EFFICIENT SCHEDULERS IN MULTIVERSION DATABASE SYSTEMS

by

Carrie Sy

B.Sc., Simon Fraser University, 1983
B.Sc., University of the Philippines, 1979

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

© Carrie Sy 1986

SIMON FRASER UNIVERSITY

April 1986

All rights reserved. This thesis may not be
reproduced in whole or in part, by photocopy
or other means, without permission of the author.

APPROVAL

Name: Carrie Sy

Degree: Master of Science

Title of Thesis: Efficient Schedulers in
Multiversion Database Systems

Examining Committee:

Chairperson: Dr. Binay Bhattacharya

Senior Supervisor: Dr. Tiko Kameda

Dr. Wo-Shun Luk

External Examiner: Dr. Naoki Katoh (in absentia)
Department of Management Science
Kobe University of Commerce

Date Approved: 8 April 1986

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Efficient Schedulers in Multiversion Database Systems

Author: _____
(signature)

Carrie Sy
(name)

16 April 1986
(date)

ABSTRACT

One way of achieving a higher level of concurrency in transaction scheduling is to keep past values of each data item in the database. This thesis will focus itself on efficient multiversion cautious schedulers which generates output sequences in classes *MWW* and *MWRW*.

In the first part of the thesis, we present the results of a simulation study, which compare the multiversion *MWW*-scheduler with *MWRW*-scheduler, assuming that an unbounded number of versions are available for each data item.

In practice, however, an unlimited number of versions cannot be maintained. In the second part of the thesis, we develop a theory which will enable us to develop efficient scheduling algorithms for K-version databases, where K is fixed. We propose a K-version cautious *MWW*-scheduler and a K-version cautious *MWRW*-scheduler, which make the scheduling decision in polynomial time.

ACKNOWLEDGEMENTS

I am most grateful to Dr. Tiko Kameda whose support, patience, and invaluable insights have enabled me to complete an endeavor which I thought would never finish. I would also like to thank Dr. Wo-Shun Luk and Dr. Naoki Katoh for their useful comments, as well as the faculty, staff, and graduate students of the School of Computing Science for making my stay in SFU both motivating and enjoyable.

To my family and

my very good friends,

Kimi and Mawie.

TABLE OF CONTENTS

APPROVAL ii

ABSTRACT iii

ACKNOWLEDGEMENTS iv

LIST OF FIGURES viii

LIST OF TABLES ix

CHAPTER 1: INTRODUCTION 1

1.1 Methods of Achieving Serializability 1

1.2 Statement of the Problem 4

1.3 Organization of the Thesis 5

CHAPTER 2: FORMAL CHARACTERIZATION OF MULTIVERSION SERIALIZABILITY 6

2.1 Database System Model 6

2.2 Serialization Constraints 7

2.3 TIO Graph and DITS 8

2.4 Cautious Schedulers 11

CHAPTER 3: MULTIVERSION SCHEDULING 14

3.1 Definitions 14

3.2 Algorithm MVCS 16

3.3 Concept of OLS 18

CHAPTER 4: SIMULATION STUDY 20

4.1 Algorithms Used 20

4.2 Simulation Model 21

4.3 Experiments and Results 25

CHAPTER 5: LIMITED VERSION SCHEDULING 42

5.1 K-version *MWW* Scheduling 43

5.2 K-version *MWRW* Scheduling 46

CHAPTER 6: CONCLUSION 56

BIBLIOGRAPHY 58

LIST OF FIGURES

Figure 2.1(a): TIO graph for Example 2.1. 9

Figure 2.1(b): DITS for TIO graph in Example 2.1. 10

Figure 2.2: WRW-augmented, exclusion-closed TIO graph of Example 2.1. 12

Figure 4.1: Average response time vs. mean transaction inter-arrival time 33

Figure 4.2: Normalized transaction delay vs. mean transaction inter-arrival time 34

Figure 4.3: Percentage old versions read vs. mean transaction inter-arrival time 35

Figure 4.4: Average response time vs. overlap of writeset with readset 36

Figure 4.5: Normalized transaction delay vs. overlap of writeset with readset 37

Figure 4.6: Illustration for explanation of results of Experiment 1. 28

Figure 4.7(a): X-segments 29

Figure 4.7(b): X-chains 29

Figure 4.7(c): Illustration of constraint arcs when degree of overlap = 100%. 29

Figure 4.8: Percentage old versions read vs. overlap of writeset with readset 38

Figure 4.9: Average response time vs. database set size 39

Figure 4.10: Normalized transaction delay vs. database set size 40

Figure 4.11: Percentage old versions read vs. database set size 41

Figure 5.1: Illustration for Proof of Lemma 5.1. 49

Figure 5.2: Illustration for Example 5.1. 52

LIST OF TABLES

Table 4.1: Varied Parameters 21

Table 4.2: Fixed Parameters 21

Table 4.3: Effects of Varying Mean Transaction Inter-arrival Time 26

Table 4.4: Effects of Varying Degree of Overlap 29

Table 4.5: Effects of Varying Database Set Size 31

CHAPTER 1

INTRODUCTION

The desire to improve user response time and throughput of database systems has led to concurrent executions of many transactions. Although concurrency does improve overall system performance, there are many problems such as lost updates and data inconsistencies [EGL76] which occur in uncontrolled concurrent accesses to a database. Therefore, some sort of control must be exercised in order to maintain the integrity of the database.

In a database system, a **transaction scheduler** checks each arriving read and write request and delays it, if necessary, to make sure the database will eventually reach a correct and consistent state. This function is commonly known as **concurrency control**.

So far, the most accepted criterion for correctness in concurrent database systems is **serializability** [BSW79, Pap79, SLR76]. Serializability can be defined as the property of a sequence of read and write operations of several transactions which guarantees that the execution of that sequence will produce the same effect on the database as that produced by a schedule in which the transactions are executed serially one after another. The concept of serializability is based on the fact that any sequential execution of transactions will leave the database in a consistent state. (We assume that each transaction, executed alone, transforms a consistent state into another consistent state). Hence, any given schedule which is equivalent to a serial schedule, i.e., a serializable schedule, will also leave the database in a consistent state.

1.1. Methods of Achieving Serializability

Locking

Locking is a common tool for concurrency control [EGL76]. Each transaction must lock a data item before any access to that item. In this method, the scheduler is simply a lock manager

that keeps track of the locks and makes sure that no two transactions simultaneously lock the same data in conflicting mode. Several locking policies have been proposed to guarantee serializability, the first of which is the two-phase locking policy [EGL76]. Here, each transaction must lock all data items to be accessed before unlocking any item. Several non-two-phase locking protocols have since been proposed. These allow a transaction to unlock some data item before all desired items are locked. Examples of these include the tree policy [SiK80], protocols based on directed acyclic graphs [KeS79], hypergraph protocol [Yan82b], and the entry point protocol [BuS85].

It is important to note that although locking is very commonly used to achieve serializability, deadlocks can occur in the system and if this happens, transactions need to be aborted. Furthermore, its power is limited only to a small subset of serializable schedules, i.e., those that are serialized under write-write constraints [Yan81]. This class has been called ***CPSR*** [BSW79], ***DSR*** [Pap79], and ***WW*** [IKM83].

Timestamping

Another commonly used method for concurrency control is timestamping. In this method, each transaction is assigned a unique timestamp when it begins executing [Ree78]. Each read and write carries the timestamp of the transaction that issued it and conflicting operations are processed in timestamp order. Hence, serialization order is determined by the order in which the transactions begin executing. Unlike locking, deadlocks do not occur in timestamping but rollbacks of already granted operations are still required.

Certification

This approach has been presented in [BHR80, StR81] and uses a technique similar to locking. However, reads and writes are processed on the first-come, first-served basis and synchronization occurs only when a transaction attempts to terminate. Thereupon the system will decide whether or not to *certify*, and therefore, commit the transaction. This method works well if very few

run-time conflicts occur, since executions are then mostly serializable. This method has the advantage in that is does not delay a transaction while it is being processed and hence, the only test for correctness is done at the transaction's termination. This method is also called *optimistic concurrency control* [KuR81].

Cautious Scheduling

Unfortunately, when conflicts are not rare, it was shown in a simulation study by Carey [Car84] that transaction rollbacks degrade performance considerably. Hence, algorithms which do not require rollbacks have been investigated.

A **cautious scheduler** is one that grants input requests if and only if it will never necessitate rollbacks. Casanova and Bernstein [CaB80, Cas81] discuss a cautious scheduler for class *CPSR* (also called *DSR* or *WW*) which is a well-known subclass of the set of all serializable schedules, **SR**. Katoh, et al. [KIK85] proposed cautious schedulers which output schedules in class ***WRW***. ***WRW*** is the class of schedules serializable under write-read and read-write constraints. They showed that although *WRW*-scheduling is, in general, NP-complete, it can be performed efficiently if all transactions are of type 1R (i.e., no more than one read step followed by multiple write steps) and if admission control (i.e., the scheduler rejects a transaction if its first request cannot be immediately granted) can be exercised.

For general, multi-step transactions, [KKI86] have introduced additional constraints in the form of a subset of *ww*-constraints to define a new class, ***WRW****. Scheduling for this class can be done in polynomial time and since it properly contains the class *WW*, it allows the most concurrency among all the efficient cautious transaction schedulers currently known.

Maintaining Multiple Versions

Still another way of achieving serializability with increased level of concurrency is by supporting multiple versions of data items. In this model, a write operation on a data item X does not overwrite the old value of X, but creates a new version. Hence, if another transaction wants

to read X, there is an option of supplying to it whichever version that will achieve serializability.

Studies on the theoretical aspects of multiversion serializability have been done [BeG83, IbK83, PaK84] and they have shown that multiversion algorithms are able to provide more concurrency than their single version counterparts. Several multiversion scheduling algorithms have also been proposed [BeG83, BuS83, Car83, Ree78, Sil82, SLR76, StR81]. A simulation study by Carey [CaM84] on three kinds of multiversion algorithms confirmed that all the multiversion algorithms outperformed their single version counterparts if the conflicts were between transactions which only read (readers) and those which read and write (updaters). Most of these algorithms [BeG83, Car83, Ree78, SLR76, StR81] utilize locking, timestamping, certification, or a variation or combination of these methods. However, they require transactions to be aborted if they fail to satisfy the serialization criterion. Since rollbacks are costly, other algorithms which do not necessitate rollbacks have been proposed. Among these are the multiversion tree protocol by [Sil82] and a generalized "progressive" (i.e. those that do not require rollbacks) protocol by [BuS83] of which the tree protocol is a special case. This latter protocol requires predeclaration of the writesets by transactions and uses timestamping and a subset of write-read constraints in order to determine the correct version to assign to each read request. Write operations are allowed to proceed without delay but read operations are sometimes required to wait if there is an update transaction with a smaller timestamp that has not written on that same data item yet. This delay prevents the possible rollbacks that occur in Reed's timestamping scheme [Ree78].

1.2. Statement of the Problem

This thesis will focus itself on efficient multiversion schedulers which do not resort to rollbacks to achieve serializability. The first part of this thesis will present results of a simulation study which compare the relative performance, in terms of response time and transaction delay, of a multiversion scheduler which makes use of read-write and write-read constraints to order conflicting transactions (***MWRW*** scheduler) and one that utilizes only write-write constraints (***MWW*** scheduler). Both schedulers assume that an unbounded number of versions are maintained

for each data item.

Since, in practice, an unlimited number of versions cannot be maintained, this thesis will propose a limited K-version cautious *MWW* scheduler and a K-version cautious *MWRW* scheduler which run in polynomial time.

1.3. Organization of the Thesis

This thesis is organized as follows: Chapter 2 presents a characterization of multiversion serializability by first giving a formal definition of the database system model and then reviewing the concepts of DITS and the TIO graph which are given in [IKM82, IKM83]. Chapter 3 discusses multiversion scheduling and the concept of OLS (on-line schedulable) [PaK84]. Chapter 4 presents the results of a simulation study, while Chapter 5 examines how the classes *MWW* and *MWRW* can be scheduled efficiently in a K-version database system, where K is fixed. The final chapter concludes the thesis and states some open problems.

CHAPTER 2

FORMAL CHARACTERIZATION OF MULTIVERSION SERIALIZABILITY

2.1. Database System Model

This thesis will adopt the same database system model and hence the same definitions as used in [KIK85]. Let our database system consist of a set **D** of **data items** and a set $\mathbf{T} = \{T_0, T_1, T_2, \cdots, T_f\}$ of **transactions**. A transaction T_i can execute a **read operation**, denoted by $R_i[X]$, on the data item X and a **write operation**, denoted by $W_i[X]$, on X. A write operation, $W_i[X]$, by T_i creates a new **version** of data item X, instead of overwriting the existing value, which is the case for single version database systems. The items written by a transaction need not be a subset of those read. A **read** (respectively, **write**) **step**, $R_i[S]$ (respectively, $W_i[S]$), where S is a subset of **D**, is an indivisible set of read (respectively, write) operations of transaction T_i, and a transaction is simply a totally ordered set of read and write steps. We assume that each data item is accessed by at most one read and at most one write operation of each transaction, and therefore, every version of any data item can be uniquely identified by the transaction that wrote it. In our model, there are two fictitious transactions, T_0, the **initial transaction** and T_f, the **final transaction**. T_0 consists of a single step, W_0[D], which "writes" the initial values of all data items. T_f consists of a single read step, R_f[D], which "reads" the final values of all data items after all other transactions have completed.

Let us define a **log**, **h**, as a sequence over the set of all the read and write steps of a set **T** of transactions. A function $\mathbf{LAST}_h^K$ is the set of all mappings associated with h, from the set of all read operations of h into the set of the write operations of h such that for each $I \in LAST_h^K$, if $I(R_j[X]) = W_i[X]$, then $W_i[X]$ is one of the last K write operations preceding $R_j[X]$ in h. For a particular I, if $I(R_j[X]) = W_i[X]$, then we say that T_j **reads** X **from** T_i in log h. Each $I \in LAST_h^K$ is

called a **K-interpretation** for log h. If $K = 1$, the set $LAST_h^K$ contains just one element. In this case, its only member is called the **standard interpretation** and is denoted by I* [Pap79]. Intuitively, the standard interpretation maps each read operation on X to the most recent write operation on X in h. If $K = \infty$, on the other hand, a read operation can read any version that has been written so far (i.e., the corresponding write operation precedes the read operation in the log.).

A **schedule**, **s**, over **T** is a pair $<h, I>$, where h is a log over **T** and I is a K-interpretation. Two schedules $s = <h, I>$ and $s' = <h', I'>$ are said to be **equivalent** (or *view-equivalent* [Yan82a]), written $s \equiv s'$, if for all pairs of indices i and j, whenever transaction T_j reads X from transaction T_i in s, T_j also reads X from T_i in s', and vice versa. In other words, $s \equiv s'$ iff $I = I'$.

A schedule $s' = <h', I'>$ is said to be **serial,** if all the steps of each transaction in h' are ordered consecutively and $I' = I^*$. A log h is said to be **(multiversion) serializable**, if there exist an interpretation $I \in LAST_h^\infty$ and a serial schedule $s' = <h', I^*>$ over the same set of transactions such that $<h, I> \equiv s'$ [BSW79, Pap79, PaK84]. To determine if a given log is serializable is, in general, known to be NP-complete [Pap79, PaK84].

2.2. Serialization Constraints

To test a given log h for serializability, we look for a serial log h' satisfying certain constraints. In a serial log h', if the operations of transaction T_i appear before those of T_j, we say that T_i is **serialized before** T_j.

Let **MVSR** denote the set of all multiversion serializable logs. The following constraints enable us to define subclasses of **MVSR**, where $<$ denotes the total order for a given log h.

(a) [*ww—constraint*] If $W_i[X] < W_j[X]$ for some data item X, then T_i must be serialized before T_j in h'.

(b) [*wr—constraint*] If $W_i[X] < R_j[X]$ for some X, then T_i must be serialized before T_j in h'.

(c) [*rw—constraint*] If $R_i[X] < W_j[X]$ for some X, then T_i must be serialized before T_j in h'.

A log h is said to belong to classes ***MWW***, ***MWR***, and ***MRW***, if h is serializable under conditions (a), (b), and (c), respectively. That is, there exists an interpretation I for h such that $<h, I>$ is equivalent to a serial schedule $<h', I^*>$ satisfying the imposed constraints (*ww*, *wr*, or *rw*). The single version counterpart of these classes are ***WW***, ***WR***, and ***RW***, respectively.

Since this thesis will be concerned with logs serializable under both *wr* and *rw* constraints, we shall abbreviate the union of these constraints as ***wrw*** and denote the class of logs serializable under these constraints as ***MWRW***.

2.3. TIO Graph and DITS

Several graph-theoretic models have been proposed in literature to formally characterize serializability of logs. Papadimitriou [Pap79] used the **polygraph**, Stearns, et al., [SLR76] the **version graph**, Sethi [Set81] the **transaction dag**, Bernstein [BeG81] the **serialization graph**, and Ibaraki, et al., [IKM83] the **TIO graph**. Recently, Vidyasankar [Vid85] introduced his **TRW** and **TP** graphs which are essentially a hybrid of the TIO graph and polygraph. Although most of them differ in their notions of serializability and the sets of constraints, all of them characterize serializability in terms of the acyclicity of appropriate graphs.

Since this thesis extends the work of [IKM82, IKM83, IKK86, KKI86], we will use the model presented there.

Definition 2.1. TIO GRAPH. The **transaction IO graph**, denoted by **TIO(s)**, for a schedule $s = <h, I>$ over a set **T** of transactions, is a labeled multigraph with the node set **T** $\cup$ **T'** and the arc set **A**, where T' is defined below. For any pair of transaction indices i and j, there is an arc $(T_i, T_j) \in$ **A** labeled by X (this arc is denoted by $(T_i, T_j):X$) whenever T_j reads X from T_i. A **dummy node** $T'_i \in$ **T'** together with a **dummy arc** $(T_i, T'_i):Y \in$ **A** are introduced if T_i writes a data item Y and if no other transaction reads Y from T_i. □

Example 2.1. The TIO graph for the following log h with the standard interpretation is shown

in Fig. 2.1 (a). The steps of h are shown in the order given by its total order, and $R_j[X_i]$ indicates that $I^*(R_j[X]) = W_i[X]$.

$$<h, I^*> = W_0[X,Y,Z]\ R_1[X_0]\ W_2[Z]\ R_2[Y_0]\ W_1[X,Z]\ W_2[X]\ R_3[X_2]\ W_2[Y]\ W_3[Z]\ R_f[X_2,Y_2,Z_3]$$

An **interval** is defined to be a set of all arcs that have the same label and originate from the same node [IKM82].

Definition 2.2. DITS. The total order << on the set of nodes of $TIO(s)$ is a **disjoint-interval topological sort** (**DITS**, for short), if it satisfies the following two conditions:

(a) If $T_i << T_j$ then there is no path from T_j to T_i in $TIO(s)$, and

(b) Let $(T_h, T_i):X$ and $(T_j, T_k):X$ be any two arcs labeled by X in $TIO(s)$ such that $h \neq j$. Then either $T_k << T_h$ or $T_i << T_j$. □

Condition (b) is referred to as the **exclusion rule.** An unlabeled **exclusion arc** (T_i, T_j) is introduced if there are two arcs $(T_h, T_i):X$ and $(T_j, T_k):X$ such that there is a path in $TIO(s)$ from T_h to T_k (possibly through T_i or T_j).

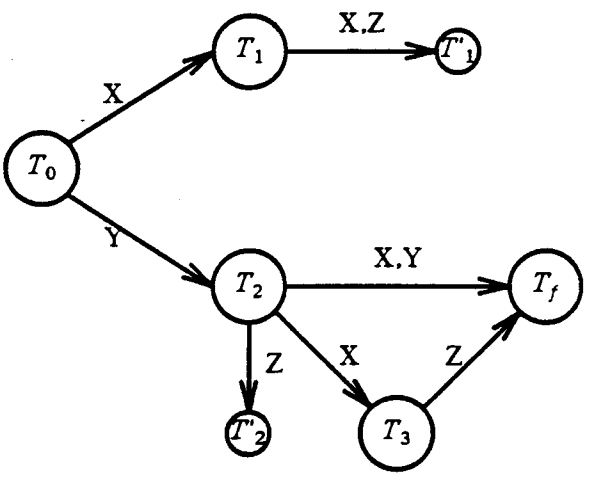


Figure 2.1(a) TIO graph, TIO(s) for Example 2.1.

Intuitively, $TIO(s)$ has a DITS if the nodes can be linearly arranged horizontally in such a way that all arcs are directed from left to right and no two intervals with the same label "overlap", i.e., no imaginary vertical line intersects more than one interval with the same label, no matter where it is placed.

The importance of the concept of DITS in serializability theory can be seen in the following theorem.

Theorem 2.1. [IKM82]. *A schedule s is serializable if and only if $TIO(s)$ has a DITS which orders T_0 first and T_f last.* □

Consider the schedule $s = <h, I^*>$ given in Example 2.1. The TIO graph for s has a DITS as shown in Fig. 2.1 (b) and hence, log h is serializable. The **serialization order** implied in this DITS gives rise to the following serial schedule that is equivalent to s.

$$<h', I^*> = W_0[X,Y,Z]R_1[X_0]W_1[X,Z]R_2[Y_0]W_2[X,Y,Z]R_3[X2]W_3[Z]R_f[X_2,Y_2,Z_3].$$

The equivalence of two schedules, s and s', can be established by testing if $TIO(s) = TIO(s')$. One of the main reasons of using the concept of DITS and the TIO graph in this thesis is that they not only provide a useful characterization of serializability, but also help in the characterization and construction of cautious schedulers which we are interested in.

Constraints such as *ww*, *wr*, *rw*, and *wrw* which are imposed on the serialization order can be indicated in the TIO graph by **constraint arcs**. Depending on the constraint it represents, such an

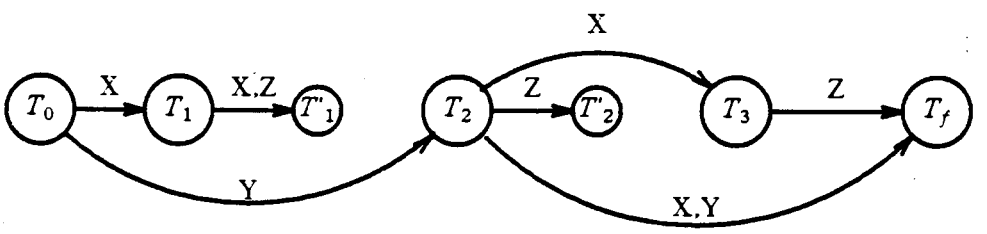


Figure 2.1(b) DITS for TIO(s) of Example 2.1.

arc is called, a **ww-arc, wr-arc** or **rw-arc.** By adding all the constraint arcs corresponding to a set c of constraints to $TIO(s)$, we obtain the **c-augmented TIO graph,** denoted by $TIO_c(s)$. Two schedules, s and s', are said to be *c—equivalent* if they satisfy the set c of constraints such that $TIO_c(s) = TIO_c(s')$. C-equivalence of these two schedules is denoted by $s \equiv_c s'$. If we repeatedly introduce exclusion arcs due to the exclusion rule until the rule is no longer applicable, the resulting graph is said to be **exclusion closed,** and is denoted by $TIO^*_c(s)$.

Theorem 2.2. [IKM82] *If c is any set of constraints (ww, wr, etc.), and C stands for the class of serializable logs satisfying the constraints in c, then a log h belongs to C, if and only if $TIO_c(<h, I^*>)$ has a DITS.* □

Theorem 2.3.

(a) [BSW79, Pap79] $TIO_{ww}(s)$ has a DITS iff $TIO^*_{ww}(s)$ is acyclic.

(b) [IKM82, IKM83] $TIO_{wrw}(s)$ has a DITS iff $TIO^*_{wrw}(s)$ is acyclic. □

Example 2.2. Consider schedule $s = <h, I>$ of Example 2.1 again. Fig. 2.2 illustrates the wr- and rw-arcs, as well as some exclusion arcs (unlabeled). Note that the exclusion arc (T'_2, T_3) is due to the reads-from arcs $(T_2, T'_2):Z$, $(T_3, T_f):Z$ and a path from T_2 to T_f. Since the graph is acyclic, log h belongs to WRW.

Now, if we impose the *ww*-constraints on the serialization of h, we obtain a *ww*-arc from T_1 to T_2 as well as from T_2 to T_1, because $W_1[X] < W_2[X]$ while $W_2[Z] < W_1[Z]$ in h. Therefore, by Theorem 2.2, h is not serializable under the *ww*-constraints. It is known that single version WW is a proper subset of the single version WRW [IKM82].

2.4. Cautious Schedulers

So far, we have concerned ourselves with the testing of membership in different serializability classes. However, testing for membership in particular classes has no practical use, per se. What is practically important in a database system with concurrent transactions is the

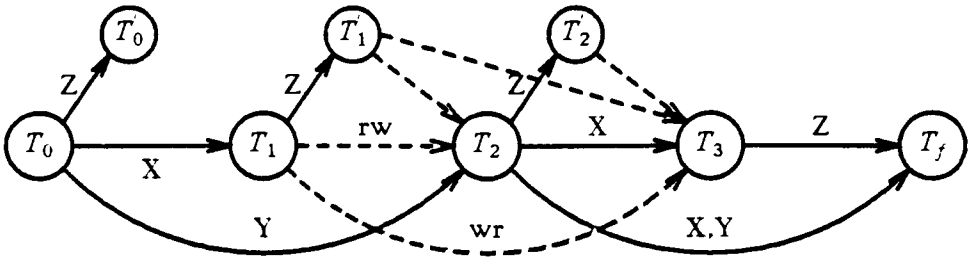


Figure 2.2. The wrw-augmented, exclusion-closed TIO(s) of Example 2.1.

database scheduler. Although the scheduler for class C makes sure that its output log is in C, the information available to it is not the same as in membership testing. Whereas in a membership test the entire log to be tested is assumed to be known, in a database scheduler, the decision of whether to grant or delay the current request must be made based solely on the information that is currently known. In addition, a multiversion scheduler must also decide which of the available versions to assign to each read request.

As mentioned earlier, no operation is ever rolled back in **cautious-scheduling**. A cautious scheduler may delay a request if granting it immediately leads to nonserializable output, but it never aborts or rolls back any operation that has been granted. In order to realize a "reasonable"[1] cautious scheduler, we assume that, upon arrival, each transaction informs the scheduler of its read set (i.e., the set of data items that it is going to read) and write set (i.e., the set of data items that it is going to write).

The most essential element and usually the most time-consuming part of any cautious scheduler is the **completion test** [KIK85]. It determines whether or not it is "safe" to grant the current request by testing whether the partial schedule consisting of the already output sequence followed by the current request can be augmented by a sequence of all pending steps in such a way that the resulting schedule belongs to the desired class, e.g., *WRW* or *CPSR*. Here, we may

[1] A scheduler that allows only the serial execution of transactions is clearly "unreasonable".

assume that the pending steps arrive in the most favorable sequence; otherwise, the scheduler can delay some of them in order to force them into such a sequence.

In the following chapter, we take a closer look at multiversion scheduling by presenting an algorithm for a general multiversion scheduler and also discussing the intrinsic limitations of any multiversion scheduler.

CHAPTER 3

MULTIVERSION SCHEDULING

Before presenting a general multiversion scheduling algorithm, we first define a few terms that we will use frequently.

3.1. Definitions

Definition 3.1 A **partial schedule** $<P, I'>$ is the log P together with an interpretation I' that has so far been granted and output by the multiversion scheduler. The **current request**, q, is the step of operations which is being examined for granting or delaying. There are two groups of steps that are examined by the scheduler.

The first group consists of **delayed** steps and they are all kept in a list ***DEL***. The steps in ***DEL*** have already arrived and have been delayed by the multiversion scheduler. Note that there is at most one step from any one transaction since a transaction does not send its next request if the last one is delayed. The second group, ***ANT***, consists of **anticipated** steps. Each step in ***ANT*** belongs to a transaction whose initial request has already arrived, irrespective of whether that request has been granted or not. All steps in ANT have not arrived yet but they are all expected to. Lastly, we call the steps in $\boldsymbol{PEND} = DEL \cup ANT - \{q\}$ **pending steps**.

In the previous chapter, we mentioned that the most crucial part of cautious scheduling is the completion test. Now, we give a formal definition of a multiversion completion test.

Definition 3.2. Multiversion Completion Test. Given $<P, I'>$, q, and $PEND$, the multiversion scheduler determines whether it is possible to **complete** the partial schedule $<P, I'>$, by appending to it a sequence $qQR_f[\mathbf{D}]$ with an interpretation I, such that

(i) Q is a sequence over $PEND$,

(ii) the order of steps in Q is consistent with that among the steps of each T_i,

(iii) the resulting schedule $<PqQR_f[\mathbf{D}], I>$ belongs to a given class C, and

(iv) I is an **extension** of I'.

Let I and I' be interpretations for logs h and h', respectively, such that h' is a prefix of h. I is said to be an **extension** of I' if $I = I'$ in the domain h'.

This test is called the **multiversion C-completion test**, where C refers to the class mentioned in (iii) above. A C-completion can be defined as $<PqQR_f[\mathbf{D}], I>$ such that $<PqQR_f[\mathbf{D}], I> \in C$. In this paper, the class C is either *MWW* or *MWRW*. One must remember that, besides deciding on-line whether to grant or delay the current request, as all ordinary (single version) schedulers do, the multiversion scheduler must also decide which of the versions available to assign to each read request; that is, it must also select an interpretation.

Since completion test also requires some sort of membership test, it will be natural to modify the TIO graph to reflect the information that is currently available to the multiversion scheduler. This graph is called the **ATIO graph** and it is with the use of this graph and DITS that this thesis will construct a multiversion scheduler.

Definition 3.3. ATIO Graph [IKK86] The **active TIO graph**, denoted by ***ATIO(<P, I'>, q, PEND)***, has a node set consisting of the transactions currently known to the scheduler, as well as some **dummy nodes**. The reads-from and dummy arcs corresponding to the steps in P are constructed in the same way as those in the TIO graph. For each $W_i[X]$ in *PEND*, a dummy arc $(T_i, T''_i):X$ is introduced, where T''_i is a dummy node.

For convenience, the arcs corresponding to the steps in P are drawn thick and those in *PEND* thin. Moreover, if q is a write step, the corresponding dummy arc is also drawn thick. Although it is not included in the formal definition of the ATIO graph, we will indicate a pending read operation on X by a transaction T_i as a dangling arc to node T_i labeled by X. These arcs simply serve to remind us pictorially which transactions still have pending read operations.

Since we are interested in a *MWRW* (respectively, *MWW*)-completion, we need to test only those serial schedules which satisfy the constraints implied by the ATIO graph augmented by the *rw*- and *wr*-arcs (respectively, *ww*-arcs). To take the *wrw*-constraints into account, we modify the existing ATIO graph by adding the following arcs. For each $X \in D$, add an *rw*-arc (T_i,T_j) if either $R_i[X]$ precedes $W_j[X]$ in Pq, or $R_i[X] \in Pq$ and $W_j[X] \in PEND$. Similarly, a *wr*-arc is introduced if either $W_i[X]$ precedes $R_j[X]$ in Pq, or $W_i[X] \in Pq$ and $R_j[X] \in PEND$. The resulting graph is called the **wrw-augmented ATIO graph** and is denoted by $ATIO_{wrw}(<P, I'>, q, PEND)$. Similarly, if we repeatedly add exclusion arcs due to the exclusion rule (condition b of Definition 2.2), until the rule is no longer applicable, we obtain the exclusion closure of the graph and this is denoted by $ATIO^*_{wrw}(<P, I'>, q, PEND)$. To take the *ww*-constraints into account, we add, for each $X \in \mathbf{D}$, a *ww*-arc (T_i, T_j), if either $W_i[X]$ precedes $W_j[X]$ in Pq, or $W_i[X] \in Pq$ and $W_j[X] \in PEND$. The exclusion rule is similarly applied in order to obtain the $ATIO^*_{ww}(<P, I'>, q, PEND)$. A DITS for an ATIO graph is defined in exactly the same way as for a TIO graph.

Now, we are ready to present our multiversion scheduling algorithm. This algorithm is similar to the one presented in [IKK86]

3.2. Algorithm MVCS

MV0 : [Initialization]

$P := W_0[D]$, $q :=$ the first request, $DEL := nil$,

$ANT := \emptyset$.

Include a node representing the first transaction in the ATIO graph

MV1 : [Test the current request, q]

Draw all necessary constraint and exclusion arcs.

Perform *MV—COMPLETION TEST*

If completion test is successful, go to MV3.

else restore the ATIO graph to the state just before

q arrived.

MV2 : [q was delayed / rejected]

If $q \in DEL$

then if all steps in *DEL* have been unsuccessfully tested

then go to MV5;

else do; let q be the next step of *DEL*

return to MV1

end;

else if $q \in ANT$

then do;

delete q from *ANT* and add it to the end of *DEL*

go to MV4

end;

MV3 : [q was granted]

$P := Pq$;

if $q \in DEL$

then delete q from *DEL*;

else if $q \in ANT$

then delete q from *ANT*;

MV4 : [Pick the next q in *DEL*]

if $DEL \neq nil$

then do;

q := next step in *DEL*

go to MV1

end;

MV5 : [Pick the next arriving request]

q := next arriving request

if q is the first step of a new transaction T_j

then do;

$ANT := ANT \cup \{steps\ of\ T_j\} - \{q\}$

include node representing T_j in the $ATIO$ graph and

draw all the necessary constraint arcs due to T_j

go to MV1

end;

Before going any further into constructing multiversion schedulers, we digress a little bit by discussing the intrinsic limitations of multiversion schedulers.

3.3. Concept of OLS

Papadimitriou [PaK84] defined the concept of **OLS** (**on-line schedulable**) in relation to multiversion concurrency control algorithms. Informally, a set S of log is **OLS** iff the scheduler can, as long as it sees a legal prefix of a log in S, decide on one interpretation that is guaranteed to be good *for all* possible continuations. OLS is the basic requirement for a set of logs to be output by a multiversion scheduler. Unfortunately, it was proved that *DMVSR* (= *MWW* [IbK83]) is not OLS [PaK84]. Since *MRW*, which the the largest polynomially recognizable subclass for multiversion schedules, is a superset of *MWW* [IbK83], then it is also not OLS [HaP85].

Since we are also interested in class *MWRW* in this thesis, we would like to know whether this class is OLS or not. We prove that *MWRW* is not OLS by an example.

Example 3.1. Consider

$$h_1 = W_0[X,Y]W_1[X]W_3[X]R_2[X]R_1[Y]W_3[Y]R_f[X,Y]$$

$$h_2 = W_0[X,Y]W_1[X]W_3[X]R_2[X]W_3[Y]R_1[Y]R_f[X,Y]$$

It is easy to see that both schedules are in *MWRW* but h_1 is *wrw*-equivalent to $T_0T_1T_3T_2T_f$ if T_2 reads X from T_3 and this is the only interpretation that makes h_1 *wrw*-equivalent to a serial schedule. On the other hand, h_2 is *wrw*-equivalent to $T_0T_3T_1T_2T_f$ only if T_2 reads X from T_1 and this is again the only way to serialize h_2. At the time $R_2[X]$ arrives, the scheduler must select a version of X to assign to this read operation. If it selects X_3 and $W_3[Y]$ arrives next, then it must be delayed, even though h_1 belongs to *MWRW*. If it selects X_1, on the other hand, and $R_1[Y]$ arrives next, $R_1[Y]$ must be delayed, even though h_2 also belongs to *MWRW*. Hence, we see that *MWRW* is not OLS. □

Since neither *MWW*, *MRW*, nor *MWRW* is OLS, we would like to find subclasses of these classes which are *maximal* with respect to the OLS property. A subset of *MVSR* is a **maximal set of OLS logs** if, when any log $\in$ *MVSR* is added to the set, it ceases to be OLS. We would, of course, like to design multiversion schedulers which output these maximal sets. However, [HaP85] proved that there cannot be any efficient multiversion scheduler which can do this. Hence, any practical scheduler will output a non-maximal subset.

In summary, the significance of the concept of OLS in multiversion scheduling is that the class of logs that we want our scheduler to output must first of all be OLS. If it is not, then it is possible that the scheduler will delay some steps even though the given input sequence, in fact, belongs to the desired subclass. Furthermore, any efficient scheduler that one can come up with will form a non-maximal class.

Now that we know the functions and limitations of a general multiversion scheduler, we present, in the next chapter, the results of a simulation study aimed at comparing the relative performance of an *MWW*-scheduler and an *MWRW*-scheduler, assuming unlimited number of versions for each data item.

CHAPTER 4

SIMULATION STUDY

4.1. Algorithms Used

This section briefly points out the differences between the two scheduling algorithms that we are going to compare by simulation.

4.1.1. *MWW Scheduling*

The class *MWW* is also called *DMVSR* [PaK84]. Basically, an *MWW*-scheduler reorders input requests, if necessary, so that the output log it produces is serializable under the write-write (*ww*, for short) constraints. The simulation program which updates the ATIO graph, implements this by drawing constraint arcs, called *ww*-arcs, from those transactions which have already written a data item to a newly arrived transaction which will eventually write that data item. We also draw a *ww*-arc from the current transaction with a write request on a data item to all transactions with pending writes on that data item. Namely, we pretend as if the current request had been granted. If the current request contains a read operation $R_j[X]$, an exclusion arc is drawn from T_j to T_k, where $W_k[X]$ is a pending write request. This exclusion arc is due to two intervals labeled by X, $(T_i, T_j):X$ and $(T_k, T'_k):X$, and a *ww*-arc (T_i, T_k), where T_i is the transaction from which T_j reads X. If the scheduler concludes that the current request cannot be granted, the current request is placed in a queue of delayed requests and all arcs introduced due to the current request are deleted.

4.1.2. *MWRW Scheduling*

This scheduler uses both the read-write and write-read constraints to serialize interacting transactions. The *rw*-arcs are drawn from those transactions whose read operations on a data

item have been granted, to those which will write on that item. Similarly, a *wr*-arc is drawn from each transaction whose write request on a data item has been granted, to those transactions which will read the same data item. As in the *MWW*-scheduler, we pretend as if the current request had been granted, and if it cannot be granted, we undo the changes caused by it.

For both schedulers, a request is granted if the exclusion-closed ATIO graph augmented by the necessary constraint arcs is acyclic. [IKK86].

4.2. Simulation Model

The program was written in the C language and run under UNIX 4.2 BSD on SUN-2.

4.2.1. Parameters

Table 4.1 shows the parameters which were varied in the different runs of the simulation program in order to see how interaction among transactions affects performance. Table 4.2 shows the fixed parameters.

NumT (see Table 4.2) is the number of transactions that are generated in one simulation run. The number of data items available for access in the simulated database system is given by *DSize* and these data items are represented by integers ranging from 1 to *DSize*. The size of the writeset

Parameters	Range	Description
DSize	20-100	number of data items in dbs
OV	0-100	percentage of writeset that overlaps with readset
T_Int_Arr	6-15	mean transaction interarrival time

TABLE 4.1: VARIED PARAMETERS

Parameters	Set Values	Description
NumT	750	maximum number of transactions
MXWSIZE	6	maximum size of a transaction's writeset
S_Int_Arr	5	mean step interarrival time
MXDPERSTEP	3	max number of data items in a step
TLIMIT	25	max number of concurrently active processes

TABLE 4.2: FIXED PARAMETERS

of transactions, *WSize*, is assumed to be a random variable having a uniform distribution over the range [1,*MXWSIZE*] with mean (1+*MXWSIZE*)/2. The size of the readset is assumed to be, on the average, 20 % larger than the writeset. *OV* is the average percentage of a transaction's writeset that overlaps with its readset. More precisely, (*OV*/100)*(1+*MXWSIZE*)/2 is mean of the number of data items that are in both readset and writeset of a transaction. *MXDPERSTEP* is the maximum number of data items that a step may access and the actual numbers are assumed to be uniformly distributed over [1,*MXDPERSTEP*] with mean (1+*MXDPERSTEP*)/2. The inter-arrival times of the transactions and of the steps of a transaction are assumed to have exponential distribution with means, *T_Int_Arr* and *S_Int_Arr*, respectively. The ratio of these two will determine to what degree the interleaving of the steps of different transactions will occur in the request arrival sequence. *TLIMIT* sets the limit on the number of concurrently running transactions in the system. This limit had to be artificially imposed because UNIX on the SUN-2 workstation which was used to run the simulation allows only 30 file descriptors, including the standard input, standard output, and standard error file, that a program can maintain at any one time. However, this limit was never reached in any of the simulation runs whose results we have presented in this thesis.

4.2.2. Over-all Description of Simulation Process

To simulate transaction arrivals, the first transaction is generated at logical time 0. Whenever a new transaction begins execution, the logical time when the next transaction will start is generated using an exponential distribution with mean *T_Int_Arr*. The anticipated arrival of a new transaction is recorded in an event queue so that when the time for arrival of the new transaction is reached, a new child process is spawned to generate the requests of that transaction. For each transaction, the number of data items that it accesses, that is, the size of the union of its readset and writeset is given by:

$$rw\_union = WSize + 1.2*WSize - WSize*(OV/100) = (2.2 - OV/100)*WSize$$

where *WSize* is determined by randomly picking a number from a uniform distribution over the

range [1,*MXWSIZE*]. $(2.2 - OV/100)$**WSize* distinct integers are randomly selected from a uniform distribution over the range [1,*DSize*]. These data items are then "marked" as either for read-only, for write-only, or for read/write by the following method. For each data item, an integer is randomly selected from a uniform distribution over the range [1,10000]. If the number generated is less than *ro_bound* = $10000*(OV/100) / (2.2 - OV/100)$, then that data item will be marked as for read/write. If the number generated falls between the range [*ro_bound*, *rw_bound*], where *rw_bound* = $10000*1.2/(2.2 - OV/100)$, then it is marked as read-only; otherwise, it is marked as write-only. This method will, on the average, generate readsets that are 20% larger than the writesets but it may also generate read-only as well as write-only transactions.

A step of a transaction is generated by first randomly choosing the type of step, i.e., read or write, with equal probability. If it is a read step, a random number (between 1 and *MXDPERSTEP*) of data items that are tagged as either read-only or read/write are selected to form the read step. On the other hand, if it is a write step, a random number (between 1 and *MXDPERSTEP*) of data items tagged write-only or those tagged read/write (provided they have already been used in a read step), are selected to form the write step. Therefore, whenever an item is both read and written, the read always precedes the write. Once a step has been formed, the child process writes the description of the step onto a pipe for the main parent process to read. The description also contains the time increment from the current step to the next step of that transaction, which is generated by the child process with the use of *S_Int_Arr*. This enables the main process to expect the arrival of a transaction's next request and to know when to read the pipe associated with that transaction. Step generation for a transaction continues until all data items in its readset and writeset have been used. During step generation, if all data items are for read-only, then only read steps are generated. Similarly, for the case where all remaining data items are for write-only.

There are several queues that the program maintains. One of them is an event queue, the head of which is pointed to by *qhead*. An event may either be the arrival of a new transaction or the arrival of the next request from a transaction that is already in the system. This queue is ordered according to the time that the events are scheduled to occur.

The scheduler follows basically the same algorithm as described in Chapter 3. In order to decide whether or not to grant the current request, the scheduler "draws" the necessary arcs in the ATIO graph and tests its exclusion-closure for acyclicity. If the request cannot be granted, the state which the system was in before the arcs were put in is restored and the request is appended to the tail of the *DEL* queue.

4.2.3. Major Data Structures

The ATIO graph is represented as an adjacency structure called *TABLE*. This table contains the pointers to the nodes and has twice as many entries as the number of transactions, *NumT*. The nodes adjacent to transaction i are chained together and pointed to by *TABLE*[2**i*].*head*. *TABLE*[2**i*+1] points to the dummy node, if it exists, associated with transaction i. To facilitate the "drawing" of exclusion arcs, a structure called *dlist* is used. It is an array of size *MAXD* (upper bound on *DSize*), and each element of the array points to the list of arcs labeled by a particular data item.

To determine the version of the data item to assign to a read request, the ATIO graph is topologically sorted. The standard interpretation for the log corresponding to the resulting transaction order is used to assign appropriate versions to the current read request.

In order to keep the graph within a manageable size, we merge with T_0 (the initial transaction) any transaction whose last request has already been granted and which has only T_0 as its predecessor in the ATIO graph. Once a transaction is merged with T_0, all the reads-from arcs coming out of it now come out of T_0 and that transaction is, in effect, forgotten by the scheduler.

4.3. Experiments and Results

In this section we present the results of three simulation experiments designed to compare the performance of the two multiversion schedulers, *MWW*-scheduler and *MWRW*-scheduler, for different sets of parameters. Experiment 1 examines the effect of varying the mean transaction inter-arrival time on the performance of the two schedulers. Experiment 2 examines the effect of varying the degree of the mean overlap of the writeset with the readset. Lastly, Experiment 3 examines the effect of varying the number of items available in the database, *DSize*.

The "typical" values assumed for the three parameters varied in the experiments are 80 % for the degree of overlap (*OV*), 8 for the mean transaction inter-arrival time (therefore, *T_Int_Arr*/*S_Int_Arr* = 8/5 =1.6), and 45 for database size (*DSize*). *DSize* = 45 should be contrasted to *MXWSIZE* = 6. Performance of the schedulers is measured in terms of *average response time* and *average normalized transaction delay*. **Average response time** is the time a step has to wait from the time it submits a request to the time the request is granted. In all the experiments, this value has been normalized with respect to the mean step interarrival time, *S_Int_Arr*. This is done to give an indication of how much delay is incurred in relation to the mean step interarrival time. **Average normalized transaction delay** gives a measure of the delay that a transaction suffers due to interaction among the transactions, with respect to its length. In other words,

$$Norm.\ Tdelay = \frac{actual\ time\ it\ took\ a\ transaction\ to\ finish - length\ of\ the\ transaction}{length\ of\ the\ transaction}$$

where actual time refers to the duration of time when a transaction's first request is submitted to the time when its last request is granted. Length of the transaction refers to the time it would take from the start of a transaction until its last request is granted, assuming there were no delays. If the transaction has only one step, then normalized transaction delay is 0 because a one step transaction is always granted immediately. Another aspect that we investigated in the 3 experiments is the percentage of reads that are assigned versions other than the most recent. We wanted to see whether varying the different parameters would have any effect on the percentage.

Experiment 1:

This experiment examines the effect of the mean transaction inter-arrival time, *T_Int_Arr*, on the performance of the schedulers as it is varied from 6-15. The ratio, *T_Int_Arr / S_Int_Arr*, (indirectly) determines the degree of interleaving among the steps of different transactions. Figure 4.1 and columns 2-3 of Table 4.3 show the relation of the average response time versus the mean transaction inter-arrival time, while Figure 4.2 and columns 4-5 of Table 4.3 show the relation of the normalized transaction delay versus the same parameter.

The effect of varying the mean transaction inter-arrival time on both measures of performance is as expected. When the mean transaction inter-arrival time is short (relative to the mean step inter-arrival time), transactions tend to come into the system before existing transactions have had a chance to finish or do much. This causes interference among the transactions, resulting in increased delay for requests. When the mean transaction inter-arrival time is long, transactions are executed almost serially since the next transaction does not usually arrive until most, if not all, of the steps of the previous transactions have been carried out. Another point to note in both graphs is that *MWRW* scheduler generally performs only slightly better than the *MWW* scheduler.

The effect of varying the mean transaction inter-arrival time on the percentage of version assignments from the old versions can be seen in Figure 4.3 and columns 6-7 of Table 4.3. These

Mean transaction inter-arrival time	Average response time		Normalized transaction delay		Percentage of old versions read	
	ww	wrw	ww	wrw	ww	wrw
6	1.22	1.14	2.75	2.35	2.81	4.42
8	0.80	0.78	1.74	1.71	2.19	2.61
10	0.61	0.63	1.25	1.25	1.58	3.19
12	0.54	0.51	1.03	0.87	1.44	2.13
14	0.48	0.44	0.96	0.77	1.13	2.30
15	0.40	0.38	0.84	0.67	1.10	2.13

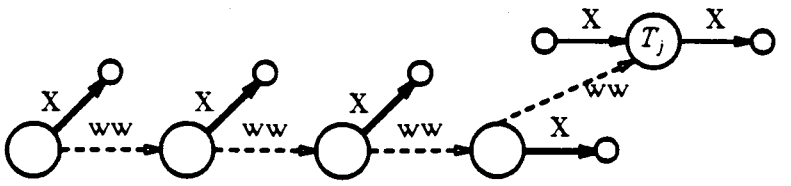
TABLE 4.3: Effects of Varying Mean Transaction Inter-arrival Time

results are also as expected. When the mean transaction inter-arrival time is long, transactions arrive almost serially; hence, the interpretation given by the scheduler to these transactions closely resembles the standard interpretation. The fact that the percentage for the *MWRW*-scheduler is consistently higher than that for the *MWW*-scheduler indicates that the *wrw*-constraints allow more flexibility in version assignment. This is because for the *MWW* scheduler, all transactions writing data item X will be ordered with *ww*-arcs with the most recent version on the rightmost end (Figure 4.6a), whereas such ordering is not imposed by the *MWRW* scheduler. Now, consider a read request $R_j[X]$. Since the degree of overlap of the writeset with the readset is 80%, many of these read requests will also be writing X. So, a "typical" transaction like T_j will have to read the most recent version of X in the *MWW* scheduler (because of the *ww* arc from the most recent version to T_j (Figure 4.6a)). On the other hand, in the *MWRW* scheduler, these transactions may read from older versions (Figure 4.6b).

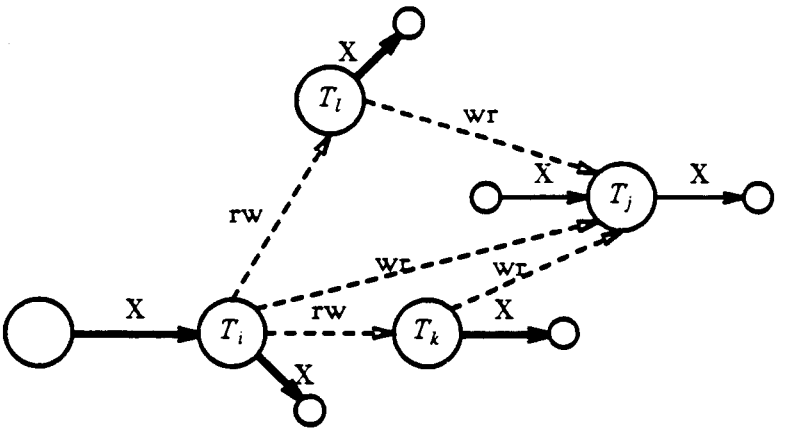
Experiment 2:

This experiment examines the effect of *OV*, the degree of overlap of the writeset with the readset, on the performance of the two schedulers. Note that since the average sizes of the readset and writeset are kept constant as *OV* is varied, the total number of data accessed by a transaction decreases as *OV* increases.

The effect of varying the degree of overlap on the average response time and the normalized transaction delay can be seen in Figures 4.4, 4.5, and Table 4.4. Although it is difficult to discern a clear-cut trend, it is probably safe to conclude for both schedulers that at lower (respectively, higher) degree of overlap, average response time and normalized transaction delay are shorter (respectively, longer). At lower *OV*, say $OV = 0$, a transaction that reads a data item X never writes X and those that write X never read X. This gives rise to isolated **X-segments** (Figure 4.7a) in the ATIO graph, as contrasted to the **X-chains** (Figure 4.7b) that may be formed when *OV* is higher. Therefore, a transaction that reads but does not write X will be less constrained in the sense that it has more choices of versions to read from without violating the DITS property of the



(a) *ww* arcs forces T_j to read from the most recent version.



(b): T_j reads X from either T_k or T_l but
these need not be the most recent version.

Figure 4.6: Illustration for explanation of the results of Experiment 1.

ATIO graph. On the other hand, if the transaction both reads and writes X, the constraint arcs will order this transaction into some DITS position and this severely limits its choices of versions. To be more specific, in the case of the *MWW* scheduler, although the *X–segments* are linked by *ww*-arcs, a transaction that reads but does not write X could read X from any of the available versions, because there are no *ww* arcs to constrain it to read from the most recent version. As for the *MWRW* scheduler, transactions that write X are not ordered by *ww*-arcs, so a read request may read from any of them within the constraints given by the ATIO graph. When *OV* is very high, say 100%, then the *ww*– (*wrw*–) arc to a transaction T_i that both reads and writes X forces T_i to read only from the tail end of the chain, that is, the transaction that wrote the most recent

version of X (Figure 4.7c).

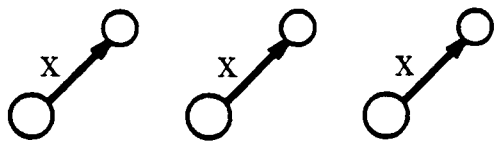


Figure 4.7(a): X-segments

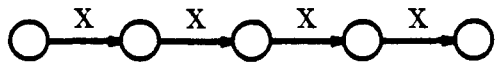


Figure 4.7(b): X-chain

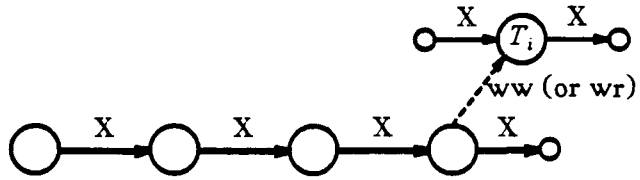


Figure 4.7(c): Constraint arcs forces T_i to read from the most recent version.

Overlap of writeset with readset	Average response time		Normalized transaction delay		Percentage of old version read	
	ww	wrw	ww	wrw	ww	wrw
0	0.53	0.43	1.17	0.83	5.81	13.64
10	0.69	0.61	1.39	1.12	6.24	12.96
20	0.79	0.64	1.81	1.25	6.25	11.05
30	0.64	0.54	1.29	1.26	4.83	9.82
40	0.99	0.86	1.95	1.58	4.44	10.37
50	0.95	0.82	1.66	1.37	4.23	10.20
60	1.12	1.03	2.22	1.95	3.98	7.96
70	1.33	1.29	2.81	2.78	3.11	4.11
80	0.80	0.78	1.74	1.71	2.19	2.61
90	1.06	0.96	2.39	2.07	2.59	1.86
100	1.01	1.05	2.34	2.45	0.81	0.00

TABLE 4.4: Effects of Varying Degree of Overlap

The results also show that except for degree of overlap very close to 100%, the *MWRW* scheduler performs slightly better than the *MWW* scheduler. We now give an explanation for the slightly better performance of *MWW* at degrees of overlap close to 100%. Although a transaction that reads and writes X is constrained in both schedulers to read only from the tail end of the chain (i.e., the most recent version), there are still some transactions which read but do not write X. (Recall from the definition of *OV* that $OV = 100\%$ means that any transaction writing X also reads X but not vice versa). These transactions are still constrained in the *MWRW* scheduler, through *wr*-arcs, to read the most recent version of X; but they are not in the *MWW* scheduler. These findings and the fact that Ibaraki et al. [IbK83], showed that *MWW* properly contains *MWRW* when the writeset is a subset of the readset seem to indicate that the *MWW* scheduler delays fewer requests than the *MWRW* scheduler whenever the writeset is a subset of the readset.

The sudden peaks and dips of the values in the middle range are most likely caused by random variations in the numbers generated by the random number generator, and probably have no definitely identifiable causes.

The results shown in Figure 4.7 and columns 5-6 of Table 4.4 support our earlier explanation that at lower degree of overlap, a read request is allowed to read any of a number of versions and at higher degrees of overlap, more restrictions force most transactions to read from the most recent versions. Notice that at 100% overlap, only the most recent versions are ever read in the *MWW*-scheduler.

Experiment 3:

This experiment examines the effect of the database set size, *DSize*, on the performance of the two schedulers. The simulation results are shown in Figures 4.8, 4.9, and Table 4.5. When *DSize* is small, the same data item is accessed by many transactions; hence, more conflicts occur among transactions, which, in turn, cause more delay. On the other hand, when *DSize* is large, there are more data available for a transaction to access; hence, fewer transactions will access the same data

more data available for a transaction to access; hence, fewer transactions will access the same data items, resulting in fewer conflicts and in a decrease in delay.

Although not very pronounced, the trend shown in Figure 4.10 and columns 6 & 7 of Table 4.5 shows that older versions tend to be read more often when the database size *DSize* is smaller. We now attempt a plausible explanation to these results. Imagine an ATIO graph for a particular value of *DSize* and consider the relative position of a transaction requesting to read the data item X with respect to the transactions which have written X. Assume that it is constrained to follow the one that wrote the most recent version of X. Now, if *DSize* is decreased, more transactions will be accessing X. So, another transaction that might have written another data item may now write the most recent version of X. Hence, that version of X which was the most recent when *DSize* was larger has become an "older" version when *DSize* decreased. This argument is somewhat oversimplified, but is probably one reason for the observed behavior.

Summary:

We have seen above that both schedulers seem to perform just about equally with the *MWRW* scheduler slightly better in most cases, except when the degree of overlap is very close to 100%. What we have simulated so far are schedulers which assume unlimited number of avail-

Database set size (DSize)	Average response time		Normalized transaction delay		Percentage of old versions read	
	ww	wrw	ww	wrw	ww	wrw
20	1.42	1.56	3.86	4.22	2.76	4.20
25	1.28	1.20	2.70	2.51	2.61	5.28
35	1.08	1.01	2.29	1.91	2.95	4.28
45	0.80	0.78	1.74	1.71	2.19	2.61
55	0.80	0.81	1.93	2.02	2.22	3.79
65	0.79	0.74	2.18	1.93	1.60	2.73
75	0.80	0.74	1.86	1.71	2.05	3.11
85	0.92	0.81	1.89	1.64	1.88	3.11
95	0.74	0.69	1.62	1.55	1.37	3.02
105	0.74	0.68	1.60	1.52	1.71	2.15

TABLE 4.5: Effects of Varying Database Set Size

able versions. This enabled us to obtain easily implementable efficient scheduling algorithms; but this assumption is not practical in the real systems. In the next chapter, we present conditions necessary to implement schedulers efficiently for classes *MWW* and *MWRW*, assuming a bounded number of versions maintained by the system.

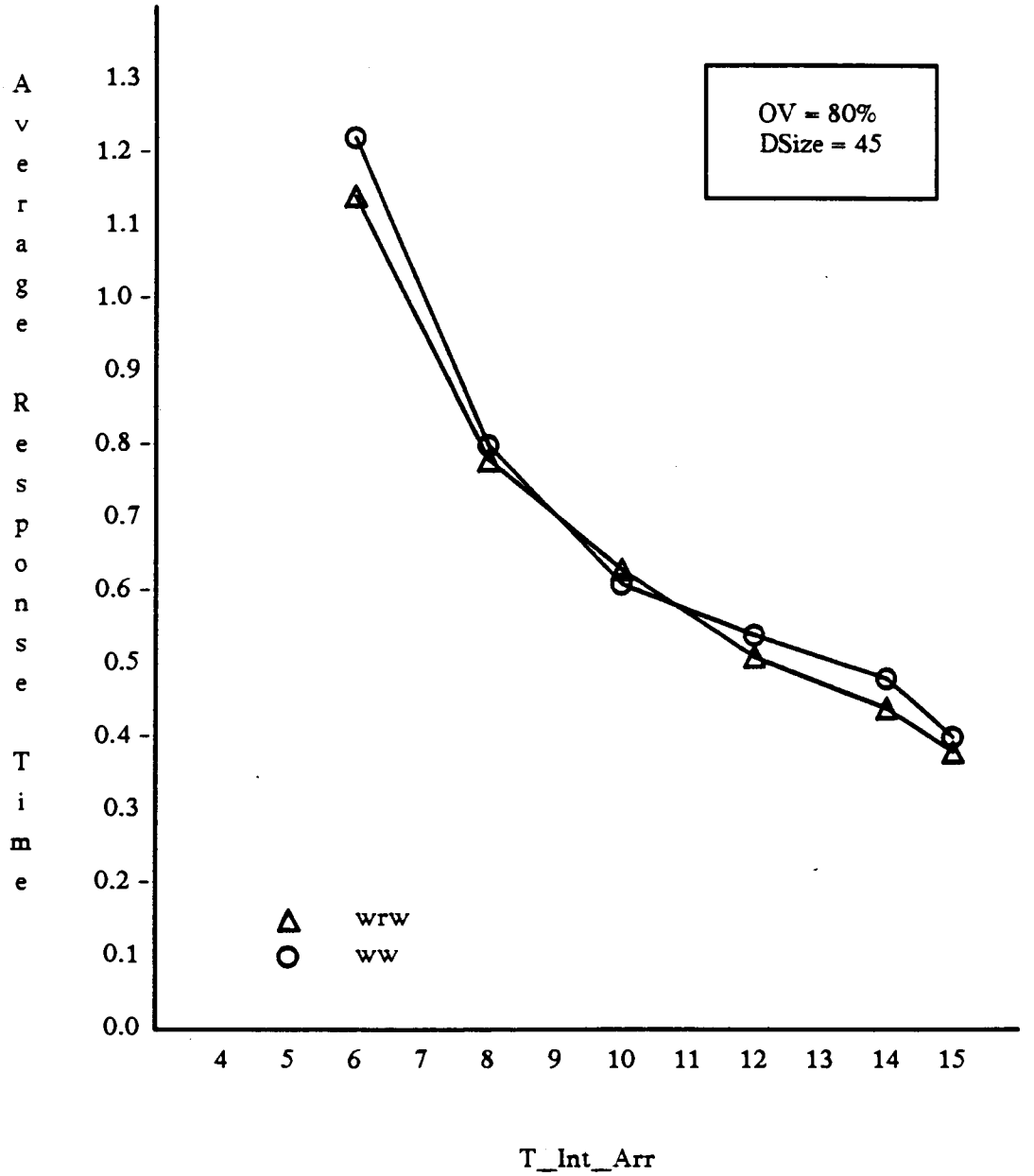


Figure 4.1. Average response time vs. mean transaction inter-arrival time.

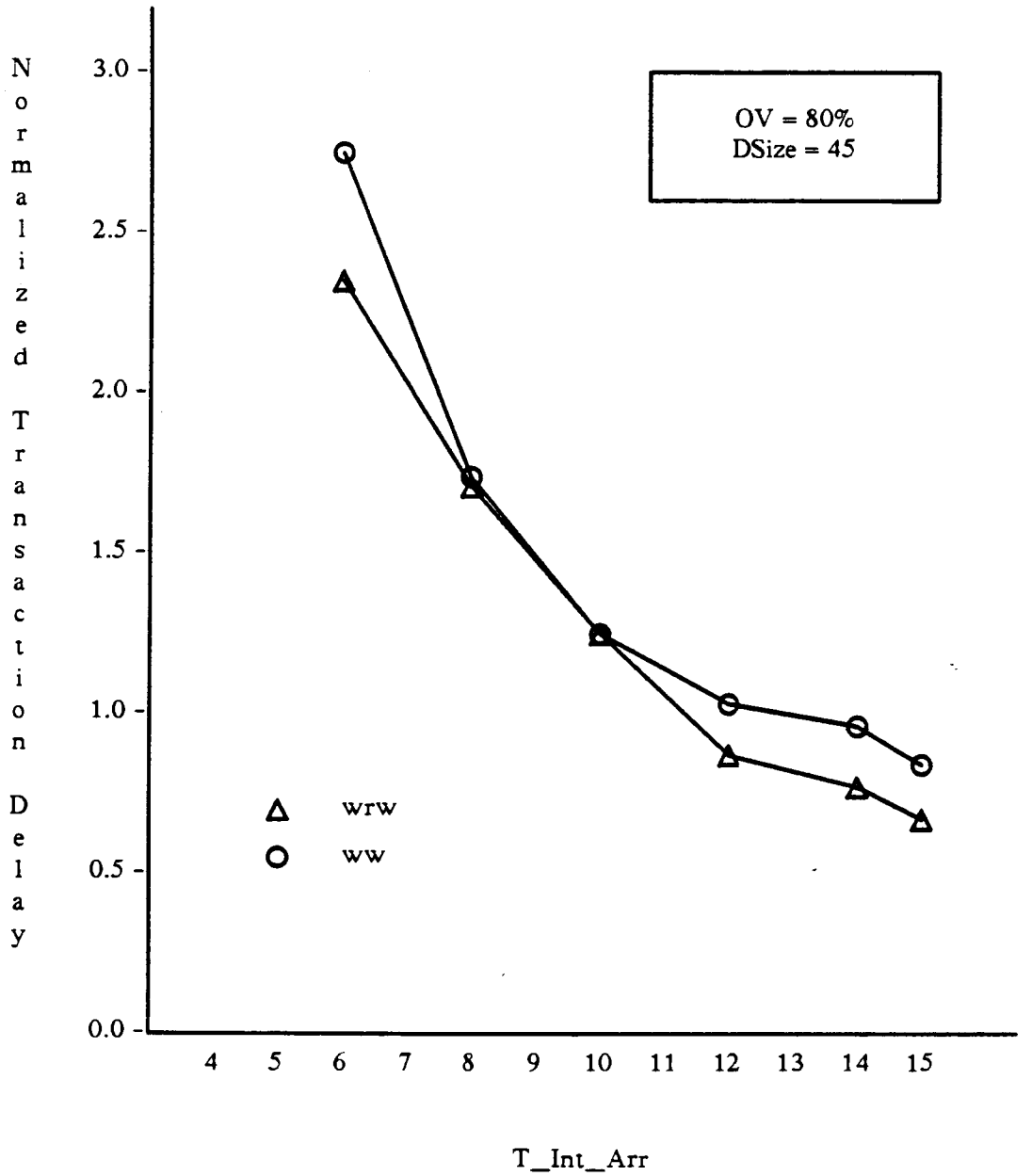


Figure 4.2. Normalized transaction delay vs. mean transaction inter-arrival time.

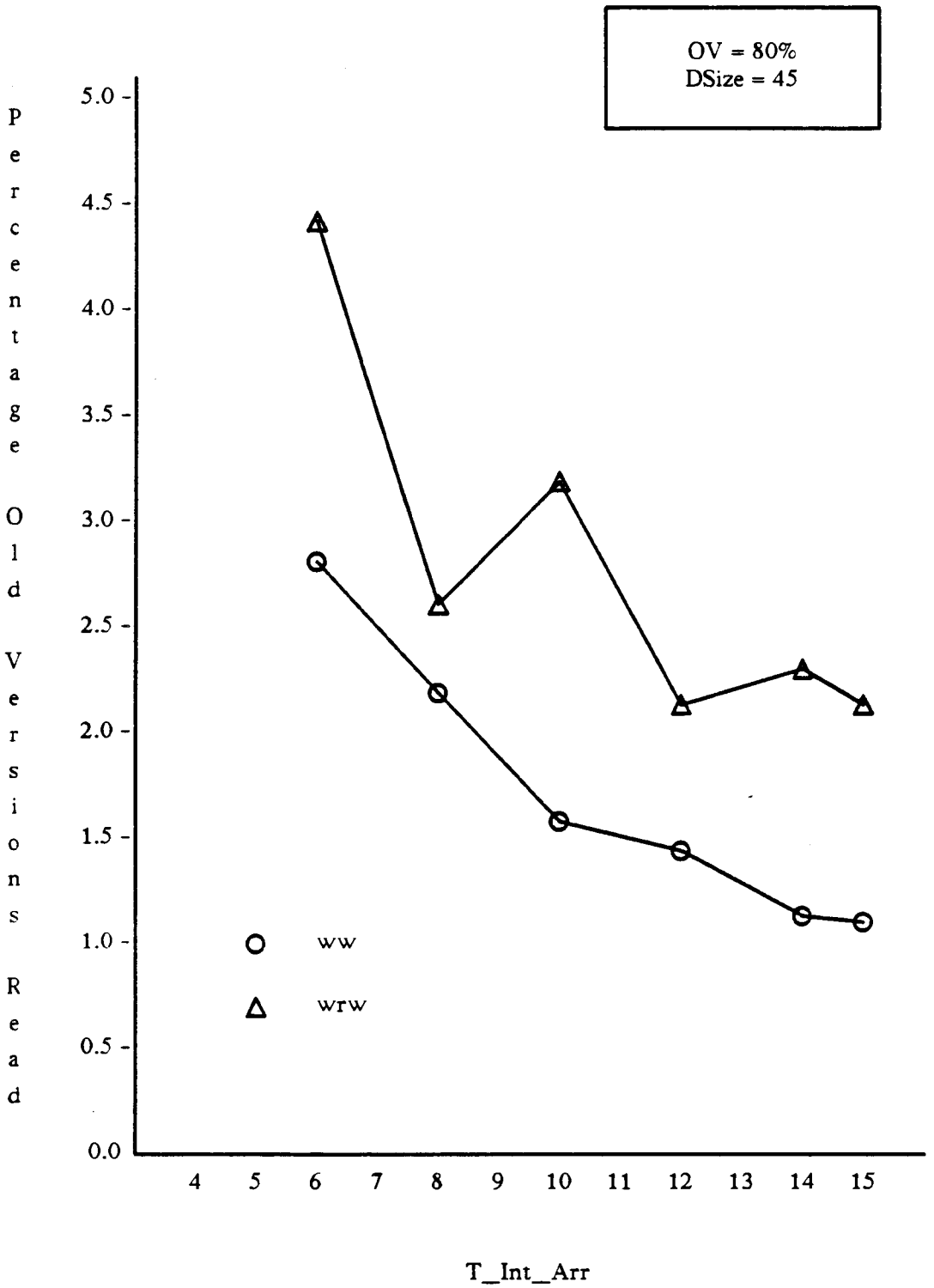


Figure 4.3. Percentage of old versions read vs. mean tansaction inter-arrival time.

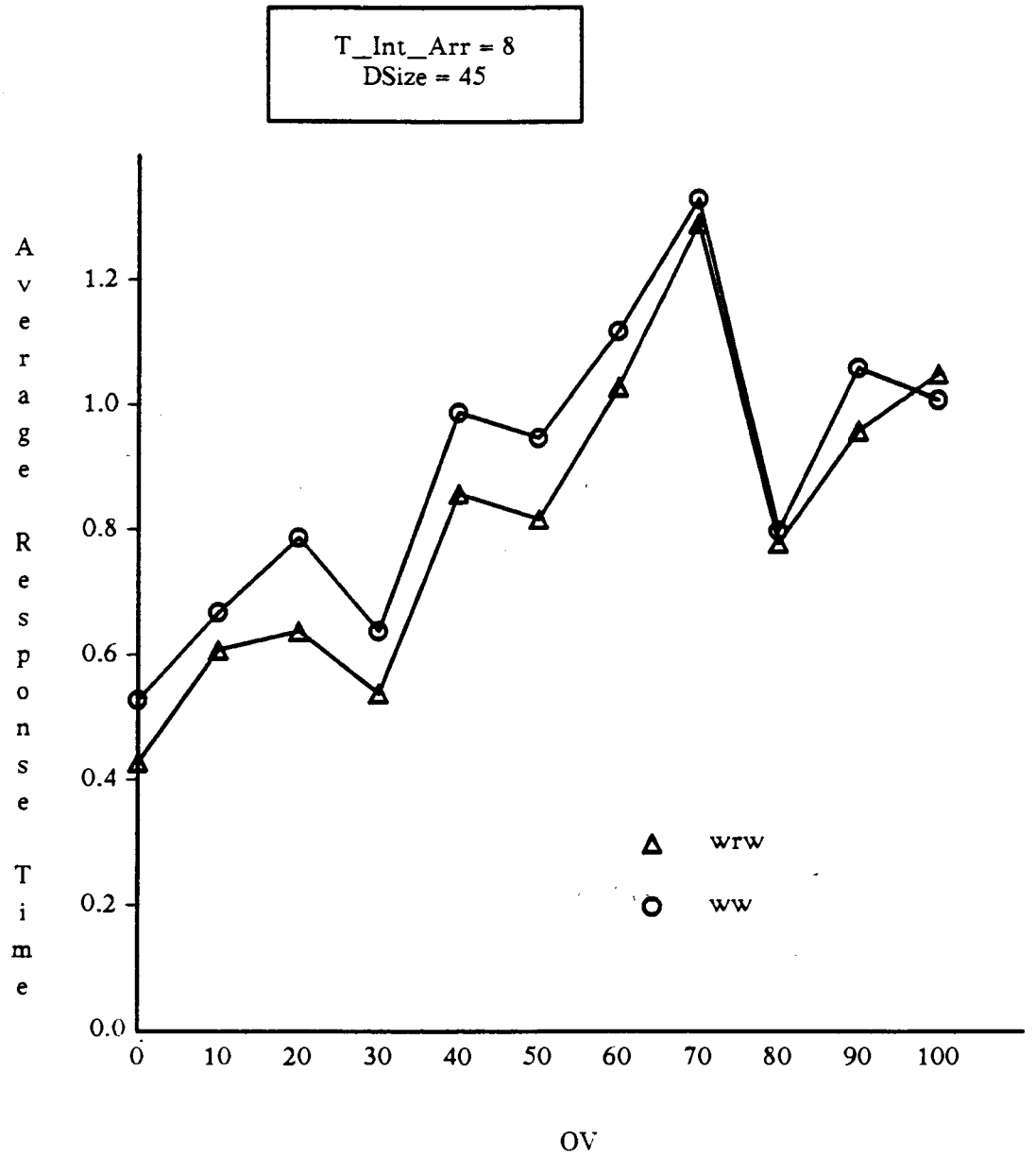


Figure 4.4. Average response time vs. overlap of writeset with readset.

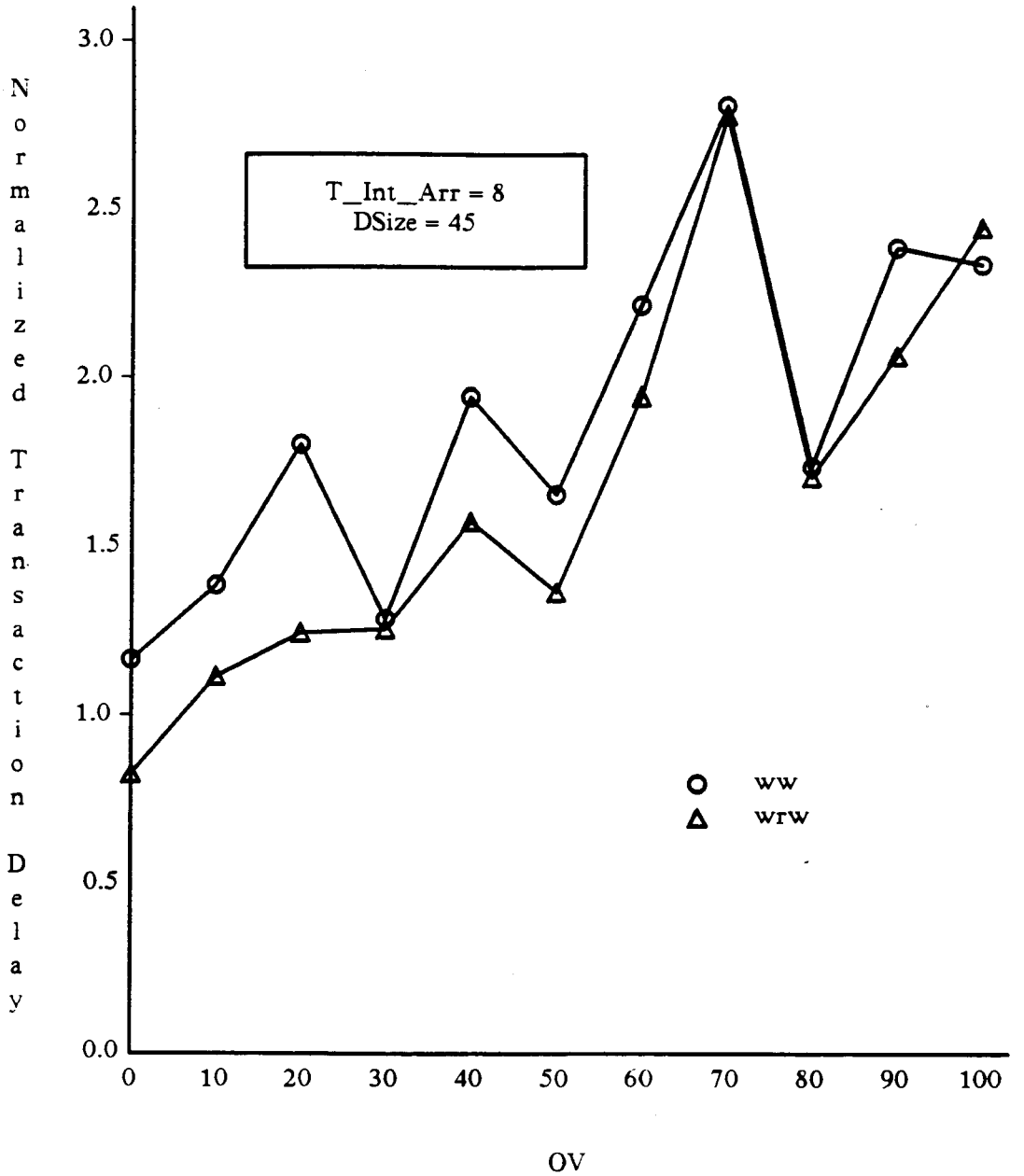


Figure 4.5. Normalized transaction delay vs. overlap of writeset with readset.

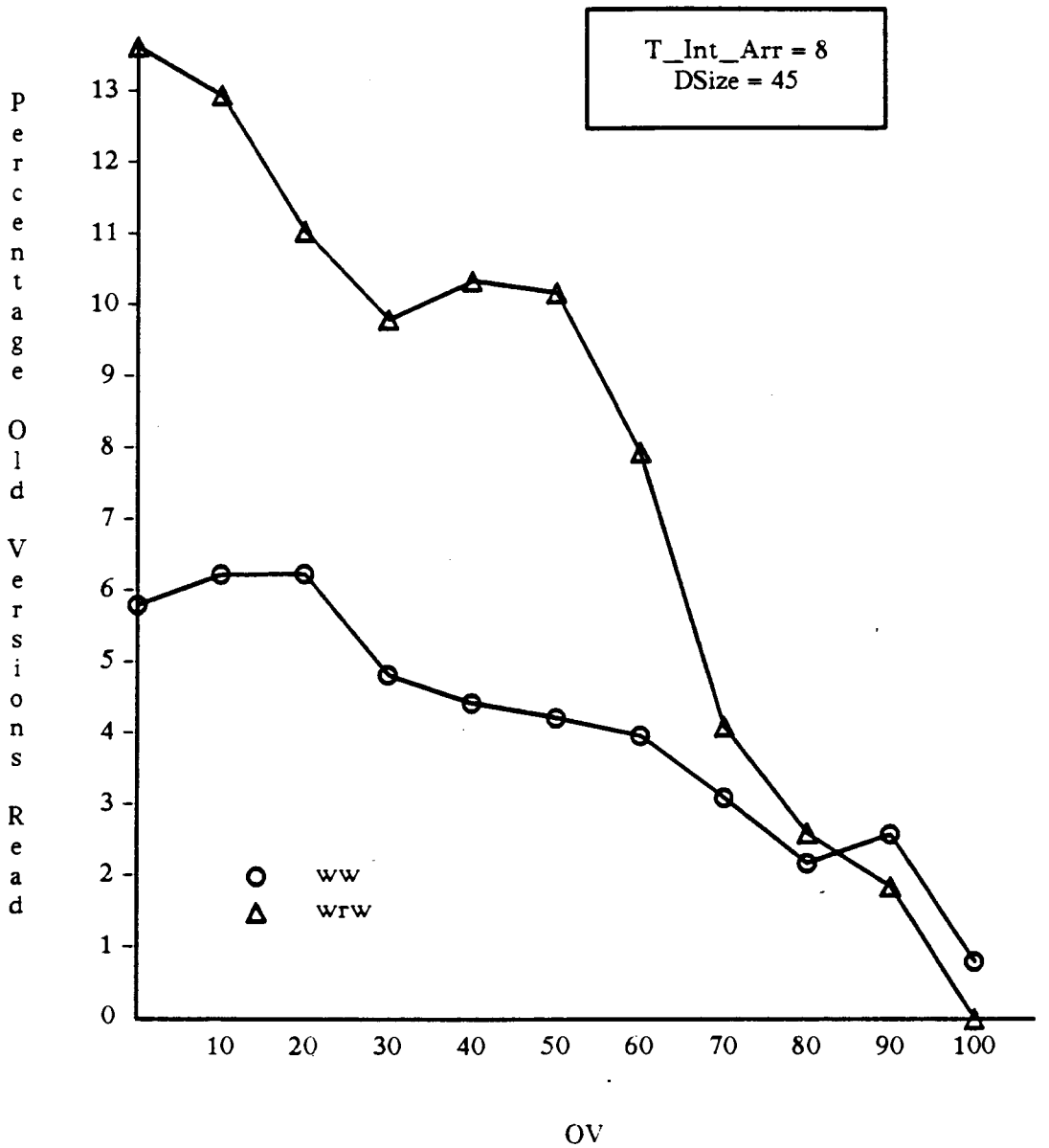


Figure 4.8. Percentage of old versions read vs. Overlap of writeset wrt readset

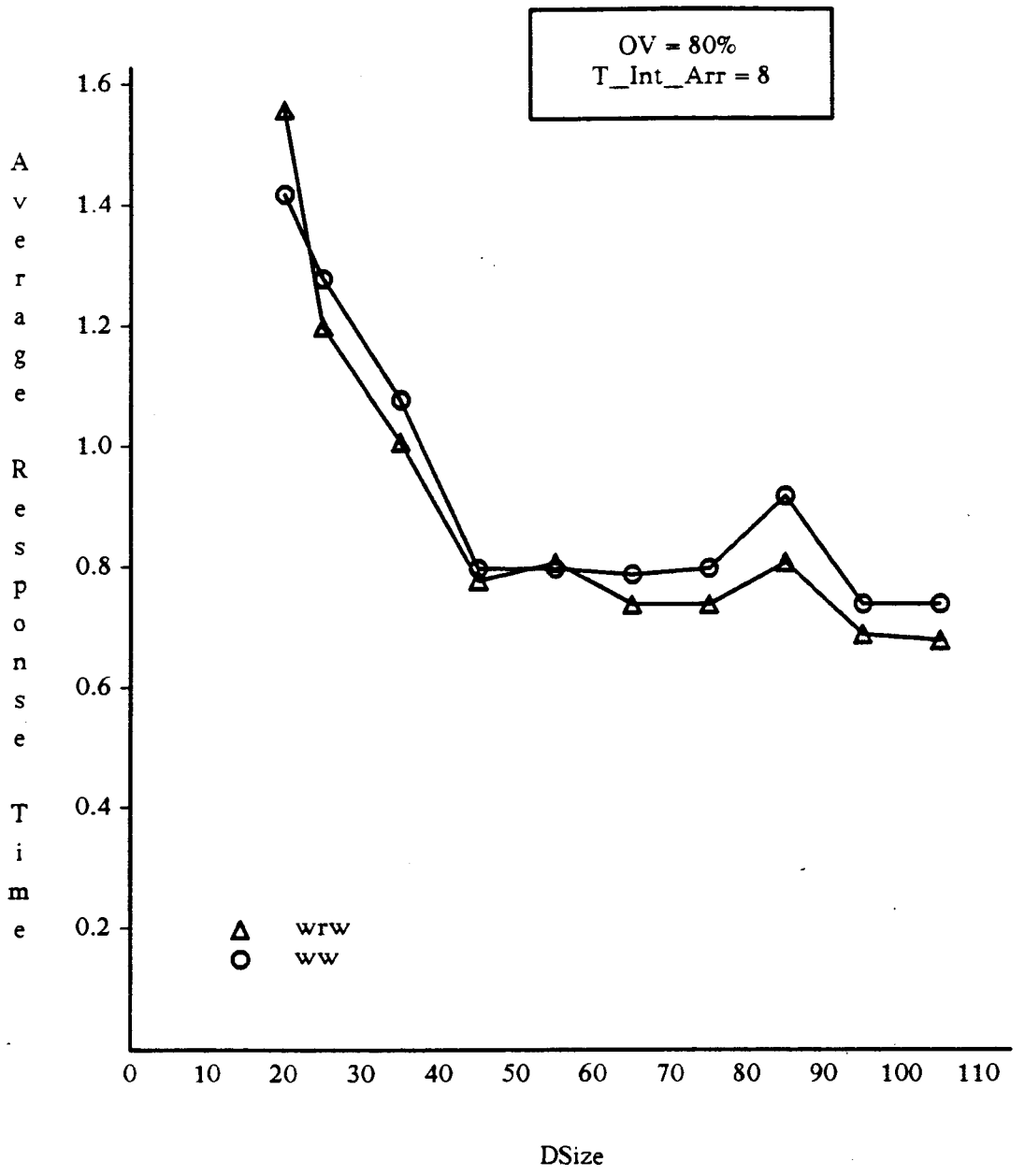


Figure 4.9. Average response time vs. database set size.

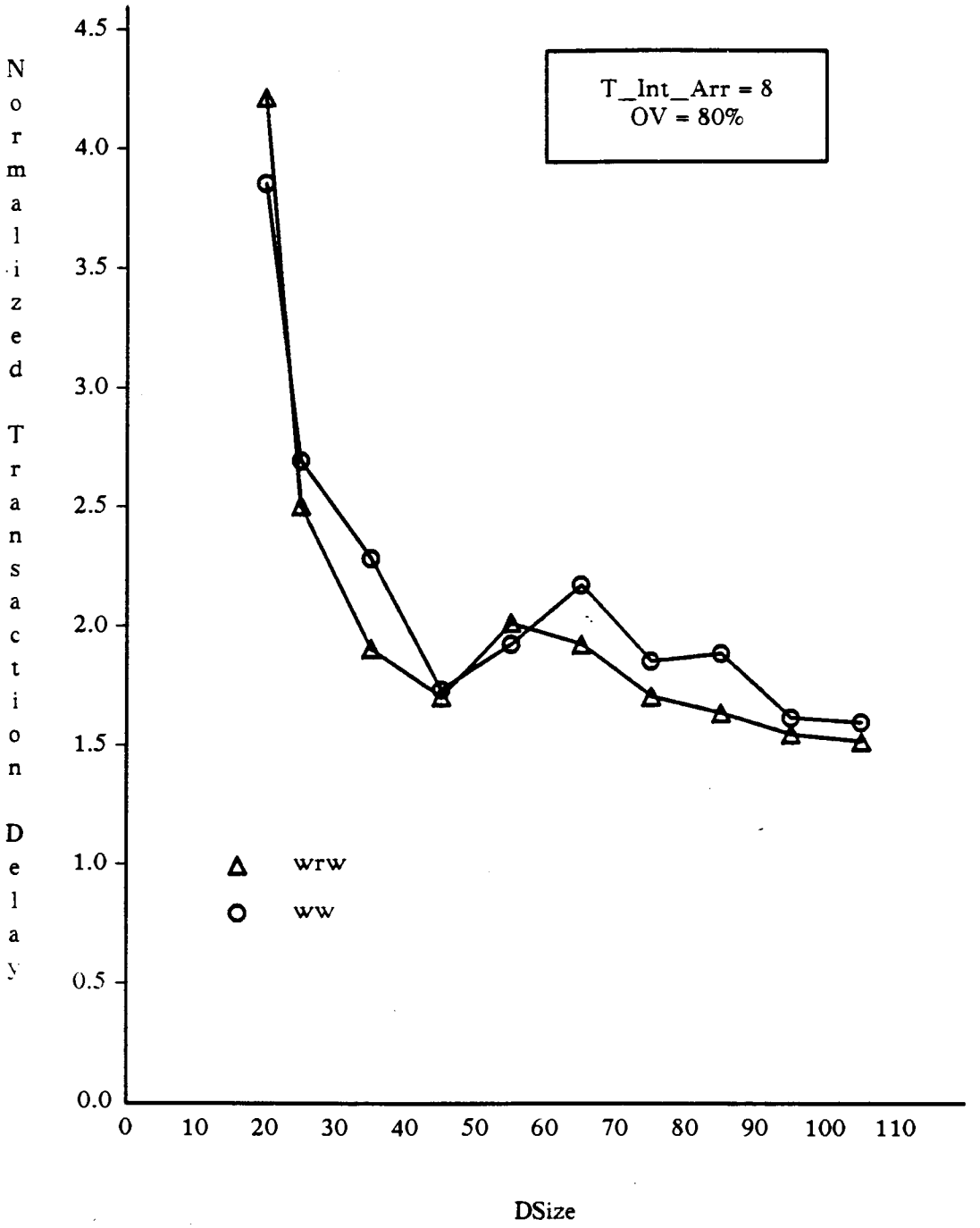


Figure 4.10. Normalized transaction delay vs. database set size.

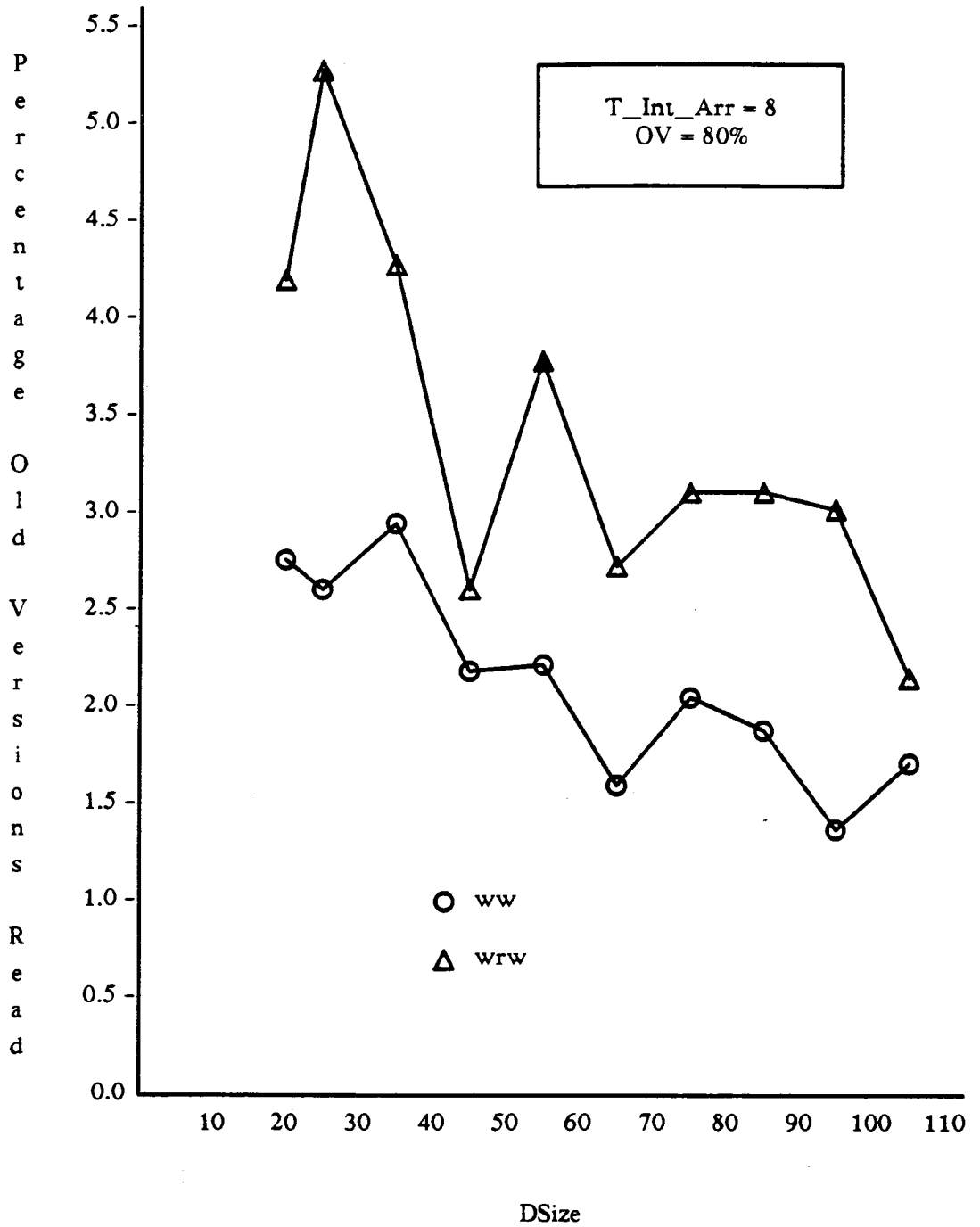


Figure 4.11. Percentage of old versions read vs. database set size.

CHAPTER 5

LIMITED VERSION SCHEDULING

As we have mentioned earlier, the completion test is the most crucial step in cautious scheduling. If an unbounded number of versions are available, the completion test for class *MWRW* (or *MWW*) simply entails checking for the acyclicity of the exclusion-closed, *wrw*-augmented (or *ww*-augmented) ATIO graph [IKK86]. This follows from the following theorem.

Theorem 5.1. [IKK86]. *Let c be a set of constraints of the sort defined in Chapter 2 and let MC denote the set of logs which is serializable under c. A partial schedule* $<P, I>$, *where* $I \in LAST_P^{\infty}$, *together with the current request q and a set of pending steps PEND has a completion in MC iff* $ATIO_c(<P, I>, q, PEND)$ *has a DITS.* □

Suppose the *c*-augmented ATIO graph referred to in the above theorem has a DITS. The serial schedule corresponding to such a DITS order is called an **eligible serial schedule with respect to** $ATIO_c(<P, I>, q, PEND)$. According to the above theorem, if an unbounded number of versions are allowed, a completion in class *MC* exists if and only if an eligible serial schedule exists. The proof for Theorem 5.1 in [IKK86] shows that every eligible serial schedule is equivalent to some completion. However, for a fixed K, no K-version completion (i.e., a completion in a K-version system) may exist even if the ATIO graph has a DITS. In other words, an eligible serial schedule may not be equivalent to any K-version completion. Additional conditions must be satisfied by an eligible serial schedule to guarantee that only the available K versions are assigned to current and pending read requests. It is mentioned in [IKK86] that $c = ww$ and *wrw* are the only practical cases. Hence, in this thesis, we discuss only classes *MWW* and *MWRW*. A completion in class *MC* will be called an ***MC* -completion.**

In this chapter, we shall discuss the additional conditions to guarantee the existence of a K-version completion as well as algorithms for K-version completion tests for classes *MWW* and *MWRW*. Basically, we add extra constraints in order to make sure that versions older than the Kth are not assigned to read requests. In the *MWW* scheduler, it is possible to represent the condition precisely by means of some *wr* constraints, whereas in the *MWRW* scheduler, we need to use a subset of *ww* constraints as a sufficient condition, thus imposing more constraints than is necessary to have a K-version completion in *MWRW*. We start with the simpler case, namely, K-version *MWW*-scheduling. In what follows, a partial schedule $<P, I>$ with $I \in LAST_P^K$ is called a **partial K— version schedule.**

5.1. K-version *MWW* Scheduling

In K-version *MWW* scheduling, the additional conditions are easily incorporated into the ATIO graph by additional constraint arcs that we introduce below. We shall prove that any eligible serializable schedule with respect to the modified ATIO graph is equivalent to a K-version *MWW*-completion. For each data item X, a *wr*-arc is drawn from the transaction which wrote the Kth most recent version of X to each transaction with a pending read request on X and also to the transaction which issued the current request, if it contains a read on X. (As in the unlimited-version case [IKK86], if the current request contains a read operation $R_j[X]$, an exclusion arc is drawn from T_j to T_k where $W_k[X]$ is a pending write request).

Let *MWWK* denote the class of logs serializable under the *ww* constraints in a K-version system as described above. We shall present a polynomial time completion test for class *MWWK*.

5.1.1. Algorithm *MWWK*-Complete

Input: A partial K-version schedule $<P, I>$, where $I \in LAST_P^K$, the current request q, the set *PEND* of pending steps, and the exclusion-closed active TIO graph $ATIO^*_{wwk}(<P, I>, q, PEND)$.

Output: A *MWWK*-completion of Pq, if one exists; otherwise an indication that the test has failed.

(1) [Initialization] For each data item X, draw a *wr*-arc in the input ATIO graph from the transaction which wrote the Kth last version of X (if it exists) in P to each transaction with a pending or current read on X, and take the exclusion closure of the graph. Let $\tau = T_0$, $t := \sigma(T_0)$, where $\sigma(T_i)$ denotes the sequence of steps forming T_i, and remove T_0 and all dummy nodes, together with the arcs incident on them, from $ATIO^*_{wwk}(<P, I>, q, PEND)$.

(2) If the set of transactions which form τ is equal to **T**, then go to step 5.

(3) For each source node T_i in the truncated $ATIO^*_{wwk}(<P, I>, q, PEND)$, extend τ and t by $\tau = \tau T_i$ and $t = t\sigma(T_i)$, and remove T_i, its dummy node T'_i, together with all the arcs incident on them, from the graph.

(4) If τ and t did not change in step 3, then stop (Pq has no *MWWK*-completion); else go to step 2.

(5) Completion test is successful and $QR_f[\mathbf{D}] = t/(Pq)$, where $t/(Pq)$ denote the sequence obtained from t by removing the steps belonging to Pq. For each $R_k[X] \in q$, draw a reads-from arc (together with their exclusion arcs) from the last transaction preceding T_k in the serial order specified by t that has written X. Stop.

In the remainder of this section we shall prove that the above completion test is correct. The correctness proof will have to show that if the test stops in step 5, then $PqQR_f[\mathbf{D}]$, where Q is generated by the above algorithm, indeed belongs to *MWWK*, and if it stops in step 4, then $<P, I>$ has no *MWWK*-completion starting in Pq.

Theorem 5.2 *Algorithm MWWK-Complete is correct and runs in polynomial time.*

Proof: The algorithm obviously runs in polynomial time since its most time-consuming part is the computation of exclusion closure in step 1. This entails updating the existing exclusion-closed graph upon the addition of a new edge. This is basically an incremental transitive closure problem which can be done in $O(N^2)$ where N is the number of nodes in the ATIO graph

[IKK86].

To prove that the algorithm is correct, we first show that if it stops in step 5 then $ATIO_{ww}(<P, I>, q, PEND)$ has a DITS. We claim that τ (produced by the above algorithm) is a DITS order for $ATIO_{ww}(<P, I>, q, PEND)$. To prove this claim, arrange the nodes of this ATIO graph linearly from left to right in the order given by τ. The dummy nodes, if any, are placed immediately to the right of their "parent nodes". It is clear from the way τ was constructed (i.e., topological sort in step 3) that all arcs (both reads-from and constraint arcs) are directed from left to right in this linear order. We must now show that no two intervals (Definition 2.1) labeled by the same data item overlap. Consider any reads-from arc due to a read operation on X. It cannot overlap with another thick arc (unless they belong to the same interval), since the *ww*-arcs and the exclusion arcs in $ATIO^*_{ww}(<P, I>, q, PEND)$ force such intervals to be ordered serially in any topological sort.

To complete the proof, we must show that there exists an interpretation for $qQR_f[\mathbf{D}]$ which maps each pending read to a version not older than the Kth last in P. Such an interpretation is defined by assigning to each pending or current read operation $R_j[X]$, the version of X created by the last transaction with a write operation on X which precedes T_j in the transaction order given by τ. In order to realize this version assignment, the scheduler can "force" the pending steps into the sequence Q. That is to say, if the arrival sequence of the pending steps does not agree with Q, then the scheduler can delay some steps as necessary. Thus, if τ dictates that a pending read $R_j[X]$ be assigned a version to be created by a pending write request, it will be assigned the most recent version available at the time it is granted. On the other hand, if τ dictates that $R_j[X]$ be assigned a version created by an already granted write operation, then the extra *wr*-arcs that we introduced in step 1 ensure that this version is not older than the Kth most recent. It is clear that such a reads-from arc cannot overlap any other reads-from arc.

Lastly, it is easy to show that if the algorithm stops at step 4, no *MWWK*-completion can

exist. This is so because the algorithm stopping at step 4 means that there are no more source nodes in the ATIO graph, i.e., it has a cycle. This implies that the serialization constraints being imposed conflict with one another, and hence there can be no DITS. □

5.2. *K*-version *MWRW* Scheduling

Unfortunately, the modifications to the completion test required for the limited version *MWRW* scheduler is not as straightforward as that for the *MWWK* scheduler. We cannot simply add a constraint arc from the Kth newest write on each data item X to the current and pending read requests on X because it doesn't guarantee that during the topological sort, older (than K) versions are not ordered between the Kth newest and the current read request. This possibility now arises because *ww* constraints are no longer imposed. Hence, some more conditions will have to be introduced in order to find a K-version completion in *MWRW*. We first prove a necessary and sufficient condition for an eligible serial schedule with respect to the *wrw*-augmented ATIO graph to be equivalent to a K-version *MWRW*-completion (Theorem 5.3). Some implications of the necessary and sufficient condition will be proved in Lemmas 5.1-5.3. Based on these lemmas, we introduce a set of new constraints that are sufficient to guarantee the two conditions in Theorem 5.3.

Theorem 5.3. *Let* $<h, I^*>$ *be an eligible serial schedule with respect to a given* $ATIO_{wrw}(<P, I>, q, PEND)$, *where* $<P, I>$ *is a K-version partial schedule. There exists a K-version completion in MWRW which is equivalent to* $<h, I^*>$ *iff the following two conditions are satisfied for each current or pending read operation* $R_j[X]$ *such that* $I^*(R_j[X]) = W_i[X]$ *in h.*

(A) Either $W_i[X] \in PEND$ *or* $W_i[X]$ *is one of the last K write operations on X in P.*

(B) If $W_i[X]$ *is the nth* ($1 \leq n \leq K$) *last write operation on X in P, then no more than K-n write operations on X in PEND precede* $W_i[X]$ *in h.*

Proof: The necessity of condition (A) follows from the definition of a K-version completion. The necessity of condition (B) can be proved by contradiction. Let $<PqQR_j[D], I'>$ be a K-version

completion *wrw*-equivalent to $<h, I^*>$, i.e., $TIO_{wrw}(<PqQR_f[\mathbf{D}], I'>) = TIO_{wrw}(<h, I^*>)$. Suppose more than K-n write operations on X in *PEND* precede $W_i[X]$ in h. If such a write $W_k[X]$ occurs after $R_j[X]$ in Q, then there'll be an rw-arc from T_j to T_k in $TIO_{wrw}(<PqQR_f[\mathbf{D}], I'>)$. Since there is a wr-arc from T_k to T_j in $TIO_{wrw}(<h, I^*>)$ and it is cycle-free, this contradicts the assumption that $<PqQR_f[\mathbf{D}], I'> \equiv_{wrw} <h, I^*>$. Therefore, all such write operations precede $R_j[X]$ in Q. Since $W_i[X]$ is in P, and there are more than K-1 writes on X between $W_i[X]$ and $R_j[X]$ in $PqQR_f[\mathbf{D}]$, T_j cannot read from T_i in a K-version database, thus contradicting the assumption.

We now prove the sufficiency, that is, if an eligible serial schedule with respect to the ATIO graph, $<h, I^*>$, satisfies (A) and (B), then there is a K-version completion equivalent to $<h, I^*>$. We claim that $<P(h/P), I'>$ is a desired K-version completion to $<P, I>$, where h/P denotes the sequence obtained from h by deleting the steps belonging to P from it, and I' is an extension to I such that for $R_j[X] \in h/P$, $I'(R_j[X]) = W_i[X]$ if and only if $I^*(R_j[X]) = W_i[X]$ in h. To prove this claim, we first show that $TIO_{wrw}(<h, I^*>)$ and $TIO_{wrw}(<P(h/P), I'>)$ have the same reads from arcs as well as rw- and wr- arcs. By definition of I', they share the same reads-from arcs. Each *wrw*-arc in $TIO_{wrw}(<h, I^*>)$ is due to a pair of operations, a write operation $W_i[X]$ and a read operation $R_j[X]$ in h, hence in $P(h/P)$ because $P(h/P)$ contains the same set of operations as h. If one of these operations belongs to P, then such a *wrw*-arc already exists in $ATIO_{wrw}(<P, I>, q, PEND)$ and both $TIO_{wrw}(<h, I^*>)$ and $TIO_{wrw}(<P(h/P), I'>)$ simply "inherit" it. If both belong to h/P, then clearly both $TIO_{wrw}(<h, I^*>)$ and $TIO_{wrw}(<P(h/P), I'>)$ have the same *wrw*-arc. What remains to be shown is that, if $I'(R_j[X]) = W_i[X]$ for a $R_j[X] \in h/P$, then there are no more than K-1 intervening writes on X between $W_i[X]$ and $R_j[X]$. If $W_i[X] \in PEND$, then obviously $R_j[X]$ follows $W_i[X]$ in h/P with no intervening writes on X. If $W_i[X]$ is the nth last write in P, by condition (B), there are no more than $K-n$ pending writes on X preceding $R_j[X]$ in h. Therefore, h/P has no more than $K-n$ writes on X preceding $R_j[X]$ and hence, there are at most

$(n-1)+(K-n)=K-1$ writes on X between $W_i[X]$ and $R_j[X]$ in $P(h/P)$. □

With this theorem, a possible K-version completion test would test each eligible serial schedule against conditions (A) and (B) of Theorem 5.3. If we find one eligible serial schedule which satisfies both, then we know that there is a K-version completion in *MWRW*. Unfortunately, this approach can be very time-consuming. In fact, *WRW*-completion test is, in general, NP-complete [KKI86], even for single version ($K = 1$) systems. To achieve an efficient K-version completion test, we shall impose some additional constraints on the serialization order.

Before specifying exactly what these additional constraints are, let us first define some concepts which will be helpful in justifying their introduction.

Let $<P, I>$ be a partial K-version schedule, q be the current request, and *PEND* be a set of pending steps. For each $X \in \mathbf{D}$, the last K write operations, if any, on X in P are said to be **ripe.** ***Ripe(X)*** denotes the set of the ripe write operations on X.

We call a write operation $W_i[X]$ **fresh** if either it is in *PEND* or it is ripe. Fresh writes are those which can potentially be assigned to the current or pending reads. More formally, the set of fresh write operations on X can be defined as:

$$FRESH(X) = \{W_k[X] \mid W_k[X] \in PEND\} \cup Ripe(X)$$

We call a write operation **old**, if it is not in *PEND*. More formally, the set of old write operations can be defined as:

$$OLD(X) = \{W_k[X] \mid W_k[X] \notin PEND\}$$

Note that a ripe operation is both fresh and old. In order to define a "readable" write operation on X, we introduce a fictitious read-only transaction T_x with only one read operation $R_x[X]$, and we let $\mathbf{T} = \mathbf{T} \cup \{T_x\}$ and $PEND = PEND \cup \{R_x[X]\}$. $W_i[X]$ is **unreadable** if no K-version completion in *MWRW* has interpretation which maps $R_x[X]$ to $W_i[X]$.

Although one might suspect no *fresh* write operation to be unreadable, this is not so because of condition (B) of Theorem 5.3. Such exceptions can be discussed in terms of a *spoiler*. A **spoiler**

is defined as a pair of write operations, $W_h[X] \in FRESH(X)$ and $W_k[X] \in OLD(X)$, such that there exists a path from T_h to T_k in the ATIO graph. Such a pair is a member of ***SPOILER(X)*** and $W_h[X]$ and $W_k[X]$ are called the **fresh end** and **old end** of the spoiler, respectively.

Lemma 5.1. *The fresh end of any spoiler is unreadable.*

Proof: Suppose there is a completion with an interpretation which maps the pending fictitious read, $R_x[X]$, to the fresh end of a spoiler $(W_h[X], W_k[X]) \in SPOILER(X)$. The wr-arc from T_k to T_x induces an exclusion arc from T'_k (which reads X from T_k) to T_h, thus creating a cycle and hence no DITS can exist. (Fig. 5.1) □

Lemma 5.2. *Let* $(W_h[X], W_k[X]) \in SPOILER(X)$ *and let* $W_i[X]$ *be the nth* $(1 \leqslant n \leqslant K)$ *last write on* X *in* P. $W_i[X]$ *(*$k=i$ *possible) is unreadable if there are more than* $K-n$ *spoilers in* $SPOILER(X)$ *whose fresh ends are pending.*

Proof: The fresh ends of all spoilers are unreadable (Lemma 5.1), so assume $W_i[X]$ is not a fresh end. We claim that in no K-version completion does the fictitious read $R_x[X]$ appear before the fresh end of any spoiler. To prove this, assume $R_x[X]$ precedes $W_h[X]$ in a K-version completion. Then the rw-arc from T_x to T_h and the wr-arc from T_k to T_x, together with a path from T_h to T_k, create a cycle. Hence all such writes must precede $R_x[X]$. If there are more than $K-n$

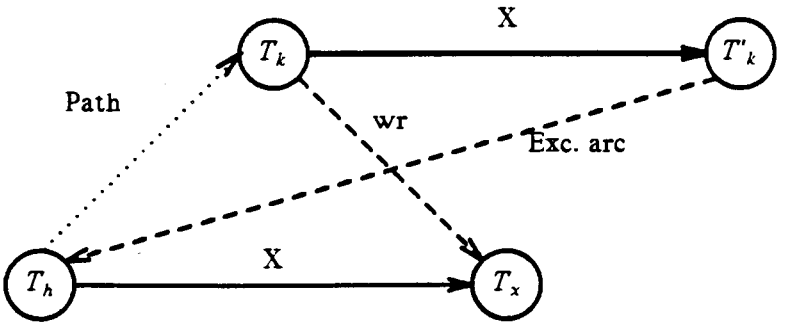


Figure 5.1. Illustration for Proof of Lemma 5.1.

spoilers in $SPOILER(X)$ whose fresh ends are pending, then this makes $W_i[X]$ unreadable (Theorem 5.3(B)) because there are now more than K-1 writes between $W_i[X]$ and $R_x[X]$. □

We can now specify the set of unreadable write operations. If $SPOILER(X) = \emptyset$, then $\boldsymbol{UNR(X)} = OLD(X) - Ripe(X)$. Otherwise, Lemmas 5.1 and 5.2 specify the additional unreadable operations in $UNR(X)$.

Let us call a write operation that is not in $UNR(X)$ **potentially readable** and let $\boldsymbol{RBL(X)}$ denote the set of all potentially readable writes on X. We use the term "potential", because even if assigning a potentially readable write operation to a pending read does not directly cause a cycle in the graph, there maybe no K-version completion.

As mentioned earlier, we shall impose additional constraints in the form of a subset of *ww* constraints to make the K-version completion test more efficient. A similar type of constraint was also imposed by [KKI86] in their single version *WRW*-scheduler.

Definition 5.1. Let *wrw*k* denote the set of *wrw* constraints augmented by the following subset of *ww* constraints. The *ww* constraints used are of the form "T_k must be serialized before T_i" if:

a) $W_k[X] \in UNR(X)$ and $W_i[X] \in RBL(X)$, or

b) $W_k[X] \in Ripe(X) \cap RBL(X)$ and $W_i[X] \in RBL(X) - Ripe(X)$.

Let $MWRW^*K$ denote the set of logs that are K-version serializable under these constraints.

The completion test, then, is to determine if there is a serial schedule $<h, I^*>$ satisfying the relation $<PqQR_f[\mathbf{D}], I> \equiv_{wrw^*_k} <h, I^*>$ for some Q over $PEND$, where $I \in LAST^K_{PqQR_f[\mathbf{D}]}$. Obviously, if the exclusion closed *wrw*k*—augmented ATIO graph has a cycle, then no such $<h, I^*>$ exists. It is important to remember that unlike the *wrw*-arcs, the *ww*-arcs due to Definition 5.1 are not permanent in the ATIO graph but are deleted after a request is processed.

Intuitively, what these added constraints do is to make sure that all unreadable writes precede readable ones so that during the construction of a serial schedule in the completion test, once

a readable write on X is in the partially formed serial schedule, we are assured that any subsequent pending read on the same data item will be able to read from a potentially readable write (Definition 5.1 (a)). Furthermore, it makes sure that all potentially readable pending writes on X appear after any ripe writes on X (Definition 5.1 (b)).

Let us define $\boldsymbol{RBL}_j(X) \subset RBL(X)$ as the set of write operations $W_i[X]$ which are potentially readable by a pending read $R_j[X]$ or final read $R_f[X]$ without creating a cycle in the wrw^*k-augmented ATIO graph. In order words,

$RBL_j(X) = \{W_i[X] \in RBL(X) \mid T_i$ *is not a descendant of* T_j *in the* $ATIO^*_{wrw^*k}(<P, I>, q, PEND)\}$.

Although it is necessary that $RBL_j(X) \neq \emptyset$ for each pending and final read operation $R_j[X]$ and $R_f[X]$, it is not sufficient because of condition (B) of Theorem 5.3. As an example, consider the following 3-version partial schedule.

Example 5.1.

$$<P, I> = W_0[D]\ R_1[X_0,Y_0]\ R_2[V_0,U_0]\ W_1[Z]\ R_3[Z_0]\ W_4[Z]\ W_2[U,V]\ R_5[U_2]\ R_6[U_2]\ R_3[V_2]\ W_4[X]$$

$$q = W_4[Y]$$

$$PEND = \{W_1[X,Y],\ W_2[X,Y],\ W_3[X,Y],\ W_4[X,Y],\ R_5[X],\ W_5[Y],\ R_6[Y],\ W_6[X]\}.$$

The wrw^*k—augmented ATIO graph is illustrated in Fig. 5.2. The potentially readable writes for the six read operations not in P are: $RBL_5(X) = RBL_f(X) = \{W_6[X]\}$, $RBL_6(Y) = RBL_f(Y) = \{W_5[Y]\}$, $RBL_f(Z) = \{W_4[Z]\}$, and $RBL_f(V) = \{W_2[V]\}$. Note that the pending writes of T_1, T_2, and T_3 are not potentially readable because of Lemma 5.1, and $W_4[X]$ and $W_4[Y]$ are also not potentially readable because of Lemma 5.2. Although there is a potentially readable write for each pending and final read operation, there is no serial schedule that satisfies condition (B) of Theorem 5.3 for all $X \in \mathbf{D}$. To see this, consider the only two possible transaction orderings, $T_0T_1T_2T_3T_4T_5T_6T_f$ and $T_0T_1T_2T_3T_4T_6T_5T_f$: the first (resp. the second) ordering does not satisfy condition (B) for $R_5[X]$ (resp. $R_6[Y]$). □

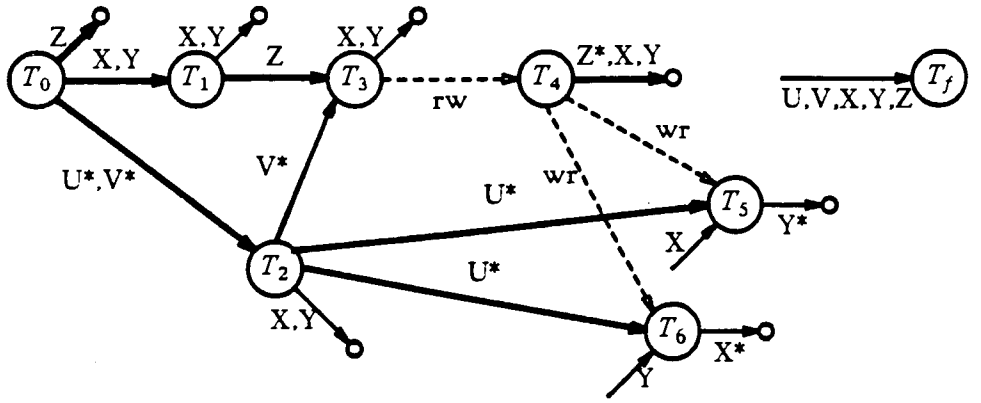


Fig. 5.2. Illustration for Example 5.1.
(* indicates a readable write)

Now, we present a polynomial time algorithm for $MWRW^*K$-completion test. The approach used here is to try to construct a serial schedule h such that $<P(h/P), I'> \equiv_{wrw^*k} <h, I^*>$, where, as a function, I' is the same as I^*.

The construction is very similar to the one used in $MWWK$ scheduling. Recall the transaction sequence τ and partial log t defined in the completion test for $MWWK$. In testing for $MWRW^*K$-completion, we will select a source node, T_i, in the truncated ATIO graph such that t contains at least one write operation from $RBL_i(X)$ for all $X \in D$ such that $R_i[X] \in q \cup PEND \cup \{R_f[\mathbf{D}]\}$.

This condition can be more formally restated as

$$NEED(T_i) \subseteq D_{RBL}(t)$$

where

$$D_{RBL}(t) = \{X \in \mathbf{D} \mid t \text{ contains a write operation} \in RBL(X)\}$$

and

$$NEED(T_i) = \{X \in \mathbf{D} \mid R_i[X] \in q \cup PEND \cup \{R_f[\mathbf{D}]\}$$

If we find such a node, T_i, we append T_i to τ and the steps of T_i to t, and delete T_i as well as all the arcs incident on it from the wrw^*k-augmented ATIO graph. If we are not able to append all

the transactions in **T** to τ, then the completion test fails. Otherwise, the resulting $<t, I^*>$ gives the serial schedule which is *wrw*-equivalent to a K-version completion of $<P, I>$.

Algorithm *MWRW*K*-Complete

Input: Partial K-version schedule $<P, I>$, current request q, set *PEND* of pending requests, and ATIO graph $ATIO^*_{wrw^*k}(<P, I>, q, PEND)$, and set **T** of currently known transactions.

Output: A completion of $<P, I>$ in *MWRW*K* if one exists; otherwise an indication that the test has failed.

(1) [Initialization] Compute $RBL(X)$ for all $X \in \mathbf{D}$ and $RBL_j(X)$ for each read operation $R_j[X] \in q \cup PEND \cup \{R_f[\mathbf{D}]\}$. If $RBL_j(X) = \emptyset$ for some current or pending $R_j[X]$, or if $ATIO^*_{wrw^*k}(<P, I>, q, PEND)$ has a cycle, then stop (P has no completion); else, let $\tau := T_0$ and $t := \sigma(T_0)$. Compute $NEED(T_i)$.

(2) If all the transactions in **T** appear in τ, then go to step 6.

(3) Compute $D_{RBL}(t)$.

(4) For each source node T_i in the truncated $ATIO_{wrw^*k}(<P, I>, q, PEND)$, let $\tau = \tau T_i$ and $t = t\sigma(T_i)$ provided $NEED(T_i) \subseteq D_{RBL}(t)$. Remove T_i, its dummy node T'_i, and all the arcs incident on them from the graph.

(5) If τ and t did not change in step 4, then stop (P has no completion); else go to step 2.

(6) Completion test is successful and $<t, I^*>$ is the serial schedule *wrw*k*-equivalent to some completion of $<P, I>$ in *MWRW*K*. For each $R_k[X] \in q$, draw a reads-from arc (together with their exclusion arcs) from the last transaction preceding T_k in τ that has written X.

ANALYSIS OF ALGORITHM.

Lemma 5.4. *Given* $<P, I>$, q, *and PEND, let* $<h, I^*>$ *be an eligible serial schedule with respect*

to $ATIO^*_{wrw^*_k}(<P, I>,q,PEND)$. *If, for every* $X \in \mathbf{D}$, *the first read operation on* X *in* h *that is not in* P, $R_k[X]$, *satisfies* $I^*(R_k[X]) \in RBL_k(X)$, *then* $<h, I^*>$ *satisfies the conditions (A) and (B) of Theorem 5.3.*

Proof: Condition (A) is satisfied because if $I^*(R_k[X]) = W_i[X] \in RBL_k(X)$, then due to the extra *ww* constraints that we have imposed (see Definition 5.1 (a)), no write operation in $UNR(X)$ can follow $W_i[X]$ in h and any pending read operation $R_j[X]$ that follows $R_k[X]$ in h will read from a write in $RBL(X)$. Thus if $I^*(R_j[X])$ is ripe, then it must be the nth last write on X for some n $(1 \leqslant n \leqslant K)$.

To prove that $<h, I^*>$ also satisfies condition (B), let $R_j[X]$ be any pending read operation such that $I^*(R_j[X]) = W_i[X] \in Ripe(X)$. It was proved in the previous paragraph that $W_i[X] \in RBL_j(X)$. This implies, by Lemma 5.2, that there are in $SPOILER(X)$ no more than $K-n$ spoilers whose fresh ends are pending. Since $W_i[X]$ is ripe, due to the extra *ww*-constraints of Definition 5.1 (b), no member of $RBL(X) - Ripe(X)$ can precede $W_i[X]$ in h. Therefore, there are no more than $K-n$ pending writes on X preceding $R_j[X]$ in h.

Theorem 5.4. *Algorithm MWRW*K-Complete is correct.*

Proof: The algorithm halts either in step 1, 5, or 6. First, we show that if it halts in step 6, then the t constructed by the above algorithm is an eligible serial schedule with respect to $ATIO_{wrw^*_k}(<P, I>,q,PEND)$ which is equivalent to a K-version completion in *MWRW*. To prove that t is indeed such an eligible serial schedule, we show that τ is a DITS order for $ATIO_{wrw^*_k}(<P, I>,q,PEND)$. Arrange the nodes of the ATIO graph from left to right in the order of τ. Since only source nodes are considered in step 4, it is assured that all the arcs are directed from left to right. Furthermore, no two intervals labeled by the same data item overlap, because the *wr*, *rw*, and exclusion arcs in $ATIO_{wrw^*_k}(<P, I>,q,PEND)$ force such intervals to be ordered serially in any topological sort. Hence, $<t, I^*>$ is an eligible serial schedule with respect to $ATIO_{wrw^*_k}(<P, I>,q,PEND)$. Since t was constructed in a way that satisfies Lemma 5.4, there exists a K-version completion in *MWRW*.

Clearly, if the algorithm stops in step 1, no completion exists, since it implies that the serialization constraints conflict with one another. What remains to be proven is that that is also the case if the algorithm stops in step 5. We prove this by contradiction. Let us assume that there exists an eligible serial schedule $<h, I^*>$ which is equivalent to a K-version completion in *MWRW*. Let t be the partial schedule output so far when the algorithm stopped in step 5. If T_i is the first transaction in h, among those transactions whose steps are not in t, then T_i has a pending read $R_i[X]$ such that no write on X in t belongs to $RBL(X)$. Let h' be a prefix of h preceding the operations of T_i in h. From the definition of T_i, the transactions in h' are clearly a subset of the transactions in t. So, if t contains no element of $RBL(X)$, neither does h'. Therefore, T_i couldn't have read from any potentially readable writes in h' and this contradicts our assumption that $<h, I^*>$ is equivalent to a K-version completion in *MWRW*. □

Theorem 5.5 *Algorithm MWRW*K-Complete runs in polynomial time.*

Proof: Most of the steps in our algorithms can be executed in a straightforward manner. Computing $D_{RBL}(t)$ involves computing RBL(X) for each $X \in \mathbf{D}$, which in turn involves determination of spoilers in an exclusion-closed ATIO graph. As shown in Theorem 5.2, exclusion closure can be computed in polynomial time. Transitive closure also facilitates the determination of spoilers, since for $W_i[X] \in PEND \cup Ripe(X)$, a pair $(W_i[X], W_j[X])$ belongs to SPOILER(X) iff there is a path from T_i to T_j and $W_j[X] \in OLD(X)$. Hence, the algorithm can be executed in polynomial time. □

CHAPTER 6

CONCLUSION

In the first part of this thesis, we have examined and compared the performance of two multiversion schedulers, *MWW* and *MWRW* schedulers, assuming that an unlimited number of versions are available.

Experiment 1 examined the effect of the mean transaction inter-arrival time, *T_Int_Arr*, on the performance of the two said schedulers. It was found that for both schedulers, the average response time and the normalized transaction delay improve as *T_Int_Arr* increases. It was also seen that both schedulers tend to perform roughly equally, although the *MWRW* scheduler, in general, slightly outperforms the *MWW* scheduler. In Experiment 2, the degree of overlap of the writeset with the readset was varied from 0-100%. It was found that both schedulers perform better at lower degree of overlap. Furthermore, when the overlap is close to 100%, the *MWW*-scheduler performs better; but at lower degree of overlap, the opposite is true. Finally, Experiment 3 varied the database set size and it was found that performance of both schedulers improves generally with increase in *DSize* and that *MWRW*-scheduler slightly outperforms the *MWW*-scheduler.

Since it is not practical to maintain an unlimited number of versions, we have proposed efficient scheduling algorithms for classes *MWRW* and *MWW*, that use only a bounded number (K) of versions. Basically, we add additional constraint arcs to the ATIO graph, which ensure that the current and pending read requests always read from within the K most recent versions. In the K-version *MWW*-scheduler, a *wr*-arc is added from the Kth oldest version to each current or pending read request. In the K-version *MWRW* scheduler, a *ww*-arc is added from "unreadable" writes to "readable" ones.

Even though the *MWRW*-scheduler, in general, slightly outperformed the *MWW*-scheduler in the unbounded version case, the extra complexity of implementing a K-version *MWRW*-scheduler would add so much overhead to this scheduler that it will probably be more practical to use a K-version *MWW*-scheduler. We also know that when the overlap of the writeset with the readset is very close to 100% (which is generally the case for most real world applications), the *MWW*-scheduler performs better than the *MWRW*-scheduler.

During the entire simulation project involving more than one hundred runs, it was found that more than 90% of the versions read were the most recent version and the oldest version read ever was the 6th oldest. With these results, an obvious extension to this work is to fix the number of versions, K, for the *MWW*-scheduler at $1, 2, \cdots, 6$, *etc.*, and compare the increase in the degree of concurrency achieved as K is increased. Such results could help a database designer decide how many versions he would want to maintain in order to achieve a desired degree of concurrency. As for the *MWRW*-scheduler, it will be interesting to examine just how much more overhead, in terms of CPU time, memory, etc., the *MWRW*-scheduler incurs. Finally, it is not sure yet, at this time, whether we have imposed too much constraint on the *MWRW*-scheduler in order to ensure that current and pending reads are assigned only the K most recent versions. There may be less stringent ways to guarantee this goal and it would certainly improve the concurrency of the K-version *MWRW*-scheduler.

BIBLIOGRAPHY

[BHR80] R. Bayer, H. Heller and A. Reiser, Parallelism and recovery in database systems, *ACM Trans. Database Systems 5*, 2 (June 1980), 139-156.

[BSW79] P. A. Bernstein, D. W. Shipman and S. W. Wong, Formal aspects of serializability in database concurrency control, *IEEE Trans. on Software Eng. SE-5*, 3 (May 1979), 203-216.

[BeG81] P. A. Bernstein and N. Goodman, Concurrency Control in Distributed Database Systems, *ACM Computing Surveys 13*, 2 (June 1981), 185-221.

[BeG83] P. A. Bernstein and N. Goodman, Multiversion Concurrency Control-Theory and Algorithms, *ACM Trans. Database Systems 8*, 4 (Dec. 1983), 465-483.

[BuS83] G. N. Buckley and A. Silberschatz, A complete characterization of a multiversion database model with effective schedulers, Tech. Rept. Tech. Rep.-217, Dept. Computer Science, Univ. of Texas, Austin, 1983.

[BuS85] G. N. Buckley and A. Silberschatz, Beyond Two-Phase Locking, *J. ACM 32*, 2 (April, 1985), 314-326.

[Car83] M. J. Carey, Multiple Versions and the Performance of Optimistic Concurrency Control, Computer Sciences Tech. Rept. No. 517, University of Wisconsin-Madison, October 1983.

[CaM84] M. J. Carey and W. A. Muhanna, The Performance of Multiversion Concurrency Control Algorithms, Computing Sciences Tech Report No. 550, University of Wisconsin-Madison, Aug 1984.

[Car84] M. J. Carey, The performance of concurrency control algorithms for database management systems, Computer Sciences Tech. Rept. No. 530, University of Wisconsin-Madison, 1984.

[CaB80] M. A. Casanova and P. A. Bernstein, General purpose schedulers for database systems, *Acta Informatica 14*, (1980), 195-220.

[Cas81] M. A. Casanova, The concurrency control problem for database systems, in *Lecture Notes in Computer Science 116*, Springer Verlag, Berlin, 1981.

[EGL76] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, The notions of consistency and predicate locks in a database system, *Comm. ACM 19*, 11 (Nov. 1976), 624-633.

[HaP85] T. Hadzilacos and C. H. Papadimitriou, Algorithmic Aspects of Multiversion Concurrency Control, Proc. 4th ACM SIGACT News-SIGOPS SPODS, Oregon, Mar 1985.

[IKM82] T. Ibaraki, T. Kameda and T. Minoura, Serializability made simple, TR82-12, Department of CS, Simon Fraser Univ., Dec. 1982.

[IKM83] T. Ibaraki, T. Kameda and T. Minoura, Disjoint-interval topological sort: a useful concept in serializability theory, Proc. 9th Int. Conf. on VLDB, Florence, Italy, 89-91, Oct/Nov 1983.

[IbK83] T. Ibaraki and T. Kameda, Multi-version vs. Single-version serializability, LCCR TR83-1, Lab for Computer & Communication Research, Simon Fraser University, 1983.

[IKK86] T. Ibaraki, T. Kameda and N. Katoh, Multiversion Cautious Schedulers for Database Concurrency Control, LCCR TR86-2, Lab for Computer & Communication Research, Simon Fraser University, 1986.

[KIK85] N. Katoh, T. Ibaraki and T. Kameda, Cautious transaction schedulers with admission control, *ACM ACM Trans. Database Systems 10*, 2 (June 1985), 205-229.

[KKI86] N. Katoh, T. Kameda and T. Ibaraki, A cautious scheduler for multi-step transactions, *To appear in Algorithmica 1*, , 1986.

[KeS79] Z. Kedem and A. Silberschatz, Controlling Concurrency using Locking Protocols, Proc of the 20th IEEE Symposium on Foundations of Computer Science, IEEE, New York, Oct 1979.

[KuR81] H. T. Kung and J. T. Robinson, On optimistic methods for concurrency control, *ACM Trans. Database Systems 6*, 2 (June 1981), 213-227.

[Pap79] C. H. Papadimitriou, The serializability of concurrent database updates, *J. ACM 26*, 4 (Oct. 1979), 631-653.

[PaK84] C. H. Papadimitriou and P. C. Kanellakis, On concurrency control by multiple versions, *ACM Trans. Database Systems 9*, 1 (March 1984), 89-99.

[Ree78] D. P. Reed, Naming and Synchronization in a Decentralized Computer System, Technical Report MIT/LCS/Tech. Rep.-205, Dept. of Electrical Engineering and Computer Science, M.I.T., Sept. 1978.

[Set81] R. Sethi, A model of concurrent database transactions, Proc. 22nd IEEE Symp. Foundation of Comp. Sci., Oct. 1981.

[SiK80] A. Silberschatz and Z. Kedem, Consistency in hierarchichal database systems, *J. ACM 27*, 1 (Jan 1980), 72-80.

[Sil82] A. Silberschatz, A Multi-version Concurrency Scheme with No Rollbacks, SIGACT News-SIGOPS Symposium on Principles of Distributed Computing, Aug 1982.

[SLR76] R. E. Stearns, P. M. I. Lewis and D. J. Rosenkrantz, Concurrency control for database systems, Proc. 17th IEEE Symp. Foundation of Computer Sci., Houston, Texas, 19-32, Oct. 1976.

[StR81] R. E. Stearns and D. J. Rosenkrantz, Distributed Database Concurrency Controls Using Before-Values, Proc. 1981 ACM-SIGMOD Conference, 1981.

[Vid85] K. Vidyasankar, A Simple Characterization of Database Serializability, Technical Report #8509, Dept. of Computer Science, Memorial University of Newfoundland, May, 1985.

[Yan81] M. Yannakakis, Issues of Correctness in Database Concurrency Control by Locking, Proc. ACM Symposium Theory of Computing, 1981.

[Yan82a] M. Yannakakis, A Theory of Safe Locking Policies in Database Systems, *J. ACM 29*, 3 (Jul 1982), 718-740.

[Yan82b] M. Yannakakis, Freedom from Deadlock of Safe Locking Policies, *SIAM Journal of Computing 11*, 2 (May 1982), 391-408.