Computer Science and Systems Analysis

Computer Science and Systems Analysis

Technical Reports

Miami University *Year* 1994

A Survey of Traditional and Practical Concurrency Control in Relational Database Management Systems

Patricia Geschwent
Miami University, commons-admin@lib.muohio.edu

MIAMI UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & SYSTEMS ANALYSIS

TECHNICAL REPORT: MU-SEAS-CSA-1994-006

A Survey of Traditional and Practical Concurrency Control in Relational Database Management Systems

Patricia K. Geschwent

School of Engineering & Applied Science | Oxford, Ohio 45056 | 513-529-5928

A Survey of Traditional and Practical Concurrency Control in Relational Database Management Systems

by

Patricia K. Geschwent
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #94-006 1994/12

Table of Contents

ABSTRACT

Traditionally, database theory has focused on concepts such as atomicity and serializability, asserting that concurrent transaction management must enable correctness above all else. Textbooks and academic journals detail a vision of unbounded rationality, where reduced throughput because of concurrency protocols is not of tremendous concern.

This thesis seeks to survey the traditional basis for concurrency in relational database management systems and contrast that with actual practice. SQL-92, the current standard for concurrency in relational database management systems has defined isolation, or allowable concurrency levels, and these are examined. Some ways in which DB2, a popular database, interprets these levels and finesses extra concurrency through performance enhancement are detailed.

SQL-92 standardizes de facto relational database management systems features. Given this and a superabundance of articles in professional journals detailing steps for fine-tuning transaction concurrency, the expansion of performance tuning seems bright, even at the expense of serializabilty.

Are the practical changes wrought by non-academic professionals killing traditional database concurrency ideals? Not really. Reasoned changes for performance gains advocate compromise, using complex concurrency controls when necessary for the job at hand and relaxing standards otherwise. The idea of relational database management systems is only twenty years old, and standards are still evolving. Is there still an interplay between tradition and practice? Of course. Current practice uses tradition pragmatically, not idealistically. Academic ideas help drive the systems available for use, and perhaps current practice now will help academic ideas define concurrency control concepts for relational database management systems.

"Many serious endeavors would never get off the ground if pioneers were limited to discussing in public only what they could demonstrate rigorously." [Papert, 5]

1. Introduction

According to Jeffrey D. Ullman, two fundamental qualities set database management systems apart from other types of programming systems: the capacity to manage persistent data and to access efficiently large quantities of data. The ability to quickly access arbitrary chunks of data is central to today's database management systems. Additional capabilities ubiquituous in database management systems include:

(1) Data modeling support, by which users can view data, including dynamic schema changes

(2) Data Definition and Data Manipulation Languages (most probably nonprocedural); automatic query optimization

(3) Transaction Management, or the ability to provide automatic, correct and concurrent access to the database.

(4) Security and integrity features, including authorization capabilities

(5) Resiliency, or lossless recovery capabilities in the case of system failure

Of these, concurrency is important for transaction management, integrity, and resiliency.

Figure 1.1 illustrates transaction management and resiliency in the Database Manager function. Data definition, data manipulation languages and data modeling support are seen in the Database Scheme group. Security and integrity come under the auspices of the Authorization Tables and Concurrent Access Tables.

The importance of concurrent transaction management has increased with the popularity of relational databases and faster, cheaper hardware on which to run relational database management systems. Relational databases are inherently slower than their predecessors, hierarchical and network databases, because of the computational expense of joining tables to answer queries. The development and use of SQL, a high-level, non-procedural query

and command language, has both fostered and enabled greater concurrency in database systems. Although it was estimated in 1992 that fewer than five Fortune 500 companies had more than 10 percent of their total corporate data stored in relational databases [Sayles 1992], relational database management systems (with some object extensions) remain the vision of many companies today. With the exception of object-oriented articles, academic theory and proprietary papers have focused predominantly on relational systems for at least the past fifteen years.

There is substantial divergence, however, between the goals of academic or traditional theory and proprietary research regarding the most important functions of a database management system. The traditional viewpoint regards integrity as the sina qua non of such systems. Article after published article proclaims a new, improved concurrency control mechanism. The database industry, not surprisingly, reflects their customers' concerns by balancing integrity concerns with performance issues. When relational database management systems began selling, traditional locking methods were employed. Performance concerns now mandate a greater pragmatism, with continued questioning of how much serializability is required in a given query or database, and spawning another set of protocols where serializability may not be as important as throughput.

This thesis proposes to survey the traditional, academic view of concurrency in databases and contrast it with a survey of concurrency actually used by relational database management systems, using DB2 as the primary example. After the introduction, a discussion of concepts basic to database concurrency is provided. Next, a brief history of academic papers concerning concurrency, descriptions of the traditional methods of concurrency control, and a short history of SQL and its extensions to the practical world are presented. Then an analysis of how locking policies are actually used in several relational database management systems today is given. Finally, some projections toward

the future are presented in this paper. At this point, a re-discussion of the term "concurrency" seems in order.

Concurrency is a term burdened with context, carrying meaning in every branch of applied computer science. Database concurrency is implemented when multiple transactions or different executions of the same transaction run simultaneously. If multiple transactions modify the same data item, the integrity of the database might be compromised if there is not a control mechanism to order their changes safely (for example, in a bank's accounts database). If, on the other hand, the simultaneous transactions are read-only and do not modify the database, quick, maximum concurrent operation is possible without compromising the integrity of the database (for example, title and author lookup in a library's database). The efficiency of today's relational database management systems is very dependent on the level of concurrency that it permits.

To keep the interaction of multiple transactions from compromising the integrity of a database, we employ concurrency control protocols or schemes which can be implemented either from the database management system or by a combination of the database management system and the operating system, depending on the application.

The serializable execution of several transactions ensures the integrity of today's relational database management systems. Serializability is the main criteria used for defining correct, concurrent programming. In essence, the concept of serializability guarantees that a set of transactions finish executing with the same values that would be achieved by any serial (one transaction running at a time, to completion, before another starts) execution of the same set of transactions.

The choice of what kind of concurrency control to use is an important implementation and performance issue as evidenced in the following quote. "According to a Sentry Market Research study, users find that DB2 performance is 37 percent dependent on system factors and 63 percent dependent on database and application design. . . the most common

cause of response time problems is not resource consumption, but concurrency. Designing systems that share resources and work together smoothly is a critical objective." [Viehman 1994]

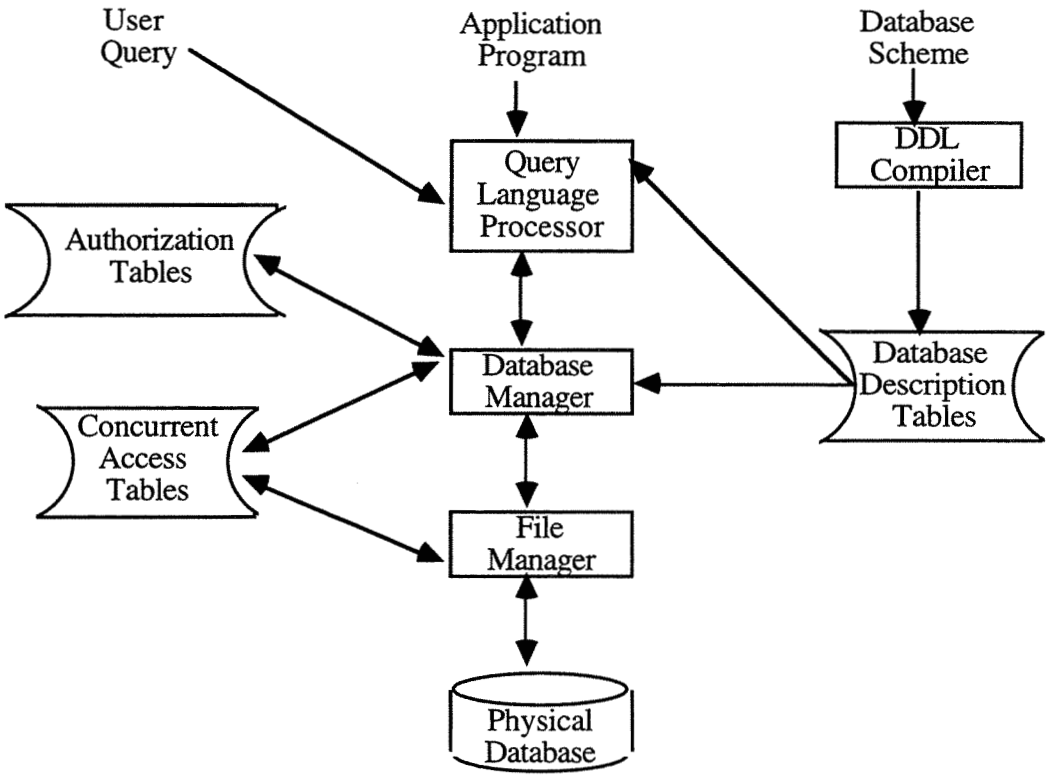


Figure 1.1. Diagram of a database system [Ullman 1988, 17]

2. Concepts and Terminology

There are a number of general concepts one must understand to be able to discuss concurrency in databases. Nine are presented here: transactions, atomicity, serializability (or isolation), locking protocols (general case), granularity of locking, deadlock, livelock, referential integrity constraints and recovery.

2.1 Transactions

A transaction is one execution of a program, where a program can be represented by a single query or through embedded calls in a host language. In Figure 2.3.1, both T1 and T2 are transactions, each comprised of two database operations (READ, WRITE). If more than one execution of the same program is running, each execution is a different transaction. Transactions are comprised of small steps, such as read, write, and arithmetic operations. When concurrency control is enforced, other small steps are needed--steps which set and release locks, mark timestamps, or finish transactions [O'Neil 1994].

2.2 Atomicity

The principle of atomicity emphasizes transaction correctness, so that a transaction, or group of several database operations, is taken together as a single, uninterruptable, all or nothing, package. We assume that the small steps mentioned above are atomic, even though the end of a time slice could occur in the middle of an arithmetic step. The step is atomic because it occurs in a local workspace and nothing affects that space until the transaction performing the step resumes. This operation should be transparent to the user--if the transaction occurs, the user does not know if another transaction was executing at the same time or not.

Why would a transaction not be atomic? In both of the following cases, it is the responsibility of the database management system to ensure correctness.

(1) More than one transaction might occur simultaneously or be interleaved in a time-shared system. If several disk units were reading and writing data to or from the database at the same time, the time slice for one transaction might end during a computation, and part of another transaction might be performed before the first transaction completes.

(2) A transaction might be aborted and not complete at all. For example, it may have requested data for which it lacked access privileges. Parts of the transaction have completed but due to denied access the entire transaction cannot complete.

2.3 Serializability (or Isolation) of Schedules

Serializability, or isolation, is the standard for ensuring atomicity. Serializability "forces transactions to run concurrently in a way that makes it appear that they ran one-at-a-time (serially)" [Ullman 1988]. When transactions run concurrently, their operations are interleaved. Without the property of serializability, the interleaving of operations can result in inconsistent data. Assume two transactions, T1 and T2, shown in Figure 2.3.1. If each accesses a single item A and adds 10 to A's value, we may say they are both executions of the same program P where:

```
P:  READ(A);
    A:=A+10;
    WRITE(A);
```

T1:	T2:	Value of A in database (after operation)	Value of A in T1's workspace	Value of A in T2's workspace
READ(A)		5	5	
	READ(A)	5	5	5
A:=A+10		5	15	5
	A:=A+10	5	15	15
	WRITE(A)	15	15	15
WRITE(A)		15	15	

Figure 2.3.1.

Even though any serial execution of T1 and T2, where T1 executes in its entirety with T2 following, or T2 executes in its entirety before T1 follows, would yield the result 25, in Figure 2.3.1 the lack of concurrency control has allowed T1 to overwrite T2's WRITE(A), resulting in incorrect data. This is an example of the "lost update problem." Other problems which may occur in interleaving of transaction operations are known as the "dirty read," "incorrect summary," and "unrepeatable read."

Dirty reads occur when a transaction updates an item, then the transaction fails. If the item is accessed by another transaction before this update is undone, "dirty" or "temporary" data is read. Incorrect summary describes the situation where one transaction calculates an aggregate summary on some records while an interleaved transaction updates the same records. If the first transaction uses some records that have been updated and others that have not, the calculation will be incorrect. Unrepeatable read refers to different values seen by a transaction which reads an item before and after the item is changed by another transaction [Elmasri and Navathe 1994].

A schedule is a specific order in which the elementary steps of a set of transactions (read, write) are interleaved. The steps of any transaction must appear in the schedule in the same order as the program executed by the transaction. If every step of one transaction occurs before the steps of another transaction, we call the schedule serial. In Figures 2.3.2 and 2.3.3, the possible serial schedules for T1 and T2 are shown. If an interleaved schedule is the equivalent to either serial schedule, i.e., A=15, B=125 and C=-3, or A=15, B=85 and C=1, the interleaved schedule is serializable.

There are numerous ways to define schedule equivalence [Korth and Silberschatz 1986]. Here the two main forms are discussed.

The first is conflict equivalence. In a schedule S, two consecutive operations Ii and Ij are in conflict if they are in different transactions on the same data item and at least one is a write. Given a schedule S, it may be transformed into a schedule S' by a series of swaps of

non-conflicting operations. S is said to be conflict equivalent to S'. A schedule S is conflict serializable if it is conflict equivalent to a serial schedule. Figure 2.3.4 can be easily seen to be conflict equivalent to the serial schedule in Figure 2.3.3 (T2, T1) since all the operations of T2 can be swapped to precede all the operations of T1. Figure 2.3.5 shows a non-serializable schedule for T1 and T2.

T1:	T2:	Database value (after operation) A	B	C	Value in T1's workspace A	B	C	Value in T2's workspace A	B	C
READ(A)		5			5					
A:=A+10		5			15					
WRITE(A)		15			15					
READ(B)			15			15				
B:=B+10			15			25				
WRITE(B)			25			25				
READ(C)				10				10		
C:=C*.2				10			.2			
WRITE(C)				2			.2			
	READ(B)	25							25	
	B:=B*5	25							125	
	WRITE(B)	125							125	
	READ(C			2						.2
	C:=C-5			2						-3
	WRITE(C)			-3						-3

Figure 2.3.2. Serial schedule where T1 executes first, followed by T2.

T1:	T2:	Database value (after operation) A	B	C	Value in T1's workspace A	B	C	Value in T2's workspace A	B	C
	READ(B)		15						15	
	B:=B*		15						75	
	WRITE(B)		75						75	
	READ(C)			10						10
	C:=C-5			10						5
	WRITE(C)			5						5
READ(A)		5			5					
A:=A+10		5			15					
WRITE(A)		15			15					
READ(B)			75			75				
B:=B+10			75			85				
WRITE(B)			85			85				
READ(C)				5			5			
C:=C*.2				5			1			
WRITE(C)				1			1			

Figure 2.3.3. Serial schedule where T2 executes first, followed by T1.

T1:	T2:	Database value (after operation) A	B	C	Value in T1's workspace A	B	C	Value in T2's workspace A	B	C
READ(A)		5			5					
	READ(B)		15						15	
A:=A+10		5			15					
	B:=B*5		15						75	
WRITE(A)		15			15					
	WRITE(B)		75						75	
READ(B)			75			75				
	READ(C)			10						10
B:=B+10			75			85				
	C:=C-5			10						5
WRITE(B)			85			85				
	WRITE(C)			5						5
READ(C)				5			5			
C:=C*.20				5			1			
WRITE(C)				1			1			

Figure 2.3.4. A conflict serializable schedule for T1 and T2.

T1:	T2:	Database value (after operation) A	B	C	Value in T1's workspace A	B	C	Value in T2's workspace A	B	C
READ(A)		5			5					
	READ(B)		15						15	
A:=A+10		5			15					
	B:=B*5		15						75	
WRITE(A)		15			15					
READ(B)			15			15				
	WRITE(B)		75						75	
	READ(C)			10						10
B:=B+10			15			25				
	C:=C-5			10						5
WRITE(B)			25			25				
	WRITE(C)			5						5
READ(C)				5			5			
C:=C*.20				5			1			
WRITE(C)				1			1			

Figure 2.3.5. A non-serializable schedule for T1 and T2.

The second form of equivalence is that of view equivalence. Two schedules S and S' are view equivalent if all the following requirements are met:

(1) For each data item I, if transaction Ti reads an initial value of I in schedule S1, then transaction Ti must also read the initial value of I in schedule S2.

(2) For each data item I, if transaction Ti produces a value I, and Tj then executes READ(I) in schedule S1, the same order must be followed in schedule S2.

(3) For each data item I, if there is a transaction performing a final WRITE(I) in schedule S1, the same transaction must perform the final WRITE(I) in schedule S2 [Korth and Silberschatz 1986].

It is called view equivalence because rules 1 and 2 guarantee that transactions see the same "view" in both schedules and rule 3 guarantees the final view of the database is the same. A schedule S is view serializable if it is view equivalent to a serial schedule.

While both conflict and view serializability are based on the read and write operations of transactions, view serializabilty is less stringent than conflict equivalence, i.e., conflict serializable schedules imply view serializability, but not vice versa. Any view serializable schedule that is not conflict serializable contains blind or unconstrained writes. A blind write occurs when WRITE(I) is performed without a preceding READ(I).

There are some general observations still to be made regarding serializability. If random operations on items are allowed, with scores of initial values, it is not possible in real time to test whether multiple schedules have the same effect for all initial item values. Thus, in practice, a commonly-agreed upon assumption regarding serializability is that ". . .values cannot be the same unless they are produced by exactly the same sequence of operations. Thus, we do not regard (A+10)-20 and (A+20)-30 as producing the same values" [Ullman 1988]. Ignoring algebraic properties of transactions such as commutativity of certain operations may cause "nonfatal" errors, i.e., force us to call a serializable schedule nonserializable (slowing concurrency), but will not cause a "fatal" error, i.e., calling a nonserializable schedule serializable. The traditional protocols detailed in this thesis will allow serializability and concurrency, while guaranteeing correctness, but we will probably never reach a state where every schedule of a collection of transactions is allowed if it is serializable.

In practice, database management systems do not test for serializability, instead relying on protocols to ensure serializability. For a set of N transactions, the number of serializable schedules is N (N-1) (N-2)...1 = N!. In theory, algorithms to test for conflict serializability use directed acyclic precedence graphs, where the set of vertices consists of all transactions in a schedule, and the set of edges are built from the following conditions:

(1) Transaction T1 executes WRITE(I) before T2 executes READ(I)

(2) Transaction T1 executes READ(I) before T2 executes WRITE(I)

(3) Transaction T1 executes WRITE(I) before T2 executes WRITE(I)

To complete the testing process, one must construct the precedence graph and use a cycle detection algorithm [Ullman 1988]. If a cycle is detected, then the schedule is not conflict serializable. If no cycle is detected, then the equivalent serial order can be achieved by topological sorting.

2.4 Locking

Locking is the principal technique used to control concurrency. A lock is an access privilege, which can be granted or denied by a lock manager. A lock manager is that part of a database management system which records for a single item, I (row, table, page, or tablespace), whether any transactions are executing on any part of I. Then, given the particular locking protocol, the lock manager may block access to another transaction, given the possibility of conflict. The lock manager stores current locks in a lock table, under the data points (<item>, <lock type>, <transaction>) [Ullman 1988, 470], signifying that transaction T holds a certain kind of lock L on item I. Several transactions can hold certain locks on the same item at the same time. Efficient management of locks may be achieved by, for example, using a hash table or comparable data structure with the item field as a key.

Locks control concurrency by acting as synchronization primitives, i.e., access to data items are done in a mutually exclusive manner. Locking of data items by itself does not guarantee serializability. Locking protocols establish a set of rules that must be followed by a transaction. These rules tell a transaction when it may lock and unlock each of the data items. By following a locking protocol, a serializable schedule is ensured. For example, a simple locking protocol would prevent the result of the Figure 2.3.1 by placing a lock on A for T1, preventing another transaction from accessing A until T1 has finished. In a locking protocol, we assume every transaction eventually unlocks any items it locks.

As an example of a very general type of locking, two possible scenarios follow. In Figure 2.4.1, if T1 sets a lock, T2 is unable to access the data item (A) until the lock is released. In Figure 2.4.2, when T2 sets a lock, T1 is unable to access the data item (A) until the lock is released. In both cases serializability in ensured, but concurrency is inhibited. More sophisticated locking protocols are discussed in Section 3.2.1.

T1:	T2:	Value of A in database (after operation)	Value of A in T1's workspace	Value of A in T2's workspace
LOCK(A)				
READ(A)		5	5	
A:=A+10		5	15	
WRITE(A)		15	15	
UNLOCK(A)			15	
	LOCK(A)			
	READ(A)	15		15
	A:=A+10	15		25
	WRITE(A)	25		25
	UNLOCK(A)	25		

Fig. 2.4.1. General locking protocol with T1 executing as the first transaction.

T1:	T2:	Value of A in database (after operation)	Value of A in T1's workspace	Value of A in T2's workspace
	LOCK(A)			
	READ(A)	5		5
	A:=A+10	5		15
	WRITE(A)	15		15
	UNLOCK(A)	15		
LOCK(A)				
READ(A)		15	15	
A:=A+10		15	25	
WRITE(A)		25	25	
UNLOCK(A)		25		

Figure 2.4.2. General locking protocol with T2 executing as the first transaction.

2.5 Granularity

Granularity is a term that describes the nature and size of units of data to which access is controlled, typically with the use of locks. For example, in a relational database management system, one may choose tables (coarse-grained granularity) or rows, sets of attributes in a row, or some combination of rows for fine-grained granularity. Using coarse-grained granularity may cut down on system overhead because less space is needed to store locks and fewer actions with locks need to be performed. Fine granularity may increase response time and throughput--with fewer locks on smaller pieces of data, more transactions can execute simultaneously. One heuristic used in today's database management systems is to set the granularity according to the average access of common transactions in the application. If a typical transaction in a relational database management system accesses one row, then rows become the item size. Likewise, if a typical transaction joins tables, whole tables may become the item size.

2.6 Deadlock

A potential problem in using a locking protocol for concurrency control is deadlock. Deadlock occurs in a set S where two or more transactions are waiting to lock an item already locked by another transaction in the set, as illustrated in Figure 2.6.1. Imagine two transactions which begin operating at roughly the same time. Their significant points are:

T1:	T2:	: Fig. 2.6.1
LOCK(A)	LOCK(B)	: Operation begins, a lock on an item is
.	.	: granted to each transaction
.	.	
.	.	
LOCK(B)	LOCK(A)	: Now each operation requests a lock on the item
		: locked by the other transaction. Since neither
		: transaction may proceed without the other locked
		: item, deadlock ensues--they both wait forever.
UNLOCK(A)	UNLOCK(B)	
UNLOCK(B)	UNLOCK(A)	

Several common solutions to prevent deadlock are:

(1) A protocol which requires each transaction to request all its locks at once. The lock manager grants all or none. The process waits if specific locks are already enforced. This protocol would have prevented the deadlock in the previous example, but would severely hamper concurrency.

(2) A protocol which assigns an arbitrary linear ordering to items, and forces transactions to request locks in this order. For example, this would work in Figure 2.6.1--assume A precedes B in the locking order. T2 would then be required to request a lock on A before B, and find A already locked. T2 would not get to lock B, so that it would be available for T1. Again, possible concurrent execution of transactions is greatly diminished.

(3) A transaction timestamping (TS) protocol assigns a unique identifier to each transaction, based on the order in which the transactions begin. The older the transaction, the smaller the timestamp. Two schemes utilizing transaction timestamping are wait-die and wound-wait. Assume Tj is holding a lock desired by Ti. In wait-die, if TS(Ti) < TS(Tj), then Ti is allowed to wait, else Ti dies and is restarted later with the same timestamp. Hence, the older transaction waits on a younger transaction while a younger transaction requesting an older transaction's item is aborted and restarted. In wound-wait, if TS(Ti) <

TS(Tj), Ti wounds Tj (i.e., aborts it). Tj restarts later with the same timestamp. If TS(Ti) > TS(Tj), then Ti is allowed to wait. In this scheme a younger transaction waits for an older one, while an older transaction can preempt a younger transaction.

(4) No waiting and cautious waiting schemes do not require timestamps. A no waiting algorithm will abort and restart (after a time delay) any transaction unable to obtain a lock, regardless of whether deadlock actually occurred or not. Cautions waiting only allows transactions to wait on transactions that are not blocked [Elmasri and Navathe 1994, Korth and Silberschatz 1986, O'Neil 1994].

Another approach to deadlock handling is detection. This general protocol does nothing to prevent deadlock, but periodically examines lock requests and checks for deadlock. Imagine a waits-for graph, with nodes representing transactions and arcs signifying a transaction T1 is waiting to lock item I, on which T2 holds the lock. Every cycle indicates a deadlock; if there are no cycles, the graph is deadlock-free. If a deadlock is discovered, at least one of the deadlocked transactions will be aborted and restarted, its effects on the database cancelled.

2.7 Livelock

Livelock is another problem that can arise when a locking protocol is used. Livelock is the database equivalent of starvation in operating systems. It occurs when a set of transactions are waiting for access to an item, and as described previously in deadlock or timestamp resolution, one transaction keeps getting bumped to the back of the line.

2.8 Referential Integrity Constraints

A key advantage of relational database management systems is their support of integrity constraints. These constraints may be as simple as enforcing data types for data items, or more to the point for this survey, involve the relationship between tuples in one relation and

tuples in another relation. Of course, tuples in one relation may have many relationships with multiple other relations. Concurrency implications become considerable since a lock on a tuple in one relation expands to cover the related tuples.

Structured Query Language (SQL), the standard non-procedural query language used in today's relational databases, supports explicit definition of such relationships by assertion of foreign keys, or links to the unique primary keys of other relations. Assertions are made at the creation of a table and can be updated or changed by the owner or DBA at will.

2.9 Recovery

A database management system is responsible for recovering from failure, whether hardware- (e.g., system crashes) or software-based (e.g., transaction errors). Typically, a log buffer is shared by all transactions. In concurrent systems, if a transaction is aborted or rolled back, it is necessary to compel any transaction which depends upon that transaction to perform the same action. Consider a schedule which contains transactions T1 and T2. T1 writes a value for item A, which T2 reads. If T1 fails at this point, it will be rolled back to a point before WRITE(A). T2 must also be rolled back to the point before it executed READ(A). The circumstance in which one transaction failure leads to a series of transaction rollbacks is known as cascading rollback, and may occur under two-phase locking and timestamp-based protocols, as described later in this survey. Cascading rollback is inexpedient because it may negate a noteworthy amount of work.

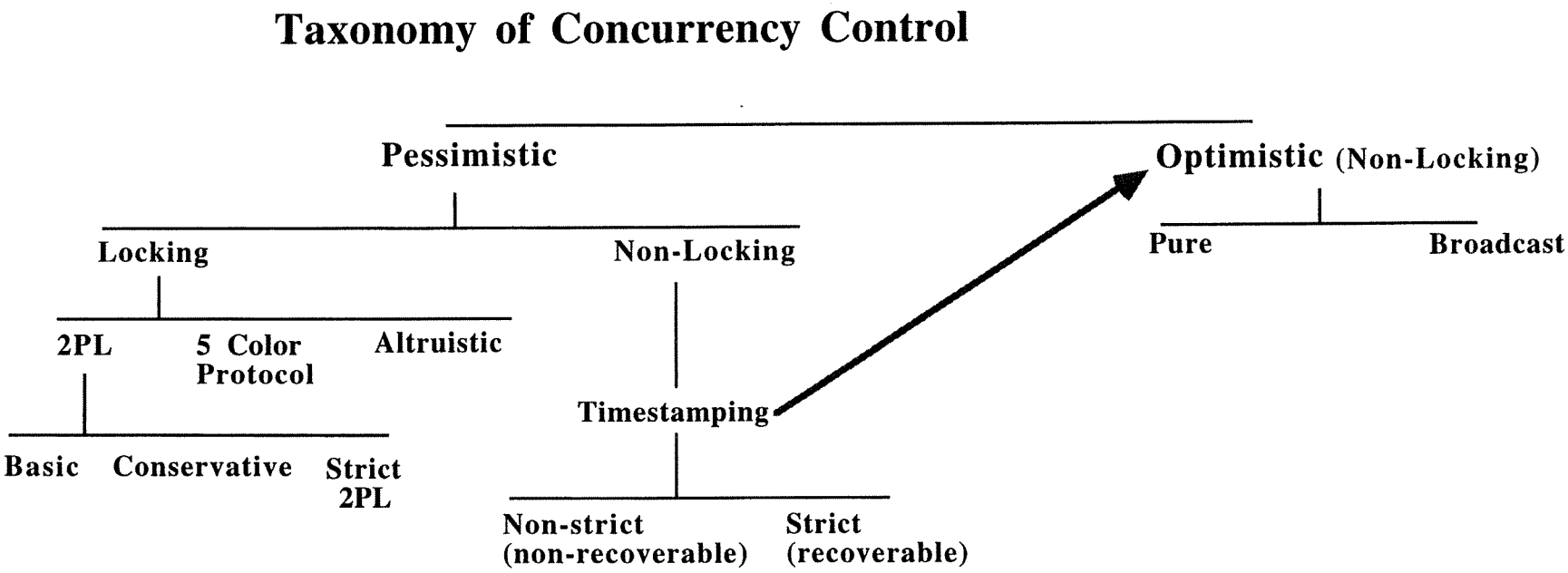


Figure 3.1 Taxonomy of Concurrency Control

PESSIMISTIC concurrency controls assume there will be contention for data, and prevents conflict by locking items in use or by executing transaction operations in the order of the transactions' timestamps.

OPTIMISTIC concurrency controls assume data contention will be small, and arbitrates possible conflict by checking a series of rules before the transaction is allowed to complete.

3. Taxonomy of Traditional Concurrency Control Protocols

Figure 3.1 illustrates the taxonomy of traditional concurrency control protocols. The difference between the major categories, pessimistic and optimistic, is based on a differing philosophy as their names imply. Pessimistic techniques assume a worst case scenario: every piece of data examined by a transaction might be required by some other concurrent transaction, therefore it needs to be locked or checked before committing. Optimistic techniques assume a best case scenario: no piece of data examined by a transaction is required by another concurrent transaction; therefore the transactions can run to completion, and is then checked to see whether conflict occurred. If so, the transaction simply aborts and starts again. More details concerning each specific protocol are contained in Sections 3.2 through 3.2.3, after a short chronology of concurrency protocols.

3.1 Overview and Brief History

Pessimistic controls automatically assume there will exist contention for data and *prevent* conflict by locking items or executing transaction operations in the order of the transactions' timestamps. Optimistic controls assume data contention will be small, and *arbitrate* possible conflict by checking a series of rules [O'Neil 1994].

Both types of concurrency control ensure serializability, or that access to data items is carried out in a mutually exclusive manner. Pessimistic concurrency controls include locking and timestamping. Locking pessimistic concurrency control requires each transaction to secure access rights to data before beginning an operation. Shared (read) and exclusive (write) locks follow a locking protocol, which places restrictions on the number of possible serializable schedules, as well as requiring a directed acyclic graph algorithm for deadlock detection. Concurrency is reduced; deadlock is possible and handled by prevention or detection algorithms.

In a standard two-phase locking protocol (2PL), lock and unlock requests are issued in a growing phase and a shrinking phase, that is, all locks are acquired before any are released. In 2PL, two shared (read) locks can be held at the same time, otherwise the first transaction holds an exclusive-mode lock so that it can both read and write a data item. Subsequent exclusive and shared lock requests must wait until the currently-held lock releases, i.e., when the data item is no longer accessed; this is different from releasing locks when an item is last accessed, which may not ensure serializability. Although lock conversions, or upgrading from a READ LOCK to a WRITE LOCK status are allowed, the degree of concurrency provided is low. The protocol assumes problems will occur and seeks to prevent them. With locking, the number of possible schedules is reduced. Processing overheads are high, because keeping track of locks and the queue waiting for data access is difficult. Storage overhead is increased because of deadlock. If deadlock occurs, the system must roll back one of the two transactions. In order to perform a rollback it must have stored information to enable rollback. Variations of 2-phase locking (2PL) such as strict, altruistic and five-phase locking have been proposed, but because their key tenet remains serializability, concurrency is still lower than it might be [O'Neil 1994].

Timestamping protocols are also pessimistic. A unique timestamp is given to each transaction before it starts execution by the system clock or a logical counter. The timestamps of the transactions determine the serializability order. If transaction A starts before transaction B, the serial order is transaction A followed by transaction B. This serial order implies that transaction A should never see transaction B updates and A should never be permitted to update anything already seen by B. As will be seen in optimistic concurrency control, rules specify the order of transactions. There are serial schedules possible under this protocol that are not possible under locking, but the converse is also true. In timestamping protocols, processing overheads are high, and can be difficult to sustain, based on the clock use and possible synchronization problems. There does not

appear to be a storage overhead in one timestamp protocol that is better than the others [Bernstein and Goodman 1981, Korth and Silberschatz 1986].

Validation or certifier techniques use optimistic concurrency theories. Optimistic concurrency control is characterized by the assumption that most transactions will be conflict free; since there will not be much data contention, holding locks wastes time. Optimistic protocols were chiefly used for query intensive systems, but are now being used in distributed databases, with a mixture of locking/timestamping protocols [Boksenbaum et al 1987, Yu and Dias 1993].

Under optimistic concurrency control, each transaction performs updates on a local copy of data; the transaction is validated by ensuring that the original data is not being accessed simultaneously by a concurrently executing transaction. If validation is successful, the data is written from local to permanent storage. Optimistic concurrency controls monitor a system instead of defining it as pessimistic protocols do; concurrency is maximized and response time is augmented. While cascading rollbacks are impossible since actual writes to the database occur after the transaction commits, aborted transactions and transaction starvation are real possibilities. Like pessimistic concurrency control's deadlock algorithms, optimistic concurrency control requires starvation-prevention algorithms. It is superior for query intensive databases, although pure optimistic concurrency control is not good in high contention, user-updated frequently systems [Bernstein and Goodman 1981, Small et al. 1992, Korth and Silberschatz 1986]. Pure optimistic concurrency control only aborts transactions at commit time, while broadcast optimistic concurrency control aborts transactions as soon as conflict is detected. [Date with Darwen 1992, Yu and Dias 1992].

The above description of concurrency control techniques illustrate the variety of methods that have been proposed. A brief history of this development is presented next.

Until 1981, locking was the unquestioned concurrency control protocol in databases. A proposal was made then for a single-site family of non-locking concurrency controls,

which were labelled optimistic. Incremental transaction numbers validated claims to ensure serializability [Kung and Robinson 1981]. In the same year, Bernstein and Goodman [Bernstein and Goodman 1981] concluded that this new approach would not work for distributed systems because it would not handle the loads.

By 1987, timestamps were gaining acceptance as a value-added benefit to optimistic concurrency control, allowing it to function in a distributed database system [Boksenbaum et al. 1987, Bassiouni 1988].

Additional proposals have posited a large buffer, so data from aborted transactions could be accessed during rerun, thus controlling resource contention due to aborted transactions [Yu and Dias 1992].

Since late 1992, the academic trend appears to be towards hybrid concurrency control, or a mixture of optimistic and pessimistic protocols. Researchers have simulated performance of different concurrency control models and concluded these new techniques offer benefits for high levels of data contention. Methods such as allowing aborts only during the early stages of a transaction's execution, and then waiting for locks to be released were proposed [Franaszek et al. 1992, Yu and Dias 1993].

Present day database management systems may allow no choice between the protocols discussed in the following sections. Some systems, such as GemStone, advertise applications that allow the DBA to specify optimistic or pessimistic concurrency controls. The choice should end up being a function of the types of queries commonly used, because no clear advantage has been shown for any of the following protocols to be discussed. Each allows different types of schedules. Each allows some schedules that the other does not, so the ultimate choice, when there is one, should depend on the necessary balance of concurrency and throughput [O'Neil 1994, Elmasri and Navathe 1994].

3.2 Concurrency Control Algorithms

The following sections describe in more detail three well-known concurrency control techniques in current database management systems. These protocols often form the default concurrency controls in practical relational database management systems. Each of the following sections describes a concurrency algorithm. Following these sections is a summary table for each protocol which also lists their advantages and disadvantages. The last section describes how each concurrency control protocol fits within a general logical transaction system model. The information in the remainder of this section is adapted from Elmasri and Navathe 1994, Korth and Silberschatz 1986, Kung and Robinson 1981, O'Neil 1994 and Ullman 1988. Additional sources are cited as necessary.

3.2.1 Two-phase locking

Two-phase locking, one of the most prevalent concurrency control protocols, ensures transaction isolation by locking data items. The two-phase locking protocol is both simple and popular. Two requirements enable this protocol to guarantee serializability: (1) all locks precede all unlocks; and (2) the scheduler checks only for legality--that is, that two transactions cannot hold locks on the same item simultaneously. If these requirements are met, the transactions are permitted to proceed. The phases refer to the locking and unlocking phases. The lock point is the moment a transaction obtains the last of its locks. If we order transactions in the order they reach lock point, the order must be a serial schedule with the arcs in the precedence graph conforming to this transaction order.

In many transactions, while the value of an item is required, the changing of that item by the transaction is not required, i.e., it can just be read. This necessitates two types of locks: Read (or shared) and Write (or exclusive) locks.

Read or shared locks: Any transaction needing to read an item I executes RLOCK(I), which prevents *other* transactions from writing to I (WLOCK(I)) until I is unlocked. One

or many other transactions can read I while the RLOCK(I) is in effect, and RLOCK(I) can be permitted as long as there is not a WLOCK(I) being held. If only one transaction holds RLOCK(I) and wishes to upgrade to WLOCK(I), it may. UNLOCK(I) removes RLOCK(I).

Write or exclusive locks: Any transaction needing to change the value of an item I must obtain a write-lock, or WLOCK(I). Every other transaction is barred from obtaining a RLOCK(I) or WLOCK(I) while this WLOCK(I) is in effect. UNLOCK(I) removes WLOCK(I).

The lock compatibility matrices seen in Figure 3.2.1.1 illustrate these rules.

Lock Held By By Another Transaction

Lock Requested	Read	Write
Read	Yes	No
Write	No	No

Lock Held By By Same Transaction

Lock Requested	Read	Write
Read	N/A*	No
Write	Maybe**	N/A*

* N/A because of Assumption 2
** If no other transaction held a READ lock, the answer would be YES. If another transaction held a READ lock, the answer would be NO until that transaction yielded its READ lock.

Figure 3.2.1.1. Two phase lock compatibility matrices.

Two phase locking assumes that each locked item will eventually be unlocked, a transaction does not try to lock an item it already holds the same lock on, and that a transaction does not try to unlock an item it does not hold a lock on. Every lock does not need to be an exclusive lock because some transactions will not change the value of an item, so read locks are available. A transaction may upgrade the type of lock it holds on an item, e.g., from a shared or read-lock to an exclusive or write-lock, as soon as no other transaction holds a read-lock on that item. Strict two-phase locking demands that a

transaction will not be written into the database until it has reached its commit point and will not release any locks until after the commit point.

3.2.2 Timestamping

In timestamping, each transaction is assigned a unique timestamp which is used by the protocol to guarantee that conflicting operations are executed in the order of the transaction's timestamp. There are two general ways that timestamps can be issued to transactions: either they are required to pass through the scheduler, which keeps a count of the number of transactions it has scheduled, and assigns the next number to each new transaction, or the database management system may use the value of the machine's internal clock at the time a transaction initiates. If the database is distributed over numerous machines or is running on a machine which contains more than one processor (several versions of the scheduler are possible), a unique suffix of a fixed length is chosen for each processor, and appended to the timestamp issued by the processor, in order to identify the processor. Also, care must be taken that counts or clocks of each processor remain synchronized.

It is possible to describe how timestamps force transactions that are not aborted for failure of protocols to run as if they were serial in the same ways one can describe other protocols--from most restrictive to least--one could have a timestamp-based protocol which distinguishes many kinds of access, such as incrementation (least restrictive), one could have one which does not distinguish between reading and writing, but just locks the database transaction by transaction (most restrictive). Here we discuss a timestamp-based protocol based on read/write locking.

In this scenario, each item in the database is given two timestamps, the read-time, or highest timestamp held by any transaction which has read the item, and the write-time, or

the highest timestamp held by any transaction which has written the item. Using these timestamps, the following are checked.

if X = READ and T >=tw, set read time to t if t> tr OR
if X = WRITE, t >=tr, and t>=Tw, set write time to t if t>tw.

if X = WRITE and tr <=t < tw, do nothing (execute).

if X = READ and T < tw, abort the transaction OR
X = WRITE and t< tr, abort the transaction

Some phases are illustrated in the following figure:

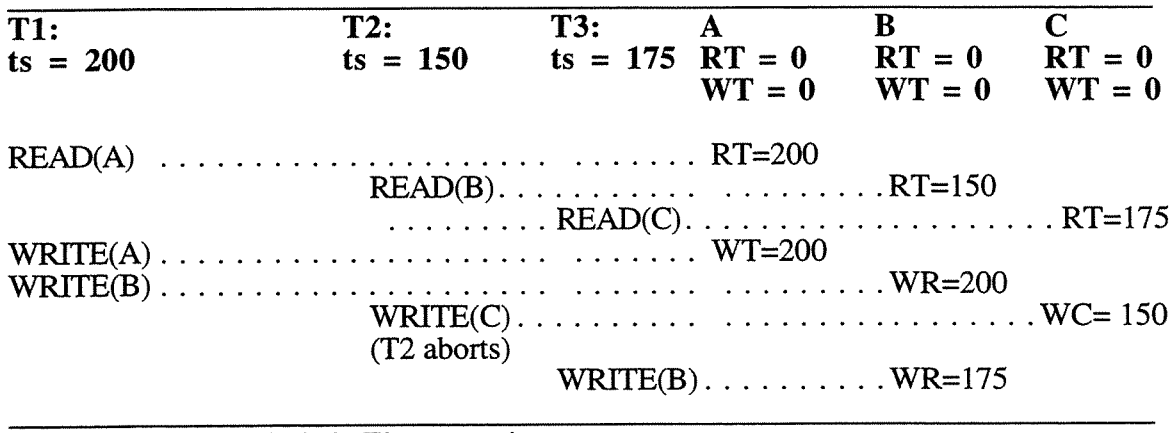


Figure 3.3.2.1. Pessimistic Timestamping.

In Figure 3.3.2.1 the transaction time of T1 is 200, T2 is 150, and T3 is 175. Initially, A, B, and C are assumed to have initial READ- and WRITE-TIMES of 0. In this example, T2 is forced to abort rather than perform a WRITE-TIME with timestamp 150, and READ-TIME of 175. Conversely, the value of WRITE(B) T3 is discarded because although the READ-TIME(B) is 150, less than ts = 175, T3's WR=175 is less than T1's WR=200.

As with locking protocols, there are additional rules for a strict timebased protocol. All updates can be performed only in the workspace, and written into the database after the transaction commits. Writes are written into the log, which is copied to stable storage, thereupon written into the database. As in locking, a commit record is written on the log between the stages. Transaction T must be given a lock on A which will hold between the

time the WRITE-TIME is changed for I, and the time it writes the value, and this lock may be held for a long time. An additional consideration with this protocol is that checking of timestamps must be accomplished prior to the commit point, because abort cannot occur after commit. So, when a transaction T reads or writes an item I, it must check the write-time of I when T reads I, and also check the read-time of I when T writes I in its workspace. If the protocol is strict, cascading rollback is avoided, and furthermore, the schedule is said to be recoverable since no transaction T1 in S commits until all transactions that have written an item that was read by T1 have committed. This is in contrast to an optimistic strategy which checks the READ-TIME or WRITE-TIME of I (if T wrote I) at the time T commits.

Assumptions made for timestamp use include that no two transactions receive the same timestamp, equivalent serial order is the order of the transactions' timestamps, and a transaction cannot read the value of an item that was not written until after the transaction executed, i.e., transaction T1 with timestamp t1 cannot read an item with write time of t2 if t2 > t1. A transaction cannot write an item if the item has its previous value read or written at a later time, i.e., transaction T1 with timestamp t1 cannot write an item with read-time t2 if t2 > t1 or write-time t2 > t1. t1 must abort, and restart with a new timestamp.

3.2.3 Certifier methods

Certifier methods are also known as optimistic or validation methods. This protocol assumes that no conflict is going to occur and proceeds with the transaction using a local copy. Then the protocol validates that the assumption was correct. If not, the transaction is aborted and restarted.

In the certifier protocol, each transaction, Ti, executes in three different phases: Read, Validation and Write. These phases are described below.

Read phase. During the read phase, Ti executes. Values of data items are read and stored in local variables. Write operations are done on these local variables, the "real" database is not updated yet, because no conflict is anticipated.

Validation phase. Ti performs a validation test to check if it can copy its local variables to the database without violating serializability. A transaction requires three timestamps, one for each of the above phases.

Three timestamps are necessary for Ti.

A. Start(Ti) denotes the time when Ti starts executing;

B. Validation(Ti) denotes the time when Ti finishes the read phase, and begins the validation phase. This timestamp determines the serializability order. There are three rules for validation:

   1. Finish(Ti) < Start(Tj). If Ti completes executing before Tj starts its read phase, then serializability is assured;

   2. Ti $\cap$ Tj = $\varnothing$. If Ti and Tj share no elements, then serializability is assured;

   3. Start(Tj) < Finish(Ti) < Validation(Tj). If Ti completes its write phase before Tj starts validation, the writes of Ti and Tj are not overlapping. Because Ti's writes do not affect Tj's read, and because Tj can not affect Ti's read, serializability is maintained;

   4. Write(Ti) $\cap$ Read(Tj) = $\varnothing$. Given different operations, if Ti and Tj share no elements, then serializability is assured.

C. Finish(Ti) denotes the time when Ti has finished its write phase and releases its locks. In contrast to strict timestamping protocols, these locks are held a short time, only long enough for other items written by I to have checks made on their read times.

Write phase. If validation is successful, then the values in the local variables are written to the database. Else, Ti is aborted.

Optimistic concurrency controls use timestamping to determine validation order and believe it is inefficient to prevent conflict that may not occur.

3.3 Summary Table of Concurrency Control Algorithms

Table 3.3.1 restates some details of the concurrency control algorithms already discussed, as well as listing their advantages and disadvantages. As before, this information is extrapolated from Elmasri and Navathe 1994, Korth and Silberschatz 1986, Kung and Robinson 1981, O'Neil 1994 and Ullman 1988.

3.3 Summary Table of Concurrency Control Protocols

2 PHASE LOCKING PROTOCOL

One of the most common concurrency control protocols ensures transaction isolation by locking data items.

Methods:

Guarantees serializability by:

1. requiring that all locks precede all unlocks
2. The scheduler checks that two transactions do not hold locks on the same item simultaneously

Types of locks:

READ locks (RLOCK)

1. prevent other transactions from writing to the held item until RLOCK is lifted
2. >1 transaction can hold RLOCKS at one time
3. cannot obtain a RLOCK if a WRITE lock is in effect

WRITE locks (WLOCK)

1. any transaction wishing to change a value must obtain a WLOCK
2. only 1 transaction may hold a WLOCK on a specific item at any time
3. if only one transaction holds a RLOCK on an item, and wishes to change the item, the lock can be escalated to a WLOCK

TIMESTAMPING PROTOCOL

Each transaction is assigned a unique timestamp to guarantee that conflicting operations are executed in the order of the transaction's timestamp.

Timestamps are issued by the scheduler assigning the next number to each new transaction, or the database management system may use the value of the machine's internal clock when a transaction initiates.

Each item in the database is given two timestamps

READ TIMESTAMP (tr)

1. highest timestamp held by any transaction which has read the item

WRITE TIMESTAMP (tw)

1. highest timestamp held by any transaction which has written the item

Rules to be checked:

if X=READ and

1. T >= tw, set read time to t if t > tr
2. T < tw, abort the transaction

if X=WRITE and

1. t >= tr and t >= Tw, set write time to t if t > tw
2. tr <= t < tw, execute
3. t < tr, abort the transaction

OPTIMISTIC PROTOCOL

Assumes no conflict will occur and proceeds with the transaction using a local copy. A validation phase checks to see if the assumption is correct. If not, the transaction is aborted and restarted.

Divides every transaction execution into three different phases:

READ PHASE:

1. Ti executes. Values of data items are read and stored in local variables. Write operations are performed here.

VALIDATION PHASE:

1. Using three timestamps, Ti performs a validation test to check if it can copy its local variables to the database without violating serializability. Start(Ti), when Ti starts executing Validation(Ti) when Ti finishes the read phase and begins validation phase Finish(Ti), when Ti finishes its write phase.

Rules for serializability:

1. Finish(Ti) < Start(Tj).
2. Ti intersection Tj = 0.
3. Start(Tj) < Finish(Ti) < Validation(Tj).
4. Write(Ti) intersection Read(Tj) = 0.

WRITE PHASE

1. If validation is successful, the values in local variables are written to the database

STRICT 2-PHASE LOCKING
requires:

1. A transaction will not be written into the database until it has reached its commit point
2. A transaction will not release any locks until after the commit point (this avoids cascading rollbacks)

Advantages:

1. Distinguishes between reads and writes and their effect on the database
2. Guarantees serializability regardless of the types of transactions which could operate concurrently with a given transaction
3. Good for update-intensive applications because it is safe

Disadvantages:

1. Inhibits concurrent execution because of locking overhead
2. May lock an item no other transaction needs
3. Inefficient in query-intensive applications because of locking overhead, possibility of deadlock and waits for locked data.

STRICT TIMEBASED PROTOCOL
requires:

1. All updates are performed only in the workspace
2. All updates are written into the database after the transaction commits. Cascading rollback is avoided

1. Different from locking, because the blocked transaction aborts rather than waits for access
2. >1 transaction can read the same item at different times, conflict-free
3. Enhanced concurrency over phased locking because transactions do not block each other needlessly

1. Inefficient where aggressive locking makes sense (where >1 transaction executing simultaneously require the same item). A large amount of rollbacks will occur
2. Timestamp checking done prior to a commit point, because you cannot abort after a commit

Updates are already placed in the database only after the transaction commits, so a strict case is not necessary.

1. Does not inhibit access prior to validation phase because emphasizes arbitration of conflict between transactions, not prevention
2. High concurrent access possible
3. Superior for query intensive databases or other systems with a low conflict rate

1. Not efficient in high contention, frequent-update systems.
2. May abort more transactions than either previous method because checks timestamps later.
3. Not as intuitively understandable as the others.

3.4 Logical Transaction System Model

This section describes how each concurrency control protocol fits within a general logical transaction system model, illustrated in Figure 3.4.1.

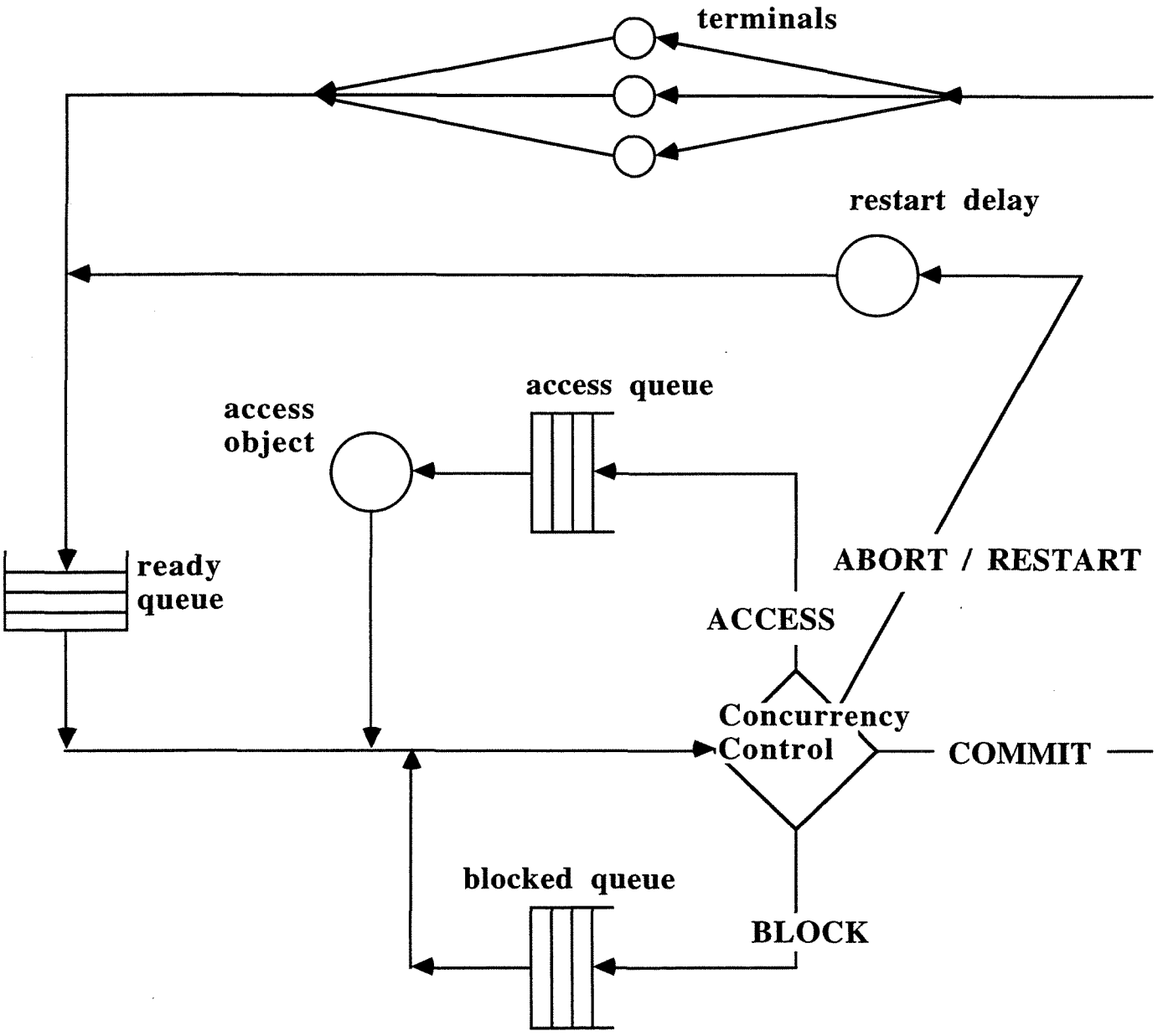


Figure 3.4.1. General Logical Transaction System Model. Adapted from Salem et al, 1994, 140.

Figure 3.4.1 represents a general logical transaction system model. In the case of locking protocols, the actual **Concurrency Control** diamond should move between the **access queue** and **access object**. If access is denied, a diagonal line could be drawn from **access object** to **BLOCK**. The transaction would move from **blocked queue** to **ACCESS**, and finally to **COMMIT**.

In a timestamping protocol, the actual **Concurrency Control** diamond stays where it is. A transaction would go through the immediate **access queue**, **access object**, then move through **Concurrency Control**. If the timestamp rules were not broken, the transaction would commit. Otherwise, the transaction would move through the **ABORT / RESTART** path.

In optimistic concurrency protocols, the actual **Concurrency Control** diamond moves to the left, immediately preceding the **COMMIT**. A transaction would go through the immediate **access queue**, **access object**, then move through **Concurrency Control**. If validation is not successful, the transaction would move through the **ABORT / RESTART** path.

3.5 Summary

The previous sections have discussed three of the most commonly used concurrency control protocols in relational database management systems today. Both pessimistic and optimistic protocols concentrate on maintaining serializability, so that the end product of concurrent transactions is correct.

In the next section, Concurrency in Practice, discussion centers on SQL-92 and DB2, and their focus on performance as well as correctness.

4. Concurrency in Practice

4.1 Structured Query Language

SQL, dubbed "Intergalactic Dataspeak" in 1990 by database guru Michael Stonebraker, is the cement in the foundation of modern Relational Database Mangement Systems. SQL is the standard language for all types of databases [Korth and Silberschatz 1986, Melton 1993a]. While vendors may add extensions aimed at their niche, SQL-compliance enables the porting of database systems from platform to platform, vendor to vendor, as least common denominator among the 50-some commercial DBMSs. The ubiquitous quality of SQL allows for more open systems in the database realm, connecting to servers, GUIs and other database management systems. In most database management systems, SQL also provides the means for concurrency control. Because concurrency standards defined by ANSI and OSI for SQL are later detailed as the concurrency standards actually used by database management systems, a brief history of the language and standards process is in order [Melton 1993a and 1993b, Melton and Kulkarni 1992a and 1992b, Sayles 1992, O'Neil 1994].

Developed in 1972 at IBM Research in Yorktown Heights, N.Y., or, depending on the citation, in 1974 at IBM's San Jose Research Laboratory, it was first called SEQUEL, became SEQUEL/2 by roughly 1976, and SQL (Structured Query Language) by the late 1970s. In 1976, the San Jose Research Laboratory published a paper which described a prototype database management system, System R. System R attempted to extend the use of SEQUEL from a simple language enabling query capabilities to one which supported concurrency and system recovery. SEQUEL's expansion included data definition, manipulation and control facilities, precursors to SQL's Data Definition Language, Data Manipulation Language and commands for explicit control of transactions. All locking was automatic [Astrahan et al. 1976].

Following the expansion of SEQUEL by IBM, in 1980 the Oracle DBMS launched the first SQL-based product to be commercially widespread. Initially, IBM developed and sold SQL/DS for a DOS/VSE environment, but it was available by 1982 for a VM/CMS environment. MVS had a version of SQL called DB2, which split from SQL in its implementation of features. Sybase SQL Server, an SQL-based implementation of the client-server model, debuted in 1986, and Digital Equipment Corp added an SQL interface to Rdb/VMS in 1986, thereupon gaining a large share of the VAX/VMS SQL market.

Critical incompatibilities between SQL dialects were bound to occur because of the lack of a standard. Recognizing this, after a series of false starts, the American National Standards Institute (ANSI) and International Organization for Standardization (ISO) both chartered committees to standardize SQL, incorporating features encompassing the least common denominator of the diverse SQL-based vendors, about 1984. The specifications admittedly contained less information than required by a working database system, but were necessary because little common ground was discovered between the products. ANSI published the first SQL standard in 1986, followed by ISO's version in 1987, now known as SQL-86. Both agreed to thoroughly revise once the basic standard was set.

In 1989, ANSI and ISO published revised standards, known as SQL-89. ANSI published a complementary standard, Database Language Embedded SQL, because SQL-89 included embedded SQL in an appendix, rather than an integral part of the document. ISO included an addendum regarding referential integrity features; they did not publish a complementary standard.

In 1992, ANSI and ISO published revised standards for SQL. SQL-92 attempted to make SQL up-to-date with various products and specify common implementation features to bring together incompatible, diverse vendor features. SQL-92 was a response to the need for increased functionality and more explicitly and completely specified SQL-89's standards. Many of the features delineated therein mirrored de facto features, or those

already implemented by leading vendors. While SQL-92 lacks standards for triggers, rules, indexes and additional data types, it does specify degrees of isolation for concurrency controls, and remains the de jure standard for SQL until SQL3, now in committee, is published. Reportedly, SQL3 is concentrating on object orientation (one part of the committee is working on MOOSE, or Major Object-Oriented SQL Extensions), advanced relational aspects, triggers, a call-level interface, and contains partitions that SQL-compliant vendors may use or not, depending on their focus.

In SQL-89, referential integrity was limited to defining primary and foreign keys, along with rules to restrict operations calling for violations of these restraints. SQL-92 provides syntax that allow referential constraints to cause DELETEs and UPDATEs to cascade to other rows. SQL-92 also improved support for multiple environments, including distributed databases, and client/server including remote connection facilities.

SQL-92 not only specifies these advanced features but also increases the flexibility in standard 2-phase latch protocols. A latch is a short-term exclusive lock taken by a transaction when it reads or writes a data item, to guarantee atomicity. After data access is complete, 2PL protocols continue to hold the lock, i.e., ensuring it is a long-term lock, held until the transaction aborts or commits. Lower isolation levels allow short-term locks, or latches, for lock release immediately after the operation is performed.

Although serializable transactions can be guaranteed by a scheduler mandating the two-phase locking (2PL) protocol, since 2PL retains all acquired locks until a commit or rollback by the transaction, concurrency is greatly diminished. As more concurrent transactions are added, lock conflicts happen, and more transactions find themselves in a WAIT state. The number of deadlocks, aborts and consequent retries increases. In some cases, increasing the number of concurrent transactions reduces the number of transactions that are active, and not blocked in the WAIT state, reducing the effective concurrency level. Thus, the CPU is not used fully because a high level of concurrency is impossible.

Given the high-pressure placed upon quick execution of today's database management systems, professionals espoused diluting locking protocols to improve performance. SQL-92 offers isolation levels to achieve this cause. Prior to executing an SQL statement initiating a transaction, a SET TRANSACTION statement sets the isolation level, e.g., SET TRANSACTION SERIALIZABLE in an ad hoc query, or EXEC SQL SET TRANSACTION SERIALIZABLE in embedded SQL. Additionally, there are provisions for declaring subsequent transactions will be READ ONLY or READ WRITE. Each level codifies the degree to which one transaction's operation is affected by concurrent transactions by specifying phenomena which may occur during concurrent SQL transactions. Each level is more restrictive, with transactions executing at isolation level 4 being serializable [Celko 1992, O'Neil 1994]. The DBA may also simply allow the system's default locking capabilities to run, but the presence and use of these isolation levels in current database management systems indicates their general use. Relational database mangement systems may implement these isolation levels in any manner they wish, from traditional 2PL for isolation level 4, and in fact there are different proprietary solutions. Compliance with SQL-92 is an assertion that these isolation levels are achievable, not that traditional concurrency protocols are unavailable. The isolation levels are discussed in more detail following Figure 4.1.1, but are listed here:

1. Read Uncommitted, or "dirty read;"
2. Read Committed, or "cursor stability;"
3. Repeatable Read;
4. Serializable.

Figure 4.1.1 defines which locks are held long-term for each isolation level. Different locking behavior is exhibited for rows in tables and predicates in WHERE clauses. "Notice that it is possible for a scheduler to support concurrently executing transactions of different isolation levels in the same transactional workload" [O'Neil 1994, 680-681].

	Write locks on rows of a table are long term	Read locks on rows of a table are long term	Read and write locks on predicates (in WHERE clauses) are long term
Read Uncommitted (dirty reads)	N/A (read only)	NO	NO
Read Committed (cursor stability)	YES	NO	NO
Repeatable Read	YES	YES	NO
Serializable	YES	YES	YES

Figure 4.1.1. Locking Behavior of SQL-92 Isolation Levels.

4.1.1. Read Uncommitted, or "dirty read"

This isolation level supports no long-term locks. Dirty reads may occur when transaction T1 changes a row and T2 reads T1's change before T1's commit. If T1 performs a rollback before commit, the row T2 has read never really existed. If T1 eventually commits, the values T2 has read still might not have fulfilled atomicity requirements, and may be incorrect. In a large, constantly changing database, where absolute accuracy is not essential, this may not be a significant error, and the database administrator may decide to go for speed. Transactions at this isolation level are not permitted by SQL-92 to perform updates. READ/WRITE is not possible at this level by definition, so "dirty writes" can not occur. A dirty write would occur if T1(WRITE) could be overwritten by T2(WRITE) where T2(READ) < T1(READ) [Celko 1992, O'Neil 1994, Melton and Kulkarni 1992b].

4.1.2. Read Committed, or "cursor stability"

The second row of Figure 4.1.1 shows that while write locks on rows of a table are long term, read locks on rows of a table and read and write locks on WHERE predicates are short term. A transaction at this isolation level cannot read or write to rows written by another transaction until the other transaction commits. Unlike the previous level, where reading uncommitted (dirty) updates was possible, here it is not. This level will allow the "nonrepeatable read" anomaly to occur, where a transaction, T1, opens a cursor on a set of rows, fetches and reads rows. The read lock is held only while the cursor is in a row, and released immediately afterward. If T1 reads the rows a second time, some of the values may have been changed by another transaction. DB2 and other database management systems have offered this level as a feature for some time, calling it cursor stability, in the interests of concurrency. In practice, a type of read lock is held on rows selected by the cursor. When the cursor moves past a row, the lock is released unless the row has been updated, in which case a write lock is imposed.

Also, what if T1 depended on the relationship between two rows? T1 reads Row 1, T2 changes the value of Row 2 and commits, and T1 now reads the changed value of Row 2.

A more serious anomaly occurs when transaction T1 reads >= one row, using them to update or insert another row into the same table. T2 concurrently performs a similar operation. The interleaving of the transactions does not match serial execution. Although SQL-92 enables these faults, it is assumed database administrators (DBAs) will be aware of possible faults, and balance their importance against production needs [Celko 1992, O'Neil 1994, Melton and Kulkarni 1992b].

4.1.3. Repeatable Read

This isolation level supports 2PL for locks on rows of tables, and solves all the problems mentioned for isolation level 2. Long-term locks are held on all accessed records.

Phantom updates remain a problem in this level. A phantom update occurs when transaction T1 reads n rows which satisfy a search condition; T2 generates additional row(s) which satisfy the search condition T1 used; If T1 repeats the search, a different set of rows is found. For example,

T1: SELECT NAME FROM STUDENTS WHERE HOME.STATE = "OHIO"

RESULT: N names of students

T2: INSERT INTO STUDENTS VALUES ("CAROL WILZ", 111-11-1111, "OHIO")

T1: SELECT NAME FROM STUDENTS WHERE HOME.STATE = "OHIO"

RESULT: N+1 names of students

4.1.4. Serializable

This isolation level supports everything isolation level 3 does, plus adds predicate locking, which eliminates phantom updates. A basic problem with 2PL is that prior to this point it has been assumed that locking table rows is enough. Serializability demands that all data accessed by a specific transaction be included in the precedence graph. In other words, in isolation level 4 serializability demands long-term read and write locks on predicates. Some systems lock tables to achieve long-term read and write locks on predicates, using coarse granularity.

SQL-92 contains a rule that if a database management system desires SQL-92-compliance but does not wish to support one of these isolation levels, it must yield one at least as secure as the one missing. It is apparent in advertisements in professional magazines that these standards are taken seriously. Most database managements systems cite their outright compliance with SQL-89 or SQL-92 [Celko 1992, O'Neil 1994, Melton and Kulkarni 1992b].

4.2 Serializability in SQL-92

It is difficult not to notice that SQL-92 encourages some of the very problems cited as an argument for serializability, i.e., dirty reads and unrepeatable (or nonrepeatable) reads. SQL-92 enables the DBA to make a rational decision concerning the importance of correctness versus concurrency.

4.3 Granularity in SQL Locking

How important is the granularity at which SQL obtains locks? Granularity defines the level at which locking is accomplished, be it row or page, and is often cited in the articles detailing how to tweak databases for performance. Automatic locking may occur at the tablespace, page, table, or row level. Although the DBA may specify one of these in some cases, SQL may upgrade the granularity level depending on the transaction. Section 4.3 goes into greater detail regarding advice to professionals for optimization of locking granularity.

A tablespace in a database management system is the "basic allocation medium . . . out of which tables and indexes as well as other objects requiring disk space receive their allocations. A tablespace corresponds to one or more operating system files and can span disks." [Salem et al. 1994, 456]. Most database management systems contain several tablespaces, including one called SYSTEM, which contains system catalog, index and authorization tables. Creating different tablespaces allows DBAs to load balance disk drives, and to bring down part of the system while allowing the remaining database to remain on-line. An optional clause exists in SQL to allow the DBA, creating a table or index, to specify the tablespace from which disk space will be appropriated. If none is specified, the tablespace defaults to the user's default tablespace. The SQL command to create a tablespace, while specifying contained file sizes in megabytes is:

```
CREATE TABLESPACE TSPACE2 DATAFILE 'FILE1' SIZE 200M,
    DATAFILE 'FILE2' SIZE 300M;
```

Pages are physical sequences of disk storage, equal-sized, and are utilized from a main memory buffer shared by all concurrent transactions. A page may contain the contents of part of a table (or tablespace), or an entire table (or tablespace). Its size is determined by the

database management system; the optimizer's purpose in a database management system is to minimize disk page accesses.

What defines a table in a database management system? Most likely, it is comprised of an entity, or class, along with its attributes, including a "primary key," or unique identifier for each specific instance of an entity. At a university, for example, "student" or "professor" would be a separate entity in a relational schema. Attributes such as "Social Security number" and "name" would serve as the primary key, and other attributes such as "address," "phone," or "salary" become accumulate to hold additional data. Locking at the table level would hold an entire table hostage, as well as any related tables defined in foreign keys (the primary key of another table, referenced by a table as an integrity constraint). A row is a specific instance in a table--in this case, it would be a certain person and their pertinent data. An example of each follows:

```
CREATE TABLE STUDENT
(SSN            CHAR(10)        NOT NULL,
NAME            CHAR(20),
GRAD_YEAR       INTEGER,
MAJOR           CHAR(10)
PRIMARY KEY (SSN)
FOREIGN KEY (NAME) REFERENCES APPLICANT)
```

Rows in this table might include:
"111-11-1111", "JANE DOE", 1994, "SAN"
"222-22-2222", "JOHN SMITH", 1995, "ACCT"

As mentioned in the granularity section, the less restrictive the locking mechanisms employed, the greater the possible concurrency. As such, tablespace is the most restrictive, then table or page, then row [Celko 1992, O'Neil 1994, Melton and Kulkarni 1992b].

4.4 DB2

Current articles abound in professional journals concerning fine-tuning database performance. Tradeoffs between concurrency and performance are both expected and

implemented to a degree unrecognized in textbook theory. Articles detail ways of balancing concurrency with the overhead required to support integrity and isolation, including management of locking conflicts and resolving deadlocks and timeouts to avoid degradation of performance because of unrealizable resources, and excessive CPU or virtual storage utilization.

DB2 is a well-known database management system which is SQL-92-compliant, since it provides for the necessary isolation levels. DB2 also attempts to meet the current definition of open standards. It runs on top of numerous operating systems, is accessible through different proprietary middleware, and can be matched with a number of servers. While DB2 contains default values for transaction management, it also allows the Database Administrator to tweak these values for better execution through the DB2 system, the database and the application.

In DB2, locking contention may occur for batch or online processes and will be exacerbated in systems requiring concurrent batch and online processing. Locks consist of a size, mode and duration. More than one type of lock may exist for a specified user in a specific tablespace, depending on the mode of lock. For example, locks might be held simultaneously on the tablespace, table and page for a specific tablespace [for this whole section: Donoghue 1992, Mullins 1991, Ingrassia 1991, Fosdick and Garcia-Rose 1992, Fratarcangeli 1992, Bischoff and Yevich 1992, Backs 1991].

Sections 4.3.1 through 4.4 specify and define the lock sizes, scope and mode in DB2. Figure 4.3.2.1 illustrates the varying types of locks an SQL statement can use, and section 4.5 discusses the future expansion of performance tuning in DB2.

4.4.1. Locks

Lock size and scope describe the locking granularity employed by DB2. SQL sets the lock size by default; if the DBA specifies a certain isolation level, the granularity at which locks are set may be different than the default granularity.

Lock Size	**Scope**
Tablespace	Locks the entire tablespace
Partition	Locks only a partition. At this time, is used solely by the recover utility, but future releases expect all processes to use partition locks on indices and data components
Table	Locks a single table, but only if the tablespace is partitioned
Page	Locks a single page in a tablespace
Subpage	Locks one subpage contained in an index. If the value of subpage = 1, the entire page will be locked

4.4.2 Sanctioned Lock Modes

Each lock size utilizes a mode while it is in force, so these modes may be used with lock sizes of tablespace, partition, table, page or subpage. Sanctioned lock modes in DB2 include:

Mode	Meaning
IS	Intent Share. Page locks in use on individual pages. The lock holder has S locks on the pages, and other users may read and update.
S	Share. Table or Tablespace lock is in use; cannot be updated, for read-only use.
IX	Intent Exclusive. Page locks in use on individual pages. The lock holder may have S, U, or X locks on pages. Other users may read and update.
U	Update. Table or Tablespace lock is in use; may be read by anyone, but only the user holding the lock can update.
SIX	Share with Intent Exclusive. Page locks in use on individual pages. An IX locks already exists, and an S must be acquired. The lock holder may have share S, U or X locks on the pages. Other users may read, but no user other than the holder may update.
X	Exclusive use for update of table or tablespace. Table or Tablespace lock is in use, with no other users permitted.

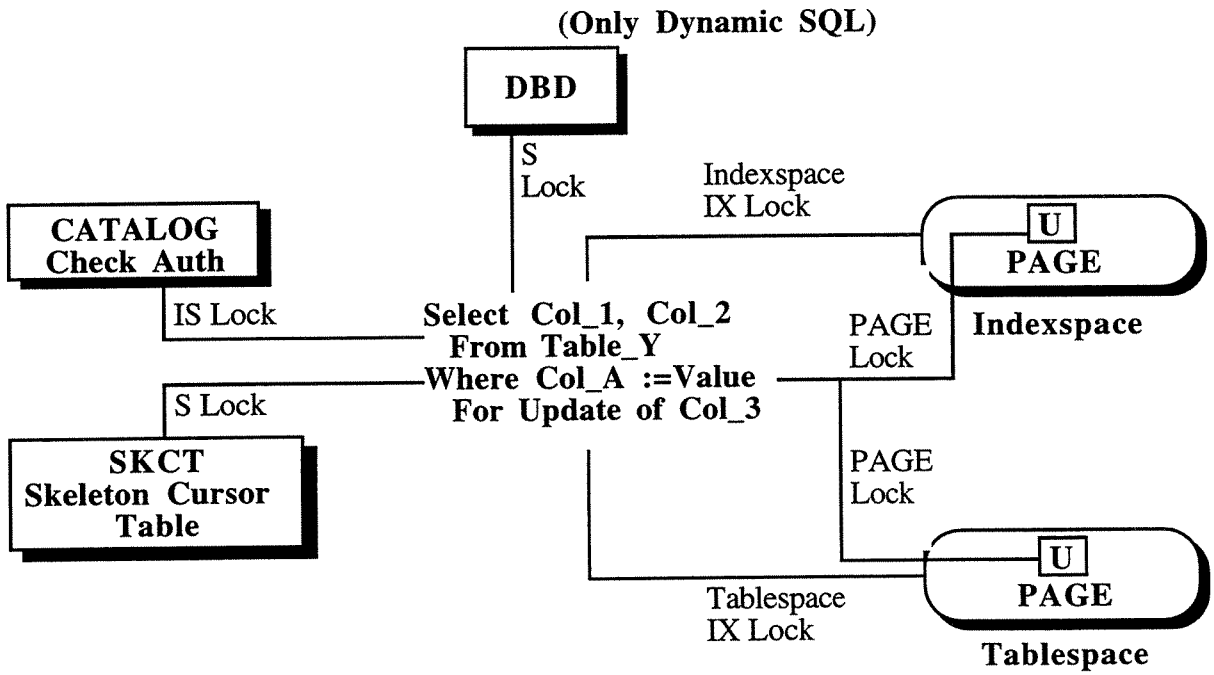


Figure 4.4.2.1. Varying Types of Locks Used by an SQL Statement

[Bischoff and Yevich 1992, 27-38].

4.4.3. Table and Tablespace Locks

An SQL statement acquires a table or tablespace lock whenever it accesses a table or index, including a tablespace lock on all indexes for all tables in a tablespace. Furthermore, if the lock's mode contains an "I", signifying "intent," page locks will also be acquired during execution. An acquired lock's mode is the highest needed to bolster any SQL statement in the plan. Figure 4.4.3.1, placed at the end of this section, illustrates how division of tablespaces can enhance concurrency.

If ACQUIRE(USE) is specified when the query plan is bound, table or tablespace locks are procured when the first SQL statement for a table is issued; only the mode required for that statement is used. The lock may be upgraded upon the issuance of another SQL statement requiring a higher mode. If ACQUIRE(ALLOCATE) is specified when the plan is bound, table or tablespace lock acquisition is mandated for all tables in the plan, at the time of plan allocation. Use of RELEASE (DEALLOCATE) is mandatory to bind the plan, thus freeing table or tablespace locks when deallocation of the plan occurs. RELEASE(COMMIT), used at bind time for a plan, allows SQL to free table or tablespace locks only upon issuance of a COMMIT or ROLLBACK, directly through SQL or indirectly through some proprietary relational database management systems.

Under some relational database management systems, table or tablespace locks may be held for an extended period by specifying ACQUIRE(ALLOCATE) or RELEASE(DEALLOCATE) at bind time.

4.4.4. Page Locks

Page locks can be one of three different modes: S, U, or X (see section 4.3.2). S confines other users to read-only use of a given page. S or U locks may be acquired by other users. In SQL, using a SELECT or FETCH command without UPDATE procures an S lock; an S lock is released when: (1) a TSO process issues an SQL COMMIT or

ROLLBACK is issued; (2) an IMS process issues another SYNCH or ROLB; (3) in CICS, when a synch point call occurs; (4) when a noncursor, non-update SQL statement completes; or (5) when the cursor position moves onto a new page without update, or the cursor closes. It is possible to hold locks over COMMIT and ROLLBACK processes in several cases, but in any case, the lock will be held on a page until DB2 gets a new lock on a new page.

U permits users to UPDATE. Other users can acquire only S locks. In SQL,using an UPDATE or DELETE statement without a cursor procures a U lock.

X permits users to UPDATE in an exclusive mode. The table or tablespace is completely locked to other transactions.

4.4.5. Index Locks

Index locking may occur at the page or subpage level. Table or tablespace locks set by an SQL statement are used equally on all indexes for all tables in a tablespace.

4.5. Simulation of Row Level Locking

Given high concurrency requirements and lack of row locking in DB2, many articles recommend simulating row-level locking. Dummy columns are added to increase a row length, so it is larger than half a page; when the row size is greater than 2K, each page will contain one row. One advantage of this procedure is seen in data synchronization processes where one process has a well-defined domain, but must allow other transactions to use other domains. Another is seen when a transaction updating one row does not lock more than that row.

4.6 Expansion of Performance Tuning

In a survey conducted in 1992 by Howard Fosdick and Linda Garcia-Rose, a quarter of the 545 respondents from metropolitan-area companies, comprising approximately 5,000 DB2 production applications, planned to install DB2 performance tuning tools. This percentage was matched by plans to acquire SQL Optimization tools.

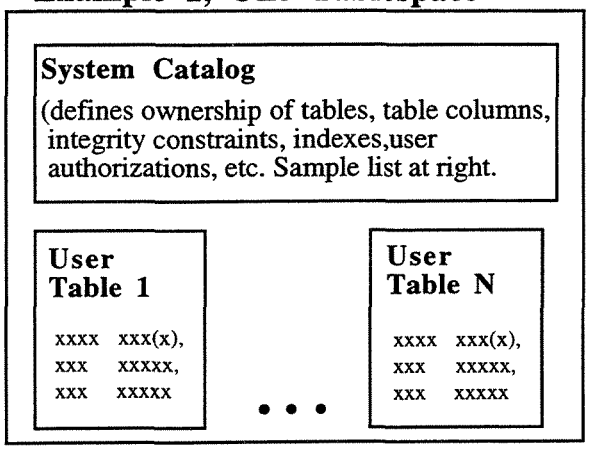
Performance tuning tools extend performance optimization choices to DBAs and programmers. Professionals will shortly be required to understand differences between previously defaulted tablespace scans, indexed access, and simple and partitioned tablespaces, as well as concurrency controls. [Fosdick and Garcia-Rose 1992, Backs 1991, Bischoff and Yevich 1992].

4.7 Summary

In Section 3, Taxonomy of Traditional Concurrency Control Protocols, serializability was demonstrated to be the most important consideration of traditional concurrency control theory. Earlier in this section, Concurrency in Practice, the concept of serializability is still important, but performance considerations intrude. Professional SQL standards committees have codified de facto proprietary standards which allow weakened serializability, in the form of SQL-92's isolation levels 1, 2 and 3. Absolute serializability is assured in SQL-92's isolation level 4. In DB2, this level of isolation employs a 2 phase locking protocol, in a merger of traditional and practical concepts.

The next section, 5, Future Trends, expands a bit on section 4.6, and theorizes about other possibilities in relational database management systems. Division 6 will further discuss the relationship between traditional and practical concurrency controls in relational database management systems, and draw some conclusions based on this work.

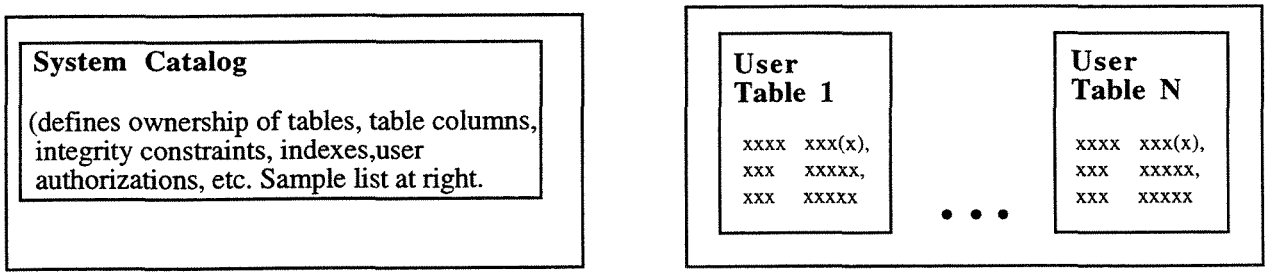
Example 1, One Tablespace



The System Catalog contains 16 files that define a system's authorizations, definitions, usage and views at a given time. They are: SYSACCESS, SYSCATALOG, SYSCHARSETS, SYSCOLAUTH, SYSCOLUMNS, SYSDBSPACES, SYSDROP, SYSINDEXES, SYSOPTIONS, SYSPROGAUTH, SYSSYNONYMS, SYSTABAUTH, SYSUSAGE, SYSUSERAUTH, SYSUSERAUTH, SYSUSERLIST, AND SYSVIEWS.

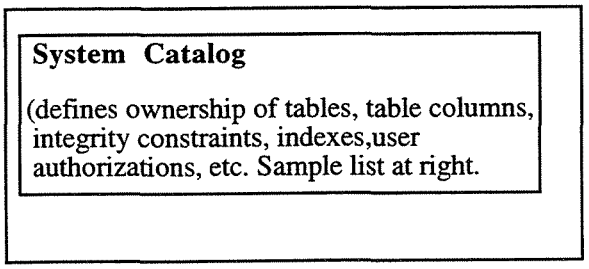
This type of "one tablespace fits all" structure denotes an inexperienced DBA. If a table is locked for data access, the whole system may be locked, through escalating locks.

Example 2, Two Tablespaces



This is the more standard way of specifying tablespaces, with the System Catalog in one tablespace, and user tables in another. Note that placing multiple tables in one tablespace conserves resources and may speed access, but reduces possible concurrency if the tablespace is locked.

Example 3, Multiple Tablespaces



This sample shows tablespaces arranged for maximum optimization under locking. Every table is in a separate tablespace. Although locking is an expensive operation, made cheaper by taking table or tablespace locks rather than page locks, concurrency is reduced.

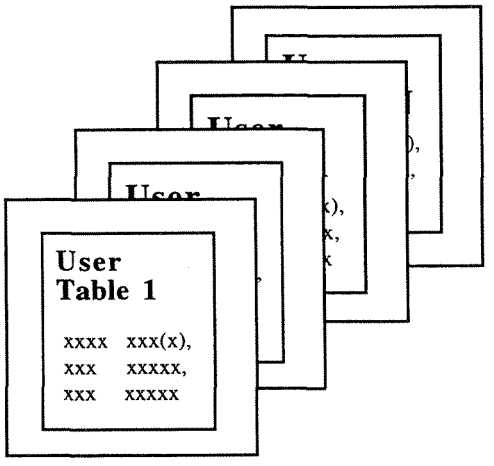


Figure 4.4.3.1 Examples of Tablespace Division

5. Future Trends

As stated, there is divergence between academia and proprietary research concerning the future of relational database management systems. While the issues of atomic concurrency vs. performance will continue to be debated and implemented through successive issues of SQL committees and database management systems, there are some indications of the direction of relational database management systems in general, as well as concurrency optimization suggestions and mandates for performance tuning.

5.1 Relational Database Management Systems

". . . Relational technology is the strategic direction in virtually all Fortune 500 companies, and DB2 is a fait accomplis in the MVS-based relational world (some predict that by the end of 1993, nearly 10,000 DB2 licenses will exist worldwide)" [Sayles 1992]. Despite claims of slow access and bad response time in relational systems, there are many success stories. Jim Melton states, ". . . It's like these people have never heard of the hundreds of banks using relational systems that run hundreds of queries per second on gigabytes (or even terabytes) of data." (Melton 1993).

The push toward object-oriented database management systems is hampered by lack of standards among the many small companies producing the technology. The reluctance of recently-become relational businesses to switch again has improved the position of the relational database management system companies, who have satisfied their customers by adding object extensions onto relational systems, for example, Oracle7 includes object-oriented extensions and supports C++ [Fratarcangeli 1992].

Although some applications require total object-orientation, it is likely that the hybrid database management systems will proliferate [Cattell 1991, Elmasri and Navathe 1994, Herlihy 1990, Melton 1993b].

5.2 Optimization of locking

Query optimization, which occurs at bind time and seeks to optimize the query path for SQL statements, is such a part of relational database management systems that is difficult to envision the problems their lack would create. Given faster hardware, it is only a matter of time before companies test locking optimizers. The tests may prove that the overhead is still too expensive, but an expert system approach to concurrency control, given a mixture of pessimistic and optimistic controls, is a fascinating offshoot of today's hybrid concurrency control protocols and SQL's in-place query optimization.

5.3 Continued Expansion in Field of Performance Tuning

As many computer professionals become more specialized than in the past, it is not inconceivable that certifications equivalent to CNEs (Certified Novell Engineer) will become necessary. Given the sleight of hand performance tuning tricks written about now, as performance tuning tools increase, expertise will be at a premium. As relational databases take over mission-critical applications, performance becomes more important, and there will always be limitations to hardware solutions. Given that one reason relational database management systems have succeeded is their ability to be part of an open systems framework, rather than be based on specific hardware configurations, software tweaking will continue--as will the database industry's attempt to catch up to actual use requirements.

6. Conclusion

Academic and traditional theories regarding concurrency have centered on a narrow band of concepts that assume serializability is a necessary factor for transactional concurrency in a relational database management system. Journals and textbooks acknowledge that using concurrency controls inhibits concurrency, but are unwilling to acknowledge that other performance factors may be equally compelling, depending on the specific database application.

Perhaps this view is a holdover from the days when databases were always an expensive proposition. Businesses which needed absolutely correct information were pioneers in the purchase of database management systems, and their needs drove the industry. With the advent of inexpensive hardware and user-friendly software, database management systems came within the reach of anyone with a need. If one takes an analytical look at the needs of some applications, serializability is not always necessary. Performance, on the other hand, sells systems and jobs.

ANSI and OSI have codified real needs, balancing concurrency with performance, in SQL-92, which supports all serializable and some non-serializable schedules. Figure 6.1a shows the different levels of possible concurrency. Serializable schedules, ones that are supported by traditional protocols, are shown on figure b. Non-serializable schedules, such as those which support lost updates, dirty reads, nonrepeatable reads and incorrect summaries, are excluded. Figure c shows the SQL-92 isolation levels that are non-serializable such as dirty reads and non-repeatable reads or cursor stability, and figure d shows all the schedules that SQL-92 supports.

Academia and traditional computing gave relational database management systems their start. Until the middle of the last decade, locking protocols such as 2-phase locking were practical for all types of concurrency. However, the increasing use of relational systems and the particular burden they place on supporting concurrent transactions while executing computationally expensive joins places system performance close to concurrency in terms

of importance. While 2PL is still in use, its use is increasingly reserved for a smaller set of operations, those demanding absolute serializability.

Relational database management systems are a new technology. The notion of serializability comes from a time when only serial execution was possible. Serializability is not deterministic since there are N! possible serial schedules for N transactions, and every one of them qualifies as correct according to the traditional viewpoint. Although almost to date, practical relational database management systems have leaned on academic or traditional concepts, maybe it is possible for academic theory to learn from actual practitioners now. Is there a better idea than serializability to use as criteria for a good system? Is it possible for a combination of performance and serializability to replace plain serializability? Until performance and serializability work hand in hand, practitioners will continue to finesse performance issues and the practical will continue to shift away from conventional wisdom.

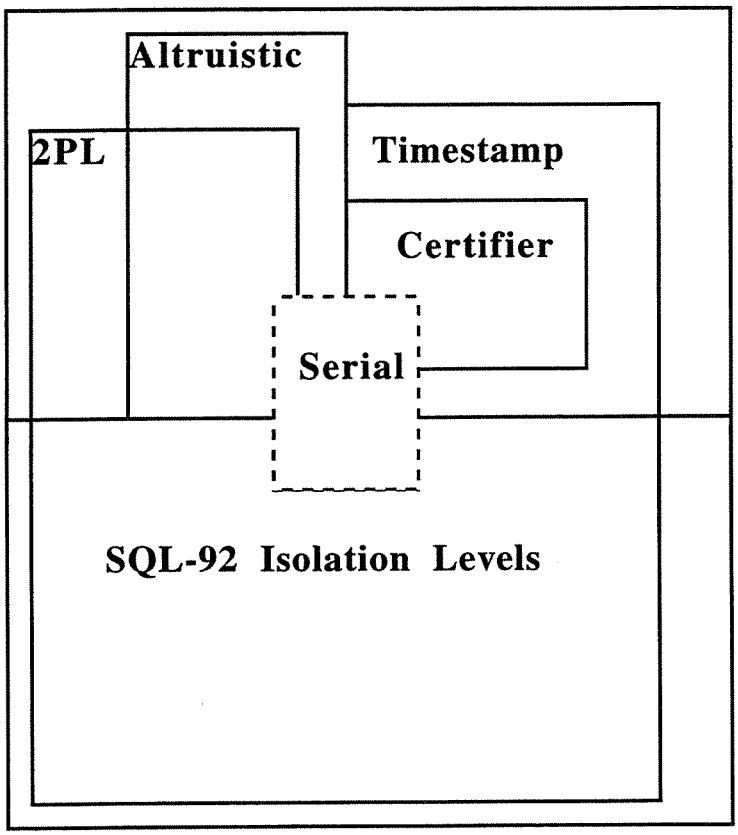
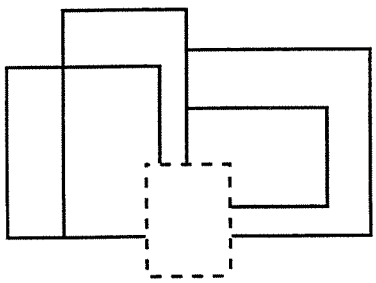
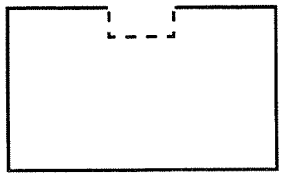


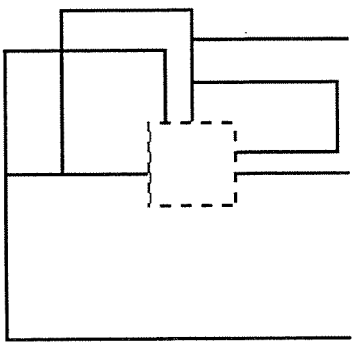
Figure 6.1a All Histories



b. Serializable Schedules



c. Non-Serializable Schedules



d. SQL-92 and Practical Concurre Use All Levels

7. Annotated Bibliography

Agrawal, R. and D. J. DeWitt . 1985. "Integrated concurrency control and recovery mechanisms: design and performance evaluation," *ACM Trans. On Database Systems* 10:4, pp. 529-564.

Backs, William. 1991. "DB2 Free Space," *Database Programming & Design,* June, pp. 22-30. Details how free space is important to accommodate growth, mitigate locking concurrency problems and improve performance. Well written, and a good precursor to Bischoff and Yevich's later article.

Bacon, Jean. 1993. *Concurrent Systems*, Addison-Wesley Publishing Company.

Badrinath, B.R. and Ramamritham, Krithi. 1992. "Semantics-Based Concurrency Control: Beyond Commutativity," *ACM Transactions on Database Systems*, Vol. 17, No. 1, March, pp. 163-199.

Bassiouni, M.A. 1988. "Single-Site and Distributed Optimistic Protocols for Concurrency Control," *IEEE Transactions on Software Engineering*, Vol. 14, No. 8, August, pp. 1071-1080. Focuses on pure optimistic concurrency (as opposed to schemes which integrate it with pessimistic), to increase its concurency and improve performance. Concludes timestamps for both fully and partially duplicated database networks will enhance performance and reduce overhead.

Bell, David and Grimson, Jane. 1992. *Distributed Database Systems*, Addison-Wesley Publishing Company.

Bernstein, Philip A. and Goodman, Nathan. 1981. "Concurrency Control in Distributed Database Systems," *Computing Surveys*, Vol. 13, No. 2, June, pp. 185-221. Presents distributed database concurrency control; decomposes concurrency control into read-write and write-write to find a basis for comparison between the >20 proposed concurrency control algorithms proposed in the literature at this time; includes good summaries. At this early date, concludes optimistic concurrency will not work for distributed systems without a lot of revision.

Bernstein, P.A. and Goodman, N. 1983. "Multiversion concurrency control--theory and algorithms, *ACM Trans on Database Systems* 8:4, pp. 463-483.

Bernstein, P.A. and Goodman, N. 1984. "An algorithm for concurrency control and recovery in replicated, distributed databases," *ACM Trans. on Database Systems* 9:4, pp. 596-615.

Bernstein, P.A., Hadzilacos, V. and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading Mass. Contains extensive references on concurrency control and distributed systems.

Bernstein, Arthur J. and Lewis, Philip M. 1993. *Concurrency in Programming and Database Systems*, Jones and Bartlett Publishers.

Bhargava, Bharat K. 1987. *Concurrency Control and Reliability in Distributed Systems*, Van Nostrand Reinhold Company.

Birman, Kenneth P. 1993. "The Process Group Approach to Reliable Distributed Computing," *Communications of the ACM*, Vol. 36, No. 12, December, pp. 36(17).

Bischoff, Joyce and Yevich, Richard. 1992. "The DB2 Dilemma: Managing Locking Contention," Database Programming & Design, May, pp. 27-38. Specifics on how to tweak better performance in DB2 by innovative definitions for tables and indexes; includes an overview of locking protocols, and a small section on latches.

Boksenbaum, Claude, Cart, Michele, Ferrie, Jean, and Pons, Jean-Francois. 1987. "Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems," *IEEE Transactions on Software Engineering*, Vol. 13, No. 4, April, pp. 409-418. Proposes new certification method for optimistic concurrency, timestamps, to make it viable in a distributed database system; allows a chronological commit order and dependency graph proof.

Cattell, R.G.G. 1991. *Object Data Management*, Addison-Wesley Publishing Company.

Casanova, Marco Antonio. 1981. *The Concurrency Control Problem for Database Systems*, Springer-Verlag.

Celko, Joe. 1992. "An Introduction to Concurrency Control," *DBMS* , September, pp. 70-83. Gives overview of SQL-92 isolation standards, and sometimes facile explanations of concurrency in general. Celko is a member of the SQL standards comittee.

Ciciani, Bruno, Dias, Daniel M., Yu, Philip S. 1992. "Analysis of Concurrency-Coherency Control Protocols for Distributed Transaction Processing Systems with Regional Locality," *IEEE Transactions on Software Engineering*, Vol. 18, No. 10, October, pp. 899-913.

Dan, Asit. 1992. *Performance Analysis of Data Sharing Environments*, The MIT Press.

Date, C.J. 1993. "A Matter of Integrity, Part III," *Database Programming & Design*, December, pp. 19-21. The father of relational database theory insists that SQL-89 and SQL-92 are ridden with inaccuracies and arbitrary complexities, as well as including a diatribe against object-oriented databases.

Date, C.J., with Darwen, Hugh. 1992. *Relational Database Writings 1989-1991*, Addison-Wesley Publishing Company, pp. 485-516.

Davidson, S.B. 1984. "Optimism and consistency in partitioned distributed operations on relational views," *ACM Trans. on Database Systems* 7:3, pp. 381-416.

Donoghue, Michael. 1992. "Committing to Commit," *Database Programming & Design,* October, pp. 48-56. Discusses running batch and online programs concurrently in DB2, including restarting after failure.

Eich, Margaret H., and Wells, David L. 1988. "Database Concurrency Control Using Data Flow Graphs," *ACM Transactions on Database Systems*, Vol. 13, No. 2, June, pp. 197-227.

Elmasri, Ramez, and Navathe, Shamkant B. 1994. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc. This books is a highly-readable survey of database concepts. Sections are included on SQL, Design, System Implementation and Advanced Data Models.

Fosdick, Howard and Garcia-Rose, Linda. 1992. "DB2: Growing in Expertise and Mission-Critical Applications," *DBMS,* pp. 76-80. This interesting examines survey results from DB2 users in metropolitan areas. Includes sections on methodology, numbers and size of respondents, responsibilities, performance problems and DB2 add-on products. Shows cross-section of actual DB2 users, as well as making predictions about the importance of performance tuning in the future.

Franaszek, Peter and Robinson, John T. 1985. "Limitations of Concurrency in Transaction Processing," *ACM Transactions on Database Systems*, Vol. 10, No. 1, March, pp. 1-28. Defines the effective level of concurrency given the pairwise probability of conflict among transactions with the total number of concurrent transactions. Shows results for standard locking, strict priority scheduling, and optimistic methods. Proposes three new concurrency control methods, which partially include optimistic methods.

Franaszek, Peter, Robinson, John T., and Thomasian, Alexander. 1992. "Concurrency Control for High Contention Environments," *ACM Transactions on Database Systems*, Vol. 17, No. 2, June, pp. 304-345. Summarizes current pure optimistic methods as die-based (transaction restarted only when reaches commit and validation is unsuccessful) and kill-based (all transactions conflicting with one which is successful restart immediately). Simulates performance of different concurrency control models, and concludes new techniques offer benefits for high levels of data contention.

Fratarcangeli, Claudio. 1992. "Locking and Referential Integrity in Oracle," *DBMS,* December, pp. 81-99. This article discusses the relationship between integrity constraints and serializability, as well as general concurrency control issues. Includes SQL referential integrity examples, as well as deadlock considerations.

Haerder, T. and Reuter, A. 1983. "Principles of transaction oriented database recovery--a taxonomy," *ACM Computing Surveys* 15:4, pp. 287-217. Organized conceptual discussion of various transaction-oriented recovery protocols. Includes a taxonomy of logging techniques. Purpose of paper is to establish a cogent terminology for this topic.

Herlihy, Maurice. 1990. "Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types," *ACM Transactions on Database Sysems*, Vol 15, No. 1, March, pp. 96-124 .

Horner, Donald R. 1989. *Operating Systems Concepts and Applications*, Scott, Foresman and Company.

Hsu, Meichun and Zhang, Bin. 1992. "Performance Evaluation of Cautious Waiting," *ACM Transactions on Database Systems*, Vol. 17, No. 3, September, pp. 477-512.

Ibaraki, Twoshihide, Kameda, Tiko and Minoura, Toshimi. 1987. "Serializability with Constraints," *ACM Transactions on Database Systems*, Vol. 12, No. 3, September, pp. 429-452.

Ingrassia, Frank J. 1991. "The Day RDS Stood Still," *Database Programming & Design,* March, pp. 54-61. Discusses the evolution of DB2's optimizer, which determines selection of access path. Details how performance may be improved through use of tablespace, index scans, and buffer pools.

Janson, Philippe A. 1985. *Operating Systems: Structures and Mechanisms*, Academic Press Inc.

Kiernan, Casey. 1993. "Treating Them Like Guests," *Database Programming & Design*, Vol. 6, No. 7, July, pp. 32(7).

Kirkwood, John. 1992. *High Performance Relational Database Design*, Ellis Horwood.

Korth, Henry R., and Silberschatz, Abraham. 1986. *Database System Concepts*, Second Edition, McGraw-Hill, Inc. Standard, somewhat outdated, text on databases, focusing mainly on relational model. Appendices on hierarchical, network and object-oriented systems. Tends to go into great detail on traditional concepts, with few real-world examples.

Kung, H.T., and Robinson, John T. 1981. "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, Vol. 6, No. 2, June, pp. 213-226. This is the primary document of optimistic methods for the single site model. Presents the general problem clearly, and proposes families of nonlocking concurrency controls. Uses incremental transaction numbers instead of timestamps.

Lynch, Nancy A. 1983. "Multilevel Atomicity--A new Correctness Criterion for Database Concurrency Control," *ACM Transactions on Database Systems*, Vol. 8, No. 4, December, pp. 484-502.

Manber, U., and Ladner, R.E. 1984. "Concurrency control in a dynamic search structure," *ACM Trans. on Database Systems* 9:3, pp. 439-455.

Melton, Jim. 1993a. "Intergalactic Dataspeak?," *Database Programming & Design*, January, pp. 35-37. Summarizes the history of SQL and its standardization. Delineates where ANSI and OSI documents differ in some cases, and cites exact titles and publication citations for versions of SQL standards. Explains how U.S. National Institute of Standards and Technology and Federal Information Processing Standard Publications specify vendor conformance.

Melton, Jim. 1993b. "The Structure of SQL3," *Database Programming & Design*, July 1993, pp. 65-67. Describes partitioning of SQL3 document, factors determining publication date, and how to order copies of SQL-92 and SQL3.

Melton, Jim, and Kulkarni, Krishna. 1992a. "Is the SQL Standard Too Large?," *Database Programming & Design,* July pp. 21-26. Seeks to explain how 600 page SQL-92 is really only 47% larger than the 120 page SQL-89 should have been, had it covered standards in sufficient detail. Includes comparison chart of selected features and SQL-

86, SQL-89 and SQL-92 number of pages for each feature, as well as analyzing degree of similarity between successive standards.

Melton, Jim, and Kulkarni, Krishna. 1992b. "Out With the Old," *Database Programming & Design*, August, pp. 26-27. Followup to "Is the SQL Standard Too Large?" Continues discussion of new features of SQL-92, including additional relational operators such as SET DIFFERENCE, SET INTERSECTION, join variants, predicates and data types. Discusses increased orthogonality of SQL-92 (where capabilities employed in one part of the language cannot be used in ostensibly equivalent parts).

Mullins, Craig S. 1991. "Mastering Disaster Recovery," *Database Programming & Design,* January, pp. 42-49. Focuses on details of creating a disaster recovery plan. Discusses logs for tablespaces and other recovery mechanisms.

O'Neil, Patrick. 1994. *Database Principles, Programming, Performance* , Morgan Kaufmann Publishers, San Francisco, California. An outstanding reference. Extremely up-to-date. References current products as practical examples of theory.

Papadimitriou, Christos. 1986. *The Theory of Database Concurrency Control*, Computer Science Press.

Papadimitriou, C.H. and Kanellakis, P. C. 1984. "On concurrency control by multiple versions," *ACM Trans. on Database Systems* 9:1, pp. 89-99.

Ries, D.R. and Stonebraker, M. 1979. "Locking granularity revisited," *ACM Trans. on Database Systems* 4:2, pp. 210-227.

Salem, Kenneth, Garcia-Molina, Hector, and Shands, Jeannie. 1994. "Altruistic Locking," *ACM Transactions on Database Systems*, Vol. 19, No. 1, March, pp. 117-165. This well-written and interesting paper proposes "altruistic locking," an extension to two-phase locking, for the case of long-lived transactions (LLTs). LLTs retain database resources for long periods of time; traditional locking greatly inhibits concurrency. Special locking rules are proposed, and altruistic locking is guaranteed to be simple to use and serializable. The term "altruistic" is used because transactions which yield locks early do not benefit from this release--other concurrently executing transactions will.

Sayles, Jonathan. 1992. "Universal Dataspeak Becomes a Relational Reality," *Database Programming & Design*, August, pp. 39-45. Focuses on Information Builders Inc. (IBI), whose FOCUS provides open connectivity and interoperability to various types of computers, using EDA/SQL. EDA/SQL is a suite of client/server products that access nonrelational and relational DBMSs on different networks; general discussion is provided of EDA/SQL benefits, including API/SQL calls.

Shasha, Dennis. 1993. "Database Tuning: Principles and Surprises: Strategies for Database Optimization," *Dr. Dobbs Journal,* Vol. 18, No. 4, April, pp. 532(3).

Silberschatz, A. and Kedem, Z. 1980. "Consistency in hierarchical database systems," *J. ACM* 27:1, pp. 72-80.

Singhal, Mukesh and Shivaratri, Niranjan G. 1994. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*, McGraw-Hill, Inc.

Small, Carol, Poulovassilis, Alexandra and Derakhshan, Mir. 1992. "The Role of a Repository Manager in Deductive Database Systems," in *Expert Database Systems*, Academic Press Limited, pp. 83-116.

Talen, Waldemar. 1993. "Client-Side Performance Tuning," *DBMS* , Vol. 6, No. 12, November, pp. 62(5).

Tate, Bruce, Malkemus, Tim and Gray, Terry. 1993. *Comprehensive Database Performance For OS/2 2.0's Extended Services*, Van Nostrand Reinhold.

Thomas, R.H. 1979. "A majority consensus approach to concurrency control," *ACM Trans. on Database Systems* 4:2, pp. 180-219.

Ullman, Jeffrey D. 1988. *Principles of Database and Knowledge-Base Systems, Vol. I*, Computer Science Press.

Viehman, Peggy. 1994. "24 Ways to Improve Database Performance," *Database Programming & Design*, Vol. 7, No. 2, February, pp. 32(7).

Vogt, F.H., ed. 1988. *Concurrency 88: International Conference on Concurrency, Hamburg, FRGT, October 18-19, 1988 Proceedings*, Springer-Verlag.

Wertz, Charles J. 1993. *Relational Database Design: A Practitioner's Guide*, CRC Press.

Wolfson, O. 1987. "The overhead of locking (and commit) protocols in distributed databases," *ACM Trans. on Database Systems* 12:3, pp. 453-471.

Yasin, Asad M. 1993. "Data Conversion and Audits," *Database Programming & Design*, February, pp. 25-27. Able argument for data audits, or a methodical process of classifying and validating data, both usable and nonusable, and their components. Proposes data audits should be performed before conversion, and on a regular basis thereafter, just as accountants verify monetary transactions. Purpose of audit: identify possible errors, thus certifying data is accurate and trustworthy.

Yonezawa, A., Ito, T., eds. 1991. "Concurrency: Theory, Language, and Architecture," *UK/Japan Workshop, Oxford, UK, September 25-27, 1989, Proceedings*, Springer-Verlag.

Yu, Philip S., and Dias, Daniel M. 1992. "Analysis of Hybrid Concurrency Control Schemes for A High Data Contention Environment," *IEEE Transactions on Software Engineering*, Vol. 18, No. 2, February, pp. 118-129. Contends that optimistic concurrency control is subject to resource contention because of aborted transactions. Proposes a large buffer, so data from these transactions can be accessed during rerun. Proposal ensures pure optimistic concurrency control which aborts at commit, performs better than broadcast optimistic concurrency control, which aborts as soon as conflict is detected.

Yu, Philip S., and Dias, Daniel M. 1993. "Performance Analysis of Concurrency Control Using Locking with Deferred Blocking," *IEEE Transactions on Software Engineering*, Vol. 19., No. 10, October, pp. 982-995. Proposes a combination of pessimistic and optimistic methods, whereby optimistic concurrency control allows aborts only during the early stages of a transaction's execution, and then waits for locks to be released.

TRADITIONAL VIEWS OF DATABASE CONCURRENCY VS. PRACTICAL CONCURRENCY IN RELATIONAL DATABASE MANAGEMENT SYSTEMS: A SURVEY

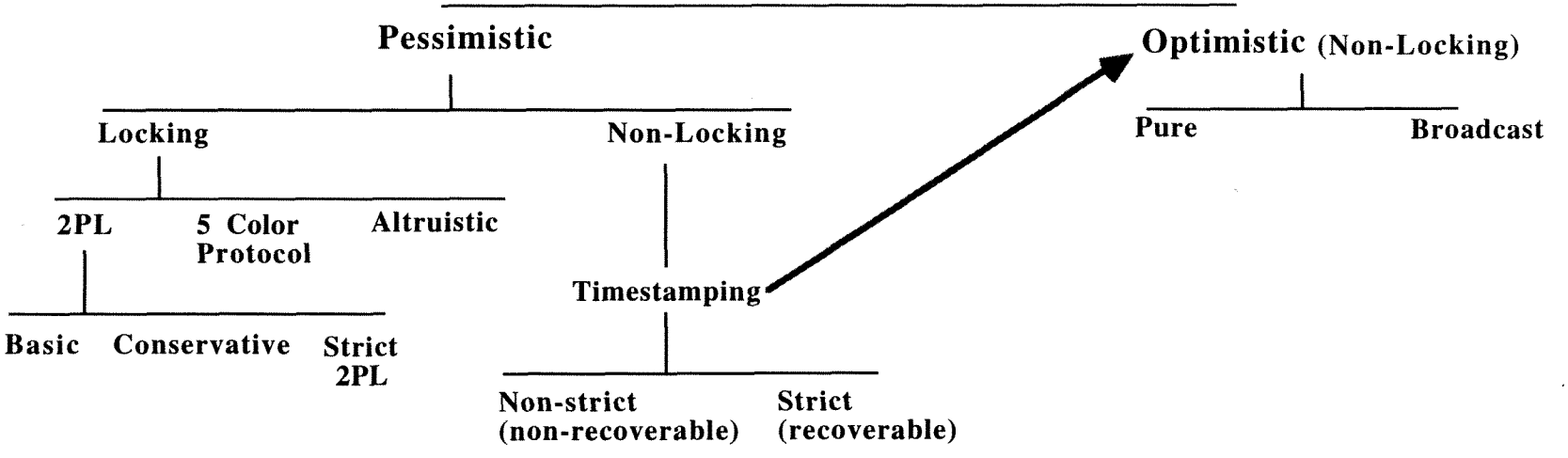
Department of Systems Analysis

Patricia K. Geschwent

Fall, 1994

Table of Contents

Taxonomy of Concurrency Control



PESSIMISTIC concurrency controls assume there will be contention for data, and prevents conflict by locking items in use or by executing transaction operations in the order of the transactions' timestamps.

OPTIMISTIC concurrency controls assume data contention will be small, and arbitrates possible conflict by checking a series of rules at validation phase.

2 PHASE LOCKING PROTOCOL

One of the most common concurrency control protocols ensures transaction isolation by locking data items.

Methods:

Guarantees serializability by:

1. requiring that all locks precede all unlocks
2. The scheduler checks that two transactions do not hold locks on the same item simultaneously

Types of locks:

READ locks (RLOCK)

1. prevent other transactions from writing to the held item until RLOCK is lifted
2. >1 transaction can hold RLOCKS at one time
3. cannot obtain a RLOCK if a WRITE lock is in effect

WRITE locks (WLOCK)

1. any transaction wishing to change a value must obtain a WLOCK
2. only 1 transaction may hold a WLOCK on a specific item at any time
3. if only one transaction holds a RLOCK on an item, and wishes to change the item, the lock can be escalated to a WLOCK

TIMESTAMPING PROTOCOL

Each transaction is assigned a unique timestamp to guarantee that conflicting operations are executed in the order of the transaction's timestamp.

Timestamps are issued by the scheduler assigning the next number to each new transaction, or the database management system may use the value of the machine's internal clock when a transaction initiates.

Each item in the database is given two timestamps

READ TIMESTAMP (tr)

1. highest timestamp held by any transaction which has read the item

WRITE TIMESTAMP (tw)

1. highest timestamp held by any transaction which has written the item

Rules to be checked:

if X=READ and

1. T >= tw, set read time to t if t > tr
2. T < tw, abort the transaction

if X=WRITE and

1. t >= tr and t >= Tw, set write time to t if t > tw
2. tr <= t < tw, execute
3. t < tr, abort the transaction

OPTIMISTIC PROTOCOL

Assumes no conflict will occur and proceeds with the transaction using a local copy. A validation phase checks to see if the assumption is correct. If not, the transaction is aborted and restarted.

Divides every transaction execution into three different phases:

READ PHASE:

1. Ti executes. Values of data items are read and stored in local variables. Write operations are performed here.

VALIDATION PHASE:

1. Ti performs a validation test to check if it can copy its local variables to the database without violating serializability. Three timestamps are necessary:
   Start(Ti), when Ti starts executing
   Validation(Ti) when Ti finishes the read phase and begins validation phase
   Finish(Ti), when Ti finishes its write phase.

Rules for serializability:

1. Finish(Ti) < Start(Tj).
2. Ti intersection Tj = 0.
3. Start(Tj) < Finish(Ti) < Validation(Tj).
4. Write(Ti) intersection Read(Tj) = 0.

WRITE PHASE

1. If validation is successful, the values in local variables are written to the database

STRICT 2-PHASE LOCKING
requires:

1. A transaction will not be written into the database until it has reached its commit point
2. A transaction will not release any locks until after the commit point (this avoids cascading rollbacks)

STRICT TIMEBASED PROTOCOL
requires:

1. All updates are performed only in the workspace
2. All updates are written into the database after the transaction commits. Cascading rollback is avoided

Advantages:

1. Distinguishes between reads and writes and their effect on the database
2. Guarantees serializability regardless of the types of transactions which could operate concurrently with a given transaction
3. Good for update-intensive applications because it is safe

1. Different from locking, because the blocked transaction aborts rather than waits for access
2. >1 transaction can read the same item at different times, conflict-free
3. Enhanced concurrency over phased locking because transactions do not block each other needlessly

1. Does not inhibit access prior to validation phase because emphasizes arbitration of conflict between transactions, not prevention
2. High concurrent access possible
3. Superior for query intensive databases or other systems with a low conflict rate

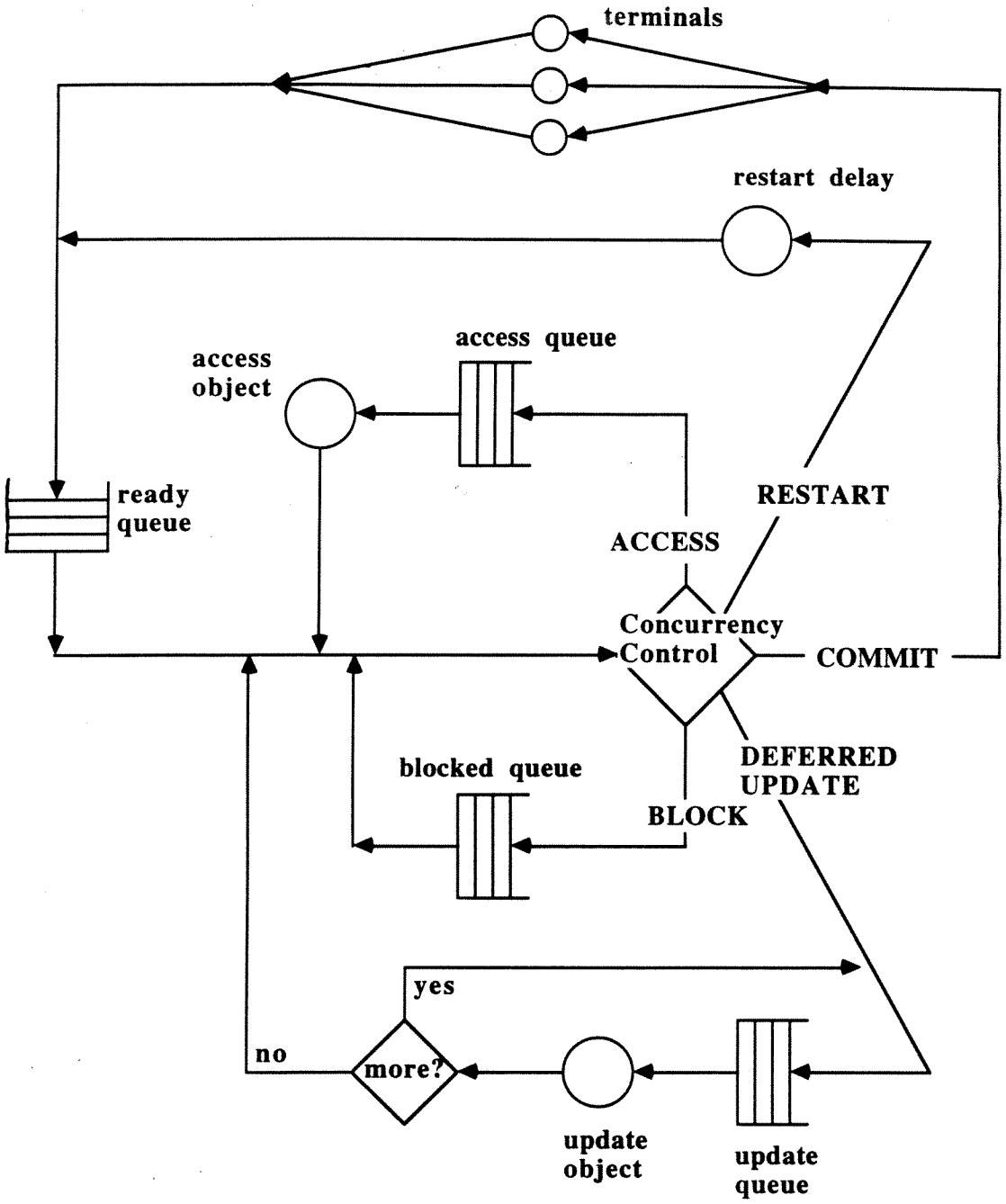
Disadvantages:

1. Inhibits concurrent execution because of locking overhead
2. May lock an item no other transaction needs
3. Inefficient in query-intensive applications because of locking overhead, possibility of deadlock and waits for locked data.

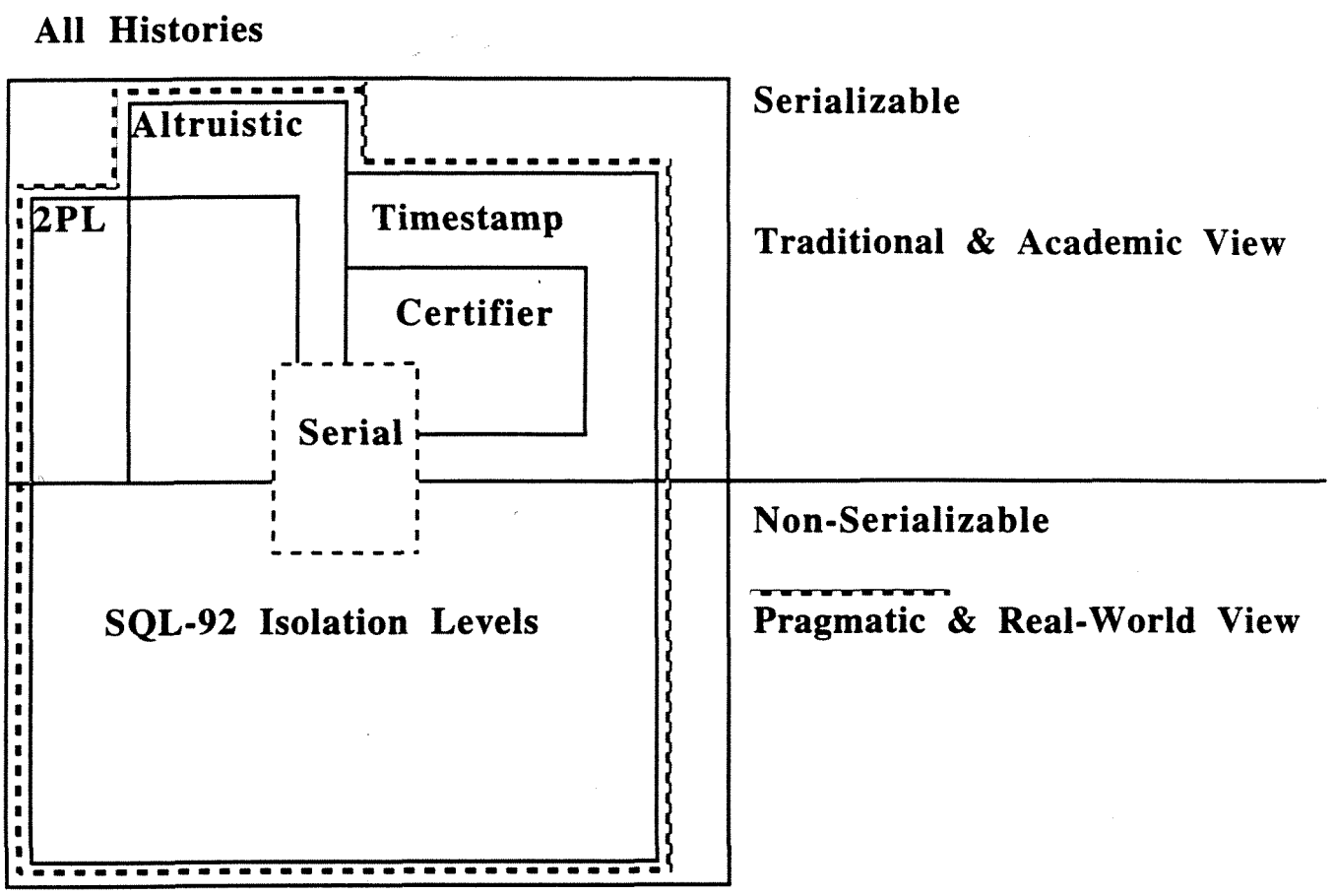
1. Inefficient where aggressive locking makes sense (where >1 transaction executing simultaneously require the same item). A large amount of rollbacks will occur
2. Timestamp checking done prior to a commit point, because you cannot abort after a commit

1. Not efficient in high contention, frequent-update systems.
2. May abort more transactions than either previous method because checks timestamps later.
3. Not as intuitively understandable as the others.

Logical Transaction System Model



Taken from Salem, Kenneth, Garcia-Molina, Hector, and Shands, Jeannie, "Altruistic Locking," ACM Transactions on Database Systems, Vol. 19, No. 1, March 1994, pp. 140.



	Write locks on rows of a table are long term	Read locks on rows of a table are long term	Read and write locks on predicates (in WHERE clauses) are long term
Read Uncommitted (dirty reads)	N/A (read only)	NO	NO
Read Committed (cursor stability)	YES	NO	NO
Repeatable Read	YES	YES	NO
Serializable	YES	YES	YES

Figure 4.1.1. Locking Behavior of SQL-92 Isolation Levels. Different locking behavior is exhibited for rows in tables and predicates in WHERE clauses. "Notice that it is possible for a scheduler to support concurrently executing transactions of different isolation levels in the same transactional workload." (O'Neil 1994, p. 680-681)

Varying Types of Locks Used by an SQL Statement

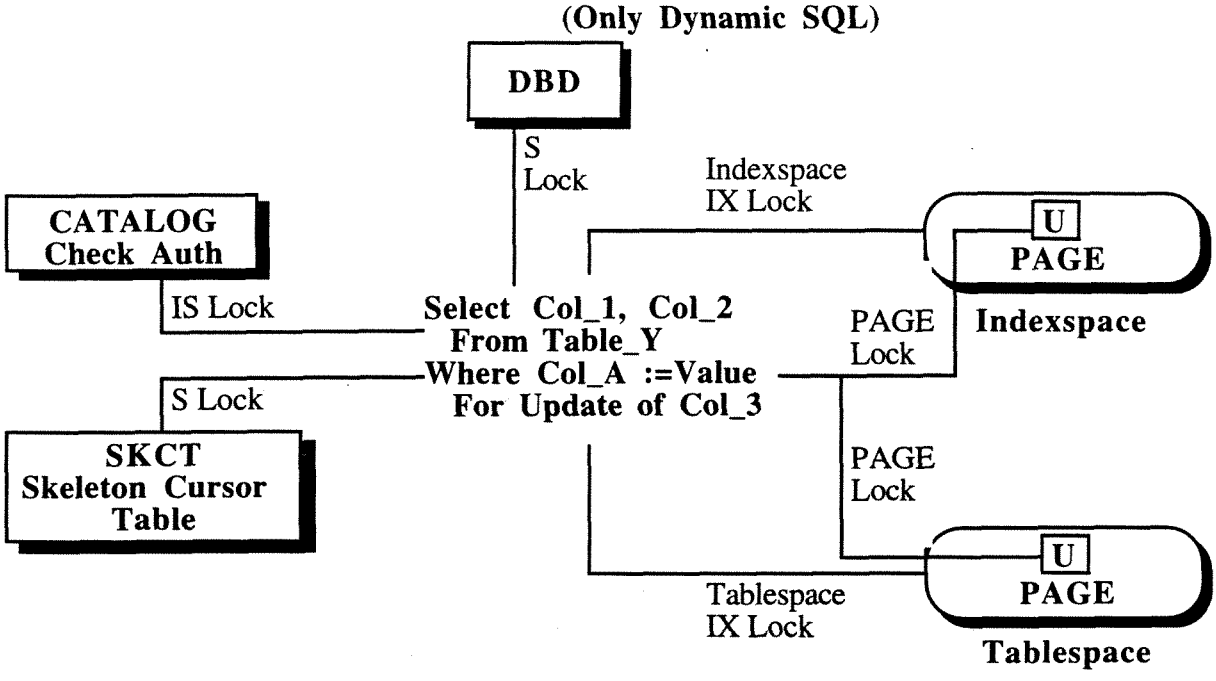
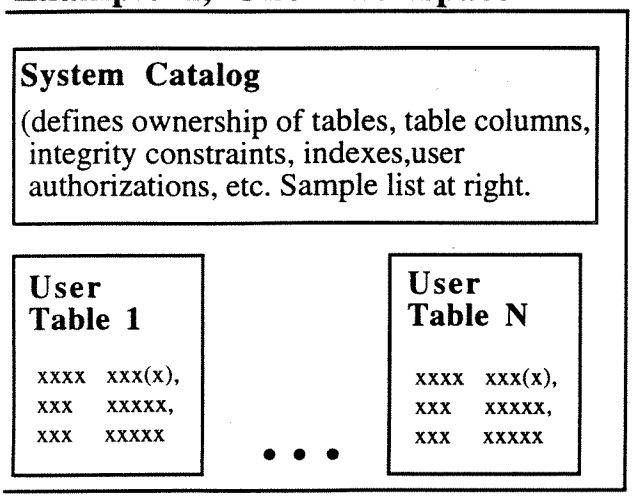


Figure 4.3.2.1. Illustrates the varying types of locks an SQL statement can use. While some locks are beyond control of the DBA, others can be controlled to some degree.

Taken from Bischoff, Joyce and Yevich, Richard, "The DB2 Dilemma: Managing Locking Contention," Database Programming & Design, May 1992, pp. 27-38.

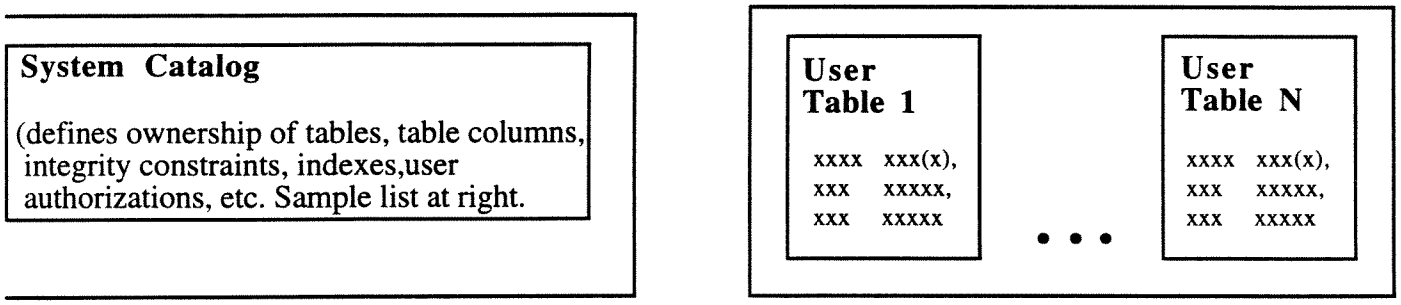
Example 1, One Tablespace



The System Catalog contains 16 files that define a system's authorizations, definitions, usage and views at a given time. They are: SYSACCESS, SYSCATALOG, SYSCHARSETS, SYSCOLAUTH, SYSCOLUMNS, SYSDBSPACES, SYSDROP, SYSINDEXES, SYSOPTIONS, SYSPROGAUTH, SYSSYNONYMS, SYSTABAUTH, SYSUSAGE, SYSUSERAUTH, SYSUSERAUTH, SYSUSERLIST, AND SYSVIEWS.

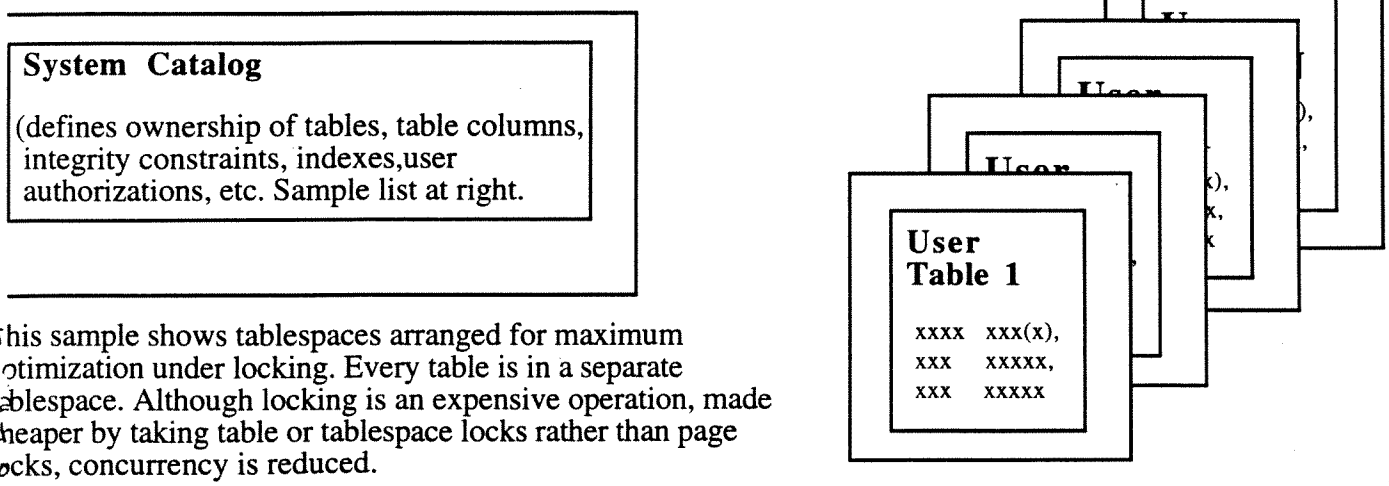
This type of "one tablespace fits all" structure denotes an inexperienced DBA. If a table is locked for data access, the whole system may be locked, through escalating locks.

Example 2, Two Tablespaces



This is the more standard way of specifying tablespaces, with the System Catalog in one tablespace, and user tables in another. Note that placing multiple tables in one tablespace conserves resources and may speed access, but reduces possible concurrency if the tablespace is locked.

Example 3, Multiple Tablespaces



This sample shows tablespaces arranged for maximum optimization under locking. Every table is in a separate tablespace. Although locking is an expensive operation, made cheaper by taking table or tablespace locks rather than page locks, concurrency is reduced.