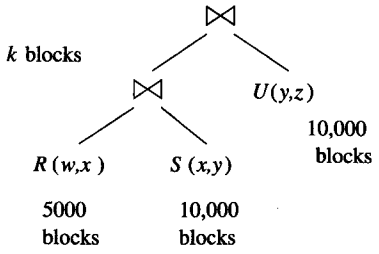


Figure 16.37: Logical query plan and parameters for Example 16.36

A sketch of the expression with key parameters is in Fig. 16.37.

First, consider the join $R \bowtie S$. Neither relation fits in main memory, so we need a two-pass hash-join. If the smaller relation $R$ is partitioned into the maximum-possible 100 buckets on the first pass, then each bucket for $R$ occupies 50 blocks.[10] If $R$'s buckets have 50 blocks, then the second pass of the hash-join $R \bowtie S$ uses 51 buffers, leaving 50 buffers to use for the join of the result of $R \bowtie S$ with $U$.

Now, suppose that $k \leq 49$; that is, the result of $R \bowtie S$ occupies at most 49 blocks. Then we can pipeline the result of $R \bowtie S$ into 49 buffers, organize them for lookup as a hash table, and we have one buffer left to read each block of $U$ in turn. We may thus execute the second join as a one-pass join. The total number of disk I/O's is:

a) 45,000 to perform the two-pass hash join of $R$ and $S$.

b) 10,000 to read $U$ in the one-pass hash-join of $(R \bowtie S) \bowtie U$.

The total is 55,000 disk I/O's.

Now, suppose $k > 49$, but $k \leq 5000$. We can still pipeline the result of $R \bowtie S$, but we need to use another strategy, in which this relation is joined with $U$ in a 50-bucket, two-pass hash-join.

1. Before we start on $R \bowtie S$, we hash $U$ into 50 buckets of 200 blocks each.

2. Next, we perform a two-pass hash join of $R$ and $S$ using 51 buckets as before, but as each tuple of $R \bowtie S$ is generated, we place it in one of the 50 remaining buffers that is used to help form the 50 buckets for the join of $R \bowtie S$ with $U$. These buffers are written to disk when they get full, as is normal for a two-pass hash-join.

3. Finally, we join $R \bowtie S$ with $U$ bucket by bucket. Since $k \leq 5000$, the buckets of $R \bowtie S$ will be of size at most 100 blocks, so this join is feasible. The fact that buckets of $U$ are of size 200 blocks is not a problem, since

[10] We shall assume for convenience that all buckets wind up with exactly their fair share of tuples.

we are using buckets of $R \bowtie S$ as the build relation and buckets of $U$ as the probe relation in the one-pass joins of buckets.

The number of disk I/O's for this pipelined join is:

a) 20,000 to read $U$ and write its tuples into buckets.

b) 45,000 to perform the two-pass hash-join $R \bowtie S$.

c) $k$ to write out the buckets of $R \bowtie S$.

d) $k$ + 10,000 to read the buckets of $R \bowtie S$ and $U$ in the final join.

The total cost is thus 75,000 + $2k$. Note that there is an apparent discontinuity as $k$ grows from 49 to 50, since we had to change the final join from one-pass to two-pass. In practice, the cost would not change so precipitously, since we could use the one-pass join even if there were not enough buffers and a small amount of thrashing occurred.

Last, let us consider what happens when $k > 5000$. Now, we cannot perform a two-pass join in the 50 buffers available if the result of $R \bowtie S$ is pipelined. We could use a three-pass join, but that would require an extra 2 disk I/O's per block of either argument, or 20,000 + $2k$ more disk I/O's. We can do better if we instead decline to pipeline $R \bowtie S$. Now, an outline of the computation of the joins is:

1. Compute $R \bowtie S$ using a two-pass hash join and store the result on disk.

2. Join $R \bowtie S$ with $U$, also using a two-pass hash-join. Note that since $B(U)$ = 10,000, we can perform a two-pass hash-join using 100 buckets, regardless of how large $k$ is. Technically, $U$ should appear as the left argument of its join in Fig. 16.37 if we decide to make $U$ the build relation for the hash join.

The number of disk I/O's for this plan is:

a) 45,000 for the two-pass join of $R$ and $S$.

b) $k$ to store $R \bowtie S$ on disk.

c) 30,000 + $3k$ for the two-pass hash-join of $U$ with $R \bowtie S$.

The total cost is thus 75,000 + $4k$, which is less than the cost of going to a three-pass join at the final step. The three complete plans are summarized in the table of Fig. 16.38. □

| Range of $k$ | Pipeline or Materialize | Algorithm for final join | Total Disk I/O's |
|---|---|---|---|
| $k \leq 49$ | Pipeline | one-pass | 55,000 |
| $50 \leq k \leq 5000$ | Pipeline | 50-bucket, two-pass | $75{,}000 + 2k$ |
| $5000 < k$ | Materialize | 100-bucket, two-pass | $75{,}000 + 4k$ |

Figure 16.38: Costs of physical plans as a function of the size of $R \bowtie S$

### 16.7.6 Notation for Physical Query Plans

We have seen many examples of the operators that can be used to form a physical query plan. In general, each operator of the logical plan becomes one or more operators of the physical plan, and leaves (stored relations) of the logical plan become, in the physical plan, one of the scan operators applied to that relation. In addition, materialization would be indicated by a `Store` operator applied to the intermediate result that is to be materialized, followed by a suitable scan operator (usually `TableScan`, since there is no index on the intermediate relation unless one is constructed explicitly) when the materialized result is accessed by its consumer. However, for simplicity, in our physical-query-plan trees we shall indicate that a certain intermediate relation is materialized by a double line crossing the edge between that relation and its consumer. All other edges are assumed to represent pipelining between the supplier and consumer of tuples.

We shall now catalog the various operators that are typically found in physical query plans. Unlike the relational algebra, whose notation is fairly standard, each DBMS will use its own internal notation for physical query plans.

#### Operators for Leaves

Each relation $R$ that is a leaf operand of the logical-query-plan tree will be replaced by a scan operator. The options are:

1. `TableScan(R)`: All blocks holding tuples of $R$ are read in arbitrary order.
2. `SortScan(R,L)`: Tuples of $R$ are read in order, sorted according to the attribute(s) on list $L$.
3. `IndexScan(R,C)`: Here, $C$ is a condition of the form $A\theta c$, where $A$ is an attribute of $R$, $\theta$ is a comparison such as $=$ or $<$, and $c$ is a constant. Tuples of $R$ are accessed through an index on attribute $A$. If the comparison $\theta$ is not $=$, then the index must be one, such as a B-tree, that supports range queries.
4. `IndexScan(R,A)`: Here $A$ is an attribute of $R$. The entire relation $R$ is retrieved via an index on $R.A$. This operator behaves like `TableScan`,

but may be more efficient if $R$ is not clustered.

### Physical Operators for Selection

A logical operator $\sigma_C(R)$ is often combined, or partially combined, with the access method for relation $R$, when $R$ is a stored relation. Other selections, where the argument is not a stored relation or an appropriate index is not available, will be replaced by the corresponding physical operator we have called `Filter`. Recall the strategy for choosing a selection implementation, which we discussed in Section 16.7.1. The notation we shall use for the various selection implementations are:

1. We may simply replace $\sigma_C(R)$ by the operator `Filter(C)`. This choice makes sense if there is no index on $R$, or no index on an attribute that condition $C$ mentions. If $R$, the argument of the selection, is actually an intermediate relation being pipelined to the selection, then no other operator besides `Filter` is needed. If $R$ is a stored or materialized relation, then we must use an operator, `TableScan` or `SortScan(R,L)`, to access $R$. We prefer sort-scan if the result of $\sigma_C(R)$ will later be passed to an operator that requires its argument sorted.

2. If condition $C$ can be expressed as $A\theta c$ `AND` $D$ for some other condition $D$, and there is an index on $R.A$, then we may:

   (a) Use the operator `IndexScan(R,`$A\theta$`c)` to access $R$, and

   (b) Use `Filter(D)` in place of the selection $\sigma_C(R)$.

### Physical Sort Operators

Sorting of a relation can occur at any point in the physical query plan. We have already introduced the `SortScan(R,L)` operator, which reads a stored relation $R$ and produces it sorted according to the list of attributes $L$. When we apply a sort-based algorithm for operations such as join or grouping, there is an initial phase in which we sort the argument according to some list of attributes. It is common to use an explicit physical operator `Sort(L)` to perform this sort on an operand relation that is not stored. This operator can also be used at the top of the physical-query-plan tree if the result needs to be sorted because of an `ORDER BY` clause in the original query, thus playing the same role as the $\tau$ operator of Section 5.2.6.

### Other Relational-Algebra Operations

All other operations are replaced by a suitable physical operator. These operators can be given designations that indicate:

1. The operation being performed, e.g., join or grouping.

2. Necessary parameters, e.g., the condition in a theta-join or the list of elements in a grouping.

3. A general strategy for the algorithm: sort-based, hash-based, or index-based, e.g.

4. A decision about the number of passes to be used: one-pass, two-pass, or multipass (recursive, using as many passes as necessary for the data at hand). Alternatively, this choice may be left until run-time.

5. An anticipated number of buffers the operation will require.

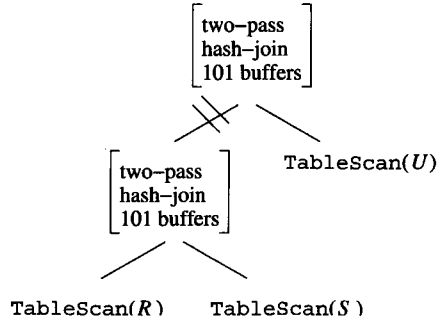


Figure 16.39: A physical plan from Example 16.36

**Example 16.37:** Figure 16.39 shows the physical plan developed in Example 16.36 for the case $k > 5000$. In this plan, we access each of the three relations by a table-scan. We use a two-pass hash-join for the first join, materialize it, and use a two-pass hash-join for the second join. By implication of the double-line symbol for materialization, the left argument of the top join is also obtained by a table-scan, and the result of the first join is stored using the `Store` operator.

In contrast, if $k \leq 49$, then the physical plan developed in Example 16.36 is that shown in Fig. 16.40. Notice that the second join uses a different number of passes, a different number of buffers, and a left argument that is pipelined, not materialized. □

**Example 16.38:** Consider the selection operation in Example 16.35, where we decided that the best of options was to use the index on $y$ to find those tuples with $y = 2$, then check these tuples for the other conditions $x = 1$ and $z < 5$. Figure 16.41 shows the physical query plan. The leaf indicates that $R$ will be accessed through its index on $y$, retrieving only those tuples with $y = 2$. The filter operator says that we complete the selection by further selecting those of the retrieved tuples that have both $x = 1$ and $z < 5$. □

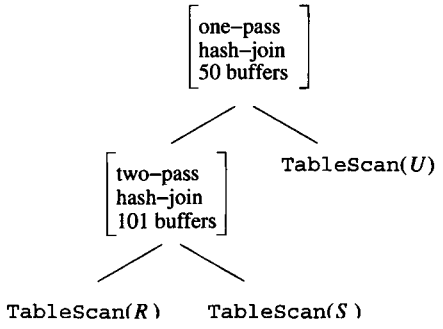


Figure 16.40: Another physical plan for the case where $R \bowtie S$ is expected to be very small

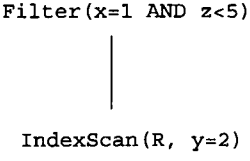


Figure 16.41: Annotating a selection to use the most appropriate index

### 16.7.7 Ordering of Physical Operations

Our final topic regarding physical query plans is the matter of order of operations. The physical query plan is generally represented as a tree, and trees imply something about order of operations, since data must flow up the tree. However, since bushy trees may have interior nodes that are neither ancestors nor descendants of one another, the order of evaluation of interior nodes may not always be clear. Moreover, since iterators can be used to implement operations in a pipelined manner, it is possible that the times of execution for various nodes overlap, and the notion of "ordering" nodes makes no sense.

If materialization is implemented in the obvious store-and-later-retrieve way, and pipelining is implemented by iterators, then we may establish a fixed sequence of events whereby each operation of a physical query plan is executed. The following rules summarize the ordering of events implicit in a physical-query-plan tree:

1. Break the tree into subtrees at each edge that represents materialization. The subtrees will be executed one-at-a-time.

2. Order the execution of the subtrees in a bottom-up, left-to-right manner. To be precise, perform a preorder traversal of the entire tree. Order the subtrees in the order in which the preorder traversal exits from the subtrees.

3. Execute all nodes of each subtree using a network of iterators. Thus, all the nodes in one subtree are executed simultaneously, with `GetNext` calls among their operators determining the exact order of events.

Following this strategy, the query optimizer can now generate executable code, perhaps a sequence of function calls, for the query.

### 16.7.8 Exercises for Section 16.7

**Exercise 16.7.1:** Consider a relation $R(a, b, c, d)$ that has a clustering index on $a$ and nonclustering indexes on each of the other attributes. The relevant parameters are: $B(R) = 1000$, $T(R) = 5000$, $V(R, a) = 20$, $V(R, b) = 1000$, $V(R, c) = 5000$, and $V(R, d) = 500$. Give the best query plan (index-scan or table-scan followed by a filter step) and the disk-I/O cost for each of the following selections:

a) $\sigma_{a=1 \text{ AND } b=2 \text{ AND } d=3}(R)$.

b) $\sigma_{a=1 \text{ AND } b=2 \text{ AND } c\geq 3}(R)$.

c) $\sigma_{a=1 \text{ AND } b\leq 2 \text{ AND } c\geq 3}(R)$.

! **Exercise 16.7.2:** In terms of $B(R)$, $T(R)$, $V(R, x)$, and $V(R, y)$, express the following conditions about the cost of implementing a selection on $R$:

a) It is better to use index-scan with a nonclustering index on $x$ and a term that equates $x$ to a constant than a nonclustering index on $y$ and a term that equates $y$ to a constant.

b) It is better to use index-scan with a nonclustering index on $x$ and a term that equates $x$ to a constant than a clustering index on $y$ and a term that equates $y$ to a constant.

c) It is better to use index-scan with a nonclustering index on $x$ and a term that equates $x$ to a constant than a clustering index on $y$ and a term of the form $y > C$ for some constant $C$.

**Exercise 16.7.3:** How would the conclusions about when to pipeline in Example 16.36 change if the size of relation $R$ were not 5000 blocks, but: (a) 2000 blocks ! (b) 10,000 blocks ! (c) 100 blocks?

! **Exercise 16.7.4:** Suppose we want to compute $\big(R(a, b) \bowtie S(a, c)\big) \bowtie T(a, d)$ in the order indicated. We have $M = 101$ main-memory buffers, and $B(R) = B(S) = 2000$. Because the join attribute $a$ is the same for both joins, we decide to implement the first join $R \bowtie S$ by a two-pass sort-join, and we shall use the appropriate number of passes for the second join, first dividing $T$ into some number of sublists sorted on $a$, and merging them with the sorted and pipelined stream of tuples from the join $R \bowtie S$. For what values of $B(T)$ should we choose for the join of $T$ with $R \bowtie S$:

a) A one-pass join; i.e., we read $T$ into memory, and compare its tuples with the tuples of $R \bowtie S$ as they are generated.

b) A two-pass join; i.e., we create sorted sublists for $T$ and keep one buffer in memory for each sorted sublist, while we generate tuples of $R \bowtie S$.

## 16.8 Summary of Chapter 16

- *Compilation of Queries*: Compilation turns a query into a physical query plan, which is a sequence of operations that can be implemented by the query-execution engine. The principal steps of query compilation are parsing, semantic checking, selection of the preferred logical query plan (algebraic expression), and generation from that of the best physical plan.

- *The Parser*: The first step in processing a SQL query is to parse it, as one would for code in any programming language. The result of parsing is a parse tree with nodes corresponding to SQL constructs.

- *View Expansion*: Queries that refer to virtual views must have these references in the parse tree replaced by the tree for the expression that defines the view. This expansion often introduces several opportunities to optimize the complete query.

- *Semantic Checking*: A preprocessor examines the parse tree, checks that the attributes, relation names, and types make sense, and resolves attribute references.

- *Conversion to a Logical Query Plan*: The query processor must convert the semantically checked parse tree to an algebraic expression. Much of the conversion to relational algebra is straightforward, but subqueries present a problem. One approach is to introduce a two-argument selection that puts the subquery in the condition of the selection, and then apply appropriate transformations for the common special cases.

- *Algebraic Transformations*: There are many ways that a logical query plan can be transformed to a better plan by using algebraic transformations. Section 16.2 enumerates the principal ones.

- *Choosing a Logical Query Plan*: The query processor must select that query plan that is most likely to lead to an efficient physical plan. In addition to applying algebraic transformations, it is useful to group associative and commutative operators, especially joins, so the physical query plan can choose the best order and grouping for these operations.

- *Estimating Sizes of Relations*: When selecting the best logical plan, or when ordering joins or other associative-commutative operations, we use the estimated size of intermediate relations as a surrogate for the true

running time. Knowing, or estimating, both the size (number of tuples) of relations and the number of distinct values for each attribute of each relation helps us get good estimates of the sizes of intermediate relations.

- *Histograms*: Some systems keep histograms of the values for a given attribute. This information can be used to obtain better estimates of intermediate-relation sizes than the simple methods stressed here.

- *Cost-Based Optimization*: When selecting the best physical plan, we need to estimate the cost of each possible plan. Various strategies are used to generate all or some of the possible physical plans that implement a given logical plan.

- *Plan-Enumeration Strategies*: The common approaches to searching the space of physical plans for the best include dynamic programming (tabularizing the best plan for each subexpression of the given logical plan), Selinger-style dynamic programming (which includes the sort-order of results as part of the table, giving best plans for each sort-order and for an unsorted result), greedy approaches (making a series of locally optimal decisions, given the choices for the physical plan that have been made so far), and branch-and-bound (enumerating only plans that are not immediately known to be worse than the best plan found so far).

- *Left-Deep Join Trees*: When picking a grouping and order for the join of several relations, it is common to restrict the search to left-deep trees, which are binary trees with a single spine down the left edge, with only leaves as right children. This form of join expression tends to yield efficient plans and also limits significantly the number of physical plans that need to be considered.

- *Physical Plans for Selection*: If possible, a selection should be broken into an index-scan of the relation to which the selection is applied (typically using a condition in which the indexed attribute is equated to a constant), followed by a filter operation. The filter examines the tuples retrieved by the index-scan and passes through only those that meet the portions of the selection condition other than that on which the index scan is based.

- *Pipelining Versus Materialization*: Ideally, the result of each physical operator is consumed by another operator, with the result being passed between the two in main memory ("pipelining"), perhaps using an iterator to control the flow of data from one to the other. However, sometimes there is an advantage to storing ("materializing") the result of one operator to save space in main memory for other operators. Thus, the physical-query-plan generator should consider both pipelining and materialization of intermediates.

## 16.9 References for Chapter 16

The surveys mentioned in the bibliographic notes to Chapter 15 also contain material relevant to query compilation. In addition, we recommend the survey [1], which contains material on the query optimizers of commercial systems.

Three of the earliest studies of query optimization are [4], [5], and [3]. Paper [6], another early study, incorporates the idea of pushing selections down the tree with the greedy algorithm for join-order choice. [2] is the source for "Selinger-style optimization" as well as describing the System R optimizer, which was one of the most ambitious attempts at query optimization of its day.

1. G. Graefe (ed.), *Data Engineering* **16**:4 (1993), special issue on query processing in commercial database management systems, IEEE.

2. P. Griffiths-Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database system," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1979), pp. 23–34.

3. P. A. V. Hall, "Optimization of a single relational expression in a relational database system," *IBM J. Research and Development* **20**:3 (1976), pp. 244–257.

4. F. P. Palermo, "A database search problem," in: J. T. Tou (ed.) *Information Systems COINS IV*, Plenum, New York, 1974.

5. J. M. Smith and P. Y. Chang, "Optimizing the performance of a relational algebra database interface," *Comm. ACM* **18**:10 (1975), pp. 568–579.

6. E. Wong and K. Youssefi, "Decomposition — a strategy for query processing," *ACM Trans. on Database Systems* **1**:3 (1976), pp. 223–241.

# Chapter 17

# Coping With System Failures

Starting with this chapter, we focus our attention on those parts of a DBMS that control access to data. There are two major issues to address:

1. Data must be protected in the face of a system failure. This chapter deals with techniques for supporting the goal of *resilience*, that is, integrity of the data when the system fails in some way.
2. Data must not be corrupted simply because several error-free queries or database modifications are being done at once. This matter is addressed in Chapters 18 and 19.

The principal technique for supporting resilience is a *log*, which records securely the history of database changes. We shall discuss three different styles of logging, called "undo," "redo," and "undo/redo." We also discuss *recovery*, the process whereby the log is used to reconstruct what has happened to the database when there has been a failure. An important aspect of logging and recovery is avoidance of the situation where the log must be examined into the distant past. Thus, we shall learn about "checkpointing," which limits the length of log that must be examined during recovery.

In a final section, we discuss "archiving," which allows the database to survive not only temporary system failures, but situations where the entire database is lost. Then, we must rely on a recent copy of the database (the archive) plus whatever log information survives, to reconstruct the database as it existed at some point in the recent past.

## 17.1 Issues and Models for Resilient Operation

We begin our discussion of coping with failures by reviewing the kinds of things that can go wrong, and what a DBMS can and should do about them. We

initially focus on "system failures" or "crashes," the kinds of errors that the logging and recovery methods are designed to fix. We also introduce in Section 17.1.4 the model for buffer management that underlies all discussions of recovery from system errors. The same model is needed in the next chapter as we discuss concurrent access to the database by several transactions.

### 17.1.1 Failure Modes

There are many things that can go wrong as a database is queried and modified. Problems range from the keyboard entry of incorrect data to an explosion in the room where the database is stored on disk. The following items are a catalog of the most important failure modes and what the DBMS can do about them.

#### Erroneous Data Entry

Some data errors are impossible to detect. For example, if a clerk mistypes one digit of your phone number, the data will still look like a phone number that *could* be yours. On the other hand, if the clerk omits a digit from your phone number, then the data is evidently in error, since it does not have the form of a phone number. The principal technique for addressing data-entry errors is to write constraints and triggers that detect data believed to be erroneous.

#### Media Failures

A local failure of a disk, one that changes only a bit or a few bits, can normally be detected by parity checks associated with the sectors of the disk, as we discussed in Section 13.4.2. Head crashes, where the entire disk becomes unreadable, are generally handled by one or both of the following approaches:

1. Use one of the RAID schemes discussed in Section 13.4, so the lost disk can be restored.

2. Maintain an *archive*, a copy of the database on a medium such as tape or optical disk. The archive is periodically created, either fully or incrementally, and stored at a safe distance from the database itself. We shall discuss archiving in Section 17.5.

3. Instead of an archive, one could keep redundant copies of the database on-line, distributed among several sites. These copies are kept consistent by mechanisms we shall discuss in Section 20.6.

#### Catastrophic Failure

In this category are a number of situations in which the media holding the database is completely destroyed. Examples include explosions, fires, or vandalism at the site of the database. RAID will not help, since all the data disks and their parity check disks become useless simultaneously. However, the other

approaches that can be used to protect against media failure — archiving and redundant, distributed copies — will also protect against a catastrophic failure.

### System Failures

The processes that query and modify the database are called *transactions*. A transaction, like any program, executes a number of steps in sequence; often, several of these steps will modify the database. Each transaction has a *state*, which represents what has happened so far in the transaction. The state includes the current place in the transaction's code being executed and the values of any local variables of the transaction that will be needed later on.

*System failures* are problems that cause the state of a transaction to be lost. Typical system failures are power loss and software errors. Since main memory is "volatile," as we discussed in Section 13.1.3, a power failure will cause the contents of main memory to disappear, along with the result of any transaction step that was kept only in main memory, rather than on (nonvolatile) disk. Similarly, a software error may overwrite part of main memory, possibly including values that were part of the state of the program.

When main memory is lost, the transaction state is lost; that is, we can no longer tell what parts of the transaction, including its database modifications, were made. Running the transaction again may not fix the problem. For example, if the transaction must add 1 to a value in the database, we do not know whether to repeat the addition of 1 or not. The principal remedy for the problems that arise due to a system error is logging of all database changes in a separate, nonvolatile log, coupled with recovery when necessary. However, the mechanisms whereby such logging can be done in a fail-safe manner are surprisingly intricate, as we shall see starting in Section 17.2.

## 17.1.2 More About Transactions

We introduced the idea of transactions from the point of view of the SQL programmer in Section 6.6. Before proceeding to our study of database resilience and recovery from failures, we need to discuss the fundamental notion of a transaction in more detail.

The transaction is the unit of execution of database operations. For example, if we are issuing ad-hoc commands to a SQL system, then each query or database modification statement (plus any resulting trigger actions) is a transaction. When using an embedded SQL interface, the programmer controls the extent of a transaction, which may include several queries or modifications, as well as operations performed in the host language. In the typical embedded SQL system, transactions begin as soon as operations on the database are executed and end with an explicit `COMMIT` or `ROLLBACK` ("abort") command.

As we shall discuss in Section 17.1.3, a transaction must execute atomically, that is, all-or-nothing and as if it were executed at an instant in time. Assuring

that transactions are executed correctly is the job of a *transaction manager*, a subsystem that performs several functions, including:

1. Issuing signals to the log manager (described below) so that necessary information in the form of "log records" can be stored on the log.
2. Assuring that concurrently executing transactions do not interfere with each other in ways that introduce errors ("scheduling"; see Section 18.1).

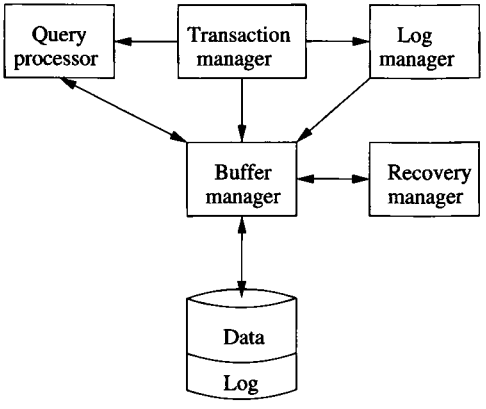


Figure 17.1: The log manager and transaction manager

The transaction manager and its interactions are suggested by Fig. 17.1. The transaction manager will send messages about actions of transactions to the log manager, to the buffer manager about when it is possible or necessary to copy the buffer back to disk, and to the query processor to execute the queries and other database operations that comprise the transaction.

The log manager maintains the log. It must deal with the buffer manager, since space for the log initially appears in main-memory buffers, and at certain times these buffers must be copied to disk. The log, as well as the data, occupies space on the disk, as we suggest in Fig. 17.1.

Finally, we show a recovery manager in Fig. 17.1. When there is a crash, the recovery manager is activated. It examines the log and uses it to repair the data, if necessary. As always, access to the disk is through the buffer manager.

### 17.1.3 Correct Execution of Transactions

Before we can deal with correcting system errors, we need to understand what it means for a transaction to be executed "correctly." To begin, we assume that the database is composed of "elements." We shall not specify precisely what an "element" is, except to say it has a value and can be accessed or modified by transactions. Different database systems use different notions of elements, but they are usually chosen from one or more of the following:

1. Relations.

2. Disk blocks or pages.

3. Individual tuples or objects.

In examples to follow, one can imagine that database elements are tuples, or in many examples, simply integers. However, there are several good reasons in practice to use choice (2) — disk blocks or pages — as the database element. In this way, buffer-contents become single elements, allowing us to avoid some serious problems with logging and transactions that we shall explore periodically as we learn various techniques. Avoiding database elements that are bigger than disk blocks also prevents a situation where part but not all of an element has been placed in nonvolatile storage when a crash occurs.

A database has a *state*, which is a value for each of its elements.[1] Intuitively, we regard certain states as *consistent*, and others as inconsistent. Consistent states satisfy all constraints of the database schema, such as key constraints and constraints on values. However, consistent states must also satisfy implicit constraints that are in the mind of the database designer. The implicit constraints may be maintained by triggers that are part of the database schema, but they might also be maintained only by policy statements concerning the database, or warnings associated with the user interface through which updates are made.

A fundamental assumption about transactions is:

- *The Correctness Principle*: If a transaction executes in the absence of any other transactions or system errors, and it starts with the database in a consistent state, then the database is also in a consistent state when the transaction ends.

There is a converse to the correctness principle that forms the motivation for both the logging techniques discussed in this chapter and the concurrency control mechanisms discussed in Chapter 18. This converse involves two points:

1. A transaction is *atomic*; that is, it must be executed as a whole or not at all. If only part of a transaction executes, then the resulting database state may not be consistent.

2. Transactions that execute simultaneously are likely to lead to an inconsistent state unless we take steps to control their interactions, as we shall in Chapter 18.

[1] We should not confuse the database state with the state of a transaction; the latter is values for the transaction's local variables, not database elements.

### Is the Correctness Principle Believable?

Given that a database transaction could be an ad-hoc modification command issued at a terminal, perhaps by someone who doesn't understand the implicit constraints in the mind of the database designer, is it plausible to assume all transactions take the database from a consistent state to another consistent state? Explicit constraints are enforced by the database, so any transaction that violates them will be rejected by the system and not change the database at all. As for implicit constraints, one cannot characterize them exactly under any circumstances. Our position, justifying the correctness principle, is that if someone is given authority to modify the database, then they also have the authority to judge what the implicit constraints are.

## 17.1.4 The Primitive Operations of Transactions

Let us now consider in detail how transactions interact with the database. There are three address spaces that interact in important ways:

1. The space of disk blocks holding the database elements.
2. The virtual or main memory address space that is managed by the buffer manager.
3. The local address space of the transaction.

For a transaction to read a database element, that element must first be brought to a main-memory buffer or buffers, if it is not already there. Then, the contents of the buffer(s) can be read by the transaction into its own address space. Writing a new value for a database element by a transaction follows the reverse route. The new value is first created by the transaction in its own space. Then, this value is copied to the appropriate buffer(s).

The buffer may or may not be copied to disk immediately; that decision is the responsibility of the buffer manager in general. As we shall soon see, one of the principal tools for assuring resilience is forcing the buffer manager to write the block in a buffer back to disk at appropriate times. However, in order to reduce the number of disk I/O's, database systems can and will allow a change to exist only in volatile main-memory storage, at least for certain periods of time and under the proper set of conditions.

In order to study the details of logging algorithms and other transaction-management algorithms, we need a notation that describes all the operations that move data between address spaces. The primitives we shall use are:

1. `INPUT(X)`: Copy the disk block containing database element $X$ to a memory buffer.

### Buffers in Query Processing and in Transactions

If you got used to the analysis of buffer utilization in the chapters on query processing, you may notice a change in viewpoint here. In Chapters 15 and 16 we were interested in buffers principally as they were used to compute temporary relations during the evaluation of a query. That is one important use of buffers, but there is never a need to preserve a temporary value, so these buffers do not generally have their values logged. On the other hand, those buffers that hold data retrieved from the database *do* need to have those values preserved, especially when the transaction updates them.

2. `READ(X,t)`: Copy the database element $X$ to the transaction's local variable $t$. More precisely, if the block containing database element $X$ is not in a memory buffer then first execute `INPUT(X)`. Next, assign the value of $X$ to local variable $t$.

3. `WRITE(X,t)`: Copy the value of local variable $t$ to database element $X$ in a memory buffer. More precisely, if the block containing database element $X$ is not in a memory buffer then execute `INPUT(X)`. Next, copy the value of $t$ to $X$ in the buffer.

4. `OUTPUT(X)`: Copy the block containing $X$ from its buffer to disk.

The above operations make sense as long as database elements reside within a single disk block, and therefore within a single buffer. If a database element occupies several blocks, we shall imagine that each block-sized portion of the element is an element by itself. The logging mechanism to be used will assure that the transaction cannot complete without the write of $X$ being atomic; i.e., either all blocks of $X$ are written to disk, or none are. Thus, we shall assume for the entire discussion of logging that

- A database element is no larger than a single block.

Different DBMS components issue the various commands we just introduced. `READ` and `WRITE` are issued by transactions. `INPUT` and `OUTPUT` are normally issued by the buffer manager. `OUTPUT` can also be initiated by the log manager under certain conditions, as we shall see.

**Example 17.1:** To see how the above primitive operations relate to what a transaction might do, let us consider a database that has two elements, $A$ and $B$, with the constraint that they must be equal in all consistent states.[2]

Transaction $T$ consists logically of the following two steps:

[2]One reasonably might ask why we should bother to have two different elements that are constrained to be equal, rather than maintaining only one element. However, this simple

```
A := A*2;
B := B*2;
```

If $T$ starts in a consistent state (i.e., $A = B$) and completes its activities without interference from another transaction or system error, then the final state must also be consistent. That is, $T$ doubles two equal elements to get new, equal elements.

Execution of $T$ involves reading $A$ and $B$ from disk, performing arithmetic in the local address space of $T$, and writing the new values of $A$ and $B$ to their buffers. The relevant steps of $T$ are thus:

```
READ(A,t); t := t*2; WRITE(A,t); READ(B,t); t := t*2; WRITE(B,t);
```

In addition, the buffer manager will eventually execute the `OUTPUT` steps to write these buffers back to disk. Figure 17.2 shows the primitive steps of $T$, followed by the two `OUTPUT` commands from the buffer manager. We assume that initially $A = B = 8$. The values of the memory and disk copies of $A$ and $B$ and the local variable $t$ in the address space of transaction $T$ are indicated for each step.

| Action | $t$ | Mem $A$ | Mem $B$ | Disk $A$ | Disk $B$ |
|---|---|---|---|---|---|
| `READ(A,t)` | 8 | 8 | | 8 | 8 |
| `t := t*2` | 16 | 8 | | 8 | 8 |
| `WRITE(A,t)` | 16 | 16 | | 8 | 8 |
| `READ(B,t)` | 8 | 16 | 8 | 8 | 8 |
| `t := t*2` | 16 | 16 | 8 | 8 | 8 |
| `WRITE(B,t)` | 16 | 16 | 16 | 8 | 8 |
| `OUTPUT(A)` | 16 | 16 | 16 | 16 | 8 |
| `OUTPUT(B)` | 16 | 16 | 16 | 16 | 16 |

Figure 17.2: Steps of a transaction and its effect on memory and disk

At the first step, $T$ reads $A$, which generates an `INPUT(A)` command for the buffer manager if $A$'s block is not already in a buffer. The value of $A$ is also copied by the `READ` command into local variable $t$ of $T$'s address space. The second step doubles $t$; it has no affect on $A$, either in a buffer or on disk. The third step writes $t$ into $A$ of the buffer; it does not affect $A$ on disk. The next three steps do the same for $B$, and the last two steps copy $A$ and $B$ to disk.

Observe that as long as all these steps execute, consistency of the database is preserved. If a system error occurs before `OUTPUT(A)` is executed, then there is no effect to the database stored on disk; it is as if $T$ never ran, and consistency is preserved. However, if there is a system error after `OUTPUT(A)` but before

numerical constraint captures the spirit of many more realistic constraints, e.g., the number of seats sold on a flight must not exceed the number of seats on the plane by more than 10%, or the sum of the loan balances at a bank must equal the total debt of the bank.

OUTPUT(B), then the database is left in an inconsistent state. We cannot prevent this situation from ever occurring, but we can arrange that when it does occur, the problem can be repaired — either both $A$ and $B$ will be reset to 8, or both will be advanced to 16. □

### 17.1.5 Exercises for Section 17.1

**Exercise 17.1.1:** Suppose that the consistency constraint on the database is $0 \leq A \leq B$. Tell whether each of the following transactions preserves consistency.

a) `A := A+B; B := A+B;`

b) `B := A+B; A := A+B;`

c) `A := B+1; B := A+1;`

**Exercise 17.1.2:** For each of the transactions of Exercise 17.1.1, add the read- and write-actions to the computation and show the effect of the steps on main memory and disk. Assume that initially $A = 5$ and $B = 10$. Also, tell whether it is possible, with the appropriate order of OUTPUT actions, to assure that consistency is preserved even if there is a crash while the transaction is executing.

## 17.2 Undo Logging

A *log* is a file of *log records*, each telling something about what some transaction has done. If log records appear in nonvolatile storage, we can use them to restore the database to a consistent state after a system crash. Our first style of logging — *undo logging* — makes repairs to the database state by undoing the effects of transactions that may not have completed before the crash.

Additionally, in this section we introduce the basic idea of log records, including the *commit* (successful completion of a transaction) action and its effect on the database state and log. We shall also consider how the log itself is created in main memory and copied to disk by a "flush-log" operation. Finally, we examine the undo log specifically, and learn how to use it in recovery from a crash. In order to avoid having to examine the entire log during recovery, we introduce the idea of "checkpointing," which allows old portions of the log to be thrown away.

### 17.2.1 Log Records

Imagine the log as a file opened for appending only. As transactions execute, the *log manager* has the job of recording in the log each important event. One block of the log at a time is filled with log records, each representing one of these events. Log blocks are initially created in main memory and are allocated

### Why Might a Transaction Abort?

One might wonder why a transaction would abort rather than commit. There are actually several reasons. The simplest is when there is some error condition in the code of the transaction itself, e.g., an attempted division by zero. The DBMS may also abort a transaction for one of several reasons. For instance, a transaction may be involved in a deadlock, where it and one or more other transactions each hold some resource that the other needs. Then, one or more transactions must be forced by the system to abort (see Section 19.2).

by the buffer manager like any other blocks that the DBMS needs. The log blocks are written to nonvolatile storage on disk as soon as is feasible; we shall have more to say about this matter in Section 17.2.2.

There are several forms of log record that are used with each of the types of logging we discuss in this chapter. These are:

1. <START $T$>: This record indicates that transaction $T$ has begun.

2. <COMMIT $T$>: Transaction $T$ has completed successfully and will make no more changes to database elements. Any changes to the database made by $T$ should appear on disk. However, because we cannot control when the buffer manager chooses to copy blocks from memory to disk, we cannot in general be sure that the changes are already on disk when we see the <COMMIT $T$> log record. If we insist that the changes already be on disk, this requirement must be enforced by the log manager (as is the case for undo logging).

3. <ABORT $T$>: Transaction $T$ could not complete successfully. If transaction $T$ aborts, no changes it made can have been copied to disk, and it is the job of the transaction manager to make sure that such changes never appear on disk, or that their effect on disk is cancelled if they do. We shall discuss the matter of repairing the effect of aborted transactions in Section 19.1.1.

For an undo log, the only other kind of log record we need is an *update record*, which is a triple $<T, X, v>$. The meaning of this record is: transaction $T$ has changed database element $X$, and its former value was $v$. The change reflected by an update record normally occurs in memory, not disk; i.e., the log record is a response to a WRITE action into memory, not an OUTPUT action to disk. Notice also that an undo log does not record the new value of a database element, only the old value. As we shall see, should recovery be necessary in a system using undo logging, the only thing the recovery manager will do is cancel the possible effect of a transaction on disk by restoring the old value.

### Preview of Other Logging Methods

In "redo logging" (Section 17.3), on recovery we redo any transaction that has a COMMIT record, and we ignore all others. Rules for redo logging assure that we may ignore transactions whose COMMIT records never reached the log on disk. "Undo/redo logging" (Section 17.4) will, on recovery, undo any transaction that has not committed, and will redo those transactions that have committed. Again, log-management and buffering rules will assure that these steps successfully repair any damage to the database.

## 17.2.2 The Undo-Logging Rules

An undo log is sufficient to allow recovery from a system failure, provided transactions and the buffer manager obey two rules:

$U_1$: If transaction $T$ modifies database element $X$, then the log record of the form $<T, X, v>$ must be written to disk *before* the new value of $X$ is written to disk.

$U_2$: If a transaction commits, then its COMMIT log record must be written to disk only *after* all database elements changed by the transaction have been written to disk, but as soon thereafter as possible.

To summarize rules $U_1$ and $U_2$, material associated with one transaction must be written to disk in the following order:

a) The log records indicating changed database elements.

b) The changed database elements themselves.

c) The COMMIT log record.

However, the order of (a) and (b) applies to each database element individually, not to the group of update records for a transaction as a whole.

In order to force log records to disk, the log manager needs a *flush-log* command that tells the buffer manager to copy to disk any log blocks that have not previously been copied to disk or that have been changed since they were last copied. In sequences of actions, we shall show FLUSH LOG explicitly. The transaction manager also needs to have a way to tell the buffer manager to perform an OUTPUT action on a database element. We shall continue to show the OUTPUT action in sequences of transaction steps.

**Example 17.2:** Let us reconsider the transaction of Example 17.1 in the light of undo logging. Figure 17.3 expands on Fig. 17.2 to show the log entries and flush-log actions that have to take place along with the actions of the transaction

| Step | Action | $t$ | M-$A$ | M-$B$ | D-$A$ | D-$B$ | Log |
|---|---|---|---|---|---|---|---|
| 1) | | | | | | | <START $T$> |
| 2) | READ(A,t) | 8 | 8 | | 8 | 8 | |
| 3) | t := t*2 | 16 | 8 | | 8 | 8 | |
| 4) | WRITE(A,t) | 16 | 16 | | 8 | 8 | <$T, A, 8$> |
| 5) | READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | t := t*2 | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <$T, B, 8$> |
| 8) | FLUSH LOG | | | | | | |
| 9) | OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| 10) | OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |
| 11) | | | | | | | <COMMIT $T$> |
| 12) | FLUSH LOG | | | | | | |

Figure 17.3: Actions and their log entries

$T$. Note we have shortened the headers to M-$A$ for "the copy of $A$ in a memory buffer" or D-$B$ for "the copy of $B$ on disk," and so on.

In line (1) of Fig. 17.3, transaction $T$ begins. The first thing that happens is that the <START $T$> record is written to the log. Line (2) represents the read of $A$ by $T$. Line (3) is the local change to $t$, which affects neither the database stored on disk nor any portion of the database in a memory buffer. Neither lines (2) nor (3) require any log entry, since they have no affect on the database.

Line (4) is the write of the new value of $A$ to the buffer. This modification to $A$ is reflected by the log entry <$T, A, 8$> which says that $A$ was changed by $T$ and its former value was 8. Note that the new value, 16, is not mentioned in an undo log.

Lines (5) through (7) perform the same three steps with $B$ instead of $A$. At this point, $T$ has completed and must commit. The changed $A$ and $B$ must migrate to disk, but in order to follow the two rules for undo logging, there is a fixed sequence of events that must happen.

First, $A$ and $B$ cannot be copied to disk until the log records for the changes are on disk. Thus, at step (8) the log is flushed, assuring that these records appear on disk. Then, steps (9) and (10) copy $A$ and $B$ to disk. The transaction manager requests these steps from the buffer manager in order to commit $T$.

Now, it is possible to commit $T$, and the <COMMIT $T$> record is written to the log, which is step (11). Finally, we must flush the log again at step (12) to make sure that the <COMMIT $T$> record of the log appears on disk. Notice that without writing this record to disk, we could have a situation where a transaction has committed, but for a long time a review of the log does not tell us that it has committed. That situation could cause strange behavior if there were a crash, because, as we shall see in Section 17.2.3, a transaction that appeared to the user to have completed long ago would then be undone and effectively aborted. □

### Background Activity Affects the Log and Buffers

As we look at a sequence of actions and log entries like Fig. 17.3, it is tempting to imagine that these actions occur in isolation. However, the DBMS may be processing many transactions simultaneously. Thus, the four log records for transaction $T$ may be interleaved on the log with records for other transactions. Moreover, if one of these transactions flushes the log, then the log records from $T$ may appear on disk earlier than is implied by the flush-log actions of Fig. 17.3. There is no harm if log records reflecting a database modification appear earlier than necessary. The essential policy for undo logging is that we don't write the <COMMIT $T$> record until the OUTPUT actions for $T$ are completed.

A trickier situation occurs if two database elements $A$ and $B$ share a block. Then, writing one of them to disk writes the other as well. In the worst case, we can violate rule $U_1$ by writing one of these elements prematurely. It may be necessary to adopt additional constraints on transactions in order to make undo logging work. For instance, we might use a locking scheme where database elements are disk blocks, as described in Section 18.3, to prevent two transactions from accessing the same block at the same time. This and other problems that appear when database elements are fractions of a block motivate our suggestion that blocks *be* the database elements.

## 17.2.3 Recovery Using Undo Logging

Suppose now that a system failure occurs. It is possible that certain database changes made by a given transaction were written to disk, while other changes made by the same transaction never reached the disk. If so, the transaction was not executed atomically, and there may be an inconsistent database state. The *recovery manager* must use the log to restore the database to some consistent state.

In this section we consider only the simplest form of recovery manager, one that looks at the entire log, no matter how long, and makes database changes as a result of its examination. In Section 17.2.4 we consider a more sensible approach, where the log is periodically "checkpointed," to limit the distance back in history that the recovery manager must go.

The first task of the recovery manager is to divide the transactions into committed and uncommitted transactions. If there is a log record <COMMIT $T$>, then by undo rule $U_2$ all changes made by transaction $T$ were previously written to disk. Thus, $T$ by itself could not have left the database in an inconsistent state when the system failure occurred.

However, suppose that we find a <START $T$> record on the log but no <COMMIT $T$> record. Then there could have been some changes to the database

made by $T$ that were written to disk before the crash, while other changes by $T$ either were not made, or were made in the main-memory buffers but not copied to disk. In this case, $T$ is an *incomplete transaction* and must be *undone*. That is, whatever changes $T$ made must be reset to their previous value. Fortunately, rule $U_1$ assures us that if $T$ changed $X$ on disk before the crash, then there will be a $<T, X, v>$ record on the log, and that record will have been copied to disk before the crash. Thus, during the recovery, we must write the value $v$ for database element $X$. Note that this rule begs the question whether $X$ had value $v$ in the database anyway; we don't even bother to check.

Since there may be several uncommitted transactions in the log, and there may even be several uncommitted transactions that modified $X$, we have to be systematic about the order in which we restore values. Thus, the recovery manager must scan the log from the end (i.e., from the most recently written record to the earliest written). As it travels, it remembers all those transactions $T$ for which it has seen a <COMMIT $T$> record or an <ABORT $T$> record. Also as it travels backward, if it sees a record $<T, X, v>$, then:

1. If $T$ is a transaction whose COMMIT record has been seen, then do nothing. $T$ is committed and must not be undone.

2. Otherwise, $T$ is an incomplete transaction, or an aborted transaction. The recovery manager must change the value of $X$ in the database to $v$, in case $X$ had been altered just before the crash.

After making these changes, the recovery manager must write a log record <ABORT $T$> for each incomplete transaction $T$ that was not previously aborted, and then flush the log. Now, normal operation of the database may resume, and new transactions may begin executing.

**Example 17.3:** Let us consider the sequence of actions from Fig. 17.3 and Example 17.2. There are several different times that the system crash could have occurred; let us consider each significantly different one.

1. The crash occurs after step (12). Then the <COMMIT $T$> record reached disk before the crash. When we recover, we do not undo the results of $T$, and all log records concerning $T$ are ignored by the recovery manager.

2. The crash occurs between steps (11) and (12). It is possible that the log record containing the COMMIT got flushed to disk; for instance, the buffer manager may have needed the buffer containing the end of the log for another transaction, or some other transaction may have asked for a log flush. If so, then the recovery is the same as in case (1) as far as $T$ is concerned. However, if the COMMIT record never reached disk, then the recovery manager considers $T$ incomplete. When it scans the log backward, it comes first to the record $<T, B, 8>$. It therefore stores 8 as the value of $B$ on disk. It then comes to the record $<T, A, 8>$ and makes $A$ have value 8 on disk. Finally, the record <ABORT $T$> is written to the log, and the log is flushed.

### Crashes During Recovery

Suppose the system again crashes while we are recovering from a previous crash. Because of the way undo-log records are designed, giving the old value rather than, say, the change in the value of a database element, the recovery steps are *idempotent*; that is, repeating them many times has exactly the same effect as performing them once. We already observed that if we find a record $<T, X, v>$, it does not matter whether the value of $X$ is already $v$ — we may write $v$ for $X$ regardless. Similarly, if we repeat the recovery process, it does not matter whether the first recovery attempt restored some old values; we simply restore them again. The same reasoning holds for the other logging methods we discuss in this chapter. Since the recovery operations are idempotent, we can recover a second time without worrying about changes made the first time.

3. The crash occurs between steps (10) and (11). Now, the `COMMIT` record surely was not written, so $T$ is incomplete and is undone as in case (2).

4. The crash occurs between steps (8) and (10). Again, $T$ is undone. In this case the change to $A$ and/or $B$ may not have reached disk. Nevertheless, the proper value, 8, is restored for each of these database elements.

5. The crash occurs prior to step (8). Now, it is not certain whether any of the log records concerning $T$ have reached disk. However, we know by rule $U_1$ that if the change to $A$ and/or $B$ reached disk, then the corresponding log record reached disk. Therefore if there were changes to $A$ and/or $B$ made on disk by $T$, then the corresponding log record will cause the recovery manager to undo those changes.

□

### 17.2.4 Checkpointing

As we observed, recovery requires that the entire log be examined, in principle. When logging follows the undo style, once a transaction has its `COMMIT` log record written to disk, the log records of that transaction are no longer needed during recovery. We might imagine that we could delete the log prior to a `COMMIT`, but sometimes we cannot. The reason is that often many transactions execute at once. If we truncated the log after one transaction committed, log records pertaining to some other active transaction $T$ might be lost and could not be used to undo $T$ if recovery were necessary.

The simplest way to untangle potential problems is to *checkpoint* the log periodically. In a simple checkpoint, we:

1. Stop accepting new transactions.
2. Wait until all currently active transactions commit or abort and have written a COMMIT or ABORT record on the log.
3. Flush the log to disk.
4. Write a log record <CKPT>, and flush the log again.
5. Resume accepting transactions.

Any transaction that executed prior to the checkpoint will have finished, and by rule $U_2$ its changes will have reached the disk. Thus, there will be no need to undo any of these transactions during recovery. During a recovery, we scan the log backwards from the end, identifying incomplete transactions as in Section 17.2.3. However, when we find a <CKPT> record, we know that we have seen all the incomplete transactions. Since no transactions may begin until the checkpoint ends, we must have seen every log record pertaining to the incomplete transactions already. Thus, there is no need to scan prior to the <CKPT>, and in fact the log before that point can be deleted or overwritten safely.

**Example 17.4:** Suppose the log begins:

<START $T_1$>
<$T_1, A, 5$>
<START $T_2$>
<$T_2, B, 10$>

At this time, we decide to do a checkpoint. Since $T_1$ and $T_2$ are the active (incomplete) transactions, we shall have to wait until they complete before writing the <CKPT> record on the log.

A possible extension of the log is shown in Fig. 17.4. Suppose a crash occurs at this point. Scanning the log from the end, we identify $T_3$ as the only incomplete transaction, and restore $E$ and $F$ to their former values 25 and 30, respectively. When we reach the <CKPT> record, we know there is no need to examine prior log records and the restoration of the database state is complete. □

### 17.2.5 Nonquiescent Checkpointing

A problem with the checkpointing technique described in Section 17.2.4 is that effectively we must shut down the system while the checkpoint is being made. Since the active transactions may take a long time to commit or abort, the system may appear to users to be stalled. Thus, a more complex technique known as *nonquiescent checkpointing*, which allows new transactions to enter the system during the checkpoint, is usually preferred. The steps in a nonquiescent checkpoint are:

<START $T_1$>
<$T_1, A, 5$>
<START $T_2$>
<$T_2, B, 10$>
<$T_2, C, 15$>
<$T_1, D, 20$>
<COMMIT $T_1$>
<COMMIT $T_2$>
<CKPT>
<START $T_3$>
<$T_3, E, 25$>
<$T_3, F, 30$>

Figure 17.4: An undo log

1. Write a log record <START CKPT $(T_1, \ldots, T_k)$> and flush the log. Here, $T_1, \ldots, T_k$ are the names or identifiers for all the *active* transactions (i.e., transactions that have not yet committed and written their changes to disk).

2. Wait until all of $T_1, \ldots, T_k$ commit or abort, but do not prohibit other transactions from starting.

3. When all of $T_1, \ldots, T_k$ have completed, write a log record <END CKPT> and flush the log.

With a log of this type, we can recover from a system crash as follows. As usual, we scan the log from the end, finding all incomplete transactions as we go, and restoring old values for database elements changed by these transactions. There are two cases, depending on whether, scanning backwards, we first meet an <END CKPT> record or a <START CKPT $(T_1, \ldots, T_k)$> record.

- If we first meet an <END CKPT> record, then we know that all incomplete transactions began after the previous <START CKPT $(T_1, \ldots, T_k)$> record. We may thus scan backwards as far as the next START CKPT, and then stop; previous log is useless and may as well have been discarded.

- If we first meet a record <START CKPT $(T_1, \ldots, T_k)$>, then the crash occurred during the checkpoint. However, the only incomplete transactions are those we met scanning backwards before we reached the START CKPT and those of $T_1, \ldots, T_k$ that did not complete before the crash. Thus, we need scan no further back than the start of the earliest of these incomplete transactions. The previous START CKPT record is certainly prior to any of these transaction starts, but often we shall find the starts of the

### Finding the Last Log Record

It is common to recycle blocks of the log file on disk, since checkpoints allow us to drop old portions of the log. However, if we overwrite old log records, then we need to keep a serial number, which may only increase, as suggested by:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ~~1~~ 9 | ~~2~~ 10 | ~~3~~ 11 | 4 | 5 | 6 | 7 | 8 |

Then, we can find the record whose serial number is greater than that of the next record; the latter record will be the current end of the log, and the entire log is found by ordering the current records by their present serial numbers.

In practice, a large log may be composed of many files, with a "top" file whose records indicate the files that comprise the log. Then, to recover, we find the last record of the top file, go to the file indicated, and find the last record there.

incomplete transactions long before we reach the previous checkpoint.[3] Moreover, if we use pointers to chain together the log records that belong to the same transaction, then we need not search the whole log for records belonging to active transactions; we just follow their chains back through the log.

As a general rule, once an <END CKPT> record has been written to disk, we can delete the log prior to the previous START CKPT record.

**Example 17.5:** Suppose that, as in Example 17.4, the log begins:

<START $T_1$>
<$T_1, A, 5$>
<START $T_2$>
<$T_2, B, 10$>

Now, we decide to do a nonquiescent checkpoint. Since $T_1$ and $T_2$ are the active (incomplete) transactions at this time, we write a log record

<START CKPT $(T_1, T_2)$>

Suppose that while waiting for $T_1$ and $T_2$ to complete, another transaction, $T_3$, initiates. A possible continuation of the log is shown in Fig. 17.5.

Suppose that at this point there is a system crash. Examining the log from the end, we find that $T_3$ is an incomplete transaction and must be undone.

[3]Notice, however, that because the checkpoint is nonquiescent, one of the incomplete transactions could have begun between the start and end of the previous checkpoint.

<START $T_1$>
<$T_1, A, 5$>
<START $T_2$>
<$T_2, B, 10$>
<START CKPT $(T_1, T_2)$>
<$T_2, C, 15$>
<START $T_3$>
<$T_1, D, 20$>
<COMMIT $T_1$>
<$T_3, E, 25$>
<COMMIT $T_2$>
<END CKPT>
<$T_3, F, 30$>

Figure 17.5: An undo log using nonquiescent checkpointing

The final log record tells us to restore database element $F$ to the value 30. When we find the <END CKPT> record, we know that all incomplete transactions began after the previous START CKPT. Scanning further back, we find the record <$T_3, E, 25$>, which tells us to restore $E$ to value 25. Between that record, and the START CKPT there are no other transactions that started but did not commit, so no further changes to the database are made.

<START $T_1$>
<$T_1, A, 5$>
<START $T_2$>
<$T_2, B, 10$>
<START CKPT $(T_1, T_2)$>
<$T_2, C, 15$>
<START $T_3$>
<$T_1, D, 20$>
<COMMIT $T_1$>
<$T_3, E, 25$>

Figure 17.6: Undo log with a system crash during checkpointing

Now suppose the crash occurs during the checkpoint, and the end of the log after the crash is as shown in Fig. 17.6. Scanning backwards, we identify $T_3$ and then $T_2$ as incomplete transactions and undo changes they have made. When we find the <START CKPT $(T_1, T_2)$> record, we know that the only other possible incomplete transaction is $T_1$. However, we have already scanned the <COMMIT $T_1$> record, so we know that $T_1$ is *not* incomplete. Also, we have already seen the <START $T_3$> record. Thus, we need only to continue backwards until we meet the START record for $T_2$, restoring database element $B$ to value

10 as we go. □

### 17.2.6 Exercises for Section 17.2

**Exercise 17.2.1:** Show the undo-log records for each of the transactions (call each $T$) of Exercise 17.1.1, assuming that initially $A = 5$ and $B = 10$.

**Exercise 17.2.2:** For each of the sequences of log records representing the actions of one transaction $T$, tell all the sequences of events that are legal according to the rules of undo logging, where the events of interest are the writing to disk of the blocks containing database elements, and the blocks of the log containing the update and commit records. You may assume that log records are written to disk in the order shown; i.e., it is not possible to write one log record to disk while a previous record is not written to disk.

a) <START $T$>; <$T, A, 10$>; <$T, B, 20$>; <COMMIT $T$>;

b) <START $T$>; <$T, A, 10$>; <$T, B, 20$>; <$T, C, 30$><COMMIT $T$>;

! **Exercise 17.2.3:** The pattern introduced in Exercise 17.2.2 can be extended to a transaction that writes new values for $n$ database elements. How many legal sequences of events are there for such a transaction, if the undo-logging rules are obeyed?

**Exercise 17.2.4:** The following is a sequence of undo-log records written by two transactions $T$ and $U$: <START $T$>; <$T, A, 10$>; <START $U$>; <$U, B, 20$>; <$T, C, 30$>; <$U, D, 40$>; <COMMIT $U$>; <$T, E, 50$>; <COMMIT $T$>. Describe the action of the recovery manager, including changes to both disk and the log, if there is a crash and the last log record to appear on disk is:

(a) <START $U$> (b) <COMMIT $U$> (c) <$T, E, 50$> (d) <COMMIT $T$>.

**Exercise 17.2.5:** For each of the situations described in Exercise 17.2.4, what values written by $T$ and $U$ *must* appear on disk? Which values *might* appear on disk?

! **Exercise 17.2.6:** Suppose that the transaction $U$ in Exercise 17.2.4 is changed so that the record <$U, D, 40$> becomes <$U, A, 40$>. What is the effect on the disk value of $A$ if there is a crash at some point during the sequence of events? What does this example say about the ability of logging by itself to preserve atomicity of transactions?

**Exercise 17.2.7:** Consider the following sequence of log records: <START $S$>; <$S, A, 60$>; <COMMIT $S$>; <START $T$>; <$T, A, 10$>; <START $U$>; <$U, B, 20$>; <$T, C, 30$>; <START $V$>; <$U, D, 40$>; <$V, F, 70$>; <COMMIT $U$>; <$T, E, 50$>; <COMMIT $T$>; <$V, B, 80$>; <COMMIT $V$>. Suppose that we begin a nonquiescent checkpoint immediately after one of the following log records has been written (in memory):

(a) <$S, A, 60$> (b) <$T, A, 10$> (c) <$U, B, 20$>
(d) <$U, D, 40$> (e) <$T, E, 50$>

For each, tell:

*i.* When the <`END CKPT`> record is written, and

*ii.* For each possible point at which a crash could occur, how far back in the log we must look to find all possible incomplete transactions.

## 17.3 Redo Logging

Undo logging has a potential problem that we cannot commit a transaction without first writing all its changed data to disk. Sometimes, we can save disk I/O's if we let changes to the database reside only in main memory for a while. As long as there is a log to fix things up in the event of a crash, it is safe to do so.

The requirement for immediate backup of database elements to disk can be avoided if we use a logging mechanism called *redo logging.* The principal differences between redo and undo logging are:

1. While undo logging cancels the effect of incomplete transactions and ignores committed ones during recovery, redo logging ignores incomplete transactions and repeats the changes made by committed transactions.

2. While undo logging requires us to write changed database elements to disk before the `COMMIT` log record reaches disk, redo logging requires that the `COMMIT` record appear on disk before any changed values reach disk.

3. While the old values of changed database elements are exactly what we need to recover when the undo rules $U_1$ and $U_2$ are followed, to recover using redo logging, we need the new values instead.

### 17.3.1 The Redo-Logging Rule

In redo logging the meaning of a log record <$T, X, v$> is "transaction $T$ wrote new value $v$ for database element $X$." There is no indication of the old value of $X$ in this record. Every time a transaction $T$ modifies a database element $X$, a record of the form <$T, X, v$> must be written to the log.

For redo logging, the order in which data and log entries reach disk can be described by a single "redo rule," called the *write-ahead logging rule.*

$R_1$: Before modifying any database element $X$ on disk, it is necessary that all log records pertaining to this modification of $X$, including both the update record <$T, X, v$> and the <`COMMIT` $T$> record, must appear on disk.

The COMMIT record for a transaction can only be written to the log when the transaction completes, so the commit record must follow all the update log records. Thus, when redo logging is in use, the order in which material associated with one transaction gets written to disk is:

1. The log records indicating changed database elements.
2. The COMMIT log record.
3. The changed database elements themselves.

**Example 17.6:** Let us consider the same transaction $T$ as in Example 17.2. Figure 17.7 shows a possible sequence of events for this transaction.

| Step | Action | $t$ | M-$A$ | M-$B$ | D-$A$ | D-$B$ | Log |
|---|---|---|---|---|---|---|---|
| 1) | | | | | | | <START $T$> |
| 2) | READ(A,t) | 8 | 8 | | 8 | 8 | |
| 3) | t := t*2 | 16 | 8 | | 8 | 8 | |
| 4) | WRITE(A,t) | 16 | 16 | | 8 | 8 | <$T, A, 16$> |
| 5) | READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | t := t*2 | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <$T, B, 16$> |
| 8) | | | | | | | <COMMIT $T$> |
| 9) | FLUSH LOG | | | | | | |
| 10) | OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| 11) | OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |

Figure 17.7: Actions and their log entries using redo logging

The major differences between Figs. 17.7 and 17.3 are as follows. First, we note in lines (4) and (7) of Fig. 17.7 that the log records reflecting the changes have the new values of $A$ and $B$, rather than the old values. Second, we see that the <COMMIT $T$> record comes earlier, at step (8). Then, the log is flushed, so all log records involving the changes of transaction $T$ appear on disk. Only then can the new values of $A$ and $B$ be written to disk. We show these values written immediately, at steps (10) and (11), although in practice they might occur later. □

### 17.3.2 Recovery With Redo Logging

An important consequence of the redo rule $R_1$ is that unless the log has a <COMMIT $T$> record, we know that no changes to the database made by transaction $T$ have been written to disk. Thus, incomplete transactions may be treated during recovery as if they had never occurred. However, the committed transactions present a problem, since we do not know which of their database changes have been written to disk. Fortunately, the redo log has exactly the

### Order of Redo Matters

Since several committed transactions may have written new values for the same database element $X$, we have required that during a redo recovery, we scan the log from earliest to latest. Thus, the final value of $X$ in the database will be the one written last, as it should be. Similarly, when describing undo recovery, we required that the log be scanned from latest to earliest. Thus, the final value of $X$ will be the value that it had before any of the incomplete transactions changed it.

However, if the DBMS enforces atomicity, then we would not expect to find, in an undo log, two uncommitted transactions, each of which had written the same database element. In contrast, with redo logging we focus on the committed transactions, as these need to be redone. It is quite normal for there to be two *committed* transactions, each of which changed the same database element at different times. Thus, order of redo is always important, while order of undo might not be if the right kind of concurrency control were in effect.

information we need: the new values, which we may write to disk regardless of whether they were already there. To recover, using a redo log, after a system crash, we do the following.

1. Identify the committed transactions.
2. Scan the log forward from the beginning. For each log record $<T, X, v>$ encountered:
   (a) If $T$ is not a committed transaction, do nothing.
   (b) If $T$ is committed, write value $v$ for database element $X$.
3. For each incomplete transaction $T$, write an <ABORT $T$> record to the log and flush the log.

**Example 17.7:** Let us consider the log written in Fig. 17.7 and see how recovery would be performed if the crash occurred after different steps in that sequence of actions.

1. If the crash occurs any time after step (9), then the <COMMIT $T$> record has been flushed to disk. The recovery system identifies $T$ as a committed transaction. When scanning the log forward, the log records $<T, A, 16>$ and $<T, B, 16>$ cause the recovery manager to write values 16 for $A$ and $B$. Notice that if the crash occurred between steps (10) and (11), then the write of $A$ is redundant, but the write of $B$ had not occurred and

changing $B$ to 16 is essential to restore the database state to consistency. If the crash occurred after step (11), then both writes are redundant but harmless.

2. If the crash occurs between steps (8) and (9), then although the record <COMMIT $T$> was written to the log, it may not have gotten to disk (depending on whether the log was flushed for some other reason). If it did get to disk, then the recovery proceeds as in case (1), and if it did not get to disk, then recovery is as in case (3), below.

3. If the crash occurs prior to step (8), then <COMMIT $T$> surely has not reached disk. Thus, $T$ is treated as an incomplete transaction. No changes to $A$ or $B$ on disk are made on behalf of $T$, and eventually an <ABORT $T$> record is written to the log.

□

### 17.3.3 Checkpointing a Redo Log

Redo logs present a checkpointing problem that we do not see with undo logs. Since the database changes made by a committed transaction can be copied to disk much later than the time at which the transaction commits, we cannot limit our concern to transactions that are active at the time we decide to create a checkpoint. Regardless of whether the checkpoint is quiescent or nonquiescent, between the start and end of the checkpoint we must write to disk all database elements that have been modified by committed transactions. To do so requires that the buffer manager keep track of which buffers are *dirty*, that is, they have been changed but not written to disk. It is also required to know which transactions modified which buffers.

On the other hand, we can complete the checkpoint without waiting for the active transactions to commit or abort, since they are not allowed to write their pages to disk at that time anyway. The steps to perform a nonquiescent checkpoint of a redo log are as follows:

1. Write a log record <START CKPT ($T_1, \ldots, T_k$)>, where $T_1, \ldots, T_k$ are all the active (uncommitted) transactions, and flush the log.

2. Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the START CKPT record was written to the log.

3. Write an <END CKPT> record to the log and flush the log.

**Example 17.8:** Figure 17.8 shows a possible redo log, in the middle of which a checkpoint occurs. When we start the checkpoint, only $T_2$ is active, but the value of $A$ written by $T_1$ may have reached disk. If not, then we must copy $A$

<START $T_1$>
<$T_1, A, 5$>
<START $T_2$>
<COMMIT $T_1$>
<$T_2, B, 10$>
<START CKPT ($T_2$)>
<$T_2, C, 15$>
<START $T_3$>
<$T_3, D, 20$>
<END CKPT>
<COMMIT $T_2$>
<COMMIT $T_3$>

Figure 17.8: A redo log

to disk before the checkpoint can end. We suggest the end of the checkpoint occurring after several other events have occurred: $T_2$ wrote a value for database element $C$, and a new transaction $T_3$ started and wrote a value of $D$. After the end of the checkpoint, the only things that happen are that $T_2$ and $T_3$ commit. □

### 17.3.4 Recovery With a Checkpointed Redo Log

As for an undo log, the insertion of records to mark the start and end of a checkpoint helps us limit our examination of the log when a recovery is necessary. Also as with undo logging, there are two cases, depending on whether the last checkpoint record is START or END.

Suppose first that the last checkpoint record on the log before a crash is <END CKPT>. Now, we know that every value written by a transaction that committed before the corresponding <START CKPT ($T_1, \ldots, T_k$)> has had its changes written to disk, so we need not concern ourselves with recovering the effects of these transactions. However, any transaction that is either among the $T_i$'s or that started after the beginning of the checkpoint can still have changes it made not yet migrated to disk, even though the transaction has committed. Thus, we must perform recovery as described in Section 17.3.2, but may limit our attention to the transactions that are either one of the $T_i$'s mentioned in the last <START CKPT ($T_1, \ldots, T_k$)> or that started after that log record appeared in the log. In searching the log, we do not have to look further back than the earliest of the <START $T_i$> records. Notice, however, that these START records could appear prior to any number of checkpoints. Linking backwards all the log records for a given transaction helps us to find the necessary records, as it did for undo logging.

Now, suppose the last checkpoint record on the log is

<START CKPT $(T_1, \ldots, T_k)$>

We cannot be sure that committed transactions prior to the start of this checkpoint had their changes written to disk. Thus, we must search back to the previous <END CKPT> record, find its matching <START CKPT $(S_1, \ldots, S_m)$> record,[4] and redo all those committed transactions that either started after that START CKPT or are among the $S_i$'s.

**Example 17.9:** Consider again the log of Fig. 17.8. If a crash occurs at the end, we search backwards, finding the <END CKPT> record. We thus know that it is sufficient to consider as candidates to redo all those transactions that either started after the <START CKPT $(T_2)$> record was written or that are on its list (i.e., $T_2$). Thus, our candidate set is $\{T_2, T_3\}$. We find the records <COMMIT $T_2$> and <COMMIT $T_3$>, so we know that each must be redone. We search the log as far back as the <START $T_2$> record, and find the update records $<T_2, B, 10>$, $<T_2, C, 15>$, and $<T_3, D, 20>$ for the committed transactions. Since we don't know whether these changes reached disk, we rewrite the values 10, 15, and 20 for $B$, $C$, and $D$, respectively.

Now, suppose the crash occurred between the records <COMMIT $T_2$> and <COMMIT $T_3$>. The recovery is similar to the above, except that $T_3$ is no longer a committed transaction. Thus, its change $<T_3, D, 20>$ must *not* be redone, and no change is made to $D$ during recovery, even though that log record is in the range of records that is examined. Also, we write an <ABORT $T_3$> record to the log after recovery.

Finally, suppose that the crash occurs just prior to the <END CKPT> record. In principal, we must search back to the next-to-last START CKPT record and get its list of active transactions. However, in this case there is no previous checkpoint, and we must go all the way to the beginning of the log. Thus, we identify $T_1$ as the only committed transaction, redo its action $<T_1, A, 5>$, and write records <ABORT $T_2$> and <ABORT $T_3$> to the log after recovery. □

Since transactions may be active during several checkpoints, it is convenient to include in the <START CKPT $(T_1, \ldots, T_k)$> records not only the names of the active transactions, but pointers to the place on the log where they started. By doing so, we know when it is safe to delete early portions of the log. When we write an <END CKPT>, we know that we shall never need to look back further than the earliest of the <START $T_i$> records for the active transactions $T_i$. Thus, anything prior to that START record may be deleted.

### 17.3.5 Exercises for Section 17.3

**Exercise 17.3.1:** Show the redo-log records for each of the transactions (call each $T$) of Exercise 17.1.1, assuming that initially $A = 5$ and $B = 10$.

[4] There is a small technicality that there could be a START CKPT record that, because of a previous crash, has no matching <END CKPT> record. Therefore, we must look not just for the previous START CKPT, but first for an <END CKPT> and then the previous START CKPT.

**Exercise 17.3.2:** Repeat Exercise 17.2.2 for redo logging.

**Exercise 17.3.3:** Repeat Exercise 17.2.4 for redo logging.

**Exercise 17.3.4:** Repeat Exercise 17.2.5 for redo logging.

**Exercise 17.3.5:** Using the data of Exercise 17.2.7, answer for each of the positions (a) through (e) of that exercise:

*i.* At what points could the <END CKPT> record be written, and

*ii.* For each possible point at which a crash could occur, how far back in the log we must look to find all possible incomplete transactions. Consider both the case that the <END CKPT> record was or was not written prior to the crash.

## 17.4 Undo/Redo Logging

We have seen two different approaches to logging, differentiated by whether the log holds old values or new values when a database element is updated. Each has certain drawbacks:

- Undo logging requires that data be written to disk immediately after a transaction finishes, perhaps increasing the number of disk I/O's that need to be performed.

- On the other hand, redo logging requires us to keep all modified blocks in buffers until the transaction commits and the log records have been flushed, perhaps increasing the average number of buffers required by transactions.

- Both undo and redo logs may put contradictory requirements on how buffers are handled during a checkpoint, unless the database elements are complete blocks or sets of blocks. For instance, if a buffer contains one database element $A$ that was changed by a committed transaction and another database element $B$ that was changed in the same buffer by a transaction that has not yet had its COMMIT record written to disk, then we are required to copy the buffer to disk because of $A$ but also forbidden to do so, because rule $R_1$ applies to $B$.

We shall now see a kind of logging called *undo/redo logging*, that provides increased flexibility to order actions, at the expense of maintaining more information on the log.

### 17.4.1 The Undo/Redo Rules

An undo/redo log has the same sorts of log records as the other kinds of log, with one exception. The update log record that we write when a database element changes value has four components. Record $<T, X, v, w>$ means that transaction $T$ changed the value of database element $X$; its former value was $v$, and its new value is $w$. The constraints that an undo/redo logging system must follow are summarized by the following rule:

$UR_1$ Before modifying any database element $X$ on disk because of changes made by some transaction $T$, it is necessary that the update record $<T, X, v, w>$ appear on disk.

Rule $UR_1$ for undo/redo logging thus enforces only the constraints enforced by *both* undo logging and redo logging. In particular, the <COMMIT $T$> log record can precede or follow any of the changes to the database elements on disk.

**Example 17.10:** Figure 17.9 is a variation in the order of the actions associated with the transaction $T$ that we last saw in Example 17.6. Notice that the log records for updates now have both the old and the new values of $A$ and $B$. In this sequence, we have written the <COMMIT $T$> log record in the middle of the output of database elements $A$ and $B$ to disk. Step (10) could also have appeared before step (8) or step (9), or after step (11). □

| Step | Action | $t$ | M-$A$ | M-$B$ | D-$A$ | D-$B$ | Log |
|---|---|---|---|---|---|---|---|
| 1) | | | | | | | <START $T$> |
| 2) | READ(A,t) | 8 | 8 | | 8 | 8 | |
| 3) | t := t*2 | 16 | 8 | | 8 | 8 | |
| 4) | WRITE(A,t) | 16 | 16 | | 8 | 8 | $<T, A, 8, 16>$ |
| 5) | READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| 6) | t := t*2 | 16 | 16 | 8 | 8 | 8 | |
| 7) | WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | $<T, B, 8, 16>$ |
| 8) | FLUSH LOG | | | | | | |
| 9) | OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| 10) | | | | | | | <COMMIT $T$> |
| 11) | OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |

Figure 17.9: A possible sequence of actions and their log entries using undo/redo logging

### 17.4.2 Recovery With Undo/Redo Logging

When we need to recover using an undo/redo log, we have the information in the update records either to undo a transaction $T$ by restoring the old values of

### A Problem With Delayed Commitment

Like undo logging, a system using undo/redo logging can exhibit a behavior where a transaction appears to the user to have been completed (e.g., they booked an airline seat over the Web and disconnected), and yet because the <COMMIT $T$> record was not flushed to disk, a subsequent crash causes the transaction to be undone rather than redone. If this possibility is a problem, we suggest the use of an additional rule for undo/redo logging:

$UR_2$ A <COMMIT $T$> record must be flushed to disk as soon as it appears in the log.

For instance, we would add FLUSH LOG after step (10) of Fig. 17.9.

the database elements that $T$ changed, or to redo $T$ by repeating the changes it has made. The undo/redo recovery policy is:

1. Redo all the committed transactions in the order earliest-first, and
2. Undo all the incomplete transactions in the order latest-first.

Notice that it is necessary for us to do both. Because of the flexibility allowed by undo/redo logging regarding the relative order in which COMMIT log records and the database changes themselves are copied to disk, we could have either a committed transaction with some or all of its changes not on disk, or an uncommitted transaction with some or all of its changes on disk.

**Example 17.11:** Consider the sequence of actions in Fig. 17.9. Here are the different ways that recovery would take place on the assumption that there is a crash at various points in the sequence.

1. Suppose the crash occurs after the <COMMIT $T$> record is flushed to disk. Then $T$ is identified as a committed transaction. We write the value 16 for both $A$ and $B$ to the disk. Because of the actual order of events, $A$ already has the value 16, but $B$ may not, depending on whether the crash occurred before or after step (11).
2. If the crash occurs prior to the <COMMIT $T$> record reaching disk, then $T$ is treated as an incomplete transaction. The previous values of $A$ and $B$, 8 in each case, are written to disk. If the crash occurs between steps (9) and (10), then the value of $A$ was 16 on disk, and the restoration to value 8 is necessary. In this example, the value of $B$ does not need to be undone, and if the crash occurs before step (9) then neither does the value of $A$. However, in general we cannot be sure whether restoration is necessary, so we always perform the undo operation.

□

### Strange Behavior of Transactions During Recovery

You may have noticed that we did not specify whether undo's or redo's are done first during recovery using an undo/redo log. In fact, whether we perform the redo's or undo's first, we are open to the following situation: a transaction $T$ has committed and is redone. However, $T$ read a value $X$ written by some transaction $U$ that has not committed and is undone. The problem is not whether we redo first, and leave $X$ with its value prior to $U$, or we undo first and leave $X$ with its value written by $T$. The situation makes no sense either way, because the final database state does not correspond to the effect of any sequence of atomic transactions.

In reality, the DBMS must do more than log changes. It must assure that such situations do not occur at all. In Chapter 18, there is a discussion about the means to isolate transactions like $T$ and $U$, so the interaction between them through database element $X$ cannot occur. In Section 19.1, we explicitly address means for preventing this situation where $T$ reads a "dirty" value of $X$ — one that has not been committed.

## 17.4.3 Checkpointing an Undo/Redo Log

A nonquiescent checkpoint is somewhat simpler for undo/redo logging than for the other logging methods. We have only to do the following:

1. Write a <START CKPT ($T_1, \ldots, T_k$)> record to the log, where $T_1, \ldots, T_k$ are all the active transactions, and flush the log.
2. Write to disk all the buffers that are *dirty*; i.e., they contain one or more changed database elements. Unlike redo logging, we flush all dirty buffers, not just those written by committed transactions.
3. Write an <END CKPT> record to the log, and flush the log.

Notice in connection with point (2) that, because of the flexibility undo/redo logging offers regarding when data reaches disk, we can tolerate the writing to disk of data written by incomplete transactions. Therefore we can tolerate database elements that are smaller than complete blocks and thus may share buffers. The only requirement we must make on transactions is:

- A transaction must not write any values (even to memory buffers) until it is certain not to abort.

As we shall see in Section 19.1, this constraint is almost certainly needed anyway, in order to avoid inconsistent interactions between transactions. Notice that under redo logging, the above condition is not sufficient, since even if the transaction that wrote $B$ is certain to commit, rule $R_1$ requires that the transaction's COMMIT record be written to disk before $B$ is written to disk.

**Example 17.12:** Figure 17.10 shows an undo/redo log analogous to the redo log of Fig. 17.8. We have changed only the update records, giving them an old value as well as a new value. For simplicity, we have assumed that in each case the old value is one less than the new value.

<START $T_1$>
<$T_1, A, 4, 5$>
<START $T_2$>
<COMMIT $T_1$>
<$T_2, B, 9, 10$>
<START CKPT ($T_2$)>
<$T_2, C, 14, 15$>
<START $T_3$>
<$T_3, D, 19, 20$>
<END CKPT>
<COMMIT $T_2$>
<COMMIT $T_3$>

Figure 17.10: An undo/redo log

As in Example 17.8, $T_2$ is identified as the only active transaction when the checkpoint begins. Since this log is an undo/redo log, it is possible that $T_2$'s new $B$-value 10 has been written to disk, which was not possible under redo logging. However, it is irrelevant whether or not that disk write has occurred. During the checkpoint, we shall surely flush $B$ to disk if it is not already there, since we flush all dirty buffers. Likewise, we shall flush $A$, written by the committed transaction $T_1$, if it is not already on disk.

If the crash occurs at the end of this sequence of events, then $T_2$ and $T_3$ are identified as committed transactions. Transaction $T_1$ is prior to the checkpoint. Since we find the <END CKPT> record on the log, $T_1$ is correctly assumed to have both completed and had its changes written to disk. We therefore redo both $T_2$ and $T_3$, as in Example 17.8, and ignore $T_1$. However, when we redo a transaction such as $T_2$, we do not need to look prior to the <START CKPT ($T_2$)> record, even though $T_2$ was active at that time, because we know that $T_2$'s changes prior to the start of the checkpoint were flushed to disk during the checkpoint.

For another instance, suppose the crash occurs just before the <COMMIT $T_3$> record is written to disk. Then we identify $T_2$ as committed but $T_3$ as incomplete. We redo $T_2$ by setting $C$ to 15 on disk; it is not necessary to set $B$ to 10 since we know that change reached disk before the <END CKPT>. However, unlike the situation with a redo log, we also undo $T_3$; that is, we set $D$ to 19 on disk. If $T_3$ had been active at the start of the checkpoint, we would have had to look prior to the START-CKPT record to find if there were more actions by $T_3$ that may have reached disk and need to be undone. □

### 17.4.4 Exercises for Section 17.4

**Exercise 17.4.1:** Show the undo/redo-log records for each of the transactions (call each $T$) of Exercise 17.1.1, assuming that initially $A = 5$ and $B = 10$.

**Exercise 17.4.2:** For each of the sequences of log records representing the actions of one transaction $T$, tell all the sequences of events that are legal according to the rules of undo/redo logging, where the events of interest are the writing to disk of the blocks containing database elements, and the blocks of the log containing the update and commit records. You may assume that log records are written to disk in the order shown; i.e., it is not possible to write one log record to disk while a previous record is not written to disk.

a) <START $T$>; <$T, A, 10, 11$>; <$T, B, 20, 21$>; <COMMIT $T$>;

b) <START $T$>; <$T, A, 10, 21$>; <$T, B, 20, 21$>; <$T, C, 30, 31$>; <COMMIT $T$>;

**Exercise 17.4.3:** The following is a sequence of undo/redo-log records written by two transactions $T$ and $U$: <START $T$>; <$T, A, 10, 11$>; <START $U$>; <$U, B, 20, 21$>; <$T, C, 30, 31$>; <$U, D, 40, 41$>; <COMMIT $U$>; <$T, E, 50, 51$>; <COMMIT $T$>. Describe the action of the recovery manager, including changes to both disk and the log, if there is a crash and the last log record to appear on disk is:

(a) <START $U$> (b) <COMMIT $U$> (c) <$T, E, 50, 51$> (d) <COMMIT $T$>.

**Exercise 17.4.4:** For each of the situations described in Exercise 17.4.3, what values written by $T$ and $U$ *must* appear on disk? Which values *might* appear on disk?

**Exercise 17.4.5:** Consider the following sequence of log records: <START $S$>; <$S, A, 60, 61$>; <COMMIT $S$>; <START $T$>; <$T, A, 61, 62$>; <START $U$>; <$U, B, 20, 21$>; <$T, C, 30, 31$>; <START $V$>; <$U, D, 40, 41$>; <$V, F, 70, 71$>; <COMMIT $U$>; <$T, E, 50, 51$>; <COMMIT $T$>; <$V, B, 21, 22$>; <COMMIT $V$>. Suppose that we begin a nonquiescent checkpoint immediately after one of the following log records has been written (in memory):

(a) <$S, A, 60, 61$> (b) <$T, A, 61, 62$> (c) <$U, B, 20, 21$>
(d) <$U, D, 40, 41$> (e) <$T, E, 50, 51$>

For each, tell:

*i.* At what points could the <END CKPT> record be written, and

*ii.* For each possible point at which a crash could occur, how far back in the log we must look to find all possible incomplete transactions. Consider both the case that the <END CKPT> record was or was not written prior to the crash.

## 17.5 Protecting Against Media Failures

The log can protect us against system failures, where nothing is lost from disk, but temporary data in main memory is lost. However, as we discussed in Section 17.1.1, more serious failures involve the loss of one or more disks. An archiving system, which we cover next, is needed to enable a database to survive losses involving disk-resident data.

### 17.5.1 The Archive

To protect against media failures, we are thus led to a solution involving *archiving* — maintaining a copy of the database separate from the database itself. If it were possible to shut down the database for a while, we could make a backup copy on some storage medium such as tape or optical disk, and store the copy remote from the database, in some secure location. The backup would preserve the database state as it existed at the time of the backup, and if there were a media failure, the database could be restored to this state.

To advance to a more recent state, we could use the log, provided the log had been preserved since the archive copy was made, and the log itself survived the failure. In order to protect against losing the log, we could transmit a copy of the log, almost as soon as it is created, to the same remote site as the archive. Then, if the log as well as the data is lost, we can use the archive plus remotely stored log to recover, at least up to the point that the log was last transmitted to the remote site.

Since writing an archive is a lengthy process, we try to avoid copying the entire database at each archiving step. Thus, we distinguish between two levels of archiving:

1. A *full dump*, in which the entire database is copied.
2. An *incremental dump*, in which only those database elements changed since the previous full or incremental dump are copied.

It is also possible to have several levels of dump, with a full dump thought of as a "level 0" dump, and a "level $i$" dump copying everything changed since the last dump at a level less than or equal to $i$.

We can restore the database from a full dump and its subsequent incremental dumps, in a process much like the way a redo or undo/redo log can be used to repair damage due to a system failure. We copy the full dump back to the database, and then in an earliest-first order, make the changes recorded by the later incremental dumps.

### 17.5.2 Nonquiescent Archiving

The problem with the simple view of archiving in Section 17.5.1 is that most databases cannot be shut down for the period of time (possibly hours) needed

### Why Not Just Back Up the Log?

We might question the need for an archive, since we have to back up the log in a secure place anyway if we are not to be stuck at the state the database was in when the previous archive was made. While it may not be obvious, the answer lies in the typical rate of change of a large database. While only a small fraction of the database may change in a day, the changes, each of which must be logged, will over the course of a year become much larger than the database itself. If we never archived, then the log could never be truncated, and the cost of storing the log would soon exceed the cost of storing a copy of the database.

to make a backup copy. We thus need to consider *nonquiescent archiving*, which is analogous to nonquiescent checkpointing. Recall that a nonquiescent checkpoint attempts to make a copy on the disk of the (approximate) database state that existed when the checkpoint started. We can rely on a small portion of the log around the time of the checkpoint to fix up any deviations from that database state, due to the fact that during the checkpoint, new transactions may have started and written to disk.

Similarly, a nonquiescent dump tries to make a copy of the database that existed when the dump began, but database activity may change many database elements on disk during the minutes or hours that the dump takes. If it is necessary to restore the database from the archive, the log entries made during the dump can be used to sort things out and get the database to a consistent state. The analogy is suggested by Fig. 17.11.

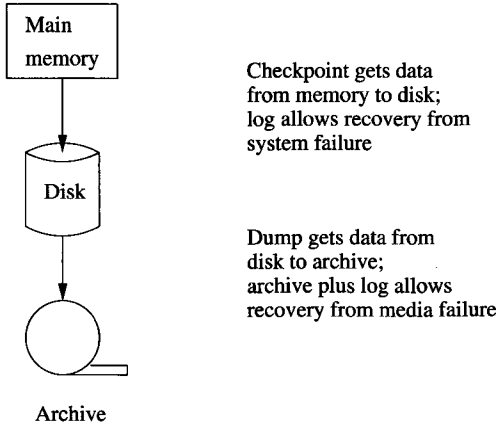


Figure 17.11: The analogy between checkpoints and dumps

A nonquiescent dump copies the database elements in some fixed order, possibly while those elements are being changed by executing transactions. As a result, the value of a database element that is copied to the archive may or may not be the value that existed when the dump began. As long as the log for the duration of the dump is preserved, the discrepancies can be corrected from the log.

**Example 17.13:** For a very simple example, suppose that our database consists of four elements, $A$, $B$, $C$, and $D$, which have the values 1 through 4, respectively, when the dump begins. During the dump, $A$ is changed to 5, $C$ is changed to 6, and $B$ is changed to 7. However, the database elements are copied in order, and the sequence of events shown in Fig. 17.12 occurs. Then although the database at the beginning of the dump has values $(1, 2, 3, 4)$, and the database at the end of the dump has values $(5, 7, 6, 4)$, the copy of the database in the archive has values $(1, 2, 6, 4)$, a database state that existed at no time during the dump. □

| Disk | Archive |
|---|---|
| | Copy $A$ |
| `A := 5` | |
| | Copy $B$ |
| `C := 6` | |
| | Copy $C$ |
| `B := 7` | |
| | Copy $D$ |

Figure 17.12: Events during a nonquiescent dump

In more detail, the process of making an archive can be broken into the following steps. We assume that the logging method is either redo or undo/redo; an undo log is not suitable for use with archiving.

1. Write a log record <START DUMP>.
2. Perform a checkpoint appropriate for whichever logging method is being used.
3. Perform a full or incremental dump of the data disk(s), as desired, making sure that the copy of the data has reached the secure, remote site.
4. Make sure that enough of the log has been copied to the secure, remote site that at least the prefix of the log up to and including the checkpoint in item (2) will survive a media failure of the database.
5. Write a log record <END DUMP>.

At the completion of the dump, it is safe to throw away log prior to the beginning of the checkpoint *previous* to the one performed in item (2) above.

**Example 17.14:** Suppose that the changes to the simple database in Example 17.13 were caused by two transactions $T_1$ (which writes $A$ and $B$) and $T_2$ (which writes $C$) that were active when the dump began. Figure 17.13 shows a possible undo/redo log of the events during the dump.

<START DUMP>
<START CKPT ($T_1, T_2$)>
<$T_1, A, 1, 5$>
<$T_2, C, 3, 6$>
<COMMIT $T_2$>
<$T_1, B, 2, 7$>
<END CKPT>
Dump completes
<END DUMP>

Figure 17.13: Log taken during a dump

Notice that we did not show $T_1$ committing. It would be unusual that a transaction remained active during the entire time a full dump was in progress, but that possibility doesn't affect the correctness of the recovery method that we discuss next. □

### 17.5.3 Recovery Using an Archive and Log

Suppose that a media failure occurs, and we must reconstruct the database from the most recent archive and whatever prefix of the log has reached the remote site and has not been lost in the crash. We perform the following steps:

1. Restore the database from the archive.
   (a) Find the most recent full dump and reconstruct the database from it (i.e., copy the archive into the database).
   (b) If there are later incremental dumps, modify the database according to each, earliest first.
2. Modify the database using the surviving log. Use the method of recovery appropriate to the log method being used.

**Example 17.15:** Suppose there is a media failure after the dump of Example 17.14 completes, and the log shown in Fig. 17.13 survives. Assume, to make the process interesting, that the surviving portion of the log does not include a <COMMIT $T_1$> record, although it does include the <COMMIT $T_2$> record shown

in that figure. The database is first restored to the values in the archive, which is, for database elements $A$, $B$, $C$, and $D$, respectively, $(1, 2, 6, 4)$.

Now, we must look at the log. Since $T_2$ has completed, we redo the step that sets $C$ to 6. In this example, $C$ already had the value 6, but it might be that:

a) The archive for $C$ was made before $T_2$ changed $C$, or

b) The archive actually captured a later value of $C$, which may or may not have been written by a transaction whose commit record survived. Later in the recovery, $C$ will be restored to the value found in the archive *if* the transaction was committed.

Since $T_1$ does not have a `COMMIT` record, we must undo $T_1$. We use the log records for $T_1$ to determine that $A$ must be restored to value 1 and $B$ to 2. It happens that they had these values in the archive, but the actual archive value could have been different because the modified $A$ and/or $B$ had been included in the archive. □

### 17.5.4 Exercises for Section 17.5

**Exercise 17.5.1:** If a redo log, rather than an undo/redo log, were used in Examples 17.14 and 17.15:

a) What would the log look like?

! b) If we had to recover using the archive and this log, what would be the consequence of $T_1$ not having committed?

c) What would be the state of the database after recovery?

## 17.6 Summary of Chapter 17

- ✦ *Transaction Management*: The two principal tasks of the transaction manager are assuring recoverability of database actions through logging, and assuring correct, concurrent behavior of transactions through the scheduler (discussed in the next chapter).

- ✦ *Database Elements*: The database is divided into elements, which are typically disk blocks, but could be tuples or relations, for instance. Database elements are the units for both logging and scheduling.

- ✦ *Logging*: A record of every important action of a transaction — beginning, changing a database element, committing, or aborting — is stored on a log. The log must be backed up on disk at a time that is related to when the corresponding database changes migrate to disk, but that time depends on the particular logging method used.

- *Recovery*: When a system crash occurs, the log is used to repair the database, restoring it to a consistent state.

- *Logging Methods*: The three principal methods for logging are undo, redo, and undo/redo, named for the way(s) that they are allowed to fix the database during recovery.

- *Undo Logging*: This method logs the old value, each time a database element is changed. With undo logging, a new value of a database element can be written to disk only after the log record for the change has reached disk, but before the commit record for the transaction performing the change reaches disk. Recovery is done by restoring the old value for every uncommitted transaction.

- *Redo Logging*: Here, only the new value of database elements is logged. With this form of logging, values of a database element can be written to disk only after both the log record of its change and the commit record for its transaction have reached disk. Recovery involves rewriting the new value for every committed transaction.

- *Undo/Redo Logging* In this method, both old and new values are logged. Undo/redo logging is more flexible than the other methods, since it requires only that the log record of a change appear on the disk before the change itself does. There is no requirement about when the commit record appears. Recovery is effected by redoing committed transactions and undoing the uncommitted transactions.

- *Checkpointing*: Since all recovery methods require, in principle, looking at the entire log, the DBMS must occasionally checkpoint the log, to assure that no log records prior to the checkpoint will be needed during a recovery. Thus, old log records can eventually be thrown away and their disk space reused.

- *Nonquiescent Checkpointing*: To avoid shutting down the system while a checkpoint is made, techniques associated with each logging method allow the checkpoint to be made while the system is in operation and database changes are occurring. The only cost is that some log records prior to the nonquiescent checkpoint may need to be examined during recovery.

- *Archiving*: While logging protects against system failures involving only the loss of main memory, archiving is necessary to protect against failures where the contents of disk are lost. Archives are copies of the database stored in a safe place.

- *Incremental Backups*: Instead of copying the entire database to an archive periodically, a single complete backup can be followed by several incremental backups, where only the changed data is copied to the archive.

- *Nonquiescent Archiving*: We can create a backup of the data while the database is in operation. The necessary techniques involve making log records of the beginning and end of the archiving, as well as performing a checkpoint for the log during the archiving.

- *Recovery From Media Failures*: When a disk is lost, it may be restored by starting with a full backup of the database, modifying it according to any later incremental backups, and finally recovering to a consistent database state by using an archived copy of the log.

## 17.7 References for Chapter 17

The major textbook on all aspects of transaction processing, including logging and recovery, is by Gray and Reuter [5]. This book was partially fed by some informal notes on transaction processing by Jim Gray [3] that were widely circulated; the latter, along with [4] and [8] are the primary sources for much of the logging and recovery technology.

[2] is an earlier, more concise description of transaction-processing technology. [7] is a recent treatment of recovery.

Two early surveys, [1] and [6] both represent much of the fundamental work in recovery and organized the subject in the undo-redo-undo/redo tricotomy that we followed here.

1. P. A. Bernstein, N. Goodman, and V. Hadzilacos, "Recovery algorithms for database systems," *Proc. 1983 IFIP Congress*, North Holland, Amsterdam, pp. 799–807.

2. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading MA, 1987.

3. J. N. Gray, "Notes on database operating systems," in *Operating Systems: an Advanced Course*, pp. 393–481, Springer-Verlag, 1978.

4. J. N. Gray, P. R. McJones, and M. Blasgen, "The recovery manager of the System R database manager," *Computing Surveys* **13**:2 (1981), pp. 223–242.

5. J. N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, San Francisco, 1993.

6. T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery — a taxonomy," *Computing Surveys* **15**:4 (1983), pp. 287–317.

7. V. Kumar and M. Hsu, *Recovery Mechanisms in Database Systems*, Prentice-Hall, Englewood Cliffs NJ, 1998.

8. C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. on Database Systems* **17**:1 (1992), pp. 94–162.

# Chapter 18

# Concurrency Control

Interactions among concurrently executing transactions can cause the database state to become inconsistent, even when the transactions individually preserve correctness of the state, and there is no system failure. Thus, the timing of individual steps of different transactions needs to be regulated in some manner. This regulation is the job of the *scheduler* component of the DBMS, and the general process of assuring that transactions preserve consistency when executing simultaneously is called *concurrency control.* The role of the scheduler is suggested by Fig. 18.1.

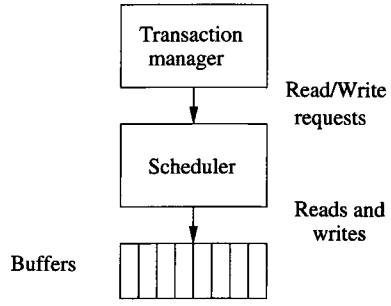


Figure 18.1: The scheduler takes read/write requests from transactions and either executes them in buffers or delays them

As transactions request reads and writes of database elements, these requests are passed to the scheduler. In most situations, the scheduler will execute the reads and writes directly, first calling on the buffer manager if the desired database element is not in a buffer. However, in some situations, it is not safe for the request to be executed immediately. The scheduler must delay the request; in some concurrency-control techniques, the scheduler may even abort the transaction that issued the request.

We begin by studying how to assure that concurrently executing transactions preserve correctness of the database state. The abstract requirement is called *serializability*, and there is an important, stronger condition called *conflict-serializability* that most schedulers actually enforce. We consider the most important techniques for implementing schedulers: locking, timestamping, and validation. Our study of lock-based schedulers includes the important concept of "two-phase locking," which is a requirement widely used to assure serializability of schedules.

# 18.1 Serial and Serializable Schedules

Recall the "correctness principle" from Section 17.1.3: every transaction, if executed in isolation (without any other transactions running concurrently), will transform any consistent state to another consistent state. In practice, transactions often run concurrently with other transactions, so the correctness principle doesn't apply directly. This section introduces the notion of "schedules," the sequence of actions performed by transactions and "serializable schedules," which produce the same result as if the transactions executed one-at-a-time.

## 18.1.1 Schedules

A *schedule* is a sequence of the important actions taken by one or more transactions. When studying concurrency control, the important read and write actions take place in the main-memory buffers, not the disk. That is, a database element $A$ that is brought to a buffer by some transaction $T$ may be read or written in that buffer not only by $T$ but by other transactions that access $A$.

| $T_1$ | $T_2$ |
|---|---|
| `READ(A,t)` | `READ(A,s)` |
| `t := t+100` | `s := s*2` |
| `WRITE(A,t)` | `WRITE(A,s)` |
| `READ(B,t)` | `READ(B,s)` |
| `t := t+100` | `s := s*2` |
| `WRITE(B,t)` | `WRITE(B,s)` |

Figure 18.2: Two transactions

**Example 18.1:** Let us consider two transactions and the effect on the database when their actions are executed in certain orders. The important actions of the transactions $T_1$ and $T_2$ are shown in Fig. 18.2. The variables $t$ and $s$ are local variables of $T_1$ and $T_2$, respectively; they are *not* database elements.

We shall assume that the only consistency constraint on the database state is that $A = B$. Since $T_1$ adds 100 to both $A$ and $B$, and $T_2$ multiplies both

$A$ and $B$ by 2, we know that each transaction, run in isolation, will preserve consistency. □

### 18.1.2 Serial Schedules

A schedule is *serial* if its actions consist of all the actions of one transaction, then all the actions of another transaction, and so on. No mixing of the actions is allowed.

| $T_1$ | $T_2$ | $A$ | $B$ |
|---|---|---|---|
| | | 25 | 25 |
| READ(A,t) | | | |
| t := t+100 | | | |
| WRITE(A,t) | | 125 | |
| READ(B,t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 125 |
| | READ(A,s) | | |
| | s := s*2 | | |
| | WRITE(A,s) | 250 | |
| | READ(B,s) | | |
| | s := s*2 | | |
| | WRITE(B,s) | | 250 |

Figure 18.3: Serial schedule in which $T_1$ precedes $T_2$

**Example 18.2:** For the transactions of Fig. 18.2, there are two serial schedules, one in which $T_1$ precedes $T_2$ and the other in which $T_2$ precedes $T_1$. Figure 18.3 shows the sequence of events when $T_1$ precedes $T_2$, and the initial state is $A = B = 25$. We shall take the convention that when displayed vertically, time proceeds down the page. Also, the values of $A$ and $B$ shown refer to their values in main-memory buffers, not necessarily to their values on disk.

Figure 18.4 shows another serial schedule in which $T_2$ precedes $T_1$; the initial state is again assumed to be $A = B = 25$. Notice that the final values of $A$ and $B$ are different for the two schedules; they both have value 250 when $T_1$ goes first and 150 when $T_2$ goes first. In general, we would not expect the final state of a database to be independent of the order of transactions. □

We can represent a serial schedule as in Fig. 18.3 or Fig. 18.4, listing each of the actions in the order they occur. However, since the order of actions in a serial schedule depends only on the order of the transactions themselves, we shall sometimes represent a serial schedule by the list of transactions. Thus, the schedule of Fig. 18.3 is represented $(T_1, T_2)$, and that of Fig. 18.4 is $(T_2, T_1)$.

| $T_1$ | $T_2$ | $A$ | $B$ |
|---|---|---|---|
| | | 25 | 25 |
| | READ(A,s) | | |
| | s := s*2 | | |
| | WRITE(A,s) | 50 | |
| | READ(B,s) | | |
| | s := s*2 | | |
| | WRITE(B,s) | | 50 |
| READ(A,t) | | | |
| t := t+100 | | | |
| WRITE(A,t) | | 150 | |
| READ(B,t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 150 |

Figure 18.4: Serial schedule in which $T_2$ precedes $T_1$

### 18.1.3 Serializable Schedules

The correctness principle for transactions tells us that every serial schedule will preserve consistency of the database state. But are there any other schedules that also are guaranteed to preserve consistency? There are, as the following example shows. In general, we say a schedule $S$ is *serializable* if there is a serial schedule $S'$ such that for every initial database state, the effects of $S$ and $S'$ are the same.

| $T_1$ | $T_2$ | $A$ | $B$ |
|---|---|---|---|
| | | 25 | 25 |
| READ(A,t) | | | |
| t := t+100 | | | |
| WRITE(A,t) | | 125 | |
| | READ(A,s) | | |
| | s := s*2 | | |
| | WRITE(A,s) | 250 | |
| READ(B,t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 125 |
| | READ(B,s) | | |
| | s := s*2 | | |
| | WRITE(B,s) | | 250 |

Figure 18.5: A serializable, but not serial, schedule

**Example 18.3:** Figure 18.5 shows a schedule of the transactions from Example 18.1 that is serializable but not serial. In this schedule, $T_2$ acts on $A$ after $T_1$ does, but before $T_1$ acts on $B$. However, we see that the effect of the two transactions scheduled in this manner is the same as for the serial schedule $(T_1, T_2)$ from Fig. 18.3. To convince ourselves of the truth of this statement, we must consider not only the effect from the database state $A = B = 25$, which we show in Fig. 18.5, but from any consistent database state. Since all consistent database states have $A = B = c$ for some constant $c$, it is not hard to deduce that in the schedule of Fig. 18.5, both $A$ and $B$ will be left with the value $2(c + 100)$, and thus consistency is preserved from any consistent state.

| $T_1$ | $T_2$ | $A$ | $B$ |
|---|---|---|---|
| | | 25 | 25 |
| READ(A,t) | | | |
| t := t+100 | | | |
| WRITE(A,t) | | 125 | |
| | READ(A,s) | | |
| | s := s*2 | | |
| | WRITE(A,s) | 250 | |
| | READ(B,s) | | |
| | s := s*2 | | |
| | WRITE(B,s) | | 50 |
| READ(B,t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 150 |

Figure 18.6: A nonserializable schedule

On the other hand, consider the schedule of Fig. 18.6, which is not serializable. The reason we can be sure it is not serializable is that it takes the consistent state $A = B = 25$ and leaves the database in an inconsistent state, where $A = 250$ and $B = 150$. Notice that in this order of actions, where $T_1$ operates on $A$ first, but $T_2$ operates on $B$ first, we have in effect applied different computations to $A$ and $B$, that is $A := 2(A + 100)$ versus $B := 2B + 100$. The schedule of Fig. 18.6 is the sort of behavior that concurrency control mechanisms must avoid. □

### 18.1.4 The Effect of Transaction Semantics

In our study of serializability so far, we have considered in detail the operations performed by the transactions, to determine whether or not a schedule is serializable. The details of the transactions do matter, as we can see from the following example.

| $T_1$ | $T_2$ | $A$ | $B$ |
|---|---|---|---|
| | | 25 | 25 |
| READ(A,t) | | | |
| t := t+100 | | | |
| WRITE(A,t) | | 125 | |
| | READ(A,s) | | |
| | s := s+200 | | |
| | WRITE(A,s) | 325 | |
| | READ(B,s) | | |
| | s := s+200 | | |
| | WRITE(B,s) | | 225 |
| READ(B,t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 325 |

Figure 18.7: A schedule that is serializable only because of the detailed behavior of the transactions

**Example 18.4:** Consider the schedule of Fig. 18.7, which differs from Fig. 18.6 only in the computation that $T_2$ performs. That is, instead of multiplying $A$ and $B$ by 2, $T_2$ adds 200 to each. One can easily check that regardless of the consistent initial state, the final state is the one that results from the serial schedule $(T_1, T_2)$. Coincidentally, it also results from the other serial schedule, $(T_2, T_1)$. □

Unfortunately, it is not realistic for the scheduler to concern itself with the details of computation undertaken by transactions. Since transactions often involve code written in a general-purpose programming language as well as SQL or other high-level-language statements, it is impossible to say for certain what a transaction is doing. However, the scheduler does get to see the read and write requests from the transactions, so it can know what database elements each transaction reads, and what elements it *might* change. To simplify the job of the scheduler, it is conventional to assume that:

- Any database element $A$ that a transaction $T$ writes is given a value that depends on the database state in such a way that no arithmetic coincidences occur.

An example of a "coincidence" is that in Example 18.4, where $A + 100 + 200 = B + 200 + 100$ whenever $A = B$, even though the two operations are carried out in different orders on the two variables. Put another way, if there is something that $T$ could do to a database element to make the database state inconsistent, then $T$ will do that.

### 18.1.5 A Notation for Transactions and Schedules

If we assume "no coincidences," then only the reads and writes performed by the transaction matter, not the actual values involved. Thus, we shall represent transactions and schedules by a shorthand notation, in which the actions are $r_T(X)$ and $w_T(X)$, meaning that transaction $T$ reads, or respectively writes, database element $X$. Moreover, since we shall usually name our transactions $T_1, T_2, \ldots$, we adopt the convention that $r_i(X)$ and $w_i(X)$ are synonyms for $r_{T_i}(X)$ and $w_{T_i}(X)$, respectively.

**Example 18.5:** The transactions of Fig. 18.2 can be written:

$$T_1\text{: } r_1(A);\ w_1(A);\ r_1(B);\ w_1(B);$$
$$T_2\text{: } r_2(A);\ w_2(A);\ r_2(B);\ w_2(B);$$

As another example,

$$r_1(A);\ w_1(A);\ r_2(A);\ w_2(A);\ r_1(B);\ w_1(B);\ r_2(B);\ w_2(B);$$

is the serializable schedule from Fig. 18.5. □

To make the notation precise:

1. An *action* is an expression of the form $r_i(X)$ or $w_i(X)$, meaning that transaction $T_i$ reads or writes, respectively, the database element $X$.
2. A *transaction* $T_i$ is a sequence of actions with subscript $i$.
3. A *schedule* $S$ of a set of transactions $\mathcal{T}$ is a sequence of actions, in which for each transaction $T_i$ in $\mathcal{T}$, the actions of $T_i$ appear in $S$ in the same order that they appear in the definition of $T_i$ itself. We say that $S$ is an *interleaving* of the actions of the transactions of which it is composed.

For instance, the schedule of Example 18.5 has all the actions with subscript 1 appearing in the same order that they have in the definition of $T_1$, and the actions with subscript 2 appear in the same order that they appear in the definition of $T_2$.

### 18.1.6 Exercises for Section 18.1

**Exercise 18.1.1:** A transaction $T_1$, executed by an airline-reservation system, performs the following steps:

*i.* The customer is queried for a desired flight time and cities. Information about the desired flights is located in database elements (perhaps disk blocks) $A$ and $B$, which the system retrieves from disk.

*ii.* The customer is told about the options, and selects a flight whose data, including the number of reservations for that flight is in $B$. A reservation on that flight is made for the customer.

*iii.* The customer selects a seat for the flight; seat data for the flight is in database element $C$.

*iv.* The system gets the customer's credit-card number and appends the bill for the flight to a list of bills in database element $D$.

*v.* The customer's phone and flight data is added to another list on database element $E$ for a fax to be sent confirming the flight.

Express transaction $T_1$ as a sequence of $r$ and $w$ actions.

! **Exercise 18.1.2:** If two transactions consist of 4 and 6 actions, respectively, how many interleavings of these transactions are there?

## 18.2 Conflict-Serializability

Schedulers in commercial systems generally enforce a condition, called "conflict-serializability," that is stronger than the general notion of serializability introduced in Section 18.1.3. It is based on the idea of a *conflict*: a pair of consecutive actions in a schedule such that, if their order is interchanged, then the behavior of at least one of the transactions involved can change.

### 18.2.1 Conflicts

To begin, let us observe that most pairs of actions do *not* conflict. In what follows, we assume that $T_i$ and $T_j$ are different transactions; i.e., $i \neq j$.

1. $r_i(X)$; $r_j(Y)$ is never a conflict, even if $X = Y$. The reason is that neither of these steps change the value of any database element.

2. $r_i(X)$; $w_j(Y)$ is not a conflict provided $X \neq Y$. The reason is that should $T_j$ write $Y$ before $T_i$ reads $X$, the value of $X$ is not changed. Also, the read of $X$ by $T_i$ has no effect on $T_j$, so it does not affect the value $T_j$ writes for $Y$.

3. $w_i(X)$; $r_j(Y)$ is not a conflict if $X \neq Y$, for the same reason as (2).

4. Similarly, $w_i(X)$; $w_j(Y)$ is not a conflict as long as $X \neq Y$.

On the other hand, there are three situations where we may not swap the order of actions:

a) Two actions of the same transaction, e.g., $r_i(X)$; $w_i(Y)$, always conflict. The reason is that the order of actions of a single transaction are fixed and may not be reordered.

b) Two writes of the same database element by different transactions conflict. That is, $w_i(X)$; $w_j(X)$ is a conflict. The reason is that as written, the value of $X$ remains afterward as whatever $T_j$ computed it to be. If we swap the order, as $w_j(X)$; $w_i(X)$, then we leave $X$ with the value computed by $T_i$. Our assumption of "no coincidences" tells us that the values written by $T_i$ and $T_j$ will be different, at least for some initial states of the database.

c) A read and a write of the same database element by different transactions also conflict. That is, $r_i(X)$; $w_j(X)$ is a conflict, and so is $w_i(X)$; $r_j(X)$. If we move $w_j(X)$ ahead of $r_i(X)$, then the value of $X$ read by $T_i$ will be that written by $T_j$, which we assume is not necessarily the same as the previous value of $X$. Thus, swapping the order of $r_i(X)$ and $w_j(X)$ affects the value $T_i$ reads for $X$ and could therefore affect what $T_i$ does.

The conclusion we draw is that any two actions of different transactions may be swapped unless:

1. They involve the same database element, and

2. At least one is a write.

Extending this idea, we may take any schedule and make as many nonconflicting swaps as we wish, with the goal of turning the schedule into a serial schedule. If we can do so, then the original schedule is serializable, because its effect on the database state remains the same as we perform each of the nonconflicting swaps.

We say that two schedules are *conflict-equivalent* if they can be turned one into the other by a sequence of nonconflicting swaps of adjacent actions. We shall call a schedule *conflict-serializable* if it is conflict-equivalent to a serial schedule. Note that conflict-serializability is a sufficient condition for serializability; i.e., a conflict-serializable schedule is a serializable schedule. Conflict-serializability is not required for a schedule to be serializable, but it is the condition that the schedulers in commercial systems generally use when they need to guarantee serializability.

**Example 18.6:** Consider the schedule

$$r_1(A);\ w_1(A);\ r_2(A);\ w_2(A);\ r_1(B);\ w_1(B);\ r_2(B);\ w_2(B);$$

from Example 18.5. We claim this schedule is conflict-serializable. Figure 18.8 shows the sequence of swaps in which this schedule is converted to the serial schedule $(T_1, T_2)$, where all of $T_1$'s actions precede all those of $T_2$. We have underlined the pair of adjacent actions about to be swapped at each step. □

$r_1(A)$; $w_1(A)$; $r_2(A)$; $\underline{w_2(A)}$; $\underline{r_1(B)}$; $w_1(B)$; $r_2(B)$; $w_2(B)$;
$r_1(A)$; $w_1(A)$; $\underline{r_2(A)}$; $\underline{r_1(B)}$; $\underline{w_2(A)}$; $w_1(B)$; $r_2(B)$; $w_2(B)$;
$r_1(A)$; $w_1(A)$; $r_1(B)$; $r_2(A)$; $\underline{w_2(A)}$; $\underline{w_1(B)}$; $r_2(B)$; $w_2(B)$;
$r_1(A)$; $w_1(A)$; $r_1(B)$; $\underline{r_2(A)}$; $\underline{w_1(B)}$; $w_2(A)$; $r_2(B)$; $w_2(B)$;
$r_1(A)$; $w_1(A)$; $r_1(B)$; $\underline{w_1(B)}$; $\underline{r_2(A)}$; $w_2(A)$; $r_2(B)$; $w_2(B)$;

Figure 18.8: Converting a conflict-serializable schedule to a serial schedule by swaps of adjacent actions

### 18.2.2 Precedence Graphs and a Test for Conflict-Serializability

It is relatively simple to examine a schedule $S$ and decide whether or not it is conflict-serializable. When a pair of conflicting actions appears anywhere in $S$, the transactions performing those actions must appear in the same order in any conflict-equivalent serial schedule as the actions appear in $S$. Thus, conflicting pairs of actions put constraints on the order of transactions in the hypothetical, conflict-equivalent serial schedule. If these constraints are not contradictory, we can find a conflict-equivalent serial schedule. If they are contradictory, we know that no such serial schedule exists.

Given a schedule $S$, involving transactions $T_1$ and $T_2$, perhaps among other transactions, we say that $T_1$ *takes precedence over* $T_2$, written $T_1 <_S T_2$, if there are actions $A_1$ of $T_1$ and $A_2$ of $T_2$, such that:

1. $A_1$ is ahead of $A_2$ in $S$,
2. Both $A_1$ and $A_2$ involve the same database element, and
3. At least one of $A_1$ and $A_2$ is a write action.

Notice that these are exactly the conditions under which we cannot swap the order of $A_1$ and $A_2$. Thus, $A_1$ will appear before $A_2$ in any schedule that is conflict-equivalent to $S$. As a result, a conflict-equivalent serial schedule must have $T_1$ before $T_2$.

We can summarize these precedences in a *precedence graph*. The nodes of the precedence graph are the transactions of a schedule $S$. When the transactions are $T_i$ for various $i$, we shall label the node for $T_i$ by only the integer $i$. There is an arc from node $i$ to node $j$ if $T_i <_S T_j$.

**Example 18.7:** The following schedule $S$ involves three transactions, $T_1$, $T_2$, and $T_3$.

$$S\text{: } r_2(A);\ r_1(B);\ w_2(A);\ r_3(A);\ w_1(B);\ w_3(A);\ r_2(B);\ w_2(B);$$

If we look at the actions involving $A$, we find several reasons why $T_2 <_S T_3$. For example, $r_2(A)$ comes ahead of $w_3(A)$ in $S$, and $w_2(A)$ comes ahead of both

## Why Conflict-Serializability is not Necessary for Serializability

Consider three transactions $T_1$, $T_2$, and $T_3$ that each write a value for $X$. $T_1$ and $T_2$ also write values for $Y$ before they write values for $X$. One possible schedule, which happens to be serial, is

$$S_1\text{: } w_1(Y);\ w_1(X);\ w_2(Y);\ w_2(X);\ w_3(X);$$

$S_1$ leaves $X$ with the value written by $T_3$ and $Y$ with the value written by $T_2$. However, so does the schedule

$$S_2\text{: } w_1(Y);\ w_2(Y);\ w_2(X);\ w_1(X);\ w_3(X);$$

Intuitively, the values of $X$ written by $T_1$ and $T_2$ have no effect, since $T_3$ overwrites their values. Thus, $X$ has the same value after either $S_1$ or $S_2$, and likewise $Y$ has the same value after either $S_1$ or $S_2$. Since $S_1$ is serial, and $S_2$ has the same effect as $S_1$ on any database state, we know that $S_2$ is serializable. However, since we cannot swap $w_1(Y)$ with $w_2(Y)$, and we cannot swap $w_1(X)$ with $w_2(X)$, therefore we cannot convert $S_2$ to any serial schedule by swaps. That is, $S_2$ is serializable, but not conflict-serializable.

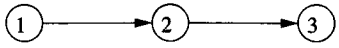


Figure 18.9: The precedence graph for the schedule $S$ of Example 18.7

$r_3(A)$ and $w_3(A)$. Any one of these three observations is sufficient to justify the arc in the precedence graph of Fig. 18.9 from 2 to 3.

Similarly, if we look at the actions involving $B$, we find that there are several reasons why $T_1 <_S T_2$. For instance, the action $r_1(B)$ comes before $w_2(B)$. Thus, the precedence graph for $S$ also has an arc from 1 to 2. However, these are the only arcs we can justify from the order of actions in schedule $S$. □

To tell whether a schedule $S$ is conflict-serializable, construct the precedence graph for $S$ and ask if there are any cycles. If so, then $S$ is not conflict-serializable. But if the graph is acyclic, then $S$ is conflict-serializable, and moreover, any topological order of the nodes[1] is a conflict-equivalent serial order.

[1] A *topological order* of an acyclic graph is any order of the nodes such that for every arc $a \to b$, node $a$ precedes node $b$ in the topological order. We can find a topological order for any acyclic graph by repeatedly removing nodes that have no predecessors among the remaining nodes.

**Example 18.8:** Figure 18.9 is acyclic, so the schedule $S$ of Example 18.7 is conflict-serializable. There is only one order of the nodes or transactions consistent with the arcs of that graph: $(T_1, T_2, T_3)$. Notice that it is indeed possible to convert $S$ into the schedule in which all actions of each of the three transactions occur in this order; this serial schedule is:

$$S'\text{: } r_1(B);\ w_1(B);\ r_2(A);\ w_2(A);\ r_2(B);\ w_2(B);\ r_3(A);\ w_3(A);$$

To see that we can get from $S$ to $S'$ by swaps of adjacent elements, first notice we can move $r_1(B)$ ahead of $r_2(A)$ without conflict. Then, by three swaps we can move $w_1(B)$ just after $r_1(B)$, because each of the intervening actions involves $A$ and not $B$. We can then move $r_2(B)$ and $w_2(B)$ to a position just after $w_2(A)$, moving through only actions involving $A$; the result is $S'$. □

**Example 18.9:** Consider the schedule

$$S_1\text{: } r_2(A);\ r_1(B);\ w_2(A);\ r_2(B);\ r_3(A);\ w_1(B);\ w_3(A);\ w_2(B);$$

which differs from $S$ only in that action $r_2(B)$ has been moved forward three positions. Examination of the actions involving $A$ still give us only the precedence $T_2 <_{S_1} T_3$. However, when we examine $B$ we get not only $T_1 <_{S_1} T_2$ [because $r_1(B)$ and $w_1(B)$ appear before $w_2(B)$], but also $T_2 <_{S_1} T_1$ [because $r_2(B)$ appears before $w_1(B)$]. Thus, we have the precedence graph of Fig. 18.10 for schedule $S_1$.

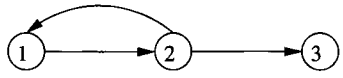


Figure 18.10: A cyclic precedence graph; its schedule is not conflict-serializable

This graph evidently has a cycle. We conclude that $S_1$ is not conflict-serializable. Intuitively, any conflict-equivalent serial schedule would have to have $T_1$ both ahead of and behind $T_2$, so therefore no such schedule exists. □

### 18.2.3 Why the Precedence-Graph Test Works

If there is a cycle involving $n$ transactions $T_1 \to T_2 \to \ldots \to T_n \to T_1$, then in the hypothetical serial order, the actions of $T_1$ must precede those of $T_2$, which precede those of $T_3$, and so on, up to $T_n$. But the actions of $T_n$, which therefore come after those of $T_1$, are also required to precede those of $T_1$ because of the arc $T_n \to T_1$. Thus, if there is a cycle in the precedence graph, then the schedule is not conflict-serializable.

The converse is a bit harder. We must show that if the precedence graph has no cycles, then we can reorder the schedule's actions using legal swaps of adjacent actions, until the schedule becomes a serial schedule. If we can do so, then we have our proof that every schedule with an acyclic precedence graph is conflict-serializable. The proof is an induction on the number of transactions involved in the schedule.

**BASIS**: If $n = 1$, i.e., there is only one transaction in the schedule, then the schedule is already serial, and therefore surely conflict-serializable.

**INDUCTION**: Let the schedule $S$ consist of the actions of $n$ transactions

$$T_1, T_2, \ldots, T_n$$

We suppose that $S$ has an acyclic precedence graph. If a finite graph is acyclic, then there is at least one node that has no arcs in; let the node $i$ corresponding to transaction $T_i$ be such a node. Since there are no arcs into node $i$, there can be no action $A$ in $S$ that:

1. Involves any transaction $T_j$ other than $T_i$,
2. Precedes some action of $T_i$, and
3. Conflicts with that action.

For if there were, we should have put an arc from node $j$ to node $i$ in the precedence graph.

It is thus possible to swap all the actions of $T_i$, keeping them in order, but moving them to the front of $S$. The schedule has now taken the form

$$(\text{Actions of } T_i)(\text{Actions of the other } n-1 \text{ transactions})$$

Let us now consider the tail of $S$ — the actions of all transactions other than $T_i$. Since these actions maintain the same relative order that they did in $S$, the precedence graph for the tail is the same as the precedence graph for $S$, except that the node for $T_i$ and any arcs out of that node are missing.

Since the original precedence graph was acyclic, and deleting nodes and arcs cannot make it cyclic, we conclude that the tail's precedence graph is acyclic. Moreover, since the tail involves $n - 1$ transactions, the inductive hypothesis applies to it. Thus, we know we can reorder the actions of the tail using legal swaps of adjacent actions to turn it into a serial schedule. Now, $S$ itself has been turned into a serial schedule, with the actions of $T_i$ first and the actions of the other transactions following in some serial order. The induction is complete, and we conclude that every schedule with an acyclic precedence graph is conflict-serializable.

### 18.2.4 Exercises for Section 18.2

**Exercise 18.2.1**: Below are two transactions, described in terms of their effect on two database elements $A$ and $B$, which we may assume are integers.

```
T1: READ(A,t); t:=t+2; WRITE(A,t); READ(B,t); t:=t*3; WRITE(B,t);
T2: READ(B,s); s:=s*2; WRITE(B,s); READ(A,s); s:=s+3; WRITE(A,s);
```

We assume that, whatever consistency constraints there are on the database, these transactions preserve them in isolation. Note that $A = B$ is *not* the consistency constraint.

a) It turns out that both serial orders have the same effect on the database; that is, $(T_1, T_2)$ and $(T_2, T_1)$ are equivalent. Demonstrate this fact by showing the effect of the two transactions on an arbitrary initial database state.

b) Give examples of a serializable schedule and a nonserializable schedule of the 12 actions above.

c) How many serial schedules of the 12 actions are there?

!! d) How many serializable schedules of the 12 actions are there?

**Exercise 18.2.2:** The two transactions of Exercise 18.2.1 can be written in our notation that shows read- and write-actions only, as:

$$T_1\text{: } r_1(A);\ w_1(A);\ r_1(B);\ w_1(B);$$
$$T_2\text{: } r_2(B);\ w_2(B);\ r_2(A);\ w_2(A);$$

Answer the following:

! a) Among the possible schedules of the eight actions above, how many are conflict-equivalent to the serial order $(T_1, T_2)$?

b) How many schedules of the eight actions are equivalent to the serial order $(T_2, T_1)$?

!! c) How many schedules of the eight actions are equivalent (not necessarily conflict-equivalent) to the serial schedule $(T_1, T_2)$, assuming the transactions have the effect on the database described in Exercise 18.2.1?

! d) Why are the answers to (c) above and Exercise 18.2.1(d) different?

! **Exercise 18.2.3:** Suppose the transactions of Exercise 18.2.2 are changed to be:

$$T_1\text{: } r_1(A);\ w_1(A);\ r_1(B);\ w_1(B);$$
$$T_2\text{: } r_2(A);\ w_2(A);\ r_2(B);\ w_2(B);$$

That is, the transactions retain their semantics from Exercise 18.2.1, but $T_2$ has been changed so $A$ is processed before $B$. Give:

a) The number of conflict-serializable schedules.

b) The number of serializable schedules, assuming the transactions have the same effect on the database state as in Exercise 18.2.1.

**Exercise 18.2.4:** For each of the following schedules:

a) $r_1(A)$; $r_2(A)$; $r_3(B)$; $w_1(A)$; $r_2(C)$; $r_2(B)$; $w_2(B)$; $w_1(C)$;

b) $r_1(A)$; $w_1(B)$; $r_2(B)$: $w_2(C)$; $r_3(C)$; $w_3(A)$;

c) $w_3(A)$; $r_1(A)$; $w_1(B)$; $r_2(B)$: $w_2(C)$; $r_3(C)$;

d) $r_1(A)$; $r_2(A)$; $w_1(B)$; $w_2(B)$; $r_1(B)$; $r_2(B)$; $w_2(C)$; $w_1(D)$;

e) $r_1(A)$; $r_2(A)$; $r_1(B)$; $r_2(B)$; $r_3(A)$; $r_4(B)$; $w_1(A)$; $w_2(B)$;

Answer the following questions:

*i.* What is the precedence graph for the schedule?

*ii.* Is the schedule conflict-serializable? If so, what are all the equivalent serial schedules?

! *iii.* Are there any serial schedules that must be equivalent (regardless of what the transactions do to the data), but are not conflict-equivalent?

!! **Exercise 18.2.5:** Say that a transaction $T$ *precedes* a transaction $U$ in a schedule $S$ if every action of $T$ precedes every action of $U$ in $S$. Note that if $T$ and $U$ are the only transactions in $S$, then saying $T$ precedes $U$ is the same as saying that $S$ is the serial schedule $(T, U)$. However, if $S$ involves transactions other than $T$ and $U$, then $S$ might not be serializable, and in fact, because of the effect of other transactions, $S$ might not even be conflict-serializable. Give an example of a schedule $S$ such that:

*i.* In $S$, $T_1$ precedes $T_2$, and

*ii.* $S$ is conflict-serializable, but

*iii.* In every serial schedule conflict-equivalent to $S$, $T_2$ precedes $T_1$.

! **Exercise 18.2.6:** Explain how, for any $n > 1$, one can find a schedule whose precedence graph has a cycle of length $n$, but no smaller cycle.

## 18.3 Enforcing Serializability by Locks

In this section we consider the most common architecture for a scheduler, one in which "locks" are maintained on database elements to prevent unserializable behavior. Intuitively, a transaction obtains locks on the database elements it accesses to prevent other transactions from accessing these elements at roughly the same time and thereby incurring the risk of unserializability.

In this section, we introduce the concept of locking with an (overly) simple locking scheme. In this scheme, there is only one kind of lock, which transactions must obtain on a database element if they want to perform any operation whatsoever on that element. In Section 18.4, we shall learn more realistic locking schemes, with several kinds of lock, including the common shared/exclusive locks that correspond to the privileges of reading and writing, respectively.

### 18.3.1 Locks

In Fig. 18.11 we see a scheduler that uses a lock table to help perform its job. Recall from the chapter introduction that the responsibility of the scheduler is to take requests from transactions and either allow them to operate on the database or block the transaction until such time as it is safe to allow it to continue. A lock table will be used to guide this decision in a manner that we shall discuss at length.

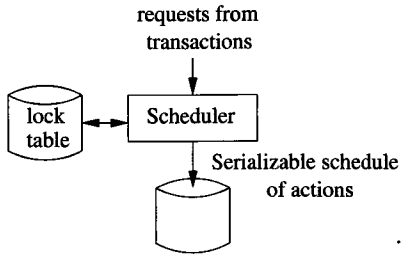


Figure 18.11: A scheduler that uses a lock table to guide decisions

Ideally, a scheduler would forward a request if and only if its execution cannot possibly lead to an inconsistent database state after all active transactions commit or abort. A locking scheduler, like most types of scheduler, instead enforces conflict-serializability, which as we learned is a more stringent condition than correctness, or even than serializability.

When a scheduler uses locks, transactions must request and release locks, in addition to reading and writing database elements. The use of locks must be proper in two senses, one applying to the structure of transactions, and the other to the structure of schedules.

- *Consistency of Transactions*: Actions and locks must relate in the expected ways:
  1. A transaction can only read or write an element if it previously was granted a lock on that element and hasn't yet released the lock.
  2. If a transaction locks an element, it must later unlock that element.
- *Legality of Schedules*: Locks must have their intended meaning: no two transactions may have locked the same element without one having first released the lock.

We shall extend our notation for actions to include locking and unlocking actions:

$l_i(X)$: Transaction $T_i$ requests a lock on database element $X$.

$u_i(X)$: Transaction $T_i$ releases ("unlocks") its lock on database element $X$.

Thus, the consistency condition for transactions can be stated as: "Whenever a transaction $T_i$ has an action $r_i(X)$ or $w_i(X)$, then there is a previous action $l_i(X)$ with no intervening action $u_i(X)$, and there is a subsequent $u_i(X)$." The legality of schedules is stated: "If there are actions $l_i(X)$ followed by $l_j(X)$ in a schedule, then somewhere between these actions there must be an action $u_i(X)$."

**Example 18.10:** Let us consider the two transactions $T_1$ and $T_2$ that we introduced in Example 18.1. Recall that $T_1$ adds 100 to database elements $A$ and $B$, while $T_2$ doubles them. Here are specifications for these transactions, in which we have included lock actions as well as arithmetic actions to help us remember what the transactions are doing.[2]

$T_1$: $l_1(A)$; $r_1(A)$; `A := A+100`; $w_1(A)$; $u_1(A)$; $l_1(B)$; $r_1(B)$; `B := B+100`; $w_1(B)$; $u_1(B)$;

$T_2$: $l_2(A)$; $r_2(A)$; `A := A*2`; $w_2(A)$; $u_2(A)$; $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $u_2(B)$;

Each of these transactions is consistent. They each release the locks on $A$ and $B$ that they take. Moreover, they each operate on $A$ and $B$ only at steps where they have previously requested a lock on that element and have not yet released the lock.

| $T_1$ | $T_2$ | $A$ | $B$ |
|---|---|---|---|
| | | 25 | 25 |
| $l_1(A)$; $r_1(A)$; | | | |
| `A := A+100;` | | | |
| $w_1(A)$; $u_1(A)$; | | 125 | |
| | $l_2(A)$; $r_2(A)$; | | |
| | `A := A*2;` | | |
| | $w_2(A)$; $u_2(A)$; | 250 | |
| | $l_2(B)$; $r_2(B)$; | | |
| | `B := B*2;` | | |
| | $w_2(B)$; $u_2(B)$; | | 50 |
| $l_1(B)$; $r_1(B)$; | | | |
| `B := B+100;` | | | |
| $w_1(B)$; $u_1(B)$; | | | 150 |

Figure 18.12: A legal schedule of consistent transactions; unfortunately it is not serializable

Figure 18.12 shows one legal schedule of these two transactions. To save space we have put several actions on one line. The schedule is legal because

[2]Remember that the actual computations of the transaction usually are not represented in our current notation, since they are not considered by the scheduler when deciding whether to grant or deny transaction requests.

the two transactions never hold a lock on $A$ at the same time, and likewise for $B$. Specifically, $T_2$ does not execute $l_2(A)$ until after $T_1$ executes $u_1(A)$, and $T_1$ does not execute $l_1(B)$ until after $T_2$ executes $u_2(B)$. As we see from the trace of the values computed, the schedule, although legal, is not serializable. We shall see in Section 18.3.3 the additional condition, "two-phase locking," that we need to assure that legal schedules are conflict-serializable. □

### 18.3.2 The Locking Scheduler

It is the job of a scheduler based on locking to grant requests if and only if the request will result in a legal schedule. If a request is not granted, the requesting transaction is delayed; it waits until the scheduler grants its request at a later time. To aid its decisions, the scheduler has a *lock table* that tells, for every database element, the transaction (if any) that currently holds a lock on that element. We shall discuss the structure of a lock table in more detail in Section 18.5.2. However, when there is only one kind of lock, as we have assumed so far, the table may be thought of as a relation `Locks(element, transaction)`, consisting of pairs $(X,T)$ such that transaction $T$ currently has a lock on database element $X$. The scheduler has only to query and modify this relation.

**Example 18.11:** The schedule of Fig. 18.12 is legal, as we mentioned, so the locking scheduler would grant every request in the order of arrival shown. However, sometimes it is not possible to grant requests. Here are $T_1$ and $T_2$ from Example 18.10, with simple but important changes, in which $T_1$ and $T_2$ each lock $B$ before releasing the lock on $A$.

$T_1$: $l_1(A)$; $r_1(A)$; `A := A+100`; $w_1(A)$; $l_1(B)$; $u_1(A)$; $r_1(B)$; `B := B+100`; $w_1(B)$; $u_1(B)$;

$T_2$: $l_2(A)$; $r_2(A)$; `A := A*2`; $w_2(A)$; $l_2(B)$; $u_2(A)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $u_2(B)$;

In Fig. 18.13, when $T_2$ requests a lock on $B$, the scheduler must deny the lock, because $T_1$ still holds a lock on $B$. Thus, $T_2$ is delayed, and the next actions are from $T_1$. Eventually, $T_1$ executes $u_1(B)$, which unlocks $B$. Now, $T_2$ can get its lock on $B$, which is executed at the next step. Notice that because $T_2$ was forced to wait, it wound up multiplying $B$ by 2 after $T_1$ added 100, resulting in a consistent database state. □

### 18.3.3 Two-Phase Locking

There is a surprising condition, called *two-phase locking* (or *2PL*) under which we can guarantee that a legal schedule of consistent transactions is conflict-serializable:

- In every transaction, all lock actions precede all unlock actions.

| $T_1$ | $T_2$ | $A$ | $B$ |
|---|---|---|---|
| | | 25 | 25 |
| $l_1(A)$; $r_1(A)$; | | | |
| `A := A+100;` | | | |
| $w_1(A)$; $l_1(B)$; $u_1(A)$; | | 125 | |
| | $l_2(A)$; $r_2(A)$; | | |
| | `A := A*2;` | | |
| | $w_2(A)$; | 250 | |
| | $l_2(B)$ **Denied** | | |
| $r_1(B)$; `B := B+100;` | | | |
| $w_1(B)$; $u_1(B)$; | | | 125 |
| | $l_2(B)$; $u_2(A)$; $r_2(B)$; | | |
| | `B := B*2;` | | |
| | $w_2(B)$; $u_2(B)$; | | 250 |

Figure 18.13: The locking scheduler delays requests that would result in an illegal schedule

The "two phases" referred to by 2PL are thus the first phase, where locks are obtained, and the second phase, where locks are relinquished. Two-phase locking is a condition, like consistency, on the order of actions in a transaction. A transaction that obeys the 2PL condition is said to be a *two-phase-locked transaction*, or 2PL transaction.

**Example 18.12:** In Example 18.10, the transactions do not obey the two-phase locking rule. For instance, $T_1$ unlocks $A$ before it locks $B$. However, the versions of the transactions found in Example 18.11 *do* obey the 2PL condition. Notice that $T_1$ locks both $A$ and $B$ within the first five actions and unlocks them within the next five actions; $T_2$ behaves similarly. If we compare Figs. 18.12 and 18.13, we see how the 2PL transactions interact properly with the scheduler to assure consistency, while the non-2PL transactions allow non-conflict-serializable behavior. □

### 18.3.4 Why Two-Phase Locking Works

Intuitively, each two-phase-locked transaction may be thought to execute in its entirety at the instant it issues its first unlock request, as suggested by Fig. 18.14. Thus, there is always at least one conflict-equivalent serial schedule for a schedule $S$ of 2PL transactions: the one in which the transactions appear in the same order as their first unlocks.

We shall show how to convert any legal schedule $S$ of consistent, two-phase-locked transactions to a conflict-equivalent serial schedule. The conversion is best described as an induction on $n$, the number of transactions in $S$. In what follows, it is important to remember that the issue of conflict-equivalence refers

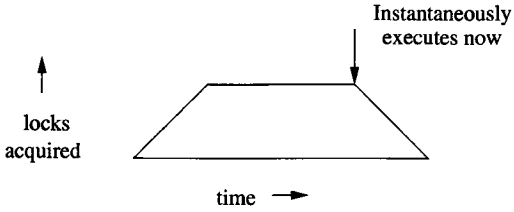


Figure 18.14: Every two-phase-locked transaction has a point at which it may be thought to execute instantaneously

to the read and write actions only. As we swap the order of reads and writes, we ignore the lock and unlock actions. Once we have the read and write actions ordered serially, we can place the lock and unlock actions around them as the various transactions require. Since each transaction releases all locks before its end, we know that the serial schedule is legal.

**BASIS**: If $n = 1$, there is nothing to do; $S$ is already a serial schedule.

**INDUCTION**: Suppose $S$ involves $n$ transactions $T_1, T_2, \ldots, T_n$, and let $T_i$ be the transaction with the first unlock action in the entire schedule $S$, say $u_i(X)$. We claim it is possible to move all the read and write actions of $T_i$ forward to the beginning of the schedule without passing any conflicting reads or writes.

Consider some action of $T_i$, say $w_i(Y)$. Could it be preceded in $S$ by some conflicting action, say $w_j(Y)$? If so, then in schedule $S$, actions $u_j(Y)$ and $l_i(Y)$ must intervene, in a sequence of actions

$$\cdots w_j(Y); \cdots ; \ u_j(Y); \cdots ; \ l_i(Y); \cdots ; \ w_i(Y); \cdots$$

Since $T_i$ is the first to unlock, $u_i(X)$ precedes $u_j(Y)$ in $S$; that is, $S$ might look like:

$$\cdots ; \ w_j(Y); \cdots ; \ u_i(X); \cdots ; \ u_j(Y); \cdots ; \ l_i(Y); \cdots ; \ w_i(Y); \cdots$$

or $u_i(X)$ could even appear before $w_j(Y)$. In any case, $u_i(X)$ appears before $l_i(Y)$, which means that $T_i$ is *not* two-phase-locked, as we assumed. While we have only argued the nonexistence of conflicting pairs of writes, the same argument applies to any pair of potentially conflicting actions, one from $T_i$ and the other from another $T_j$.

We conclude that it is indeed possible to move all the actions of $T_i$ forward to the beginning of $S$, using swaps of nonconflicting read and write actions, followed by restoration of the lock and unlock actions of $T_i$. That is, $S$ can be written in the form

$$(\text{Actions of } T_i)(\text{Actions of the other } n-1 \text{ transactions})$$

The tail of $n - 1$ transactions is still a legal schedule of consistent, 2PL transactions, so the inductive hypothesis applies to it. We convert the tail to a

### A Risk of Deadlock

One problem that is not solved by two-phase locking is the potential for deadlocks, where several transactions are forced by the scheduler to wait forever for a lock held by another transaction. For instance, consider the 2PL transactions from Example 18.11, but with $T_2$ changed to work on $B$ first:

$T_1$: $l_1(A)$; $r_1(A)$; `A := A+100`; $w_1(A)$; $l_1(B)$; $u_1(A)$; $r_1(B)$; `B := B+100`; $w_1(B)$; $u_1(B)$;

$T_2$: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $l_2(A)$; $u_2(B)$; $r_2(A)$; `A := A*2`; $w_2(A)$; $u_2(A)$;

A possible interleaving of the actions of these transactions is:

| $T_1$ | $T_2$ | $A$ | $B$ |
|---|---|---|---|
| | | 25 | 25 |
| $l_1(A)$; $r_1(A)$; | | | |
| | $l_2(B)$; $r_2(B)$; | | |
| `A := A+100;` | | | |
| | `B := B*2;` | | |
| $w_1(A)$; | | 125 | |
| | $w_2(B)$; | | 50 |
| $l_1(B)$ **Denied** | $l_2(A)$ **Denied** | | |

Now, neither transaction can proceed, and they wait forever. In Section 19.2, we shall discuss methods to remedy this situation. However, observe that it is not possible to allow both transactions to proceed, since if we do so the final database state cannot possibly have $A = B$.

conflict-equivalent serial schedule, and now all of $S$ has been shown conflict-serializable.

## 18.3.5 Exercises for Section 18.3

**Exercise 18.3.1:** Below are two transactions, with lock requests and the semantics of the transactions indicated. Recall from Exercise 18.2.1 that these transactions have the unusual property that they can be scheduled in ways that are not conflict-serializable, but, because of the semantics, are serializable.

$T_1$: $l_1(A)$; $r_1(A)$; `A := A+2`; $w_1(A)$; $u_1(A)$; $l_1(B)$; $r_1(B)$; `B := B*3`; $w_1(B)$; $u_1(B)$;

$T_2$: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $u_2(B)$; $l_2(A)$; $r_2(A)$; `A := A+3`; $w_2(A)$; $u_2(A)$;

In the questions below, consider only schedules of the read and write actions, not the lock, unlock, or assignment steps.

a) Give an example of a schedule that is prohibited by the locks.

! b) Of the $\binom{8}{4} = 70$ orders of the eight read and write actions, how many are legal schedules (i.e., they are permitted by the locks)?

! c) Of the legal schedules, how many are serializable (according to the semantics of the transactions given)?

! d) Of those schedules that are legal and serializable, how many are conflict-serializable?

!! e) Since $T_1$ and $T_2$ are not two-phase-locked, we would expect that some nonserializable behaviors would occur. Are there any legal schedules that are unserializable? If so, give an example, and if not, explain why.

! **Exercise 18.3.2:** Here are the transactions of Exercise 18.3.1, with all unlocks moved to the end so they are two-phase-locked.

$T_1$: $l_1(A)$; $r_1(A)$; `A := A+2`; $w_1(A)$; $l_1(B)$; $r_1(B)$; `B := B*3`; $w_1(B)$; $u_1(A)$; $u_1(B)$;

$T_2$: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $l_2(A)$; $r_2(A)$; `A := A+3`; $w_2(A)$; $u_2(B)$; $u_2(A)$;

How many legal schedules of all the read and write actions of these transactions are there?

**Exercise 18.3.3:** For each of the schedules of Exercise 18.2.4, assume that each transaction takes a lock on each database element immediately before it reads or writes the element, and that each transaction releases its locks immediately after the last time it accesses an element. Tell what the locking scheduler would do with each of these schedules; i.e., what requests would get delayed, and when would they be allowed to resume?

! **Exercise 18.3.4:** For each of the transactions described below, suppose that we insert one lock and one unlock action for each database element that is accessed.

a) $r_1(A)$; $w_1(B)$;

b) $r_2(A)$; $w_2(A)$; $w_2(B)$;

Tell how many orders of the lock, unlock, read, and write actions are:

*i*. Consistent and two-phase locked.

*ii*. Consistent, but not two-phase locked.

*iii*. Inconsistent, but two-phase locked.

*iv*. Neither consistent nor two-phase locked.

## 18.4 Locking Systems With Several Lock Modes

The locking scheme of Section 18.3 illustrates the important ideas behind locking, but it is too simple to be a practical scheme. The main problem is that a transaction $T$ must take a lock on a database element $X$ even if it only wants to read $X$ and not write it. We cannot avoid taking the lock, because if we didn't, then another transaction might write a new value for $X$ while $T$ was active and cause unserializable behavior. On the other hand, there is no reason why several transactions could not read $X$ at the same time, as long as none is allowed to write $X$.

We are thus motivated to introduce the most common locking scheme, where there are two different kinds of locks, one for reading (called a "shared lock" or "read lock"), and one for writing (called an "exclusive lock" or "write lock"). We then examine an improved scheme where transactions are allowed to take a shared lock and "upgrade" it to an exclusive lock later. We also consider "increment locks," which treat specially write actions that increment a database element; the important distinction is that increment operations commute, while general writes do not. These examples lead us to the general notion of a lock scheme described by a "compatibility matrix" that indicates what locks on a database element may be granted when other locks are held.

### 18.4.1 Shared and Exclusive Locks

The lock we need for writing is "stronger" than the lock we need to read, since it must prevent both reads and writes. Let us therefore consider a locking scheduler that uses two different kinds of locks: ***shared locks*** and ***exclusive locks***. For any database element $X$ there can be either one exclusive lock on $X$, or no exclusive locks but any number of shared locks. If we want to write $X$, we need to have an exclusive lock on $X$, but if we wish only to read $X$ we may have either a shared or exclusive lock on $X$. If we want to read $X$ but not write it, it is better to take only a shared lock.

We shall use $sl_i(X)$ to mean "transaction $T_i$ requests a shared lock on database element $X$" and $xl_i(X)$ for "$T_i$ requests an exclusive lock on $X$." We continue to use $u_i(X)$ to mean that $T_i$ unlocks $X$; i.e., it relinquishes whatever lock(s) it has on $X$.

The three kinds of requirements — consistency and 2PL for transactions, and legality for schedules — each have their counterpart for a shared/exclusive lock system. We summarize these requirements here:

1. *Consistency of transactions*: A transaction may not write without holding an exclusive lock, and you may not read without holding some lock. More precisely, in any transaction $T_i$,

   (a) A read action $r_i(X)$ must be preceded by $sl_i(X)$ or $xl_i(X)$, with no intervening $u_i(X)$.

   (b) A write action $w_i(X)$ must be preceded by $xl_i(X)$, with no intervening $u_i(X)$.

   All locks must be followed by an unlock of the same element.

2. *Two-phase locking of transactions*: Locking must precede unlocking. To be more precise, in any two-phase locked transaction $T_i$, no action $sl_i(X)$ or $xl_i(X)$ can be preceded by an action $u_i(Y)$, for any $Y$.

3. *Legality of schedules*: An element may either be locked exclusively by one transaction or by several in shared mode, but not both. More precisely:

   (a) If $xl_i(X)$ appears in a schedule, then there cannot be a following $xl_j(X)$ or $sl_j(X)$, for some $j$ other than $i$, without an intervening $u_i(X)$.

   (b) If $sl_i(X)$ appears in a schedule, then there cannot be a following $xl_j(X)$, for $j \neq i$, without an intervening $u_i(X)$.

   Note that we *do* allow one transaction to request and hold both shared and exclusive locks on the same element, provided its doing so does not conflict with the lock(s) of other transactions. If transactions know in advance their needs for locks, then only the exclusive lock would have to be requested, but if lock needs are unpredictable, then it is possible that one transaction would request both shared and exclusive locks at different times.

**Example 18.13:** Let us examine a possible schedule of the following two transactions, using shared and exclusive locks:

$$T_1\text{: } sl_1(A);\ r_1(A);\ xl_1(B);\ r_1(B);\ w_1(B);\ u_1(A);\ u_1(B);$$
$$T_2\text{: } sl_2(A);\ r_2(A);\ sl_2(B);\ r_2(B);\ u_2(A);\ u_2(B);$$

Both $T_1$ and $T_2$ read $A$ and $B$, but only $T_1$ writes $B$. Neither writes $A$.

In Fig. 18.15 is an interleaving of the actions of $T_1$ and $T_2$ in which $T_1$ begins by getting a shared lock on $A$. Then, $T_2$ follows by getting shared locks on both $A$ and $B$. Now, $T_1$ needs an exclusive lock on $B$, since it will both read and write $B$. However, it cannot get the exclusive lock because $T_2$ already has a shared lock on $B$. Thus, the scheduler forces $T_1$ to wait. Eventually, $T_2$ releases the lock on $B$. At that time, $T_1$ may complete. □

| $T_1$ | $T_2$ |
|---|---|
| $sl_1(A)$; $r_1(A)$; | |
| | $sl_2(A)$; $r_2(A)$; |
| | $sl_2(B)$; $r_2(B)$; |
| $xl_1(B)$ **Denied** | |
| | $u_2(A)$; $u_2(B)$ |
| $xl_1(B)$; $r_1(B)$; $w_1(B)$; | |
| $u_1(A)$; $u_1(B)$; | |

Figure 18.15: A schedule using shared and exclusive locks

Notice that the resulting schedule in Fig 18.15 is conflict-serializable. The conflict-equivalent serial order is $(T_2, T_1)$, even though $T_1$ started first. The argument we gave in Section 18.3.4 to show that legal schedules of consistent, 2PL transactions are conflict-serializable applies to systems with shared and exclusive locks as well. In Fig. 18.15, $T_2$ unlocks before $T_1$, so we would expect $T_2$ to precede $T_1$ in the serial order.

## 18.4.2 Compatibility Matrices

If we use several lock modes, then the scheduler needs a policy about when it can grant a lock request, given the other locks that may already be held on the same database element. A *compatibility matrix* is a convenient way to describe lock-management policies. It has a row and column for each lock mode. The rows correspond to a lock that is already held on an element $X$ by another transaction, and the columns correspond to the mode of a lock on $X$ that is requested. The rule for using a compatibility matrix for lock-granting decisions is:

- We can grant the lock on $X$ in mode $C$ if and only if for every row $R$ such that there is already a lock on $X$ in mode $R$ by some other transaction, there is a "Yes" in column $C$.

| | | Lock requested | |
|---|---|---|---|
| | | S | X |
| Lock held | S | Yes | No |
| in mode | X | No | No |

Figure 18.16: The compatibility matrix for shared and exclusive locks

**Example 18.14:** Figure 18.16 is the compatibility matrix for shared (S) and exclusive (X) locks. The column for $S$ says that we can grant a shared lock on

an element if the only locks held on that element currently are shared locks. The column for $X$ says that we can grant an exclusive lock only if there are no other locks held currently. □

### 18.4.3 Upgrading Locks

A transaction $T$ that takes a shared lock on $X$ is being "friendly" toward other transactions, since they are allowed to read $X$ at the same time $T$ is. Thus, we might wonder whether it would be friendlier still if a transaction $T$ that wants to read and write a new value of $X$ were first to take a shared lock on $X$, and only later, when $T$ was ready to write the new value, *upgrade* the lock to exclusive (i.e., request an exclusive lock on $X$ in addition to its already held shared lock on $X$). There is nothing that prevents a transaction from issuing requests for locks on the same database element in different modes. We adopt the convention that $u_i(X)$ releases all locks on $X$ held by transaction $T_i$, although we could introduce mode-specific unlock actions if there were a use for them.

**Example 18.15:** In the following example, transaction $T_1$ is able to perform its computation concurrently with $T_2$, which would not be possible had $T_1$ taken an exclusive lock on $B$ initially. The two transactions are:

$T_1$: $sl_1(A)$; $r_1(A)$; $sl_1(B)$; $r_1(B)$; $xl_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;
$T_2$: $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$; $u_2(A)$; $u_2(B)$;

Here, $T_1$ reads $A$ and $B$ and performs some (possibly lengthy) calculation with them, eventually using the result to write a new value of $B$. Notice that $T_1$ takes a shared lock on $B$ first, and later, after its calculation involving $A$ and $B$ is finished, requests an exclusive lock on $B$. Transaction $T_2$ only reads $A$ and $B$, and does not write.

| $T_1$ | $T_2$ |
|---|---|
| $sl_1(A)$; $r_1(A)$; | |
| | $sl_2(A)$; $r_2(A)$; |
| | $sl_2(B)$; $r_2(B)$; |
| $sl_1(B)$; $r_1(B)$; | |
| $xl_1(B)$ **Denied** | |
| | $u_2(A)$; $u_2(B)$ |
| $xl_1(B)$; $w_1(B)$; | |
| $u_1(A)$; $u_2(B)$; | |

Figure 18.17: Upgrading locks allows more concurrent operation

Figure 18.17 shows a possible schedule of actions. $T_2$ gets a shared lock on $B$ before $T_1$ does, but on the fourth line, $T_1$ is also able to lock $B$ in shared

mode. Thus, $T_1$ has both $A$ and $B$ and can perform its computation using their values. It is not until $T_1$ tries to upgrade its lock on $B$ to exclusive that the scheduler must deny the request and force $T_1$ to wait until $T_2$ releases its lock on $B$. At that time, $T_1$ gets its exclusive lock on $B$, writes $B$, and finishes.

Notice that had $T_1$ asked for an exclusive lock on $B$ initially, before reading $B$, then the request would have been denied, because $T_2$ already had a shared lock on $B$. $T_1$ could not perform its computation without reading $B$, and so $T_1$ would have more to do after $T_2$ releases its locks. As a result, $T_1$ finishes later using only an exclusive lock on $B$ than it would if it used the upgrading strategy. □

**Example 18.16:** Unfortunately, indiscriminate use of upgrading introduces a new and potentially serious source of deadlocks. Suppose, that $T_1$ and $T_2$ each read database element $A$ and write a new value for $A$. If both transactions use an upgrading approach, first getting a shared lock on $A$ and then upgrading it to exclusive, the sequence of events suggested in Fig. 18.18 will happen whenever $T_1$ and $T_2$ initiate at approximately the same time.

| $T_1$ | $T_2$ |
|---|---|
| $sl_1(A)$ | |
| | $sl_2(A)$ |
| $xl_1(A)$ **Denied** | |
| | $xl_2(A)$ **Denied** |

Figure 18.18: Upgrading by two transactions can cause a deadlock

$T_1$ and $T_2$ are both able to get shared locks on $A$. Then, they each try to upgrade to exclusive, but the scheduler forces each to wait because the other has a shared lock on $A$. Thus, neither can make progress, and they will each wait forever, or until the system discovers that there is a deadlock, aborts one of the two transactions, and gives the other the exclusive lock on $A$. □

### 18.4.4 Update Locks

We can avoid the deadlock problem of Example 18.16 with a third lock mode, called *update locks*. An update lock $ul_i(X)$ gives transaction $T_i$ only the privilege to read $X$, not to write $X$. However, only the update lock can be upgraded to a write lock later; a read lock cannot be upgraded. We can grant an update lock on $X$ when there are already shared locks on $X$, but once there is an update lock on $X$ we prevent additional locks of any kind — shared, update, or exclusive — from being taken on $X$. The reason is that if we don't deny such locks, then the updater might never get a chance to upgrade to exclusive, since there would always be other locks on $X$.

This rule leads to an asymmetric compatibility matrix, because the update (U) lock looks like a shared lock when we are requesting it and looks like an

exclusive lock when we already have it. Thus, the columns for U and S locks are the same, and the rows for U and X locks are the same. The matrix is shown in Fig. 18.19.[3]

| | S | X | U |
|---|---|---|---|
| S | Yes | No | Yes |
| X | No | No | No |
| U | No | No | No |

Figure 18.19: Compatibility matrix for shared, exclusive, and update locks

**Example 18.17:** The use of update locks would have no effect on Example 18.15. As its third action, $T_1$ would take an update lock on $B$, rather than a shared lock. But the update lock would be granted, since only shared locks are held on $B$, and the same sequence of actions shown in Fig. 18.17 would occur.

However, update locks fix the problem shown in Example 18.16. Now, both $T_1$ and $T_2$ first request update locks on $A$ and only later take exclusive locks. Possible descriptions of $T_1$ and $T_2$ are:

$$T_1\text{: } ul_1(A);\ r_1(A);\ xl_1(A);\ w_1(A);\ u_1(A);$$
$$T_2\text{: } ul_2(A);\ r_2(A);\ xl_2(A);\ w_2(A);\ u_2(A);$$

The sequence of events corresponding to Fig. 18.18 is shown in Fig. 18.20. Now, $T_2$, the second to request an update lock on $A$, is denied. $T_1$ is allowed to finish, and then $T_2$ may proceed. The lock system has effectively prevented concurrent execution of $T_1$ and $T_2$, but in this example, any significant amount of concurrent execution will result in either a deadlock or an inconsistent database state. □

| $T_1$ | $T_2$ |
|---|---|
| $ul_1(A);\ r_1(A);$ | |
| | $ul_2(A)$ **Denied** |
| $xl_1(A);\ w_1(A);\ u_1(A);$ | |
| | $ul_2(A);\ r_2(A);$ |
| | $xl_2(A);\ w_2(A);\ u_2(A);$ |

Figure 18.20: Correct execution using update locks

[3]Remember, however, that there is an additional condition regarding legality of schedules that is not reflected by this matrix: a transaction holding a shared lock but not an update lock on an element $X$ cannot be given an exclusive lock on $X$, even though we do not in general prohibit a transaction from holding multiple locks on an element.

### 18.4.5 Increment Locks

Another interesting kind of lock that is useful in some situations is an "increment lock." Many transactions operate on the database only by incrementing or decrementing stored values. For example, consider a transaction that transfers money from one bank account to another.

The useful property of increment actions is that they commute with each other, since if two transactions add constants to the same database element, it does not matter which goes first, as the diagram of database state transitions in Fig. 18.21 suggests. On the other hand, incrementation commutes with neither reading nor writing; If you read $A$ before or after it is incremented, you leave different values, and if you increment $A$ before or after some other transaction writes a new value for $A$, you get different values of $A$ in the database.

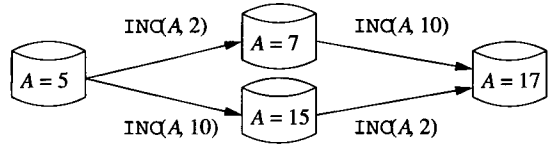


Figure 18.21: Two increment actions commute, since the final database state does not depend on which went first

Let us introduce as a possible action in transactions the *increment* action, written `INC(A,c)`. Informally, this action adds constant $c$ to database element $A$, which we assume is a single number. Note that $c$ could be negative, in which case we are really decrementing $A$. In practice, we might apply `INC` to a component of a tuple, while the tuple itself, rather than one of its components, is the lockable element. More formally, we use `INC(A,c)` to stand for the atomic execution of the following steps: `READ(A,t); t := t+c; WRITE(A,t);`.

Corresponding to the increment action, we need an *increment lock*. We shall denote the action of $T_i$ requesting an increment lock on $X$ by $il_i(X)$. We also use shorthand $inc_i(X)$ for the action in which transaction $T_i$ increments database element $X$ by some constant; the exact constant doesn't matter.

The existence of increment actions and locks requires us to make several modifications to our definitions of consistent transactions, conflicts, and legal schedules. These changes are:

a) A consistent transaction can only have an increment action on $X$ if it holds an increment lock on $X$ at the time. An increment lock does not enable either read or write actions, however.

b) In a legal schedule, any number of transactions can hold an increment lock on $X$ at any time. However, if an increment lock on $X$ is held by some transaction, then no other transaction can hold either a shared or exclusive lock on $X$ at the same time. These requirements are expressed

by the compatibility matrix of Fig. 18.22, where $I$ represents a lock in increment mode.

c) The action $inc_i(X)$ conflicts with both $r_j(X)$ and $w_j(X)$, for $j \neq i$, but does not conflict with $inc_j(X)$.

| | S | X | I |
|---|---|---|---|
| S | Yes | No | No |
| X | No | No | No |
| I | No | No | Yes |

Figure 18.22: Compatibility matrix for shared, exclusive, and increment locks

**Example 18.18:** Consider two transactions, each of which read database element $A$ and then increment $B$.

$$T_1\text{: } sl_1(A);\ r_1(A);\ il_1(B);\ inc_1(B);\ u_1(A);\ u_1(B);$$
$$T_2\text{: } sl_2(A);\ r_2(A);\ il_2(B);\ inc_2(B);\ u_2(A);\ u_2(B);$$

Notice that the transactions are consistent, since they only perform an incrementation while they have an increment lock, and they only read while they have a shared lock. Figure 18.23 shows a possible interleaving of $T_1$ and $T_2$. $T_1$ reads $A$ first, but then $T_2$ both reads $A$ and increments $B$. However, $T_1$ is then allowed to get its increment lock on $B$ and proceed.

| $T_1$ | $T_2$ |
|---|---|
| $sl_1(A);\ r_1(A);$ | |
| | $sl_2(A);\ r_2(A);$ |
| | $il_2(B);\ inc_2(B);$ |
| $il_1(B);\ inc_1(B);$ | |
| | $u_2(A);\ u_2(B);$ |
| $u_1(A);\ u_1(B);$ | |

Figure 18.23: A schedule of transactions with increment actions and locks

Notice that the scheduler did not have to delay any requests in Fig. 18.23. Suppose, for instance, that $T_1$ increments $B$ by $A$, and $T_2$ increments $B$ by $2A$. They can execute in either order, since the value of $A$ does not change, and the incrementations may also be performed in either order.

Put another way, we may look at the sequence of non-lock actions in the schedule of Fig. 18.23; they are:

$$S\text{: } r_1(A);\ r_2(A);\ inc_2(B);\ inc_1(B);$$

We may move the last action, $inc_1(B)$, to the second position, since it does not conflict with another increment of the same element, and surely does not conflict with a read of a different element. This sequence of swaps shows that $S$ is conflict-equivalent to the serial schedule $r_1(A)$; $inc_1(B)$; $r_2(A)$; $inc_2(B)$;. Similarly, we can move the first action, $r_1(A)$ to the third position by swaps, giving a serial schedule in which $T_2$ precedes $T_1$. □

## 18.4.6 Exercises for Section 18.4

**Exercise 18.4.1:** For each of the schedules of transactions $T_1$, $T_2$, and $T_3$ below:

a) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(D)$;

b) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(A)$;

c) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(C)$; $w_2(D)$; $w_3(E)$;

d) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;

e) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(A)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;

do each of the following:

*i.* Insert shared and exclusive locks, and insert unlock actions. Place a shared lock immediately in front of each read action that is not followed by a write action of the same element by the same transaction. Place an exclusive lock in front of every other read or write action. Place the necessary unlocks at the end of every transaction.

*ii.* Tell what happens when each schedule is run by a scheduler that supports shared and exclusive locks.

*iii.* Insert shared and exclusive locks in a way that allows upgrading. Place a shared lock in front of every read, an exclusive lock in front of every write, and place the necessary unlocks at the ends of the transactions.

*iv.* Tell what happens when each schedule from (*iii*) is run by a scheduler that supports shared locks, exclusive locks, and upgrading.

*v.* Insert shared, exclusive, and update locks, along with unlock actions. Place a shared lock in front of every read action that is not going to be upgraded, place an update lock in front of every read action that will be upgraded, and place an exclusive lock in front of every write action. Place unlocks at the ends of transactions, as usual.

*vi.* Tell what happens when each schedule from (*v*) is run by a scheduler that supports shared, exclusive, and update locks.

! **Exercise 18.4.2:** Consider the two transactions:

$$T_1\text{: } r_1(A);\ r_1(B);\ inc_1(A);\ inc_1(B);$$
$$T_2\text{: } r_2(A);\ r_2(B);\ inc_2(A);\ inc_2(B);$$

Answer the following:

a) How many interleavings of these transactions are serializable?

b) If the order of incrementation in $T_2$ were reversed [i.e., $inc_2(B)$ followed by $inc_2(A)$], how many serializable interleavings would there be?

**Exercise 18.4.3:** For each of the following schedules, insert appropriate locks (read, write, or increment) before each action, and unlocks at the ends of transactions. Then tell what happens when the schedule is run by a scheduler that supports these three types of locks.

a) $r_1(A);\ r_2(B);\ inc_1(B);\ inc_2(C);\ w_1(C);\ w_2(D);$

b) $r_1(A);\ r_2(B);\ inc_1(B);\ inc_2(A);\ w_1(C);\ w_2(D);$

c) $inc_1(A);\ inc_2(B);\ inc_1(B);\ inc_2(C);\ w_1(C);\ w_2(D);$

**Exercise 18.4.4:** In Exercise 18.1.1, we discussed a hypothetical transaction involving an airline reservation. If the transaction manager had available to it shared, exclusive, update, and increment locks, what lock would you recommend for each of the steps of the transaction?

**Exercise 18.4.5:** The action of multiplication by a constant factor can be modeled by an action of its own. Suppose `MC(X,c)` stands for an atomic execution of the steps `READ(X,t); t := c*t; WRITE(X,t);`. We can also introduce a lock mode that allows only multiplication by a constant factor.

a) Show the compatibility matrix for read, write, and multiplication-by-a-constant locks.

! b) Show the compatibility matrix for read, write, incrementation, and multiplication-by-a-constant locks.

! **Exercise 18.4.6:** Suppose for sake of argument that database elements are two-dimensional vectors. There are four operations we can perform on vectors, and each will have its own type of lock.

*i.* Change the value along the $x$-axis (an $X$-lock).

*ii.* Change the value along the $y$-axis (a $Y$-lock).

*iii.* Change the angle of the vector (an $A$-lock).

*iv.* Change the magnitude of the vector (an $M$-lock).

Answer the following questions.

a) Which pairs of operations commute? For example, if we rotate the vector so its angle is $120^o$ and then change the $x$-coordinate to be 10, is that the same as first changing the $x$-coordinate to 10 and then changing the angle to $120^o$?

b) Based on your answer to (a), what is the compatibility matrix for the four types of locks?

!! c) Suppose we changed the four operations so that instead of giving new values for a measure, the operations incremented the measure (e.g., "add 10 to the $x$-coordinate," or "rotate the vector $30^o$ clockwise"). What would the compatibility matrix then be?

! **Exercise 18.4.7:** Here is a schedule with one action missing:

$$r_1(A);\ r_2(B);\ ???;\ w_1(C);\ w_2(A);$$

Your problem is to figure out what actions of certain types could replace the ??? and make the schedule not be serializable. Tell all possible nonserializable replacements for each of the following types of action: (a) Read (b) Write (c) Update (d) Increment.

## 18.5 An Architecture for a Locking Scheduler

Having seen a number of different locking schemes, we next consider how a scheduler that uses one of these schemes operates. We shall consider here only a simple scheduler architecture based on several principles:

1. The transactions themselves do not request locks, or cannot be relied upon to do so. It is the job of the scheduler to insert lock actions into the stream of reads, writes, and other actions that access data.

2. Transactions do not release locks. Rather, the scheduler releases the locks when the transaction manager tells it that the transaction will commit or abort.

### 18.5.1 A Scheduler That Inserts Lock Actions

Figure 18.24 shows a two-part scheduler that accepts requests such as read, write, commit, and abort, from transactions. The scheduler maintains a lock table, which, although it is shown as secondary-storage data, may be partially or completely in main memory. Normally, the main memory used by the lock table is not part of the buffer pool that is used for query execution and logging. Rather, the lock table is just another component of the DBMS, and will be

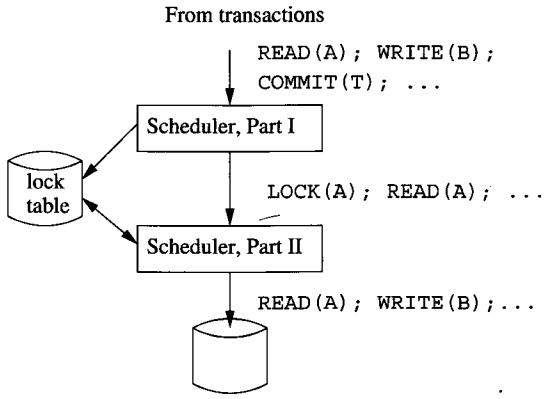


Figure 18.24: A scheduler that inserts lock requests into the transactions' request stream

allocated space by the operating system like other code and internal data of the DBMS.

Actions requested by a transaction are generally transmitted through the scheduler and executed on the database. However, under some circumstances a transaction is *delayed*, waiting for a lock, and its requests are not (yet) transmitted to the database. The two parts of the scheduler perform the following actions:

1. Part I takes the stream of requests generated by the transactions and inserts appropriate lock actions ahead of all database-access operations, such as read, write, increment, or update. The database access actions are then transmitted to Part II. Part I of the scheduler must select an appropriate lock mode from whatever set of lock modes the scheduler is using.

2. Part II takes the sequence of lock and database-access actions passed to it by Part I, and executes each appropriately. If a lock or database-access request is received by Part II, it determines whether the issuing transaction $T$ is already delayed, because a lock has not been granted. If so, then the action is itself delayed and added to a list of actions that must eventually be performed for transaction $T$. If $T$ is *not* delayed (i.e., all locks it previously requested have been granted already), then

   (a) If the action is a database access, it is transmitted to the database and executed.

   (b) If a lock action is received by Part II, it examines the lock table to see if the lock can be granted.

      i. If so, the lock table is modified to include the lock just granted.

ii. If not, then an entry must be made in the lock table to indicate that the lock has been requested. Part II of the scheduler then delays transaction $T$ until such time as the lock is granted.

3. When a transaction $T$ commits or aborts, Part I is notified by the transaction manager, and releases all locks held by $T$. If any transactions are waiting for any of these locks, Part I notifies Part II.

4. When Part II is notified that a lock on some database element $X$ is available, it determines the next transaction or transactions that can now be given a lock on $X$. The transaction(s) that receive a lock are allowed to execute as many of their delayed actions as can execute, until they either complete or reach another lock request that cannot be granted.

**Example 18.19:** If there is only one kind of lock, as in Section 18.3, then the task of Part I of the scheduler is simple. If it sees any action on database element $X$, and it has not already inserted a lock request on $X$ for that transaction, then it inserts the request. When a transaction commits or aborts, Part I can forget about that transaction after releasing its locks, so the memory required for Part I does not grow indefinitely.

When there are several kinds of locks, the scheduler may require advance notice of what future actions on the same database element will occur. Let us reconsider the case of shared-exclusive-update locks, using the transactions of Example 18.15, which we now write without any locks at all:

$$T_1\text{: } r_1(A);\ r_1(B);\ w_1(B);$$
$$T_2\text{: } r_2(A);\ r_2(B);$$

The messages sent to Part I of the scheduler must include not only the read or write request, but an indication of future actions on the same element. In particular, when $r_1(B)$ is sent, the scheduler needs to know that there will be a later $w_1(B)$ action (or might be such an action). There are several ways the information might be made available. For example, if the transaction is a query, we know it will not write anything. If the transaction is a SQL database modification command, then the query processor can determine in advance the database elements that might be both read and written. If the transaction is a program with embedded SQL, then the compiler has access to all the SQL statements (which are the only ones that can access the database) and can determine the potential database elements written.

In our example, suppose that events occur in the order suggested by Fig. 18.17. Then $T_1$ first issues $r_1(A)$. Since there will be no future upgrading of this lock, the scheduler inserts $sl_1(A)$ ahead of $r_1(A)$. Next, the requests from $T_2$ — $r_2(A)$ and $r_2(B)$ — arrive at the scheduler. Again there is no future upgrade, so the sequence of actions $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$ are issued by Part I.

Then, the action $r_1(B)$ arrives at the scheduler, along with a warning that this lock may be upgraded. The scheduler Part I thus emits $ul_1(B)$; $r_1(B)$ to

Part II. The latter consults the lock table and finds that it can grant the update lock on $B$ to $T_1$, because there are only shared locks on $B$.

When the action $w_1(B)$ arrives at the scheduler, Part I emits $xl_1(B)$; $w_1(B)$. However, Part II cannot grant the $xl_1(B)$ request, because there is a shared lock on $B$ for $T_2$. This and any subsequent actions from $T_1$ are delayed, stored by Part II for future execution. Eventually, $T_2$ commits, and Part I releases the locks on $A$ and $B$ that $T_2$ held. At that time, it is found that $T_1$ is waiting for a lock on $B$. Part II of the scheduler is notified, and it finds the lock $xl_1(B)$ is now available. It enters this lock into the lock table and proceeds to execute stored actions from $T_1$ to the extent possible. In this case, $T_1$ completes. □

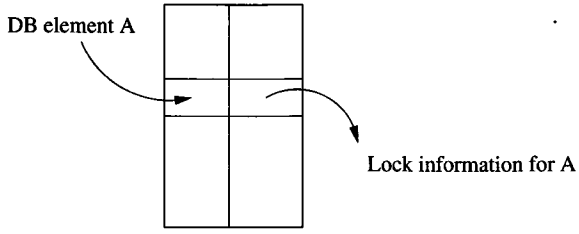


Figure 18.25: A lock table is a mapping from database elements to their lock information

### 18.5.2 The Lock Table

Abstractly, the lock table is a relation that associates database elements with locking information about that element, as suggested by Fig. 18.25. The table might, for instance, be implemented with a hash table, using (addresses of) database elements as the hash key. Any element that is not locked does not appear in the table, so the size is proportional to the number of locked elements only, not to the size of the entire database.

In Fig. 18.26 is an example of the sort of information we would find in a lock-table entry. This example structure assumes that the shared-exclusive-update lock scheme of Section 18.4.4 is used by the scheduler. The entry shown for a typical database element $A$ is a tuple with the following components:

1. The *group mode* is a summary of the most stringent conditions that a transaction requesting a new lock on $A$ faces. Rather than comparing the lock request with every lock held by another transaction on the same element, we can simplify the grant/deny decision by comparing the request with only the group mode.[4] For the shared-exclusive-update ($SXU$) lock scheme, the rule is simple: the group mode:

[4]The lock manager must, however, deal with the possibility that the requesting transaction already has a lock in another mode on the same element. For instance, in the $SXU$ lock system discussed, the lock manager may be able to grant an $X$-lock request if the requesting

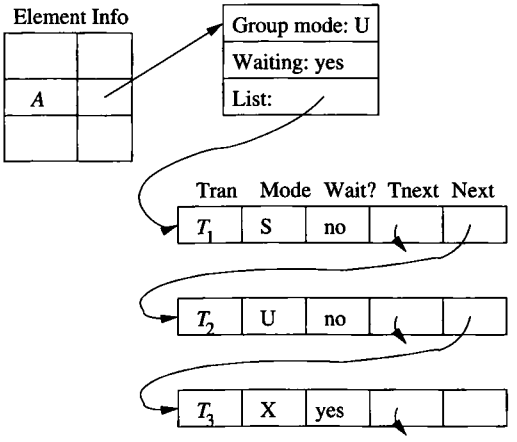


Figure 18.26: Structure of lock-table entries

(a) $S$ means that only shared locks are held.

(b) $U$ means that there is one update lock and perhaps one or more shared locks.

(c) $X$ means there is one exclusive lock and no other locks.

For other lock schemes, there is usually an appropriate system of summaries by a group mode; we leave examples as exercises.

2. The *waiting* bit tells that there is at least one transaction waiting for a lock on $A$.

3. A list describing all those transactions that either currently hold locks on $A$ or are waiting for a lock on $A$. Useful information that each list entry has might include:

   (a) The name of the transaction holding or waiting for a lock.

   (b) The mode of this lock.

   (c) Whether the transaction is holding or waiting for the lock.

   We also show in Fig. 18.26 two links for each entry. One links the entries themselves, and the other links all entries for a particular transaction (`Tnext` in the figure). The latter link would be used when a transaction commits or aborts, so that we can easily find all the locks that must be released.

transaction is the one that holds a $U$ lock on the same element. For systems that do not support multiple locks held by one transaction on one element, the group mode always tells what the lock manager needs to know.

### Handling Lock Requests

Suppose transaction $T$ requests a lock on $A$. If there is no lock-table entry for $A$, then surely there are no locks on $A$, so the entry is created and the request is granted. If the lock-table entry for $A$ exists, we use it to guide the decision about the lock request. We find the group mode, which in Fig. 18.26 is $U$, or "update." Once there is an update lock on an element, no other lock can be granted (except in the case that $T$ itself holds the $U$ lock and other locks are compatible with $T$'s request). Thus, this request by $T$ is denied, and an entry will be placed on the list saying $T$ requests a lock (in whatever mode was requested), and `Wait? = 'yes'`.

If the group mode had been $X$ (exclusive), then the same thing would happen, but if the group mode were $S$ (shared), then another shared or update lock could be granted. In that case, the entry for $T$ on the list would have `Wait? = 'no'`, and the group mode would be changed to $U$ if the new lock were an update lock; otherwise, the group mode would remain $S$. Whether or not the lock is granted, the new list entry is linked properly, through its `Tnext` and `Next` fields. Notice that whether or not the lock is granted, the entry in the lock table tells the scheduler what it needs to know without having to examine the list of locks.

### Handling Unlocks

Now suppose transaction $T$ unlocks $A$. $T$'s entry on the list for $A$ is deleted. If the lock held by $T$ is not the same as the group mode (e.g., $T$ held an $S$ lock, while the group mode was $U$), then there is no reason to change the group mode. On the other hand, if $T$'s lock is in the group mode, we may have to examine the entire list to find the new group mode. In the example of Fig. 18.26, we know there can be only one $U$ lock on an element, so if that lock is released, the new group mode could be only $S$ (if there are shared locks remaining) or nothing (if no other locks are currently held).[5] If the group mode is $X$, we know there are no other locks, and if the group mode is $S$, we need to determine whether there are other shared locks.

If the value of `Waiting` is `'yes'`, then we need to grant one or more locks from the list of requested locks. There are several different approaches, each with its advantages:

1. *First-come-first-served*: Grant the lock request that has been waiting the longest. This strategy guarantees no *starvation*, the situation where a transaction can wait forever for a lock.

2. *Priority to shared locks*: First grant all the shared locks waiting. Then, grant one update lock, if there are any waiting. Only grant an exclusive lock if no others are waiting. This strategy can allow starvation, if a transaction is waiting for a $U$ or $X$ lock.

[5] We would never actually see a group mode of "nothing," since if there are no locks and no lock requests on an element, then there is no lock-table entry for that element.

3. *Priority to upgrading*: If there is a transaction with a $U$ lock waiting to upgrade it to an $X$ lock, grant that first. Otherwise, follow one of the other strategies mentioned.

### 18.5.3 Exercises for Section 18.5

**Exercise 18.5.1:** What are suitable group modes for a lock table if the lock modes used are:

a) Shared and exclusive locks.

! b) Shared, exclusive, and increment locks.

!! c) The lock modes of Exercise 18.4.6.

**Exercise 18.5.2:** For each of the schedules of Exercise 18.2.4, tell the steps that the locking scheduler described in this section would execute.

## 18.6 Hierarchies of Database Elements

Let us now return to the exploration of different locking schemes that we began in Section 18.4. In particular, we shall focus on two problems that come up when there is a tree structure to our data.

1. The first kind of tree structure we encounter is a hierarchy of lockable elements. We shall discuss in this section how to allow locks on both large elements, e.g., relations, and smaller elements contained within these, such as blocks holding several tuples of the relation, or individual tuples.

2. The second kind of hierarchy that is important in concurrency-control systems is data that is itself organized in a tree. A major example is B-tree indexes. We may view nodes of the B-tree as database elements, but if we do, then as we shall see in Section 18.7, the locking schemes studied so far perform poorly, and we need to use a new approach.

### 18.6.1 Locks With Multiple Granularity

Recall that the term "database element" was purposely left undefined, because different systems use different sizes of database elements to lock, such as tuples, pages or blocks, and relations. Some applications benefit from small database elements, such as tuples, while others are best off with large elements.

**Example 18.20:** Consider a database for a bank. If we treated relations as database elements, and therefore had only one lock for an entire relation such as the one giving account balances, then the system would allow very little concurrency. Since most transactions will change an account balance either positively or negatively, most transactions would need an exclusive lock on the

accounts relation. Thus, only one deposit or withdrawal could take place at any time, no matter how many processors we had available to execute these transactions. A better approach is to lock individual pages or data blocks. Thus, two accounts whose tuples are on different blocks can be updated at the same time, offering almost all the concurrency that is possible in the system. The extreme would be to provide a lock for every tuple, so any set of accounts whatsoever could be updated at once, but this fine a grain of locks is probably not worth the extra effort.

In contrast, consider a database of documents. These documents may be edited from time to time, but most transactions will retrieve whole documents. The sensible choice of database element is a complete document. Since most transactions are *read-only* (i.e., they do not perform any write actions), locking is only necessary to avoid the reading of a document that is in the middle of being edited. Were we to use smaller-granularity locks, such as paragraphs, sentences, or words, there would be essentially no benefit but added expense. The only activity a smaller-granularity lock would support is the ability for two people to edit different parts of a document simultaneously. □

Some applications could use both large- and small-grained locks. For instance, the bank database discussed in Example 18.20 clearly needs block- or tuple-level locking, but might also at some time need a lock on the entire accounts relation in order to audit accounts (e.g., check that the sum of the accounts is correct). However, permitting a shared lock on the accounts relation, in order to compute some aggregation on the relation, while at the same time there are exclusive locks on individual account tuples, can lead easily to unserializable behavior. The reason is that the relation is actually changing while a supposedly frozen copy of it is being read by the aggregation query.

### 18.6.2 Warning Locks

The solution to the problem of managing locks at different granularities involves a new kind of lock called a "warning." These locks are useful when the database elements form a nested or hierarchical structure, as suggested in Fig. 18.27. There, we see three levels of database elements:

1. Relations are the largest lockable elements.
2. Each relation is composed of one or more block or pages, on which its tuples are stored.
3. Each block contains one or more tuples.

The rules for managing locks on a hierarchy of database elements constitute the *warning protocol*, which involves both "ordinary" locks and "warning" locks. We shall describe the lock scheme where the ordinary locks are $S$ and $X$ (shared and exclusive). The warning locks will be denoted by prefixing $I$ (for "intention

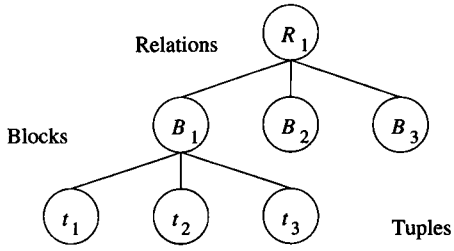


Figure 18.27: Database elements organized in a hierarchy

to") to the ordinary locks; for example $IS$ represents the intention to obtain a shared lock on a subelement. The rules of the warning protocol are:

1. To place an ordinary $S$ or $X$ lock on any element, we must begin at the root of the hierarchy.

2. If we are at the element that we want to lock, we need look no further. We request an $S$ or $X$ lock on that element.

3. If the element we wish to lock is further down the hierarchy, then we place a warning at this node; that is, if we want to get a shared lock on a subelement we request an $IS$ lock at this node, and if we want an exclusive lock on a subelement, we request an $IX$ lock on this node. When the lock on the current node is granted, we proceed to the appropriate child (the one whose subtree contains the node we wish to lock). We then repeat step (2) or step (3), as appropriate, until we reach the desired node.

| | IS | IX | S | X |
|---|---|---|---|---|
| IS | Yes | Yes | Yes | No |
| IX | Yes | Yes | No | No |
| S | Yes | No | Yes | No |
| X | No | No | No | No |

Figure 18.28: Compatibility matrix for shared, exclusive, and intention locks

In order to decide whether or not one of these locks can be granted, we use the compatibility matrix of Fig. 18.28. To see why this matrix makes sense, consider first the $IS$ column. When we request an $IS$ lock on a node $N$, we intend to read a descendant of $N$. The only time this intent could create a problem is if some other transaction has already claimed the right to write a new copy of the entire database element represented by $N$; thus we see "No" in the row for $X$. Notice that if some other transaction plans to write only a subelement, indicated by an $IX$ lock at $N$, then we can afford to grant the $IS$

## Group Modes for Intention Locks

The compatibility matrix of Fig. 18.28 exhibits a situation we have not seen before regarding the power of lock modes. In prior lock schemes, whenever it was possible for a database element to be locked in both modes $M$ and $N$ at the same time, one of these modes *dominates* the other, in the sense that its row and column each has "No" in whatever positions the other mode's row or column, respectively, has "No." For example, in Fig. 18.19 we see that $U$ dominates $S$, and $X$ dominates both $S$ and $U$. An advantage of knowing that there is always one dominant lock on an element is that we can summarize the effect of many locks with a "group mode," as discussed in Section 18.5.2.

As we see from Fig. 18.28, neither of modes $S$ and $IX$ dominate the other. Moreover, it is possible for an element to be locked in both modes $S$ and $IX$ at the same time, provided the locks are requested by the same transaction (recall that the "No" entries in a compatibility matrix only apply to locks held by some *other* transaction). A transaction might request both locks if it wanted to read an entire element and then write a few of its subelements. If a transaction has both $S$ and $IX$ locks on an element, then it restricts other transactions to the extent that either lock does. That is, we can imagine another lock mode $SIX$, whose row and column have "No" everywhere except in the entry for $IS$. The lock mode $SIX$ serves as the group mode if there is a transaction with locks in $S$ and $IX$ modes, but not $X$ mode.

Incidentally, we might imagine that the same situation occurs in the matrix of Fig 18.22 for increment locks. That is, one transaction could hold locks in both $S$ and $I$ modes. However, this situation is equivalent to holding a lock in $X$ mode, so we could use $X$ as the group mode in that situation.

lock at $N$, and allow the conflict to be resolved at a lower level, if indeed the intent to write and the intent to read happen to involve a common element.

Now consider the column for $IX$. If we intend to write a subelement of node $N$, then we must prevent either reading or writing of the entire element represented by $N$. Thus, we see "No" in the entries for lock modes $S$ and $X$. However, per our discussion of the $IS$ column, another transaction that reads or writes a subelement can have potential conflicts dealt with at that level, so $IX$ does not conflict with another $IX$ at $N$ or with an $IS$ at $N$.

Next, consider the column for $S$. Reading the element corresponding to node $N$ cannot conflict with either another read lock on $N$ or a read lock on some subelement of $N$, represented by $IS$ at $N$. Thus, we see "Yes" in the rows for both $S$ and $IS$. However, either an $X$ or an $IX$ means that some other transaction will write at least a part of the element represented by $N$. Thus,

we cannot grant the right to read all of $N$, which explains the "No" entries in the column for $S$.

Finally, the column for $X$ has only "No" entries. We cannot allow writing of all of node $N$ if any other transaction already has the right to read or write $N$, or to acquire that right on a subelement.

**Example 18.21:** Consider the relation

```
Movie(title, year, length, studioName)
```

Let us postulate a lock on the entire relation and locks on individual tuples. Then transaction $T_1$, which consists of the query

```
SELECT *
FROM Movie
WHERE title = 'King Kong';
```

starts by getting an $IS$ lock on the entire relation. It then moves to the individual tuples (there are three movies with the title *King Kong*), and gets $S$ locks on each of them.

Now, suppose that while we are executing the first query, transaction $T_2$, which changes the year component of a tuple, begins:

```
UPDATE Movie
SET year = 1939
WHERE title = 'Gone With the Wind';
```

$T_2$ needs an $IX$ lock on the relation, since it plans to write a new value for one of the tuples. $T_1$'s $IS$ lock on the relation is compatible, so the lock is granted. When $T_2$ goes to the tuple for *Gone With the Wind*, it finds no lock there, and so gets its $X$ lock and rewrites the tuple. Had $T_2$ tried to write a new value in the tuple for one of the *King Kong* movies, it would have had to wait until $T_1$ released its $S$ lock, since $S$ and $X$ are not compatible. The collection of locks is suggested by Fig. 18.29. □

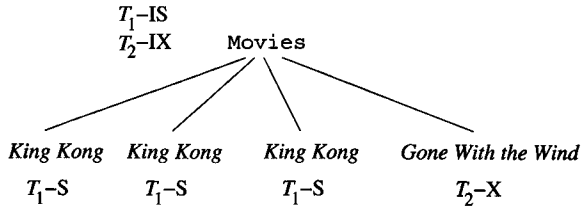


Figure 18.29: Locks granted to two transactions accessing `Movie` tuples

### 18.6.3 Phantoms and Handling Insertions Correctly

When transactions create new subelements of a lockable element, there are some opportunities to go wrong. The problem is that we can only lock existing items; there is no easy way to lock database elements that do not exist but might later be inserted. The following example illustrates the point.

**Example 18.22:** Suppose we have the same `Movie` relation as in Example 18.21, and the first transaction to execute is $T_3$, which is the query

```
SELECT SUM(length)
FROM Movie
WHERE studioName = 'Disney';
```

$T_3$ needs to read the tuples of all the Disney movies, so it might start by getting an $IS$ lock on the `Movie` relation and $S$ locks on each of the tuples for Disney movies.[6]

Now, a transaction $T_4$ comes along and inserts a new Disney movie. It seems that $T_4$ needs no locks, but it has made the result of $T_3$ incorrect. That fact by itself is not a concurrency problem, since the serial order $(T_3, T_4)$ is equivalent to what actually happened. However, there could also be some other element $X$ that both $T_3$ and $T_4$ write, with $T_4$ writing first, so there *could* be an unserializable behavior of more complex transactions.

To be more precise, suppose that $D_1$ and $D_2$ are pre-existing Disney movies, and $D_3$ is the new Disney movie inserted by $T_4$. Let $L$ be the sum of the lengths of the Disney movies computed by $T_3$, and assume the consistency constraint on the database is that $L$ should be equal to the sum of all the lengths of the Disney movies that existed the last time $L$ was computed. Then the following is a sequence of events that is legal under the warning protocol:

$$r_3(D_1);\ r_3(D_2);\ w_4(D_3);\ w_4(X);\ w_3(L);\ w_3(X);$$

Here, we have used $w_4(D_3)$ to represent the creation of $D_3$ by transaction $T_4$. The schedule above is not serializable. In particular, the value of $L$ is not the sum of the lengths of $D_1$, $D_2$, and $D_3$, which are the current Disney movies. Moreover, the fact that $X$ has the value written by $T_3$ and not $T_4$ rules out the possibility that $T_3$ was ahead of $T_4$ in a supposed equivalent serial order. □

The problem in Example 18.22 is that the new Disney movie has a *phantom* tuple, one that should have been locked but wasn't, because it didn't exist at the time the locks were taken. There is, however, a simple way to avoid the occurrence of phantoms. We must regard the insertion or deletion of a tuple as a write operation on the relation as a whole. Thus, transaction $T_4$ in Example 18.22 must obtain an $X$ lock on the relation `Movie`. Since $T_3$ has already locked this relation in mode $IS$, and that mode is not compatible with mode $X$, $T_4$ would have to wait until after $T_3$ completes.

[6] However, if there were many Disney movies, it might be more efficient just to get an $S$ lock on the entire relation.

### 18.6.4 Exercises for Section 18.6

**Exercise 18.6.1:** Consider, for variety, an object-oriented database. The objects of class $C$ are stored on two blocks, $B_1$ and $B_2$. Block $B_1$ contains objects $O_1$ and $O_2$, while block $B_2$ contains objects $O_3$, $O_4$, and $O_5$. The entire set of objects of class $C$, the blocks, and the individual objects form a hierarchy of lockable database elements. Tell the sequence of lock requests and the response of a warning-protocol-based scheduler to the following sequences of requests. You may assume all requests occur just before they are needed, and all unlocks occur at the end of the transaction.

a) $r_1(O_1)$; $w_2(O_2)$; $r_2(O_3)$; $w_1(O_4)$;

b) $r_1(O_5)$; $w_2(O_5)$; $r_2(O_3)$; $w_1(O_4)$;

c) $r_1(O_1)$; $r_1(O_3)$; $r_2(O_1)$; $w_2(O_4)$; $w_2(O_5)$;

d) $r_1(O_1)$; $r_2(O_2)$; $r_3(O_1)$; $w_1(O_3)$; $w_2(O_4)$; $w_3(O_5)$; $w_1(O_2)$;

**Exercise 18.6.2:** Change the sequence of actions in Example 18.22 so that the $w_4(D_3)$ action becomes a write by $T_4$ of the entire relation `Movie`. Then, show the action of a warning-protocol-based scheduler on this sequence of requests.

!! **Exercise 18.6.3:** Show how to add increment locks to a warning-protocol-based scheduler.

## 18.7 The Tree Protocol

Like Section 18.6, this section deals with data in the form of a tree. However, here, the nodes of the tree do not form a hierarchy based on containment. Rather, database elements are disjoint pieces of data, but the only way to get to a node is through its parent; B-trees are an important example of this sort of data. Knowing that we must traverse a particular path to an element gives us some important freedom to manage locks differently from the two-phase locking approaches we have seen so far.

### 18.7.1 Motivation for Tree-Based Locking

Let us consider a B-tree index in a system that treats individual nodes (i.e., blocks) as lockable database elements. The node is the right level of lock granularity, because treating smaller pieces as elements offers no benefit, and treating the entire B-tree as one database element prevents the sort of concurrent use of the index that can be achieved via the mechanisms that form the subject of this section.

If we use a standard set of lock modes, like shared, exclusive, and update locks, and we use two-phase locking, then concurrent use of the B-tree is almost impossible. The reason is that every transaction using the index must begin by

locking the root node of the B-tree. If the transaction is 2PL, then it cannot unlock the root until it has acquired all the locks it needs, both on B-tree nodes and other database elements.[7] Moreover, since in principle any transaction that inserts or deletes could wind up rewriting the root of the B-tree, the transaction needs at least an update lock on the root node, or an exclusive lock if update mode is not available. Thus, only one transaction that is not read-only can access the B-tree at any time.

However, in most situations, we can deduce almost immediately that a B-tree node will not be rewritten, even if the transaction inserts or deletes a tuple. For example, if the transaction inserts a tuple, but the child of the root that we visit is not completely full, then we know the insertion cannot propagate up to the root. Similarly, if the transaction deletes a single tuple, and the child of the root we visit has more than the minimum number of keys and pointers, then we can be sure the root will not change.

Thus, as soon as a transaction moves to a child of the root and observes the (quite usual) situation that rules out a rewrite of the root, we would like to release the lock on the root. The same observation applies to the lock on any interior node of the B-tree. Unfortunately, releasing the lock on the root early will violate 2PL, so we cannot be sure that the schedule of several transactions accessing the B-tree will be serializable. The solution is a specialized protocol for transactions that access tree-structured data such as B-trees. The protocol violates 2PL, but uses the fact that accesses to elements must proceed down the tree to assure serializability.

### 18.7.2 Rules for Access to Tree-Structured Data

The following restrictions on locks form the *tree protocol*. We assume that there is only one kind of lock, represented by lock requests of the form $l_i(X)$, but the idea generalizes to any set of lock modes. We assume that transactions are consistent, and schedules must be legal (i.e., the scheduler will enforce the expected restrictions by granting locks on a node only when they do not conflict with locks already on that node), but there is no two-phase locking requirement on transactions.

1. A transaction's first lock may be at any node of the tree.[8]
2. Subsequent locks may only be acquired if the transaction currently has a lock on the parent node.
3. Nodes may be unlocked at any time.
4. A transaction may not relock a node on which it has released a lock, even if it still holds a lock on the node's parent.

[7] Additionally, there are good reasons why a transaction will hold all its locks until it is ready to commit; see Section 19.1.

[8] In the B-tree example of Section 18.7.1, the first lock would always be at the root.

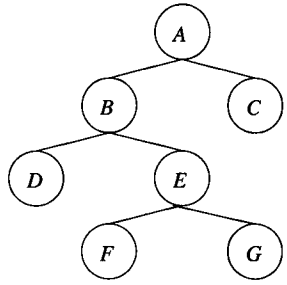


Figure 18.30: A tree of lockable elements

**Example 18.23:** Figure 18.30 shows a hierarchy of nodes, and Fig. 18.31 indicates the action of three transactions on this data. $T_1$ starts at the root $A$, and proceeds downward to $B$, $C$, and $D$. $T_2$ starts at $B$ and tries to move to $E$, but its move is initially denied because of the lock by $T_3$ on $E$. Transaction $T_3$ starts at $E$ and moves to $F$ and $G$. Notice that $T_1$ is not a 2PL transaction, because the lock on $A$ is relinquished before the lock on $D$ is acquired. Similarly, $T_3$ is not a 2PL transaction, although $T_2$ happens to be 2PL. □

### 18.7.3 Why the Tree Protocol Works

The tree protocol implies a serial order on the transactions involved in a schedule. We can define an order of precedence as follows. Say that $T_i <_S T_j$ if in schedule $S$, the transactions $T_i$ and $T_j$ lock a node in common, and $T_i$ locks the node first.

**Example 18.24:** In the schedule $S$ of Fig 18.31, we find $T_1$ and $T_2$ lock $B$ in common, and $T_1$ locks it first. Thus, $T_1 <_S T_2$. We also find that $T_2$ and $T_3$ lock $E$ in common, and $T_3$ locks it first; thus $T_3 <_S T_2$. However, there is no precedence between $T_1$ and $T_3$, because they lock no node in common. Thus, the precedence graph derived from these precedence relations is as shown in Fig. 18.32. □

If the precedence graph drawn from the precedence relations that we defined above has no cycles, then we claim that any topological order of the transactions is an equivalent serial schedule. For example, either $(T_1, T_3, T_2)$ or $(T_3, T_1, T_2)$ is an equivalent serial schedule for Fig. 18.31. The reason is that in such a serial schedule, all nodes are touched in the same order as they are in the original schedule.

To understand why the precedence graph described above must always be acyclic if the tree protocol is obeyed, observe the following:

- If two transactions lock several elements in common, then they are all locked in the same order.

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| $l_1(A)$; $r_1(A)$; | | |
| $l_1(B)$; $r_1(B)$; | | |
| $l_1(C)$; $r_1(C)$; | | |
| $w_1(A)$; $u_1(A)$; | | |
| $l_1(D)$; $r_1(D)$; | | |
| $w_1(B)$; $u_1(B)$; | | |
| | $l_2(B)$; $r_2(B)$; | |
| | | $l_3(E)$; $r_3(E)$; |
| $w_1(D)$; $u_1(D)$; | | |
| $w_1(C)$; $u_1(C)$; | | |
| | $l_2(E)$ **Denied** | |
| | | $l_3(F)$; $r_3(F)$; |
| | | $w_3(F)$; $u_3(F)$; |
| | | $l_3(G)$; $r_3(G)$ |
| | | $w_3(E)$; $u_3(E)$; |
| | $l_2(E)$; $r_2(E)$; | |
| | | $w_3(G)$; $u_3(G)$ |
| | $w_2(B)$; $u_2(B)$; | |
| | $w_2(E)$; $u_2(E)$; | |

Figure 18.31: Three transactions following the tree protocol

To see why, consider some transactions $T$ and $U$, which lock two or more items in common. First, notice that each transaction locks a set of elements that form a tree, and the intersection of two trees is itself a tree. Thus, there is some one highest element $X$ that both $T$ and $U$ lock. Suppose that $T$ locks $X$ first, but that there is some other element $Y$ that $U$ locks before $T$. Then there is a path in the tree of elements from $X$ to $Y$, and both $T$ and $U$ must lock each element along the path, because neither can lock a node without having a lock on its parent.

Consider the first element along this path, say $Z$, that $U$ locks first, as suggested by Fig. 18.33. Then $T$ locks the parent $P$ of $Z$ before $U$ does. But then $T$ is still holding the lock on $P$ when it locks $Z$, so $U$ has not yet locked

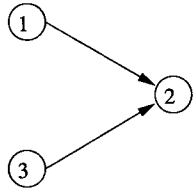


Figure 18.32: Precedence graph derived from the schedule of Fig. 18.31

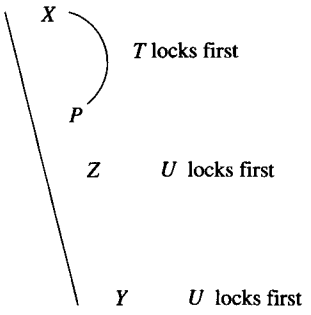


Figure 18.33: A path of elements locked by two transactions

$P$ when it locks $Z$. It cannot be that $Z$ is the first element $U$ locks in common with $T$, since they both lock ancestor $X$ (which could also be $P$, but not $Z$). Thus, $U$ cannot lock $Z$ until after it has acquired a lock on $P$, which is after $T$ locks $Z$. We conclude that $T$ precedes $U$ at every node they lock in common.

Now, consider an arbitrary set of transactions $T_1, T_2, \ldots, T_n$ that obey the tree protocol and lock some of the nodes of a tree according to schedule $S$. First, among those that lock the root, they do so in some order, and by the rule just observed:

- If $T_i$ locks the root before $T_j$, then $T_i$ locks every node in common with $T_j$ before $T_j$ does. That is, $T_i <_S T_j$, but not $T_j <_S T_i$.

We can show by induction on the number of nodes of the tree that there is some serial order equivalent to $S$ for the complete set of transactions.

**BASIS**: If there is only one node, the root, then as we just observed, the order in which the transactions lock the root serves.

**INDUCTION**: If there is more than one node in the tree, consider for each subtree of the root the set of transactions that lock one or more nodes in that subtree. Note that transactions locking the root may belong to more than one subtree, but a transaction that does not lock the root will belong to only one subtree. For instance, among the transactions of Fig. 18.31, only $T_1$ locks the root, and it belongs to both subtrees — the tree rooted at $B$ and the tree rooted at $C$. However, $T_2$ and $T_3$ belong only to the tree rooted at $B$.

By the inductive hypothesis, there is a serial order for all the transactions that lock nodes in any one subtree. We have only to blend the serial orders for the various subtrees. Since the only transactions these lists of transactions have in common are the transactions that lock the root, and we established that these transactions lock every node in common in the same order that they lock the root, it is not possible that two transactions locking the root appear in different orders in two of the sublists. Specifically, if $T_i$ and $T_j$ appear on the list for some child $C$ of the root, then they lock $C$ in the same order as they lock

the root and therefore appear on the list in that order. Thus, we can build a serial order for the full set of transactions by starting with the transactions that lock the root, in their appropriate order, and interspersing those transactions that do not lock the root in any order consistent with the serial order of their subtrees.

**Example 18.25:** Suppose there are 10 transactions $T_1, T_2, \ldots, T_{10}$, and of these, $T_1$, $T_2$, and $T_3$ lock the root in that order. Suppose also that there are two children of the root, the first locked by $T_1$ through $T_7$ and the second locked by $T_2$, $T_3$, $T_8$, $T_9$, and $T_{10}$. Hypothetically, let the serial order for the first subtree be $(T_4, T_1, T_5, T_2, T_6, T_3, T_7)$; note that this order must include $T_1$, $T_2$, and $T_3$ in that order. Also, let the serial order for the second subtree be $(T_8, T_2, T_9, T_{10}, T_3)$. As must be the case, the transactions $T_2$ and $T_3$, which locked the root, appear in this sequence in the order in which they locked the root.

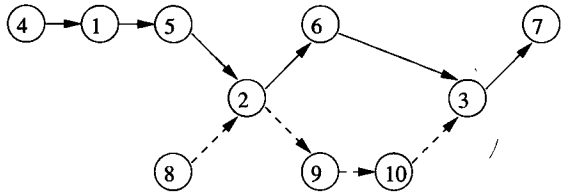


Figure 18.34: Combining serial orders for the subtrees into a serial order for all transactions

The constraints imposed on the serial order of these transactions are as shown in Fig. 18.34. Solid lines represent constraints due to the order at the first child of the root, while dashed lines represent the order at the second child. $(T_4, T_8, T_1, T_5, T_2, T_9, T_6, T_{10}, T_3, T_7)$ is one of the many topological sorts of this graph. □

### 18.7.4 Exercises for Section 18.7

**Exercise 18.7.1:** Suppose we perform the following actions on the B-tree of Fig. 14.13. If we use the tree protocol, when can we release a write-lock on each of the nodes searched?

(a) Insert 10 (b) Insert 20 (c) Delete 5 (d) Delete 23.

! **Exercise 18.7.2:** Consider the following transactions that operate on the tree of Fig. 18.30.

$T_1$: $r_1(A)$; $r_1(B)$; $r_1(E)$;
$T_2$: $r_2(A)$; $r_2(C)$; $r_2(B)$;
$T_3$: $r_3(B)$; $r_3(E)$; $r_3(F)$;

If schedules follow the tree protocol, in how many ways can we interleave: (a) $T_1$ and $T_2$ (b) $T_1$ and $T_3$ !! (c) all three?

! **Exercise 18.7.3:** Suppose there are eight transactions $T_1, T_2, \ldots, T_8$, of which the odd-numbered transactions, $T_1$, $T_3$, $T_5$, and $T_7$, lock the root of a tree, in that order. There are three children of the root, the first locked by $T_1$, $T_2$, $T_3$, and $T_4$ in that order. The second child is locked by $T_3$, $T_6$, and $T_5$, in that order, and the third child is locked by $T_8$ and $T_7$, in that order. How many serial orders of the transactions are consistent with these statements?

!! **Exercise 18.7.4:** Suppose we use the tree protocol with shared and exclusive locks for reading and writing, respectively. Rule (2), which requires a lock on the parent to get a lock on a node, must be changed to prevent unserializable behavior. What is the proper rule (2) for shared and exclusive locks? *Hint*: Does the lock on the parent have to be of the same type as the lock on the child?

## 18.8 Concurrency Control by Timestamps

Next, we shall consider two methods other than locking that are used in some systems to assure serializability of transactions:

1. *Timestamping.* Assign a "timestamp" to each transaction. Record the timestamps of the transactions that last read and write each database element, and compare these values with the transactions timestamps, to assure that the serial schedule according to the transactions' timestamps is equivalent to the actual schedule of the transactions. This approach is the subject of the present section.

2. *Validation.* Examine timestamps of the transaction and the database elements when a transaction is about to commit; this process is called "validation" of the transaction. The serial schedule that orders transactions according to their validation time must be equivalent to the actual schedule. The validation approach is discussed in Section 18.9.

Both these approaches are *optimistic*, in the sense that they assume that no unserializable behavior will occur and only fix things up when a violation is apparent. In contrast, all locking methods assume that things will go wrong unless transactions are prevented in advance from engaging in nonserializable behavior. The optimistic approaches differ from locking in that the only remedy when something does go wrong is to abort and restart a transaction that tries to engage in unserializable behavior. In contrast, locking schedulers delay transactions, but do not abort them.[9] Generally, optimistic schedulers are

[9]That is not to say that a system using a locking scheduler will never abort a transaction; for instance, Section 19.2 discusses aborting transactions to fix deadlocks. However, a locking scheduler never uses a transaction abort simply as a response to a lock request that it cannot grant.

better than locking when many of the transactions are read-only, since those transactions can never, by themselves, cause unserializable behavior.

### 18.8.1 Timestamps

To use timestamping as a concurrency-control method, the scheduler needs to assign to each transaction $T$ a unique number, its *timestamp* TS($T$). Timestamps must be issued in ascending order, at the time that a transaction first notifies the scheduler that it is beginning. Two approaches to generating timestamps are:

a) We can use the system clock as the timestamp, provided the scheduler does not operate so fast that it could assign timestamps to two transactions on one tick of the clock.

b) The scheduler can maintain a counter. Each time a transaction starts, the counter is incremented by 1, and the new value becomes the timestamp of the transaction. In this approach, timestamps have nothing to do with "time," but they have the important property that we need for any timestamp-generating system: a transaction that starts later has a higher timestamp than a transaction that starts earlier.

Whatever method of generating timestamps is used, the scheduler must maintain a table of currently active transactions and their timestamps.

To use timestamps as a concurrency-control method, we need to associate with each database element $X$ two timestamps and an additional bit:

1. RT($X$), the *read time* of $X$, which is the highest timestamp of a transaction that has read $X$.

2. WT($X$), the *write time* of $X$, which is the highest timestamp of a transaction that has written $X$.

3. C($X$), the *commit bit* for $X$, which is true if and only if the most recent transaction to write $X$ has already committed. The purpose of this bit is to avoid a situation where one transaction $T$ reads data written by another transaction $U$, and $U$ then aborts. This problem, where $T$ makes a "dirty read" of uncommitted data, certainly can cause the database state to become inconsistent, and any scheduler needs a mechanism to prevent dirty reads.[10]

### 18.8.2 Physically Unrealizable Behaviors

In order to understand the architecture and rules of a timestamp scheduler, we need to remember that the scheduler assumes the timestamp order of transactions is also the serial order in which they must appear to execute. Thus,

[10] Although commercial systems generally give the user an option to allow dirty reads, as suggested by the SQL isolation level `READ UNCOMMITTED` in Section 6.6.5.

the job of the scheduler, in addition to assigning timestamps and updating $RT$, $WT$, and $C$ for the database elements, is to check that whenever a read or write occurs, what happens in real time *could* have happened if each transaction had executed instantaneously at the moment of its timestamp. If not, we say the behavior is *physically unrealizable*. There are two kinds of problems that can occur:

1. *Read too late*: Transaction $T$ tries to read database element $X$, but the write time of $X$ indicates that the current value of $X$ was written after $T$ theoretically executed; that is, $\text{TS}(T) < \text{WT}(X)$. Figure 18.35 illustrates the problem. The horizontal axis represents the real time at which events occur. Dotted lines link the actual events to the times at which they theoretically occur — the timestamp of the transaction that performs the event. Thus, we see a transaction $U$ that started after transaction $T$, but wrote a value for $X$ before $T$ reads $X$. $T$ should not be able to read the value written by $U$, because theoretically, $U$ executed after $T$ did. However, $T$ has no choice, because $U$'s value of $X$ is the one that $T$ now sees. The solution is to abort $T$ when the problem is encountered.

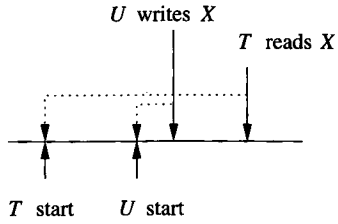


Figure 18.35: Transaction $T$ tries to read too late

2. *Write too late*: Transaction $T$ tries to write database element $X$. However, the read time of $X$ indicates that some other transaction should have read the value written by $T$, but read some other value instead. That is, $\text{WT}(X) < \text{TS}(T) < \text{RT}(X)$. The problem is shown in Fig. 18.36. There we see a transaction $U$ that started after $T$, but read $X$ before $T$ got a chance to write $X$. When $T$ tries to write $X$, we find $\text{RT}(X) > \text{TS}(T)$, meaning that $X$ has already been read by a transaction $U$ that theoretically executed later than $T$. We also find $\text{WT}(X) < \text{TS}(T)$, which means that no other transaction wrote into $X$ a value that would have overwritten $T$'s value, thus, negating $T$'s responsibility to get its value into $X$ so transaction $U$ could read it.

### 18.8.3 Problems With Dirty Data

There is a class of problems that the commit bit is designed to solve. One of these problems, a "dirty read," is suggested in Fig. 18.37. There, transaction

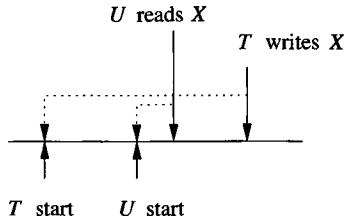


Figure 18.36: Transaction $T$ tries to write too late

$T$ reads $X$, and $X$ was last written by $U$. The timestamp of $U$ is less than that of $T$, and the read by $T$ occurs after the write by $U$ in real time, so the event seems to be physically realizable. However, it is possible that after $T$ reads the value of $X$ written by $U$, transaction $U$ will abort; perhaps $U$ encounters an error condition in its own data, such as a division by 0, or as we shall see in Section 18.8.4, the scheduler forces $U$ to abort because it tries to do something physically unrealizable. Thus, although there is nothing physically unrealizable about $T$ reading $X$, it is better to delay $T$'s read until $U$ commits or aborts. We can tell that $U$ is not committed because the commit bit $\mathrm{C}(X)$ will be false.

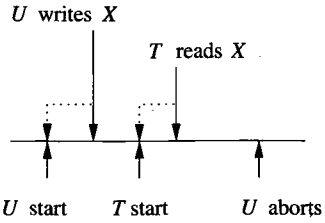


Figure 18.37: $T$ could perform a dirty read if it reads $X$ when shown

A second potential problem is suggested by Fig. 18.38. Here, $U$, a transaction with a later timestamp than $T$, has written $X$ first. When $T$ tries to write, the appropriate action is to do nothing. Evidently no other transaction $V$ that should have read $T$'s value of $X$ got $U$'s value instead, because if $V$ tried to read $X$ it would have aborted because of a too-late read. Future reads of $X$ will want $U$'s value or a later value of $X$, not $T$'s value. This idea, that writes can be skipped when a write with a later write-time is already in place, is called the *Thomas write rule.*

There is a potential problem with the Thomas write rule, however. If $U$ later aborts, as is suggested in Fig. 18.38, then its value of $X$ should be removed and the previous value and write-time restored. Since $T$ is committed, it would seem that the value of $X$ should be the one written by $T$ for future reading. However, we already skipped the write by $T$ and it is too late to repair the damage.

While there are many ways to deal with the problems just described, we

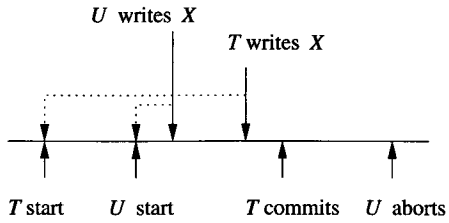


Figure 18.38: A write is cancelled because of a write with a later timestamp, but the writer then aborts

shall adopt a relatively simple policy based on the following assumed capability of the timestamp-based scheduler.

- When a transaction $T$ writes a database element $X$, the write is "tentative" and may be undone if $T$ aborts. The commit bit $C(X)$ is set to false, and the scheduler makes a copy of the old value of $X$ and its previous $\text{WT}(X)$.

### 18.8.4 The Rules for Timestamp-Based Scheduling

We can now summarize the rules that a scheduler using timestamps must follow to make sure that nothing physically unrealizable may occur. The scheduler, in response to a read or write request from a transaction $T$ has the choice of:

a) Granting the request,

b) Aborting $T$ (if $T$ would violate physical reality) and restarting $T$ with a new timestamp (abort followed by restart is often called *rollback*), or

c) Delaying $T$ and later deciding whether to abort $T$ or to grant the request (if the request is a read, and the read might be dirty, as in Section 18.8.3).

The rules are as follows:

1. Suppose the scheduler receives a request $r_T(X)$.
   (a) If $\text{TS}(T) \geq \text{WT}(X)$, the read is physically realizable.
      i. If $\text{C}(X)$ is true, grant the request. If $\text{TS}(T) > \text{RT}(X)$, set $\text{RT}(X) := \text{TS}(T)$; otherwise do not change $\text{RT}(X)$.
      ii. If $\text{C}(X)$ is false, delay $T$ until $\text{C}(X)$ becomes true, or the transaction that wrote $X$ aborts.
   (b) If $\text{TS}(T) < \text{WT}(X)$, the read is physically unrealizable. Rollback $T$; that is, abort $T$ and restart it with a new, larger timestamp.
2. Suppose the scheduler receives a request $w_T(X)$.

(a) If $\text{TS}(T) \geq \text{RT}(X)$ and $\text{TS}(T) \geq \text{WT}(X)$, the write is physically realizable and must be performed.

i. Write the new value for $X$,

ii. Set $\text{WT}(X) := \text{TS}(T)$, and

iii. Set $\text{C}(X) :=$ `false`.

(b) If $\text{TS}(T) \geq \text{RT}(X)$, but $\text{TS}(T) < \text{WT}(X)$, then the write is physically realizable, but there is already a later value in $X$. If $C(X)$ is true, then the previous writer of $X$ is committed, and we simply ignore the write by $T$; we allow $T$ to proceed and make no change to the database. However, if $C(X)$ is false, then we must delay $T$ as in point 1(a)ii.

(c) If $\text{TS}(T) < \text{RT}(X)$, then the write is physically unrealizable, and $T$ must be rolled back.

3. Suppose the scheduler receives a request to commit $T$. It must find (using a list the scheduler maintains) all the database elements $X$ written by $T$, and set $\text{C}(X) :=$ `true`. If any transactions are waiting for $X$ to be committed (found from another scheduler-maintained list), these transactions are allowed to proceed.

4. Suppose the scheduler receives a request to abort $T$ or decides to rollback $T$ as in 1b or 2c. Then any transaction that was waiting on an element $X$ that $T$ wrote must repeat its attempt to read or write, and see whether the action is now legal after $T$'s writes are cancelled.

**Example 18.26:** Figure 18.39 shows a schedule of three transactions, $T_1$, $T_2$, and $T_3$ that access three database elements, $A$, $B$, and $C$. The real time at which events occur increases down the page, as usual. We have also indicated the timestamps of the transactions and the read and write times of the elements. At the beginning, each of the database elements has both a read and write time of 0. The timestamps of the transactions are acquired when they notify the scheduler that they are beginning. Notice that even though $T_1$ executes the first data access, it does not have the least timestamp. Presumably $T_2$ was the first to notify the scheduler of its start, and $T_3$ did so next, with $T_1$ last to start.

In the first action, $T_1$ reads $B$. Since the write time of $B$ is less than the timestamp of $T_1$, this read is physically realizable and allowed to happen. The read time of $B$ is set to 200, the timestamp of $T_1$. The second and third read actions similarly are legal and result in the read time of each database element being set to the timestamp of the transaction that read it.

At the fourth step, $T_1$ writes $B$. Since the read time of $B$ is not bigger than the timestamp of $T_1$, the write is physically realizable. Since the write time of $B$ is no larger than the timestamp of $T_1$, we must actually perform the write. When we do, the write time of $B$ is raised to 200, the timestamp of the writing transaction $T_1$.

| $T_1$ | $T_2$ | $T_3$ | $A$ | $B$ | $C$ |
|---|---|---|---|---|---|
| 200 | 150 | 175 | RT=0<br>WT=0 | RT=0<br>WT=0 | RT=0<br>WT=0 |
| $r_1(B)$; | | | | RT=200 | |
| | $r_2(A)$; | | RT=150 | | |
| | | $r_3(C)$; | | | RT=175 |
| $w_1(B)$; | | | | WT=200 | |
| $w_1(A)$; | | | WT=200 | | |
| | $w_2(C)$; | | | | |
| | **Abort**; | | | | |
| | | $w_3(A)$; | | | |

Figure 18.39: Three transactions executing under a timestamp-based scheduler

Next, $T_2$ tries to write $C$. However, $C$ was already read by transaction $T_3$, which theoretically executed at time 175, while $T_2$ would have written its value at time 150. Thus, $T_2$ is trying to do something that s physically unrealizable, and $T_2$ must be rolled back.

The last step is the write of $A$ by $T_3$. Since the read time of $A$, 150, is less than the timestamp of $T_3$, 175, the write is legal. However, there is already a later value of $A$ stored in that database element, namely the value written by $T_1$, theoretically at time 200. Thus, $T_3$ is not rolled back, but neither does it write its value. □

### 18.8.5 Multiversion Timestamps

An important variation of timestamping maintains old versions of database elements in addition to the current version that is stored in the database itself. The purpose is to allow reads $r_T(X)$ that otherwise would cause transaction $T$ to abort (because the current version of $X$ was written in $T$'s future) to proceed by reading the version of $X$ that is appropriate for a transaction with $T$'s timestamp. The method is especially useful if database elements are disk blocks or pages, since then all that must be done is for the buffer manager to keep in memory certain blocks that might be useful for some currently active transaction.

**Example 18.27:** Consider the set of transactions accessing database element $A$ shown in Fig. 18.40. These transactions are operating under an ordinary timestamp-based scheduler, and when $T_3$ tries to read $A$, it finds WT($A$) to be greater than its own timestamp, and must abort. However, there is an old value of $A$ written by $T_1$ and overwritten by $T_2$ that would have been suitable for $T_3$ to read; this version of $A$ had a write time of 150, which is less than $T_3$'s timestamp of 175. If this old value of $A$ were available, $T_3$ could be allowed to read it, even though it is not the "current" value of $A$. □

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $A$ |
|---|---|---|---|---|
| 150 | 200 | 175 | 225 | RT=0<br>WT=0 |
| $r_1(A)$ | | | | RT=150 |
| $w_1(A)$ | | | | WT=150 |
| | $r_2(A)$ | | | RT=200 |
| | $w_2(A)$ | | | WT=200 |
| | | $r_3(A)$ | | |
| | | **Abort** | | |
| | | | $r_4(A)$ | RT=225 |

Figure 18.40: $T_3$ must abort because it cannot access an old value of $A$

A multiversion-timestamp scheduler differs from the scheduler described in Section 18.8.4 in the following ways:

1. When a new write $w_T(X)$ occurs, if it is legal, then a new version of database element $X$ is created. Its write time is $\text{TS}(T)$, and we shall refer to it as $X_t$, where $t = \text{TS}(T)$.

2. When a read $r_T(X)$ occurs, the scheduler finds the version $X_t$ of $X$ such that $t \leq \text{TS}(T)$, but there is no other version $X_{t'}$ with $t < t' \leq \text{TS}(T)$. That is, the version of $X$ written immediately before $T$ theoretically executed is the version that $T$ reads.

3. Write times are associated with *versions* of an element, and they never change.

4. Read times are also associated with versions. They are used to reject certain writes, namely one whose time is less than the read time of the previous version. Figure 18.41 suggests the problem, where $X$ has versions $X_{50}$ and $X_{100}$, the former was read by a transaction with timestamp 80, and a new write by a transaction $T$ whose timestamp is 60 occurs. This write must cause $T$ to abort, because its value of $X$ should have been read by the transaction with timestamp 80, had $T$ been allowed to execute.

5. When a version $X_t$ has a write time $t$ such that no active transaction has a timestamp less than $t$, then we may delete any version of $X$ *previous* to $X_t$.

**Example 18.28:** Let us reconsider the actions of Fig. 18.40 if multiversion timestamping is used. First, there are three versions of $A$: $A_0$, which exists before these transactions start, $A_{150}$, written by $T_1$, and $A_{200}$, written by $T_2$. Figure 18.42 shows the sequence of events, when the versions are created, and when they are read. Notice in particular that $T_3$ does not have to abort, because it can read an earlier version of $A$. □

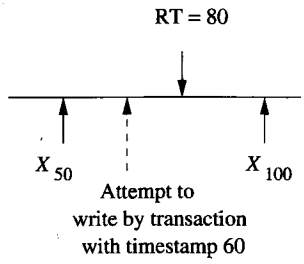


Figure 18.41: A transaction tries to write a version of $X$ that would make events physically unrealizable

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $A_0$ | $A_{150}$ | $A_{200}$ |
|---|---|---|---|---|---|---|
| 150 | 200 | 175 | 225 | | | |
| $r_1(A)$ | | | | Read | | |
| $w_1(A)$ | | | | | Create | |
| | $r_2(A)$ | | | | Read | |
| | $w_2(A)$ | | | | | Create |
| | | $r_3(A)$ | | | Read | |
| | | | $r_4(A)$ | | | Read |

Figure 18.42: Execution of transactions using multiversion concurrency control

### 18.8.6 Timestamps Versus Locking

Generally, timestamping is superior in situations where either most transactions are read-only, or it is rare that concurrent transactions will try to read and write the same element. In high-conflict situations, locking performs better. The argument for this rule-of-thumb is:

- Locking will frequently delay transactions as they wait for locks.
- But if concurrent transactions frequently read and write elements in common, then rollbacks will be frequent in a timestamp scheduler, introducing even more delay than a locking system.

There is an interesting compromise used in several commercial systems. The scheduler divides the transactions into read-only transactions and read/write transactions. Read/write transactions are executed using two-phase locking, to keep all transactions from accessing the elements they lock.

Read-only transactions are executed using multiversion timestamping. As the read/write transactions create new versions of a database element, those versions are managed as in Section 18.8.5. A read-only transaction is allowed to read whatever version of a database element is appropriate for its timestamp. A read-only transaction thus never has to abort, and will only rarely be delayed.

### 18.8.7 Exercises for Section 18.8

**Exercise 18.8.1:** Below are several sequences of events, including *start* events, where $st_i$ means that transaction $T_i$ starts. These sequences represent real time, and the timestamp scheduler will allocate timestamps to transactions in the order of their starts. Tell what happens as each executes.

a) $st_1$; $st_2$; $r_1(A)$; $r_2(B)$; $w_2(A)$; $w_1(B)$;

b) $st_1$; $r_1(A)$; $st_2$; $w_2(B)$; $r_2(A)$; $w_1(B)$;

c) $st_1$; $st_2$; $st_3$; $r_1(A)$; $r_2(B)$; $w_1(C)$; $r_3(B)$; $r_3(C)$; $w_2(B)$; $w_3(A)$;

d) $st_1$; $st_3$; $st_2$; $r_1(A)$; $r_2(B)$; $w_1(C)$; $r_3(B)$; $r_3(C)$; $w_2(B)$; $w_3(A)$;

**Exercise 18.8.2:** Tell what happens during the following sequences of events if a multiversion, timestamp scheduler is used. What happens instead, if the scheduler does not maintain multiple versions?

a) $st_1$; $st_2$; $st_3$; $st_4$; $w_1(A)$; $w_2(A)$; $w_3(A)$; $r_2(A)$; $r_4(A)$;

b) $st_1$; $st_2$; $st_3$; $st_4$; $w_1(A)$; $w_3(A)$; $r_4(A)$; $r_2(A)$;

c) $st_1$; $st_2$; $st_3$; $st_4$; $w_1(A)$; $w_4(A)$; $r_3(A)$; $w_2(A)$;

!! **Exercise 18.8.3:** We observed in our study of lock-based schedulers that there are several reasons why transactions that obtain locks could deadlock. Can a timestamp scheduler using the commit bit $C(X)$ have a deadlock?

## 18.9 Concurrency Control by Validation

Validation is another type of optimistic concurrency control, where we allow transactions to access data without locks, and at the appropriate time we check that the transaction has behaved in a serializable manner. Validation differs from timestamping principally in that the scheduler maintains a record of what active transactions are doing, rather than keeping read and write times for all database elements. Just before a transaction starts to write values of database elements, it goes through a "validation phase," where the sets of elements it has read and will write are compared with the write sets of other active transactions. Should there be a risk of physically unrealizable behavior, the transaction is rolled back.

### 18.9.1 Architecture of a Validation-Based Scheduler

When validation is used as the concurrency-control mechanism, the scheduler must be told for each transaction $T$ the sets of database elements $T$ reads and writes, the *read set*, RS($T$), and the *write set*, WS($T$), respectively. Transactions are executed in three phases:

1. *Read.* In the first phase, the transaction reads from the database all the elements in its read set. The transaction also computes in its local address space all the results it is going to write.

2. *Validate.* In the second phase, the scheduler validates the transaction by comparing its read and write sets with those of other transactions. We shall describe the validation process in Section 18.9.2. If validation fails, then the transaction is rolled back; otherwise it proceeds to the third phase.

3. *Write.* In the third phase, the transaction writes to the database its values for the elements in its write set.

Intuitively, we may think of each transaction that successfully validates as executing at the moment that it validates. Thus, the validation-based scheduler has an assumed serial order of the transactions to work with, and it bases its decision to validate or not on whether the transactions' behaviors are consistent with this serial order.

To support the decision whether to validate a transaction, the scheduler maintains three sets:

1. *START*, the set of transactions that have started, but not yet completed validation. For each transaction $T$ in this set, the scheduler maintains START($T$), the time at which $T$ started.

2. *VAL*, the set of transactions that have been validated but not yet finished the writing of phase 3. For each transaction $T$ in this set, the scheduler maintains both START($T$) and VAL($T$), the time at which $T$ validated. Note that VAL($T$) is also the time at which $T$ is imagined to execute in the hypothetical serial order of execution.

3. *FIN*, the set of transactions that have completed phase 3. For these transactions $T$, the scheduler records START($T$), VAL($T$), and FIN($T$), the time at which $T$ finished. In principle this set grows, but as we shall see, we do not have to remember transaction $T$ if FIN($T$) < START($U$) for any active transaction $U$ (i.e., for any $U$ in *START* or *VAL*). The scheduler may thus periodically purge the *FIN* set to keep its size from growing beyond bounds.

## 18.9.2 The Validation Rules

The information of Section 18.9.1 is enough for the scheduler to detect any potential violation of the assumed serial order of the transactions — the order in which the transactions validate. To understand the rules, let us first consider what can be wrong when we try to validate a transaction $T$.

1. Suppose there is a transaction $U$ such that:

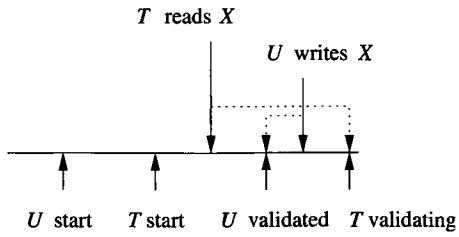


Figure 18.43: $T$ cannot validate if an earlier transaction is now writing something that $T$ should have read

(a) $U$ is in *VAL* or *FIN*; that is, $U$ has validated.

(b) $\text{FIN}(U) > \text{START}(T)$; that is, $U$ did not finish before $T$ started.[11]

(c) $\text{RS}(T) \cap \text{WS}(U)$ is not empty; in particular, let it contain database element $X$.

Then it is possible that $U$ wrote $X$ after $T$ read $X$. In fact, $U$ may not even have written $X$ yet. A situation where $U$ wrote $X$, but not in time is shown in Fig. 18.43. To interpret the figure, note that the dotted lines connect the events in real time with the time at which they would have occurred had transactions been executed at the moment they validated. Since we don't know whether or not $T$ got to read $U$'s value, we must rollback $T$ to avoid a risk that the actions of $T$ and $U$ will not be consistent with the assumed serial order.

2. Suppose there is a transaction $U$ such that:

   (a) $U$ is in *VAL*; i.e., $U$ has successfully validated.

   (b) $\text{FIN}(U) > \text{VAL}(T)$; that is, $U$ did not finish before $T$ entered its validation phase.

   (c) $\text{WS}(T) \cap \text{WS}(U) \neq \emptyset$; in particular, let $X$ be in both write sets.

   Then the potential problem is as shown in Fig. 18.44. $T$ and $U$ must both write values of $X$, and if we let $T$ validate, it is possible that it will write $X$ before $U$ does. Since we cannot be sure, we rollback $T$ to make sure it does not violate the assumed serial order in which it follows $U$.

The two problems described above are the only situations in which a write by $T$ could be physically unrealizable. In Fig. 18.43, if $U$ finished before $T$ started, then surely $T$ would read the value of $X$ that either $U$ or some later transaction wrote. In Fig. 18.44, if $U$ finished before $T$ validated, then surely

[11] Note that if $U$ is in *VAL*, then $U$ has not yet finished when $T$ validates. In that case, $\text{FIN}(U)$ is technically undefined. However, we know it must be larger than $\text{START}(T)$ in this case.

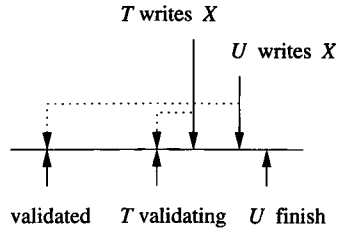


Figure 18.44: $T$ cannot validate if it could then write something ahead of an earlier transaction

$U$ wrote $X$ before $T$ did. We may thus summarize these observations with the following rule for validating a transaction $T$:

- Check that $\text{RS}(T) \cap \text{WS}(U) = \emptyset$ for any previously validated $U$ that did not finish before $T$ started, i.e., if $\text{FIN}(U) > \text{START}(T)$.
- Check that $\text{WS}(T) \cap \text{WS}(U) = \emptyset$ for any previously validated $U$ that did not finish before $T$ validated, i.e., if $\text{FIN}(U) > \text{VAL}(T)$.

**Example 18.29:** Figure 18.45 shows a time line during which four transactions $T$, $U$, $V$, and $W$ attempt to execute and validate. The read and write sets for each transaction are indicated on the diagram. $T$ starts first, although $U$ is the first to validate.

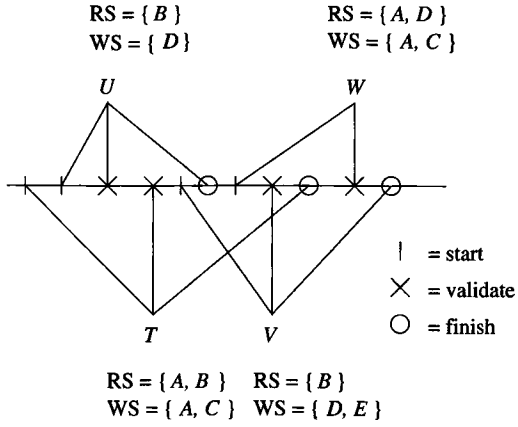


Figure 18.45: Four transactions and their validation

1. Validation of $U$: When $U$ validates there are no other validated transactions, so there is nothing to check. $U$ validates successfully and writes a value for database element $D$.

2. Validation of $T$: When $T$ validates, $U$ is validated but not finished. Thus, we must check that neither the read nor write set of $T$ has anything in common with $\text{WS}(U) = \{D\}$. Since $\text{RS}(T) = \{A, B\}$, and $\text{WS}(T) = \{A, C\}$, both checks are successful, and $T$ validates.

3. Validation of $V$: When $V$ validates, $U$ is validated and finished, and $T$ is validated but not finished. Also, $V$ started before $U$ finished. Thus, we must compare both $\text{RS}(V)$ and $\text{WS}(V)$ against $\text{WS}(T)$, but only $\text{RS}(V)$ needs to be compared against $\text{WS}(U)$. we find:

   - $\text{RS}(V) \cap \text{WS}(T) = \{B\} \cap \{A, C\} = \emptyset$.
   - $\text{WS}(V) \cap \text{WS}(T) = \{D, E\} \cap \{A, C\} = \emptyset$.
   - $\text{RS}(V) \cap \text{WS}(U) = \{B\} \cap \{D\} = \emptyset$.

   Thus, $V$ also validates successfully.

4. Validation of $W$: When $W$ validates, we find that $U$ finished before $W$ started, so no comparison between $W$ and $U$ is performed. $T$ is finished before $W$ validates but did not finish before $W$ started, so we compare only $\text{RS}(W)$ with $\text{WS}(T)$. $V$ is validated but not finished, so we need to compare both $\text{RS}(W)$ and $\text{WS}(W)$ with $\text{WS}(V)$. These tests are:

   - $\text{RS}(W) \cap \text{WS}(T) = \{A, D\} \cap \{A, C\} = \{A\}$.
   - $\text{RS}(W) \cap \text{WS}(V) = \{A, D\} \cap \{D, E\} = \{D\}$.
   - $\text{WS}(W) \cap \text{WS}(V) = \{A, C\} \cap \{D, E\} = \emptyset$.

   Since the intersections are not all empty, $W$ is not validated. Rather, $W$ is rolled back and does not write values for $A$ or $C$.

□

### 18.9.3 Comparison of Three Concurrency-Control Mechanisms

The three approaches to serializability that we have considered — locks, timestamps, and validation — each have their advantages. First, they can be compared for their storage utilization:

- *Locks*: Space in the lock table is proportional to the number of database elements locked.

- *Timestamps*: In a naive implementation, space is needed for read- and write-times with every database element, whether or not it is currently accessed. However, a more careful implementation will treat all timestamps that are prior to the earliest active transaction as "minus infinity" and not record them. In that case, we can store read- and write-times in a table analogous to a lock table, in which only those database elements that have been accessed recently are mentioned at all.

### Just a Moment

You may have been concerned with a tacit notion that validation takes place in a moment, or indivisible instant of time. For example, we imagine that we can decide whether a transaction $U$ has already validated before we start to validate transaction $T$. Could $U$ perhaps finish validating while we are validating $T$?

If we are running on a uniprocessor system, and there is only one scheduler process, we can indeed think of validation and other actions of the scheduler as taking place in an instant of time. The reason is that if the scheduler is validating $T$, then it cannot also be validating $U$, so all during the validation of $T$, the validation status of $U$ cannot change.

If we are running on a multiprocessor, and there are several scheduler processes, then it might be that one is validating $T$ while the other is validating $U$. If so, then we need to rely on whatever synchronization mechanism the multiprocessor system provides to make validation an atomic action.

- *Validation*: Space is used for timestamps and read/write sets for each currently active transaction, plus a few more transactions that finished after some currently active transaction began.

Thus, the amounts of space used by each approach is approximately proportional to the sum over all active transactions of the number of database elements the transaction accesses. Timestamping and validation may use slightly more space because they keep track of certain accesses by recently committed transactions that a lock table would not record. A potential problem with validation is that the write set for a transaction must be known before the writes occur (but after the transaction's local computation has been completed).

We can also compare the methods for their effect on the ability of transactions to complete without delay. The performance of the three methods depends on whether *interaction* among transactions (the likelihood that a transaction will access an element that is also being accessed by a concurrent transaction) is high or low.

- Locking delays transactions but avoids rollbacks, even when interaction is high. Timestamps and validation do not delay transactions, but can cause them to rollback, which is a more serious form of delay and also wastes resources.

- If interference is low, then neither timestamps nor validation will cause many rollbacks, and may be preferable to locking because they generally have lower overhead than a locking scheduler.

- When a rollback is necessary, timestamps catch some problems earlier than validation, which always lets a transaction do all its internal work before considering whether the transaction must rollback.

### 18.9.4 Exercises for Section 18.9

**Exercise 18.9.1:** In the following sequences of events, we use $R_i(X)$ to mean "transaction $T_i$ starts, and its read set is the list of database elements $X$." Also, $V_i$ means "$T_i$ attempts to validate," and $W_i(X)$ means that "$T_i$ finishes, and its write set was $X$." Tell what happens when each sequence is processed by a validation-based scheduler.

a) $R_1(A,B)$; $R_2(B,C)$; $V_1$; $R_3(C,D)$; $V_3$; $W_1(A)$; $V_2$; $W_2(A)$; $W_3(B)$;

b) $R_1(A,B)$; $R_2(B,C)$; $V_1$; $R_3(C,D)$; $V_3$; $W_1(A)$; $V_2$; $W_2(A)$; $W_3(D)$;

c) $R_1(A,B)$; $R_2(B,C)$; $V_1$; $R_3(C,D)$; $V_3$; $W_1(C)$; $V_2$; $W_2(A)$; $W_3(D)$;

d) $R_1(A,B)$; $R_2(B,C)$; $R_3(C)$; $V_1$; $V_2$; $V_3$; $W_1(A)$; $W_2(B)$; $W_3(C)$;

e) $R_1(A,B)$; $R_2(B,C)$; $R_3(C)$; $V_1$; $V_2$; $V_3$; $W_1(C)$; $W_2(B)$; $W_3(A)$;

f) $R_1(A,B)$; $R_2(B,C)$; $R_3(C)$; $V_1$; $V_2$; $V_3$; $W_1(A)$; $W_2(C)$; $W_3(B)$;

## 18.10 Summary of Chapter 18

- ✦ *Consistent Database States*: Database states that obey whatever implied or declared constraints the designers intended are called consistent. It is essential that operations on the database preserve consistency, that is, they turn one consistent database state into another.

- ✦ *Consistency of Concurrent Transactions*: It is normal for several transactions to have access to a database at the same time. Transactions, run in isolation, are assumed to preserve consistency of the database. It is the job of the scheduler to assure that concurrently operating transactions also preserve the consistency of the database.

- ✦ *Schedules*: Transactions are broken into actions, mainly reading and writing from the database. A sequence of these actions from one or more transactions is called a schedule.

- ✦ *Serial Schedules*: If transactions execute one at a time, the schedule is said to be serial.

- ✦ *Serializable Schedules*: A schedule that is equivalent in its effect on the database to some serial schedule is said to be serializable. Interleaving of actions from several transactions is possible in a serializable schedule that is not itself serial, but we must be very careful what sequences of actions

we allow, or an interleaving will leave the database in an inconsistent state.

- *Conflict-Serializability*: A simple-to-test, sufficient condition for serializability is that the schedule can be made serial by a sequence of swaps of adjacent actions without conflicts. Such a schedule is called conflict-serializable. A conflict occurs if we try to swap two actions of the same transaction, or to swap two actions that access the same database element, at least one of which actions is a write.

- *Precedence Graphs*: An easy test for conflict-serializability is to construct a precedence graph for the schedule. Nodes correspond to transactions, and there is an arc $T \rightarrow U$ if some action of $T$ in the schedule conflicts with a later action of $U$. A schedule is conflict-serializable if and only if the precedence graph is acyclic.

- *Locking*: The most common approach to assuring serializable schedules is to lock database elements before accessing them, and to release the lock after finishing access to the element. Locks on an element prevent other transactions from accessing the element.

- *Two-Phase Locking*: Locking by itself does not assure serializability. However, two-phase locking, in which all transactions first enter a phase where they only acquire locks, and then enter a phase where they only release locks, will guarantee serializability.

- *Lock Modes*: To avoid locking out transactions unnecessarily, systems usually use several lock modes, with different rules for each mode about when a lock can be granted. Most common is the system with shared locks for read-only access and exclusive locks for accesses that include writing.

- *Compatibility Matrices*: A compatibility matrix is a useful summary of when it is legal to grant a lock in a certain lock mode, given that there may be other locks, in the same or other modes, on the same element.

- *Update Locks*: A scheduler can allow a transaction that plans to read and then write an element first to take an update lock, and later to upgrade the lock to exclusive. Update locks can be granted when there are already shared locks on the element, but once there, an update lock prevents other locks from being granted on that element.

- *Increment Locks*: For the common case where a transaction wants only to add or subtract a constant from an element, an increment lock is suitable. Increment locks on the same element do not conflict with each other, although they conflict with shared and exclusive locks.

- *Locking Elements With a Granularity Hierarchy*: When both large and small elements — relations, disk blocks, and tuples, perhaps — may need to be locked, a warning system of locks enforces serializability. Transactions place intention locks on large elements to warn other transactions that they plan to access one or more of its subelements.

- *Locking Elements Arranged in a Tree*: If database elements are only accessed by moving down a tree, as in a B-tree index, then a non-two-phase locking strategy can enforce serializability. The rules require a lock to be held on the parent while obtaining a lock on the child, although the lock on the parent can then be released and additional locks taken later.

- *Optimistic Concurrency Control*: Instead of locking, a scheduler can assume transactions will be serializable, and abort a transaction if some potentially nonserializable behavior is seen. This approach, called optimistic, is divided into timestamp-based, and validation-based scheduling.

- *Timestamp-Based Schedulers*: This type of scheduler assigns timestamps to transactions as they begin. Database elements have associated read- and write-times, which are the timestamps of the transactions that most recently performed those actions. If an impossible situation, such as a read by one transaction of a value that was written in that transaction's future is detected, the violating transaction is rolled back, i.e., aborted and restarted.

- *Multiversion Timestamps*: A common technique in practice is for read-only transactions to be scheduled by timestamps, but with multiple versions, where a write of an element does not overwrite earlier values of that element until all transactions that could possibly need the earlier value have finished. Writing transactions are scheduled by conventional locks.

- *Validation-Based Schedulers*: These schedulers validate transactions after they have read everything they need, but before they write. Transactions that have read, or will write, an element that some other transaction is in the process of writing, will have an ambiguous result, so the transaction is not validated. A transaction that fails to validate is rolled back.

## 18.11 References for Chapter 18

The book [6] is an important source for material on scheduling, as well as locking. [3] is another important source. Two recent surveys of concurrency control are [12] and [11].

Probably the most significant paper in the field of transaction processing is [4] on two-phase locking. The warning protocol for hierarchies of granularity is from [5]. Non-two-phase locking for trees is from [10]. The compatibility matrix was introduced to study behavior of lock modes in [7].

Timestamps as a concurrency control method appeared in [2] and [1]. Scheduling by validation is from [8]. The use of multiple versions was studied by [9].

1. P. A. Bernstein and N. Goodman, "Timestamp-based algorithms for concurrency control in distributed database systems," *Intl. Conf. on Very Large Databases*, pp. 285–300, 1980.

2. P. A. Bernstein, N. Goodman, J. B. Rothnie, Jr., and C. H. Papadimitriou, "Analysis of serializability in SDD-1: a system of distributed databases (the fully redundant case)," *IEEE Trans. on Software Engineering* **SE-4**:3 (1978), pp. 154–168.

3. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading MA, 1987.

4. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Comm. ACM* **19**:11 (1976), pp. 624–633.

5. J. N. Gray, F. Putzolo, and I. L. Traiger, "Granularity of locks and degrees of consistency in a shared data base," in G. M. Nijssen (ed.), *Modeling in Data Base Management Systems*, North Holland, Amsterdam, 1976.

6. J. N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, San Francisco, 1993.

7. H. F. Korth, "Locking primitives in a database system," *J. ACM* **30**:1 (1983), pp. 55–79.

8. H.-T. Kung and J. T. Robinson, "Optimistic concurrency control," *ACM Trans. on Database Systems* **6**:2 (1981), pp. 312–326.

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

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

11. A. Thomasian, "Concurrency control: methods, performance, and analysis," *Computing Surveys* **30**:1 (1998), pp. 70–119.

12. B. Thuraisingham and H.-P. Ko, "Concurrency control in trusted database management systems: a survey," *SIGMOD Record* **22**:4 (1993), pp. 52–60.

# Chapter 19

# More About Transaction Management

In this chapter we cover several issues about transaction management that were not addressed in Chapters 17 or 18. We begin by reconciling the points of view of these two chapters: how do the needs to recover from errors, to allow transactions to abort, and to maintain serializability interact? Then, we discuss the management of deadlocks among transactions, which typically result from several transactions each having to wait for a resource, such as a lock, that is held by another transaction.

Finally, we consider the problems that arise due to "long transactions." There are applications, such as CAD systems or "workflow" systems, in which human and computer processes interact, perhaps over a period of days. These systems, like short-transaction systems such as banking or airline reservations, need to preserve consistency of the database state. However, the concurrency-control methods discussed in Chapter 18 do not work reasonably when locks are held for days, or human decisions are part of a "transaction."

## 19.1 Serializability and Recoverability

In Chapter 17 we discussed the creation of a log and its use to recover the database state when a system crash occurs. We introduced the view of database computation in which values move between nonvolatile disk, volatile main-memory, and the local address space of transactions. The guarantee the various logging methods give is that, should a crash occur, it will be able to reconstruct the actions of the committed transactions on the disk copy of the database. A logging system makes no attempt to support serializability; it will blindly reconstruct a database state, even if it is the result of a nonserializable schedule of actions. In fact, commercial database systems do not always insist on serializability, and in some systems, serializability is enforced only on explicit

request of the user.

On the other hand, Chapter 18 talked about serializability only. Schedulers designed according to the principles of that chapter may do things that the log manager cannot tolerate. For instance, there is nothing in the serializability definition that forbids a transaction with a lock on an element $A$ from writing a new value of $A$ into the database before committing, and thus violating a rule of the logging policy. Worse, a transaction might write into the database and then abort without undoing the write, which could easily result in an inconsistent database state, even though there is no system crash and the scheduler theoretically maintains serializability.

### 19.1.1 The Dirty-Data Problem

Recall from Section 6.6.5 that data is "dirty" if it has been written by a transaction that is not yet committed. The dirty data could appear either in the buffers, or on disk, or both; either can cause trouble.

| $T_1$ | $T_2$ | $A$ | $B$ |
|---|---|---|---|
| | | 25 | 25 |
| $l_1(A)$; $r_1(A)$; | | | |
| `A := A+100;` | | | |
| $w_1(A)$; $l_1(B)$; $u_1(A)$; | | 125 | |
| | $l_2(A)$; $r_2(A)$; | | |
| | `A := A*2;` | | |
| | $w_2(A)$; | 250 | |
| | $l_2(B)$ **Denied** | | |
| $r_1(B)$; | | | |
| **Abort**; $u_1(B)$; | | | |
| | $l_2(B)$; $u_2(A)$; $r_2(B)$; | | |
| | `B := B*2;` | | |
| | $w_2(B)$; $u_2(B)$; | | 50 |

Figure 19.1: $T_1$ writes dirty data and then aborts

**Example 19.1:** Let us reconsider the serializable schedule from Fig. 18.13, but suppose that after reading $B$, $T_1$ has to abort for some reason. Then the sequence of events is as in Fig. 19.1. After $T_1$ aborts, the scheduler releases the lock on $B$ that $T_1$ obtained; that step is essential, or else the lock on $B$ would be unavailable to any other transaction, forever.

However, $T_2$ has now read data that does not represent a consistent state of the database. That is, $T_2$ read the value of $A$ that $T_1$ changed, but read the value of $B$ that existed prior to $T_1$'s actions. It doesn't matter in this case whether or not the value 125 for $A$ that $T_1$ created was written to disk or not; $T_2$

gets that value from a buffer, regardless. Because it read an inconsistent state, $T_2$ leaves the database (on disk) with an inconsistent state, where $A \neq B$.

The problem in Fig. 19.1 is that $A$ written by $T_1$ is dirty data, whether it is in a buffer or on disk. The fact that $T_2$ read $A$ and used it in its own calculation makes $T_2$'s actions questionable. As we shall see in Section 19.1.2, it is necessary, if such a situation is allowed to occur, to abort and roll back $T_2$ as well as $T_1$. □

| $T_1$ | $T_2$ | $T_3$ | $A$ | $B$ | $C$ |
|---|---|---|---|---|---|
| 200 | 150 | 175 | RT=0 | RT=0 | RT=0 |
| | | | WT=0 | WT=0 | WT=0 |
| | $w_2(B)$; | | | WT=150 | |
| $r_1(B)$; | | | | | |
| | $r_2(A)$; | | RT=150 | | |
| | | $r_3(C)$; | | | RT=175 |
| | $w_2(C)$; | | | | |
| | **Abort**; | | | WT=0 | |
| | | $w_3(A)$; | WT=175 | | |

Figure 19.2: $T_1$ has read dirty data from $T_2$ and must abort when $T_2$ does

**Example 19.2:** Now, consider Fig. 19.2, which shows a sequence of actions under a timestamp-based scheduler as in Section 18.8. However, we imagine that this scheduler does not use the commit bit that was introduced in Section 18.8.1. Recall that the purpose of this bit is to prevent a value that was written by an uncommitted transaction to be read by another transaction. Thus, when $T_1$ reads $B$ at the second step, there is no commit-bit check to tell $T_1$ to delay. $T_1$ can proceed and could even write to disk and commit; we have not shown further details of what $T_1$ does.

Eventually, $T_2$ tries to write $C$ in a physically unrealizable way, and $T_2$ aborts. The effect of $T_2$'s prior write of $B$ is cancelled; the value and write-time of $B$ is reset to what it was before $T_2$ wrote. Yet $T_1$ has been allowed to use this cancelled value of $B$ and can do anything with it, such as using it to compute new values of $A$, $B$, and/or $C$ and writing them to disk. Thus, $T_1$, having read a dirty value of $B$, can cause an inconsistent database state. Note that, had the commit bit been recorded and used, the read $r_1(B)$ at step (2) would have been delayed, and not allowed to occur until after $T_2$ aborted and the value of $B$ had been restored to its previous (presumably committed) value. □

### 19.1.2 Cascading Rollback

As we see from the examples above, if dirty data is available to transactions, then we sometimes have to perform a *cascading rollback*. That is, when a