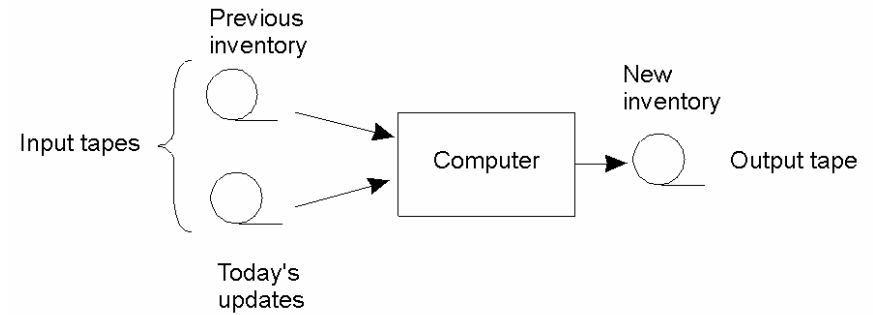


Distributed Transactions

The Transaction Model (1)



Updating a master tape is fault tolerant.

The Transaction Model (2)

| Primitive | Description |
|-------------------|---|
| BEGIN_TRANSACTION | Make the start of a transaction |
| END_TRANSACTION | Terminate the transaction and try to commit |
| ABORT_TRANSACTION | Kill the transaction and restore the old values |
| READ | Read data from a file, a table, or otherwise |
| WRITE | Write data to a file, a table, or otherwise |

Examples of primitives for transactions.

The Transaction Model (3)

```
BEGIN_TRANSACTION
reserve WP -> JFK;
reserve JFK -> Nairobi;
reserve Nairobi -> Malindi;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
reserve WP -> JFK;
reserve JFK -> Nairobi;
reserve Nairobi -> Malindi full =>
ABORT_TRANSACTION
```

(b)

- a) Transaction to reserve three flights commits
- b) Transaction aborts when third flight is unavailable

ACID Properties

- A transaction is a collection of operations on the state of an object (database, object composition, etc.) that satisfies the following properties:
- Atomicity: All operations either succeed, or all of them fail. When the transaction fails, the state of the object will remain unaffected by the transaction.
- Consistency: A transaction establishes a valid state transition. This does not exclude the possibility of invalid, intermediate states during the transaction's execution.
- Isolation: Concurrent transactions do not interfere with each other. It appears to each transaction T that other transactions occur either before T, or after T, but never both.
- Durability: After the execution of a transaction, its effects are made permanent: changes to the state survive failures.

6/23/2005

5

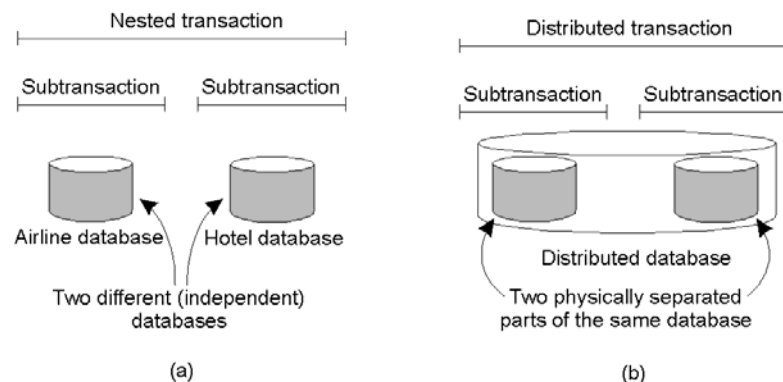
Classification of Transactions

- Flat transactions: The most familiar one: a sequence of operations that satisfies the ACID properties.
- Nested transactions: A hierarchy of transactions that allows
 - concurrent processing of subtransactions, and
 - recovery per subtransaction.
- Distributed transactions: A (flat) transaction that is executed on distributed data, often implemented as a two-level nested transaction with one subtransaction per node.

6/23/2005

6

Distributed Transactions



(a) A nested transaction

(b) A distributed transaction

6/23/2005

7

Flat Transactions: Limitations

- Flat transactions constitute a very simple and clean model for dealing with a sequence of operations that satisfies the ACID properties.
- However, after a series of successful operations all changes should be undone in the case of failure. Sometimes unnecessary:
- Example 1: Trip planning:
 - Plan a intercontinental trip where all flights have been reserved, but filling in the last part requires some "experimentation." The first reservations are known to be in order, but cannot yet be committed.
- Example 2: Bulk updates:
 - When updating bank accounts for monthly interests we have to lock the entire database (every account should be updated exactly once: it is a transaction over the entire database.)
 - Better: each update is immediately committed. However, in the case of failure, we'll have to be able to continue where we left off.

6/23/2005

8

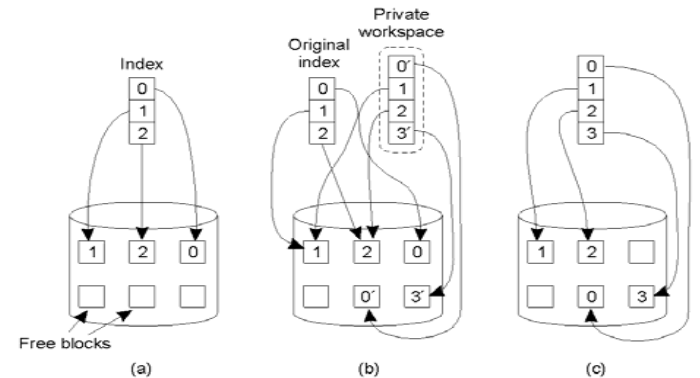
Implementing Transactions

- Use a **private workspace**, by which the client gets its own copy of the (part of the) database. When things go wrong delete copy, otherwise commit the changes to the original.
- Use a **writeahead log** in which changes are recorded allowing you to roll back when things go wrong

6/23/2005

9

Private Workspace



- The file index and disk blocks for a three-block file
- The situation after a transaction has modified block 0 and appended block 3
- After committing

6/23/2005

10

Writeahead Log

| | | | |
|--------------------|-------------|-------------|-------------|
| x = 0; | Log | Log | Log |
| y = 0; | | | |
| BEGIN_TRANSACTION; | | | |
| x = x + 1; | [x = 0 / 1] | [x = 0 / 1] | [x = 0 / 1] |
| y = y + 2 | | [y = 0/2] | [y = 0/2] |
| x = y * y; | | | [x = 1/4] |
| END_TRANSACTION; | | | |
| (a) | (b) | (c) | (d) |

- A transaction
- d) The log before each statement is executed

6/23/2005

11

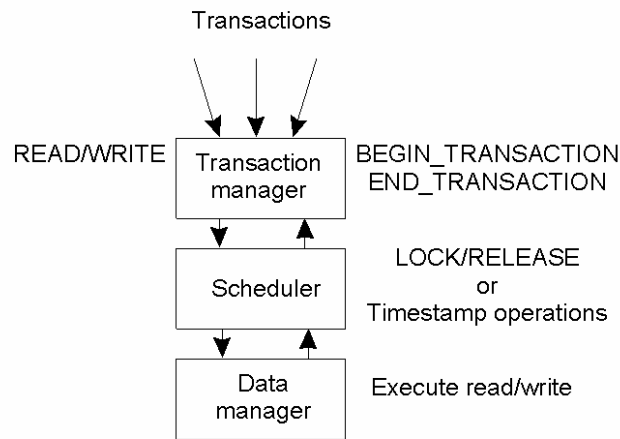
Concurrency Control (1)

- Concurrent transactions – transactions that are executed at the same time on shared data
- Increases efficiency by allowing several transactions to execute at the same time
- Constraint: Effect should be the same as if the transactions were executed in some serial order
- Concurrency control – provides the basic properties of consistency and isolation by properly controlling execution of concurrent transactions

6/23/2005

12

Concurrency Control (2)

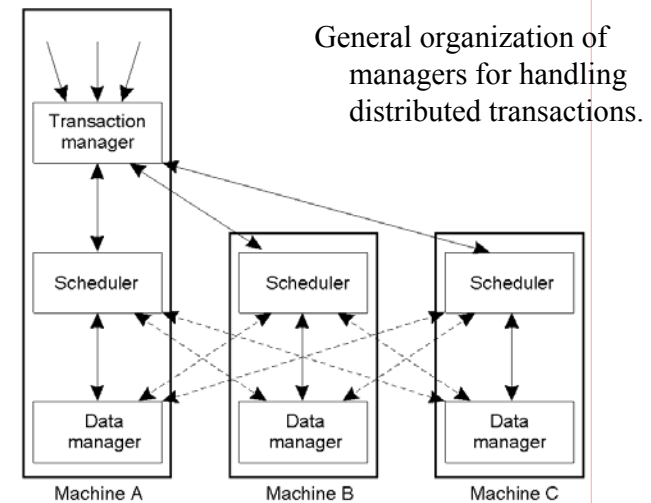


General organization of managers for handling transactions.

6/23/2005

13

Concurrency Control (3)



6/23/2005

14

Serializability (1)

- Consider a collection E of transactions $T_1 \dots T_n$. Goal is to conduct a **serializable execution** of E :
- Transactions in E are possibly concurrently executed according to some schedule S .
- Schedule S is equivalent to some *totally ordered* execution of $T_1 \dots T_n$.

| | | |
|--|--|--|
| BEGIN_TRANSACTION $x = 0;$ $x = x + 1;$ END_TRANSACTION | BEGIN_TRANSACTION $x = 0;$ $x = x + 2;$ END_TRANSACTION | BEGIN_TRANSACTION $x = 0;$ $x = x + 3;$ END_TRANSACTION |
|--|--|--|

(a) Transaction-1

(b) Transaction-2

(c) Transaction-3

| | | |
|------------|---|---------|
| Schedule 1 | $x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3$ | Legal |
| Schedule 2 | $x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;$ | Legal |
| Schedule 3 | $x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;$ | Illegal |

(d) Possible Schedules

6/23/2005

15

Serializability (2)

- Because we're not concerned with the computations of each transaction, a transaction can be modeled as a log of read and write operations.
- Two operations $OPER(T_i, x)$ and $OPER(T_j, x)$ on the same data item x , and from a set of logs may conflict at a data manager:
 - read-write conflict (rw): One is a read operation while the other is a write operation on x .
 - write-write conflict (ww): Both are write operations on x .
- The important thing is that we process conflicting reads and writes in certain relative orders. This is what concurrency control is all about.

6/23/2005

16

Synchronization Techniques

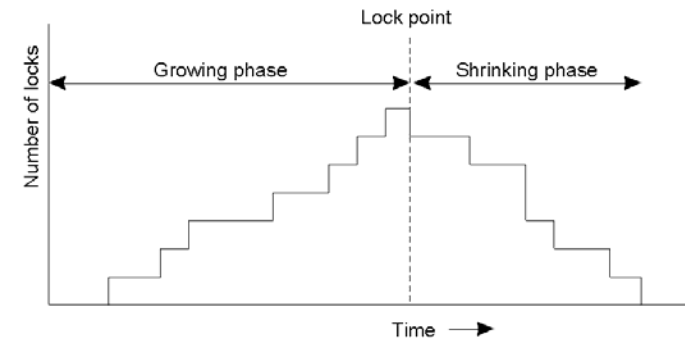
- **Two-phase locking:** Before reading or writing a data item, a lock must be obtained. After a lock is given up, the transaction is not allowed to acquire any more locks.
- **Timestamp ordering:** Operations in a transaction are timestamped, and data managers are forced to handle operations in timestamp order.
- **Optimistic control:** Don't prevent things from going wrong, but correct the situation if conflicts actually did happen. Basic assumption: you can pull it off in most case

6/23/2005

17

Two-Phase Locking (1)

- Clients do only READ and WRITE operations within transactions
- Locks are **granted** and **released** only by scheduler
- Locking policy is to avoid **conflicts** between operations



6/23/2005

18

Two-Phase Locking (2)

- **Rule 1:** When client submits a $OPER(T_i, x)$, scheduler tests whether it conflicts with an operation $OPER(T_j, x)$ from some other client. If no conflict then grant $LOCK(T_i, x)$, otherwise delay execution of $OPER(T_i, x)$. *Conflicting operations are executed in the same order as that locks are granted.*
- **Rule 2:** If $LOCK(T_i, x)$ has been granted, do not release the lock until $OPER(T_i, x)$ has been executed by data manager. *Guarantees LOCK, OPER, RELEASE order.*
- **Rule 3:** If $RELEASE(T_i, x)$ has taken place, no more locks for T_i may be granted. *Combined with rule 1, guarantees that all pairs of conflicting operations of two transactions are done in the same order.*

6/23/2005

19

Two-Phase Locking (3)

- Centralized 2PL: A single site handles all locks
- Primary 2PL: Each data item is assigned a primary site to handle its locks. Data is not necessarily replicated
- Distributed 2PL: Assumes data can be replicated. Each primary is responsible for handling locks for its data, which may reside at remote data managers.

6/23/2005

20

Two-Phase Locking: Problems

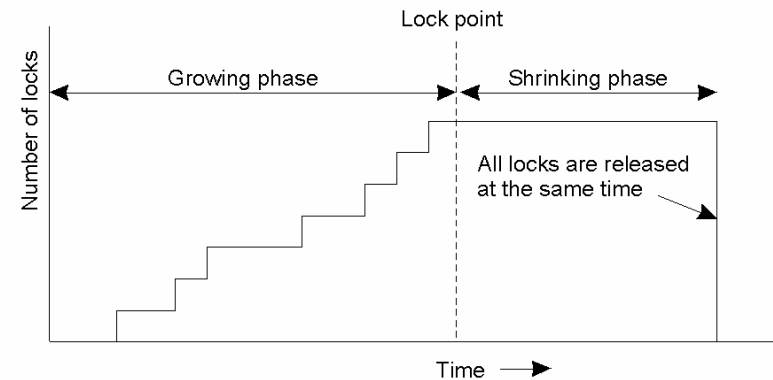
- Problem 1: System can come into a deadlock. Practical solution: put a timeout on locks and abort transaction on expiration.
- Problem 2: When should the scheduler actually release a lock:
 - (1) when operation has been executed
 - (2) when it knows that no more locks will be requested
- No good way of testing condition (2) unless transaction has been committed or aborted.
- Moreover: Assume the following execution sequence takes place: $\text{RELEASE}(T_i, x) \rightarrow \text{LOCK}(T_j, x) \rightarrow \text{ABORT}(T_i)$
- Consequence: Scheduler will have to abort T_j as well (cascaded aborts).
- Solution: Release all locks only at commit/abort time (strict two-phase locking).

6/23/2005

21

Strict Two-Phase Locking

- A transaction always reads value written by a committed transaction – no need to perform cascaded aborts
- All locking operations can be handled transparently



6/23/2005

22

Timestamp Ordering

- Transaction manager assigns a unique timestamp $\text{TS}(T_i)$ to each transaction T_i
- Each operation $\text{OPER}(T_i, x)$ submitted by the transaction manager to the scheduler is timestamped $\text{TS}(\text{OPER}(T_i, x)) \leftarrow \text{TS}(T_i)$
- Scheduler follows the rule:
if $\text{OPER}(T_i, x)$ and $\text{OPER}(T_j, x)$ conflict then
data manager processes $\text{OPER}(T_i, x)$
iff $\text{TS}(\text{OPER}(T_i, x)) < \text{TS}(\text{OPER}(T_j, x))$
- If $\text{TS}(\text{OPER}(T_i, x)) < \text{TS}(\text{OPER}(T_j, x))$, but $\text{OPER}(T_j, x)$ is already processed by the data manager then the scheduler has to reject $\text{OPER}(T_i, x)$
- If $\text{TS}(\text{OPER}(T_i, x)) < \text{TS}(\text{OPER}(T_j, x))$, but $\text{OPER}(T_i, x)$ is already processed by the data manager then the scheduler would submit $\text{OPER}(T_j, x)$ to the data manager

6/23/2005

23

Optimistic Timestamp Ordering

- Observation: (1) Maintaining locks costs a lot; (2) in practice not many conflicts.
- Alternative: Go ahead immediately with all operations, use tentative writes everywhere (shadow copies), and solve conflicts later on.
- Phases: (1) allow operations tentatively; (2) validate effects; (3) make updates permanent
- Validation: Check two basic rules for each of active transactions T_i and T_j (abort if one of the rules does not hold):
 - Rule 1: T_i must not read or write data that has been written by T_j
 - Rule 2: T_j must not read or write data that has been written by T_i

6/23/2005

24