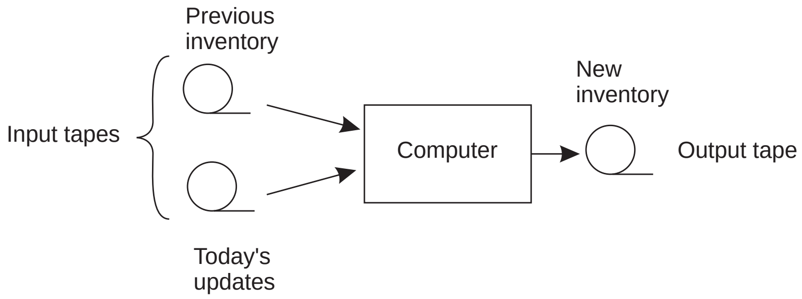
Distributed Systems
Synchronization (II)

[2] **Distributed Transactions**

1. The transaction model
   - ACID properties
2. Classification of transactions
   - flat transactions,
   - nested transactions,
   - distributed transactions.
3. Concurrency control
   - serializability,
   - synchronization techniques
     - two-phase locking,
     - pessimistic timestamp ordering,
     - optimistic timestamp ordering.

[3] **The Transaction Model (1)**



Updating a master tape is fault tolerant.

[4] **The Transaction Model (2)**

Primitive	Description
BEGIN_TRANSACTION	Mark 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.

[5] **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;
   Nairobi → Malindi full
ABORT_TRANSACTION
                (b)
```

a. transaction to reserve three flights commits,

b. transaction aborts when third flight is unavailable.

[6] **ACID Properties**

Transaction
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.

[7] **Transaction Classification**

Flat transactions
The most familiar one: a sequence of operations that satisfies the ACID properties.

Nested transactions
A hierarchy of transactions that allows (1) concurrent processing of subtransactions, and (2) recovery per subtransaction.

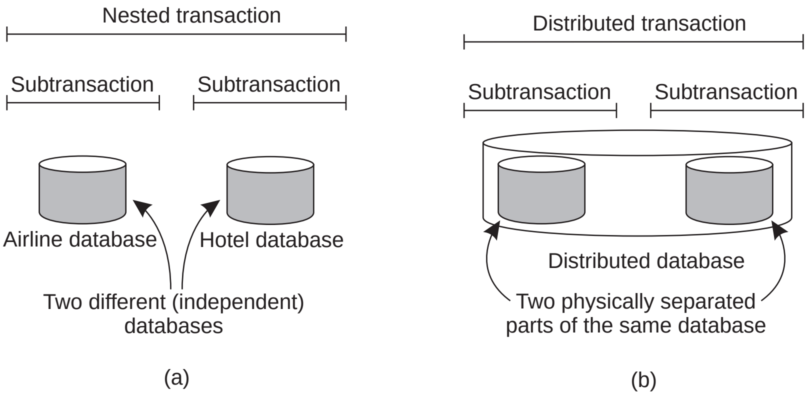
Distributed transactions
A (flat) transaction that is executed on distributed data. Often implemented as a two-level nested transaction with one subtransaction per node.

[8] **Flat Transactions – Limitations**

- they do not allow partial results to be committed or aborted,
- the strength of the atomicity property of a flat transaction also is partly its weakness,
- solution: usage of nested transactions,
- difficult scenarios:
  - subtransaction committed but the higher-level transaction aborted,
  - if a subtransaction commits and a new subtransaction is started, the second one has to have available results of the first one.

[9] **Distributed Transactions**

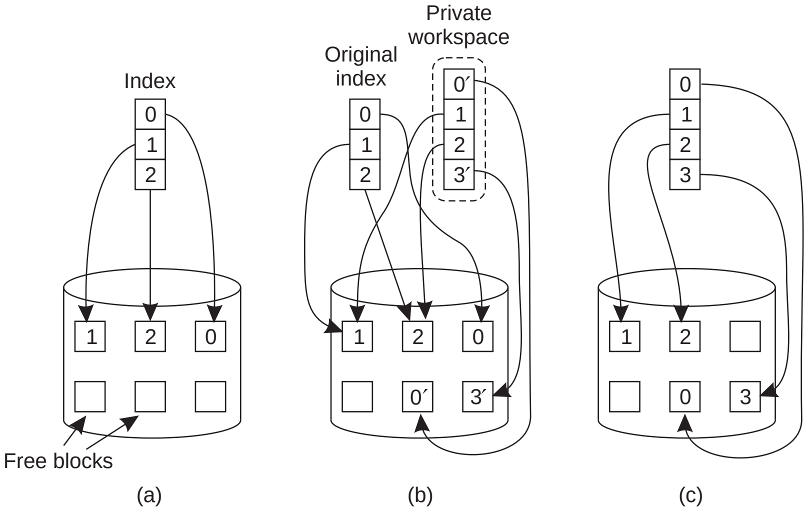
- **nested transaction** is logically decomposed into a hierarchy of subtransactions,
- **distributed transaction** is logically flat, indivisible transaction that operates on distributed data. Separate distributed algorithms required for (1) handling the locking of data and (2) committing the entire transaction.



[10] **Transaction Implementation**

1. **private workspace**
   - 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,
   - optimization by not getting everything.
2. **write-ahead log**
   - use a writeahead log in which changes are recorded allowing you to roll back when things go wrong.

[11] **TransImpl: Private Workspace**



a. The file index and disk blocks for a three-block file,

b. The situation after a transaction has modified block 0 and appended block 3,

c. After committing.

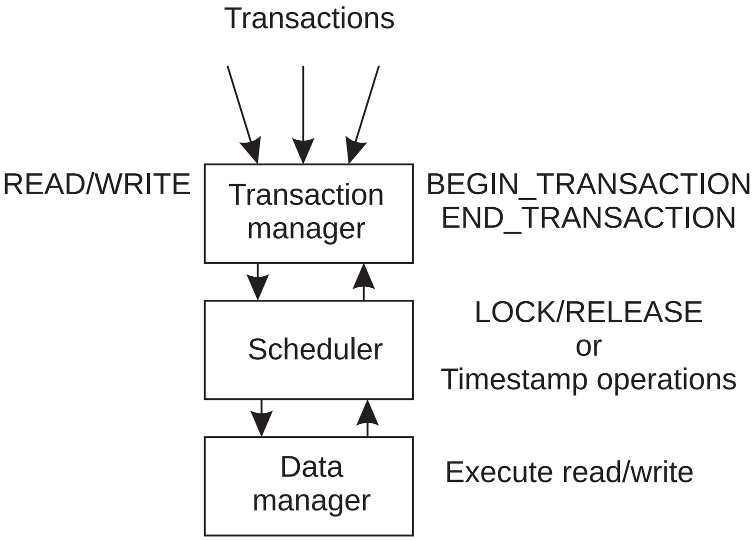
[12] **TransImpl: Writeahead Log**

```
x = 0;
y = 0;                             Log          Log          Log
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. A transaction,

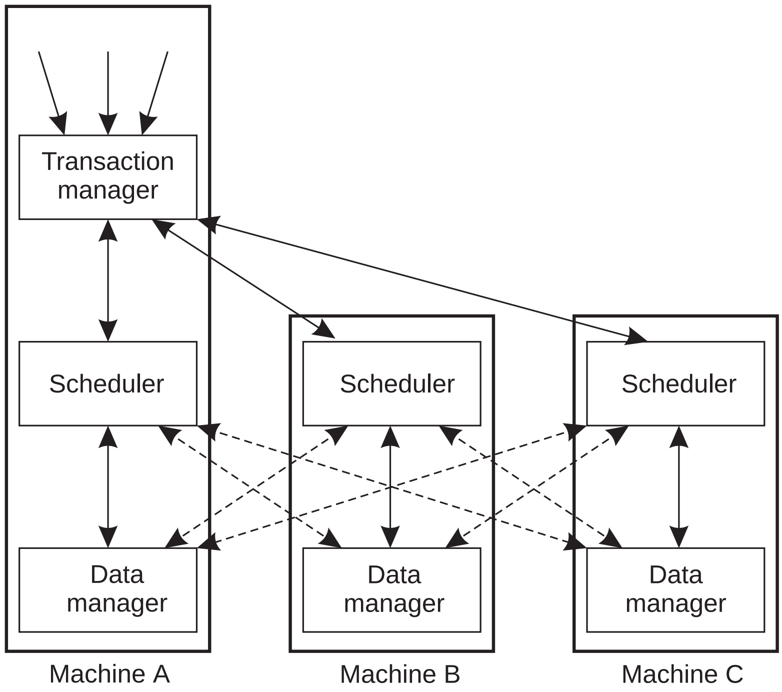
b.-d. The log before each statement is executed.

[13] **Transactions: Concurrency Control (1)**



General organization of managers for handling transactions.

[14] **Transactions: Concurrency Control (2)**



General organization of managers for handling distributed transactions.

[15] **Serializability (1)**

```
BEGIN_TRANSACTION      BEGIN_TRANSACTION      BEGIN_TRANSACTION
   x = 0;                 x = 0;                 x = 0;
   x = x + 1;             x = x + 2;             x = x + 3;
END_TRANSACTION        END_TRANSACTION        END_TRANSACTION

        (a)                    (b)                    (c)
```

Time →

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)

a.-c. Three transactions T1, T2, and T3,

d. Possible schedules.

[16] **Serializability (2)**

Consider a collection E of transactions $T_1, \ldots, T_n$. 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 $T_1, \ldots, T_n$.

Because we are 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.

[17] **Synchronization Techniques**

1. **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.
2. **Timestamp ordering**
   Operations in a transaction are time-stamped, and data managers are forced to handle operations in timestamp order.
3. **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 cases.

[18] **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.

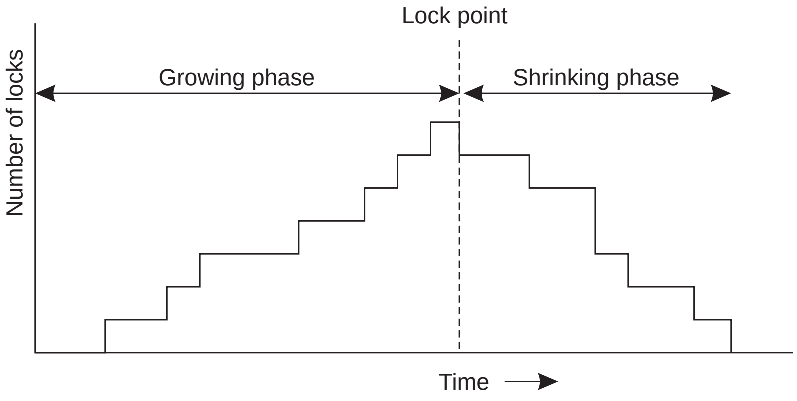
1. When client submits $OPER(T_i, x)$, scheduler tests whether it conflicts with an operation $OPER(T_j, x)$ from any other client. If no conflict then grant $LOCK(T_i, x)$, otherwise delay execution of $OPER(T_i, x)$.

   - conflicting operations are executed in the same order as that locks are granted.

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.

3. If $RELEASE(T_i, x)$ has taken place, no more locks for T_i may be granted.

[19] **Two-Phase Locking (2)**



Two-phase locking.

[20] **Two-Phase Locking (3)**

Types of **2PL**

Centralized 2PL A single site handles all locks,

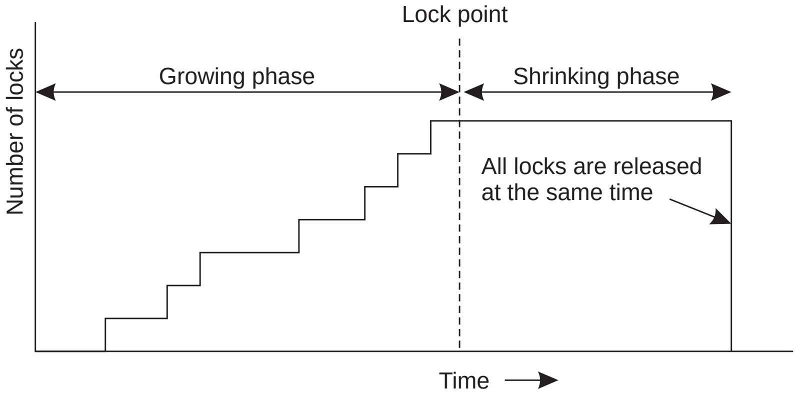
Primary 2PL Each data item is assigned a primary site to handle its locks. Data is not necessarily replicated,

Distributed 2PL Assumes data can be replicated. Each primary is responsible for handling locks for its data, which may reside at remote data managers.

Problems:

- **deadlock** possible – order of acquiring, deadlock detection, a timeout scheme,
- **cascaded aborts** – strict two-phase locking.

## [21] 2PL: Strict 2PL

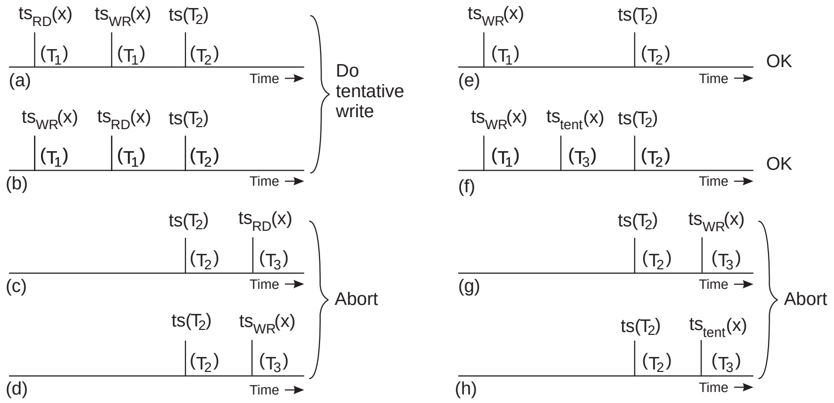


Strict two-phase locking.

[22] Pessimistic Timestamp Ordering (1)

- each transaction T has a timestamp $ts(T)$ assigned,
- timestamps are unique (Lamport's algorithm),
- every operation, part of T, timestamped with $ts(T)$,
- every data item x has a read timestamp $ts_{RD}(x)$ and a write timestamp $ts_{WR}(x)$,
- if operations conflicts, the data manager processes the one with the lowest timestamp,
- comparing to locking (like 2PL): aborts possible but deadlock free.

[23] Pessimistic Timestamp Ordering (2)



(a)-(d) $T2$ is trying to write an item, (e)-(f) $T2$ is trying to read an item.

[24] **Optimistic Timestamp Ordering**

Assumptions:

- conflicts are relatively rare,
- go ahead and do whatever you want, solve conflicts later on,
- keep track of which data items have been read and written (private workspaces, shadow copies),
- check possible conflicts at the time of committing.

Features:

- deadlock free with maximum parallelism,
- under conditions of heavy load, the probability of failure (and abort) goes up substantially,
- focused on nondistributed systems,
- hardly implemented in commercial or prototype systems.

[25] **MySQL: Transactions (1)**

By default, MySQL runs with autocommit mode enabled. This means that as soon as you execute a statement that updates (modifies) a table, MySQL stores the update on disk.

- `SET AUTOCOMMIT = {0 | 1}`

Start and stop transaction:

- `START TRANSACTION | BEGIN [WORK]`
- `COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]`
- `ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]`

[26] MySQL: Transactions (2)

- If you issue a ROLLBACK statement after updating **a non-transactional table** within a transaction, warning occurs. Changes to transaction-safe tables are rolled back, but not changes to non-transaction-safe tables.
- **InnoDB** – transaction-safe storage engine,
- MySQL uses table-level locking for MyISAM and MEMORY tables, page-level locking for BDB tables, and row-level locking for InnoDB tables.
- **Some statements cannot be rolled back**. In general, these include data definition language (DDL) statements, such as those that create or drop databases, those that create, drop, or alter tables or stored routines.
- **Transactions cannot be nested**. This is a consequence of the implicit COMMIT performed for any current transaction when you issue a START TRANSACTION statement or one of its synonyms.

[27] MySQL: Savepoints

The savepoints syntax:

- `SAVEPOINT identifier`
- `ROLLBACK [WORK] TO SAVEPOINT identifier`
- `RELEASE SAVEPOINT identifier`

Description:

- The **ROLLBACK TO SAVEPOINT** statement rolls back a transaction to the named savepoint. Modifications that the current transaction made to rows after the savepoint was set are undone in the rollback, but InnoDB does not release the row locks that were stored in memory after the savepoint.
- All savepoints of the current transaction are deleted if you execute a COMMIT, or a ROLLBACK that does not name a savepoint.

[28] MySQL: Isolation Levels in InnoDB (1)

Isolation levels:

- `SET [SESSION | GLOBAL] TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE}`
- `SELECT @@global.tx_isolation;`
- `SELECT @@tx_isolation;`
- Suppose that you are running in the default REPEATABLE READ isolation level. When you issue a consistent read (that is, an ordinary SELECT statement), InnoDB gives your transaction a timepoint according to which your query sees the database. If another transaction deletes a row and commits after your timepoint was assigned, you do not see the row as having been deleted. Inserts and updates are treated similarly.

[29] MySQL: Isolation Levels in InnoDB (2)

READ UNCOMMITTED SELECT statements are performed in a non-locking fashion, but a possible earlier version of a record might be used. Thus, using this isolation level, such reads are not consistent. This is also called a *dirty read*. Otherwise, this isolation level works like READ COMMITTED.

READ COMMITTED Consistent reads behave as in other databases: Each consistent read, even within the same transaction, sets and reads its own fresh snapshot.

REPEATABLE READ This is the default isolation level of InnoDB. All consistent reads within the same transaction read the snapshot established by the first such read in that transaction. You can get a fresher snapshot for your queries by committing the current transaction and after that issuing new queries.

SERIALIZABLE This level is like REPEATABLE READ, but InnoDB implicitly commits all plain SELECT statements to SELECT ... LOCK IN SHARE MODE.