Distributed Operating Systems

Synchronization in Databases

dr inż. Adam Kozakiewicz

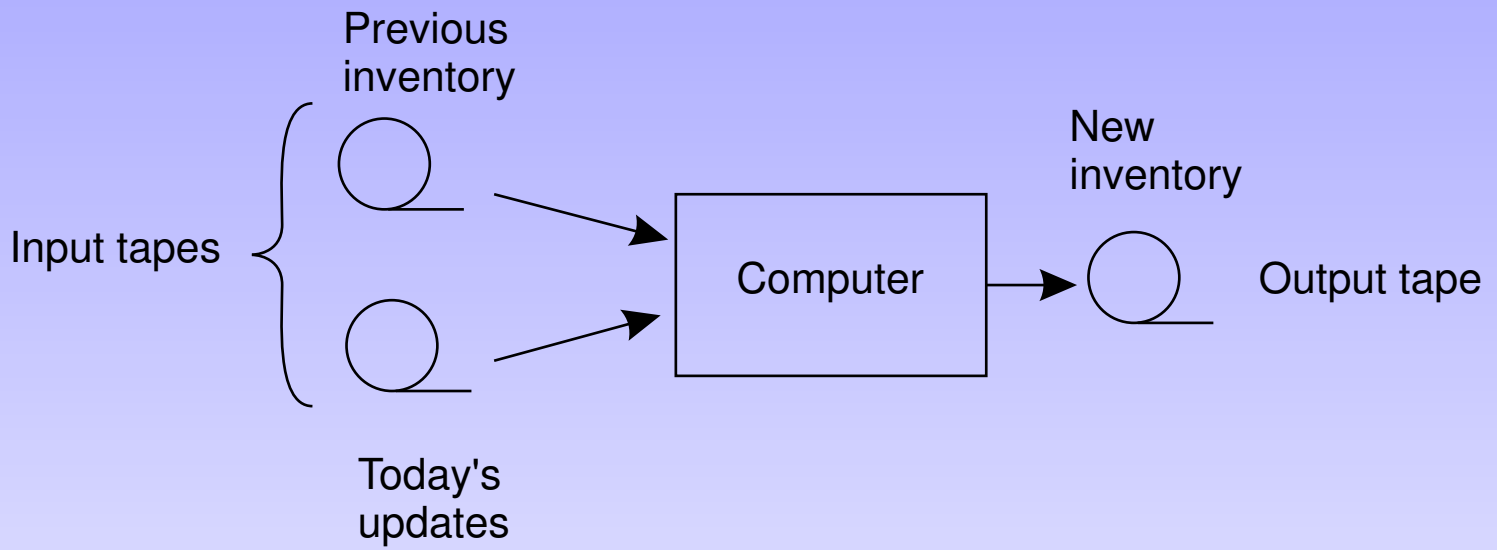
akozakie@ia.pw.edu.pl

Institute of Control and Information Engineering
Warsaw University of Technology

Distributed Transactions

1. 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.

The Transaction Model (1)



A well prepared update operation is fault resistant. Even if something goes wrong, nothing bad happens.

The Transaction Model (2)

Examples of primitives for transactions:

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

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. a transaction to reserve three flights, successful – ends in commit,

b. transaction aborted when the third flight is unavailable, no changes to the database made.

ACID Properties

Transaction

Collection of operations on the state of an object (database or other) that satisfies the following properties:

Atomicity Either **all** operations succeed, or **all** of them fail. Failure of one operation decides that the whole transaction must fail. The object state is unaffected by a failed transaction.

Consistency A transaction establishes a valid state transition, that is given an initially consistent object it will leave it in a consistent state. Intermediate states during the transaction's execution may be invalid.

Isolation Concurrent transactions do not interfere with each other. To each transaction it appears, that other transations occur either before or after it – never both.

Durability A successful transaction has permanent effects – changes to the object's state survive any subsequent failures.

Transaction Classification

Flat transactions

The simplest and most familiar case – a sequence of operations that satisfies the ACID properties.

Nested transactions

A hierarchy of transactions that allows:

- √ concurrent execution of subtransactions,
- √ recovery (rollback) per subtransaction.

Distributed transactions

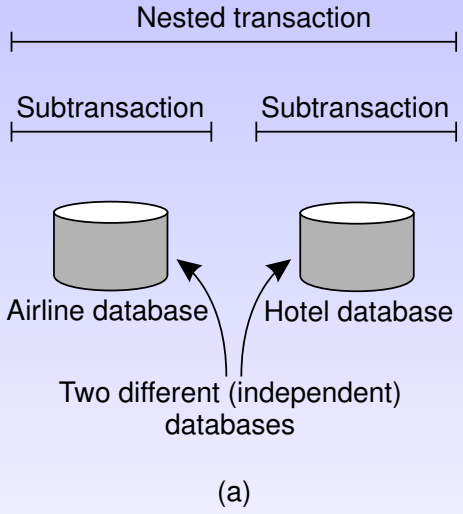
Flat transactions executed on distributed data. Usually implemented as a two-level nested transaction with one subtransaction per node.

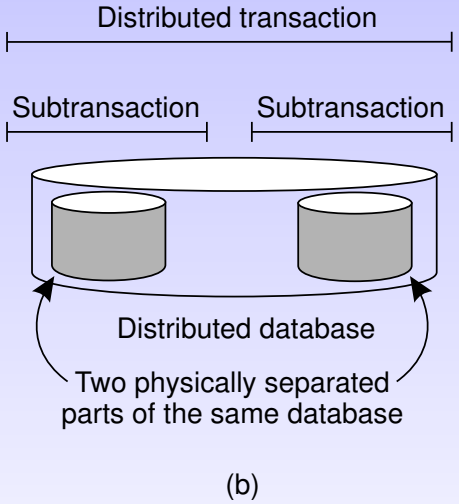
Flat Transactions – Limitations

- √ Partial results cannot be independently committed or aborted.
- √ Strong atomicity may be a weakness.
- √ Solution: nested transactions.
- √ Difficult scenarios:
  - ⋆ Erroneus partial commit – subtransaction is already committed, but the higher-level transaction is subsequently aborted.
  - ⋆ Sequence of subtransactions – if one subtransaction commits and a new one is started, the new one must have available the results of the first one.

Distributed Transactions

- **a nested transaction** is logically decomposed into a hierarchy of subtransactions
- **a distributed transaction** is logically **flat**, indivisible, but it operates on distributed data. Separate distributed algorithms are required to handle the locking of data and committing of the entire transaction.

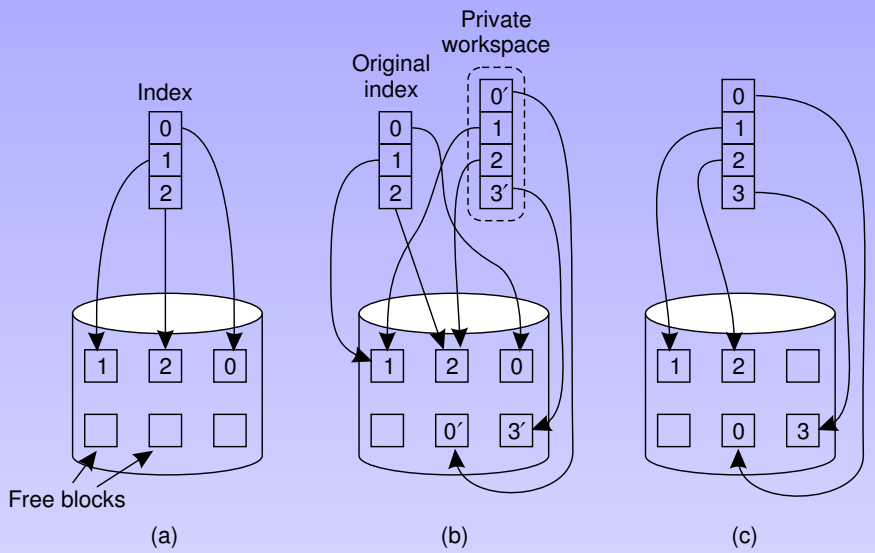




Transactions – Implementation Methods

1. **private workspace**
   - √ each transaction uses a private workspace, with a copy of the part of the database that it uses; commit operation is a simple copy command (or just a change in the index), abort operation is even easier – just delete the workspace,
   - √ only copy what you really need – a lot of room for optimization.
2. **write-ahead log**
   - √ all changes recorded in a write-ahead log, allowing a rollback.

Transactions: Private Workspace



a. The file index and disk blacks for a three-block file.

b. Situation during transaction, after it has modified block 0 and appended block 3.

c. After committing.

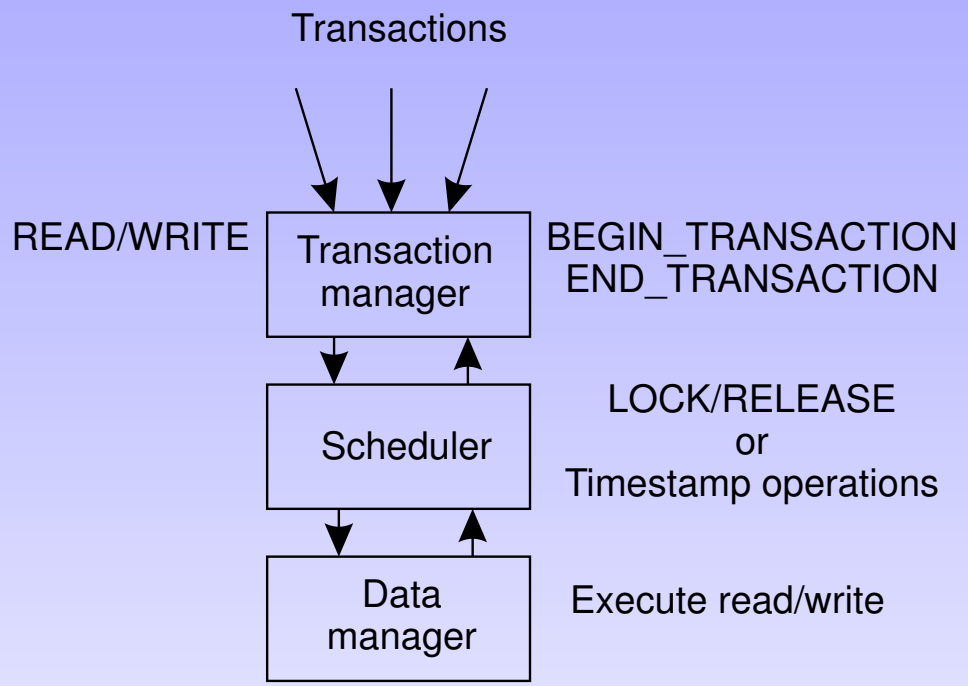
Transactions: Write-Ahead Log

```
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,

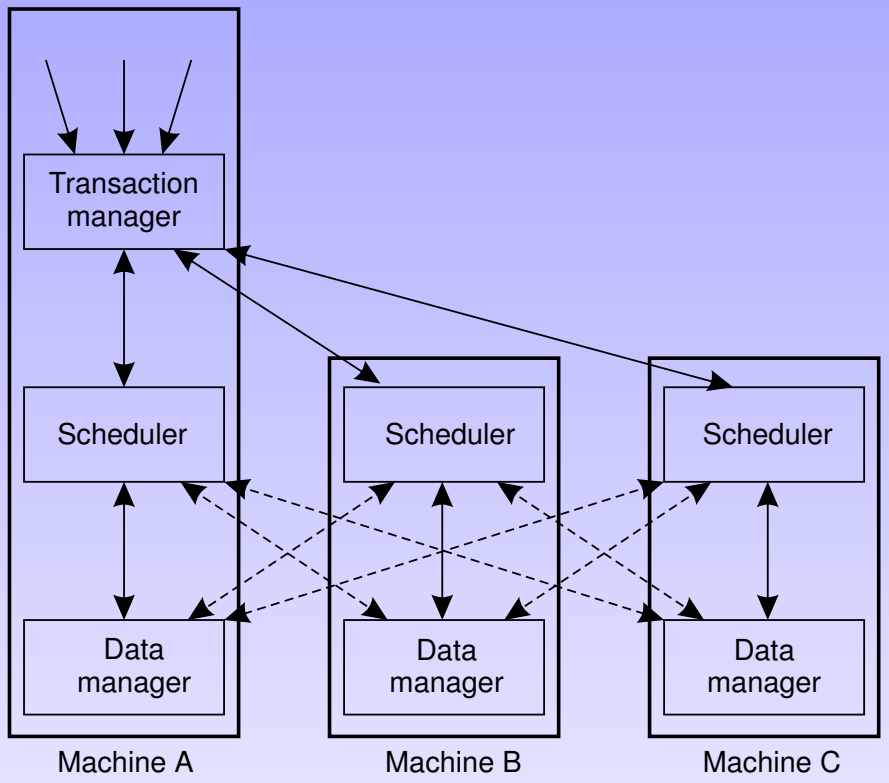
b.-d. The write-ahead log after each operation.

Transactions: Concurrency Control (1)



General organization of managers in a transaction handling mechanism.

Transactions: Concurrency Control (2)



General organization of managers in a distributed transaction handling mechanism.

Serializability (1)

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

(a)

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

(b)

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

(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 i T3,

d. Possible schedules.

Serializability (2)

Consider a collection E of transactions $T_1, \ldots, T_n$. Conduct a serializable execution of E:

- √ transactions in E may be executed concurrently, according to some schedule S,
- √ schedule S is equivalent to some totally ordered execution of $T_1, \ldots, T_n$.

We are not concerned with any computations done within each transaction, so a transaction can be represented as a sequence of read and write operations.

Two operations $OPER(T_i, x)$ and $OPER(T_j, x)$ on the same data item x from two transactions T_i and T_j may **conflict** at a data manager:

read-write conflict (rw) one is a read operation on x and the other is a write operation on x,

write-write conflict (ww) both are write operations on x.

Synchronization Techniques

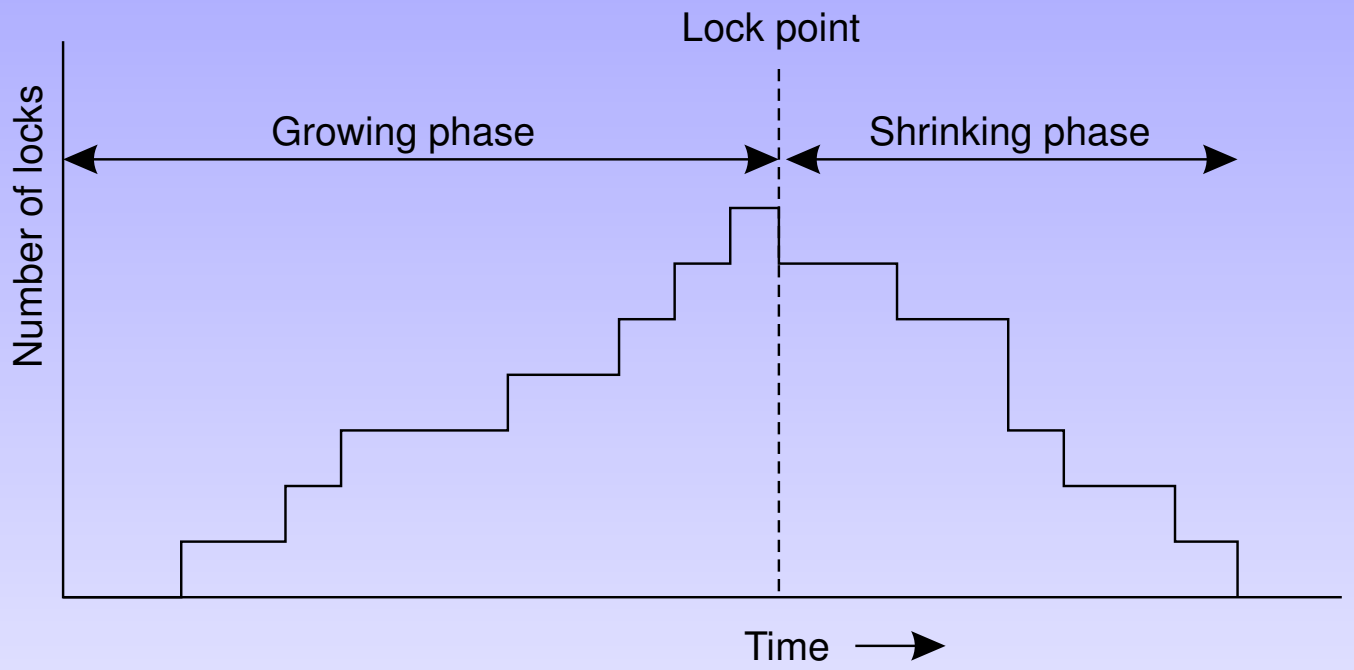
1. **Two-phase locking**
   Before reading or writing a data item, a lock must be obtained. After one lock is given up, the transaction **may not** acquire any more locks.
2. **Pessimistic timestamp ordering**
   Operations in a transaction are time-stamped. Data managers are required to handle operations in timestamp order.
3. **Optimistic control**
   Don't prevent any conflicts, detect them instead, then correct the situation. Optimistic assumption: you can pull it off in most cases (and get away with it without causing a major breakdown). In other words – conflicts must be rare.

Two-Phase Locking (1)

- √ clients only do READ and WRITE operations within transactions,
- √ locks are granted and released only by the scheduler,
- √ locking policy avoids conflicts between operations.

1. When a client submits $OPER(T_i, x)$, the scheduler checks whether it conflicts with any operation $OPER(T_j, x), i \neq j$. If not, a lock $LOCK(T_i, x)$ is granted, otherwise execution of $OPER(T_i, x)$ is delayed.
   - √ conflicting operations are executed in the order of granting locks.
2. If $LOCK(T_i, x)$ has been granted, it may not be released until the data manager has executed $OPER(T_i, x)$.
3. If $RELEASE(T_i, x)$ has taken place, no more locks will be granted to T_i.

Two-Phase Locking (2)



Two-phase locking.

Two-Phase Locking (3)

Types of two-phase locking (2PL):

Centralized 2PL one scheduler handles all locks.

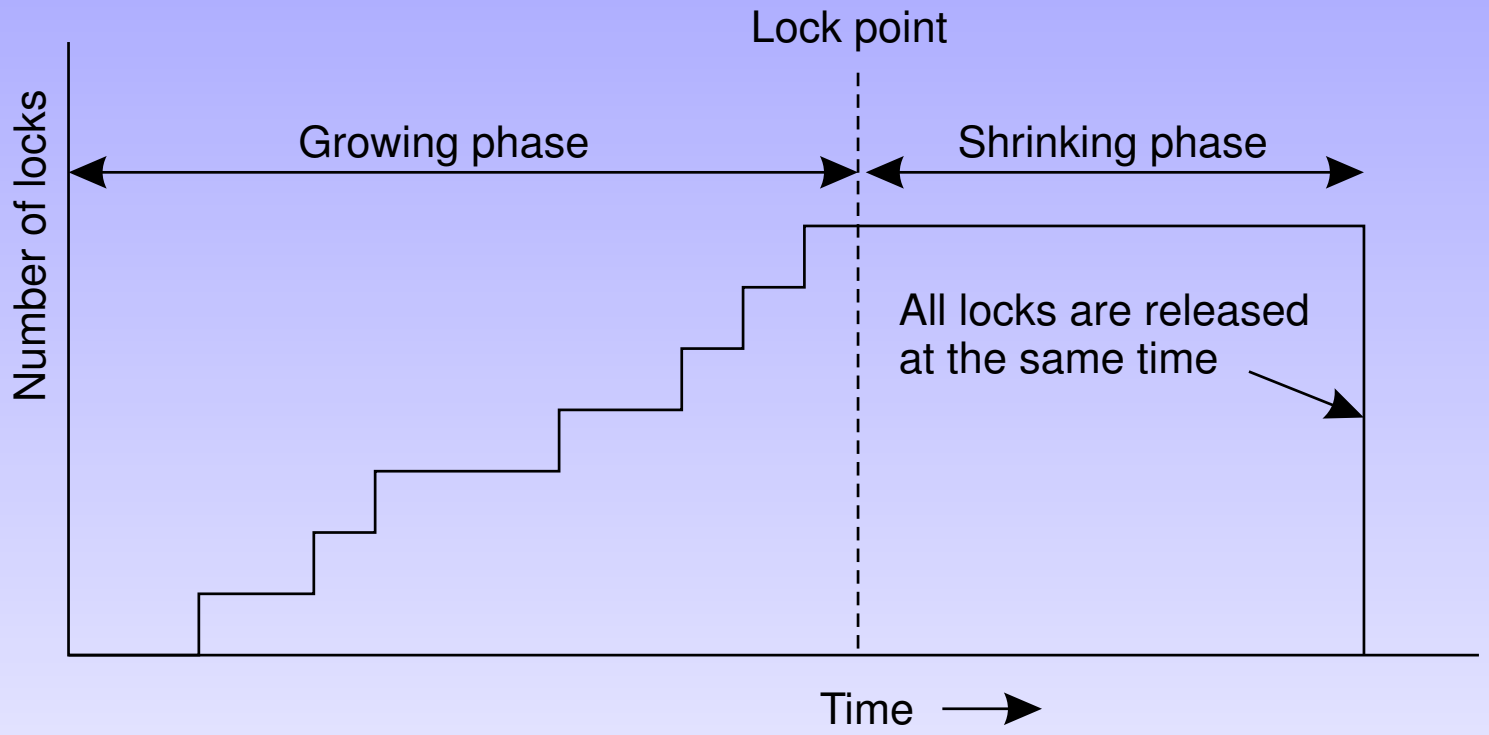
Primary 2PL Each data item x is assigned a primary site (scheduler) to handle its locks. Data is not replicated.

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

Problems:

- √ Possibility of a **deadlock** – solved by ordered acquiring, deadlock detection or timeouts,
- √ **cascaded aborts** – solved by **strict two-phase locking**.

Strict Two-Phase Locking

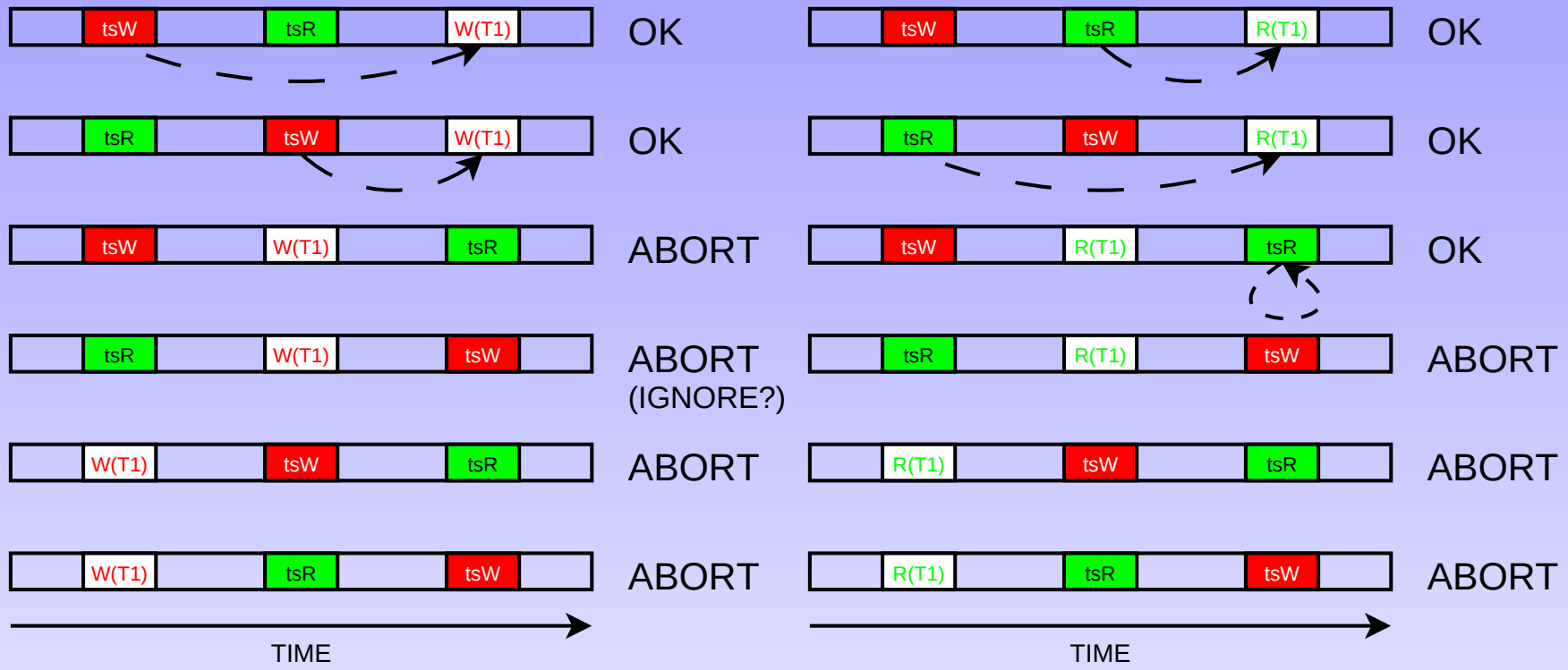


Strict two-phase locking.

Pessimistic Timestamp Ordering (1)

- √ each transaction T has a timestamp $ts(T)$,
- √ timestamps are unique (Lamport's algorithm with client numbers),
- √ each operation within T is stamped with $ts(T)$,
- √ each data item x has a read timestamp $ts_{RD}(x)$ and a write timestamp $ts_{WR}(x)$,
- √ in case of conflict, the operation with the lowest timestamp is processed first,
- √ compared to 2PL: cascading aborts are now possible, but there are no deadlocks.

Pessimistic Timestamp Ordering (2)



Possible orderings of timestamps, excl. equal (same transaction) timestamps

Many variants – revisions, tentative writes, etc.

Optimistic Timestamp Ordering

Assumptions:

- √ conflicts are rare,
- √ just go ahead, do whatever you like, conflicts will be resolved later,
- √ all read and write operations are logged (private workspaces,shadow copies),
- √ on commit attempt, check for possible conflicts.

Features:

- √ no deadlocks, great parallelism,
- √ breaks down easily under load – too many conflicts,
- √ not for distributed systems,
- √ academic approach – hardly implemented in practice, even in prototype systems.

MySQL: Transactions (1)

By default, MySQL runs in autocommit mode – each modification to the database is immediately executed.

- √ `SET AUTOCOMMIT = {0 | 1}`

Using transactions:

- √ `START TRANSACTION [WITH CONSISTENT SNAPSHOT]| BEGIN [WORK]`
- √ `COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]`
- √ `ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]`

MySQL: Transactions (2)

- A `ROLLBACK` command after updating a non-transactional table causes a warning. Changes to transaction-safe tables are rolled back, but the rest stays.
- **InnoDB** – a transaction-safe storage engine for MySQL,
- MySQL uses table-level locking for MyISAM and MEMORY engines, page-level locking for BDB and row-level locking for InnoDB.
- **Not all statements may be rolled back!** This mainly concerns DDL statements (create/drop table, database, alter table, stored routines...)
- **No nested transactions!** `START TRANSACTION` statement and its synonyms invoke `COMMIT` at the beginning.
- `SELECT ... FOR UPDATE` and `SELECT ... LOCK IN SHARE MODE` influence the interation between transactions (in lower isolation levels)

MySQL: Savepoints

Savepoint syntax:

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

- √ `ROLLBACK TO SAVEPOINT` only rolls back to the named savepoint. Row locks (InnoDB) are not released.
- √ `COMMIT` or `ROLLBACK` without a named savepoint delete all savepoints.

MySQL: Isolation Levels in InnoDB (1)

- `SET [SESSION | GLOBAL] TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE}`
- `SELECT @@global.tx_isolation;`
- `SELECT @@tx_isolation;`
- In the default `REPEATABLE READ` isolation level read operations (`SELECT`) create a per-transaction timestamp – from that moment the state is fixed. Modifications from other transactions are invisible.

MySQL: Isolation Levels in InnoDB (2)

READ UNCOMMITTED `SELECT` does not use locks, but it may use an earlier version of the row. Reads are not consistent (*dirty read*). Otherwise its just like `READ COMMITTED`.

READ COMMITTED As in other databases – every read works on its own fresh snapshot.

REPEATABLE READ The default, a consistent snapshot is established by the first read operation. New snapshot requires closing (committing or aborting) the transaction.

SERIALIZABLE Same as `REPEATABLE READ`, but InnoDB executes all `SELECT` operations as `SELECT ... LOCK IN SHARE MODE`.