# Database Concurrency Control and Recovery: Outline

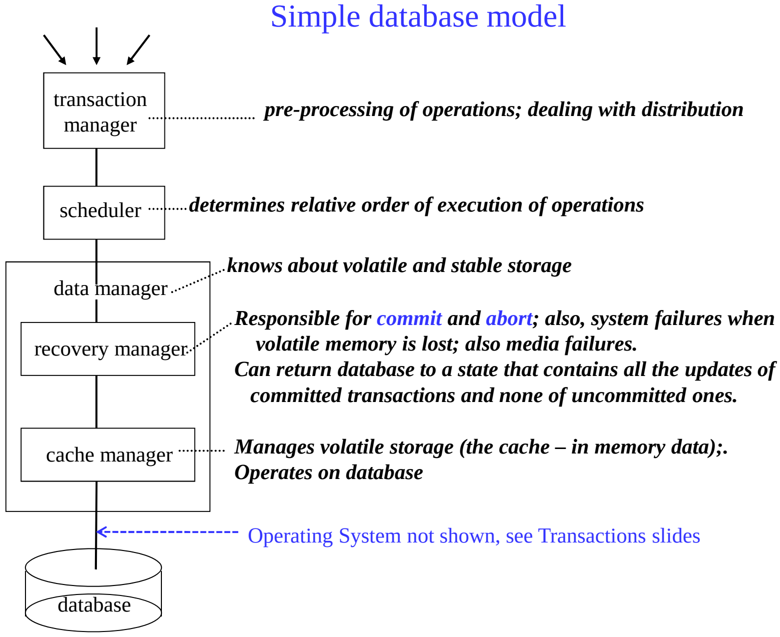
Pessimistic concurrency control
- Two-phase locking (2PL) and Strict 2PL
- Timestamp ordering (TSO) and Strict TSO

Optimistic concurrency control (OCC)
- definition
- validator operation – phases 1 and 2

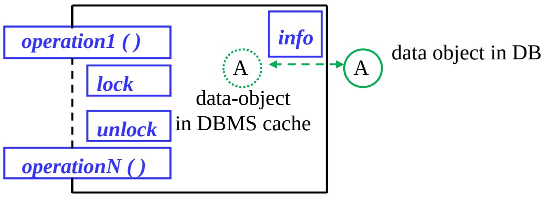
Recovery – see 14



# Simple database model



## Concurrency control: Two-phase locking




Locking all potentially conflicting objects at transaction start reduces concurrency. Also, some of the transaction's objects may be determined dynamically.

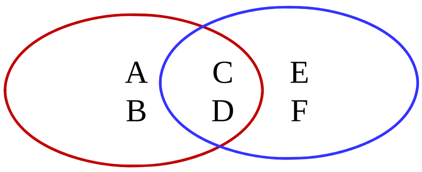
Usually, some form of two-phase locking ( 2PL ) is used:

1. Non-strict 2PL:
   - a) phase of acquiring locks: locks are acquired as the objects are needed
   - b) phase of releasing locks: once all locks have been acquired, locks are released when the object operations complete.
   - ensures a serialisable execution schedule (*serialisation graph cycles* are prevented because locks cannot be released in phase a) ).
   - subject to deadlock (see conditions for deadlock to exist) but a deadlock occurs when the serialisation graph would have had a cycle.
   - subject to cascading aborts
2. Strict 2PL:
   - a) phase of acquiring locks as above
   - b) hold locks and release after ***commit*** – enforces **Isolation** - prevents cascading aborts



## Two-phase locking - visualisation

transaction T1
objects A, B, C, D




transaction T2
objects C, D, E, F

T1 and T2 execute conflicting operations on C and D
suppose T1 locks C and T2 locks D
deadlock is inevitable
but the schedule is not serialisable (neither T1⟶T2 nor T2⟶T1)

1. Non-strict 2PL:
   - a) phase of acquiring locks: locks are acquired as the objects are needed
   - b) phase of releasing locks: once all locks have been acquired, locks are released when the object operations complete.
2. Strict 2PL:
   - a) phase of acquiring locks as above
   - b) hold locks and release after ***commit*** – enforces **Isolation** - prevents cascading aborts

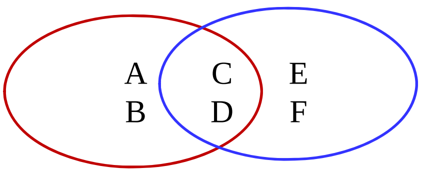
## Concurrency control: Timestamp ordering (TSO)

- Each transaction is allocated a timestamp, e.g. its start time
- An object records the timestamp of the invoking transaction with the info' it holds on the object
- A request for a conflicting operation from a transaction with a **later** timestamp is **accepted**
- A request for a conflicting operation from a transaction with an **earlier** timestamp
  is **rejected** - TOO LATE ! Transaction is aborted and restarted.
  All its operations that have completed must be undone.
- One serialisable order is achieved – that of the transactions' timestamps
- Decisions are based on information local to the objects – transaction IDs and timestamps
- TSO is *not subject to deadlock* – the TSO prevents cycles
- BUT serialisable executions can be rejected – those where concurrent transactions request
  to invoke *all conflicting operations on shared objects in reverse timestamp order*



## Timestamp ordering - visualisation

transaction T1
objects A, B, C, D

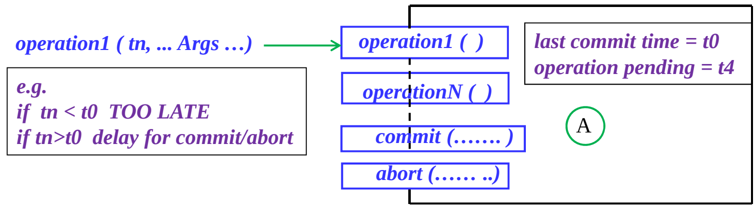



transaction T2
objects C, D, E, F

Suppose T1's timestamp is before T2's (T1 < T2 )
T1 and T2 execute conflicting operations on C and D
Suppose T2 invokes each of C and D before T1
the schedule is serialisable (T2 ⟶ T1)
but both objects C and D reject T1's invocation as "TOO LATE"

## Concurrency control: TSO and Strict TSO

- Cascading aborts are possible with TSO unless **Isolation** is enforced by Strict TSO
- For Strict TSO, objects need a ***commit*** operation, invoked by the transaction manager when it ***commit***s a transaction.
  An invocation can only execute after the previous invocation's result is committed.
  Invocations that are not "TOO LATE" may be delayed.
- TSO and Strict TSO are *not subject to deadlock* – the TSO prevents cycles
- BUT, as with TSO, serialisable executions can be rejected
- TSO and Strict TSO are simple to implement – invocation decisions are local to objects
- Because invocation decisions are local to each object, TSO distributes well






## Optimistic concurrency control (OCC)

In some applications **conflicts are rare**: OCC avoids overhead e.g. locking, and delay.

OCC definition:
At transaction start, or on demand, take a "shadow copy" of all objects invoked by it
Do they represent a **consistent system state**?
How can this be achieved?
NOTE: atomic commitment is part of a pessimistic approach
OCC does not lock all a transaction's objects during ***commit***
NOTE: **Isolation** is enforced – the transaction invokes the shadow objects

The transaction requests ***commit***. The system (validator) must ensure:
1. the transaction's shadow objects were consistent at the start
2. no other transaction has committed an operation at an object that conflicts with one of this committing transaction's invocations.

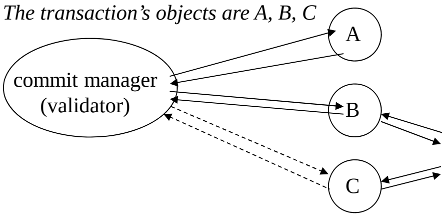
If both of these conditions are satisfied then ***commit*** the updates at the persistent objects in the same order of transactions at every object
If not, ***abort*** – discard the shadow copies and restart the transaction

Used in IBM's IMS Fast Track in the 1980's and improved performance greatly

## OCC: How inconsistent shadows can be taken

At transaction start, or on demand, take a "shadow copy" of all objects invoked by it
- Do they represent a consistent system state?
- How could inconsistent copies be taken?

*e.g.validator* ***commit****s*
*updates for a transaction,*
*creating object versions* ***$T_K$***
*The transaction's objects are A, B, C*




*at this point a new transaction takes* ***shadow copies of B and C***
*B is at version* ***$T_K$***
*C is at some earlier version, e.g.* ***$T_{K-1}$****, or earlier.*
*B and C's shadows represent an inconsistent state*

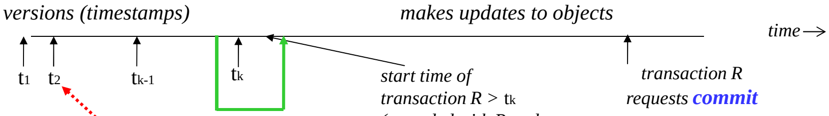


## OCC: Inconsistent shadows – another visualisation

We assume a single centralised validator.
Assume a timestamp $t_n$ is allocated to a transaction by the validator when it decides it can ***commit*** the transaction
Therefore every object has a version number comprising its "most recent timestamp" (transaction timestamp (at ***commit***) that operated most recently on this object)
The validator can use the version numbers of the set of objects (shadows) used by a transaction to decide whether they represent a consistent system state.
Note that the validator has no control over the making of shadow copies.
What it has available is the timestamps of transaction ***commit***s.

*transaction R's objects' versions (timestamps)*

*transaction R's execution phase: makes updates to objects*




*start time of transaction R > $t_k$ (recorded with R and available to validator)*

*transaction R requests* ***commit***

*Transaction Q* ***commit*** *in progress shares some of R's objects e.g. may update object with timestamp $t_2$ giving it a new timestamp $t_k$ but after R has taken the $t_2$ version*

## OCC: Transaction data

| validated transaction | timestamp | objects and updates | all updates acknowledged? |
|---|---|---|---|
| previous transactions | ……. | …….. | ….. |
| P | ti | A, B, C, D, E | Yes |
| Q | ti+1 | B, C, E, F | Yes |
| R | ti+2 | B, C, D | Yes |
| S | ti+3 | A, C, E | No ….. Yes |

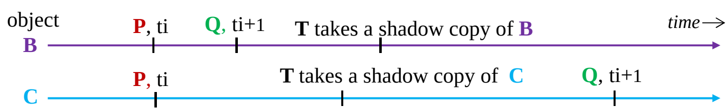
object versions before and after S is committed:

| object | version before S's updates | version after S's updates |
|---|---|---|
| **A** | **P, ti** | **S, ti+3** |
| B | R, ti+2 | R, ti+2 |
| **C** | **R, ti+2** | **S, ti+3** |
| D | R, ti+2 | R, ti+2 |
| **E** | **Q, ti+1** | **S, ti+3** |
| F | Q, ti+1 | Q, ti+1 |

This degree of contention is not expected to occur in practice in systems where OCC is used



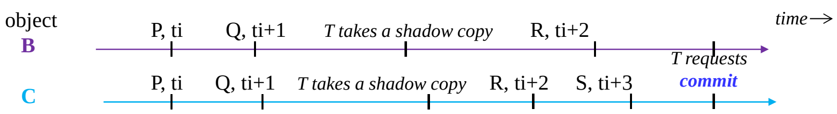
## OCC: Validation phase 1




Transaction **T** operates on objects **B** and **C**
Transaction **P** has completed ***commit*** when **T** takes its shadows of **B** and **C**
Transaction **Q**'s ***commit*** is in progress.
Transaction **T**'s objects' timestamps are **B**, ti+1 committed by **Q**
and **C**, ti committed by **P**
While **T** operates on these versions, **Q** commits its update to **C**

**validation phase 1**: **T** has taken **inconsistent versions** of objects **B** and **C**. ***abort*** **T**.
Transaction restarts and takes new shadows.

## OCC: Validation phases 1 and 2




**validation phase 1**: T has taken consistent versions of objects B, ti+1 and C, ti+1

**phase 2**: during T's execution phase, updates have been committed at B and C.
If any of these **conflict** with T's updates then T is aborted.
If none conflict, T is assigned an update timestamp and its updates are queued for application at the objects B and C.



## Recovery

We give a short overview of how recovery can be implemented:

- Requirements for recovery
- A practical approach to recovery – keep a **recovery log** – must be **write-ahead**
- Example showing system components with values in DB and in-memory cache
- Checkpoint procedure: to aid processing of the very large recovery log
- Transaction categories for recovery
- An algorithm for the recovery manager

## Requirements for Recovery

- Media failure, e.g. disc-head crash.
  Part of persistent store is lost – need to restore it.
  Transactions in progress may be using this area – ***abort*** uncommitted transactions.

- System failure e.g. crash - main memory lost.
  Persistent store is not lost but may have been changed by uncommitted transactions.
  Also, committed transactions' effects may not yet have reached persistent objects.

- Transaction abort
  Need to undo any changes made by the aborted transaction.

Our object model assumed all invocations are recorded with the object.
It was not made clear how this was to be implemented – synchronously in persistent store?
**We need to optimise for performance reasons - not write-out every operation synchronously.**

We consider one method – a **recovery log**. i.e. update data objects in place in persistent store, as and when appropriate, and make a (recovery) log of the updates.



## Recovery Log

1. Assume a periodic (daily?) dump of the database (e.g. Op. Sys. backup)
2. Assume that a record of every change to the database is written to a log
   ***{transaction-ID, data-object-ID, operation (arguments), old value, new value }***
3. If a failure occurs the log can be used by the Recovery manager to REDO or UNDO selected operations. UNDO and REDO must be idempotent (repeatable), e.g. contain before and after values, not just "add 3". Further crashes might occur at any time.

Transaction abort:
UNDO the operations – roll back the transaction

System failure
REDO committed transactions, UNDO uncommitted transactions

Media failure
reload the database from the last dump
REDO the operations of all the transactions that committed since then

But the log is very large to search for this information
so, to assist rapid recovery, take a CHECKPOINT at "small" time intervals
e.g. after 5 mins or after n log items – see 18

## Recovery Log must be “write-ahead”

Two distinct operations:
- write a change to an object in the database
- write the log record of the change

A failure could occur between them – in which order should they be done?
If an object is updated in the database, there is no record of the previous value,
so no means of UNDOing the operation on abort.
*The log must be written first.*

Also, *a transaction is not allowed to* ***commit***
*until the* ***log records*** *for all its operations have been* ***written out*** *to the log.*

Note: we can’t, and needn’t, take time to update in the database on every ***commit***
the (few) objects involved in a transaction.
Note: a log can be written efficiently, because:
- there are enough records from the many transactions in progress at any time,
- the writes are to one place – the log file.

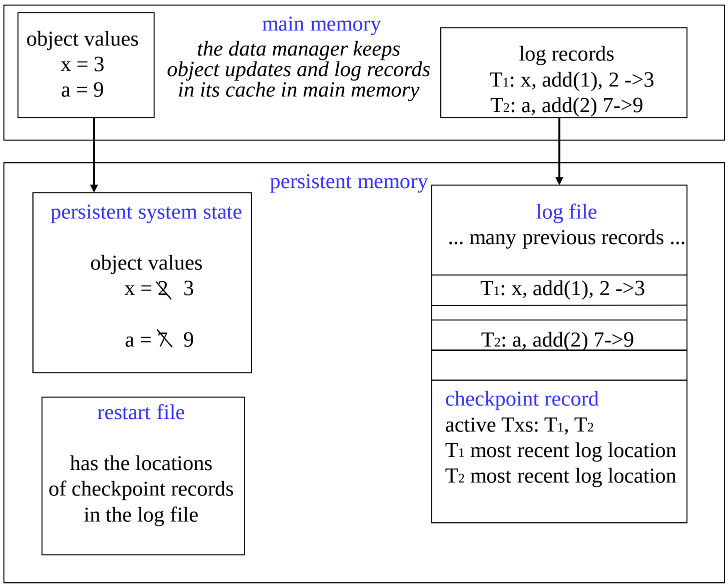


## Checkpoints and the checkpoint procedure

From 16:
The log is very large to search for this information on transactions
especially for abort of a single transaction,
so take a CHECKPOINT at “small” time intervals
e.g. After 5 mins or after n log items.

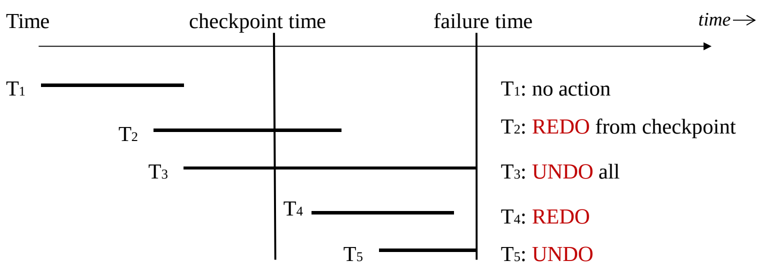
Checkpoint procedure :
- Force-write any **log records** in main memory **out to the log** (OS *must* do this)
- Force-write a checkpoint record to the log, containing:
  - list of all transactions active (started but not committed) at the time of the checkpoint
  - address within the log of each transaction’s most recent log record
  - note: the log records of a given transaction are chained
- Force-write database buffers (database updates still in main memory) out to the database.
- Write the address of the checkpoint record within the log into a restart file.

## A recovery log with a checkpoint record






## Transaction categories for recovery




Checkpoint record says $T_2$ and $T_3$ are active

$T_1$: its log records were written out before ***commit***.

Any remaining DB updates were written out at checkpoint time. No action required.

$T_2$: any updates made after the checkpoint are in the log and can be re-applied (REDO)

$T_4$: log records are written on ***commit*** – can be re-applied (REDO is idempotent)

$T_3$ and $T_5$: any changes that might have been made can be found in the log and previous state recovered (undone using UNDO operation)

$T_3$ requires log to be searched before the checkpoint – checkpoint contains pointer to previous log record.

# Algorithm for recovery manager

Keeps: UNDO list - initially contains all transactions listed in the checkpoint record
REDO list – initially empty

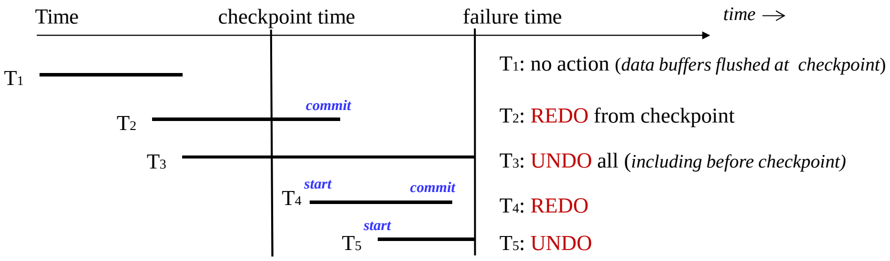
Searches forward through the log starting from the checkpoint record, to the end of the log

- If it finds a ***start-transaction*** record it adds that transaction to the UNDO list
- If it finds a ***commit*** record it moves that transaction from the UNDO list to the REDO list

Then, works backwards through the log
UNDOing transactions on the UNDO list (restores state)
Finally, works forward again through the log
REDOing transactions on the REDO list



# Algorithm for recovery manager




Checkpoint records $T_2$, $T_3$ active
Restart after crash:

| | | | | |
|---|---|---|---|---|
| UNDO list: $T_2$, $T_3$ | add $T_4$ | remove $T_2$ | add $T_5$ | remove $T_4$ |
| REDO list: | | add $T_2$ | | add $T_4$ |

After processing the log from the checkpoint to the end:
UNDO list: $T_3$, $T_5$ (***start*** found but no ***commit***)
REDO list: $T_2$, $T_4$ (***commit*** found)
Work back through the log UNDOing and REDOing these transactions, including UNDO before the checkpoint for $T_3$. $T_2$ was up-to-date at the checkpoint.

Reference for correctness of two-phase locking (pp.486 – 488):
Database System Implementation
Hector Garcia-Molina, Jeffrey Ullman, Jennifer Widom
Prentice-Hall, 2000

References for OCC

Optimistic Concurrency Control
H-T Kung and J T Robinson
ACM Transactions on Database Systems, **6**–2 (1981), 312-326

Apologizing versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types
Maurice Herlihy
ACM Transactions on Database Systems, **15**–1 (1990), 96-124



## Database Concurrency Control and Recovery: Summary

Pessimistic concurrency control
- Two-phase locking (2PL) and Strict 2PL
- Timestamp ordering (TSO) and Strict TSO

Optimistic concurrency control (OCC)
- definition
- validator operation – phases 1 and 2

Recovery using a write-ahead log

## Concurrent Systems Summary

1. Introduction and overview
   Concurrency in and supported by OS. Thread models.

2. Shared memory – low level concurrency control

3. Shared memory – high-level language concurrency control
3a. Lock-free programming, if time allows (not to be examined)

4. Inter-process communication with no shared memory

5. Liveness properties – Deadlock

*
6. Transactions: composite operations on persistent objects

7. Concurrency control and recovery for transaction systems

* (8). FreeBSD case study
  given by Dr Robert Watson