10. Distributed Concurrency Control

ACID properties

- A tomicity: A transaction is executed completely or not at all.
- C onsistency: Consistency constraints defined on the data are preserved.
- I **solation: Each transaction behaves as if it were operating alone on the data.**
- D urability: All effects will survive all software and hardware failures.

Challenges of Distributed/Replicated Data

- Storing copies of data on different nodes enables availability, performance and reliability
- Data needs be consistent
  - **Synchronizing concurrent access**
  - Detecting and recovering from failures
  - **Deadlock management**

Discussion of ACID guarantees

- classical, "all-inclusive" guarantee in (relational) database systems
- solves the problems demonstrated in Examples (and more)
- well-established theory and clear semantics
- mature and well-engineered implementations
  - *Recovery* for *A*, *D*
  - *Concurrency Control* for *I*

We are looking at a fork in the road:

- provide ACID, but limit scalability and availability
- favour scalability and availability, but sacrifice on isolation/consistency: NoSQL, BASE

This chapter will focus ACID consistency

Concurrency Control Refresh

Page Model

- All operations on data will be eventually mapped into read and write operations on pages.
- To study the concurrent execution of transactions it is sufficient to inspect the interleavings of the resulting page operations.
- Independently whether a page resides in cache memory or resides on disk, read and write are considered as indivisible. ~ atomic
- Set of transactions $\mathcal{T} = \{T_1, \ldots, T_n\}$.
- A transaction is given as a sequence of read (R) - and write (W)-actions over database objects $\{A, B, C, \ldots\}$, e.g. no complex update semantics (increase)

$$T_1 = R_1A\ W_1A\ R_1B\ W_1B$$
$$T_2 = R_2A\ W_2A\ R_2B\ W_2B$$
$$T_3 = R_3A\ W_3B$$

Ordering and dependencies within a transaction

- In the basic definition, operations within a transaction are totally ordered.
- Write operations possibly depend on all read inputs seen before:
  - Let WX be the j-th action of transaction T
  - Let $RA_1, \ldots, RA_n$ are the read actions of T being processed in the indicated order before WX.
  - Then the value of X written by T is given by $f_{T,j}(a_1, \ldots, a_n)$, where $f_{T,j}$ is an arbitrary, however unknown function and the a's are the values read in the indicated order by the preceding read actions.

Complete transaction

We call a transaction *complete*, if its first action is begin b and its last action either is commit c or abort a.

Histories

Let $\mathcal{T} = \{T_1, \ldots, T_n\}$ be a (finite) set of complete transactions, where for each T_i we have $T_i = (OP_i, <_i)$.

A *history* of $\mathcal{T}$ is a pair $S = (OP_S, <_S)$, where

- $OP_S = \cup_{i=1}^{n} OP_i$ and $<_S$ a partial order on OP_S such that $<_S \subseteq \cup_{i=1}^{n} <_i$. sorted within each TA
- Let $p, q \in OP_S$, where p and q belong to distinct transactions, however access the same data object. If p or q is a write action, then either $p <_S q$ or $q <_S p$

Schedules

- A *schedule* of $\mathcal{T}$ is a prefix of a history.

$$S_1 = R_1A\ W_1A\ R_3A\ R_1B\ W_1B\ R_2A\ W_2A\ W_3B\ R_2B\ W_2B$$
$$S_2 = R_1A\ W_1A\ R_3A\ R_1B\ W_1B\ R_2A\ W_2A\ W_3B\ R_2B\ W_2B$$
$$S_3 = R_3A\ R_1A\ W_1A\ R_1B\ W_1B\ R_2A\ W_2A\ R_2B\ W_2B\ W_3B$$

- A schedule is called serial, if it is not interleaved. no disturbance

$$S_4 = \underbrace{R_3A\ W_3B}_{T_3}\ \underbrace{R_1A\ W_1A\ R_1B\ W_1B}_{T_1}\ \underbrace{R_2A\ W_2A\ R_2B\ W_2B}_{T_2}$$ slow

Serializability

A schedule is called (conflict-)serializable,[1] if there exists a (conflict-)equivalent serial schedule over the same set of transactions.

<-> Linear order of transaction start / commit

Conflict graph

The conflict graph of a schedule S is given as $G(S) = (V, E)$, where V is the set of transactions in S and the set of edges E is given by the conflicts in S: $T_i \rightarrow T_j \in E$, iff there are conflicting actions $p \in OP_i$, $q \in OP_j$ and $p <_S q$.

- $S = \ldots W_iA \ldots R_jA \ldots \;\Rightarrow\; T_i \rightarrow T_j \in E$, if there is no other write-action to A between W_iA und R_jA in S.
- $S = \ldots W_iA \ldots W_jA \ldots \;\Rightarrow\; T_i \rightarrow T_j \in E$, if there is no other write-action to A between W_iA und W_jA in S.
- $\widehat{S} = \ldots R_iA \ldots W_jA \ldots \;\Rightarrow\; T_i \rightarrow T_j \in E$, if there is no other write-action to A between R_iA und W_jA in S.

Serializability Testing

A schedule is serializable iff its conflict graph is acyclic.

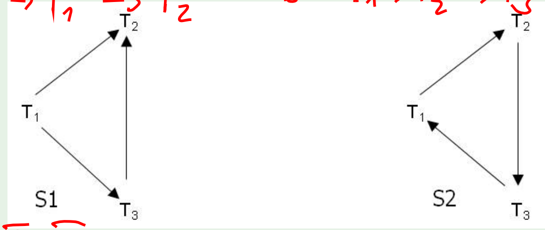
[1] We consider only conflict-serializability and therefore talk about serializability in the sequel, for short.

A: $T_3 - T_2$ $T_1 \to T_3$

Example   A: $T_1 \to T_3$ $T_1 \to T_2$

Schedule S_1: $R_1A\ W_1A\ R_3A\ R_1B\ W_1B\ R_2A\ W_2A\ W_3B\ R_2B\ W_2B$
Schedule S_2: $R_3A\ R_1A\ W_1A\ R_1B\ W_1B\ R_2A\ W_2A\ R_2B\ W_2B\ W_3B$

A: $T_3 \to T_1 \to T_2$ B: $T_1 \to T_2 \to T_3$



$T_1\ T_3\ T_2$

S_1 is serializable, S_2 is not.

To exclude not serializable schedules, a so called *transaction manager* enforces certain transaction behaviour.

2-Phase Locking (2PL)

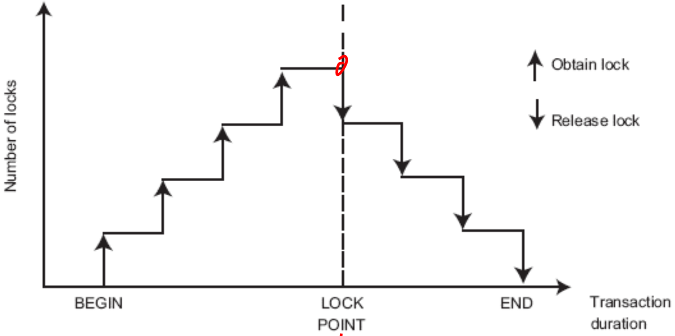
- Serializable schedules are guaranteed, if all transactions obey the 2PL-protocol:
  - For each transaction T, each RA and WA has to be surrounded by a lock/unlock pair LA, UA:

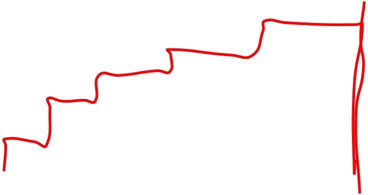
$$T = \dots R/WA \dots \Longrightarrow T = \dots LA \dots R/WA \dots UA \dots$$

  - For each A read or written in T there exists at most one pair LA and UA.
  - In any schedule S, the same object A cannot be locked at the same time by more than one transaction: (exclusive lock)

$$S = \dots L_i A \dots L_j A \dots \Longrightarrow S = \dots L_i A \dots U_i A \dots L_j A \dots$$

  - For each T and any LA_1, UA_2 there holds: $T = \dots LA_1 \dots UA_2 \dots$.
    $\Longrightarrow$ No more locking after the first unlock!
- Every schedule according to 2PL is serializable, however
  - Not every serializable schedule can be produced by 2PL.
  - Deadlocks may occur.


Number of locks
Obtain lock
Release lock
BEGIN
LOCK
POINT
END
Transaction
duration



strict 2PL

Example 1

$$T_1 = L_1A\ R_1A\ L_1B\ U_1A\ W_1B\ U_1B,$$
$$T_2 = L_2A\ R_2A\ W_2A\ U_2A,$$
$$T_3 = L_3C\ R_3C\ U_3C.$$

L_1B U_1B UA U_1B

$$S = L_1A\ R_1A\ L_1B\ U_1A\ L_2A\ R_2A\ L_3C\ R_3C\ U_3C\ W_1B\ U_1B\ W_2A\ U_2A$$

Example 2

$$T_1 = L_1A\ R_1A\ L_1B\ U_1A\ W_1B\ U_1B,$$
$$T_2 = L_2A\ R_2A\ W_2A\ U_2A,$$
$$T_3 = L_3C\ R_3C\ U_3C.$$

$$S = L_1A\ R_1A\ L_1B\ U_1A\ L_2A\ R_2A\ L_3C\ R_3C\ U_3C\ W_1B\ U_1B\ W_2A\ U_2A$$

The *lock point* of a transaction using 2PL is given by the first unlock of the transaction.

2PL guarantees serializability of schedules.

Let S be a schedule of a set of 2PL-transactions $\mathcal{T} = \{T_1, \ldots, T_n\}$.

Assume, S is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \rightarrow T_1$.

- Each edge $T \rightarrow T'$ implies T and T' having conflicting actions, where the action of T preceds the one of T'.
- Because of surrounding actions by lock/unlock and the 2PL-rule, T' can execute its action only after the lock-point of T. This implies the following structure of S, where $A_1, \ldots, A_k$ are data items:

$$
\begin{aligned}
S &= \ldots U_1 A_1 \ldots L_2 A_1 \ldots, \\
&\vdots \\
S &= \ldots U_{k-1} A_{k-1} \ldots L_k A_{k-1} \ldots, \\
S &= \ldots U_k A_k \ldots L_1 A_k \ldots.
\end{aligned}
$$

- Let $l_1, \ldots, l_k$ be the lock points of the involved transactions. Then we have l_1 before l_2, $\ldots$, l_{k-1} before l_k and l_k before l_1. However this is a contradiction to the structure of a schedule. Therefore S is serializable.

2PL guarantees serializability of schedules.

Let S be a schedule of a set of 2PL-transactions $\mathcal{T} = \{T_1, \ldots, T_n\}$.

Assume, S is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $T_1 \to T_2 \to \cdots \to T_k \to T_1$.

- Each edge $T \to T'$ implies T and T' having conflicting actions, where the action of T precedes the one of T'.
- Because of surrounding actions by lock/unlock and the 2PL-rule, T' can execute its action only after the lock-point of T. This implies the following structure of S, where $A_1, \ldots, A_k$ are data items:

$$
\begin{aligned}
S &= \ldots U_1 A_1 \ldots L_2 A_1 \ldots, \\
&\vdots \\
S &= \ldots U_{k-1} A_{k-1} \ldots L_k A_{k-1} \ldots, \\
S &= \ldots U_k A_k \ldots L_1 A_k \ldots.
\end{aligned}
$$

- Let $l_1, \ldots, l_k$ be the lock points of the involved transactions. Then we have l_1 before l_2, $\ldots$, l_{k-1} before l_k and l_k before l_1. However this is a contradiction to the structure of a schedule. Therefore S is serializable.

2PL guarantees serializability of schedules.

Let S be a schedule of a set of 2PL-transactions $\mathcal{T} = \{T_1, \ldots, T_n\}$.

Assume, S is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \rightarrow T_1$.

- Each edge $T \rightarrow T'$ implies T and T' having conflicting actions, where the action of T precedes the one of T'. conflict = guarded by locks
- Because of surrounding actions by lock/unlock and the 2PL-rule, T' can execute its action only after the lock-point of T. This implies the following structure of S, where $A_1, \ldots, A_k$ are data items:

$$
\begin{aligned}
&S = \ldots U_1 A_1 \ldots L_2 A_1 \ldots, \\
&\vdots \\
&S = \ldots U_{k-1} A_{k-1} \ldots L_k A_{k-1} \ldots, \\
&S = \ldots U_k A_k \ldots L_1 A_k \ldots.
\end{aligned}
$$

- Let $l_1, \ldots, l_k$ be the lock points of the involved transactions. Then we have l_1 before l_2, $\ldots$, l_{k-1} before l_k and l_k before l_1. However this is a contradiction to the structure of a schedule. Therefore S is serializable.

2PL guarantees serializability of schedules.

Let S be a schedule of a set of 2PL-transactions $\mathcal{T} = \{T_1, \ldots, T_n\}$.

Assume, S is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $T_1 \to T_2 \to \cdots \to T_k \to T_1$.

- Each edge $T \to T'$ implies T and T' having conflicting actions, where the action of T precedes the one of T'.
- Because of surrounding actions by lock/unlock and the 2PL-rule, T' can execute its action only after the lock-point of T. This implies the following structure of S, where $A_1, \ldots, A_k$ are data items:

$$\begin{aligned} S &= \ldots U_1 A_1 \ldots L_2 A_1 \ldots, \\ &\vdots \\ S &= \ldots U_{k-1} A_{k-1} \ldots L_k A_{k-1} \ldots, \\ S &= \ldots U_k A_k \ldots L_1 A_k \ldots. \end{aligned}$$

- Let $l_1, \ldots, l_k$ be the lock points of the involved transactions. Then we have l_1 before l_2, $\ldots$, l_{k-1} before l_k and l_k before l_1. However this is a contradiction to the structure of a schedule. Therefore S is serializable.

2PL guarantees serializability of schedules.

Let S be a schedule of a set of 2PL-transactions $\mathcal{T} = \{T_1, \dots, T_n\}$.

Assume, S is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $T_1 \to T_2 \to \cdots \to T_k \to T_1$.

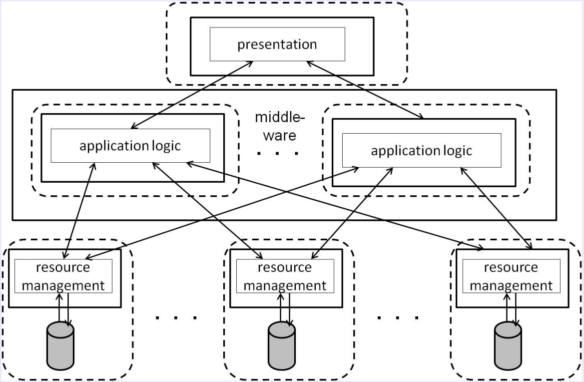
- Each edge $T \to T'$ implies T and T' having conflicting actions, where the action of T precedes the one of T'.
- Because of surrounding actions by lock/unlock and the 2PL-rule, T' can execute its action only after the lock-point of T. This implies the following structure of S, where $A_1, \dots, A_k$ are data items:

$$
\begin{aligned}
S &= \dots \overset{l_1}{U_1} A_1 \dots L_2 A_1 \dots, \\
&\vdots \\
S &= \dots \overset{l_{n-1}}{U_{k-1}} A_{k-1} \dots L_k A_{k-1} \dots, \\
S &= \dots \overset{l_n}{U_k} A_k \dots L_1 A_k \dots.
\end{aligned}
$$

- Let $l_1, \dots, l_k$ be the lock points of the involved transactions. Then we have l_1 before l_2, $\dots$, l_{k-1} before l_k and l_k before l_1. However this is a contradiction to the structure of a schedule. Therefore S is serializable.

10.2: Preliminaries of Distributed Concurrency Control

General reference architecture.



Federated system

Sites and subtransactions

- Let be given a fixed number of sites across which the data is distributed. The server at site i, $1 \leq i \leq n$ is responsible for a (finite) set D_i of data items. The corresponding global database is given as $D = \cup_{i=1}^{n} D_i$.
- Data items are not replicated; thus $D_i \cap D_j = \emptyset$, $i \neq j$.
- Let $\mathcal{T} = \{T_1, \ldots, T_m\}$ be a set of transactions, where $T_i = (OP_i, <_i)$, $1 \leq i \leq m$.
- Transaction T_i is called *global*, if its actions are running at more than one server; otherwise it is called *local*. (2PL + ARIES)
- The part of a transaction T_i being executed at a certain site j is called *subtransaction* and is denoted by T_{ij}. (? + 2PC)

Parallelism as prerequisite for distributed execution

- Basic definitions of transactions (and most visualizations) assume a total order.
- This is insufficient to express distributed execution of a transaction: Fine-grained coordination needed
- Relaxed model needed: partial ordering among operations

Formal definition:
A transaction T is defined as $(OP, <)$

OP is a finite set of T's actions RX and WX, where X is a data item.

$< \subseteq OP \times OP$ is a partial order on OP which fulfills the following properties:

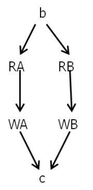
- Each data item is read and written by T at most once.
- If p is a read action and q is a write actions of T and both access the same data item, then $p < q$.

A parallel debit/credit transaction. *b*: BEGIN; *c*: COMMIT.

b - RA WA RB WA c

Source A   Dest B

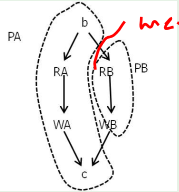
total



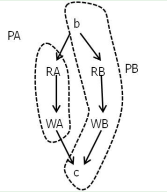
no order between RA and RB etc.

When transactions are depicted as directed graphs, we omit transitive edges.

Two parallel debit/credit transactions, each prepared for parallel execution.

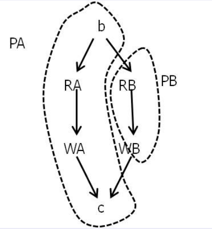


message delay

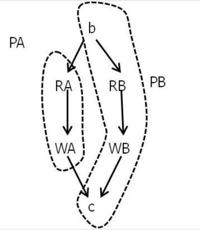


$\Longrightarrow$ Definition of a schedule? Definition of serializability?

Two parallel debit/credit transactions, each prepared for parallel execution.



Transaction T_1



Transaction T_2

Locally observable schedules of the two transactions when executed in parallel by CPU PA and CPU PB.

(i) $PA: \quad R_1A\ W_1A\ R_2A\ W_2A$
$PB: \quad R_1B\ W_1B\ R_2B\ W_2B$

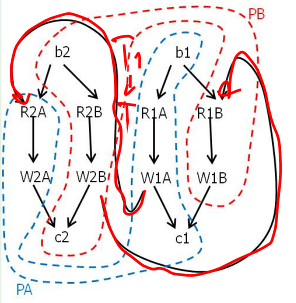
$T_1 \to T_2$
$T_1 \to T_2$ $\Rightarrow$ globally serializable

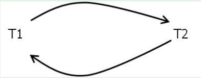
(ii) $PA: \quad R_1A\ W_1A\ R_2A\ W_2A$
$PB: \quad R_2B\ W_2B\ R_1B\ W_1B$

$T_1 \to T_2$
$T_2 \to T_1$ $\Rightarrow$ NOT

On each CPU in both cases the local schedules are serializable - however, globally, in the second case the transactions are not executed in a serializable manner!

A schedule/history of the two parallel debit/credit transactions.





The schedule is not serializable as its conflict graph is cyclic.

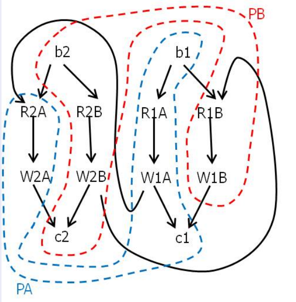
Local and global schedules

We are interested in deciding whether or not the execution of a set of transactions is serializable or not.

- At the local sites we can observe an evolving sequence of the respective transactions' actions.
- We would like to decide whether or not all these locally observable sequences imply a (globally) serializable schedule.
- However, on the global level we cannot observe an evolving sequence, as there does not exist a notion of global physical time.

Example

Schedule:



Observed local schedules:

$$\begin{array}{ll} \text{Site 1 (PA)}: & R_1A \ W_1A \ R_2A \ W_2A \\ \text{Site 2 (PB)}: & R_2B \ W_2B \ R_1B \ W_1B \end{array}$$

Can schedules be represented as action sequences, as well?

… yes, we call them *global schedules*.

From now on local and global schedules are sequences of actions!

Let $\mathcal{T} = \{T_1, \ldots, T_m\}$ be a set of transactions being executed at n sites. Let $S_1, \ldots, S_n$ be the corresponding local schedules.

A *global schedule* of $\mathcal{T}$ with respect to $S_1, \ldots, S_n$ is any sequence S of the actions of the transactions in $\mathcal{T}$, such that its projection onto the local sites equals the corresponding local schedules $S_1, \ldots, S_n$.

Example

Consider local schedules $S_1 = R_1A\ W_2A$ and $S_2 = W_1B\ R_2B$.

Global schedules:
$S: R_1A\ W_1B\ W_2A\ R_2B$
$S': R_1A\ W_1B\ R_2B\ W_2A$

Not a global schedule: $S'': R_1A\ R_2B\ W_1B\ W_2A$

preserve local ordering

Examples where there does not exist a serializable global schedule

- $T_1 = R_1A\ W_1B$, $T_2 = R_2C\ W_2A$ are global transactions and $T_3 = R_3B\ W_3C$ is a local transaction.

  $T_1 \rightarrow T_2$ $\quad S_1: A \quad S_2: B\ C$

  $S_1: \quad R_1A \quad W_2A$

  $S_2: \quad R_3B \quad W_1B \quad R_2C \quad W_3C$

  $T_2 \rightarrow T_3 \rightarrow T_1$

  Note, in S_2 subtransactions T_{12} and T_{22} have no confliciting actions!

- $T_1 = RA\ RD$ und $T_2 = RB\ RC$ are global transactions, while $T_3 = RA\ RB\ WA\ WB$ and $T_4 = RD\ WD\ RC\ WC$ are local transactions.

  $S_1: \quad R_1A \quad R_3A \quad R_3B \quad W_3A \quad W_3B \quad R_2B$

  $S_2: \quad R_4D \quad W_4D \quad R_1D \quad R_2C \quad R_4C \quad W_4C$

  Note, both global transactions are only reading and, in particular, disjoint data sets!

In both examples the local schedules are serializable, however no serializable global schedule exists.

Serializability of global schedules

- As we do not have replication of data items, whenever there is a conflict in a global schedule, the same conflict must be part of exactly one local schedule.
- Consequently, the conflict graph of a global schedule is given as the union of the conflict graphs of the respective local schedules.
- In particular, given a set of local schedules, either all or none corresponding global schedule is serializable.

Examples

$T_1 \to T_2$

- $$\begin{array}{llllll} S_1: & R_1A & W_1A & R_2A & W_2A \\ S_2: & R_2B & W_2B & R_1B & W_1B \end{array}$$

  NOT

  $T_2 \to T_1$

- $$\begin{array}{llllll} S_1: & R_1A & W_2A & & \\ S_2: & R_3B & W_1B & R_2C & W_3C \end{array}$$

- $$\begin{array}{lllllll} S_1: & R_1A & R_3A & R_3B & W_3A & W_3B & R_2B \\ S_2: & R_4D & W_4D & R_1D & R_2C & R_4C & W_4C \end{array}$$

Types of federation

- *homogeneous* federation:

  Same services and protocols at all servers. Characterized by *distribution transparency*: the federation is perceived by the outside world as if it were not distributed at all.

- *heterogenous* federation:

  Servers are autonomous and independent of each other; no uniformity of services and protocols across the federation.

Interface to recovery

Every global transactions runs the 2-phase-commit protocol. By that protocol the subtransactions of a global transaction synchronize such that either all subtransactions commit, or none of them, i.e. all abort.