Transactions: Definition, Properties, Classification

Concurrency, Conflicts, and Schedules

Locking Based Algorithms

Timestamp Ordering Algorithms

Transactions/1

- **Transaction**: A collection of actions that transforms the DB from one consistent state into another consistent state; during the execution the DB might be inconsistent.

\[
\begin{align*}
\text{Database in a consistent state} & \quad \text{Database may be temporarily in an inconsistent state during execution} & \quad \text{Database in a consistent state} \\
\text{Begin Transaction} & \quad \text{Execution of Transaction} & \quad \text{End Transaction}
\end{align*}
\]

Transactions/2

- **States** of a transaction
  - **Active**: Initial state and during the execution
  - **Partially committed**: After the final statement has been executed
  - **Committed**: After successful completion
  - **Failed**: After the discovery that normal execution can no longer proceed
  - **Aborted**: After the transaction has been rolled back and the DB restored to its state prior to the start of the transaction. Restart it again or kill it.

Transaction Example/1

- **Example**: Consider an airline DB with the following relations:
  
  \[
  \begin{align*}
  \text{FLIGHT}(\text{FNO}, \text{DATE}, \text{SRC}, \text{DEST}, \text{STSOLD}, \text{CAP}) \\
  \text{CUST}(\text{CNAME}, \text{ADDR}, \text{BAL}) \\
  \text{FC}(\text{FNO}, \text{DATE}, \text{CNAME}, \text{SPECIAL})
  \end{align*}
  \]

  Consider the reservation of a ticket, where a travel agent enters flight number, date and customer name, and then asks for a reservation.

  \[
  \begin{align*}
  \text{Begin\_transaction} \quad \text{Reservation} \\
  \text{begin} \\
  \text{input}(\text{flight\_no}, \text{date}, \text{customer\_name}); \\
  \text{EXEC SQL UPDATE} \quad \text{FLIGHT} \\
  \text{SET} \quad \text{STSOLD} = \text{STSOLD} + 1 \\
  \text{WHERE} \quad \text{FNO} = \text{flight\_no} \text{ AND DATE} = \text{date}; \\
  \text{...}
  \end{align*}
  \]
Transaction Example/2

▶ (cont’d)

EXEC SQL INSERT
    INTO FC(FNO, DATE, CNAME, SPECIAL);
VALUES (flight_no, date, customer_name, null);
    output ("reservation completed")
end.

▶ Example (contd.):
▶ A transaction always terminates: commit (successful completion) or abort (does not do the task).
▶ The above transaction assumes that there are always free seats.
▶ A better solution is to check the availability of free seats and terminate the transaction appropriately.

Transaction Example/3

Begin_transaction Reservation
begin
input (flight_no, date, customer_name);
EXEC SQL SELECT STSOLD, CAP
    INTO temp1, temp2 FROM FLIGHT
    WHERE FNO = flight_no AND DATE = date;
if temp1 = temp2 then
    output ("no free seats");
    Abort
else
    EXEC SQL UPDATE FLIGHT
        SET STSOLD = STSOLD + 1
        WHERE FNO = flight_no AND DATE = date;
    EXEC SQL INSERT
        INTO FC(FNO, DATE, CNAME, SPECIAL);
    VALUES (flight_no, date, customer_name, null);
    Commit
    output ("reservation completed")
endif
end.

Definition of a Transaction/1

▶ We use the following notation:
▶ TI is a transaction; x is a data item (attribute value, tuple, relation)
▶ Oij = operation O of transaction TI
▶ Oij ∈ {R(x), W(x)} be an atomic read/write operation of TI on data item x
▶ OSi = ∪j Oij be the set of all operations of TI
▶ Ni ∈ {A, C} be the termination operation, i.e., abort/commit
▶ Two operations Oij(x) and Oik(x) on the same data item are in conflict if at least one of them is a write operation
▶ A transaction TI is a partial order over its operations, i.e.,
    TI = {Σi, ≺i}, where
    1. Σi = OSi ∪ Ni
    2. For Oij = {R(x) ∨ W(x)} and Oik = W(x), either Oij ≺i Oik or Oik ≺i Oij
    3. ∀Oij ∈ OSi (Oi j ≺i Ni )
Definition of a Transaction/2

- Remarks
  - The partial order $\prec$ is given and is actually application dependent
  - It has to specify the execution order between the conflicting operations and between all operations and the termination operation

- Example: Consider the following transaction $T$
  
  Read(x)
  Read(y)
  $x \leftarrow x + y$
  Write(x)
  Commit

- The transaction is formally represented as

$$\Sigma = \{R(x), R(y), W(x), C\}$$

$$\prec = \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$$

Definition of a Transaction/3

Example (contd.): A transaction can also be specified/represented as a directed acyclic graph (DAG), where the vertices are the operations and the edges indicate the ordering.

- Assume

$$\prec = \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$$

- The DAG is

![DAG representation](image)

- Often a total order given by the relative ordering of operations is used to simplify notation, e.g., $T = \{R(x), R(y), W(x), C\}$

Definition of a Transaction/4

Example: The reservation transaction is more complex, as it has two possible termination conditions, but a transaction allows only one

- BUT, a transaction is the execution of a program which has obviously only one termination

- Thus, it can be represented as two transactions, one that aborts and one that commits

- Transaction $T_1$:

$$\Sigma = \{R(STSOLD), R(CAP), R(FNO), R(DATE), A\}$$

$$\prec = \{(R(STSOLD), A), (R(CAP), A), (R(FNO), A), (R(DATE), A)\}$$

- Transaction $T_2$:

$$\Sigma = \{R(STSOLD), R(CAP), R(FNO), R(DATE), W(STSOLD), W(FNO), W(DATE), W(CNAME), W(SPECIAL), C\}$$

$$\prec = \{(R(STSOLD), W(STSOLD)), \ldots\}$$

Transaction Processing Issues

- A transaction is a collection of actions that transforms the system from one consistent state into another consistent state.

- Transaction $T$ can be viewed as a partial order: $T = \{\Sigma, \prec\}$, where $\Sigma$ is the set of all operations, and $\prec$ denotes the order of operations. $T$ can also be represented as a directed acyclic graph (DAG).

- Transaction manager aims to achieve four properties of transactions: atomicity, consistency, isolation, and durability.

- Transactions can be classified according to (i) organization of reads and writes, (ii) time, and (iii) structure.

- Transaction processing involves concurrency, reliability, and replication protocols to ensure the ACID properties:
  - Atomicity: execute all operations or no operation
  - Consistency: transform consistent state into new consistent state
  - Isolation: behave as if transactions are executed one after the other
  - Durability: committed changes are permanent
Classification of Transactions

- **Classification** of transactions according to various criteria (cnt’d)
  - **Duration** of transaction
    - On-line (short-life)
    - Batch (long-life)
  - **Structure** of transaction
    - Flat transaction: Consists of a sequence of primitive operations between a begin and end marker
    - Nested transaction: The operations of a transaction may themselves be transactions.
    - **Workflows**: A collection of tasks organized to accomplish a given business process

Concurrency

- Concurrency deals with operations from multiple transactions.
- **Concurrency control** is the problem of synchronizing concurrent transactions (i.e., order the operations of concurrent transactions) such that the following two properties are achieved:
  - the consistency of the DB is maintained
  - the maximum degree of concurrency of operations is achieved
- Obviously, the serial execution of a set of transaction achieves consistency, if each single transaction is consistent

Conflicts/1

- **Conflicting operations**: Two operations $O_{ij}(x)$ and $O_{kl}(x)$ of transactions $T_i$ and $T_k$ are in conflict iff at least one of the operations is a write, i.e.,
  - $O_{ij} = \text{read}(x)$ and $O_{kl} = \text{write}(x)$
  - $O_{ij} = \text{write}(x)$ and $O_{kl} = \text{read}(x)$
  - $O_{ij} = \text{write}(x)$ and $O_{kl} = \text{write}(x)$
- Intuitively, a conflict between two operations indicates that their order of execution is important.
- Read operations do not conflict with each other, hence the ordering of read operations does not matter.
- The **conflict graph** contains a node for each transaction. It contains an edge from $T_i$ to $T_j$ if there is an operation $p$ in $T_i$ that conflicts with an operation $q$ in $T_j$ and $p \prec q$.

Conflicts/2

- **Example**: Consider the following two transactions
  $T_1$: Read($x$) $\quad x \leftarrow x + 1$ $\quad$ Write($x$) $\quad$ Commit
  $T_2$: Read($x$) $\quad x \leftarrow x + 1$ $\quad$ Write($x$) $\quad$ Commit
- To preserve DB consistency, it is important that the read($x$) of one transaction is not between read($x$) and write($x$) of the other transaction.
A schedule (history) specifies a possibly interleaved order of execution of all operations $O$ of a set of transactions $T = \{ T_1, T_2, \ldots, T_n \}$, where $T_i$ is specified by a partial order $(\Sigma_i, \prec_i)$.

A schedule can be specified as a partial order over $O$, where

1. $\Sigma_T = \bigcup_{i=1}^{n} \Sigma_i$
2. $\prec_T \supseteq \bigcup_{i=1}^{n} \prec_i$
3. For conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, either $O_{ij} \prec_T O_{kl}$ or $O_{kl} \prec_T O_{ij}$.

A concurrent schedule is said to be conflict equivalent if they have the same effect on the DB.

Conflict equivalence: Two schedules $S_1$ and $S_2$ defined over the same set of transactions $T = \{ T_1, T_2, \ldots, T_n \}$ are said to be conflict equivalent if for each pair of conflicting operations $O_{ij}$ and $O_{kl}$, whenever $O_{ij} \prec_T O_{kl}$ then $O_{ij} \prec_T O_{kl}$.

i.e., conflicting operations must be executed in the same order in both schedules.

A concurrent schedule is said to be (conflict-)serializable iff it is conflict equivalent to a serial schedule.
Serializability/2

- A conflict-serializable schedule can be transformed into a serial schedule by swapping non-conflicting operations
- **Example:** Consider the following two schedules

\[
T_1: \begin{align*}
&\text{Read}(x) \\
x &\leftarrow x + 1 \\
&\text{Write}(x) \\
&\text{Write}(z) \\
&\text{Commit}
\end{align*}
\]

\[
T_2: \begin{align*}
&\text{Read}(x) \\
x &\leftarrow x + 1 \\
&\text{Write}(x) \\
&\text{Commit}
\end{align*}
\]

- The schedule \(\{R_1(x), W_1(x), R_2(x), W_2(x), W_1(z), C_2, C_1\}\) is conflict-equivalent to \(\{T_1, T_2\}\) but not to \(\{T_2, T_1\}\)

Serializability/3

- **The primary function** of a concurrency controller is to generate a serializable schedule for the execution of pending transactions.
- In a DDBMS two schedules must be considered
  - Local schedule
  - Global schedule
- **Serializability** in DDBMS
  - The key concepts of transactions, histories, schedules, and conflicts remain valid.
  - Serializability extends to a DDBMS if local schedules are replaced by global schedules, etc.
  - Without global schedules local schedules can be modified (e.g., by introducing explicit conflicts) to get global guarantees.
  - Requires care if data is replicated and transactions are run on all sites with replicated data items:

Serializability/4

- **Example:** Consider two sites and a data item \(x\) which is replicated at both sites.

\[
T_1: \begin{align*}
\text{Read}(x) \\
x &\leftarrow x + 5 \\
\text{Write}(x) \\
\end{align*}
\]

\[
T_2: \begin{align*}
\text{Read}(x) \\
x &\leftarrow x + 10 \\
\text{Write}(x) \\
\end{align*}
\]

- Both transactions need to run on both sites
- The following two schedules might have been produced at both sites:
  - Site1: \(S_1 = \{R_1(x), W_1(x), R_2(x), W_2(x)\}\)
  - Site2: \(S_2 = \{R_1(x), W_2(x), R_2(x), W_1(x)\}\)
- Both schedules are (trivially) serializable, thus are correct in the local context
- But they produce different results for \(x\) and are therefore not correct.
- Solution: same serialization order of transactions on both sites

Concurrency Control Algorithms

- The most common concurrency control algorithms are
  - locking: ensure mutually exclusive access to data
  - ordering: order execution of transactions according to a protocol
- Two-Phase Locking (2PL)
  - Centralized (primary site) 2PL
  - Primary copy 2PL
  - Distributed 2PL
- Timestamp Ordering (TO)
  - Multiversion TO
Locking-Based Algorithms/1

- **Locking-based concurrency algorithms** ensure that data items shared by conflicting operations are accessed in a mutually exclusive way. This is accomplished by associating a “lock” with each such data item.
- Two types of **locks** (lock modes)
  - **read lock** (rl) – also called shared lock
  - **write lock** (wl) – also called exclusive lock
- **Compatibility matrix** of locks

\[
\begin{array}{c|cc}
  & rl_i(x) & wl_i(x) \\
\hline
rl_j(x) & \text{compatible} & \text{not compatible} \\
wl_j(x) & \text{not compatible} & \text{not compatible}
\end{array}
\]

- General locking algorithm
  1. Before using a data item \( x \), transaction requests lock for \( x \) from the lock manager
  2. If \( x \) is already locked and the existing lock is incompatible with the requested lock, the transaction is delayed
  3. Otherwise, the lock is granted

Two-Phase Locking (2PL)/1

- **Two-phase locking** protocol
  - Each transaction is executed in two phases
    - **Growing phase**: the transaction obtains locks
    - **Shrinking phase**: the transaction releases locks
  - The **lock point** is the moment when transitioning from the growing phase to the shrinking phase

Two-Phase Locking (2PL)/2

- **Properties** of the 2PL protocol
  - Generates **conflict-serializable** schedules
  - But schedules may cause **cascading aborts**
  - If a transaction aborts after it releases a lock, it may cause other transactions that have accessed the unlocked data item to abort as well
- **Strict 2PL locking** protocol
  - Holds the locks till the end of the transaction
  - Cascading aborts are avoided

Example: Consider the following two transactions

\[
\begin{align*}
T_1: & \quad \text{Read}(x) \\
     & \quad x \leftarrow x + 1 \\
     & \quad \text{Write}(x) \\
     & \quad \text{Read}(y) \\
     & \quad y \leftarrow y - 1 \\
     & \quad \text{Write}(y)
\end{align*}
\]

\[
\begin{align*}
T_2: & \quad \text{Read}(x) \\
     & \quad x \leftarrow x + 2 \\
     & \quad \text{Write}(x) \\
     & \quad \text{Read}(y) \\
     & \quad y \leftarrow y + 2 \\
     & \quad \text{Write}(y)
\end{align*}
\]

The following schedule is a valid locking-based schedule (\( lr_1(x) \) indicates the release of a lock on \( x \)):

\[
S = \{wl_1(x), R_1(x), W_1(x), lr_1(x), w_2(x), R_2(x), W_2(x), lr_2(x), w_1(y), R_1(y), W_1(y), lr_1(y)\}
\]

However, \( S \) is not serializable

- \( S \) cannot be transformed into a serial schedule by using only non-conflicting swaps
- The result is different from the result of any serial execution
Two-Phase Locking (2PL)/3

▶ Example: The schedule $S$ of the previous example is not valid in the 2PL protocol:

$S = \{wl_1(x), R_1(x), W_1(x), lr_1(x)
\}
wl_2(x), R_2(x), W_2(x), lr_2(x)
wl_2(y), R_2(y), W_2(y), lr_2(y)
wl_1(y), R_1(y), W_1(y), lr_1(y)\}$

▶ e.g., after $lr_1(x)$ (in line 1) transaction $T_1$ cannot request the lock $wl_1(y)$ (in line 4).

▶ Valid schedule in the 2PL protocol

$S = \{wl_1(x), R_1(x), W_1(x),\}
wl_1(y), R_1(y), W_1(y), lr_1(x), lr_1(y)
wl_2(x), R_2(x), W_2(x),\}
wl_2(y), R_2(y), W_2(y), lr_2(x), lr_2(y)\}$

2PL for DDBMS/1

▶ Various extensions of the 2PL to DDBMS

▶ Centralized 2PL

▶ A single site is responsible for the lock management, i.e., one lock manager for the whole DDBMS
▶ Lock requests are sent to the lock manager
▶ Coordinating transaction manager (TM at site where the transaction is initiated) makes lock requests on behalf of local transaction managers

▶ Advantage: Easy to implement
▶ Disadvantages: Bottlenecks and lower reliability
▶ Replica control protocol is additionally needed if data are replicated (see also primary copy 2PL)

2PL for DDBMS/2

▶ Primary copy 2PL

▶ Several lock managers are distributed to a number of sites
▶ Each lock manager is responsible for managing the locks for a set of data items
▶ For replicated data items, one copy is chosen as primary copy, others are slave copies
▶ Only the primary copy of a data item that is updated needs to be write-locked
▶ Once primary copy has been updated, the change is propagated to the slaves

▶ Advantages
▶ Lower communication costs and better performance than the centralized 2PL

▶ Disadvantages
▶ Deadlock handling is more complex

2PL for DDBMS/3

▶ Distributed 2PL

▶ Lock managers are distributed to all sites
▶ Each lock manager responsible for locks for data at that site
▶ If data is not replicated, it is equivalent to primary copy 2PL
▶ If data is replicated, the Read-One-Write-All (ROWA) replica control protocol is implemented

▶ Read($x$): Any copy of a replicated item $x$ can be read by obtaining a read lock on the copy
▶ Write($x$): All copies of $x$ must be write-locked before $x$ can be updated

▶ Disadvantages
▶ Deadlock handling more complex
▶ Communication costs higher than primary copy 2PL
### 2PL for DDBMS/4

- Communication structure of the distributed 2PL
  - The coordinating TM sends the lock request to the lock managers of all participating sites
  - The LMs pass the operations to the data processors
  - The end of the operation is signaled to the coordinating TM

![Communication Structure Diagram](image_url)

### Timestamp Ordering/1

- **Timestamp-ordering** based algorithms do not maintain serializability by mutual exclusion, but select (a priori) a serialization order and execute transactions accordingly.
  - Transaction $T_i$ is assigned a globally unique timestamp $ts(T_i)$
  - Conflicting operations $O_{ij}$ and $O_{kl}$ are resolved by timestamp order, i.e., $O_{ij}$ is executed before $O_{kl}$ if $ts(T_i) < ts(T_k)$.
  - To allow for the scheduler to check whether operations arrive in correct order, each data item is assigned a write timestamp ($wts$) and a read timestamp ($rts$):
    - $rts(x)$: largest timestamp of any transaction that read $x$
    - $wts(x)$: largest timestamp of any transaction that wrote $x$

### Timestamp Ordering/2

- Then the scheduler has to perform the following checks:
  - **Read operation**, $R_i(x)$:
    - If $ts(T_i) < wts(x)$: $T_i$ attempts to read overwritten data; abort $T_i$
    - If $ts(T_i) \geq wts(x)$: the operation is allowed and $rts(x)$ is updated
  - **Write operations**, $W_i(x)$:
    - If $ts(T_i) < rts(x)$: $x$ was needed before by other transaction; abort $T_i$
    - If $ts(T_i) < wts(x)$: $T_i$ writes an obsolete value; abort $T_i$
    - Otherwise, execute $W_i(x)$

### Timestamp Ordering/3

- **Generation of timestamps (TS)** in a distributed environment
  - TS needs to be locally and globally unique and monotonically increasing
  - System clock, incremental event counter at each site, or global counter are unsuitable (difficult to maintain)
  - Concatenate local timestamp/counter with a unique site identifier: `<local timestamp, site identifier>`
    - site identifier is in the least significant position in order to distinguish only if the local timestamps are identical
Schedules generated by the basic TO protocol have the following properties:
- Serializable
- Since transactions never wait (but are rejected), the schedules are deadlock-free
- The price to pay for deadlock-free schedules is the potential restart of a transaction several times
- Cascading rollbacks are possible
- Recoverability requires additional steps that ensure that commits happen in the correct order

Multiversion timestamp ordering
- Write operations do not modify the DB; instead, a new version of the data item is created: $x_1, x_2, \ldots, x_n$
- $R_i(x)$ is always successful and is performed on the appropriate version of $x$, i.e., the version of $x$ (say $x_v$) such that $wts(x_v)$ is the largest timestamp less than $ts(T_i)$
- $W_i(x)$ produces a new version $x_w$ with $ts(x_w) = ts(T_i)$ if the scheduler has not yet processed any $R_j(x_r)$ on a version $x_r$ such that $ts(T_i) < rts(x_r)$
  - i.e., the write is too late.
  - Otherwise, the write is rejected.

Conclusion
- A transaction is a set of operations with a partial order
- The transaction manager ensures atomicity, consistency, isolation, and durability
- A schedule is some order of the operations of the given transactions. If a set of transactions is executed one after the other, we have a serial schedule.
- There are two main groups of serializable concurrency control algorithms: lock based and timestamp based
- Local properties do not ensure global properties but can be generalized.

Course Exam
- Exam date: Monday 16.01.2017
- Exam time: 14:00 - 17:00
- Exam location: BIN 2.E.13
- Exam form and procedure
  - oral, 20 minutes
  - 10 minutes about project (demo, code, algorithm)
  - 10 about a topic of the course
    - Distributed Database Systems
    - Distributed Database Design
    - Distributed Query Processing
    - Distributed Query Optimization
    - Distributed Transactions and Concurrency Control
- During exam: present solutions on examples
- Prepare suitable examples beforehand