Chapter 6: Introduction

Distributed Applications - Motivation

Why do we want to make our applications distributed?

- Applications are inherently distributed.
- A distributed system is more reliable.
- A distributed system performs better.
- A distributed system scales better.

Only paradigm to support use cases like Google, Facebook and Amazon!

Data-Centric Distributed Applications

Union of two technologies:

- Database Systems + Distributed Systems
- Database systems provide
  - data independence (physical & logical)
  - centralized and controlled data access
  - integration
- Distributed System provide scaling

Chapter 6: Introduction

Distributed Applications - Motivation

Why do we want to make our applications distributed?

- Applications are inherently distributed.
- A distributed system is more reliable.
- A distributed system performs better.
- A distributed system scales better.

Only paradigm to support use cases like Google, Facebook and Amazon!

Data-Centric Distributed Applications

Union of two technologies:

- Database Systems + Distributed Systems
- Database systems provide
  - data independence (physical & logical)
  - centralized and controlled data access
  - integration
- Distributed System provide scaling

Goals of Data-Centric Distributed Applications

1. Transparent management of distributed and replicated data
2. Reliability/availability through distributed transactions
3. Improved performance
4. Easier and more economical system expansion

Focus of this course: Distributed transactions!
If you are interested in the other aspects, visit my course in the winter term.

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
- Fundamental conflicts between requirements (see CAP theorem)

Goals of Data-Centric Distributed Applications

1. Transparent management of distributed and replicated data
2. Reliability/availability through distributed transactions
3. Improved performance
4. Easier and more economical system expansion

Focus of this course: Distributed transactions!
If you are interested in the other aspects, visit my course in the winter term.

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
- Fundamental conflicts between requirements (see CAP theorem)

Why *transactions?*

Transactions form a reasonable *abstraction concept* for many classes of real-life data processing problems.

- Transactions cope in a very elegant way with the subtle and often difficult issues of keeping data consistent in the presence of highly concurrent data accesses and despite all sorts of failures.
- This is achieved in a generic way invisible to the application logic so that application developers are freed from the burden of dealing with such system issues.
  - The application program simply has to indicate the boundaries of a transaction by issuing `BEGIN TRANSACTION` and `COMMIT TRANSACTION` calls.
  - The execution environment considers all requests receiving from the application programs execution within this dynamic scope as belonging to the same transaction.
  - For the transaction's requests and effects on the underlying data certain properties are guaranteed: ACID properties.

Challenges inherent to the transaction concept demonstrated by some examples

(Expl.1a) Debit/credit

Consider a debit/credit-program of a bank which transfers a certain amount of money between two accounts. Executing the program will give us the following transaction T:

```
BEGIN
% Withdraw
READ current value VA of account A from disk into T's local main memory;
decrement VA by amount X;
WRITE new value VA' = VA - X of account A from T's local main memory onto disk;
% Deposit
READ current value VB of account B from disk into T's local main memory;
increment VB by amount X;
WRITE new value VB' = VB + X of account B from T's local main memory onto disk;
COMMIT;
```

- Assume when executing T the system runs into a failure, e.g. after writing A and before reading B. A customer of the bank has lost X money!
- Assume debit/credit-transaction T1 is running concurrently to a transaction T2, which computes the balance of the accounts A and B. Then the READ and WRITE accesses of both transactions may be interleaved. Assume that T2 is executed after T1 writing A and before T1 writing B, then the balance computed will be incorrect.

(Expl.1b) Distributed debit/credit

Assume that different branches of the bank are involved, where each branch maintains its own server. Assume further, at Branch1 a debit/credit-transaction is started and at Branch2 a balancing transaction, where both involve the same accounts. Transactions shall have access to accounts on remote server via remote procedure calls (RPC), a synchronous communication mechanism transparent to the programmer. We assume procedures *withdraw*(*account*, *amount*), *deposit*(*account*, *amount*) and *getBalance*(*account*).

A possible interleaving when both transactions are running in parallel.

*Branch*1(*accountA*)	*Branch*2(*accountB*)	
T_1 : withdraw(A,10)		
T_1 : call(deposit(B,10))		
	T_2 : getBalance(B)	↓ *time*
	T_2 : call(getBalance(A))	
	T_1 : deposit(B,10)	
T_2 : getBalance(A)		
	T_2 : display A+B	

An incorrect balance will be displayed!

(Expl.1c) Distributed debit/credit

Assume that different branches of the bank are involved, where each branch maintains it own servers. Assume further, at Branch1 a debit/credit-transaction is started and at Branch2 a balancing transaction is started, where both involve the same accounts. Finally assume, that each transaction implements exclusive access to both accounts during execution. Communication is explicitly implemented by exchanging messages between the involved servers.

A possible interleaving when both transactions are running in parallel.

*Branch*1	*Branch*2
T_1 : {{lock(A); withdraw(A,10)} || {send {lock(B); deposit(B,10)} to Branch2}}	
	T_2 : {{lock(B); getBalance(B)} || {send {lock(A); getBalance(A)} to Branch1}}
T_1 : {wait for ACK of deposit at Branch2}	
T_2 : {wait until lock(A) granted}	
	T_2 : {wait for balance of A}
	T_1 : {wait until lock(B) granted}

$\downarrow$ *time*

A deadlock has occured which is difficult to detect!

(Expl.2) Electronic commerce

Consider the following purchasing activity, which covers several different servers located at different sites:

- A client connects to a bookstore's server and starts browsing and querying the catalog.
- The client gradually fills a shopping card with items intended to purchase.
- When the client is about to check out she makes final decisions what to purchase.
- The client provides all necessary information for placing a legally binding order, e.g. shipping address and credit card.
- The merchants's server forwards the payment information to the customer's bank or to a clearinghouse. When the payment is accepted, the inventory is updated, shipping is initiated and the client is notified about successful completion of her order.

- The final step of the purchasing is the most critical one. Several servers maintained by different institutions are involved.
- Most importantly it has to be guaranteed, that either all the tasks of the final step are processed correctly, or the whole purchasing activity is undone.

(Expl.3) Mobile computing

Assume that the described purchasing activity is performed via a smartphone. Then the described picture is even more complicated.

- The smartphone might be temporarily disconnected from the mobile net. Thus it is not guaranteed, that the state of the catalog as seen by the client reflects the state of the catalog at the server.
- If the client enters a dead spot during processing of the final step of the purchasing activity, confusion may arise, e.g. the purchasing is started again resulting in double orders.

Transaction Concept

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.

$\Longrightarrow$ *Concurrency Control* (I) and *Recovery* (A, D) provide the mechanisms needed to cope with the problems demonstrated by Expl.1-3.

Concurrency Control Refresh

Basics

- 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.

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

- Let WX be the j-th action of transaction T and assume that $RA_1, \ldots, RA_n$ 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.
- A concurrent execution of a set of transactions is called schedule and is given as a - possibly interleaved - sequence of the respective actions, e.g.

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

- A schedule is called serial, if it is not interleaved.

Correctness

- A schedule is called (conflict-)serializable,[1] if there exists a (conflict-)equivalent serial schedule over the same set of transactions.
- For a given schedule S over a set of transactions, the conflict graph $G(S)$ is given as $G(S) = (V, E)$, where the node set V is the set of transactions in S and the set of edges E is given by so called conflicts as follows:
  - $S = \ldots W_i A \ldots R_j A \ldots \Rightarrow T_i \rightarrow T_j \in E$, if there is no other write-action to A between $W_i A$ und $R_j A$ in S.
  - $S = \ldots W_i A \ldots W_j A \ldots \Rightarrow T_i \rightarrow T_j \in E$, if there is no other write-action to A between $W_i A$ und $W_j A$ in S.
  - $\widehat{S} = \ldots R_i A \ldots W_j A \ldots \Rightarrow T_i \rightarrow T_j \in E$, if there is no other write-action to A between $R_i A$ und $W_j A$ in S.
- 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.

Example

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

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:

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

  - For each A read or written in T there exists at most one pair LA and UA.
  - For each T and any LA_1, UA_2 there holds: $T = \ldots LA_1 \ldots UA_2 \ldots$.
    $\Longrightarrow$ No more locking after the first unlock!
  - In any schedule S, the same object A cannot be locked at the same time by more than one transaction:

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

- Every schedule according to 2PL is serializable, however
  - Not every serializable schedule can be produced by 2PL.
  - Deadlocks may occur.

Example 1

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

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

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. $T_1 \to T_2 \to \cdots \to T_k \to T_1$.

- Each edge $T \to T'$ implies T and T' having conflicting actions, where the action of T 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. $T_1 \to T_2 \to \cdots \to T_k \to T_1$.

- 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. $T_1 \to T_2 \to \cdots \to T_k \to T_1$.

- Each edge $T \to T'$ implies T and T' having conflicting actions, where the action of T 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 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.

Recovery Refresh

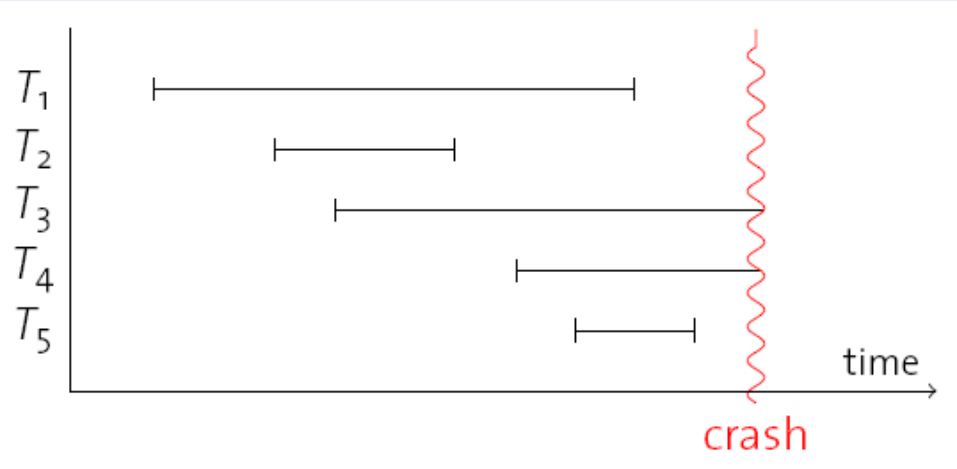
Failure Recovery

We want to deal with three types of failures:

- **transaction failure (also: process failure)**: A transaction voluntarily or involuntarily aborts. All of its updates need to be*undone*
- **system failure**: Database or operating system crash, power outage, etc. All information in main memory is lost. Must make sure that no committed transaction is lost (or *redo* their effects) and that all other transactions are *undone*.
- **media failure (also: device failure)**: Hard disk crash, catastrophic error (fire, water, ...). Must recover database from *stable storage*

In spite of all these failures, we we want to guarantee *atomicity* and *durability*.

Example: System Failure



- Transactions T_1, T_2, and T_5 were committed before the crash.
  - **Durability**: Ensure that updates are *preserved* (or *redone*).
- Transactions T3 and T4 were not (yet) committed.
  - **Atomicity**: All of their effects need to be *undone*.

Types of Storage

We assume three different types of storage:

- **volatile storage**: This is essentially the *buffer manager* in *main memory*. We are going to use volatile storage to cache the "*write-ahead log*" in a moment.
- **non-volatile storage**: Typical candidate is a hard disk or SSD
- **stable storage**: Non-volatile storage that survives all types of failures which is hard to achieve in practice. Stability can be improved using, e.g., (network) *replication* of disk data. Backup tapes are another example.

Observe how these storage types correspond to the three types of failures.

Interaction between volatile and non-volatile storage

Coordination policies between transactions and storage on non-volatile memory

- Can modified pages written to disk even if there is no commit (**Steal**)?
- Can we delay writing modified pages after commit (**No-Force**)?

Steal+No-Force

- improve throughput and latency,
- but make recovery more complicated

Types of Storage

We assume three different types of storage:

- **volatile storage**: This is essentially the *buffer manager* in *main memory*. We are going to use volatile storage to cache the "*write-ahead log*" in a moment.
- **non-volatile storage**: Typical candidate is a hard disk or SSD
- **stable storage**: Non-volatile storage that survives all types of failures which is hard to achieve in practice. Stability can be improved using, e.g., (network) *replication* of disk data. Backup tapes are another example.

Observe how these storage types correspond to the three types of failures.

Interaction between volatile and non-volatile storage

Coordination policies between transactions and storage on non-volatile memory

- Can modified pages written to disk even if there is no commit (**Steal**)?
- Can we delay writing modified pages after commit (**No-Force**)?

Steal+No-Force

- improve throughput and latency,
- but make recovery more complicated

Effects of TA/storage coordination on recovery

The decisions force/no force and steal/no steal have implications on what we have to do during recovery:

	force	no force
no steal	no redo no undo	must redo no undo
steal	no redo must undo	must redo must undo

If we want to use steal and no force (to increase concurrency and performance), we have to implement redo and undo routines.

ARIES Algorithm

- **A**lgorithm for **R**ecovery and **I**solation **E**xploiting **S**emantics
- A better alternative to shadow paging which switches between active/committed page
- Works with steal and no-force
- Data pages are updated in place
- Uses "logging"
  - Log: An ordered list of REDO/UNDO actions.
  - Record REDO and UNDO information for every update.
  - Sequential writes to log (usually kept on separate disk(s)).
  - Minimal info written to log $\rightsquigarrow$ multiple updates fit in a single log page.

Three main principles of ARIES

1. Write-Ahead Logging
   - Record database changes in the log at stable storage before the actual change.
2. Repeating History During Redo
   - After a crash, bring the system back to the exact state at crash time; undo the transactions that were still active at crash time.
3. Logging Changes During Undo
   - Log the database changes during a transaction undo so that they are not repeated in case of repeated failures and restarts (i.e., never undo an undo action).

Three main principles of ARIES

1. Write-Ahead Logging
   - Record database changes in the log at stable storage before the actual change.
2. Repeating History During Redo
   - After a crash, bring the system back to the exact state at crash time; undo the transactions that were still active at crash time.
3. Logging Changes During Undo
   - Log the database changes during a transaction undo so that they are not repeated in case of repeated failures and restarts (i.e., never undo an undo action).

Three main principles of ARIES

1. Write-Ahead Logging
   - Record database changes in the log at stable storage before the actual change.
2. Repeating History During Redo
   - After a crash, bring the system back to the exact state at crash time; undo the transactions that were still active at crash time.
3. Logging Changes During Undo
   - Log the database changes during a transaction undo so that they are not repeated in case of repeated failures and restarts (i.e., never undo an undo action).

Three main principles of ARIES

1. Write-Ahead Logging
   - Record database changes in the log at stable storage before the actual change.
2. Repeating History During Redo
   - After a crash, bring the system back to the exact state at crash time; undo the transactions that were still active at crash time.
3. Logging Changes During Undo
   - Log the database changes during a transaction undo so that they are not repeated in case of repeated failures and restarts (i.e., never undo an undo action).

Write-Ahead Log (WAL)

- The ARIES recovery method uses a "write-ahead log" to implement the necessary redundancy.
  - Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead LoggingW, ACM TODS, 17(1), 1992.
- WAL: Any change to a database object is **first** recorded in the log, which must be written to stable storage **before** the change itself is written to disk.
  - To ensure atomicity and prepare for undo, undo information must be written to stable storage before a page update is written back to disk.
  - To ensure durability, redo information must be written to stable storage at commit time (no-force policy: the on-disk data page may still contain old information).

Write-Ahead Log (WAL)

- The ARIES recovery method uses a "write-ahead log" to implement the necessary redundancy.
  - Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead LoggingW, ACM TODS, 17(1), 1992.
- WAL: Any change to a database object is **first** recorded in the log, which must be written to stable storage **before** the change itself is written to disk.
  - To ensure atomicity and prepare for undo, undo information must be written to stable storage before a page update is written back to disk.
  - To ensure durability, redo information must be written to stable storage at commit time (no-force policy: the on-disk data page may still contain old information).

Write-Ahead Log (WAL)

- The ARIES recovery method uses a "write-ahead log" to implement the necessary redundancy.
  - Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead LoggingW, ACM TODS, 17(1), 1992.
- WAL: Any change to a database object is **first** recorded in the log, which must be written to stable storage **before** the change itself is written to disk.
  - To ensure atomicity and prepare for undo, undo information must be written to stable storage before a page update is written back to disk.
  - To ensure durability, redo information must be written to stable storage at commit time (no-force policy: the on-disk data page may still contain old information).

Write-Ahead Log (WAL)

- The ARIES recovery method uses a "write-ahead log" to implement the necessary redundancy.
  - Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead LoggingW, ACM TODS, 17(1), 1992.
- WAL: Any change to a database object is **first** recorded in the log, which must be written to stable storage **before** the change itself is written to disk.
  - To ensure atomicity and prepare for undo, undo information must be written to stable storage before a page update is written back to disk.
  - To ensure durability, redo information must be written to stable storage at commit time (no-force policy: the on-disk data page may still contain old information).

Log Information

The log consists of entries in the followowing forrm:

$$< LSN, Type, TOD, PrevLSN, PageID, NextLSN, Redo, Undo >$$

- LSN: Log Sequence Number: Monotonically increasing number to identify each log record.
- Type (Record Type): Begin, Commit, Abort, Update, Compensation
- TID: Transaction Identifier
- PrevLSN: Previous LSN of the same transaction
- PageID: Page which was modified
- NextLSN: Next LSN of the same transaction
- Redo Information described by this log entry
- Undo Information described by this log entry

Example of transactions and logs

Transaction 1	Transaction 2
$a \leftarrow \texttt{read}(A)$;	
	$c \leftarrow \texttt{read}(C)$;
$a \leftarrow a - 50$;	
	$c \leftarrow c + 10$;
$\texttt{write}(a,A)$;	
	$\texttt{write}(c,C)$;
$b \leftarrow \texttt{read}(B)$;	
$b \leftarrow b + 50$;	
$\texttt{write}(b,B)$;	
commit;	
	$a \leftarrow \texttt{read}(A)$;
	$a \leftarrow a - 10$;
	$\texttt{write}(a,A)$;
	commit;

LSN	Type	TX	Prev	Page	UNxt	Redo	Undo
1	UPD	T_1	–	···		$A := A - 50$	$A := A + 50$
2	UPD	T_2	–	···		$C := C + 10$	$C := C - 10$
3	UPD	T_1	1	···		$B := B + 50$	$B := B - 50$
4	EOT	T_1	3	···			
5	UPD	T_2	2	···		$A := A - 10$	$A := A + 10$
6	EOT	T_2	5	···			

Redo Information

- ARIES assumes **page-oriented** redo
- stores byte images of the pages
- *before* and *after* the modification
- Restore exact same pages as execution without failures

Undo Information

- ARIES assumes **logical undo**
- Record the actual tuple changes, e.g. account A increased by 50
- Faster undo

Redo Information

- ARIES assumes **page-oriented** redo
- stores byte images of the pages
- *before* and *after* the modification
- Restore exact same pages as execution without failures

Undo Information

- ARIES assumes **logical undo**
- Record the actual tuple changes, e.g. account A increased by 50
- Faster undo

Writing Log Records

- For performance reasons, all log records are first written to volatile storage.
- At certain times, the log is forced to stable storage up to a certain LSN:
  - Commit of a transaction for Redo
  - Page writing of uncommitted for Undo
- Committed transaction = all log records (including commit) are on stable storage

Normal Processing

- During normal transaction processing, keep two pieces of information in each transaction control block:
  - LastLSN: LSN of the last log record written for this transaction.
  - NextLSN: LSN of the next log record to be processed during rollback.
- Whenever an update to a page p is performed
  - a log record r is written to the WAL, and
  - the LSN of r is recorded in the page header of p.

Writing Log Records

- For performance reasons, all log records are first written to volatile storage.
- At certain times, the log is forced to stable storage up to a certain LSN:
  - Commit of a transaction for Redo
  - Page writing of uncommitted for Undo
- Committed transaction = all log records (including commit) are on stable storage

Normal Processing

- During normal transaction processing, keep two pieces of information in each transaction control block:
  - LastLSN: LSN of the last log record written for this transaction.
  - NextLSN: LSN of the next log record to be processed during rollback.
- Whenever an update to a page p is performed
  - a log record r is written to the WAL, and
  - the LSN of r is recorded in the page header of p.

ARIES Transaction Rollback

- To roll back a transaction T after a **transaction failure** (e.g. ABORT):
  - Process the log in a backward fashion.
  - Start the undo operation at the log entry pointed to by the UNxt field in the transaction control block of T.
  - Find the remaining log entries for T by following the Prev and UNxt fields in the log.
  - Perform the changes in the Undo part of the log entry
- Undo operations modify pages, too!
  - Log all undo operations to the WAL.
  - Use compensation log records (CLRs) for this purpose.
  - Note: We never undo an undo action, but we might need to redo an undo action.

ARIES Crash Recovery

Restart after a system failure is performed in three phases

1. Analysis Phase:
   - Read log in forward direction.
   - Determine all transactions that were active when the failure happened. Such transactions are called "losers".
2. Redo Phase:
   - Replay the log (in forward direction) to bring the system into the state as of the time of system failure.
   - Put "after images" in place of before images
   - Also restores the losers
3. Undo Phase
   - Roll back all loser transactions, reading the log in a backward fashion (similar to "normal" rollback).

Media Recovery

- To allow for recovery from non-volatile media failure, periodically back up data to stable storage.
- Can be done during normal processing, if WAL is archived, too.
- Other approach: Use log to mirror database on a remote host (send log to network and to stable storage).

Checkpointing

- WAL file keeps growing unbounded
- For recovery, we need to visit entire WAL file
- Generate checkpoints with current transaction state
  - Recovery only from checkpoint
  - Bound WAL file and allow truncation

Media Recovery

- To allow for recovery from non-volatile media failure, periodically back up data to stable storage.
- Can be done during normal processing, if WAL is archived, too.
- Other approach: Use log to mirror database on a remote host (send log to network and to stable storage).

Checkpointing

- WAL file keeps growing unbounded
- For recovery, we need to visit entire WAL file
- Generate checkpoints with current transaction state
  - Recovery only from checkpoint
  - Bound WAL file and allow truncation