

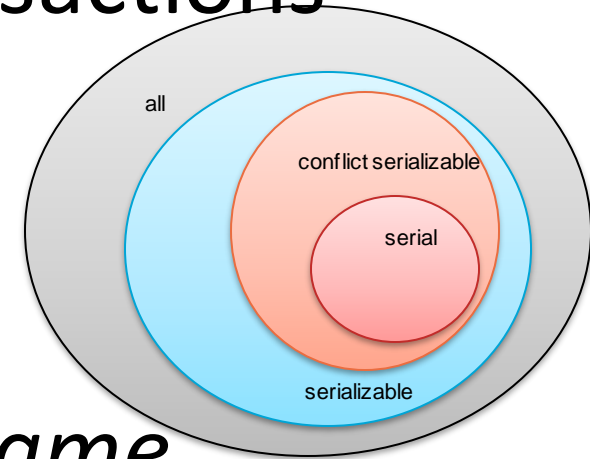
# Information & Data Management

## Recovery & Distributed Transactions

**Christoph Lofi**  
**Asterios Katsifodimos**  
**Rihan Hai**

# Last Lecture

- **Serial** schedules: the holy grail of transactions
  - They ensure correct execution
  - Avoid concurrency issues or anomalies
  - But they are slow
- **Serializable**: a schedule that has *the same effects* on a database as a *serial* schedule.
- **Conflict-serializable**: a schedule that has exactly the same conflicting operations (in the same order) as a serial schedule.



# Next

- Part 1
  - How do we recover from failures?
    - Logging
    - UNDO/REDO logging
    - Recovery from logs
- Part 2
  - How do we perform distributed transactions?
    - Two-Phase Commit
    - CAP Theorem

# Recovery Using Logging

# Introduction to Recovery

- **Concurrency control** is mostly concerned with **isolation** and **consistency** issues
  - Transactions should never adversely influence each other
  - The database should stay in a consistent state
- **But what happens if a transaction does not complete?**
  - Either the transaction is aborted by the system/user
  - Or the operating system crashes, a harddrive fails, etc.

# Introduction to Recovery

- **Recovery** deals with the problems of **atomicity** and **durability**
  - Each transaction has to be **completed entirely** or must not be performed at all (rollback)
  - Results of **committed transactions** need to be **persistent** in the database
  - Database systems have to be protected against **transaction failures** and **hardware crashes**



# Today: Recovery from System Failures

- **Failure Modes**
- Undo Logging
- Redo Logging
- Undo/Redo Logging
- Archiving

# Repetition: Error Classification

1. Transaction failure

2. System failure

3. Media failure



# Repetition: Error Classification

## 1. Transaction failure

- Leads to abort of a transaction
- Error in an application (e.g., division by zero)
- **abort** command
- Abort by DBMS (e.g., deadlock)

## 2. System failure

- Crash in DBMS, OS, Hardware
- Data in main memory is lost

## 3. Media failure

- Head crash, Controller Failure, Catastrophy
- Data on disk is destroyed
- Now more ...

# System Failure

- Transactions
  - Are stateful during execution
    - Values of variables
    - Actual piece of code
  - State maintained in main memory
  - State will be lost upon system failure
- Power outage
  - State destroyed
- Software failure
  - State overwritten
- Solution
  - Logging – this lecture!

# Repetition: Transactions

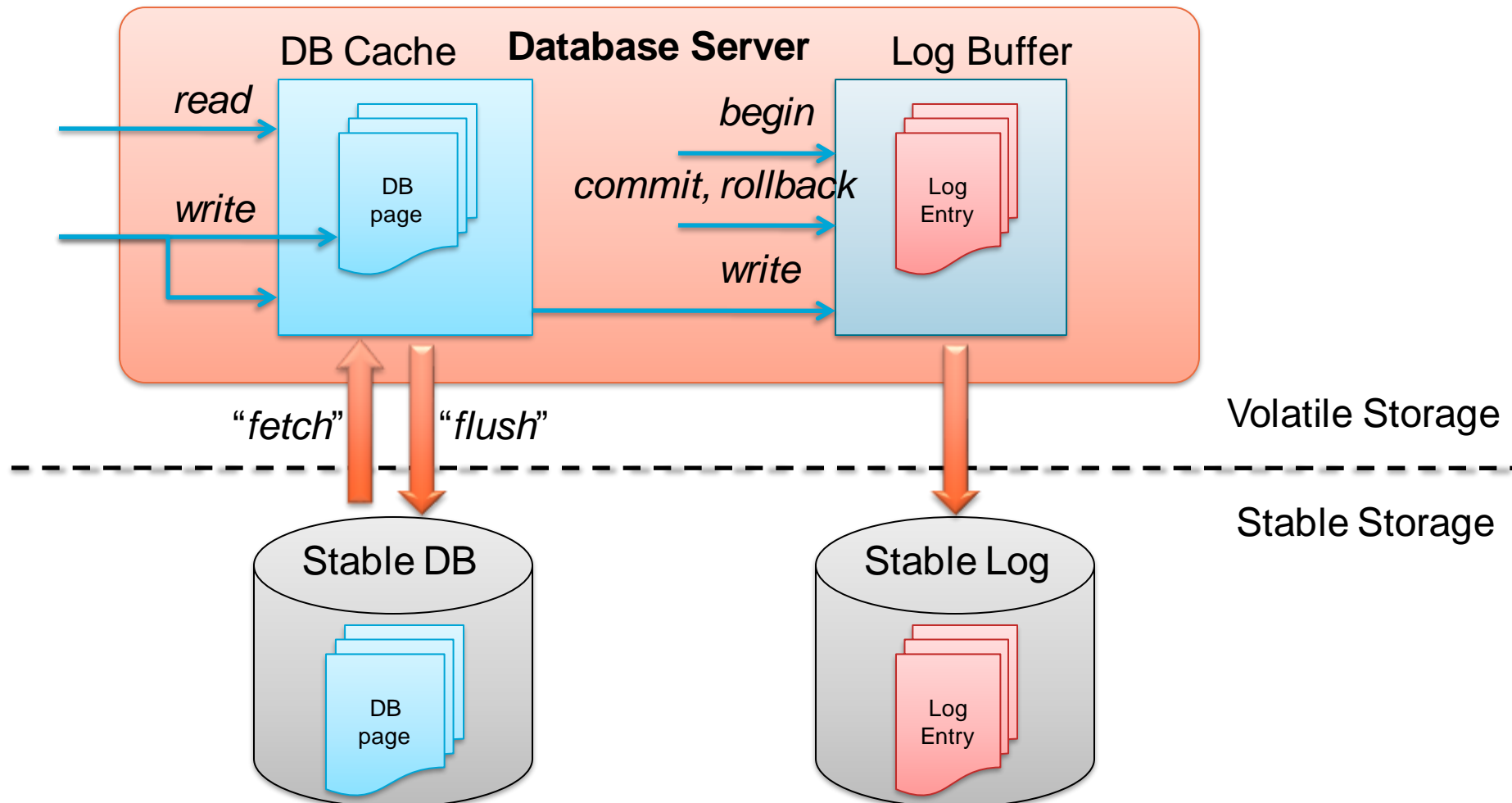
- A transaction is a sequence of operations (actions) which transform a database from a consistent state into a consistent state (perhaps a different one) following the ACID principle.
- ACID:
  - Atomicity and Durability solved by logging
  - Consistency solved by integrity constraints that may abort transactions.
  - Isolation solved by typically by locking / time-stamping.

# Transactions

- So far, developer point of view
  - Single SQL commands are transactions
  - Embedded SQL: Sequence of SQL commands
    - Transactions finished with COMMIT or ROLLBACK
- Atomicity is necessary
- Now, from the standpoint of the transaction manager
  - Responsible for correct processing of transactions
    - Sends log messages to the log manager
    - Schedules the execution order of operations
    - Must guarantee atomicity

# System Concepts

- **Components** involved in recovery



# System Concepts

- Generally there are two ways to write a page to disc
  - In-place updating overwrites the original copy on disk (before image, BFIM) with a new version (after image, AFIM), thus only a single copy of the data exists
    - Depending on the recovery protocol, we need UNDO and/or REDO logs to recover
    - In order to have a reliable log, log entries need to be written before the AFIM is applied
      - Called Write-Ahead-Logging – really write to disc, not only to buffer
        - » ...it is beneficial if logs have their own disc because of that
  - Shadow paging writes a new item at a different disk location, thus both the AFIM and BFIM exist on disc
    - Classic shadow-paging is not used that often anymore, but record-based (not page-based) multi-version protocols are popular now
      - This can work without logs to a certain extend

# System Concepts

- When can pages be written back?
- **Force:** Always directly write / flush changes to disc after a page in the buffer was modified
  - Might be slow due to frequent writes
  - Writes partial transaction results to disc which might need to be undone
    - e.g., in case of system failure or transaction rollback
- **No-Force:** Writes can be deferred to a later time
  - We might lose pages in the (volatile) buffer

# System Concepts

- When can pages be written back?  
**Steal-No Steal:** Can other transactions evict blocks when the buffer is full?
  - **No-Steal:** A page modified by a transaction cannot be written back to disc as part of buffer management as long as the transaction is not committed
    - Use pin attribute for this
    - We might lose the buffer content
  - **Steal:** A page modified by a transaction can be flushed / written to disc before it is committed
    - i.e. another transaction can steal its space in the buffer
    - Disc can have changes of partial transactions, which potentially need to be undone



# Executing Transactions

- Database consists of „Elements“
  - Definition of „Element“ depends on DBMS
    - Complete table
    - Blocks
    - Row (record)
- Database has a state: Value for every element
  - Either consistent or inconsistent
  - Consistent = all explicit and implicit integrity constraints are satisfied
    - Implicit: Trigger, Transactions, etc.
- Assumption on correctness: If a transaction shall be executed on a consistent database in an isolated way with no failures, the database will be consistent afterwards
  - The „C“ in ACID

# Processing Transactions

- Processing within a transaction
  - Transaction requests a database element from buffer manager
  - Buffer manager retrieves the element from disk if needed
  - Element will be fetched into local address space of transaction
  - New/changed element will be created inside address space if necessary
  - Transaction returns new element to buffer manager
  - Eventually: Buffer manager writes element back to disk
- Failure tolerance requires frequent writing on disk
- Efficiency demands little writing on disk

# Primitive Operations

- Move elements between address spaces
- INPUT(X)
  - Copy X from disk into buffer
  - Executed by buffer manager
- READ(X, t)
  - Copy X from buffer into transaction address space for transaction t
  - Executed by transaction
- WRITE(X, t)
  - Copy value produced by t to element X in buffer
  - Executed by transaction
- OUTPUT(X)
  - Copy containing X from buffer to disk
  - Executed by buffer manager
- Assumption: Elements always fit into a single block
  - If not: Partition elements into single blocks
  - Be accurate/careful: Always write ALL blocks

# Operations – Example

- Two database elements: A and B
  - Integrity constraint:  $A = B$
  - More realistic constraints:
    - Sold flights  $< 110\%$  of seats
    - Sum of account balances = total balance
- Transaction T
  - $A := A \cdot 2$
  - $B := B \cdot 2$
- T is consistent

# Operations – Example

Action	t	Mem A	Mem B	Disc A	Disc B
READ(A,t)	8	8		8	8
$t := t \cdot 2$	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
$t := t \cdot 2$	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

- Where may a system failure occur?
- Where must a system failure not occur?

# Recovery from System Failures

- Failure Modes
- **Undo Logging**
- Redo Logging
- Undo/Redo Logging

# Logging

- Objective: Atomicity of transaction
- Log = Order of „log records“
- Activities of several transactions may overlap
  - Hence: Do not wait for end of transaction to log
  - System fault: Use Log for reconstructing a consistent state
- In general
  - Some transactions must be repeated (redo).
  - Some transactions must be rolled back (undo).
- Now: Undo-logging
  - If it is not clear, that the result of a transaction was completely written on disc: Undo!
  - Only undo, no redo

# Log Records

- Log file: Append-only
- Log-Manager saves each important event
- Log file also organized into blocks
  - First in main memory
  - Then written to disc
  - This makes logging more complex!



# Log Records

- Several log record/event types:
  - $\langle \text{START } T \rangle$  - Transaction T has begun
  - $\langle \text{COMMIT } T \rangle$  - Transaction T has completed and does not change anymore
    - General: Changes are not yet on disc necessarily (depends on buffer management strategy)
    - But undo logging requires this
  - $\langle \text{ABORT } T \rangle$  - Transaction T has aborted
    - Transaction manager must guarantee that T has no effect on disc
  - $\langle T, X, v \rangle$  - Update: Transaction T has changed element X  
v is the old (and overwritten) value of X
    - Especially for undo logging: New value is not necessary
    - Will be written after WRITE
    - Must be on disk before the OUTPUT
      - Why?

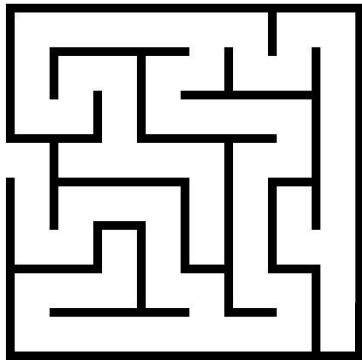
# Rules of Undo Logging

- U1: If transaction  $T$  changes element  $X$ ,  $\langle T, X, v \rangle$  must have been written in the log on disk BEFORE writing new value of  $X$  on disc
- U2: If transaction  $T$  commits,  $\langle \text{COMMIT } T \rangle$  can only be written into log after all changed elements have been written to disk.
- Writing to disk is carried out in the following order:
  1. Write log record for changed elements
  2. Write elements to disk
  3. Write COMMIT log record
  - 1. and 2. separately for each element!

# Undo Logging - Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8>
READ(B,t)	8	16	8	8	8	
$t := t \cdot 2$	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8>
<b>FLUSH LOG</b>						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
<b>FLUSH LOG</b>						

# Undo Logging – Theseus & Ariadne's thread!



“Ariadne gave Theseus a ball of red thread, and Theseus unrolled it as he penetrated the labyrinth, which allowed him to find his way back out. He found the minotaur deep in the recesses of the labyrinth, killed it with his sword, and followed the thread back to the entrance.”

# Log Manager

- Log manager: FLUSH LOG command instructs buffer manager to write all log blocks to disc
- Transaction manager: OUTPUT command instructs buffer manager to write element on disc
  - That implies: Every data block in the buffer manager has a position of the log (log entry number) associated with it.
  - Log must be written to that point before the block is written on disk.

# Undo Logging - Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8>
READ(B,t)	8	16	8	8	8	
$t := t \cdot 2$	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Necessary?

Transaction can be reported as committed, only after flushing log. Required for correct recovery.

# Recovery via Undo Logging

- Problem: System fault in the middle of transactions
  - Atomicity not fulfilled
  - Database inconsistent
  - → **Recovery manager to the rescue**
- Naive approach - Examine complete log
- Group all transactions
  - „Committed“ if <COMMIT T> exists
  - „Uncommitted“ otherwise (<START T> without <COMMIT T>)
    - Undo necessary!
    - Apply update records in log for undo
- Write old value *v* for element *X* with respect to the update records
  - Regardless of actual value that is currently stored

# Recovery via Undo Logging

- Problem
  - *Uncommitted* transactions made changes to database
- Solution: Undo recovery
  - Process complete log file backwards from end to start
    - „Chronologically backwards“
  - Most recent values first
- When walking back
  - Memorize all transactions with COMMIT or ABORT
  - When update record  $\langle T, X, v \rangle$ 
    - If a COMMIT or ABORT exists for T: Do nothing
    - Otherwise: Write v on X
- At the end
  - Write  $\langle \text{ABORT } X \rangle$  for all uncommitted transactions into log
  - FLUSH LOG



# Recovery via Undo Logging - Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8>
READ(B,t)	8	16		8	8	
$t := t \cdot 2$	16	16		8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

At Recovery:

- T will be recognized as committed
- No need to restore

Crash

# Recovery via Undo Logging - Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(A,t)						<T, A, 8>
READ(B,t)						
t := t · 2						
WRITE(B,t)						<T, B, 8>
FLUSH LOG						
OUTPUT(A)						
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

At Recovery:

- Perhaps <COMMIT T> record is already written to disc.
- Will be recognized as committed
- Do nothing
- Else: T will be recognized as uncommitted
- Value 8 will be written to B
- Value 8 will be written to A
- <ABORT T> will be written to log
- Log will be flushed

Crash

# Recovery via Undo Logging - Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8>
READ(B,t)			8	8	8	
$t := t \cdot 2$				8	8	
WRITE(B,t)				8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)				16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

At Recovery:

- T (definitely) will be recognized as uncommitted
  - Value 8 will be written to B
  - Value 8 will be written to A
  - <ABORT T> will be written to log
  - Log will be flushed

Crash

# Recovery via Undo Logging - Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2				8	8	
WRITE(A,t)				8	8	<T, A, 8>
READ(B,t)					8	
t := t · 2					8	
WRITE(B,t)					8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

At Recovery:

- T will be recognized as uncommitted
- Value 8 will be written to B (even when change is not on disc)
- Value 8 will be written to A (even when change is not on disc)
- <ABORT T> will be written to log
- Log will be flushed

Crash

# Recovery via Undo Logging - Example

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)				8	8	
t := t · 2				8	8	
WRITE(A,t)		16		8	8	<T, A, 8>
READ(B,t)				8	8	
t := t · 2				8	8	
WRITE(B,t)		16		8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)		16		16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

At Recovery:

- T will be recognized as uncommitted
- Problem: Log records already on disc?
- Never mind, because no data change could have happened before log write
- Value 8 will be written to B (even if change is not on disc)
- Value 8 will be written to A (even if change is not on disc)
- <ABORT T> will be written to log
- Log will be flushed

Crash


# System Fault during Recovery

- What to do?
- At system fault: Simply rerun log from the beginning
- Recovery steps are idempotent.
  - Repeated execution is equivalent to single execution

# Checkpointing

- Problem: Recovery forces to read complete log
  - Idea 1: Abort as soon as reading a COMMIT
    - But: However some other concurrent transactions could be still uncommitted
    - Does not work!
  - Idea 2: Set checkpoint periodically
    - Reject new transactions temporarily
    - Wait until all running transactions are committed or aborted and corresponding log records are written
    - Flush log
    - Write log record <CKPT>
    - Flush log
    - Accept new transactions
  - Changes of all previous transactions are written on disc
  - Recovery has to be executed till reading <CKPT> log record

# Checkpointing - Example

1. <START T1>
  2. <T1, A, 5>
  3. <START T2>
  4. <T2, B, 10>
  5. <T2, C, 15>
  6. <T1, D, 20>
  7. <COMMIT T1>
  8. <COMMIT T2>
  9. <CKPT>
  10. <START T3>
  11. <T3, E, 25>
  12. <T3, F, 30>
- Decision to  
Checkpoint
- 



# Non-Blocking Checkpointing

- Problem: Database is blocked during checkpointing
  - No new transactions will be accepted
  - Already running transactions may consume much time
- Idea: Checkpoint only for certain transactions
  - Write log record <START CKPT (T1, ..., Tk)>
    - All active transactions
  - Flush log
  - Wait till T1, ..., Tk are committed or aborted
    - But allow new transactions!
  - Write log record <END CKPT>
  - Flush log

# Non-Blocking Checkpointing - Recovery

- Reading from back first <END CKPT> or first <START CKPT ...>?
  - First <END CKPT>
    - Recovery only till next <START CKPT>
  - First <START CKPT T1,..., Tk>
    - Means that we had a system fault during checkpointing
    - T1, ..., Tk are the single active transaction at this time
    - Recovery more backwards, but only till starting of the earliest *uncommitted* transactions of T1, ..., Tk
      - Fast through appropriate pointer structure inside log file

# Non-Blocking Checkpointing - Example

1. <START T1>

2. <T1, A, 5>

3. <START T2>

4. <T2, B, 10>

5. ~~<START CKPT (T1, T2)>~~

Decision to  
Checkpoint  
(non-blocking)

6. <T2, C, 15>

7. <START T3>

8. <T1, D, 20>

9. <COMMIT T1>

10. <T3, E, 25>

11. <COMMIT T2>

12. <END CKPT>

13. <T3, F, 30>

**CRASH**

Recovery  
until row 5

1. <START T1>

2. <T1, A, 5>

3. <START T2>

Decision to  
Checkpoint  
(non-blocking)

4. ~~<T2, B, 10>~~

5. <START CKPT (T1, T2)>

6. <T2, C, 15>

7. <START T3>

8. <T1, D, 20>

9. <COMMIT T1>

10. <T3, E, 25>

**CRASH**

Recovery  
until row 3

# This Lecture: Recovery from System Failures

- Failure Modes
- Undo Logging
- **Redo Logging**
- Undo/Redo Logging

# Undo vs. Redo Logging

- Problem with Undo-Logging: „Commit“ not until having written to disc
  - Potentially high I/O costs
- Undo log: allowed to “commit” when all changes are *on disk*
  - COMMIT is written to log when all values are on disk
  - Log holds old values
  - Incomplete transactions are rolled back
  - Complete transactions are ignored
- Redo log: allowed to commit when changes are *on log*
  - COMMIT is written to log before any value is written to disk
  - Log holds new values
  - Incomplete transactions are ignored
  - Complete transactions are repeated

# Redo Logging Rules

- Log record  $\langle T, X, v \rangle$ 
  - For database element  $X$ , transaction  $T$  has written new value  $v$
- Redo Rule („write-ahead logging“ rule)
  - R1: Before any db element  $X$  changed by  $T$  is written to disc, all log records of  $T$  and COMMIT  $T$  must be written to log
- Writing to disk is carried out in the following order:
  - Write update log records on disc
  - Write COMMIT log record on disc
  - Write changed database elements on disc

# Redo Logging - Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 16>
READ(B,t)	8	16	8	8	8	
$t := t \cdot 2$	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

# Redo Logging - Recovery

- Observation
  - If no COMMIT in log, elements on disc are untouched
    - There is no need to recover them
  - => Incomplete transactions can be ignored
- Committed transactions are a problem
  - It is not clear, which changes are on disk right now
  - But: Log records hold all information about that
- Process
  - Identify committed transactions
  - Read log data from the beginning to the end (chronological)
    - For each update record  $\langle T, X, v \rangle$
    - If T is not committed: Ignore
    - If T is committed: Write v as element X
  - For each uncommitted transaction write  $\langle \text{ABORT } T \rangle$  into log
  - Flush log



# Redo Logging – Recovery Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 16>
READ(B,t)	8	16	8	8	8	
$t := t \cdot 2$	16	16	16	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

At Recovery:

- T will be recognized as committed
- Value 16 will be written for A (possibly redundant)
- Value 16 will be written for B (possibly redundant)

Crash

# Redo-Logging – Recovery Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 16>
READ(B,t)	8	16	8	8	8	
$t := t \cdot 2$	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

At Recovery:

- If <COMMIT T> (randomly) written to disc
- As before
- If not
  - As on next slide

Crash

# Redo Logging – Recovery Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 16>
READ(B,t)	8	16	8	8	8	
$t := t \cdot 2$	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

At Recovery:

- <COMMIT T> can not have been written to disc
- Hence, T is incomplete (uncommitted)
  - Do nothing at first
- Write <ABORT T> in log on disc

Crash

# Redo Logging - Checkpointing

- New problem in contrast to undo logging:
  - Limitation on active transactions will not suffice:
    - COMMIT in log while result has not written on disc yet
- Idea: Write all database elements on disc during checkpointing, that have been changed by committed but incomplete transactions
  - Dirty buffer must be known by the buffer manager
  - Log manager must know, which transactions have changed which buffer
- Advantage: There is no need to wait on completion of active (uncommitted) transactions
  - Uncommitted transactions are not written to disc

# Redo Logging - Checkpointing

- Procedure for non-blocking checkpointing
  - Write `<START CKPT (T1, ..., Tk)>` in log
    - $T_1, \dots, T_k$  are all active transactions
  - Flush log
  - Write all elements on disc
    - which are changed in buffer by committed transactions, but not written on disc yet.
  - Write `<END CKPT>` in log
  - Flush log

# Redo-Logging – Checkpointing

## Beispiel

1. <START T1>
2. <T1, A, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 10>
6. <START CKPT (T2)>
7. <T2, C, 15>
8. <START T3>
9. <T3, D, 20>
10. <END CKPT>
11. <COMMIT T2>
12. <COMMIT T3>

Decision  
For Non-Blocking  
Checkpoint

Before end of checkpointing,  
value A must be written on  
disc.

# Redo Logging - Checkpointing

## Recovery

- As for undo logging, see whether the last checkpoint record in log is a START or an END:
- **<END CKPT>**
  - All transactions that are committed before **<START CKPT (T1, ..., Tk)>** are on disc
  - T1,..., Tk and all transactions that have been started after START are unsafe
    - Even when COMMIT
  - It is sufficient to consider the earliest **<START Ti>**
    - Backward-linked log is helpful
- **<START CKPT (T1, ..., Ti)>**
  - System fault occurred during checkpointing
  - Even committed transactions before this point are unsafe
  - Backward oriented search to next **<END CKPT>** and then continue backwards to the appropriate **<START CKPT (S1, ..., Sj)>**
  - Then redo of all transactions, which have been committed after the START and redo of Si

# Redo Logging - Checkpointing Recovery Example

1. <START T1>
2. <T1, A, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 10>
6. <START CKPT (T2)>
7. <T2, C, 15>
8. <START T3>
9. <T3, D, 20>
10. <END CKPT>
11. <COMMIT T2>
12. <COMMIT T3>

**CRASH**

- Reverse search finds <END CKPT> at 10.
- Redo of all TAs, which started after the appropriate <START CKPT (T2)> (Row 6.) (so T3)
- Redo all TAs of list in 6. (so T2)
- T2 and T3 have been committed
  - So redo for both
- Search backwards till <START T2> (row 3)
  - Redo the three update entries in rows 5, 7, and 9.



# Redo Logging - Checkpointing Recovery Example

1. <START T1>
2. <T1, A, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 10>
6. <START CKPT (T2)>
7. <T2, C, 15>
8. <START T3>
9. <T3, D, 20>
10. <END CKPT>
11. <COMMIT T2>

## CRASH

12. <COMMIT T3>

- As before
- But T3 is not committed
  - So, no redo for T3
  - Add **<ABORT T3>** in log

# Redo Logging - Checkpointing Recovery Example

1. <START T1>
2. <T1, A, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 10>
6. <START CKPT (T2)>
7. <T2, C, 15>
8. <START T3>
9. <T3, D, 20>

## CRASH

10. <END CKPT>
11. <COMMIT T2>
12. <COMMIT T3>

- Search for the last  
    <START CKPT (T1, ..., Tk)>
  - Not here
  - So, start at the beginning of the log
- T1 is the only committed transaction
- Redo for T1
- <ABORT T2> in log
- <ABORT T3> in log

# This Lecture: Recovery from System Failures

- Failure Modes
- Undo Logging
- Redo Logging
- **Undo/Redo Logging**

# Best of both Worlds

- Disadvantage Undo: Data must be written immediately after end of transaction
  - => Too many I/Os
- Disadvantage Redo: All changed blocks must be retained in buffer till COMMIT and log records are on disc
  - => High memory requirement
- Undo/Redo logging is more flexible
  - But more information in log records are needed

# Undo/Redo Rules

- Log record:  $\langle T, X, v, w \rangle$ 
  - Old value ( $v$ ) **and** new value ( $w$ )
- Rule UR1:
  - Update record  $\langle T, X, v, w \rangle$  must have been written on disc BEFORE  $X$  changed by  $T$  has been written on disc
- This rule is also required by undo and redo as mentioned before
  - Reminder U1: If transaction  $T$  changes element  $X$ ,  $\langle T, X, v \rangle$  must be written on disc BEFORE the new value of  $X$  will be written on disc
  - Reminder R1: Before a database element  $X$  can be changed on disc, all appropriate log records must be written on disc
- What is missing in contrast to undo or redo?
  - $\langle \text{COMMIT } T \rangle$  can be written on disc BEFORE and AFTER OUTPUT of elements to disk

# Undo/Redo Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)	8	8		8	8	
$t := t \cdot 2$	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8, 16>
READ(B,t)	8	16	8	8	8	
$t := t \cdot 2$	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

Right  
the mic

It could have be  
occured before  
after.

# Undo/Redo Recovery

- Log allows redo and undo through storage of new and old value
  - Redo of all committed transactions in a chronological order
  - Undo of all uncommitted transactions in a reverse chronological order
- Both is important: Extreme cases
  - Committed transaction with no change on disc
  - Uncommitted transaction with all changes on disc

# Undo/Redo Recovery - Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)				8	8	
t := t · 2				8	8	
WRITE(A,t)				8	8	<T, A, 8, 16>
READ(B,t)				8	8	
t := t · 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

- <COMMIT T> possibly already on disc
- It seems to be that T is committed
- But B is possibly not yet on disc
- At recovery, both values „16“ will be written

Crash



# Undo/Redo Recovery - Example

Action	t	Mem A	Mem B	Disc A	Disc B	Log
						<START T>
READ(A,t)						
t := t · 2						
WRITE(A,t)						<T, A, 8, 16>
READ(B,t)	8	16	8	8	8	
t := t · 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

- <COMMIT T> cannot have reached disc
- Hence, T is uncommitted
- Whether it is necessary or not, recovery restores both value „8“
- Write <ABORT T> on disc

Crash

# Undo/Redo Recovery - Checkpointing

- Simpler approach
  - Write  $\langle \text{START CKPT } (T_1, \dots, T_k) \rangle$  to log
    - $T_i$  are all active transactions
  - Flush log
  - Write all dirty buffers on disc
    - The ones that contain at least one changed element
  - Write  $\langle \text{END CKPT} \rangle$  in log
  - Flush log

# Undo/Redo Recovery - Checkpointing

## Example

1. <START T1>
2. <T1, A, 4, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 9, 10>
6. <START CKPT (T2)>
7. <T2, C, 14, 15>
8. <START T3>
9. <T3, D, 19, 20>
10. <END CKPT>
11. <COMMIT T2>
12. <COMMIT T3>

- Only T2 is active at CKPT
- <T2, B, 9, 10> possibly on disc right now
  - Cp: Not possible with redo logging
- But it will be flushed on disc anyway
  - Like all dirty blocks
  - E.g. like <T1, A, 4, 5>, if not done yet

# Undo/Redo Recovery - Checkpointing

## Example

1. <START T1>
2. <T1, A, 4, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 9, 10>
6. <START CKPT (T2)>
7. <T2, C, 14, 15>
8. <START T3>
9. <T3, D, 19, 20>
10. <END CKPT>
11. <COMMIT T2>
12. <COMMIT T3>

**CRASH**

- Find <END CKPT> and appropriate <START CKPT...>
- T1 irrelevant – is committed before <START CKPT...>
- Redo of all committed TAs of the list and all TAs started after <START CKPT...>
  - Here: T2 and T3

# Undo/Redo Recovery - Checkpointing

## Example

1. <START T1>
  2. <T1, A, 4, 5>
  3. <START T2>
  4. <COMMIT T1>
  5. <T2, B, 9, 10>
  6. <START CKPT (T2)>
  7. <T2, C, 14, 15>
  8. <START T3>
  9. <T3, D, 19, 20>
  10. <END CKPT>
  11. <COMMIT T2>
- CRASH**T3>

- T2 is committed, T3 is active
- Redo for T2
  - Only till <START CKPT...>
- Undo for T3
  - In contrast to redo logging: Redo is not necessary there
  - Here the new value can already be on disc

## Undo/Redo Recovery - Checkpointing Example

1. <START T1>
2. <T1, A, 4, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 9, 10>
6. <START CKPT (T2)>
7. <T2, C, 14, 15>
8. <START T3>
9. <T3, D, 19, 20>
10. <END CKPT>

**CRASH**

- Undo for T2 and T3
- Since T2 is active at start of checkpoint, actions before the checkpoint must also be restored
  - <T2, B, 9, 10> must be undone

# Summary - Recovery

- Transaction Management
    - Atomicity through logging
    - Isolation through scheduling/locking
  - Database elements
    - Unit for Logging, Scheduling and Locking
    - Mostly disc blocks
    - But also relations or tuples
  - Logging
    - Data record for each important action
    - Log must be on disc when logged values come to disc
      - Before (undo) resp. after (redo) commit record
  - Recovery
    - Using log to restore a consistent state after system fault
- Undo Logging
    - Stores old values
      1. Log record on disc
      2. New value on disc
      3. Commit log on disc
    - Recovery through restoring old values for uncommitted TAs
  - Redo Logging
    - Stores new values
      1. Log record on disc
      2. Commit log on disc
      3. New value on disc
    - Recovery through writing new values of all committed TAs
  - Undo/Redo Logging
    - Stores old and new values
      1. Log record on disc
      2. New value on disc
    - Commit log whenever

# Distributed Transactions



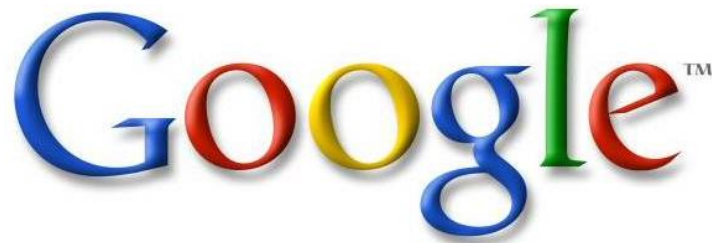
# Some motivating examples ...



- Hundreds of millions of products.
- ~ 300 sales per second.



- ~ 1 billion million active users.
- ~ 200k events per second.



- > 1 trillion ( $10^{12}$ ) pages indexed.
- ~ 40k queries per second.
- Search must complete in ~200ms.

# Challenges

- Scale out to 1000+ of machines
  - Data, query load (or both) are too large for a single machine.
  - Many services are deployed in multiple locations.
- Highly available even on unreliable (commodity) hardware:
  - Typical first year for a new cluster (J. Dean, LADIS 2009):
    - ~ 0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
    - ~ 1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours)
    - ~ 1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hrs)
    - ~ 1 network rewiring (rolling ~5% of machines down over 2-day span)
    - ~ 5 racks go wonky (40-80 machines see 50% packetloss)
    - ~ 1000 individual machine failures
    - Thousands of hard drive failures slow disks, bad memory, misconfigured machines
    - ...
  - ...

# In Summary

- Two “definitions” of distributed systems:
  - **Optimist:** A distributed system is a collection of independent computers that appears to its users as a single coherent system.
  - **Pessimist:** “You know you have one (a distributed system) when the crash of a computer you’ve never heard of stops you from getting any work done.”  
(Leslie Lamport)

# Transactions Spanning Multiple Machines/Data Centers - how?

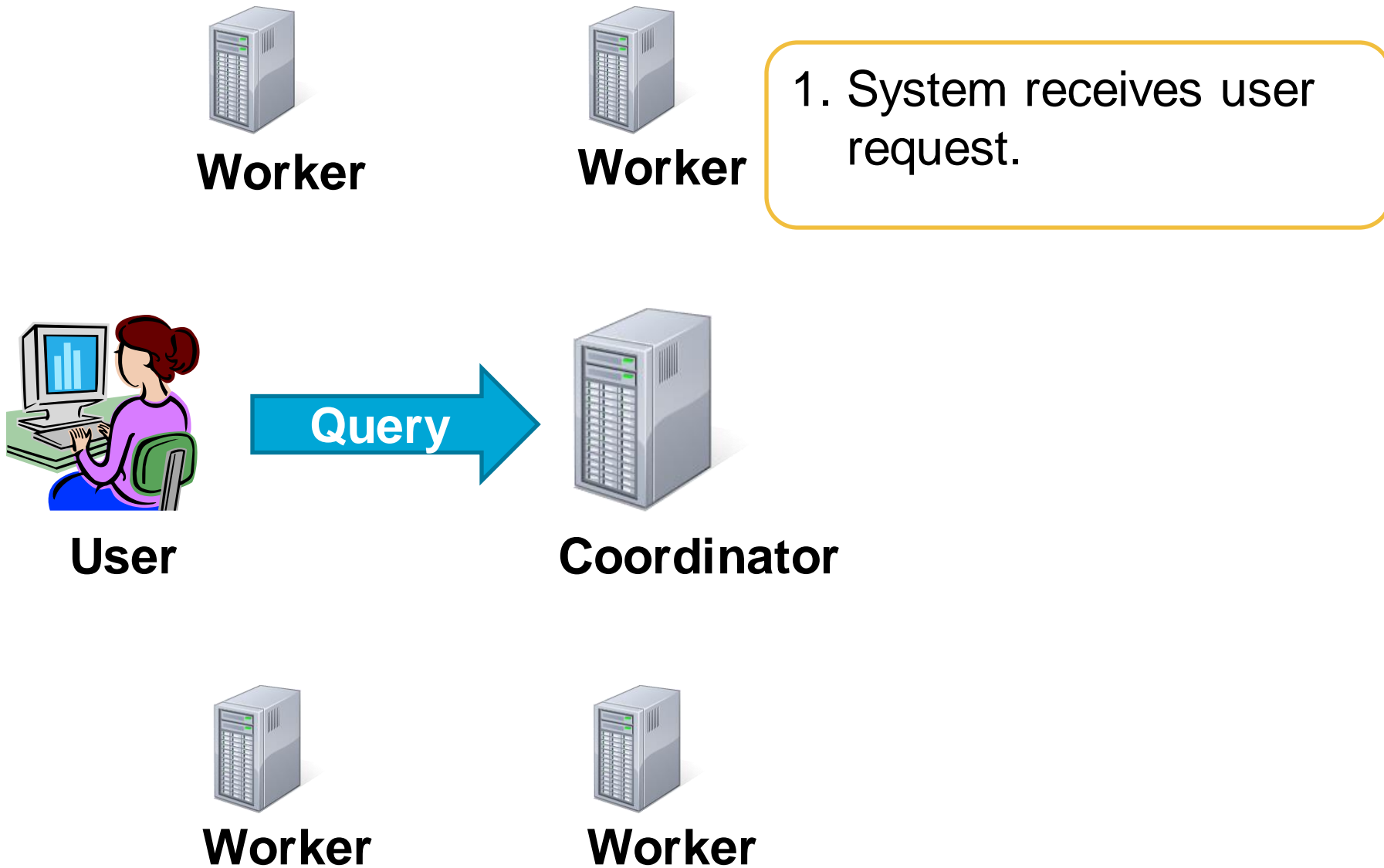
- ACID semantics for distributed transactions?
  - Distributed Commit Protocols
  - Distributed Lock Protocols

# Two Phase Commit (2PC) - Overview

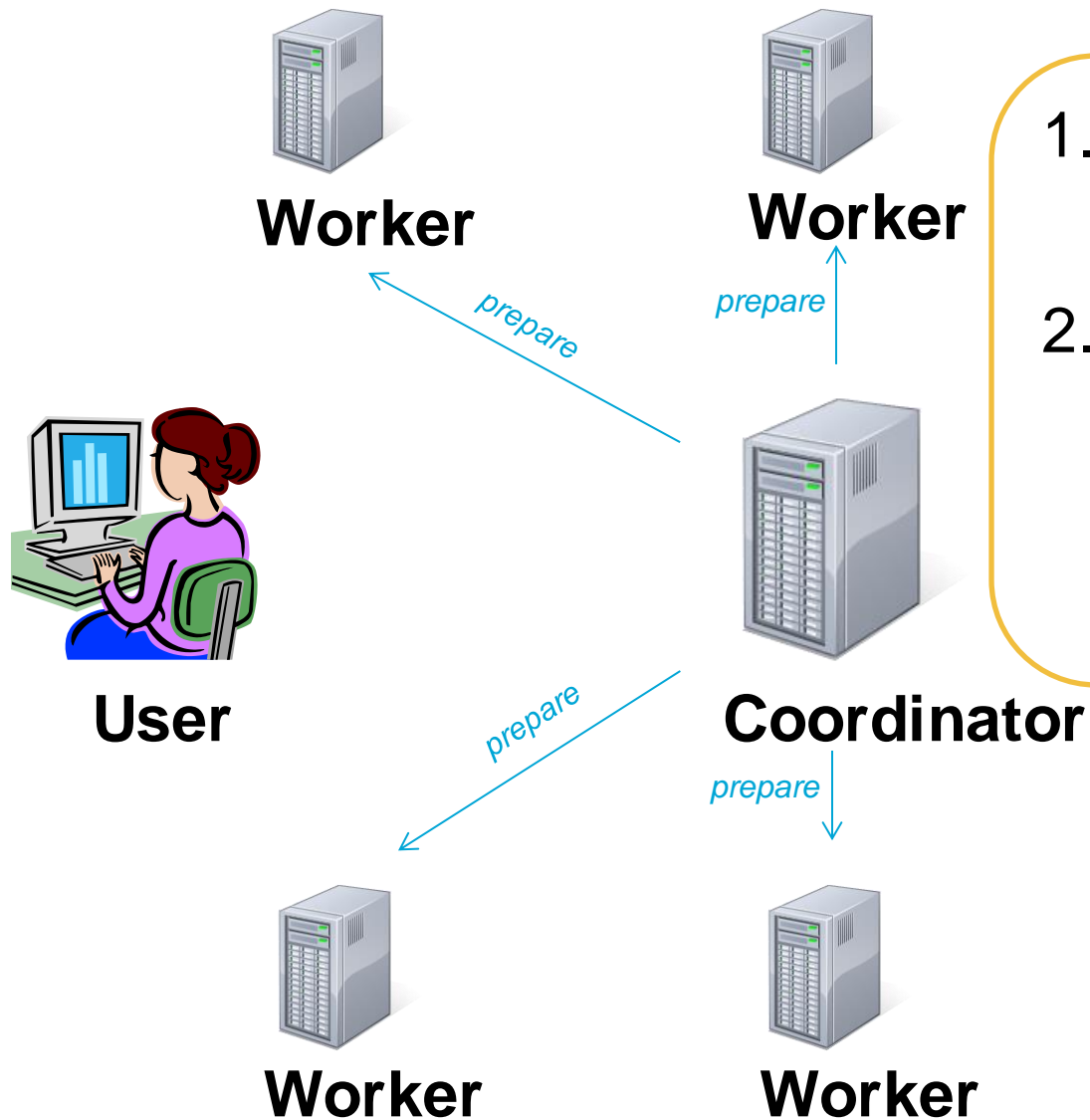
- Widely used family of protocols in distributed databases.
- Two Roles:
  - One coordinator.
  - $n$  workers (Cohorts).
- Assumptions:
  - Write-ahead log at each node (Thursday!).
  - No node crashes forever.
  - Any two nodes can communicate.
- Transactions are processed in two phases:
  - All nodes try to apply the transaction (writing it to the log).
  - The transaction is only committed if all nodes succeeded.
  - ➔ All or nothing: Either all nodes commit, or none does.



# Two Phase Commit (2PC) – Commit Request Phase

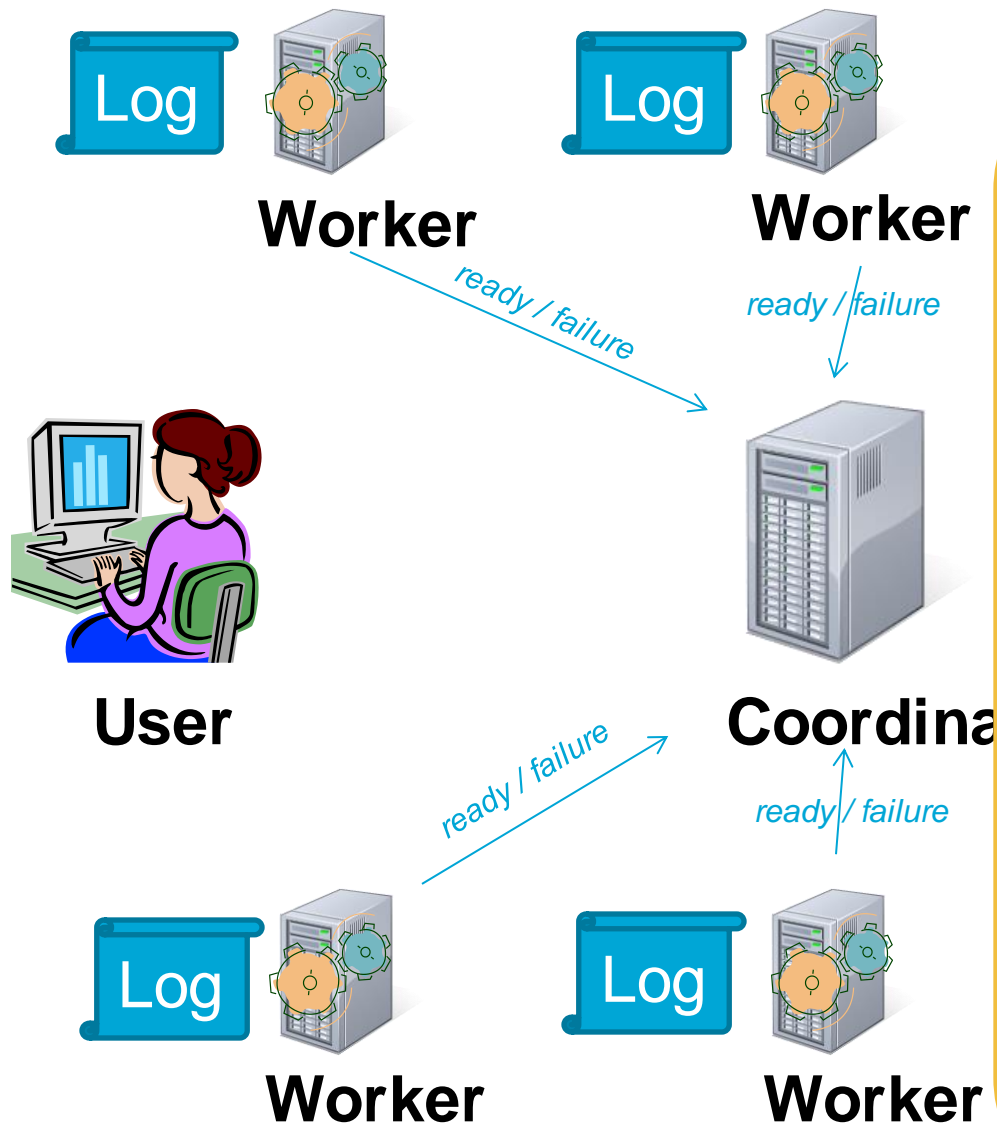


# Two Phase Commit (2PC) – Commit Request Phase



1. System receives user request.
2. Coordinator sends „*Prepare Commit*“ message to all workers.

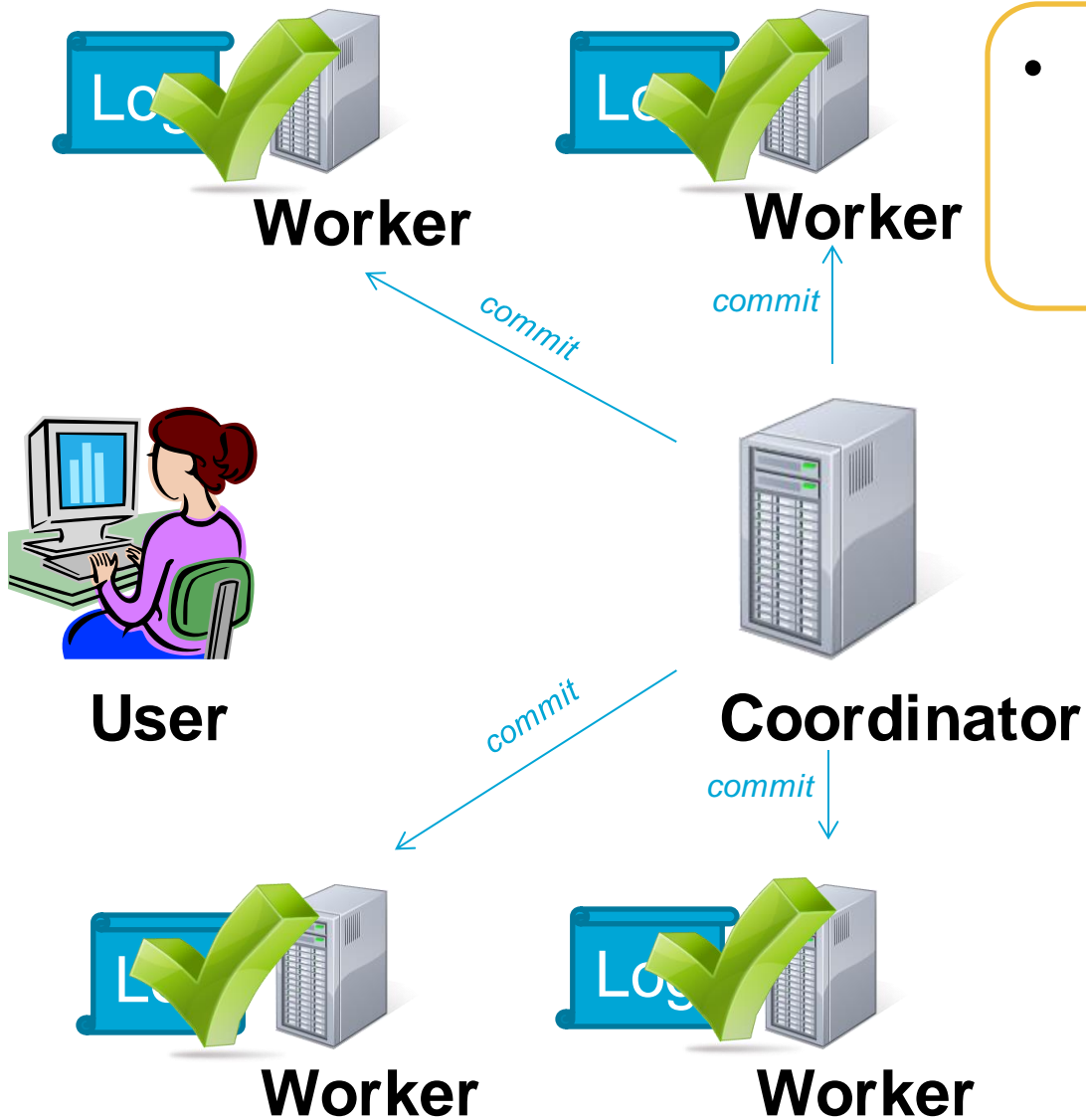
# Two Phase Commit (2PC) – Commit Request Phase



1. System receives user request.
2. Coordinator sends „*Prepare Commit*“ message to all workers.
3. Workers process request and write pending changes to log.
  - If successful, worker answers „*Ready*“.
  - In case of errors, worker answers „*Failure*“.

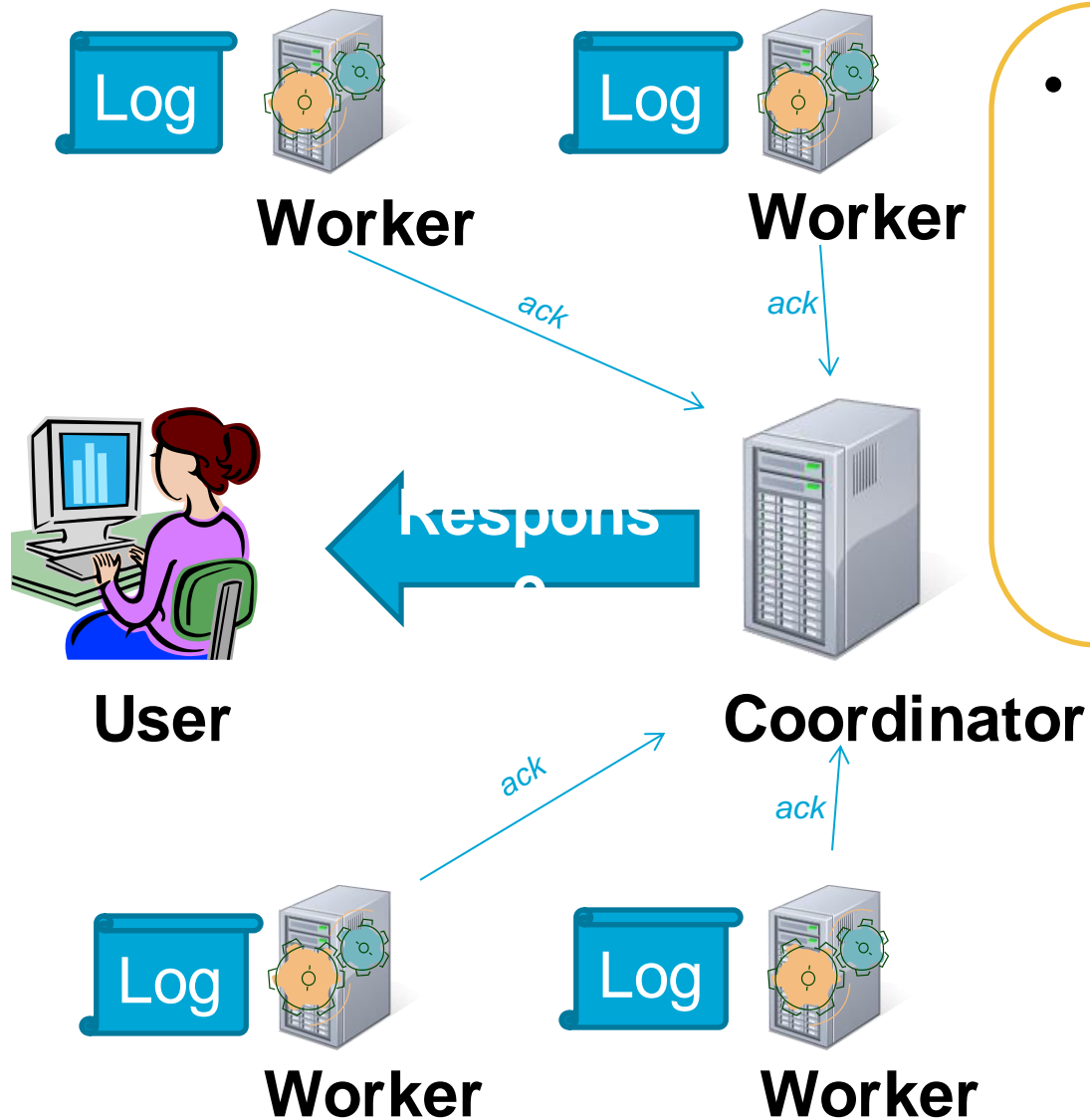


# Two Phase Commit (2PC) – Commit Phase (success)



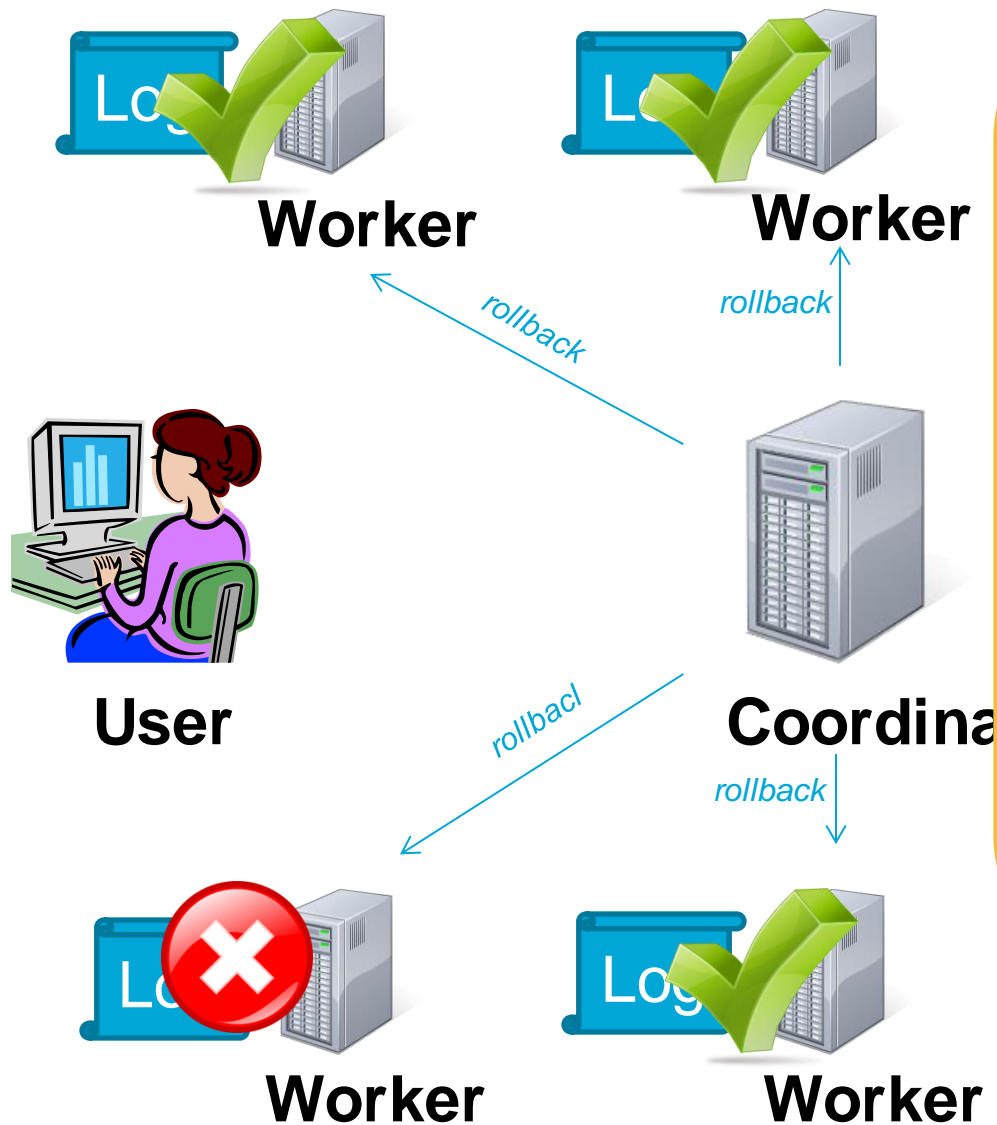
- All workers answered *“ready”*:
  1. Coordinator sends *“commit”* to all workers.

# Two Phase Commit (2PC) – Commit Phase (success)



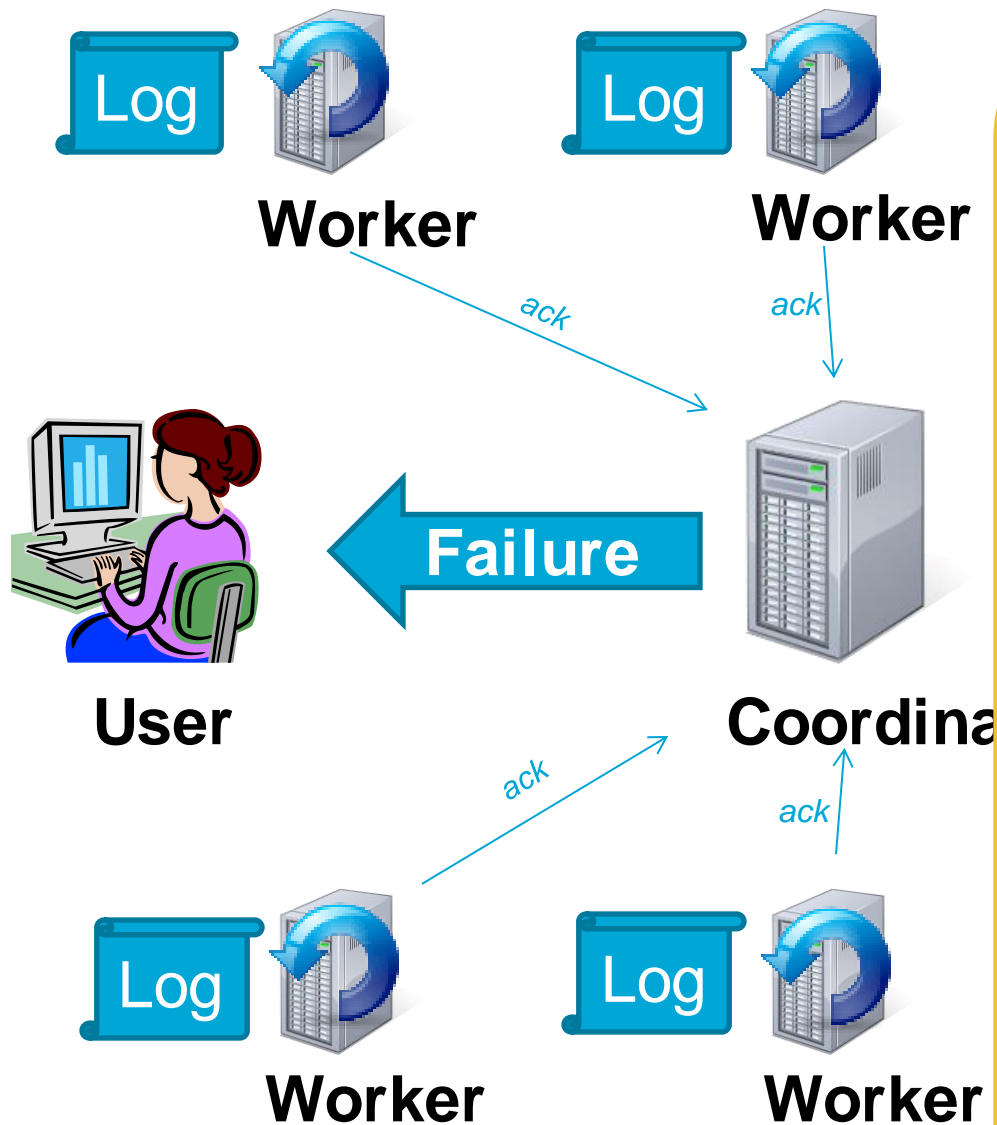
- All workers answered “*ready*”:
  1. Coordinator sends “*commit*” to all workers.
  2. Workers commit pending changes from log; Send “*ack*” to coordinator.
  3. Coordinator completes transaction, once all workers finished.

# Two Phase Commit (2PC) – Commit Phase (abort)



- All workers answered “*ready*”:
  1. Coordinator sends “*commit*” to all workers.
  2. Workers commit pending changes from log; Send “*ack*” to coordinator.
  3. Coordinator completes transaction, once all workers finished.
- At least one answered “*failure*”:
  1. Coordinator sends “*rollback*” to all workers.

# Two Phase Commit (2PC) – Commit Phase (abort)



- All workers answered *“ready”*:
  1. Coordinator sends *“commit”* to all workers.
  2. Workers commit pending changes from log; Send *“ack”* to coordinator.
  3. Coordinator completes transaction, once all workers finished.
- At least one answered *“failure”*:
  1. Coordinator sends *“rollback”* to all workers.
  2. Workers undo pending changes from log; Send *“ack”* to coordinator.
  3. Coordinator undoes transaction, once all workers finished.

# Two Phase Commit (2PC) – Failure Recovery I

- Simplest case: One (or more) recoverable worker failure.
  - Assumption: Workers keep log of sent and received messages.
- Failure occurs in Phase 1:
  - Directly after receiving “Prepare” from Coordinator:
    - Transaction either timed out (failed) or Coordinator is still waiting. Make local decision and check with Coordinator.
  - After sending “Failure” to Coordinator:
    - The transaction has failed. Undo from log.
  - After sending “Ready” to Coordinator:
    - The transaction might have succeeded. Check back with coordinator or neighboring nodes.
- Failure occurs in Phase 2:
  - After receiving “Abort” from Coordinator:
    - Undo transaction from log.

# Non-Acid Transactions



 **riak**

**ORACLE®**  
NOSQL  
DATABASE

  
**CouchDB**

 **mongoDB**

**APACHE**  
**HBASE**

  
**cassandra**

# BASE Transactions

- Many modern data storage systems (NoSQL systems) do not support ACID transactions
  - NoSQL system are usually build for scale, running on many nodes
    - Usually, the same data item is stored with many redundant copies
      - Replica consistency: All replicas always have the same value
    - ACID transaction management has issues with scale and many replica

# CAP-Theorem

- In the following, we will examine some **trade-offs** involved when designing high performance **distributed and replicated** databases
  - **CAP Theorem**
    - “You can’t have a highly available partition-tolerant and consistent system”
  - **BASE Transactions**
    - Weaker than ACID transaction model following from the CAP theorem





# CAP-Theorem

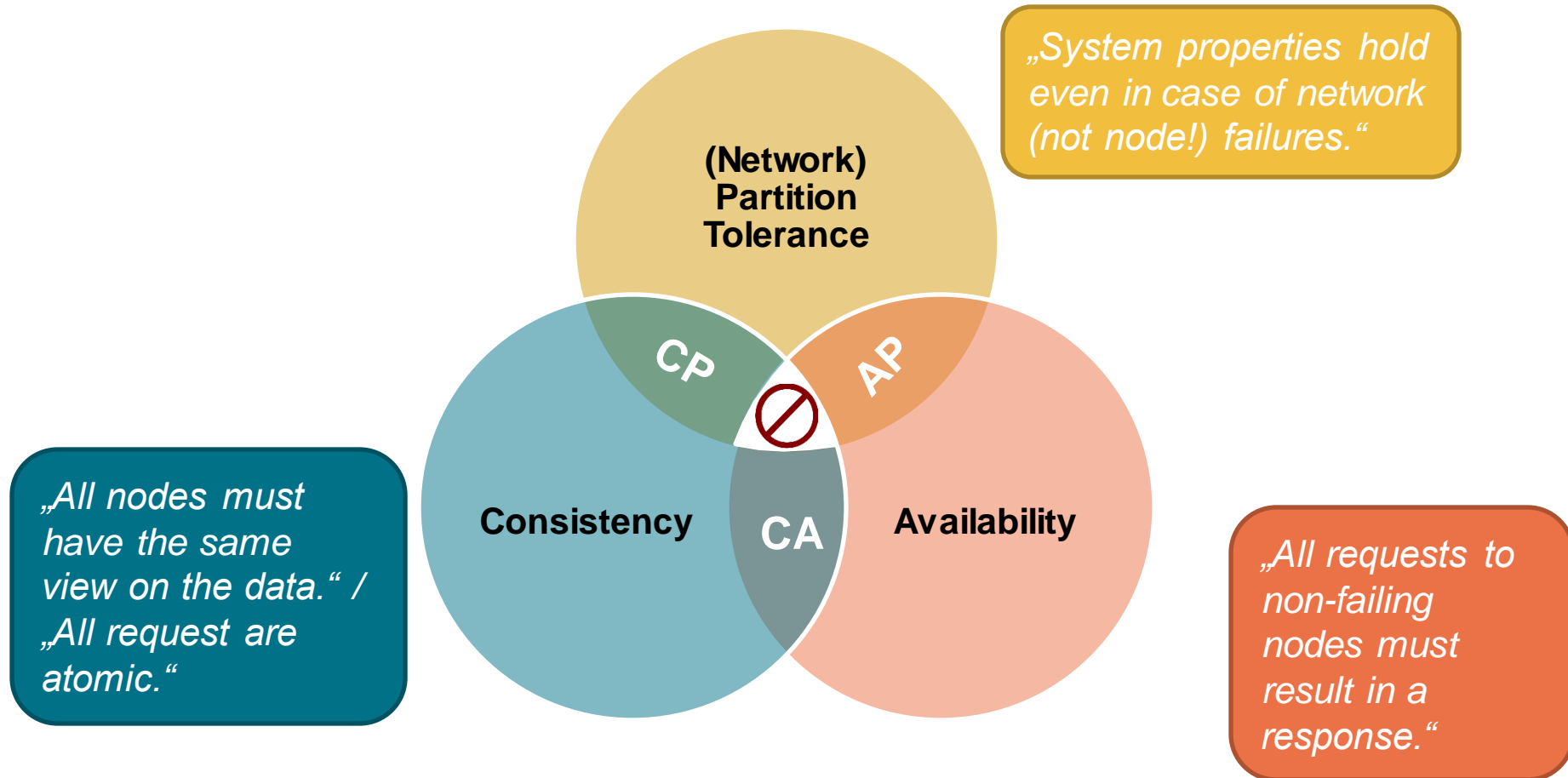
- The **CAP theorem** was made popular by **Eric Brewer** at the ACM Symposium of Distributed Computing (PODC)
  - Started as a conjecture, was later proven by Gilbert and Lynch
    - Seth Gilbert, Nancy Lynch. “*Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*”. ACM SIGACT News, 2002
  - CAP theorem limits the design space for highly-available distributed systems



# CAP-Theorem

- Assumption:
  - High-performance distributed storage system with replicated data fragments
- **CAP: Consistency, Availability, Partition Tolerance**
- **Consistency**
  - Not to be confused with ACID consistency
    - CAP is not about transactions, but about the design space of highly available data storage
  - Consistent means that all replicas of a fragment are always equal
    - Thus, CAP consistency is more similar to ACID atomicity: an update to the system atomically updates all replicas
    - Replica Consistency!
  - At a given time, all nodes see the same data

# CAP-Theorem



# CAP-Theorem

- **Availability**

- The data service is **available and fully operational**
- Any node failure will allow the survivors to continue operation without any restrictions
  - Especially: If you can **reach** a node, it can also be used for **reading** and **writing**
- Common problem with availability:  
**Availability most often fails when you need it most**
  - i.e. failures during busy periods because the system is busy



# CAP-Theorem

- **Partition Tolerance**
  - No set of **network failures** less than total network crash is allowed to cause the system to respond incorrectly
  - **Partition**
    - Set of nodes which can communicate with each other
    - The whole node set should always be one big partition
  - However, often multiple **partitions** may form
    - Assumption: short-term network partitions form very frequently
    - Thus, not all nodes can communicate with each other
    - Partition tolerant system must either
      - prevent this case of ever happening
      - or tolerate forming and merging of partitions without producing failures

# CAP-Theorem

- Finally: **The CAP theorem**
  - “Any **highly-scalable** distributed storage system using replication can only achieve a **maximum of two** properties out of **consistency**, **availability** and **partition tolerance**”
    - Thus, only compromises are possible
  - In most cases, **consistency** is sacrificed
    - Availability and partition tolerance keeps your business (and money) running
    - Many application can live with minor inconsistencies

# BASE Transactions

- **Consideration:**  
Maybe full ACID properties are not always necessary?
  - Allow the availability counter to be out-of sync?
    - Use a cached availability which is updated eventually
  - Allow rare cases where a user buys a book while unfortunately the last copy was already sold?
    - Cancel the order and say you are very sorry...
- These consideration lead to the **BASE** transaction model!
  - Sacrifice transactional consistency for scalability and features!

# BASE Transactions

- Accepting **eventual consistency** leads to new application and transaction paradigms
- **BASE transactions**
  - Directly follows from the CAP theorem
    - To be honest: BASE is both a silly acronym and also a silly theorem...
  - **Basic Availability**
    - Focus on availability – even if data is outdated, it should be available
  - **Soft-State**
    - Allow inconsistent states
  - **Eventual Consistent**
    - Sooner or later, all data will be consistent and in-sync
    - In the meantime, data is **stale** and queries return just approximate answers
    - Consistency here means replica consistency



# Requirements for distributed systems

- Instead providing ACID requirements, modern internet systems focus on BASE:
  - Basically Available.
  - Soft State.
  - Eventually Consistent.

## ACID

- Strong consistency for transactions highest priority
- Availability less important
- Pessimistic
- Rigorous analysis
- Complex mechanisms

## BASE

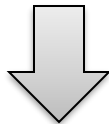
- Availability and scaling highest priorities
- Weak consistency
- Optimistic
- Best effort
- Simple and fast

# BASE Transactions

- The transition between **ACID** and **BASE** is a continuum
  - You may place your application wherever you need it to between ACID and BASE



You?



**ACID**

**BASE**



+ Guaranteed Transactional Consistency  
- Severe Scalability issues

+ High scalability and performance  
- Eventually consistent, approximate query results