

# Transactions

# Overview

- Transactions
  - Concept
  - ACID properties
  - Examples and counter-examples
- Implementation techniques
- Weak isolation issues

# Further Reading

- Transaction concept: 《A First Course in Database Systems》 Garcia-Molina et al Chapter 6.6
- Implementation techniques: Garcia-Molina et al Chapters 6-8
- Big picture: “Principles of Transaction Processing” by P. Bernstein and E. Newcomer
- The gory details: “Transaction Processing” by J. Gray and A. Reuter

# Definition

- A transaction is a collection of one or more operations on one or more databases, which reflects a single real-world transition
  - In the real world, this happened (completely) or it didn't happen at all (**Atomicity**)
- Commerce examples
  - Transfer money between accounts
  - Purchase a group of products
- Student record system
  - Register for a class (either waitlist or allocated)

# Coding a transaction

- Typically a computer-based system doing OLTP has a collection of *application programs*
- Each program is written in a high-level language, which calls DBMS to perform individual SQL statements
  - Either through embedded SQL converted by preprocessor
  - Or through Call Level Interface where application constructs appropriate string and passes it to DBMS

# Why write programs?

- Why not just write a SQL statement to express “what you want”?
- An individual SQL statement can’t do enough
  - It can’t update multiple tables
  - It can’t perform complicated logic (conditionals, looping, etc)

# COMMIT

- As app program is executing, it is “in a transaction”
- Program can execute COMMIT
  - SQL command to finish the transaction successfully
  - The next SQL statement will automatically start a new transaction

# Warning

- The idea of a transaction is hard to see when interacting directly with DBMS, instead of from an app program
- Using an interactive query interface to DBMS, by default each SQL statement is treated as a separate transaction (with implicit COMMIT at end) unless you explicitly say “START TRANSACTION”



# A Limitation

- Some systems rule out having both DML and DDL statements in a single transaction
- I.E., you can change the schema, or change the data, but not both

# ROLLBACK

- If the app gets to a place where it can't complete the transaction successfully, it can execute ROLLBACK
- This causes the system to “abort” the transaction
  - The database returns to the state without any of the previous changes made by activity of the transaction

# Reasons for Rollback

- User changes their mind (“ctl-C”/cancel)
- App program finds a problem
  - Eg qty on hand < qty being sold
- System-initiated abort
  - System crash
  - Housekeeping
    - Eg due to timeouts

# Atomicity

- Two possible outcomes for a transaction
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made
- That is, transaction's activities are **all** or **nothing**

# Integrity

- A real world state is reflected by collections of values in the tables of the DBMS
- But not every collection of values in a table makes sense in the real world
- The state of the tables is restricted by **integrity constraints**
- Eg account number is unique
- Eg stock amount can't be negative

# Integrity (ctd)

- Many constraints are explicitly declared in the schema
  - So the DBMS will enforce them
  - Especially: primary key (some column's values are non null, and different in every row)
  - And referential integrity: value of foreign key column is actually found in another “referenced” table
- Some constraints are not declared
  - They are business rules that are supposed to hold

# Consistency

- Each transaction can be written on the assumption that all integrity constraints hold in the data, before the transaction runs
- It must make sure that its changes leave the integrity constraints still holding
  - However, there are allowed to be intermediate states where the constraints do not hold
- A transaction that does this, is called **consistent**
- This is an obligation on the programmer
  - Usually the organization has a testing/checking and sign-off mechanism before an application program is allowed to get installed in the production system

# System obligations

- Provided the app programs have been written properly,
- Then the DBMS is supposed to make sure that the state of the data in the DBMS reflects the real world accurately, as affected by all the committed transactions



# Local to global reasoning

- Organization checks each app program as a separate task
  - Each app program running on its own moves from state where integrity constraints are valid to another state where they are valid
- System makes sure there are no nasty interactions
- So the final state of the data will satisfy all the integrity constraints

# Example - Tables

- System for managing inventory
- InStore(prodID, storeID, qty)
- Product(prodID, desc, mnfr, ..., WarehouseQty)
- Order(orderNo, prodID, qty, rcvd, ....)
  - Rows never deleted!
  - Until goods received, rcvd is null
- Also Store, Staff, etc etc

# Example - Constraints

- Primary keys
  - InStore: (prodID, storeID)
  - Product: prodID
  - Order: orderId
  - etc
- Foreign keys
  - Instore.prodID references Product.prodID
  - etc

# Example - Constraints

- Data values
  - $\text{Instore.qty} \geq 0$
  - $\text{Order.rcvd} \leq \text{current\_date}$  or  $\text{Order.rcvd}$  is null
- Business rules
  - for each  $p$ , (Sum of qty for product  $p$  among all stores and warehouse)  $\geq 50$
  - for each  $p$ , (Sum of qty for product  $p$  among all stores and warehouse)  $\geq 70$  or there is an outstanding order of product  $p$

# Example - transactions

- MakeSale(store, product, qty)
- AcceptReturn(store, product, qty)
- RcvOrder(order)
- Restock(store, product, qty)
  - // move from warehouse to store
- ClearOut(store, product)
  - // move all held from store to warehouse
- Transfer(from, to, product, qty)
  - // move goods between stores

# Example - ClearOut

- Validate Input (appropriate product, store)
- ```
SELECT qty INTO :tmp
FROM InStore
WHERE StoreID = :store AND prodID = :product
```
- ```
UPDATE Product
SET WarehouseQty = WarehouseQty + :tmp
WHERE prodID = :product
```
- ```
UPDATE InStore
SET Qty = 0
WHERE StoreID = :store AND prodID = :product
```
- COMMIT

# Example - Restock

- Input validation
  - Valid product, store, qty
  - Amount of product in warehouse  $\geq$  qty
- UPDATE Product
  - SET WarehouseQty = WarehouseQty - :qty
  - WHERE prodID = :product
- If no record yet for product in store
  - INSERT INTO InStore (:product, :store, :qty)
- Else, UPDATE InStore
  - SET qty = qty + :qty
  - WHERE prodID = :product and storeID = :store
- COMMIT

# Example - Consistency

- How to write the app to keep integrity holding?
- MakeSale logic:
  - Reduce Instore.qty
  - Calculate sum over all stores and warehouse
  - If  $\text{sum} < 50$ , then ROLLBACK // Sale fails
  - If  $\text{sum} < 70$ , check for order where date is null
    - If none found, insert new order for say 25



# Example - Consistency

- We don't need any fancy logic for the business rules in AcceptReturn, Restock, ClearOut, Transfer
  - *Why?*
- What is logic needed for RcvOrder?

# Threats to data integrity

- Need for application rollback
  - System crash
  - Concurrent activity
- 
- The system has mechanisms to handle these

# Application rollback

- A transaction may have made changes to the data before discovering that these aren't appropriate
  - the data is in state where integrity constraints are false
  - Application executes ROLLBACK
- System must somehow return to earlier state
  - Where integrity constraints hold
- So aborted transaction has no effect at all

# Example

- While running MakeSale, app changes InStore to reduce qty, then checks new sum
- If the new sum is below 50, txn aborts
- System must change InStore to restore previous value of qty
  - Somewhere, system must remember what the previous value was!

# System crash

- At time of crash, an application program may be part-way through (and the data may not meet integrity constraints)
- Also, buffering can cause problems
  - Note that system crash loses all buffered data, restart has only disk state
  - Effects of a committed txn may be only in buffer, not yet recorded in disk state
  - Lack of coordination between flushes of different buffered pages, so even if current state satisfies constraints, the disk state may not

# Example

- Suppose crash occurs after
  - MakeSale has reduced InStore.qty
  - found that new sum is 65
  - found there is no unfilled order
  - // but before it has inserted new order
- At time of crash, integrity constraint did not hold
- Restart process must clean this up (effectively aborting the txn that was in progress when the crash happened)

# Concurrency

- When operations of concurrent threads are interleaved, the effect on shared state can be unexpected
- Well known issue in operating systems, thread programming
  - see OS textbooks on critical section
  - Java use of synchronized keyword

# Famous anomalies

- Dirty data
  - One task T reads data written by T' while T' is running, then T' aborts (so its data was not appropriate)
- Lost update
  - Two tasks T and T' both modify the same data
  - T and T' both commit
  - Final state shows effects of only T, but not of T'
- Inconsistent read
  - One task T sees some but not all changes made by T'
  - The values observed may not satisfy integrity constraints
  - This was not considered by the programmer, so code moves into absurd path



# Example – Dirty data

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 25  |
| p1  | s2  | 70  |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 10  |
| p2  | etc | 44  |
| etc | etc | etc |

Initial state of InStore, Product

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 25  |
| p1  | s2  | 5   |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 10  |
| p2  | etc | 44  |
| etc | etc | etc |

Final state of InStore, Product

- AcceptReturn(p1,s1,50) MakeSale(p1,s2,65)
- Update row 1: 25 -> 75
- update row 2: 70->5
- find sum: 90
- // no need to insert
- // row in Order
- Abort
- // rollback row 1 to 25
- COMMIT

Integrity constraint is false:  
Sum for p1 is only 40!

# Example – Lost update

- ClearOut(p1,s1) **AcceptReturn(p1,s1,60)**
- Query InStore; qty is 25
- Add 25 to WarehouseQty: 40->65
- **Update row 1: 25->85**
- Update row 1, setting it to 0
- COMMIT
- **COMMIT**

60 returned p1's have vanished from system; total is still 115

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 25  |
| p1  | s2  | 50  |
| p2  | s1  | 45  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 40  |
| p2  | etc | 55  |
| etc | etc | etc |

Initial state of InStore, Product

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 0   |
| p1  | s2  | 50  |
| p2  | s1  | 45  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 65  |
| p2  | etc | 55  |
| etc | etc | etc |

Final state of InStore, Product

# Example – Inconsistent read

- ClearOut(p1,s1)      **MakeSale(p1,s2,60)**
- Query InStore: qty is 30
- Add 30 to WarehouseQty: 10->40
- **update row 2: 65->5**
- **find sum: 75**
- **// no need to insert**
- **// row in Order**
- Update row 1, setting it to 0
- COMMIT
- **COMMIT**

Integrity constraint is false:  
Sum for p1 is only 45!

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 30  |
| p1  | s2  | 65  |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 10  |
| p2  | etc | 44  |
| etc | etc | etc |

Initial state of InStore, Product

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 0   |
| p1  | s2  | 5   |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 40  |
| p2  | etc | 44  |
| etc | etc | etc |

Final state of InStore, Product

# Serializability

- To make isolation precise, we say that an execution is serializable when
- There exists some serial (ie batch, no overlap at all) execution of the same transactions which has the same final state
  - Hopefully, the real execution runs faster than the serial one!
- NB: different serial txn orders may behave differently; we ask that *some* serial order produces the given state
  - Other serial orders may give different final states

# Example – Serializable execution

- ClearOut(p1,s1)      MakeSale(p1,s2,20)
- Query InStore: qty is 30
- update row 2: 45->25
- find sum: 65
- no order for p1 yet
- Add 30 to WarehouseQty: 10->40
- Update row 1, setting it to 0
- COMMIT
- Insert order for p1
- COMMIT

Execution is like serial  
MakeSale; ClearOut

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 30  |
| p1  | s2  | 45  |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 10  |
| p2  | etc | 44  |
| etc | etc | etc |

Order: empty

Initial state of InStore, Product, Order

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 0   |
| p1  | s2  | 25  |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 40  |
| p2  | etc | 44  |
| etc | etc | etc |

|    |    |      |     |
|----|----|------|-----|
| p1 | 25 | Null | etc |
|----|----|------|-----|

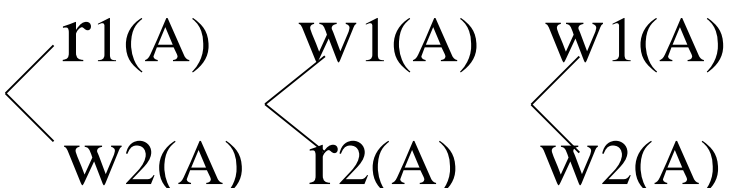
Final state of InStore, Product, Order

# Serializability Theory

- There is a beautiful mathematical theory, based on formal languages
  - Eg the task of testing whether an execution is serializable is NP
- There is a nice sufficient condition (ie a conservative approximation) called **conflict serializable**, which can be efficiently tested
  - based on absence of cycles in a graph
- Most people and books use the approximation, usually without mentioning it!

# Concepts

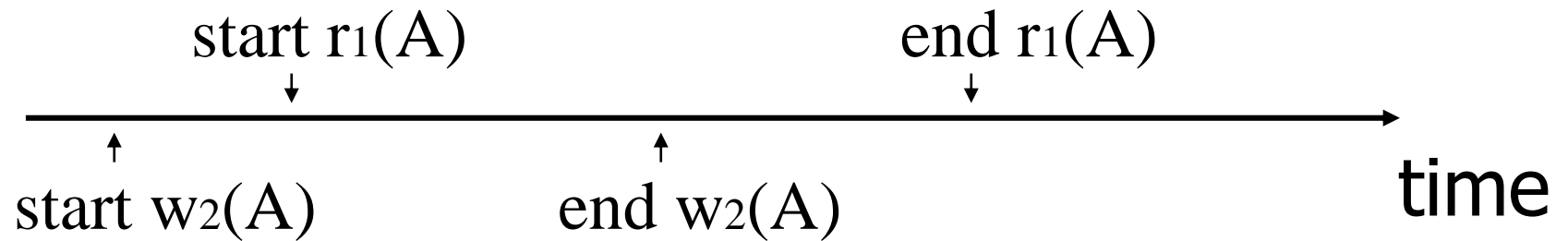
*Transaction*: sequence of  $r_i(x)$ ,  $w_i(x)$  actions

*Conflicting actions*: 

*Schedule*: represents chronological order  
in which actions are executed

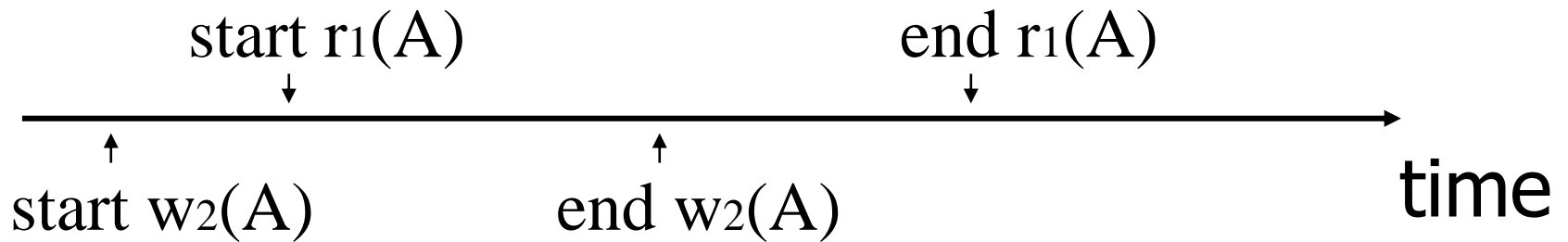
*Serial schedule*: no interleaving of actions  
or transactions

What about conflicting, concurrent actions on same object?





What about conflicting, concurrent actions on same object?



- Assume equivalent to either  $r_1(A) w_2(A)$   
or  $w_2(A) r_1(A)$
- $\Rightarrow$  low level synchronization mechanism
- Assumption called "atomic actions"

## Definition

$S_1, S_2$  are conflict equivalent schedules  
if  $S_1$  can be transformed into  $S_2$  by a series  
of swaps on non-conflicting actions.

## Definition

A schedule is conflict serializable if it is conflict equivalent to some serial schedule.

## Exercise:

$S = w_3(A) \ w_2(C) \ r_1(A) \ w_1(B) \ r_1(C) \ w_2(A) \ r_4(A) \ w_4(D)$

Is  $S$  conflict serializable ?

# ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

# Big Picture

- If programmer writes applications so each txn is consistent
- And DBMS provides atomic, isolated, durable execution
  - Ie actual execution has same effect as some serial execution of those txns that committed (but not those that aborted)
- Then the final state will satisfy all the integrity constraints

NB true even though system does not know all integrity constraints!

# Overview

- Transactions
- Implementation Techniques
  - Ideas, not details!
  - Implications for application programmers
  - Implications for DBAs
- Weak isolation issues

# Main implementation techniques

- Logging
  - Interaction with buffer management
  - Use in restart procedure
- Locking
- Distributed Commit



# Logging

- The log is an append-only collection of entries, showing all the changes to data that happened, in order as they happened
- Eg when T1 changes field qty in row 3 from 15 to 75, this fact is recorded as a log entry
- Log also shows when txns start/commit/abort

# A log entry

- LSN: identifier for entry, increasing values
- Txn id
- Data item involved
- Old value
- New value
  - Sometimes there are separate logs for old values and new values

# Extra features

- Log also records changes made by system itself
  - Eg when old value is restored during rollback
- Log entries are linked for easier access to past entries
  - Link to previous log entry
  - Link to previous entry for the same txn

# Buffer management

- Each page has place for LSN of most recent change to that page
- When a page is fetched into buffer, DBMS remembers latest LSN at that time
- Log itself is produced in buffer, and flushed to disk (appending to previously flushed parts) from time to time
- Important rules govern when buffer flushes can occur, relative to LSNs involved
  - Sometimes a flush is forced (eg log flush forced when txn commits)

# Using the log

- To rollback txn T
  - Follow chain of T's log entries, *backwards*
  - For each entry, restore data to old value, and produce new log record showing the restoration
  - Produce log record for “abort T”

# Restart

- After a crash, follow the log *forward*, replaying the changes
  - i.e. re-install new value recorded in log
- Then rollback all txns that were active at the end of the log
- Now normal processing can resume

# Optimizations

- Use LSNs recorded in each page of data, to avoid repeating changes already reflected in page
- Checkpoints: flush pages that have been in buffer too long
  - Record in log that this has been done
  - During restart, only repeat history since last (or second-last) checkpoint

# Checkpointing

- What is the problem
- Checkpoints
- Recovery by checkpointing



# What is the problem

Much time cost for

- retrieving the whole log
- redo processing

# Solution

- Recovery by checkpointing
  - Add checkpoint records in the log
  - Add restart file
  - Maintain log file dynamically

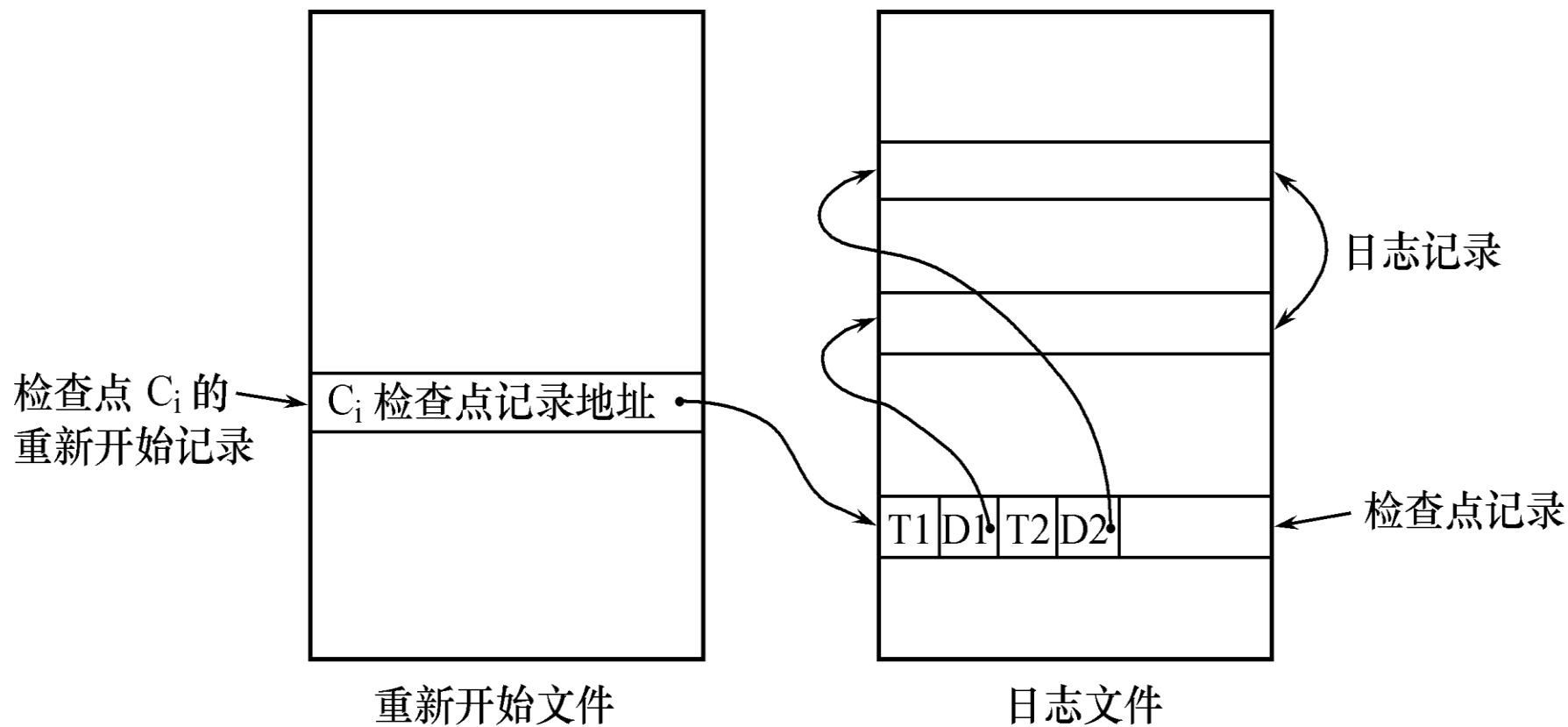
# Checkpoints

## ■ Checkpoint record

- 1. a list for all active transactions when checkpointing
- 2. addresses for the latest log record of active transactions

## ■ Restart file

- Record the addresses for checkpoint records in log



Log file with checkpoint record and restart file

# Maintain log dynamically

Execute the following operations periodically for building checkpoints and saving database status:

- 1.flush log in log buffer to disk
- 2.write a log record <CKPT>
- 3.flush data in data buffer to disk
- 4.write the address for <CKPT> record in log file into the restart file

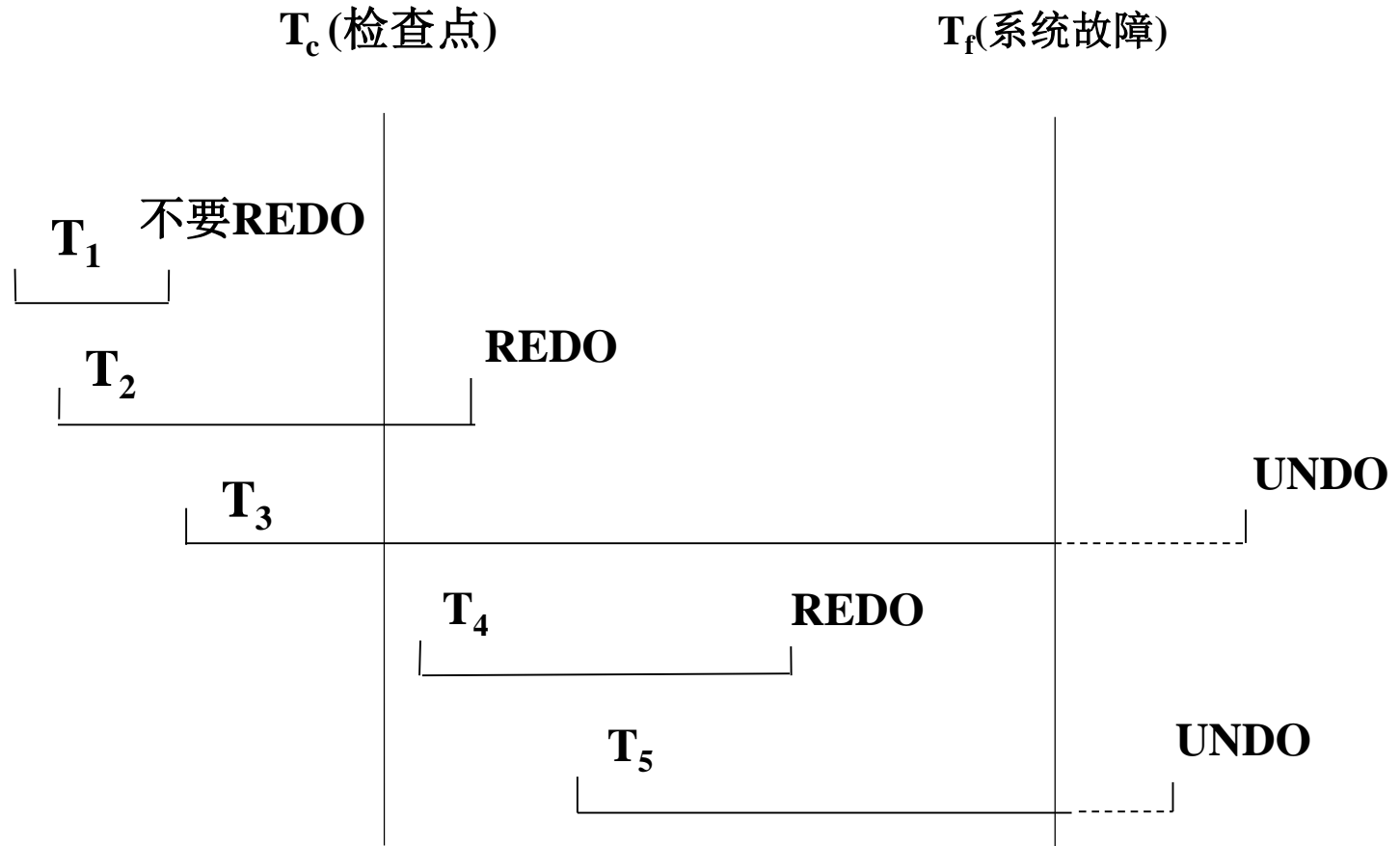
# Building checkpoints

- recovery system can build checkpoints and save database status periodically or from time to time
  - periodically  
according to a preset time interval. For example, build checkpoint hourly
  - From time to time  
according to some rule, For example, build checkpoint when log file is half full

# Recovery by checkpointing

- By checkpointing, the efficiency of recovery can be improved
  - When transaction T commits before checkpointing
  - the modification by T has been written into disk
  - The written time is when checkpoing or before checkpoing
  - So no REDO processing is needed for T during recovery

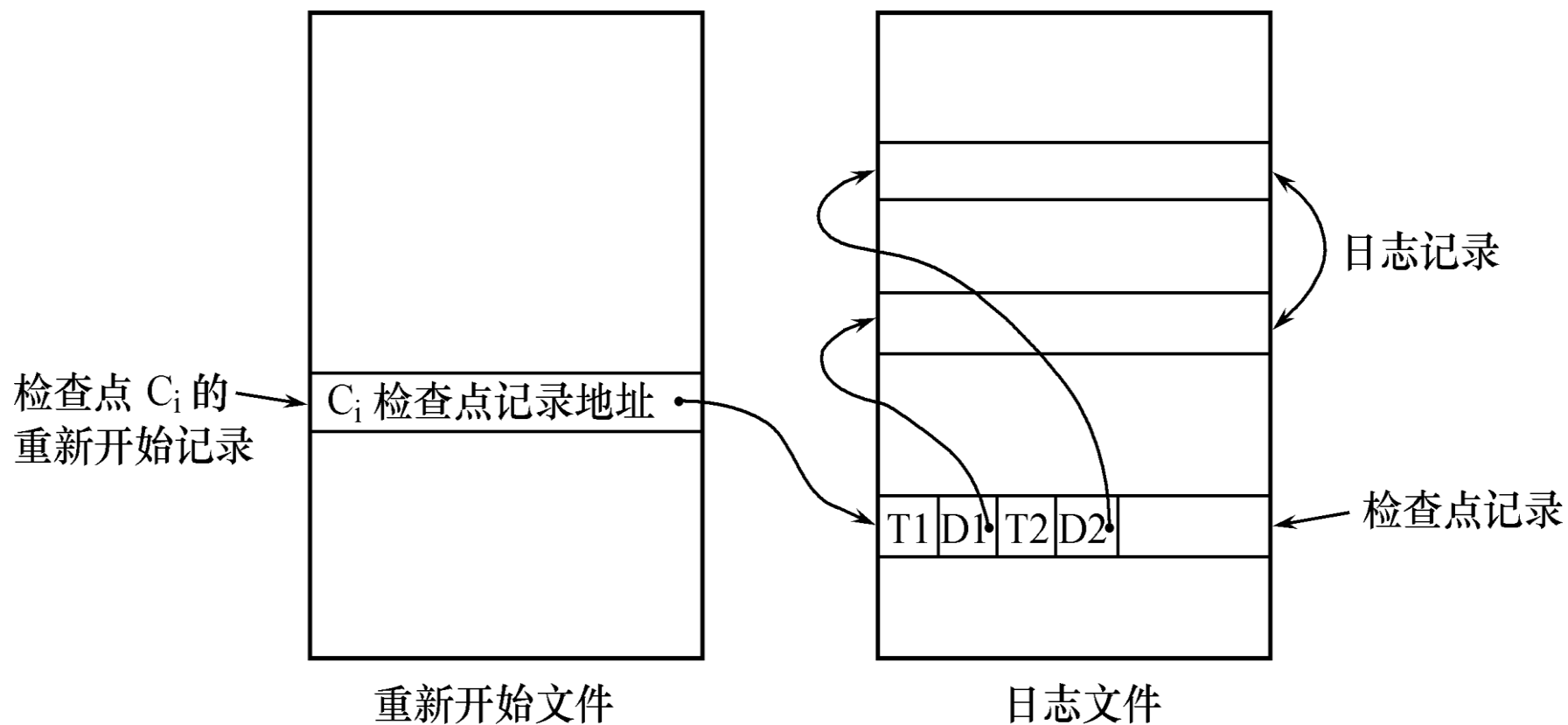
**When system failure occurs, recovery system will use different strategies according to transaction status**





# Recovery steps using checkpoints

1. Find from restart file the address of the latest checkpoint record in log, from which we can get the latest checkpoint record in log
2. From the checkpoint record, we can obtain a list of active transactions called ACTIVE-LIST when building checkpoint
  - Build two transaction lists
    - UNDO-LIST
    - REDO-LIST
  - Put all transactions in ACTIVE-LIST into UNDO-LIST, that is, REDO-LIST is empty in that time.



# Recovery steps using checkpoints

3. Follow the log forward from the checkpoint record to end
  - If we meet new transaction  $T_i$ , put  $T_i$  into UNDO-LIST temporarily
  - If transaction  $T_j$  is committed, move  $T_j$  from UNDO-LIST to REDO-LIST
4. Execute UNDO for each transaction in UNDO-LIST, and Execute REDO for each transaction in REDO-LIST

# Don't be too confident

- Crashes can occur during rollback or restart!
  - Algorithms must be **idempotent**
- Must be sure that log is stored separately from data (on different disk array; often replicated off-site!)
  - In case disk crash corrupts data, log allows fixing this
  - Also, since log is append-only, don't want have random access to data moving disk heads away

# Complexities

- Changes to index structures
  - Avoid logging every time index is rearranged
- Multithreading in log writing
  - Use standard OS latching to prevent different tasks corrupting the log's structure

# ARIES

- Until 1992, textbooks and research papers described only simple logging techniques that did not deal with complexities
- Then C. Mohan (IBM) published a series of papers describing ARIES algorithms
  - Papers are very hard to read, and omit crucial details, but at least the ideas of real systems are now available!

# Implications

- For application programmer
  - Choose txn boundaries to include everything that must be atomic
  - Use ROLLBACK to get out from a mess
- For DBA
  - Tune for performance: adjust checkpoint frequency, amount of buffer for log, etc
  - Look after the log!

# Main implementation techniques

- Logging
- Locking
  - Lock manager
  - Lock modes
  - Granularity
  - User control
- Distributed Commit



# Lock manager

- A structure in (volatile memory) in the DBMS which remembers which txns have set locks on which data, in which modes
- It rejects a request to get a new lock if a conflicting lock is already held by a different txn
- NB: a lock does not actually prevent access to the data, it only prevents getting a conflicting lock
  - So data protection only comes if the right lock is requested before every access to the data

# Lock modes

- Locks can be for writing (W), reading (R) or other modes
- Standard conflict rules: two W locks on the same data item conflict, so do one W and one R lock on the same data
  - However, two R locks do not conflict
- Thus W=exclusive, R=shared

# Automatic lock management

- DBMS requests the appropriate lock whenever the app program submits a request to read or write a data item
- If lock is available, the access is performed
- If lock is not available, the whole txn is **blocked** until the lock is obtained
  - After a conflicting lock has been released by the other txn that held it

# Strict two-phase locking

- Locks that a txn obtains are kept until the txn completes
  - Once the txn commits or aborts, then all its locks are released (as part of the commit or rollback processing)
- Two phases:
  - Locks are being obtained (while txn runs)
  - Locks are released (when txn finished)

# Serializability

- If each transaction does strict two-phase locking (requesting all appropriate locks), then executions are serializable
- However, performance does suffer, as txns can be blocked for considerable periods
  - Deadlocks can arise, requiring system-initiated aborts

# Example – No Dirty data

- AcceptReturn(p1,s1,50) MakeSale(p1,s2,65)
- Update row 1: 25 -> 75
- //t1 W-locks InStore. row 1
- update row 2: 70->5
- //t2 W-locks Instore.row2
- try find sum:// blocked
- // as R-lock on Instore.row1
- // can't be obtained
- Abort
- // rollback row 1 to 25; release lock
- // now get locks
- find sum: 40
- ROLLBACK
- // row 2 restored to 70
- 

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 25  |
| p1  | s2  | 70  |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 10  |
| p2  | etc | 44  |
| etc | etc | etc |

Initial state of InStore, Product

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 25  |
| p1  | s2  | 70  |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 10  |
| p2  | etc | 44  |
| etc | etc | etc |

Final state of InStore, Product

Integrity constraint is valid

# Example – No Lost update

- ClearOut(p1,s1)      AcceptReturn(p1,s1,60)
- Query InStore; qty is 25
- //t1 R-lock InStore.row1
- Add 25 to WarehouseQty: 40->65
- // t1 W-lock Product.row 1
- try Update row 1
- // blocked
- // as W-lock on InStore.row1
- // can't be obtained
- Update row 1, setting it to 0
- //t1 upgrades to W-lock on InStore.row1
- COMMIT // release t1's locks
- // now get W-lock
- Update row 1: 0->60
- COMMIT

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 25  |
| p1  | s2  | 50  |
| p2  | s1  | 45  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 40  |
| p2  | etc | 55  |
| etc | etc | etc |

Initial state of InStore, Product

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 60  |
| p1  | s2  | 50  |
| p2  | s1  | 45  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 65  |
| p2  | etc | 55  |
| etc | etc | etc |

Outcome is same as serial  
ClearOut; AcceptReturn

Final state of InStore, Product

# Granularity

- What is a data item (on which a lock is obtained)?
  - Most times, in most systems: item is a tuple in a table
  - Sometimes: item is a page (with several tuples)
  - Sometimes: item is a whole table
- In order to manage conflicts properly, system gets “intention” mode locks on larger granules before getting actual R/W locks on smaller granules



# Explicit lock management

- With most DBMS, the application program can include statements to set or release locks on a table
  - Details vary
- Eg LOCK TABLE InStore IN EXCLUSIVE MODE

# Implications

- For application programmer
  - If txn reads many rows in one table, consider locking the whole table first
  - Consider weaker isolation (see later)
- For DBA
  - Tune for performance: adjust max number of locks, granularity factors
  - Possibly redesign schema to prevent unnecessary conflicts
  - Possibly adjust query plans if locking causes problems

# Implementation mechanisms

- Logging
- Locking
- Distributed Commit

# Transactions across multiple DBMS

- Within one transaction, there can be statements executed on more than one DBMS
- To be atomic, we still need all-or-nothing
- That means: every involved system must produce the same outcome
  - All commit the txn
  - Or all abort it

# Why it's hard

- Imagine sending to each DBMS to say “commit this txn T now”
- Even though this message is on its way, any DBMS might abort T spontaneously
  - e.g. due to a system crash

NB unrelated to “two-phase locking”

# Two-phase commit

- The solution is for each DBMS to first move to a special situation, where the txn is “prepared”
- A crash won’t abort a prepared txn, it will leave it in prepared state
  - So all changes made by prepared txn must be recovered during restart (including any locks held before the crash!)

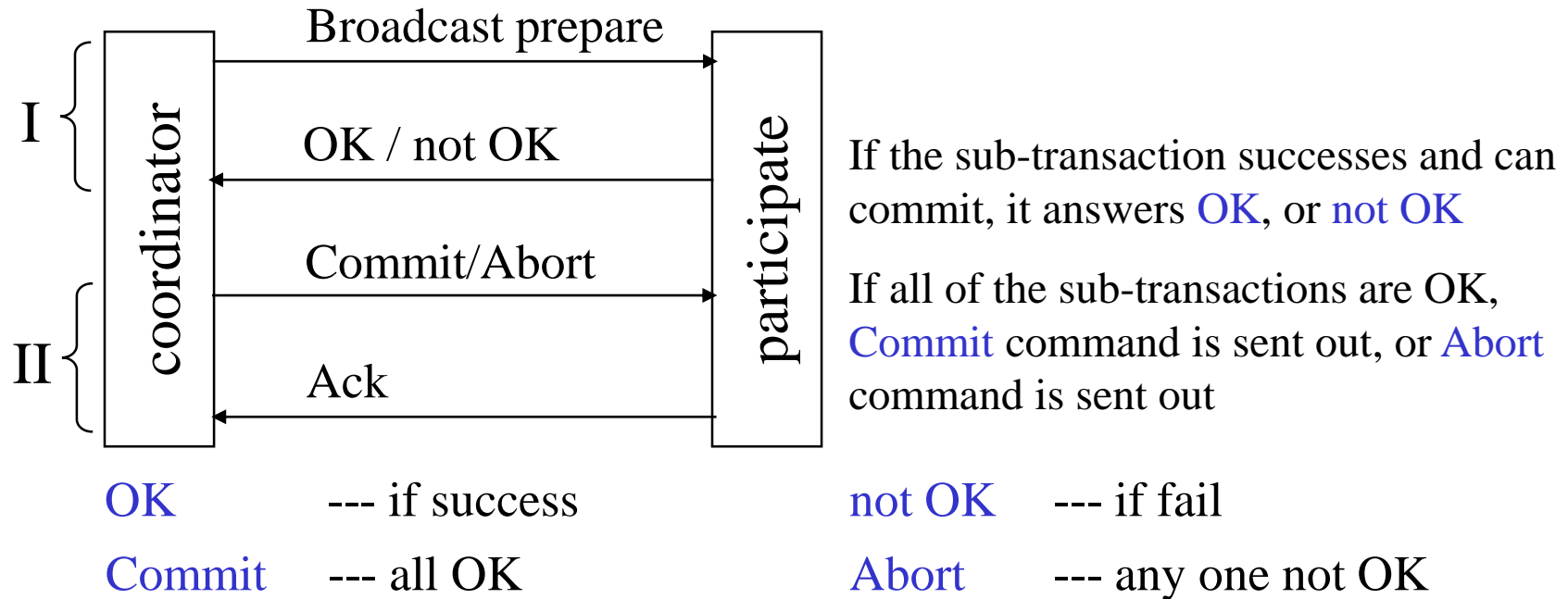
# Basic idea

- Two round-trips of messages
  - Request to prepare/ prepared or aborted
  - Either Commit/committed or Abort/aborted



Only if all DBMSs are already prepared!

# Two Phase Commit



- Every participant is self-determining before answering **OK**, it can abort by itself. Once answers **OK**, it can only wait for the command come from the coordinator.
- If the coordinator has failure after the participates answer **OK**, the participates have to wait, and is in blocked state. This is the disadvantage of 2PC.



# Read-only optimisation

- If a txn has involved a DBMS only for reading (but no modifications at that DBMS), then it can drop out after first round, without preparing
  - The outcome doesn't matter to it!
  - Special phase 1 reply: ReadOnly

# Fault-tolerant protocol

- The interchange of messages between the “coordinator” (part of the TPMonitor software) and each DBMS is tricky
  - Each participant must record things in log at specific times
  - But the protocol copes with lost messages, inopportune crashes etc

# Implications

- For application programmer
  - Avoid putting modifications to multiple databases in a single txn
    - Performance suffers a lot
    - W-Locks are held during the message exchanges, which take much longer than usual txn durations
- For DBA
  - Monitor performance carefully
  - Make sure you have DBMS that support protocol