More About Transaction Management

Introduction

- How transactions recover by maintaining serializability ?
- Distributed databases
- Long Transactions
  - Workflow systems

Outline

- **Transactions that read uncommitted data**
- View serializability
- Resolving deadlocks
- Distributed databases
- Distributed Commit
- Distributed locking
- Long duration transactions

Transactions that read uncommitted data

- While computation, values move between nonvolatile disk, volatile memory and local address space of a transaction.
- Logging system
  - Logging system able to reconstruct the state of committed transactions.
  - Logging system makes no effort to support serializability.
    - It will blindly reconstruct the state of the database, even the result is not serializable.
- The scheduler
  - Ignores the logging and write the new value of the data to database before committing, thus a violating the rule of logging policy.
  - If a transaction writes into database and then aborts, it will result into inconsistent database.

The Dirty-Data Problem

- Dirty data: if it is written by the transaction that is not yet committed.
  - The dirty data could appear either in the buffers, or on disk, or both.

Example

- T1 writes dirty data and aborts

T1	T2	A	B
l1(A); r1(A);			
A:=A+100;			
w1(A);l1(B);u1(A);			
	l2(A);r2(A)		
	A:=A+2;		
	w2(A)		
	l2(B) **denied**		
r1(B)			
Abort;u1(B)			
	l2(B);u2(A);r2(B)		
	B:=B*2;		
	w2(B); u2(B)		

Example

- T1 has read dirty data from T2 and must abort when T2 does

T1	T2	T3
	w2(B);	
r1(B);		
	r2(A);	
		r3(C);
	w2(C);	
	Abort;	
		w3(A);

Isolation levels in SQL

- SQL2 standard does not assume that every transaction runs in a serializable manner
- Serializable-level is the highest isolation level
- Problems
  - Dirty read problem: reading uncommitted data
  - UnRepeatable read problem: Successive reads get a different values
  - Phantom problem: Successive reads get additional tuples (not different values).

Transaction Support in SQL-92

- Each transaction has an access mode, a diagnostics size, and an isolation level.

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No

Cascading rollback

- If the dirty data is available to transactions we have to perform a cascading rollback.
  - When a T aborts, all transactions which have read the data written by T have to be aborted.
- Timestamp scheduler with commit bit avoids cascading rollback.
- Validation scheduler avoids cascading rollback

Managing Rollbacks in Lock-based Scheduler and Recoverable Schedules

- Strict locking
  - Do not release any locks until the transaction has committed or aborted, and commit or abort record is flushed to disk.
  - If a failure occurs after flushing the commit, the recovery manager commits the transaction.
- Recoverable schedules
  - A schedule of transactions that obey the strict locking rule is called recoverable.
- In a recoverable schedule, it is not possible for a transaction to read dirty data.

Managing Rollbacks

- If the lockable elements are blocks, no need of using the disk.
  - Pin the uncommitted writes in the main memory
  - If the transaction aborts, ignore the value. No other transaction reads the data as it is already locked.
- If the lockable elements are tuples, the above mechanism does not work.
  - Buffer may contain the changes made by more than one transaction.
- Options
  - Read the original value of A from disk
  - Obtain former value from the log itself
  - Maintain a separate main memory log for the active transactions

Group Commit

- We can avoid reading dirty data even if we do not flush every commit record on the log to disk immediately.
- If we flush the log records in the order they are written, we can release the locks as soon as commit record is written to the log in the buffer.
- Example: If T1 writes X and writes commit record in the buffer. T2 reads X and commits, but its commit record should follow the commit record flushing of T1.
  - Recovery manager
    - Finds neither T1 and T2 commit records
    - T1 is committed, T2 not
    - Both are committed
- Group Commit
  - Do not release locks unless commit record appears in the buffer.
  - Flush the log records in the order they were created

Logical Logging..

- Several problems with data elements as blocks
  - Great deal of redundant information is stored in the log.
  - Releasing locks only after the commit reduces concurrency.
- Logical logging: Only changes to the blocks are described

Logical logging

- A small number of bytes of database element changed
  - Tuple t has its attribute a changed from v1 to v2.
  - Sometimes sliding problem occurs. Use of overflow blocks may be required.
  - B-tree; only changes to the nodes can be logged.

Outline

- Transactions that read uncommitted data
- **View serializability**
- Resolving deadlocks
- Distributed databases
- Distributed Commit
- Distributed locking
- Long duration transactions

View Serializability

- We have studied the conflict serializability
- View serializability is another correctness criteria (weaker than conflict serializability)

Conflict Serializability

- Scheduler ensures that schedules are serializable by ensuring a condition conflict-serializability.
- It is based on the idea of conflict
- Conflict: Let Ti and Tj are transactions
  - ri(X); rj(Y) is never conflict even X=Y as they do not change any value
  - ri(X); wj(Y) is not a conflict provided that X≠Y.
  - wi(X); rj(Y) is not a conflict if X≠Y.
  - Wi(X); wj(Y) is not a conflict if X≠Y

- The actions of the same transaction conflict.; ri(X); wi(Y) conflict; order is fixed by a transaction. DBMS can not reorder!
- Two writes of different transaction on the same element conflict: wi(X), wj(X) conflict.
- Read and write of the same database element by different transactions also conflict. ri(X); wj(X) conflict. Also, wi(X) and ri(X) conflict.

Conflict Serializability

- Two actions of different transactions may be swapped unless
  - They involve the same database element, and
  - at least one of them is write.
- Carry out the non conflicting swaps and try to convert the schedule into a serial schedule.
- Conflict-equivalent
  - Two schedules are conflict equivalent if they can be turned one into other by a sequence of non-conflicting swaps of ADJACENT ACTIONS.
- Conflict serializable
  - If a schedule is conflict equivalent to a serial schedule.
- Conflict serializable schedule is a serializable schedule.
- Note:
  - Conflict serializability is not required for a schedule to be serializable.
  - But many commercial system use serializability criteria to guarantee serializabilty

View Serializability

- We have studied the conflict serializability
- View serializability is another correctness criteria (weaker than conflict serializability)
- View serializability considers all the connection of T and U, where T writes a database element whose value U reads.
- Difference
  - T writes own value of A and no other transaction reads. (some other transaction write own value of A)

View Equivalence

- Let S1 and S2 are two schedules of the same set of transactions
- Imagine T0 wrote intial values for each database element. Tf reads every element.
- Rules for view equivalence. Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

Example

T1:			r1(A)		w1(B)		
T2:	r2(B)	w2(A)				w2(B)	
T3:				r3(A)			w3(B)

- S is not confict serializable: T1 and T2 are deadlocked.
  - But w1(B) and w2(B) do not have long-term effects.

Example

T1: r1(A) w1(B)

T2: r2(B) w2(A) w2(B)

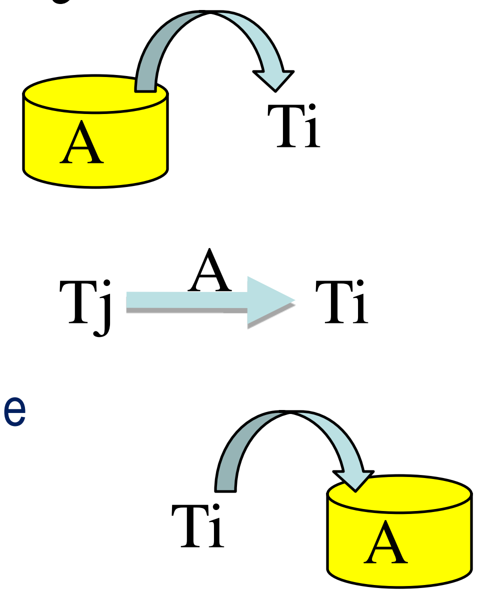
T3: r3(A) w2(B)

- Source r2(B) is T0
- Source of r1(A) is T2
- Source of r2(A) is T2.
- Source of hypothetical read of A by Tf is T2
- Source of hypothetical read of B by Tf is T3, the last writer of B.
- If we order the transactions (T2, T1, T3) the sources of all reads is the same as S
- So we conclude that S is a view equivalent serializable schedule.

View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

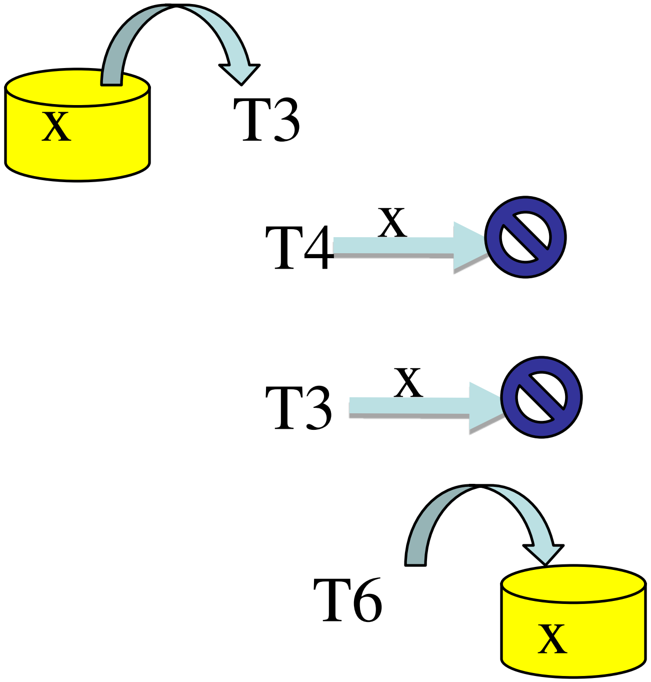
Time line

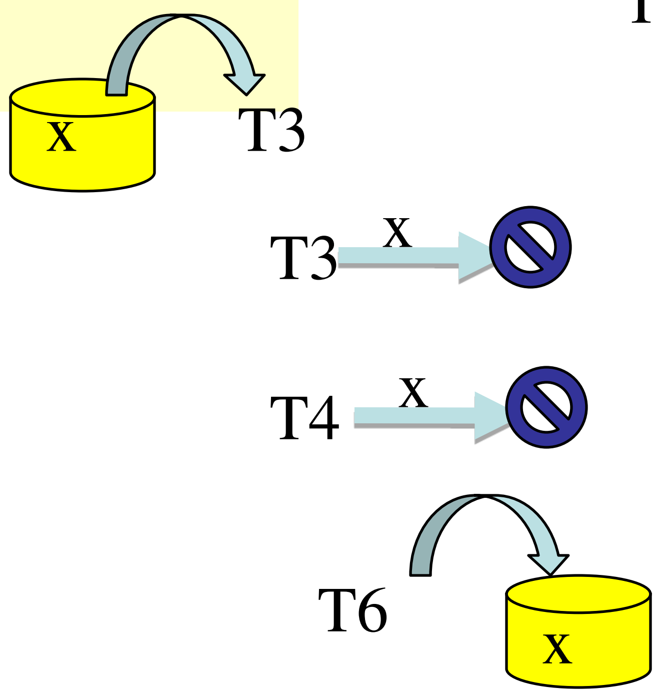


T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

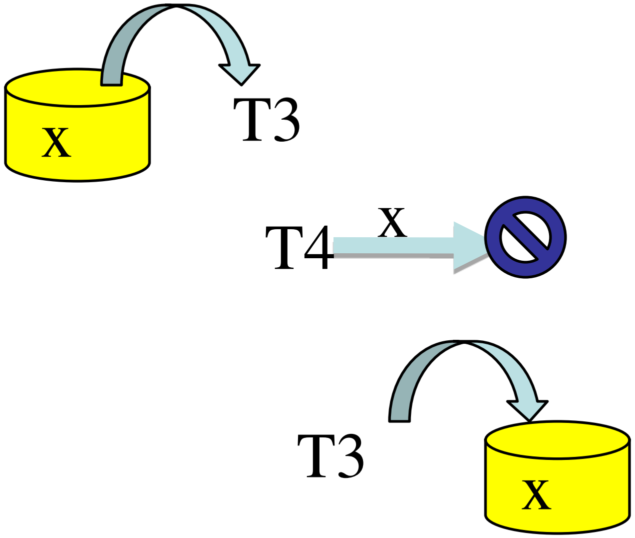
T1:	R(A),W(A)		
T2:		W(A)	
T3:			W(A)

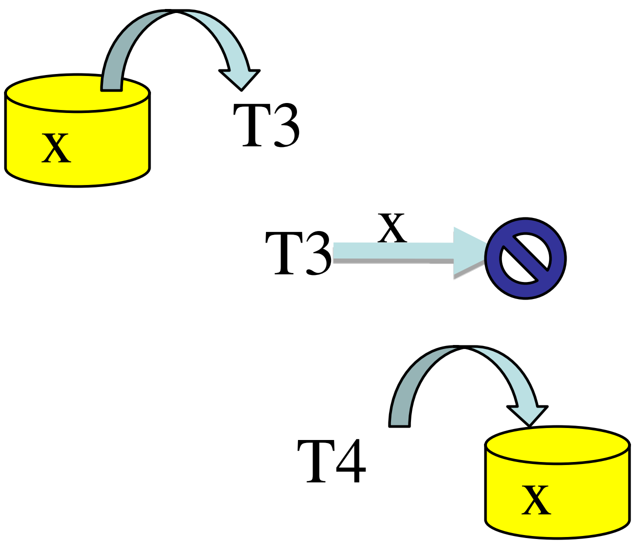
- $r_3[x]w_4[x]w_3[x]w_6[x]$
- T_3 read-from T_b.
- The final write for *x* is $w_6[x]$.
- View equivalent to T_3 T_4 T_6:
- r3[x]w3[x]w4[x]w6[x]

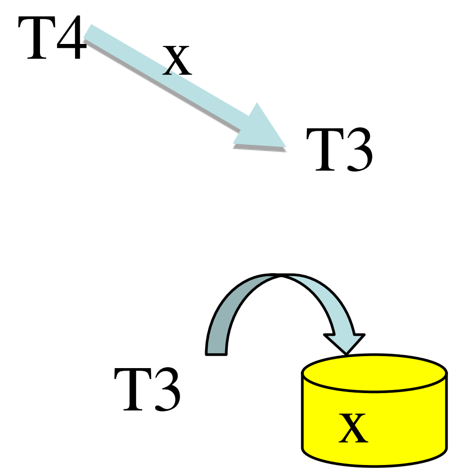




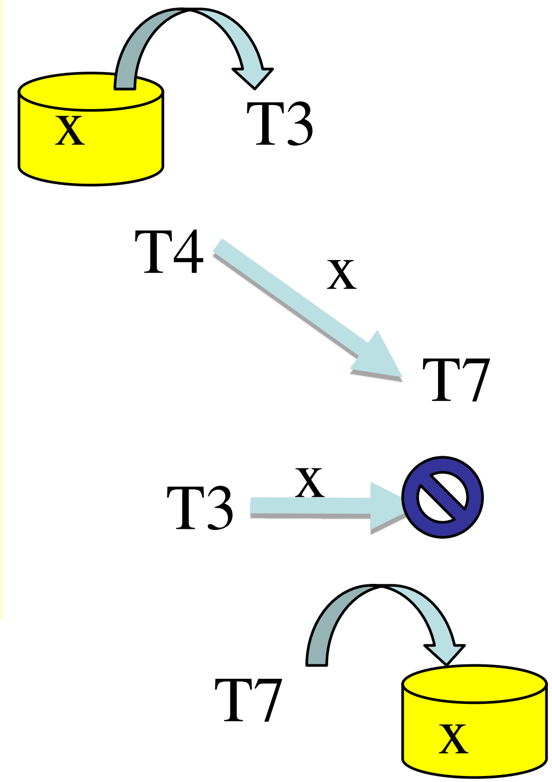
- $r_3[x]\ w_4[x]\ w_3[x]$
- T_3 read-from T_b.
- The final write for *x* is $w_3[x]$.
- Not serializable.
- r3[x] w3[x] w4[x]
- w4[x] r3[x] w3[x]

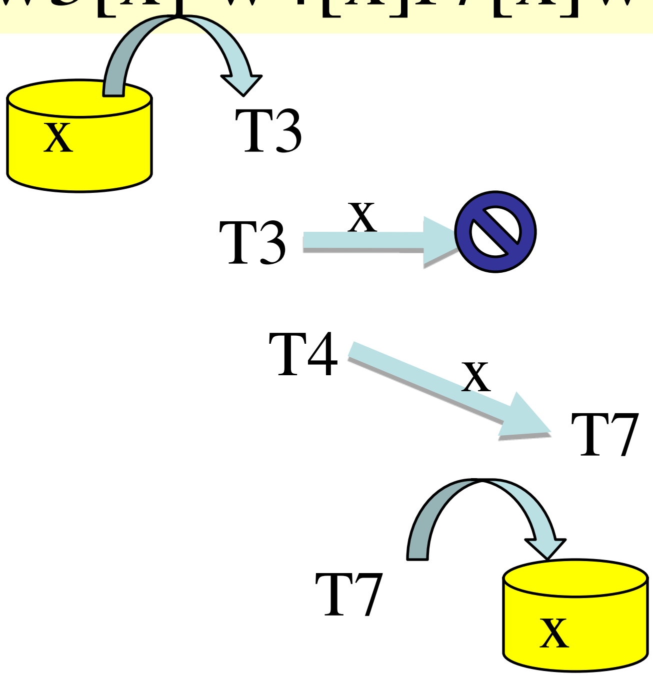






- $r_3[x]\ w_4[x]\ r_7[x]\ w_3[x]\ w_7[x]$
- T_3 read-from T_b.
- T_7 read-from T_4.
- The final write for x is $w_7[x]$.
- View equivalent to $T_3\ T_4\ T_7$.
- $r3[x]\ w3[x]\ w4[x] r7[x] w7[x]$



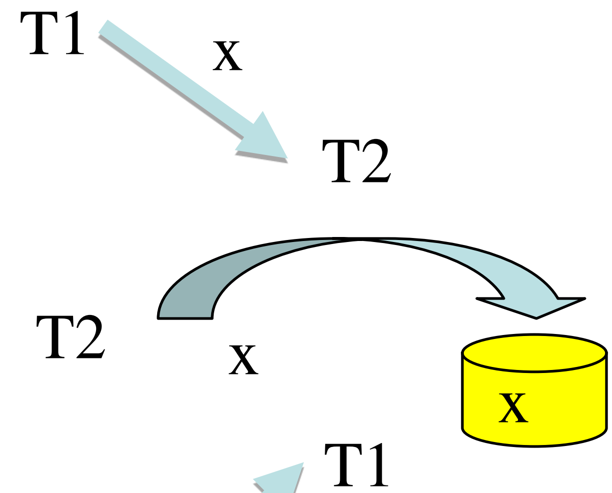


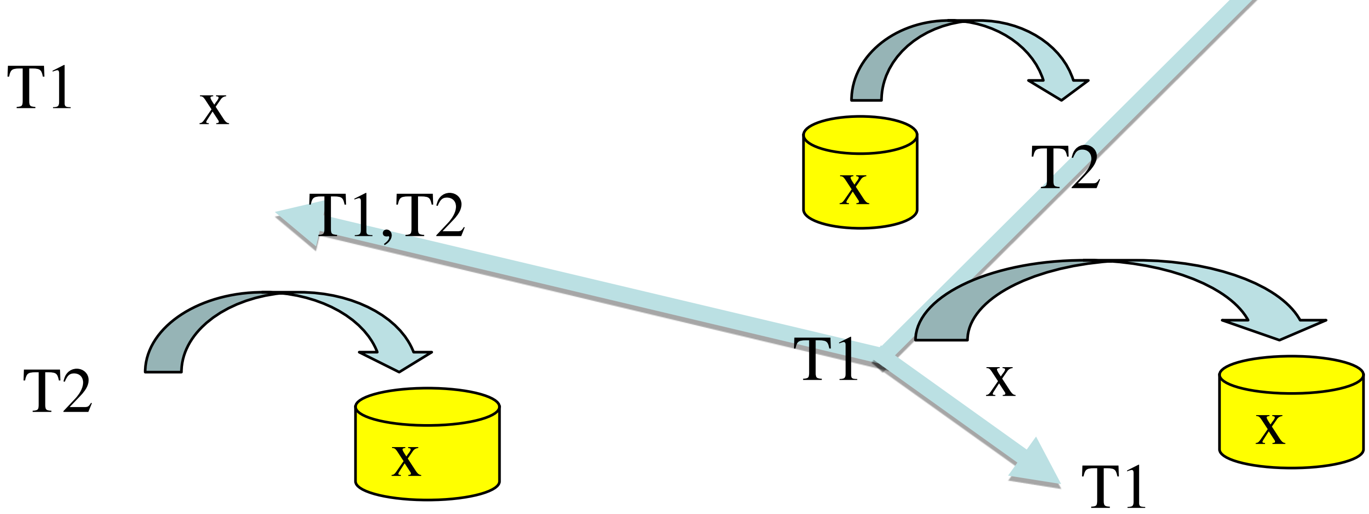
$w_1[x]\ r_2[x]\ w_2[x]\ r_1[x]$

- T_2 read-from T_1.
- T_1 read-from T_2
- The final write for *x* is $w_2[x]$.
- Not serializable.

w1[x] r1[x] r2[x] w2[x]

r2[x] w2[x] w1[x] r1[x]

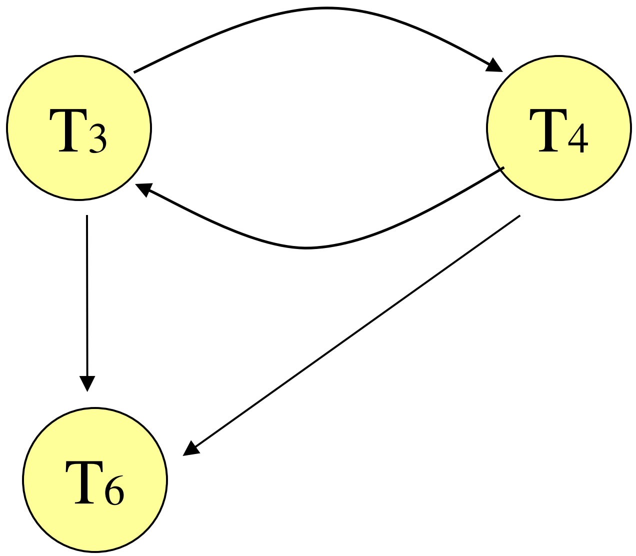




Testing whether a schedule is view serializable is NP-complete

Enforcing is expensive. So, not used.

- $r_3[x]\ w_4[x]\ w_3[x]\ w_6[x]$



- Not conflict serializable, as there is a cycle in the precedence graph.
- But view serializable, equivalent to T3, T4, T6
- $r_3[x]\ w_3[x]\ w_4[x]\ w_6[x]$

Polygraphs and the test for view serializability

- Section 10.2.2 and 10.2.3 are not included.

Outline

- Transactions that read uncommitted data
- View serializability
- **Resolving deadlocks**
- Distributed databases
- Distributed Commit
- Distributed locking
- Long duration transactions

Resolving Deadlocks

- Due to lock upgradation, 2PL
  - Deadlock detection
  - Deadlock prevention

Deadlock detection by timeout

- If transaction waits more than L sec., roll it back!
- Simple scheme
- Hard to select L

Deadlock Detection by Waits-for Graph

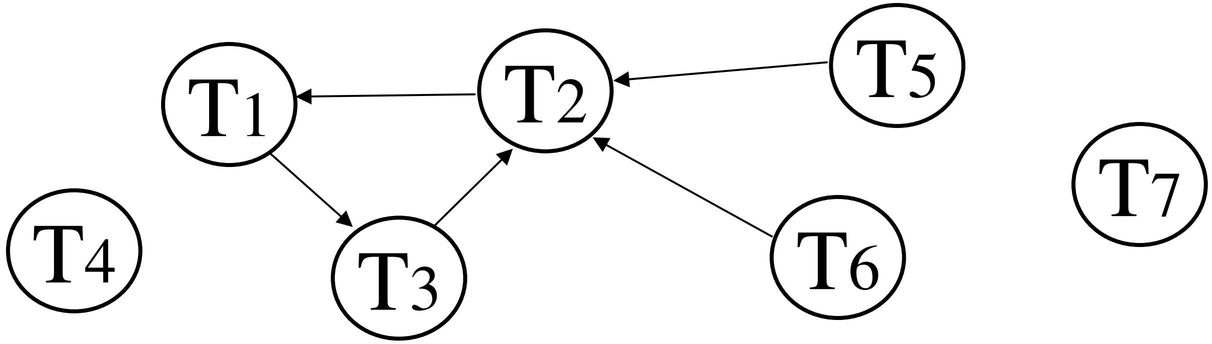
- Build Wait-For graph
  - Which transactions are waiting for other transactions.
- Use lock table structures
- Build incrementally or periodically
- When cycle found, rollback victim

Deadlocks

- Detection
  - Wait-for graph
- Prevention
  - Resource ordering
  - Timeout
  - Wait-die
  - Wound-wait

Deadlock Detection

- Build Wait-For graph
- Use lock table structures
- Build incrementally or periodically
- When cycle found, rollback victim



Example waits-for graph

Deadlock prevention Resource Ordering

- Order all elements $A_1, A_2, \ldots, A_n$
- A transaction T can lock A_i after A_j only if $i > j$

Detecting deadlocks by timestamps

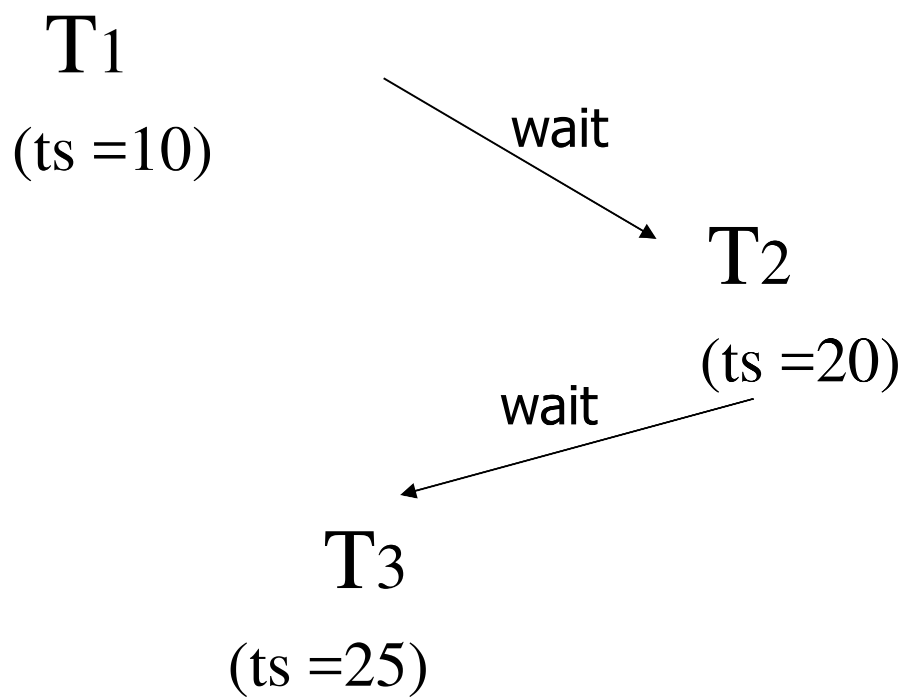
- Two schemes
- Wait-die
  - Transactions given a timestamp when they arrive …. $ts(T_i)$
  - T_i can only wait for T_j if $ts(T_i) < ts(T_j)$
    ...else die
    - That is If Ti is older it waits otherwise dies; it is rolled back.
- Wound-wait
  - Transactions given a timestamp when they arrive … $ts(T_i)$
  - T_i wounds T_j if $ts(T_i) < ts(T_j)$

    else T_i waits

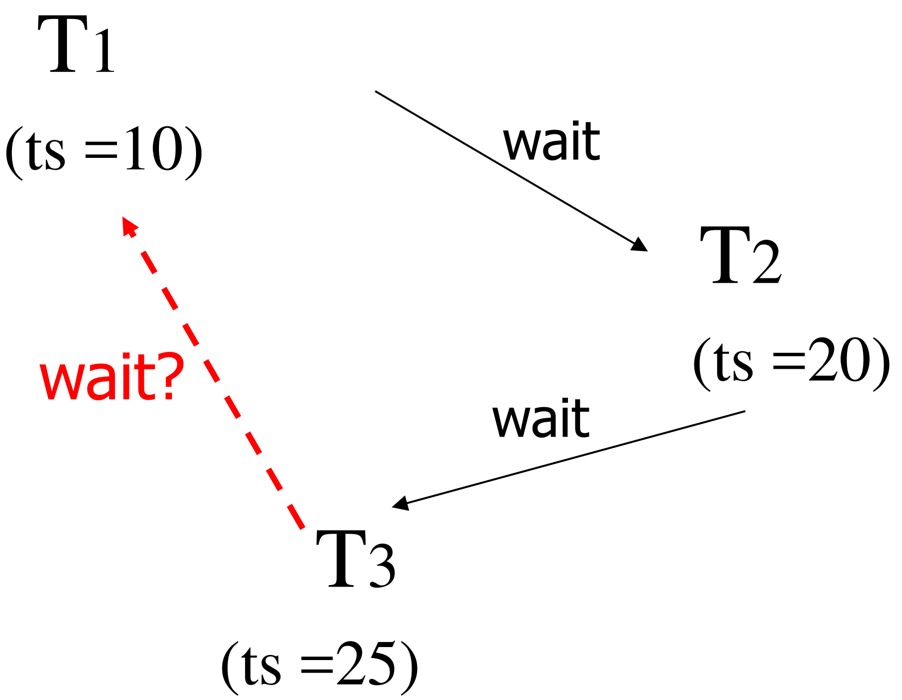
  “Wound”: T_j rolls back and gives lock to T_i

  (If Ti is older than U, it wounds U, otherwise Ti waits.)

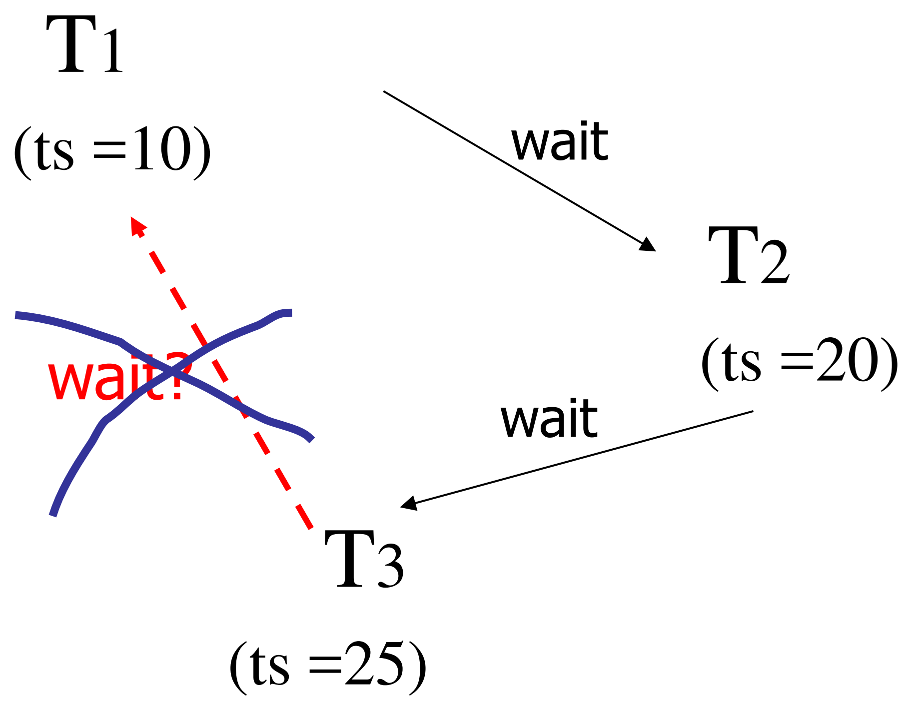
Example: Wait-die



Example:



Example:



Starvation with Wait-Die

- When transaction dies, re-try later with what timestamp?
  - original timestamp
  - new timestamp (time of re-submit)

Starvation with Wait-Die

- Resubmit with original timestamp
- Guarantees no starvation
  - Transaction with oldest ts never dies
  - A transaction that dies will eventually have oldest ts and will complete...

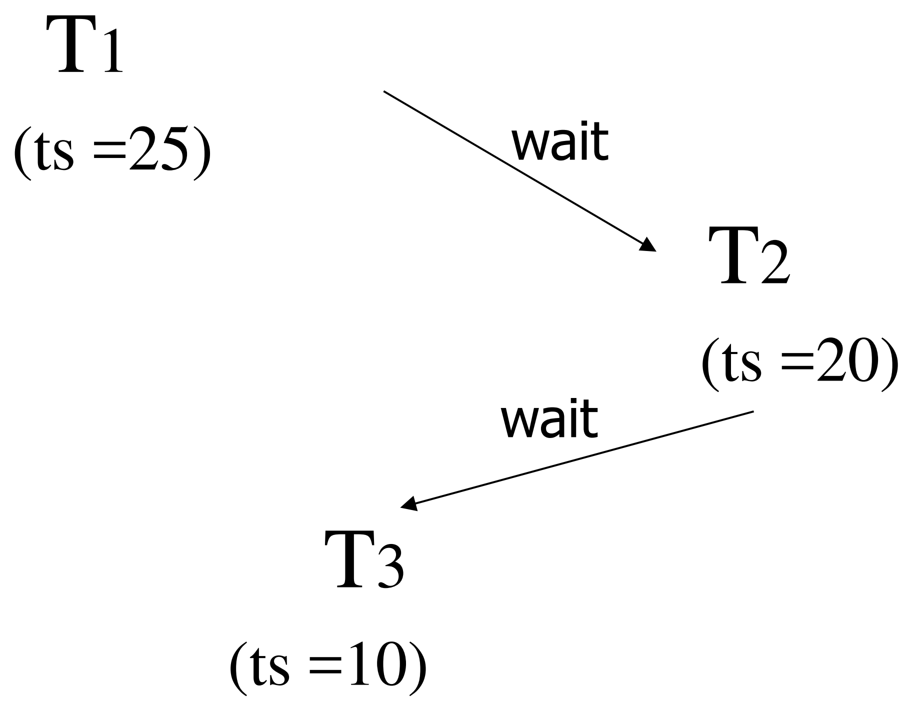
Wound-wait

- Transactions given a timestamp when they arrive … $ts(T_i)$
- T_i wounds T_j if $ts(T_i) < ts(T_j)$

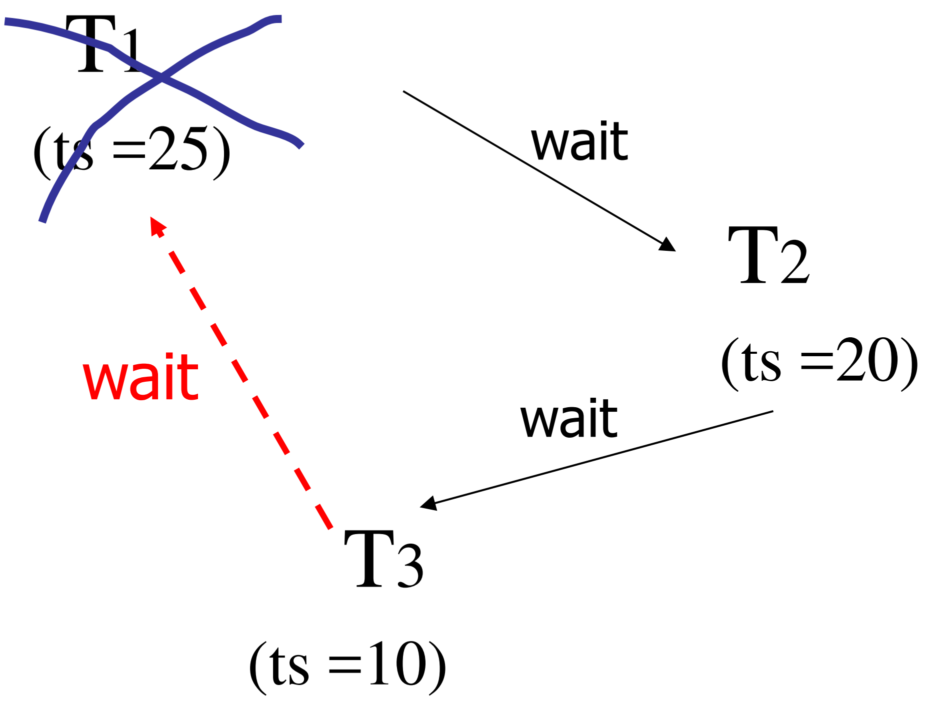
  else T_i waits

"Wound": T_j rolls back and gives lock to T_i

Example:



Example:



Starvation with Wound-Wait

- When transaction dies, re-try later with what timestamp?
  - original timestamp
  - new timestamp (time of re-submit)

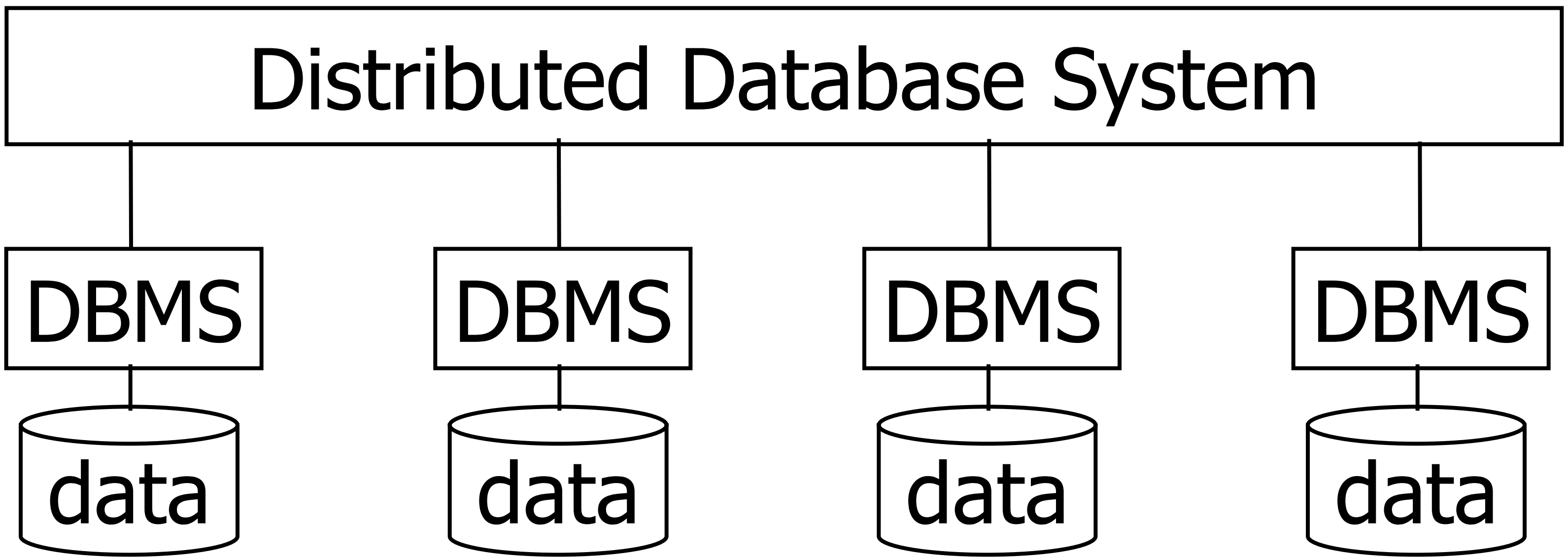
Comparison of Deadlock-Management Methods

- In wait-die and wound-wait, older transactions kill newer transactions.
- Since transactions restart with old timestamp, there is no starvation..
  - However, if several transactions arrive at the same starvation problem may occur.
- But
  - Timestamped schemes are easier to implement
    - Aborts occur even there is no deadlock.
  - Waits-for graph methods minimizes aborts.

Outline

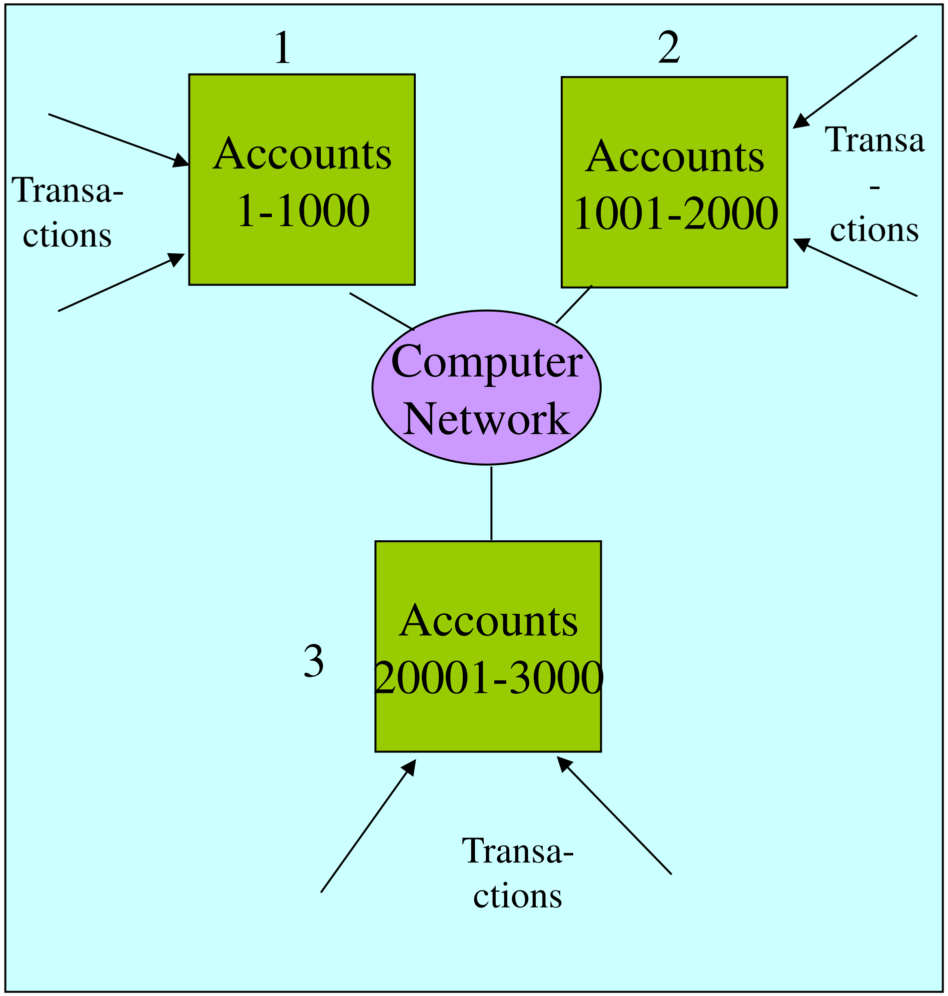
- Transactions that read uncommitted data
- View serializability
- Resolving deadlocks
- **Distributed databases**
- Distributed Commit
- Distributed locking
- Long duration transactions

Distributed Databases



Distributed Database Systems (DDBSs)

- In DDBSs, database is stored at number of sites, separated geographically by a Computer Network.
- A distributed transaction may access data at more than one site.
- For concurrency control, distributed two-phase locking may be employed.
- After execution, two-phase commit protocol is followed for commit processing.



Advantages of a DDBS

- Modularity
- Fault Tolerance
- High Performance
- Data Sharing
- Low Cost Components

Issues

- Data Distribution
- Exploiting Parallelism
- Concurrency and Recovery
- Heterogeneity

DDBS

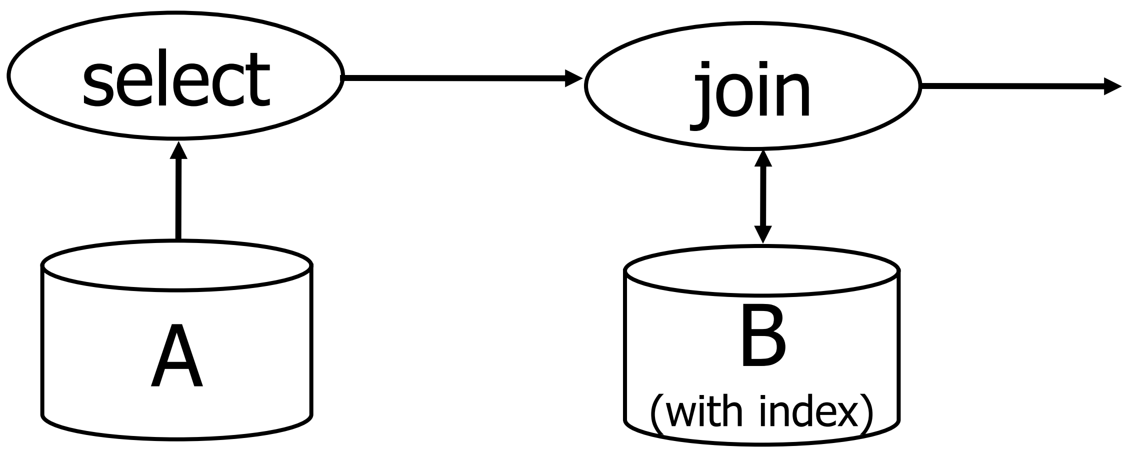
- Distributed processing increases complexity on every aspect of a database system
- We have to redesign most of the DBMS components.
- Cost of communication dominates the cost of processing

Distribution of Data

- Sales (item, date, price, purchaser)
- The relation does not exists physically
  - It is a union of number of smaler relations called fragments
  - Horizontal fragmentation
    - Tuples
  - Vertical fragmentation.
    - Subsets of attributes

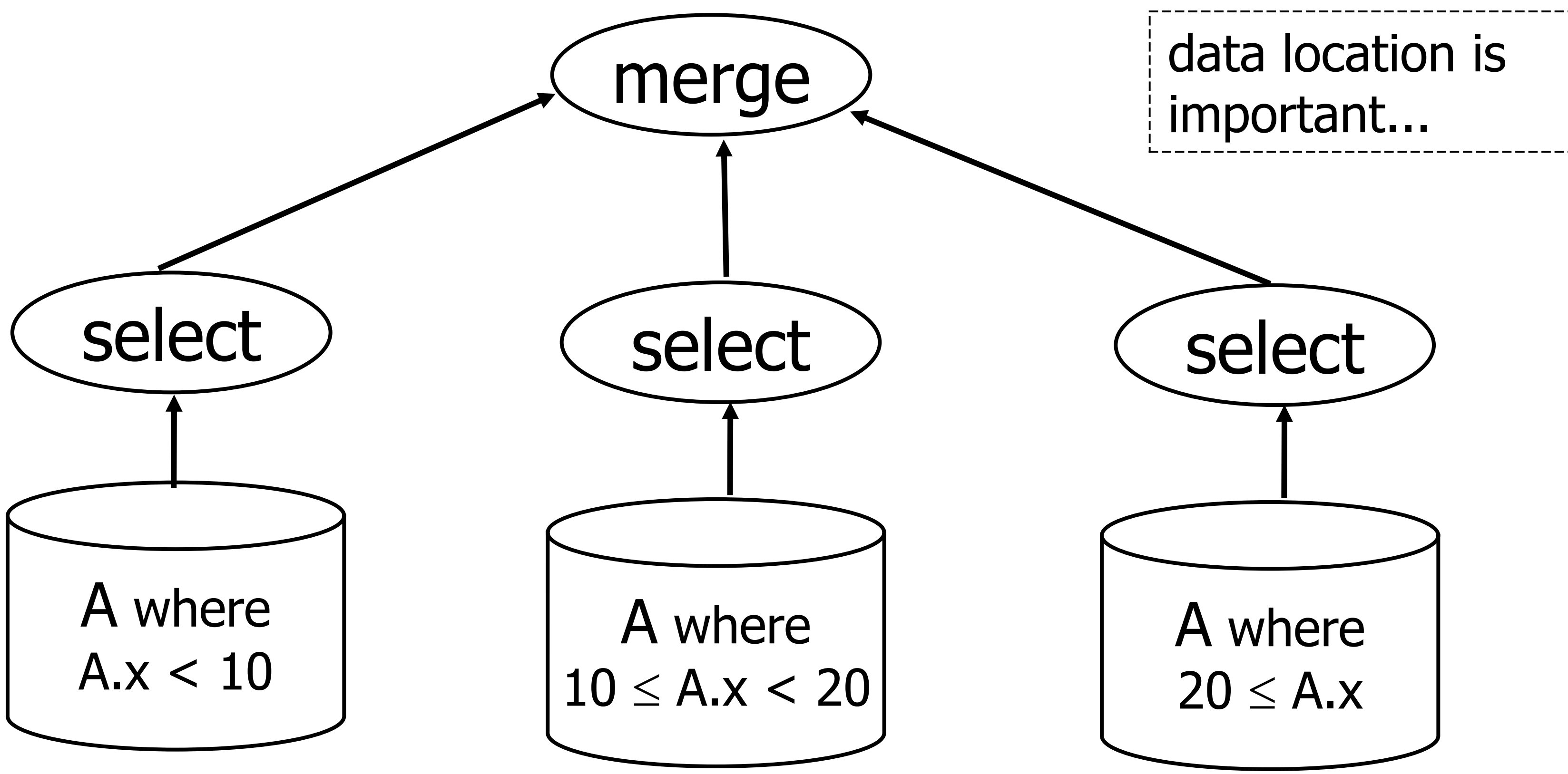
Parallelism: Pipelining

- Example:
  - $T_1 \leftarrow$ SELECT *
    FROM A WHERE cond
  - $T_2 \leftarrow$ JOIN T_1 and B



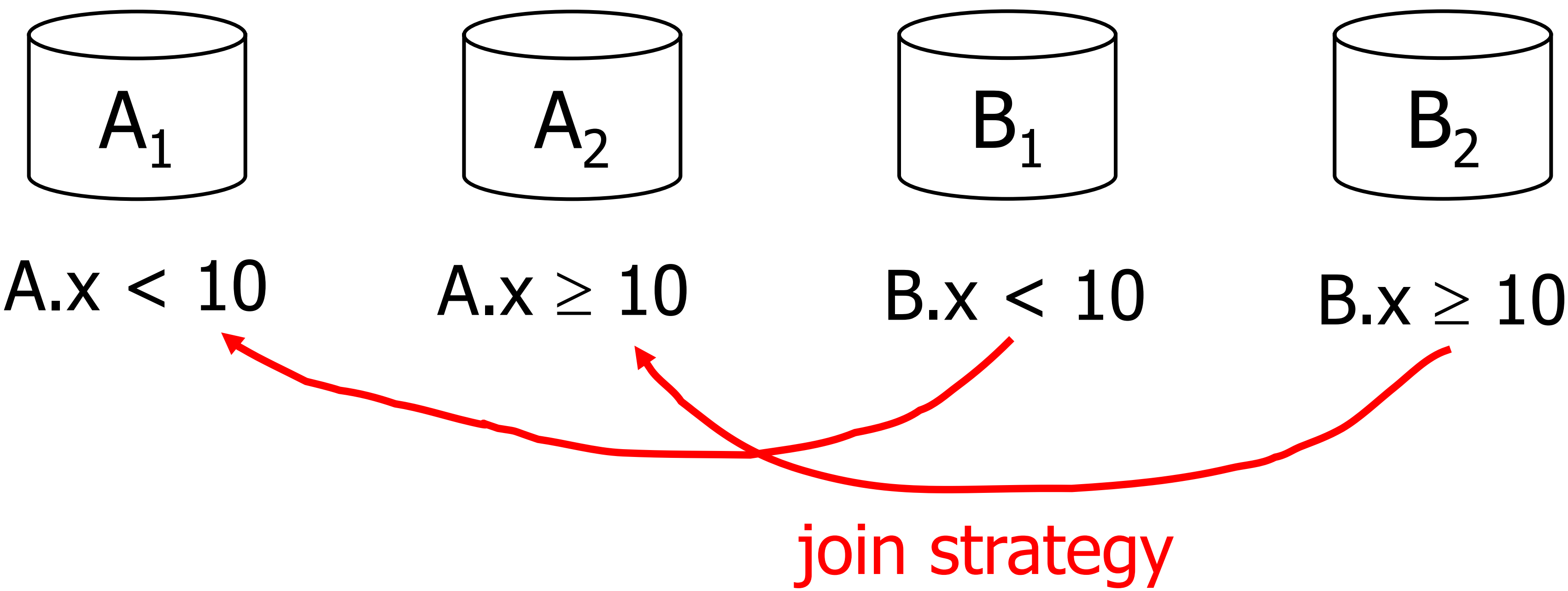
Parallelism: Concurrent Operations

- Example: SELECT * FROM A WHERE cond



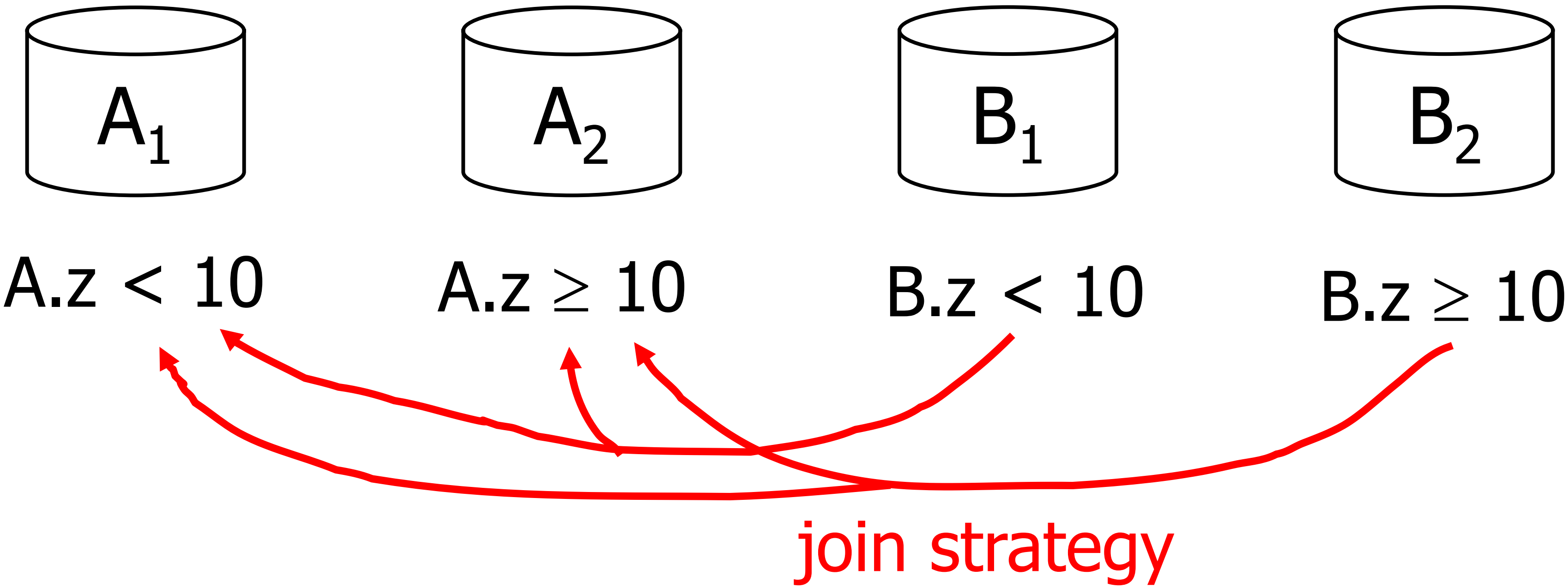
Join Processing

- Example: JOIN A, B over attribute X



Join Processing

- Example: JOIN A, B over attribute X



Distributed Transactions

- Transaction may involve processes at multiple sites.
  - How to manage commit/abort decision
  - How to ensure serializability ?
  - Maintaining lock tables
    - Local locks support global locks

Data Replication

- Advantage of distributed system
  - Data replication
  - If a site fails, other sites can take care
- Issues
  - How to keep the copies identical ?
    - Update is a distributed transaction
  - How do we decide how many copies to keep ?
  - What happens when a communication failure occurs or partitioning occurs.

Distributed Query Optimization

- Physical plan generation
  - Out of several copies of a relation which to use ?
  - For a join of R and S
    - Ship S to R’s site
    - Ship R to S’s site
    - Move R and S to third site

Outline

- Transactions that read uncommitted data
- View serializability
- Resolving deadlocks
- Distributed databases
- **Distributed Commit**
- Distributed locking
- Long duration transactions

Distributed Commit

- Even though logging is provided at every site, things can go wrong.
- Simply sending the commit will not work.
  - Local sites may not have received.
  - How to ensure that all the sites will take the same decision ?
- Two-phase commit is the solution

2-phase commit

- It is a distributed commit protocol to ensure atomicity and recovery
- There is no global log, each site logs actions at that site.
- Coordinator site takes the decision to commit or abort.

Two-Phase Commit: Phase 1

- Coordinator
  - The cooridator places a log record <Prepare T> on the log at the site.
  - The coordinator sends <prepare T>
- Site
  - Each site which receives <prepare T> decides whether to commit or abort. The site can delay if it is not completed.
  - If the site wants to commit its component, write a log record <ready T> in the local log and send the message <ready T>. Or the site can send <Donot Commit T> to the coordinator.

Two-phase Commit-Phase II

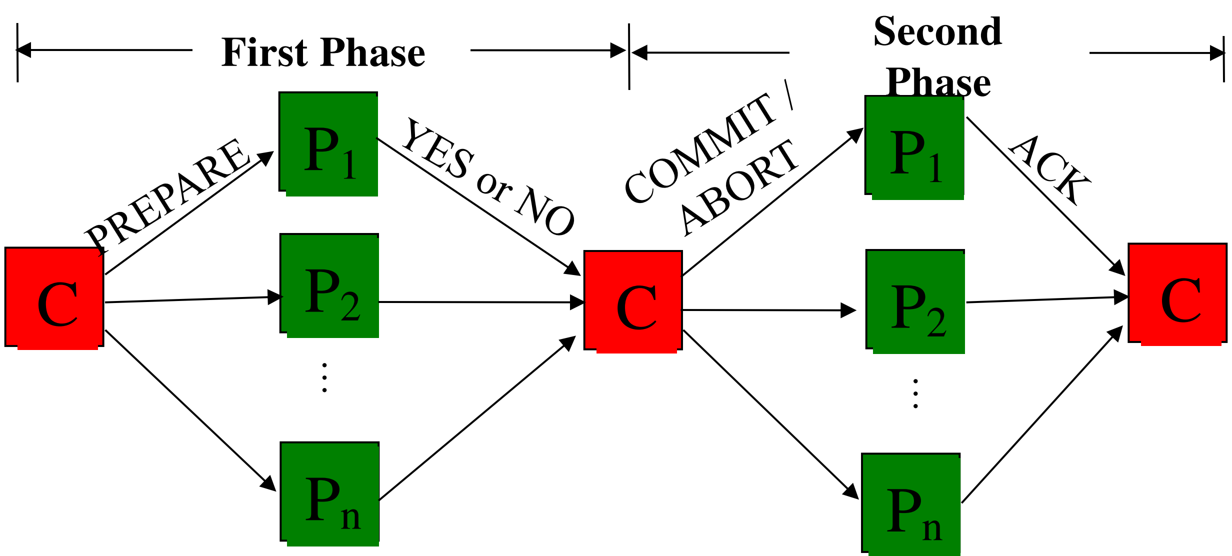
- Coordinator
  - If the coordinator receives ready T messages from all the components of T
    - Log <commit T> at its site.
    - Send the commit T to all sites involved in T.
  - If the coordinator has received don't commit T messages from one or more sites
    - Log <Abort T> at its site, and
    - Sends abort T messages to all sites involved in T.
- Site
  - If the site receives commit T message, it commits the components of T at this site, logging <Commit T>.
  - If the site receives abort T, it aborts T and writes the log record <Abort T>.

Two-Phase Commit Protocol

- 2PC ensures that all the sites involved in a distributed transaction reach a consistent decision either to **accept** or **reject** the transaction.
- One site (arrival site of a transaction) acts as a coordinator and the remaining sites act as participants.

- **First Phase**
  - **Coordinator** : Sends **PREPARE** messages to all the participant sites
  - **Participant:** If it is willing to commit, it sends **YES** message; Otherwise it sends **NO** message to the coordinator.

- **Second Phase**
- **Coordinator:** If it receives **YES** messages from all, it sends **GLOBAL_COMMIT** messages to all participants. Otherwise, even it receives **NO** message from one or time_out, it sends **GLOBAL_ABORT** message to all the participants.
- **Participant:** If it receives **GLOBAL_COMMIT**, it commits the transaction. Otherwise, if it receives **GLOBAL_ABORT**, it aborts the transaction.



Transaction Processing in DDBSs

- 2PL is widely employed for concurrency control.
- In case of 2PL, a transaction obtains locks by sending lock requests messages to object sites during execution and releases them only after completion of commit processing.
- The processing of a transaction T_i is depicted as follows.

s_i : Start of the execution

e_i: Completion of execution

c_i: Commit

a_i: Abort

Recovery of Distributed Transactions..

- Case 1: If the site has <Commit T> record, T must have been committed by the coordinator (REDO).
- Case 2: If the site has <Abort T> record, T must have been aborted by the coordinator (UNDO).
- If the last log record is <Don't Commit>, T is aborted (UNDO).

Recovery of Distributed Transactions

- Site Fails: If the log record for T is <ready T>, the site does not know whether the coordinator has committed or aborted.
  - If the site is up, the site can know from the coordinator.
  - If the coordinator is not-up, talk to other site to know the status of T.
  - Otherwise, wait for the recovery of the coordinator site. (Blocking problem.)

Recovery of Distributed Transactions

- Coordinator site fails.
  - The sites have to wait for the recovery of the coordinator, or elect a new coordinator.
- Leader election is another problem.
  - Send the messages and receive the conformation and announce the decision.
- New leader polls the sites for information about each distributed transaction T.
- For the difficult situation, DBA intervenes. DBA notifies the problem to blocking transactions to take some compensating action.

Outline

- Transactions that read uncommitted data
- View serializability
- Resolving deadlocks
- Distributed databases
- Distributed Commit
- **Distributed locking**
- Long duration transactions

Distributed Locking

- Central locking
  - One site maintains lock table for logical elements.
  - Problems:
    - Lock table may become bottleneck.
    - If the lock table crashes, the entire operation is stalled.

Cost model for Distributed Locking Algorithms

- A message to request the lock.
- A reply message granting the lock
- A message to the site of X releasing the lock.

Locking Replicated Elements

- If the element has replicas at several places, locking is different.
- Primary copy locking
  - Each logical element X has one of its copies designated the "primary copy".
- Global locks
  - Read-locks one, Write locks all
  - Majority locking

Outline

- Transactions that read uncommitted data
- View serializability
- Resolving deadlocks
- Distributed databases
- Distributed Commit
- Distributed locking
- **Long duration transactions**

Long-duration Transactions

- It takes too long period
  - Minutes to hours
- Design systems
  - Design of automobile, microprocessor or a software system.
- Workflow systems
  - Collection of processes some are executed by software alone and some involve human intervention.

Sagas Model

- A collection of actions
- A graph whose nodes are actions or special abort and complete nodes. Arcs link pair of nodes. No arch links two special nodes called terminal nodes.
- It has a start node.
- Each action may be a short transaction
- Transaction can be any of the path from start node to terminal node.

Compensating Transactions

- For each action A there will by A^{-1}, if we execute both the resulting state is the same.
- If saga execution leads to abort node, then rollback the saga by executing the compensating transactions for each action.