Unit IV-Transactions

Terms

- Transaction Concept
- Transaction State
- Serializability
- Recoverability
- Testing for Serializability.

Transaction Concept

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be inconsistent.
- When the transaction is committed, the database must be consistent.
- Two main issues to deal with:
  - ! Failures of various kinds, such as hardware failures and system crashes
  - ! Concurrent execution of multiple transactions

ACID Properties

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are. !
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - ! That is, for every pair of transactions Ti and Tj, it appears to Ti that either Tj, finished execution before Ti started, or Tj started execution after Ti finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example of Fund Transfer

- Transaction to transfer $50 from account A to account B:

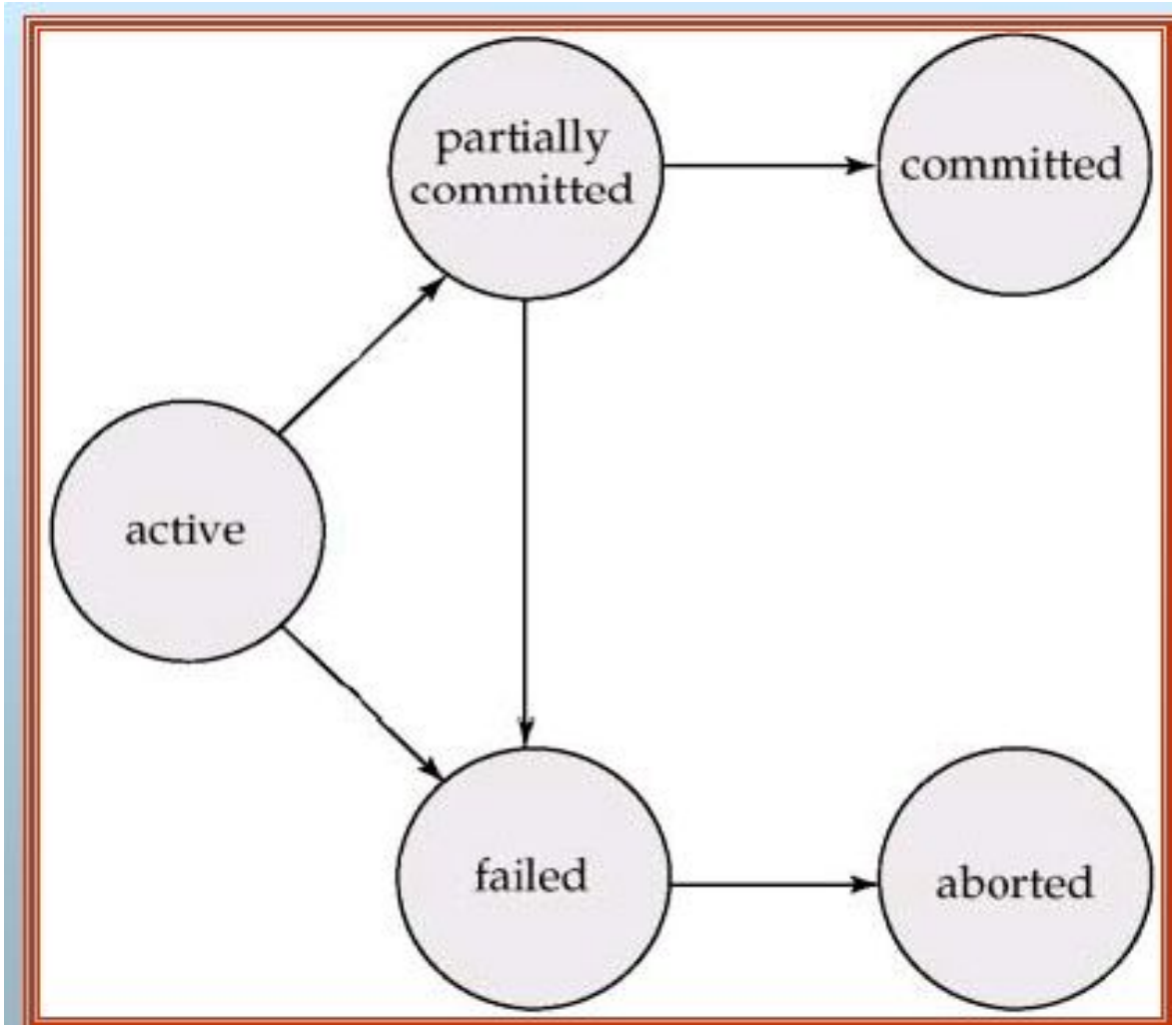
1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

- **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.
- **Atomicity requirement** — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

Transaction State

- **Active,** the initial state; the transaction stays in this state while it is executing
- Partially committed, after the final statement has been executed.
- **Failed,** after the discovery that normal execution can no longer proceed.
- **Aborted,** after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction – only if no internal logical error
  - kill the transaction
- **Committed,** after successful completion.

Transaction State



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - increased processor and disk utilization, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk
  - reduced average response time for transactions: short transactions need not wait behind long ones.
- Concurrency control schemes – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

Schedules

- Schedules – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

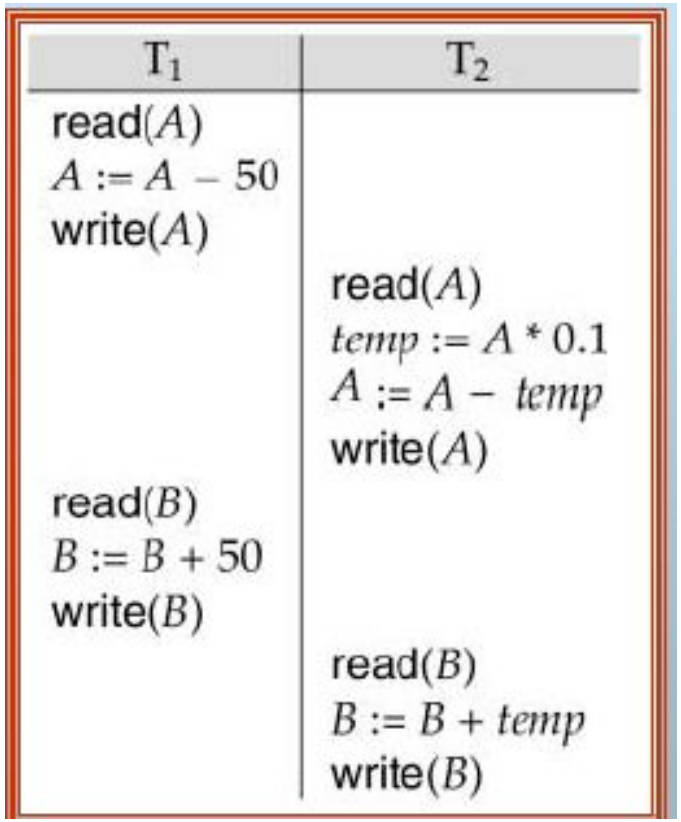
Example Schedules

- Let T1 transfer $50 from A to B, and T2 transfer 10% of the balance from A to B. The following is a serial schedule, in which T1 is followed by T2.

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

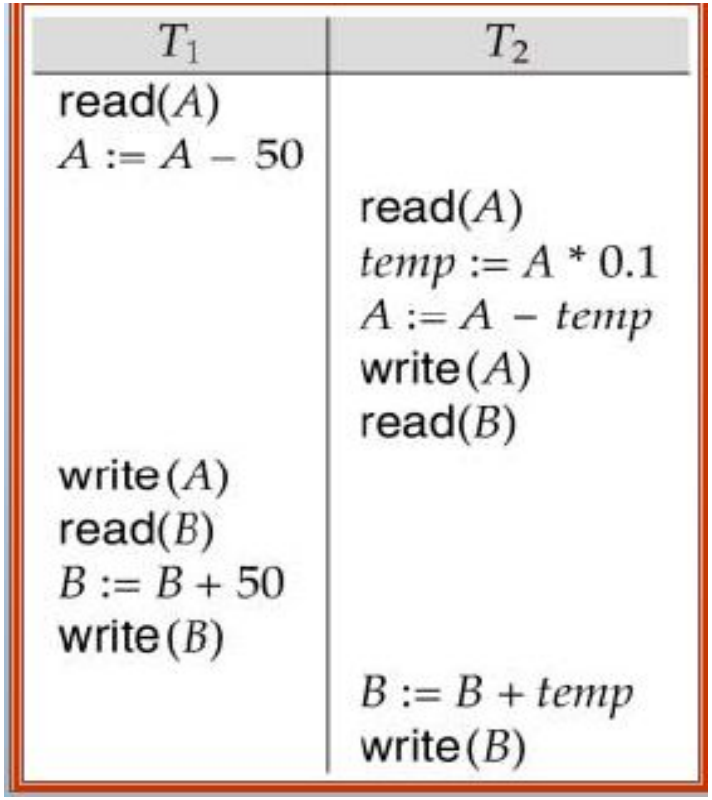
Continue….

- Let T1 and T2 be the transactions defined previously. The following schedule is not a serial schedule, but it is equivalent to Schedule 1.



T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

The following concurrent schedule (Schedule 4 in the text) does not preserve the value of the sum A + B



T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serializability

- Basic Assumption – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
- Different forms of schedule equivalence give rise to the notions of:
  - 1. conflict serializability
  - 2. view serializability

Conflict Serializability

- Instructions li and lj of transactions Ti and Tj respectively, conflict if and only if there exists some item Q accessed by both li and lj, and at least one of these instructions wrote Q.
- 1. li = read(Q), lj = read(Q). li and lj don't conflict.
- 2. li = read(Q), lj = write(Q). They conflict.
- 3. li = write(Q), lj = read(Q). They conflict.
- 4. li = write(Q), lj = write(Q). They conflict.

- Intuitively, a conflict between li and lj forces a (logical) temporal order between them.
- If li and lj are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

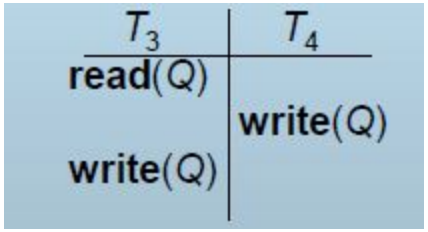
Conflict Serializability (Cont.)

Schedule 3 below can be transformed into Schedule 1, a serial schedule where *T2 follows T1, by series of swaps of* non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Conflict Serializability (Cont.)

- If a schedule S can be transformed into a schedule S´ by a series of swaps of non-conflicting instructions, we say that S and S´ are conflict equivalent.
- We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule ! Example of a schedule that is not conflict serializable:



We are unable to swap instructions in the above schedule to obtain either the serial schedule < *T3, T4* >, *or the serial* schedule < *T4, T3* >.

View Serializability

- Let *S and S′ be two schedules with the same set of* transactions. *S and S′ are* ***view equivalent if the following*** three conditions are met:

1. For each data item *Q, if transaction Ti reads the initial value of Q in* schedule *S, then transaction Ti must, in schedule S′, also read the* initial value of *Q*.

2. For each data item Q if transaction Ti executes ***read(Q) in schedule*** *S, and that value was produced by transaction Tj (if any), then* transaction *Ti must in schedule S′ also read the value of Q that* was produced by transaction *Tj* .

3. For each data item *Q, the transaction (if any) that performs the final* **write(*Q*) *operation in schedule S must perform the final write(Q)*** operation in schedule *S′*.

- As can be seen, view equivalence is also based purely on **reads** and **writes alone.**

View Serializability (Cont.)

- A schedule *S is* ***view serializable it is view equivalent to a serial*** schedule.
- ! Every conflict serializable schedule is also view serializable.
- ! Schedule 9 (from text) — a schedule which is view-serializable but *not conflict serializable.*

! Every view serializable schedule that is not conflict serializable has **blind writes.**

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Recoverability

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule — if a transaction *Tj reads a data items*** previously written by a transaction *Ti , the commit operation of Ti* appears before the commit operation of *Tj.*
- !The following schedule (Schedule 11) is not recoverable if *T9* commits immediately after the read
- If *T8 should abort, T9 would have read (and possibly shown to the* user) an inconsistent database state. Hence database must ensure that schedules are recoverable.

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Recoverability (Cont.)

Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

If *T10 fails, T11 and T12 must also be rolled back.*

- Can lead to the undoing of a significant amount of work

Recoverability (Cont.)

- **Cascadeless schedules — cascading rollbacks cannot occur;** for each pair of transactions *Ti and Tj such that Tj reads a data* item previously written by *Ti, the commit operation of Ti appears* before the read operation of *Tj.*

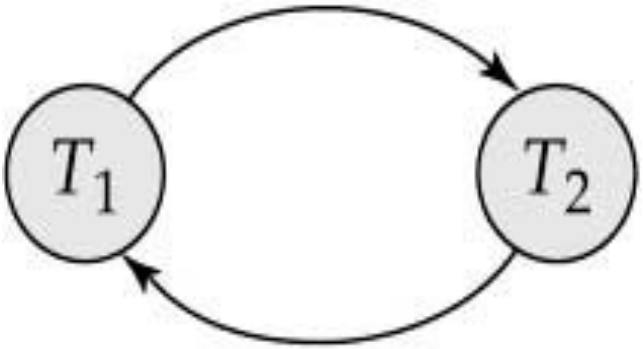
- Every cascadeless schedule is also recoverable.

- It is desirable to restrict the schedules to those that are cascadeless.

Testing of Conflict Serializability

Consider some schedule of a set of transactions *T1, T2,..., Tn*

- **Precedence graph — a direct graph where the** vertices are the transactions (names).
- We draw an arc from *Ti to Tj if the two transaction* conflict, and *Ti accessed the data item on which the* conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**



A schedule is conflict serializable if and only if its precedence graph is acyclic.

Precedence Graph for
(a) Schedule 1 and (b) Schedule 2

END

Concurrency control

Concurrency control

- Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another.
- **Concurrency Control Protocols**
- Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose.
- **Lock-Based Protocol**
- **Two Phase Locking Protocol**
- **Timestamp-Based Protocols**
- Validation-Based Protocol
- Graph-Based Protocols
- Tree-Based Protocols

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. ***exclusive (X) mode***. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. ***shared (S) mode***. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to **concurrency-control manager.**
- Transaction do not access data items before having acquired a lock on that data item.
- Transactions release their locks on a data item only after they have accessed a data item.

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.

- Any number of transactions can hold shared locks on an item, but if any transaction holds an **exclusive lock** on the item no other transaction may hold any lock on the item.

- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

Example of a transaction performing locking:

T2 : lock-S(A);
read (A);
unlock(A);
lock-S(B);
read (B);
unlock(B);
display(A+B)

- Locking as above is not sufficient to guarantee serializability — if A and B get updated in-between the read of A and B, the displayed sum would be wrong.
- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Neither *T3nor T4can make progress —executing* ***lock-S(B)causes T4to wait for T3to release its lock on B, while executing lock-X(A)causes T3to wait for T4to release its lock on A.***
- Such a situation is called a **deadlock.**
  - To handle a deadlock one of *T3or T4must be rolled back* and its locks released.

Pitfalls of Lock-Based Protocols (Cont.)

- **Starvation is also possible if concurrency control manager is badly designed. For example:**
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

Example

- Suppose a transaction ***T2 has a shared-mode*** *lock on a data item, and another transaction* ***T1*** **requests an exclusive-mode** lock on the data item. Clearly, ***T1 has to wait for T2*** *to release* the shared-mode lock. Meanwhile, a transaction ***T3 may request a shared-mode*** lock on the same data item.

- The lock request is compatible with the lock granted to *T2, so* ***T3 may be granted the shared-mode lock****. At this point T2 may release the lock,* but still ***T1 has to wait for T3 to finish****.*

- *But again, there may be a new transaction* ***T4 that requests a shared-mode*** *lock on the same data item, and is granted the lock* before *T3 releases it.*

- *In fact, it is possible that there is a* ***sequence of transactions*** *that* each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but ***T1 never gets the exclusive-mode lock on the*** **data item.** The transaction *T1 may never make progress, and is said to be* ***starved.***

- **We can avoid starvation of transactions by granting locks in the following manner:**
- When a transaction *Ti requests a lock on a data item Q in a particular mode M, the* concurrency-control manager grants the lock provided that
  - **1. There is no other other transaction holding a lock on *Q in a mode that conflicts*** with *M.*
  - **2. There is no other transaction that is waiting for a lock on *Q, and that made its*** lock request before *Ti.*

- Thus, a lock request will never get blocked by a lock request that is made later.

The Two Phase Locking Protocol

- This protocol requires that each transaction issue lock and unlock requests in two phases:
- **1. Growing phase.** A transaction may obtain locks, but may not release any lock.
- **2. Shrinking phase.** A transaction may release locks, but may not obtain any new locks.

- The protocol assures serializability.
- Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.
- **Cascading rollback** may occur under two-phase locking.

- Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol. This protocol requires not only that locking** be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits.

- This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.

- Another variant of two-phase locking is the **rigorous two-phase locking protocol,** which requires that all locks be held until the transaction commits.

- We can easily verify that, with **rigorous two-phase locking, transactions can be serialized in the order in which they commit**. Most database systems implement either strict or rigorous two-phase locking.

- Consider the following two transactions, for which we have shown only some of the significant read and write operations:

  - *T8: read(a1);*
  - read(*a2);*
  - . . .
  - read(*an);*
  - write(*a1).*

  - *T9: read(a1);*
  - read(*a2);*
  - display(*a1 + a2).*

- If we employ the two-phase locking protocol**, then *T8 must lock a1 in exclusive* mode.** Therefore, any concurrent execution of both transactions amounts to a serial execution.

- However, that *T8 needs an exclusive lock on a1 only at the end of* its execution, when it writes *a1*.

- *Thus, if T8 could initially lock a1 in shared mode, and* then could later change the lock to exclusive mode, we could get more concurrency, since *T8 and T9 could access a1 and a2 simultaneously.*

- This observation leads us to a refinement of the basic two-phase locking protocol, in which **lock conversions are allowed.**

Lock conversions

- **Two-phase locking with lock conversions:**

- **–First Phase:**
  - can acquire a lock-S on item
  - can acquire a lock-X on item
  - can convert a lock-S to a lock-X (upgrade)

- **–Second Phase:**
  - can release a lock-S
  - can release a lock-X
  - can convert a lock-X to a lock-S (downgrade)

Timestamp Ordering Protocol

- With each transaction *Ti in the system, we associate a unique fixed timestamp, denoted* by TS(*Ti). This timestamp is assigned by the database system before the transaction Ti starts execution.*
- *If a transaction Ti has been assigned timestamp TS(Ti), and* a new transaction *Tj enters the system, then TS(Ti) < TS(Tj).*
- *There are two simple* methods for implementing this scheme:
- **1. Use the value of the *system clock* as the timestamp; that is, a *transaction's timestamp*** is equal to the value of the clock when the transaction enters the system.
- **2. Use a logical counter that is incremented after a new timestamp has been** assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

Timestamp Ordering Protocol

- Each transaction is issued a timestamp when it enters the system. If an old transaction Ti has time-stamp TS(Ti), a new transaction Tj is assigned time-stamp TS(Tj) such that TS(Ti) <TS(Tj).

- The protocol manages concurrent execution such that the time-stamps determine the serializability order.

- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp(Q)** is the largest time-stamp of any transaction that executed write(Q) successfully.
  - **R-timestamp(Q)** is the largest time-stamp of any transaction that executed read(Q) successfully.

- The timestamp ordering protocol ensures that any conflicting **readand writeoperations are executed in timestamp order.**
- Suppose a transaction Ti issues a **read(*Q*)**
  - 1.If TS(*Ti*) ≤***W-timestamp(Q), then Ti needs to read a value of Q that was already overwritten.***
    - Hence, the **read operation is rejected, and *Ti is rolled back.***
  - 2.If TS(*Ti*)≥***W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to max of R-timestamp(Q) and TS(Ti).***

- Suppose that transaction *Ti issues* ***write(Q).***
  - 1.If TS(*Ti*) < *R-timestamp(Q), then the value of Q that Ti is producing was needed previously, and the system assumed that the value would never be produced.*
    - Hence, the **write operation is rejected, and *Ti is rolled back.***
  - 2.If TS(*Ti*) < *W-timestamp(Q), then Ti is attempting to write an obsolete value of Q.*
    - Hence, this **write operation is rejected, and *Ti is rolled back.***
  - 3.Otherwise, the **write operation is executed, and W-timestamp(*Q*) *is set to TS(Ti).***

- If a transaction *Ti is rolled back by the concurrency-control scheme as result of issuance* of either a read or write operation, the system assigns it a new timestamp and restarts it.

- The timestamp-ordering protocol ensures conflict serializability.

- This is because conflicting operations are processed in timestamp order.

Thomas' Write Rule

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

- Since *T16 starts* before *T17, we shall assume that TS(T16) < TS(T17). The read(Q) operation of T16 succeeds,* as does the write(*Q) operation of T17.*

- *When T16 attempts its write(Q) operation,* we find that TS(*T16) < W-timestamp(Q), since W-timestamp(Q) = TS(T17).*
-
- *Thus, the* write(*Q) by T16 is rejected and transaction T16 must be rolled back.*

- Although the rollback of *T16 is required by the timestamp-ordering protocol, it* is unnecessary.

- Since *T17 has already written Q, the value that T16 is attempting to* write is one that will never need to be read.

- Any transaction *Ti with TS(Ti) < TS(T17)* that attempts a read(*Q) will be rolled back, since TS(Ti) < W-timestamp(Q).*

- Any transaction *Tj with TS(Tj) > TS(T17) must read the value of Q written by T17, rather* than the value written by *T16.*

- This observation leads to a modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances.

- The protocol rules for read operations remain unchanged. The protocol rules for write operations, however, are slightly different from the timestamp-ordering protocol.

- The modification to the timestamp-ordering protocol, called **Thomas' write rule,** is this: Suppose that transaction *Ti issues write(Q).*
- **1. If TS(*Ti) < R-timestamp(Q), then the value of Q that Ti is producing was previously*** needed, and it had been assumed that the value would never be produced.
  - Hence, the system rejects the write operation and rolls *Ti back.*

- **2. If TS(*Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value*** of *Q*. ***Hence, this write operation can be ignored.***

- **3. Otherwise, the system executes the write operation and sets W-timestamp(*Q)*** to TS(*Ti)*.

- The timestamp-ordering protocol requires that *Ti be rolled back if Ti issues* write(*Q) and TS(Ti) < W-timestamp(Q).*

- *However, here, in those cases where TS(Ti) ≥ R-timestamp(Q), we ignore the obsolete write.*

END

Recovery System

Storage Structure

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory, non-volatile RAM
- **Stable storage**:
  - A form of storage that survives all failures

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.

- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated.

- Protecting storage media from failure during data transfer:-
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition.
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock).

- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash

- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage.

Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures.

- **Recovery algorithms have two parts:-**
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures.
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability.

Log-Based Recovery

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.

- When transaction T_i starts, it registers itself by writing a

  $<T_i$ **start**$>$ log record

- *Before* T_i executes **write**(X), a log record $<T_i, X, V_1, V_2>$ is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.

  - Log record notes that T_i has performed a write on data item X_j. X_j had value V_1 before the write, and will have value V_2 after the write.

- When T_i finishes its last statement, the log record <T_i **commit**> is written.

- Two approaches using logs:-
  1) Deferred database modification
  2) Immediate database modification

(1)Deferred Database Modification

- All logs are written on to the stable storage and the database is updated when a transaction commits.
- The **deferred database modification** scheme records all modifications to the log, but defers all the **writes** to after partial commit.
- Assume that transactions execute serially.
- Transaction starts by writing $<T_i$ ***start***$>$ record to log.
- A **write**(X) operation results in a log record $<T_i, X, V>$ being written, where V is the new value for X
  - **Note: old value is not needed for this scheme**
- The write is not performed on X at this time, but is deferred.
- When T_i partially commits, $<T_i$ **commit**$>$ is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.

- During recovery after a crash, a transaction needs to be **redone** if and only if both <T_i **start**> and<T_i **commit**> are there in the log.
- Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.

- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken.

- example transactions T_0 and T_1 (T_0 executes before T_1):

T_0	T_1
read (A)	**read** (C)
A: - A - 50	C:-C- 100
Write (A)	**write** (C)
read (B)	
B:- B + 50	
write (B)	

- Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

- If log on stable storage at time of crash is as in case:
  (a) No redo actions need to be taken
  (b) redo(T_0) must be performed since <T_0 **commit**> is present
  (c) **redo**(T_0) must be performed followed by redo(T_1) since <T_0 **commit**> and <T_i commit> are present.

Immediate Database Modification

- Each log follows an actual database modification. That is, the database is modified immediately after every operation.

- The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value.

- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage.

- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

- Recovery procedure has two operations instead of one:
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i.

- Both operations must be **idempotent.**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once

- When recovering after failure:
  - Transaction T_i needs to be **undone** if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
  - Transaction T_i needs to be **redone** if the log contains both the record <T_i **start**> and the record <T_i **commit**>.
- Undo operations are performed first, then redo operations.

Immediate Database Modification Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	A = 950 B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B, B_C
<T_1 **commit**>		
		B_A

B_C output before T_1 commits

B_A output after T_0 commits

- Note: B_X denotes block containing X.

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then A and B are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600

Checkpoints

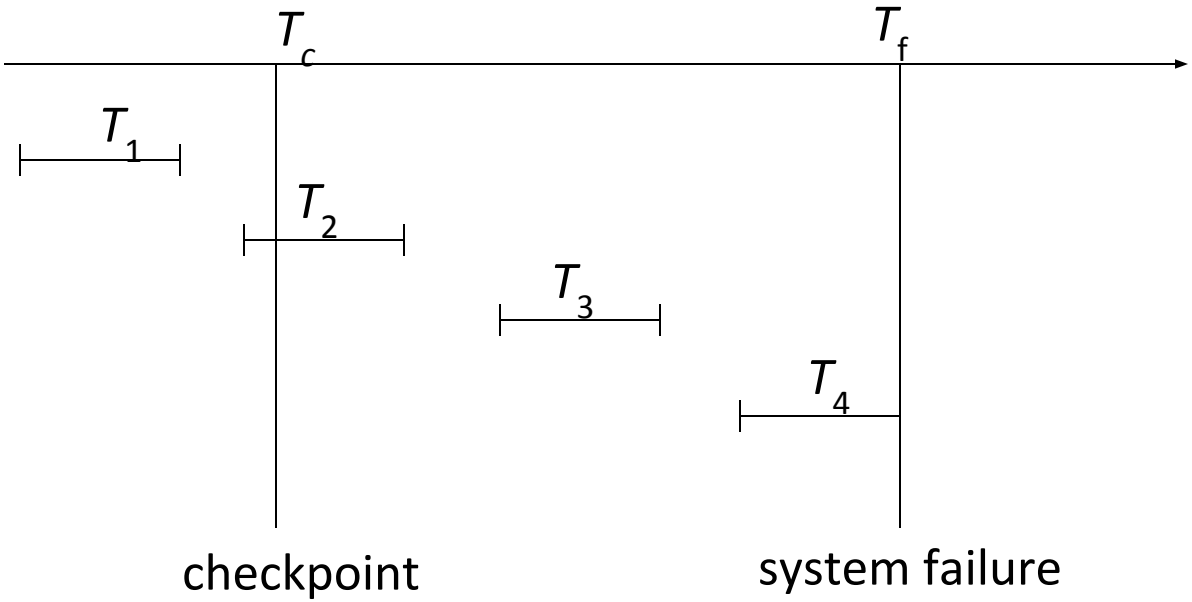
- **Checkpoint** is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. **Checkpoint** declares a point before which the **DBMS** was in consistent state, and all the transactions were committed.

- Problems in recovery procedure :
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already output their updates to the database.

- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint**> onto stable storage.

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  1. Scan backwards from end of log to find the most recent <**checkpoint**> record
  2. Continue scanning backwards till a record <T_i **start**> is found.
  3. Need only consider the part of log following above **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  4. For all transactions (starting from T_i or later) with no <T_i **commit**>, execute **undo(***T_i***)**. (Done only in case of immediate modification.)
  5. Scanning forward in the log, for all transactions starting from T_i or later with a <T_i **commit**>, execute **redo(***T_i***)**.

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an **undo-list and a redo-list.**
- If the recovery system sees a log with $<T_n$, Start> and $<T_n$, Commit> or just $<T_n$, Commit>, it puts the transaction in the **redo-list.**
- If the recovery system sees a log with $<T_n$, Start> but no commit or abort log found, it puts the transaction in **undo-list.**

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

- Checkpoints are performed as before, except that the checkpoint log record is now of the form
  < **checkpoint** *L*>
  where *L* is the list of transactions active at the time of the checkpoint
  - We assume no updates are in progress while the checkpoint is carried out.

- When the system recovers from a crash, it first does the following:
  1. Initialize *undo-list* and *redo-list* to empty
  2. Scan the log backwards from the end, stopping when the first <**checkpoint** *L*> record is found.
     For each record found during the backward scan:
     - if the record is <T_i **commit**>, add T_i to *redo-list*
     - if the record is <T_i **start**>, then if T_i is not in *redo-list*, add T_i to *undo-list*
  3. For every T_i in *L*, if T_i is not in *redo-list*, add T_i to *undo-list*

Example of Recovery

- Go over the steps of the recovery algorithm on the following log:

<T_0 **start**>
<T_0, A, 0, 10>
<T_0 **commit**>
<T_1 **start**> --stop backward for undo
<T_1, B, 0, 10>
<T_2 **start**>
<T_2, C, 0, 10>
<T_2, C, 10, 20>
<checkpoint {T_1, T_2}> -- start forward for redo
<T_3 **start**>
<T_3, A, 10, 20>
<T_3, D, 0, 10>
<T_3 **commit**>

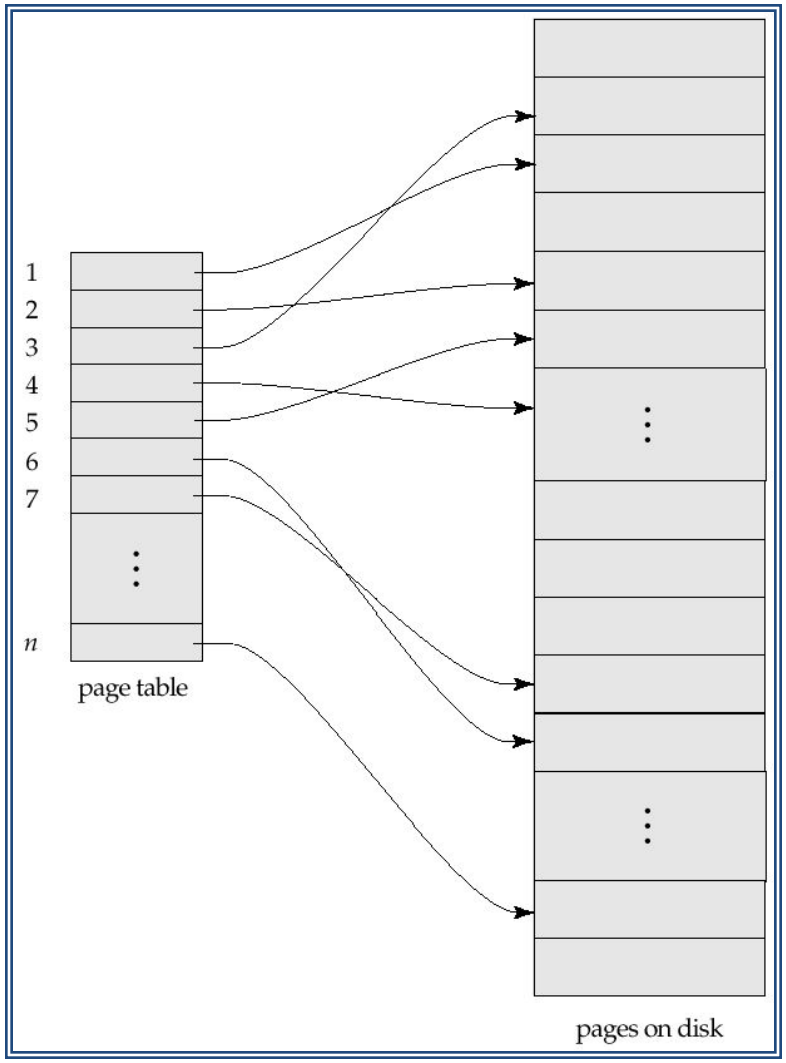
Summary-Checkpoint

- Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system.
- As time passes, the log file may grow too big to be handled at all.
- Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk.
- Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

Shadow Paging

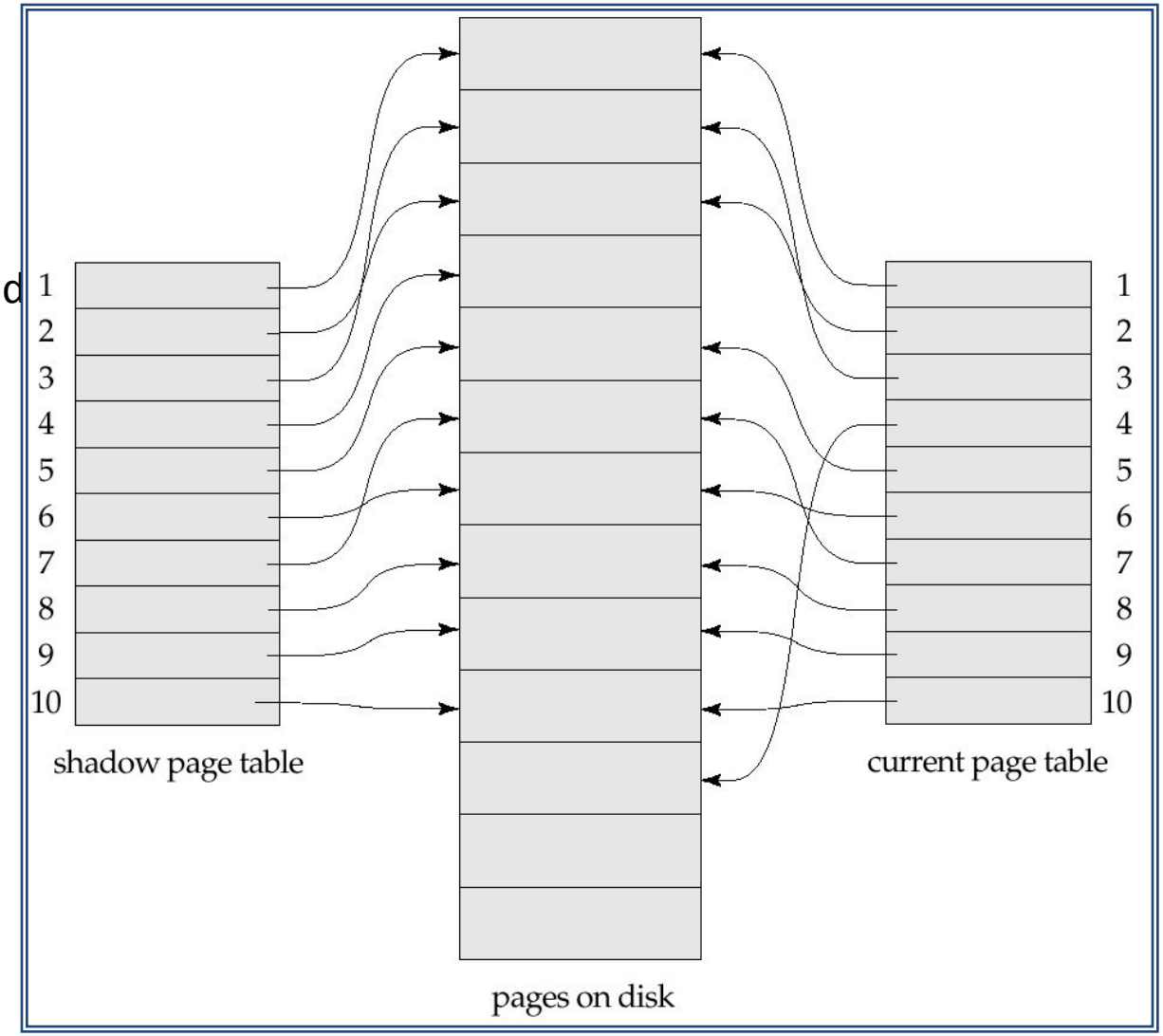
- **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially.
- Idea: maintain *two* page tables during the lifetime of a transaction –the **current page table**, and the **shadow page table**
- Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
  - Shadow page table is never modified during execution
- To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
- Whenever any page is about to be written for the first time
  - A copy of this page is made onto an unused page. [to be a new current page]
  - The current page table is then made to point to the copy
  - The update is performed on the copy

Sample Page Table



Example of Shadow Paging

Shadow and



- To commit a transaction :
  1. Flush all modified pages in main memory to disk
  2. Output current page table to disk
  3. Make the current page table the new shadow page table, as follows:
     - keep a pointer to the shadow page table at a fixed (known) location on disk.
     - to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
- Once pointer to shadow page table has been written, transaction is committed.
- No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
- Pages not pointed to from current/shadow page table should be freed (garbage collected).

- Advantages of shadow-paging over log-based schemes
  - no overhead of writing log records
  - recovery is faster
- Disadvantages :
  - Copying the entire page table is very expensive
  - Commit overhead is high.
    - Need to flush every updated page, and page table
  - Data gets fragmented (related pages get separated on disk)
  - After every transaction completion, the database pages containing old versions of modified data need to be garbage collected.

Deadlocks

- Consider the partial schedule

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- Neither T_3 nor T_4 can make progress — executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

- Two-phase locking *does not* ensure freedom from deadlocks.
- In addition to deadlocks**,** there is a possibility of **starvation.**
- **Starvation** occurs if the concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

- The potential for deadlock exists in most locking protocols.

- When a deadlock occurs there is a possibility of cascading roll-backs.

- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking** -- a transaction must hold all its exclusive locks till it commits/aborts.

- **Rigorous two-phase locking** is even stricter. Here, *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Deadlock Detection

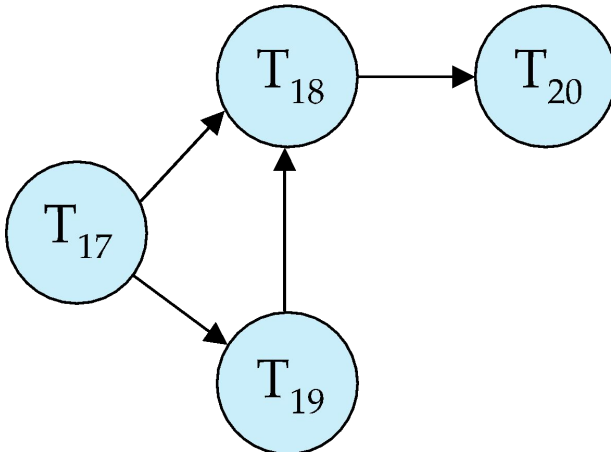
- Deadlocks can be described as a ***wait-for*** *graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.

- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.

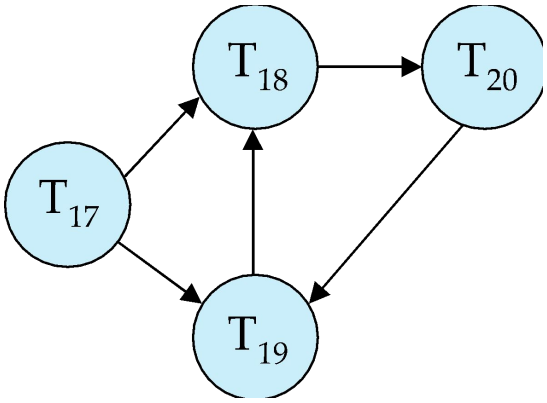
- When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.

- The system is in a **deadlock** state if and only if the **wait-for graph has a cycle**. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  - **Starvation** happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation.

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order.

More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- **wait-die** scheme — non-preemptive
  - older transaction may wait for younger one to release data item. (older means smaller timestamp).
  - Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme.

1. The **wait–die scheme is a nonpreemptive technique.** When transaction *Ti requests* a data item currently held by *Tj , Ti is allowed to wait only if it has a* timestamp **smaller than** that of *Tj (that is, Ti is older than Tj). Otherwise, Ti is* rolled back (dies).

For example, suppose that transactions *T22, T23, and T24 have timestamps* 5, 10, and 15, respectively. If *T22 requests a data item held by T23, then T22 will* wait. If *T24 requests a data item held by T23, then T24 will be rolled back.*

- **2. The wound–wait scheme is a preemptive technique. It is a counterpart to the** wait–die scheme.
  - When transaction Ti requests a data item currently held by Tj , Ti is allowed to wait only if it has a **timestamp larger than** that of Tj (that is, Ti is younger than Tj). Otherwise, Tj is rolled back (Tj is wounded by Ti).

- **For example** with transactions T22, T23, and T24, if T22 requests a data item held by T23, then the data item will be preempted from T23, and **T23 will be rolled back**. If T24 requests a data item held by T23, then T24 will wait.

- **Timeout-Based Schemes:**
  - a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted,
  - Thus, deadlocks are not possible
  - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

END

Query Processing and Optimization

- **Query processing** refers to the range of activities involved in extracting data from a database.

- The activities include translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries.

- The steps involved in processing a query are:
  1. Parsing and translation.
  2. Optimization.
  3. Evaluation.

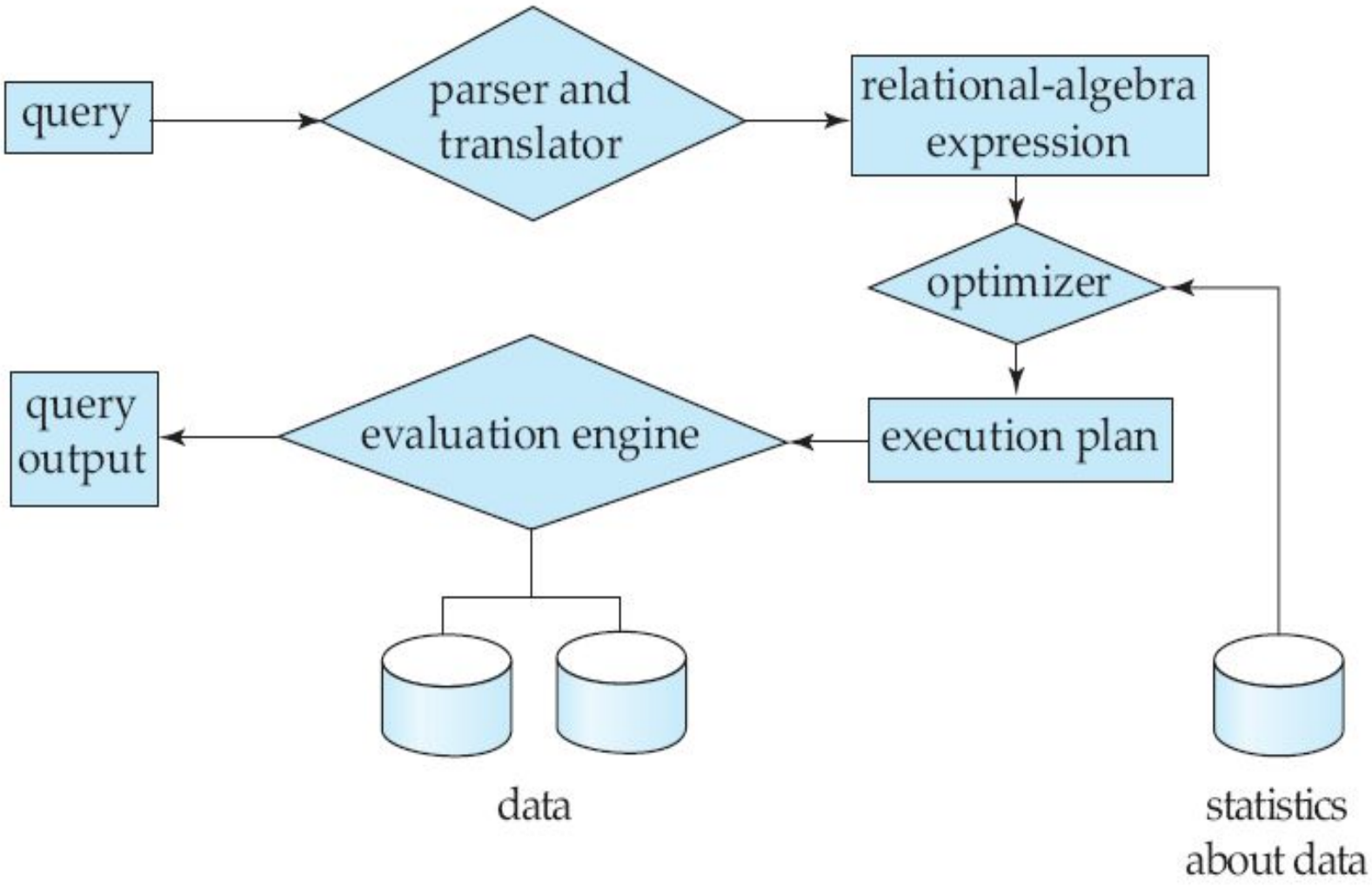
- Before query processing can begin, the system must **translate the query into a usable form.**
- A more useful internal representation is one based on the extended relational algebra.
- Thus, the first action the system must take in query processing is to translate a given **query into its internal form.**
- This translation process is similar to the work performed by the **parser of a compiler.**

- In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on.

- The system constructs a **parse-tree representation** of the query, which it then translates into a **relational-algebra expression.**

- If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view.

Steps in query processing



- Given a query, there are generally a variety of methods for computing the answer.
- Each SQL query can itself be translated into a relational algebra expression in one of several ways.
- Furthermore, the relational-algebra representation of a query specifies only partially how to evaluate a query; there are usually several ways to evaluate relational-algebra expressions.
- consider the query:

```
select salary
from instructor
where salary < 75000;
```

- This query can be translated into either of the following relational-algebra expressions:

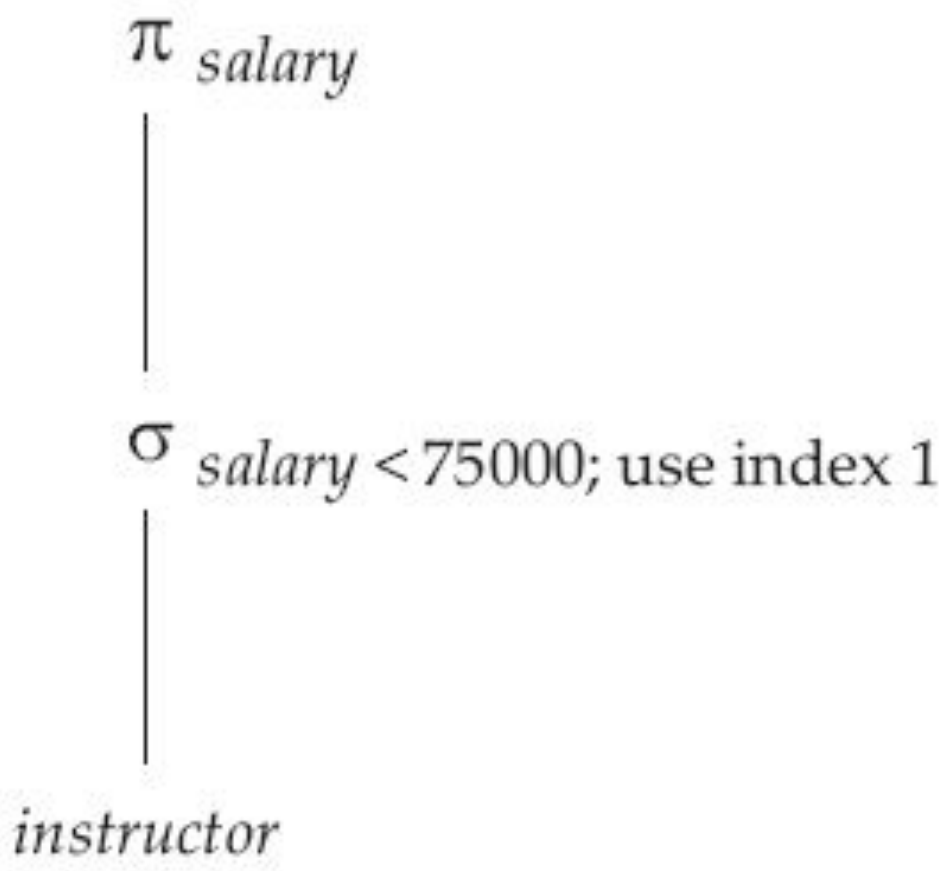
$$\sigma_{salary < 75000}(\pi_{salary}(instructor))$$

$$\pi_{salary}(\sigma_{salary < 75000}(instructor))$$

- Further, we can execute each relational-algebra operation by one of several different algorithms.

- For example, to implement this, we can search every tuple in instructor to find tuples with salary less than 75000.

Query evaluation plan



- A relational algebra operation interpreted with instructions on how to evaluate it is called an **evaluation primitive.**

- **A sequence of primitive operations** that can be used to evaluate a query is a **query-execution plan or query-evaluation plan.**

- In previous figure the evaluation plan for given query, in which a particular index (denoted in the figure as "index 1") is specified for the selection operation.

- The **query-execution engine takes a query-evaluation plan,** executes that plan, and returns the answers to the query.

- The different evaluation plans for a given query can have different costs.

- We do not expect users to write their queries in a way that suggests the most efficient evaluation plan.

- Rather, it is the responsibility of the system to construct a query evaluation plan that minimizes the cost of query evaluation; this task is called **query optimization.**

- Once the query plan is chosen, the query is evaluated with that plan, and the result of the query is output.

Query optimization

- **Query optimization** is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex.

- We do not expect users to write their queries so that they can be processed efficiently. Rather, we expect the system to construct a query-evaluation plan that minimizes the cost of query evaluation.

- One aspect of optimization occurs at the relational-algebra level, where the system attempts to find an expression that is equivalent to the given expression, but more efficient to execute.

- Other aspect is selecting a detailed strategy for processing the query, such as choosing the algorithm to use for executing an operation, choosing the specific indices to use, and so on.

- The difference in cost (in terms of evaluation time) between a good strategy and a bad strategy is often large, and may be several orders of magnitude.

- Hence, it is worthwhile for the system to spend a substantial amount of time on the selection of a good strategy for processing a query, even if the query is executed only once.

- Consider the following relational-algebra expression, for the query,

"Find the names of all instructors in the Music department together with the course title of all the courses that the instructors teach."

- $\pi_{name,title}(\sigma_{dept\text{-}name=\text{“Music”}}(instructor \bowtie teaches) \bowtie \pi_{course\_id,title}(course))$

- The projection of course on (name,title) is required since course shares an attribute **dept name** with **instructor.**

- (if we did not remove this attribute using the projection, the above expression using natural joins would return only courses from the Music department, even if some Music department instructors taught courses in other departments.)

- The above expression constructs a large intermediate relation,

instructor ⋈ teaches ⋈ πcourse id ,title (course).

- However, we are interested in only a few tuples of this relation (those pertaining to instructors in the Music department), and in only two of the ten attributes of this relation.

- Since we are concerned with only those tuples in the instructor relation that pertain to the **Music** department, we do not need to consider those tuples that do not have dept name = “Music”.

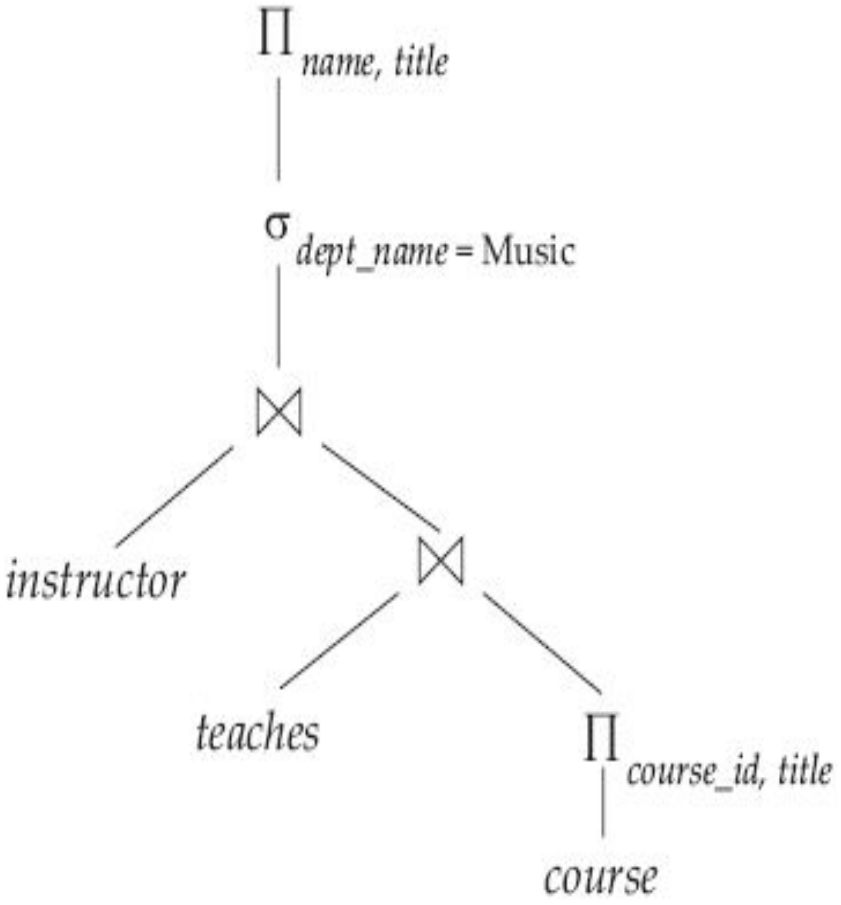
- By reducing the number of tuples of the instructor relation that we need to access, we reduce the size of the intermediate result.

- This query is now represented by the relational-algebra expression:

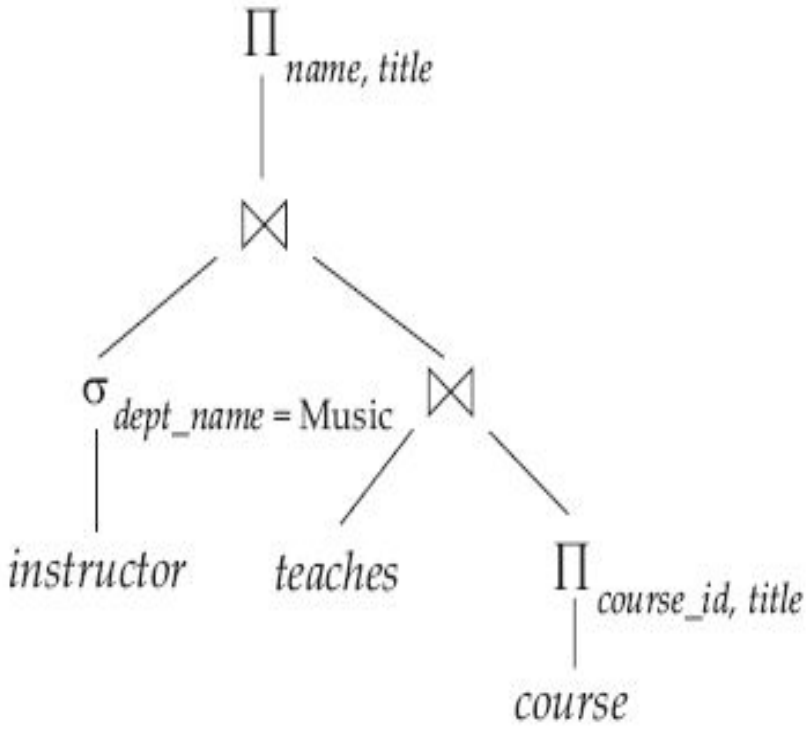
- $\Pi_{name,title}((\sigma_{dept\ name = \text{“Music”}}(instructor)) \bowtie teaches \bowtie \Pi_{course\ id, title}(course)))$.

- which is equivalent to our original algebra expression, but which generates smaller intermediate relations.

Equivalent expressions



(a) Initial expression tree



(b) Transformed expression tree

Consider the relational-algebra expression for the query

“Find the names of all customers who have an account at any branch located in Brooklyn.”

$$\Pi_{customer\text{-}name}\left(\sigma_{branch-city=\text{“Brooklyn”}}\left(branch \bowtie \left(account \bowtie depositor\right)\right)\right)$$

- This expression constructs a large intermediate relation, branch ⋈ account ⋈ depositor.

- However, we are interested in only a few tuples of this relation (those pertaining to branches located in Brooklyn), and in only one of the six attributes of this relation.

- Since we are concerned with only those tuples in the branch relation that pertain to branches located in Brooklyn, we do not need to consider those tuples that do not have branch-city = “Brooklyn”.

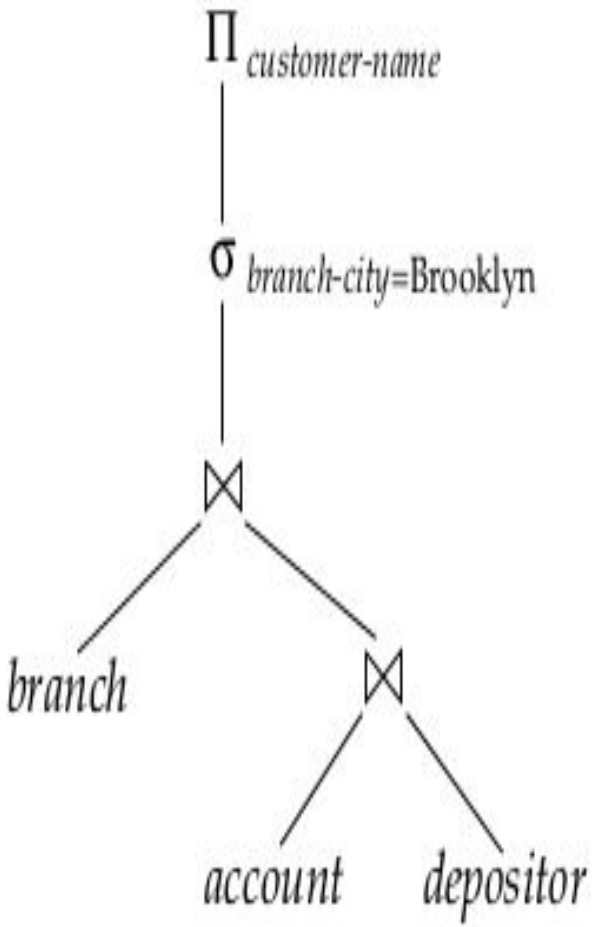
- By reducing the number of tuples of the branch relation that we need to access, we reduce the size of the intermediate result.

- This query is now represented by the relational-algebra expression,

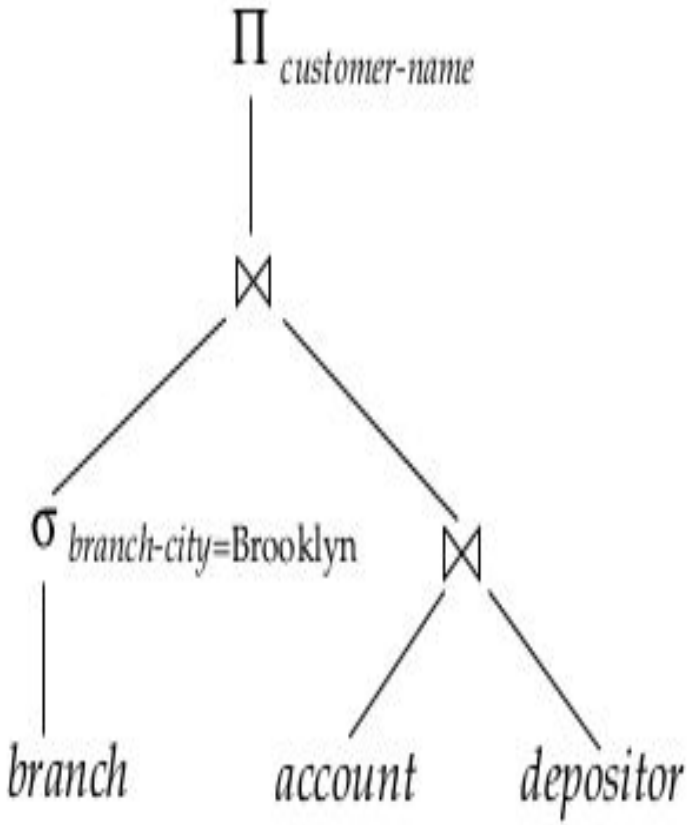
$$\Pi_{customer\text{-}name}\left(\left(\sigma_{branch\text{-}city = \text{“Brooklyn”}}(branch)\right) \bowtie (account \bowtie depositor)\right)$$

which is equivalent to our original algebra expression,
but which generates smaller intermediate relations.

•Following figure shows the initial and transformed expressions.



(a) Initial expression tree



(b) Transformed expression tree

Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use heuristics to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

References

https://www.wisdomjobs.com/e-university/database-system-concepts-tutorial-528/query-processing-15035.html

https://www.youtube.com/watch?v=Sn_Wkf9KNEg

END