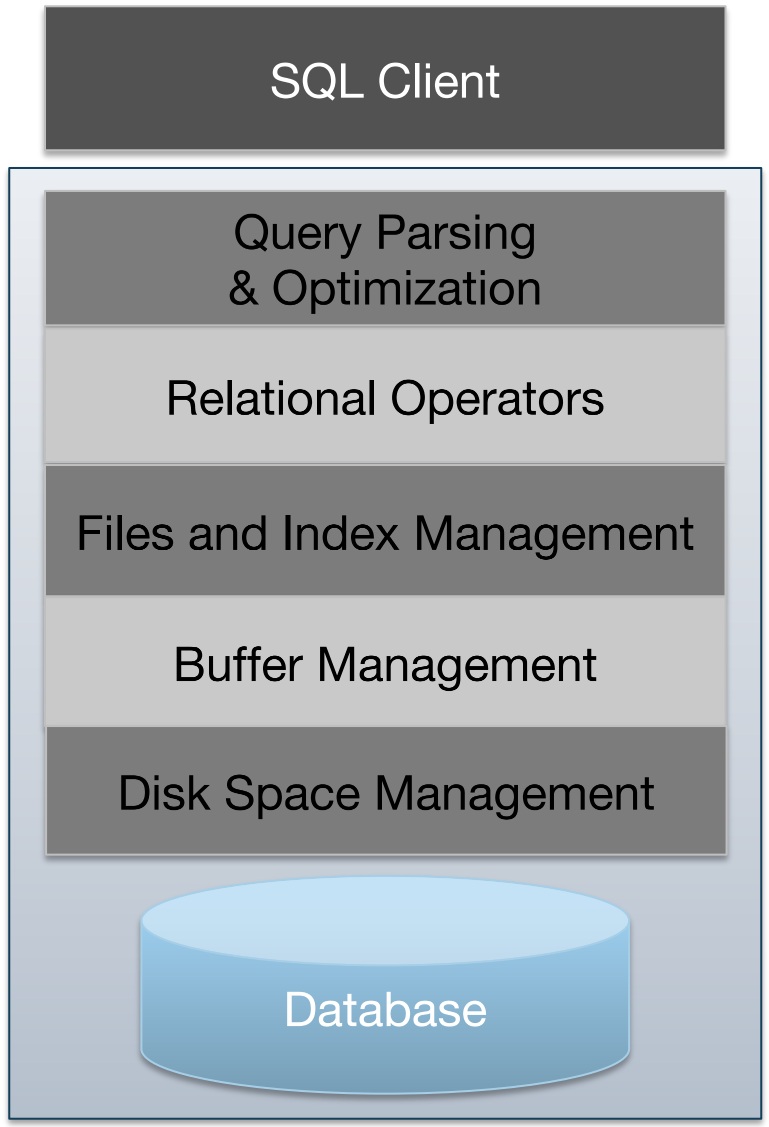
Introduction to Database Systems

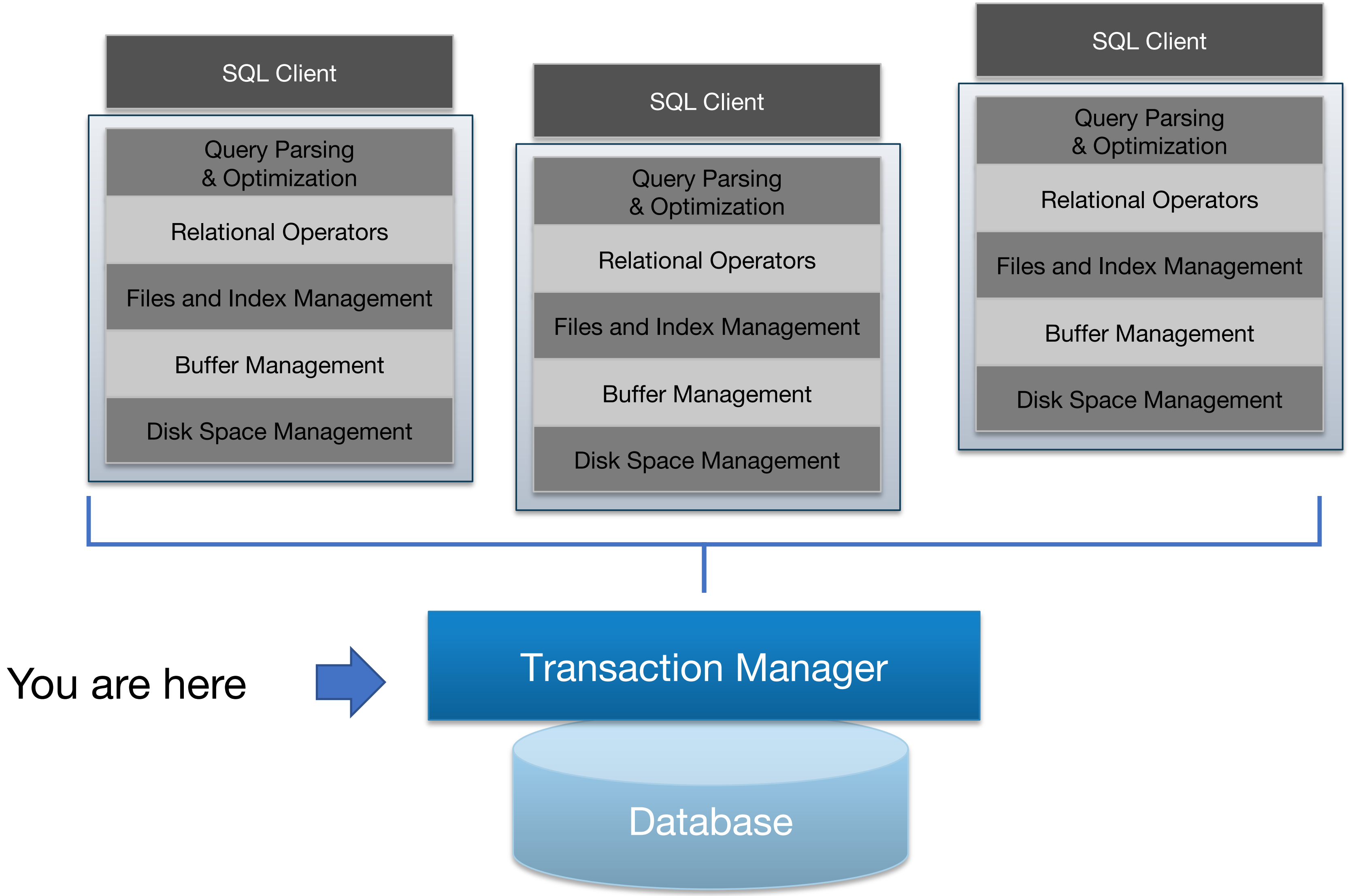
2023-Fall

Transactions Management

Architecture of a DBMS

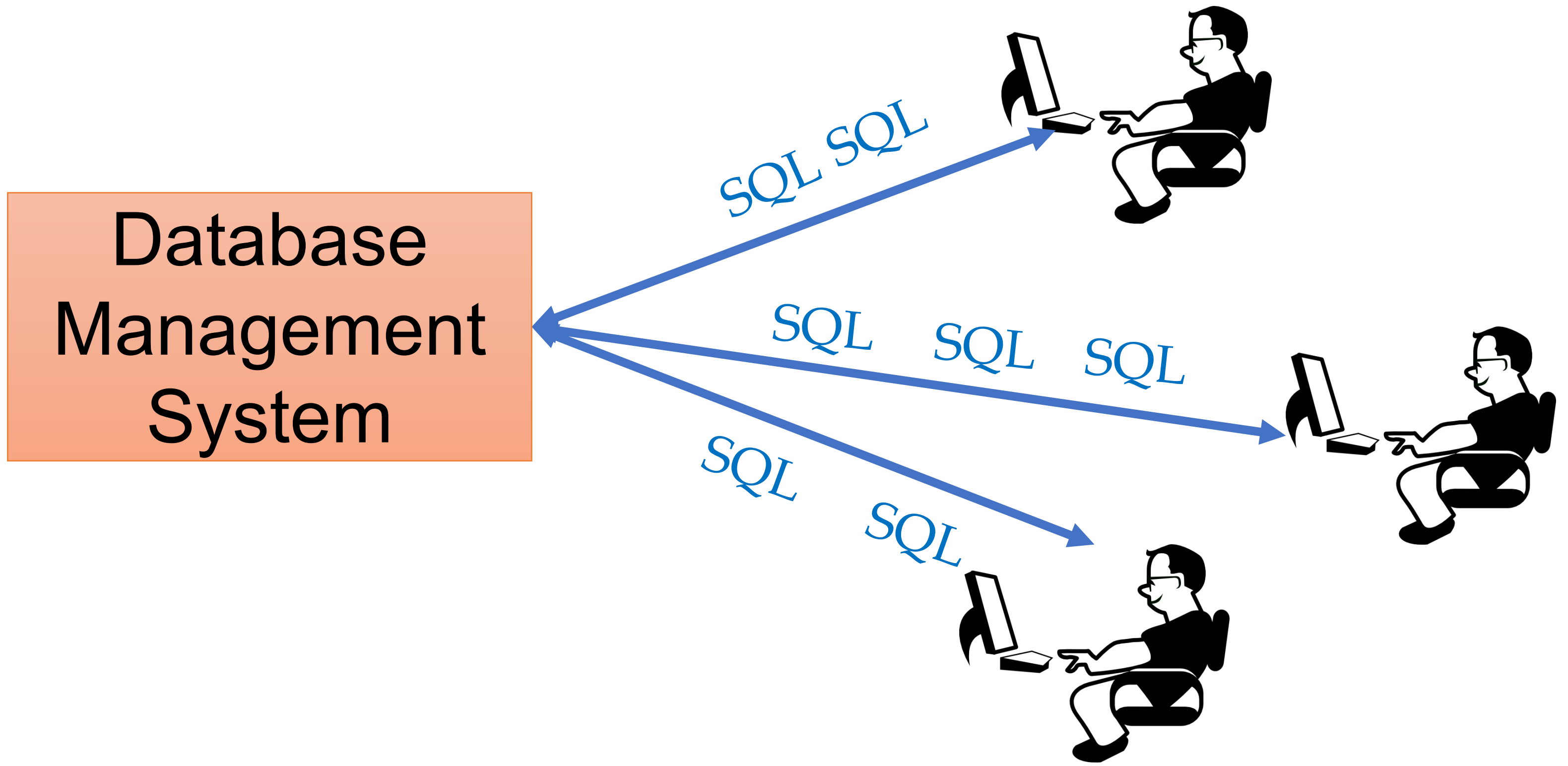


Architecture of a DBMS, Part 2



Applications Want Something from the DBMS

- Queries and updates of course: what you learned so far!
- Real applications are composed of many statements being generated by user behaviors
- Many users work with the application at the same time



Transactions

Transaction: Concept and Implementation

- Major component of database systems
- Critical for most applications; arguably more so than SQL

What is a Transaction?

- A *sequence* of *multiple actions* to be executed as an ***atomic* unit**
- Application View (SQL View):
  - Begin transaction
  - Sequence of SQL statements
  - End transaction
- Examples
  - Transfer money between accounts
  - Book a flight, a hotel and a car together on Expedia

Transaction Example

- Transaction to transfer $100 from account R to account S

Not seen by the DBMS transaction manager!

1. start transaction
2. read(R)
3. R = R – 100
4. write(R)
5. read(S)
6. S = S + 100
7. write(S)
8. end transaction

ACID: High-Level Properties of Transactions

- **Atomicity:** *All* actions in the **transaction** happen, or *none* happen.
- **Consistency:** If the DB *starts* out *consistent*, it *ends* up *consistent* at the end of the **transaction**
- **Isolation:** Execution of *each* **transaction** is *isolated from* that of *others*
- **Durability:** If a **transaction** *commits*, its effects *persist*.

Note: This is a mnemonic, not a formalism. We'll do some formalisms shortly.

Isolation (Concurrency)

- DBMS interleaves actions of many transactions
  - Actions = reads/writes of DB objects
- DBMS ensures 2 transactions do not “interfere”
- Each transaction executes as if it ran by itself.
  - Concurrent accesses have no effect on transaction’s behavior
  - Net effect must be identical to executing all transactions in some serial order
  - Users & programmers think about transactions in isolation
    - Without considering effects of other concurrent transactions!

Atomicity and Durability

- **A transaction ends in one of two ways:**
  - **Commit** after completing all its actions
    - “commit” is a contract with the caller of the DB
  - **Abort** (or be aborted by the DBMS) after executing some actions
    - Or **system crash** while the transaction is in progress; treat as abort.
- **Two key properties** for a transaction
  - **Atomicity**: Either execute all its actions, or none of them
  - **Durability**: The effects of a committed transaction must survive failures.
- DBMS typically ensures the above by **logging** all actions:
  - **Undo** the actions of aborted/failed transactions.
  - **Redo** actions of committed transactions not yet propagated to disk when system crashes

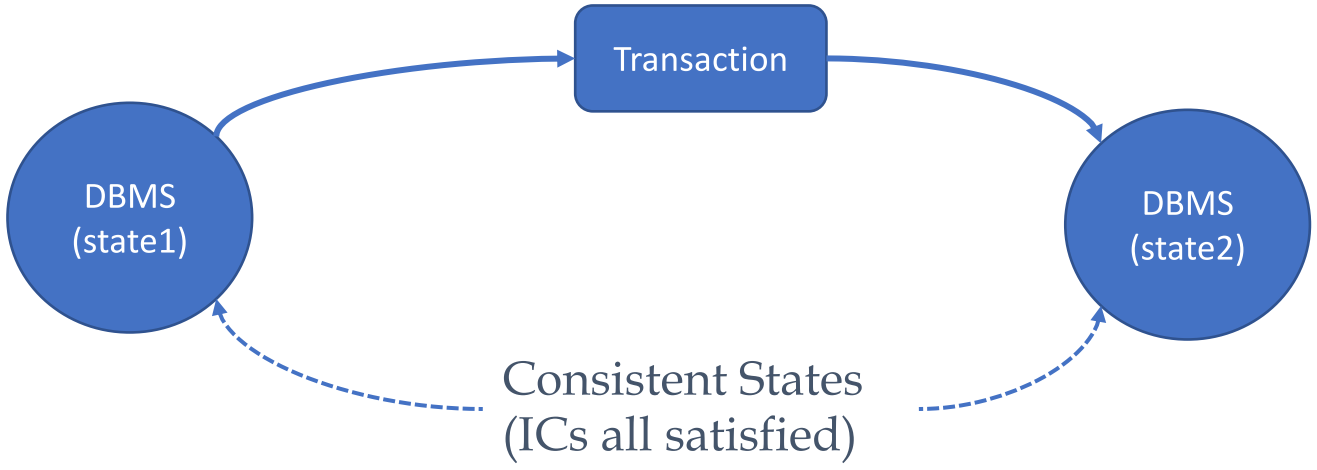
Atomicity and Durability, cont.

- Atomicity
  - If the transaction fails after step 4 and before step 7
    - Money will be "lost" → inconsistent database
  - DBMS should ensure that updates of a partially executed transaction are not reflected
- Durability
  - Once the user hears that the transaction is complete, can rest easy that the \$100M was xferred from R to S.

```
1. start
   transaction
2. read(R)
3. R = R – 100
4. write(R)
5. read(S)
6. S = S + 100
7. write(S)
8. end
   transaction
```

Transaction Consistency

- **Transactions preserve DB consistency**
  - Given a consistent DB state, produce another consistent DB state
- DB consistency expressed as a set of **declarative integrity constraints**
  - CREATE TABLE/ASSERTION statements
- **Transactions that violate integrity are aborted**
  - That's all the DBMS can automatically check!



Summary

- We have seen an overview
- ACID Transactions make guarantees that
  - Improve performance (via concurrency)
  - Relieve programmers of correctness concerns
    - Hide concurrency and failure handling!
- Two key issues to consider, and mechanisms
  - Recovery (via write-ahead logging (WAL))
  - Concurrency control (via two-phase locking)

Recovery

Recovery

Introduction

The main roles of recovery mechanism in DBMS are:

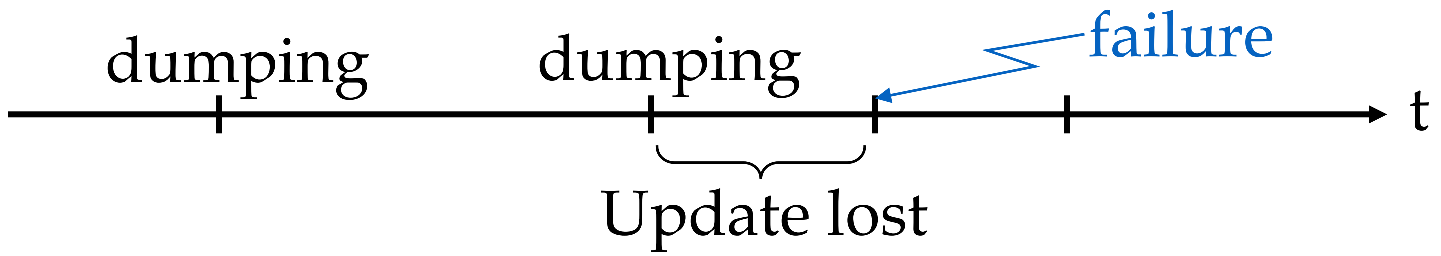
(1) Reducing the likelihood of failures (prevention)

(2) Recover from failures (solving)

Restore DB to a consistent state after some failures.

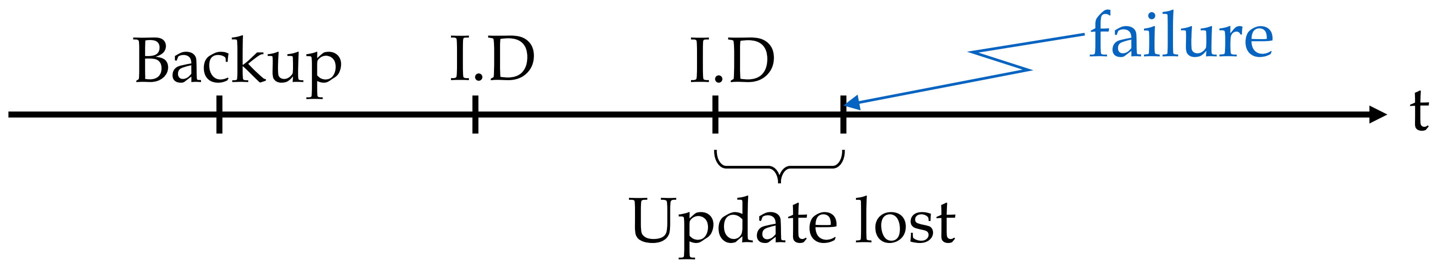
- Redundancy is necessary.
- Should inspect all possible failures.
- General method:

1) Periodical dumping



- Variation : Backup + Incremental dumping

  I.D --- updated parts of DB



This method is easy to be implemented and the overhead is low, but the update maybe lost after failure occurring.

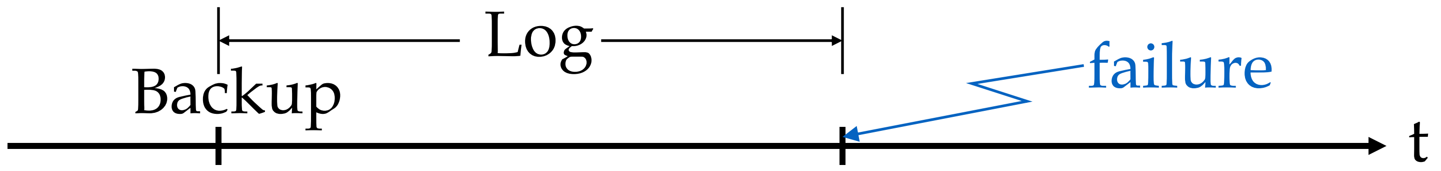
So it is often used in file system or small DBMS.

2) Backup + Log

Log : record of all changes on DB since the last backup copy was made.

$$\text{Change: } \left\{ \begin{array}{l} \text{Old value (Before Image --- B.I)} \\ \text{New value (After Image --- A.I)} \end{array} \right\} \begin{array}{c} \text{Recorded} \\ \text{into Log} \end{array}$$

For	update op.	:	B.I	A.I
	insert op.	:	----	A.I
	delete op.	:	B.I	----



While recovering:

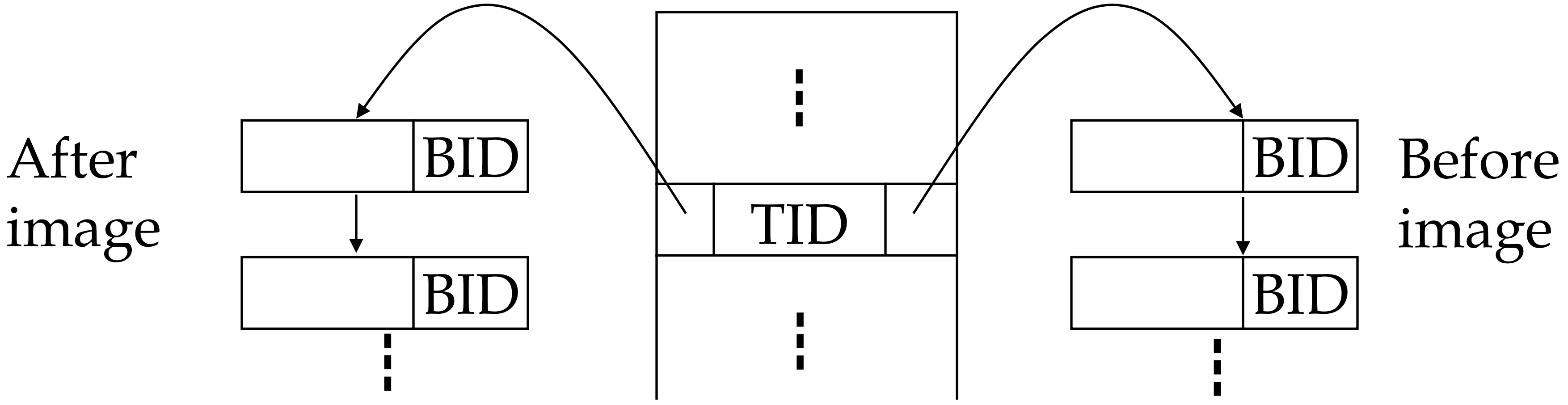
- Some transactions maybe half done, should undo them with B.I recorded in Log.
- Some transactions have finished but the results have not been written into DB in time, should redo them with A.I recorded in Log. (finish writing into DB)

  It is possible to recover DB to the most recent consistent state with Log.

Some Structures to Support Recovery

Recovery information (such as Log) should be stored in ***nonvolatile*** storage. The following information need to be stored in order to support recovery:

1) Commit list : list of TID which have been committed.
2) Active list : list of TID which is in progress.
3) Log :



Commit Rule and Log Ahead Rule

1) Commit Rule

A.I must be written to nonvolatile storage before commit of the transaction.

2) Log Ahead Rule

If A.I is written to DB before commit then B.I must first written to log.

3) Recovery strategies

The features of undo and redo (are *idempotent*) :

undo(undo(undo ⋯ undo(x) ⋯)) = undo(x)

redo(redo(redo ⋯ redo(x) ⋯)) = redo(x)

Three kinds of update strategy-1

a) A.I→DB before commit

TID →active list

B.I →Log (Log Ahead Rule)

A.I →DB, Log

┇

TID →commit list

delete TID from active list

The recovery after failure in this situation

- Check two lists for every TID while restarting after failure:

Commit list	Active list	
	✓	Undo, delete TID from active list
✓	✓	delete TID from active list
✓		nothing to do

Three kinds of update strategy-2

b) A.I→DB after commit

TID →active list

A.I →Log (Commit Rule)

╏

TID →commit list

A.I →DB

delete TID from active list

The recovery after failure in this situation

- Check two lists for every TID while restarting after failure:

Commit list	Active list	
	✓	delete TID from active list
✓	✓	redo, delete TID from active list
✓		nothing to do

Three kinds of update strategy-3

c) A.I→DB concurrently with commit

TID →active list

A.I, B.I →Log (Two Rules)

A.I →DB (partially done)

┇

TID →commit list

A.I →DB (completed)

delete TID from active list

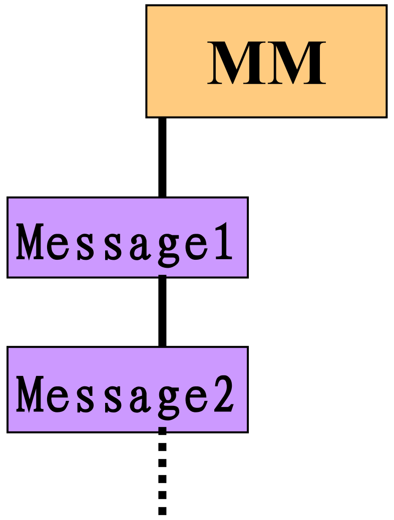
The recovery after failure in this situation

- Check two lists for every TID while restarting after failure:

Commit list	Active list	
	✓	Undo, delete TID from active list
✓	✓	redo, delete TID from active list
✓		nothing to do

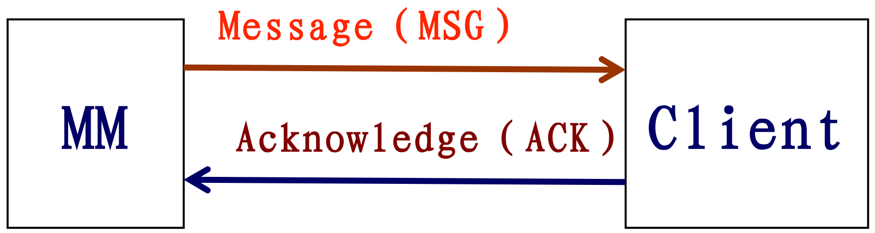
Message Management

- A transaction often needs to send impactful messages to users, which are not the kind of messages that require user input for direction. The task of message sending is delegated to a Message Manager (MM) subsystem.
- During the execution of a transaction, the messages that need to be sent are passed to the MM, which establishes a message queue for each transaction.



Message Management

- The MM allows for the addition and deletion of messages entrusted for sending by the transaction until the normal conclusion of the transaction; once the transaction ends, the MM stores the messages in non-volatile memory.
- When a transaction ends normally (including commit and rollback), it notifies the MM to send the messages; if the transaction is aborted due to a failure, the MM will discard the messages for that transaction.
- To ensure reliable delivery of messages to the concerned users, the MM employs a "send-acknowledge" method of transmission.



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
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovering from Failure

- When recovering after failure:
  - Transaction T_i needs to be undone if the log
    - Contains the record <T_i **start**>,
    - But does not contain either the record <T_i **commit**> *or* <T_i **abort**>.
  - Transaction T_i needs to be redone if the log
    - Contains the records <T_i **start**>
    - And contains the record <T_i **commit**> *or* <T_i **abort**>

Recovering from Failure (Cont.)

- Suppose that transaction T_i was undone earlier and the <T_i **abort**> record was written to the log, and then a failure occurs,
- On recovery from failure transaction T_i is redone
  - Such a **redo** redoes all the original actions of transaction T_i *including the steps that restored old values*
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly

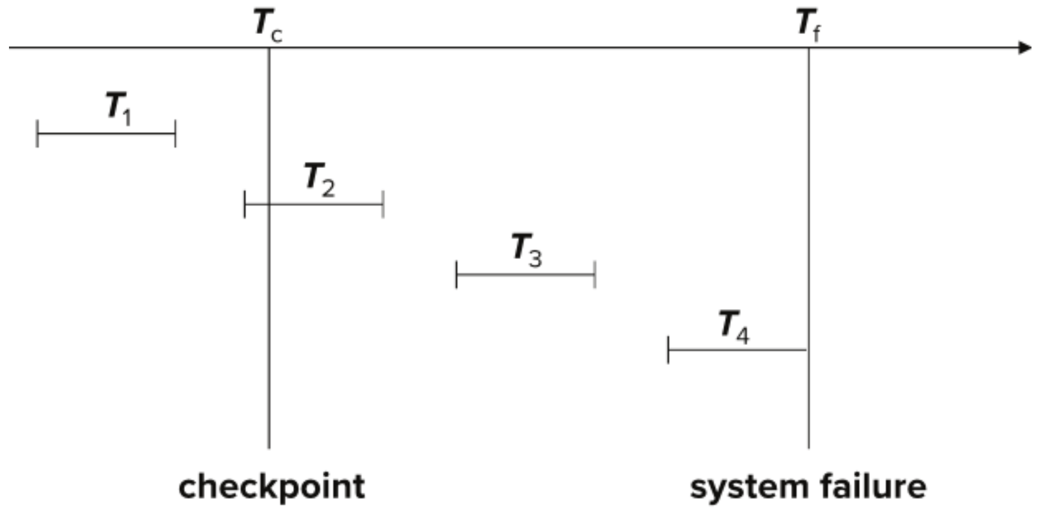
Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  - Processing the entire log is time-consuming if the system has run for a long time
  - 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** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.
  4. All updates are stopped while doing checkpointing

Checkpoints (Cont.)

- 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.
  - Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
  - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
  - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.
- Some earlier part of the log may be needed for undo operations
  - Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in *L*.
  - Parts of log prior to earliest <T_i **start**> record above are not needed for recovery, and can be erased whenever desired.

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

Concurrency Control

Concurrency Control: Providing Isolation

- **Naïve approach - serial execution**
  - One transaction runs at a time
  - Safe but slow
- **Execution must be interleaved for better performance**
- With concurrent executions, how does one **define** and **ensure** correctness?

Transaction Schedules

T1	T2
begin	
read(A)	
write(A)	
read(B)	
write(B)	
commit	
	begin
	read(A)
	write(A)
	read(B)
	write(B)
	commit

A **schedule** is a sequence of actions on data from one or more transactions.

Actions: Begin, Read, Write, Commit and Abort.

$$R_1(A)\ W_1(A)\ R_1(B)\ W_1(B)\ R_2(A)\ W_2(A)\ R_2(B)\ W_2(B)$$

By convention we only include committed transactions, and omit Begin and Commit.

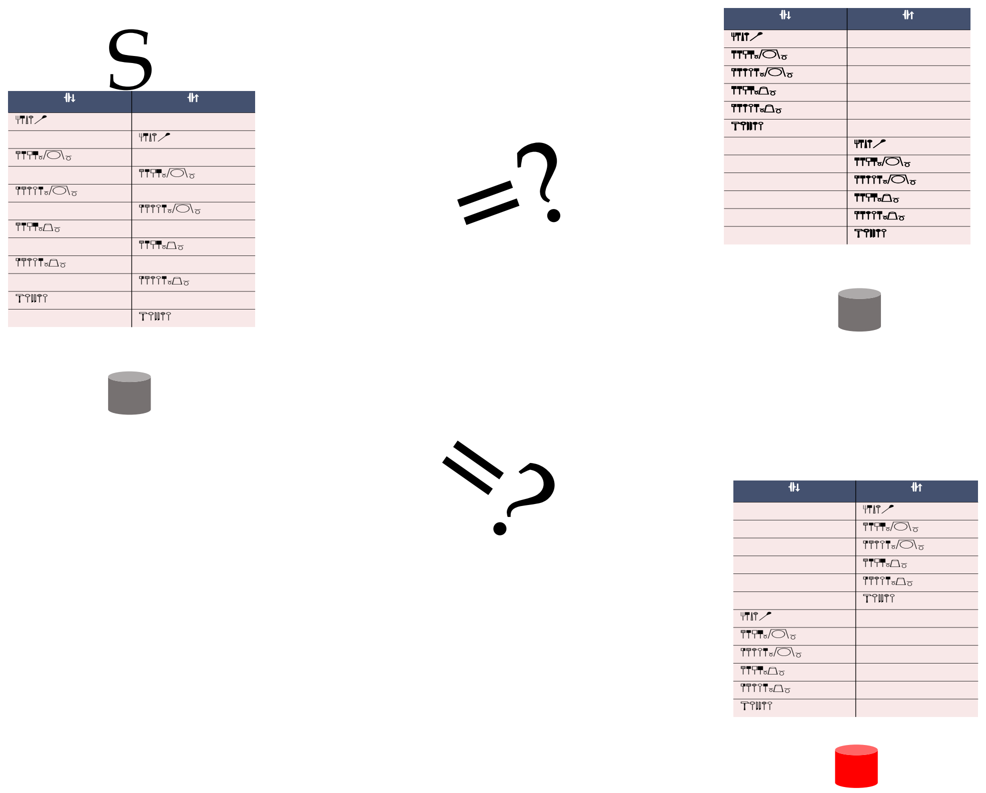
Serial Equivalence

- We need a “touchstone” concept for correct behavior
- **Definition**: **Serial schedule**
  - Each transaction runs from start to finish without any intervening actions from other transactions
- **Definition**: 2 schedules are **equivalent** if they:
  - involve the same transactions
  - each individual transaction’s actions are ordered the same
  - both schedules leave the DB in the same final state

View equivalence

Serializability

- **Definition**: Schedule S is **serializable** if:
  - S is equivalent to some serial schedule



Schedule 1

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
read(B)	
B = B + 100	
write(B)	
commit	
	begin
	read(A)
	A = A * 1.1
	write(A)
	read(B)
	B = B * 1.1
	write(B)
	commit

- Let T1 transfer $100 from A to B
- Let T2 add 10% interest to A & B
- Serial schedule in which T1 is followed by T2
  - Final outcome:
    - A := 1.1*(A-100)
    - B := 1.1*(B+100)

Schedule 2

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
	begin
	read(A)
	A = A * 1.1
	write(A)
	read(B)
	B = B * 1.1
	write(B)
	commit
begin	
read(A)	
A = A - 100	
write(A)	
read(B)	
B = B + 100	
write(B)	
commit	

- Serial schedule in which T2 is followed by T1
  - Final outcome:
    - A := (1.1*A)-100
    - B := (1.1*B)+100
  - Different!
    - But still understandable

Schedule 3

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
	begin
	read(A)
	A = A * 1.1
	write(A)
read(B)	
B = B + 100	
write(B)	
commit	
	read(B)
	B = B * 1.1
	write(B)
	commit

- Schedule in which actions of T1 and T2 are interleaved.
- This is not a serial schedule
- But it is equivalent to schedule 1
  - A := (A-100)*1.1
  - B := (B+100)*1.1
- Hence **serializable**!

Conflicting Operations

- Tricky to check property **“leaves the DB in the same final state”**

- **Use notion of “conflicting” operations (read/write)**

- **Definition: Two operations conflict if they:**
  - Are by different transactions,
  - Are on the same object,
  - At least one of them is a write.

- The order of non-conflicting operations has no effect on the final state of the database!
  - Focus our attention on the order of conflicting operations

Conflict Serializable Schedules

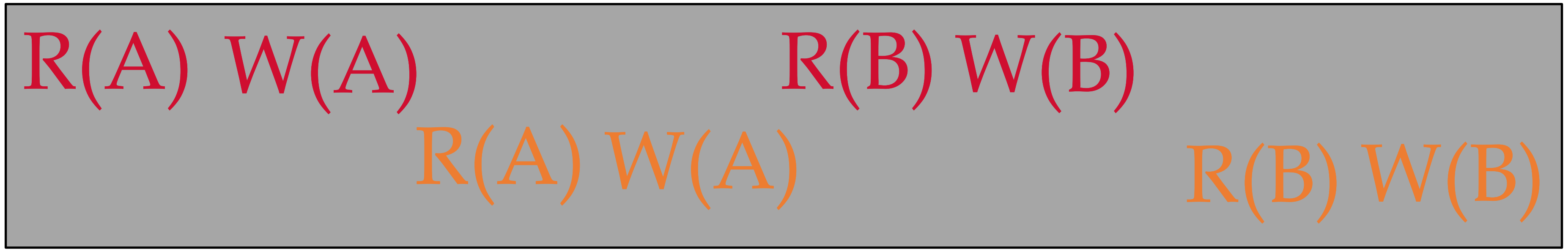
- **Definition: Two schedules are *conflict equivalent* if:**
  - They involve the same actions of the same transactions, and
  - Every pair of conflicting actions is ordered the same way
- **Definition: Schedule S is *conflict serializable* if:**
  - S is conflict equivalent to some serial schedule
  - Implies S is also Serializable

Note: some serializable schedules are NOT conflict serializable

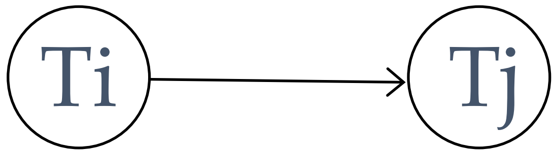
- Conflict serializability gives false negatives as a test for serializability!
- The cost of a conservative test
- A price we pay to achieve efficient enforcement

Conflict Serializability - Intuition

- **A schedule S is conflict serializable if**
  - You are able to transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different transactions
- ***Example***



Conflict Dependency Graph


$$\begin{tikzcd} \boxed{Ti} \arrow[r] & \boxed{Tj} \end{tikzcd}$$

- **Dependency Graph:**
  - One node per Xact
  - Edge from Ti to Tj if:
    - An operation Oi of Ti conflicts with an operation Oj of Tj **and**
    - Oi appears earlier in the schedule than Oj

- **Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic.**

Proof Sketch: Conflicting operations prevent us from “swapping” operations into a serial schedule

Example

- **A schedule that is not conflict serializable**

T1: R(A), W(A)

Dependency graph

Example, pt 2

- **A schedule that is not conflict serializable**

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

$$\begin{tikzcd} T1 \arrow[r] & T2 \end{tikzcd}$$

Dependency graph

Example, pt 3

- **A schedule that is not conflict serializable**

```
T1:   R(A), W(A),
T2:               R(A), W(A), R(B), W(B)
```

$$\begin{tikzcd} T1 \arrow[rr, "A"] & & T2 \end{tikzcd}$$

Dependency graph

Example, pt 4

- **A schedule that is not conflict serializable**

```
T1:   R(A), W(A),                              R(B)
T2:               R(A), W(A), R(B), W(B)
```

$$\begin{tikzcd} T1 \arrow[r, shift left, "A"] & T2 \arrow[l, shift left, "B"] \end{tikzcd}$$

Dependency graph

Notes on Serializability Definitions

- **View Serializability allows (a few) more schedules than conflict serializability**
  - But V.S. is difficult to enforce efficiently.

- **Neither definition allows all schedules that are actually serializable.**
  - Because they don’t understand the meanings of the operations or the data

- **Conflict Serializability is what gets used, because it can be enforced efficiently**
  - To allow more concurrency, some special cases do get handled separately.
  - (Search the web for “Escrow Transactions” for example)

Locking Protocol

Locking Protocol

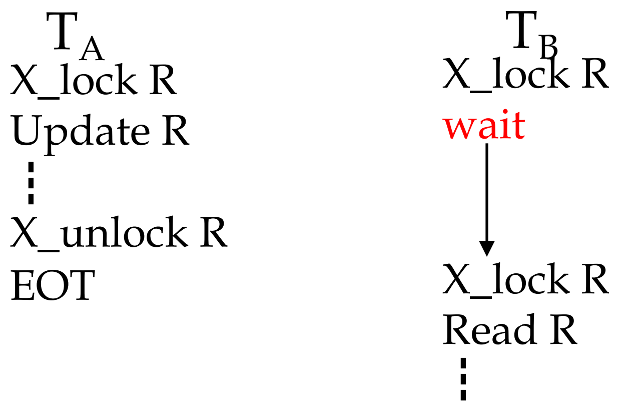
Locking method is the most basic concurrency control method. There maybe many kinds of locking protocols.

(1) X locks

Only one type of lock, for both read and write.

Compatibility matrix : NL—no lock X—X lock
Y —compatible N—incompatible

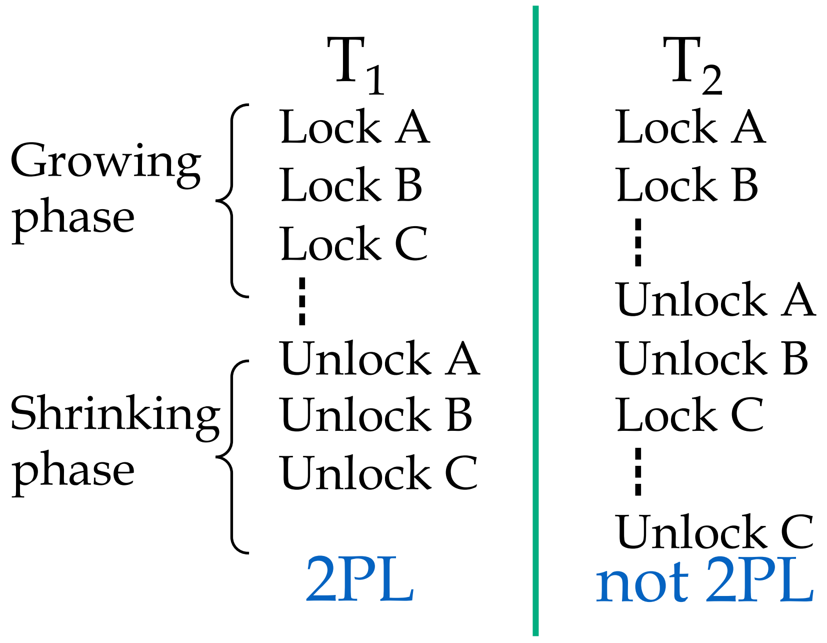
	NL	X
NL	Y	Y
X	Y	N



*Two Phase Locking

- **Definiton1***:* In a transaction, if all locks precede all unlocks, then the transaction is called two phase transaction. This restriction is called ***two phase locking protocol***.
- **Definition2***:* In a transaction, if it first acquires a lock on the object before operating on it, it is called ***well-formed***.

- *Theorem:* If S is any schedule of well-formed and two phase transactions, then S is serializable.
  (proving is omitted)



Two Phase Locking (2PL)

- **The most common scheme for enforcing conflict serializability**
- **A bit “pessimistic”**
  - Sets locks for fear of conflict… Some cost here.
  - Alternative schemes use multiple versions of data and “optimistically” let transactions move forward
    - Abort when conflicts are detected.
    - Some names to know/look up:
      - Optimistic Concurrency Control
      - Timestamp-Ordered Multi-version Concurrency Control

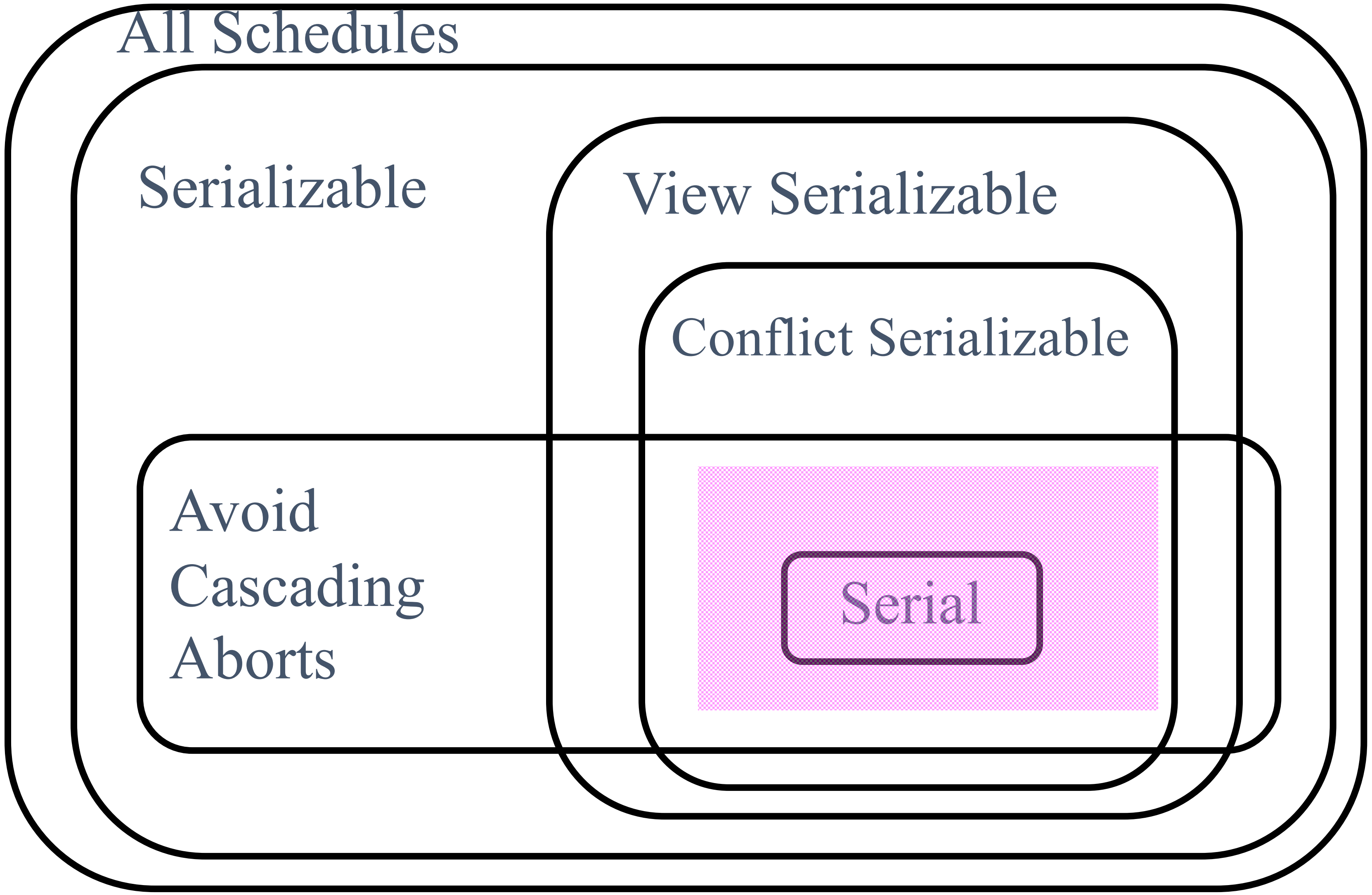
Why 2PL guarantees conflict serializability

- When a committing transaction has reached the end of its acquisition phase…
  - Call this the "lock point"
  - At this point, it has *everything it needs* locked…
  - … and any conflicting transactions either:
    - started release phase before this point
    - are blocked waiting for this transaction
- Visibility of actions of two conflicting transactions are ordered by their lock points
- The order of lock points gives us an equivalent serial schedule!

Conclusions :

1) Well-formed + 2PL : serializable
2) Well-formed + 2PL + unlock update at EOT: serializable and recoverable. (without domino phenomena)
3) Well-formed + 2PL + holding all locks to EOT: strict two phase locking transaction.

Which schedules does Strict 2PL allow?



(S,X) locks

(2) (S,X) locks

S lock --- if read access is intended.

X lock --- if update access is intended.

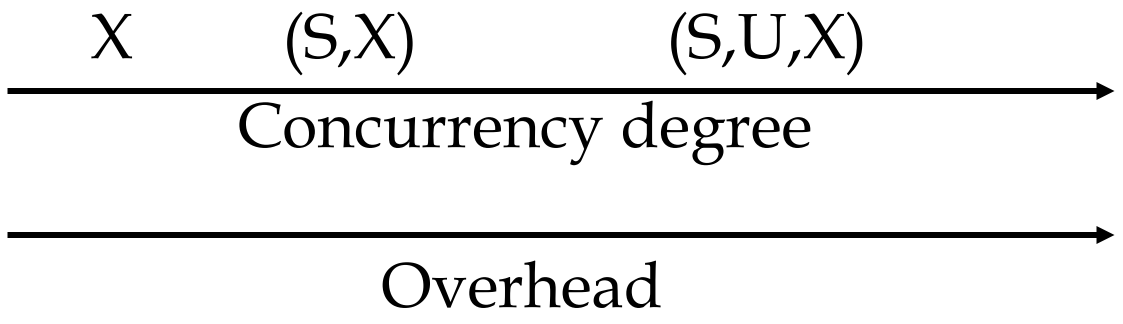
	NL	S	X
NL	Y	Y	Y
S	Y	Y	N
X	Y	N	N

(S,U,X) locks

U lock --- update lock. For an update access the transaction first acquires a U-lock and then promote it to X-lock.

Purpose: shorten the time of exclusion, so as to boost concurrency degree, and reduce deadlock.

	NL	S	U	X
NL	Y	Y	Y	Y
S	Y	Y	Y	N
U	Y	Y	N	N
X	Y	N	N	N



Lock Management

- Lock and unlock requests handled by ***Lock Manager***
- LM maintains a ***hash table***, keyed on names of objects being locked.
- LM keeps an entry for each currently held lock
- Entry contains
    - Granted set: Set of xacts currently granted access to the lock
    - Lock mode: Type of lock held (shared or exclusive)
    - Wait Queue: Queue of lock requests

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3(X) ← T4(X)
B	{T6}	X	T5(X) ← T7(S)

Lock Management (continued)

- **When lock request arrives:**
  - Does any xact in Granted Set or Wait Queue want a conflicting lock?
    - If no, put the requester into "granted set" and let them proceed
    - If yes, put requester into wait queue (typically FIFO)

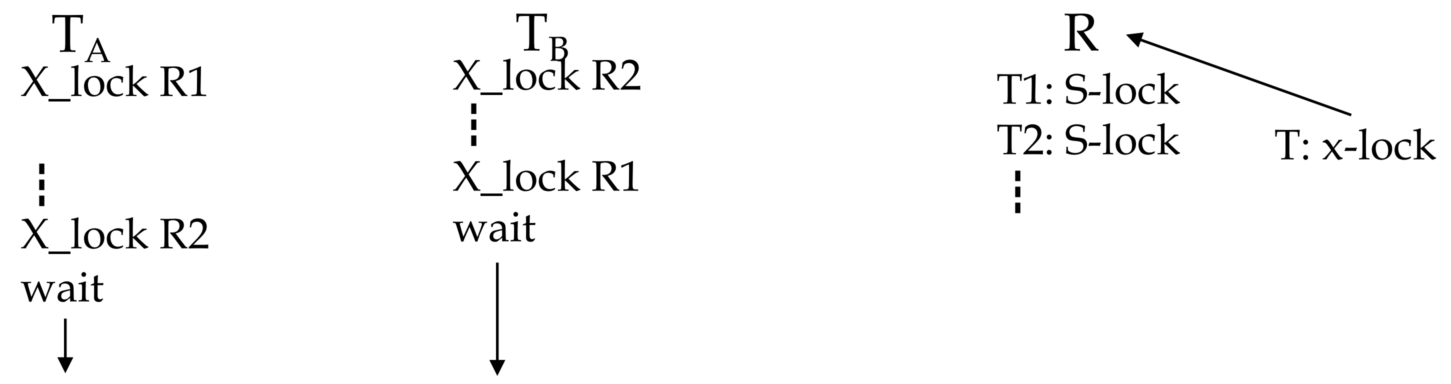
- **Lock upgrade:**
  - Xact with shared lock can request to upgrade to exclusive

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	**T2(X)** ← T3(X) ← T4(X)
B	{T6}	X	T5(X) ← T7(S)

Deadlock & Live Lock

Dead lock: wait in cycle, no transaction can obtain all of resources needed to complete.

Live lock: although other transactions release their resource in limited time, some transaction can not get the resources needed for a very long time.



- Live lock is simpler, only need to adjust schedule strategy, such as FIFO
- Deadlock: (1) Prevention(don't let it occur); (2) Solving(permit it occurs, but can solve it)

Deadlocks

- **Deadlock: Cycle of Xacts waiting for locks to be released by each other.**
- **Three ways of dealing with deadlocks:**
  - Prevention
  - Avoidance
  - Detection and Resolution
- **Many systems just punt and use timeouts**
  - What are the dangers with this approach?

Deadlock Detection

1. Timeout: If a transaction waits for some specified time then deadlock is assumed and the transaction should be aborted.

2. Detect deadlock by wait-for graph G=<V,E>

V : set of transactions $\{T_i | T_i$ is a transaction in DBS $(i=1,2,\ldots n)\}$

E : $\{<T_i,T_j> | T_i$ waits for T_j $(i \neq j)\}$

- If there is cycle in the graph, the deadlock occurs.
- When to detect?
  1) whenever one transaction waits.
  2) periodically

What to do when detected?

1) Pick a victim (youngest, minimum abort cost, …)
2) Abort the victim and release its locks and resources
3) Grant a waiter
4) Restart the victim (automatically or manually)

Deadlock avoidance

1) Requesting all locks at initial time of transaction.
2) Requesting locks in a specified order of resource.
3) Abort once conflicted.
4) Transaction Retry

Deadlock avoidance

- Every transaction is uniquely time stamped. If T_A requires a lock on a data object that is already locked by T_B, one of the following methods is used:

a) ***Wait-die***: T_A waits if it is older than T_B, otherwise it "dies", i.e. it is aborted and automatically retried with original timestamp.

b) ***Wound-wait***: T_A waits if it is younger than T_B, otherwise it "wound" T_B, i.e. T_B is aborted and automatically retried with original timestamp.

In above, both have only one direction wait, either older → younger or younger → older. It is impossible to occur wait in cycle, so the dead lock is avoided.

Lock Granularity

Lock Granularity

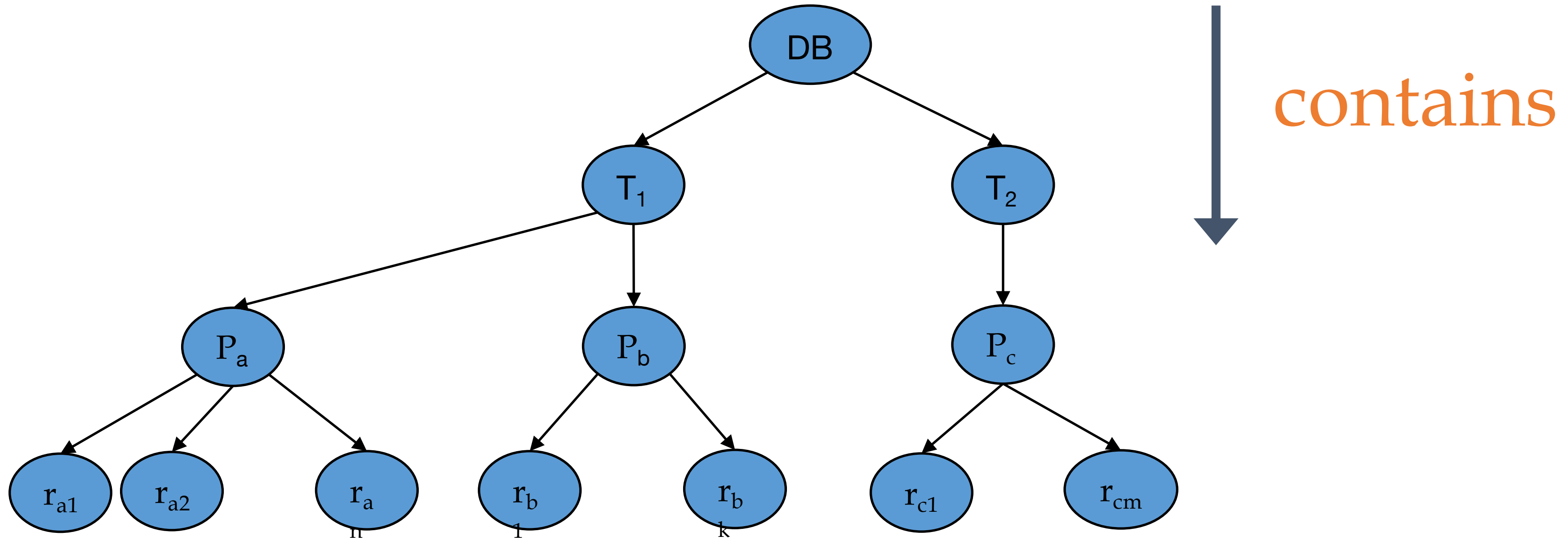
- Hard to decide what granularity to lock
    - Tuples vs Pages vs Tables?
- What is the tradeoff?
    - Fine-grained availability of resources would be nice (e.g. lock per tuple)
    - Small # of locks to manage would also be nice (e.g. lock per table)
    - Can’t have both!
        - Or can we???

Multiple Locking Granularity

- **Shouldn’t have to make same decision for all transactions!**
- Allow data items to be of various sizes
- Define a hierarchy of data granularities, small nested within large
  - Can be represented graphically as a tree.

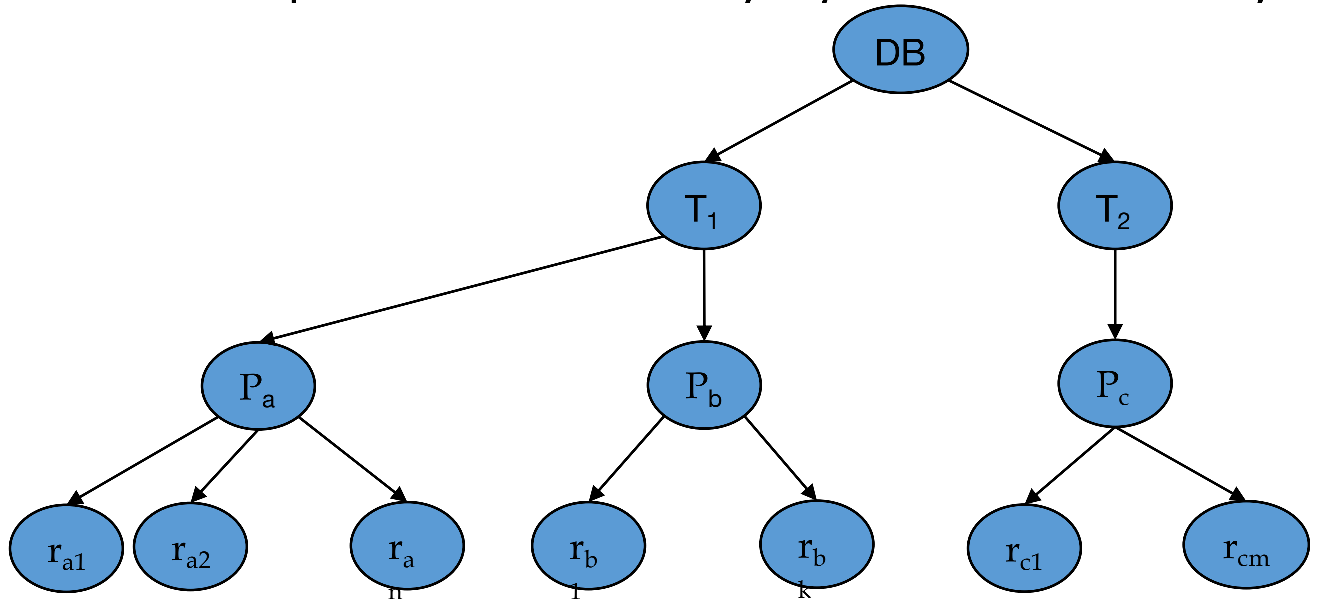
Example of Granularity Hierarchy (RDBMS)

- Data "containers" can be viewed as nested.
- The levels, starting from the coarsest (top) level are
  - Database, Tables, Pages, Records
- When a transaction locks a node in the tree ***explicitly***, it ***implicitly*** locks all the node's descendants in the same mode.



Multiple Locking Granularity

- Granularity of locking (level in tree where locking is done):
  - **Fine granularity** (lower in tree): High concurrency, lots of locks (overhead)
  - **Coarse granularity** (higher in tree): Few locks (low overhead), lost concurrency
    - Lost potential concurrency if you don't need everything inside the coarse grain



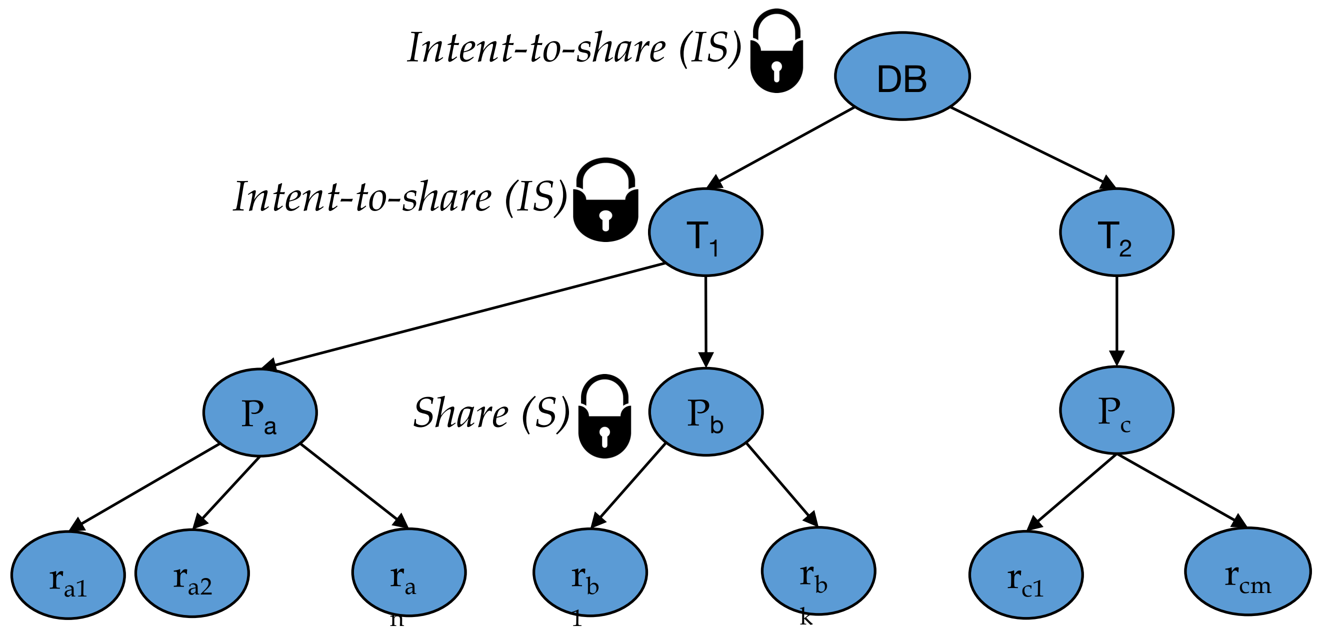
Real-World Locking Granularities

Resource	Description
RID	A row identifier used to lock a single row within a heap.
KEY	A row lock within an index used to protect key ranges in serializable transactions.
PAGE	An 8-kilobyte (KB) page in a database, such as data or index pages.
EXTENT	A contiguous group of eight pages, such as data or index pages.
HoBT	A heap or B-tree. A lock protecting a B-tree (index) or the heap data pages in a table that does not have a clustered index.
TABLE	The entire table, including all data and indexes.
FILE	A database file.
APPLICATION	An application-specified resource.
METADATA	Metadata locks.
ALLOCATION_UNIT	An allocation unit.
DATABASE	The entire database.

From MS SQL Server
https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx

Solution: New Lock Modes, Protocol

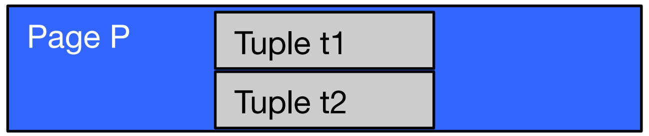
- Allow xacts to lock at each level, but with a special protocol using new “intent” locks:
- Before getting S or X lock, Xact must have proper intent locks on all its ancestors in the granularity hierarchy.



New Lock Modes – Intention Lock Modes

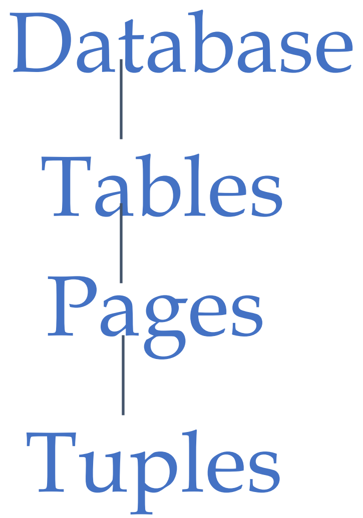
- 3 additional lock modes:
  - ***IS:*** *Intent to get S lock(s) at finer granularity.*
  - ***IX:*** *Intent to get X lock(s) at finer granularity.*
  - ***SIX:*** *Like S & IX at the same time. Why useful?*

- Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes



Multiple Granularity Locking Protocol

- Each Xact starts from the root of the hierarchy.
- To get S or IS lock on a node, must hold IS or IX on parent node.
  - What if Xact holds S on parent? SIX on parent?
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.
- Must release locks in bottom-up order.

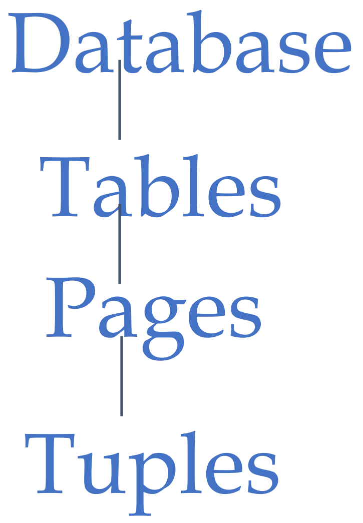


- 2-phase and lock compatibility matrix rules enforced as well
- Protocol is correct in that it is ***equivalent to directly setting locks at leaf levels of the hierarchy.***

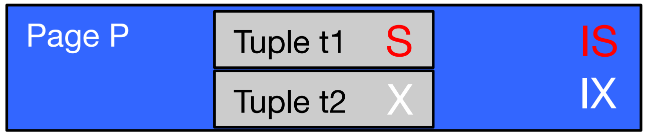
Lock Compatibility Matrix

- IS – Intent to get S lock(s) at finer granularity.
- IX – Intent to get X lock(s) at finer granularity.
- SIX mode: Like S & IX at the same time.

	IS	IX	S	SIX	X
IS					
IX					
S			true		false
SIX					
X			false		false



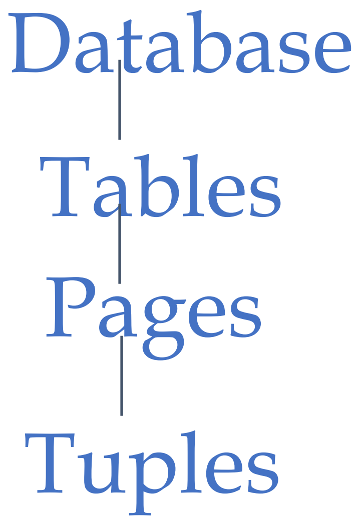
Handy simple case to remember:
Could 2 intent locks be compatible?



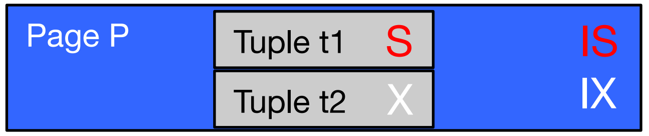
Lock Compatibility Matrix, Cont

- IS – Intent to get S lock(s) at finer granularity.
- IX – Intent to get X lock(s) at finer granularity.
- SIX mode: Like S & IX at the same time.

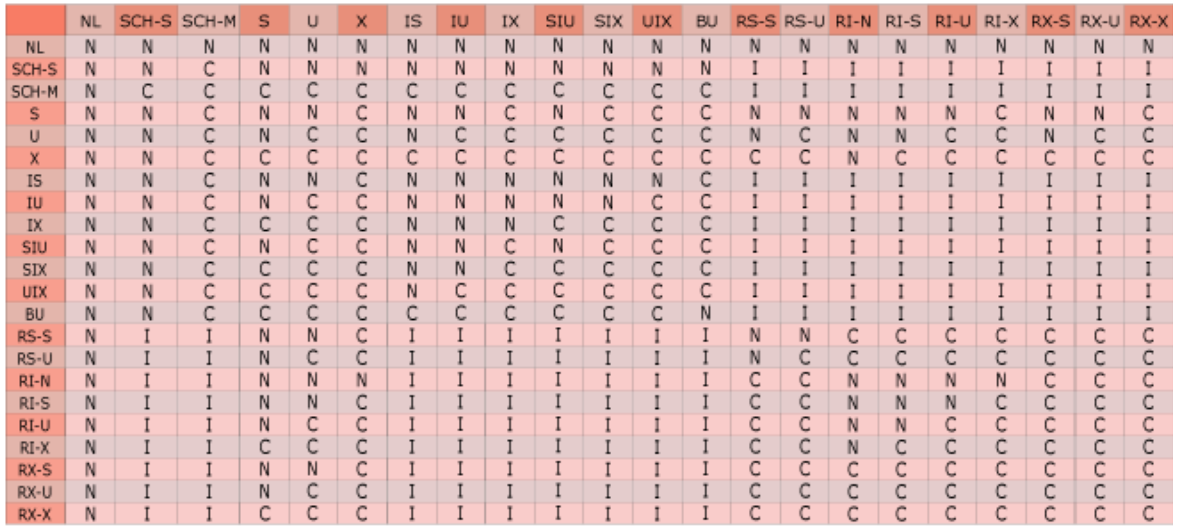
	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false



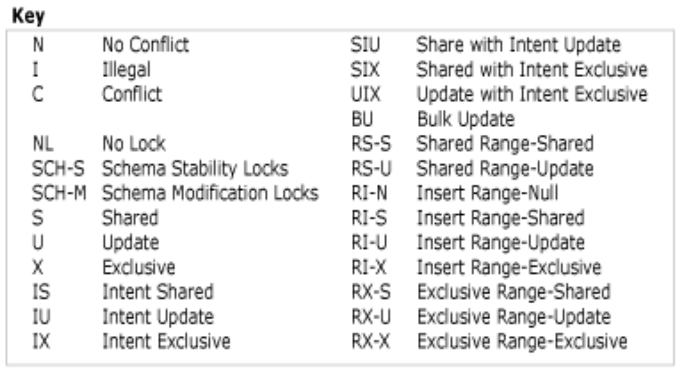
Handy simple case to remember:
Could 2 intent locks be compatible?



Real-World Lock Compatibility Matrix



	NL	SCH-S	SCH-M	S	U	X	IS	IU	IX	SIU	SIX	UIX	BU	RS-S	RS-U	RI-N	RI-S	RI-U	RI-X	RX-S	RX-U	RX-X
NL	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
SCH-S	N	N	C	N	N	N	N	N	N	N	N	N	N	I	I	I	I	I	I	I	I	I
SCH-M	N	C	C	C	C	C	C	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I
S	N	N	C	N	N	C	N	N	C	N	C	C	C	N	N	N	N	N	C	N	N	C
U	N	N	C	N	C	C	N	C	C	C	C	C	C	N	C	N	N	C	C	N	C	C
X	N	N	C	C	C	C	C	C	C	C	C	C	C	C	C	N	C	C	C	C	C	C
IS	N	N	C	N	N	C	N	N	N	N	N	N	C	I	I	I	I	I	I	I	I	I
IU	N	N	C	N	C	C	N	N	N	N	N	C	C	I	I	I	I	I	I	I	I	I
IX	N	N	C	C	C	C	N	N	N	C	C	C	C	I	I	I	I	I	I	I	I	I
SIU	N	N	C	N	C	C	N	N	C	N	C	C	C	I	I	I	I	I	I	I	I	I
SIX	N	N	C	C	C	C	N	N	C	C	C	C	C	I	I	I	I	I	I	I	I	I
UIX	N	N	C	C	C	C	N	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I
BU	N	N	C	C	C	C	C	C	C	C	C	C	N	I	I	I	I	I	I	I	I	I
RS-S	N	I	I	N	N	C	I	I	I	I	I	I	I	N	N	C	C	C	C	C	C	C
RS-U	N	I	I	N	C	C	I	I	I	I	I	I	I	N	C	C	C	C	C	C	C	C
RI-N	N	I	I	N	N	N	I	I	I	I	I	I	I	C	C	N	N	N	N	C	C	C
RI-S	N	I	I	N	N	C	I	I	I	I	I	I	I	C	C	N	N	N	C	C	C	C
RI-U	N	I	I	N	C	C	I	I	I	I	I	I	I	C	C	N	N	C	C	C	C	C
RI-X	N	I	I	C	C	C	I	I	I	I	I	I	I	C	C	N	C	C	C	C	C	C
RX-S	N	I	I	N	N	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C
RX-U	N	I	I	N	C	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C
RX-X	N	I	I	C	C	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C



Key

N	No Conflict	SIU	Share with Intent Update
I	Illegal	SIX	Shared with Intent Exclusive
C	Conflict	UIX	Update with Intent Exclusive
		BU	Bulk Update
NL	No Lock	RS-S	Shared Range-Shared
SCH-S	Schema Stability Locks	RS-U	Shared Range-Update
SCH-M	Schema Modification Locks	RI-N	Insert Range-Null
S	Shared	RI-S	Insert Range-Shared
U	Update	RI-U	Insert Range-Update
X	Exclusive	RI-X	Insert Range-Exclusive
IS	Intent Shared	RX-S	Exclusive Range-Shared
IU	Intent Update	RX-U	Exclusive Range-Update
IX	Intent Exclusive	RX-X	Exclusive Range-Exclusive

From MS SQL Server
https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx

Summary

- **Correctness criterion for isolation is “serializability”.**
  - In practice, we use “conflict serializability” which is conservative but easy to enforce
- **Two Phase Locking and Strict 2PL: Locks implement the notions of conflict directly**
  - The lock manager keeps track of the locks issued.
  - **Deadlocks** may arise; can either be prevented or detected.
- **Multi-Granularity Locking:**
  - Allows flexible tradeoff between lock “scope” in DB, and # of lock entries in lock table
- **More to the story**
  - Optimistic/Multi-version/Timestamp CC
  - Index “latching”, phantoms
  - Actually, there’s much much more :-)