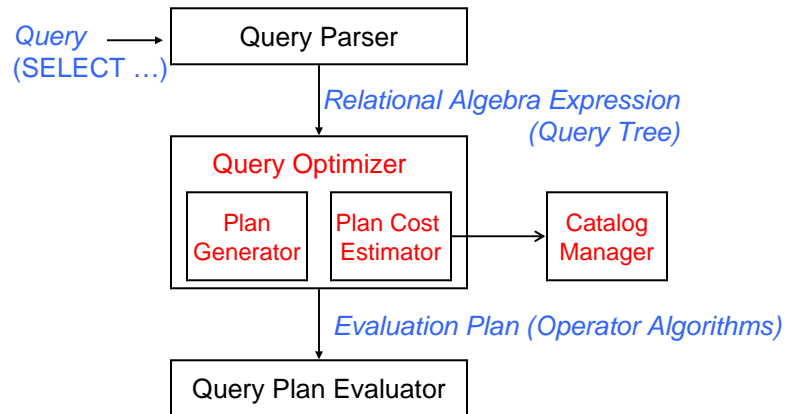


## Lecture Notes 8: Query Optimization Concurrency Control and Recovery

- Query Optimization
  - RA equivalences
  - Generating Plans
  - Costing Plans
    - Statistics
  - Enumerating Plans
  - Nested Queries

- Concurrency Control
  - Transaction
  - ACID
  - Isolation
    - Schedules
    - Guaranteeing isolation
      - Serializability
      - Conflict Serializability
      - Precedence Graph
      - Locking
      - Two Phase Locking
      - Deadlock
      - Isolation levels
- Recovery
  - ACID: Atomicity & Durability
  - Trading execution time for recovery time
  - Mechanics of recovery
    - What is recovered and how
    - Write ahead logging
    - Mechanics of logging
      - Log Record Contents
      - How to commit, abort and redo
    - Big Picture

## Query Evaluation



## Query Optimization

- **Translate SQL query into a query tree**  
(operators: relational algebra plus a few other ones)
- **Generate other, equivalent query trees**  
(e.g., using relational algebra equivalences)
- **For each possible query tree:**
  - select an algorithm for each operator (producing a query *plan*)
  - estimate the cost of the plan
- **Choose the plan with lowest cost – of the plans considered (which is not necessarily all possible plans)**

## Initial Query Tree - Equivalent to SQL (without any algorithms selected)

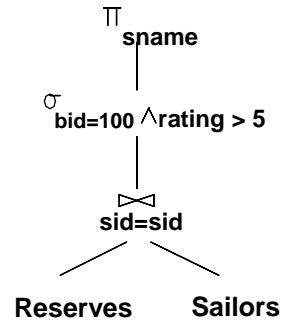
### SQL Query:

```

SELECT S.sname
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid AND
       R.bid = 100 AND
       S.rating > 5;
    
```

Sailors(sid, sname, rating, age)  
 Reserves(sid, bid, day)

### Relational Algebra Tree:



## Relational Algebra Equivalence

$$\sigma_{c_1 \wedge c_2}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(R))$$

This symbol means equivalence.

So you can replace  $\sigma_{c_1}(\sigma_{c_2}(R))$  with  $\sigma_{c_1 \wedge c_2}(R)$

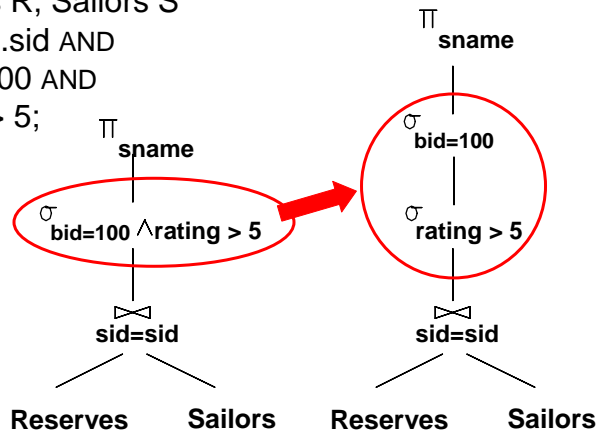
And you can replace  $\sigma_{c_1 \wedge c_2}(R)$  with  $\sigma_{c_1}(\sigma_{c_2}(R))$

If you have two conditions connected by “AND” in a select operator, then you can apply them one at a time.

Example:  $\sigma_{c_1 \wedge c_2}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(R))$

```

SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid AND
      R.bid = 100 AND
      S.rating > 5;
    
```



## Another Relational Algebra Equivalence

$$\sigma_c(R \bowtie S) \equiv R \bowtie \sigma_c(S)$$

You can apply this from left to right or from right to left.

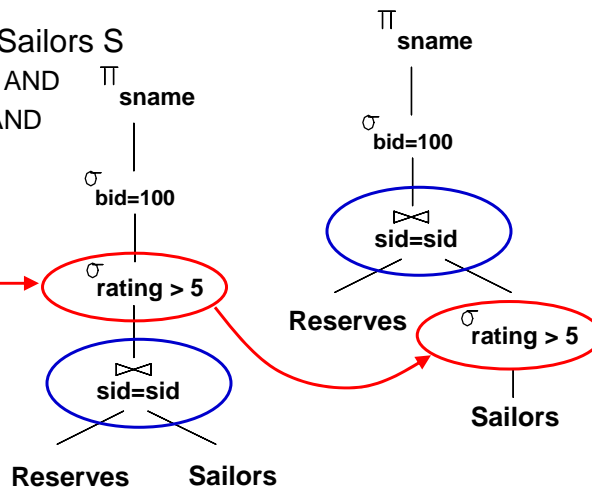
If you have a select operation following a join, and the select condition applies ONLY to one of the tables (S, in this example), then you can “push” that select operator before the join operator.

Example:  $\sigma_c(R \bowtie S) \equiv R \bowtie \sigma_c(S)$

SELECT S.sname  
 FROM Reserves R, Sailors S  
 WHERE R.sid = S.sid AND  
 R.bid = 100 AND  
 S.rating > 5;

This applies only to  
 the Sailors table!

This is called  
 "Pushing a Select" because  
 you push it past a join.

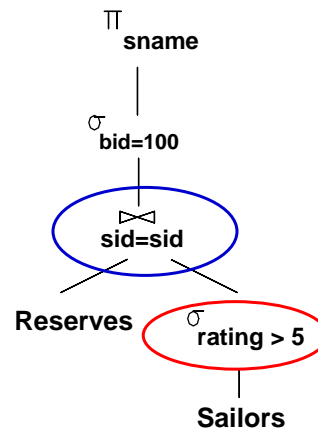


Example:  $\sigma_c(R \bowtie S) \equiv R \bowtie \sigma_c(S)$  (cont.)

SELECT S.sname  
 FROM Reserves R, Sailors S  
 WHERE R.sid = S.sid AND  
 R.bid = 100 AND  
 S.rating > 5;

What are the advantages of "pushing" a  
 select past a join operator?

What are the disadvantages of "pushing" a  
 select past a join operator?



Relational Algebra Equivalences –  
to help you rewrite SQL queries

```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid AND
      R.bid = 100 AND
      S.rating > 5;
```

Are these  
equivalent  
SQL queries?

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE R.sid = S.sid AND
      R.bid = 100 AND
      S.rating > 5;
```

$R \bowtie S \equiv S \bowtie R$  Join is commutative

```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid AND
      R.bid = 100 AND
      S.rating > 5;
```

Are these  
equivalent  
SQL queries?

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE R.sid = S.sid AND
      R.bid = 100 AND
      S.rating > 5;
```

YES! Because  
of the relational  
algebra  
equivalence shown  
above.

## Relational Algebra Equivalences

- **Selections:**

$$\sigma_{c_1 \wedge \dots \wedge c_n}(R) \equiv \sigma_{c_1}(\dots \sigma_{c_n}(R)) \quad \text{Selects Cascade}$$

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R)) \quad \text{Select Commute}$$

- **Projections:**

$$\pi_a(R) \equiv \pi_a(\pi_{a_1}(\dots \pi_{a_n}(R))) \quad \text{Only last project matters}$$

- **Joins:**

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \quad \text{Joins are Associative}$$

$$R \bowtie S \equiv S \bowtie R \quad \text{Joins Commute}$$

Try to prove that:  $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

## Some Rel. Algebra Equivalences have Constraints

$$\sigma_{c_1 \wedge c_2}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(R))$$

Is this always true? No matter what the conditions are ... no matter what table R we use?

What about this one?

$$\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$$

Is this always true? No matter what the condition is? No matter what project list is used?

## Equivalences with 2 or More Operations

- **Projection commutes with a selection**  
 PROVIDED that the selection uses attributes that are retained by the projection ( $c$ 's *attrs*  $\subseteq a$ ):  

$$\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$$
- **A cross-product can be converted to a join**  

$$\sigma_c(R \times S) \equiv R \bowtie_c S$$
- **A selection can be pushed past a cross product (or join)**  
 PROVIDED the select uses just attributes of S  

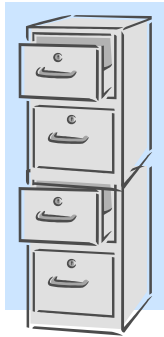
$$\sigma_c(R \bowtie S) \equiv R \bowtie \sigma_c(S)$$

## Equivalences with 2 or More Operations

- Join distributes over the various set operators (union, intersection, difference. For example, with union:  

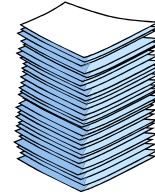
$$(Q \bowtie S) \cup (R \bowtie S) \equiv (Q \cup R) \bowtie S$$

## A Clerical Task



Cabinet with customer accounts:  
Cust. name, cust#, account balance

Stack of payment vouchers  
Cust. name, cust#, amount paid



Apply payments to account balances

## Which is Fastest?

1. Go through stack of payment vouchers one at a time. Find customer account in cabinet. Apply payment.
2. Sort payment vouchers by cust# first. Then go through vouchers, find customer account for each, apply payment.
3. Sort payment vouchers by customer name. For all payment vouchers for the same customer, add up payment amounts to get a total for each customer. Go through totals, update each customer account by total.

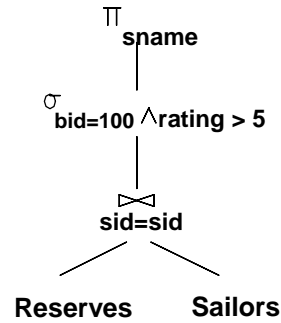
## Query Tree - Equivalent to SQL (without any algorithms selected)

**SQL Query:**

```

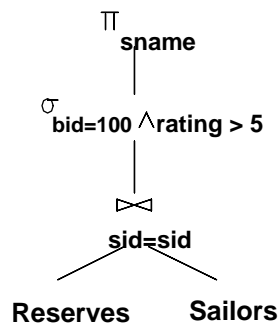
SELECT S.sname
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid AND
       R.bid = 100 AND
       S.rating > 5;
    
```

**RA Tree:**

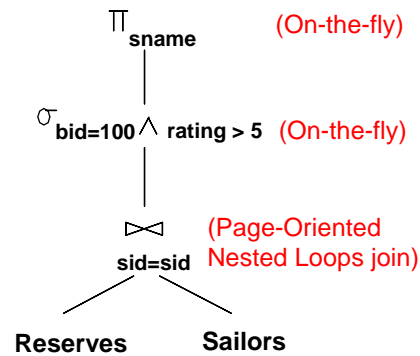


## Choosing algorithms for each operator (algorithms shown in red)

**RA Tree:**



**One Possible Plan:**



## “On the fly”

```

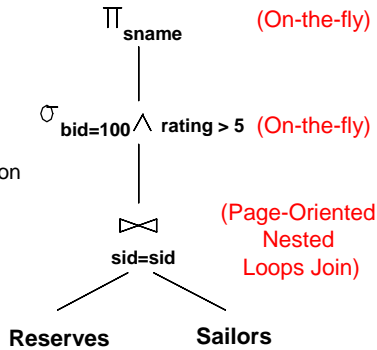
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid AND
      R.bid = 100 AND
      S.rating > 5;
    
```

On the fly means that we evaluate the operator in memory - while we have the tuple available.

“On the fly” induces no I/O cost!

Relation on left is assumed to be the outer relation - for any algorithm that uses nested loops.

Plan:



## Limitations of “On the fly”

- Can only happen if:
  - Computation can be done entirely on tuples in memory
  - Results do not need to be materialized
- Cannot apply to all operations!

**Cost of plan 1**  
no index (Sailors – inner loop)

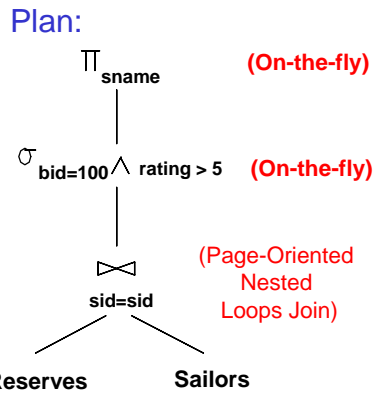
```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid AND
      R.bid = 100 AND
      S.rating > 5;
```

M = # of pages in outer table  
N = # of pages in inner table

Cost of page-oriented nested loops join is:

$M + M * N$   
 $1000 + 1000 * 500 = 501,000$

And the “on-the-fly” operations have no I/O - so plan cost is 501,000



**Cost of plan 2**  
no index (Reserves: inner loop)

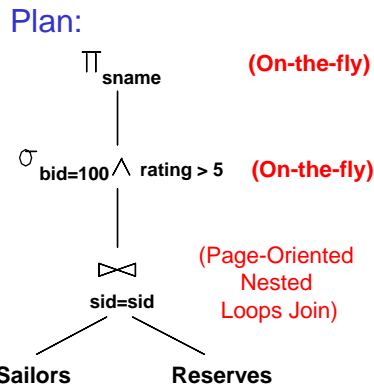
```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid AND
      R.bid = 100 AND
      S.rating > 5;
```

Sailors as the outer relation rather than Reserves.

M = # of pages in outer table  
N = # of pages in inner table

Cost of page-oriented nested loops join is:

$500 + 500 * 1000 = 500,500$   
And the “on-the-fly” operations have no I/O - so plan cost is 500,500



### Cost of plan 3 Push down selects

Apply this equivalence:

$$\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$$

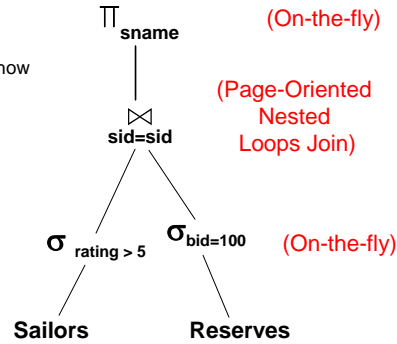
To the previous query tree to get an equivalent query tree.

```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid AND
      R.bid = 100 AND
      S.rating > 5;
```

Plan:

What is the cost of the plan shown?

Scanning sailors and reserves cost M+N I/Os.  
 What about the cost of the join? It depends on how many reservations are for boat 100 and how many sailors have a rating >5.  
 How would you find this information?  
 Statistics will help.



### To estimate cost, we need table sizes

- For all operators beyond the leaf level of the query plan, the input tables are the result of some earlier query.
- Thus, **we need to estimate the size of intermediate results!**

(This can be difficult. This is one reason why the cost estimates may not be very good. Estimation errors tend to compound.)

## DBMS Usually Maintains Some Statistics in the DB Catalog

- *Catalogs* typically contain at least:
  - # tuples and # pages for each table.
  - # distinct key values and # pages for each index.
  - Index height, low/high key values for each tree index.
- Catalogs are updated periodically - say, once a week or once a month. Perhaps they're updated during the backup.
- Simplest case: assume that all attribute values are uniformly distributed. Thus if gender was an attribute, the optimizer would assume that half of the rows had the male value and half had the female value. (This might be grossly inaccurate.)

## Calculating Selectivities

- Assume that rating values range from 1 to 10, and that bid values range from 1 to 100.
- What percentage of the incoming tuples, to the operator  $\sigma_{bid=100}$ , will be output?
- What about  $\sigma_{rating > 5}$  ?

## Doing better than a uniform distribution:

- The DBMS might gather more detailed information about how the values of attributes are distributed (e.g., **histograms** of the values in a field) and store it in the catalog.

Suppose there was an attribute **degree-program** with three possible values: "BS CS" "MS CS" "PhD CS"

- Then the DBMS might count the values and know that there are 428 "BS CS" values, 98 "MS CS" values and 25 "PhD CS" values.
- This allows much better estimate of the reduction factor.

## Independence of Reduction Factors

- We **assume** that all **terms are independent!**

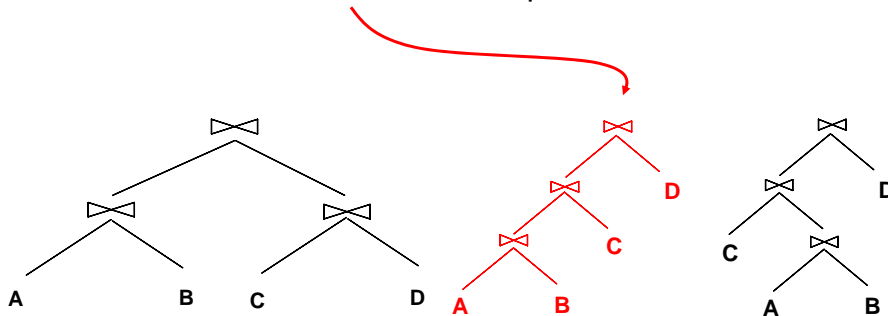
Thus, if one attribute is **class** and the other is **number-of-hours**, the query optimizer might assume that class is uniformly distributed over {Fresh, Soph, Jun, Sen} and that number-of-hours is uniformly distributed over {0, 1, ..., 205}

[But, we know that only class correlates with number-of-hours! Might even be that number-of-hours determines which class you are in.]

## Enumerating Plans for Multiple Joins

- In System R: **only left-deep join trees** are considered.

This one is left-deep - the other two are not.



CS386 Some slides copied from R. Ramakrishnan,  
L. Delcambre, L. Shapiro, D. Maier © 2006

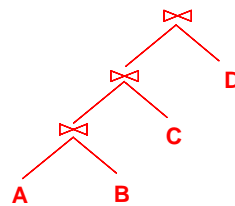
5/23/2006

Slide 31  
Lecture 8

## Queries Over Multiple Relations (Joins)

**Left-deep** trees allow us to generate all **fully pipelined plans**.

- Intermediate results not written to temporary files.
- Not all left-deep trees are fully pipelined (e.g., SM join).
- Using only left-deep plans (obviously) restricts the search space. (So optimizer may not find the optimal plan.)



CS386 Some slides copied from R. Ramakrishnan,  
L. Delcambre, L. Shapiro, D. Maier © 2006

5/23/2006

Slide 32  
Lecture 8

## Enumeration of Left-Deep Plans

- Need to consider all possible order for the relations.
- For each order, consider the access method for each relation and the join method for each join.
- Enumerated using N passes (if N relations joined):
  - **Pass 1:** Find best 1-relation plan for each relation.
  - **Pass 2:** Find best way to join result of each 1-relation plan (as outer) to another relation. (All 2-relation plans.)
  - **Pass N:** Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation. (All N-relation plans.)
- At the end of each pass, retain only:
  - Cheapest plan overall, plus
  - Cheapest plan for each **interesting order** of the tuples.

## Enumeration of Plans (Contd.)

- **ORDER BY, GROUP BY, aggregates** etc. handled as a **final** step, using either an “interestingly ordered” plan or an additional sorting operator.
- An N-1 way plan is not combined with an additional relation unless there is a join condition between them, that is unless all predicates in WHERE have been used up.
  - i.e., **avoid Cartesian products if possible.**
- In spite of the fact that we are pruning the plan space, this approach is **still exponential** in the # of tables (n!).

## Nested Queries

- **Nested block** is optimized **independently**, with the outer tuple considered as providing a selection condition.
- **Outer block** is optimized with the **cost of 'calling' the nested block** computation taken into account.
- Implicit ordering of these blocks means that some good strategies are not considered.
- The **non-nested version** of the query is typically **optimized better**. Optimizer might not find it from nested version, so may need to explicitly unnest the query.

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS
  (SELECT *
   FROM Reserves R
   WHERE R.bid=103
   AND R.sid=S.sid)
```

Nested block to optimize:

```
SELECT *
FROM Reserves R
WHERE R.bid=103
AND S.sid= outer value
```

Equivalent non-nested query:

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
AND R.bid=103
```

## Query Optimizers Don't (Always) Find the Best Plan

- There are usually more plans than you can consider.
- The optimizer might not even try to generate all possible plans (it won't be able to consider all of them anyway).
- Sometimes the optimizer will compare the optimization cost to the estimated execution cost

## Concurrency Control and Recovery

---

CS386 Some slides copied from R. Ramakrishnan,  
L. Delcambre, L. Shapiro, D. Maier © 2006

5/23/2006

Lecture 8

Slide 37

### Why Have Concurrent Processing (from multiple users) against a Database?

- **Higher throughput**, lower response times
- **Better utilization of resources**: While one processes is doing a disk read, another can be using the CPU or reading another disk.
- **But...** concurrent processing can lead to incorrect data in the database or incorrect data returned to users!

---

CS386 Some slides copied from R. Ramakrishnan,  
L. Delcambre, L. Shapiro, D. Maier © 2006

5/23/2006

Lecture 8

Slide 38

## Transaction

A transaction is:

- one “complete” set of actions
- **defined** by the **user** (meaningful to the user)
- establishes where certain **integrity constraints are enforced**.

For concurrency control purposes (inside DBMS):

- a **transaction** is one atomic unit of work.
  - Thus we must be able to redo and undo it
- DBMS cares only about the reads/writes to the DB
- DBMS views a transaction as (only) a sequence of **reads, writes plus commit & abort** (ignoring the rest of the program)

## Example Transaction

**Transfer \$100 from one account to another:**

- read balance in first account
- check balance to see if there's enough money
- read balance in second account
- subtract \$100 from first account balance
- add \$100 to second account balance
- update first account record
- update second account record
- print receipt

## Example Transaction

Transfer \$100 from one account to another:

- read first account
- check balance to see if there's enough
- read second account
- subtract 100 from first account balance
- add 100 to second account balance
- update first account balance
- update second account balance
- print receipt

**all or nothing!**

## Transaction (cont.)

User (application developer) must indicate:

- **Begin transaction**
- **read/write/modify statements** intermixed with other programming language statements

plus either

- **commit** - indicates successful completion **or**
- **abort** - indicates program wants to roll back (erase the transaction)

## The ACID Properties of Transactions

- **Atomicity:** All actions in a transaction happen in their entirety or not at all.
- **Consistency:** If the DB starts in a consistent state, (this notion is defined by the user; some of it may be enforced by integrity constraints) and a transaction executes with no other queries active, then the DB ends up in a consistent state.
- **Isolation:** Each transaction is isolated from other transactions. The effect on the DB is (as if) the transaction executed by itself.
- **Durability:** If a transaction commits, its changes to the database state persist (changes are permanent).

## Supporting the ACID Properties of Transactions

- **Atomicity:** All actions in a transaction happen in their entirety or not at all.
  - **Consistency:** If the DB starts in a consistent state, (this notion is defined by the user; some of it may be enforced by integrity constraints) and a transaction executes with no other queries active, then the DB ends up in a consistent state.
  - **Isolation:** The transaction is isolated from other transactions. The effect on the DB is (as if) the transaction executed by itself.
  - **Durability:** If a transaction commits, its changes to the database state persist.
- Recovery System*
- Programmers*
- Concurrency Control System*
- Recovery System*

## Example re Isolation

- Consider two transactions:

```
T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END
```

- Intuitively, the first transaction is transferring \$100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.

## Which Transactions are Isolated?

	T2 then T1	T1 then T2	T1, T2 mixed																								
Initial values → for A, B	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td>A</td><td>B</td></tr> <tr><td>1000</td><td>500</td></tr> <tr><td>1060</td><td>530</td></tr> <tr><td>1160</td><td>430</td></tr> </table>	A	B	1000	500	1060	530	1160	430	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td>A</td><td>B</td></tr> <tr><td>1000</td><td>500</td></tr> <tr><td>1100</td><td>400</td></tr> <tr><td>1166</td><td>424</td></tr> </table>	A	B	1000	500	1100	400	1166	424	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td>A</td><td>B</td></tr> <tr><td>1000</td><td>500</td></tr> <tr><td>1100</td><td>530</td></tr> <tr><td>1166</td><td>430</td></tr> </table>	A	B	1000	500	1100	530	1166	430
A	B																										
1000	500																										
1060	530																										
1160	430																										
A	B																										
1000	500																										
1100	400																										
1166	424																										
A	B																										
1000	500																										
1100	530																										
1166	430																										
	<p style="color: magenta;">These are isolated</p>		<p style="color: gray;">Illegal values! (NOT Isolated)</p>																								

```
T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END
```

## Interleaved Schedules:

Consider this interleaved *schedule*:

T1:	A=A+100,	B=B-100
T2:	A=1.06*A,	B=1.06*B

equivalent to T1 then T2 on prior page. This is OK.

Consider another interleaved *schedule*:

T1:	A=A+100,	B=B-100
T2:	A=1.06*A, B=1.06*B	

equivalent to third schedule on prior page. This is illegal.

## Schedules

- **Schedule**: An interleaving of actions from a set of transactions, where the actions of any 1 transaction are in the original order.
  - Represents an actual sequence of database actions.
  - Example:  $R_1(A), W_1(A), R_2(B), W_2(B), R_1(C), W_1(C)$
  - In a *complete* schedule, each transaction ends in *commit* or *abort*.
- Initial State + Schedule → Final State

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

## Serializable Schedule $\Leftrightarrow$ Isolated Transactions

- **Serial** schedules:
  - Run transactions one at a time, in a series.
- **Serializable** schedules:
  - Final state must be the same as the state produced by **one of the serial** schedules.
  - Must look to each transaction like the transactions that precede it ran sequentially
  - (ignore aborted transactions for now — they are made to “disappear” by using the recovery subsystem).
- **Serializable  $\Leftrightarrow$  Isolated**
  - $\Rightarrow$ : Serializable  $\Rightarrow$  equivalent to some serial schedule, and in a serial schedule, each Xact. is isolated
  - $\Leftarrow$ : If each xact runs alone, the schedule must be serial

## Examples on Serializable Schedules

Which of these schedules is serializable?

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

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

## Conflicting Actions

Two actions from different transactions *conflict* if at least one of them is a write and they access the same object

- R(A), W(A): conflict
- W(B), W(B): conflict
- R(A), R(A): two reads, no conflict
- R(A), W(B): different object, no conflict

## Conflict Serializable Schedules

- **Commuting actions:**
  - In different transactions, operate on different objects, or both are reads.
  - If actions commute then we can interchange them and get an equivalent schedule
- Two schedules S1 and S2 are **conflict equivalent** if:
  - There is a sequence of interchanges of commuting actions that transforms S1 to S2.
- Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule
  - i.e., if there is a sequence of interchanges of commuting actions that transforms S to a serial schedule

## Which are conflict serializable?

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

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

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

## Conflict Serializable $\Rightarrow$ Serializable

- Every conflict serializable schedule S is serializable.

### Proof:

- We can interchange nonconflicting actions to transform S to a serial schedule R.
- The interchanges produce equivalent schedules
- Therefore the serial schedule R is equivalent to S.
- Thus S is serializable.

- Not true in the other direction:

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

## Precedence Graph

- **Precedence graph**: One node per  $X_{act}$ ; edge from  $T_i$  to  $T_j$  if an action in  $T_i$  occurs before an action in  $T_j$  and they conflict.
- **Theorem**: Schedule is conflict serializable if and only if its dependency graph is **acyclic**
- Draw the precedence graph for the previous 3 schedules.

## Is a schedule $S$ serializable or not?

- How to prove that  $S$  is serializable
  - Based on the definition of serializable, we can find a serial schedule that is equivalent to it.
  - We can also show that its precedence graph is acyclic.
    - This proves that it is conflict serializable, therefore serializable.
- How to prove that  $S$  is not serializable
  - Enumerate all serial schedules and show that  $S$  is not equivalent to any of them - impractical

## Locking: Used in Concurrency Control

- **Transaction must get a lock** – before then can read or update data
- There are two kinds of locks:  
**shared (S) locks** and **exclusive (X) locks**
- To read a record you **MUST** get an S lock  
To modify or delete a record you **MUST** get an X lock
- Lock info maintained by a “lock manager”

## How Locks Work

- If an object has an **S** lock, new transactions can get **S** locks but not **X** locks.
- If an object has an **X** lock, no other transaction can get any lock (**S** or **X**) on that object.
- If a transaction can't get a lock, it **waits** (in a **queue**).

lock on data item

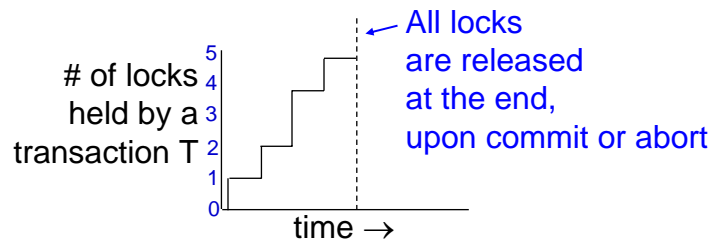
	--	S	X
--	ok	ok	ok
S	ok	ok	no
X	ok	no	no

Lock compatibility

## Strict Two Phase Locking Protocol (2PL)

Strict 2PL is a way of managing locks during a transaction

- T gets (S and X) locks gradually, as needed
- T hold all locks until end of transaction (commit/abort)



## Try to enforce 2PL on each schedule

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

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

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

## Strict 2PL guarantees serializability

- Can prove that a Strict 2PL schedule is equivalent to the serial schedule in which each transaction runs instantaneously at the time that it commits
- This is **huge**: A property of each transaction (2PL) implies a property of any set of transactions (serializability)
  - No need to check serializability of any schedules
- Real DBMSs use 2PL to enforce serializability

## Deadlock

Moira: "I'll wait for you to call and then we'll go eat."

Tom (not paying much attention): "OK, I'll wait for you to call and then we'll go eat."

Moira (also not paying much attention): "Fine."

DEADLOCK! They'll never eat.

## Deadlock in DBMSs

- What is a deadlock?
  - A *cycle* of transactions,  $T_1, T_2, \dots, T_n=T_1$  where each  $T_i$  is *waiting* for  $T_{i-1}$  to release a lock.
  - Causes these transactions to sleep forever.
- A **Deadlock** can happen whenever you allow a transaction to **wait for a lock**.
  - Even with strict 2PL, can have deadlocks
- How can you prevent deadlock?
  - Avoid by careful use of resources
    - E.g. lock objects in a fixed order
  - Detect, then abort one transaction
    - look for a cycle in waitfor graph, perhaps using a timeout

## 3 problems with concurrent transactions

- The motivation here is to design environments that are less strict than serializability.
- **Write-Write** conflicts
  - 3<sup>rd</sup> example, or any write over uncommitted data.
- **Read-Write** conflicts (**unrepeatable reads**)
  - You read a value
  - Then some other transaction modifies it
  - Then you read it again (IT'S DIFFERENT!)
- **Write-Read** conflicts (**dirty reads**) –
  - Some transaction modifies data
  - Then you read it
  - Then the other transaction aborts – and their modifications to the database are **UNDONE**
  - You have read a tuple value that never “officially” existed.

## PHANTOMS!!!

- Gasp!
- Occurs when different reads of a collection see two different sets of objects
  - New rows show up, despite the current transaction not making any changes
- Here, conflict-serializability does not guarantee serializability

Bob	5	True
Ted	7	True
Sally	4	False
Pratibha	6	True

Bob	5	True
Ted	7	True
Sally	4	False
Pratibha	6	True
Andrew	5	True

## Isolation levels (DBMS's may differ...)

Developers can choose how much isolation (protection) they want ... There are four isolation levels defined in SQL:

- **“READ UNCOMMITTED”** – allows dirty read, unrepeatable read, and “phantoms”
- **“READ COMMITTED”** – allows unrepeatable reads and phantoms
- **“REPEATABLE READ”** – allows phantoms
- **“SERIALIZABLE”** – full isolation

*typical* →

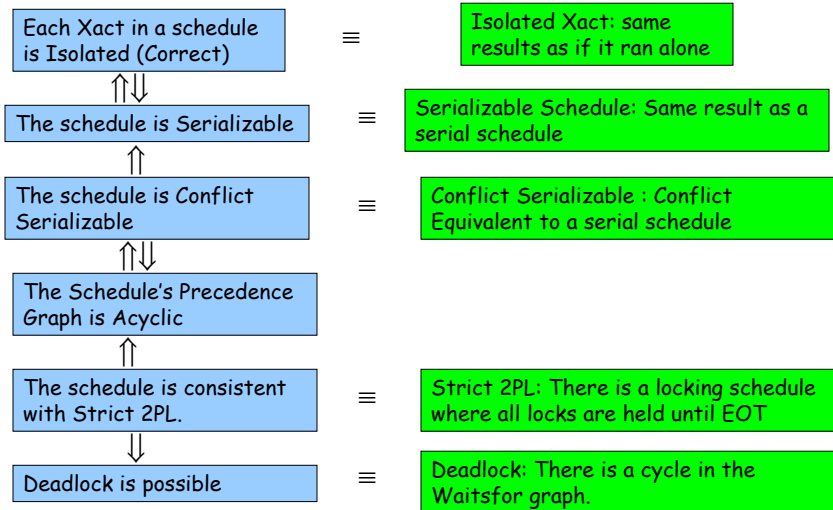
## Isolation levels: How they work

- **READ UNCOMMITTED**
  - Does not bother with locks when reading
- **READ COMMITTED**
  - Obtains write locks, releases read locks immediately
- **REPEATABLE READ**
  - Locks only individual objects
- **SERIALIZABLE**
  - Locks sets of objects (tables, pages)

## What should we lock?

- Lock **tables** – not very much concurrency – lock table is small
- **typical** → Lock **rows** – much more concurrency – lock table is much larger
- Lock **attribute values** – even more concurrency – much larger lock table – difficult to implement
- Lock a **predicate** ... e.g., lock all students with age > 25 (locks students who appear later – avoids “phantoms”); **hard to implement**

## Summary



## Review: The ACID properties

- Recovery System**
- Atomicity:** All actions in the transaction happen in their entirety or none of them happen.
- Consistency:** If each transaction is consistent, and the DB starts in a consistent state, it ends in a consistent state.
- Programmers**
- Concurrency Control System**
- Isolation:** Execution of one transaction is isolated from that of other transactions.
- Durability:** If a transaction commits, its effects persist.
- Recovery System**

## Trading execution time for recovery time

- **Force:** writes to disk immediately on commit
  - Why do you have to warn your OS before removing an external device? Because of the OS's No Force Policy.
  - Force gives poor response time. But it implies durability.
  - How can we have durability without force?
- **Steal** buffer-pool frames from uncommitted transactions?
  - If DBMS doesn't steal
    - Poor throughput.
    - Atomicity is easy
  - How can we have atomicity with Steal?

	No Steal	Steal
Force	Slow Execution; Easy recovery	
No Force		Fast Execution; Tough Recovery

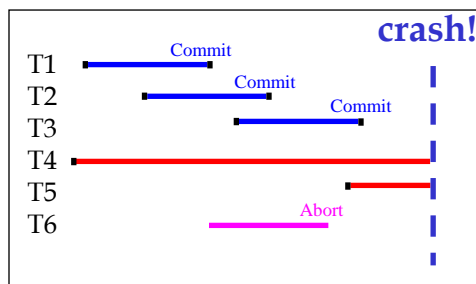
## Consequences of *No Force*

- Dirty pages may remain in memory **after commit.**
  - What if we lose memory because of a crash?
  - Updates in the dirty pages are lost. This could **kill durability.**
- **Example:**
  - Xact withdrawing and depositing \$100 runs and commits
  - The page containing the withdrawal is forced/written to disk.
  - The page containing the deposit is **not** forced to disk.
  - The page containing the deposit of \$100 stays in memory after commit.
  - Crash.
  - **Have you lost \$100?**
  - **Would it help if the withdrawal info had been written to a log?**

## Consequences of *Steal*

- Dirty pages may be written to disk **before commit**.
  - What if the transaction is aborted?
  - Changes written to disk can't be reversed. This could **kill atomicity**.
- **Example:**
  - Xact deposits \$100
  - Page with deposit info is written to disk
  - Xact aborts
  - **Have you gained \$100?**
  - **Would it help if the deposit had been logged?**

## Which Transactions are Recovered and How?

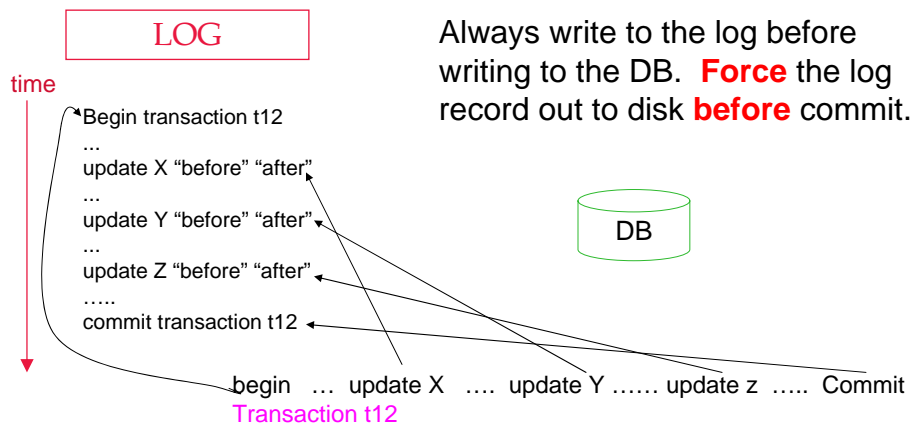


- Desired Behavior after system restarts:
  - **T1, T2 & T3** committed and must be **durable**
    - Ensure durability by **redoing** each Xact using the **log**.
  - **T4 & T5** were still running and must be **atomic**.
    - Ensure this by **undoing** the Xact using the **log**.
  - **T6** was aborted before the crash and must be atomic
    - Ensure this by **undoing** the Xact using the **log**.

## Assumptions

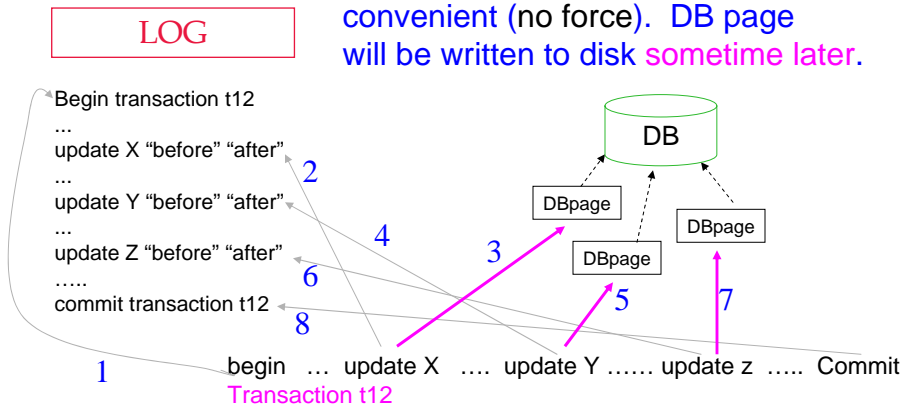
- Concurrency control is in effect - **Strict 2PL**
- Updates are happening “**in place**”. That is, data is overwritten on (or deleted from) the disk.

## Write Ahead Logging



## Write Ahead Logging

After each log record is written, write  
The change to the DB page when  
convenient (no force). DB page  
will be written to disk **sometime later**.



CS386 Some slides copied from R. Ramakrishnan,  
L. Delcambre, L. Shapiro, D. Maier © 2006

5/23/2006

Lecture 8

Slide 77

## Logging mechanics (simplified)

- Put the log on a separate disk.
- Write a log entry for **insert**, **update**, **delete**, **begin-trans**, **commit**, **abort**, and **checkpoint**.
- Log record contains:

<XID, pageID, offset, length, old data, new data>

identification of  
the record

before  
image      after  
image

CS386 Some slides copied from R. Ramakrishnan,  
L. Delcambre, L. Shapiro, D. Maier © 2006

5/23/2006

Lecture 8

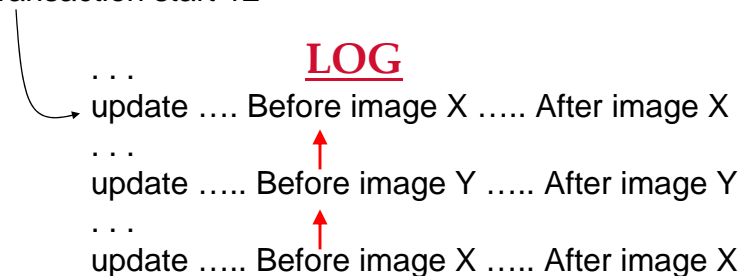
Slide 78

## Transaction Commit (simple model)

- Push all log records for this transaction to disk. Write commit record to log.

## Transaction Abort (UNDO) (simple model)

Transaction start 12



We must reapply BEFORE images in reverse order  
How far back do we need to go?

### Redo (simple model)

Transaction start 12

```

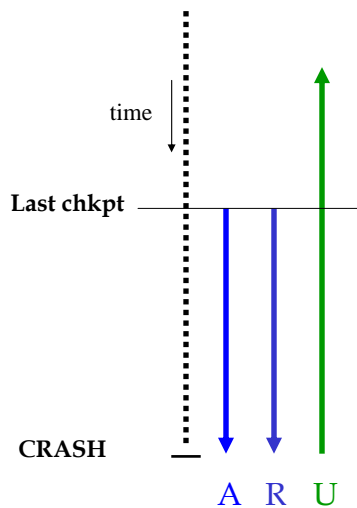
...
update .... Before image X ..... After image X
...
update ..... Before image Y ..... After image Y
...
update ..... Before image Z ..... After image Z
...
Commit
...
end
    
```

We must reapply AFTER images in forward order



How far back do we have to go to start the redo? How many transactions do we need to redo?

### Crash Recovery: Big Picture (simple model)

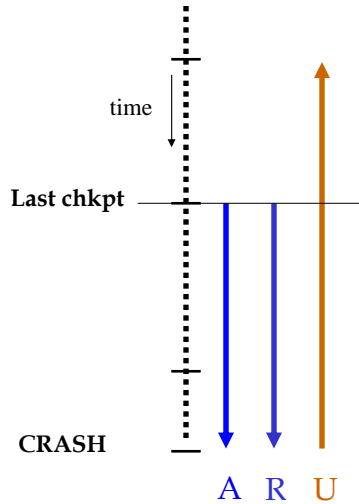


Start from a **checkpoint**.

Three phases. Need to:

- Figure out which transactions committed and which failed since checkpoint (**Analysis**).
- **REDO** actions since the checkpoint for all transactions committed since last checkpoint.
- **UNDO** all actions of failed and incomplete transactions.

## Crash Recovery: Big Picture (simple model)



- **Analysis:** find transactions that must be **REDO** and **UNDO**
  - (read the log backwards to last checkpoint record, make 2 lists)
- **REDO** transactions  
(read log forwards from checkpoint, apply "after images" in forward order)
- **UNDO** transactions.  
(read log backwards, apply "before images" in reverse order; stop at checkpoint if quiesce point was used at the checkpoint)