Cloud Data Management: Relational Databases

(Data model, algebra, query processing, transactions)

Lecture 5

Role of a database system

- Database: **integrated, shared** data collection
- Integrated
  - Eliminate needless redundancy
  - Maintain strong consistency
- Shared
  - Application written by <u>programmers</u> in multiple languages
  - <u>End-users</u> who use applications, forms, CLI to interact
- Database systems shield users from
  - How data is stored (bits & bytes, 1 vs N files, 1 vs N disks…)
  - How data is accessed (btree, hashtable, scan, …)

What is a data model?

- Collection of application-visible constructs
  - Describe data in application & storage agnostic way
- Constructs to describe structural aspects
  - How do applications perceive the data?
  - Ex: table, graph, associative array…
- Constructs to describe manipulation aspects
  - What operators can applications use?
  - Ex: join, traverse, lookup…
- Constructs to describe data integrity aspects
  - How do we ensure that data manipulation is “correct”?

Relational Model: Structural aspect

- Database = set of named **relations** (or **tables**)
- Each relation has a set of named **attributes** (or **columns**)
- Each **tuple** (or **row**) has a value for each attribute
- Each attribute has a **type** (or **domain**)
  - integer, real, string, file formats (jpeg,…), enumerated and many more

Students

sid	name	login	age	gpa
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smit@ee	18	3.2
…	…	…	…	…

Colleges

name	location	strength
MIT	USA	10000
Oxford	UK	22000
EPFL	CH	9000
…	…	…

Relational Model: Structural aspect

- **Relation Schema**: relation name + field names + field domains
  - Students(*sid*: string, *name*: string, *login*: string, *age*: integer, *gpa*: real)
- **Relation Instance**: contents at a given point in time
  - *set* of rows or *tuples*. (all rows are distinct with no specific ordering)
  - Cardinality: # rows, Arity or degree: # attributes
- **Database Schema**: collection of relation schemas
- **Database Instance**: collection of relation instances

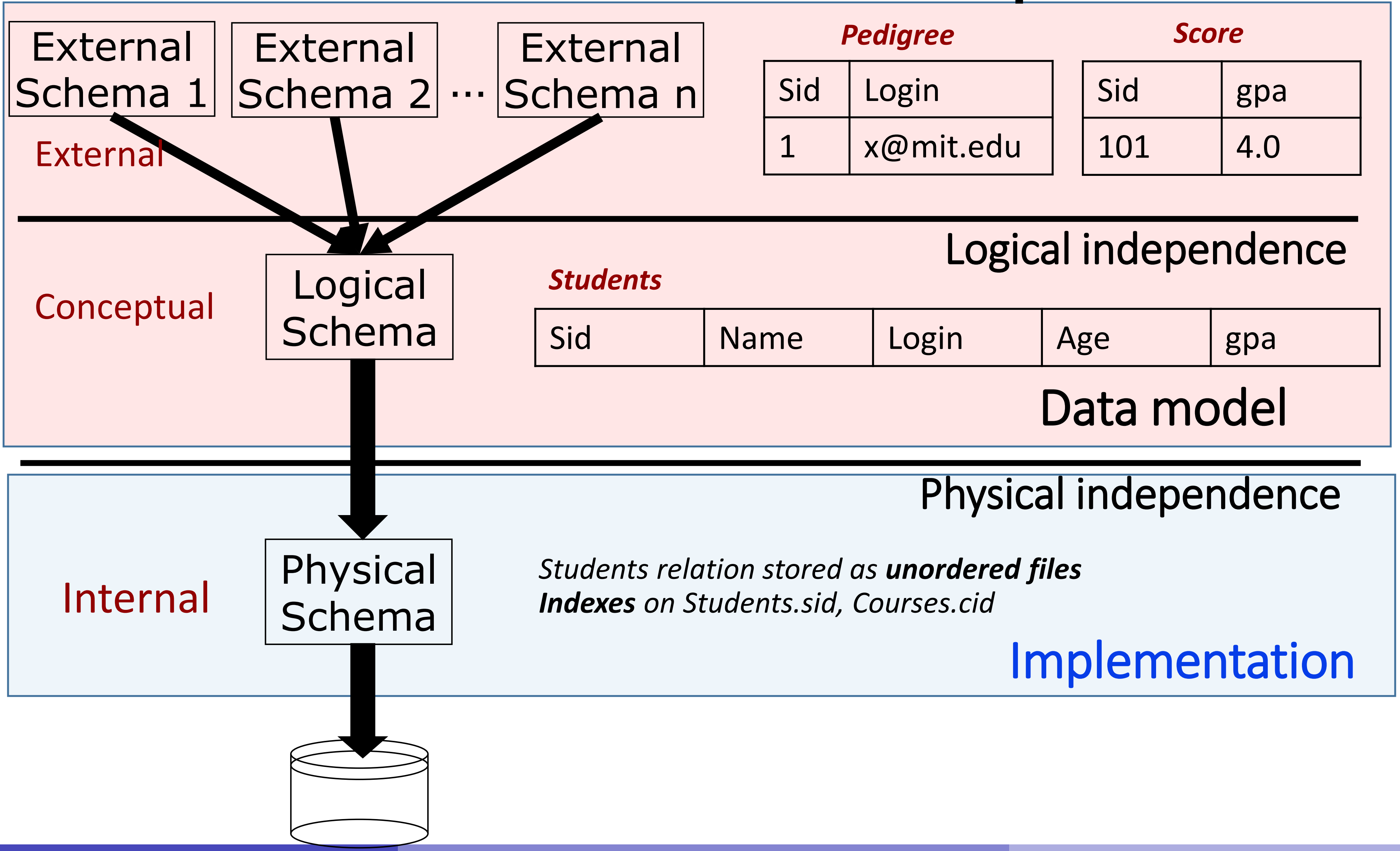
Students

sid	name	login	age	gpa
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smit@ee	18	3.2
…	…	…	…	…

Colleges

name	location	strength
MIT	USA	10000
Oxford	UK	22000
EPFL	CH	9000
…	…	…

Relational model & data independence



Pedigree

Sid	Login
1	x@mit.edu

Score

Sid	gpa
101	4.0

Students

Sid	Name	Login	Age	gpa

Relational Model: Integrity Aspect

- Relational model provides **Integrity Constraints**
  - condition specified on schema that restricts the data that can be stored in ***any*** instance
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.
- A ***legal*** instance of a relation is one that satisfies all specified ICs
  - DBMS should not allow illegal instances.
- With ICs, stored data is more faithful to real-world meaning
  - Avoids data entry errors, too!

Relational Model: **Keys**

- Attribute whose value is unique in each tuple
- Or set of attributes whose combined values are unique
- Keys specify **key constraint**
  - Enforced when tuples are inserted/updated

Students

sid	name	login	age	gpa
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smit@ee	18	3.2
…	…	…	…	…

Colleges

name	location	strength
MIT	USA	10000
Oxford	UK	22000
EPFL	CH	9000
…	…	…

Relational Model: **Foreign Keys**

- Set of fields in one relation that `refer' to a tuple in another relation (like a pointer)
- Foreign keys specify **Foreign Key Constraint**
  - FK must correspond to the primary key of the other relation
- If all foreign key constraints are enforced, **referential integrity** is achieved (i.e., no dangling references.)

Students

sid	name	login	age	gpa
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smit@ee	18	3.2
…	…	…	…	…

Enrolled

cid	sid	grade
Carnatic101	53666	C
Raggae203	50000	B
Topology112	53666	A
…	…	…

Relational Model: Manipulation Aspect

- **Query languages**: Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages != Programming Languages!
  - QLs not expected to be "Turing complete".
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.

Formal Relational Query Languages

Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:

Relational Algebra: More operational, very useful for representing execution plans.

Relational Calculus: Lets users describe what they want, rather than how to compute it. (Non-procedural, *declarative*.)

- *Understanding Algebra & Calculus is key to understanding SQL, query processing!*

Preliminaries

- A query is applied to *relation instances*, and the result of a query is also a relation instance.
  - *Schemas* of input relations for a query are fixed (but query will run over any legal instance)
  - The schema for the *result* of a given query is also fixed. It is determined by the definitions of the query language constructs.

Example Schema and Instances

- `Boats(`<u>*bid*`: integer`</u>`, `*bname*`: string, `*color*`: string)`
- `Sailors(`<u>*sid*`: integer`</u>`, `*sname*`: string, `*rating*`: integer, `*age*`: real)`
- `Reserves(`<u>*sid*`: integer, `*bid*`: integer, `*day*`:date`</u>`)`

Boats

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

Relational Algebra: 5 Basic Operations

- **Selection** (σ) Selects a subset of *rows* from relation (horizontal).
- **Projection** (π) Retains only wanted *columns* from relation (vertical).
- **Cross-product** ($\times$) Allows us to combine two relations.
- **Set-difference** ($-$) Tuples in r1, but not in r2.
- **Union** ($\cup$) Tuples in r1 and/or in r2.

Since each operation returns a relation, operations can be *composed!* (Algebra is "closed").

Selection Operator: (σ)

- Selects rows that satisfy *selection condition*.
- ***Output schema*** of result is same as that of the input relation

S2

sid	sname	rating	age
~~28~~	~~yuppy~~	~~9~~	~~35.0~~
31	Lubber	8	55.5
44	guppy	5	35.0
~~58~~	~~Rusty~~	~~10~~	~~35.0~~

$$\sigma_{rating<9}(S2)$$

Output

sid	sname	rating	age
31	Lubber	8	55.5
44	guppy	5	35.0

S2

sid	sname	rating	age
~~28~~	~~yuppy~~	~~9~~	~~35.0~~
31	Lubber	8	55.5
~~44~~	~~guppy~~	~~5~~	~~35.0~~
~~58~~	~~Rusty~~	~~10~~	~~35.0~~

$$\sigma_{\substack{rating<9\wedge \\ age>50}}(S2)$$

Output

sid	sname	rating	age
31	Lubber	8	55.5

Projection Operator (π)

- Retains only attributes that are in the *projection list.*
- **Output schema** is exactly the fields in the projection list, with the same names that they had in the input relation.

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

$$\pi_{sname,rating}(S2)$$

Output

sname	rating
yuppy	9
Lubber	8
guppy	5
Rusty	10

Projection Operator (π): Duplicate Elimination

- Relational algebra is set based while SQL is bag (multiset) based
- Projection operator *eliminates duplicates*

$S2$

sid	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

$$\pi_{age}(S2)$$

$Output$

age
35.0
55.5

Composing multiple operators

- Output of one operator can become input to another operator

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

Output

sname	rating
yuppy	9
Rusty	10

$$\pi_{sname,rating}\left(\sigma_{rating>8}(S2)\right)$$

Union and Set-Difference

- All of these operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - “Corresponding” fields have the same type.

Union operator (U)

$S1$

<u>sid</u>	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

$S2$

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

$S1 \cup S2$

<u>sid</u>	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

Set Difference Operator (–)

$S1$

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

$S2$

sid	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

$S1 - S2$

sid	sname	rating	age
22	Dustin	7	45.0

$S2 - S1$

sid	sname	rating	age
28	yuppy	9	35.0
44	guppy	5	35.0

Compound Operator: Intersection

- Alongside the 5 basic operators, there are several additional **Compound Operators**:
  - These add no computational power to the language, but are useful shorthands.
  - Can be expressed solely with the basic ops.
- Intersection takes two input relations, which must be union-compatible.
- Q: How to express it using basic operators?

$$R \cap S = R - (R - S)$$

Intersection operator (∩)

$S1$

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

$S2$

sid	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

$S2 \cap S1$

sid	sname	rating	age
31	Lubber	8	55.5
58	Rusty	10	35.0

Renaming Operator (ρ)

- Renames the list of attributes specified in the form of oldname $\rightarrow$ newname or position $\rightarrow$ newname
- **Output schema** is same as input except for the renamed attributes.
- Returns same tuples as input
- Can also be used to rename the name of the output relation

Boats

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

$$\rho_{bname \rightarrow boatname, color \rightarrow boatcolor}(Boats)$$

bid	boatname	boatcolor
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

$$\rho_{2 \rightarrow boatname, 3 \rightarrow boatcolor}(Boats)$$

Cross-Product (×)

- S1 x R1: Each row of S1 paired with each row of R1.
- Q: How many rows in the result?
- **Result schema** has one field per field of S1 and R1, with field names "inherited" if possible.
  - *May have a naming conflict*: Both S1 and R1 have a field with the same name.
  - In this case, can use the *renaming operator*:

$$\rho(C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1)$$

Call C the result of S1×R1 and respectively rename the 1^{st} & 5^{th} fields of C to sid1 & sid2

Cross-Product Example

$S1$

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

$R1$

sid	bid	day
22	101	10/10/96
58	103	11/12/96

$S1 \times R1$

sid	sname	rating	age	sid	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

$\rho_{1 \to sid1, 5 \to sid2}(S1 \times R1)$

sid1	sname	rating	age	sid2	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

Compound Operator: Join

- Joins are compound operators involving cross product, selection, and (sometimes) projection.
- Most common type of join is a **natural join** (often just called "join"). $R \bowtie S$ conceptually is:
  - Compute $R \times S$
  - Select rows where attributes that appear in both relations have equal values
  - Project all unique attributes and one copy of each of the common ones.
- Note: Usually done much more efficiently than this.
- Useful for putting "normalized" relations back together.

Natural Join Example

$$\pi_{S1.sid,sname,\ldots}\left(\sigma_{S1.sid=R1.sid}\left(\boldsymbol{S1} \times \boldsymbol{R1}\right)\right)$$

$\boldsymbol{S1}$

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

$\boldsymbol{R1}$

sid	bid	day
22	101	10/10/96
58	103	11/12/96

sid	sname	rating	age	sid	bid	day
22	Dustin	7	45.0	22	101	10/10/96
~~22~~	~~Dustin~~	~~7~~	~~45.0~~	~~58~~	~~103~~	~~11/12/96~~
~~31~~	~~Lubber~~	~~8~~	~~55.5~~	~~22~~	~~101~~	~~10/10/96~~
~~31~~	~~Lubber~~	~~8~~	~~55.5~~	~~58~~	~~103~~	~~11/12/96~~
~~58~~	~~Rusty~~	~~10~~	~~35.0~~	~~22~~	~~101~~	~~10/10/96~~
58	Rusty	10	35.0	58	103	11/12/96

$\boldsymbol{S1} \bowtie \boldsymbol{R1}$

sid	sname	rating	age	bid	day
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

Condition Join or Theta-Join

$$R \bowtie_C C = \sigma_C(R \times S)$$

- **Output schema** same as that of cross-product.
- May have fewer tuples than cross-product.

$S1$

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

$R1$

sid	bid	day
22	101	10/10/96
58	103	11/12/96

$$S1 \bowtie_{S1.sid<R1.sid} R1$$

sid	sname	rating	age	sid	bid	day
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	58	103	11/12/96

Equi-Join

- Special case of theta-join: condition *c* contains only conjunction of *equalities*.
- Find **all pairs** of sailors in S2 who have same age.

S2

<u>sid</u>	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

- $\boldsymbol{S1} \bowtie_{S1.age=S2.age} \boldsymbol{S2}$
- $\boldsymbol{S1} \bowtie_{age=age2} \rho_{age \to age2}(\boldsymbol{S2})$
- $\sigma_{\boldsymbol{sid1!=sid2}}\left(\boldsymbol{S1} \bowtie_{age=age2} \rho_{\substack{age \to age2 \\ sid \to sid2}}(\boldsymbol{S2})\right)$
- $\sigma_{\boldsymbol{sid1<sid2}}\left(\boldsymbol{S1} \bowtie_{age=age2} \rho_{\substack{age \to age2 \\ sid \to sid2}}(\boldsymbol{S2})\right)$

Grouping and Aggregation

- Grouping and Aggregation: γX (R)
  - Given a relation R, partition its tuples according to their values in one set of attributes G
    - The set G is called the grouping attributes
  - Then, for each group, aggregate the values in certain other attributes
    - Aggregation functions: SUM, COUNT, AVG, MIN, MAX, ...
- In the notation, X is a list of elements that can be:
  - A grouping attribute
  - An expression θ(A), where θ is one of the (five) aggregation functions and A is an attribute NOT among the grouping attributes

Grouping and Aggregation: Example

- Let's work with an example
  - Imagine that a social-networking site has a relation `Friends(User, Friend)`
  - The tuples are pairs *(a, b)* such that b is a friend of a
  - *Query: compute the number of friends each member has*

- $\gamma_{User,\ COUNT(Friend)}\ (Friends)$
  - This operation groups all the tuples by the value in their first component
  - There is one group for each user
  - Then, for each group, it counts the number of friends

Relational Algebra: Summary

Formal foundation for real query languages

- Helps represent and reason about execution plans

5 basic operators forming a closed algebra

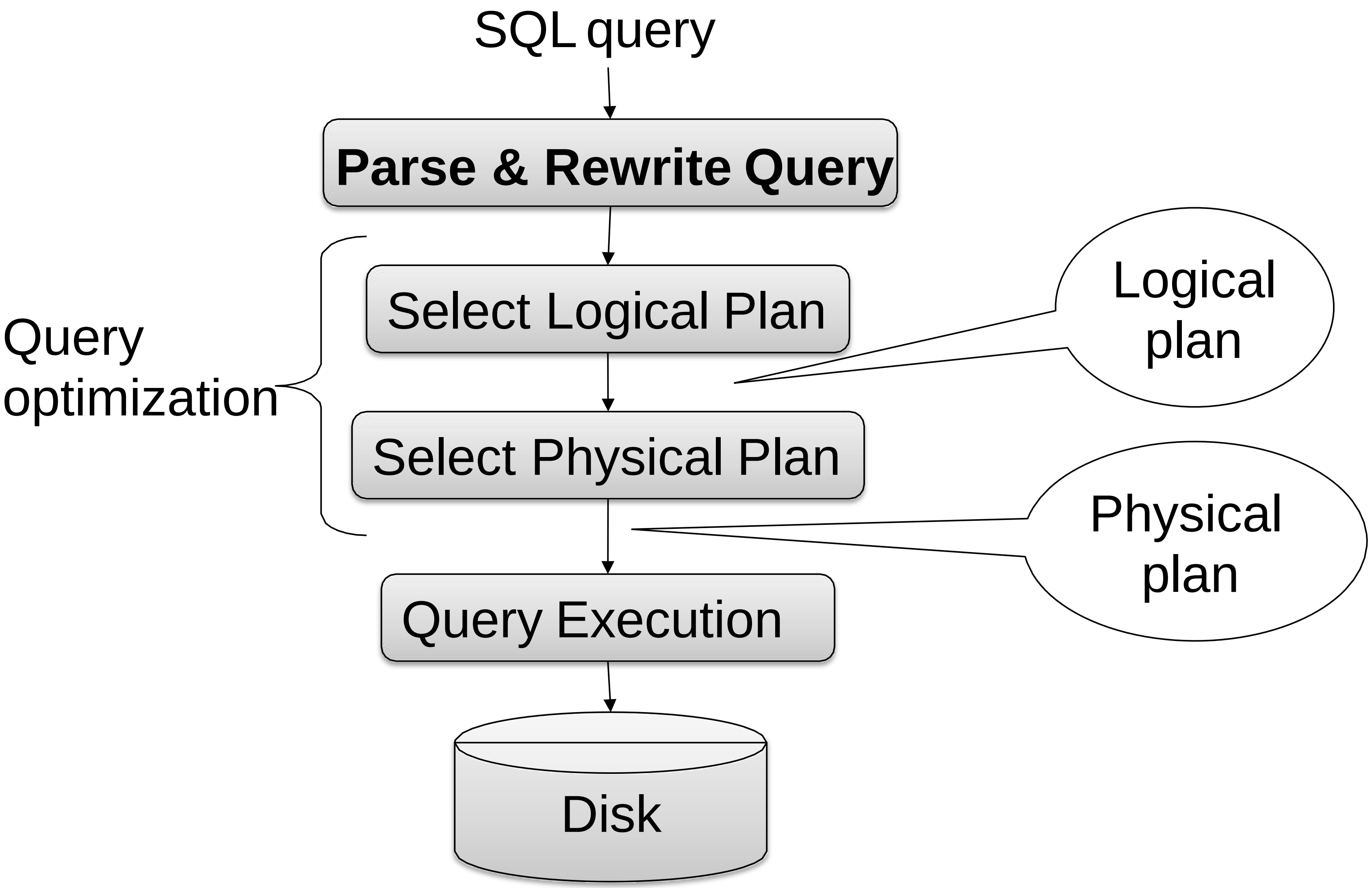
- Selection, projection, cross-product, union, set difference

Compound operators

- Useful shorthands like join and division
- Can be expressed with basic operators
- But enable faster query execution

Query Processing

Steps in Query Processing



Query parsing & transformation

A Query:

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

1. Query first broken into “blocks”
2. Each block converted to relational algebra

Step 1: Break query into Query Blocks

- Query block = unit of optimization

- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple

  (This is an over-simplification, but serves for now)

Outer block

```
SELECT  S.sname
FROM  Sailors S
WHERE  S.age IN
    (SELECT  MAX (S2.age)
     FROM  Sailors S2)
```

Nested block

Step 2: Converting query block into relational algebra expression

```
SELECT  S.sid
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = “red”
```

$$\pi_{S.sid}(\sigma_{B.color = \text{“red”}}(\text{Sailors} \bowtie \text{Reserves} \bowtie \text{Boats}))$$

Relational Algebra Equivalences

- Selections: $\sigma_{c_1 \wedge \cdots \wedge c_n}(R) \equiv \sigma_{c_1}\left(\ldots\left(\sigma_{c_n}(R)\right)\right)$ $(Cascade)$

  $\sigma_{c_1}\left(\sigma_{c_2}(R)\right) \equiv \sigma_{c_2}\left(\sigma_{c_1}(R)\right)$ *(Commute)*

- Projections: $\pi_{a_1}(R) \equiv \pi_{a_1}\left(\ldots\left(\pi_{a_n}(R)\right)\right)$ *(Cascade)*

  a_i is a set of attributes of R and $a_i \subseteq a_{i+1}$ for $i = 1 \ldots n-1$

- These equivalences allow us to 'push' selections and projections ahead of joins.

Examples …

$$\sigma_{age<18 \wedge rating>5}\ (Sailors)$$

$$\leftrightarrow \sigma_{age<18}\,(\sigma_{rating>5}\ (Sailors))$$

$$\leftrightarrow \sigma_{rating>5}\,(\sigma_{age<18}\,(Sailors))$$

$$\cancel{\pi_{age,rating}\ (Sailors) \leftrightarrow \pi_{age}\ (\pi_{rating}\ (Sailors))} \qquad (??)$$

$$\pi_{age,rating}\ (Sailors) \leftrightarrow \pi_{age,rating}\ (\pi_{age,rating,sid}\ (Sailors))$$

Another Equivalence

- A projection commutes with a selection that only uses attributes retained by the projection

$$\pi_{\text{age, rating, sid}} (\sigma_{\text{age}<18 \wedge \text{rating}>5} (\text{Sailors}))$$

$$\leftrightarrow \sigma_{\text{age}<18 \wedge \text{rating}>5} (\pi_{\text{age, rating, sid}} (\text{Sailors}))$$

Equivalences Involving Joins

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \qquad \textit{(Associative)}$$

$$(R \bowtie S) \equiv (S \bowtie R) \qquad \textit{(Commutative)}$$

- These equivalences allow us to choose different join orders

Mixing Joins with Selections & Projections

- Converting selection + cross-product to join

$$\sigma_{S.sid = R.sid}\ (\text{Sailors x Reserves})$$

$$\leftrightarrow \text{Sailors} \bowtie_{S.sid = R.sid} \text{Reserves}$$

- Selection on just attributes of S commutes with R $\bowtie$ S

$$\sigma_{S.age<18}\ (\text{Sailors} \bowtie_{S.sid = R.sid} \text{Reserves})$$

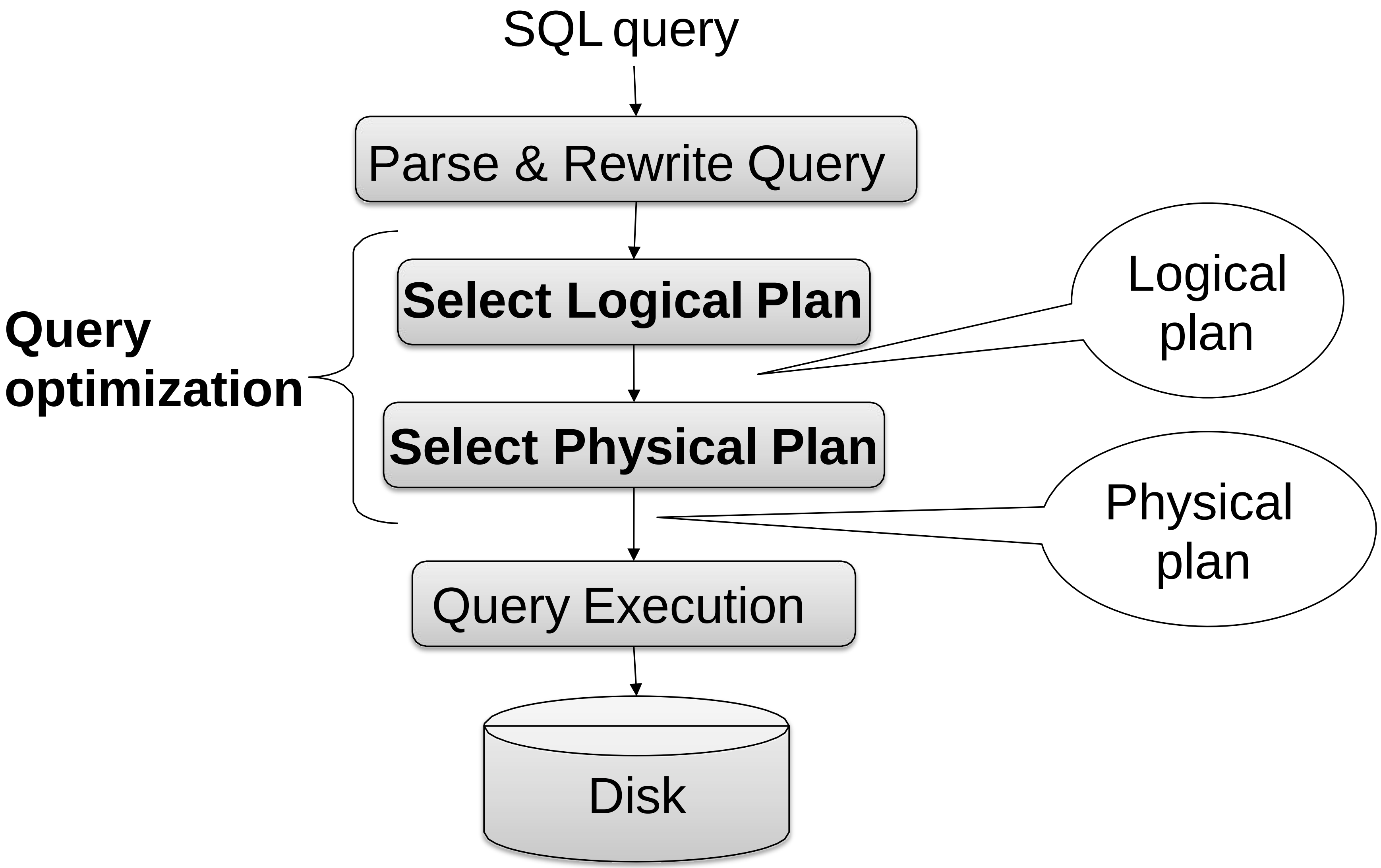
$$\leftrightarrow (\sigma_{S.age<18}\ (\text{Sailors})) \bowtie_{S.sid = R.sid} \text{Reserves}$$

- We can also “push down” projection (*but be careful…*)

$$\pi_{S.sname}\ (\text{Sailors} \bowtie_{S.sid = R.sid} \text{Reserves})$$

$$\leftrightarrow \pi_{S.sname}\,(\pi_{sname,sid}(\text{Sailors}) \bowtie_{S.sid = R.sid} \pi_{sid}(\text{Reserves}))$$

Steps in Query Processing



We know…

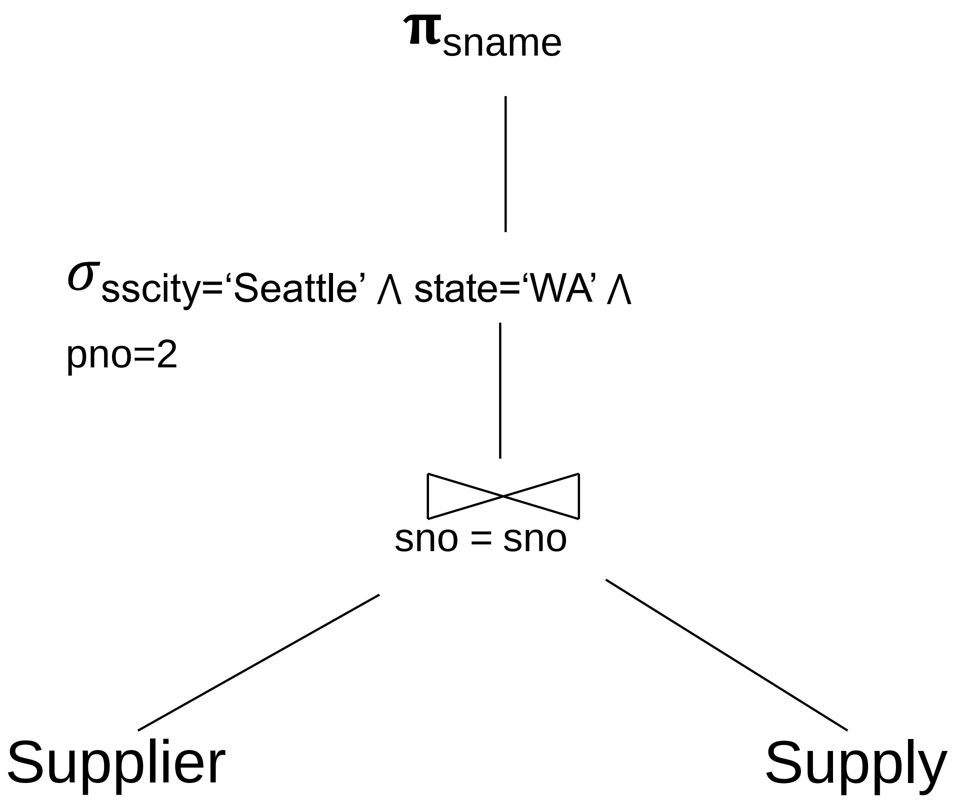
```
Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)
```

For each SQL query….

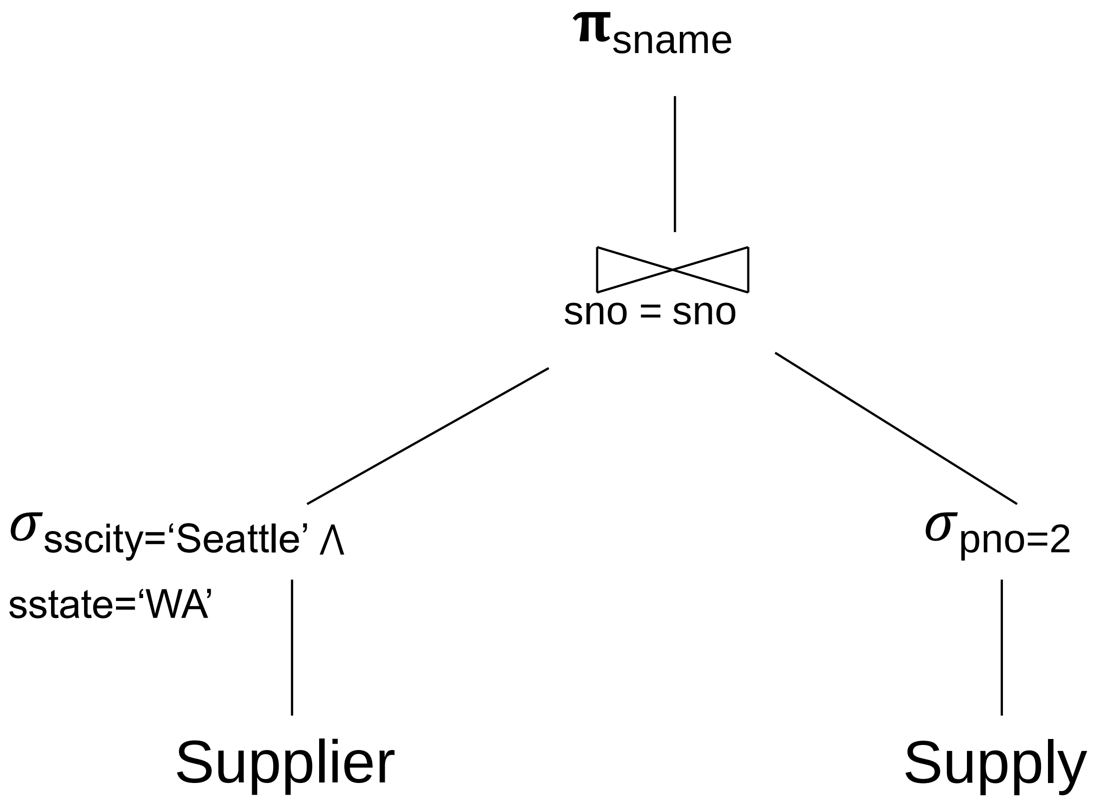
```
SELECT S.sname
FROM Supplier S, Supply U
WHERE S.scity='Seattle' AND S.sstate='WA'
AND  S.sno  = U.sno
AND  U.pno  = 2
```

There exist many logical query plans…

Example Query: Logical Plan 1



Example Query: Logical Plan 2

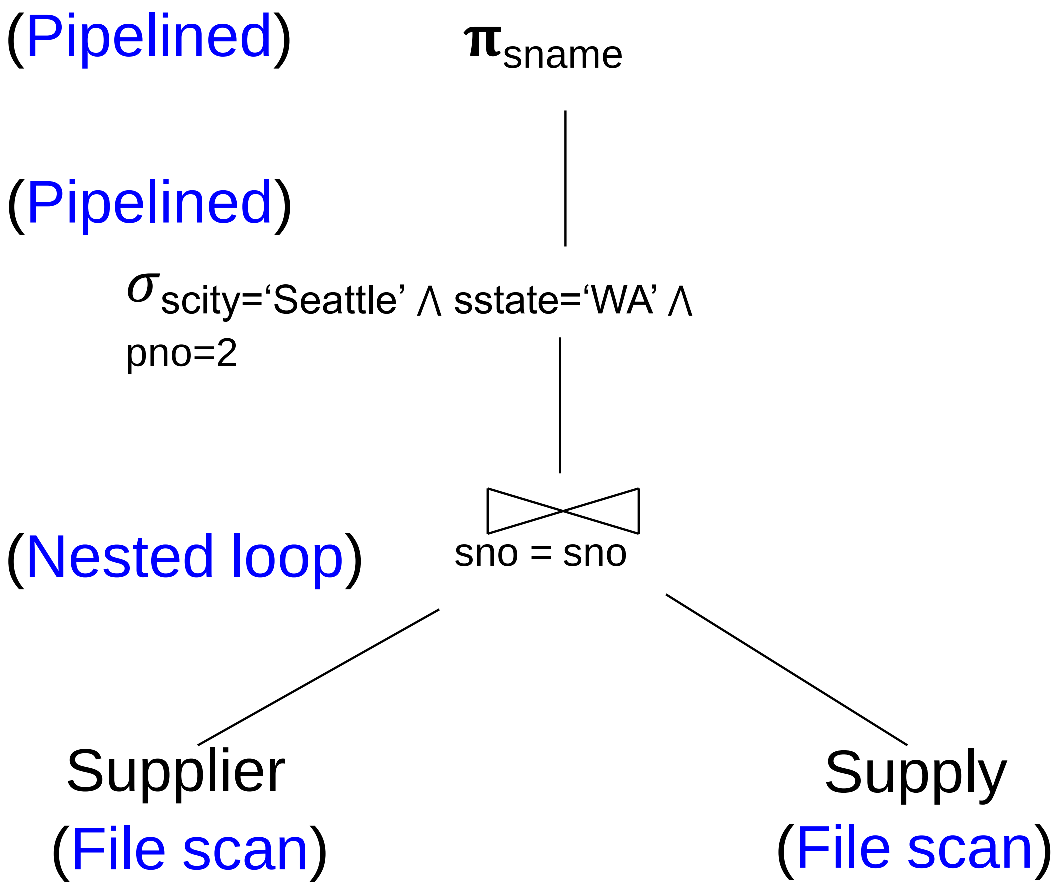


What We Also Know

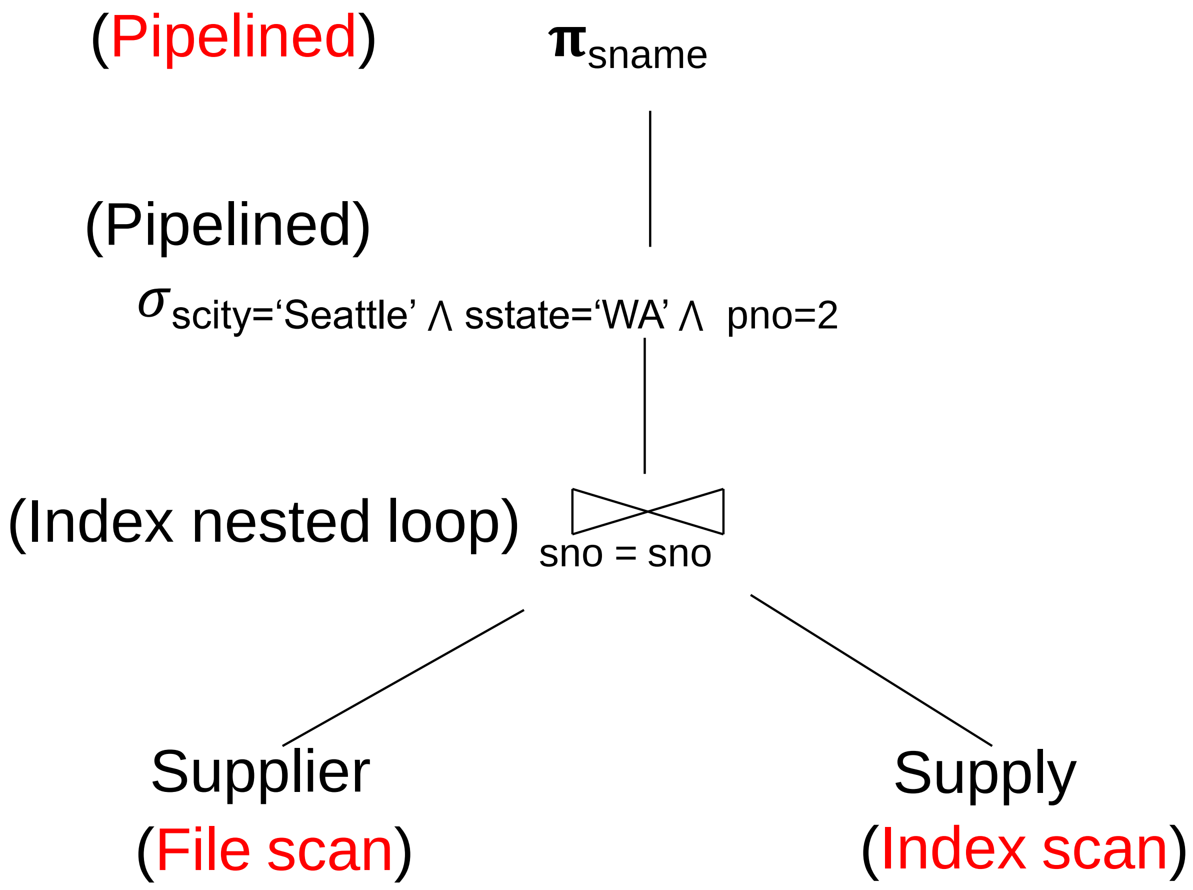
- For each logical plan…

- There exist many physical plans

Example Query: Physical Plan 1



Example Query: Physical Plan 2

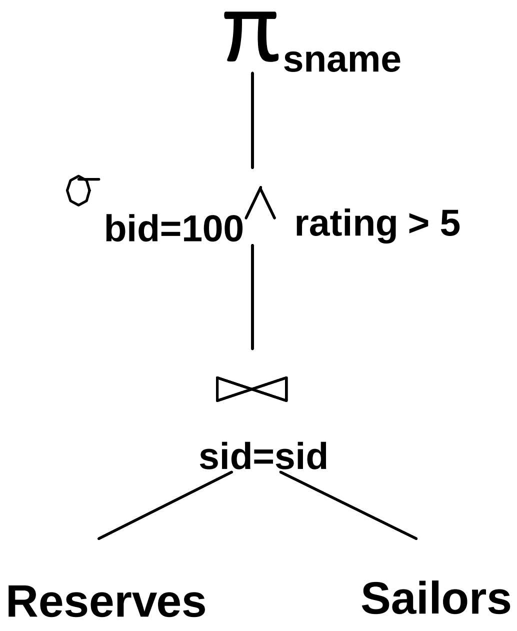


Query Optimization

1. Transformation produces relational algebra expression per "block"
2. Then, for each block, several alternative query plans are considered
3. Plan with lowest estimated cost is selected

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

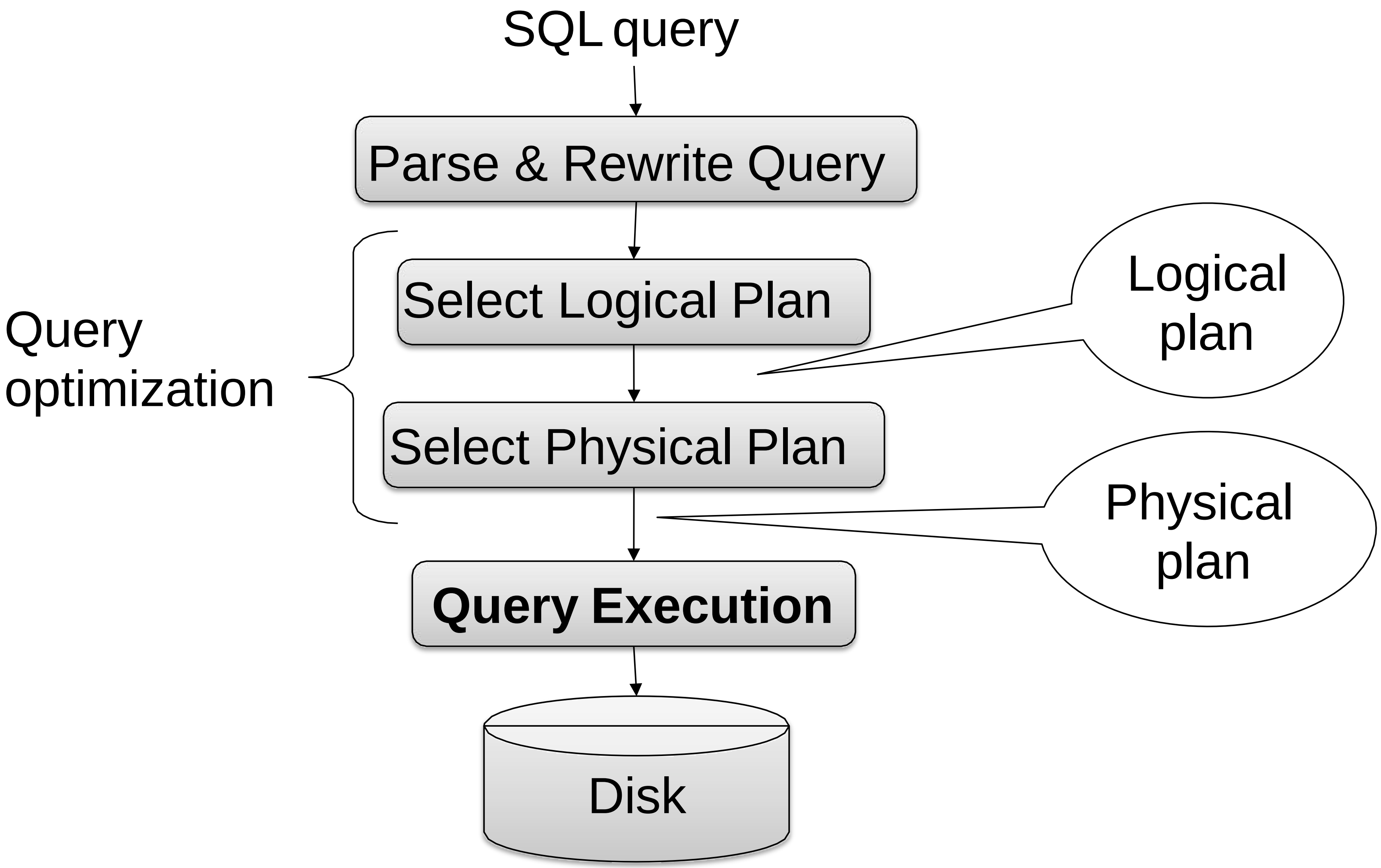
$$\pi_{(sname)}\sigma_{(bid=100 \wedge rating > 5)} (\text{Reserves} \bowtie \text{Sailors})$$



Query Optimizer Overview

- **Input**: A logical query plan
- **Output**: A good physical query plan
- **Basic query optimization algorithm**
  - Enumerate alternative plans (logical and physical)
  - Compute estimated cost of each plan
    - Compute number of I/Os
    - Optionally take into account other resources
  - Choose plan with lowest cost
  - This is called cost-based optimization

Steps in Query Processing



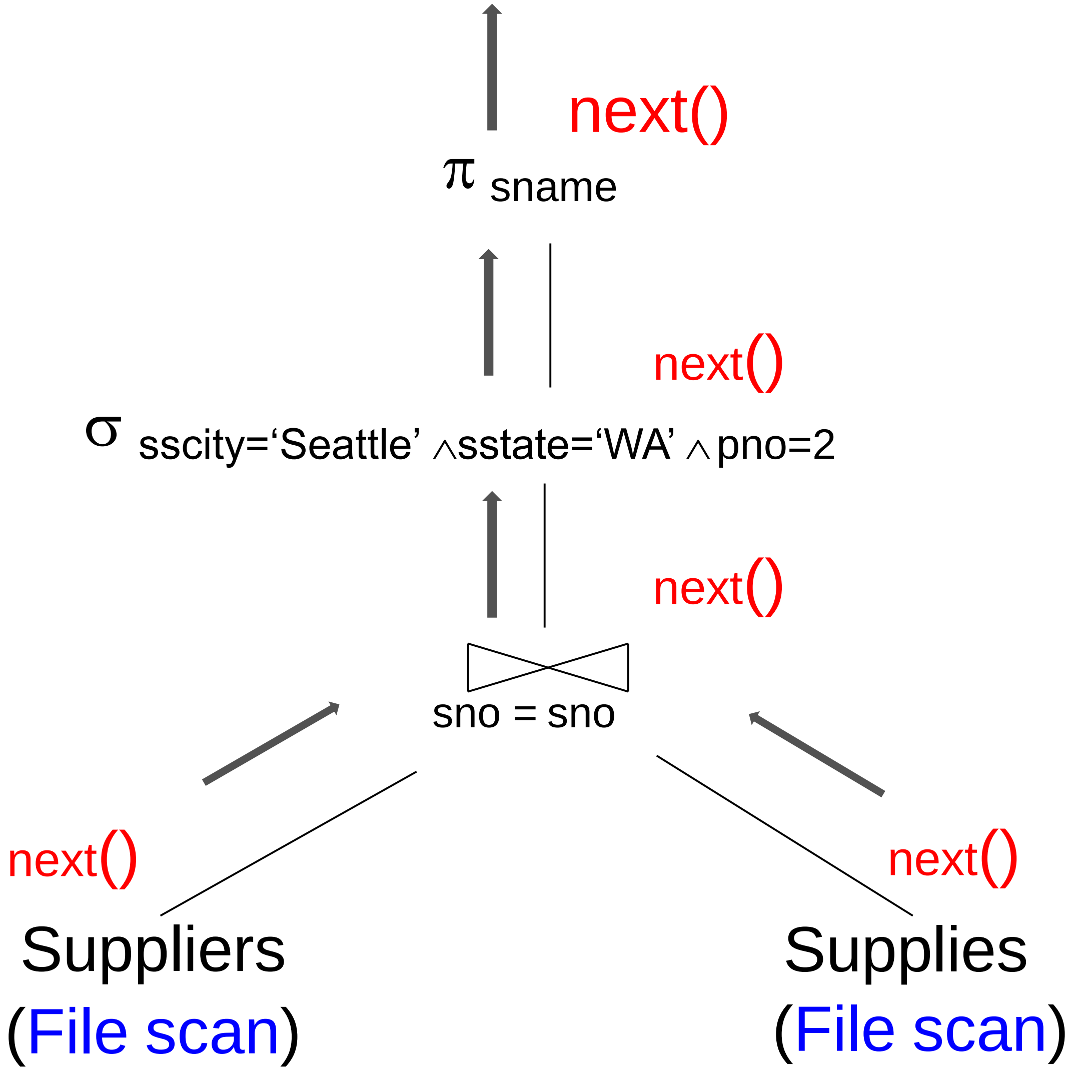
Query Execution: Volcano Model

Each operator implements an iterator interface

- **open()**
  - Initializes operator state
  - Sets parameters such as selection condition
- **next()**
  - Operator invokes get_next() recursively on its inputs
  - Performs processing and produces an output tuple
- **close()**: clean-up state

Pipelined Execution

Tuples generated by an operator are immediately sent to the parent

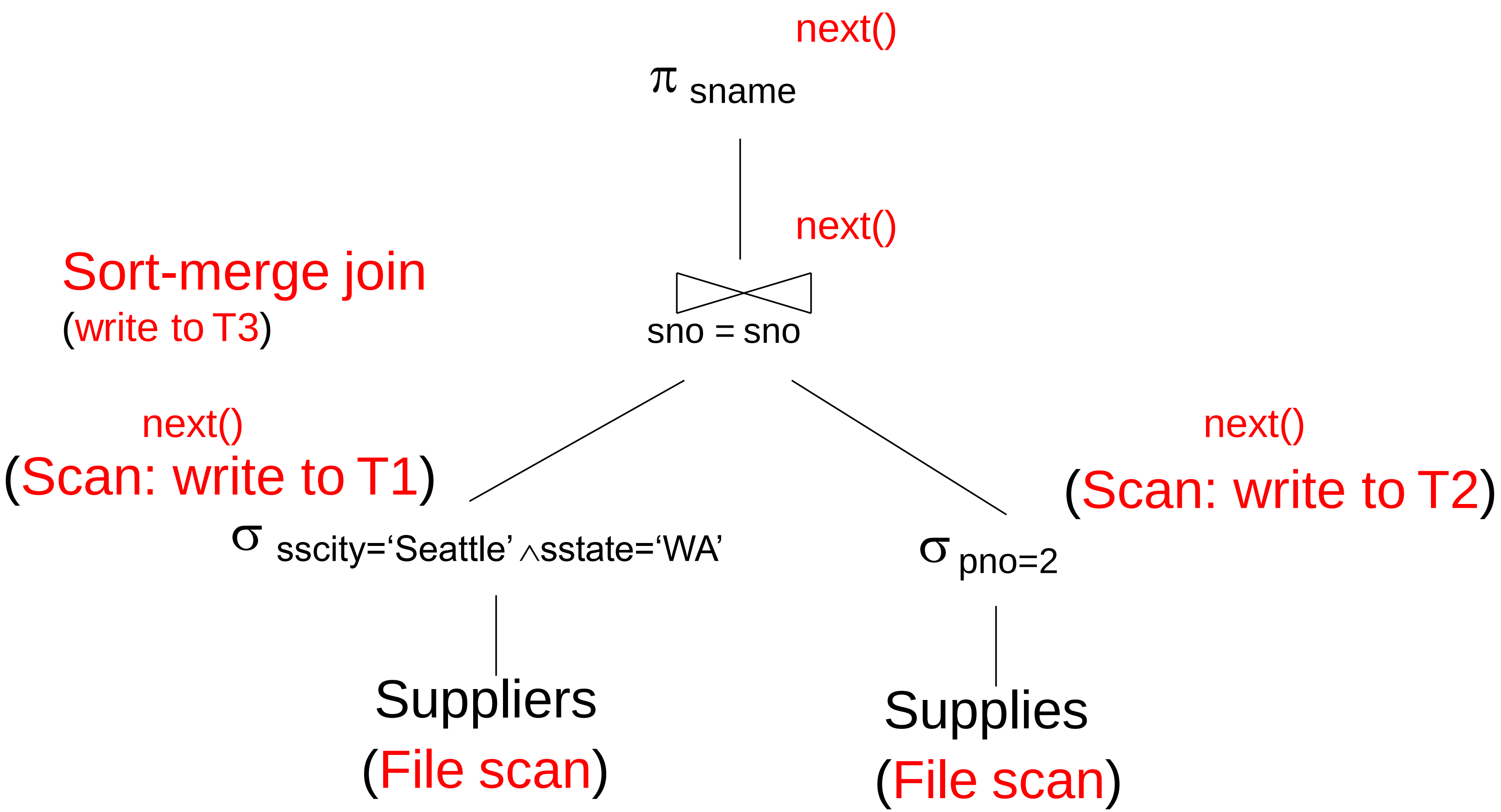


Pipelined Execution

- Tuples generated by an operator are immediately sent to the parent
- Benefits:
  - Pull based: No operator synchronization issues
  - Saves cost of writing intermediate data to disk
  - Saves cost of reading intermediate data from disk
- This approach is used whenever possible

Materialization Example

Tuples generated by an operator are written to an intermediate table



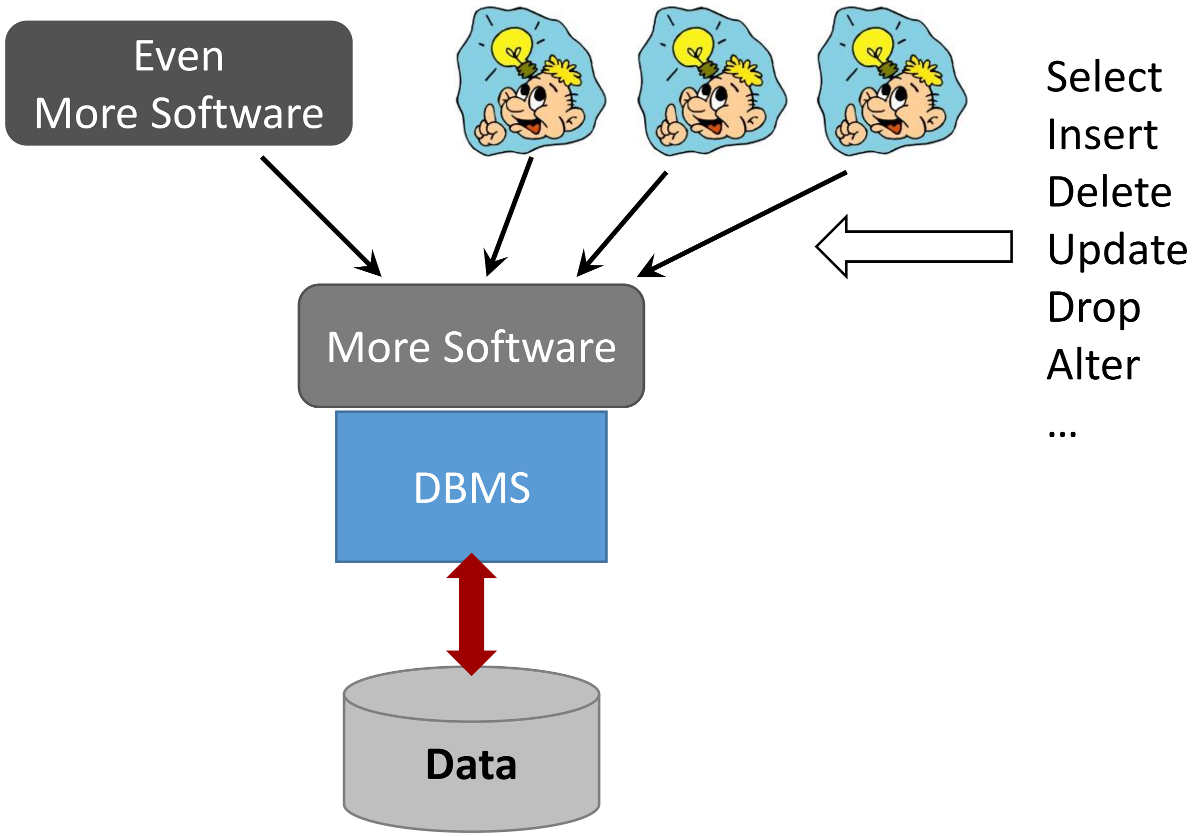
Intermediate Materialization

- Tuples generated by an operator are written to an intermediate table

- No direct benefit
- Necessary:
  - For certain operator implementations
  - When we don’t have enough memory

Transaction Management

Concurrent database access



Why concurrency is a problem?

Two concurrently executing queries

```
update account
set balance=balance-20
where accid=101
```

```
update account
set balance=balance+10
where accid=101
```

fetch; modify; update → Read; modify; Write

$R_1(X)$
$X=X-20$
$W_1(X)$

$R_2(X)$
$X=X+10$
$W_2(X)$

X

Accid	balance
101	100
102	1000
104	1000

t0:	$R_1(X)$	
t1:	$X=X-20$	
t2:	$W_1(X)$	
t3:		$R_2(X)$
t4:		
t5:		

X=90

t0:		$R_2(X)$
t1:		$X=X+10$
t2:		$W_2(X)$
t3:	$R_1(X)$	

X=90

t0:	$R_1(X)$	
t1:		$R_2(X)$
t2:	$X=X-20$	
t3:		$X=X+10$

X=110

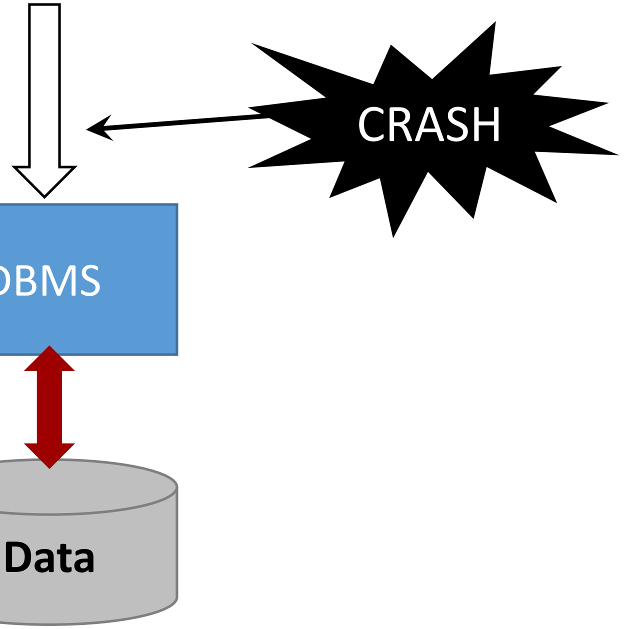
Arbitrary interleaving can lead to inconsistencies

Goal of concurrency control

- Execute sequence of SQL statements so they appear to be running in isolation
- Obvious way: execute them in isolation
  - **Is this acceptable?**
- Enable concurrency whenever possible and safe to do
  - utilization/throughput ("hide" waiting for I/Os)
  - response time
  - fairness

Resilience to system failures

```
update account set balance=balance-50 where accid=101
update account set balance=balance+50 where accid=102
```



Solution to both problems

- Concurrent database access
- Resilience to system failures

Transactions

- A transaction is a sequence of one or more SQL operations treated as a unit
  - Transactions appear to run in isolation
  - If the system fails, each transaction's changes are reflected either entirely or not at all

Correctness: The **ACID** properties

- **A**tomicity: All actions in the transaction happen, or none happen
- **C**onsistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent
- **I**solation: Execution of one transaction is isolated from that of other transactions
- **D**urability: If a transaction commits, its effects persist

A Atomicity of transactions

- Two possible outcomes of executing a transaction:
  - Transaction might *commit* after completing all its actions
  - or it could *abort* (or be aborted by the DBMS) after executing some actions

- DBMS guarantees that transactions are *atomic.*
  - From user’s point of view: transaction always either executes all its actions, or executes no actions at all

A Mechanisms for ensuring atomicity

- One approach: LOGGING
  - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions
- Another approach: SHADOW PAGING
  - (ask me after class if you're curious)

- Logging used by modern systems, because of the need for audit trail and for efficiency

D Durability - Recovering from a crash

- Three phases
  - ***Analysis***: Scan the log (forward from the most recent *checkpoint*) to identify all transactions that were active at the time of the crash
  - ***Redo***: Redo updates as needed to ensure that all logged updates are in fact carried out and written to disk
  - ***Undo***: Undo writes of all transactions that were active at the crash, working backwards in the log
- At the end – all committed updates and only those updates are reflected in the database
- Some care must be taken to handle the case of a crash occurring during the recovery process!

c Transaction consistency

- “Consistency” - data in DBMS is accurate in modeling real world and follows integrity constraints
- User must ensure that transaction is consistent
- Key point:

Transaction consistency (cont.)

- Recall: Integrity constraints
  - must be true for DB to be considered consistent
  - **Examples:**
    1. FOREIGN KEY R.sid REFERENCES S
    2. ACCT-BAL >= 0

- System checks integrity constraints and if they fail, the transaction rolls back (i.e., is aborted)
  - Beyond this, DBMS does not understand the semantics of the data
  - e.g., it does not understand how interest on a bank account is computed

Isolation of transactions

- Users submit transactions concurrently

- Each transaction executes as if it was running by itself
  - Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.

Example

- Consider two transactions:

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

A

B

Accid	balance
101	100
102	1000
104	1000

- 1st xact transfers $100 from B's account to A's
- 2nd credits both accounts with 6% interest
- Assume at first A and B each have $1000. What are the <u>legal outcomes</u> of running T1 and T2?
  - $2000 *1.06 = $2120
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect *must* be equivalent to these two transactions running serially in some order

Example (contd.)

- Legal outcome: A=1166,B=954
- Consider a possible interleaved *schedule*:

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

- This is OK (same as T1;T2). But what about:

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

- **Result: A=1166, B=960; A+B = 2126, bank loses $6**

Anomalies with interleaved execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

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

- Unrepeatable Reads (RW Conflicts):

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

- Overwriting Uncommitted Data (WW Conflicts):

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

How do we allow concurrency while preventing these anomalies? (Theory of serializability)

Transactions & Schedules: Definitions

- A program may carry out many operations on the data retrieved from the database
- The DBMS is only concerned about what data is read/written from/to the database
- *Database*
  - a fixed set of named data objects *(A, B, C, …)*
- *Transaction*
  - a sequence of actions *(read(A), write(B),commit, abort …)*
- *Schedule*
  - an interleaving of actions from various transactions

Formal properties of schedules

- *Serial schedule:* Schedule that does not interleave the actions of different transactions

T_1: $R_1(X)$, $X=X-20$, $W_1(X)$

T_2: $R_2(X)$, $X=X+10$, $W_2(X)$

T_1, T_2		
t0:	$R_1(X)$	
t1:	$X=X-20$	
t2:	$W_1(X)$	
t3:		$R_2(X)$
t4:		$X=X+10$
t5:		$W_2(X)$

T_2, T_1		
t0:		$R_2(X)$
t1:		$X=X+10$
t2:		$W_2(X)$
t3:	$R_1(X)$	
t4:	$X=X-20$	
t5:	$W_1(X)$	

Formal properties of schedules

- *Equivalent schedules*: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions
  Note: If each transaction preserves consistency, every serializable schedule preserves consistency.

Conflicting operations

- We need a formal notion of equivalence that can be implemented efficiently
  - Base it on the notion of “conflicting” operations

- Definition: Two operations conflict if:
  - They are done by different transactions,
  - And they are done on the same object,
  - And at least one of them is a write

$R_1(A)$, $W_2(A)$

$W_1(A)$, $R_2(A)$

$W_1(A)$, $W_2(A)$

$R_1(B)$, $W_2(B)$

$W_1(B)$, $R_2(B)$

$W_1(B)$, $W_2(B)$

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

Conflict serializable schedules

- <u>Definition</u>: Two schedules are conflict equivalent iff:
  - They involve the same actions of the same transactions,
  - And every pair of conflicting actions is ordered the same way

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

$S_1 \equiv S_2$

$S_1 \equiv S_3$??

S_1:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
$R_1(B)$	
$W_1(B)$	
	$R_2(B)$
	$W_2(B)$

S_2:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
$R_1(B)$	
	$W_2(A)$
$W_1(B)$	
	$R_2(B)$
	$W_2(B)$

S_3:

T_1	T_2
$R_1(A)$	
	$R_2(A)$
$W_1(A)$	
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$
$R_1(B)$	
$W_1(B)$	

Conflict serializable schedules

- <u>Definition</u>: Schedule S is conflict serializable if:
  - S is conflict equivalent to some serial schedule

T_1: $R_1(A)$, $A=A-100$, $W_1(A)$, $R_1(B)$, $B=B+100$, $W_1(B)$

T_2: $R_2(A)$, $A=1.06*A$, $W_2(A)$, $R_2(B)$, $B=1.06*B$, $W_2(B)$

S_1:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
$R_1(B)$	
$W_1(B)$	
	$R_2(B)$
	$W_2(B)$

S_2:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
$R_1(B)$	
$W_1(B)$	
	$R_2(A)$
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$

S_3:

T_1	T_2
	$R_2(A)$
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$
$R_1(A)$	
$W_1(A)$	
$R_1(B)$	
$W_1(B)$	

Conflict serializability: Definition

- 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:*

R(A) W(A) R(B) W(B)

R(A) W(A) R(B) W(B)

$\equiv$

R(A) W(A) R(B) W(B)

R(A) W(A) R(B) W(B)

Conflict serializability (cont.)

- Here's another example:

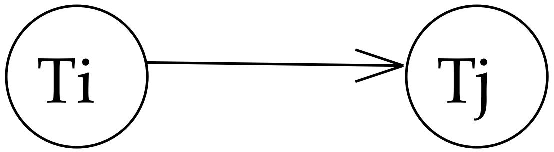
R(A) W(A)

R(A) W(A)

- Conflict serializable or not?

NOT!

Testing for conflict serializability



- *Precedence graph*:
  - One node per transaction
  - 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 precendence graph is acyclic

Precedence graph

$R_1(A), W_2(A)$

$W_1(A), R_2(A)$

$W_1(A), W_2(A)$

$R_1(B), W_2(B)$

$W_1(B), R_2(B)$

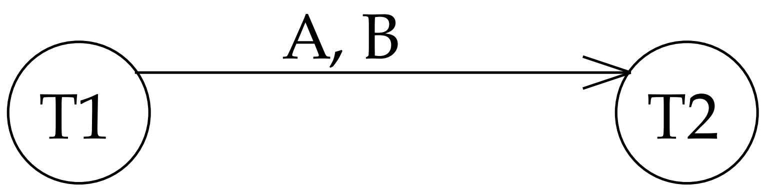
$W_1(B), W_2(B)$

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

S_1:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
$R_1(B)$	
$W_1(B)$	
	$R_2(B)$
	$W_2(B)$



Precedence graph

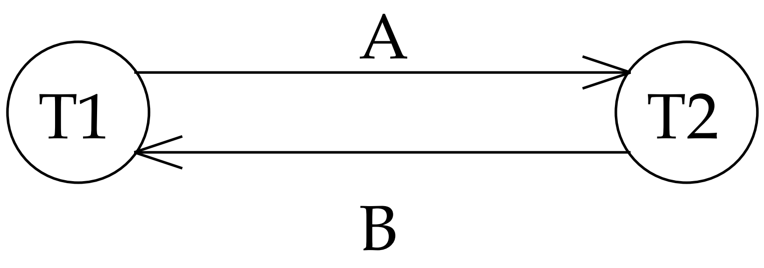
$R_1(A), W_2(A)$
$W_1(A), R_2(A)$
$W_1(A), W_2(A)$
$R_1(B), W_2(B)$
$W_1(B), R_2(B)$
$W_1(B), W_2(B)$

T_1: $R_1(A)$, A=A-100, $W_1(A)$, $R_1(B)$, B=B+100, $W_1(B)$

T_2: $R_2(A)$, A=1.06*A, $W_2(A)$, $R_2(B)$, B=1.06*B, $W_2(B)$

S_1:

T_1	T_2
$R_1(A)$	
$W_1(A)$	
	$R_2(A)$
	$W_2(A)$
	$R_2(B)$
	$W_2(B)$
$R_1(B)$	
$W_1(B)$	



Not conflict serializable

The cycle in the graph reveals the problem.
The output of T1 depends on T2, and vice-versa

Two-Phase Locking (2PL)

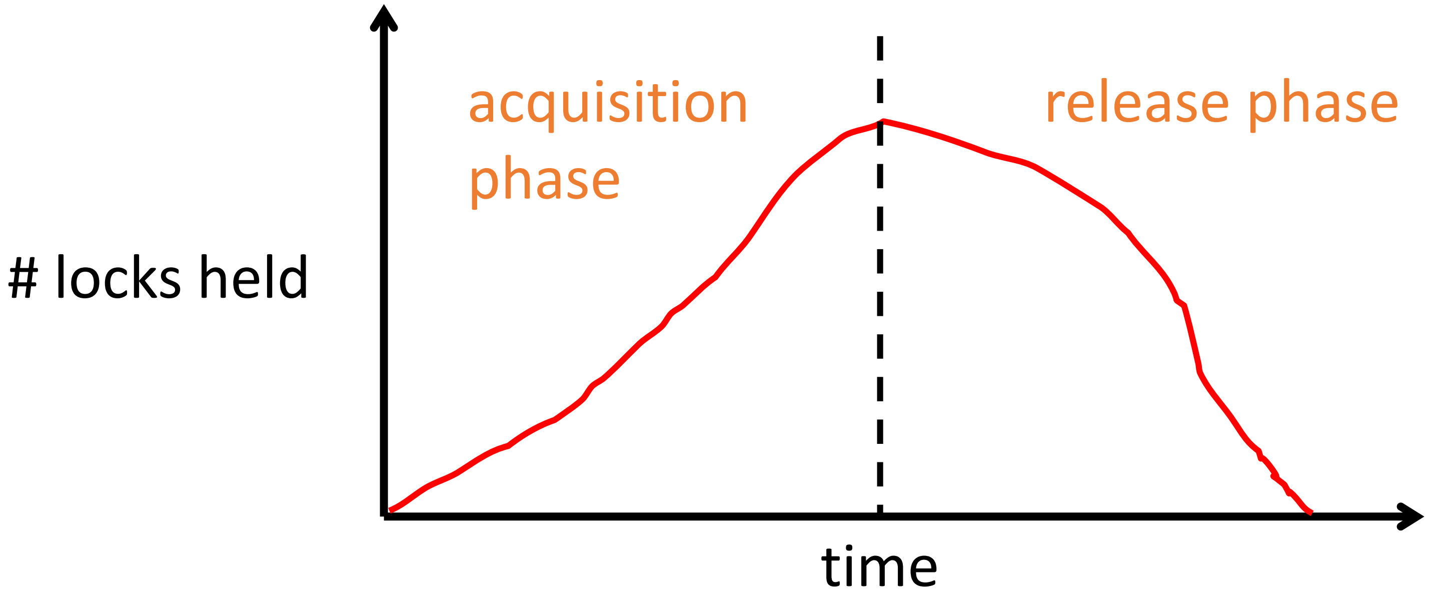
- Locking protocol
  - Each transaction must obtain an S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing
  - ***A transaction cannot request additional locks once it releases any locks***
  - Thus, there is a "growing phase" followed by a "shrinking phase"

Lock Compatibility Matrix

	S	X
S	√	–
X	–	–

2PL & Serializability

- 2PL on its own is sufficient to guarantee conflict serializability (i.e., schedules whose precedence graph is acyclic), but, it is subject to **Cascading Aborts**



Strict 2PL

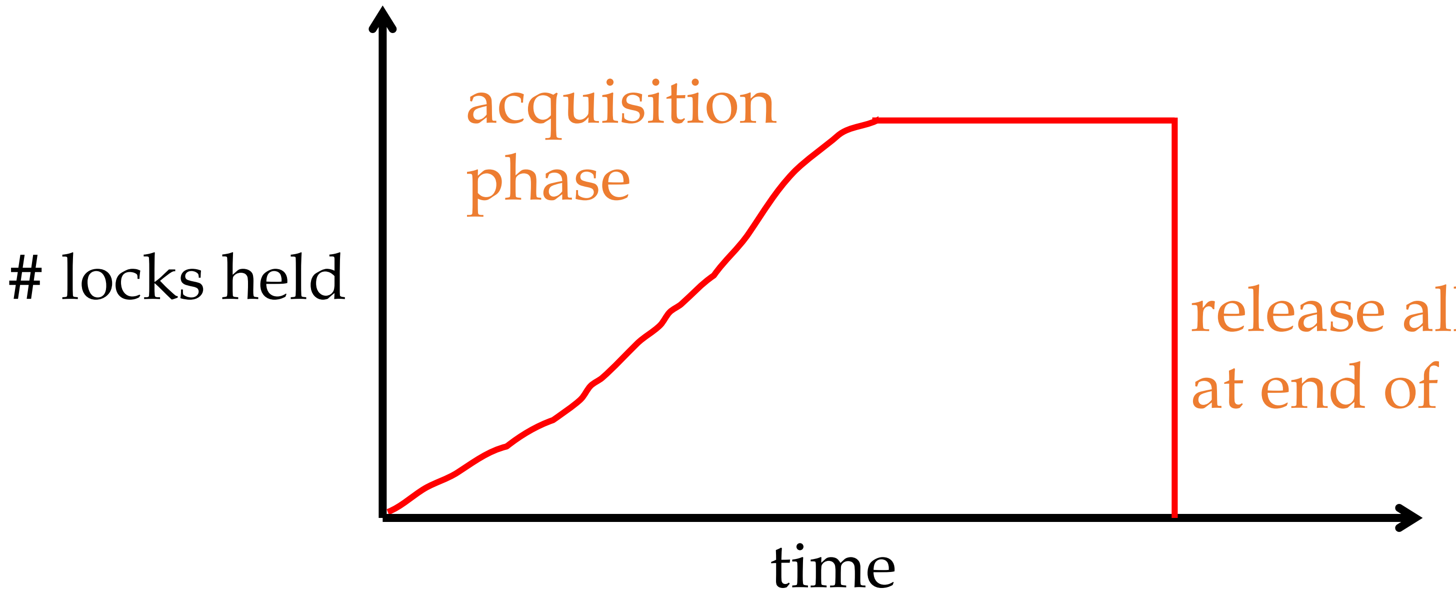
- Problem: Cascading Aborts
- Example: rollback of T1 requires rollback of T2!

T1:	$R_1(A), W_1(A), R_1(B), W_1(B),$		Abort
T2:		$R_2(A), W_2(A)$	

- To avoid Cascading Aborts, use Strict 2PL
- Strict Two-Phase Locking (Strict 2PL) Protocol:
  - Same as 2PL, except: All locks held by a transaction are released only when the transaction completes

Strict 2PL (cont.)

- Allows only conflict serializable schedules
- In effect, “shrinking phase” is delayed until
  a) Transaction has committed (commit log record on disk), or
  b) Decision has been made to abort the transaction (locks can be released after rollback)



Non-2PL, A= 100, B=200, output =?

Lock_X(A)		
Read(A)	Lock_S(A)	
A: = A-50		
Write(A)		A=50
Unlock(A)		
	Read(A)	
	Unlock(A)	
	Lock_S(B)	
Lock_X(B)		
	Read(B)	
	Unlock(B)	
	PRINT(A+B)	**250**
Read(B)		
B := B +50		
Write(B)		B=250
Unlock(B)		

2PL, A= 100, B=200, output =?

Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Unlock(A)	
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

→ A=50

→ B=250

→ **300**

Strict 2PL, A= 100, B=200, output =?

Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	
	Read(A)
	Lock_S(B)
	Read(B)
	PRINT(A+B)
	Unlock(A)
	Unlock(B)

→ A=50

→ B=250

→ **300**

2PL: Summary

- Locks implement the notions of conflict directly
- 2PL has:
  - Growing phase where locks are acquired and no lock is released
  - Shrinking phase where locks are released and no lock is acquired
- Strict 2PL requires all locks to be released at once, when transaction ends

Conclusion

- Relational databases are data management stalwarts
  - Data model provides data independence
  - Query language provides easy, declarative way to define and manipulate data
  - DBMS uses semantics and structure to optimize query processing
  - ACID properties of transactions simplifies application development

- We only grazed the surface. Take database course for more!
  - ER modeling, normalization for conceptual database design
  - Query compilation and vectorization for fast query processing
  - Optimistic & pessimistic concurrency control for high throughput txns
  - ….