Database Management Systems

Query Execution and Optimization

M. Emre Gürsoy

Assistant Professor
Department of Computer Engineering
www.memregursoy.com

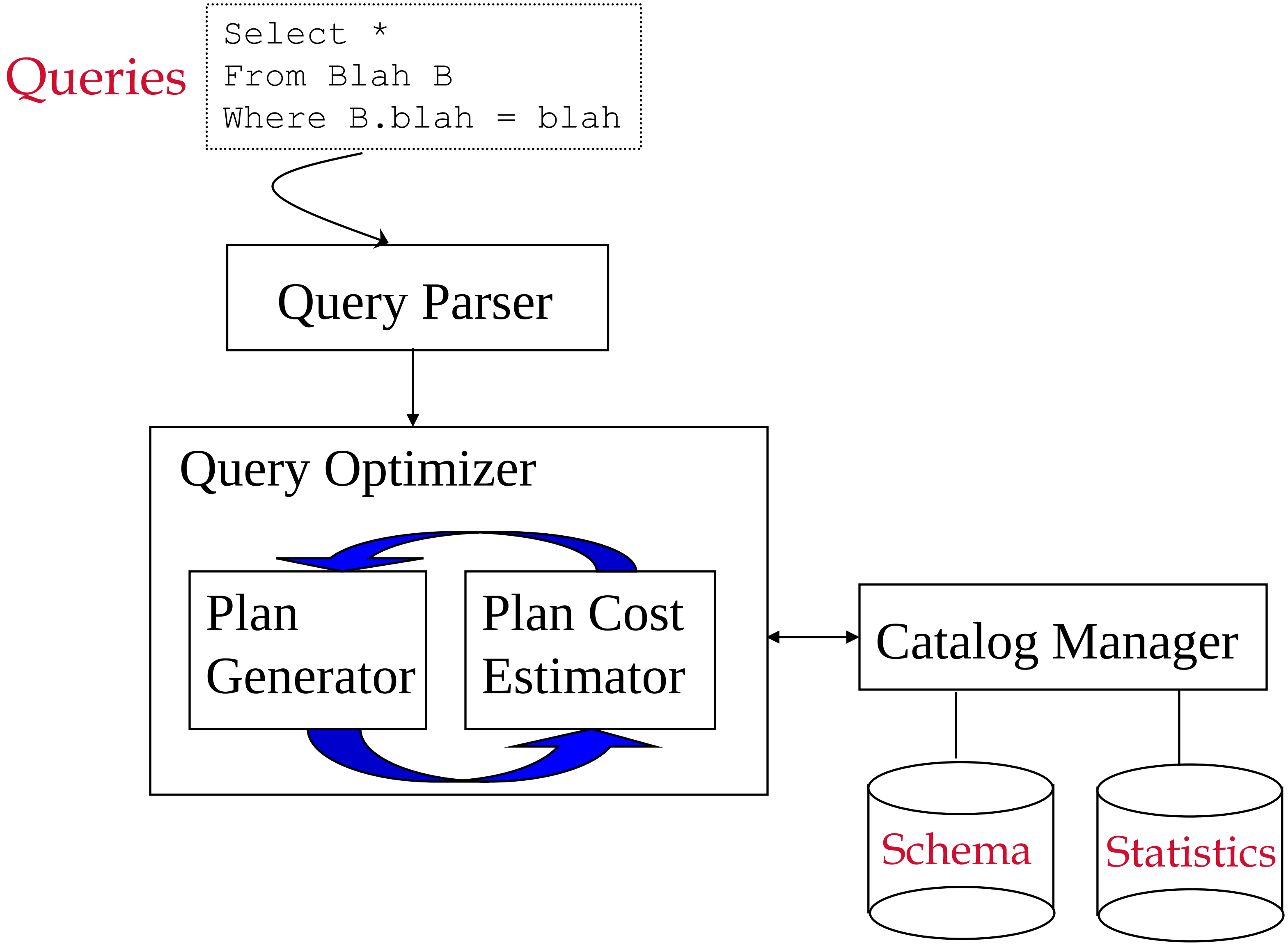
Introduction

- SQL is a **declarative** language, not procedural
  - You tell the DBMS what you want, but you don't tell the DBMS how to compute it
  - The DBMS must decide how to compute it
  - And the DBMS must find a way to compute it efficiently

```
SELECT  S.sname
FROM  Sailors S
WHERE S.rating =
  (SELECT  MAX (S2.rating)
   FROM  Sailors S2)
```

What are the alternative ways to compute the answer to this query? Which way is fastest?

Overview


Queries
Select *
From Blah B
Where B.blah = blah
Query Parser
Query Optimizer
Plan Generator
Plan Cost Estimator
Catalog Manager
Schema
Statistics

Steps of Query Optimization

- **Step 1:** SQL query is parsed and divided into blocks.

- **Step 2:** Each block is converted to its “relational algebra equivalent”, which is used to construct its Relational Algebra tree (RA Tree). This is the initial plan.
  - The DBMS may not internally draw an RA tree, but this is how we (humans) reason about it.

- **Step 3:** A subset of alternative plans are generated.

- **Step 4:** Costs of the different plans are estimated.
  - The DBMS picks and executes the plan with the lowest estimated cost.

SQL Blocks

- SQL queries are optimized by decomposing them into a collection of smaller units, called blocks.
  - Typically, query optimizer concentrates on optimizing a single block at a time.

- A block is an SQL query with:
  - No nesting
  - Exactly 1 SELECT and 1 FROM clause
  - At most 1 WHERE, 1 GROUP BY and 1 HAVING clause

```
SELECT      S.name
FROM        Student S, Takes T
WHERE       T.cid='415' and
            S.ssn=T.ssn
```

Nested Queries

- If the query is nested (but not correlated):
  - The inner query is the first block to be processed
  - Its output is finalized
  - The output is incorporated into the outer query as if it had been part of the original statement
  - The outer query is processed as the second block

```
SELECT  S.sname
FROM  Sailors S
WHERE S.rating = (SELECT  MAX (S2.rating)
                  FROM  Sailors S2)
```

Correlated Nested Queries

- If the query is a **correlated** nested query, we can't evaluate the inner query once and be done with it
- In this case, the typical strategy is to evaluate the inner query for each different tuple of Sailors
  - Repeat the inner block many times

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

RA Tree

- Say that I have an SQL block, how do I convert it into a query evaluation plan?
  - Relational Algebra tree (RA tree)
  - The RA tree is **procedural**, i.e., it tells you what to do step-by-step

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

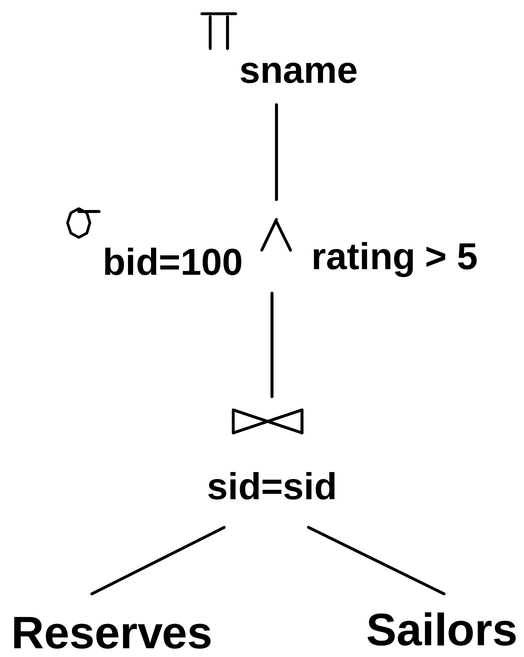
What is the corresponding RA tree of this query?

- First express the query in relational algebra:

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(\mathrm{Reserves} \bowtie_{sid=sid} Sailors))$$

RA Tree

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(\text{Re}\,serves \bowtie_{sid=sid} Sailors))$$



RA Tree

```
SELECT  S.rating, COUNT (*)
FROM  Sailors S
WHERE  S.rating > 5 AND S.age = 20
GROUP BY S.rating
```

$$\pi_{\text{rating, COUNT(*)}}$$

$$\text{GROUP BY}_{\text{rating}}$$

$$\sigma_{\text{rating} > 5 \wedge \text{age} = 20}$$

Sailors

RA Tree

- "Find the birthdate and movie title for those female stars who appeared in some movie in 1996."

SELECT Title, Birthdate
FROM MovieStar, StarIn
WHERE Year=1996 **AND** Gender='F' **AND** Name=StarName

$$\pi_{\texttt{Title,Birthdate}}$$

$$\sigma_{\texttt{Year}=1996 \wedge \texttt{Gender}='F' \wedge \texttt{StarName}=\texttt{Name}}$$

$$\times$$

MovieStar   StarIn

RA Equivalences

- Note that given one SQL query, there may be multiple equivalent ways to write it in relational algebra.
  - Cartesian product, then selection
  - Join MovieStar and StarIn wrt Name=StarName

```
SELECT Title, Birthdate
FROM   MovieStar, StarIn
WHERE  Year=1996 AND Gender='F' AND Name=StarName
```

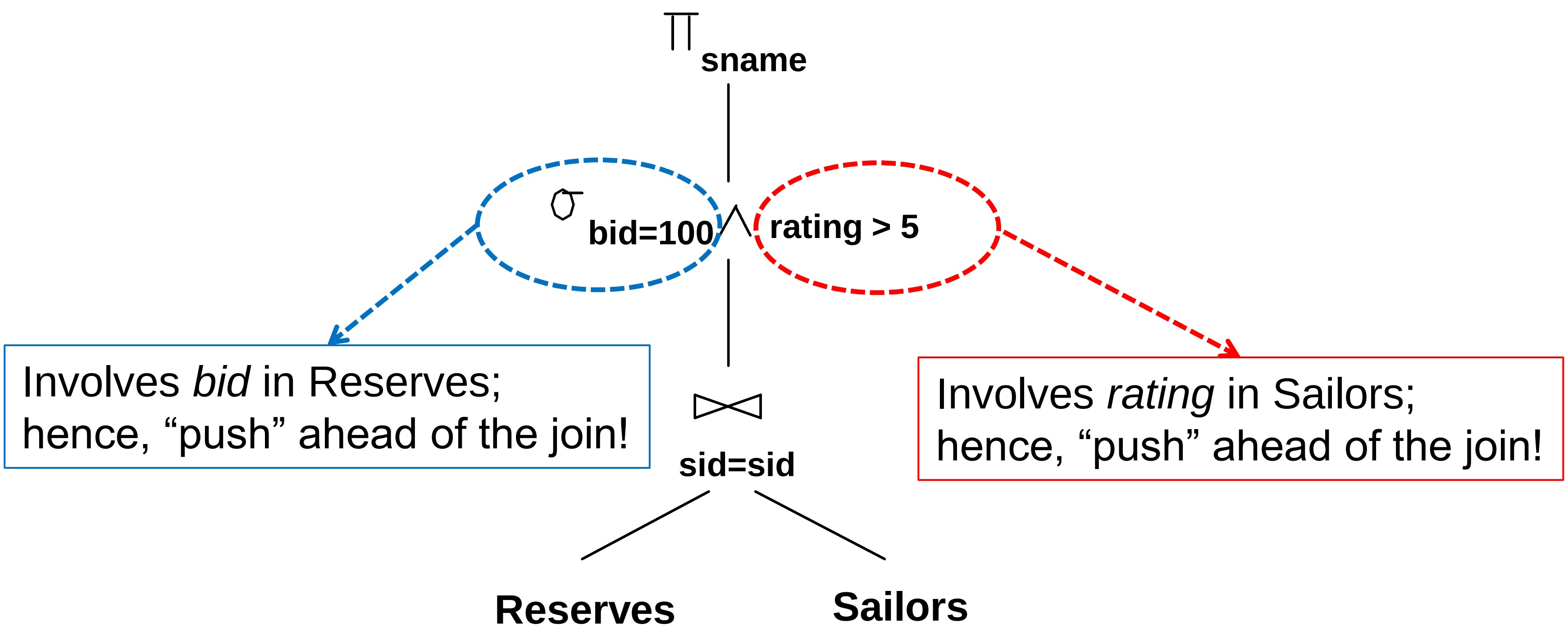
- There may be optimizations you can perform on the RA query while maintaining equivalence:

$$\pi_{sname}(\sigma_{bid=103}(Reserves \bowtie Sailors))$$

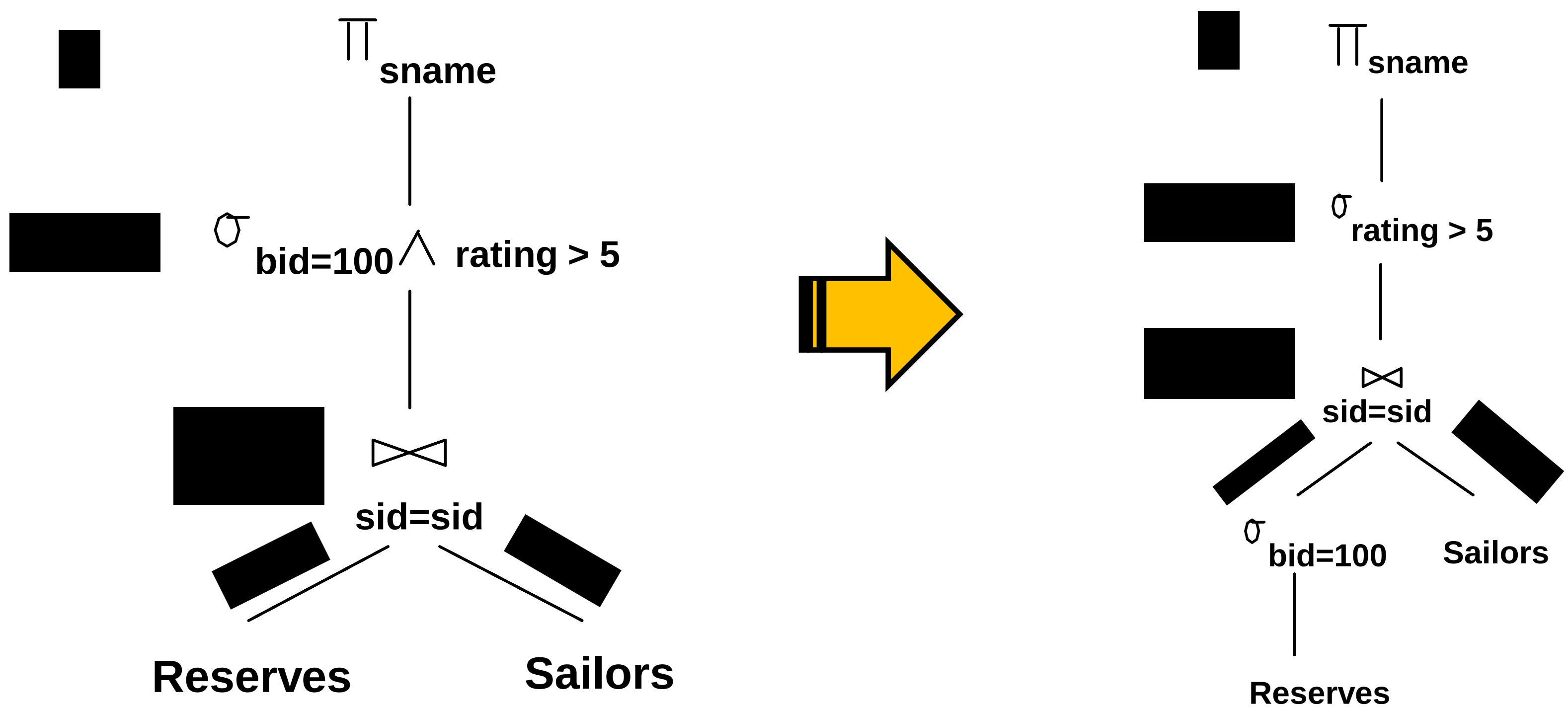
$$\pi_{sname}((\sigma_{bid=103} Reserves) \bowtie Sailors)$$

Operator Pushdown

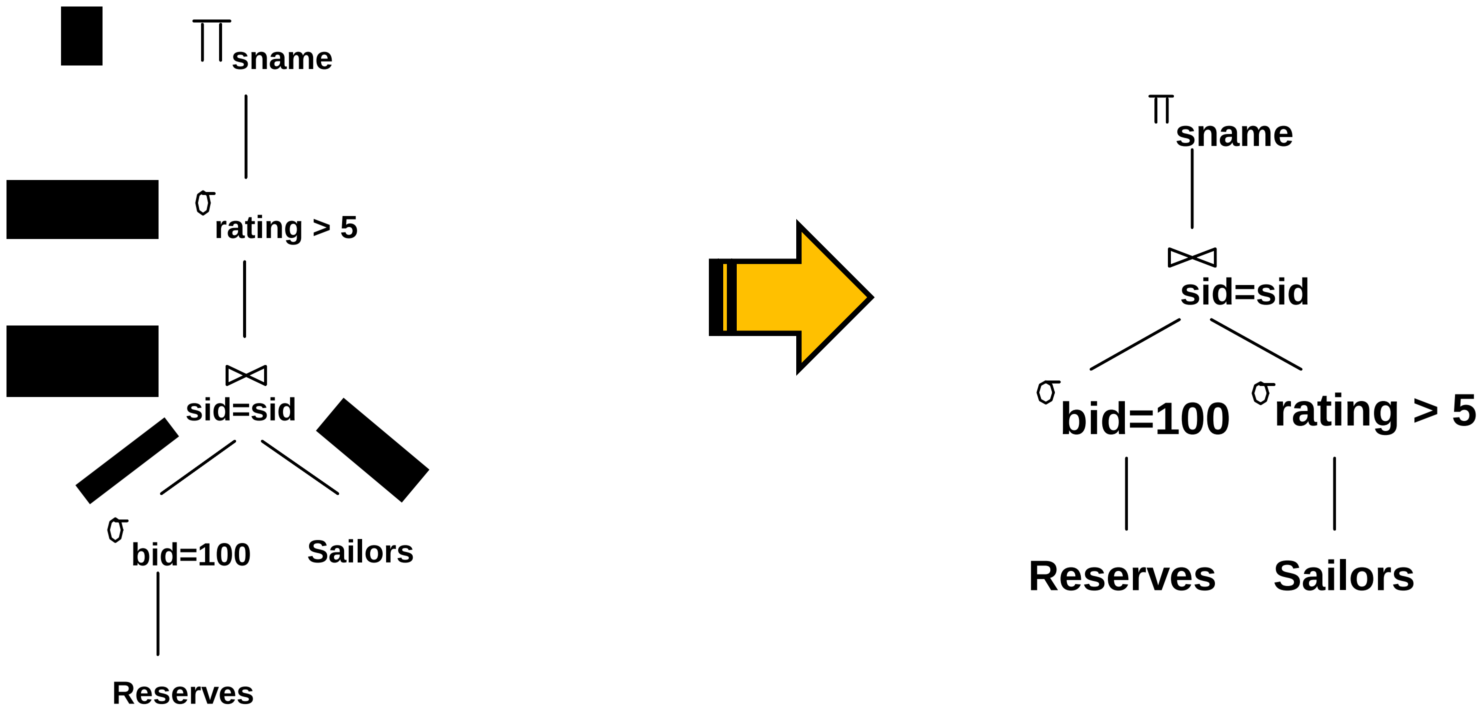
- You can decrease the cost of an RA tree (aka: query execution plan) by pushing some operators down
  - Be careful – **do not hurt equivalence!**



Operator Pushdown

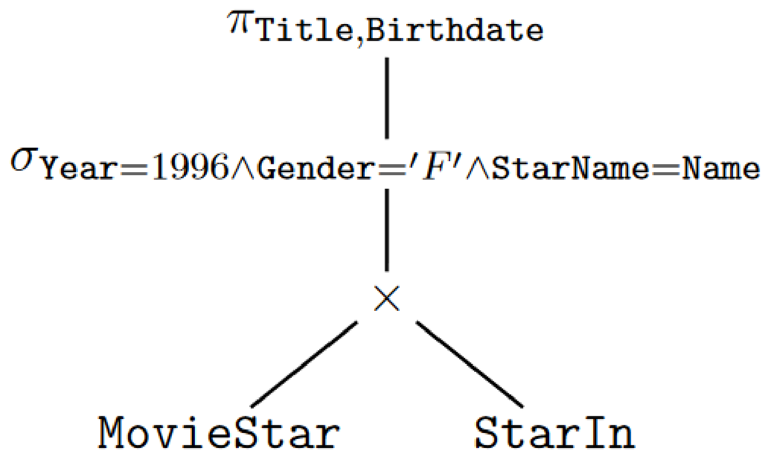


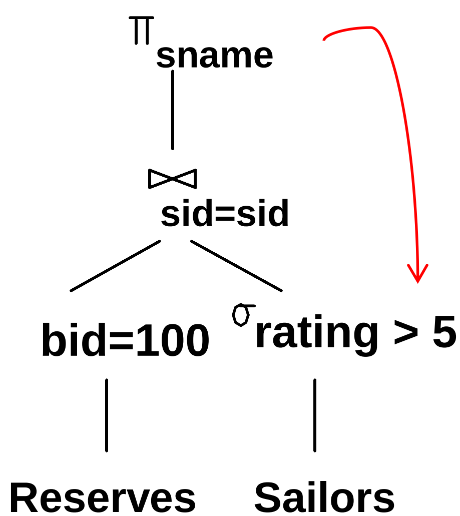
Operator Pushdown



Operator Pushdown

Replacing a {Cross Product + Select} with a Join can also be considered an instance of operator pushdown.

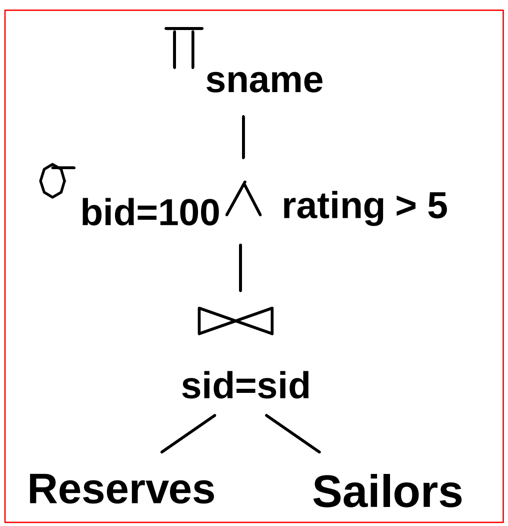


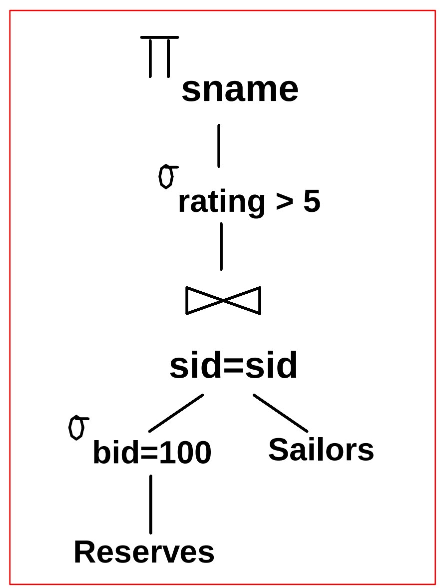


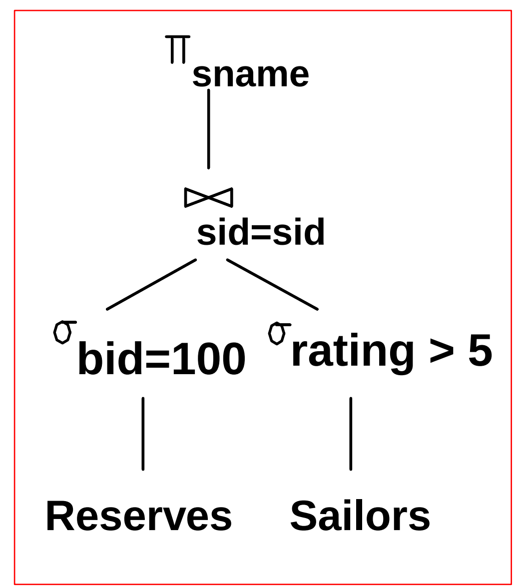
Can you push this projection down?

Ensuring Correctness

- Different RA trees can be created this way
  - Each tree yields a different query execution plan
- Some pushdowns are correct, some are not
  - Humans can distinguish, but DBMSs?
  - Then, how does a DBMS ensure correctness?







Equivalence Rules

- Query optimizer uses equivalence rules of RA to find equivalent expressions for a given query
- Two RA expressions are called equivalent if they produce the same result on all possible instances of relations

- Equivalence rules allow us to:
  - Push selections and projections ahead of joins
  - Combine selections and cross products into joins
  - Choose different operation/join orders, etc.

- There are many equivalence rules, let's see a small subset of them as examples

Equivalence Rules

- Cascading of selections
  - Allows us to combine several selections into one
  - Or replace one selection with several smaller selections

$$\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}\left(\ldots \; \sigma_{cn}(R)\right)$$

- Commutation of selections
  - Allows us to test selection conditions in different order

$$\sigma_{c1}\left(\sigma_{c2}(R)\right) \equiv \sigma_{c2}\left(\sigma_{c1}(R)\right)$$

Equivalence Rules

- Joins and cross products are commutative

$$(R \times S) \equiv (S \times R)$$

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

- Joins and cross products are also associative

$$R \times (S \times T) \equiv (R \times S) \times T$$

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

It follows: $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

This means the query optimizer can choose any order it prefers to perform the joins!

Equivalence Rules

- Selection + cross product -> Join

$$R \bowtie_{c} T \equiv \sigma_c(R \times S)$$

- Commutation of selection with cross product and join:
  - If **c** appears only in **R** but not **S**

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

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

Equivalence Rules

- Following commutation of selection w/ cross product:

$$\sigma_c(R \times S) \equiv \sigma_{c1 \wedge c2 \wedge c3}(R \times S)$$

$$\equiv \sigma_{c1}(\sigma_{c2}(\sigma_{c3}(R \times S)))$$

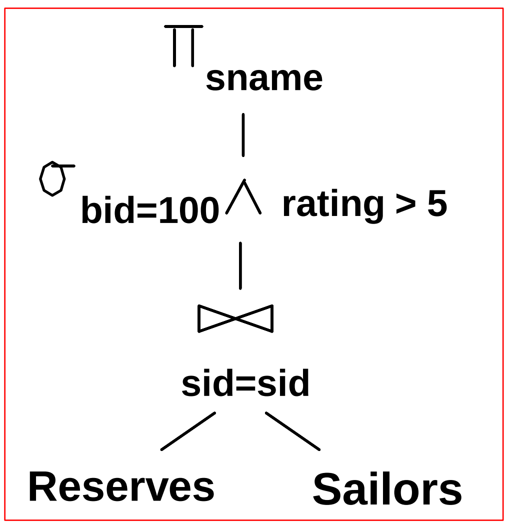
$$\equiv \sigma_{c1}(\sigma_{c2}(R) \times \sigma_{c3}(S))$$

This says we can push part of the selection condition ***c*** ahead of the cross-product

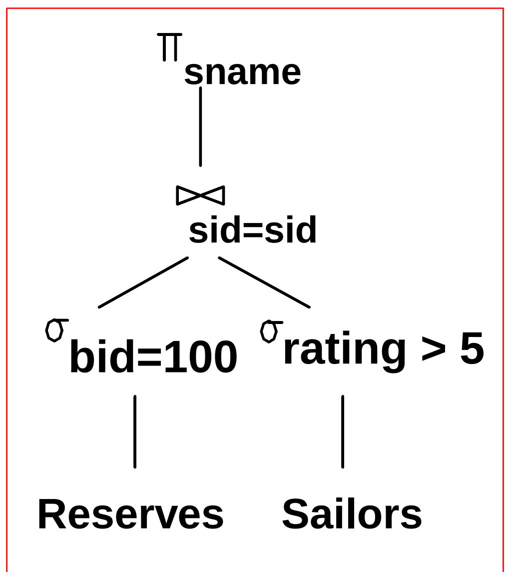
It holds for joins as well!

Cost Estimation

- Say that we have a bunch of different query plans (expressed as different RA trees). We want to pick the one with the lowest cost, i.e., the one that is fastest.
  - How can you estimate cost without executing the plan?



One large join +
one selection



Two selections + one smaller join

Is the version with operator pushdown always faster?

Cost Estimation

- Cost estimation depends on mainly two factors:
  - How fast can you implement an operator?
    - How fast is selection? Join? Projection?
  - What are the sizes of the operator inputs and outputs?
    - Early selections will reduce how much of the data? 10% or 95%?
    - This is called the **reduction factor** (or **selectivity**)
    - The DBMS can **estimate** reduction factors according to available indexes or summary statistics (e.g., histograms)

Implementing Operators

- We'll study the implementation of the following operators:
  - ***Selection*** **(** σ **)**
  - ***Projection*** **(** π **)**
  - ***Join*** **(** $\bowtie$ **)**
  - ***Set operators (union, intersect, difference)***
- Each operator returns a **relation**, therefore an RA tree is a composition of these operators
- Other SQL functionalities (e.g., SUM/MAX/MIN, GROUP BY) can be implemented in linear time
  - We won't cover them – but it's easy to verify

Selection

- Consider the following, basic selection:
  - No conjunctions (AND), no disjunctions (OR)

```
SELECT *
FROM Reserves R
WHERE R.rname = 'Joe'
```

- If a hash/tree index is available, use it
- If not, perform a full-table scan

- How about the case with **multiple conjunctions**?

```
SELECT *
FROM Reserves R
WHERE day < 24/3/2015 AND bid=5 AND sid=3
```

Selection

```
SELECT *
FROM Reserves R
WHERE day < 24/3/2015 AND bid=5 AND sid=3
```

- If no index is available for **day**, **bid** or **sid**, you must perform a full-table scan
- If an index is available for one of them, evaluate its condition before the others
  - E.g.: B+ tree index on day is available
  - Use it to retrieve tuples which satisfy day < 24/3/2015
  - Among those tuples, check bid=5 and sid=3

Reduce the amount of linear scanning as much as possible

Selection

```
SELECT  *
FROM  Reserves R
WHERE  day < 8/9/1994 OR rname="Alice_res"
```

- Case with **disjunctions** (OR):
  - Detailed algorithms exist, we won't cover them
- Two intuitive remarks:
  - Use indexes as much as possible
    - E.g., if there's an index on day and an index on rname, use them to individually compute day < 8/9/1994 condition and rname="Alice_res" condition
    - Subsequently, union their results
  - Full-table scan may become unavoidable
    - E.g., index on day exists but no index on rname
    - We can't avoid full-table scan to evaluate rname="Alice_res"

Projection

- Consider the following query which implies a projection:

```
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```

- How can we evaluate this query?
  - First remove unwanted attributes
  - Then eliminate any duplicate tuples
    - This is the more difficult step, naively, it is $O(N^2)$
- Two strategies for duplicate elimination:
  - Sorting
  - Hashing

Projection w/ Sorting

SELECT DISTINCT R.sid, R.bid
FROM Reserves R

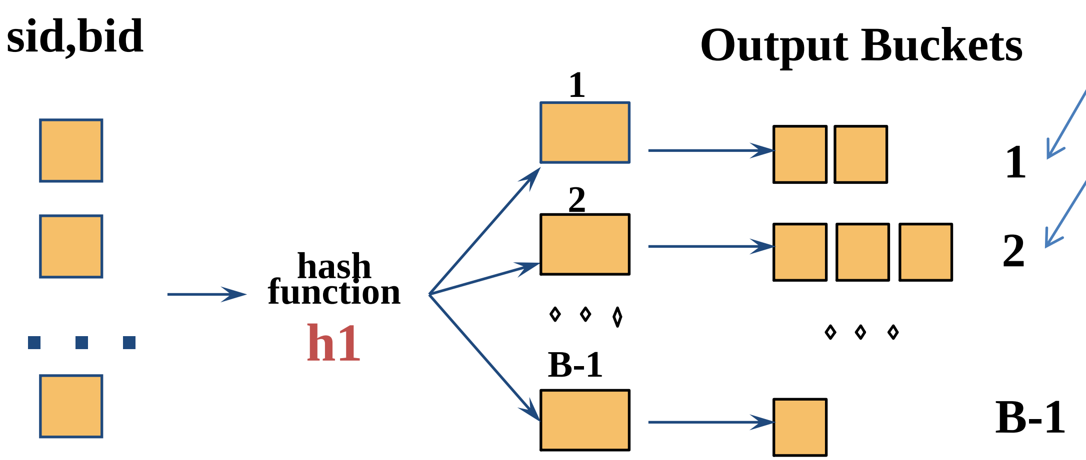
- Produce a set of tuples **S** which contains only **sid** and **bid** attributes
- Sort **S**
- Scan the sorted result, compare adjacent tuples, and discard duplicates

sid	bid
28	103
28	103
31	101
31	101
31	102
58	103

Projection w/ Hashing

SELECT DISTINCT R.sid, R.bid
FROM Reserves R

Two tuples that belong to different buckets are guaranteed not to be duplicates



Sorting vs Hashing

- Which approach to use?

- In terms of average-case time complexity (wrt data size), hashing is better: **O(N)** vs **O(NlogN)**

- Sorting is better when:
  - There is a high frequency of duplicates
  - Hash function causes too many hash collisions
  - Side benefit of sorting: the result is sorted, which can benefit downstream operators (such as joins)

Join

- Consider the following query which implies a join:

```
SELECT *
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
```

- We will study three join algorithms:
  - Nested Loop Join **O(|R|*|S|)**
  - Sort-Merge Join **O(|R|*log|R| + |S|*log|S|)**
    - If relations are pre-sorted, then complexity is: O(|R| + |S|)
  - Hash Join **O(|R| + |S|)**

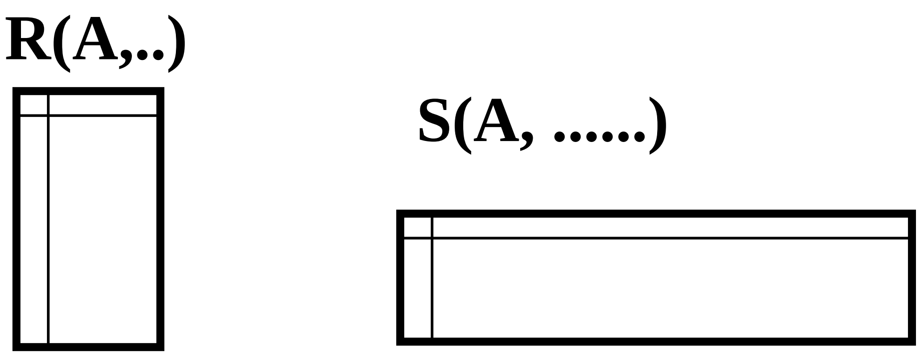
Nested Loop Join

- Implement join as a nested loop:

```
for each tuple r in R
    for each tuple s in S
        if match: print (r, s)
```

Outer Relation

Inner Relation



Sort-Merge Join

SELECT *
FROM Reserves R, Sailors S
WHERE R.sid = S.sid

- Individually sort both relations on the join attribute
  - Sort **R** according to **sid**, sort **S** according to **sid**
  - O(|R|*log|R| + |S|*log|S|)
  - This cost can be avoided if relations have been sorted beforehand (e.g., index or prior projection)
- Scan each relation and merge
  - O(|R|+|S|)
- Works only for equality join conditions

Sort-Merge Join: An Example

= ?

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Sort-Merge Join: An Example

= NO

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Sort-Merge Join: An Example

= ?

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Sort-Merge Join: An Example

= YES

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Output the two tuples

Sort-Merge Join: An Example

= ?

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Sort-Merge Join: An Example

= YES

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Sort-Merge Join: An Example

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

= **YES**

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Output the two tuples

Sort-Merge Join: An Example

$=$?

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Sort-Merge Join: An Example

= NO

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Sort-Merge Join: An Example

=?

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Sort-Merge Join: An Example

= YES

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

<u>sid</u>	<u>bid</u>	<u>day</u>	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

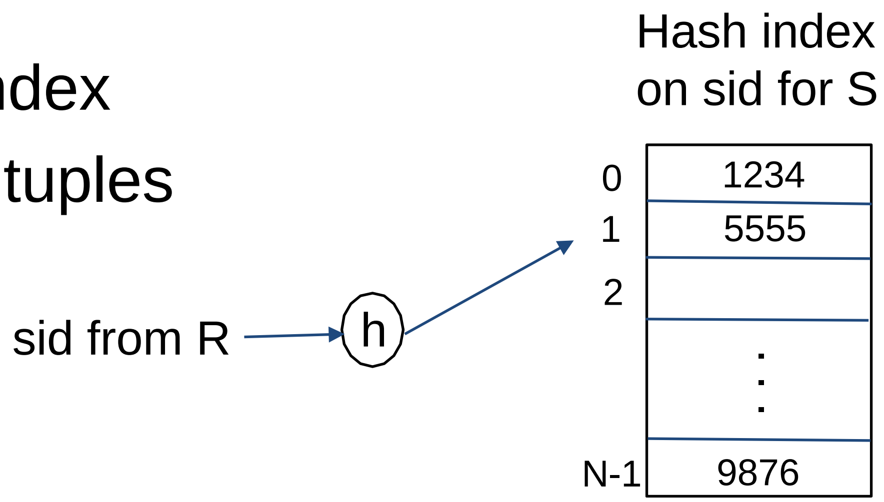
Output the two tuples

Continue the same way!

Hash Join

SELECT *
FROM Reserves R, Sailors S
WHERE R.sid = S.sid

- Create a hash index for **S** using **sid**
  - O(|S|)
- For each tuple r in R:
  - hash(r) and check the index
  - If match, output the two tuples
  - O(|R|) * O(1)
- Total: **O(|R| + |S|)**



Set Operators

- UNION: **R U S**
  - Vertically concatenate **R** and **S**
  - Eliminate duplicates (re-use ideas from projection)

- INTERSECTION: **R ∩ S**
  - Create hash index for **S**
  - For each tuple in **R**: check if it’s in that hash index; if so, include it in the output

- DIFFERENCE (EXCEPT): **R – S**
  - Create hash index for **S**
  - For each tuple in **R**: check if it’s in that hash index; if not, include it in the output