Database Applications (15-415)

DBMS Internals- Part IX
Lecture 20, April 5, 2015

Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

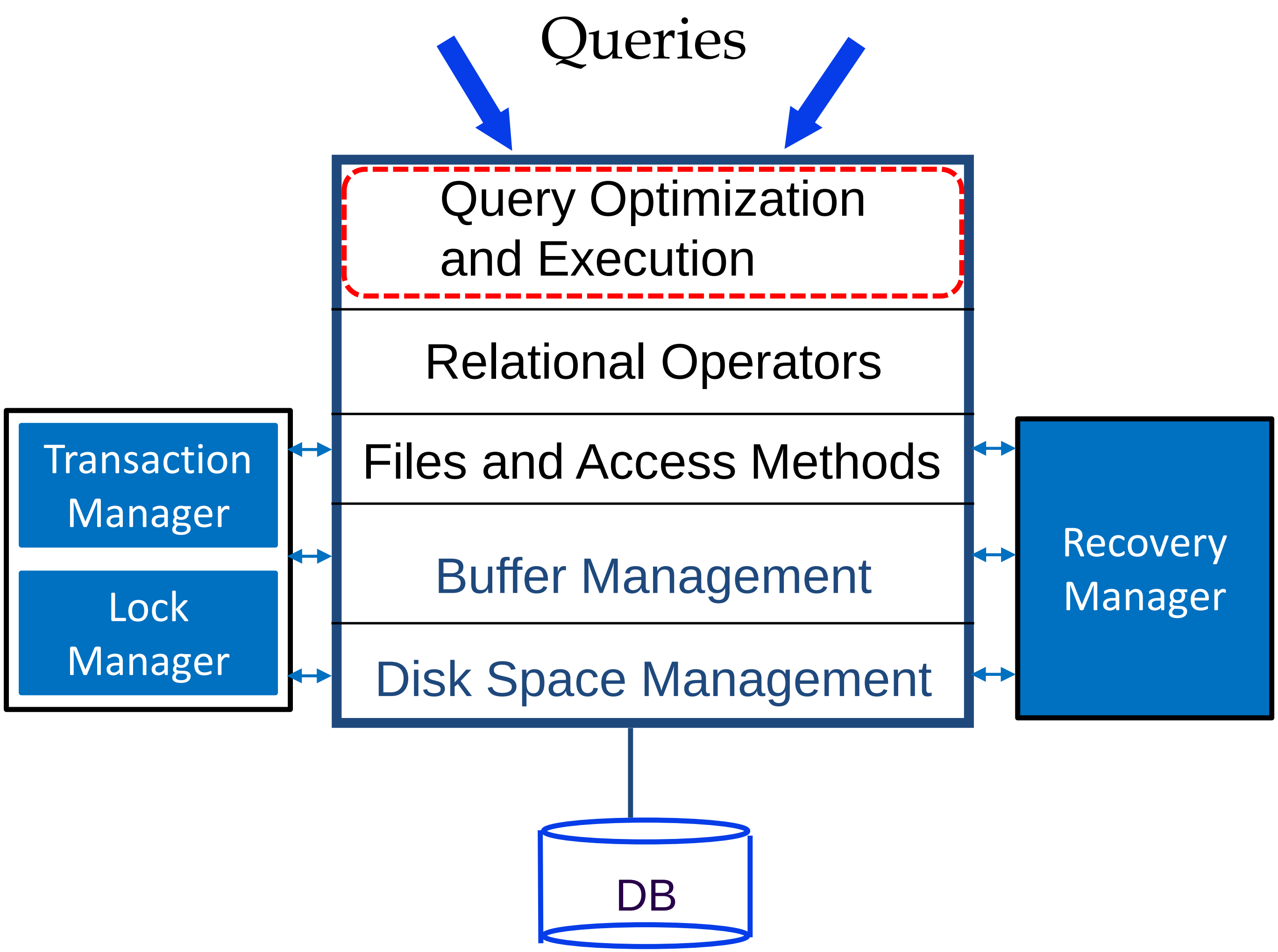
Today…

- Last Session:
  - DBMS Internals- Part VIII
    - Algorithms for Relational Operations (*Cont'd*)

- Today's Session:
  - DBMS Internals- Part IX
    - Query Optimization

- Announcements:
  - PS4 is now posted. It is due on Sunday, April 12 by midnight
  - Quiz II is on Thursday, April 9th (all concepts covered after the midterm are included)

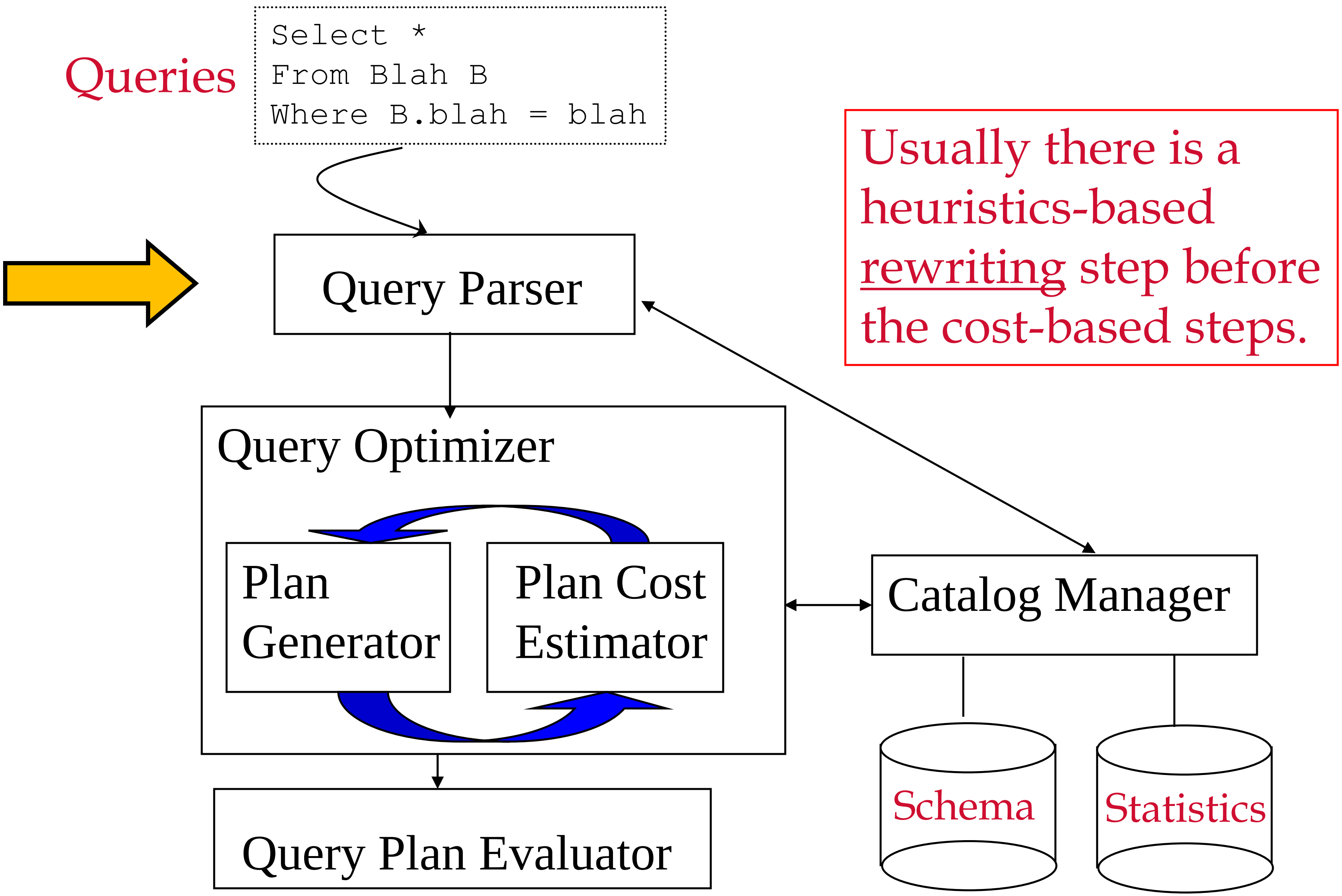
DBMS Layers


Queries
Query Optimization and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction Manager
Lock Manager
Recovery Manager
DB

Outline

- A Brief Primer on Query Optimization ✓
- Evaluating Query Plans
- Relational Algebra Equivalences
- Estimating Plan Costs
- Enumerating Plans

Cost-Based Query Sub-System



Query Optimization Steps

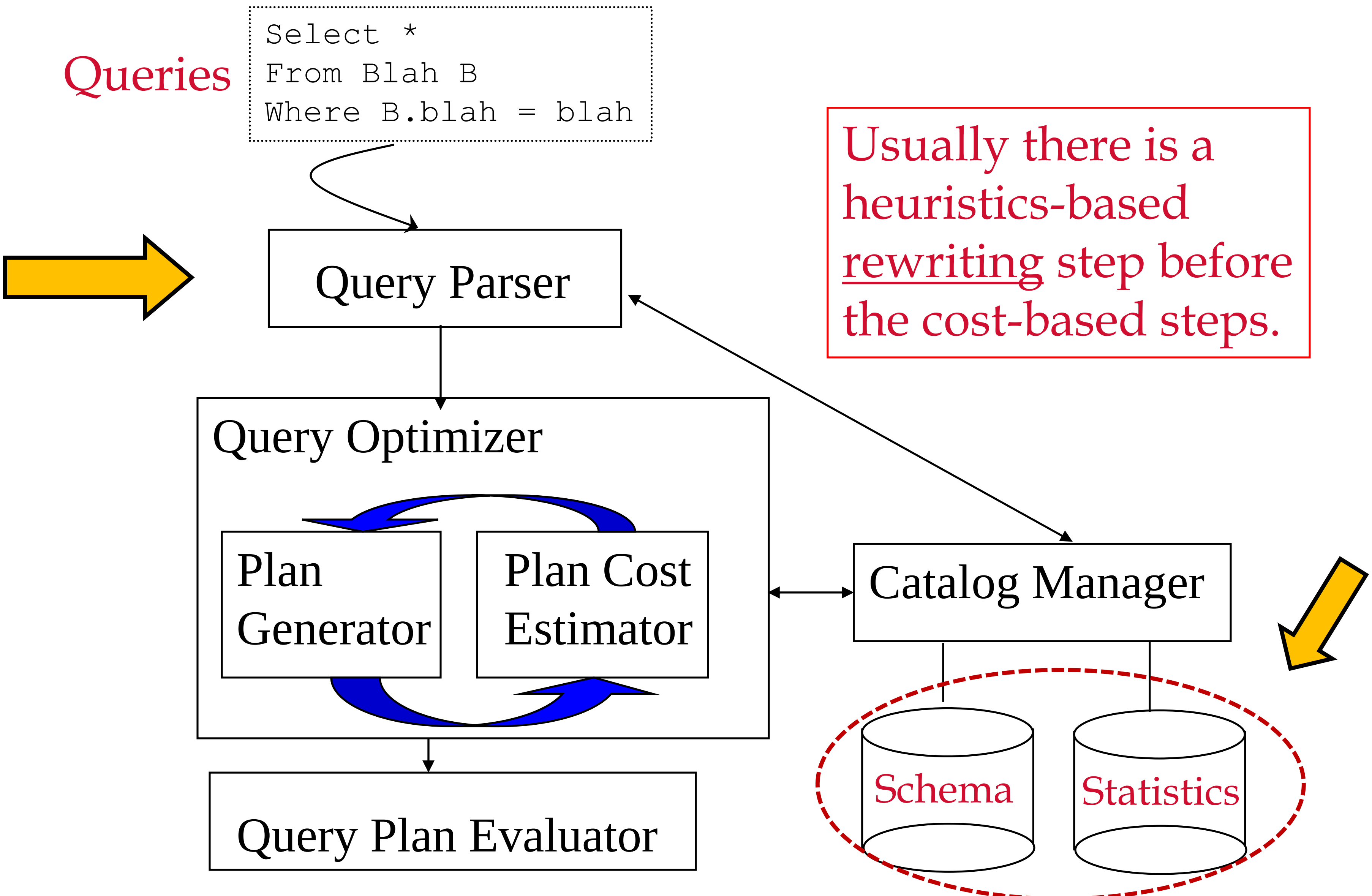
- Step 1: Queries are parsed into internal forms (e.g., parse trees)
- Step 2: Internal forms are transformed into 'canonical forms' (syntactic query optimization)
- Step 3: A ***subset*** of alternative plans are enumerated
- Step 4: Costs for alternative plans are estimated
- Step 5: The query evaluation plan with the ***least estimated cost*** is picked

Required Information to Evaluate Queries

- To estimate the costs of query plans, the query optimizer examines the system catalog and retrieves:
  - Information about the types and lengths of fields
  - Statistics about the referenced relations
  - Access paths (indexes) available for relations

- In particular, the *Schema* and *Statistics* components in the Catalog Manager are inspected to find a good enough query evaluation plan

Cost-Based Query Sub-System



Catalog Manager: The Schema

- What kind of information do we store at the Schema?
  - Information about tables (e.g., table names and integrity constraints) and attributes (e.g., attribute names and types)
  - Information about indices (e.g., index structures)
  - Information about users

- Where do we store such information?
  - In tables, hence, can be queried like any other tables
  - For example: Attribute_Cat (attr_name: **string**, rel_name: **string**; type: **string**; position: **integer**)

Catalog Manager: Statistics

- What would you store at the Statistics component?
  - NTuples(R): # records for table R
  - NPages(R): # pages for R
  - NKeys(I): # distinct key values for index I
  - INPages(I): # pages for index I
  - IHeight(I): # levels for I
  - ILow(I), IHigh(I): range of values for I
  - ...
- Such statistics are important for estimating plan costs and result sizes (*to be discussed shortly!*)

SQL Blocks

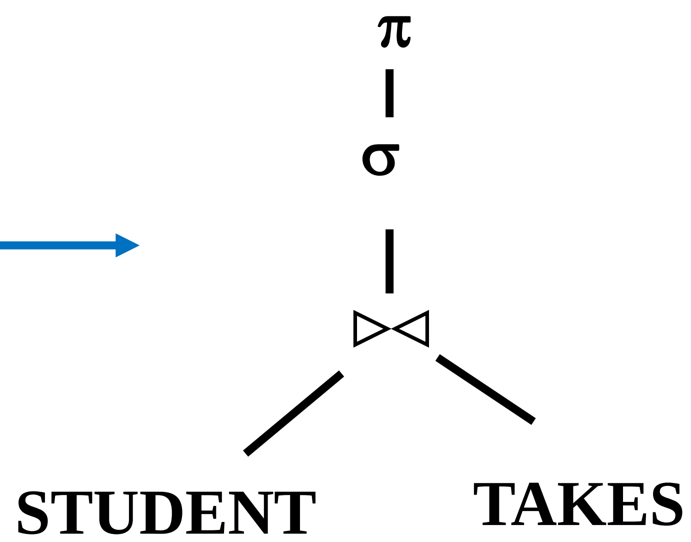
- SQL queries are optimized by *decomposing* them into a collection of smaller units, called blocks

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

- A typical relational query optimizer concentrates on optimizing a single block at a time

Translating SQL Queries Into Relational Algebra Trees

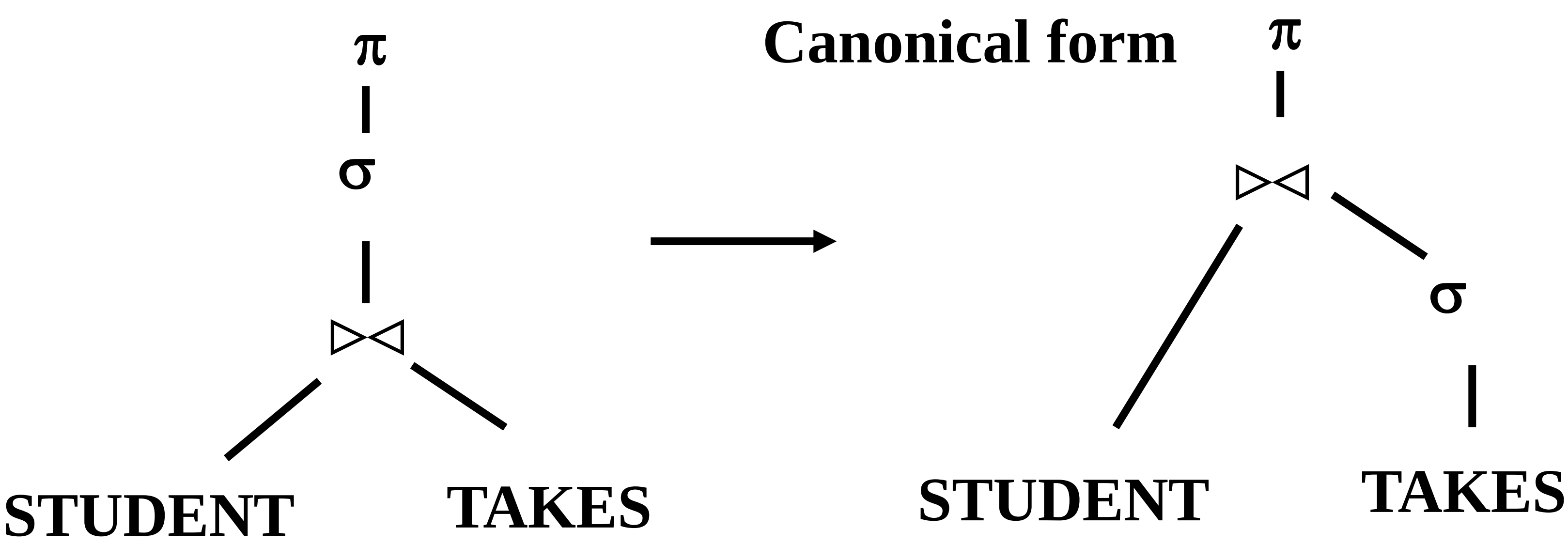
select name
from STUDENT, TAKES
where c-id=‘415’ and
STUDENT.ssn=TAKES.ssn



- An SQL block can be thought of as an algebra expression containing:
  - A cross-product of all relations in the FROM clause
  - Selections in the WHERE clause
  - Projections in the SELECT clause

- Remaining operators can be carried out on the result of such SQL block

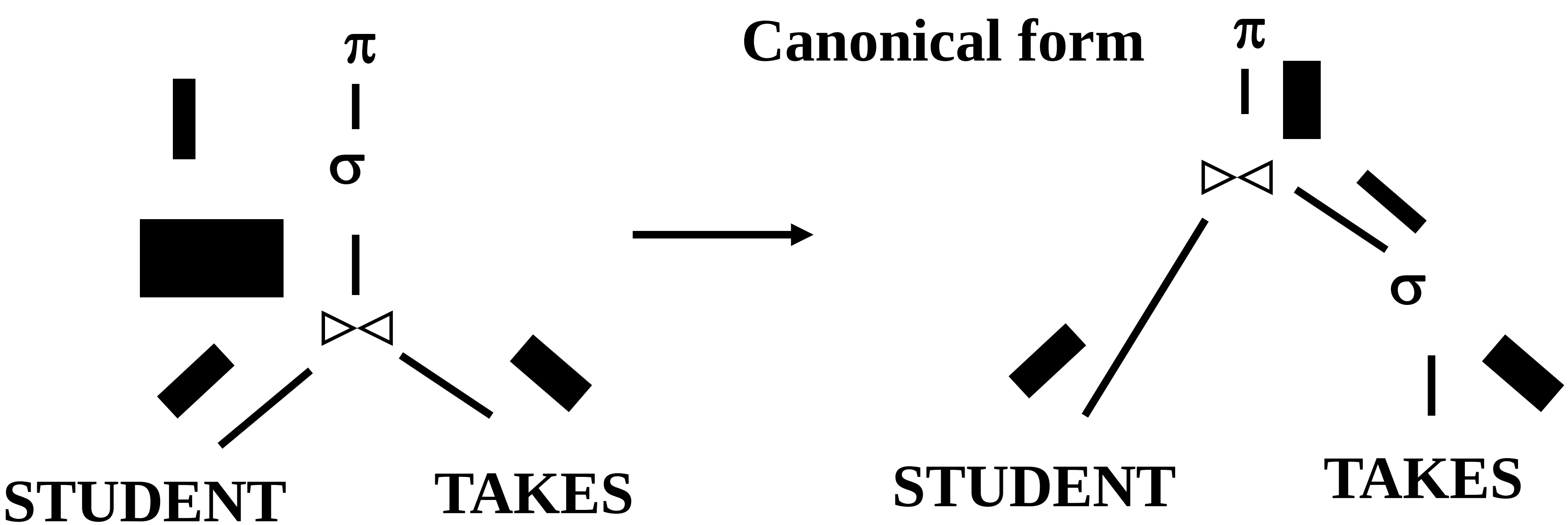
Translating SQL Queries Into Relational Algebra Trees (*Cont'd*)



Still the same result!

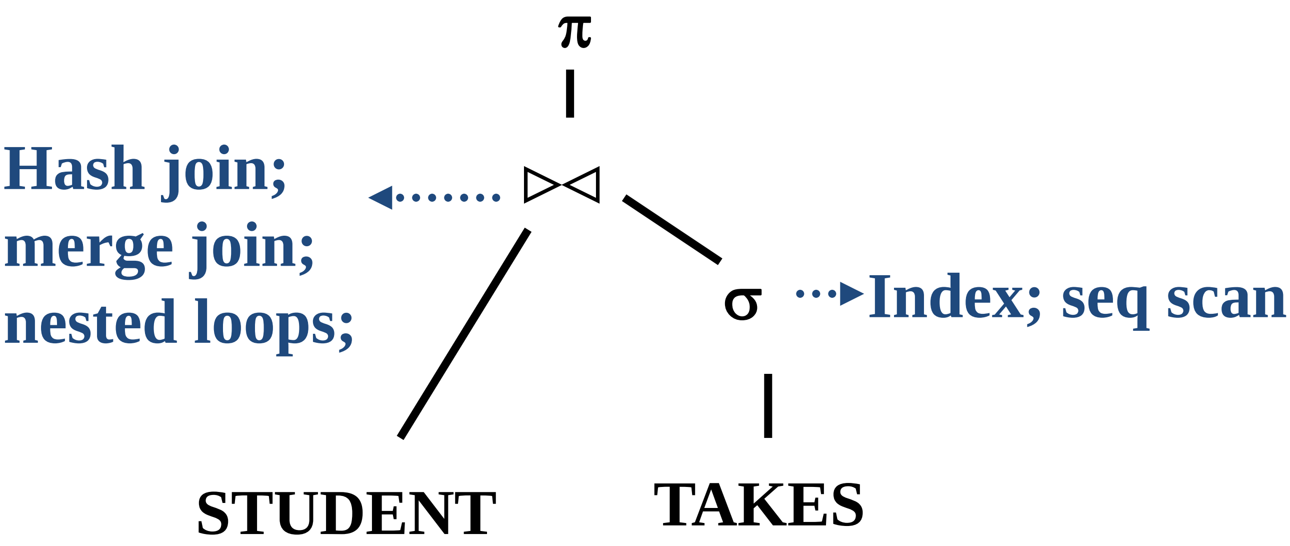
How can this be guaranteed?

Translating SQL Queries Into Relational Algebra Trees (*Cont'd*)



OBSERVATION: try to perform selections and projections early!

Translating SQL Queries Into Relational Algebra Trees (*Cont'd*)



How to evaluate a query plan (as opposed to evaluating an operator)?

Outline

- A Brief Primer on Query Optimization
- Evaluating Query Plans ✓
- Relational Algebra Equivalences
- Estimating Plan Costs
- Enumerating Plans

Query Evaluation Plans

- A *query evaluation plan* (or simply a *plan*) consists of an *extended* relational algebra tree (or simply a tree)

- A plan tree consists of annotations at each node indicating:
  - The access methods to use for each relation
  - The implementation method to use for each operator

- Consider the following SQL query ***Q***:

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

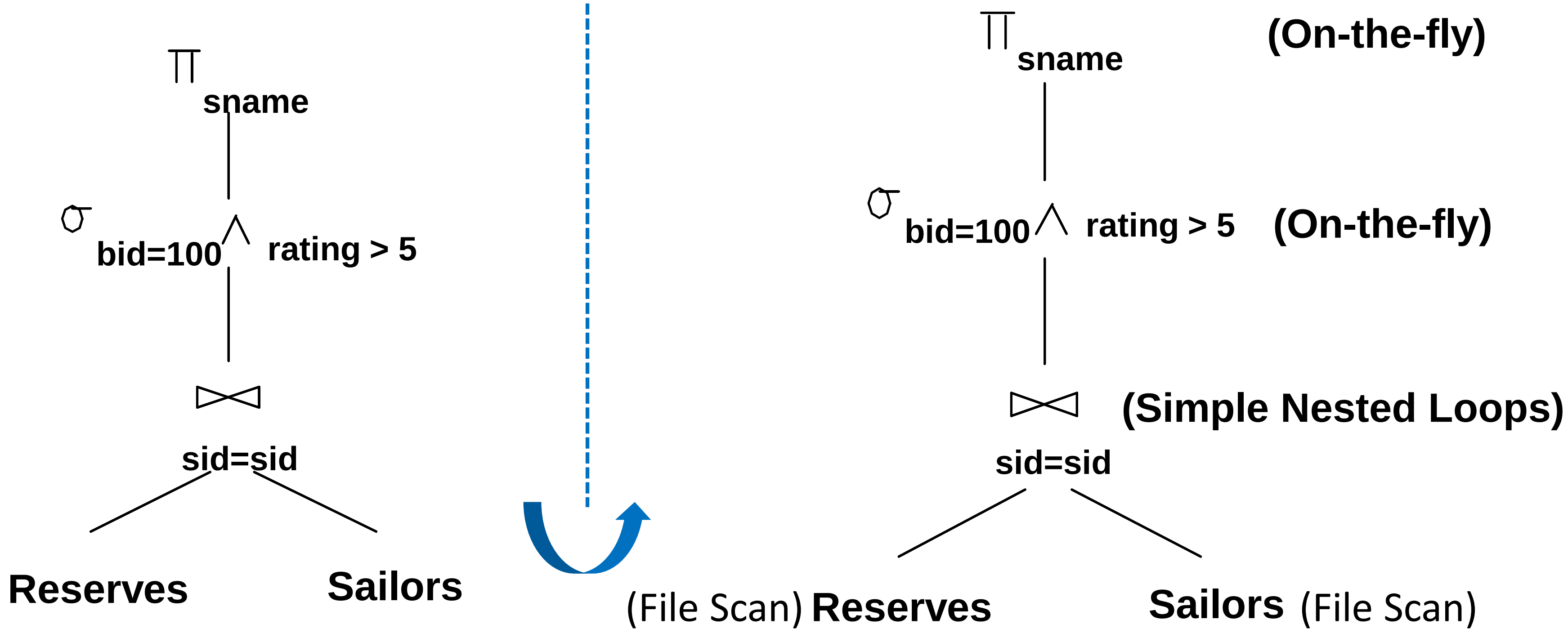
What is the corresponding RA of ***Q***?

Query Evaluation Plans (Cont'd)

- ***Q*** can be expressed in relational algebra as follows:

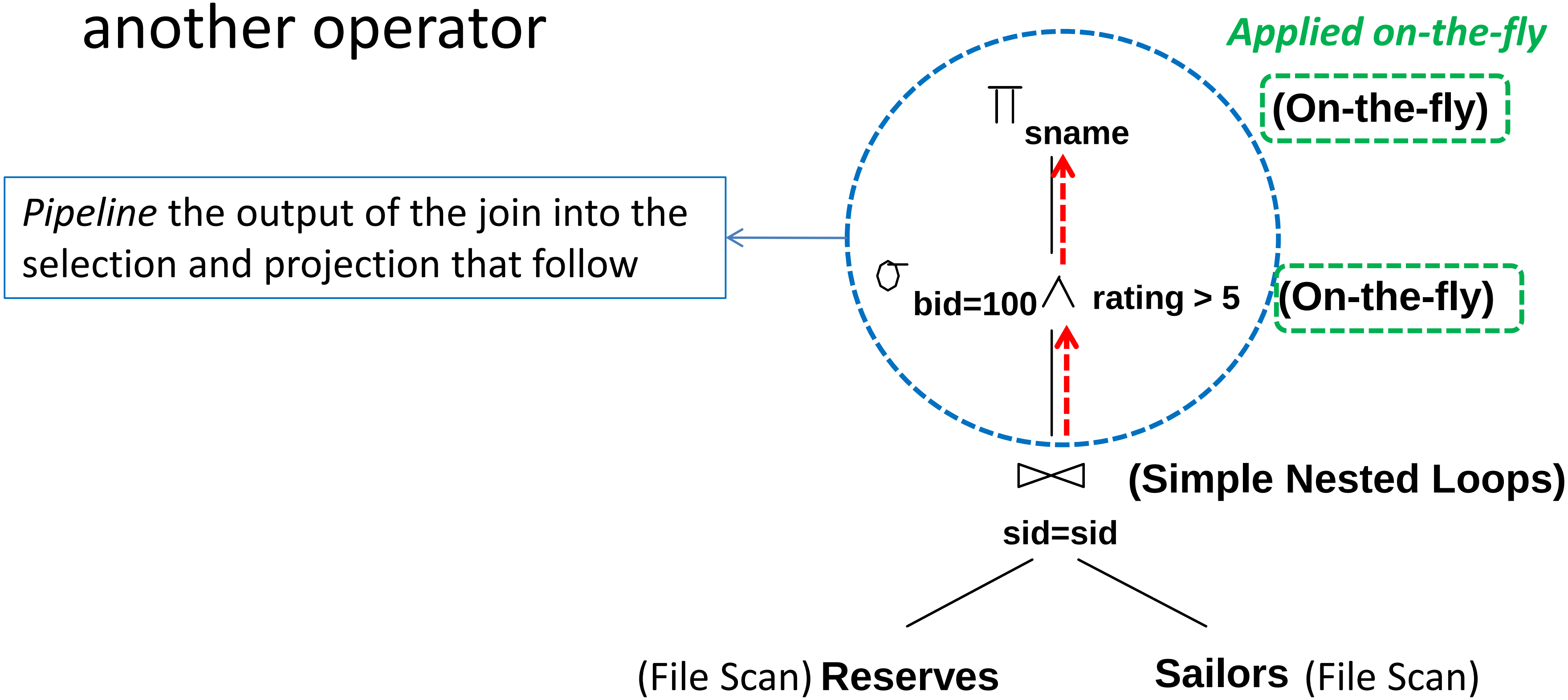
$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$

A RA Tree: **An Extended RA Tree:**



Pipelining vs. Materializing

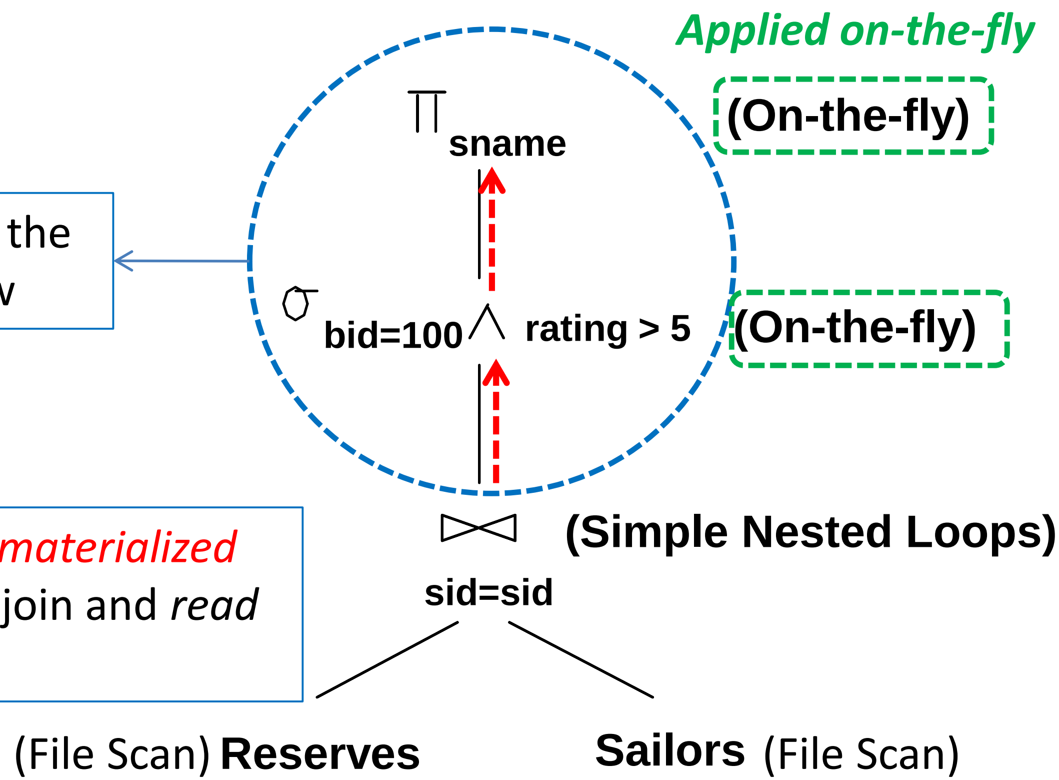
- When a query is composed of several operators, the result of one operator can sometimes be *pipelined* to another operator



Pipelining vs. Materializing

- When a query is composed of several operators, the result of one operator can sometimes be *pipelined* to another operator

Pipeline the output of the join into the selection and projection that follow



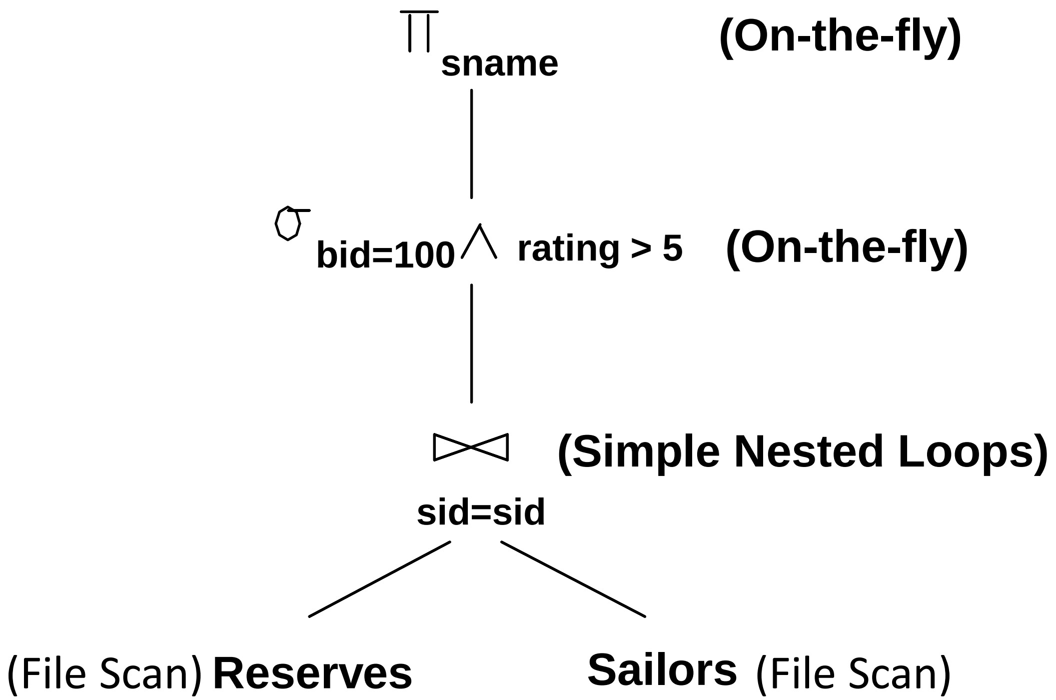
≠

In contrast, a temporary table can be *materialized* to hold the *intermediate result* of the join and *read back* by the selection operation!

Pipelining *can* significantly save I/O cost!

The I/O Cost of the *Q* Plan

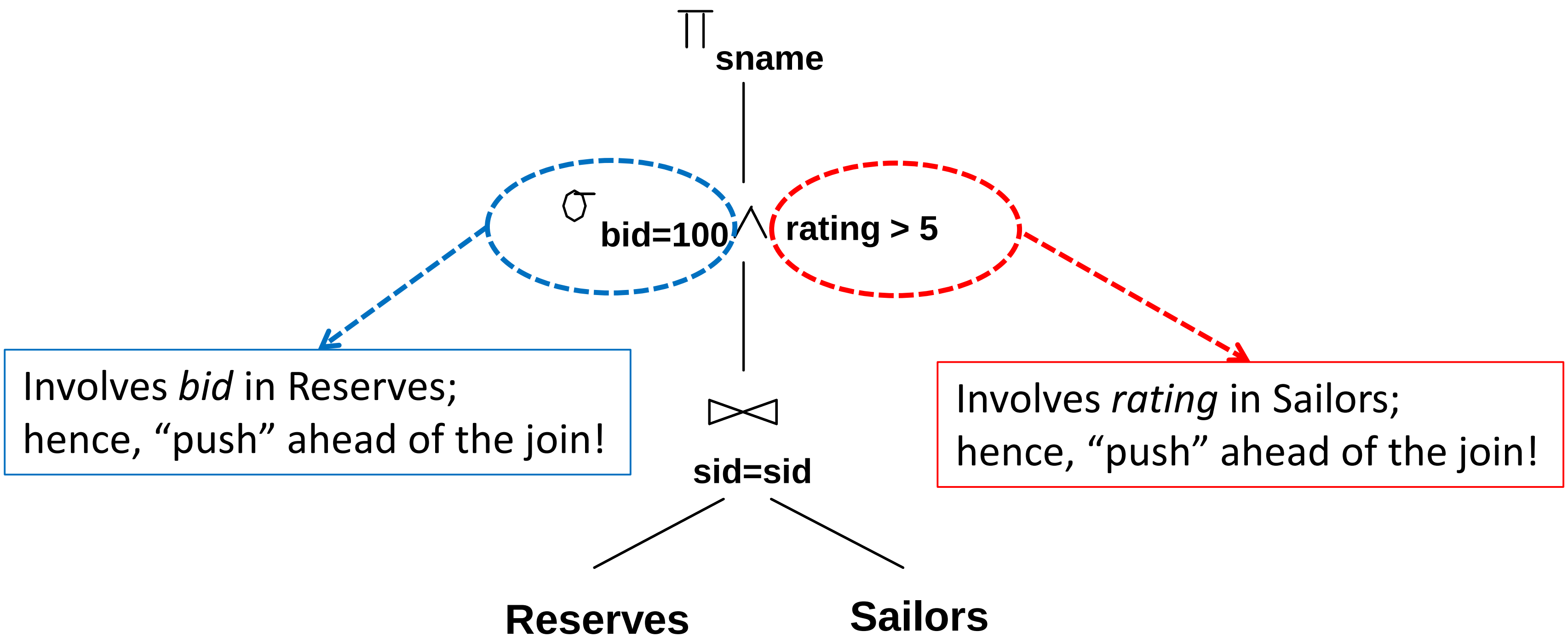
- What is the I/O cost of the following evaluation plan?



- ✓ The cost of the join is 1000 + 1000 * 500 = 501,000 I/Os (assuming page-oriented Simple NL join)
- ✓ The selection and projection are done on-the-fly; hence, do not incur additional I/Os

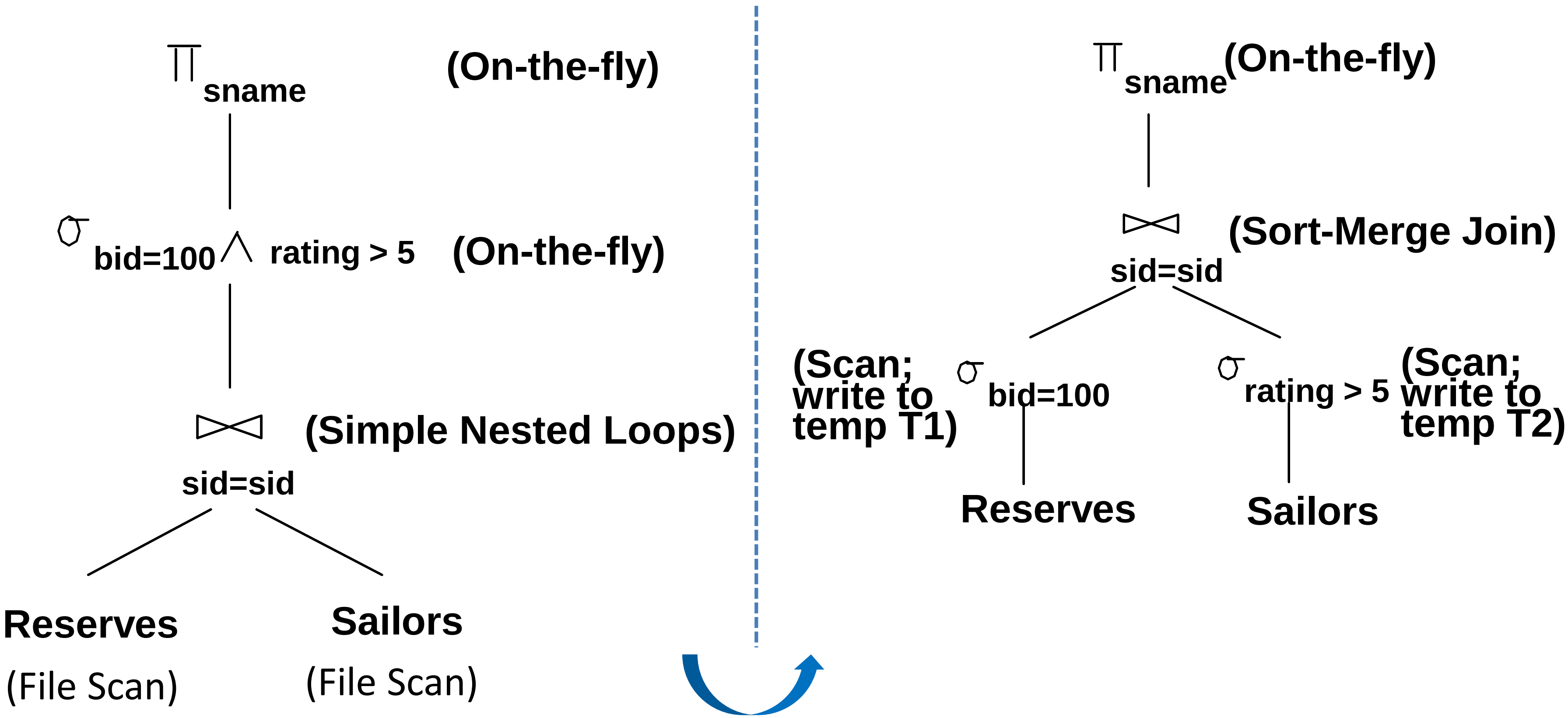
Pushing Selections

- How can we reduce the cost of a join?
  - By reducing the sizes of the input relations!



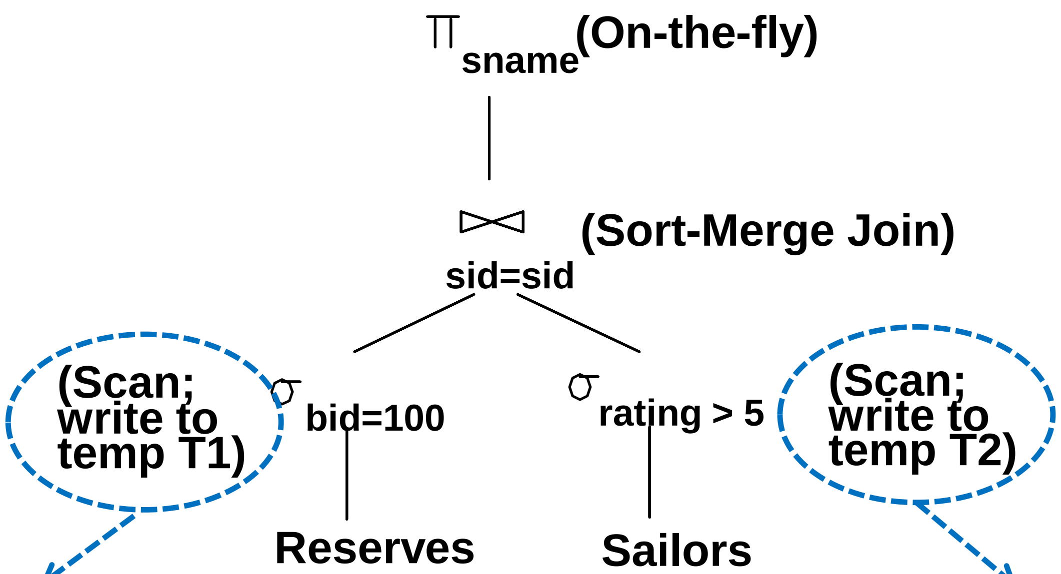
Pushing Selections

- How can we reduce the cost of a join?
  - By reducing the sizes of the input relations!



The I/O Cost of the *New* ***Q*** Plan

- What is the I/O cost of the following evaluation plan?



Cost of Scanning Reserves = 1000 I/Os
Cost of Writing T1 = 10* I/Os (*later*)

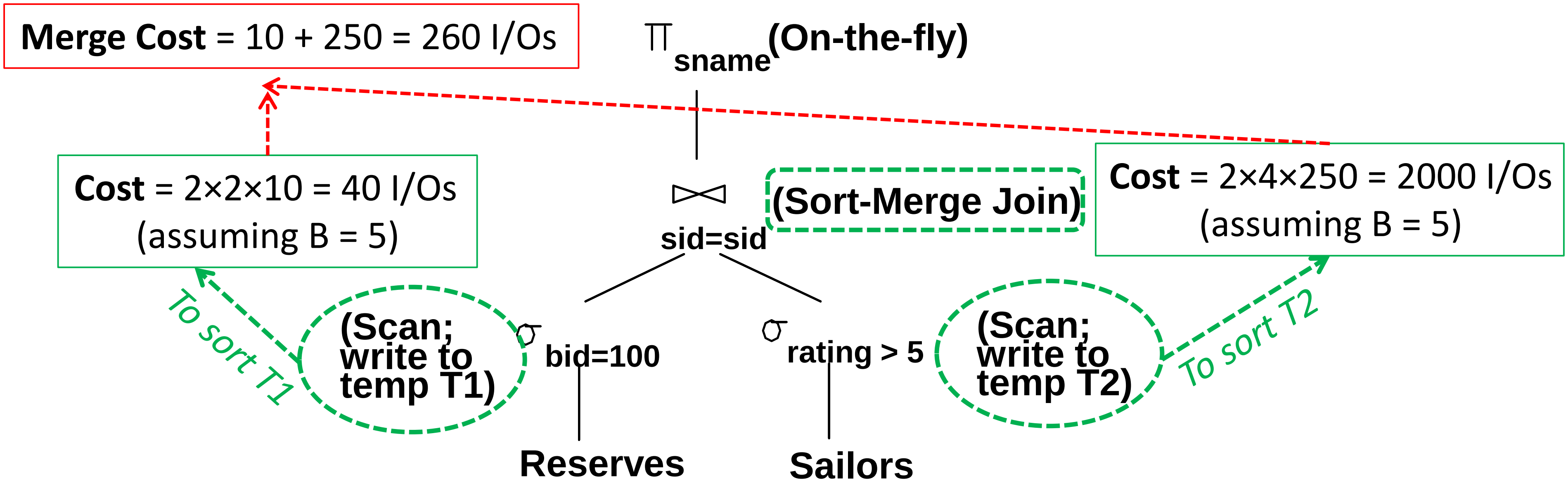
Cost of Scanning Sailors = 500 I/Os
Cost of Writing T2 = 250* I/Os (*later*)

*Assuming 100 boats and uniform distribution of reservations across boats.

*Assuming 10 ratings and uniform distribution over ratings.

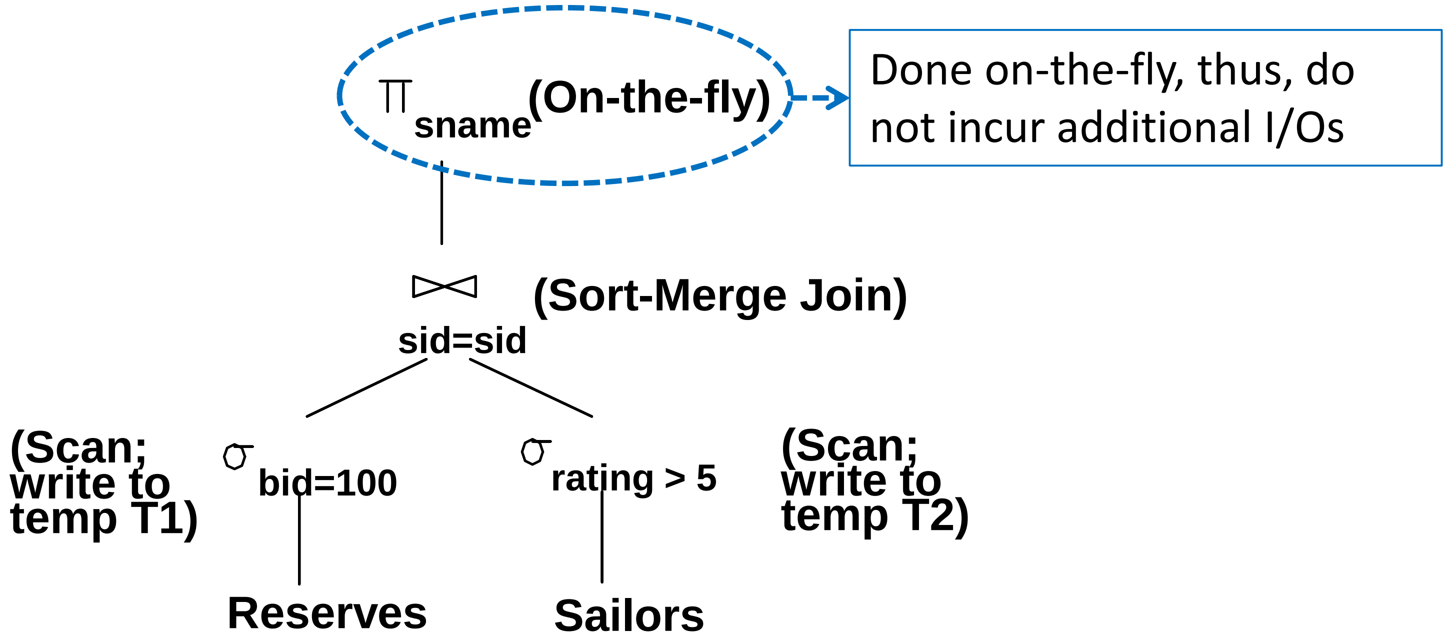
The I/O Cost of the *New* ***Q*** Plan

- What is the I/O cost of the following evaluation plan?



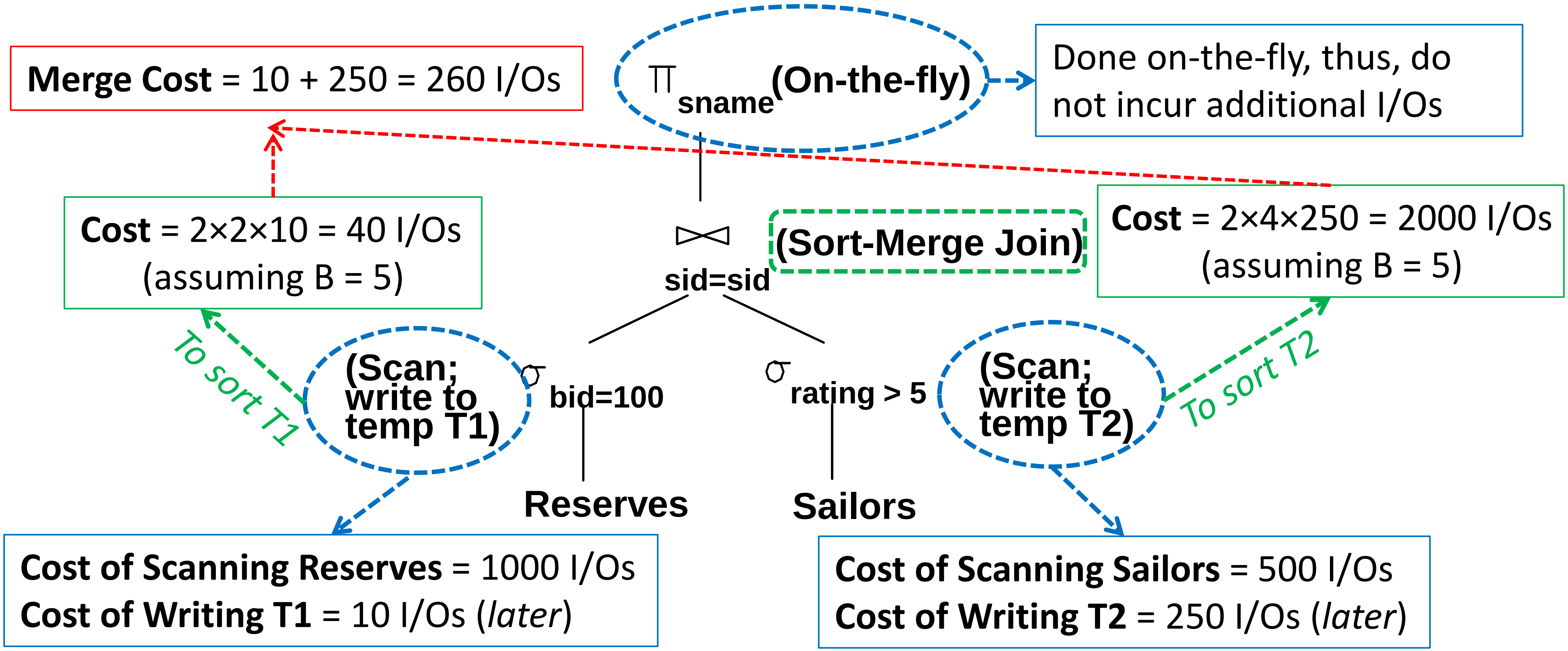
The I/O Cost of the *New* ***Q*** Plan

- What is the I/O cost of the following evaluation plan?



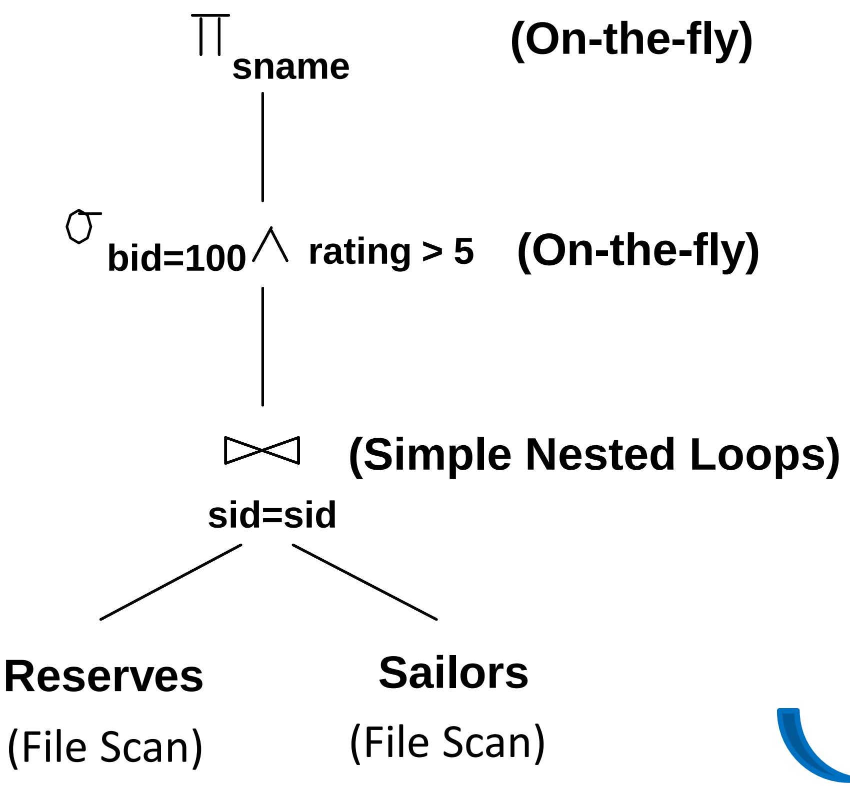
The I/O Cost of the *New* ***Q*** Plan

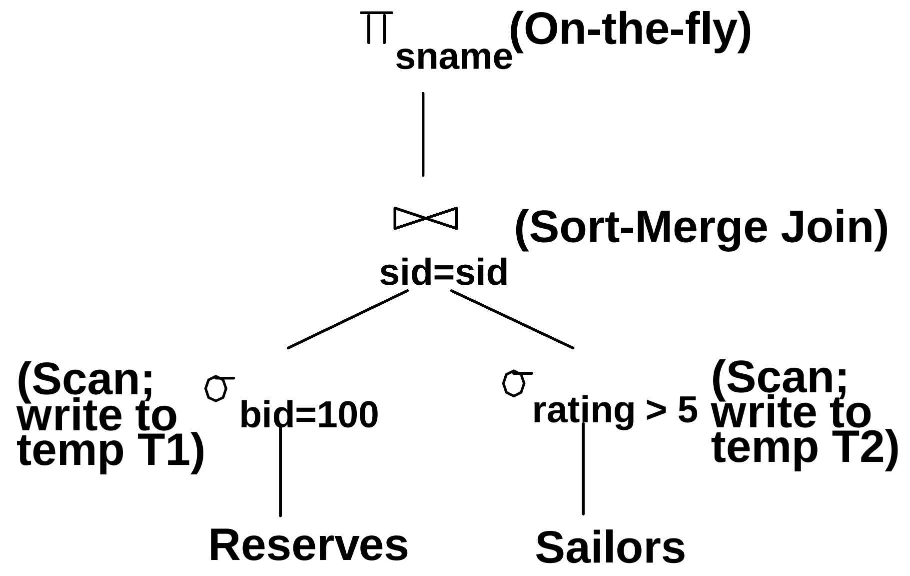
- What is the I/O cost of the following evaluation plan?



Total Cost = 1000 + 10 + 500 + 250 + 40 + 2000 + 260 = 4060 I/Os

The I/O Costs of the *Two* ***Q*** Plans





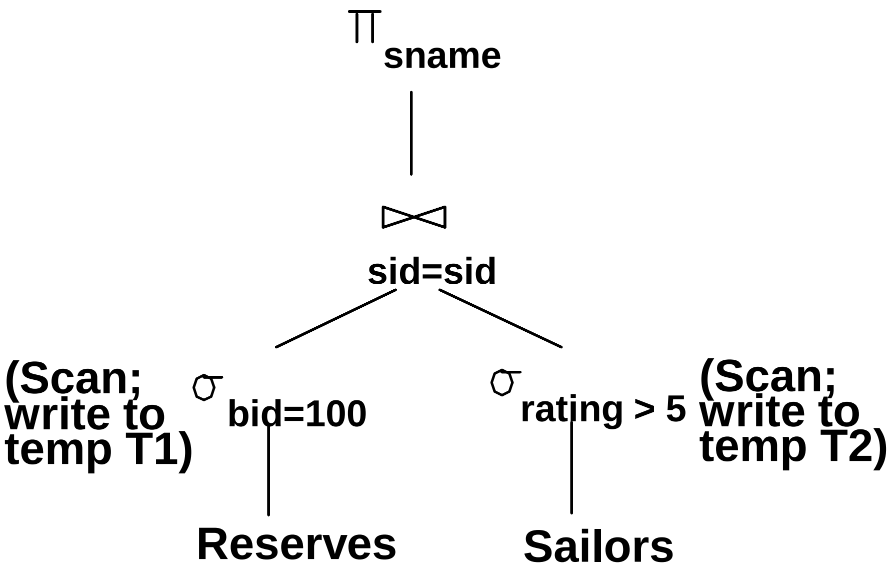
Total Cost = 501, 000 I/Os

Total Cost = 4060 I/Os

Pushing Projections

- How can we reduce the cost of a join?
  - By reducing the sizes of the input relations!

- Consider (again) the following plan:



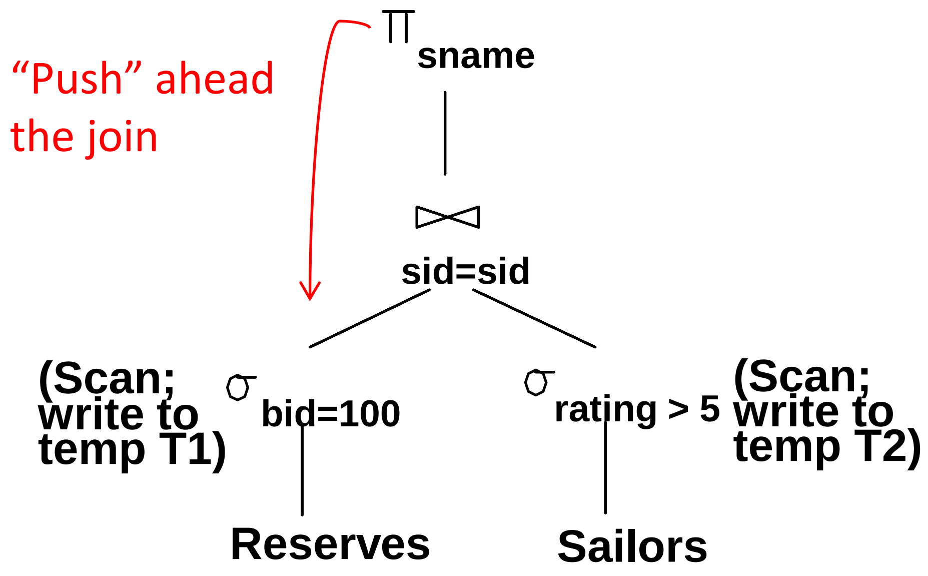
- What are the attributes required from T1 and T2?
  - *Sid* from T1
  - *Sid* and sname from T2

Hence, as we scan Reserves and Sailors we can also remove unwanted columns (i.e., "Push" the projections ahead of the join)!

Pushing Projections

- How can we reduce the cost of a join?
  - By reducing the sizes of the input relations!

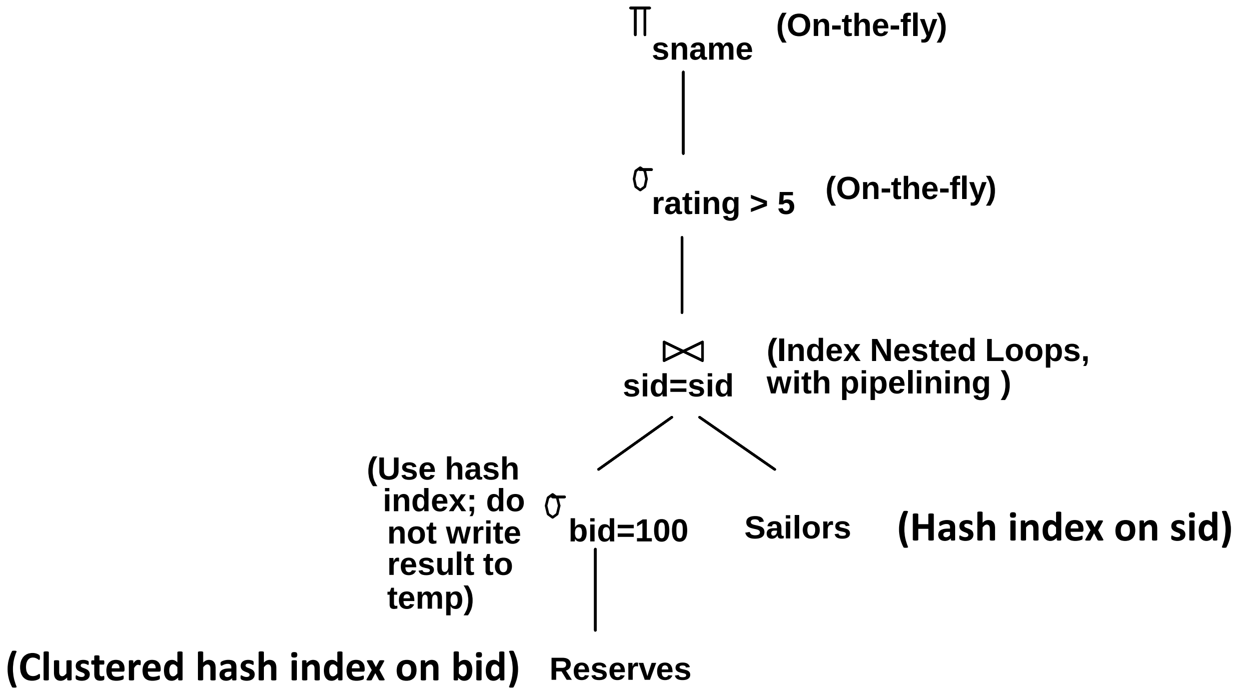
- Consider (again) the following plan:



The cost after applying this heuristic can become 2000 I/Os (as opposed to 4060 I/Os with only pushing the selection)!

Using Indexes

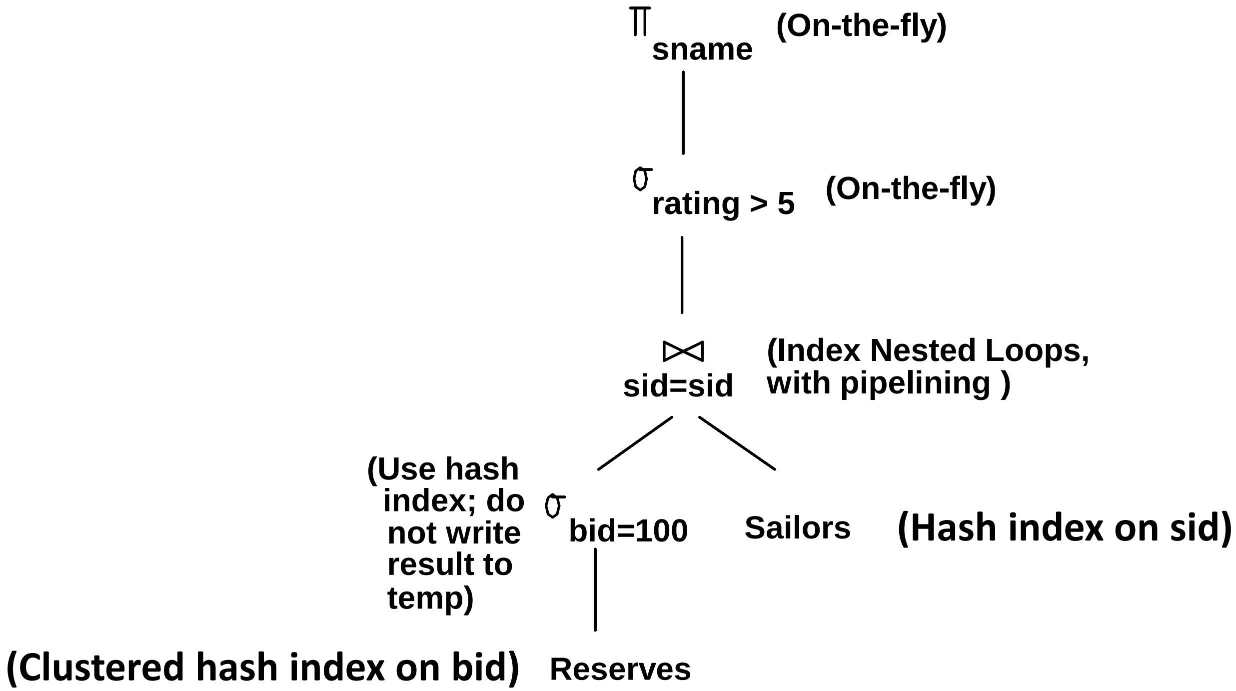
- What if indexes are available on Reserves and Sailors?



- ✓ With clustered index on *bid* of Reserves, we get 100,000/100 = 1000 tuples (assuming 100 boats and uniform distribution of reservations across boats)
- ✓ Since the index is clustered, the 1000 tuples appear consecutively within the same bucket; thus # of pages = 1000/100 = 10 pages

Using Indexes

- What if indexes are available on Reserves and Sailors?



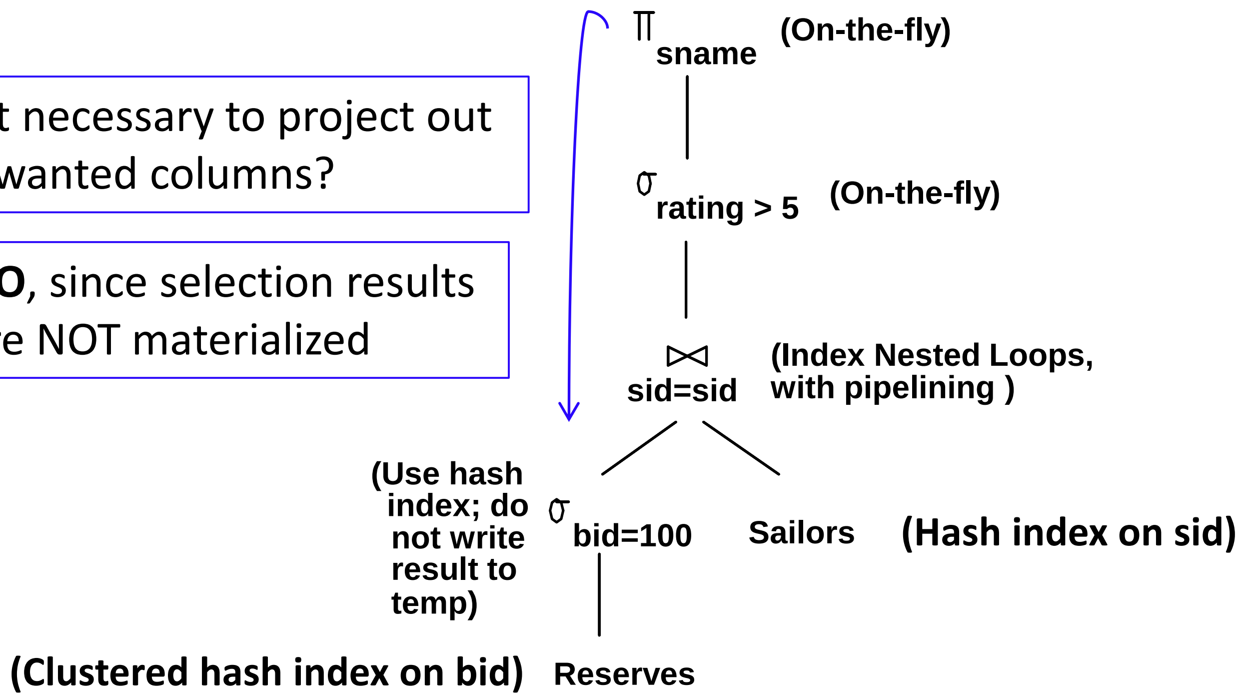
- ✓ For each selected Reserves tuple, we can retrieve matching Sailors tuples using the hash index on the *sid* field
- ✓ Selected Reserves tuples need not be materialized and the join result can be pipelined!
- ✓ For each tuple in the join result, we apply rating > 5 and the projection of *sname* on-the-fly

Using Indexes

- What if indexes are available on Reserves and Sailors?

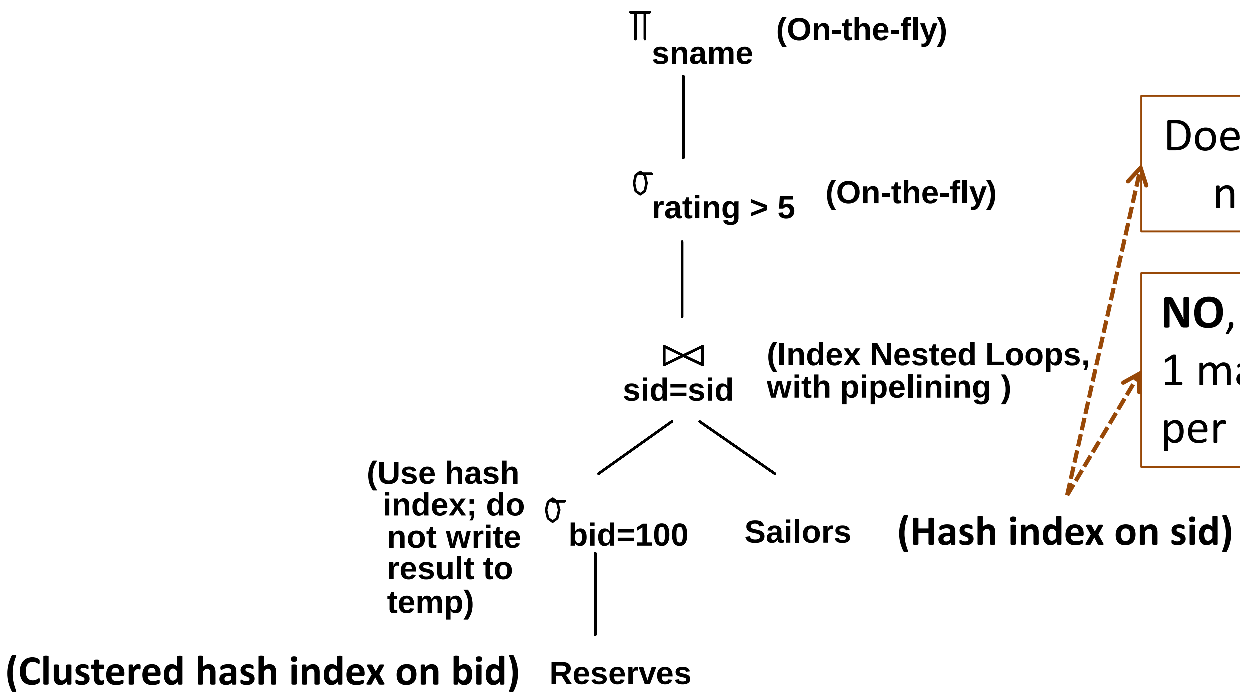
Is it necessary to project out unwanted columns?

NO, since selection results are NOT materialized



Using Indexes

- What if indexes are available on Reserves and Sailors?

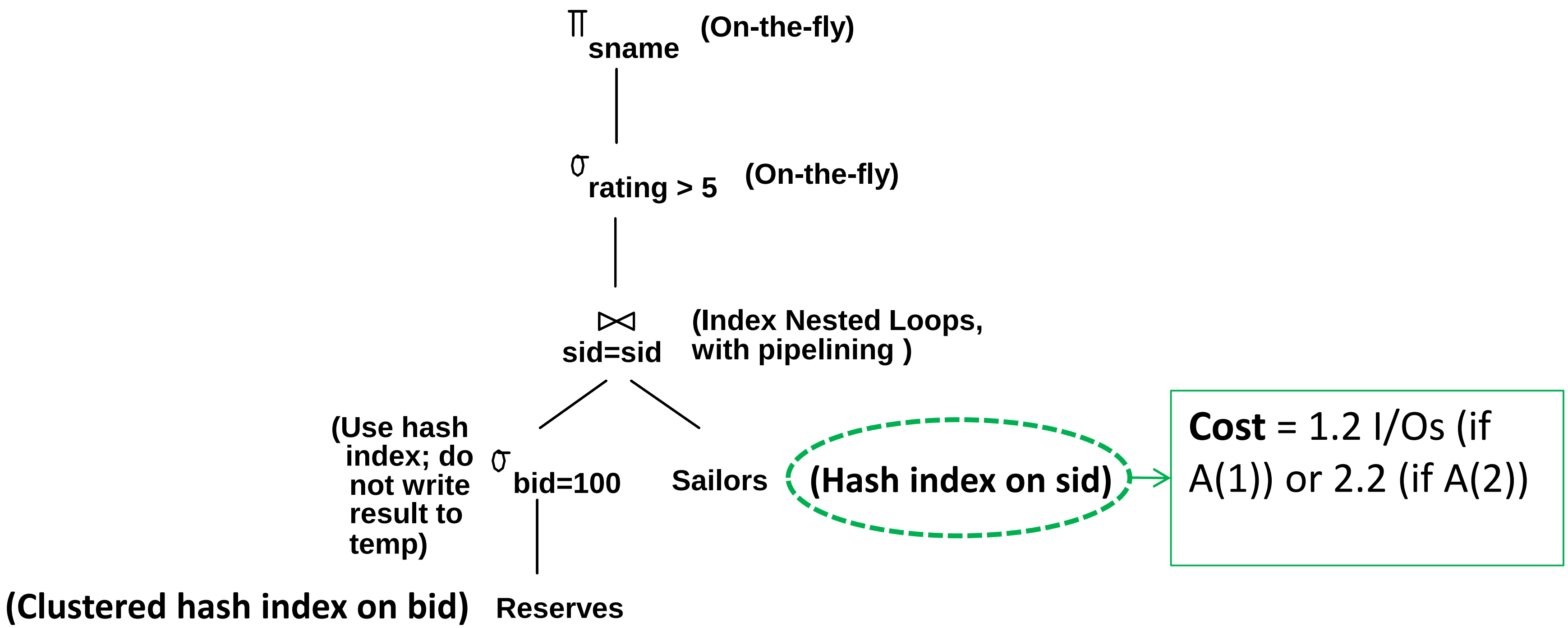


Does the hash index on sid need to be clustered?

NO, since there is at most 1 matching Sailors tuple per a Reserves tuple! Why?

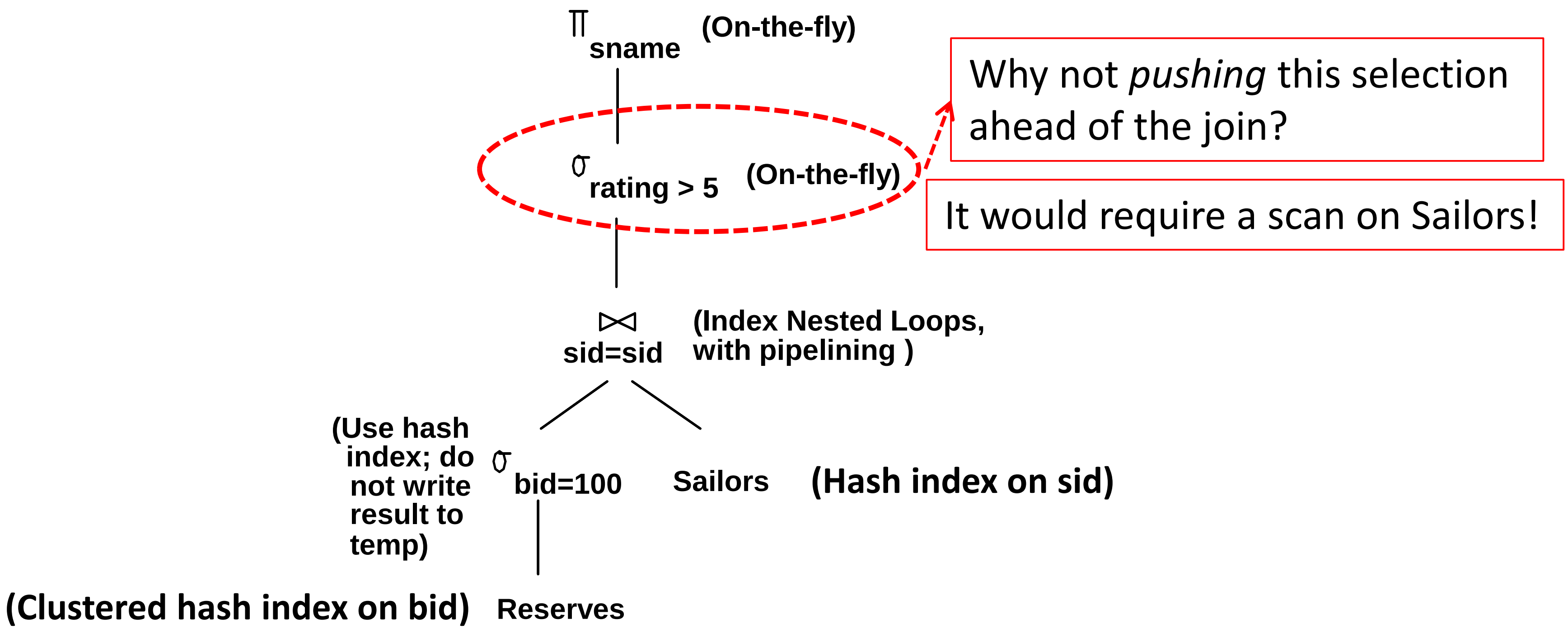
Using Indexes

- What if indexes are available on Reserves and Sailors?



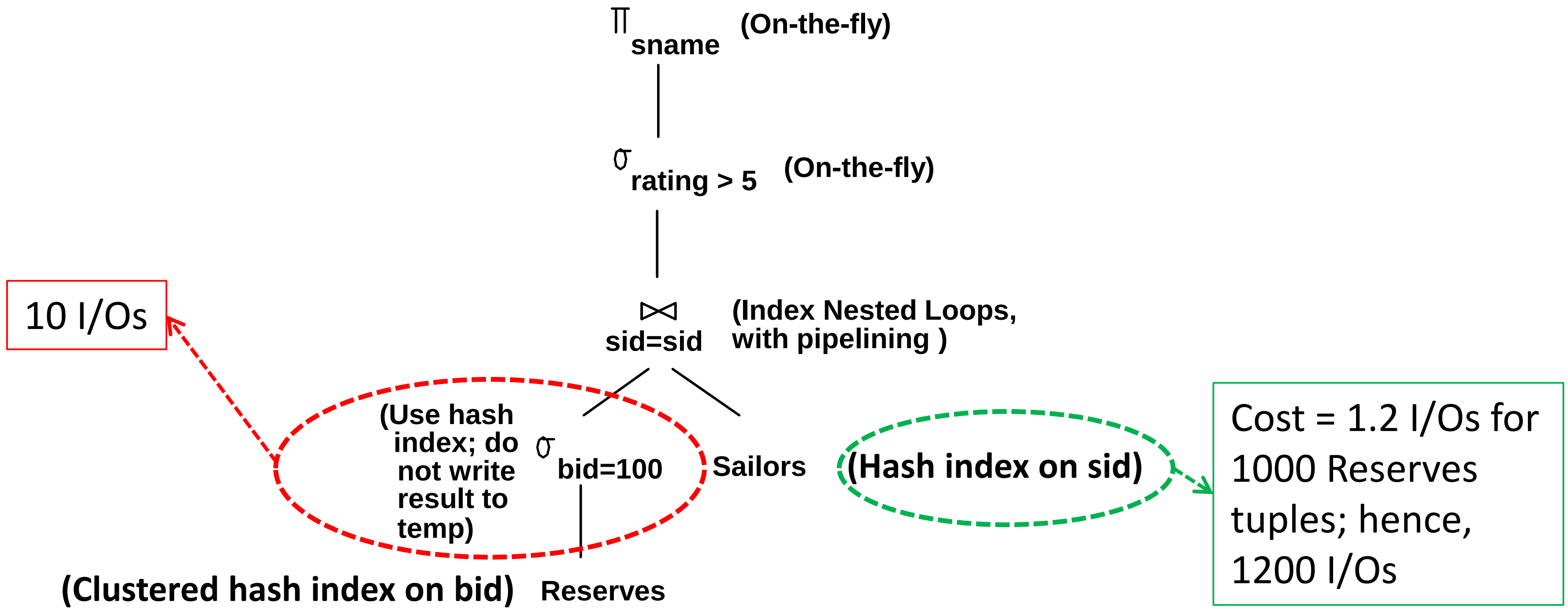
Using Indexes

- What if indexes are available on Reserves and Sailors?



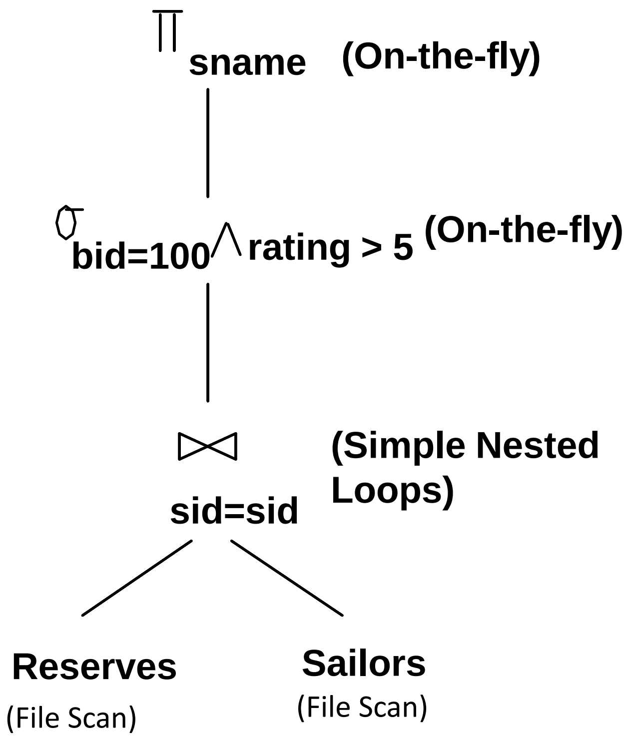
The I/O Cost of the *New* ***Q*** Plan

- What is the I/O cost of the following evaluation plan?

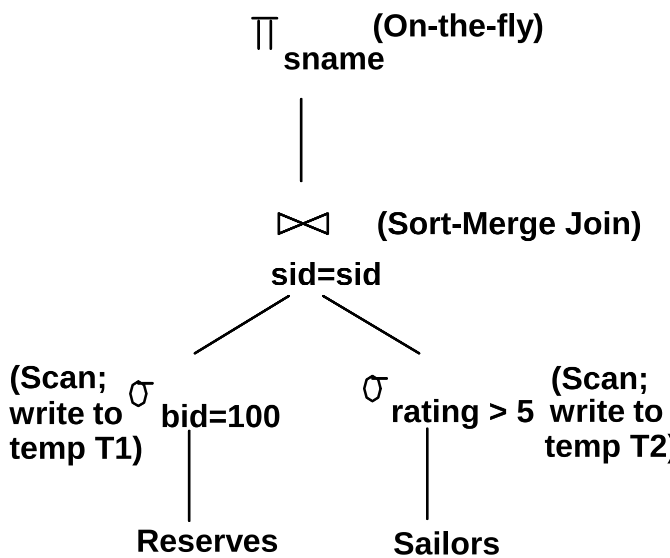


Total Cost = 10 + 1200 = ***1210*** I/Os

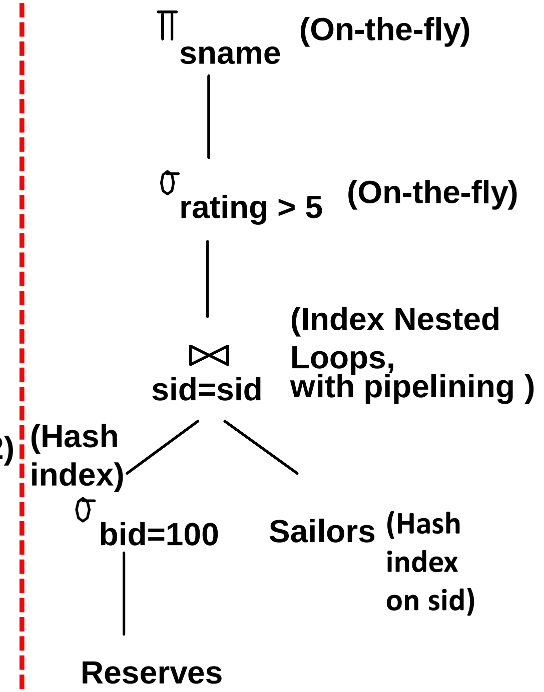
Comparing I/O Costs: Recap



Total Cost = 501, 000 I/Os

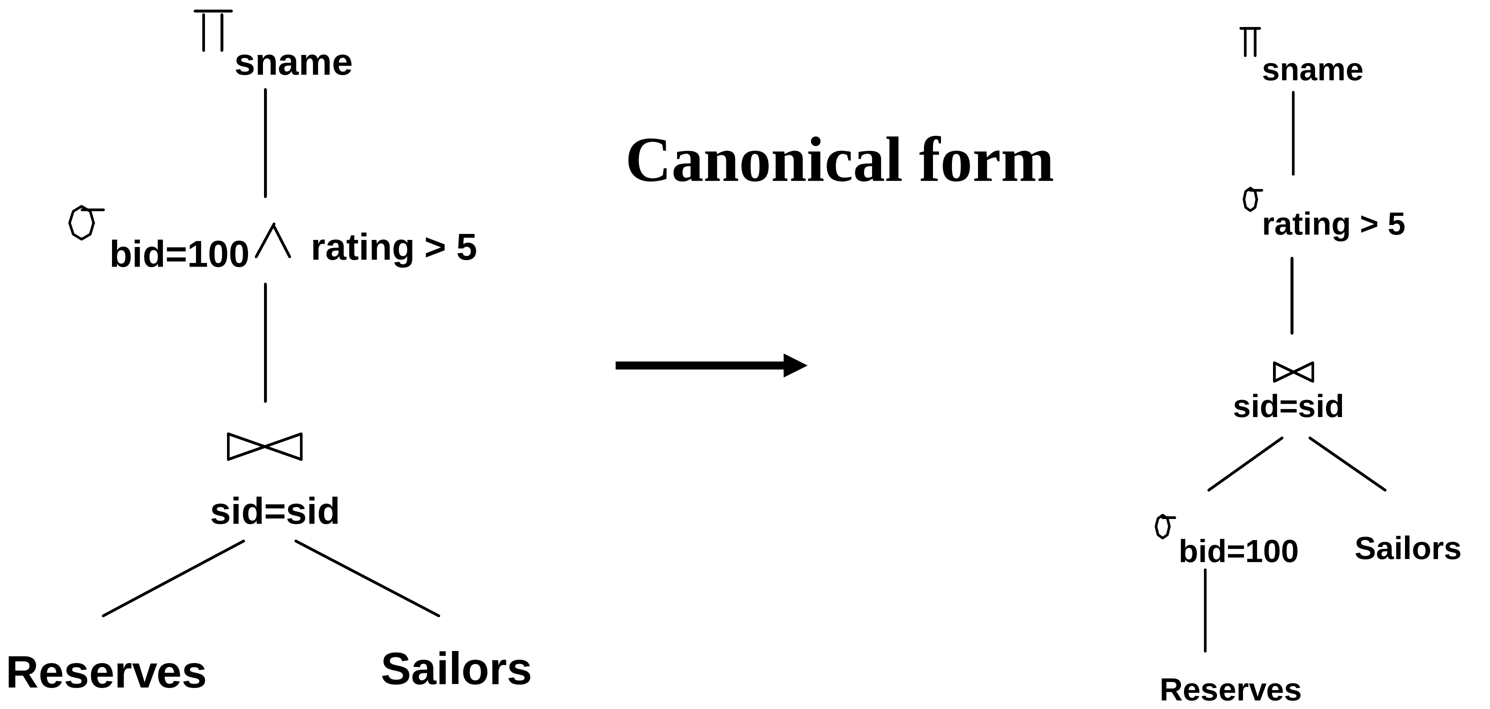


Total Cost = 4060 I/Os



Total Cost = *1210* I/Os

But, How Can we Ensure Correctness?



Still the same result!

How can this be guaranteed?

Outline

- A Brief Primer on Query Optimization
- Evaluating Query Plans
- Relational Algebra Equivalences ✓
- Estimating Plan Costs
- Enumerating Plans

Relational Algebra Equivalences

- A relational query optimizer uses *relational algebra equivalences* to identify many *equivalent* expressions for a given query

- Two relational algebra expressions over the same set of input relations are said to be *equivalent* if they produce the same result on all relations' instances

- Relational algebra equivalences allow us to:
  - Push selections and projections ahead of joins
  - Combine selections and cross-products into joins
  - Choose different join orders

RA Equivalences: Selections

- Two important equivalences involve selections:
  1. Cascading of Selections:

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

Allows us to combine several selections into one selection

OR: Allows us to replace a selection with several smaller selections

  2. Commutation of Selections:

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

Allows us to test selection conditions in either order

RA Equivalences: Projections

- One important equivalence involves projections:
  - Cascading of Projections:

$$\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{an}(R)))$$

This says that successively eliminating columns from a relation is equivalent to simply eliminating all but the columns retained by the final projection!

RA Equivalences: Cross-Products and Joins

- Two important equivalences involve cross-products and joins:
    1. Commutative Operations:

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

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

This allows us to choose which relation to be the inner and which to be the outer!

RA Equivalences: Cross-Products and Joins

- Two important equivalences involve cross-products and joins:

  2. Associative Operations:

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

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

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

This says that regardless of the order in which the relations are considered, the final result is the same!

This *order-independence* is fundamental to how a query optimizer generates alternative query evaluation plans

RA Equivalences: Selections, Projections, Cross Products and Joins

- Selections with Projections:

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

This says we can commute a selection with a projection if the selection involves only attributes retained by the projection!

- Selections with Cross-Products:

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

This says we can combine a selection with a cross-product to form a join (*as per the definition of a join*)!

RA Equivalences: Selections, Projections, Cross Products and Joins

- Selections with Cross-Products and with Joins:

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

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

Caveat: The attributes mentioned in *c* must appear only in R and *NOT* in S

This says we can commute a selection with a cross-product or a join if the selection condition involves only attributes of one of the arguments to the cross-product or join!

RA Equivalences: Selections, Projections, Cross Products and Joins

- Selections with Cross-Products and with Joins (*Cont'd*):

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

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

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

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

This applies to joins as well!

RA Equivalences: Selections, Projections, Cross Products and Joins

- Projections with Cross-Products and with Joins:

$$\pi_a(R \times S) \equiv \pi_{a1}(R) \times \pi_{a2}(S)$$

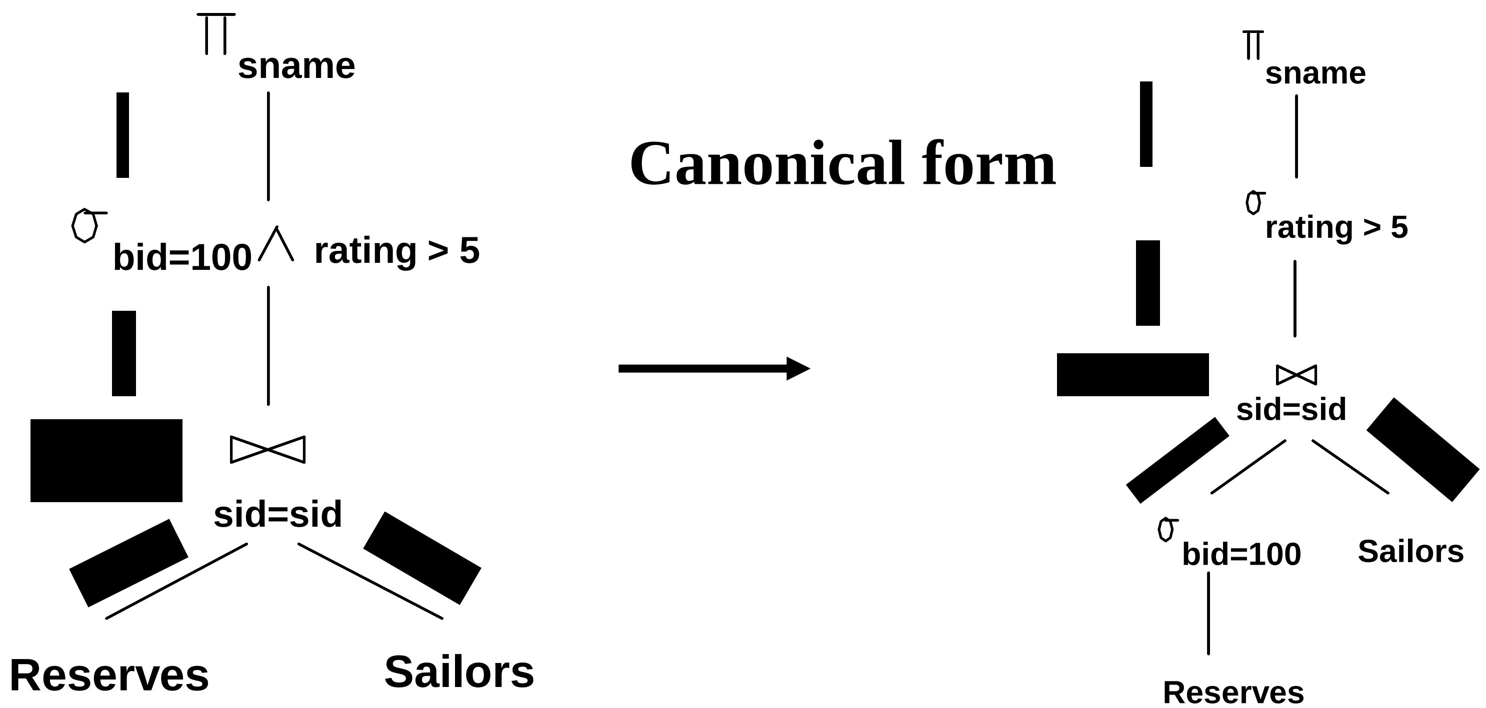
$$\pi_a(R \bowtie_c S) \equiv \pi_{a1}(R) \bowtie_c \pi_{a2}(S)$$

$$\pi_a(R \bowtie_c S) \equiv \pi_a(\pi_{a1}(R) \bowtie_c \pi_{a2}(S))$$

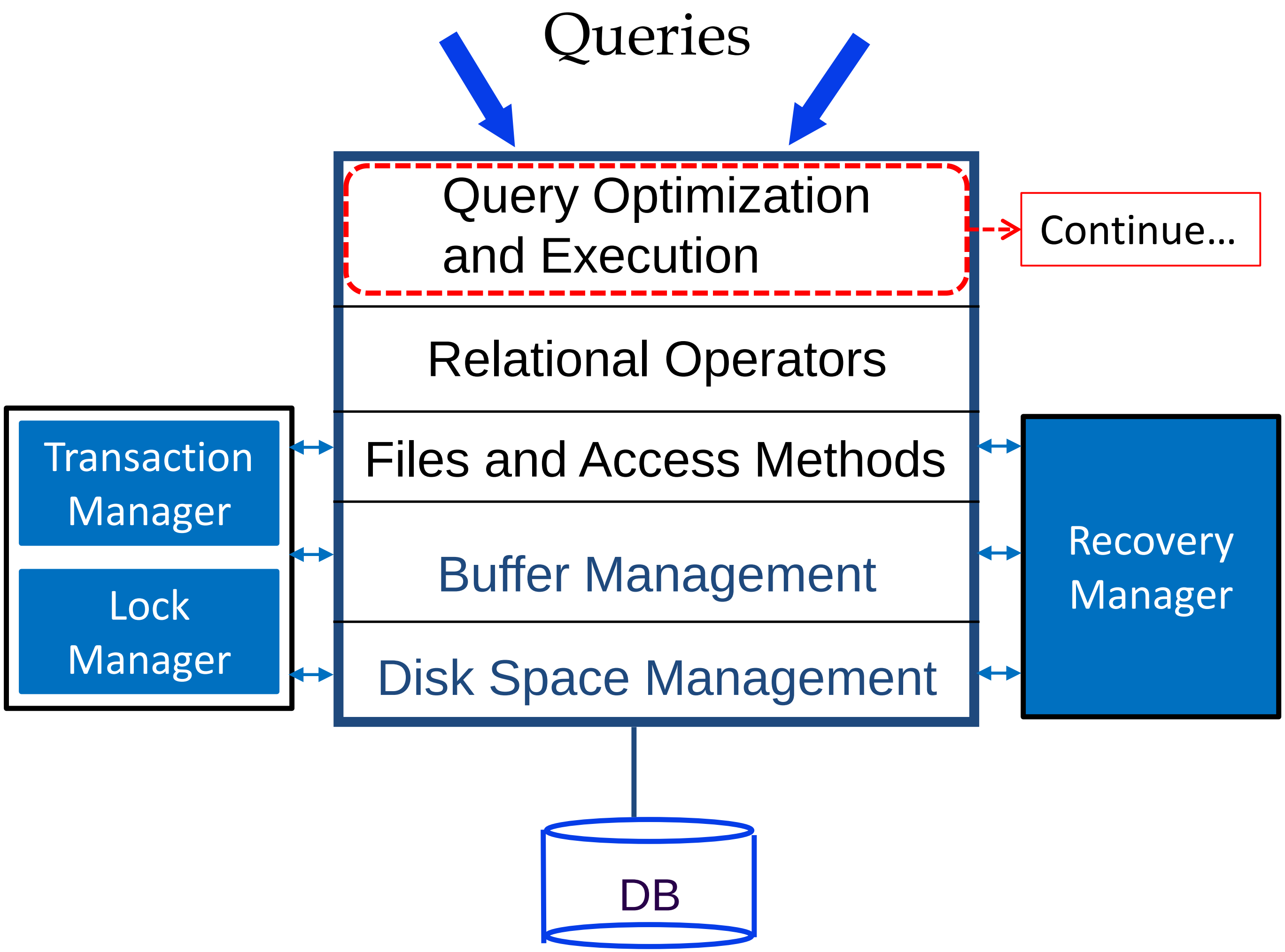
Intuitively, we need to retain only those attributes of R and S that are either mentioned in the join condition ***c*** or included in the set of attributes ***a*** retained by the projection

How to Estimate the Cost of Plans?

- Now that correctness is ensured, how can the DBMS estimate the costs of various plans?



Next Class


Queries
Query Optimization
and Execution
Continue…
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction
Manager
Lock
Manager
Recovery
Manager
DB