Database Applications (15-415)

DBMS Internals- Part X
Lecture 18, March 26, 2014

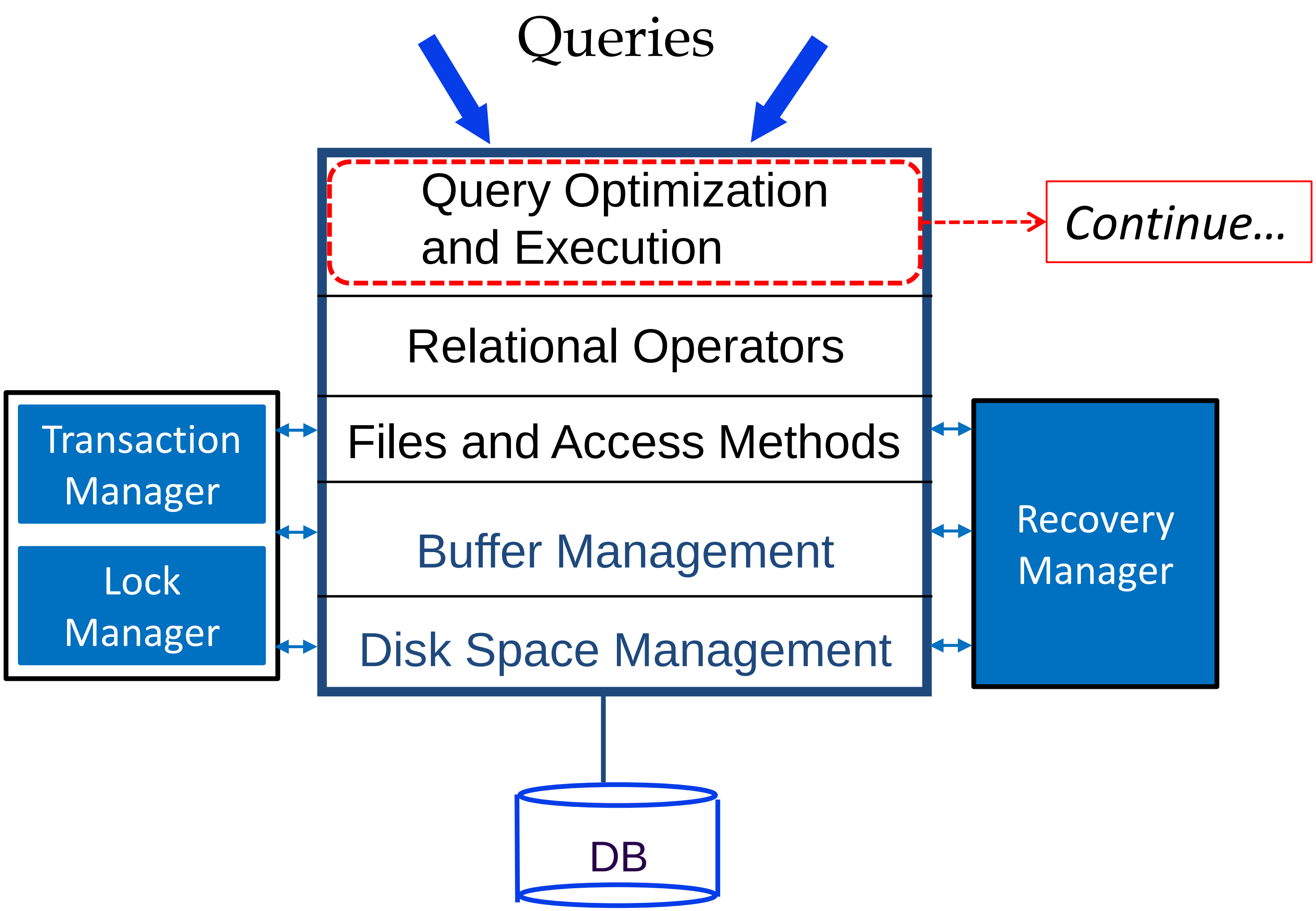
Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Today…

- Last Session:
  - DBMS Internals- Part VIII
    - Query Optimization
- Today's Session:
  - DBMS Internals- Part IX
    - Query Optimization (*Cont'd*)
- Announcements:
  - Project 3 is due on April 5th
  - Quiz 2 is on Thursday, April 3, at 5:00PM in Room 2051 (*all material covered after the midterm*)

DBMS Layers


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

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

Outline

- A Brief Primer on Query Optimization
- Query Evaluation Plans
- Relational Algebra Equivalences
- Estimating Plan Costs ✓
- Enumerating Plans
- Nested Sub-Queries

Last Session

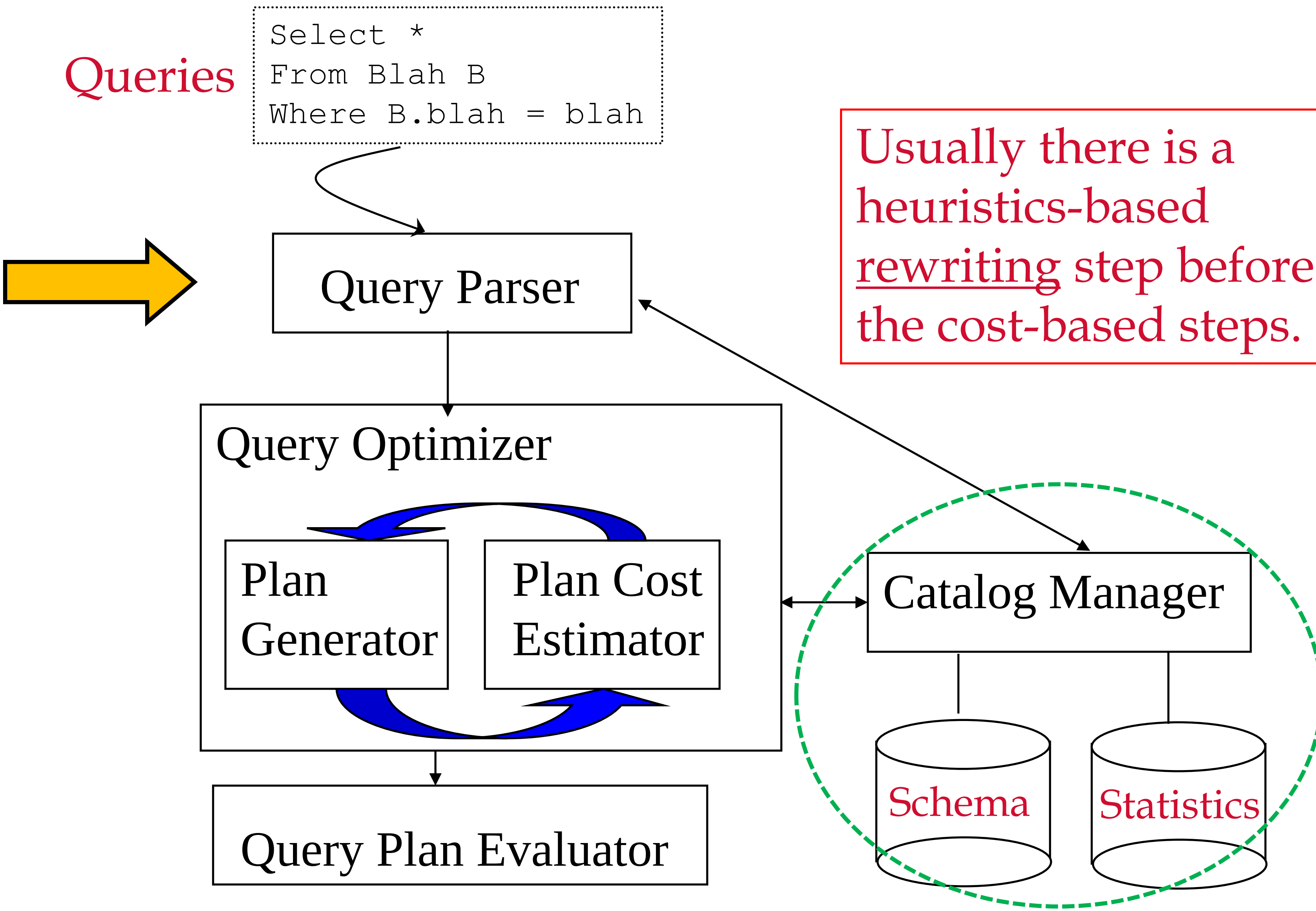
Required Information to Estimate Plan Costs

- For each enumerated plan, we have to estimate its cost

- To estimate the cost of a query plan, 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: Revisit



Usually there is a heuristics-based rewriting step before the cost-based steps.

Catalog Manager: The Schema Component

- 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: The Statistics Component

- 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 *operation costs* and *result sizes*

Estimating the Cost of a Plan

- The cost of a plan can be estimated by:
  1. Estimating *the cost of each operation* in the plan tree
     - Already covered last week (e.g., costs of various join algorithms)
  2. Estimating *the size of the result of each operation* in the plan tree
     - The output ***size*** and ***order*** of a child node affects the cost of its parent node

How can we estimate result sizes?

Estimating Result Sizes

- Consider a query block, ***QB***, of the form:

```
SELECT  attribute list
FROM  R1, R2, …., Rn
WHERE  term 1 AND ... AND term k
```

- What is the *maximum* number of tuples generated by ***QB***?
  - NTuples (R1) × NTuples (R2) × …. × NTuples(Rn)

- Every term in the WHERE clause, however, eliminates some of the possible resultant tuples
  - A *reduction factor* can be associated with each term

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Estimating Result Sizes (*Cont'd*)

- Consider a query block, ***QB***, of the form:

```
SELECT attribute list
FROM R1, R2, …., Rn
WHERE term 1 AND ... AND term k
```

- The *reduction factor* (*RF*) associated with each *term* reflects the impact of the *term* in reducing the result size

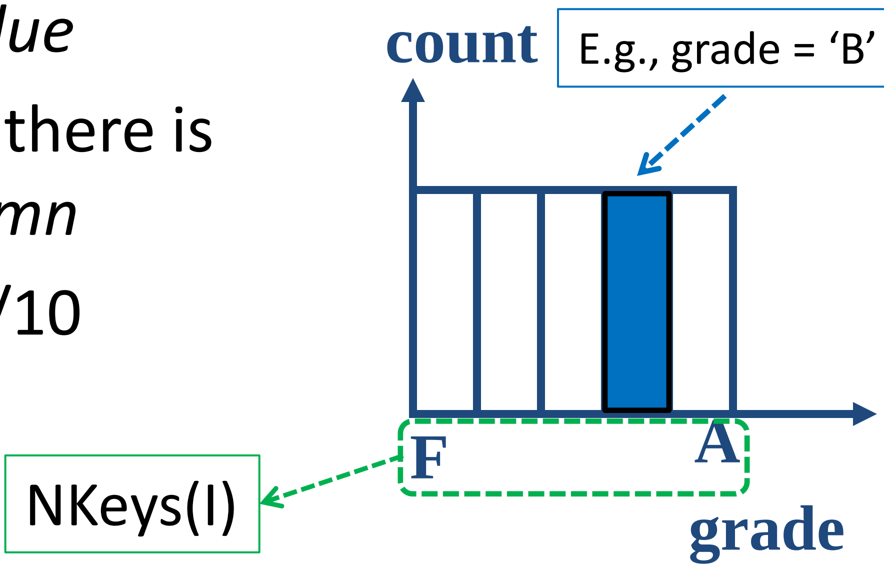
- Final (***estimated***) result cardinality = [NTuples (R1) × ... × NTuples(Rn)] × [RF(term 1) ×... × RF(term k)]
  - ***Implicit assumptions***: *terms are independent and distribution is uniform!*

But, how can we compute reduction factors?

Approximating Reduction Factors

- Reduction factors (RFs) can be *approximated* using the statistics available in the DBMS's catalog

- For different ***<u>forms</u>*** of terms, RF is computed differently
  - Form 1: *Column = Value*
    - RF = 1/NKeys(***I***), if there is an index ***I*** on *Column*
    - Otherwise, RF = 1/10

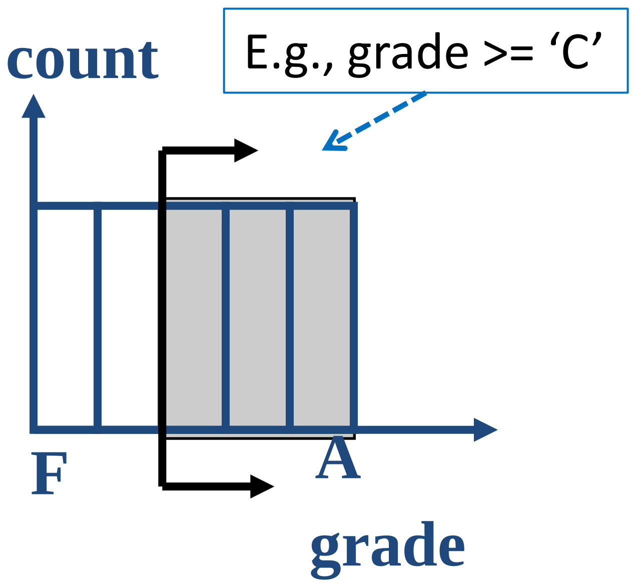


Approximating Reduction Factors (*Cont'd*)

- For different forms of terms, RF is computed differently
  - Form 2: *Column 1 = Column 2*
    - RF = 1/MAX(NKeys(***I1***), NKeys(***I2***)), if there are indices ***I1*** and ***I2*** on *Column 1* and *Column 2*, respectively
    - **Or**: RF = 1/NKeys(***I***), if there is only 1 index on *Column 1* or *Column 2*
    - **Or**: RF = 1/10, if neither *Column 1* nor *Column 2* has an index
  - Form 3: *Column* **IN** (*List of Values*)
    - RF equals to RF of "*Column = Value*" (i.e., Form 1) × # of elements in the *List of Values*

Approximating Reduction Factors (*Cont'd*)

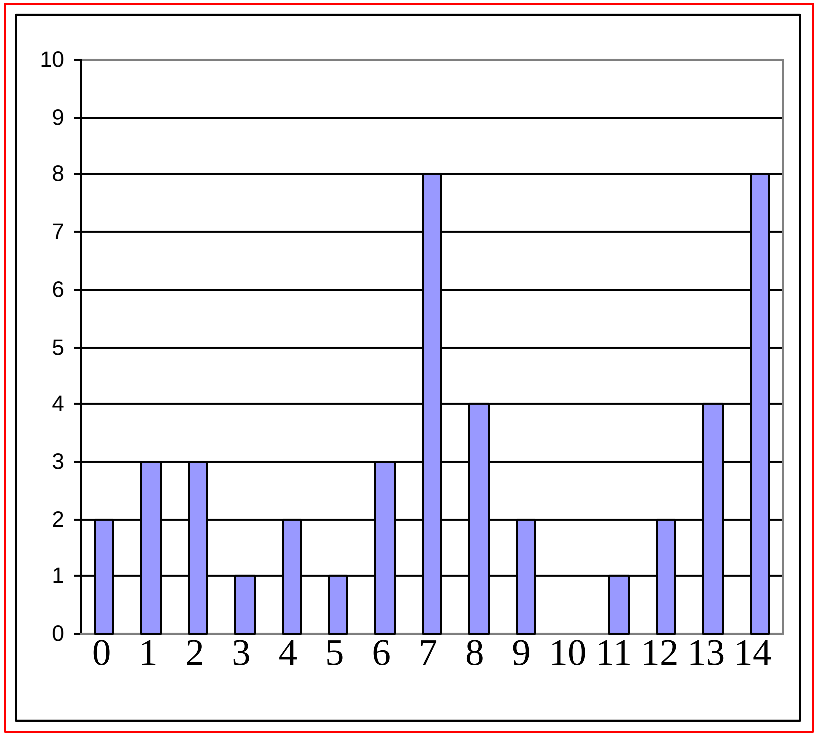
- For different forms of terms, RF is computed differently
  - Form 4: *Column* > *Value*
    - RF = (High(***I***) – *Value*)/(High(***I***) – Low(***I***)), if there is an index ***I*** on *Column*
    - Otherwise, RF equals to any fraction < 1/2



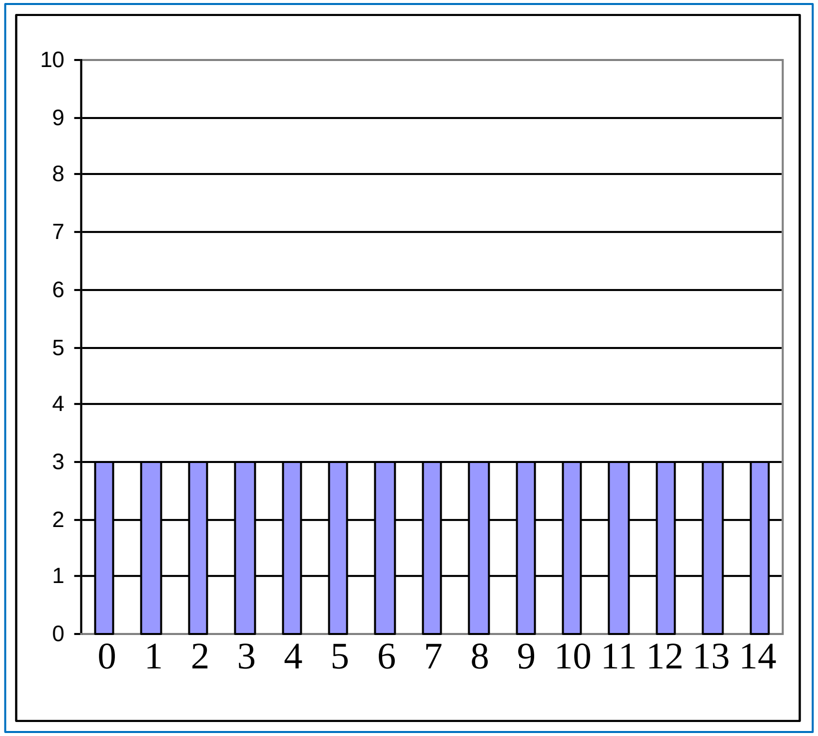
Improved Statistics: Histograms

- Estimates can be improved considerably by maintaining more detailed statistics known as *histograms*

Distribution D



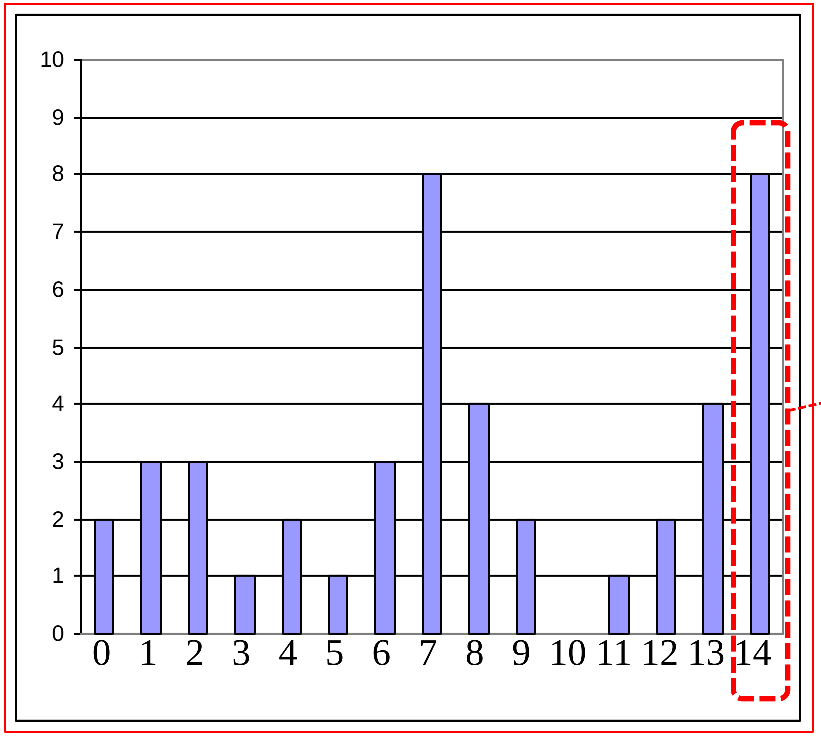
Uniform Distribution Approximating D



Improved Statistics: Histograms

- Estimates can be improved considerably by maintaining more detailed statistics known as *histograms*

Distribution D



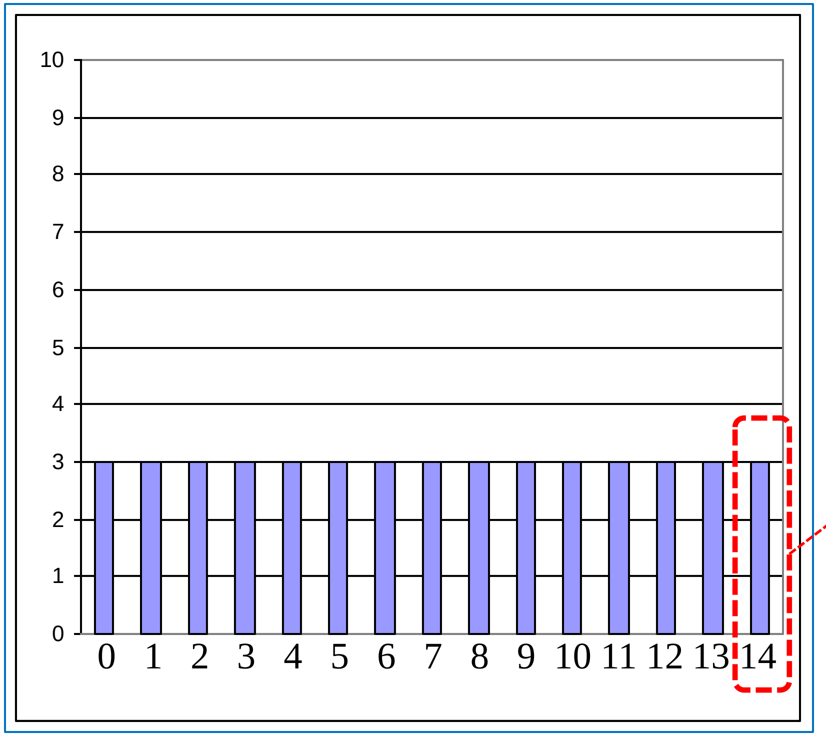
What is the result size of *term* value > 13?

8 tuples

Improved Statistics: Histograms

- Estimates can be improved considerably by maintaining more detailed statistics known as *histograms*

Uniform Distribution Approximating D



What is the (*estimated*) result size of *term* value > 13?

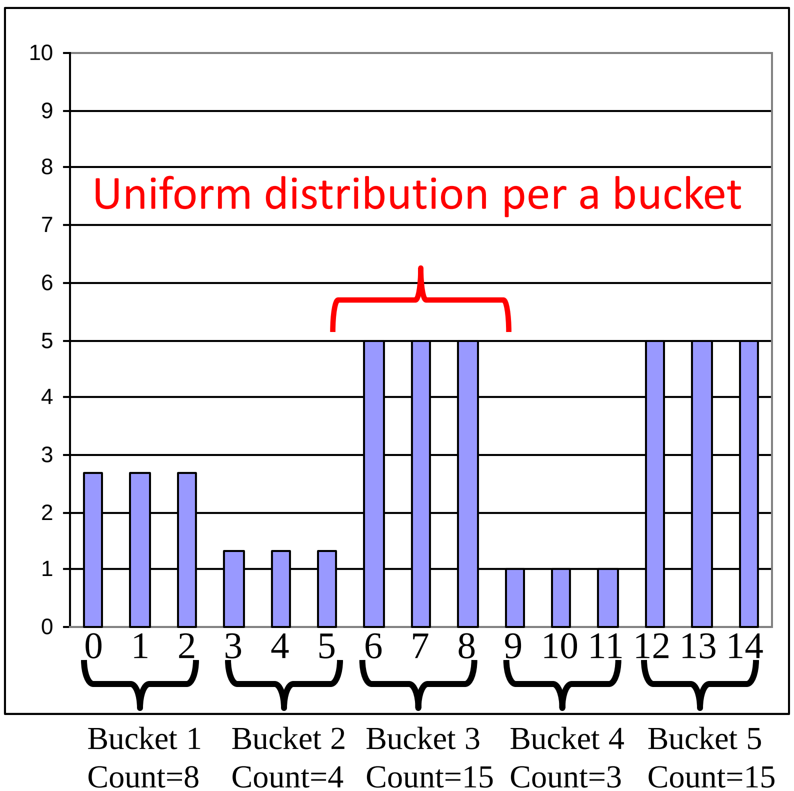
(1/15 × 44) = ~3 tuples

Clearly, this is inaccurate!

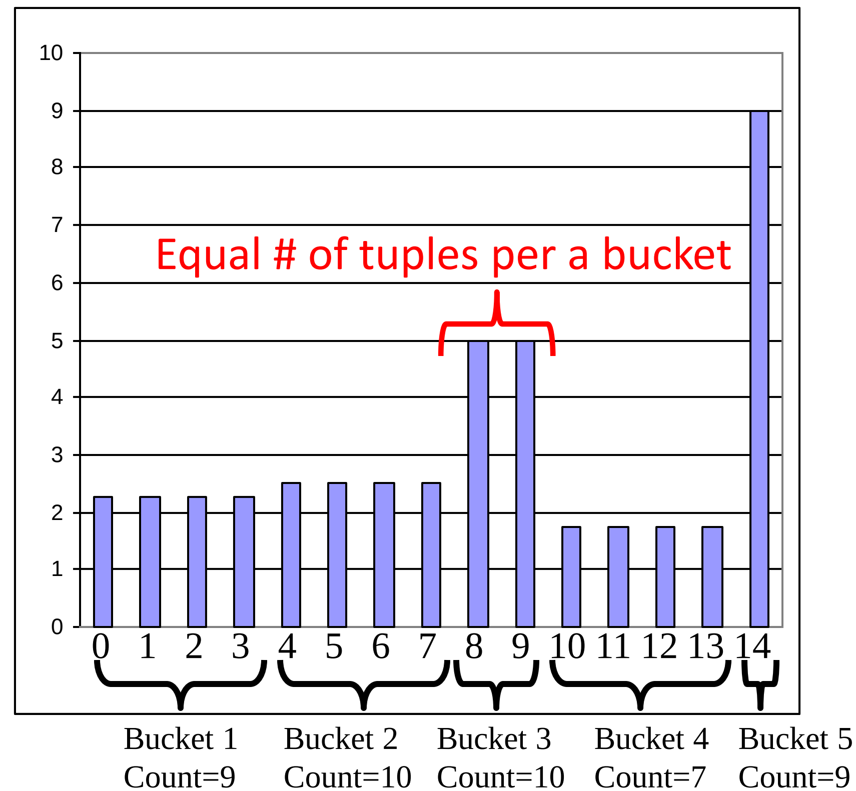
Improved Statistics: Histograms

- We can do better if we divide the range of values into *sub-ranges* called *buckets*

Equiwidth **histogram**



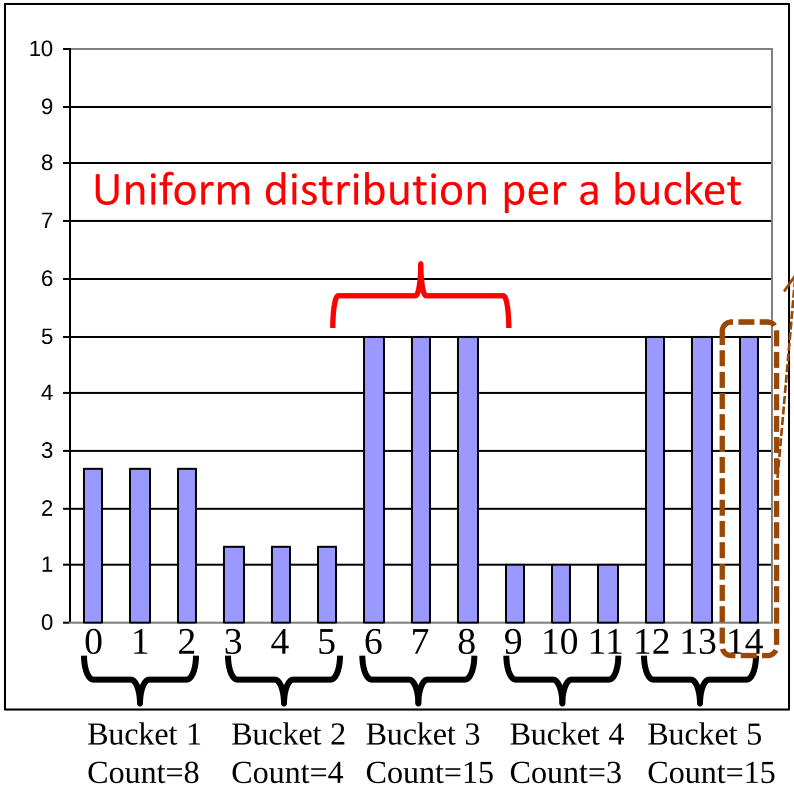
Equidepth **histogram**



Improved Statistics: Histograms

- We can do better if we divide the range of values into *sub-ranges* called *buckets*

Equiwidth histogram



What is the (*estimated*) result size of *term* value > 13?

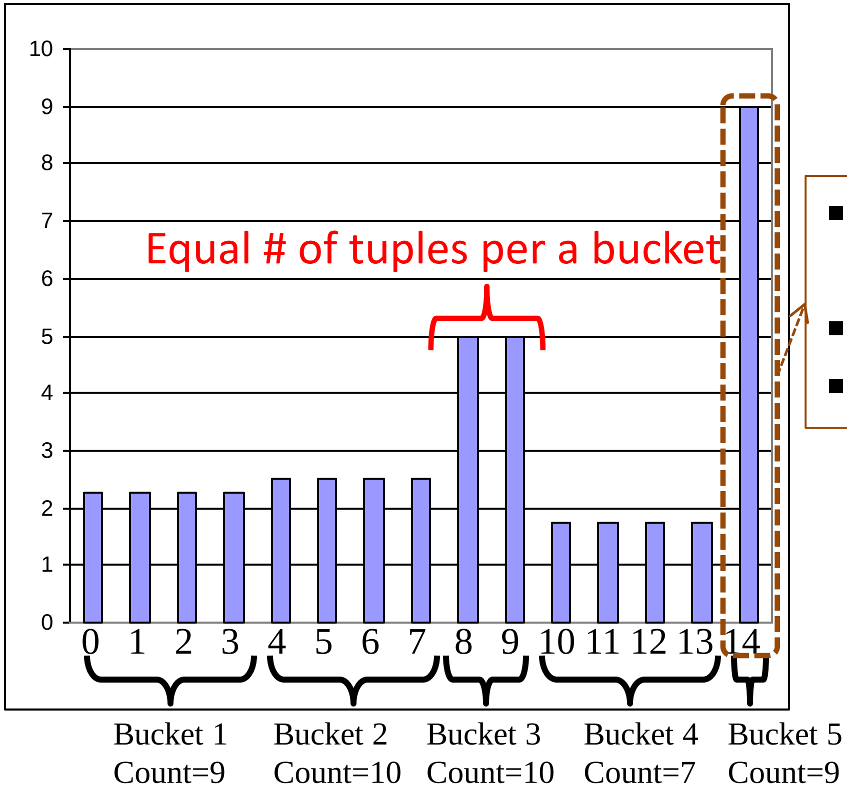
- The selected range = 1/3 of the range for bucket 5
- Bucket 5 represents a total of 15 tuples
- Estimated size = 1/3 × 15 = 5 tuples

Better than regular histograms!

Improved Statistics: Histograms

- We can do better if we divide the range of values into *sub-ranges* called *buckets*

Equidepth histogram



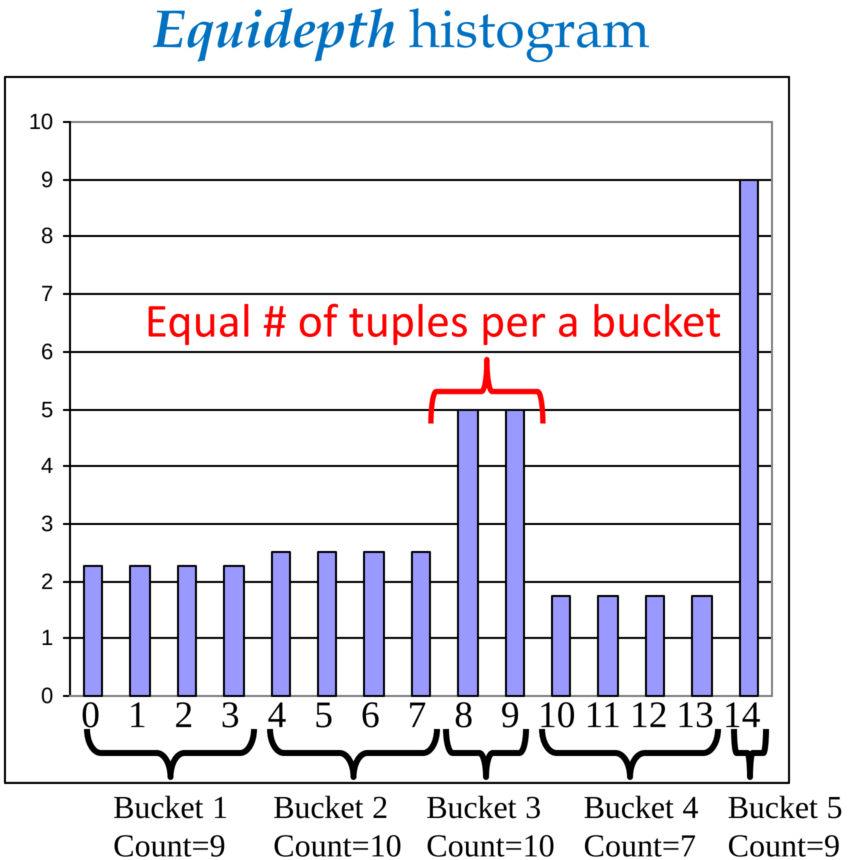
What is the (*estimated*) result size of *term* value > 13?

- The selected range = 100% of the range for bucket 5
- Bucket 5 represents a total of 9 tuples
- Estimated size = 1 × 9 = 9 tuples

Better than *equiwidth* histograms! ***Why?***

Improved Statistics: Histograms

- We can do better if we divide the range of values into *sub-ranges* called *buckets*



Because, buckets with very frequently occurring values contain fewer slots; hence, the uniform distribution assumption is applied to a smaller range of values!

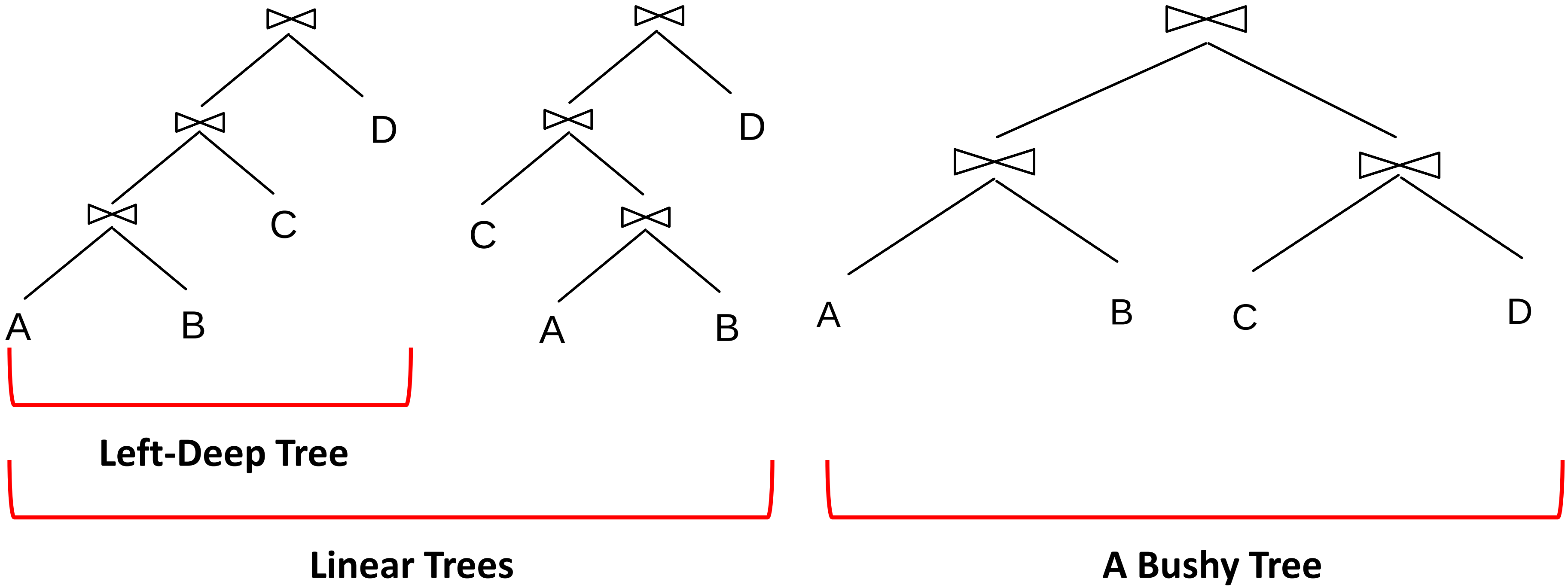
What about buckets with mostly infrequent values?
They are approximated less accurately!

Outline

- A Brief Primer on Query Optimization
- Query Evaluation Plans
- Relational Algebra Equivalences
- Estimating Plan Costs
- Enumerating Plans ✓
- Nested Sub-Queries

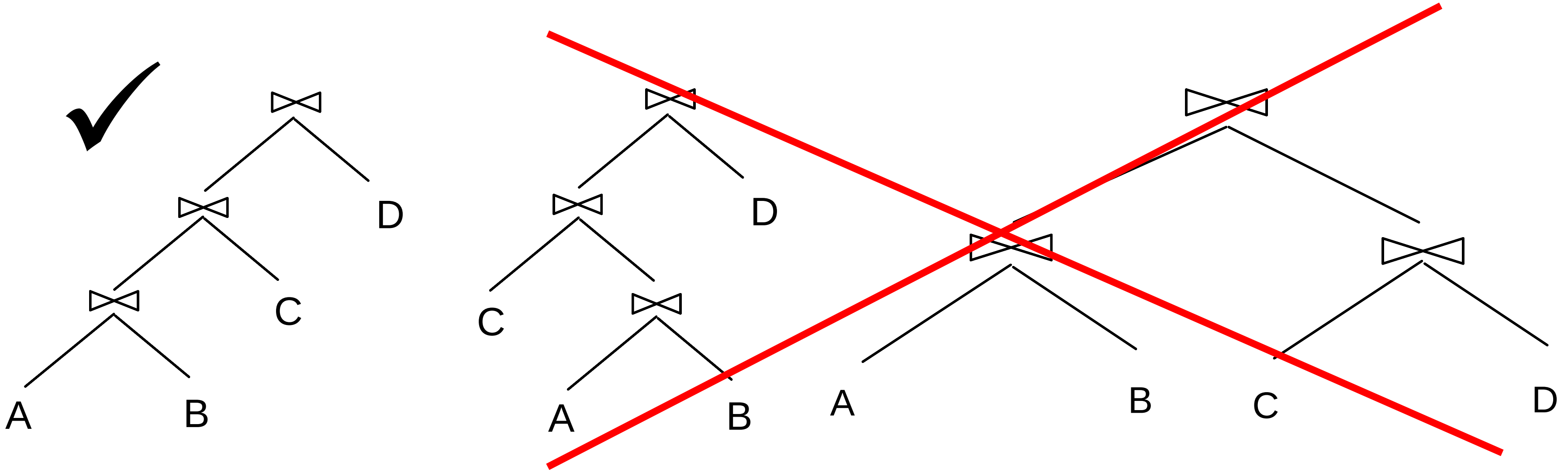
Enumerating Execution Plans

- Consider a query $\boldsymbol{Q} = A \bowtie B \bowtie C \bowtie D$
- Here are 3 plans that are *equivalent*:



Enumerating Execution Plans

- Consider a query ***Q*** = $A \bowtie B \bowtie C \bowtie D$
- Here are 3 plans that are *equivalent*:



Why?

Enumerating Execution Plans (*Cont'd*)

- There are two main reasons for concentrating only on left-deep plans:
  - As the number of joins increases, the number of plans increases rapidly; hence, it becomes necessary to prune the space of alternative plans
  - Left-deep trees allows us to generate all ***fully pipelined*** plans

- Clearly, by adding details to left-deep trees (e.g., the join algorithm per each join), several query plans can be obtained

- The query optimizer enumerates *all possible left-deep* plans using typically a *dynamic programming approach* (later), estimates the cost of each plan, and selects the one with the lowest cost!

Enumerating Execution Plans (*Cont'd*)

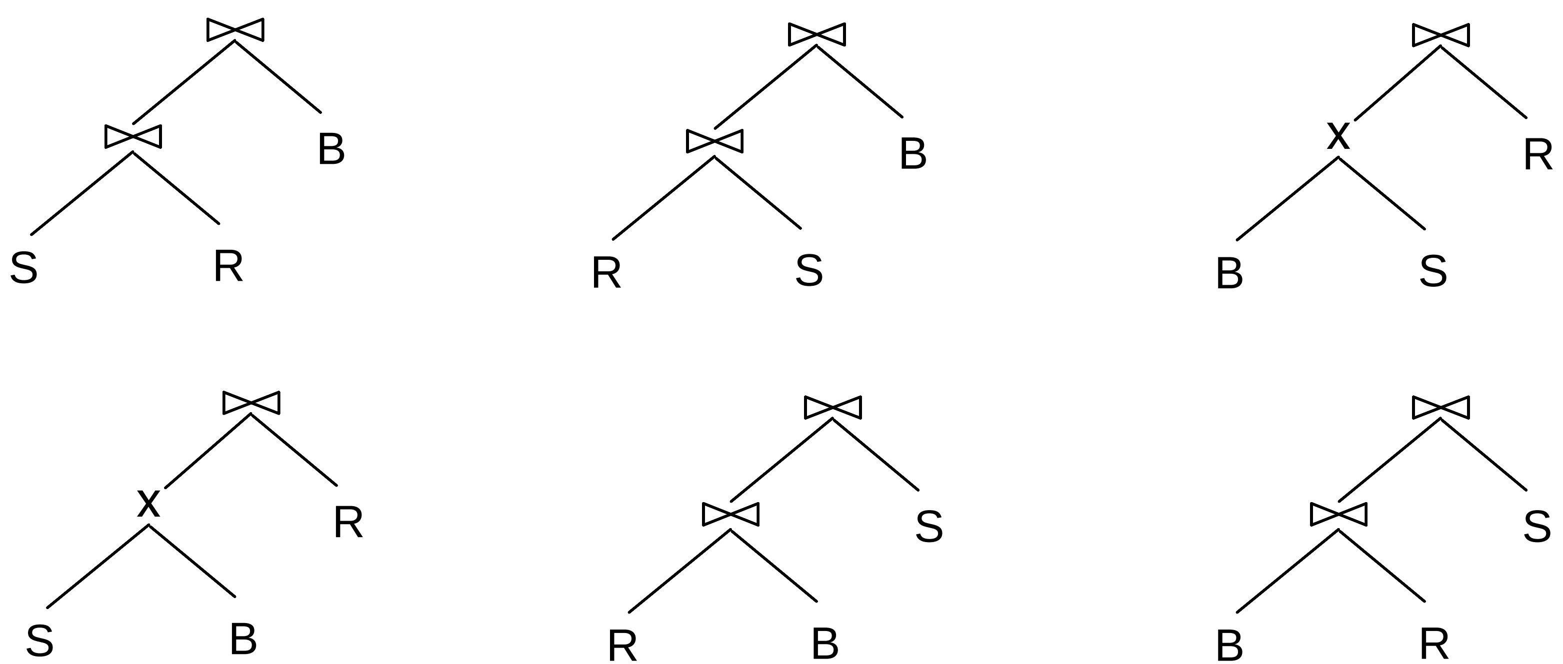
- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings
  2. The different possible ways for evaluating each operator
  3. The different access paths for each relation

- Assume the following query ***Q***:

```
SELECT  S.sname, B.bname, R.day
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid
```

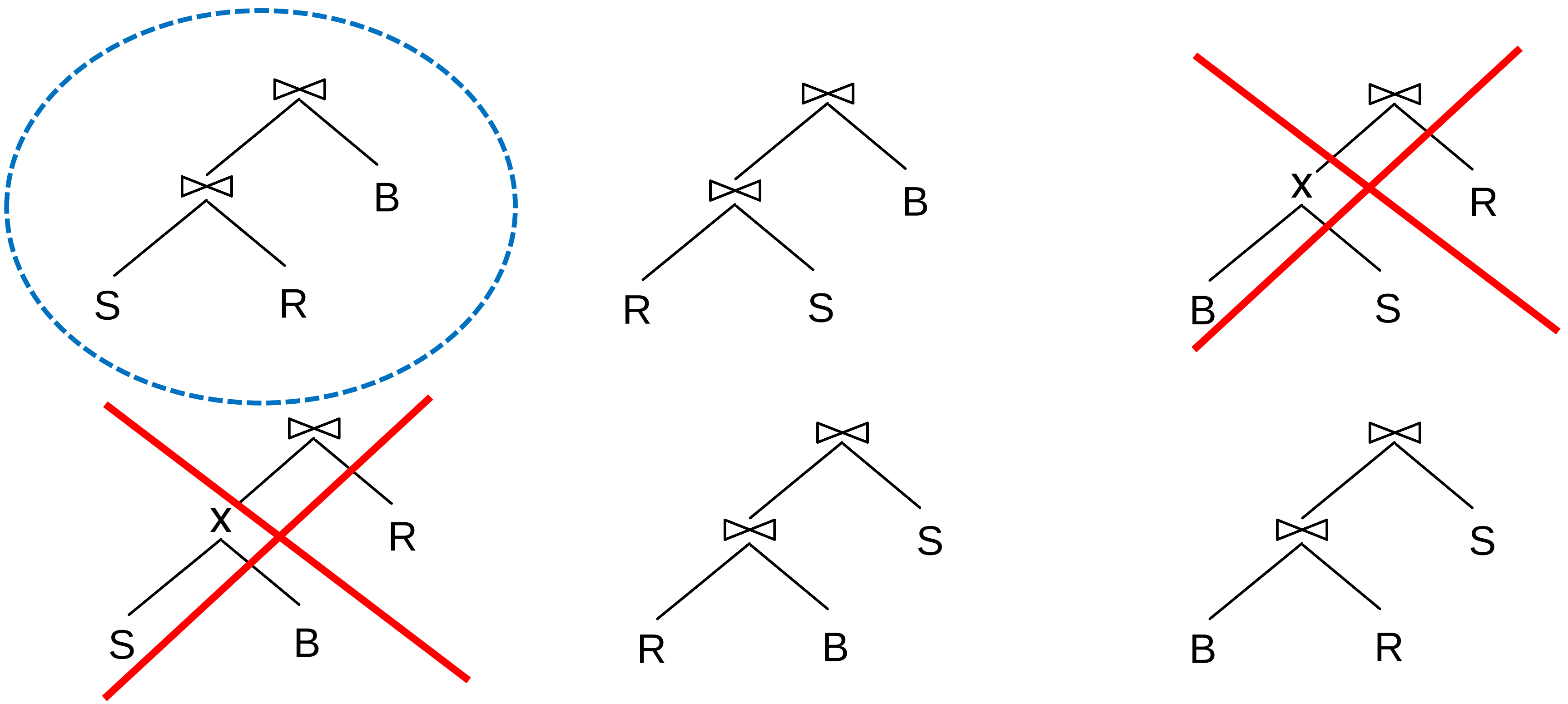
Enumerating Execution Plans (*Cont'd*)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings



Enumerating Execution Plans (*Cont'd*)

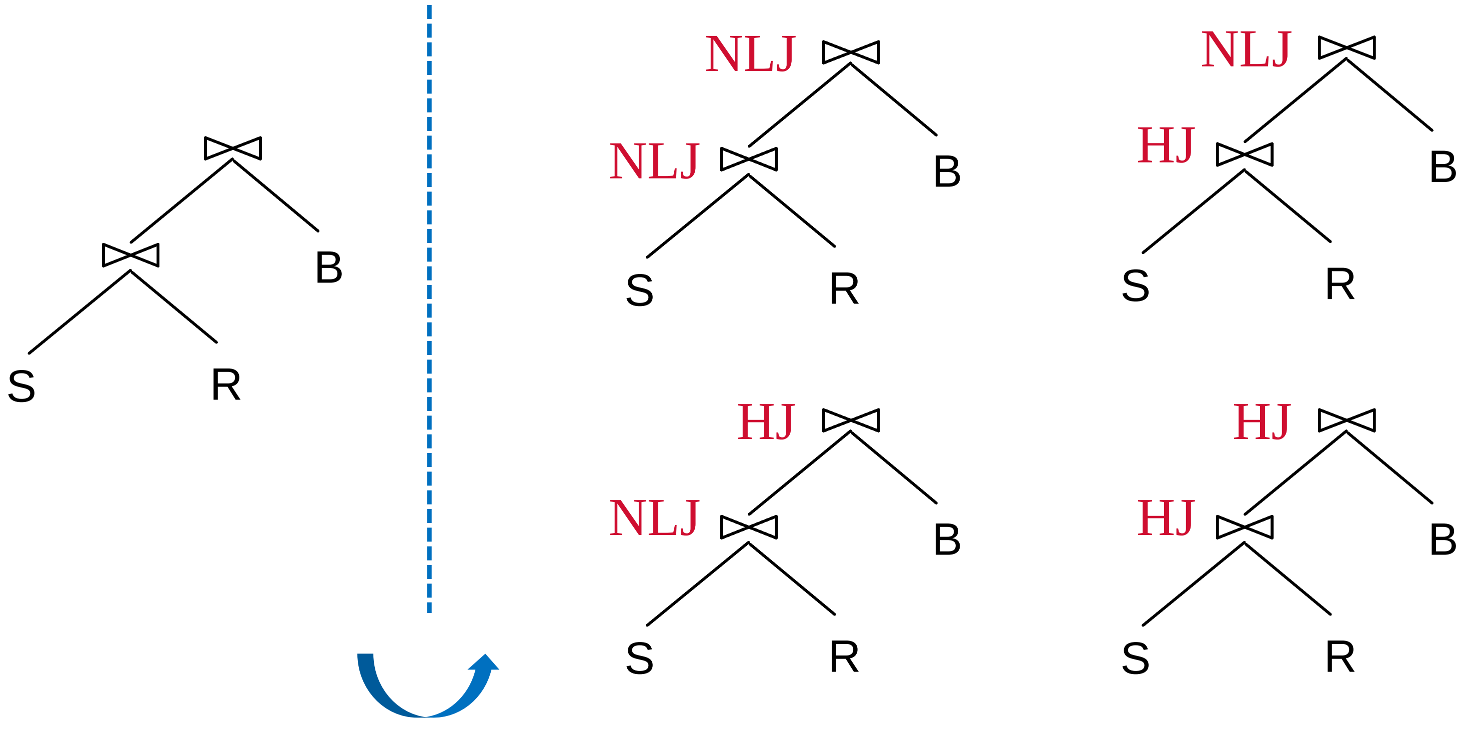
- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings



Prune plans with cross-products immediately!

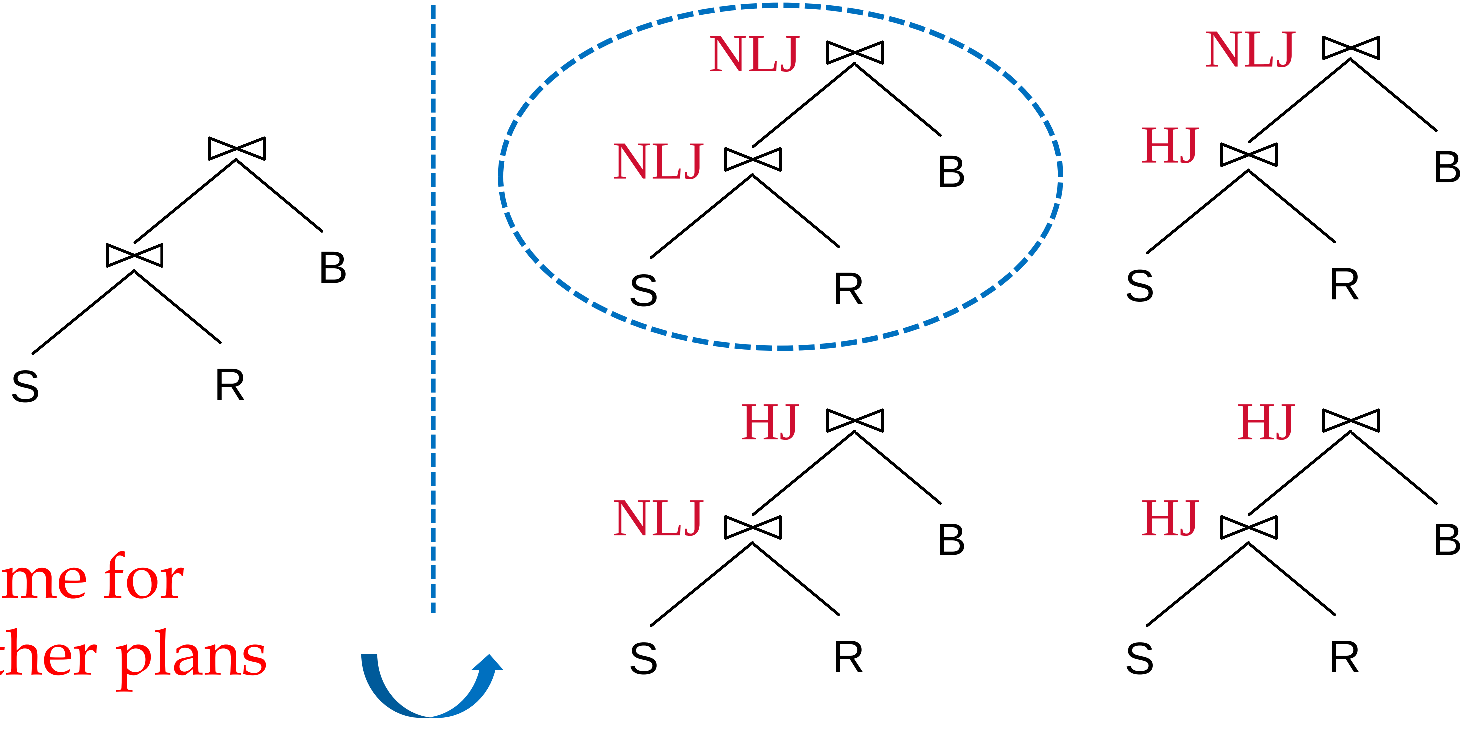
Enumerating Execution Plans (*Cont'd*)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings
  2. The different possible ways for evaluating each operator



Enumerating Execution Plans (*Cont'd*)

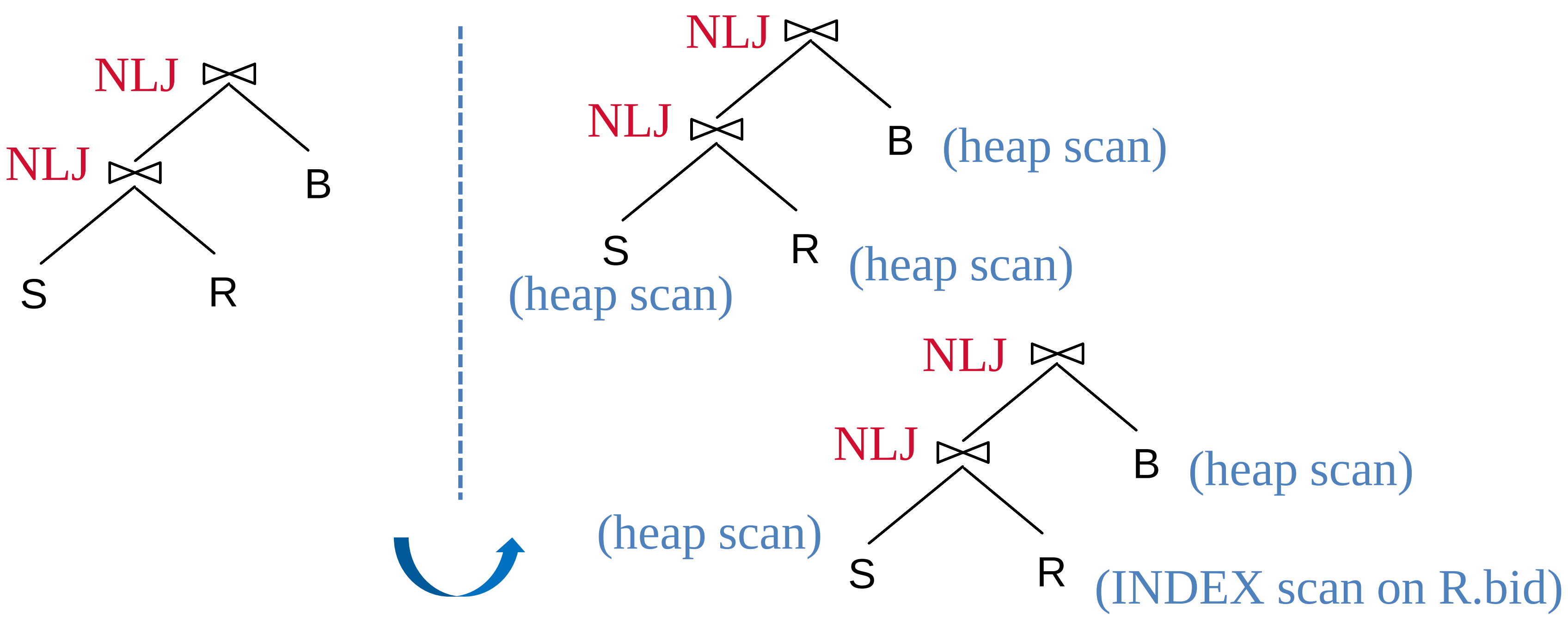
- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings
  2. The different possible ways for evaluating each operator



+ do same for
the 3 other plans

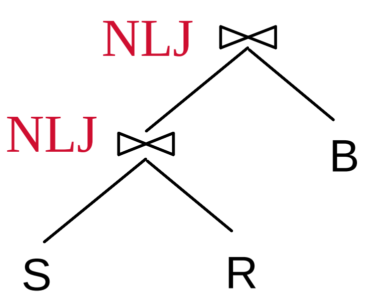
Enumerating Execution Plans (*Cont'd*)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings
  2. The different possible ways for evaluating each operator
  3. The different access paths for each relation

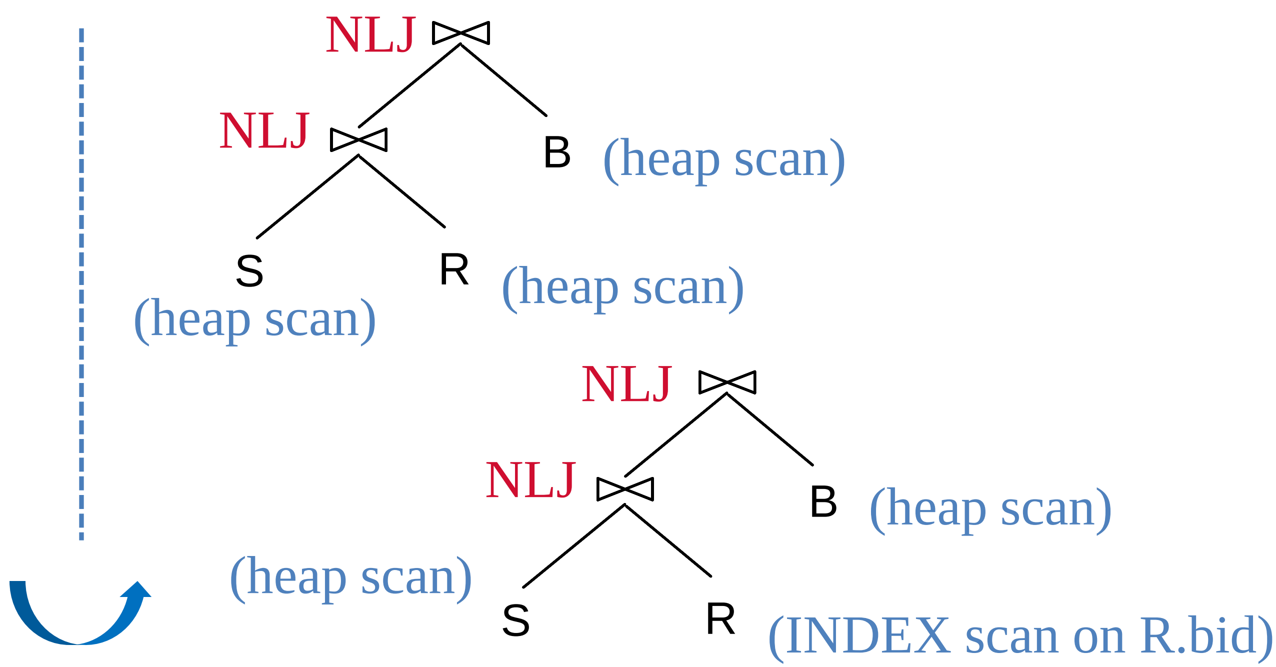


Enumerating Execution Plans (*Cont'd*)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings
  2. The different possible ways for evaluating each operator
  3. The different access paths for each relation



+ do same for the 3 other plans



Enumerating Execution Plans (*Cont'd*)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings
  2. The different possible ways for evaluating each operator
  3. The different access paths for each relation

Subsequently, estimate the cost of each plan using *statistics* collected and stored at the system catalog!

Let us now study a *dynamic programming algorithm* to effectively enumerate and estimate cost plans

Towards a Dynamic Programming Algorithm

- There are two main cases to consider:
  - CASE I: Single-Relation Queries
  - CASE II: Multiple-Relation Queries

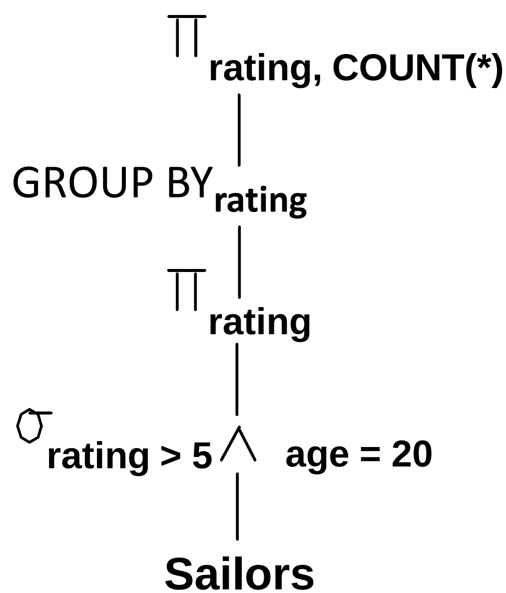
- CASE I: Single-Relation Queries
  - Only *selection*, *projection*, *grouping* and *aggregate* operations are involved (i.e., no *joins*)
  - Every available access path is considered and the one with the least estimated cost is selected
  - The different operations are carried out together
    - E.g., if an index is used for a selection, projection can be done for each retrieved tuple, and the resulting tuples can be *pipelined* into an aggregate operation (if any)

CASE I: Single-Relation Queries- An Example

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

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

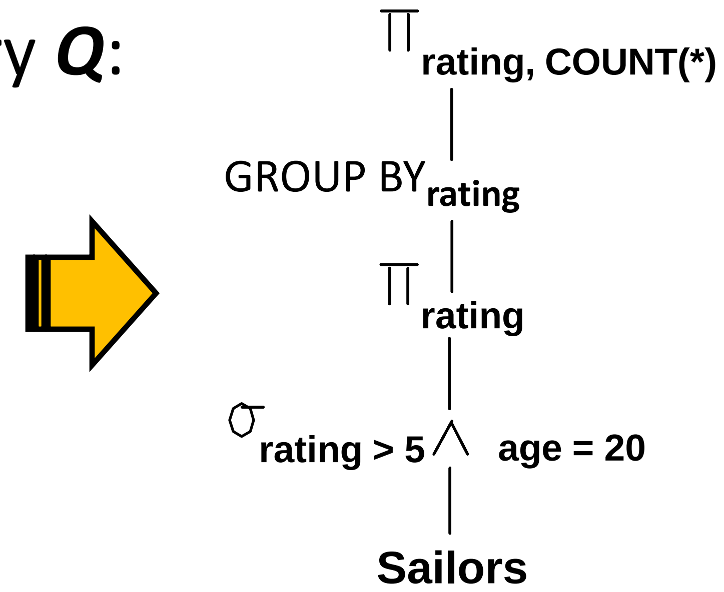
- ***Q*** can be expressed in a relational algebra tree as follows:



CASE I: Single-Relation Queries- An Example

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

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



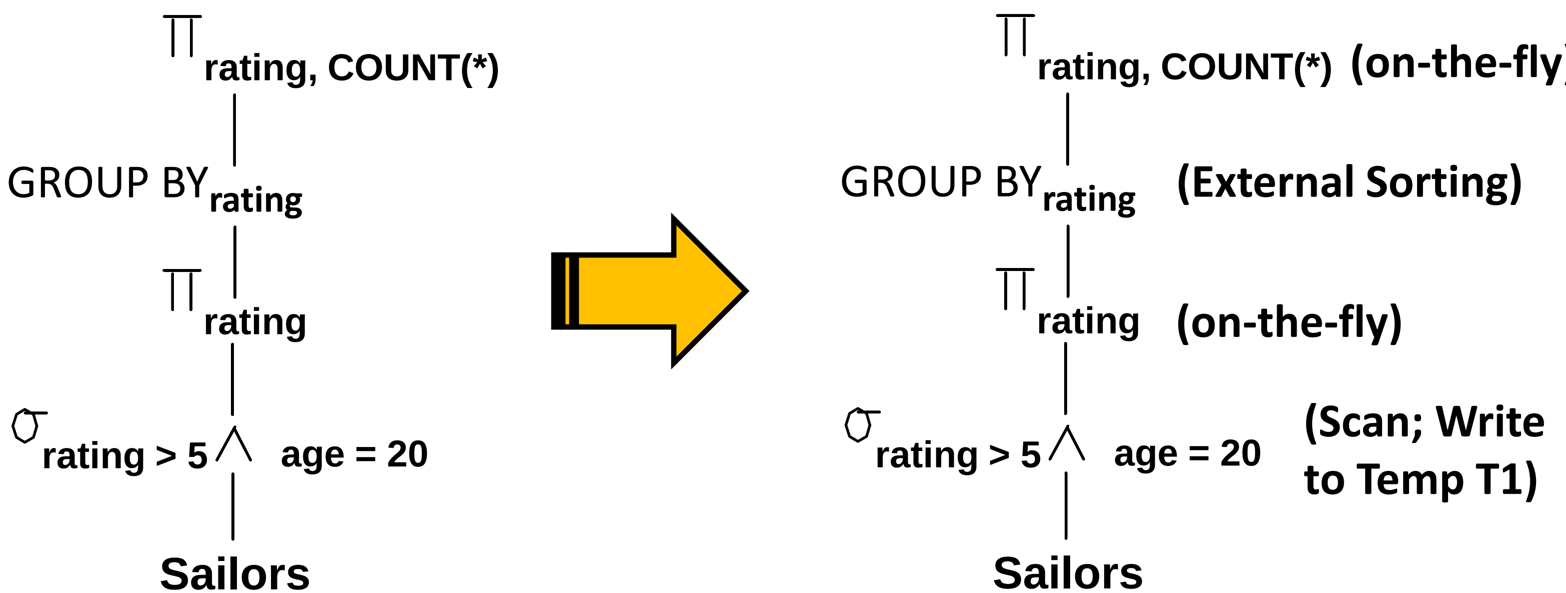
- How can ***Q*** be evaluated?
  - Apply CASE I:
    - Every available access path *for Sailors* is considered and the one with the least estimated cost is selected
    - The selection and projection operations are carried out together

CASE I: Single-Relation Queries- An Example

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

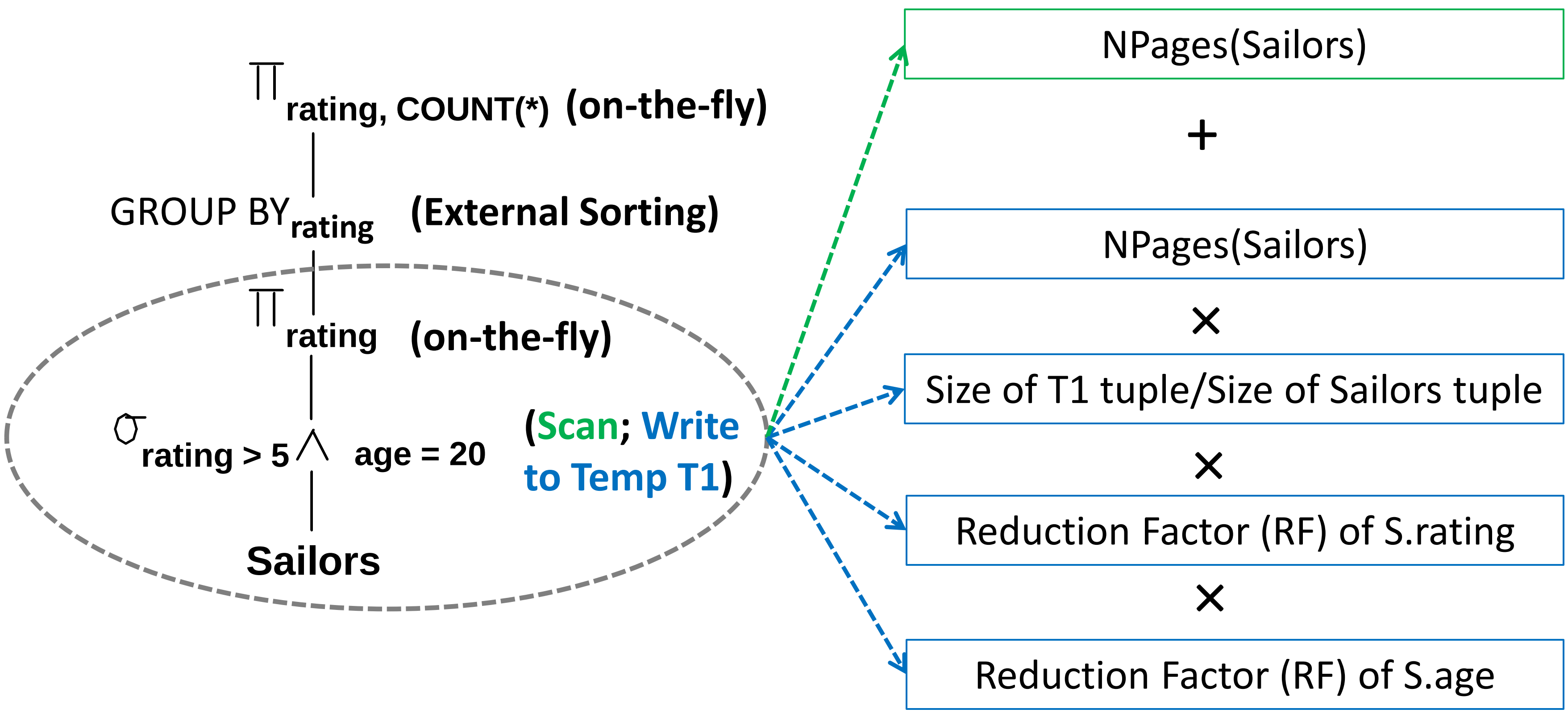
```
SELECT  S.rating, COUNT (*)
FROM  Sailors S
WHERE  S.rating > 5 AND S.age = 20
GROUP BY S.rating
```

- What would be the cost of we assume a <u>file scan</u> for sailors?



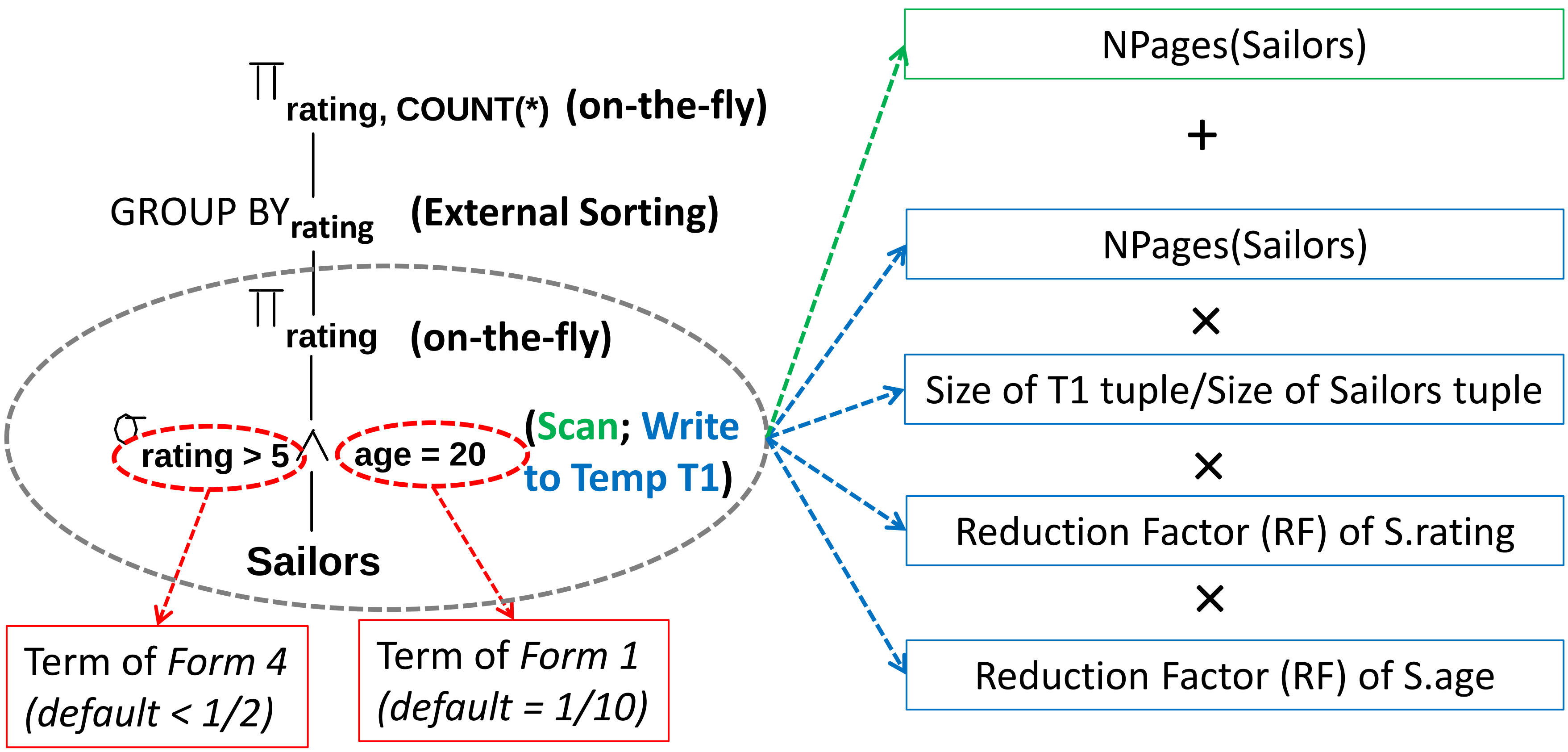
CASE I: Single-Relation Queries- An Example

- What would be the cost of we assume a <u>file scan</u> for sailors?



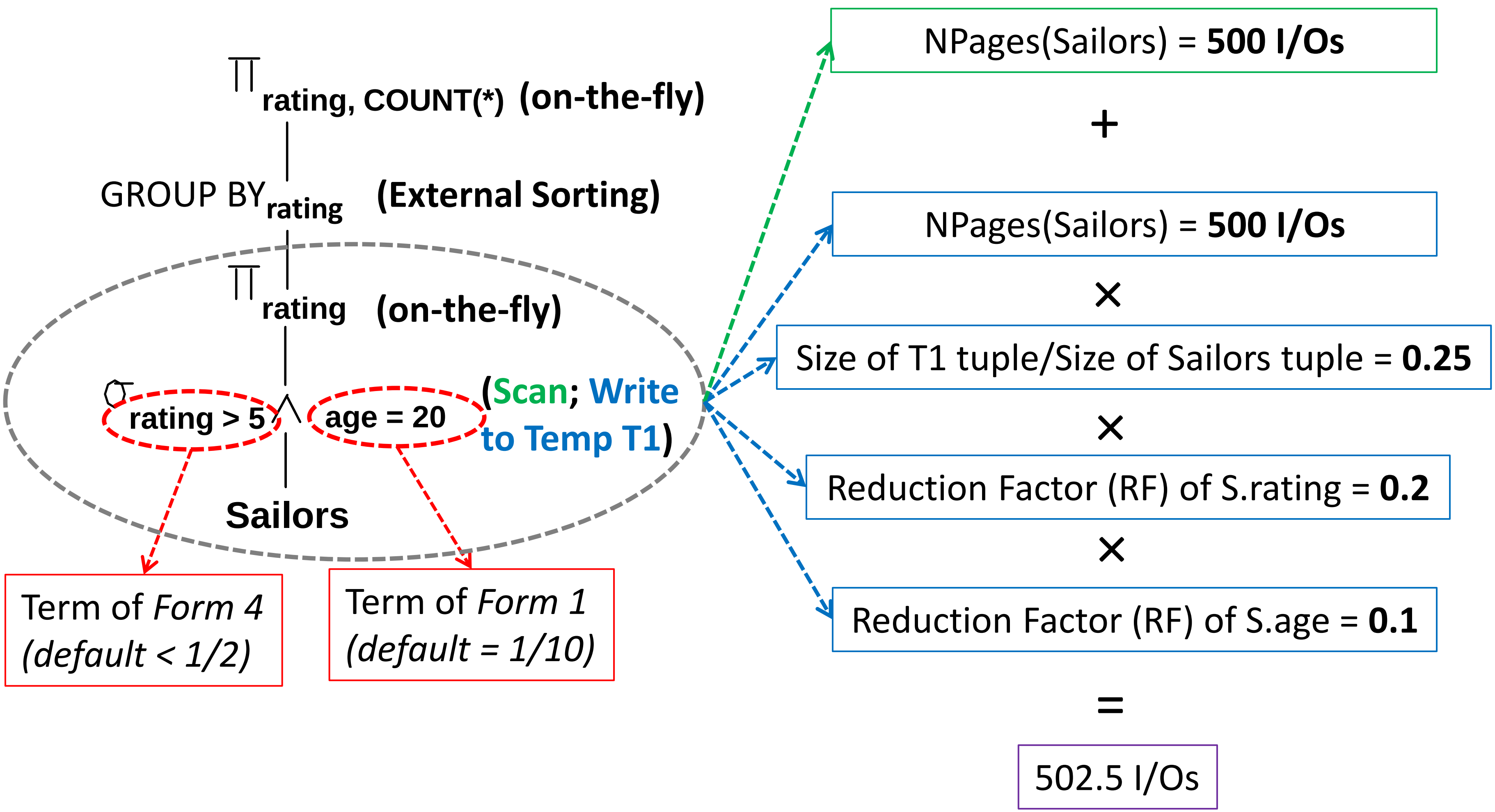
CASE I: Single-Relation Queries- An Example

- What would be the cost of we assume a <u>file scan</u> for sailors?



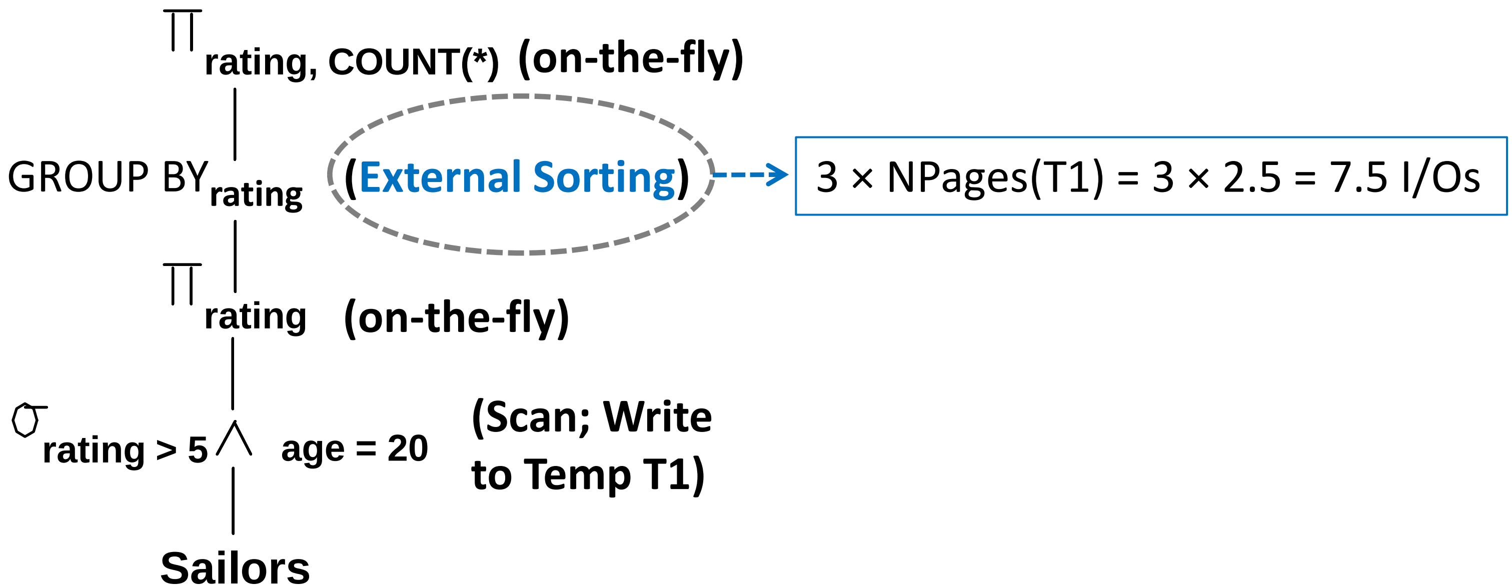
CASE I: Single-Relation Queries- An Example

- What would be the cost of we assume a file scan for sailors?



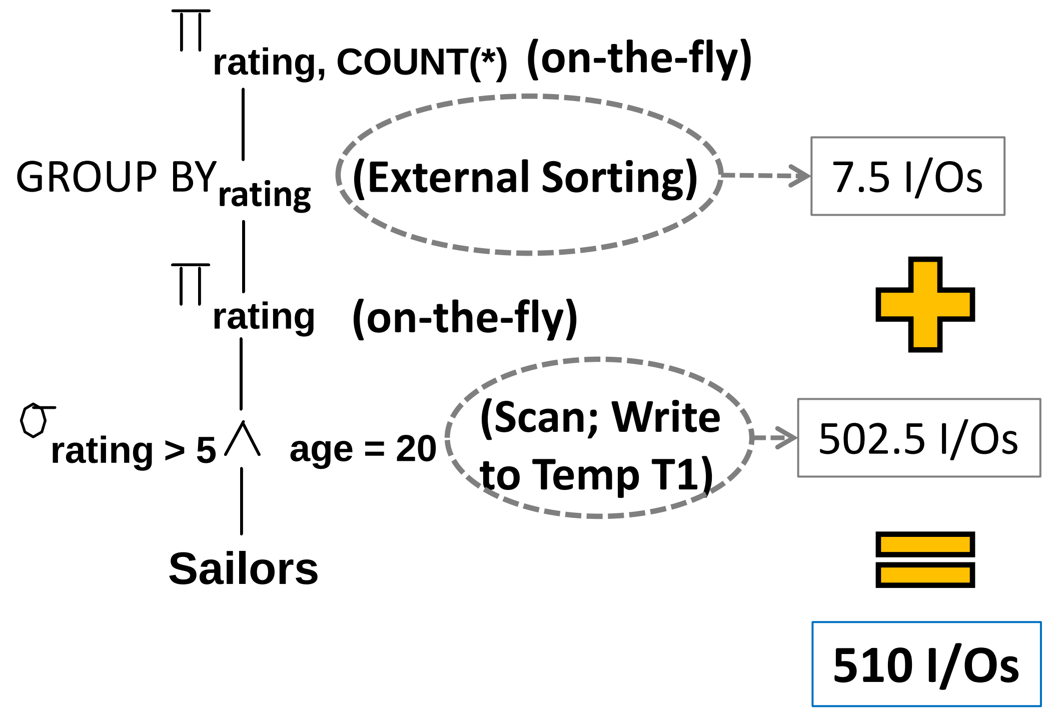
CASE I: Single-Relation Queries- An Example

- What would be the cost of we assume a <u>file scan</u> for sailors?



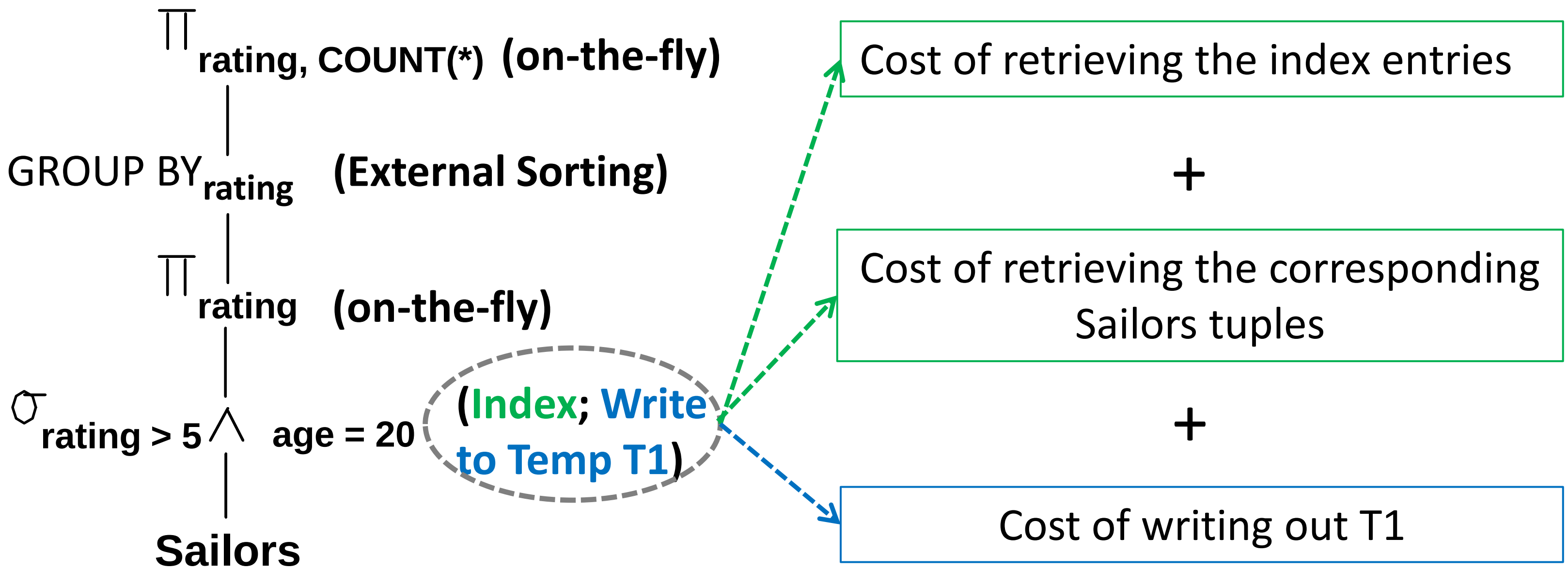
CASE I: Single-Relation Queries- An Example

- What would be the cost of we assume a file scan for sailors?



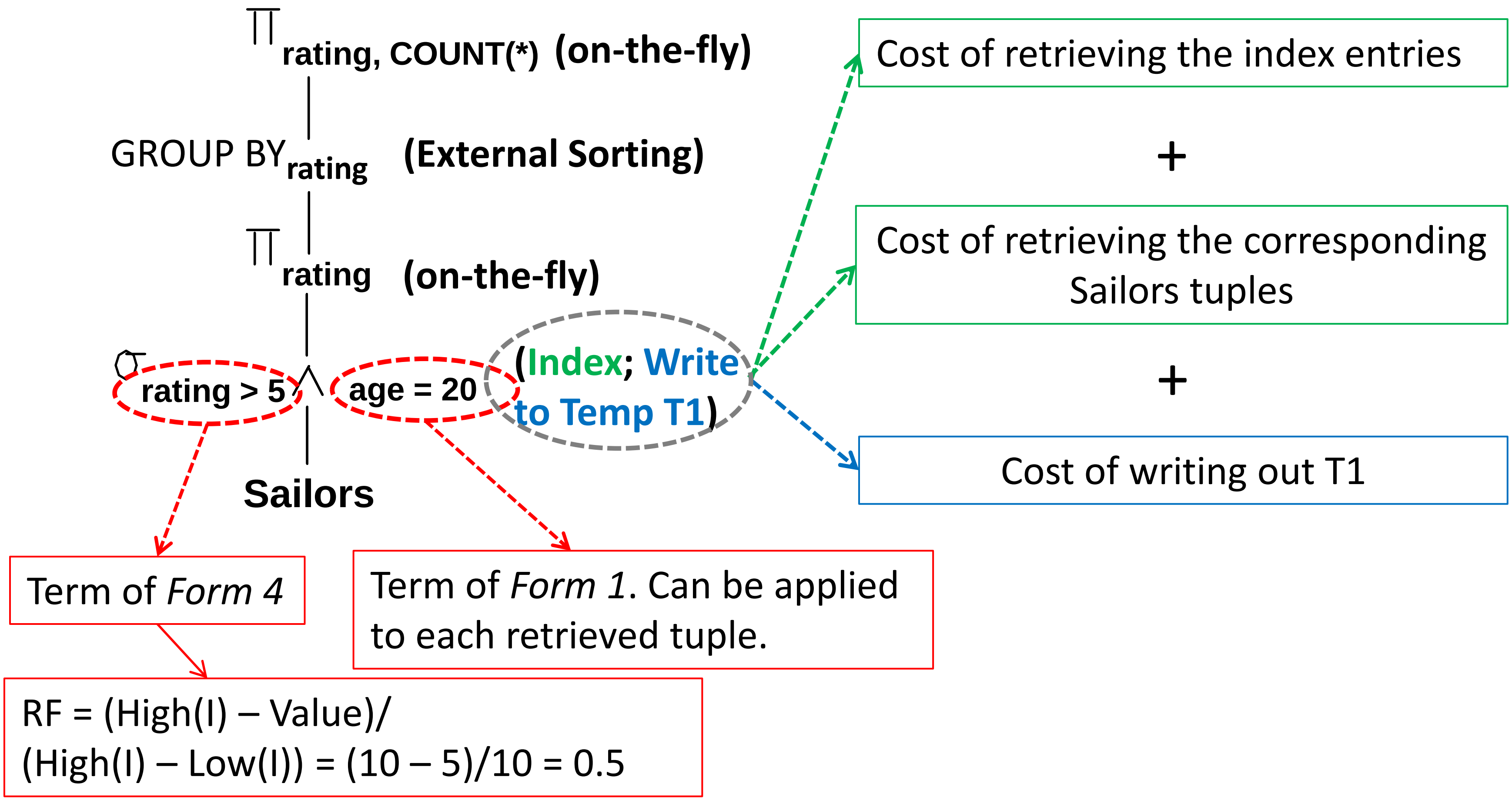
CASE I: Single-Relation Queries- An Example

- What would be the cost of we assume *a clustered index on rating with A(1)*?



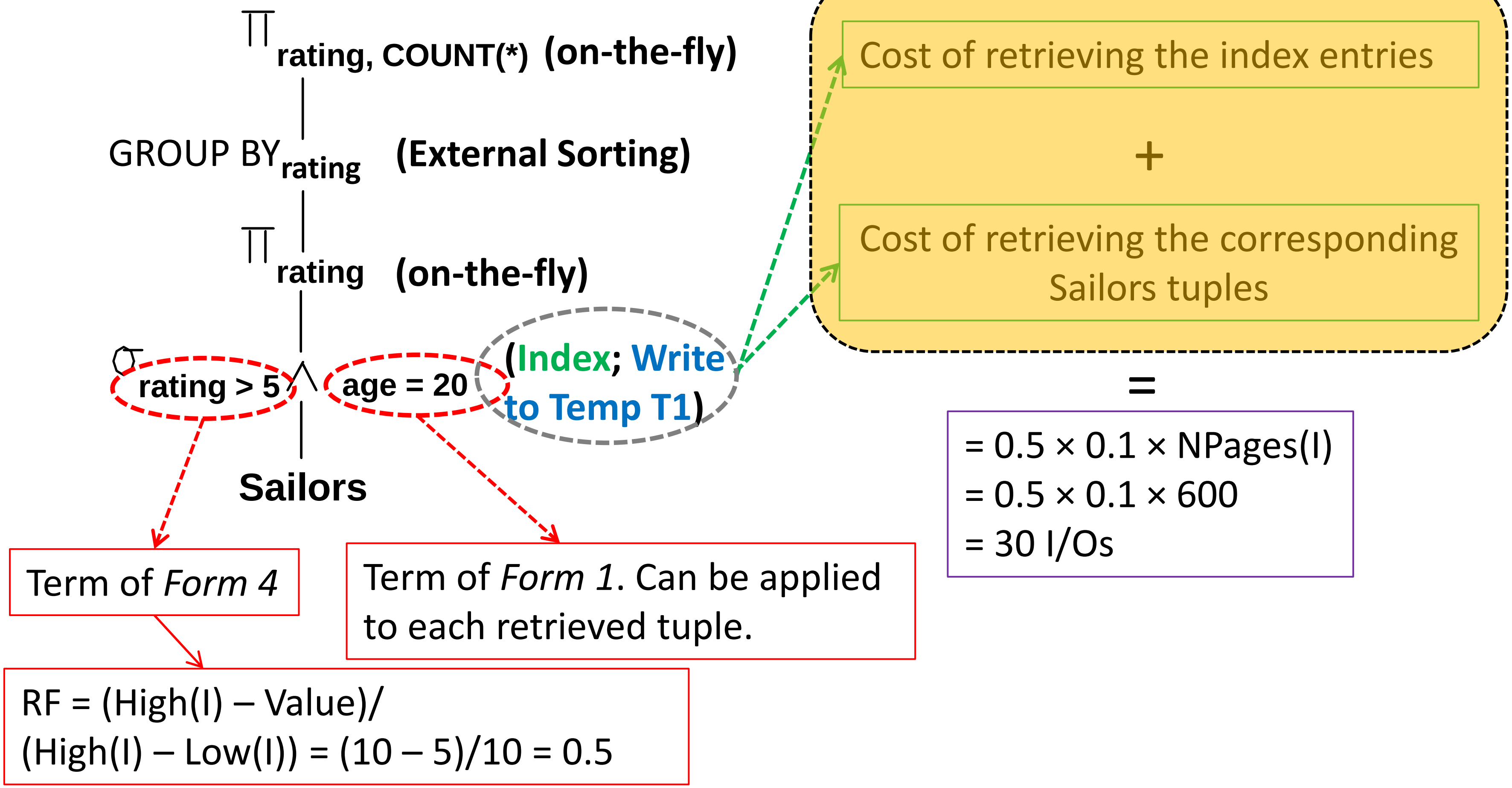
CASE I: Single-Relation Queries- An Example

- What would be the cost of we assume ***a clustered index on rating with A(1)***?



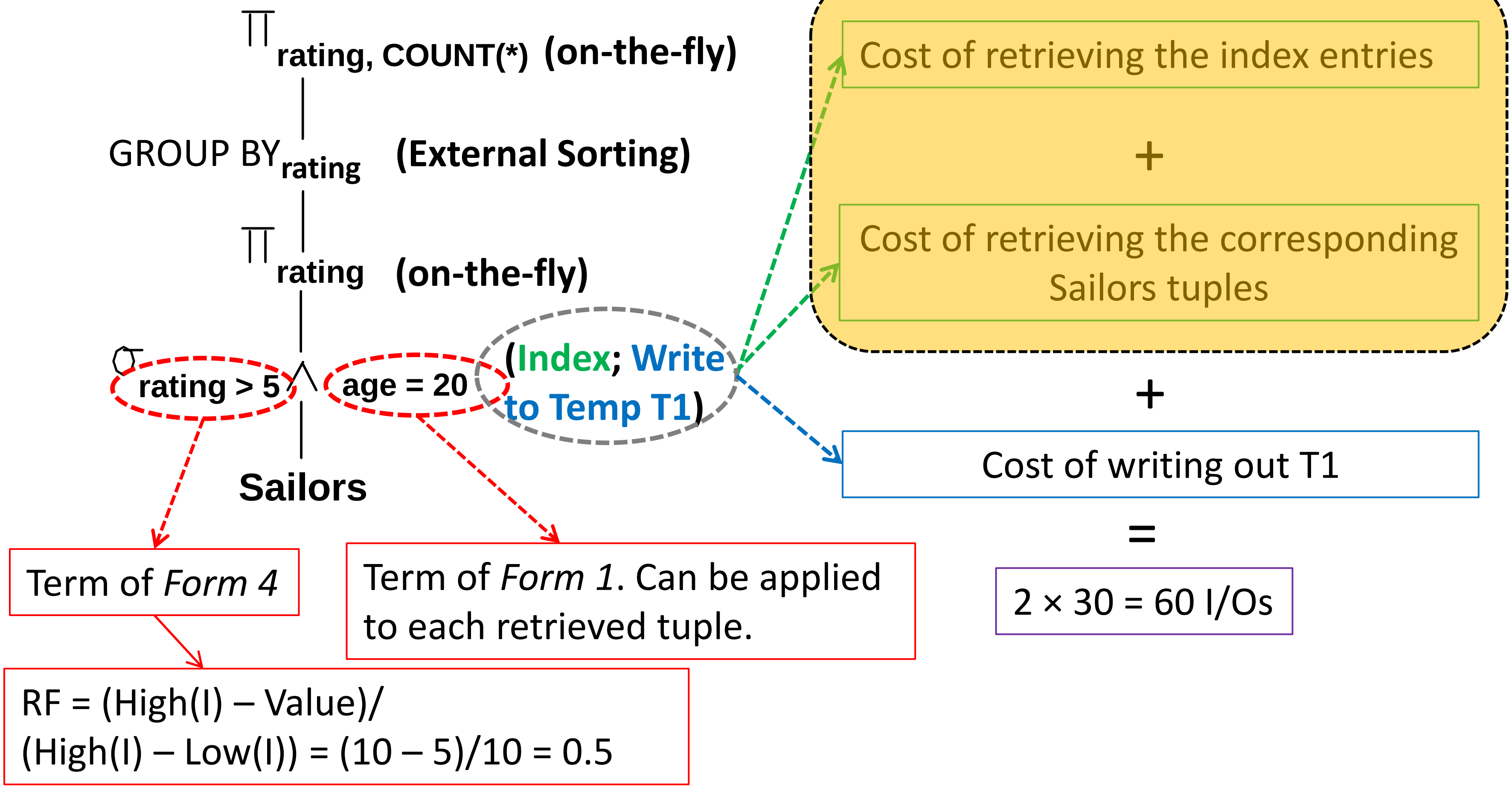
CASE I: Single-Relation Queries- An Example

- What would be the cost of we assume <u>*a clustered index on rating with A(1)*</u>?



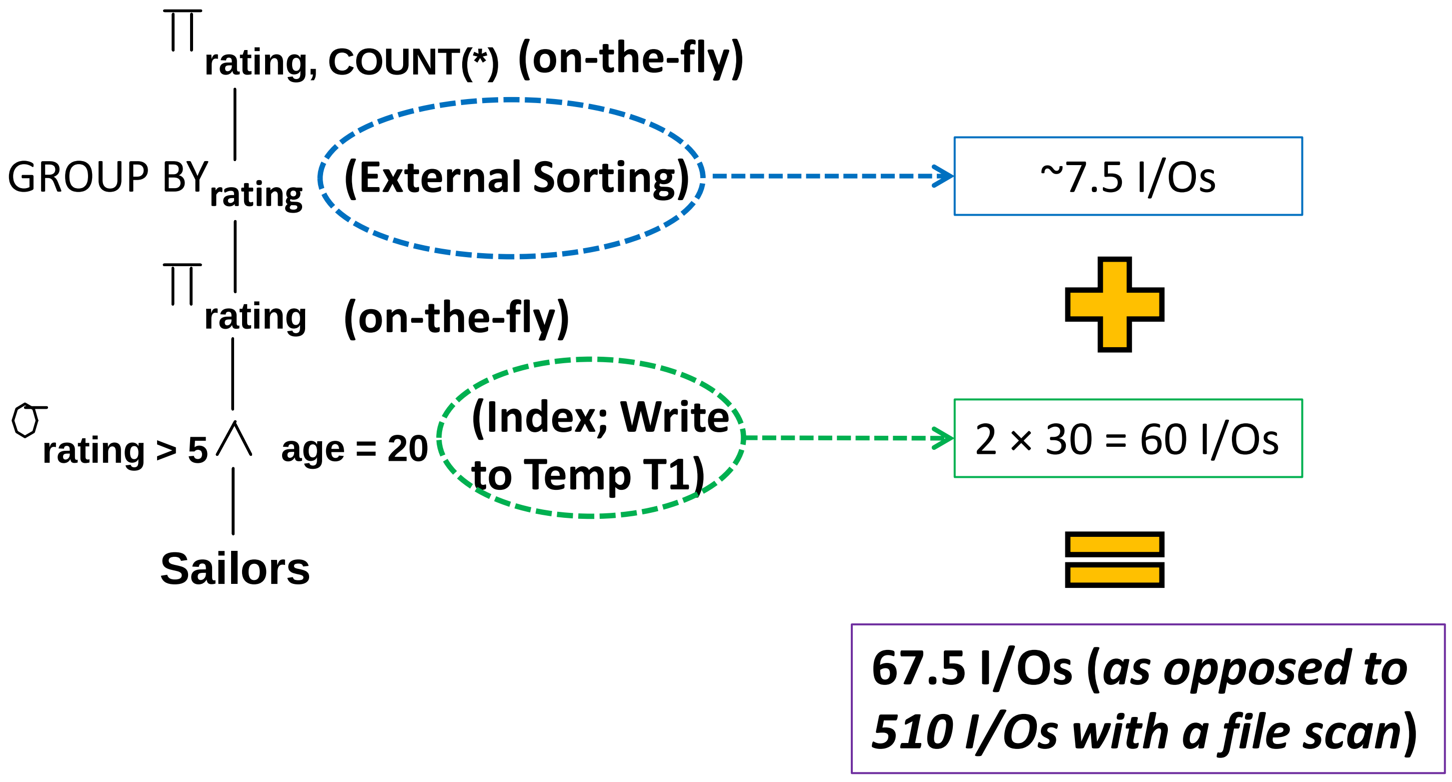
CASE I: Single-Relation Queries- An Example

- What would be the cost of we assume *a clustered index on rating with A(1)*?



CASE I: Single-Relation Queries- An Example

- What would be the cost of we assume *a clustered index on rating with A(1)*?



Towards a Dynamic Programming Algorithm

- There are two main cases to consider:
  - CASE I: Single-Relation Queries
  - CASE II: Multiple-Relation Queries

- CASE II: Multiple-Relation Queries
  - Only consider left-deep plans
  - Apply a dynamic programming algorithm

Enumeration of Left-Deep Plans Using Dynamic Programming

- Enumerate using ***N*** passes (if ***N*** relations joined):
  - **Pass 1:**
    - For each relation, enumerate all plans (all ***1***-relation plans)
    - Retain the cheapest plan per each relation
  - **Pass 2:**
    - Enumerate all ***2***-relation plans by considering each ***1***-relation plan retained in Pass 1 (as *outer*) and successively every other relation (as *inner*)
    - Retain the cheapest plan per each ***1***-relation plan
  - **Pass N:**
    - Enumerate all ***N***-relation plans by considering each (***N-1***)-relation plan retained in Pass N-1 (as outer) and successively every other relation (as inner)
    - Retain the cheapest plan per each (***N-1***)-relation plan
  - **Pick the cheapest N-relation plan**

Enumeration of Left-Deep Plans Using Dynamic Programming (*Cont'd*)

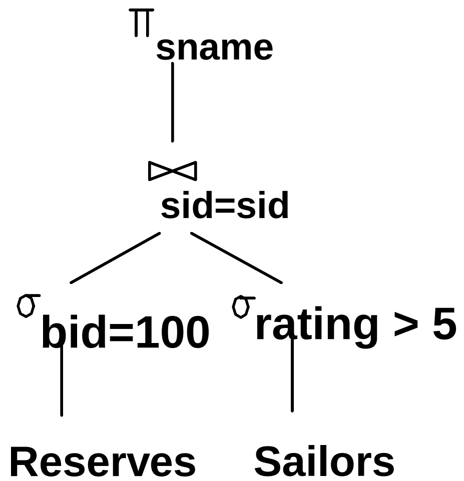
- An ***N-1*** way plan is not combined with an additional relation unless:
  - There is a join condition between them
  - All predicates in the WHERE clause have been used up

- ORDER BY, GROUP BY, and aggregate functions are handled as a final step, using either an `interestingly ordered' plan or an additional sorting operator

- In spite of pruning plan space, this approach is *still exponential* in the # of tables

CASE II: Multiple-Relation Queries- An Example

- Consider the following relational algebra tree:

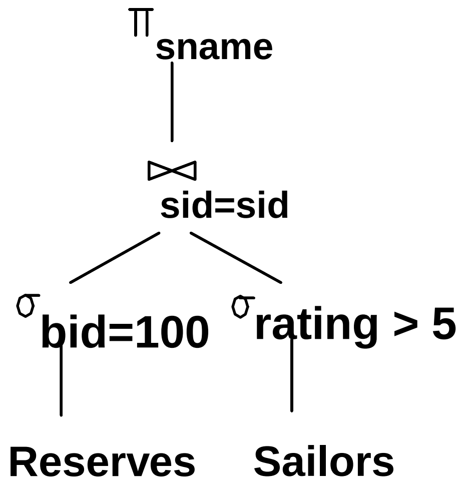


- Assume the following:

- Sailors:
  - B+ tree on *rating*
  - Hash on *sid*
- Reserves:
  - B+ tree on *bid*

CASE II: Multiple-Relation Queries- An Example

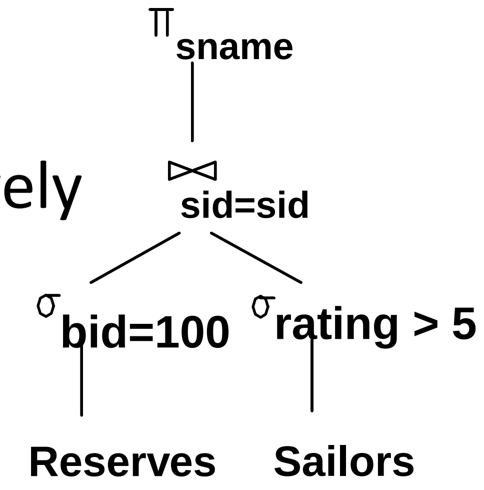
- **Pass 1:**
  - Sailors:
    - B+ tree matches rating>5, and is *probably* the cheapest
    - If this selection is expected to retrieve a lot of tuples, and the index is un-clustered, file scan might be cheaper!
  - Reserves: B+ tree on *bid* matches *bid=500*; *probably* the cheapest



- Sailors:
  - B+ tree on *rating*
  - Hash on *sid*
- Reserves:
  - B+ tree on *bid*

CASE II: Multiple-Relation Queries- An Example

- **Pass 2:**
  - Consider each plan retained from **Pass 1** as the outer, and join it effectively with every other relation
  - E.g., Reserves as outer:
    - Hash index can be used to get Sailors tuples that satisfy sid = outer tuple's sid value



- Sailors:
  - B+ tree on *rating*
  - Hash on *sid*
- Reserves:
  - B+ tree on *bid*

Outline

- A Brief Primer on Query Optimization
- Query Evaluation Plans
- Relational Algebra Equivalences
- Estimating Plan Costs
- Enumerating Plans
- Nested Sub-Queries ✓

Nested Sub-queries

- Consider the following nested query ***Q1***:

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

- The nested sub-query can be evaluated *just once*, yielding a <u>single</u> value ***V***

- ***V*** can be incorporated into the top-level query as if it had been part of the original statement of ***Q1***

Nested Sub-queries

- Now, consider the following nested query ***Q2***:

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

- The nested sub-query can still be evaluated *just once*, but it will yield a collection of *sids*

- Every *sid* value in Sailors must be checked whether it exists in the collection of sids returned by the nested sub-query
  - This entails a join, and the full range of join methods can be explored!

Nested Sub-queries

- Now, consider another nested query ***Q3***:

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

- Q3 is *correlated*; hence, we "cannot" evaluate the sub-query just once!

- In this case, the typical evaluation strategy is to evaluate the nested sub-query *for each tuple* of Sailors

The common approach, indeed, is to *always* do nested loops join!

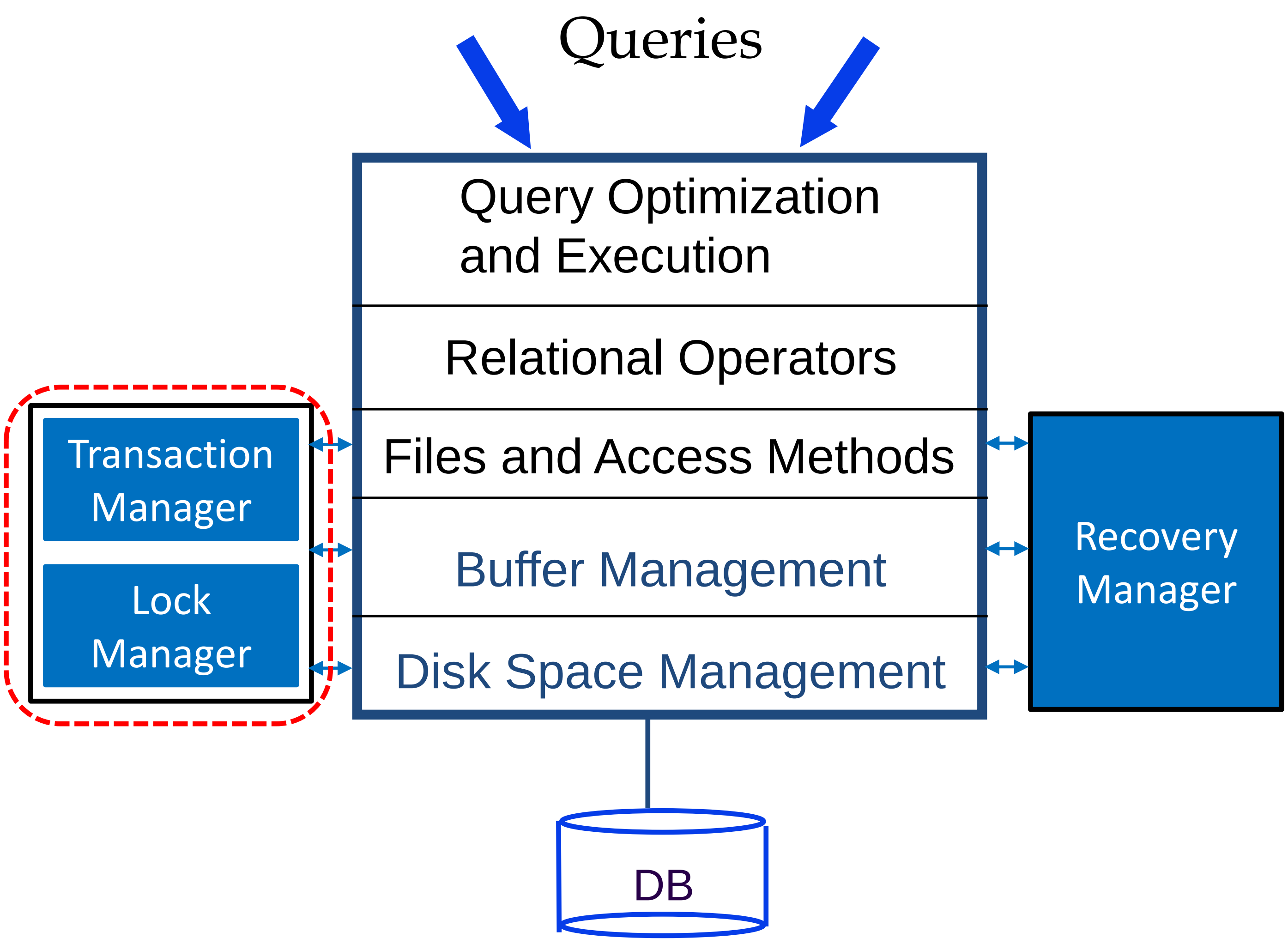
Summary

- Query optimization is a crucial task in a relational DBMSs

- We must understand query optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries)

- Two parts to optimizing a query:
  1. Consider a set of alternative plans (e.g., using dynamic programming)
     - Apply selections/projections as early as possible
     - Prune search space; typically, keep left-deep plans only
  2. Estimate the cost of each plan that is considered
     - Must estimate *size of result* and *cost of each tree node*
     - *Key issues*: Statistics, indexes, operator implementations

Next Class


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