Database Applications (15-415)

DBMS Internals- Part VIII
Lecture 19, March 31, 2015

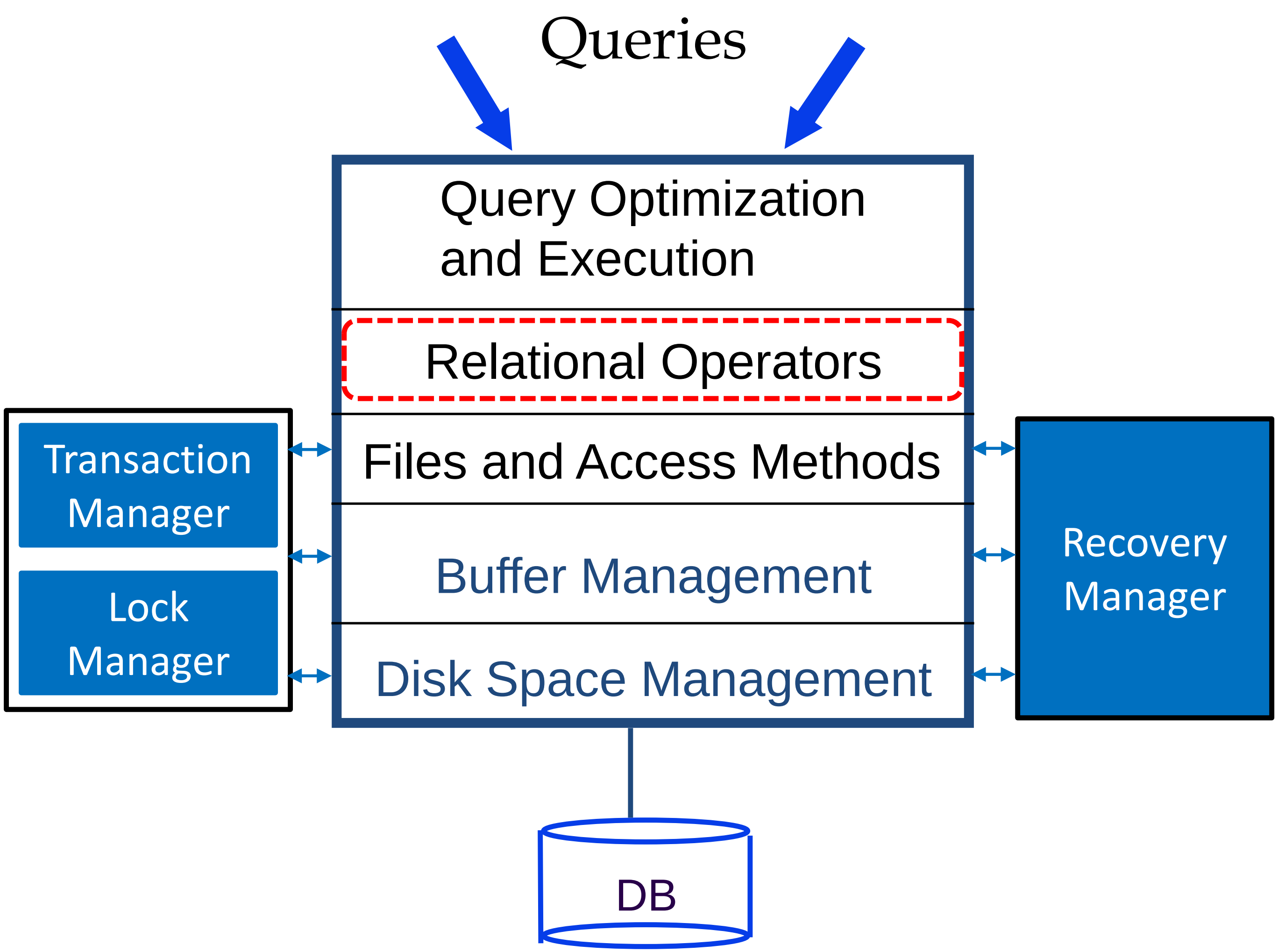
Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Today…

- Last Session:
  - DBMS Internals- Part VII
    - Algorithms for Relational Operations (*Cont'd*)
- Today's Session:
  - DBMS Internals- Part VIII
    - Algorithms for Relational Operations (*Cont'd*)
    - Introduction to Query Optimization
- Announcements:
  - Project 2 grades will be out on Thursday, April 2nd
  - Project 3 is due on Thursday, April 2nd by midnight
  - PS4 will be posted by Thursday, April 2nd. It will be due on April 16th by midnight
  - Quiz II will be held 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

- The Join Operation (Cont'd) ✓
- The Set Operations
- The Aggregate Operations
- Introduction to Query Optimization

The Join Operation

- We will study *five* join algorithms, *two* which enumerate the cross-product and *three* which do not

- Join algorithms which enumerate the cross-product:
  - Simple Nested Loops Join
  - Block Nested Loops Join

- Join algorithms which do not enumerate the cross-product:
  - Index Nested Loops Join
  - Sort-Merge Join
  - Hash Join ✓

Hash Join

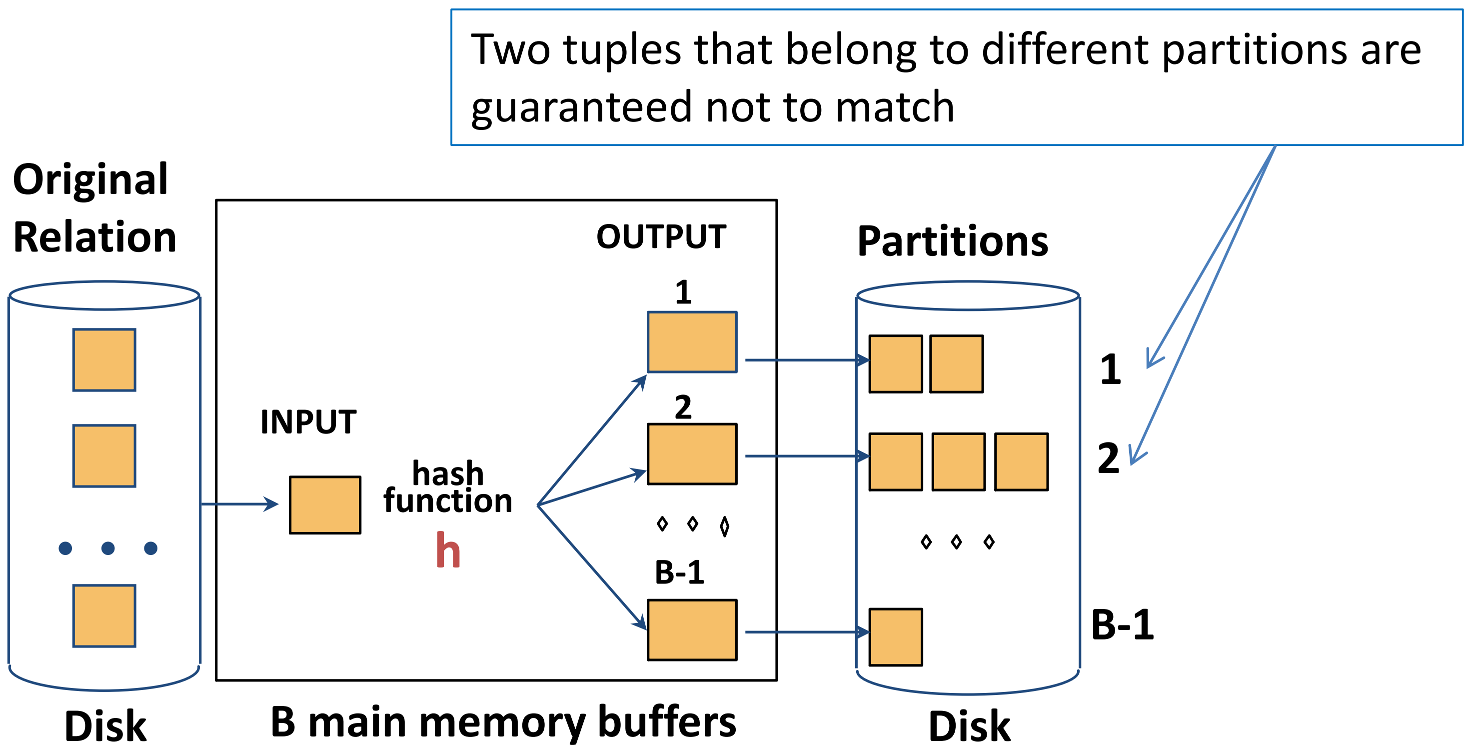
- The join algorithm based on hashing has two phases:
  - Partitioning (also called *Building*) Phase
  - Probing (also called *Matching*) Phase

- Idea: Hash both relations on the join attribute into ***k*** partitions, using the same hash function ***h***

- Premise: R tuples in partition ***i*** can join only with S tuples in the same partition ***i***

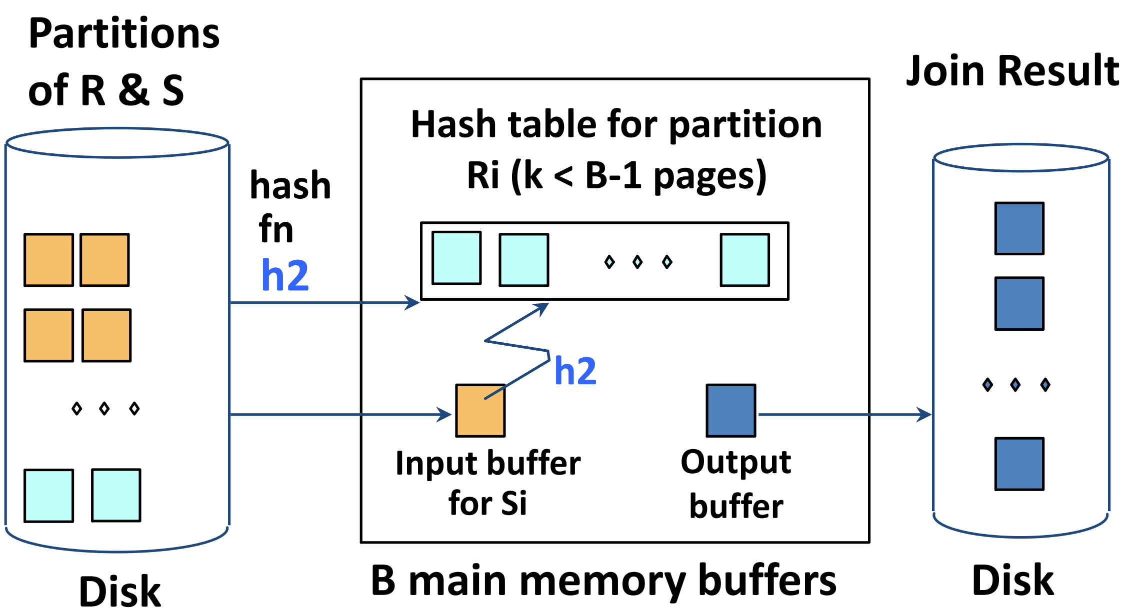
Hash Join: Partitioning Phase

- Partition both relations using hash function ***h***



Hash Join: Probing Phase

- Read in a partition of R, hash it using ***h2* (<> *h*)**
- Scan the corresponding partition of S and search for matches



Hash Join: Cost

- What is the cost of the partitioning phase?
  - We need to scan R and S, and write them out once
  - Hence, cost is 2(M+N) I/Os

- What is the cost of the probing phase?
  - We need to scan each partition once (*assuming no partition overflows*) of R and S
  - Hence, cost is M + N I/Os

- Total Cost = 3 (M + N)

Hash Join: Cost (*Cont'd*)

- Total Cost = 3 (M + N)

- Joining Reserves and Sailors would cost 3 (500 + 1000) = 4500 I/Os

- Assuming 10ms per I/O, hash join takes less than 1 minute!

- This underscores the importance of using a good join algorithm (e.g., *Simple NL Join takes ~140 hours!*)

But, so far we have been assuming that partitions fit in memory!

Memory Requirements and Overflow Handling

- How can we increase the chances for a given partition in the probing phase to fit in memory?
  - Maximize the number of partitions in the building phase

- If we partition R (or S) into ***k*** partitions, what would be the size of each partition (in terms of B)?
  - At least ***k*** output buffer pages and 1 input buffer page
  - Given B buffer pages, ***k*** = B – 1
  - Hence, the size of an R (or S) partition = M/B-1

- What is the number of pages in the (in-memory) hash table built during the probing phase per a partition?
  - ***f***.M/B-1, where ***f*** is a *fudge factor*

Memory Requirements and Overflow Handling

- What do we need else in the probing phase?
  - A buffer page for scanning the S partition
  - An output buffer page

- What is a good value of B as such?
  - B > f.M/B-1 + 2
  - Therefore, we need ~$B > \sqrt{f.M}$

- What if a partition overflows?
  - Apply the hash join technique *recursively* (as is the case with the projection operation)

Hash Join vs. Sort-Merge Join

- If $B > \sqrt{M}$ (M is the # of pages in the *smaller* relation) and we assume uniform partitioning, the cost of hash join is 3(M+N) I/Os

- If $B > \sqrt{N}$ (N is the # of pages in the *larger* relation), the cost of sort-merge join is 3(M+N) I/Os

Which algorithm to use, hash join or sort-merge join?

Hash Join vs. Sort-Merge Join

- If the available number of buffer pages falls between $\sqrt{M}$ and $\sqrt{N}$, hash join is preferred (why?)
- Hash Join shown to be highly parallelizable (*beyond the scope of the class*)
- Hash join is sensitive to data skew while sort-merge join is not
- Results are sorted after applying sort-merge join (may help "upstream" operators)
- Sort-merge join goes fast if one of the input relations is already sorted

The Join Operation

- We will study *five* join algorithms, *two* which enumerate the cross-product and *three* which do not

- Join algorithms which enumerate the cross-product:
  - Simple Nested Loops Join
  - Block Nested Loops Join

  ✓

- Join algorithms which <u>do not</u> enumerate the cross-product:
  - Index Nested Loops Join
  - Sort-Merge Join
  - Hash Join

  ✓

General Join Conditions

- Thus far, we assumed a *single equality join condition*

- Practical cases include join conditions with several equality (e.g., *R.sid=S.sid* AND *R.rname=S.sname*) and/or inequality (e.g., *R.rname < S.sname*) conditions

- We will discuss two cases:
  - Case 1: a join condition with several equalities
  - Case 2: a join condition with an inequality comparison

General Join Conditions: Several Equalities

- Case 1: a join condition with several equalities (e.g., *R.sid=S.sid* AND *R.rname=S.sname*)
  - Simple NL join and Block NL join are unaffected
  - For index NL join, we can build an index on Reserves using the composite key (sid, rname) and treat Reserves as the inner relation
  - For sort-merge join, we can sort Reserves on the composite key (sid, rname) and Sailors on the composite key (sid, sname)
  - For hash join, we can partition Reserves on the composite key (sid, rname) and Sailors on the composite key (sid, sname)

General Join Conditions: An Inequality

- Case 2: a join condition with an inequality comparison (e.g., *R.rname < S.sname*)
  - Simple NL join and Block NL join are unaffected
  - For index NL join, we require a B+ tree index
  - Sort-merge join and hash join are not applicable!

Outline

- The Join Operation (Cont'd)
- The Set Operations ✓
- The Aggregate Operations
- Introduction to Query Optimization

Set Operations

- R ∩ S is a special case of join!
  - Q: How?
  - A: With equality on *all* fields in the join condition

- R × S is a special case of join!
  - Q: How?
  - A: With no join condition

- How to implement R U S and R – S?
  - Algorithms based on sorting
  - Algorithms based on hashing

Union and Difference Based on Sorting

- How to implement R U S based on sorting?
  - Sort R and S
  - Scan sorted R and S (in parallel) and merge them, eliminating duplicates

- How to implement R – S based on sorting?
  - Sort R and S
  - Scan sorted R and S (in parallel) and write only tuples of R that do not appear in S

Union and Difference Based on Hashing

- How to implement R U S based on hashing?
  - Partition R and S using a hash function ***h***
  - For each S-partition, build in-memory hash table (using ***h2***)
  - Scan R-partition which corresponds to S-partition and write out tuples while discarding duplicates
- How to implement R – S based on hashing?
  - Partition R and S using a hash function ***h***
  - For each S-partition, build in-memory hash table (using *h2*)
  - Scan R-partition which corresponds to S-partition and write out tuples which are in R-partition but not in S-partition

Outline

- The Join Operation (Cont'd)
- The Set Operations
- The Aggregate Operations ✓
- Introduction to Query Optimization

Aggregate Operations

- Assume the following SQL query Q1:

```
SELECT AVG(S.age)
FROM Sailors S
```

- How to evaluate Q1?
  - Scan Sailors
  - Maintain the average on age

- In general, we implement aggregate operations by:
  - Scanning the input relation
  - Maintaining some *running information* (e.g., total for SUM and smaller for MIN)

Aggregate Operations

- Assume the following SQL query Q2:

```
SELECT AVG(S.age)
FROM Sailors S
GROUP BY S.rating
```

- How to evaluate Q2?
  - An algorithm based on sorting ✓
  - An algorithm based on hashing

- Algorithm based on sorting:
  - Sort Sailors on rating
  - Scan sorted Sailors and compute the average for each rating group

Aggregate Operations

- Assume the following SQL query Q2:

```
SELECT AVG(S.age)
FROM Sailors S
GROUP BY S.rating
```

- How to evaluate Q2?
  - An algorithm based on sorting
  - An algorithm based on hashing ✓

- Algorithm based on hashing:
  - Build a hash table on rating
  - Scan Sailors and for each tuple *t*, probe its corresponding hash bucket and update average

Aggregate Operations

- Assume the following SQL query Q2:

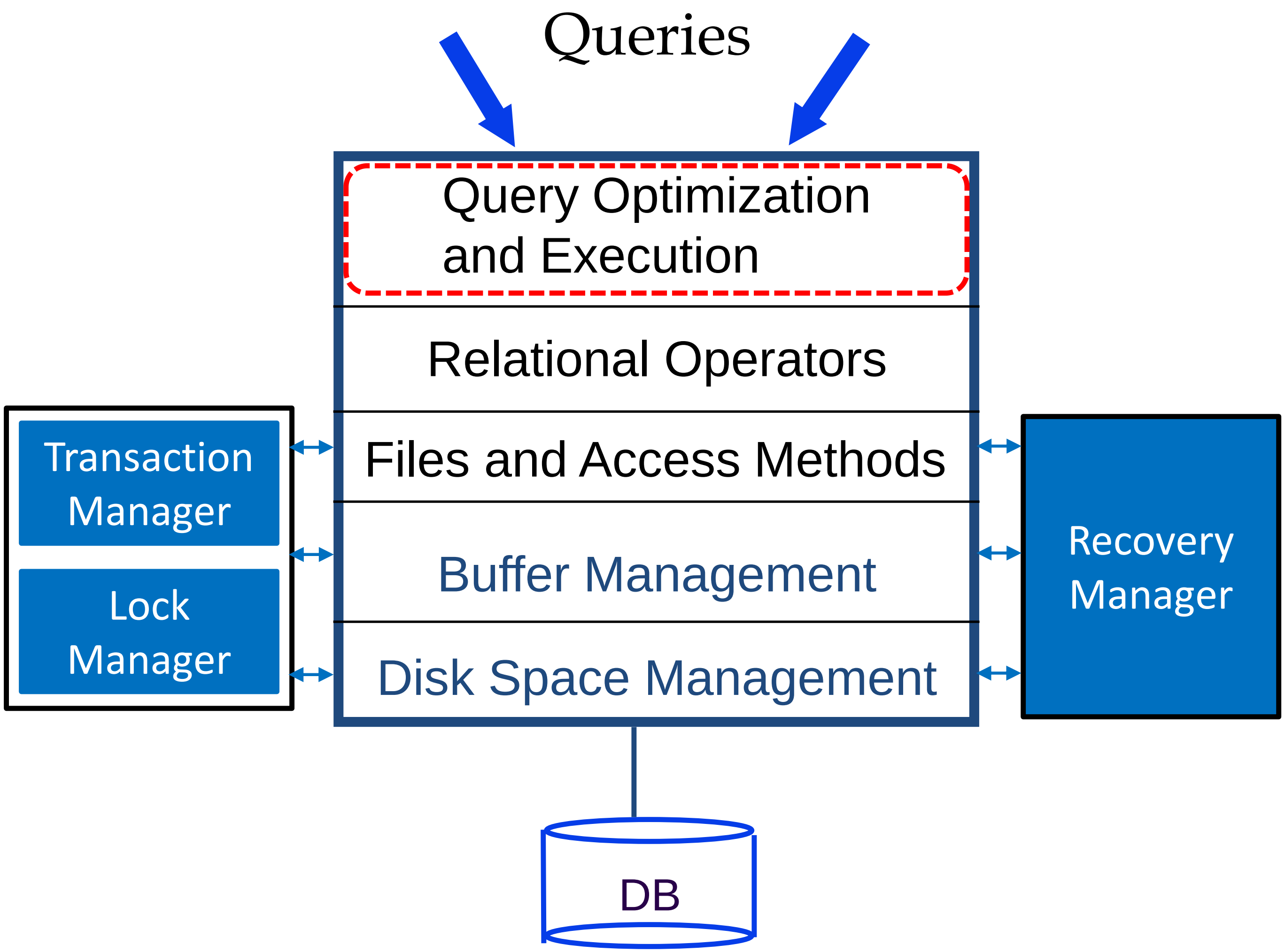
```
SELECT AVG(S.age)
FROM Sailors S
GROUP BY S.rating
```

- How to evaluate Q2 with the existence of an index?
  - If group-by attributes form *prefix* of search key, we can retrieve data entries/tuples in group-by order and thereby avoid sorting
  - If the index is a tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, we can pursue an *index-only scan*

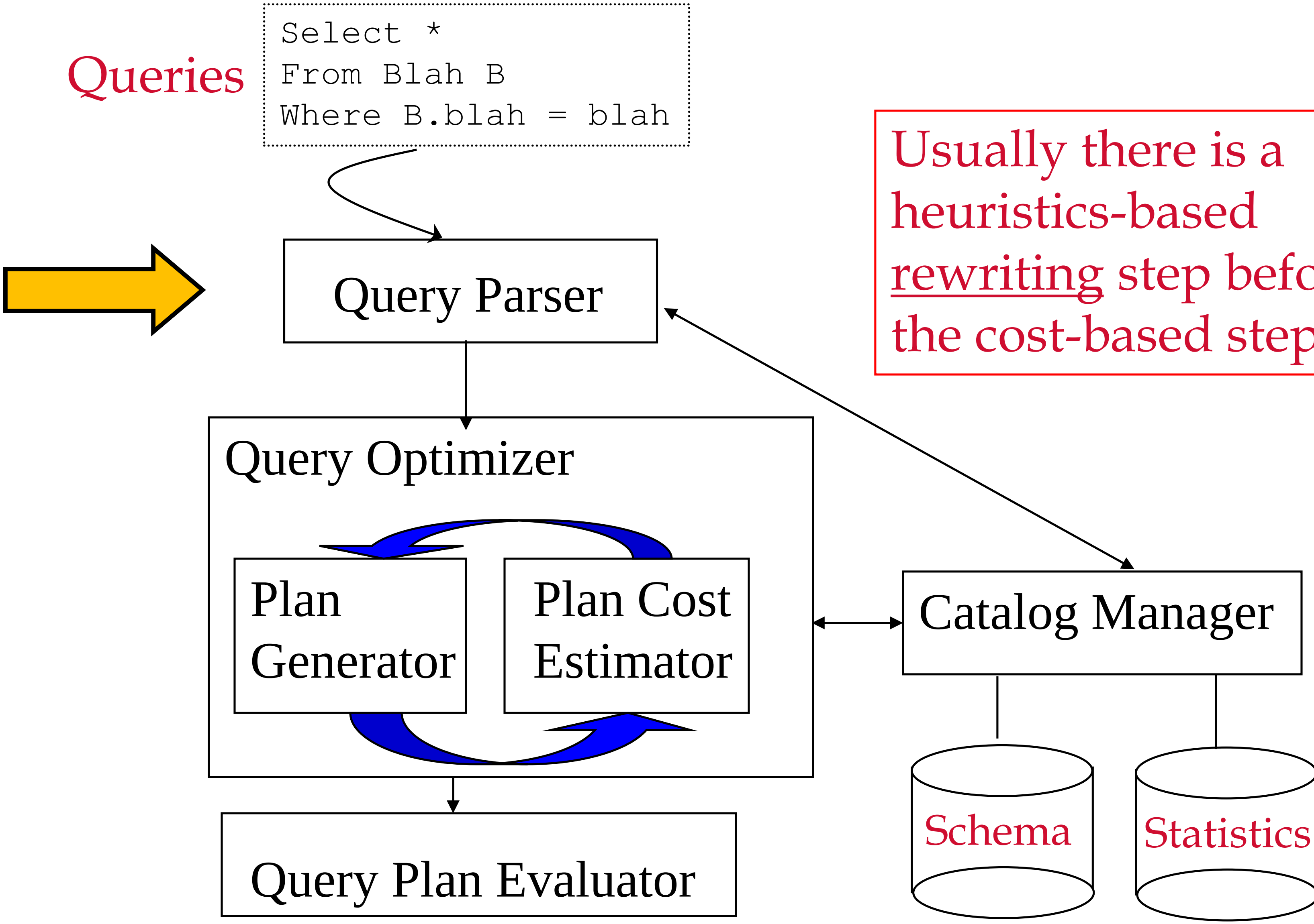
Outline

- The Join Operation (Cont'd)
- The Set Operations
- The Aggregate Operations
- Introduction to Query Optimization ✓

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

Cost-Based Query Sub-System



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

Query Optimization Steps

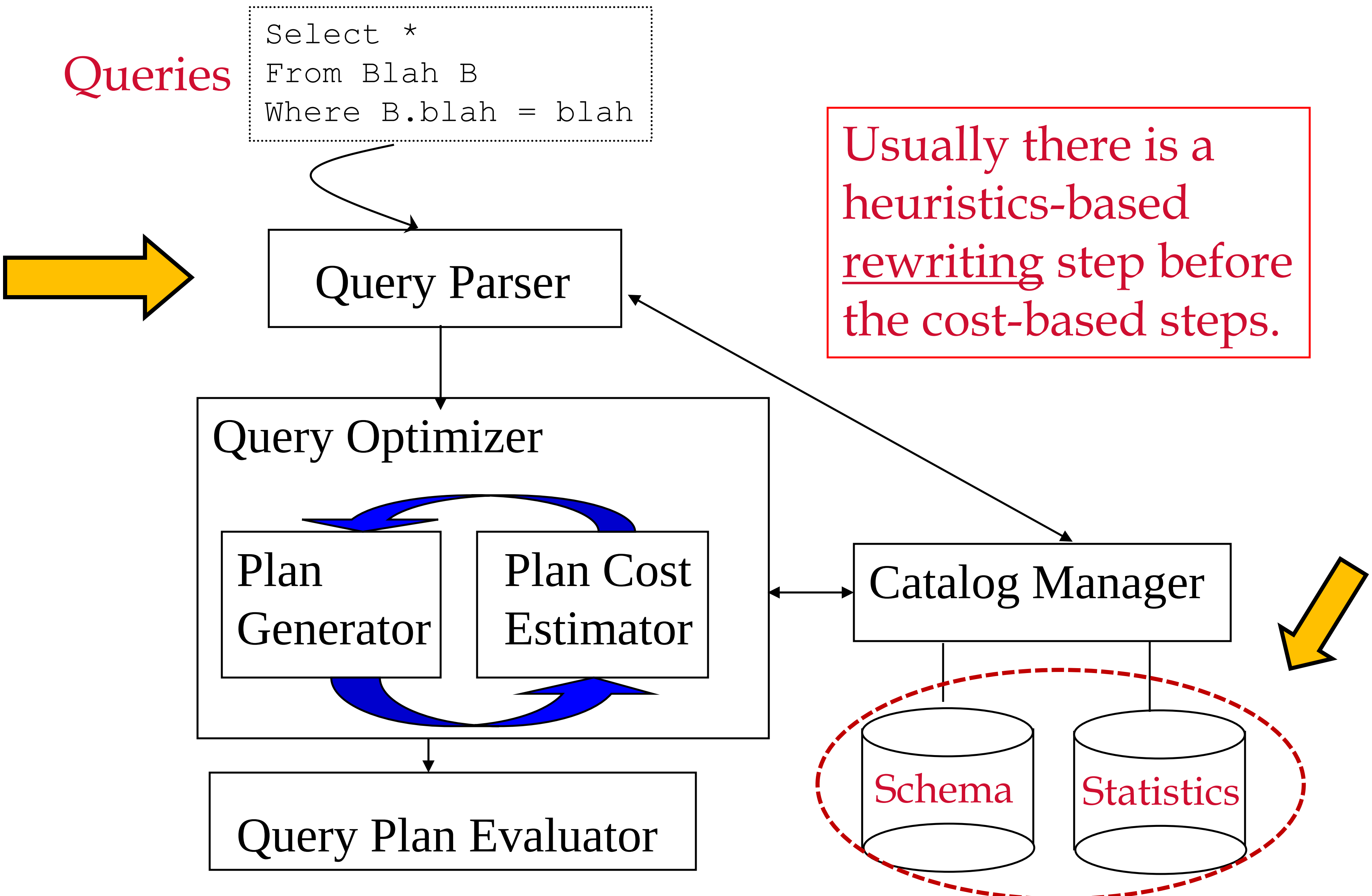
- Step 1: Queries are parsed into internal forms (e.g., parse trees)
- Step 2: Internal forms are transformed into 'canonical forms' (syntactic query optimization)
- Step 3: A *subset* of alternative plans are enumerated
- Step 4: Costs for alternative plans are estimated
- Step 5: The query evaluation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

SQL Blocks

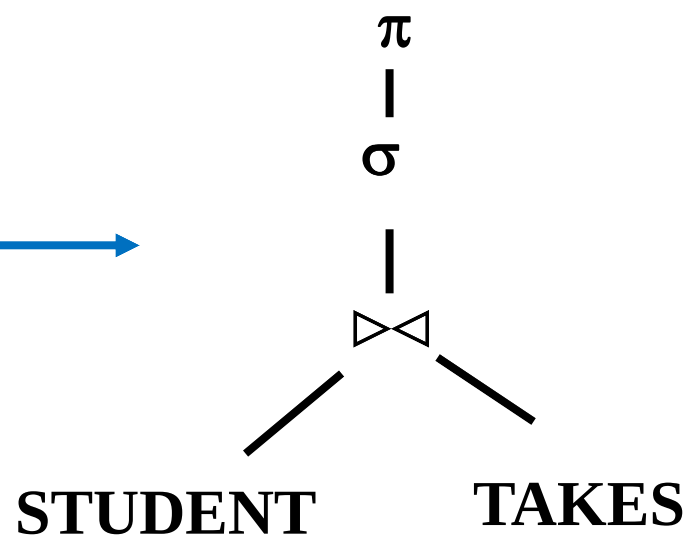
- SQL queries are optimized by *decomposing* them into a collection of smaller units, called blocks

- 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

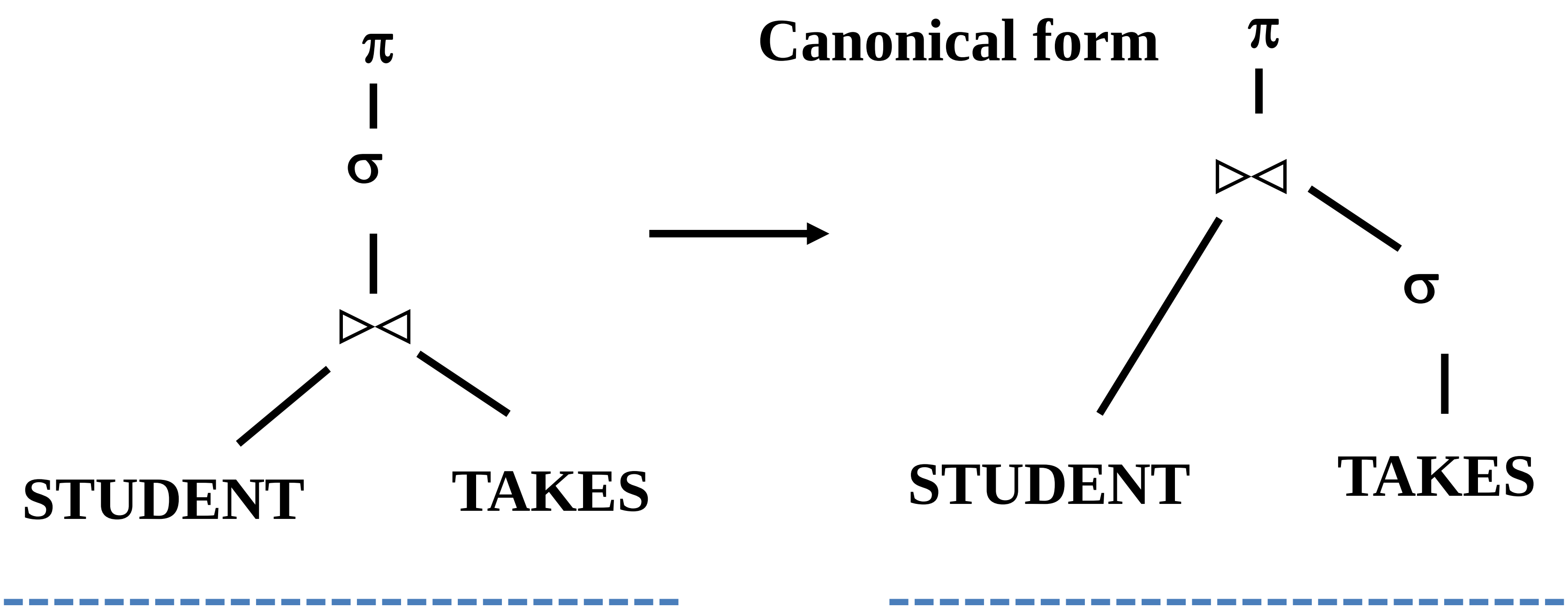
```
select name
from STUDENT, TAKES
where c-id='415' and
STUDENT.ssn=TAKES.ssn
```



- 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

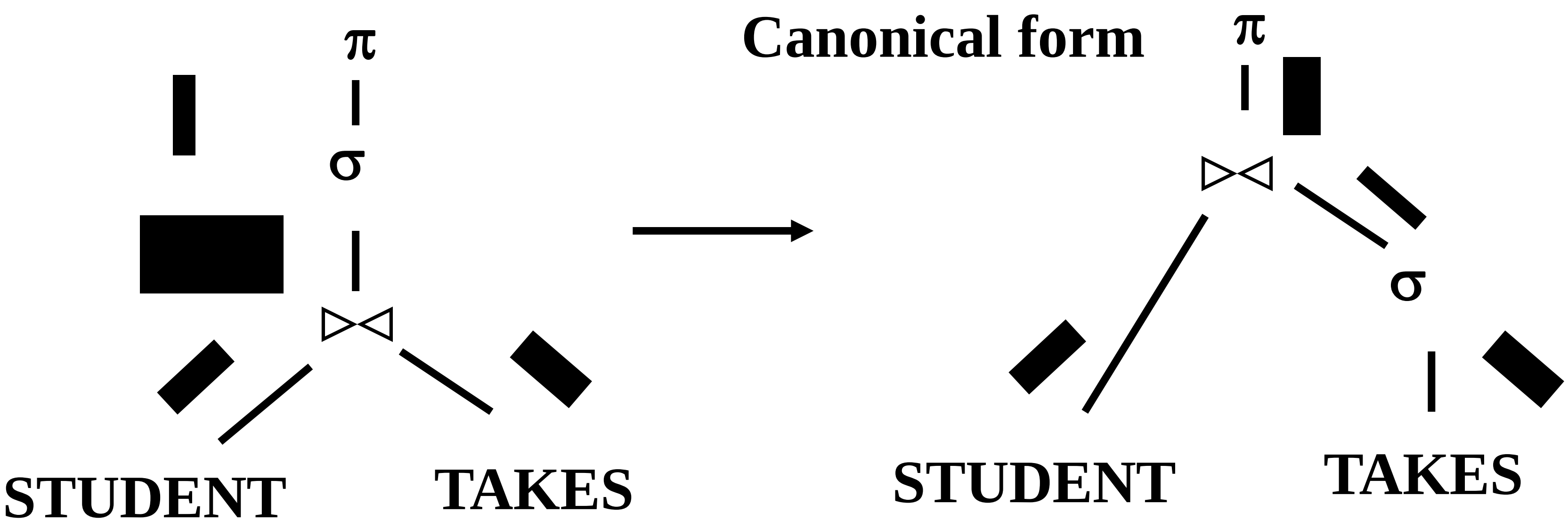
Translating SQL Queries Into Relational Algebra Trees (*Cont'd*)



Still the same result!

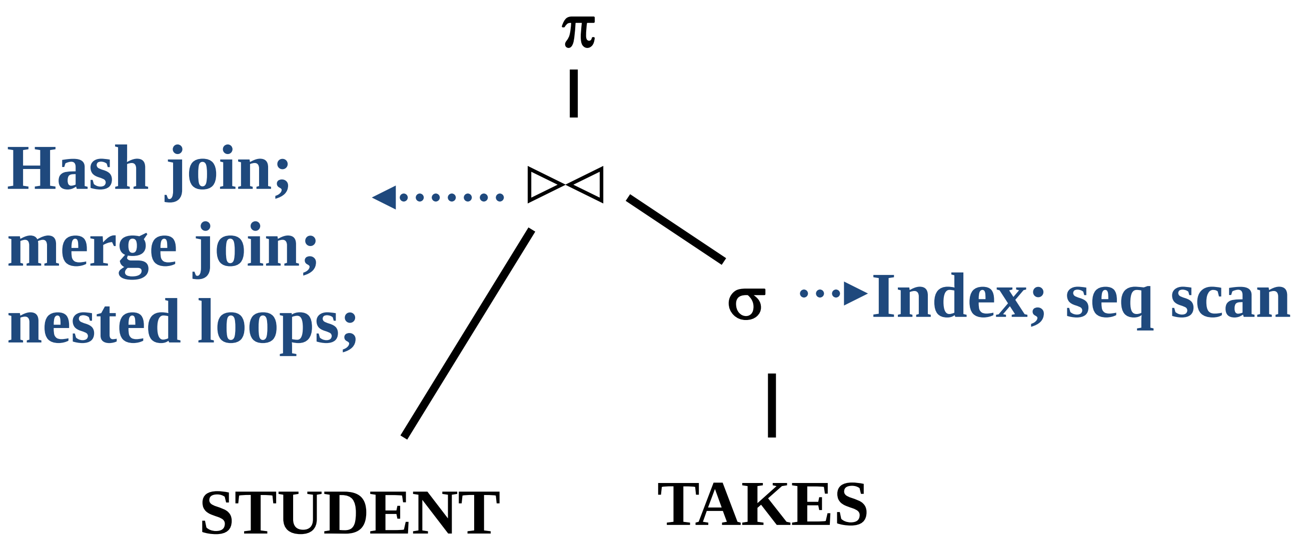
How can this be guaranteed? Next class!

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



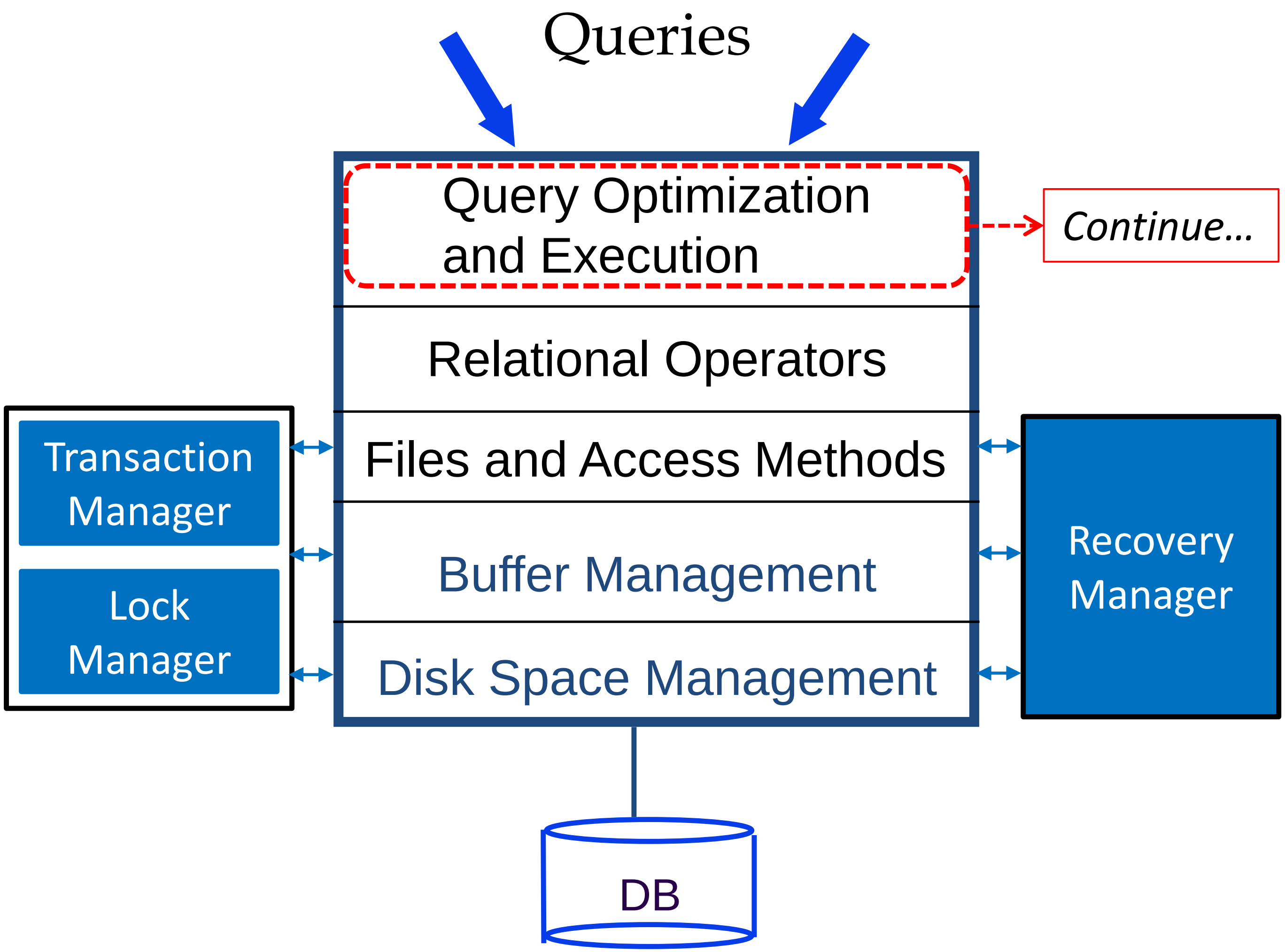
OBSERVATION: try to perform selections and projections early!

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



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

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