COMP9120 Database Management Systems

Week 11: Query Processing and Evaluation

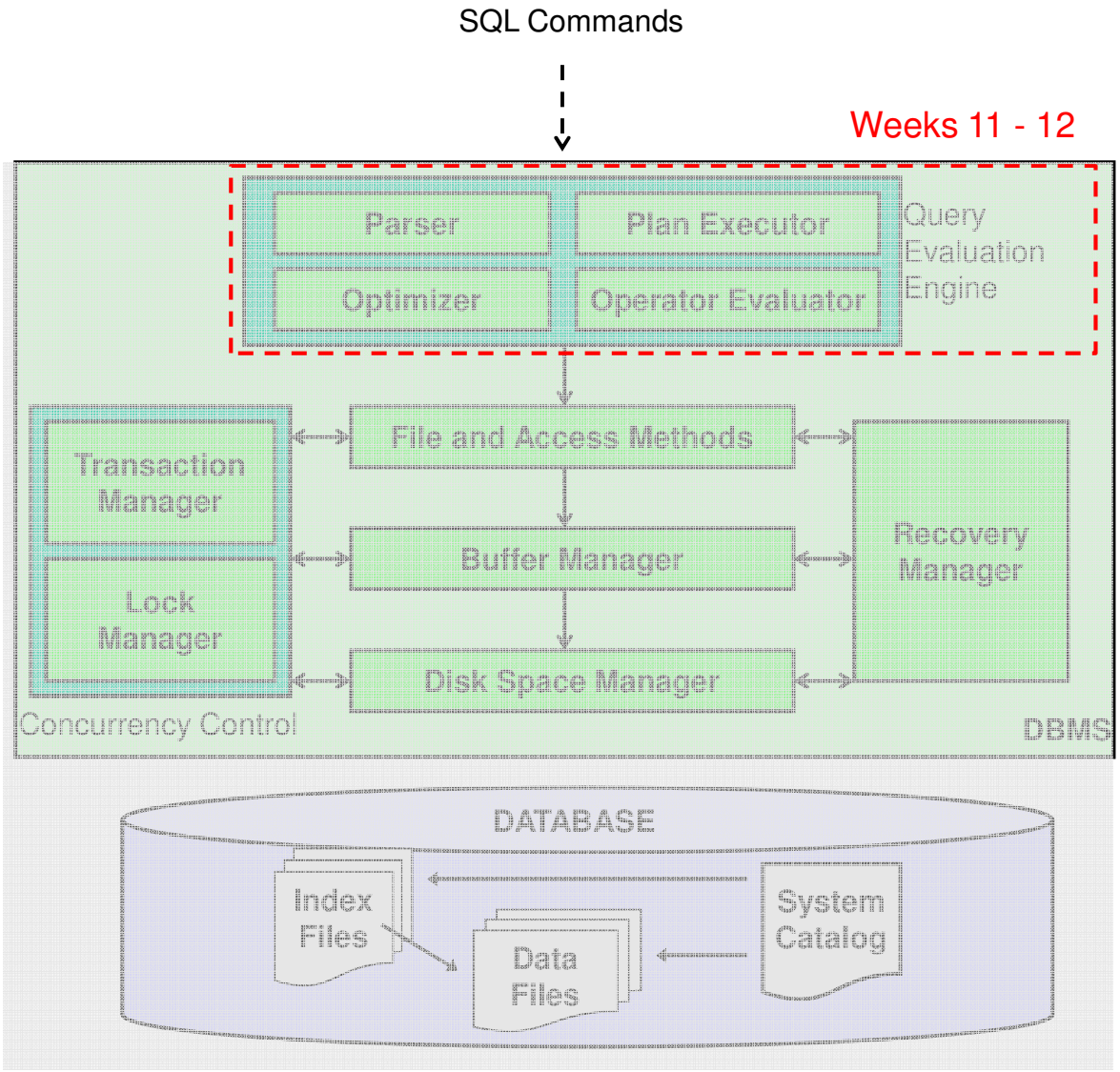
Garcia-Molina/Ullman/Widom – Chapter 15 (start);
Ramakrishnan/Gehrke – Chapters 13 and 14;
Kifer/Bernstein/Lewis – Chapter 10 (start)

Dr Bryn Jeffries
School of Information Technologies

Learning Objectives

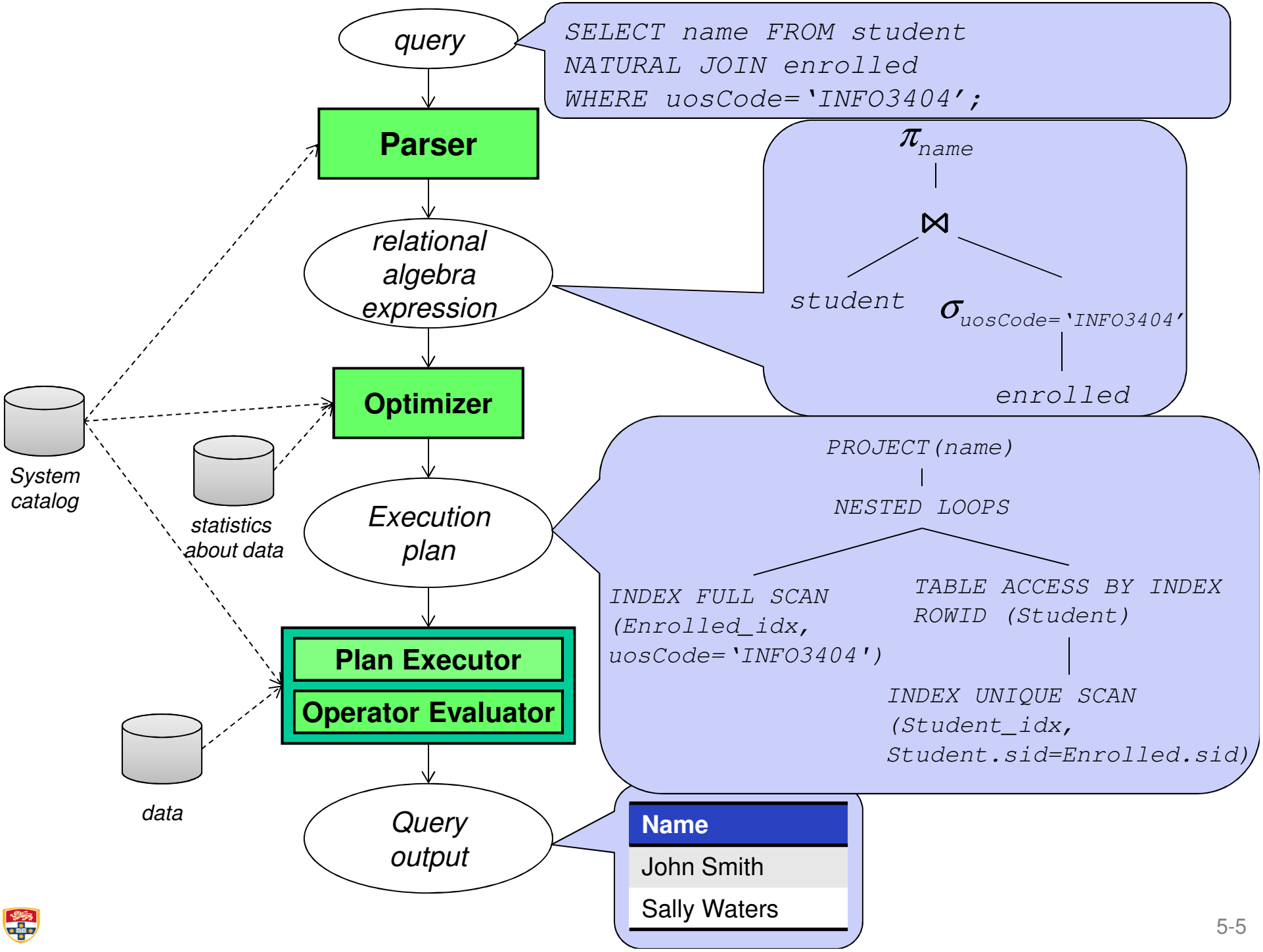
- Understanding of Role and Structure of Query Processing
  - From SQL to physical data access
  - 3 Steps: Query Parsing, Optimization, Execution
  - Expression Tree vs. Evaluation Plan
  - Query Execution Algorithms
- Operator Algorithms
  - External Merge Sort
  - Joins
    - Simple Nested
    - Block Nested
    - Index Nested
    - (mention merge joins)
  - (Pipelining and Materialization)

Based on slides from Ramakrishnan/Gehrke (2003) "Database Management Systems".


SQL Commands
Weeks 11 - 12
Parser
Plan Executor
Optimizer
Operator Evaluator
Query Evaluation Engine
Transaction Manager
File and Access Methods
Buffer Manager
Recovery Manager
Lock Manager
Disk Space Manager
Concurrency Control
DBMS
DATABASE
Index Files
Data Files
System Catalog

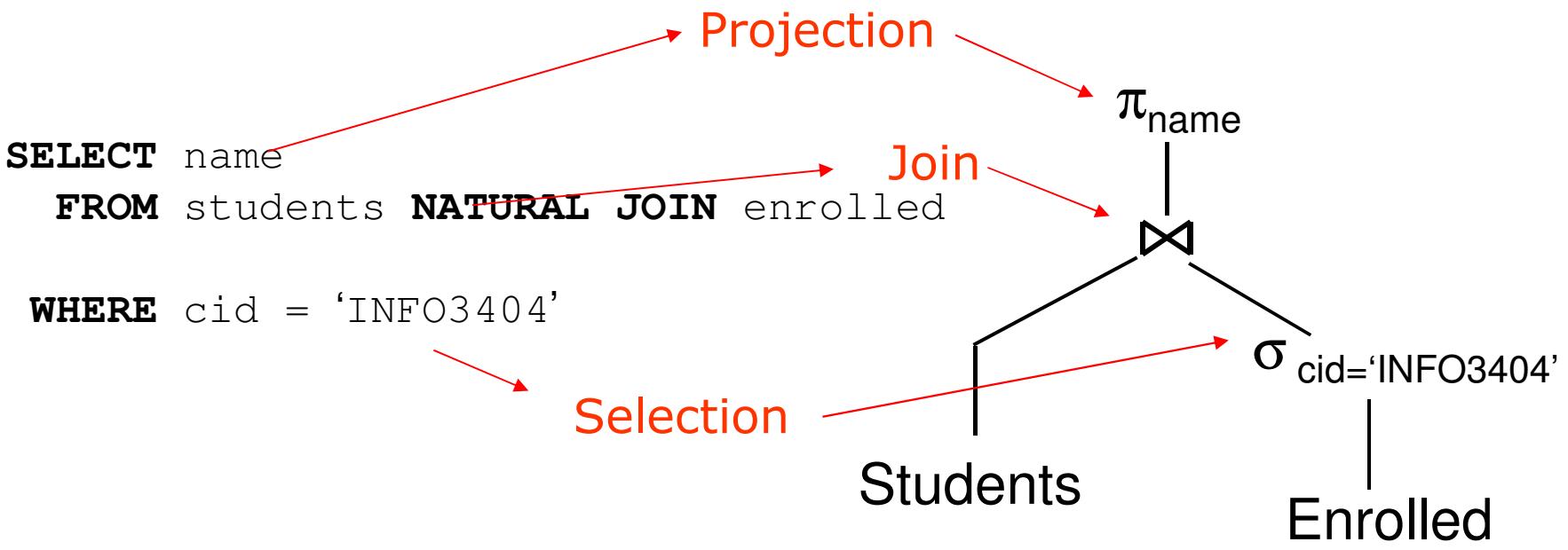
Basic Steps in Query Processing

- Parsing and Translation
  - translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
  - At the end of this part: Query Rewriter
    - Access to views is replaced with actual sub-query
- Query Optimization
  - Amongst all equivalent query-evaluation plans choose the one with lowest cost.
- Query Evaluation
  - Query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.


query
SELECT name FROM student
NATURAL JOIN enrolled
WHERE uosCode='INFO3404';
Parser
relational algebra expression
π name
⋈
student
σ uosCode='INFO3404'
enrolled
Optimizer
System catalog
statistics about data
Execution plan
PROJECT(name)
NESTED LOOPS
INDEX FULL SCAN (Enrolled_idx, uosCode='INFO3404')
TABLE ACCESS BY INDEX ROWID (Student)
INDEX UNIQUE SCAN (Student_idx, Student.sid=Enrolled.sid)
Plan Executor
Operator Evaluator
data
Query output
Name
John Smith
Sally Waters

Parsing and Translation

- SQL gets translated into relational algebra, which can be shown as **expression tree**.
  - Operators have one or more input sets and return one (or more) output set(s).
  - Leafs correspond to (base) relations.
- Example:



RA recap: Projection and Selection

Assessment

sid	uosCode	sem	year	mark
316424328	INFO2120	S1	2012	72
305678453	INFO2120	S1	2012	86
316424328	INFO3005	S1	2010	63
305678453	COMP5138	S1	2012	94

RA:

$\pi_{uosCode,\ sem}(Assessment)$

uosCode	sem
INFO2120	S1
INFO3005	S1
COMP5138	S1

RA:

$\sigma_{uosCode='INFO2120'}(Assessment)$

sid	uosCode	sem	year	mark
316424328	INFO2120	S1	2012	72
305678453	INFO2120	S1	2012	86

Example SQL (projection):

```
SELECT DISTINCT uosCode, sem
FROM Assessment;
```

Example SQL (selection):

```
SELECT *
FROM Assessment
WHERE uosCode='INFO2120';
```

RA recap: Joins

Assessment

sid	uosCode	sem	year	mark
316424328	INFO2120	S1	2012	72
305678453	INFO2120	S1	2012	86
316424328	INFO3005	S1	2010	63
305678453	COMP5138	S1	2012	94

UoSLecturer

uosCode	sem	year	lecturer
INFO2120	S1	2012	Uwe Roehm
INFO3005	S1	2010	Irena Koprinska
COMP5138	S2	2012	Bryn Jeffries

sid	mark	uosCode	sem	year	lecturer
316424328	72	INFO2120	S1	2012	Uwe Roehm
305678453	86	INFO2120	S1	2012	Uwe Roehm
316424328	63	INFO3005	S1	2010	Irena Koprinska

RA:

$Assessment \bowtie UosLecturer$

Example SQL:

```
SELECT *
FROM Assessment A, UoSLecturer L
WHERE A.uosCode=L.uosCode
AND A.sem=L.sem
AND A.year=L.year;
```

OR

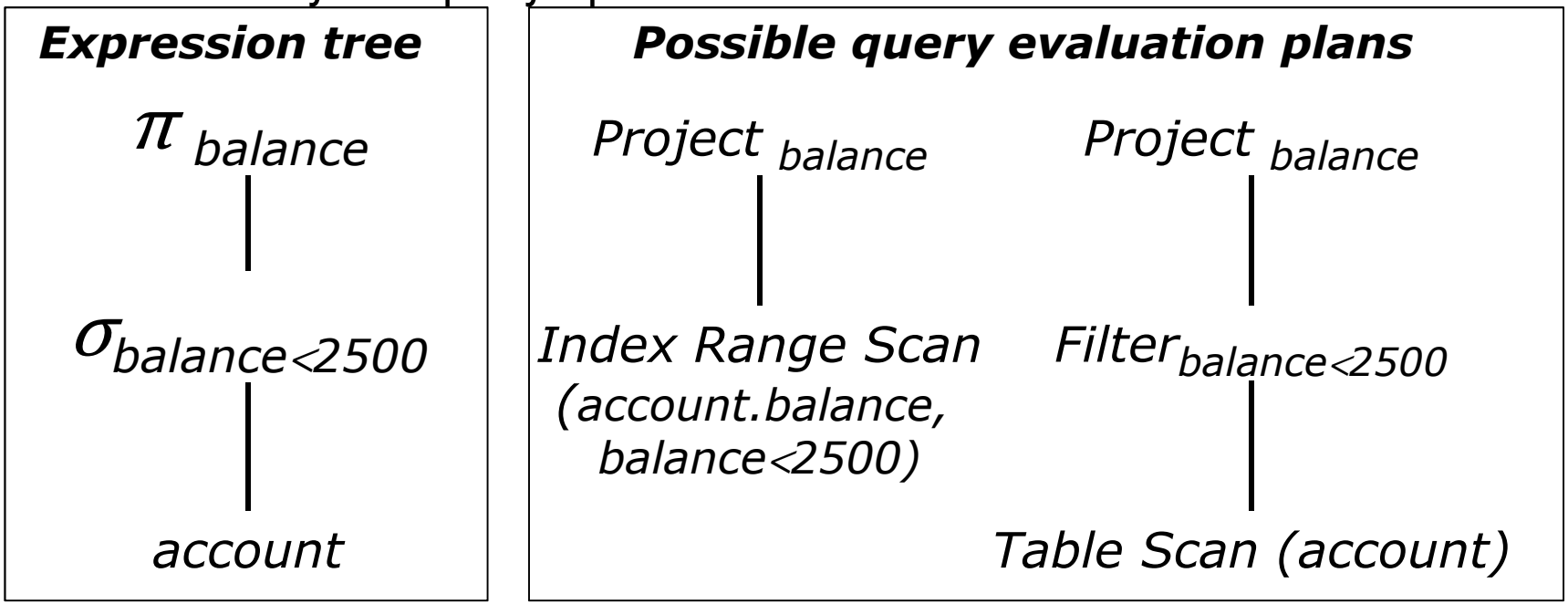
```
SELECT * FROM Assessment A NATURAL JOIN UoSLecturer L;
```

Overview of Query Optimizer

- Read in query expression tree of logical (RA) operations
- Generate *Query Execution Plan*
  - *Tree of relational algebra operators + choice of algorithm for each operator.*
- Aim:
  - Find an optimal plan
  - Avoid bad plans
  - Don't take too long choosing a plan
- Main issues (considered more in later weeks):
  - Multiple equivalent RA expressions
  - Multiple physical operators
  - How to compare plans?
    - System R approach: Lowest estimated I/O

Query Execution Plans

- **Relational algebra operators** are logical operators that represent the intermediate results
- **Physical operators** show how query is evaluated
- An annotated expression tree specifying detailed evaluation strategy with physical operators is called an **evaluation plan** or **query plan**
- Multiple possible query evaluation plans for the expression tree are considered by the query optimizer



Access Paths

- An **access path** is a method of retrieving tuples
  - Can include a search key
- Data structure + Algorithm
  - ***Heap files*** *can be accessed with a* ***file scan***
  - ***Indexes*** can be accessed with an **index scan**
    - Index must match the search key terms
  - Review previous weeks' material
- Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don't match the index:
  - **Most selective access path**: An index or file scan that we estimate will require the fewest page I/Os.

Physical Access Path Operators

- **Table scan** – search algorithms that locate and retrieve records that fulfill a selection condition.
  - Cost estimate (number of disk blocks scanned) = b_R
    - b_R denotes number of blocks containing records from relation R
  - e.g. Oracle: TABLE SCAN, TABLE ACCESS FULL
- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
  - Cost=*lookup*+*number of blocks* containing retrieved records
  - e.g. Oracle: INDEX RANGE SCAN or INDEX UNIQUE SCAN
- **TID access** – direct access via a list of tuple identifiers (tid)
  - E.g. Oracle: TABLE ACCESS BY ROWID

Selections

- Often combined with Access Path by passing search key to access path operator
- Otherwise trivial FILTER operation: only pass on those tuples that match the selection criteria
  - Read cost is input relation size
- Write cost depends upon **reduction factor**:
  - # output tuples/# input tuples

Projection

- Another trivial operation:
  - Just return subset of columns
- The expensive part is removing duplicates.
  - SQL systems don't remove duplicates unless the keyword DISTINCT is specified in a query.

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

- ***Sorting Approach***: Sort on <sid, bid> and remove duplicates. (Can optimize this by dropping unwanted information while sorting.)
- *Hashing Approach*: Hash on <sid, bid> to create **partitions**. Load partitions into memory one at a time, build in-memory hash structure, and eliminate duplicates.

External Sorting

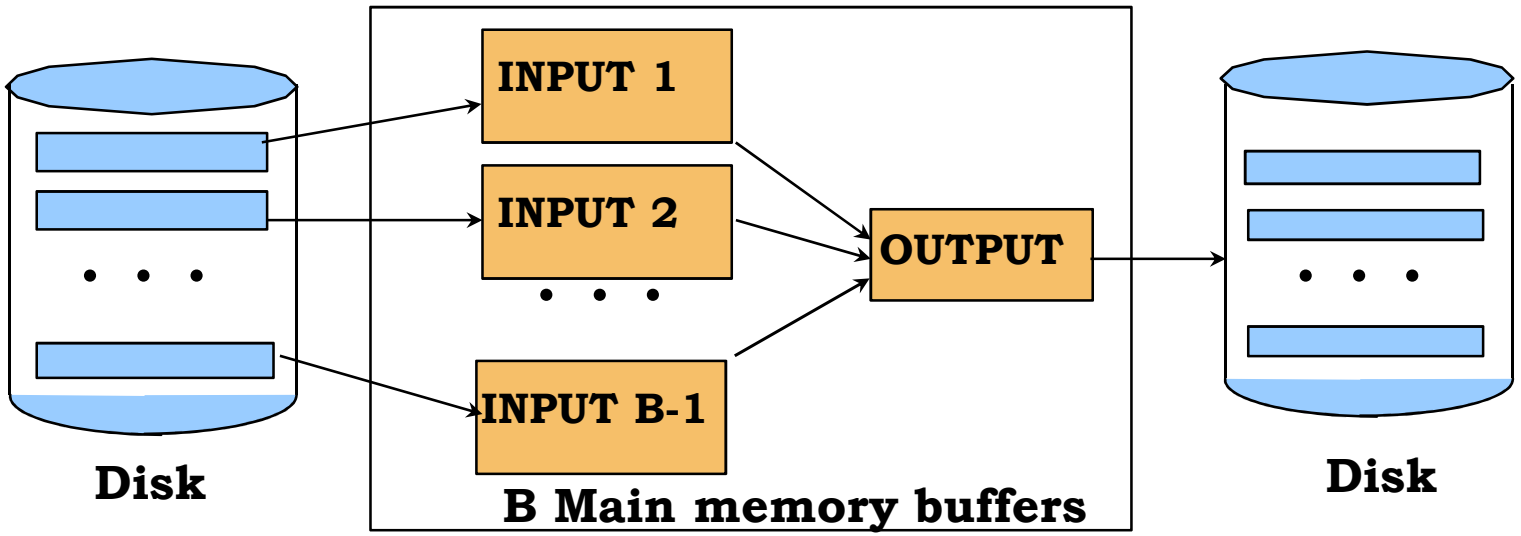
Why Sort?

- A classic problem in computer science!
- Importance of sorting for DBMS:
  - SQL queries can specify that the output be sorted (ORDER BY)
  - Duplicate elimination (why?)
  - SQL can be implemented efficiently if the input are sorted
    - e.g. ***sort-merge-join*** algorithm involves sorting
- For relations that fit in memory, techniques like *QuickSort* can be used.
- Problem: sort 1Gb of data with 1Mb of RAM…

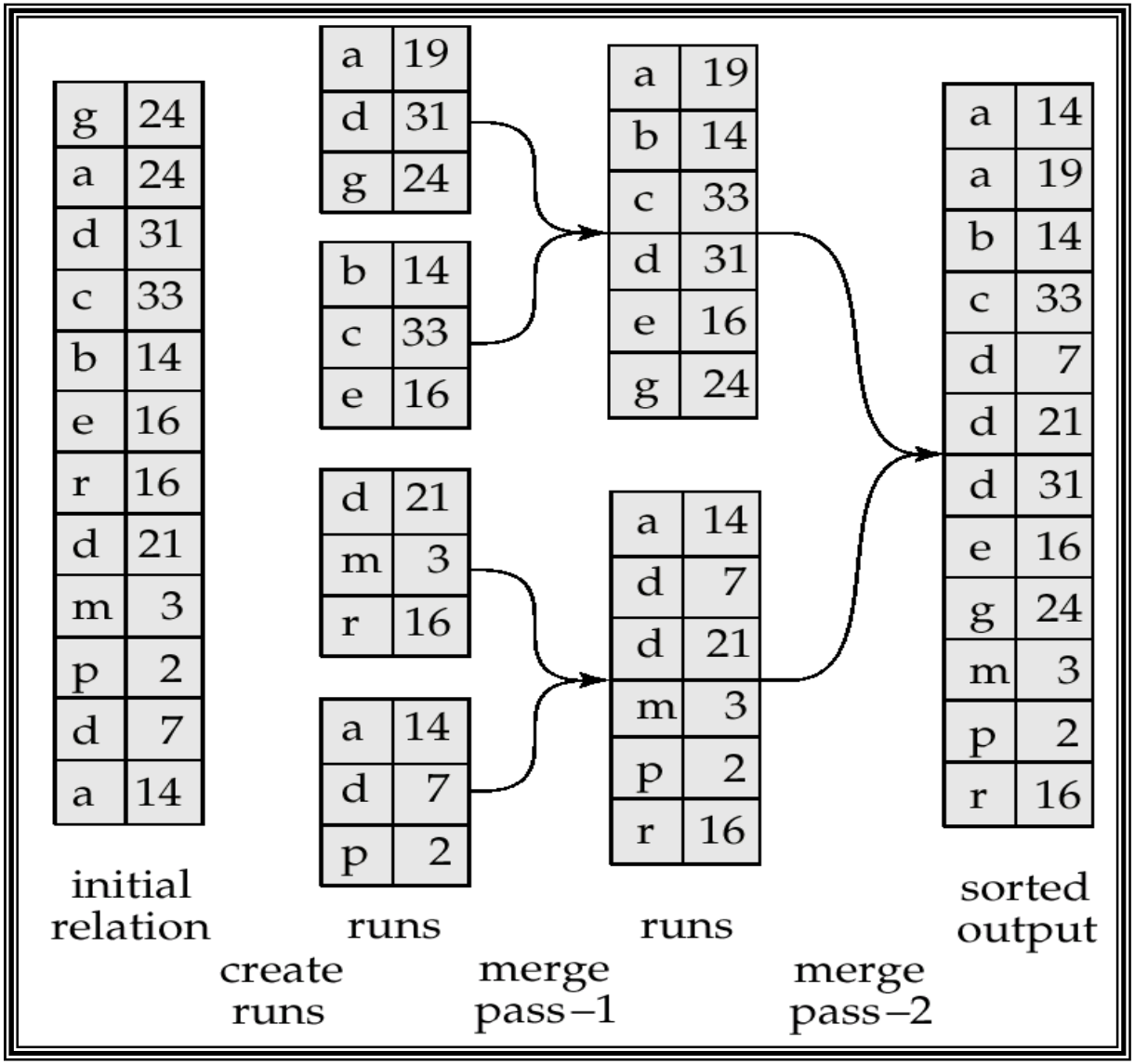
  => **external merge-sort**

General External Merge Sort

- To sort a file with *N* pages using *B* buffer pages:
  - Pass 0: use ***B*** buffer pages.
    Produce $\lceil N / B \rceil$ sorted runs of *B* pages each.
  - Pass 2, …, etc.: merge *B-1* runs.



External Sort-Merge - Example



External Merge Sort Algorithm

Let B denote memory size (in pages).

1. Create sorted runs. Let i be 0 initially.
   Repeatedly do the following till the end of the relation:
   (a) Read B blocks of relation into memory
   (b) Sort the in-memory blocks
   (c) Write sorted data to run R_i; increment i.
   Let the final value of i be m ($\lceil N / B \rceil$)
2. Merge the runs (m-way merge).
   1. Use $B-1$ blocks of memory to buffer input runs, and 1 block to buffer output.
      Read the first block of each run into its buffer page
   2. **repeat**
      1. Select the first record (in sort order) among all buffer pages
      2. Write the record to the output buffer. If output is full, write it to disk.
      3. **If** this is the last records of the input buffer page
         then read the next block (if any) of the run into the buffer.
   3. **until** all input buffer pages are empty:
3. If $m \geq B$, several merge passes are required.
   1. In each pass, contiguous groups of $B - 1$ runs are merged.

Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)

- Example:
  - with 5 buffer pages (B), to sort 108 page file (N):
  - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Scalability of External Merge Sort

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Joins

Reminder: Join types

All joins combine rows from two relations into a new relation

- **Theta join**: Rows from R and S combine if some condition θ is true
  - E.g. θ = R.age>S.minimum age
- **Equi-join**: Rows from R and S combine if specified attributes match in value
  - E.g: R.x=S.y AND R.m = S.n
  - (special case of θ)
- **Natural join**: Rows from R and S combine if all attributes of same name and compatible types match in value
  - E.g R.x=S.x AND R.m = S.m
  - (special case of equi-join)
- (also **Outer join**: Substitute NULL for row data when no match between R and S)

Equality Joins With 1 Join Column

```
SELECT  *
  FROM  Students S, Enrolled E
 WHERE  S.sid=E.sid
```

- In algebra: $\mathbf{R} \bowtie \mathbf{S}$
  Common!
  Must be carefully optimized.
  - $R \times S$ is large; so, $R \times S$ followed by a selection inefficient.

- We will consider more complex join conditions later.

Join Operations

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Sort-Merge join
  - Hash join

- Choice based on cost estimate
  - *Cost metric*: # of I/Os. By convention ignore output costs since they may be pipelined.

Example Data Sizes for Cost Estimates

- The following examples will refer to:
  - $|R|$ tuples in R, stored in b_R blocks (pages)
  - $|S|$ tuples in S, stored in b_S blocks (pages)
    - In our examples, R is Students and S is Enrolled.

- Specific values:
  - Number of tuples of **students ($|R|$)**: 1,000
    enrolled ($|S|$): 10,000
  - Number of blocks of **students (b_R)**: 100
    enrolled (b_S): 400

Simple Nested-Loops Join

- To compute the theta join $R \bowtie_\theta S$

  for each tuple r in R do
  for each tuple s in S do
  if θ(r,s)=true then add <r,s> to the result

- R is called the *outer relation*, S the *inner relation* of the join
- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
- **Pro:** Requires no indexes and can be used with any kind of join condition.
- **Con:** Expensive since it examines every pair of tuples in the two relations.

Cost-Analysis: Nested Loops Join

- In the worst case, if there is memory only to hold one block of each relation, the estimated cost is

$$b_R + |R| * b_S$$

- If the smaller relation fits entirely in memory, use that as the inner relation. Reduces cost to $b_R + b_S$ disk accesses.

- Example:
  - In the worst case scenario:
    students as outer relation: $1000 * 400 + 100 =$ **400,100 disk I/Os**
    enrolled as outer relation: $10000 * 100 + 400 =$ **1,000,400 disk I/Os**
- If smaller relation (students) fits entirely in memory, the cost estimate will be 100+400=**500** disk accesses.

Block Nested-Loops Join

- Variant of nested-loops join in which every block of inner relation is paired with every block of outer relation.
  - For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.

for each block B_R of R do
** for each block B_s of S do**
** for each tuple r in B_R do**
** for each tuple s in B_s do**
** if θ(r,s)=true then output <r,s>**

Cost Analysis: Block Nested-Loops Join

- Worst case estimate: $b_R + b_R * b_S$ block accesses.
  - Each block in the inner relation *S* read once for each block in the outer relation (instead of once for each tuple in the outer relation)
- Best case: $b_R + b_S$ block accesses (whole of inner relation fits in memory)
- Improvements to block nested loop join algorithm:
  - If equi-join attribute forms a key on inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer

Indexed Nested-Loop Join

for each tuple r in R do
for each tuple s in S where $s_j = r_i$ do
add <r,s> to result

- Index lookups can replace file scans if
  - join is an **equi-join or natural join**, and
  - an **index is available** on the inner relation's join attribute
- For each tuple *r* in the outer relation *R*, use the index to look up tuples in *S* that satisfy the join condition with tuple *r*.
- Worst case: buffer has space for only one page of *R*, and, for each tuple in R, we perform an index lookup on *S*.

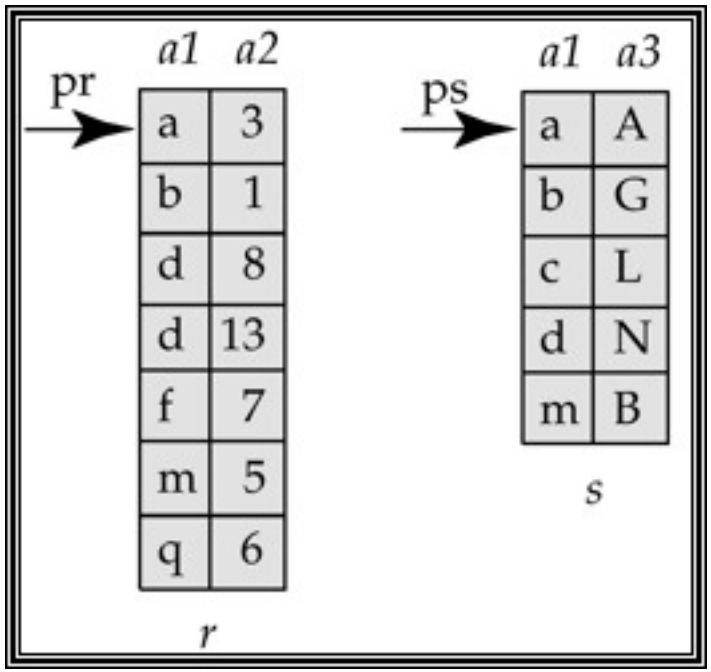
Cost Analysis: Index Nested Loops Join

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: $b_R + (|R| * c$ *(cost of finding matching S tuples))*
  - Where *c* is the cost of traversing index and fetching all matching *S* tuples for one tuple of *R*
  - *c* can be estimated as cost of a single selection on *S* using the join condition.

- If indexes are available on join attributes of both *R* and *S*, use the relation with fewer tuples as the outer relation.

Sort-Merge Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
   - Join step is similar to the merge stage of the sort-merge algorithm.
   - Main difference is handling of duplicate values in join attribute
     - Can construct an index just to compute a join.



Evaluation of Expressions

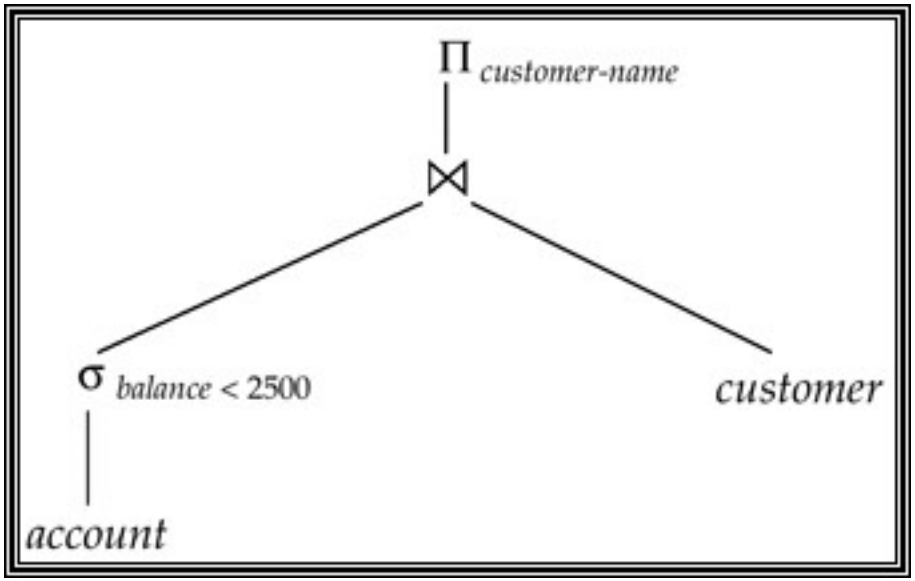
- Alternatives for evaluating an query execution plan:

- **Materialization** (also: **set-at-a-time**):
  simply evaluate one operation at a time. The result of each evaluation is materialized (stored) in a temporary relation for subsequent use.

- **Pipelining** (also: **tuple-at-a-time**):
  evaluate several operations simultaneously in a pipeline

Materialization

- **Materialized evaluation**: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below:
  1. compute and store new relation for $\sigma_{balance<2500}(account)$
  2. Compute and store result of materialized result joined with **customer**
  3. Read back new materizalized result and compute the projections on **customer-name**.



- Materialized evaluation is always applicable
- Costs can be quite high

Pipelining

- **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree
  1. Find tuple matching $\sigma_{balance<2500}(account)$
     1. Join matching tuple with tuples of customer until a new tuple is generated
     2. Project customer name from joined tuple
     3. Repeat for all join results
  2. Repeat for next selection result
- Much cheaper than materialization: no need to store a temporary relation to disk.

Next Week: Query Optimization

- Ramakrishnan & Gehrke: Ch 15, 12
- Kifer, Bernstein, Lewis: Ch 11