Big Data Infrastructures

Philippe Cudré-Mauroux

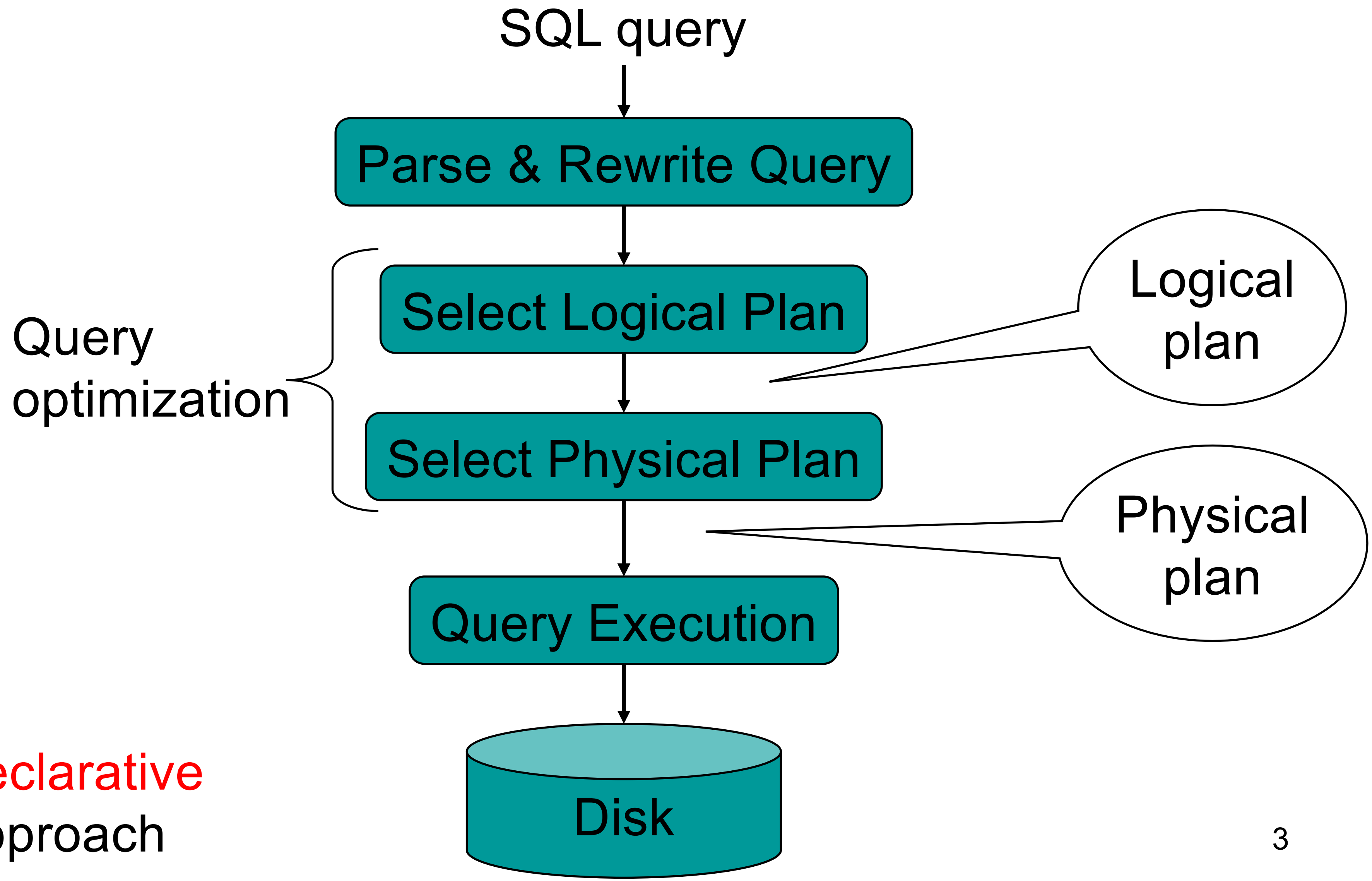
Fall 2018

Lecture 3 – Query Execution & Optimization

Outline

- **Steps involved in processing a query**
  - Logical query plan
  - Physical query plan
  - Query execution overview
- **Operator implementations**
  - One pass algorithms
  - Two-pass algorithms
  - Index-based algorithms
- **Query optimization 101**

Query Evaluation Steps



Declarative
Approach

Example Database Schema

```
Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)
```

View: Suppliers in Seattle

```
CREATE VIEW NearbySupp AS
SELECT sno, sname
FROM Supplier
WHERE scity='Seattle' AND sstate='WA'
```

Example Query

- Find the names of all suppliers in Seattle who supply part number 2

```
SELECT sname FROM NearbySupp
WHERE sno IN ( SELECT sno
               FROM Supplies
               WHERE pno = 2 )
```

Steps in Query Evaluation

- **Step 0: admission control**
  - User connects to the db with username, password
  - User sends query in text format
- **Step 1: Query parsing**
  - Parses query into an internal format
  - Performs various checks using catalog
    - Correctness, authorization, integrity constraints
- **Step 2: Query rewrite**
  - View rewriting, flattening, etc.

Rewritten Version of Our Query

Original query:

```
SELECT sname
FROM NearbySupp
WHERE sno IN ( SELECT sno
               FROM Supplies
               WHERE pno = 2 )
```

Rewritten query:

```
SELECT S.sname
FROM Supplier S, Supplies U
WHERE S.scity='Seattle' AND S.sstate='WA'
AND S.sno = U.sno
AND U.pno = 2;
```

Continue with Query Evaluation

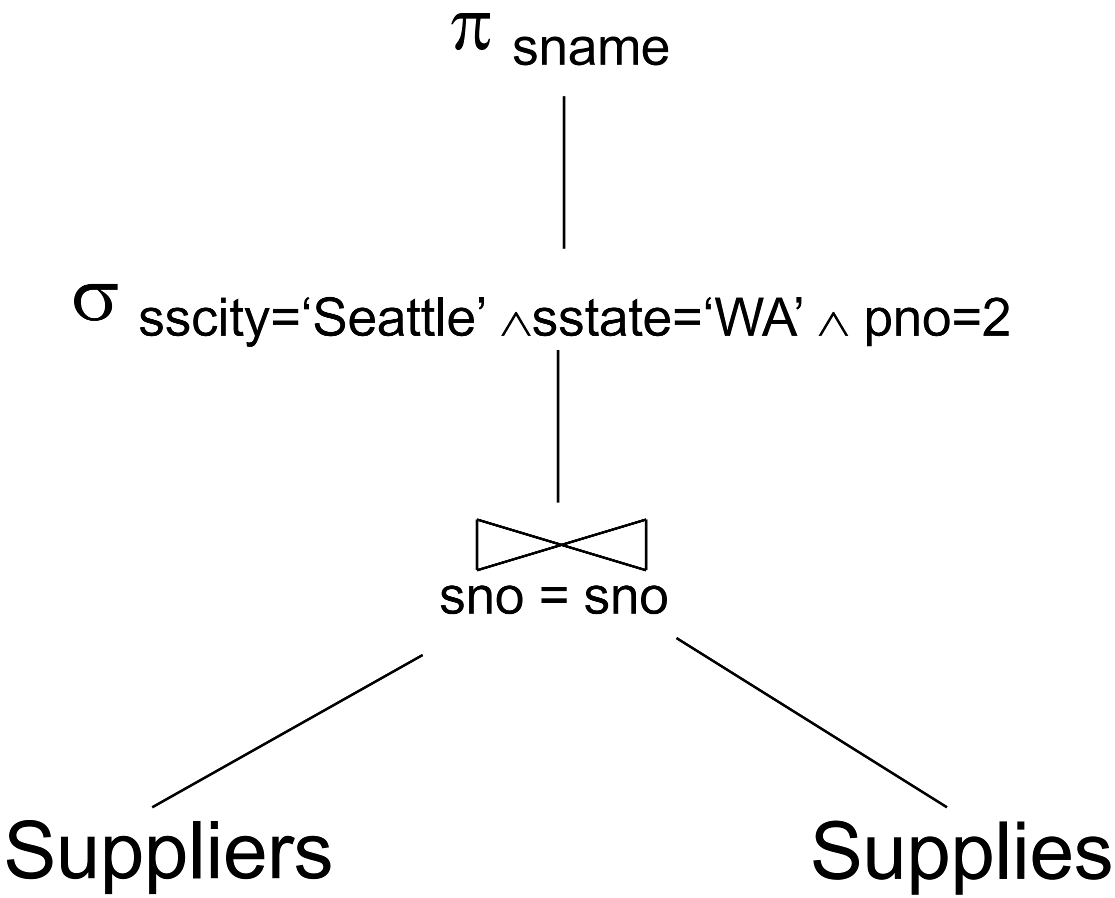
- **Step 3: Query optimization**
  - Find an efficient query plan for executing the query
  - We will spend some time on this topic later

- A **query plan** is
  - **Logical query plan**: an extended relational algebra tree
  - **Physical query plan**: with additional annotations at each node
    - Access method to use for each relation
    - Implementation to use for each relational operator

Extended Algebra Operators

- Union $\cup$, intersection $\cap$, difference $-$
- Selection σ
- Projection π
- Join $\bowtie$
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ
- Rename ρ

Logical Query Plan

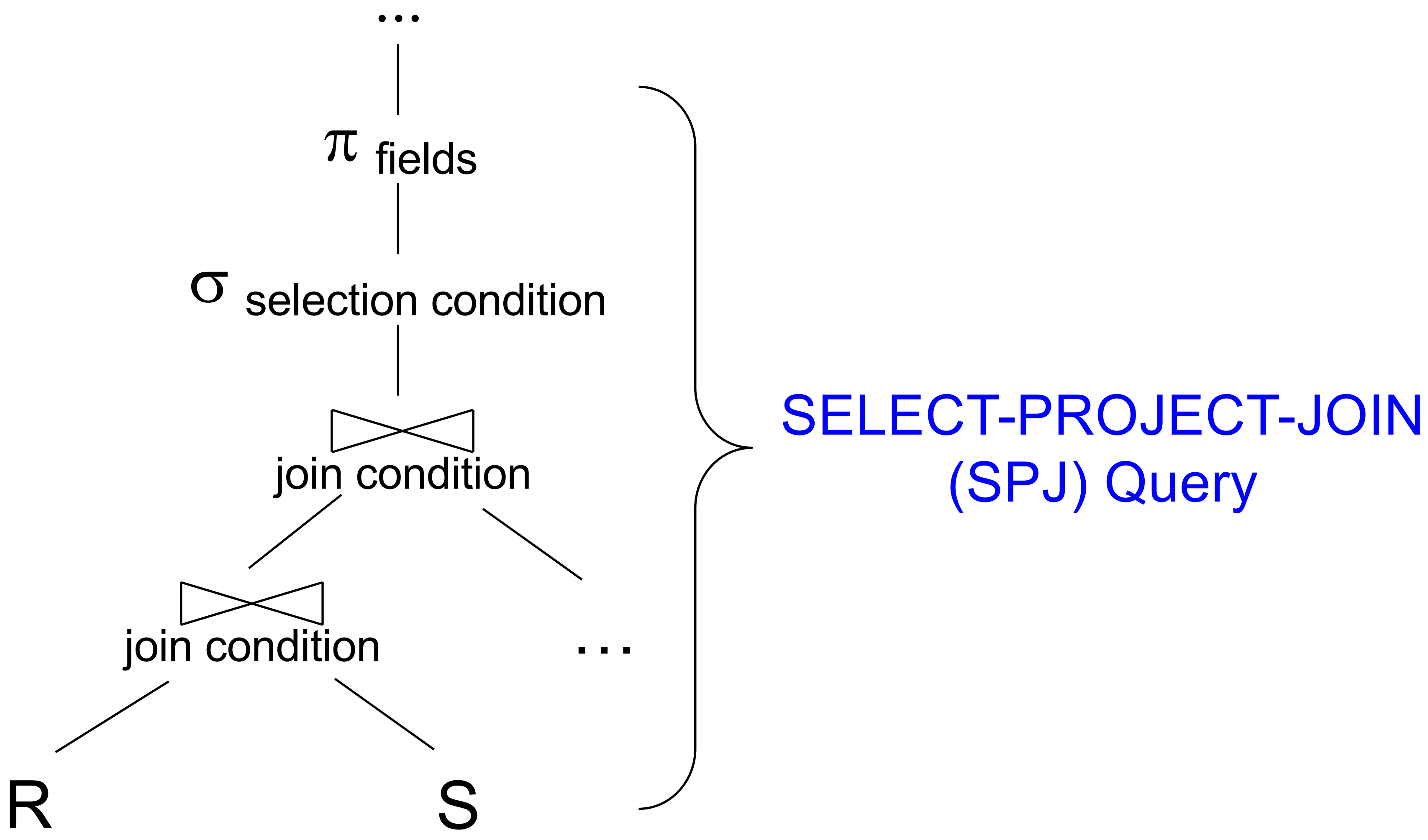


Query Block

- Most optimizers operate on individual query blocks

- A query block is a SQL query with **no nesting**
  - **Exactly one**
    - SELECT clause
    - FROM clause
  - **At most one**
    - WHERE clause
    - GROUP BY clause
    - HAVING clause (behaves like a WHERE for aggregates)

Typical Plan for Block (1/2)



Typical Plan For Block (2/2)

$having_{condition}$

$\gamma_{\text{fields, sum/count/min/max(fields)}}$

π_{fields}

$\sigma_{\text{selection condition}}$

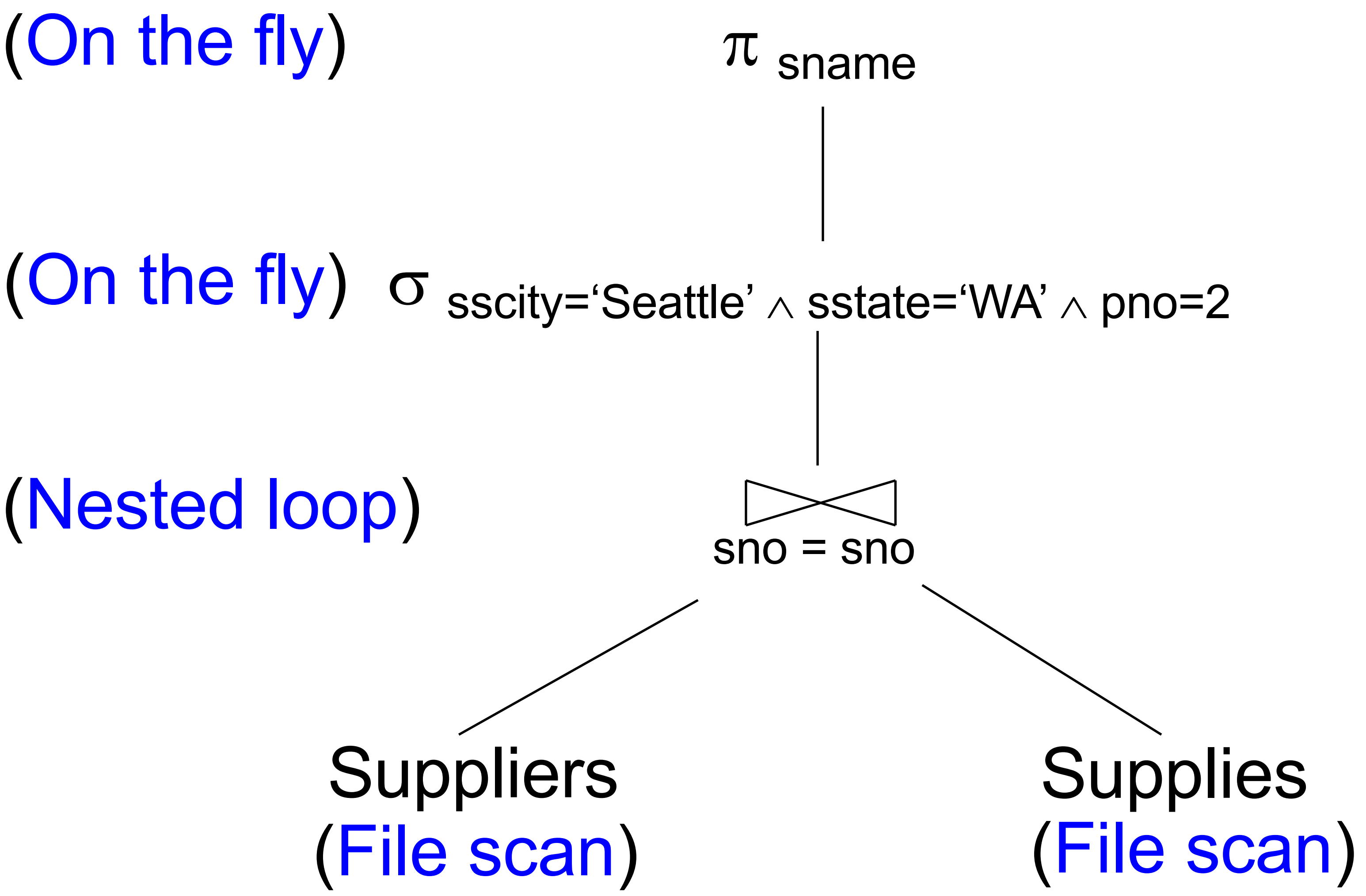
$\bowtie_{\text{join condition}}$

…   …

Physical Query Plan

- Logical query plan with extra annotations
- **Access path selection** for each relation
  - Use a file scan or use an index
- **Implementation choice** for each operator
- **Scheduling decisions** for operators

Physical Query Plan



Final Step in Query Processing

- **Step 4: Query execution**
  - How to synchronize operators?
  - How to pass data between operators?

- Standard approaches:
  - Iterator interface
  - Pipelined execution
  - Intermediate result materialization

Iterator Interface

- Each **operator implements this interface**
- Interface has only three methods
- **open()**
  - Initializes operator state
  - Sets parameters such as selection condition
- **get_next()**
  - Operator invokes get_next() iteratively on its inputs
  - Performs processing and produces an output tuple
- **close()**: clean-up state

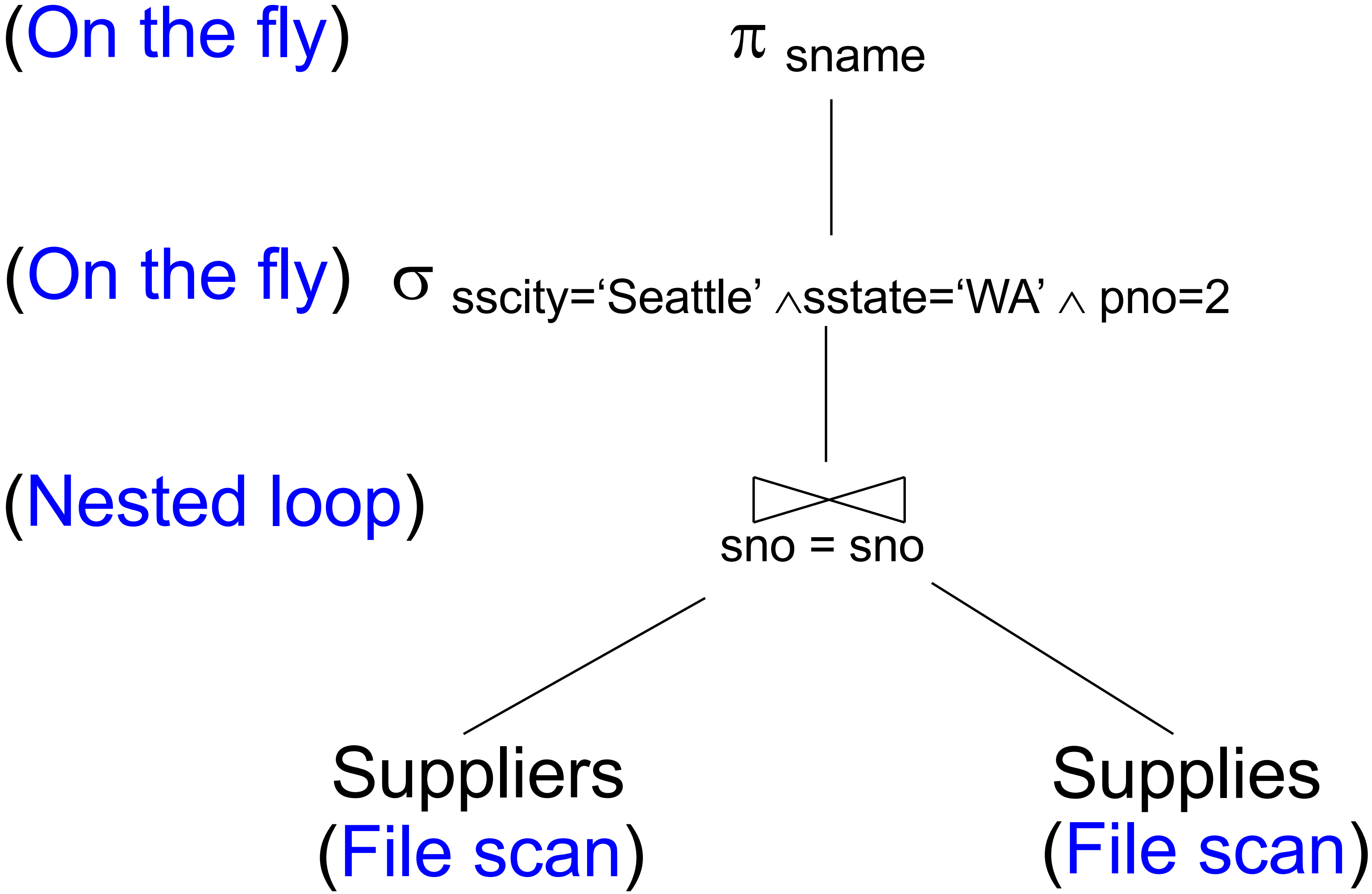
Pipelined Execution

- Applies parent operator to tuples directly as they are produced by child operators

- Benefits
  - No operator synchronization issues
  - Saves cost of writing intermediate data to disk
  - Saves cost of reading intermediate data from disk
  - Good resource utilizations on single processor

- This approach is used whenever possible

Pipelined Execution

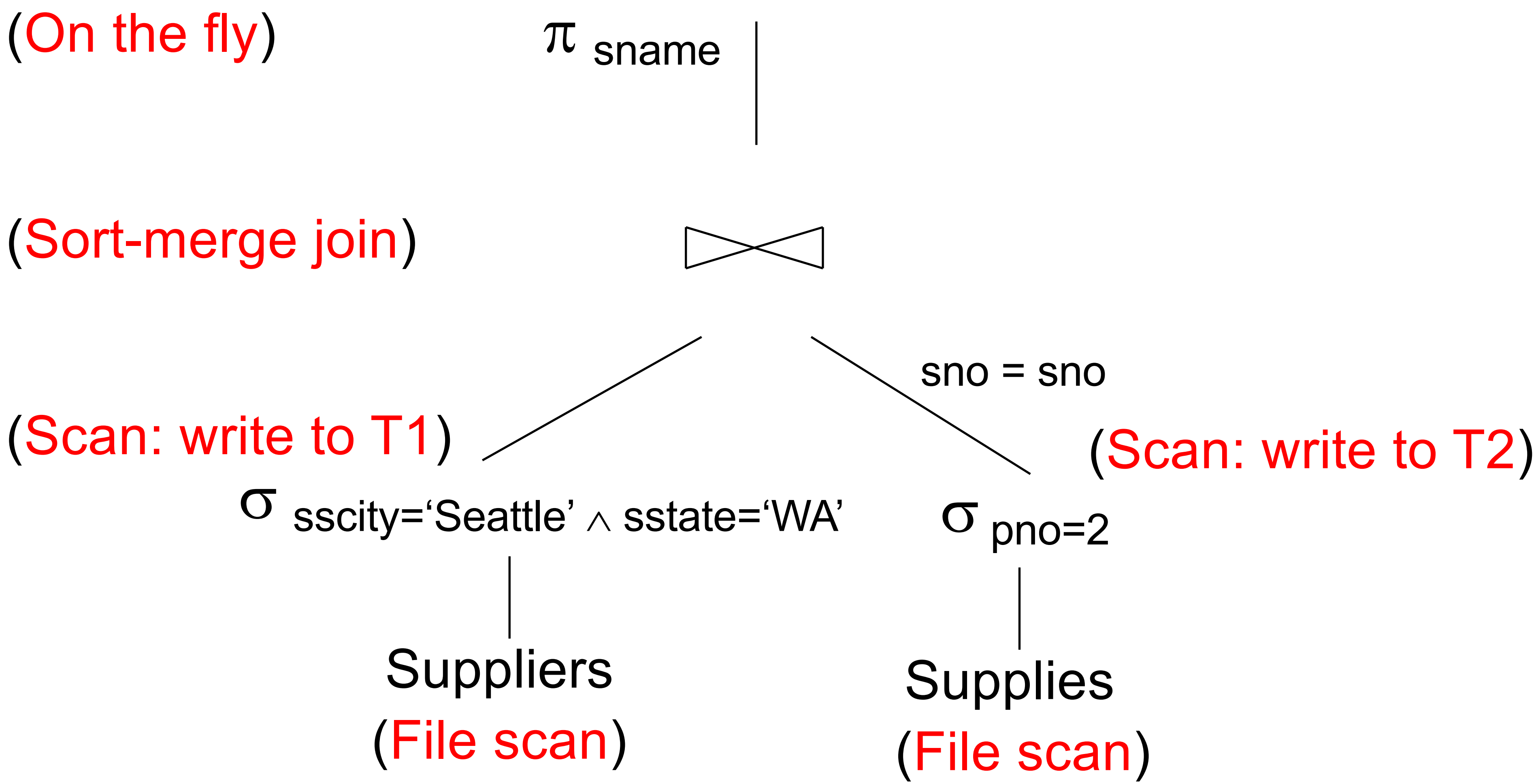


Intermediate Tuple Materialization

- Writes the results of an operator to an intermediate table on disk

- No direct benefit but necessary for some operator implementations
  - E.g., When operator needs to examine the same tuples multiple times

Intermediate Tuple Materialization



Intermediate Results

- The change in available computing resources, e.g. amount of RAM, multiple cores per CPU, may yield to additional changes in result handling
- To achieve best locality of reference, non-pipelined plans can be favored
  - Intermediate results are written only to main memory and not to disk
- Applicability depends on the size of the estimated intermediate result and available buffer size

Outline

- **Steps involved in processing a query**
  - Logical query plan
  - Physical query plan
  - Query execution overview

- **Operator implementations**
  - One pass algorithms
  - Two-pass algorithms
  - Index-based algorithms

- **Query optimization 101**

Why Learn About Op Algos?

- Implemented in commercial DBMSs an Big Data frameworks
- Different DBMSs implement different subsets of these algorithms
- Good algorithms can greatly improve performance
- Need to know about physical operators to understand query optimization

Cost Parameters

- In database systems the data is on disk
- **Cost = total number of I/Os**
  - **More complex models possible (which ones?)**
  - **Different models for main-memory / distributed DBMSs (which ones?)**

- Parameters:
  - **B(R) = # of blocks (i.e., pages) for relation R**
  - **T(R) = # of tuples in relation R**
  - **V(R, a) = # of distinct values of attribute a**

Cost

- Cost of an operation = number of disk I/Os to
  - read the operands
  - compute the result
- Cost of writing the result to disk is *not included*
  - Need to count it separately when applicable

Notions of Clustering

- **Co-clustering**
  - Tuples of one relation R are placed with a tuple of another relation S with a common value (*uncommon*)

- **Clustered relation**
  - Tuples of relation are stored on blocks predominantly devoted to storing *that* relation
  - Sometimes also called “clustered file organization”
  - We will focus on this in the following

- **Clustered index (aka clustering index)**
  - When ordering of data records is close to the ordering of data entries in the index

Cost Parameters

- Clustered relation R:
  - Blocks consists mostly of records from this table
  - $B(R) \approx T(R)$ / blockSize

- Unclustered relation R:
  - Its records are placed on blocks with other tables
  - When R is unclustered: $B(R) \approx T(R)$

- When a is a key, $V(R,a) = T(R)$
- When a is not a key, $V(R,a)$

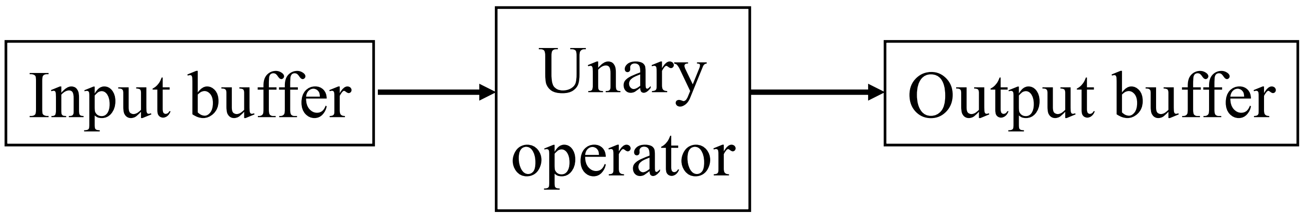
Cost of Scanning a Table

- Clustered relation:
  - Result may be unsorted: B(R)
  - Result needs to be sorted: 3B(R) (see later)

One-pass Algorithms

Selection $\sigma(R)$, projection $\Pi(R)$

- Both are ***tuple-at-a-time*** algorithms
- Cost: B(R), the cost of scanning the relation



Join Algorithms

- Logical operator:
  - Product(pname, cname) $\bowtie$ Company(cname, city)

- Propose three physical operators for the join, assuming the tables are in main memory:
  - **Hash join**
  - **Nested loop join**
  - **Sort-merge join**

Hash Join

Hash join: R $\bowtie$ S

- Scan R, build buckets in main memory
- Then scan S and join

- Cost: B(R) + B(S)

- One pass algorithm when B(R) <= M

Nested Loop Joins

- Tuple-based nested loop $R \bowtie S$
- R is the outer relation, S is the inner relation

```
for each tuple r in R do
   for each tuple s in S do
      if r and s join then output (r,s)
```

- Cost: B(R) + T(R) B(S) when S is clustered

Page-at-a-time Refinement

```
for each page of tuples r in R do
  for each page of tuples s in S do
      for all pairs of tuples
          if r and s join then output (r,s)
```

- Cost: B(R) + B(R)B(S) if S is clustered

Nested Loop Joins

- We can be cleverer

- How would you compute the join in the following cases ? What is the cost ?

  – B(R) = 1000, B(S) = 2, M = 4

  – B(R) = 1000, B(S) = 3, M = 4

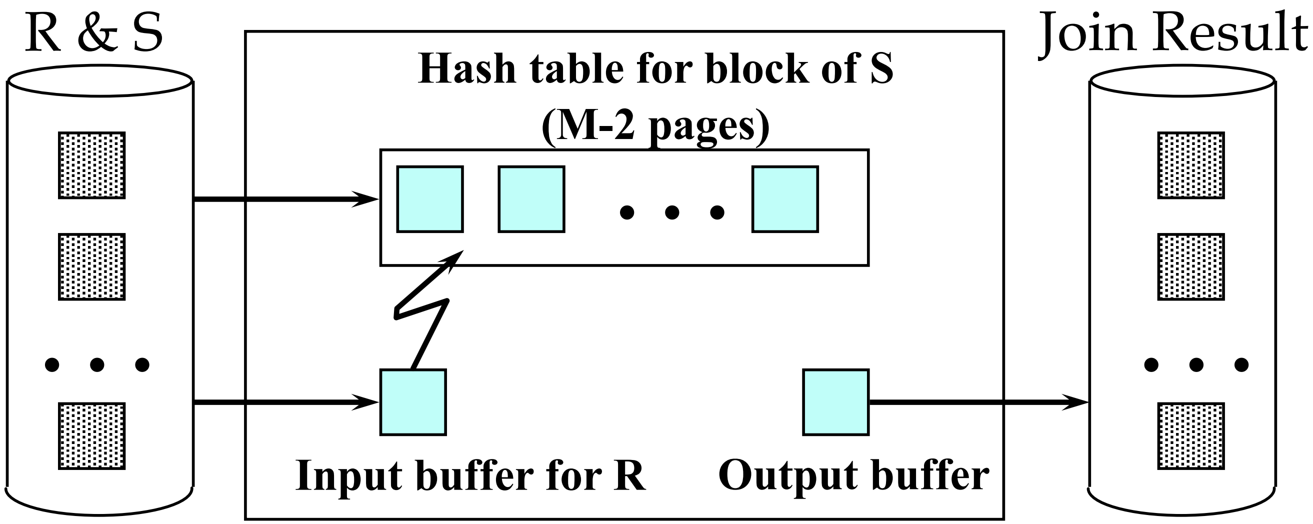
  – B(R) = 1000, B(S) = 6, M = 4

Block Nested Loop Joins

- Block Nested Loop Join
- Group of (M-2) pages of S is called a “block”

```
for each (M-2) pages ps of S do
  for each page pr of R do
      for each tuple s in ps
          for each tuple r in pr do
              if “r and s join” then output(r,s)
```

Block Nested Loop Joins



Block Nested Loop Joins

- Cost of block-based nested loop join
  - Read S once: cost B(S)
  - Outer loop runs B(S)/(M-2) times, and each time need to read R: costs B(S)B(R)/(M-2)
  - Total cost: B(S) + B(S)B(R)/(M-2)

- Notice: it is better to iterate over the smaller relation first

Sort-Merge Join

Sort-merge join: $R \bowtie S$

- Scan R and sort in main memory
- Scan S and sort in main memory
- Merge R and S

- Cost: B(R) + B(S)
- One pass algorithm when B(S) + B(R) <= M
- Typically, this is NOT a one pass algorithm

More One-pass Algorithms

Duplicate elimination $\delta(R)$

- Need to keep tuples in memory
- When new tuple arrives, need to compare it with previously seen tuples
- Balanced search tree or hash table
- Cost: $B(R)$
- Assumption: $B(\delta(R)) <= M$

Even More One-pass Algorithms

Grouping:

$\gamma_{\text{department, sum(quantity)}}$ (Product)

How can we compute this in main memory ?

Even More One-pass Algorithms

- Grouping: $\gamma_{\text{department, sum(quantity)}}(R)$

- Need to store all departments in memory
- Also store the sum(quantity) for each department
- Balanced search tree or hash table
- Cost: B(R)
- Assumption: number of depts fits in memory

Outline

- **Steps involved in processing a query**
  - Logical query plan
  - Physical query plan
  - Query execution overview
- **Operator implementations**
  - One pass algorithms
  - Two-pass algorithms
  - Index-based algorithms
- **Query optimization 101**

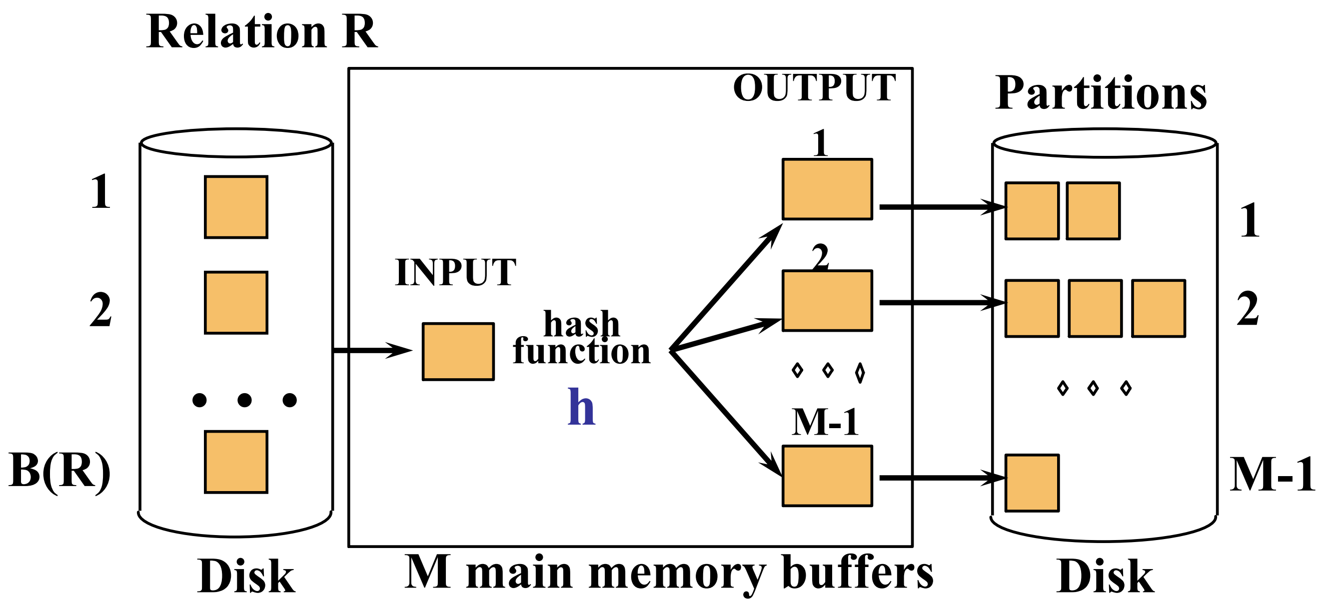
Two-Pass Algorithms

- What if data does not fit in memory?
- Need to process it in multiple passes

- Two key techniques
  - Hashing
  - Sorting

Two Pass Algorithms Based on Hashing

- Idea: partition a relation R into partitions (buckets), on disk
- Each bucket has size approx. B(R)/M



- Does each partition fit in main memory ?
  - Yes if B(R)/M <= M, i.e. $B(R) <= M^2$

Hash Based Algorithms for δ

- Recall: $\delta(R) =$ duplicate elimination

- Step 1. Partition R into buckets
- Step 2. Apply δ to each bucket

- Cost: 3B(R)
- Assumption: B(R) <= M^2

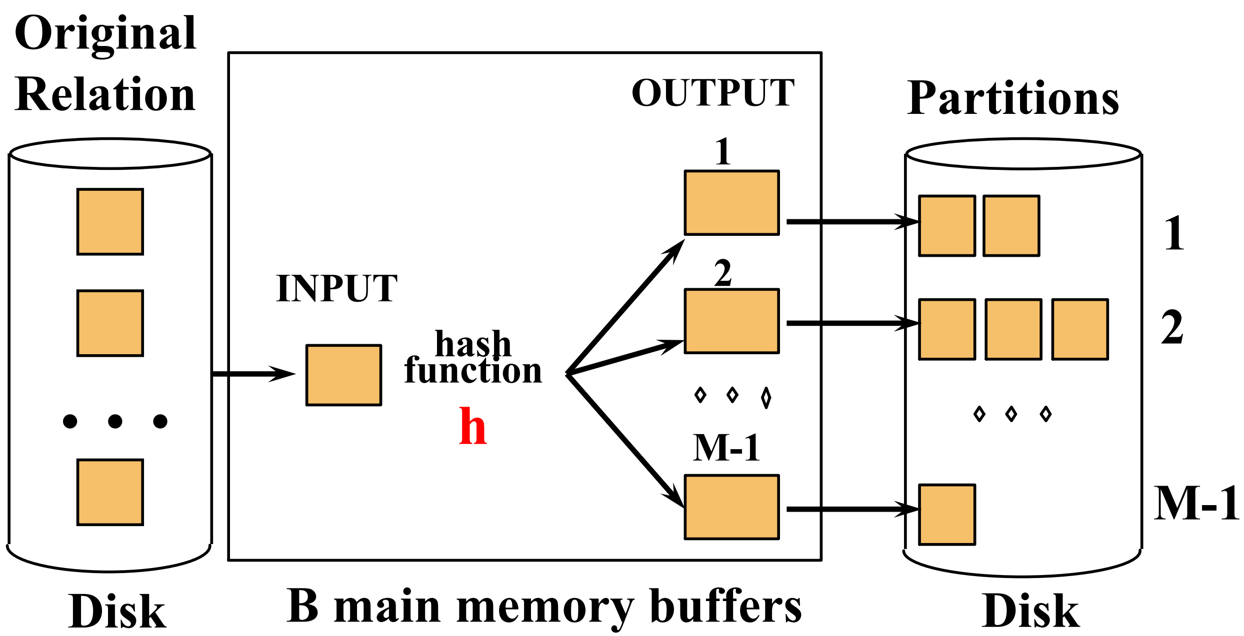
Partitioned (Grace) Hash Join

$R \bowtie S$

- Step 1:
  - Hash S into M-1 buckets
  - Send all buckets to disk
- Step 2
  - Hash R into M-1 buckets
  - Send all buckets to disk
- Step 3
  - Join every pair of buckets

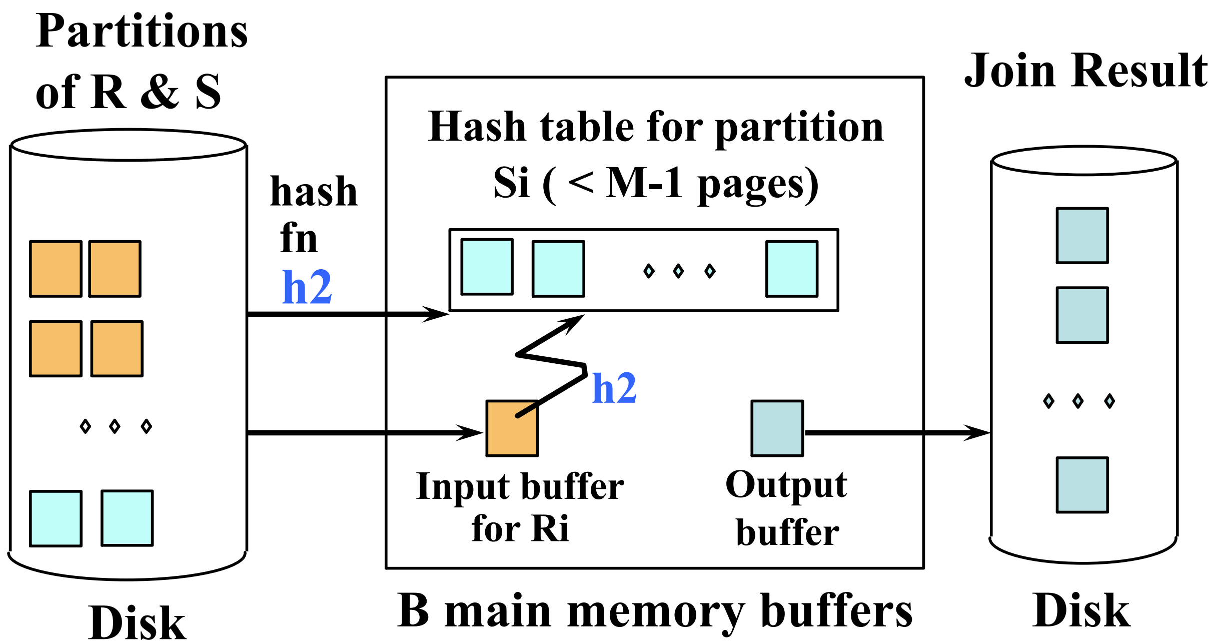
Partitioned Hash Join

- Partition both relations using hash fn **h**
- R tuples in partition i will only match S tuples in partition i.



Partitioned Hash Join

- Read in partition of R, hash it using h2 (≠ h)
  - Build phase
- Scan matching partition of S, search for matches
  - Probe phase



Partitioned Hash Join

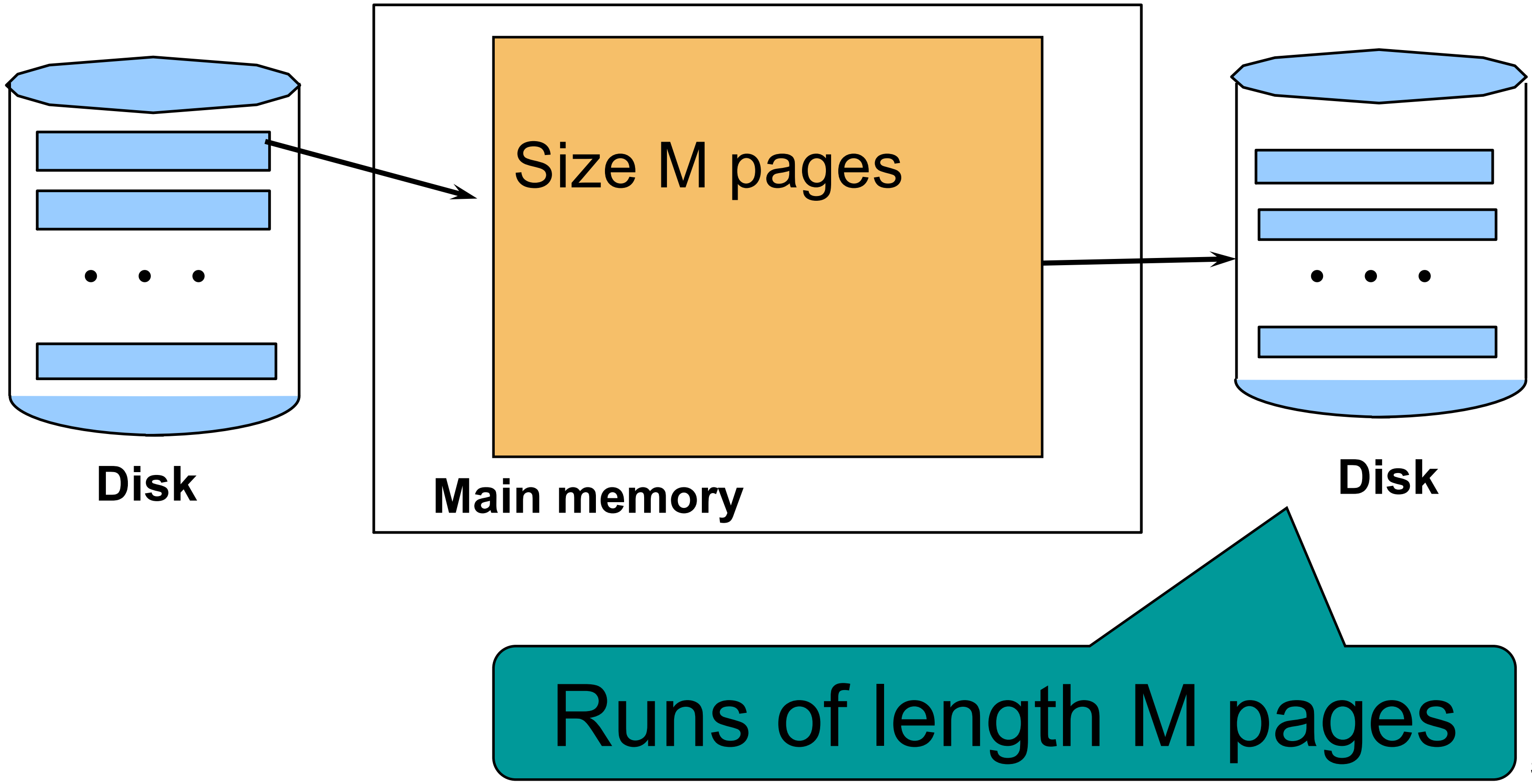
- Cost: 3B(R) + 3B(S)
- Assumption: min(B(R), B(S)) <= M^2

External Sorting

- Problem: Sort a file of size B with memory M
- Where we need this:
  - ORDER BY in SQL queries
  - Several physical operators
  - Bulk loading of B+-tree indexes.
- Will discuss only 2-pass sorting, for when $B < M^2$

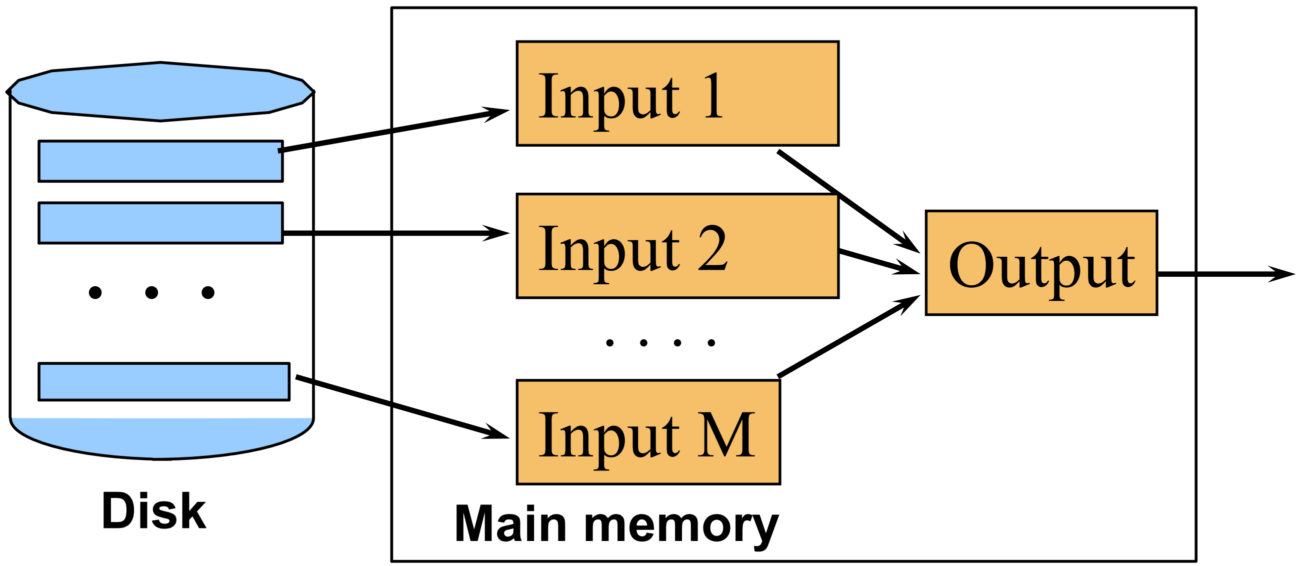
External Merge-Sort: Step 1

- Phase one: load M pages in memory, sort



External Merge-Sort: Step 2

- Merge M – 1 (max, due to memory) runs into a new run



If $B(R) < M^2$ then we have M-1 runs max and we are done (slightly more complex otherwise)

External Merge-Sort

- Cost:
  - Read+write+read = 3B(R)
  - Assumption: B(R) <= M^2

- Other considerations
  - In general, a lot of optimizations are possible

Two-Pass Algorithms Based on Sorting

Duplicate elimination $\delta(R)$

- Trivial idea: sort first, then eliminate duplicates
- Step 1: sort chunks of size M, write
  - cost 2B(R)
- Step 2: merge M-1 runs, but include each tuple only once
  - cost B(R)
- Total cost: 3B(R), Assumption: $B(R) <= M^2$

Two-Pass Algorithms Based on Sorting

Grouping: $\gamma_{a,\ sum(b)}$ (R)

- Same as before: sort, then compute the sum(b) for each group of a's
- Total cost: 3B(R)
- Assumption: B(R) <= M^2

Two-Pass Algorithms Based on Sorting

Join R $\bowtie$ S

- Start by sorting both R and S on the join attribute:
  - Cost: 4B(R)+4B(S) (because need to write to disk)
- Read both relations in sorted order, match tuples
  - Cost: B(R)+B(S)
- Total cost: 5B(R)+5B(S)
- Assumption: $B(R) <= M^2$, $B(S) <= M^2$

Two-Pass Algorithms Based on Sorting

Join $R \bowtie S$

- If the number of tuples in R matching those in S is small (or vice versa) (e.g., If $B(R) + B(S) <= M^2$) we can compute the join during the merge phase

- Total cost: $3B(R)+3B(S)$

Outline

- **Steps involved in processing a query**
  - Logical query plan
  - Physical query plan
  - Query execution overview
- **Operator implementations**
  - One pass algorithms
  - Two-pass algorithms
  - Index-based algorithms
- **Query optimization 101**

Review: Access Methods

- **Heap file**
  - Scan tuples one at the time
- **Hash-based index**
  - Efficient selection on equality predicates
- **Tree-based index**
  - Efficient selection on equality or range predicates

Index Based Selection

- Selection on equality: $\sigma_{a=v}(R)$
- V(R, a) = # of distinct values of attribute a
- Clustered index on a: cost B(R)/V(R,a)
- Unclustered index on a: cost T(R)/V(R,a)
- Note: we ignored the I/O cost for the index pages

Index Based Selection

- Example:

$B(R) = 2000$
$T(R) = 100{,}000$
$V(R, a) = 20$

cost of $\sigma_{a=v}(R) = ?$

- Table scan (assuming R is clustered)
  - B(R) = 2,000 I/Os
- Index based selection
  - If index is clustered: B(R)/V(R,a) = 100 I/Os
  - If index is unclustered: T(R)/V(R,a) = 5,000 I/Os
- Lesson
  - Don't build unclustered indexes when V(R,a) is small !

Index Nested Loop Join

$R \bowtie S$

- Assume S has an index on the join attribute
- Iterate over R, for each tuple fetch corresponding tuple(s) from S

- Cost:
  - Assuming R is clustered
  - If index on S is clustered: $B(R) + T(R)B(S)/V(S,a)$
  - If index on S is unclustered: $B(R) + T(R)T(S)/V(S,a)$

Summary of External Join Algorithms

- Block Nested Loop Join: B(R) + B(R)*B(S)/M
- Sort-Merge Join: 3B(R)+3B(S)
  Assuming B(R)+B(S) <= M^2
- Index Nested Loop Join: B(R) + T(R)B(S)/V(S,a)
  Assuming R is clustered and S has clustered index on a

Summary of Query Execution

- For each logical query plan
  - There exist many physical query plans
  - Each plan has a different cost
  - Cost depends on the data
- Additionally, for each query
  - There exist several logical plans

Outline

- **Steps involved in processing a query**
  - Logical query plan
  - Physical query plan
  - Query execution overview

- **Operator implementations**
  - One pass algorithms
  - Two-pass algorithms
  - Index-based algorithms

- **Query optimization 101**

Query Optimization Algorithm

- For a query
  - There exists many physical query plans
  - Query optimizer needs to pick a good one

- Basic query optimization algorithm
  - Enumerate alternative plans
  - Compute estimated cost of each plan
    - Compute number of I/Os
    - Optionally take into account other resources
  - Choose plan with lowest cost
  - This is called cost-based optimization

Estimating Cost of a Query Plan

- We already know how to
  - Compute the cost of different operations in terms of number IOs

- We still need to
  - Compute cost of retrieving tuples from disk with different access paths (for more sophisticated predicates than equality)
  - Compute cost of a complete plan

Access Path Selection

- Supplier(sid,sname,scity,sstate)
- Selection condition: sid > 300 $\wedge$ scity='Seattle'
- Indexes: B+-tree on sid and B+-tree on scity
- Which access path should we use?
- We should pick the **most selective** access path

Access Path Selectivity

- **Access path selectivity is the number of pages retrieved if we use this access path**
  - Most selective retrieves fewest pages

- As we saw earlier, **for equality predicates**
  - Selection on equality: $\sigma_{a=v}(R)$
  - V(R, a) = # of distinct values of attribute a
  - 1/V(R,a) is thus the reduction factor
  - Clustered index on a: cost B(R)/V(R,a)
  - Unclustered index on a: cost T(R)/V(R,a)
  - (we are ignoring I/O cost of index pages for simplicity)

Selectivity for Range Predicates

Selection on range: $\sigma_{a>v}(R)$

- How to compute the selectivity?
- Assume values are uniformly distributed
- Reduction factor X
- X = (Max(R,a) - v) / (Max(R,a) - Min(R,a))

- Clustered index on a: cost B(R)*X
- Unclustered index on a: cost T(R)*X

Example

- Selection condition: **sid > 300**
  - Index I1: B+-tree on sid clustered
- Selection condition: **scity = ‘Fribourg’**
  - Index I2: B+-tree on scity unclustered

- Let’s assume
  - V(Supplier,scity) = 20
  - Max(Supplier, sid) = 1000, Min(Supplier,sid)=1
  - B(Supplier) = 100, T(Supplier) = 1000

- **Cost I1: B(R) * (Max-v)/(Max-Min) = 100*700/999 ≈ 70**
- **Cost I2: T(R) * 1/V(Supplier,scity) = 1000/20 = 50**

Selectivity with Multiple Conditions

What if we have an index on multiple attributes?

- Example selection $\sigma_{a=v1 \wedge b=v2}(R)$ and index on <a,b>

How to compute the selectivity?

- Assume attributes are independent
- $X = 1 / (V(R,a) * V(R,b))$

- Clustered index on <a,b>: cost $B(R)*X$
- Unclustered index on <a,b>: cost $T(R)*X$

Back to Estimating Cost of a Query Plan

- We already know how to
  - Compute the cost of different operations
  - Compute cost of retrieving tuples from disk with different access paths (for more sophisticated predicates than equality)

- We still need to
  - Compute cost of a complete plan

Computing the Cost of a Plan

- Collect statistical summaries of stored data
- Compute cost in a bottom-up fashion
- For each operator compute
  - Estimate cost of executing the operation
  - Estimate statistical summary of the output data

Statistics on the Output Data

- Most important piece of information
  - **Size of operator result**
  - I.e., the number of output tuples

- Projection: output size same as input size
- Selection: multiply input size by reduction factor
  - Similar to what we did for estimating access path selectivity
  - Assume independence between conditions in the predicate
  - (use product of the reduction factors for the terms)

Estimating Result Sizes

- For joins R $\bowtie$ S
  - Take product of cardinalities of relations R and S
  - Apply reduction factors for each term in join condition
  - Terms are of the form: column1 = column2
  - Reduction: 1/ (MAX(V(R,column1), V(S,column2))
  - **Assumes each value in smaller set has a matching value in the larger set**

Our Example

- Suppliers(sno,sname,scity,sstate)
- Supplies(pno,sno,quantity)
- Some statistics
  - T(Supplier) = 1000 records
  - B(Supplier) = 100 pages
  - T(Supplies) = 10,000 records
  - B(Supplies) = 100 pages
  - V(Supplier,scity) = 20, V(Supplier,state) = 10
  - V(Supplies,pno) = 3,000
  - Both relations are clustered

Physical Query Plan 1

(On the fly) π_{sname}

(On the fly) $\sigma_{\text{scity='Seattle'} \wedge \text{sstate='WA'} \wedge \text{pno=2}}$

(Nested loop) $\bowtie_{\text{sno = sno}}$

Suppliers (File scan)

Supplies (File scan)

Selection and project on-the-fly
-> No additional cost.

Total cost of plan is thus cost of join:
=
B(Supplier)+B(Supplier)*B(Supplies)
= 100 + 100 * 100
= 10,100 I/Os

Physical Query Plan 2

(On the fly) π_{sname} (4)

(Sort-merge join) $\bowtie_{\text{sno = sno}}$ (3)

(Scan write to T1) (1) $\sigma_{\text{scity='Seattle'} \wedge \text{sstate='WA'}}$

(Scan write to T2) (2) $\sigma_{\text{pno=2}}$

Suppliers (File scan)

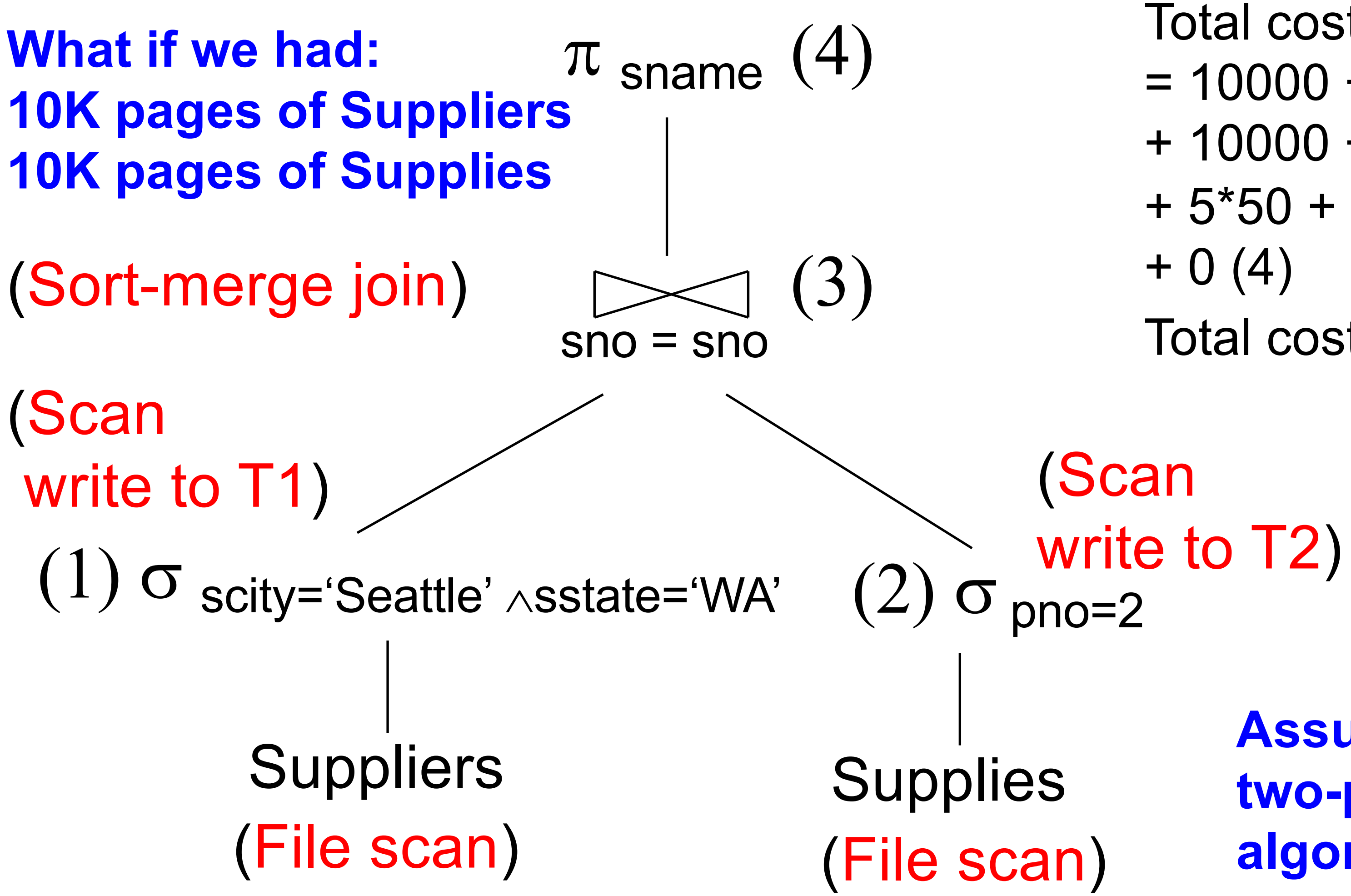
Supplies (File scan)

Total cost
= 100 + 100 * 1/20 * 1/10 (1)
+ 100 + 100 * 1/3000 (2)
+ 2 (3)
+ 0 (4)

Total cost $\approx$ **204 I/Os**

Plan 2 with Different Numbers

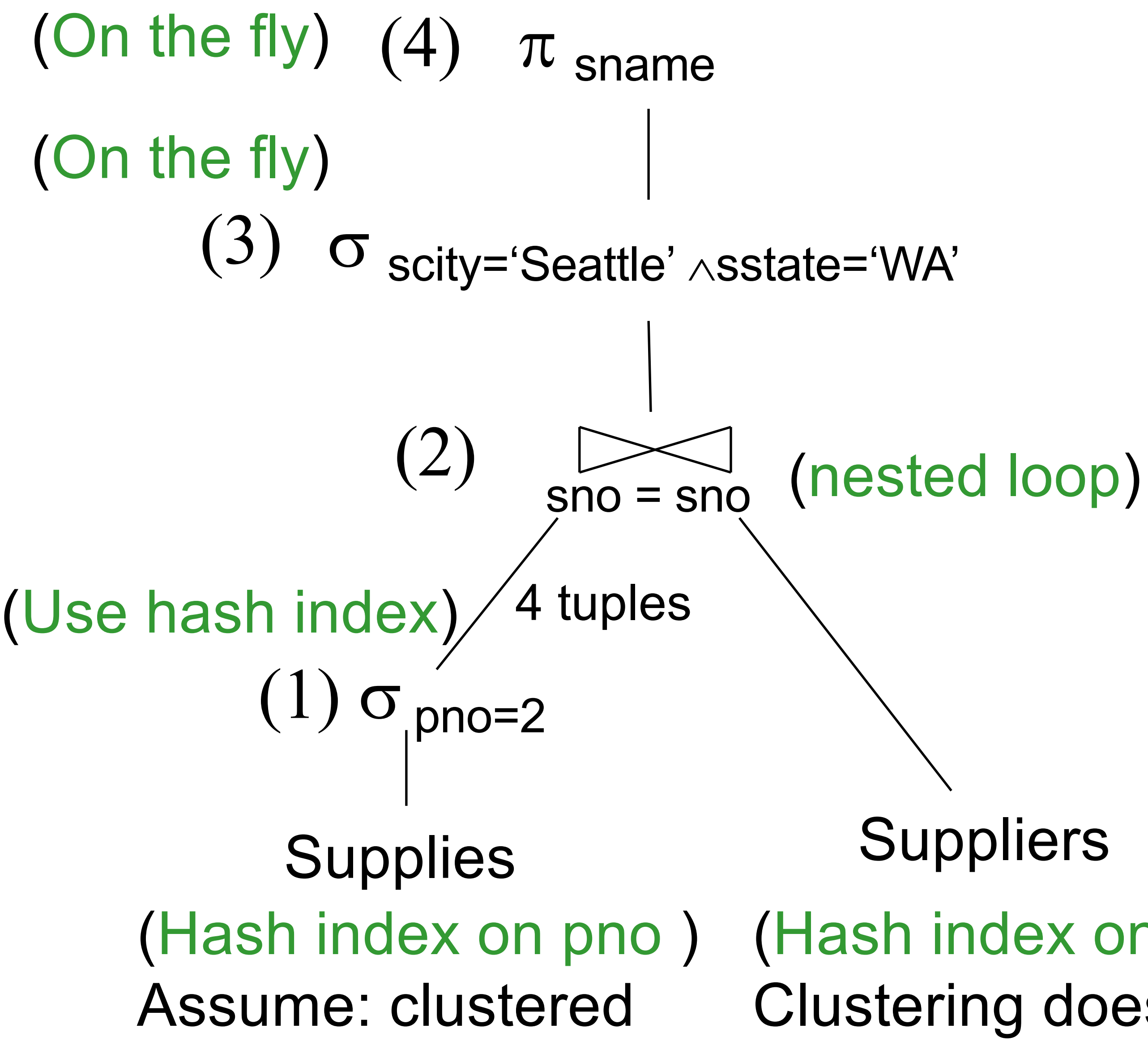
What if we had:
10K pages of Suppliers
10K pages of Supplies



Total cost
= 10000 + 50 (1)
+ 10000 + 4 (2)
+ 5*50 + 5*4 (3)
+ 0 (4)
Total cost $\approx$ 20,324 I/Os

Assuming naive two-pass sort algorithm

Physical Query Plan 3



Total cost
= 1 + 1 (1)
+ 1 + 4 (2)
+ 0 (3)
+ 0 (4)
Total cost ≈ **7 I/Os**

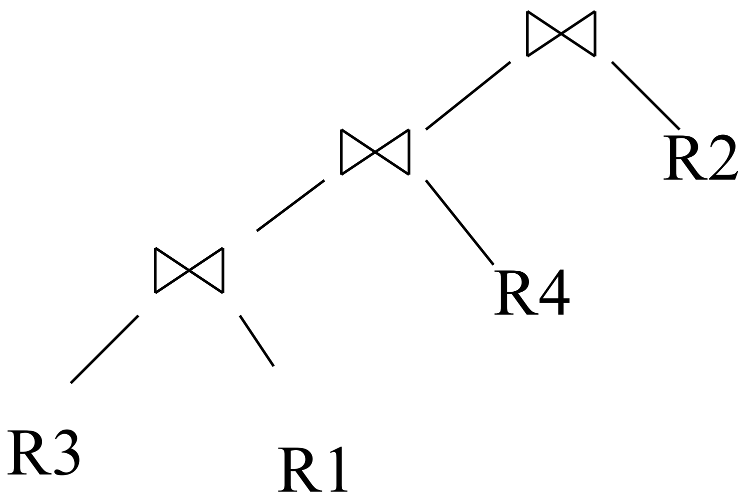
Simplifications

- In the previous examples, we assumed that all index pages were in memory
- When this is not the case, we need to add the cost of fetching index pages from disk

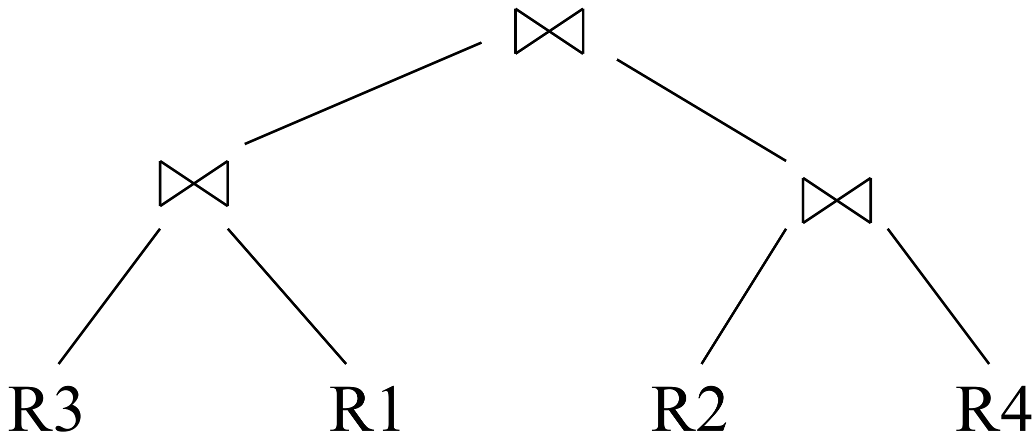
Relational Algebra Equivalences

- Selections
  - Commutative: $\sigma_{c1}(\sigma_{c2}(R))$ same as $\sigma_{c2}(\sigma_{c1}(R))$
  - Cascading: $\sigma_{c1 \wedge c2}(R)$ same as $\sigma_{c2}(\sigma_{c1}(R))$
- Projections
  - Cascading
- Joins
  - Commutative : $R \bowtie S$ same as $S \bowtie R$
  - Associative: $R \bowtie (S \bowtie T)$ same as $(R \bowtie S) \bowtie T$

Left-Deep Plans and Bushy Plans



Left-deep plan



Bushy plan

Search Space Challenges

- Search space is huge!
  - Many possible equivalent trees
  - Many implementations for each operator
  - Many access paths for each relation

- Cannot consider ALL plans
- Want a search space that includes low-cost plans

System R Search Space

- Only left-deep plans
  - Enable dynamic programming for enumeration
  - Facilitate tuple pipelining from outer relation
- Consider plans with all “interesting orders”
- Perform cross-products after all other joins (heuristic)
- Only consider nested loop & sort-merge joins
- Consider both file scan and indexes
- Try to evaluate predicates early

- Many other search strategies possible