# CS-300: Data-Intensive Systems

## Query Processing with Relational Operations

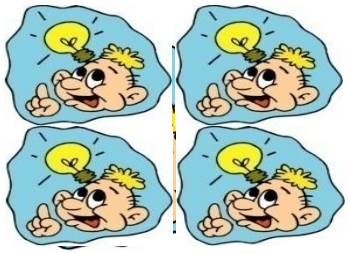
**(Chapters 15.3, 15.5.1-15.5.2, 15.6.1-15.6.2, 15.7)**

*Prof. Anastasia Ailamaki, **Prof. Sanidhya Kashyap***

EPFL

# The big picture

Want to store data



Want to access data

Relational Algebra, SQL

SQL

Result

Conceptual Design

ER Models

ER to Relational

Relational Model

Logical Design

Physical Design

Query Optimization and Execution

Relational Operators

Access Methods

Buffer Management

Disk Space Management

Query processing

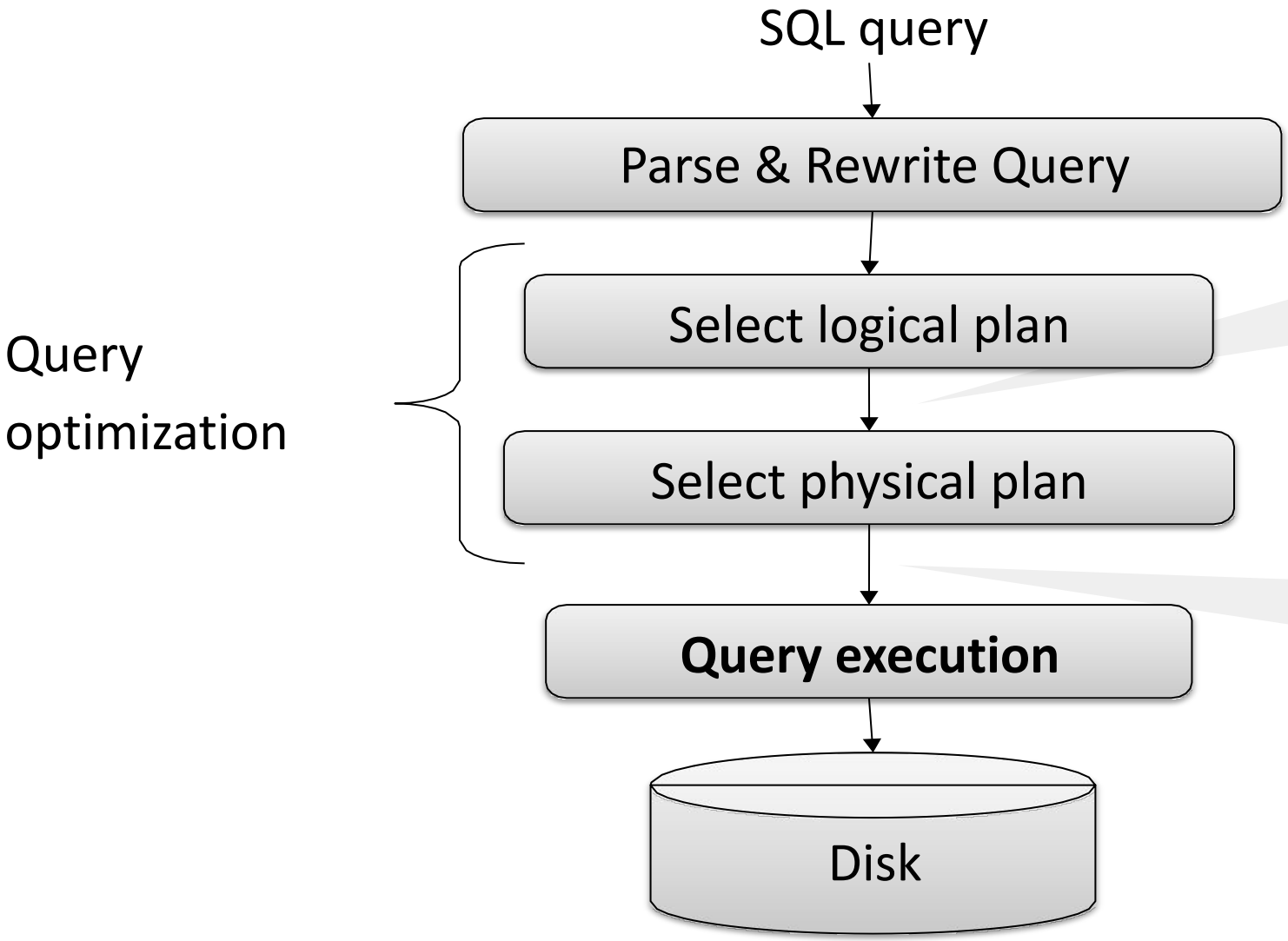
Database Storage

Disk Storage, Files

# Today's focus

- Overview
- Selections
- Projections
- Joins

# Steps in query processing




**Logical plan**
(generates parse tree; nodes looks like relational algebra operations)

**Physical plan**
(similar to logical plan but also includes implementation details)

# Physical query plan

- **Processing model** for operators (scheduling decisions)
  - Pipelined execution (*iterator model*)
  - Intermediate tuple materialization (*materialization model*)

- **Access path selection** for each relation
    - File scan
    - Index lookup with a predicate
- **Implementation choice** for each operator
  - Several algorithms exist

# Iterator model

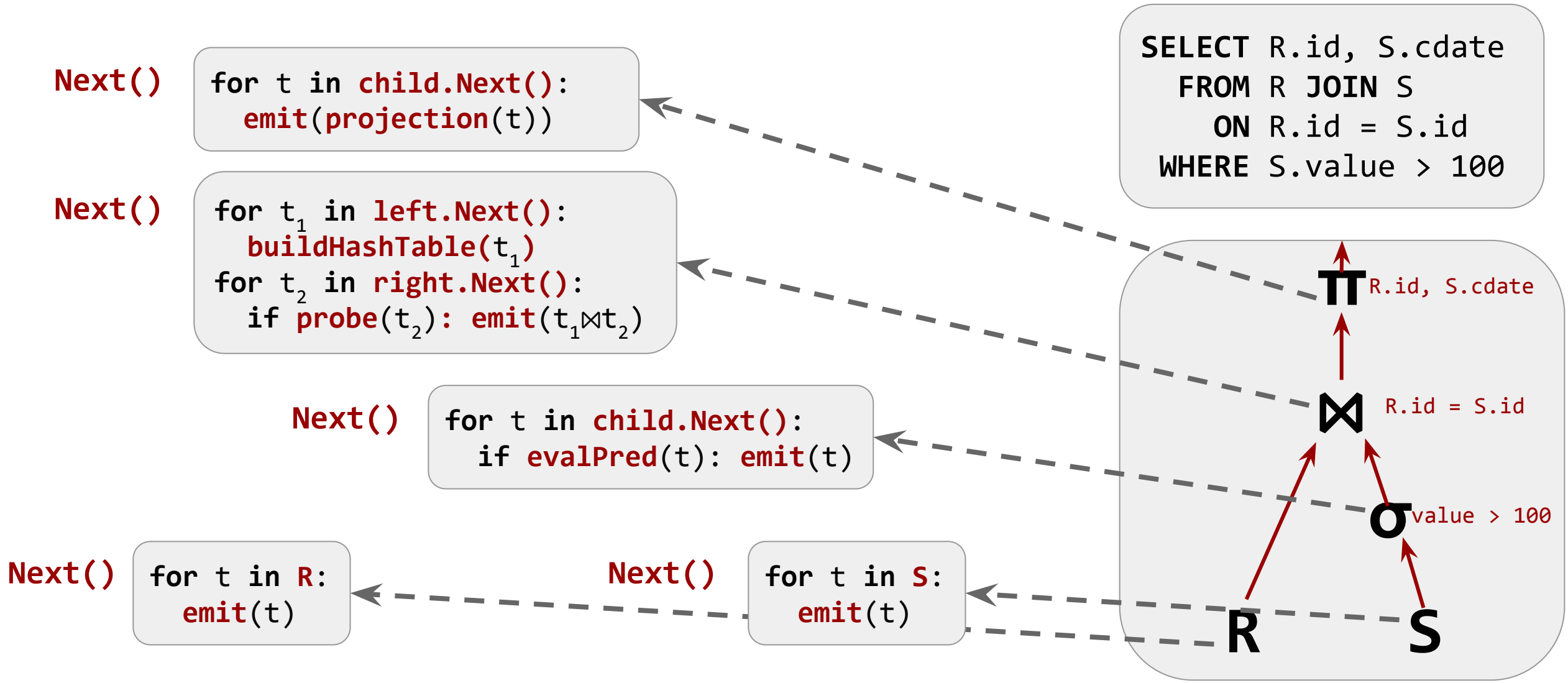
Each query plan operator implements **`Next()`** function

- On each invocation, the operator returns
  - **Single tuple** (single row of data)
  - An end-of-file (**`EOF`**) marker if there are no more tuples
- The operator implements a loop that calls **`Next()`** on its children to retrieve their tuples and then process them

- Each operator also implements **`Open()`** and **`Close()`** functions
  - Analogous to *constructors* and *destructors*, but for operators

Also called pipelined or volcano model

# Iterator model



# Iterator model

❶

```
for t in child.Next():
  emit(projection(t))
```

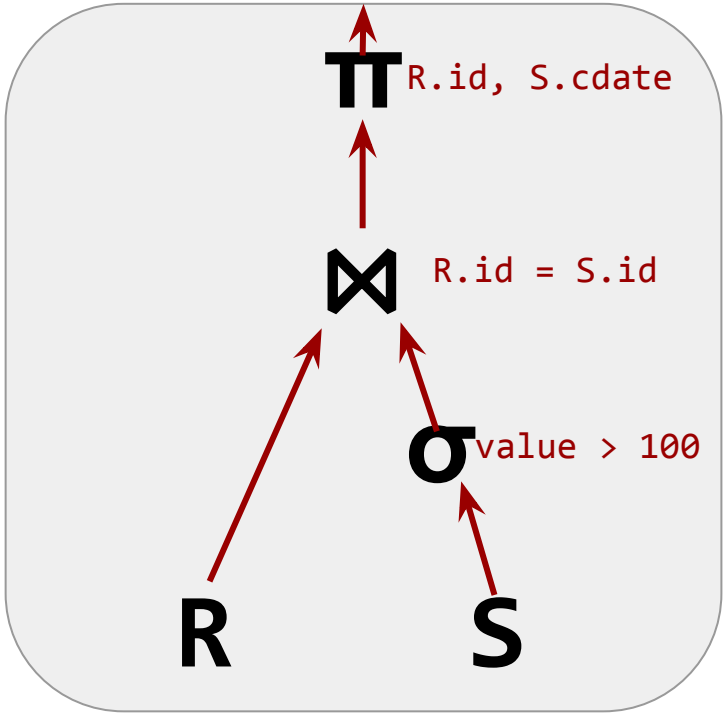
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

```
for t in child.Next():
  if evalPred(t): emit(t)
```

```
for t in R:
  emit(t)
```

```
for t in S:
  emit(t)
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Iterator model

❶
```
for t in child.Next():
  emit(projection(t))
```

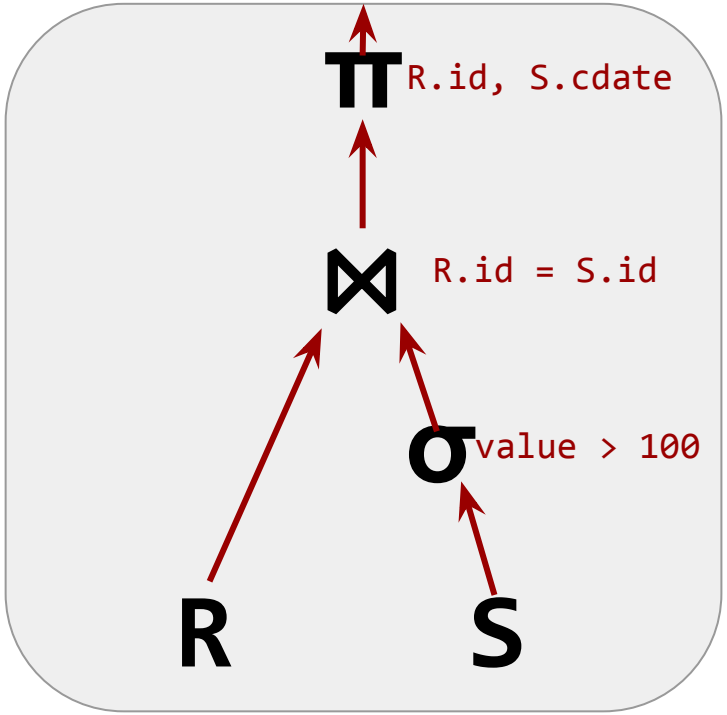
❷
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

```
for t in child.Next():
  if evalPred(t): emit(t)
```

❸
```
for t in R:
  emit(t)
```

```
for t in S:
  emit(t)
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Iterator model

❶
```
for t in child.Next():
  emit(projection(t))
```

❷
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

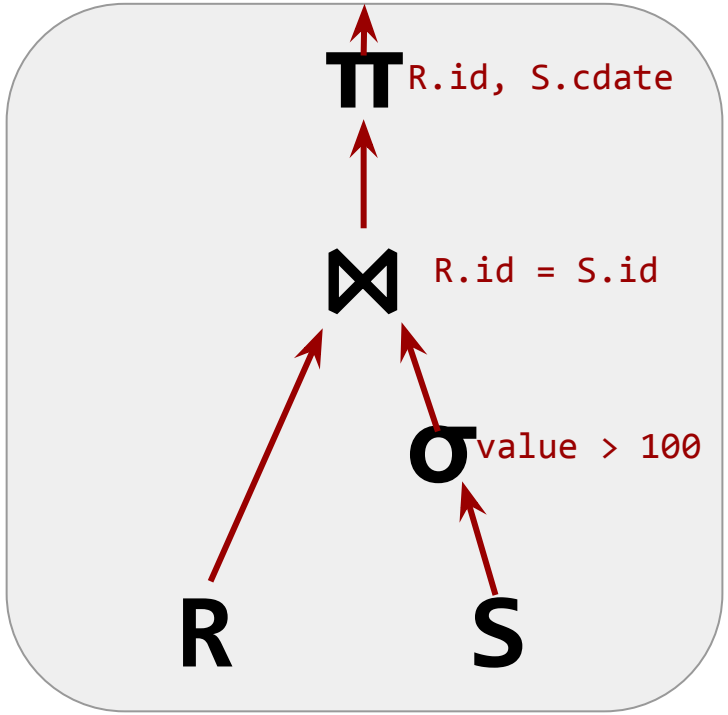
```
for t in child.Next():
  if evalPred(t): emit(t)
```

Single tuple

❸
```
for t in R:
  emit(t)
```

```
for t in S:
  emit(t)
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Iterator model

❶
```
for t in child.Next():
  emit(projection(t))
```

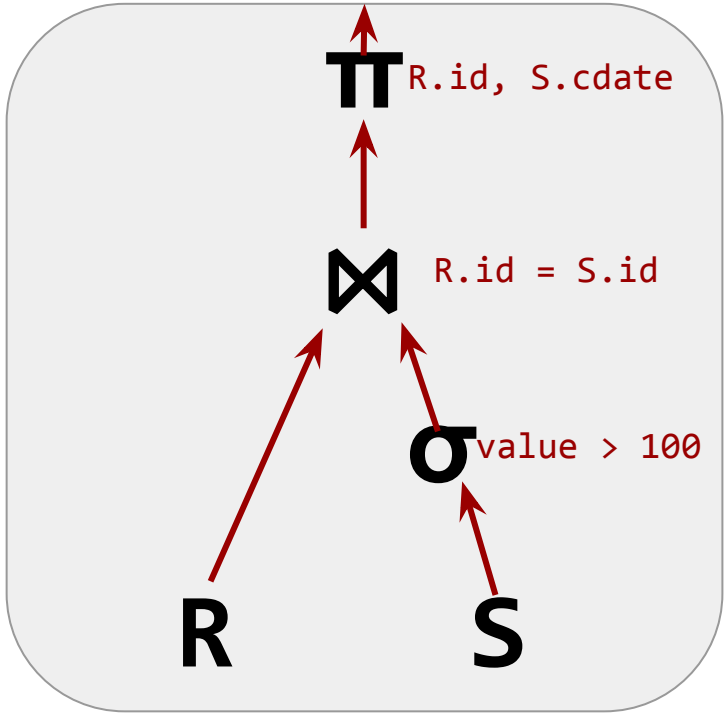
❷
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

❹
```
for t in child.Next():
  if evalPred(t): emit(t)
```

❸
```
for t in R:
  emit(t)
```

❺
```
for t in S:
  emit(t)
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Iterator model

❶
```
for t in child.Next():
  emit(projection(t))
```

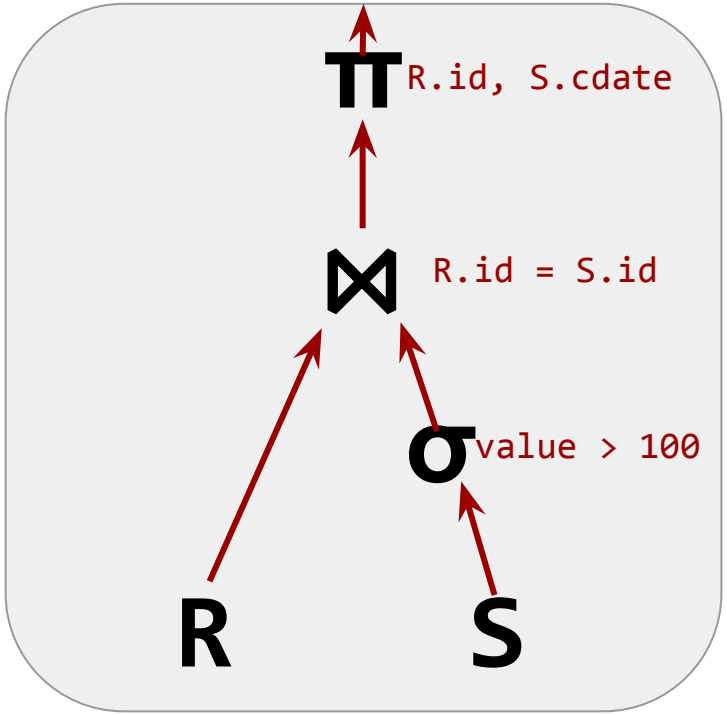
❷
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

❹
```
for t in child.Next():
  if evalPred(t): emit(t)
```

❸
```
for t in R:
  emit(t)
```

❺
```
for t in S:
  emit(t)
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Iterator model

❶
```
for t in child.Next():
  emit(projection(t))
```

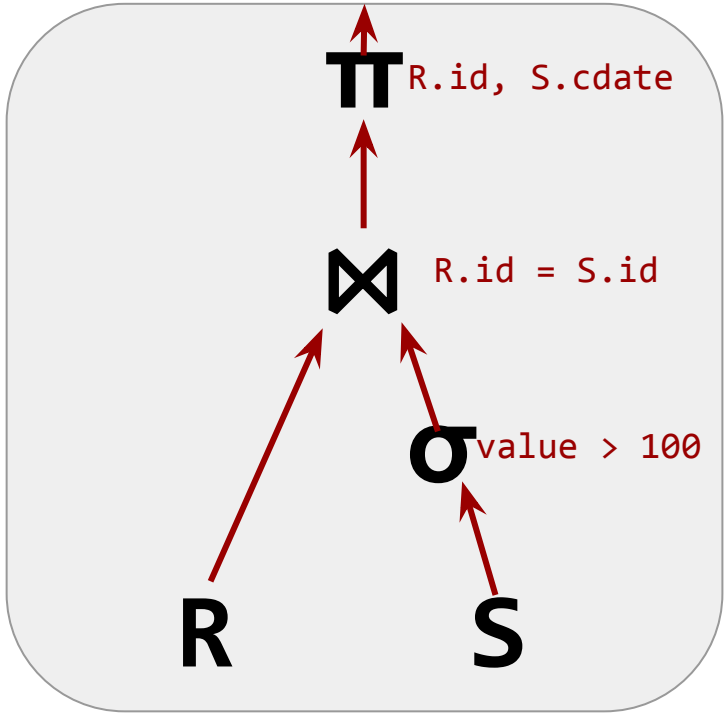
❷
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

❹
```
for t in child.Next():
  if evalPred(t): emit(t)
```

❸
```
for t in R:
  emit(t)
```

❺
```
for t in S:
  emit(t)
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Iterator model

❶
```
for t in child.Next():
  emit(projection(t))
```

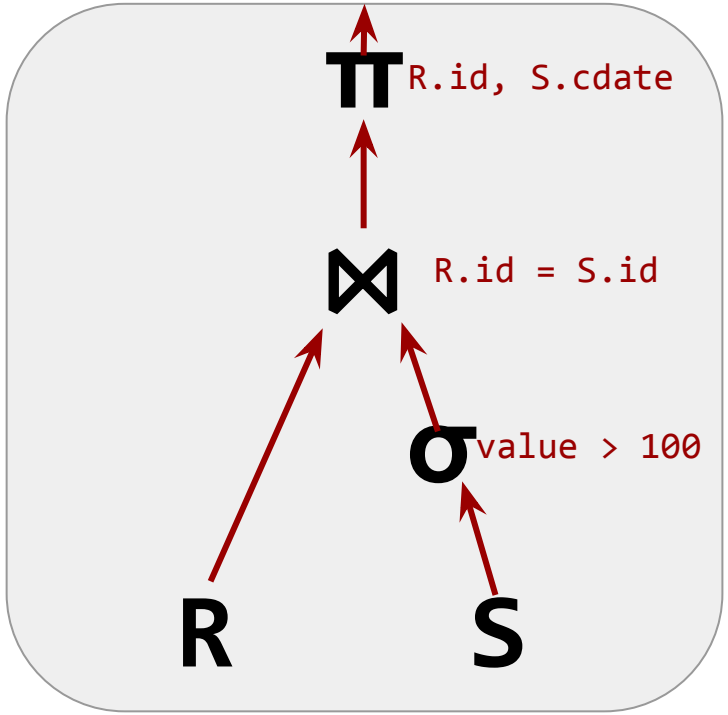
❷
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

❹
```
for t in child.Next():
  if evalPred(t): emit(t)
```

❸
```
for t in R:
  emit(t)
```

❺
```
for t in S:
  emit(t)
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Iterator model

- Tuples generated by an operator are immediately sent to the parent
  - Tuples are pipelined

**Benefits:**

- Pull-based: No operator synchronization issues
- Can save cost of writing intermediate data to disk
- Can save cost of reading intermediate date from disk

- Used by almost every DBMS
- Many operators must block until their children emit all their tuples:
  - Joins, aggregates, subqueries, order by

# Materialization model

Each operator processes its input all at once and then emits output all at once

- The operator “materializes” its output as a single result
- DBMS can push down hints (e.g., **LIMITS**) to avoid scanning too many tuples
- Can send either a materialized row or a single column

DBMS output can be either whole tuples or subsets of columns

# Materialization model

1

```
out = []
for t in child.Output():
  out.add(projection(t))
return out
```

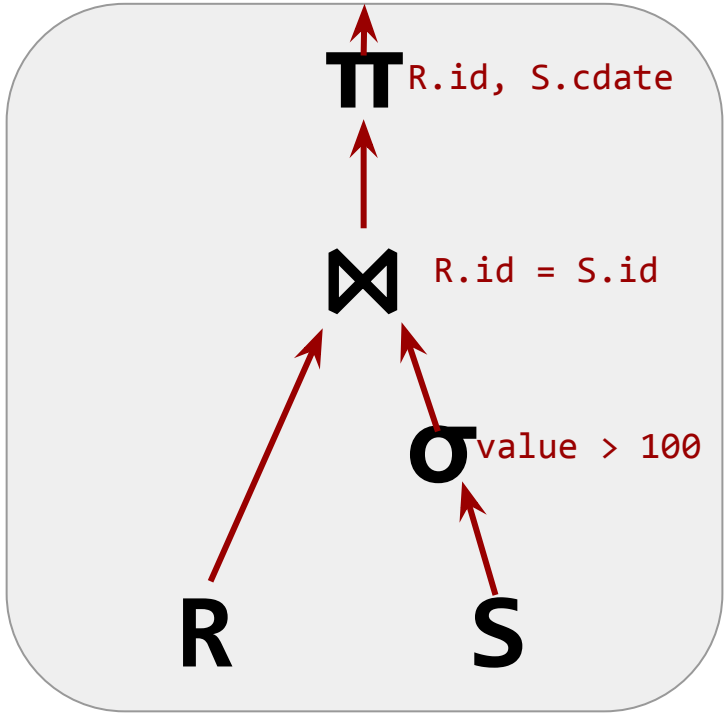
```
out = []
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
out = []
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

```
out = []
for t in R:
  out.add(t)
return out
```

```
out = []
for t in S:
  out.add(t)
return out
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Materialization model

1
```
out = []
for t in child.Output():
  out.add(projection(t))
return out
```

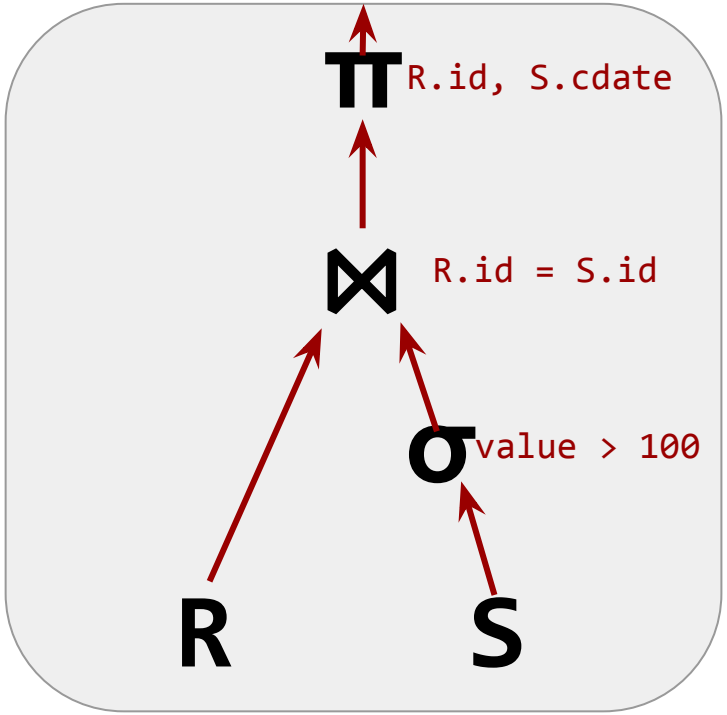
2
```
out = []
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
out = []
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

3
```
out = []
for t in R:
  out.add(t)
return out
```

```
out = []
for t in S:
  out.add(t)
return out
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Materialization model

❶
```
out = []
for t in child.Output():
  out.add(projection(t))
return out
```

❷
```
out = []
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

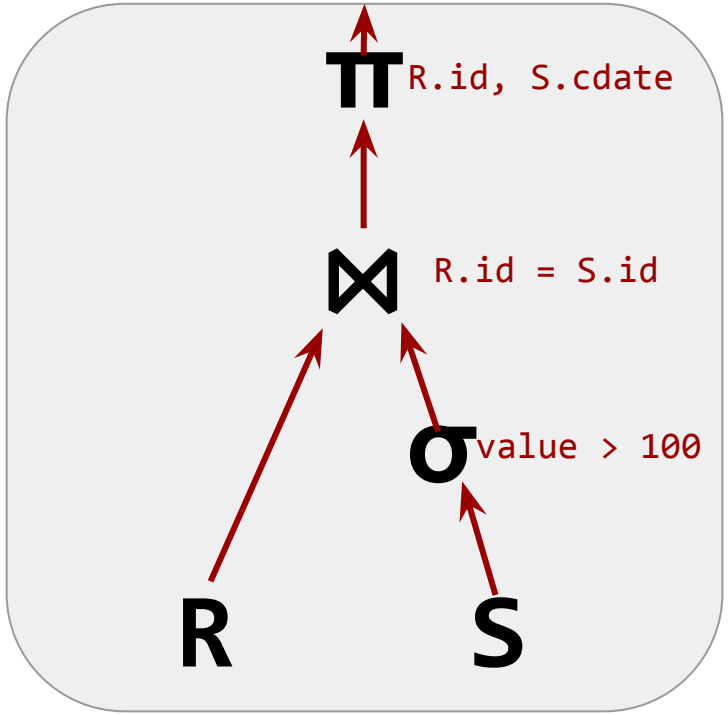
All tuples

```
out = []
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

❸
```
out = []
for t in R:
  out.add(t)
return out
```

```
out = []
for t in S:
  out.add(t)
return out
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Materialization model

1
```
out = []
for t in child.Output():
  out.add(projection(t))
return out
```

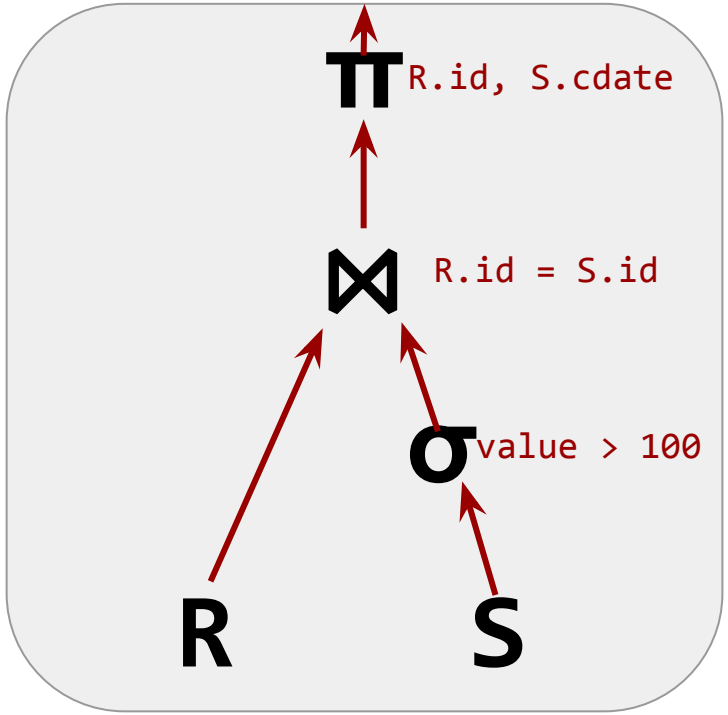
2
```
out = []
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

3
```
out = []
for t in R:
  out.add(t)
return out
```

4
```
out = []
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

```
out = []
for t in S:
  out.add(t)
return out
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Materialization model

❶
```
out = []
for t in child.Output():
  out.add(projection(t))
return out
```

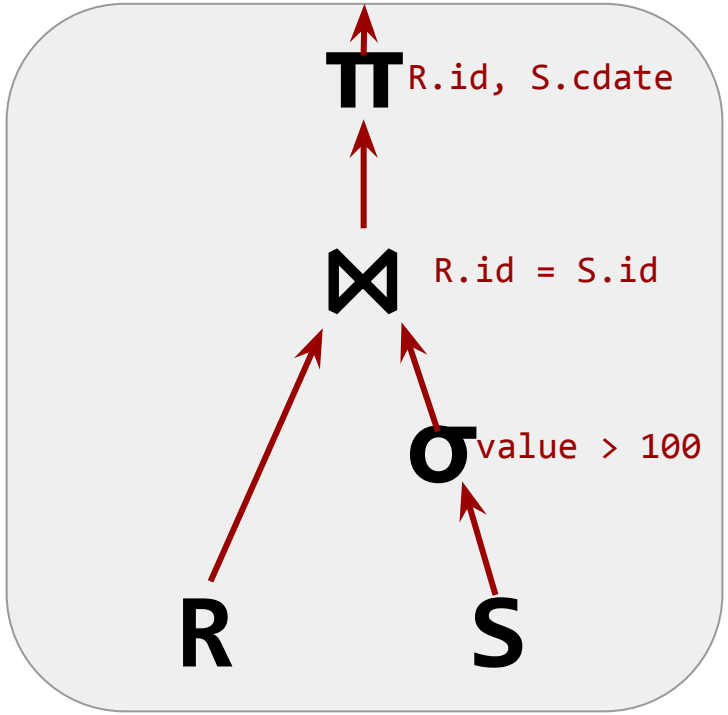
❷
```
out = []
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

❸
```
out = []
for t in R:
  out.add(t)
return out
```

❹
```
out = []
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

❺
```
out = []
for t in S:
  out.add(t)
return out
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Materialization model

❶
```
out = []
for t in child.Output():
  out.add(projection(t))
return out
```

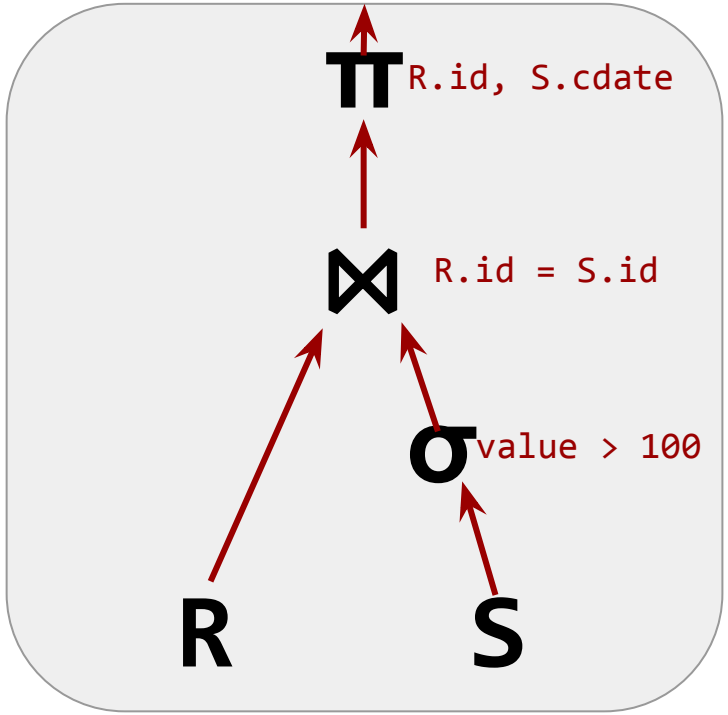
❷
```
out = []
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

❸
```
out = []
for t in R:
  out.add(t)
return out
```

❹
```
out = []
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

❺
```
out = []
for t in S:
  out.add(t)
return out
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Materialization model

❶
```
out = []
for t in child.Output():
  out.add(projection(t))
return out
```

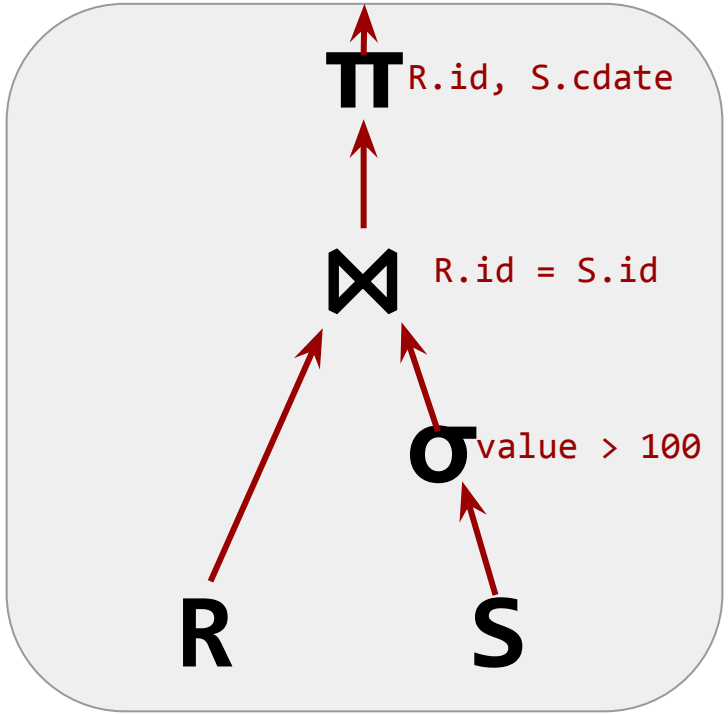
❷
```
out = []
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

❸
```
out = []
for t in R:
  out.add(t)
return out
```

❹
```
out = []
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

❺
```
out = []
for t in S:
  out.add(t)
return out
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Materialization model

❶
```
out = []
for t in child.Output():
  out.add(projection(t))
return out
```

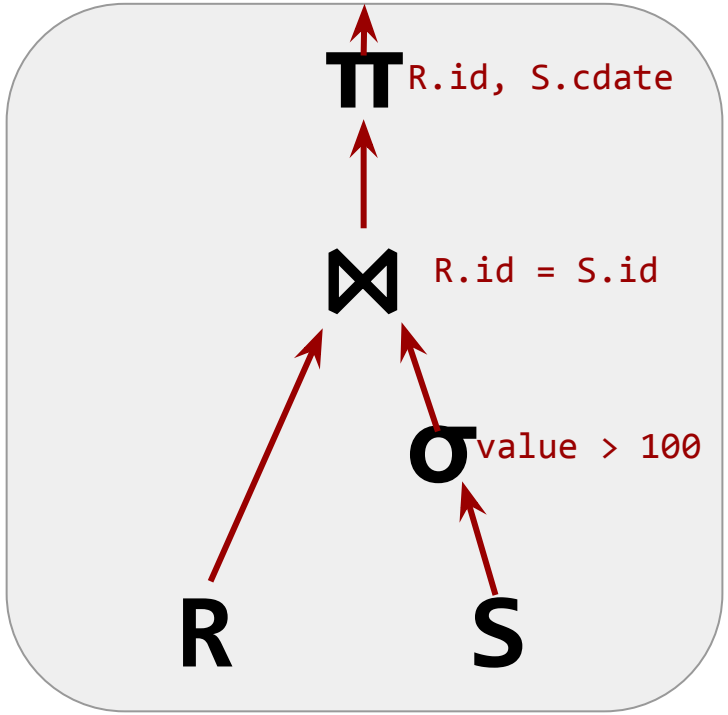
❷
```
out = []
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

❸
```
out = []
for t in R:
  out.add(t)
return out
```

❹
```
out = []
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

❺
```
out = []
for t in S:
  out.add(t)
return out
```

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Materialization model

Better for transaction processing workloads because queries only access a small number of tuples at a time

- Lower execution / coordination overhead
- Fewer function calls

Not good for analytical queries with **large intermediate results**

- Requires memory and sometimes it won't be enough

# Plan processing direction

**Approach #1:** Top-to-Bottom (Pull)

- Start with the root and “pull” date up from its children
- Tuples are always passed with function calls
- Overhead: Next() call implemented as virtual functions; CPU branching cost

**Approach #2:** Bottom-to-Top (Push)

- Start with leaf nodes and push data to their parents
- Allows for tighter control of caches / registers in pipelines
- More amenable to dynamic query re-optimization
- Difficult to control the size of intermediate result sizes and some operators too

# Physical query plan

- **Processing model** for operators (scheduling decisions)
  - Pipelined execution (*iterator model*)
  - Intermediate tuple materialization (*materialization model*)

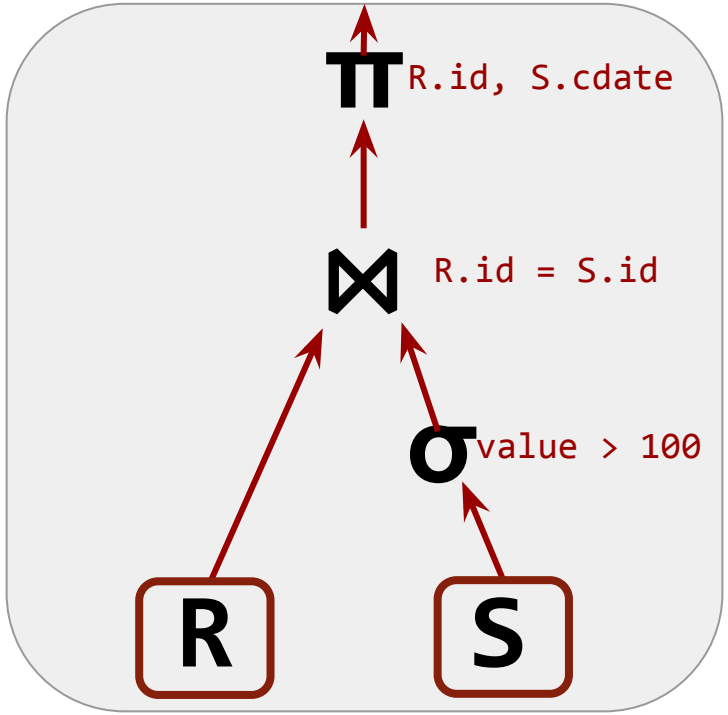
- **Access path selection** for each relation
  - File scan
  - Index lookup with a predicate
- **Implementation choice** for each operator
  - Several algorithms exist

# Access methods

- An access method is the approach how a DBMS accesses the data stored in a table
  - Not part of relational algebra
  - Required for generating physical plan

- Three-basic approaches:
  - Sequential scan: reading the whole table
  - Index scan: Scan existing indexes to access table
  - Multi-index scan: Use multiples of indexes on a set of tables

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



# Physical query plan

- **Processing model** for operators (scheduling decisions)
  - Pipelined execution (*iterator model*)
  - Intermediate tuple materialization (*materialization model*)

- **Access path selection** for each relation
    - File scan
    - Index lookup with a predicate
- **Implementation choice** for each operator
  - Several algorithms exist

# Implementing relational operators

Considering their implementation:

- ***Selection*** (σ): Selects a subset of rows from relation
- ***Projection*** (π): Deletes unwanted columns from relation
- ***Join*** (⋈): Allows us to combine two relations
- ***Set-difference*** (−): Tuples in relation 1, but not in relation 2
- ***Union*** (∪): Tuples in relation 1 and in relation 2
- ***Aggregation*** (SUM, MIN, etc.): and GROUP BY

# Today's focus

- Overview
- **Selections**
- Projections
- Joins

# Simple selection:

Operation of choosing or filtering rows from a relation based on specific criterion

- Of the form: $\sigma_{\text{R.attr } \mathbf{op} \text{ value}}$ (R)

- Best approach to implement depends on:
  - Available index / access paths
  - Expected size of the result (# tuples / # pages)

```
SELECT *
  FROM Reserves R
  WHERE R.rname < 'C%'
```

- Size of result approximated as

  **size of R * *reduction factor***

  - "Reduction factor" also known as **selectivity**
  - Selectivity estimate is based on statistics

# Alternatives to simple selections

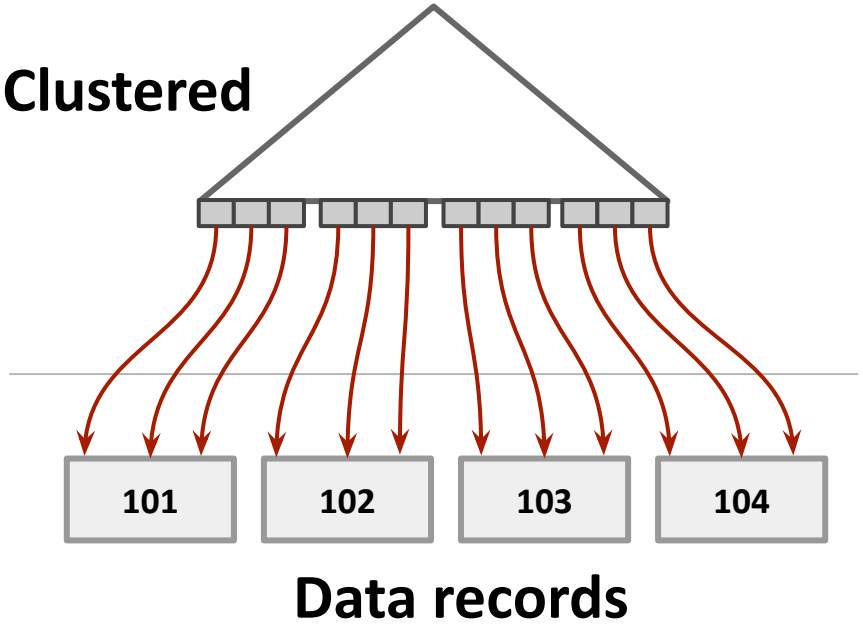
- **With no index, unsorted**
  - Must scan the whole relation
  - Cost is **M** (# pages in R); for Reserves = 1000 IOs
- **With no index, sorted:**
  - Cost of binary search + number of pages containing results
  - For reserves = ~10 IO + [**selectivity * # pages]**
  - This is rare; most likely an index will definitely exist
- **With an index on selection attribute:**
  - Use index to find qualifying data entries
  - Then retrieve corresponding data records

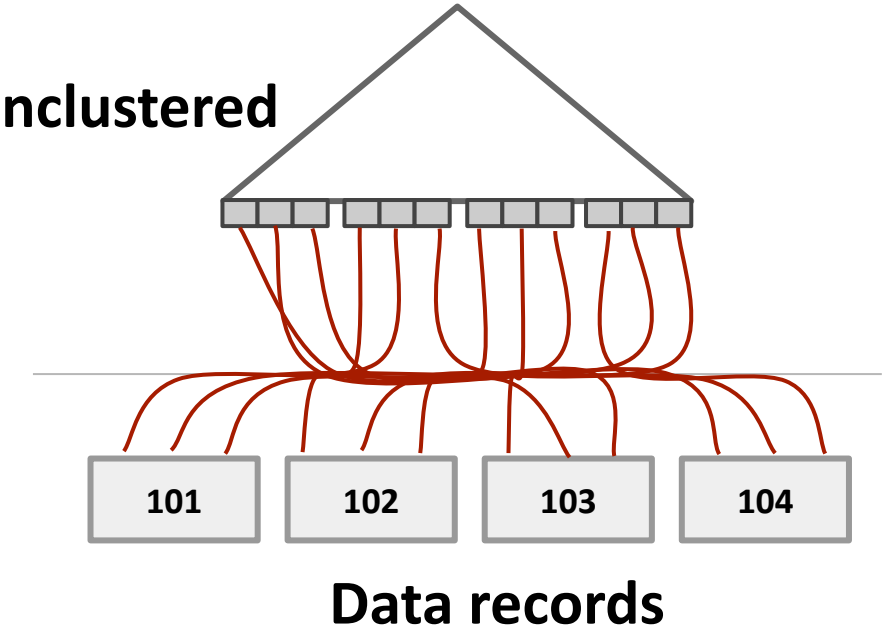
```
SELECT *
  FROM Reserves R
  WHERE R.rname < ‘C%’
```

# Selections using an index: Example

- If 10% of tuples qualify
  - 10,000 (out of 100K) tuples, 100 (out of 1K) pages
  - Clustered index: a bit more than 100 IOs
  - Unclustered index: Can go up to 10,000 IOs

```
SELECT *
  FROM Reserves R
  WHERE R.rname < 'C%'
```

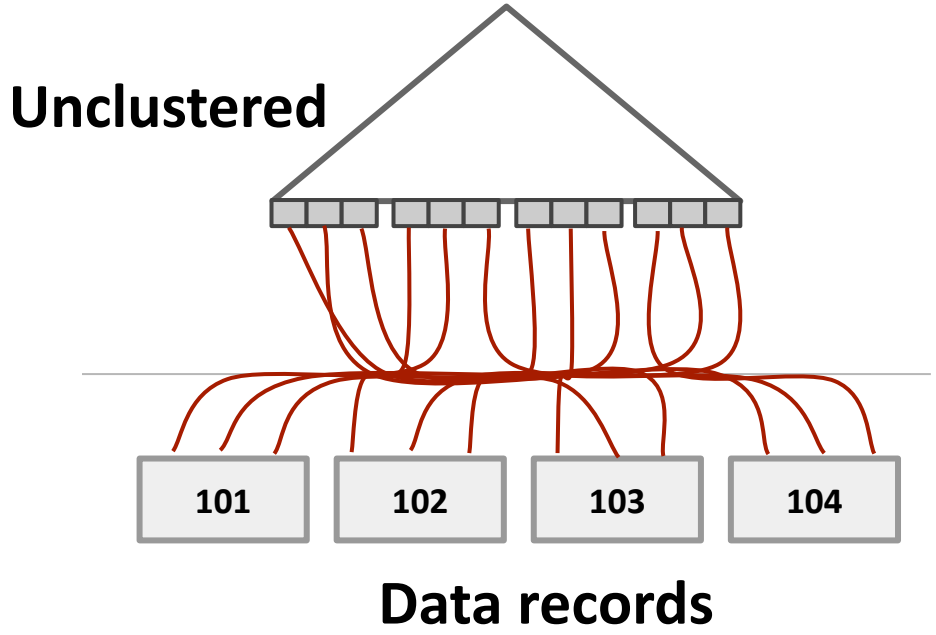


# Optimization for unclustered index

- Find qualifying data entries
- Sort the **`rids`** of the data records to be retrieved by their page-id component
- Fetch **`rids`** in order
  - Ensure that each data page is accessed only once

```
SELECT *
  FROM Reserves R
  WHERE R.rname < ‘C%’
```



# General selection conditions

Selection condition is a combination of conditions using AND / OR

```
(day < 8/9/94 AND rname=`Paul') OR bid=5 OR sid=3
```

1. First convert to conjunctive normal form (CNF) (collection of ANDs) →

```
(day < 8/9/94 OR bid=5 OR sid=3) AND (rnam=`Paul' OR bid=5 OR sid=3)
```

2. Match conditions to indexes

- A B+Tree index **works best** for prefix (start) of the indexed attributes
  - Index on `<a, b, c>` matches `a=5 AND b=3`, but not `b=3`
- Hash index must have all attributes in search keys

# Selection: 1st approach

Similar to using index for **simple selection**

1. Pick a set of conjuncts that match an index
2. Retrieve tuples using it
3. Apply the conjuncts that do not **match** the index (if any)

# Selection: 1st approach: Example

- Consider: **(day < 8/9/94 AND rname=`Paul') OR bid=5 OR sid=3**

- Simplest approach: Scan and check each tuple
- Another approach:
  - Use B+Tree on **day**
    i. Retrieve all tuples matching day < 8/9/94
    ii. Filter the results using rname=`Paul'
    iii. Get results from (bid=5 OR sid=3)
  - A hash index on **<bid>** and another hash index on **<sid>**
    i. Retrieve all tuples matching bid=3 OR sid=3
    ii. For each retrieved tuple, check **day<8/9/94 AND rname=`Paul'**

# Selection: 1st approach: Example

- Consider: **(day < 8/9/94 AND rname=`Paul') OR bid=5 OR sid=3**

Questions: What happens if we have indexes with different attributes orders

- a B+Tree on <rname, day>?
  - First sort by **rname** and then **day**
  - Good choice as equality on **rname**, and then range scan on **day** using B+tree index
- A B+Tree on <day, rname>?
  - First range scan on day
  - Cannot do equality lookup on rname (scan all tuples for day and then filter rname)
- A hash index on <day, rname>?
  - A bad choice here as we have to generate all possible rnames

# Selection: 2nd approach: Intersecting RIDs

- If we have two or more matching indexes
  a. **Retrieve** a set of **`rids`** of data records using **each matching index**
  b. Then **intersect** these sets of collected `rids`
  c. Retrieve the records and **apply any remaining terms**

Consider: **`day < 8/9/94 AND bid=5 OR sid=3`**

- With a **B+Tree index on day** and an **hash index on sid:**
  a. Retrieve `rids` of records satisfying `day<8/9/94`
  b. Retrieve `rids` of records satisfying `sid=3`
  c. Intersection of the output of the above two
  d. Retrieve records and check `bid=5`

# Selection: Summary

- Simple selections on one condition
  - On sorted or unsorted data, with and without index
- General selections
  - Use conjunctive normal form to express them
  - Retrieve tuples and then filter them through conditions
  - Intersect rids of matching tuples for non-clustered indexes
- Choice depends on **selectives**

# Today's focus

- Overview
- Selections
- **Projections**
- Joins

# The projection operation

Remove duplicates (which is challenging)

- Basic approach is to use sorting
  a. Scan R, extract only needed attributes
  b. Sort the resulting set
  c. Remove adjacent duplicates

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

# General external merge sort: Recap

To sort a file with N pages using B buffer pages:

**Pass #0**

- Produce $\lceil N/B \rceil$ sorted runs of size **B**

**Pass #1,2,3 …**

- Merge **B-1** runs (i.e., k-way merge)

Number of passes = $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$

Total IO cost = $2N * (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$

# The projection operation

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

Example cost calculation scenario

- Original table size: 1000 pages
- Size ratio = .25 (sid and bid)
- Buffer pages for sorting = 20; 2 pass sorting can be done

Steps:

1. Extract relevant attributes: 1000 reads; write smaller set: 250 pages
2. Sort the smaller set (250 pages): 2 * 2 * 250 = 1000 IO operations
3. Remove duplicates by reading all 250 pages = 250 IO operations

Total: 2500 IO operations

# The projection operation: more optimization

- Modify external sort algorithm and **apply projections on the fly**:
  1. Modify pass #0 of external sort to eliminate unwanted fields
  2. Modify merging passes to eliminate duplicates

- Cost:
  - With 20 buffer pages, Reserves (table) with ratio .25 = 250 pages
  - Read 1000 pages: 1000 IOs
  - Write out 250 page in ~13 runs of 19 pages each **with projections**: 250 IOs
    - 13 sorted groups
  - Merge 13 runs: 250 IOs (reads)
  - Total: M + T + T = 1500 pages

# Projection using hashing

Good candidate if ample amount of buffer is available

**2 phase: partitioning and duplicate elimination**

- *Partitioning phase*:
  - Read R using one input buffer
  - For each tuple:
    - Discard unwanted fields
    - Apply hash function h1 to choose one of B-1 output buffers
  - Result is B-1 partitions (of tuples with no unwanted fields)
    - Tuples in **different partitions** are guaranteed **not to be duplicates**

# Projection using hashing

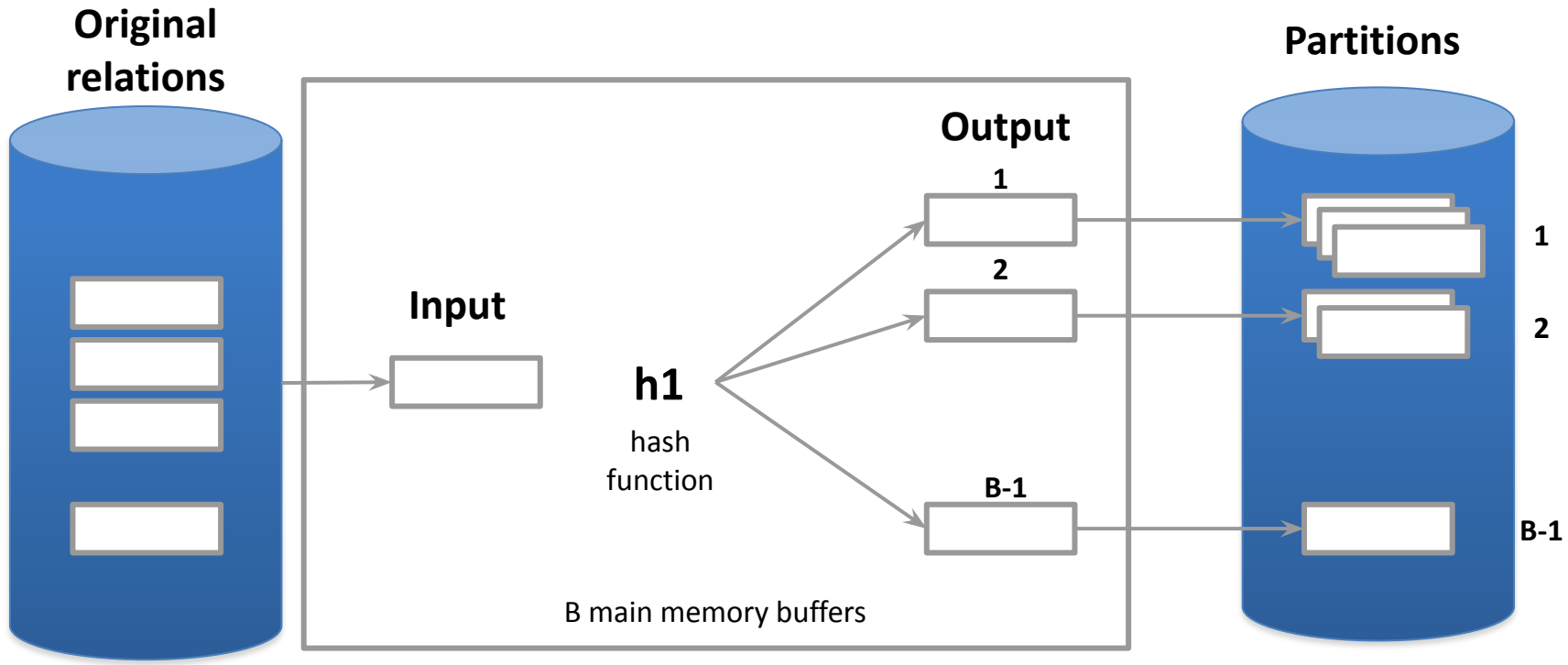
- **2 phase: partitioning and duplicate elimination**

- *Duplicate elimination phase*:
  - For each partition:
    - Read it and build an in-memory hash table
    - Use hash function h2 (!= h1) on all fields
    - Discard duplicates during this phase

  - If partition does not fit in memory:
    - Apply hash-based projection algorithm recursively to this partition

# Projection using hashing: Memory

- Assume:
  - h1 distributes tuples uniformly, T is # pages after projection
  - # pages per partition = T / (B - 1)
  - B >= T / (B - 1) or B > √T



# Projection using hashing: Cost

- Assume partitions fit in memory (i.e., B > $\sqrt{T}$)
  - Read pages: 1000 IOs
  - Write partitions of projected tuples: 250 IOs
  - Do duplicate elimination on each partition: 250 IOs
  - Total: M + T + T = 1500 IOs

# Understanding projection

- Sort-based approach is standard
  - Handles **skewed** data efficiently
  - Produces **sorted** result (can be used by subsequent joins or grouping)

- If there are more buffers, both have the same IO cost: M + 2T
  - M: # pages in R
  - T: # pages of R with unneeded attributes removed

- If all relevant attributes are indexed
  - Use **index-only** scan

# Today's focus

- Overview
- Selections
- Projections
- **Joins**

# Joins

- Combines two relations: R⋈S
- Combine multi-joins using a pair-wise joins from left to right
- At a high-level: combination of relation product followed by a selection
  - Inefficient as large intermediate results
- Join operator algorithms:
  - One pass algorithms (this lecture)
  - Index-based algorithms (next lecture)
  - Two-pass algorithms (next lecture)
- Join techniques:
  - Nested-loop joins, sort-merge join, hash join

# Simple nested loops join

- For each tuple in the **outer** relation **R**, scan the entire **inner** relation **S**

```
foreach tuple r in R do
  foreach tuple s in S do
    if r_i==s_j then add<r,s> to result
```

- If we assume:
  - M pages in R, $p_R$ tuples per page
  - N pages in S, $p_S$ tuples per page

- IO (read) cost: M + ((M * $p_R$) * N)

  We process every tuple of R, and for every tuple of R, we read N
- Ignore CPU and writing output cost

# Simple nested loops join: Cost

- R(outer) is Reserves and S(inner) is Sailors
  - M = 1000, $p_R$ = 100
  - N = 500, $p_S$ = 80
  - Cost: M + ($p_R$ * M) * N = 1K + 100*1K*500 = 50,001,000
  - At 10ms/IO, total: 140 hours

```
foreach tuple r in R do
  foreach tuple s in S do
    if r_i==s_j then add<r,s> to result
```

```
SELECT *
FROM Reserves R
WHERE R1.sid=S1.sid
```

# Simple nested loops join: Cost

- R(outer) is Reserves and S(inner) is Sailors
  - M = 1000, $p_R$ = 100
  - N = 500, $p_S$ = 80
  - Cost: M + ($p_R$ * M) * N = 1K + 100*1K*500 = 50,001,000
  - At 10ms/IO, total: 140 hours

```
foreach tuple r in R do
  foreach tuple s in S do
    if r_i==s_j then add<r,s> to result
```

```
SELECT *
FROM Reserves R
WHERE R1.sid=S1.sid
```

OR

- R(outer) is Sailors and S(inner) is Reserves
- IO Cost: 500 + 80*500*1K = 40,000,500
- At 10ms/IO, total = 111 hours

**Choice of inner/outer matters in join!**

# Page-oriented nested loops join

- Process data **page-by-page** than tuple-by-tuple

```
foreach page b_R in R do
 foreach page b_S in S do
  foreach tuple r in b_R do
   foreach tuple s in b_S do
    if r_i==s_j then add<r,s> to result
```

- For each page of outer **R**
  - Scan all the pages of inner **S**
  - For each pair of pages (one from **R** and one from **S**), match tuples
  - If tuples match (<**r**,**s**>: **r** is in **R**-page and **s** is in **S**-page), output them

- IO Cost: M + M * N
  - M: Reading each page from **R** exactly once
  - M*N: Reading all pages from **S** for every page of **R**

# Nested loops: Cost

- `R(outer)` is Reserves and `S(inner)` is Sailors
  - M = 1000, $p_R$ = 100
  - N = 500, $p_S$ = 80

```
SELECT *
FROM Reserves R
WHERE R1.sid=S1.sid
```

- Page-oriented nested loop
  - IO Cost: M + M * N = 1K + 1K * 500 = 501,000
  - At 10 ms/IO, total = 501,000 * 10 = **1.5 hours**
- Simple nested loop
  - IO Cost: M + ($p_R$ * M) * N = 1K + 100*1K*500 = 50,001,000
  - At 10 ms/IO, total: **111 hours**

# Block nested loop joins: use available buffers

- Page-oriented nested loop does not use extra buffers
- Buffer allocation: 1 page for `S(inner)`, 1 for output and the remaining for `R(outer)`($block_R$)
- Steps:
  - Read multiple pages of R into memory
  - For this block R, scan each page of S and find matching tuples
  - Output matched tuple

```
foreach block block_R in R do
 foreach page b_S in S do
  foreach tuple r in block_R do
   foreach tuple s in b_S do
    if r_i==s_j then add<r,s> to result
```

- IO Cost = M + (# outer blocks * N (inner scan))
  - # outer blocks = ⌈# outer pages / blocksize⌉

# Nested loops: IO Cost

- `R(outer)` is Reserves and `S(inner)` is Sailors
  - M = 1000, $p_R$ = 100
  - N = 500, $p_S$ = 80

```
SELECT *
FROM Reserves R
WHERE R1.sid=S1.sid
```

- Block nested loop with block size = 100:
  - IO cost of scanning R is 1000 IOs, a total of 10 blocks
  - For each block of R, we scan Sailors: 500 * 10 = 5000 IOs
  - total IO cost: 6000 IOs; **1 minute**

- Page-oriented nested loop ⟶ total: **1.5 hours**
- Simple nested loop ⟶ total: **111 hours** of IO time

# Hash join

- Hash join: R⋈S
  - Scan R, build buckets in main memory
  - Then scan S, probe and join

  - Cost: M (outer scan) + N (inner scan)

- One-pass assumption:
  - Enough memory to store M + hashtable

# Nested loops: Cost

- `R(outer)` is Reserves and `S(inner)` is Sailors
  - M = 1000, $p_R$ = 100; N = 500, $p_S$ = 80

```
SELECT *
FROM Reserves R
WHERE R1.sid=S1.sid
```

- Hash join
  - IO Cost: M + N = 1K + 500 = 1500, **15 secs**

- Block nested loop → total: **1 minute**
- Page-oriented nested loop → total: **1.5 hours**
- Simple nested loop → total: **111 hours** of IO time

# Sort-merge join

- Hash join: R⋈S
  - Scan R and sort in main memory
  - Scan S and sort in main memory
  - Merge R and S
- IO Cost: M (outer scan) + N (inner scan)
  - Same as hash join

- One-pass assumption:
  - Enough memory to store M + N

- Usually not a one-pass algorithm

# Nested loops: Cost

- `R(outer)` is Reserves and `S(inner)` is Sailors
  - M = 1000, $p_R$ = 100; N = 500, $p_S$ = 80

```
SELECT *
FROM Reserves R
WHERE R1.sid=S1.sid
```

- Sort-merge join
  - IO Cost: M + N = 1K + 500 = 1500, **15 secs**

- Hash join → total: **15 secs**
- Block nested loop → total: **1 minute**
- Page-oriented nested loop → total: **1.5 hours**
- Simple nested loop → total: **111 hours** of IO time

# Summary

- SQL query transformed into physical plan
  - Scheduling decisions for operators
  - Implementation choice for each operator
  - Access path selection for each relation

- Two scheduling types / processing model:
  - Iterator vs materialization

- Many relative implementations for each operator
  - Huge difference in cost