CS 4440 A

# Emerging Database Technologies

Lecture 8
02/05/24

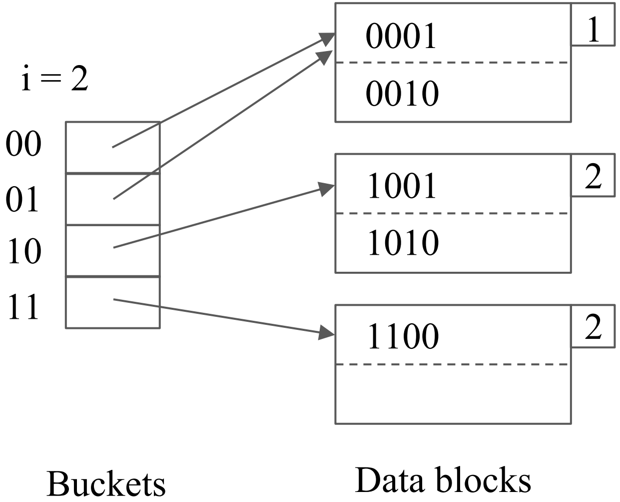
# Announcements

- Technology presentation group and schedule announced
    - Presentation schedule on course website
    - 7~8min per person (25 min for teams of 3, 35min for teams of 4, 40min for teams of 5)
    - Detailed instructions in Assignment 4, 5

- Assignment 2 (proposal draft) due this Wednesday

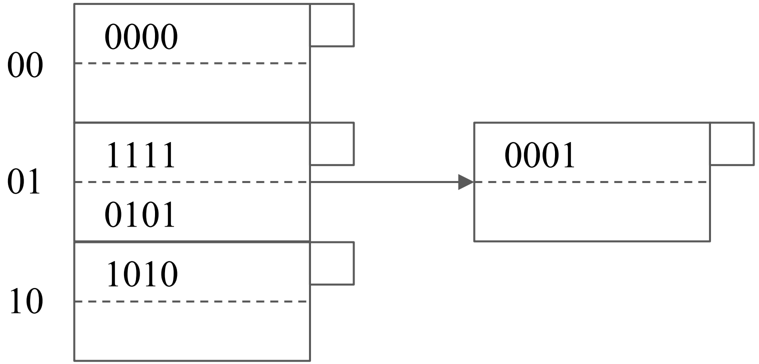
# Recap

- Static hash table
  - Linear probing hashing
  - Cuckoo hashing
- Dynamic hash table
  - Chained hashing
  - Extensible hashing
  - Linear hashing



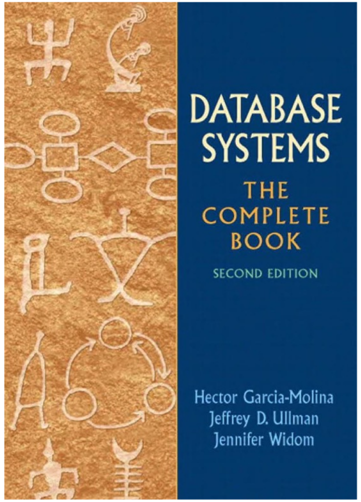
# bits used i = 2
# buckets n = 3
# records r = 5

Policy: limit r ≤ 1.7n



# Reading Materials

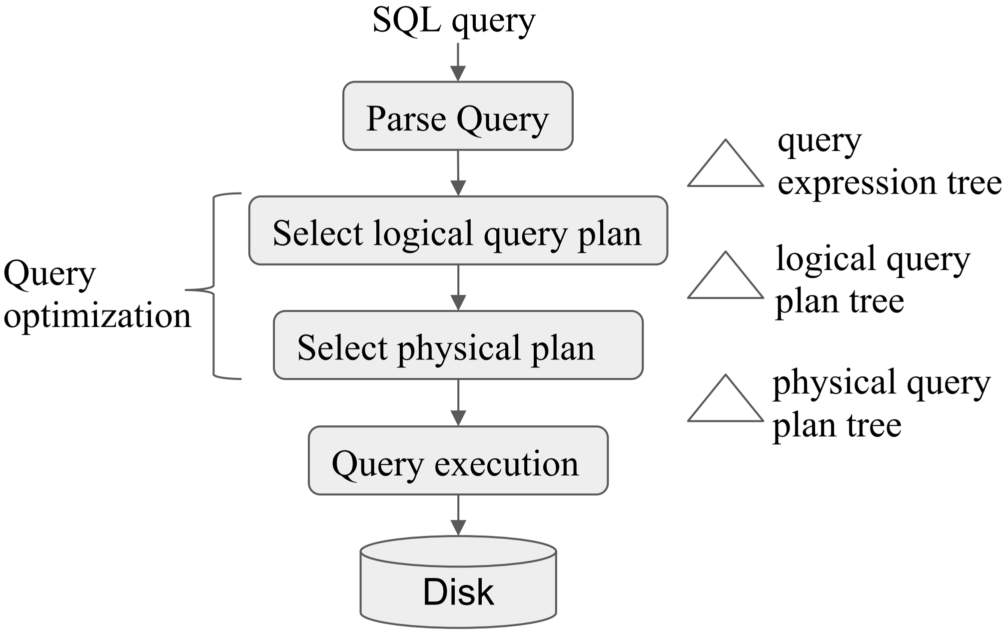
- Query execution (Chapters 15.1 - 15.6)
    - Physical operators
    - Implementing operators and estimating costs
- Query optimization (Chapters 16.1 - 16.5)
    - Parsing
    - Algebraic laws
    - Parse tree -> logical query plan
    - Estimating result sizes
    - Cost-based optimization



Acknowledgement: The following slides have been adapted from EE477 (Database and Big Data Systems) taught by Steven Whang.

# Query processor

- Group of components of a DBMS that turns user queries and data-modification commands into a sequence of database operations and executes them

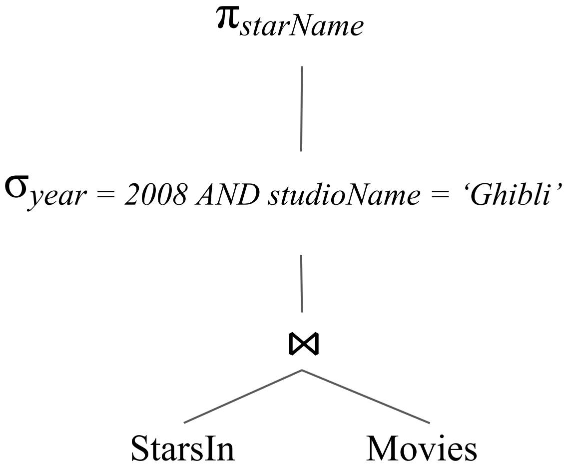


# Parse query

- SQL to relational algebra expression tree (= logical query plan)

```
StarsIn(title, year, starName)
Movies(title, length, genre, studioName, producer#)
```

```
SELECT starName
FROM StarsIn X, Movies Y
WHERE X.title = Y.title
AND studioName = ‘Ghibli’
AND year = 2008;
```



$\pi_{starName}$

$\sigma_{year\ =\ 2008\ AND\ studioName\ =\ \text{‘Ghibli’}}$

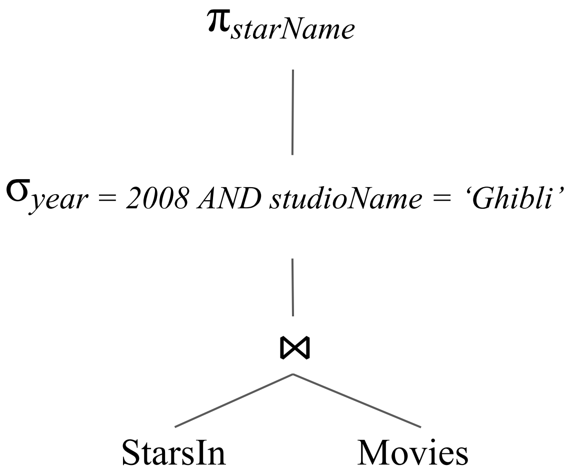
$\bowtie$

StarsIn Movies

# Select logical query plan

- Rewrite to equivalent expression that is expected to require less time to execute

```
StarsIn(title, year, starName)
Movies(title, length, genre, studioName, producer#)
```



$\pi_{starName}$

$\sigma_{year\ =\ 2008\ AND\ studioName\ =\ \text{‘Ghibli’}}$

$\bowtie$

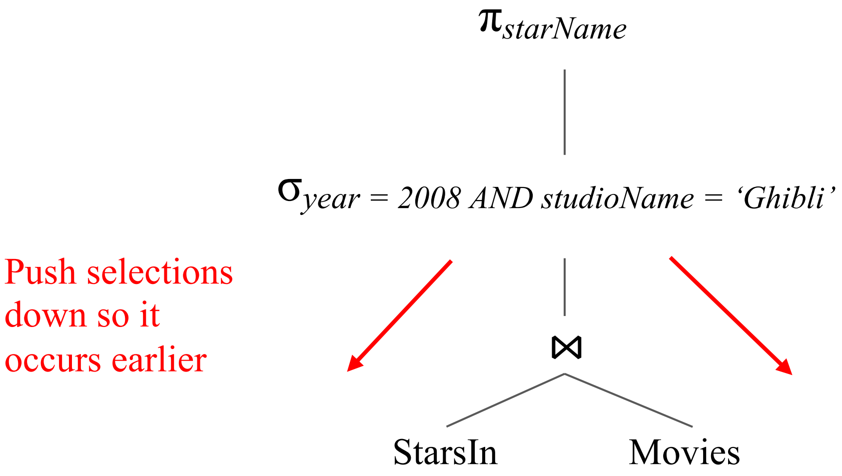
StarsIn Movies

Q: How could we rewrite this query to make it run faster?

# Select logical query plan

- Rewrite to equivalent expression that is expected to require less time to execute

```
StarsIn(title, year, starName)
Movies(title, length, genre, studioName, producer#)
```



$\pi_{starName}$

$\sigma_{year\ =\ 2008\ AND\ studioName\ =\ \text{‘Ghibli’}}$

$\bowtie$

StarsIn

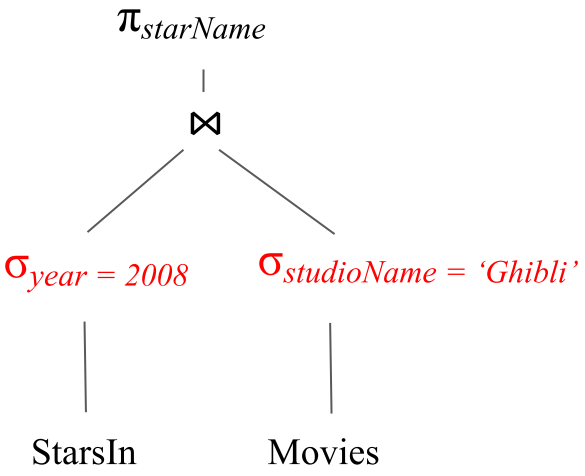
Movies

Push selections down so it occurs earlier

# Select logical query plan

- Rewrite to equivalent expression that is expected to require less time to execute

```
StarsIn(title, year, starName)
Movies(title, length, genre, studioName, producer#)
```



$\pi_{starName}$

$\bowtie$

$\sigma_{year = 2008}$ $\sigma_{studioName = 'Ghibli'}$

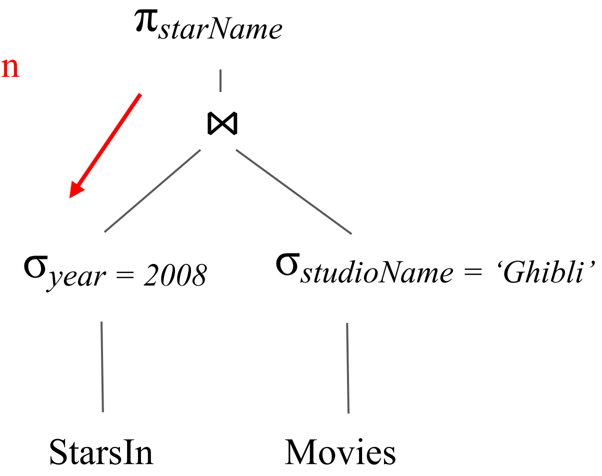
StarsIn Movies

Push selections down so it occurs earlier

# Select logical query plan

- Rewrite to equivalent expression that is expected to require less time to execute

```
StarsIn(title, year, starName)
Movies(title, length, genre, studioName, producer#)
```



Push projection down so it occurs earlier

$\pi_{starName}$

$\bowtie$

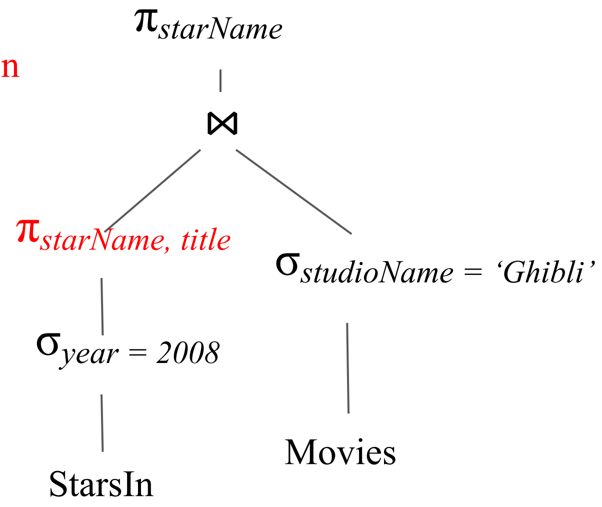
$\sigma_{year\ =\ 2008}$ $\sigma_{studioName\ =\ \text{'Ghibli'}}$

StarsIn Movies

# Select logical query plan

- Rewrite to equivalent expression that is expected to require less time to execute

```
StarsIn(title, year, starName)
Movies(title, length, genre, studioName, producer#)
```



$\pi_{starName}$

$\bowtie$

$\pi_{starName,\ title}$

$\sigma_{studioName\ =\ \text{‘Ghibli’}}$

$\sigma_{year\ =\ 2008}$

StarsIn

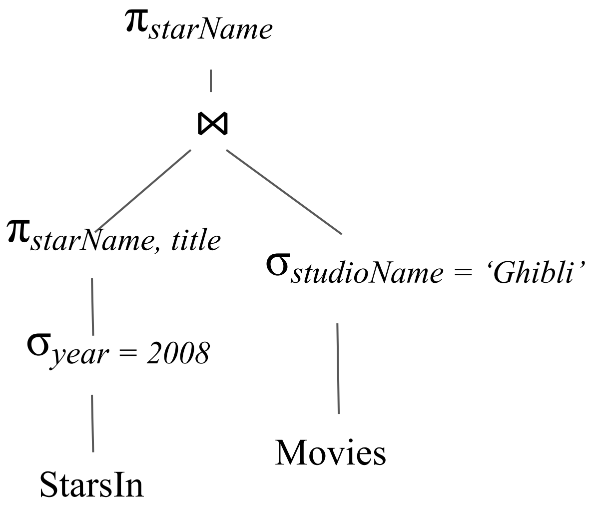
Movies

Push projection down so it occurs earlier

# Select logical query plan

- Rewrite to equivalent expression that is expected to require less time to execute

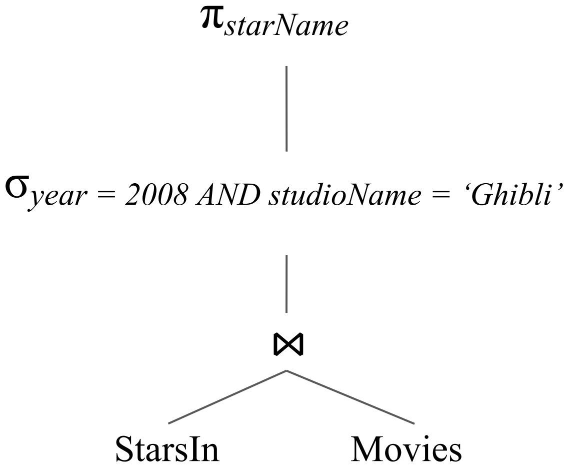
```
StarsIn(title, year, starName)
Movies(title, length, genre, studioName, producer#)
```



There can be many possible logical plans

# Select physical query plan

- A logical query plan is turned into a physical query plan
  - Algorithm for each operator
  - Order of execution
  - How to access relations



$\pi_{starName}$

$\sigma_{year\ =\ 2008\ AND\ studioName\ =\ \text{‘Ghibli’}}$

$\bowtie$

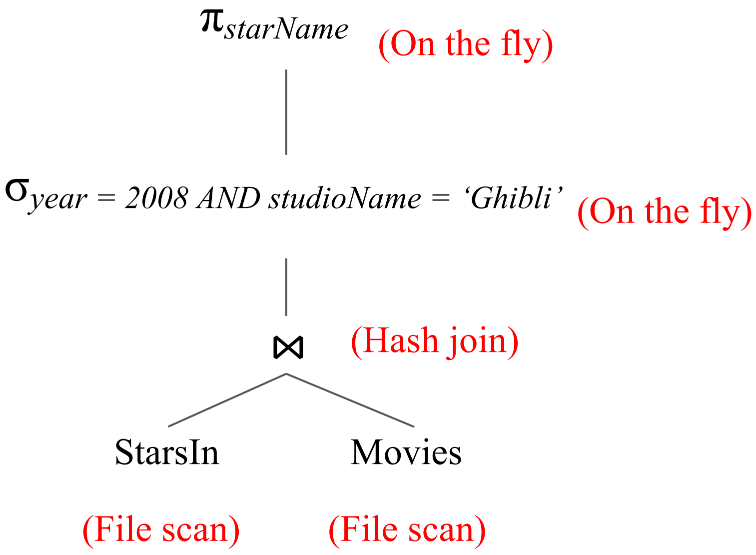
StarsIn

Movies

# Select physical query plan

- A logical query plan is turned into a physical query plan
    - Algorithm for each operator
    - Order of execution
    - How to access relations

Physical query plan 1



$\pi_{starName}$ (On the fly)

$\sigma_{year\ =\ 2008\ AND\ studioName\ =\ 'Ghibli'}$ (On the fly)

$\bowtie$ (Hash join)

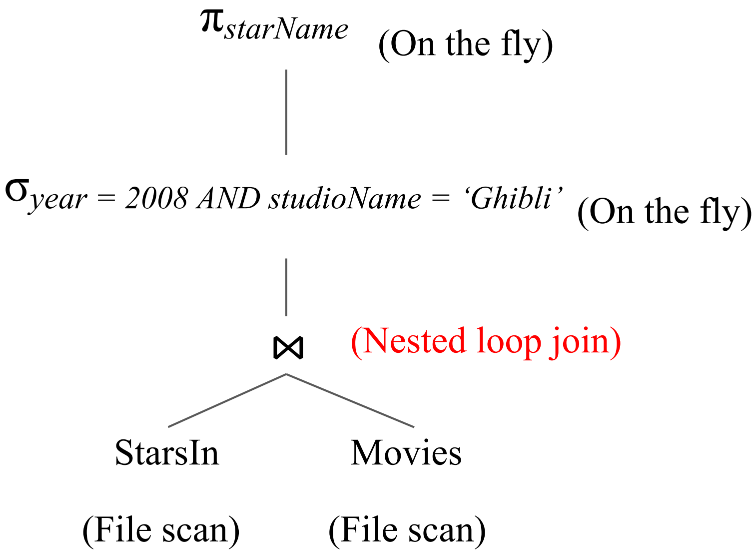
StarsIn (File scan)

Movies (File scan)

# Select physical query plan

- A logical query plan is turned into a physical query plan
    - Algorithm for each operator
    - Order of execution
    - How to access relations

Physical query plan 2



$\pi_{starName}$ (On the fly)

$\sigma_{year\ =\ 2008\ AND\ studioName\ =\ \text{‘Ghibli’}}$ (On the fly)

$\bowtie$ (Nested loop join)

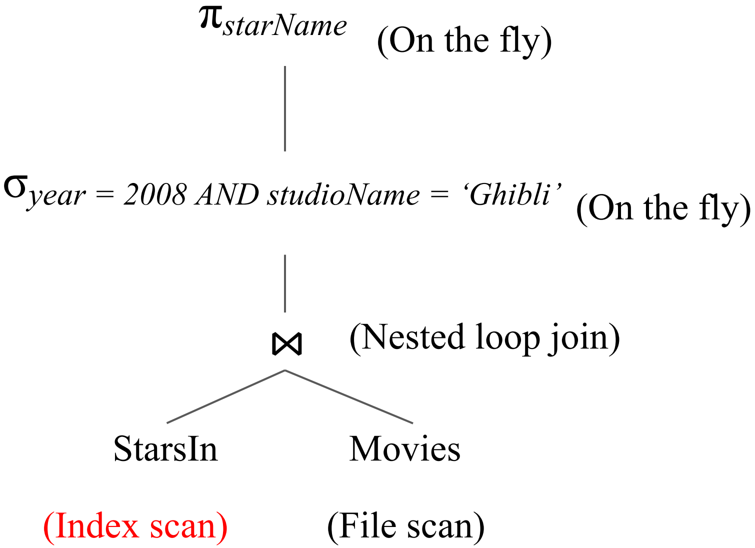
StarsIn (File scan)

Movies (File scan)

# Select physical query plan

- A logical query plan is turned into a physical query plan
  - Algorithm for each operator
  - Order of execution
  - How to access relations

Physical query plan 3



$\pi_{starName}$ (On the fly)

$\sigma_{year\ =\ 2008\ AND\ studioName\ =\ \text{‘Ghibli’}}$ (On the fly)

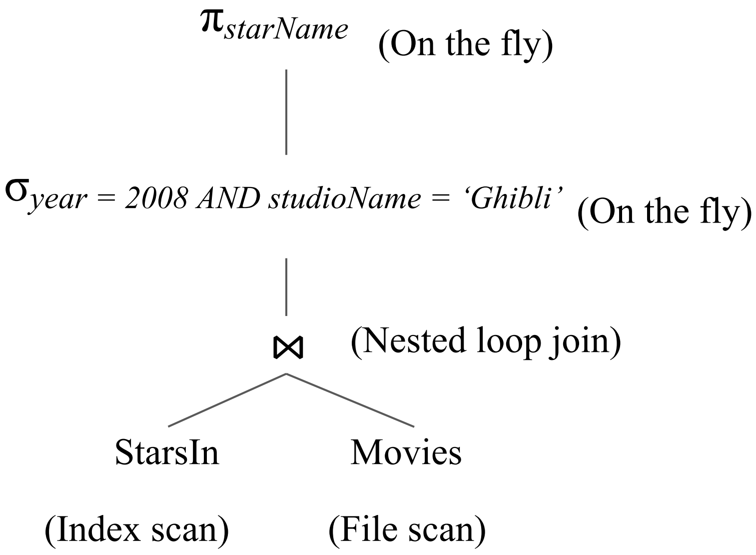
$\bowtie$ (Nested loop join)

StarsIn (Index scan)

Movies (File scan)

# Select physical query plan

- A logical query plan is turned into a physical query plan
    - Algorithm for each operator
    - Order of execution
    - How to access relations



$\pi_{starName}$ (On the fly)

$\sigma_{year\ =\ 2008\ AND\ studioName\ =\ \text{'Ghibli'}}$ (On the fly)

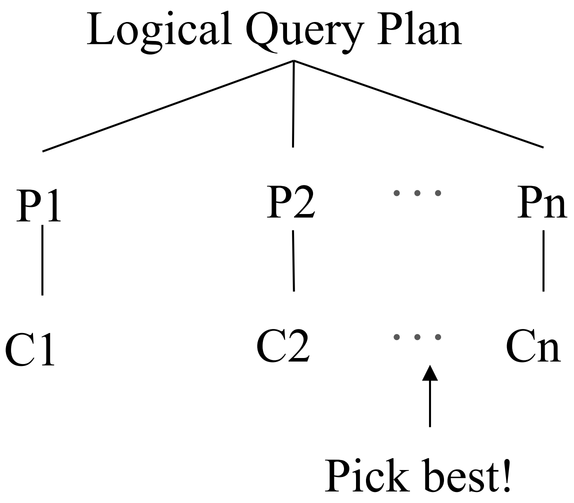
$\bowtie$ (Nested loop join)

StarsIn (Index scan)

Movies (File scan)

In general, there can be many possible physical plans

# Select physical query plan



Logical Query Plan

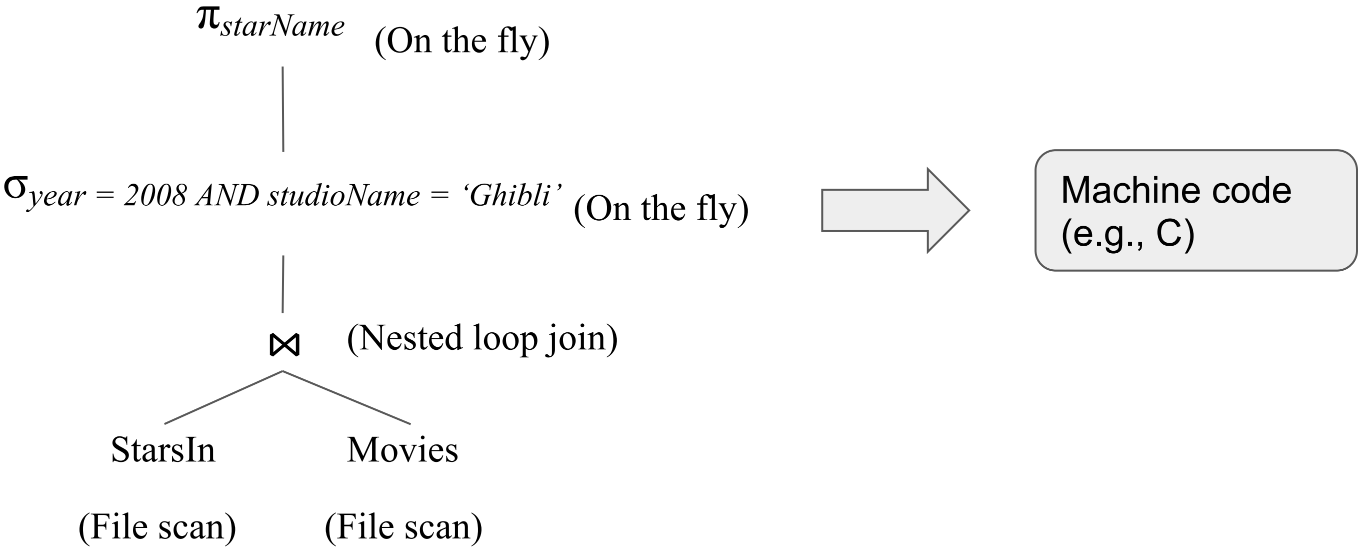
P1 P2 . . . Pn

C1 C2 . . . Cn

Pick best!

# Query execution

- The best physical plan is translated to actual machine code



$\pi_{starName}$ (On the fly)

$\sigma_{year\ =\ 2008\ AND\ studioName\ =\ \text{‘Ghibli’}}$ (On the fly)

$\bowtie$ (Nested loop join)

StarsIn (File scan)

Movies (File scan)

Machine code
(e.g., C)

# Overview summary

- Logical plan
  - An SQL query is parsed into a logical plan
  - The logical plan can be rewritten to multiple equivalent ones
  - See textbook 16.2 for laws for transforming logical plans
- Physical plan
  - A logical query plan with physical implementation details
  - Each logical plan can have multiple possible physical plans
- Query optimization
  - Find the optimal logical and physical plans

Focus of this lecture

# Estimating the cost of a physical query plan

- Estimate the size of results
  - Projection
  - Selection
  - Joins
- Estimate the # of disk I/O’s

# Size parameters

- $B(R)$: # blocks to hold tuples in $R$
- $T(R)$: # tuples in $R$
- $V(R, a)$: # distinct values of attribute $a$ in $R$

# Size parameters

- Example

$R$

| $A$ | $B$ | $C$ |
|---|---|---|
| `cat` | `1` | `2000` |
| `cat` | `1` | `2001` |
| `dog` | `1` | `2002` |

$A$: 10 byte string
$B$: 4 byte integer
$C$: 8 byte date

$T(R) = 3$
$V(R, A) = 2$
$V(R, B) = 1$
$V(R, C) = 3$
$B(R) = 1$ (if 3 tuples fit in one block)

# Estimating size of projection

- Example

*R*

| *A* | *B* | *C* |
|---|---|---|
| `cat` | `1` | `2000` |
| `cat` | `1` | `2001` |
| `dog` | `1` | `2002` |
| `...` | | |

*A*: 10 byte string
*B*: 4 byte integer
*C*: 8 byte date

Suppose each block is 100 bytes
Then a block fits 4 tuples
If $T(R) = 1000$
Then $B(R) = 1000 / 4 = 250$

# Estimating size of projection

- Example

$R$

| $A$ | $B$ | $C$ |
|---|---|---|
| `cat` | `1` | `2000` |
| `cat` | `1` | `2001` |
| `dog` | `1` | `2002` |
| `...` | | |

$A$: 10 byte string
$B$: 4 byte integer
$C$: 8 byte date

Suppose each block is 100 bytes
Then a block fits 4 tuples
If $T(R) = 1000$
Then $B(R) = 1000 / 4 = 250$

For $\pi_A(R)$, each block fits 10 tuples, so
$B(R) = 1000 / 10 = 100$

# Estimating size of projection

- Example

*R*

| *A* | *B* | *C* |
|---|---|---|
| `cat` | `1` | `2000` |
| `cat` | `1` | `2001` |
| `dog` | `1` | `2002` |
| `...` | | |

*A*: 10 byte string
*B*: 4 byte integer
*C*: 8 byte date

Suppose each block is 100 bytes
Then a block fits 4 tuples
If $T(R) = 1000$
Then $B(R) = 1000 / \ 4 = 250$

For $\pi_A$(R), each block fits 10 tuples, so
$B(R) = 1000 / \ 10 = 100$

For $\pi_{A,B,C,B/100 \to X}$(R), each block fits 3 tuples

# Estimating size of selection

- A selection generally reduces the number of tuples

Selection

$$S = \sigma_{A=c}(R)$$

Estimated result size
(without any more information)

$$T(S) = T(R)/V(R, A)$$

Assumption: values in $A = c$ are uniformly distributed over possible $V(R, A)$ values

# Estimating size of selection

- A selection generally reduces the number of tuples

Selection

$$S = \sigma_{A<c}(R)$$

Estimated result size
(without any more information)

$$T(S) = T(R)/3$$

Assumption: queries involving inequalities tend to retrieve a small fraction of possible tuples

Example: postgres/src/include/utils/selfuncs.h

# Estimating size of selection

- If selection condition is AND of conditions, multiply all selectivity factors

$$S = \sigma_{A=10 \wedge B<20}(R)$$

$$T(R) = 10,000$$

$$V(R, A) = 50$$

$$T(S) = T(R)/(50 \times 3) = 67$$

# Estimating size of selection

- If selection condition is an OR of conditions, can assume independence of conditions

$$S = \sigma_{A=10 \vee B<20}(R)$$

$$T(R) = 10,000$$

$$V(R, A) = 50$$

$$T(S) = T(R)(1 - (1 - 1/50)(1 - 1/3)) = 3466$$

# Estimating size of join

- We study $R(X, Y) \bowtie S(Y, Z)$
- Two simplifying assumptions
    - Containment of value sets: if $V(R,Y) \leq V(S,Y)$, then every $Y$-value of $R$ is a $Y$-value of $S$
    - Preservation of value sets: $V(R \bowtie S, X) = V(R, X)$

# Estimating size of join

- We study $R(X,Y) \bowtie S(Y,Z)$
- Two simplifying assumptions
    - Containment of value sets: if $V(R,Y) \leq V(S,Y)$, then every $Y$-value of $R$ is a $Y$-value of $S$
    - Preservation of value sets: $V(R \bowtie S, X) = V(R, X)$
- Case 1: $V(R, Y) \geq V(S, Y)$

$$T(R \bowtie S) = T(R)T(S)/V(R,Y)$$

*For each pair (r, s), we know that the Y-value of s is one of the Y-values of R by containment of value sets, so the probability of r having the same Y-value is 1/V(R,Y)*

# Estimating size of join

- We study $R(X, Y) \bowtie S(Y, Z)$
- Two simplifying assumptions
  - Containment of value sets: if $V(R,Y) \leq V(S,Y)$, then every $Y$-value of $R$ is a $Y$-value of $S$
  - Preservation of value sets: $V(R \bowtie S, X) = V(R, X)$
- Case 1: $V(R, Y) \geq V(S, Y)$

$$T(R \bowtie S) = T(R)T(S)/V(R,Y)$$

*For each pair (r, s), we know that the Y-value of s is one of the Y-values of R by containment of value sets, so the probability of r having the same Y-value is 1/V(R,Y)*

- Case 2: $V(R, Y) < V(S, Y)$

$$T(R \bowtie S) = T(R)T(S)/V(S,Y)$$

# Estimating size of join

- We study $R(X, Y) \bowtie S(Y, Z)$
- Two simplifying assumptions
  - Containment of value sets: if $V(R,Y) \leq V(S,Y)$, then every $Y$-value of $R$ is a $Y$-value of $S$
  - Preservation of value sets: $V(R \bowtie S, X) = V(R, X)$
- Case 1: $V(R, Y) \geq V(S, Y)$

$$T(R \bowtie S) = T(R)T(S)/V(R, Y)$$

*For each pair (r, s), we know that the Y-value of s is one of the Y-values of R by containment of value sets, so the probability of r having the same Y-value is 1/V(R,Y)*

- Case 2: $V(R, Y) < V(S, Y)$

$$T(R \bowtie S) = T(R)T(S)/V(S, Y)$$

- So in general, $T(R \bowtie S) = T(R)T(S)/\max(V(R, Y), V(S, Y))$

# Joins of many relations

- Compute intermediate $T$, $V$ results
- Example: consider $R \bowtie S \bowtie T$

$R(A, B)$

$S(B, C)$

$T(C, D)$

$T(R) = 1000$
$V(R, B) = 20$

$T(S) = 2000$
$V(S, B) = 50$
$V(S, C) = 100$

$T(T) = 5000$
$V(T, C) = 500$
$V(T, D) = 200$

Q: What is $T(R \bowtie S)$ and $V(R \bowtie S, C)$?

# Joins of many relations

- Compute intermediate $T$, $V$ results
- Example: consider $R \bowtie S \bowtie T$

$R \bowtie S\ (A, B, C)$

$T(C, D)$

$T(R \bowtie S) = 40000$
$V(R \bowtie S, C) = 100$

$T(T) = 5000$
$V(T, C) = 500$
$V(T, D) = 200$

# Joins of many relations

- Compute intermediate $T$, $V$ results
- Example: consider $R \bowtie S \bowtie T$

$$(R \bowtie S) \bowtie T$$

$$\begin{aligned} T((R \bowtie S) \bowtie T) &= 40000 \text{ x } 5000 \,/\, \max\{100, 500\} \\ &= 400000 \end{aligned}$$

# Joins of many relations

- Compute intermediate $T$, $V$ results
- Example: consider $R \bowtie S \bowtie T$

$$R \bowtie (S \bowtie T)$$

$$\begin{aligned} T(R \bowtie (S \bowtie T)) &= 1000 \text{ x } (2000 \text{ x } 5000 / \max\{100, 500\}) / \max\{20, 50\} \\ &= 400000 \end{aligned}$$

# Joins of many relations

- Compute intermediate $T$, $V$ results
- Example: consider $R \bowtie S \bowtie T$

$$R \bowtie (S \bowtie T)$$

$$\begin{aligned} T(R \bowtie (S \bowtie T)) &= 1000 \text{ x } (2000 \text{ x } 5000 / \max\{100, 500\}) / \max\{20, 50\} \\ &= 400000 \end{aligned}$$

- Assuming containment and preservation of value sets, the estimated result size is the same regardless of how we group and order the terms in a natural join of relations

# Natural joins with multiple join attributes

- Same as $R \bowtie S$ with single join attribute, but divide by $\max\{V(R, A), V(S, A)\}$ for each joining attribute A

$R(A, B, C)$

$T(R) = 1000$
$V(R, B) = 20$
$V(R, C) = 100$

$S(B, C, D)$

$T(S) = 2000$
$V(S, B) = 50$
$V(S, C) = 50$

$R \bowtie S$

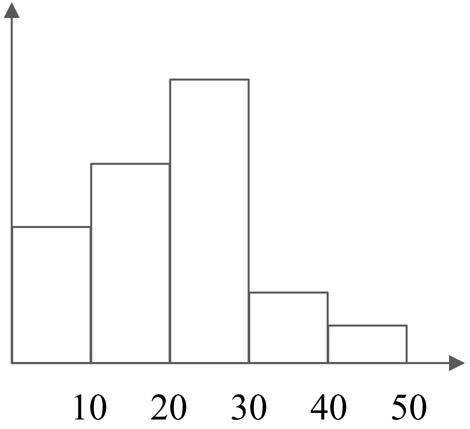
$$T(R \bowtie S) = 1000 \text{ x } 2000 \; / \max\{20, 50\} \; / \max\{100, 50\} = 400$$

# Using similar ideas, can estimate sizes of

- Union, intersect, difference, duplicate elimination, grouping [16.4.7]

# Obtaining estimates for size parameters

- Scan entire relation *R* to obtain *T*(*R*), *V*(*R*, *A*), and *B*(*R*)
- A DBMS may also compute histograms per attribute for more accurate estimations
    - e.g., equal-width histogram



$$\sigma_{A=22}(R) = \ ?$$

# Computation of statistics

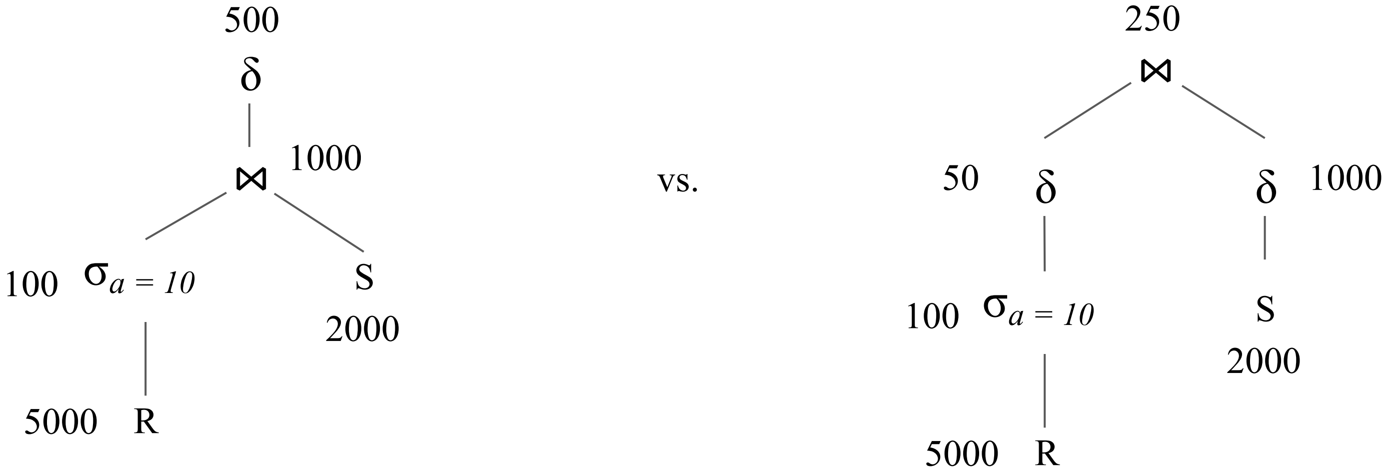
- Computed periodically or by request
- Sampling used to compute approximate statistics quickly

Example:

- `ANALYZE` command in Postgres
- See also: https://www.postgresql.org/docs/current/planner-stats.html

# Comparing logical query plan cost

- Cost estimates (sum of intermediate results) can be used to compare costs before and after transformations

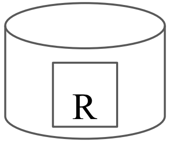


# Estimating the cost of a physical query plan

- Estimate the size of results
- Estimate the # of disk I/O’s
  - Scanning-based methods
  - Hash-based methods
  - Index-based methods

# Table scan

- Read entire contents of relation *R*
  - If table is clustered, requires *B*(*R)* I/O’s
  - If table is distributed among tuples among other relations, may require *T*(*R*) I/O’s



# Tuple-based Nested-loop Join

```
For each tuple t1 in R
    For each tuple t2 in S
        If t1.a == t2.a
            Join(t1, t2)
```

- $T(R) = 10{,}000$, $T(S) = 5{,}000$
- Suppose relations are not clustered
- Required memory $M \geq 2$

For each tuple in *R*, read all *S* blocks and join:

Total cost of $R \bowtie S$: $\overbrace{10000}^{\text{Outer Loop}} \text{ x } (1+ \overbrace{5000}^{\text{Read all } S \text{ tuples (inner loop)}}) = 50{,}010{,}000$ I/O’s

I/O: T(R) + T(R)T(S)

Memory Usage: 2 blocks

# Block-based Nested-loop Join

- $T(R) = 10{,}000$, $T(S) = 5{,}000$
- Required memory $M \geq 2$
- Suppose 10 records fit in one block:
  - B(R) = 1000, B(S) = 500

```
For each block b1 in R
    For each block b2 in S
        For each tuple t1 in b1
            For each tuple t2 in b2:
                If t1.a == t2.a
                    Join(t1, t2)
```

Outer Loop

Read all $S$ tuples (inner loop)

Total cost of $R \bowtie S$: 1000 x (1+ 500) = 501,000 I/O's

I/O: B(R) + B(R)B(S)

Memory Usage: 2 blocks

# Block-based Nested-loop Join

- $T(R) = 10{,}000$, $T(S) = 5{,}000$
- Suppose 10 records fit in one block:
  - B(R) = 1000, B(S) = 500
- Reverse join order

```
For each blocks s in S
    For each block r in R
        For each tuple t1 in s
            For each tuple t2 in r:
                If t1.a == t2.a
                  Join(t1, t2)
```

Outer Loop                Read all *R* tuples (inner loop)

Total cost of $R \bowtie S$: 500 x (1+ 1000) = 500,500 I/O’s

I/O: B(S) + B(S)B(R)

Memory Usage: 2 blocks

# Block-based Nested-loop Join

- $T(R) = 10{,}000$, $T(S) = 5{,}000$
- Suppose 10 records fit in one block:
  - B(R) = 1000, B(S) = 500
- Reverse join order
- Extra memory M=101: read 100 blocks of S at a time

```
For each M-1 blocks s in S
    For each block r in R
        For each tuple t1 in s
            For each tuple t2 in r:
                If t1.a == t2.a
                    Join(t1, t2)
```

Outer Loop

Read all *R* tuples

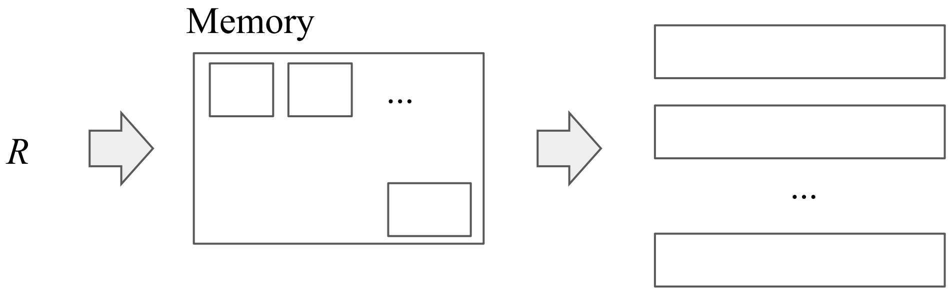
Total cost of $S \bowtie R$: 500/100 x (100+1000) = 5500 I/O’s

I/O: B(S) + B(S)B(R) / (M-1)

Memory Usage: M blocks

# Hash join

- Scan the smaller table, S, and build a hash table in memory. The hash table maps each distinct value of the join attribute to a list of tuples that have that attribute value.
- Scan R sequentially. For each tuple s in R, check the hash table to see if S has any tuples which have the same value of the join attribute.
- Join each tuple in S with any tuples in R which have the same join attribute.



# Hash join

- B(R) = 1000, B(S) = 500
- Total cost of $S \bowtie R$: $\underbrace{500}_{\text{Read all of S (step 1)}} + \underbrace{1000}_{\text{Read all of T (step 2)}} = 1{,}500$ I/O's

- Analysis of Hash join
  - Required memory: $B(S)$, assuming S is the smaller relation
  - Two pass algorithms require $\sqrt{B(S)}$
  - # Disk I/Os: $B(R) + B(S)$

# Index join

- Suppose $S$ has an index on the join attribute $Y$
  - The index is "clustering" if tuples with the same $Y$ value are clustered
- If $R$ is clustered, read $B(R)$ blocks to get all $R$ tuples
- For each tuple of $R$,
  - If $S$'s index is not clustering, read $T(S) / V(S, Y)$ blocks on average
  - If clustered, read $B(S) / V(S, Y)$ blocks
- Total join cost: $B(R) + T(R)T(S) / V(S,Y)$ or

  $B(R) + T(R)(\max(1, B(S) / V(S,Y)))$

# Query Optimization Overview

**Output:** A good physical query plan

Basic cost-based query optimization algorithm

- Enumerate candidate query plans (logical and physical)
- Compute estimated cost of each plan (e.g., number of I/Os)
  - Without executing the plan!
- Choose plan with lowest cost

# The Three Parts of an Optimizer

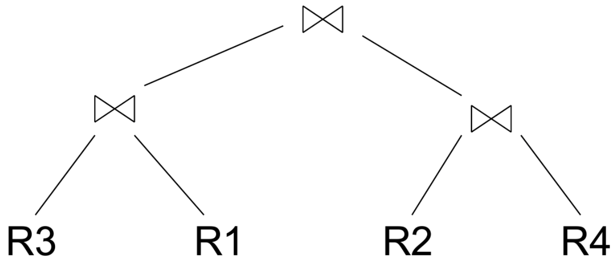
- Cost estimation
    - Estimate size of results
    - Also consider whether output is sorted/intermediate results written to disk etc.

- Search space
    - Algebraic laws, restricted types of join trees

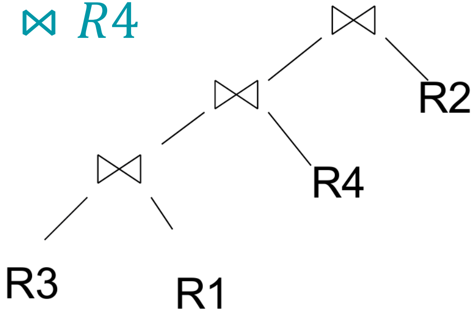
- Search algorithm
    - Example: Selinger algorithm

# Search Space

Query: $R1 \bowtie R2 \bowtie R3 \bowtie R4$



Bushy plan



Left-deep plan

Logical plan space:

- Several possible structures of the trees
- Each tree can have n! permutations of relations on leaves

Physical plan space:

- Different implementation (e.g., join algorithm) and scanning of intermediate operators for each logical plan

# Heuristic for pruning plan space

- Apply predicates as early as possible
- Avoid plans with cartesian products
  - $(R(A,B) \bowtie T(C,D)) \bowtie S(B,C)$
- Consider only left-deep join trees
  - Studied extensively in traditional query optimization literature
  - Works well with existing join algorithms such as nested-loop and hash join
    - e.g., might not need to write tuples to disk if enough memory