



Query Optimization

Databases, Aarhus University



Ira Assent

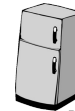
Intended learning outcomes

- ▶ Be able to
 - ▶ Describe and apply heuristics based and cost-based query optimization strategies

Review: query evaluation

SELECT
FROM
WHERE

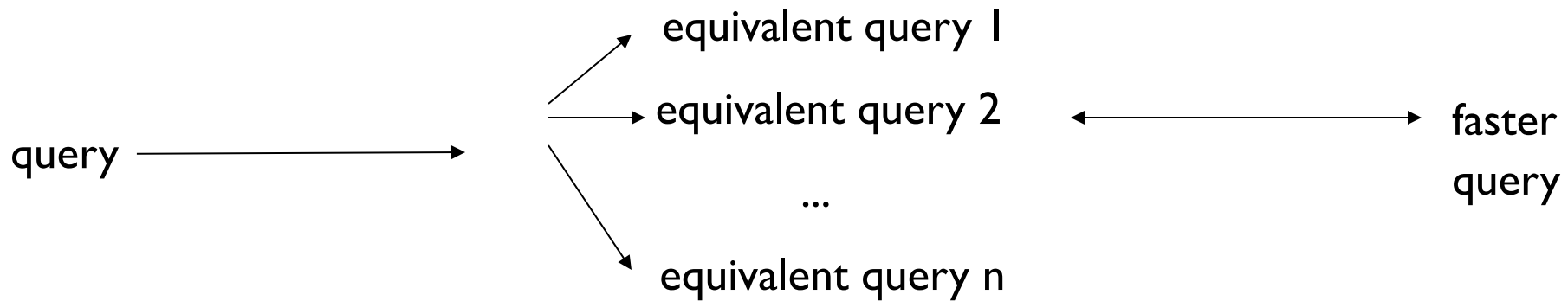
π \times σ



- ▶ SQL query translated to relational algebra, then optimized
 - ▶ queries are decomposed into query blocks
- ▶ Strategies work on a per block (not per record) basis
- ▶ Selection: linear search (brute force) and index-based search where applicable, for conjunction: use most selective first, use index pointers
- ▶ Join most time-consuming, different strategies
 - ▶ Nested loop join (brute force)
 - ▶ Index-based join
 - ▶ Sort-merge join
 - ▶ Hash join
- ▶ Projection: generally straightforward, removing duplicates expensive
- ▶ Cartesian product: try to avoid if possible
- ▶ Union, intersection, and difference: sort and scan or hash
- ▶ Pipeline the data through multiple operations to avoid materializing large intermediate results

Query Optimization

- ▶ Transform query into faster, equivalent query



- Heuristic (logical) optimization
 - Query tree (relational algebra) optimization
- Cost-based (physical) optimization

Using Heuristics in Query Optimization

- ▶ **Process for heuristics optimization**
 1. Parser of high-level query (SQL) generates initial internal representation (relational algebra)
 2. Apply heuristics rules to optimize the internal representation
 3. Generate query execution plan to execute groups of operations based on access paths available on files involved in query (e.g. index)
- ▶ **Main heuristic: apply first the operations that reduce the size of intermediate results**
 - ▶ E.g., Apply SELECT and PROJECT operations before applying the JOIN or other binary operations



Query trees

▶ Query tree

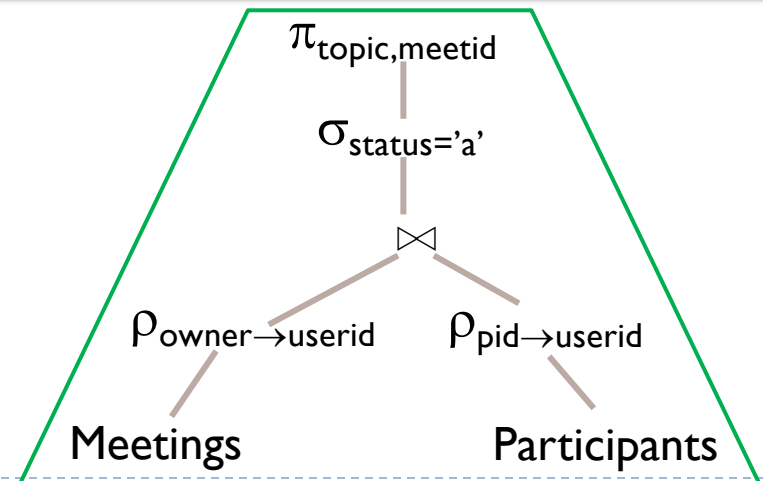
- ▶ tree data structure corresponding to relational algebra expression
- ▶ input relations as leaf nodes
- ▶ operations as internal nodes

▶ Example: in which meetings do the owners participate?


$$\pi_{\text{topic,meetid}} (\sigma_{\text{status}='a'} (\rho_{\text{owner} \rightarrow \text{userid}} (\text{Meetings}) \bowtie \rho_{\text{pid} \rightarrow \text{userid}} (\text{Participants})))$$

▶ Execution of the query tree

- ▶ Executing internal node operation whenever its operands are available
- ▶ Then replace that internal node by the relation that results from executing the operation



Canonical Query Tree



SQL query:

SELECT Name

FROM Customer CU, Purchased R,
Product P

WHERE P.Name = 'Cookie'

AND P.ProductID = R.ProductID

AND CU.CustomerID = R.CustomerID

AND CU.Street = 'Sesame'

Translate SQL query directly into tree

- leaves are the input tables
- then selection (WHERE clause) and
- projection (SELECT clause) are applied

Customer (CustomerID, Name,
Street, City)

Purchased (CustomerID, ProductID)

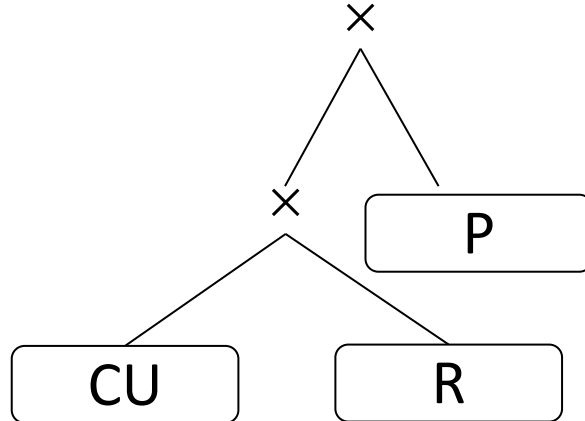
Product (ProductID, Name, CreationDate,
Price, Manufacturer, Category) ira@cs.au.dk

Canonical Query Tree



$$\pi_{\text{Name}}$$

$$\sigma_{\text{Name} = \text{'Cookie'} \wedge P.\text{ProductId} = R.\text{ProductID} \wedge$$

$$\text{CU.CustomerID} = R.\text{CustomerID} \wedge \text{CU.Street} = \text{'Sesame'}}$$


SQL query:

SELECT Name

FROM Customer CU, Purchased R,
Product P

WHERE P.Name = 'Cookie'

AND P.ProductId = R.ProductID

AND CU.CustomerID = R.CustomerID

AND CU.Street = 'Sesame'

Translate SQL query directly into tree

- leaves are the input tables
- then selection (WHERE clause) and
- projection (SELECT clause) are applied

Customer (CustomerID, Name,
Street, City)

Purchased (CustomerID, ProductID)

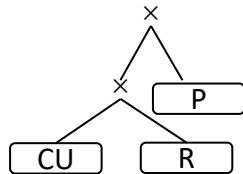
Product (ProductID, Name, CreationDate,
Price, Manufacturer, Category) ira@cs.au.dk

Apply Selections Early

π_{Name}

$\sigma_{Name = 'Cookie' \wedge P.ProductId = R.ProductID \wedge$

$CU.CustomerID = R.CustomerID \wedge CU.Street = 'Sesame'}$



π_{Name}

$\sigma_{P.ProductId = R.ProductID}$

\times

$\sigma_{CU.CustomerID = R.CustomerID}$

$\sigma_{Name = 'Cookie'}$

\times

P

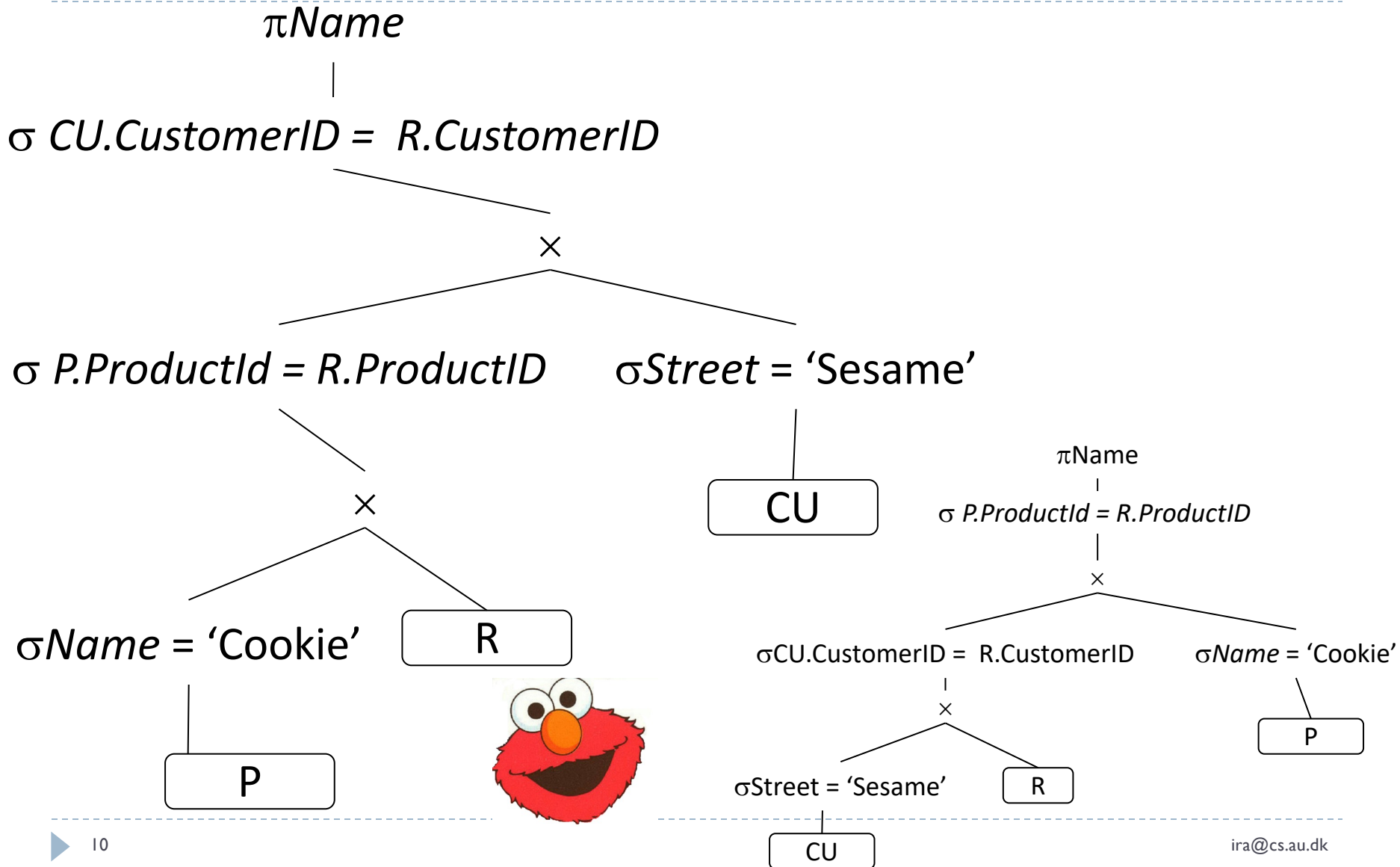
$\sigma_{Street = 'Sesame'}$

R

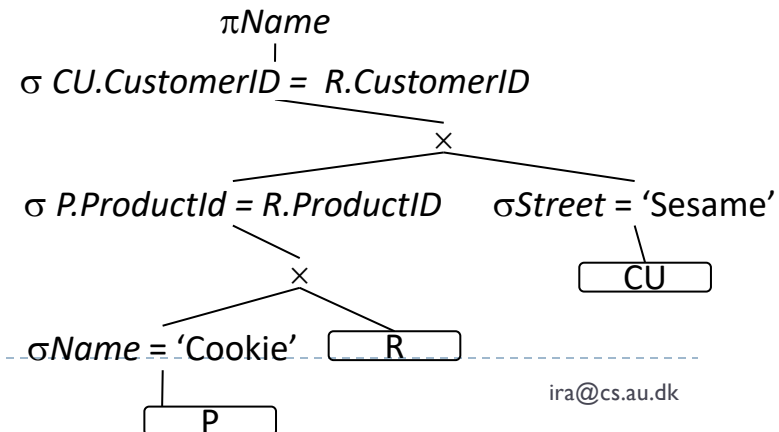
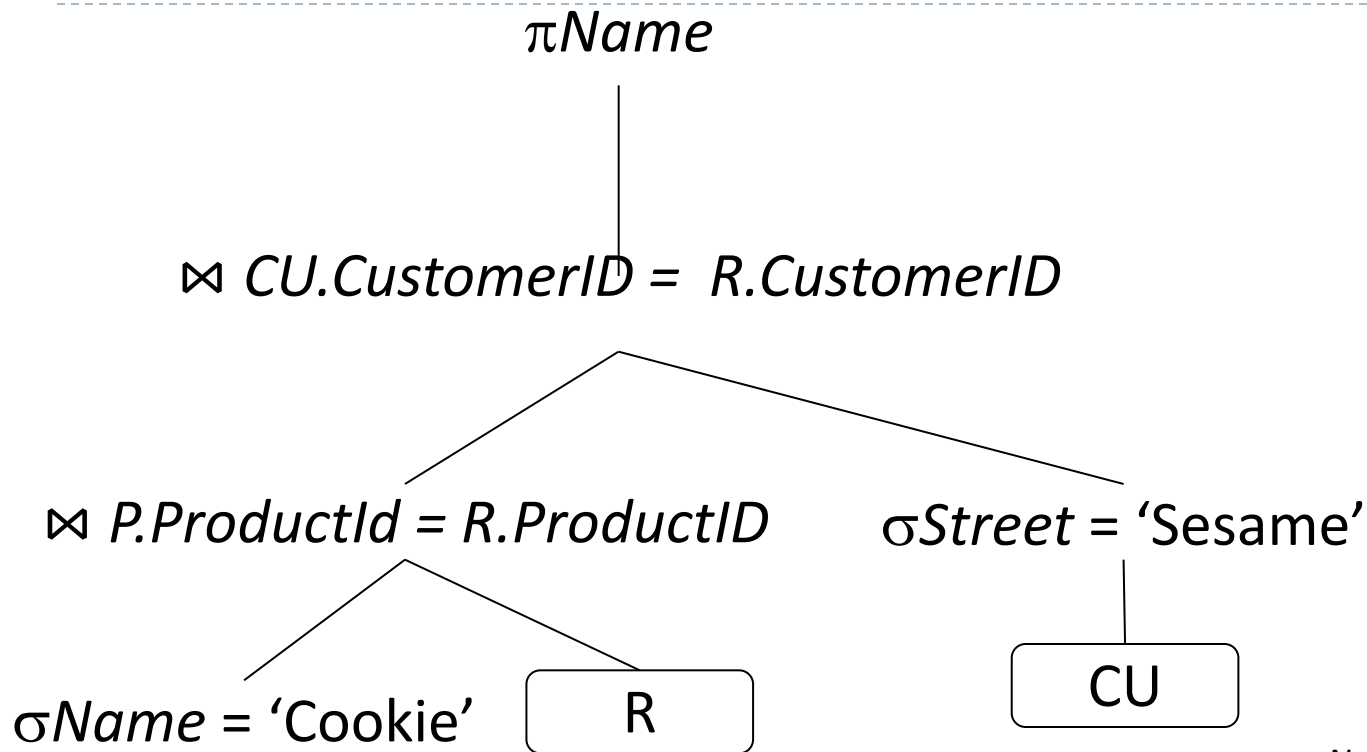
CU



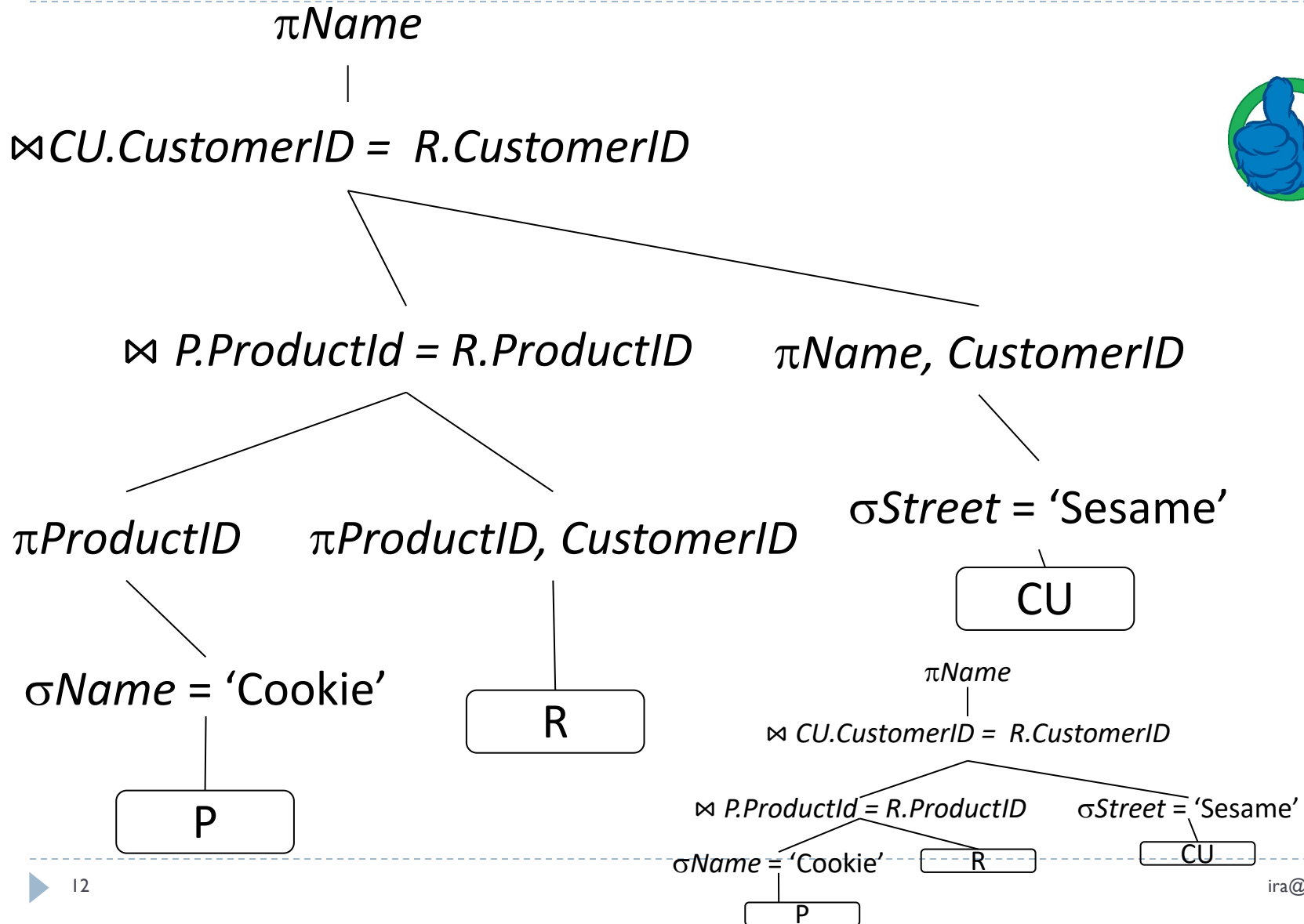
Apply More Restrictive Selections Early



Form Joins



Apply Projections Early



Algebraic Query Optimization

What is the most efficient way to

- join tables A, B, C,
- to select from the union of A and B

when A and B contain a million rows each, and C contains 10 rows?

1. $A \bowtie (B \bowtie C)$ $\sigma_C(A) \cup \sigma_C(B)$
Join B and C first, join result with A; select first individual relations, then union
2. $(A \bowtie B) \bowtie C$ $\sigma_C(A) \cup \sigma_C(B)$
Join A and B first, join result with C; select first individual relations, then union
3. $A \bowtie (B \bowtie C)$ $\sigma_C(A \cup B)$
Join B and C first, join result with A; union relations first, then select
4. $(A \bowtie B) \bowtie C$ $\sigma_C(A \cup B)$
Join A and B first, join result with C; union relations first, then select

Algebraic Query Optimization

What is the most efficient way to

- join tables A, B, C,
- to select from the union of A and B

when A and B contain a million rows each, and C contains 10 rows?

1. $A \bowtie (B \bowtie C) \quad \sigma_C(A) \cup \sigma_C(B)$

Join B and C first, join result with A;

select first individual relations, then union

- ▶ We want small intermediate results
- ▶ So, we do the smallest join first, which involves C
- ▶ Similarly, we first apply selection reduce the size of A and B prior to unioning them

1. $(A \bowtie B) \bowtie C \quad \sigma_C(A) \cup \sigma_C(B)$
individual relations, then union

Join A and B first, join result with C; select first

2. $A \bowtie (B \bowtie C) \quad \sigma_C(A \cup B)$
relations first, then select

Join B and C first, join result with A; union

3. $(A \bowtie B) \bowtie C \quad \sigma_C(A \cup B)$
union relations first, then select

Join A and B first, join result with C;

Outline of Heuristic Algebraic Optimization Algorithm

1. break up any select operations with conjunctive conditions into a cascade
2. move each select operation as far down the query tree as is permitted by the attributes involved in the select condition
3. rearrange the leaf nodes of the tree so that the leaf node relations with the most restrictive select operations are executed first
4. transform Cartesian product with subsequent select operation into join
5. break down and move lists of projection attributes down the tree as far as possible by creating new project operations as needed
6. Identify subtrees that represent groups of operations that can be executed by a single algorithm



General Transformation Rules

1. Cascade of σ : break up conjunctive selection into cascade (sequence) of σ :
 - ▶ $\sigma_{c1 \text{ AND } c2 \text{ AND } \dots \text{ AND } cn}(R) = \sigma_{c1} (\sigma_{c2} (\dots (\sigma_{cn}(R)) \dots))$
2. Commutativity of σ : $\sigma_{c1} (\sigma_{c2}(R)) = \sigma_{c2} (\sigma_{c1}(R))$
3. Cascade of π : In cascade of π operations, all but the last one can be ignored
 - ▶ $\pi_{List1} (\pi_{List2} (\dots (\pi_{Listn}(R)) \dots)) = \pi_{List1}(R)$
4. Commuting σ with π : If selection condition involves only attributes in projection list, the two operations can be commuted:
 - ▶ $\pi_{A1, A2, \dots, An} (\sigma_c (R)) = \sigma_c (\pi_{A1, A2, \dots, An} (R))$
5. Commutativity of \bowtie (and \times): $R \bowtie_c S = S \bowtie_c R$; $R \times S = S \times R$
6. Commuting σ with \bowtie (or \times): If all attributes in selection condition involve only attributes of one of the relations being joined—say, R —the two operations can be commuted: $\sigma_c (R \bowtie S) = (\sigma_c (R)) \bowtie S$
 - ▶ Or, if selection condition can be split up such that one involves only attributes of R and the other only attributes of S , the operations commute:
 - ▶ $\sigma_c (R \bowtie S) = (\sigma_{c1} (R)) \bowtie (\sigma_{c2} (S))$

More rules

7. Commuting π with \bowtie (or \times): let $L = \{A_1, \dots, A_n, B_1, \dots, B_m\}$, where A_1, \dots, A_n are attributes of R and B_1, \dots, B_m are attributes of S . If the join condition c involves only attributes in L , the two operations can be commuted:
 - ▶ $\pi_L (R \bowtie_C S) = (\pi_{A_1, \dots, A_n} (R)) \bowtie_C (\pi_{B_1, \dots, B_m} (S))$
 - ▶ If the join condition C contains additional attributes not in L , these must be added to the projection list, and a final π operation is needed
8. set operations \cup and \cap are commutative, but “ $-$ ” is not
9. Associativity of \bowtie , \times , \cup , and \cap : if θ stands for any one of these four operations (throughout the expression): $(R \theta S) \theta T = R \theta (S \theta T)$
10. Commuting σ with set operations: The σ operation commutes with \cup , \cap , and $-$. If θ stands for any one of these three operations, we have
 - ▶ $\sigma_c (R \theta S) = (\sigma_c (R)) \theta (\sigma_c (S))$
11. The π operation commutes with \cup : $\pi_L (R \cup S) = (\pi_L (R)) \cup (\pi_L (S))$
12. If the condition c of a σ that follows a \times corresponds to a join condition, convert: $(\sigma_c (R \times S)) = (R \bowtie_C S)$

Summary Heuristic Query Optimization

1. Generate initial query tree from SQL statement
2. Transform query tree into more efficient query tree, via a series of tree modifications, each of which hopefully reduces execution time





A single optimized tree is the result

- ▶ The main heuristic is to apply first the operations that reduce the size of intermediate results
 - ▶ Perform select operations as early as possible to reduce the number of tuples and perform project operations as early as possible to reduce the number of attributes
 - ▶ by moving select and project operations as far down the tree as possible
 - ▶ More restrictive select and join operations should be executed before other similar operations
 - ▶ by reordering the leaf nodes of the tree among themselves and adjusting the rest of the tree accordingly



Ignores the actual database!

Cost-based query optimization

-  Estimate and compare the costs of executing a query using different execution strategies and choose the strategy with the lowest cost estimate
-  Can “learn” from data stored in the database
 - ▶ E.g. how large are the relations actually, what is the selectivity of an operation?
 - ▶ Can differentiate between different operations with the same relational algebra operators
-  Need to consider a number of different strategies
-  Need to define cost function

Approach in Cost-Based Optimization

- ▶ Transformations to generate multiple candidate query trees from initial tree
- ▶ Statistics on the inputs to each operator needed
 - ▶ Statistics on leaf relations stored in system catalog
 - ▶ Statistics on intermediate relations must be estimated; most important relations' cardinalities
- ▶ Cost formulas estimate cost of executing each operation in each candidate query tree
 - ▶ Parameterized by statistics of the input relations
 - ▶ Also dependent on specific algorithm used by operator
 - ▶ Cost can be CPU time, I/O time, communication time, main memory usage, or a combination
- ▶ Query tree with least total cost executed

What could be relevant statistics for the cost?

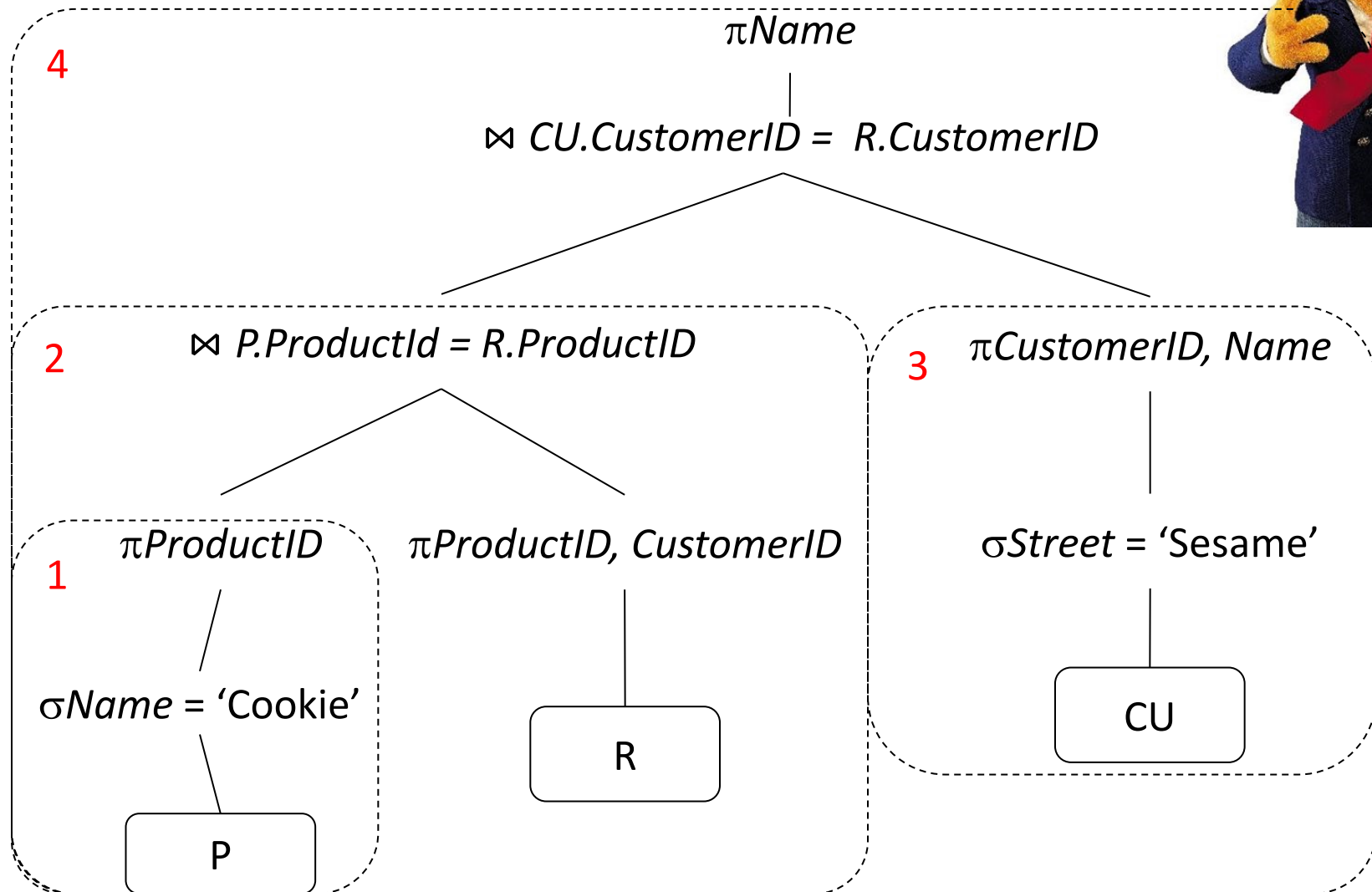
1. Number of indexing structures on an attribute
2. Number of distinct values of an attribute
3. Number of leaves in an index
4. Number of levels in an index

Statistics used as input to cost

- ▶ Catalog Information Used in Cost Functions
 - ▶ Information about the size of a file
 - ▶ number of records (tuples) (r),
 - ▶ record size (R),
 - ▶ number of blocks (b)
 - ▶ blocking factor (bfr): how many records per block
 - ▶ Information about indexes and indexing attributes of a file
 - ▶ Number of levels (x) of each multilevel index
 - ▶ i.e. how many levels to traverse before reaching data
 - ▶ Number of first-level index blocks (b_{l1})
 - ▶ Number of distinct values (d) of an attribute
 - ▶ Can be used to estimate the selectivity
 - ▶ Selectivity (sl) of an attribute
 - ▶ Selection cardinality (s) of an attribute ($s = sl * r$)

The cheat sheet
(brightspace) is
your friend

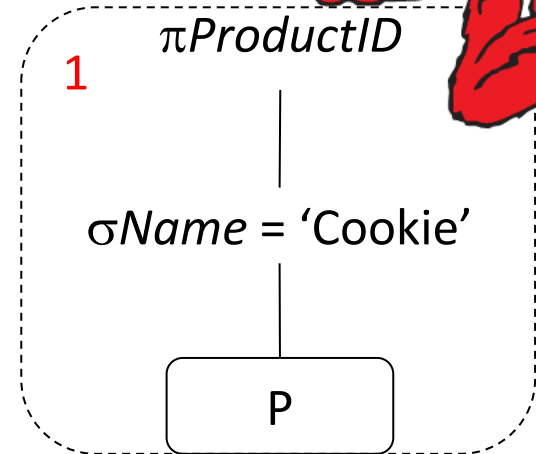
Cost Estimation Example



Operation 1: σ followed by a π



- ▶ Statistics
 - ▶ Relation statistics:
 - ▶ number of tuples in Product $r_{Product} = 5,000$
 - ▶ relation size in blocks in Film $b_{Product} = 50$
 - ▶ Attribute statistics:
 - ▶ selection cardinality in Name $s_{Name} = 1$
 - ▶ Index statistics:
 - ▶ Secondary Hash Index on *Name*
- ▶ Result relation size: 1 tuple
- ▶ Operation: use index with 'Cookie', then project on *ProductID*, leave result in main memory (1 block)
- ▶ Cost (in disk accesses): $C_1 = 1 + 1 = 2$
 - ▶ 1 block for hash index, 1 block for result tuple
 - ▶ (cost of using an index is the number of levels of the index plus the number of data blocks retrieved)



Operation 2: π followed by a \bowtie

▶ Statistics

▶ Relation statistics:

- ▶ number of tuples $r_{Purchased} = 40,000$
- ▶ relation size in blocks $b_{Purchased} = 2,000$

▶ Attribute statistics:

- ▶ selection cardinality $s_{ProductID} = 8$

▶ Index statistics:

- ▶ Secondary B+-Tree Index for *Purchased* on *ProductID* with 2 levels

▶ Result relation size: 8 tuples

▶ Operation: index join using B-Tree, then project on *CustomerID*, leave result in main memory (one block)

▶ Cost: $C_2 = 2 + 8 = 10$

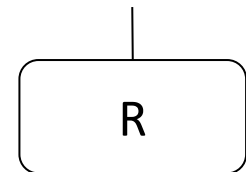
- ▶ Product is for “free”, because still in main memory, scan over it, look up R using B-tree, 2 blocks for that, plus 8 data blocks (secondary, so different blocks) to read result tuples

2

$\bowtie P.ProductID = R.ProductID$

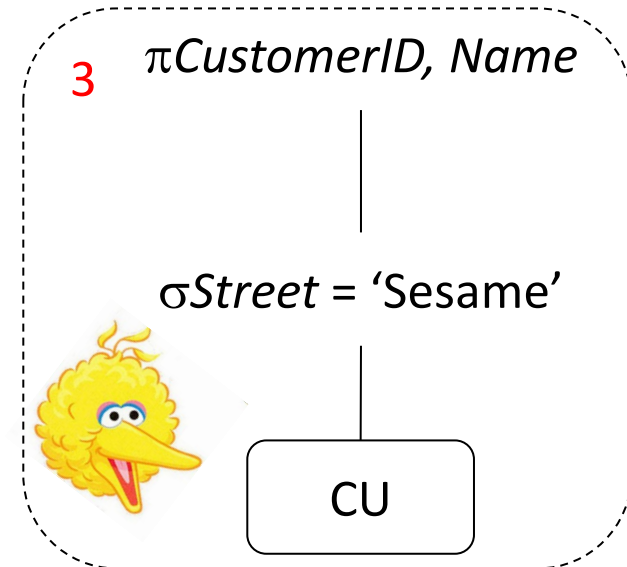


$\pi ProductID, CustomerID$



Operation 3: σ followed by a π

- ▶ Statistics
 - ▶ Relation statistics:
 - ▶ number of tuples $r_{Customer} = 200$
 - ▶ relation size in blocks $b_{Customer} = 10$
 - ▶ Attribute statistics:
 - ▶ selection cardinality $s_{Street} = 10$
- ▶ Result relation size: 10 tuples
- ▶ Operation: Linear search of *Customer*, leave result in main memory (one block)
- ▶ Cost: $C_3 = 10$
 - ▶ Linear scan: read all 10 blocks of Customer relation



Operation 4: ⋈ followed by a π

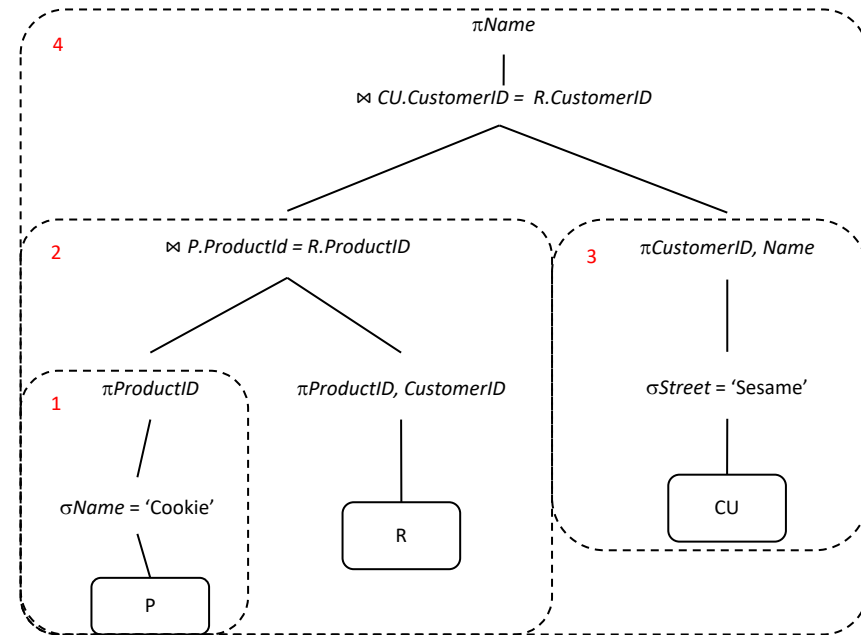
- Operation: Main memory join on relations in main memory

4 π_{Name}
|
 $\bowtie CU.CustomerID = R.CustomerID$

- Cost: $C_4 = 0$

Total cost:

$$C = \sum_{i=1}^4 C_i = 2 + 10 + 10 + 0 = 22$$



with all intermediate results still in main memory, no additional cost because no further disk accesses

Overall cost is the sum over all operations



Estimates

The size of what can we estimate most accurately for a given plan?

- A. The size of a join result
- B. The size of a selection
- C. The size of a cartesian product
- D. The size of an intersection

Textbook example cost-based optimization σ

- ▶ Consider the following query
SELECT * FROM Employee WHERE Dno=5 AND Salary>30000 AND Sex='F';
- ▶ Translate into **relational algebra** representation:
 - ▶ $\sigma_{Dno=5 \text{ AND } Salary>30000 \text{ AND } Sex='F'}(Employee)$
- ▶ We start out by checking **available statistics**
 - ▶ Employee file
 - ▶ $r_E = 10000$ records with blocking factor (number of records per block)
 $bfr_E=5$, so $b_E=2000$ blocks
 - ▶ selection cardinality for Salary of 20 on average ($s_{Salary}=20$)
 - ▶ Selection cardinality key Ssn $s_{Ssn}=1$
 - ▶ Selection cardinality for Dno $s_{Dno}=80$
 - ▶ Selection cardinality for Sex $s_{Sex}=5000$
 - ▶ Note: often such cardinality estimates are based on the number of distinct values (NDV) which is stored in system catalog:
 - ▶ e.g. $NDV(Employee, Sex)=2$, so $s_{Sex}=10000/2=5000$

Textbook example calculations

$\sigma_{Dno=5 \text{ AND } Salary > 30000 \text{ AND } Sex='F'}(\text{Employee})$

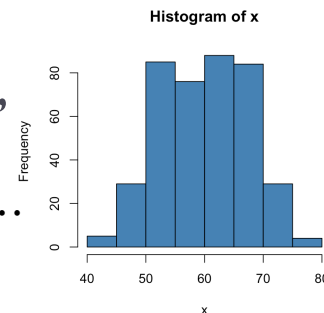
- ▶ Next consider **access paths**: denote with x the number of levels in index
 - ▶ Clustering index on Salary with 3 levels: $x_{Salary}=3$
 - ▶ Secondary index on key Ssn: $x_{Ssn}=4$
 - ▶ Secondary index on Dno: $x_{Dno}=2$
 - ▶ Secondary index on Sex: $x_{Sex}=1$
- ▶ Next **compute the cost** of different evaluation options
 - ▶ Linear scan: reads all 2000 blocks
 - ▶ Check different access paths for each of the conjunctive conditions
 - ▶ Expected selection cardinality of each of the conditions
 - ▶ Dno=5: $x_{Dno}=2$: 2 levels index, selection cardinality $s_{Dno}=80$, so estimate 82 (2 levels to reach data, then read 80 data blocks)
 - ▶ Salary<30000, assume half of records meet condition, clustering index: $x_{Salary}=3$: 3 levels index+ $2000/2=1003$
 - ▶ Sex=2, so $10000/2=5000$ $x_{Sex}+s_{Sex}=\underline{5001}$ (worse than brute force)

Textbook example choice of plan

- ▶ Linear scan: reads all 2000 blocks
- ▶ Check different access paths for each of the conjunctive conditions
 - Dno=5: 82
 - Salary>30000: 1003
 - Sex=2: 5001

$\sigma_{Dno=5 \text{ AND } Salary>30000 \text{ AND } Sex='F'}(Employee)$

- ▶ Pick cheapest option: Dno=5
 - ▶ Once the data from that condition is loaded into main memory, check other conditions there: no additional I/O
 - ▶ So, total cost estimate is 82
 - ▶ Here, we do not assume additional information, but in practice, histograms on value distribution also used
 - ▶ E.g. Sex: M=6500, F=3500, or Dno: 1=200, 2=800, 3=25, 4=10, 5=2000, ...
 - Would give potentially very different estimates on selectivity of Sex='F' and Dno=5
 - Would mean in this case that linear scan becomes cheapest estimate



Textbook example cost-based estimate of join

- ▶ **SELECT * FROM DEPARTMENT JOIN EMPLOYEE ON Dnumber=Dno;**
- ▶ **DEPARTMENT $\bowtie_{Dnumber=Dno}$ EMPLOYEE;**
- ▶ **Statistics and access paths for Department file:**
 - ▶ Department $r_D=125$ records in $b_D=13$ blocks
 - ▶ Primary index on Dnumber $x_{Dnumber}=1$
 - ▶ Secondary index on Mgr_ssn $s_{Mgr_ssn}=1$, $x_{Mgr_ssn}=2$
 - ▶ Join selectivity $1/125$ because Dnumber is key
 - ▶ Assume $bfr_{ED}=4$ for joint result relation

- ▶ **Employee file**
 - ▶ $r_E = 10000$ records with blocking factor (number of records per block) $bfr_E=5$, so $b_E=2000$ blocks
 - ▶ selection cardinality for Salary of 20 on average ($s_{Salary}=20$)
 - ▶ Selection cardinality key Ssn $s_{Ssn}=1$
 - ▶ Selection cardinality for Dno $s_{Dno}=80$
 - ▶ Selection cardinality for Sex $s_{Sex}=5000$

Textbook example cost-based estimate of join

DEPARTMENT $\bowtie_{Dnumber=Dno}$ EMPLOYEE;

- ▶ Statistics and access paths:
 - ▶ Department $r_D = 125$ records in $b_D = 13$ blocks
 - ▶ Employee as before
 - ▶ Primary index on Dnumber $x_{Dnumber} = 1$
 - ▶ Secondary index on Mgr_ssn $s_{Mgr_ssn} = 1$, $x_{Mgr_ssn} = 2$
 - ▶ Join selectivity $js_Q = 1/125$ because Dnumber is key
 - ▶ Assume $bfr_{ED} = 4$ for joint result relation
- ▶ Possible plans if 3 main memory buffers and their cost calculation:
 - ▶ Nested loop join, EMPLOYEE outer relation:
 - ▶ $b_E + (b_E * b_D) + ((js_Q * r_E * r_D) / bfr_{ED}) = 2000 + (2000 * 13) + (((1/125) * 10000 * 125) / 4) = \underline{30500}$
 - ▶ Nested loop join, DEPARTMENT outer relation
 - ▶ $b_D + (b_E * b_D) + ((js_Q * r_E * r_D) / bfr_{ED}) = 13 + (2000 * 13) + (((1/125) * 10000 * 125) / 4) = \underline{28513}$
 - ▶ Index-based join with EMPLOYEE outer relation:
 - ▶ $b_E + (r_E * (x_{Dno} + 1)) + ((js_Q * r_E * r_D) / bfr_{ED}) = 2000 + (10000 * 2) + (((1/125) * 10000 * 125) / 4) = \underline{24500}$
 - ▶ Index-based join with DEPARTMENT outer relation:
 - ▶ $b_D + (r_D * (x_{Dno} + s_{Dno})) + ((js_Q * r_E * r_D) / bfr_{ED}) = 13 + (125 * 2 + 80) + (((1/125) * 10000 * 125) / 4) = \underline{12763}$
 - ▶ Hash join: $3 * (b_D + b_E) + ((js_Q * r_E * r_D) / bfr_{ED}) = 3 * (13 + 2000) + 2500 = \underline{8539}$

If we had more main memory, what would be most cost efficient?

DEPARTMENT $\bowtie_{Dnumber=Dno}$ EMPLOYEE;

1. Hash join still
2. Nested loop join
3. Index-based join
4. Brute force approach

Statistics and access paths:

Department $r_D=125$ records in $b_D=13$ blocks

Employee as before

Primary index on Dnumber $x_{Dnumber}=1$

Secondary index on Mgr_ssn $s_{Mgr_ssn}=1$, $x_{Mgr_ssn}=2$

Join selectivity js_Q $1/125$ because Dnumber is key

Assume $bfr_{ED}=4$ for joint result relation

- ▶ Nested loop join, EMPLOYEE outer relation:
 - ▶ $b_E + (b_E * b_D) + ((js_Q * r_E * r_D) / bfr_{ED}) = 2000 + (2000 * 13) + (((1/125) * 10000 * 125) / 4) = \underline{30500}$
- ▶ Nested loop join, DEPARTMENT outer relation
 - ▶ $b_D + (b_E * b_D) + ((js_Q * r_E * r_D) / bfr_{ED}) = 13 + (2000 * 13) + (((1/125) * 10000 * 125) / 4) = \underline{28513}$
- ▶ Index-based join with EMPLOYEE outer relation:
 - ▶ $b_E + (r_E * (x_{Dno} + 1)) + ((js_Q * r_E * r_D) / bfr_{ED}) = 2000 + (10000 * 2) + (((1/125) * 10000 * 125) / 4) = \underline{24500}$
- ▶ Index-based join with DEPARTMENT outer relation:
 - ▶ $b_D + (r_D * (x_{Dno} + s_{Dno})) + ((js_Q * r_E * r_D) / bfr_{ED}) = 13 + (125 * 2 + 80) + (((1/125) * 10000 * 125) / 4) = \underline{12763}$
- ▶ Hash join: $3 * (b_D + b_E) + ((js_Q * r_E * r_D) / bfr_{ED}) = 3 * (13 + 2000) + 2500 = \underline{8539}$

If we had more main memory, what would be most cost efficient?

DEPARTMENT $\bowtie_{Dnumber=Dno}$ EMPLOYEE;

Statistics and access paths:

Department $r_D=125$ records in $b_D=13$ blocks

Employee as before

Primary index on Dnumber $x_{Dnumber}=1$

Secondary index on Mgr_ssn $s_{Mgr_ssn}=1$, $x_{Mgr_ssn}=2$

Join selectivity js_Q $1/125$ because Dnumber is key

Assume $bfr_{ED}=4$ for joint result relation

1. Hash join still
2. Nested loop join
3. Index-based join
4. Brute force approach

If we had $nB=15$ main memory buffers (or more), we could have the smaller relation Department with 13 blocks in main memory, then e.g.

▶ Nested loop join, DEPARTMENT outer relation

$$b_D + (b_E * (b_D / nB - 2)) + ((js_Q * r_E * r_D) / bfr_{ED}) = 13 + (2000 * 1) + (((1/125) * 10000 * 125) / 4) = \underline{4513}$$

▶ Nested loop join, EMPLOYEE outer relation:

$$b_E + (b_E * b_D) + ((js_Q * r_E * r_D) / bfr_{ED}) = 2000 + (2000 * 13) + (((1/125) * 10000 * 125) / 4) = \underline{30500}$$

▶ Nested loop join, DEPARTMENT outer relation

$$b_D + (b_E * b_D) + ((js_Q * r_E * r_D) / bfr_{ED}) = 13 + (2000 * 13) + (((1/125) * 10000 * 125) / 4) = \underline{28513}$$

▶ Index-based join with EMPLOYEE outer relation:

$$b_E + (r_E * (x_{Dno} + 1)) + ((js_Q * r_E * r_D) / bfr_{ED}) = 2000 + (10000 * 2) + (((1/125) * 10000 * 125) / 4) = \underline{24500}$$

▶ Index-based join with DEPARTMENT outer relation:

$$b_D + (r_D * (x_{Dno} + s_{Dno})) + ((js_Q * r_E * r_D) / bfr_{ED}) = 13 + (125 * 2 + 80) + (((1/125) * 10000 * 125) / 4) = \underline{12763}$$

▶ Hash join: $3 * (b_D + b_E) + ((js_Q * r_E * r_D) / bfr_{ED}) = 3 * (13 + 2000) + 2500 = \underline{8539}$

Comparison

- ▶ Heuristic query optimization
 - ▶ Sequence of single query plans
 - ▶ Each plan is (presumably) more efficient than the previous
 - ▶ Search is linear
- ▶ Cost-based query optimization
 - ▶ Many query plans generated
 - ▶ The cost of each is estimated, with the most efficient chosen
- ▶ Heuristic optimization is more efficient to generate, but may not yield the optimal query evaluation plan
- ▶ Cost-based optimization relies on statistics gathered on the relations

Intended learning outcomes

- ▶ Be able to
 - ▶ Describe and apply heuristics based and cost-based query optimization strategies

What was this all about?

Guidelines for your own review of today's session

- ▶ The overall idea in query optimization is to...
 - ▶ Heuristic query optimization follows the principle that...
 - ▶ It uses a specific data structure to represent the query...
 - Its canonical version is defined as...
 - ▶ Some of the steps taken in heuristic optimization use the following rules...
 - At a high level, the steps we go through in the algorithm are...
 - ▶ Cost-based query optimization on the other hand, uses...
 - ▶ It requires statistics on...
 - ▶ Cost is defined using e.g....
 - ▶ We compute the cost function as follows...
 - ▶ We then decide to pick the plan which...
 - ▶ Heuristic and cost-based query optimization differ in that...