CS 5614: (Big) Data Management Systems

B. Aditya Prakash
Lecture #6: Query Processing and Optimization
Outline

- introduction
- selection
- projection
- join
- set & aggregate operations
Introduction

- Today’s topic: QUERY PROCESSING
- Some database operations are EXPENSIVE
- Can greatly improve performance by being “smart”
  - e.g., can speed up 1,000,000x over naïve approach
Introduction (cnt’ d)

- Main weapons are:
  - clever implementation techniques for operators
  - exploiting “equivalencies” of relational operators
  - using statistics and cost models to choose among these.
A Really Bad Query Optimizer

- For each Select-From-Where query block
  - do cartesian products first
  - then do selections
  - etc, ie.:
    - GROUP BY; HAVING
    - projections
    - ORDER BY

- Incredibly inefficient
  - Huge intermediate results!
Cost-based Query Sub-System

Queries

Select *
From Blah B
Where B.blah = blah

Query Parser

Query Optimizer

Plan Generator
Plan Cost Estimator

Catalog Manager

Schema
Statistics

Usually there is a heuristics-based rewriting step before the cost-based steps.
The Query Optimization Game

- “Optimizer” is a bit of a misnomer...
- Goal is to pick a “good” (i.e., low expected cost) plan.
  - Involves choosing access methods, physical operators, operator orders, ...
  - Notion of cost is based on an abstract “cost model”
Relational Operations

- We will consider how to implement:
  - **Selection** ($\sigma$) Selects a subset of rows from relation.
  - **Projection** ($\pi$) Deletes unwanted columns from relation.
  - **Join** ($\bowtie$) Allows us to combine two relations.
  - **Set-difference** (-) Tuples in reln. 1, but not in reln. 2.
  - **Union** ($\cup$) Tuples in reln. 1 and in reln. 2.
  - **Aggregation** (SUM, MIN, etc.) and GROUP BY

- Recall: ops can be *composed*!

- Later, we’ll see how to *optimize* queries with many ops
Schema for Examples

Sailors \((sid: \text{integer}, sname: \text{string}, rating: \text{integer}, age: \text{real})\)
Reserves \((sid: \text{integer}, bid: \text{integer}, day: \text{dates}, rname: \text{string})\)

- Similar to old schema; \textit{rname} added for variations.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  - \(N=500, p_s=80\).
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
  - \(M=1000, p_R=100\).
Simple Selections

- Of the form $\sigma_{R.\text{attr} \ op \ \text{value}} (R)$
- Question: how best to perform?

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

A: Depends on:

– what indexes/access paths are available
– what is the expected size of the result (in terms of number of tuples and/or number of pages)
Simple Selections

- Size of result approximated as
  
  \[
  \text{size of } R \times \text{reduction factor}
  \]

  - “reduction factor” is also called \textit{selectivity}.
  
  - estimate of reduction factors is based on statistics – we will discuss shortly.
Alternatives for Simple Selections

- With no index, unsorted:
  - Must essentially scan the whole relation
  - cost is $M$ (#pages in $R$). For “reserves” = 1000 I/Os.
Simple Selections (cnt’ d)

- With no index, sorted:
  - cost of binary search + number of pages containing results.
  - For reserves = 10 I/Os + \([\text{selectivity} \times \#\text{pages}]\)
Simple Selections (cnt’d)

- With an index on selection attribute:
  - Use index to find qualifying data entries,
  - then retrieve corresponding data records.
  - (Hash index useful only for equality selections.)
Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering.
  - Cost:
    - finding qualifying data entries (typically small)
    - plus cost of retrieving records (could be large w/o clustering).
Selections using Index (cnt’d)

Index entries direct search for data entries

Data Records

Data entries

(Index File)

(Data file)

CLUSTERED

UNCLUSTERED

Data Records
Selections using Index (cnt’d)

– In example “reserves” relation, if 10% of tuples qualify (100 pages, 10,000 tuples).
  • With a clustered index, cost is little more than 100 I/Os;
  • if unclustered, could be up to 10,000 I/Os! unless...
Selections using Index (cnt’d)

- **Important refinement for unclustered indexes:**
  1. Find qualifying data entries.
  2. Sort the rid’s of the data records to be retrieved.
  3. Fetch rids in order. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).
The Projection Operation

- Issue is removing duplicates.
- Basic approach: sorting
  - 1. Scan R, extract only the needed attrs (why?)
  - 2. Sort the resulting set
  - 3. Remove adjacent duplicates

Cost: Reserves with size ratio 0.25 = 250 pages. With 20 buffer pages can sort in 2 passes, so

\[ 1000 + 250 + 2 \times 2 \times 250 + 250 = 2500 \text{ I/Os} \]

```sql
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```
Discussion of Projection

- If an index on the relation contains all wanted attributes in its search key, can do *index-only* scan.
  - Apply projection techniques to data entries (much smaller!)
Joins

- Joins are very common.
- Joins can be very expensive (cross product in worst case).
- Many approaches to reduce join cost.
Joins

- Join techniques we will cover:
  - Nested-loops join
  - Index-nested loops join
  - Sort-merge join
  - Hash join
Equality Joins With One Join Column

SELECT *  
FROM Reserves R1, Sailors S1  
WHERE R1.sid=S1.sid

- In algebra: $R \bowtie S$. Common! Must be carefully optimized. $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient.
- Remember, join is associative and commutative.
Equality Joins

- Assume:
  - M pages in R, \( p_R \) tuples per page, \( m \) tuples total
  - N pages in S, \( p_S \) tuples per page, \( n \) tuples total
  - In our examples, R is Reserves and S is Sailors.

- We will consider more complex join conditions later.

- *Cost metric*: \# of I/Os. We will ignore output costs.
Nested loops

- Algorithm #0: (naive) nested loop (*SLOW!*)

\[ \text{Algorithm #0: (naive) nested loop (SLOW!)} \]

- Diagram showing two nested loops, one inner loop labeled \( \text{R(A,..)} \) and one outer loop labeled \( \text{S(A, ......)} \). The inner loop iterates over a range of \( m \) values, and the outer loop iterates over a range of \( n \) values.
Nested loops

- Algorithm #0: (naive) nested loop (\textbf{SLOW}!)
  for each tuple $r$ of $R$
    for each tuple $s$ of $S$
      print, if they match
Nested loops

- **Algorithm #0**: (naive) nested loop (SLOW!)

  for each tuple r of \( R \)
  
  for each tuple s of \( S \)
  
  print, if they match

---

\[
\begin{align*}
R(A, \ldots) & \quad m \\
S(A, \ldots) & \quad n
\end{align*}
\]
Nested loops

- Algorithm #0: why is it bad?
- how many disk accesses (‘M’ and ‘N’ are the number of blocks for ‘R’ and ‘S’)?

\[
\begin{align*}
&\text{R}(A,\ldots) \\
M \text{ pages, m tuples} \\
&\text{S}(A, \ldots) \\
\text{N pages, n tuples}
\end{align*}
\]
Nested loops

- Algorithm #0: why is it bad?
- how many disk accesses (‘M’ and ‘N’ are the number of blocks for ‘R’ and ‘S’)? $M + m \times N$

M pages, m tuples

R(A,..)

S(A, ......)

N pages, n tuples
Simple Nested Loops Join

- Actual number
  \[(p_R \times M) \times N + M = 100 \times 1000 \times 500 + 1000 \text{ I/Os.}\]
  - At 10ms/IO, Total: ???
- What if smaller relation (S) was outer?
- What assumptions are being made here?
Simple Nested Loops Join

- Actual number
- \((p_R \times M) \times N + M = 100 \times 1000 \times 500 + 1000\) I/Os.
  - At 10ms/IO, Total: \(\sim 6\) days (!)
- What if smaller relation \((S)\) was outer?
  - slightly better
- What assumptions are being made here?
  - 1 buffer for each table (and 1 for output)
Page-Oriented Nested loops

• Algorithm #1: Page-or. nested-loop join
  – read in a block (=1 page) of R
    • read in a block (=1 page) of S
      – print matching tuples

COST?
Page-Oriented Nested loops

• Algorithm #1: Page-or. nested-loop join
  – read in a block (=1 page) of R
    • read in a block (=1 page) of S
      – print matching tuples

\[ \text{COST} = M + M \times N \]
Page-Oriented Nested loops

• Which one should be the outer relation?

$\text{COST} = M + M \times N$
Page-Oriented Nested loops

- Which one should be the outer relation?
- A: the smallest (page-wise)

\[ \text{COST} = M + M \times N \]
Page-Oriented Nested loops

- $M=1000$, $N=500$
- Cost $= 1000 + 1000 \times 500 = 501,000$
- $= 5010$ sec $\sim 1.4h$

$\text{COST} = M + M \times N$
Page-Oriented Nested loops

- $M=1000$, $N=500$ - if smaller is outer:
- Cost $= 500 + 1000 \times 500 = 500,500$
- $= 5005$ sec $\sim 1.4$ h

\[ \text{COST} = N + M \times N \]
Blocked-Nested loops

- What if we have B buffers available and can use all of them?
  (in contrast: page-oriented NJ only uses 3 buffers)
Blocked-Nested loops

• What if we have B buffers available?
• A: give $B-2$ buffers to outer, 1 to inner, 1 for output
Blocked-Nested loops

• Algorithm #2: Blocked nested-loop join
  – read in $B-2$ blocks of $R$
    • read in a block of $S$
      – print matching tuples

\[
\text{COST} = ?
\]

\[
\begin{array}{c}
\text{R(A,,..)} \\
\text{M pages,} \\
\text{m tuples}
\end{array}
\quad
\begin{array}{c}
\text{S(A, ......)} \\
\text{N pages,} \\
\text{n tuples}
\end{array}
\]
Nested loops

- Algorithm #2: Blocked nested-loop join
  - read in \( B-2 \) blocks of \( R \)
    - read in a block of \( S \)
      - print matching tuples

\[
\text{COST} = M + \frac{M}{B-2} \cdot N
\]
Nested loops

and, actually:

Cost = M + ceiling(M/(B-2)) * N

\[ \text{COST} = M + \frac{M}{(B-2)} \times N \]
Nested loops

• If smallest (outer) fits in memory
• (ie., $B = N + 2$),
• Cost =?

$$\text{COST} = N + \frac{N}{B-2} \times M$$
Nested loops

- If smallest (outer) fits in memory
- (ie., $B = N+2$),
- Cost $= N + M$ (minimum!) $\text{COST} = \frac{N+N}{B-2} \times M$
Nested loops - guidelines

- pick as outer the smallest table (= fewest pages)
- fit as much of it in memory as possible
- loop over the inner
Index NL join

- use an existing **index**, or even build one on the fly
- cost: $M + m \times c$  \hspace{1cm} (c: look-up cost)
Index NL join

- **cost**: $M + m \times c$ (c: look-up cost)
- ‘c’ depends whether the index is clustered or not.

$$c'$$

Index NL join $R(A,..)$ $S(A, ......)$

- **M pages**, **m tuples**
- **N pages**, **n tuples**
Joins

- Join techniques we will cover:
  - Nested-loops join
  - Index-nested loops join
  - Sort-merge join
  - Hash join
Sort-merge join

- sort both on joining attributes
- scan each and merge
- Cost, given B buffers?

\[
R(A,..) \\
\text{M pages, m tuples} \\
\downarrow \\
S(A, ......) \\
\uparrow \\
\text{N pages, n tuples}
\]
Sort-merge join

- Cost, given B buffers?
- $\sim 2M \log M / \log B + 2N \log N / \log B + M + N$
Sort-Merge Join

- Useful if
Sort-Merge Join

- Useful if
  - one or both inputs are already sorted on join attribute(s)
  - output is required to be sorted on join attributes(s)

- “Merge” phase can require some back tracking if duplicate values appear in join column
**Example of Sort-Merge Join**

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>103</td>
<td>12/4/96</td>
<td>guppy</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
<td>11/3/96</td>
<td>yuppy</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
<td>dustin</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>10/12/96</td>
<td>lubber</td>
</tr>
<tr>
<td>31</td>
<td>101</td>
<td>10/11/96</td>
<td>lubber</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>
Example of Sort-Merge Join

- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.
- (while Block Nested Loop (BNL) cost: 2,500 to 15,000 I/Os)
Sort-merge join

- Worst case for merging phase?
- Cost?
Refinements

- All the refinements of external sorting
- plus overlapping of the merging of sorting with the merging of joining.
Joins

- Join techniques we will cover:
  - Nested-loops join
  - Index-nested loops join
  - Sort-merge join
  - Hash join
- hash join: use hashing function $h()$
  - hash ‘R’ into (0, 1, ..., ‘max’) buckets
  - hash ‘S’ into buckets (same hash function)
  - join each pair of matching buckets
Hash join - details

– how to join each pair of partitions Hr-i, Hs-i ?
– A: build another hash table for Hs-i, and probe it with each tuple of Hr-i
Hash join - details

- In more detail:
- Choose the (page-wise) smallest - if it fits in memory, do ~NL
  - and, actually, build a hash table (with h2() \neq h())
  - and probe it, with each tuple of the other
what if Hs-i is too large to fit in main-memory?

A: recursive partitioning

more details (overflows, hybrid hash joins): in book

cost of hash join? (if we have enough buffers:)

$3(M + N)$ (why? See next slide)
Cost of Hash-Join

- In partitioning phase, read+write both relns; \(2(M+N)\). In matching phase, read both relns; \(M+N\) I/Os.

- In our running example, this is a total of 4500 I/Os.
Hash join details

- [cost of hash join? (if we have enough buffers:)]
  \[3(M + N)\]

- What is ‘enough’? \(\sqrt{N}\), or \(\sqrt{M}\)?
Hash join details

- [cost of hash join? (if we have enough buffers:)]
  \[3(M + N)\]

- What is ‘enough’? \(\sqrt{N}\), or \(\sqrt{M}\)?

- A: \(\sqrt{\text{smallest}}\) (why?)
  - Because you only need enough memory to hold the hash table partitions of the smaller table in memory so \(B > \text{size of smaller}/B - 1 \Rightarrow B \sim \sqrt{\text{size-of-smaller}}\)
Sort-Merge Join vs. Hash Join

- Given a minimum amount of memory both have a cost of $3(M+N)$ I/Os.

  (min. memory for sort-merge = $\sqrt{\text{larger table}}$ using aggressive refinements---in textbook)

  (min. memory for hash = $\sqrt{\text{smaller table}}$---see previous slides)
Sort-Merge vs Hash join

- **Hash Join Pros:**
  - ??
  - ??
  - ??

- **Sort-Merge Join Pros:**
  - ??
Sort-Merge vs Hash join

- **Hash Join Pros:**
  - Superior if relation sizes differ greatly
  - Shown to be highly parallelizable (*beyond scope of class*)

- **Sort-Merge Join Pros:**
  - ??
Sort-Merge vs Hash join

- **Hash Join Pros:**
  - Superior if relation sizes differ greatly
  - Shown to be highly parallelizable

- **Sort-Merge Join Pros:**
  - Less sensitive to data skew
  - Result is sorted (may help “upstream” operators)
  - Goes faster if one or both inputs already sorted
General Join Conditions

- Equalities over several attributes (e.g., $R.sid = S.sid$ AND $R.rname = S.sname$):
  - all previous methods apply, using the composite key
General Join Conditions

- Inequality conditions (e.g., $R.rname < S.sname$):
- which methods still apply?
  - NL
  - index NL
  - Sort merge
  - Hash join
General Join Conditions

- Inequality conditions (e.g., $R.rname < S.sname$):
- which methods still apply?
  - NL (probably, the best!)
  - index NL (only if clustered index)
  - Sort merge (does not apply!) (why?)
  - Hash join (does not apply!) (why?)
Set Operations

- Intersection and cross-product: special cases of join
- Union (Distinct) and Except: similar; we’ll do union:
- Effectively: concatenate; use sorting or hashing
- Sorting based approach to union:
  - Sort both relations (on combination of all attributes).
  - Scan sorted relations and merge them.
  - *Alternative*: Merge runs from Pass 0 for *both* relations.
Set Operations, cont’d

- Hash based approach to union:
  - Partition R and S using hash function $h$.
  - For each S-partition, build in-memory hash table (using $h2$), scan corresponding R-partition and add tuples to table while discarding duplicates.
Without grouping:

- In general, requires scanning the relation.
- Given index whose search key includes all attributes in the `SELECT` or `WHERE` clauses, can do index-only scan.
Summary

- A virtue of relational DBMSs:
  - queries are composed of **a few basic operators**
  - The implementation of these operators can be **carefully tuned**
  - **Important** to do this!

- Many alternative implementation techniques for each operator
  - No universally superior technique for most operators.

“it depends” [Guy Lohman (IBM)]
Summary cont’d

- Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc.
  - Part of the broader task of optimizing a query composed of several ops.
QUERY OPTIMIZATION
Some parts from (a copy of the paper is on the course webpage)

Cost-based Query Sub-System

Usually there is a heuristics-based rewriting step before the cost-based steps.
Multiple Algorithms: Range Searches

- Sequential Scan
- Hashes
- B-Trees
- ....

- Saw some of them in previous lectures
Multiple Algorithms: Joins

- Merge-Join (like merge-sort)
- Hash-Join (using hashes)
- Indexed-Join (using indexes)
- Nested loops Join (most obvious)
- ...

- Saw some of them previously
Why Query optimization?

- SQL: ~declarative
- good q-opt -> big difference
  - eg., seq. Scan vs
  - B-tree index, on P=1,000 pages
Q-opt steps

- bring query in internal form (eg., parse tree)
- ... into ‘canonical form’ (syntactic q-opt)
- generate alt. plans
- estimate cost; pick best
Q-opt - example

```
select name
from STUDENT, TAKES
where c-id = '4604' and
STUDENT.ssn = TAKES.ssn
```
Q-opt - example

```
select name
from STUDENT, TAKES
where c-id = '4604' and
STUDENT.ssn = TAKES.ssn
```

Join Predicate $\rightarrow$ STUDENT.ssn = TAKES.ssn
(is assumed to be part of the join)

Non-join Predicate $\rightarrow$ c-id = '4604'
(part of the explicit selection)
Q-opt - example

Canonical form

STUDENT  TAKES  STUDENT  TAKES
Q-opt - example

Canonical Form has the following properties:
1. Push Selections as much as possible.
2. Push Projections as much as possible
3. It is a left-deep join tree (we will see this later)
Q-opt - example

```
<table>
<thead>
<tr>
<th>π</th>
<th>Σ</th>
<th>Index; seq scan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ST: TAKES</td>
</tr>
</tbody>
</table>
```

Hash join; merge join; nested loops;
Equivalence of expressions

- A.k.a.: syntactic q-opt
- in short: perform selections and projections early
Equivalence of expressions

Q: How to prove a transformation rule?

\[ \sigma_P(R_1 \Join R_2) = \sigma_P(R_1) \Join \sigma_P(R_2) \]

A: use RA, to show that LHS = RHS, eg:

\[ \sigma_P(R_1 \cup R_2) = \sigma_P(R_1) \cup \sigma_P(R_2) \]
Equivalence of expressions

\[ \sigma_P(R1 \cup R2) = \sigma_P(R1) \cup \sigma_P(R2) \]
\[ t \in LHS \iff \]
\[ t \in (R1 \cup R2) \land P(t) \iff \]
\[ (t \in R1 \lor t \in R2) \land P(t) \iff \]
\[ (t \in R1 \land P(t)) \lor (t \in R2 \land P(t)) \iff \]
Equivalence of expressions

\[ \sigma_p(R1 \cup R2) = \sigma_p(R1) \cup \sigma_p(R2) \]

\[
\begin{align*}
&\quad (t \in R1 \land P(t)) \lor (t \in R2) \land P(t) \\
&\quad (t \in \sigma_p(R1)) \lor (t \in \sigma_p(R2)) \\
&\quad t \in \sigma_p(R1) \cup \sigma_p(R2) \\
&\quad t \in RHS
\end{align*}
\]

QED
Equivalence of expressions

- Q: how to disprove a rule??

$$\pi_A(R1 - R2) = \pi_A(R1) - \pi_A(R2)$$

Construct a counter-example!
Equivalence of expressions

- Selections
  - perform them early
  - break a complex predicate, and push
    \[ \sigma_{p_1 \land p_2 \land \ldots \land p_n}(R) = \sigma_{p_1}(\sigma_{p_2}(\ldots \sigma_{p_n}(R))\ldots) \]
  - simplify a complex predicate
    - (‘X=Y and Y=3’) -> ‘X=3 and Y=3’
Equivalence of expressions

- Projections
  - perform them early (but carefully...)
    - Smaller tuples
    - Fewer tuples (if duplicates are eliminated)
  - project out all attributes except the ones requested or required (e.g., joining attr.)
Equivalence of expressions

- Joins
  - Commutative, associative
    \[ R \bowtie S = S \bowtie R \]
    \[ (R \bowtie S) \bowtie T = R \bowtie (S \bowtie T) \]
  - Q: n-way join - how many diff. orderings?
Equivalence of expressions

- Joins - Q: n-way join - how many diff. orderings?
- A: Catalan number ~ $4^n$
  - Exhaustive enumeration: too slow.
(Some) Transformation Rules (1)

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.
\[
\sigma_{\theta_1 \land \theta_2} (E) = \sigma_{\theta_1} (\sigma_{\theta_2} (E))
\]

2. Selection operations are commutative.
\[
\sigma_{\theta_1} (\sigma_{\theta_2} (E)) = \sigma_{\theta_2} (\sigma_{\theta_1} (E))
\]

3. Only the last in a sequence of projection operations is needed, the others can be omitted.
\[
\prod_{L_1} (\prod_{L_2} (\ldots (\prod_{L_n} (E)) \ldots)) = \prod_{L_1} (E)
\]

4. Selections can be combined with Cartesian products and theta joins.
   a. \[
   \sigma_{\theta} (E_1 \times E_2) = E_1 \Join_{\theta} E_2
   \]
   b. \[
   \sigma_{\theta_1 \land \theta_2} (E_1 \Join_{\theta_1 \land \theta_2} E_2) = E_1 \Join_{\theta_1 \land \theta_2} E_2
   \]
(Some) Transformation Rules (2)

5. Theta-join operations (and natural joins) are commutative.

\[ E_1 \Join_\theta E_2 = E_2 \Join_\theta E_1 \]

6. (a) Natural join operations are associative:

\[(E_1 \Join E_2) \Join E_3 = E_1 \Join (E_2 \Join E_3)\]

(b) Theta joins are associative in the following manner:

\[(E_1 \Join_{\theta_1} E_2) \Join_{\theta_2 \land \theta_3} E_3 = E_1 \Join_{\theta_1 \land \theta_3} (E_2 \Join_{\theta_2} E_3)\]

where \(\theta_2\) involves attributes from only \(E_2\) and \(E_3\).
7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in $\theta_0$ involve only the attributes of one of the expressions ($E_1$) being joined.

   $$\sigma_{\theta_0}(E_1 \bowtie \theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie \theta E_2$$

   (b) When $\theta_1$ involves only the attributes of $E_1$ and $\theta_2$ involves only the attributes of $E_2$.

   $$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie \theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie \theta (\sigma_{\theta_2}(E_2))$$
Q-opt steps

- bring query in internal form (e.g., parse tree)
- ... into ‘canonical form’ (syntactic q-opt)
- generate alt. plans
- estimate cost; pick best
Cost-based Query Sub-System

- **Query Parser**
- **Query Optimizer**
  - **Plan Generator**
  - **Plan Cost Estimator**
- **Query Plan Evaluator**
- **Catalog Manager**
  - **Schema**
  - **Statistics**

Queries:

```
Select *
From Blah B
Where B.blah = blah
```

Usually there is a heuristics-based rewriting step before the cost-based steps.
Cost estimation

- Eg., find ssn’s of students with an ‘A’ in 4604 (using seq. scanning)
- How long will a query take?
  - CPU (but: small cost; decreasing; tough to estimate)
  - Disk (mainly, # block transfers)
- How many tuples will qualify?
- (what statistics do we need to keep?)
Cost estimation

- Statistics: for each relation ‘r’ we keep
  - nr : # tuples;
  - Sr : size of tuple in bytes
Cost estimation

- Statistics: for each relation ‘r’ we keep
  - …
  - $V(A, r)$: number of distinct values of attr. ‘A’
  - (recently, histograms, too)
Derivable statistics

- blocking factor = max# records/block (=??)
- br: # blocks (=??)
- SC(A,r) = selection cardinality = avg# of records with A=given (=??)
Derivable statistics

- blocking factor = max# records/block (= B/Sr ; B: block size in bytes)
- br: # blocks (= nr / (blocking-factor) )
Derivable statistics

- \( SC(A,r) = \text{selection cardinality} = \text{avg# of records with } A=\text{given} \ (\text{= } nr / V(A,r) \ ) \ (\text{assumes uniformity...}) \)

eg: 10,000 students, 10 departments – how many students in CS?
Additional quantities we need:

- For index ‘i’:
  - $f_i$: average fanout ($\sim 50-100$)
  - $HT_i$: # levels of index ‘i’ ($\sim 2-3$)
    - $\sim \log(#\text{entries})/\log(f_i)$
  - $LB_i$: # blocks at leaf level
Statistics

- Where do we store them?
- How often do we update them?
Q-opt steps

- bring query in internal form (eg., parse tree)
- ... into ‘canonical form’ (syntactic q-opt)
- generate alt. plans
- estimate cost; pick best
Selections

- We saw simple predicates (A=constant; eg., ‘name=Smith’)
- How about more complex predicates, like
  - ‘salary > 10K’
  - ‘age = 30 and job-code=“analyst”’
- What is their selectivity?
Selections – complex predicates

- selectivity $\text{sel}(P)$ of predicate $P$:
  - $\approx$ fraction of tuples that qualify
  - $\text{sel}(P) = \frac{\text{SC}(P)}{\text{nr}}$
Selections – complex predicates

- eg., assume that $V(\text{grade, TAKES})=5$ distinct values
- simple predicate $P$: $A=\text{constant}$
  - $\text{sel}(A=\text{constant}) = 1/V(A,r)$
  - eg., $\text{sel}(\text{grade}=\text{‘B’}) = 1/5$
- (what if $V(A,r)$ is unknown??)
Selections – complex predicates

- range query: \( \text{sel( grade } \geq \text{ ‘C’ } \) }
- \( \text{sel(A}>a) = \frac{\text{Amax} - a}{\text{Amax} - \text{Amin}} \)

![Diagram showing a range query for grades F to A]
Selections - complex predicates

- negation: \( \text{sel}( \text{grade} \neq 'C') \)
  - \( \text{sel}( \text{not } P) = 1 - \text{sel}(P) \)
  - (Observation: selectivity = \(\sim\) probability)
Selections - complex predicates

- Conjunction:
  - `sel( grade = 'C' and course = '4604' )`
  - `sel(P1 and P2) = sel(P1) * sel(P2)`
  - INDEPENDENCE ASSUMPTION
Selections - complex predicates

- Disjunction:
  - \( \text{sel}( \text{grade} = 'C' \text{ or } \text{course} = '4604') \)
  - \( \text{sel}(P1 \text{ or } P2) = \text{sel}(P1) + \text{sel}(P2) - \text{sel}(P1 \text{ and } P2) \)
  - \( = \text{sel}(P1) + \text{sel}(P2) - \text{sel}(P1) * \text{sel}(P2) \)
  - INDEPENDENCE ASSUMPTION, again
Selections - complex predicates

- disjunction: in general
  - $\text{sel}(P_1 \textbf{or} P_2 \textbf{or} \ldots \textbf{or} \ P_n) =$
    - $1 - (1 - \text{sel}(P_1)) \times (1 - \text{sel}(P_2)) \times \ldots \times (1 - \text{sel}(P_n))$
Selections Selectivity – summary

- \( \text{sel}(A=\text{constant}) = \frac{1}{V(A,r)} \)
- \( \text{sel}(A>a) = \frac{(A_{\text{max}} - a)}{(A_{\text{max}} - A_{\text{min}})} \)
- \( \text{sel}(\text{not } P) = 1 - \text{sel}(P) \)
- \( \text{sel}(P_1 \text{ and } P_2) = \text{sel}(P_1) \times \text{sel}(P_2) \)
- \( \text{sel}(P_1 \text{ or } P_2) = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1)\times\text{sel}(P_2) \)
- \( \text{sel}(P_1 \text{ or ... or } P_n) = 1 - (1-\text{sel}(P_1))\times\ldots\times(1-\text{sel}(P_n)) \)

- UNIFORMITY and INDEPENDENCE ASSUMPTIONS
Result Size Estimation for Joins

- Q: Given a join of R and S, what is the range of possible result sizes (in #of tuples)?
  - Hint: what if $\text{R}\_\text{cols} \cap \text{S}\_\text{cols} = \emptyset$?
  - $\text{R}\_\text{cols} \cap \text{S}\_\text{cols}$ is a key for R (and a Foreign Key in S)?
Result Size Estimation for Joins

- General case: R_cols ∩ S_cols = {A} (and A is key for neither)
  - match each R-tuple with S-tuples
    \[
    \text{est}_\text{size} \sim NTuples(R) \times NTuples(S) / NKeys(A,S)
    \]
    \[
    \sim nr \times ns / V(A,S)
    \]
  - symmetrically, for S:
    \[
    \text{est}_\text{size} \sim NTuples(R) \times NTuples(S) / NKeys(A,R)
    \]
    \[
    \sim nr \times ns / V(A,R)
    \]
  - Overall:
    \[
    \text{est}_\text{size} = NTuples(R) \times NTuples(S) / \text{MAX}\{NKeys(A,S), NKeys(A,R)\}
    \]

Prakash 2014
VT CS 5614
123
On the Uniform Distribution Assumption

- Assuming uniform distribution is rather crude

Distribution D

Uniform distribution approximating D
For better estimation, use a *histogram*

**Equiwidth histogram**

- Bucket 1: Count=8
- Bucket 2: Count=4
- Bucket 3: Count=15
- Bucket 4: Count=3
- Bucket 5: Count=15

**Equidepth histogram ~ quantiles**

- Bucket 1: Count=9
- Bucket 2: Count=10
- Bucket 3: Count=10
- Bucket 4: Count=7
- Bucket 5: Count=9
Q-opt Steps

- bring query in internal form (eg., parse tree)
- ... into ‘canonical form’ (syntactic q-opt)
- generate alt. plans
  - single relation
  - multiple relations
- estimate cost; pick best
- Selections – e.g.,
  
  ```sql
  select *
  from TAKES
  where grade = 'A'
  ```

- Plans?

---

**plan generation**
plan generation

- Plans?
  - seq. scan
  - binary search
    - (if sorted & consecutive)
  - index search
    - if an index exists
plan generation

seq. scan – cost?
- br (worst case)
- br/2 (average, if we search for primary key)
plan generation

binary search – cost?
if sorted and consecutive:
- $\sim \log(br) +$
- $\frac{SC(A,r)}{fr}$ (=blocks spanned by qual. tuples)
plan generation

estimation of selection cardinalities SC(A,r):
– we saw it earlier how to do it for general conditions
method#3: index – cost?

– Roughly $\log(N)$, but exact cost tricky

SKIP!
Q-opt Steps

- bring query in internal form (eg., parse tree)
- ... into ‘canonical form’ (syntactic q-opt)
- generate alt. plans
  - single relation
  - multiple relations
- estimate cost; pick best
n-way joins

- $r_1 \text{ JOIN } r_2 \text{ JOIN } ... \text{ JOIN } r_n$
- typically, break problem into 2-way joins
  - choose between NL, sort merge, hash join, ...
Queries Over Multiple Relations

- As number of joins increases, number of alternative plans grows rapidly → need to restrict search space
- Fundamental decision in System R (IBM): only left-deep join trees are considered. Advantages?
Queries Over Multiple Relations

- As number of joins increases, number of alternative plans grows rapidly → need to restrict search space

- Fundamental decision in System R (IBM): only left-deep join trees are considered. Advantages?
  - fully pipelined plans.
  - Intermediate results not written to temporary files.
Queries over Multiple Relations

- Enumerate the orderings (= left deep tree)
- enumerate the plans for each operator
- enumerate the access paths for each table

Dynamic programming, to save cost estimations
(we wont cover exact algorithm in class)
Candidate Plans

1. Enumerate relation orderings:

Prune plans with cross-products immediately!
2. Enumerate *join algorithm* choices:

+ do same for 4 other plans

\[ 4 \times 4 = 16 \text{ plans so far..} \]
Candidate Plans

3. Enumerate access method choices:

+ do same for other plans

SELECT  S.sname, B.bname, R.day 
FROM   Sailors S, Reserves R, Boats B 
Now estimate the cost of each plan

Example:

```
NLJ
  /\  /
NLJ B
 /\ /
S R
(heap scan) (heap scan) (INDEX scan on R.sid)
```
Conclusions

- Ideas to remember:
  - canonical parse tree
  - syntactic q-opt – do selections/projections early
    - More complicated rules are also used
  - How to get selectivity estimations (uniformity, independence)
    - We saw mainly range and equality predicates
    - More complicated: histograms; join selectivity
  - left-deep joins
    - dynamic programming