Database design and implementation

CMPSCI 645

Lectures 15-16: Query Evaluation and Optimization
Relational operations

We will consider how to implement:

- **Selection** ($\sigma$) Selects a subset of rows from relation.
- **Join** ($\bowtie$) Allows us to combine two relations.
- **Projection** ($\pi$) Deletes unwanted columns from relation.
- **Union** (U) Tuples in either reln. 1 or reln. 2.
- **Intersection** ($\cap$) Tuples in both reln. 1 and reln. 2.
- **Set-difference** (−) Tuples in reln. 1, but not in reln. 2.
- **GROUP BY** and **Aggregation** (SUM, MIN, etc.)
Outline

- Selections
- Sorting routine
- Joins
- Projections
- Set operators
- Group By aggregation
Schema for examples

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

- **Sailors:**
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - 500 pages.

- **Reserves:**
  - Each tuple is 40 bytes long,
  - 100 tuples per page,
  - 1000 pages.

- **Cost metric:** \# I/Os
Using an index for selections

Cost of selection includes:
1) top down search in the index
2) scan the relevant leaf nodes
3) retrieve records from file (could be large w/o clustering).

- **Step 1) top down search**: ≤ 3-4 I/Os, depending on buffer management
Cost factors of steps 2 and 3

Cost of selection includes:
1) top down search in the index
2) scan the relevant leaf nodes
3) retrieve records from file (could be large w/o clustering).

Cost factor: number of qualifying tuples
- rating > 8: if 20% of tuples qualify, 500/5=100 pages, 80*100=8,000 tuples.
- Scanning leaf nodes: if a data entry is 1/5 of a tuple, need 20 leaf nodes, so 20 I/Os.
Cost factors of selection (contd.)

- **Cost factor: clustering**
  - *rating > 8*: 20% of tuples qualify, 100 pages, 8,000 tuples.

- **Retrieving records from file ≈**
  - **Clustered index**: 100 I/Os.
  - **Unclustered index**: worst case 1 I/O per tuple; 8,000 I/Os here!
General selections

- Boolean combination of predicates using AND and OR.
  - Conjunctive Normal Form (CNF), e.g.,
    
    \[
    pred1 \text{ AND } (pred3 \text{ OR } pred4) \\
    (pred1 \text{ OR } pred2) \text{ AND } (pred3 \text{ OR } pred4)
    \]

- *File scan* always works for general selections.

- *Index scan* works when it matches a predicate that is a conjunct of CNF.
  - E.g., an index matching *pred1* can be used for
    
    \[
    pred1 \text{ AND } (pred3 \text{ OR } pred4)
    \]
Conjunctive predicates only

- CNF without OR: e.g., \(pred_1 \text{ AND } pred_2 \text{ AND } pred_3\)
  - Find the most selective access path, retrieve tuples using it
    - File scan or index scan that gives the smallest I/O cost.
    - Apply remaining terms that don’t match index on the fly.
    - Other terms do not affect I/O cost.

- \(day<8/9/94 \text{ AND } bid=5 \text{ AND } sid=3\)
  - Hash index on \(<bid, sid>\): check \(day<8/9/94\) on the fly.
  - B+ tree index on \(day\): apply \(bid=5\) and \(sid=3\) on the fly.
Improvement: intersection of rids

- 2+ matching indexes (Alternatives 2 or 3):
  1. Get sets of rids of data records using each index.
  2. *Intersect* these sets of rids.
  3. Retrieve the records and apply any remaining terms.

\[ day<8/9/94 \text{ AND bid}=5 \text{ AND } sid=3 \]

B+ tree index on *day*, a hash index on *sid*, both using Alt 2:

1. retrieve rids of records satisfying *day*<8/9/94 using first index,
   rids of records satisfying *sid*=3 using second index,
2. intersect these rids,
3. retrieve records, check *bid*=5.
Outline

- Selections
  - Sorting routine
- Joins
- Projections
- Set operators
- Group By aggregation
Why sort?

- Important utility in DBMS:
  - *Sort-merge* join algorithm involves sorting.
  - *Eliminate duplicates* in a collection of records (e.g., `SELECT DISTINCT`)
  - Request data in *sorted order* (e.g., `ORDER BY`)  
    - e.g., find students in decreasing order of `gpa`
  - Sorting is first step in *bulk loading* B+ tree index.

- **Problem:** sort 1GB of data with 1MB of RAM.
  - Limited Memory. Key is to minimize # I/Os!
Two-way sort: Requires 3 buffers

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used

- Pass 2, 3, ..., etc.: Merge two sorted subfiles
  - three buffer pages used.
Two-way external merge sort

Divide and conquer:
- sort subfiles (runs)
- and merge

A file of $N$ pages:
- Pass 1: $N$ sorted runs of 1 page each
- Pass 2: $N/2$ sorted runs of 2 pages each
- Pass 3: $N/4$ sorted runs of 4 pages each
- ... 
- Pass $P+1$: 1 sorted run of $2^P$ pages

$2^P \geq N \Rightarrow P \geq \log_2 N$
Two-way external merge sort

Divide and conquer:
- sort subfiles (runs)
- and merge

• Each pass, read + write N pages in file $\rightarrow$ 2N.

• Number of passes is:

$$\lceil \log_2 N \rceil + 1$$

• So total cost is:

$$2N(\lceil \log_2 N \rceil + 1)$$
General external merge sort

*Given B (>3) buffer pages. How can we utilize them?*

- **Pass 1:** *Use B buffer pages.* Produce $\lceil \frac{N}{B} \rceil$ sorted runs of $B$ pages each.
- **Pass 2, 3..., etc.** *Merge $B-1$ runs.*
Cost of external merge sort

- E.g., with 5 (B) buffer pages, sort 108 (N) page file:

<table>
<thead>
<tr>
<th>Pass</th>
<th>Description</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass 1</td>
<td>[\frac{108}{5}] = 22 sorted runs of 5 pages each (last run is only 3 pages)</td>
<td>[N/B] sorted runs of B pages each</td>
</tr>
<tr>
<td>Pass 2</td>
<td>[\frac{22}{4}] = 6 sorted runs of 20 pages each (last run is only 8 pages)</td>
<td>[\frac{N}{B}] /(B-1) sorted runs of (B(B-1)) pages each</td>
</tr>
<tr>
<td>Pass 3</td>
<td>2 sorted runs, 80 pages and 28 pages</td>
<td>[\frac{N}{B}] /(B-1)^2 sorted runs of (B(B-1)^2) pages</td>
</tr>
<tr>
<td>Pass 4</td>
<td>Sorted file of 108 pages</td>
<td>[\frac{N}{B}] /(B-1)^3 sorted runs of (B(B-1)^3) ((\geq N)) pages</td>
</tr>
</tbody>
</table>

- Number of passes = 1 + \([\log_{B-1} \[N/B\]\])

  Cost = \(2N \times (1 + [\log_{B-1} \[N/B\]\])\)
## Number of passes of external sort

<table>
<thead>
<tr>
<th>N</th>
<th>B=3</th>
<th>B=5</th>
<th>B=9</th>
<th>B=17</th>
<th>B=129</th>
<th>B=257</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10,000,000</td>
<td>23</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
Replacement sort

- **Replacement sort**: Produce initial sorted runs as long as possible. Used in Pass 1 of sorting.

Organize B available buffers:
- 1 buffer for *input*
- B-2 buffers for *current set*
- 1 buffer for *output*
Replacement sort

- Pick tuple \( r \) in the current set (CS) s.t. \( r \) is the *smallest value* in CS that is \( \geq \) *largest value in output*, e.g. 8, to extend the current run.
- Write output buffer out if full, extending the current run.
- Fill the space in current set by adding tuples from input.
- Current run terminates if *every tuple in the current set is < the largest tuple in output*.
Replacement sort

- When used in Pass 1 for sorting, write out sorted runs of size $2B$ on average.
- Affects calculation of the number of passes accordingly.
Using B+ trees for sorting

- Scenario: Table to be sorted has a B+ tree index on sorting attribute(s).
  - Retrieve students in increasing order of age.

- **Idea**: Can retrieve records in order by traversing leaf pages.

- Is this a good idea? Cases to consider:
  - B+ tree is clustered: *Good idea!*
  - B+ tree is not clustered: *Could be a very bad idea!*
Clustered B+ tree used for sorting

- Alternative 1: cost of retrieving all leaf pages
- Alternative 2: also cost of retrieving data records, but reading each page just once.

Almost always better than external sorting!
Unclustered B+ tree used for sorting

- Alternative 2: each data entry contains \textit{rid} of a data record. In general, \textbf{one I/O per data record}!

Worse case I/O: \textit{RN}
\textit{R}: \# records per page
\textit{N}: \# pages in file
## External sorting vs. unclustered index

<table>
<thead>
<tr>
<th>N</th>
<th>Sorting</th>
<th>R=1</th>
<th>R=10</th>
<th>R=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>200</td>
<td>100</td>
<td>1,000</td>
<td>10,000</td>
</tr>
<tr>
<td>1,000</td>
<td>2,000</td>
<td>1,000</td>
<td>10,000</td>
<td>100,000</td>
</tr>
<tr>
<td>10,000</td>
<td>40,000</td>
<td>10,000</td>
<td>100,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>100,000</td>
<td>600,000</td>
<td>100,000</td>
<td>1,000,000</td>
<td>10,000,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td>8,000,000</td>
<td>1,000,000</td>
<td>10,000,000</td>
<td>100,000,000</td>
</tr>
<tr>
<td>10,000,000</td>
<td>80,000,000</td>
<td>10,000,000</td>
<td>100,000,000</td>
<td>1,000,000,000</td>
</tr>
</tbody>
</table>

- **R**: # of records per page
- **R=100** is the more realistic value.
- **Worse case numbers** (R, N) here!

For sorting
- **B=1,000**
Outline

- Selections
- Sorting routine
- Joins
- Projections
- Set operators
- Group By aggregation
Equality joins with one join column

```
SELECT * 
FROM Reserves R, Sailors S 
WHERE R.sid = S.sid 
```

cross product (×)

selection (σ)

too expensive!

semantics
Schema for examples

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

- **Sailors:**
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - 500 pages.

- **Reserves:**
  - Each tuple is 40 bytes long,
  - 100 tuples per page,
  - 1000 pages.

- **Cost metric:** # I/Os
**Simple nested loops join (NLJ)**

foreach tuple \( r \) in \( R \) do

foreach tuple \( s \) in \( S \) do

if \( r_i = s_j \) then

add \( <r, s> \) to result

For each tuple in \( R \), read all of \( S \)

Cost: \( M + (p_R M) N = 1000 + 100 \times 1000 \times 500 = 1,000 + (5 \times 107) \) I/Os.

assuming 10ms/IO, it will take 140 hours
Page-oriented nested loops join

How can we improve Simple NLJ?

- foreach page of R do
  - foreach page of S do
    - write out each matching pair <r, s>
      - //r is in R-page, s is in S-page

For each page in R, read all of S

Cost: $M + MN =

= 1000 + 1000 \times 500

= 501,000 I/Os.

assuming 10ms/IO, it will take 1.4 hours
Page-oriented nested loops join

- How can we improve Simple NLJ?

```java
foreach page of R do
    foreach page of S do
        write out each matching pair <r, s>
        // r is in R-page, s is in S-page
```

Which should be the outer?

How many buffers do we need?
Block nested loops join

- How can we utilize additional buffer pages?
  - If the smaller reln, say R, fits in memory, use R as outer, read the inner S only once.
  - Otherwise, read a big chunk of R each time, hence reducing # times of reading S.

- Block Nested Loops Join:
  - The smaller reln R as outer, the other S as inner.
  - Buffer allocation:
    - 1 buffer for scanning the inner S
    - 1 buffer for output
    - All remaining buffers for holding a “block” of outer R
Block nested loops join (Contd.)

```plaintext
foreach block in R do
foreach page in S do
  foreach matching tuple r in R-block, s in S-page do
    add <r, s> to result
```
Block nested loops join (Contd.)

foreach block in \( R \) do
  build a hash table on \( R \)-block
foreach page in \( S \) do
  foreach matching tuple \( r \) in \( R \)-block, \( s \) in \( S \)-page do
    add \( <r, s> \) to result
Cost of block nested loops join

- **Cost**: Scan of outer + \#outer blocks * scan of inner
  - B buffer pages available
  - Cost = size of outer + \[size of outer / B-2\] * size of inner

- E.g. B=102, Sailors S = 500 pages, Reserves R = 1000 pages.
  - What is the cost if S is outer, R is inner?
    - A block = B-2 = 100 pages
    - Cost = 500 + \[500/100\] * 1000 = 5,500 I/Os.
  - What is the cost if we swap R and S?
    - Cost = 1000 + \[1000/100\] * 500 = 6,000 I/Os.
Index nested loops join

- Given an index on the join column of S:

  ```
  foreach tuple r in R do
    foreach tuple s in S where r == s (via index lookup) do
      add <r, s> to result
  ```

  For each tuple in R, check index

  Cost: $M + (p_R M) \text{index\_lookup}$
Index nested loops join

- Given an index on the join column of S:

  \[
  \text{foreach tuple r in R do} \\
  \quad \text{foreach tuple s in S where } r == s \text{ (via } \text{index lookup}) \text{ do} \\
  \quad \text{add } <r, s> \text{ to result}
  \]

1) Cost of index lookup:
   - \textit{Hash index}: \textasciitilde1.2 I/O to search + pages for matches
   - \textit{B+ tree}: 2-4 I/Os to search + extra pages for matches.

2) Cost of retrieving matching S tuples:
   - \textit{Clustered index}: one or a few I/Os (typical).
   - \textit{Unclustered}: up to 1 I/O per matching S tuple.
Examples of index nested loops

Sailors $\bowtie$ Reserves
- Sailors: tuple size is 50 bytes, 80 tuples per page, 500 pages.
- Reserves: tuple size is 40 bytes, 100 tuples per page, 1000 pages.

- Hash-index (Alt. 2) on $sid$ of Sailors (primary key index):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: # of matching Sailors tuples = 1
    - 1.2 I/Os to get data entry in index
    - 1 I/O to get the *exactly one* matching Sailors tuple.
  - Total: 1000+ 100*1000*2.2 = 221,000 I/Os.
Examples of index nested loops

Sailors \( \bowtie \) Reserves
- Sailors: tuple size is 50 bytes, 80 tuples per page, 500 pages.
- Reserves: tuple size is 40 bytes, 100 tuples per page, 1000 pages.

Hash-index (Alt. 2) on sid of Reserves:
- Scan Sailors: 500 page I/Os, 80*500 tuples.
- For each Sailors tuple: \# of matching Reserves tuples = ?
  - Uniform distribution: 2.5 Reserves tuples/sailor (100,000/40,000).
  - 1.2 I/Os to find the index page with data entries.
  - Cost of retrieving the tuples is 1 or 2.5 I/Os (cluster or not).
- Total: 500+80*500*(2.2~3.7) = 88,500~148,500 I/Os.
Sort-merge \((R \bowtie S)\) for equi-join

- **Sort** R and S on join column using external sorting.
- **Merge** R and S on join column, output result tuples.

Repeat until either R or S is finished:

- **Scanning**:
  - Advance scan of R until current R-tuple \(\geq\) current S tuple,
  - Advance scan of S until current S-tuple \(\geq\) current R tuple;
  - Do this until current R tuple = current S tuple.

- **Matching**:
  - Match all R tuples and S tuples with same value (called R-group and S-group of the current value).
  - Output \(<r, s>\) for all pairs of such tuples.
Example of sort-merge join

Cost: $\text{Sorting\_cost}(R) + \text{Sorting\_cost}(S) + \text{Merging\_cost}$

- Merging\_cost $\in [M+N, M*N]$
- M+N: *foreign key join* with the referenced reln. as inner.
- M*N: uncommon but possible. When?

<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>
Refinement of sort-merge join

- Is there a guaranteed **linear-time** algorithm?

- Observed repeated merging phases:
  - *Sorting* of R and S has respective merging phases.
  - *Join* of R and S also has a merging phase.
  - Combine all these merging phases!
Two-pass sort-merge join

- **Pass 1** *Sorting*: sort subfiles of R and S individually
- **Pass 2** *Merging*: merge sorted runs of R and S
  - merge sorted runs of R,
  - merge sorted runs of S, and
  - compare R and S tuples using the *join condition*.
- Assume that we don’t have too many duplicates...
Merging in two-pass sort-merge

Relation R

Relation S

Run1 of R
Run2 of R
RunK of R
Run1 of S
Run2 of S
RunK of S

B memory buffer pages

OUTPUT

Join Results
Memory requirement and cost

- **Memory requirement for two-pass sort-merge:**
  - Let $U$ be the size of the *larger* relation, $U = \max(M, N)$.
  - *Sorting* pass produces sorted runs of size up to $2B$. So, Number of runs per relation $\leq U/2B$.
  - *Merging* pass holds sorted runs of both relations and an output buffer. So,
    $$2^{\ast}(U/2B) + 1 \leq B \rightarrow B > \sqrt{U}$$

- **Cost:** read & write each relation in sorting pass
  + read each relation in merging pass
  (+ writing result tuples, ignore here) = $3 \ (M+N)$!
Hash-join for equi-join

- **Idea**: For an *equi-join*, partition both R and S using a hash function s.t. R tuples will only match S tuples in partition i.

- **Phase 1 ** *Partitioning*: Partition both relations using hash function $h$ (R$i$ tuples will only match with S$i$ tuples).
Hash-join

- **Phase 2 Probing:**
  - Read in partition Ri, build hash table using h2 (<> h!).
  - Scan partition Si, one page at a time, search for matches.
Memory requirement

- **Partitioning:** # partitions in memory ≤ B-1,
  **Probing:** to fit each Ri in memory, size of partition ≤ B-2.
  - A little more memory needed to build hash table, but ignored here.

- Assuming uniformly sized partitions, $L = \min(M, N)$:
  - $L / (B-1) \leq (B-2) \Rightarrow B > \sqrt{L}$
  - Use the *smaller* relation as the building relation in probing phase.

- What if hash fn h does not partition uniformly?
  - One or more R partitions may not fit in memory.
  - Can apply hash-join recursively to this R-partition and the corresponding S-partition. Higher cost, of course...
Cost of hash-join

- **Partitioning:** reads+writes both relns; \(2(M+N)\).
- **Probing:** reads both relns; \(M+N\) I/Os.

**Total cost = \(3(M+N)\).**

- In our running example, a total of 4500 I/Os using hash join (compared to 501,000 I/Os w. Page NLJ).

**Sort-Merge Join vs. Hash Join:**

- Given a minimum amount of memory (**what is this, for each?**) both have a cost of \(3(M+N)\) I/Os.
- Hash Join superior on this count if relation sizes differ greatly. Assuming \(M<N\), what if \(\sqrt{M} < B < \sqrt{N}\)?
- Sort-Merge less sensitive to data skew; result is sorted.
**Block NLJ:** $L+LU/(B-2)$

**Sort Merge:** $[2L\log_{B-1} L + 2U\log_{B-1} U + 3(L+U), L+U]$  

(Hybrid) Hash Join (no skew):  
$[3(L+U), L+U]$
General join conditions

- Equalities over several attributes (e.g., $R.sid = S.sid \text{ AND } R.rname = S.sname$):
  - Block NL works fine.
  - For Index NL,
    - use index on $<sid, sname>$ if available; or
    - use an index on $sid$ or $sname$, check the other predicate on the fly.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.
General join conditions

- Inequality conditions (e.g., \( R.rname < S.sname \)):
  - For Index NL, need B+ tree index.
    - Range probes on inner; number of matches likely to be much higher than for equality joins.
    - Clustered index is much preferred.
  - Block NL often works well.
  - Hash Join, Sort Merge Join not applicable.
Outline

- Selections
- Sorting routine
- Joins
  - Projections
- Set operators
- Group By aggregation
The projection operation

- Projection consists of two steps:
  - Remove attributes not in the projection list.
  - If `DISTINCT` is specified, eliminate any duplicate tuples.

- Algorithms: *single-relation sorting* and *hashing* based on *all* remaining attributes.

```
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```
Projection based on sorting

- **Sorting pass**: modified to remove unwanted fields.
  - Runs of about 2B pages are produced.
  - But tuples in runs are smaller than input tuples. (Size ratio depends on # and size of fields that are dropped.)

- **Merging passes**: modified to eliminate duplicates.
  - # result tuples smaller than input (depends on # of duplicates.)
Projection based on hashing

- **Partitioning phase**: Partition input relation using \( h1 \); for each tuple, discard unwanted fields.
  - Result is B-1 partitions (of tuples with only wanted fields). 2 tuples from different partitions guaranteed to be distinct.

- **Duplicate elimination phase**: Read each partition, build an in-memory hash table using \( h2 \) on all fields, discard duplicates.
Outline

- Selections
- Sorting routine
- Joins
- Projections
  - Set operators
- Group By aggregation
Intersection

- **Intersection**: Tuples in both reln. 1 and reln. 2.

  Equality join on *all* fields!
Union, Set Difference

- **Union**: Tuples in either reln. 1 or reln. 2.
  - Selects *distinct* values only.

- **Set-difference**: Tuples in reln. 1, but not in reln. 2.

- Both can be implemented using *two-relation sorting* or *hashing*.
  - Details see the textbook...
Outline

- Selections
- Sorting routine
- Joins
- Projections
- Set operators
- Group By aggregation
Aggregate operations (AVG, MIN, etc.)

```
SELECT min(S.age)
FROM   Sailors S
WHERE  S.rating = 10
```

- Aggregation without grouping
  - *File scan*: in general, requires scanning the relation.
  - *Index only scan*: if a tree index’s search key includes all attributes in the SELECT and WHERE clauses.
    - e.g. B+tree on <rating, age>
Aggregate operations (contd.)

- Aggregation with grouping (GROUP BY)
  - *Single-relation sorting*: sort by group-by attribute(s); compute aggregate for each group in last merging phase.
  - *Single-relation hashing*: hash on group-by attribute(s); compute aggregate using in-memory hash table for each partition.
  - *Index only scan*: if a tree index’s search key includes all attributes in SELECT, WHERE and GROUP BY clauses.
  - e.g. B+tree on <rating, age>

```
SELECT min(S.age)
FROM Sailors S
WHERE S.rating > 5
GROUP BY S.rating
```
I/O cost versus number of I/Os

- Cost metric has so far been the number of I/Os.
- Issue 1: effect of sequential (blocked) I/O?
  - Refine external sorting using *blocked I/O*
- Issue 2: parallelism between CPU and I/O?
  - Refine external sorting using *double buffering*
To reduce I/O cost, make each input buffer a block of pages.
- But this will reduce fan-out during merge passes! E.g. from B-1 inputs to (B-1)/2 inputs.
- In practice, most files still sorted in 2-3 passes.
**Double Buffering**

- What happens when an input block has been consumed?

- To reduce wait time for I/O request to complete, can **prefetch into `shadow block'**.
  - Potentially, more passes.
  - In practice, most files **still** sorted in 2-3 passes.
Summary so far

- Selections
- Sorting routine
- Joins
- Projections
- Set operators
- Group By aggregation

Different ways to perform these operations and their cost
\[ N = \frac{((z*2) + ((z*3) + y))}{x} \]

Given \( x = 1 \), \( y = 0 \), and \( z = 4 \), solve for \( N \)

In what order did you perform the operations?
\[ N = \frac{((z*2) + ((z*3) + y))}{x} \]

Given \( x = 1 \), \( y = 0 \), and \( z = 4 \), solve for \( N \)

**but now assume:**
- * costs 10 units
- + costs 2 units
- / costs 50 units

**How will you do the operations now?**
\[
N = \frac{((z \times 2) + ((z \times 3) + y))}{x}
\]

**Algebraic laws:**

1. (+) identity: \( x + 0 = x \)
2. (/) identity: \( x / 1 = x \)
3. (*) distributes: \( n \times (x + n \times y) = n \times (x + y) \)
4. (*) commutes: \( x \times y = y \times x \)

**Assume:**

* costs 10 units
+ costs 2 units
/ costs 50 units

\[
N = (2 + 3) \times z
\]

2 operations instead of 5, no division
Relational algebra tree

<table>
<thead>
<tr>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT S.sname</td>
</tr>
<tr>
<td>FROM Reserves R, Sailors S</td>
</tr>
<tr>
<td>WHERE R.sid=S.sid AND R.bid=100 AND S.rating&gt;5</td>
</tr>
</tbody>
</table>

Expression in Relational Algebra (RA):

\[ \pi_{\text{sname}} (\sigma_{\text{bid}=100 \land \text{rating}>5} (\text{Reserves} \bowtie_{\text{sid}=\text{sid}} \text{Sailors})) \]
Query evaluation plan

- **Query evaluation plan** extends an RA tree with:
  - *access method* for each relation;
  - *implementation method* for each other operator.

- Missed opportunities:
  - Selections could have been ‘pushed’ earlier.
  - More efficient joins.
  - Use of indexes.
Relational algebra equivalences

- **Selections:**
  \[
  \sigma_{c_1 \land \ldots \land c_n}(R) \equiv \sigma_{c_1}(\ldots \sigma_{c_n}(R)) \quad \text{(Cascade)}
  \]
  \[
  \sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R)) \quad \text{(Commute)}
  \]

- **Projections:**
  \[
  \pi_{a_1}(R) \equiv \pi_{a_1}(\ldots (\pi_{a_n}(R))) \quad \text{(Cascade)}
  \]

- **Joins:**
  \[
  (R \bowtie S) \equiv (S \bowtie R) \quad \text{(Commute)}
  \]
  \[
  R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \quad \text{(Associative)}
  \]
More equivalences

- $\sigma_c (R \times S) \equiv R \bowtie_c S$

- $\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$

- $\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$
  - holds if $\sigma$ only uses attributes retained by $\pi$

- For $\pi_b(R \bowtie_a S)$, we can push $\pi$ before $\bowtie$ by retaining only the $a$ attribute and the $b$ attribute (if existent)

☞ But, aggregates do not commute with other operators.
Query plan 1 (selection pushed down)

- Push selections below the join.
- Materialization vs. Pipelining:
  - Store a temporary relation T, if the subsequent join needs to scan T multiple times.
  - The opposite is pipelining.

- With 5 buffer pages, cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings).
  - Sort-Merge join: Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250).
  - Total = 4060 page I/Os.
Query plan 2 (different join method)

- Join uses *block nested loops join* instead.

- With 5 buffer pages, cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages).
  - Scan Sailors (500) + write temp T2 (250 pages).
  - **BNL join**: join cost = 10 + 4 * 250.
  - Total cost = 2770.
Using indexes

- A **tree** index *matches* (a conjunction of) terms if the attributes in the terms form a *prefix* of the index key.
  - Tree index on \(<a, b, c>\)
    - \(a=5\ AND\ b=3\ ?\)
    - \(a=5\ AND\ b>6\ ?\)
    - \(b=3\ ?\)

- A **hash** index *matches* (a conjunction of) terms if there is a term *attribute = value* for *every* attribute in the index key.
  - Hash index on \(<a, b, c>\)
    - \(a=5\ AND\ b=3\ AND\ c=5\ ?\)
    - \(a=5\ AND\ b=3\ ?\)
    - \(a>5\ AND\ b=3\ AND\ c=5\ ?\)
Query plan 3 (using indexes)

- **Selection using index**: clustered index on bid of Reserves.
  - Retrieve 100,000/100 = 1000 tuples
  - Clustering: read 1000/100 = 10 pages.

- Indexed NLJ: *pipeline* the outer and *index lookup* on sid of Sailors.
  - The outer: no need to materialize.
  - The inner: *sid* is a key; *at most one* match tuple, unclustered index OK.

- **Cost**:
  - Selection of Reserves tuples (10 I/Os).
  - For each tuple, get matching Sailors tuple (1000*(1.2+1)).
  - Total = 2210 I/Os.
Highlights of System R optimizer

- **Impact:** most widely used; works well for < 10 joins.

- **Plan Space:** too large, must be pruned.
  - Only considers the space of *left-deep plans*.
  - Avoids cartesian products!

- **Cost of a plan:** approximate art at best.
  - Uses statistics to estimate *cost of an operation* and its *result size*.
  - Considers a combination of CPU and I/O costs.

- **Plan Search:** dynamic programming
  - Prunes useless subplans.
(1) Plan space

- The plans considered are:
  - All *access methods*, for each reln in the FROM clause.
  - All *left-deep join trees*: all the ways to join the relns one-at-a-time, with the inner reln in the FROM clause.
    - All permutations of N relns: $N$ factorial!
(1) Plan space

The plans considered are:

- All *access methods*, for each reln in FROM clause.
- All *left-deep join trees*: all the ways to join the relns one-at-a-time, with the inner reln in the FROM clause.
  - All permutations of N relns: N *factorial*!
  - But avoid *cartesian products*!
    - R.a = S.a and R.b = T.b, how many left-deep trees?
- All *join methods*, for each join in the tree.
- Appropriate places for selections and projections.
  - Not all selections can be pushed before joins.
(2) Cost estimation

- For each plan considered, must estimate its cost.

- Estimate *cost* of each operation in a plan tree:
  - Depends on input cardinalities.
  - Depends on the method (sequential scan, index scan, join method...)

- Estimate *size of result* for each operation in tree:
  - Use statistics about input relations.
Statistics in system catalog

- **Statistics about each relation (R) and index (I):**
  - **Relation cardinality:** # tuples (NTuples) in R
  - **Relation size:** # pages (NPages) in R
  - **Index cardinality:** # distinct values (NKeys) in I
  - **Index size:** # pages (INPages) in I
  - **Index height:** # nonleaf levels (IHeight) of I
  - **Index range:** low/high key values (Low/High) in I
  - **Number of distinct values** in an attribute (NKeys)
  - **Histogram** for an attribute
Size estimation & reduction factors

SELECT attribute list
FROM relation list
WHERE term_1 AND ... AND term_k

- **Reduction factor (RF) or Selectivity** of each term reflects the impact of the term in reducing result size.
  - **Assumption 1**: uniform distribution of the values.
  - Term col=value: RF = 1/NKeys(I), if there is an index I on col.
  - Term col>value: RF = (High(I)-value)/(High(I)-Low(I))
  - Term col1=col2: RF = 1/\( \text{MAX}(\text{NKeys}(I1), \text{NKeys}(I2)) \)

- **Result cardinality** = Max # tuples * product of all RFs.
  - Max # tuples = the product of the cardinalities of relations in the FROM clause.
  - **Assumption 2**: terms are independent.
Issue with assumption 1

• “Uniform distribution of values”: often causes highly inaccurate estimates
  • E.g., distribution of gender: male (15), female (4)
  • E.g., distribution of age:

<table>
<thead>
<tr>
<th>Age</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>9</td>
</tr>
</tbody>
</table>

Nkeys = 15, count = 45.
Reduction factor of ‘age=14’: 1/15? 9/45=1/5!

• Histogram: approximates a data distribution
### Equiwidth histograms

**Equiwidth Histogram:** buckets of equal size.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>8/3</th>
<th>4/3</th>
<th>15/3</th>
<th>3/3</th>
<th>15/3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counts</td>
<td>8</td>
<td>4</td>
<td>15</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>Buckets</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Still not accurate for value 14 with true frequency 9
Equidepth histograms

**Equidepth**: roughly equal counts in buckets.

Favors *frequent* values.

What if we have 100 distinct values and 20 frequent values?
- Keep a list of (MCVs) with their frequencies.
- The histogram excludes MCVs.

Small errors for infrequent items: tolerable.

Now accurate for value 14: 9
**Equidepth histograms**

**Equidepth**: roughly equal counts in buckets.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>9/4</th>
<th>10/4</th>
<th>10/2</th>
<th>7/4</th>
<th>9/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counts</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Buckets</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

- Implementation is often sampling-based:
  - Boundaries of 5 buckets \{0, 4, 8, 10, 14, 14\}
  - Count of tuples for each bucket (optional)
  - Number of distinct values for each bucket (optional)

Small errors for infrequent items: tolerable.

Now accurate for value 14: 9
Issue with assumption 2

- **Independence of predicates**: often causes inaccurate estimates
  - E.g., Car DB: 10 makes, 100 models.
  - RF of make=‘honda’ and model=‘civic’
  - If independent, 1/10 * 1/100. In practice, much higher!

- **Multi-dimensional histograms** [PI’97, MVW’98, GKT’00]
  - Maintain counts and frequency in multi-attribute space.

- **Dependency-based histograms** [DGR’01]
  - Learn dependency between attributes and compute conditional probability $P(\text{model}=\text{‘civic’} \mid \text{make}=\text{‘honda’})$
  - Can use graphical models...
(3) Queries over a single relation

- Query involves selection, projection, and aggregation.

- **Enumeration of alternative plans:**
  1. Each *available access path* (file/index scan) is considered, the one with least estimated cost is chosen.
  2. Various operations are often carried out together:
     - If an index is used for a selection, projection is done for each retrieved tuple.
     - If no GROUP BY, the resulting tuples can be *pipelined* into the aggregate computation.
     - Otherwise, hashing or sorting is needed for GROUP BY.
(4) Queries over multiple relations

- As the number of joins increases, the number of alternative plans grows rapidly.

- System R: (1) use *only left-deep join trees*, (2) avoid *cartesian products*.
  - Allow *pipelined* plans; intermediate results not written to temporary files.
  - Not all left-deep trees are fully pipelined!
    - Sort-Merge join: sorting phase
    - Two-phase hash join: partitioning phase
Plan search

- Left-deep join plans differ in:
  - the order of relations,
  - the access path for each relation, and
  - the join method for each join.
  - where selections are placed.

- Many of these plans share common prefixes, so don’t enumerate all of them. This is a job for...

- **Dynamic Programming**
  “a method of solving problems that exhibit the properties of overlapping subproblems and optimal substructure.”
Enumeration of left-deep plans

**Enumerate with N passes** (if N relations are joined):

- **Base**: Find best 1-relation plan for each relation.
- **Pass 1**: Find best ways to join result of each 1-relation plan (as *outer*) to another relation. *(All 2-relation plans.)*
- ...
- **Pass N-1**: Find best ways to join result of a (N-1)-relation plan (as *outer*) to the N’th relation. *(All N-relation plans.)*

**For each subset of relations**, retain only:

- cheapest *unordered* plan, and
- cheapest plan for each *interesting order* (order for final output or a subsequent op. using sorting ) of the tuples.
A $k$-way ($k<N$) plan is not combined with an additional relation unless there is a join condition between them.

- Do it until all predicates in WHERE have been used.
- That is, avoid cartesian products if possible.

- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using an `interestingly ordered` plan, or an additional sorting or hashing.
What about nested queries?

- **Nested query**: appears as an operand of a predicate in WHERE.

- **Nested query with no correlation**: does not contain a reference to the tuple from the outer.
  - A nested query needs to be evaluated *only once*.
  - The optimizer arranges it to be evaluated before the top level query.

```sql
SELECT S.sname
FROM Sailors S
WHERE S.rating > (SELECT Avg(rating)
                   FROM Sailors)
```

```sql
(SELECT Avg(rating)
 FROM Sailors)
```

```sql
SELECT S.sname
FROM Sailors S
WHERE S.rating > value
```
Nested queries with correlation

- **Nested query with correlation**: contains a reference to a tuple from the outer.
  - Nested block is optimized independently, with the value of outer tuple considered as a constant.
  - The nested block is evaluated a tuple-at-a-time.

```sql
SELECT S.sname
FROM Sailors S
WHERE EXISTS
  (SELECT *
   FROM Reserves R
   WHERE R.bid=103
   AND R.sid=S.sid)
```

**Nested block to optimize:**

```sql
(SEL ECT *
 FROM Reserves R
 WHERE R.bid =103
 AND S.sid = outer value)
```
Unnesting

- **Guideline:** Use only one “query block”, if possible.

```
SELECT DISTINCT *  
FROM    Sailors S  
WHERE   S.sname IN  
        (SELECT Y.sname  
         FROM YoungSailors Y)
```

- Not always possible ...

```
SELECT *  
FROM    Sailors S  
WHERE   S.sname IN  
        (SELECT DISTINCT Y.sname  
         FROM YoungSailors Y)
```