Outline

- Selection
- Sorting routine
- Join
  - Projection
- Set operator
- Group By aggregation
Equality Joins With One Join Column

- R ⋈ S, natural join. Very common operation!
- Semantics: cross product (×) followed by selection (σ)
  - If R × S is large, R × S followed by a selection is inefficient.
  - Must be carefully optimized.
- Cost metric: # of I/Os. Ignore output cost in analysis.
  - R: M pages, pR tuples per page
  - S: N pages, pS tuples per page.

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

Sailors ($sid$: integer, $sname$: string, $rating$: integer, $age$: real)
Reserves ($sid$: integer, $bid$: integer, $day$: date, $rname$: 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
1) Page-Oriented Nested Loops Join

- A baseline approach:

```
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
```

- Cost: \( M + M \times N = 1000 + 1000 \times 500 = 501,000 \) I/Os.
  - 2M random I/Os; others are sequential I/Os.

- How many buffer pages do we need?
  - 3 buffer pages!
2) 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.)

**foreach** block in R **do**
build a hash table on R-block (optional)

**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. If M < N
  - Cost = M + \[ \left\lceil \frac{M}{B} \right\rceil - 2 \] * N

- 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 + \[ \left\lceil \frac{500}{100} \right\rceil \] * 1000 = 5,500 I/Os.
  - What is the cost if we swap R and S?
    - Cost = 1000 + \[ \left\lceil \frac{1000}{100} \right\rceil \] * 500 = 6,000 I/Os.
  - Which relation should be the outer for smaller cost?
3) Index Nested Loops Join

- Given an index on the join column of one relation, say S:

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

- **Cost:** \( M + (M \cdot p_R \cdot \text{cost of finding matching S tuples}) \)
  1) Cost of search in S index:
     - *Hash index*: about 1 I/O to search + extra pages for matches
     - *B+ tree*: 2-4 I/O's to search + extra pages for matches.
  2) Cost of retrieving matching S tuples (assuming Alt. 2 or 3):
     - *Clustered index*: one or a few I/O's (typical).
     - *Unclustered*: up to 1 I/O per matching S tuple.
4) 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

| sid | bid | day     | rname | | sid | rname | rating | age |
|-----|-----|---------|-------| |-----|-------|--------|-----|
| 28  | 103 | 12/4/96 | guppy | 22  | dustin | 7      | 45.0|
| 28  | 103 | 11/3/96 | yuppy | 28  | yuppy  | 9      | 35.0|
| 31  | 101 | 10/10/96| dustin| 31  | lubber | 8      | 55.5|
| 31  | 102 | 10/12/96| lubber| 44  | guppy  | 5      | 35.0|
| 31  | 101 | 10/11/96| lubber| 58  | rusty  | 10     | 35.0|
| 58  | 103 | 11/12/96| dustin|      |        |        |     |

- **Cost**: \(\text{Sorting\_cost}(R) + \text{Sorting\_cost}(S) + \text{Merging\_cost}\)
  - \(\text{Merging\_cost} \in [M+N, M*N]\)
  - \(M+N\): *foreign key join* with the referenced reln. as inner.
  - \(M*N\): uncommon but possible. When?
- **What is the I/O pattern in the sort-merge join?**
- **How many buffers are needed in the merge phase?**
Refinement of Sort-Merge Join

- When can we achieve a 2-pass algorithm for foreign key-primary key join?
  - Pass 1: create sorted runs of R and S like in external sorting
  - Pass 2: combine repeated merging phases into one pass
    - Sorting of R and S has respective merging phases.
    - Join of R and S also has a merging phase.
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
Merging in Two-Pass Sort-Merge

Relation R

Relation S

Join Results

B memory buffer pages

Relation R

Relation S

OUTPUT

Join Results

B memory buffer pages

Relation R

Relation S

OUTPUT

Join Results

B memory buffer pages

Relation R

Relation S

OUTPUT

Join Results

B memory buffer pages

Relation R

Relation S

OUTPUT

Join Results

B memory buffer pages

Relation R

Relation S

OUTPUT

Join Results

B memory buffer pages
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*.

- There exists a linear (2-pass) algorithm for *foreign key-primary key join* if *certain conditions hold*
Memory Requirement and Cost

- Memory requirement for two-pass sort-merge:
  - **Sorting** pass produces sorted runs of size up to $2B$. So, Number of runs = $(M+N)/2B$.
  - **Merging** pass holds sorted runs of both relations and an output buffer. So, $(M+N)/2B + 1 \leq B \rightarrow B > \sqrt{(M+N)/2}$
  - Sometimes, can see a looser bound $B > \sqrt{U}$, $U = \max(M,N)$

- **Cost:** read & write each relation in sorting pass + read each relation in merging pass
  = $3(M+N)$!
  (+ writing result tuples, ignored here)
5) **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<sub>i</sub> tuples will only match with S<sub>i</sub> 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, \( L = \min(M, N) \):
- \( L / (B-1) \leq (B-2) \rightarrow B > \sqrt{L} + 1 \)
- 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…

- What is the I/O pattern in the hash join?
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.
  - Sort-Merge less sensitive to data skew; result is sorted.
Rough Comparisons of Join Methods

- **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)**: $[3(L+U), L+U]$

# I/Os
- $L+L*U$
- $2L\log_{2}L+$
- $2U\log_{2}U+$
- $3(L+U)$
- $3(L+U)$
- $L+U$

Buffer size $B$
General Join Conditions

- Equalities over several attributes (e.g., \( R\.sid=S\.sid \) AND \( R\.rname=S\.sname \)):
  - Block NL works fine.
  - For Index NL,
    - use index on \(<\text{sid}, \text{sname}>\) if available; or
    - use an index on \text{sid} or \text{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

- Selection
- Sorting routine
- Join
- Projection
- Set operators
  - Group By aggregation
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>

```sql
SELECT min(S.age)
FROM     Sailors S
WHERE    S.rating = 10
```
Aggregate Operations (contd.)

<table>
<thead>
<tr>
<th>SELECT</th>
<th>\texttt{min}(S.age)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>Sailors S</td>
</tr>
<tr>
<td>WHERE</td>
<td>S.rating &gt; 5</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>S.rating</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SELECT</th>
<th>\texttt{count}(distinct C.userid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>Clicks C</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>C.url</td>
</tr>
</tbody>
</table>

- Aggregation with grouping (GROUP BY)
  - \textit{Single-relation sorting}: sort by group-by attribute(s); compute aggregate for each group in last merging phase.
  - \textit{Single-relation hashing}: hash on group-by attribute(s): compute aggregate using in-memory hash table for each partition.
  - \textit{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>
A Baseline Sorting Alg. for Group by

- Step 1: sort the entire relation by R.a
- Step 2: scan the sorted relation, find each group on the fly, compute aggregate on R.b

\[
\text{SELECT } \text{sum}(R.b) \\
\text{FROM } R \\
\text{GROUP BY } R.a
\]

- The above algorithm needs to scan R many times.
- Is it possible to limit the algorithm to 1 pass?
  - When memory size B >= relation size N
- Is it possible to limit the algorithm to 2 passes?
  - See next.
**Two-Pass Sorting for Group By**

- **Pass 1:** Produce $\lceil N/B \rceil$ sorted runs of $B$ pages each.
- **Pass 2:** Merge $B-1$ runs to produce a final sorted file directly. Find each group on the fly and compute the aggregate.
- To finish in 2 passes, we have $\lceil N/B \rceil <= B-1$, or $B > \sqrt{N}$
Two-Pass Hashing for Group By

- **Idea:** when $|R| = N > B$, partition R using a hash function s.t. the $i$th partition can be processed in memory later.

- Phase 1 (*Partitioning*): Partition R into $B-1$ buckets using hash function $h$.
Two-Pass Hashing for Group By

- Phase 2 (*Probing*):
  - Read in partition $R_i$, find groups (using in-memory sorting or building a hash table using $h_2 <> h_1$), compute the aggregate in each group
  - To enable $R_i$ to fit in memory, we have $\frac{N}{B-1} \leq B-1$, or $B > \sqrt{N}$

- IO cost of two-phase hashing: $3N$ (ignore the output cost)
Comparison of Hash Algorithms

Can we do better than 3N given more memory than \( \sqrt{N} \)?

Two-phase hash: 3N

One-phase hash: N

Buffer size B