IMPLEMENTATION OF RELATIONAL OPERATIONS
(BASED ON SLIDES FROM UC BERKELEY)
Join Operators
Join Operators

• Joins are a very common query operation.
• Joins can be very expensive:
  Consider an inner join of R and S each with 1M records. Q: How many tuples in the answer? (cross product in worst case, 0 in the best(?))

• Many join algorithms have been developed
• Can have very different join costs.
Equality Joins With One Join Column

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

• Assume:
  – M = 1000 pages in R, \( p_R = 100 \) tuples per page.
  – N = 500 pages in S, \( p_S = 80 \) tuples per page.
  – In our examples, R is Reserves and S is Sailors.

• Cost metric: \# of I/Os. We will ignore output costs.

• We will consider more complex join conditions later.
Simple Nested Loops Join

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 the outer relation R, we scan the entire inner relation S.
• How much does this Cost?
• \((p_R \times M) \times N + M = 100,000 \times 500 + 1000\) I/Os.
  – At 10ms/IO, Total: ???
• What if smaller relation (S) was outer?
• \((p_S \times N) \times M + N = 40,000 \times 1000 + 500\) I/Os.
• What assumptions are being made here?
Page-Oriented Nested Loops Join

foreach page b_R in R do
  foreach page b_S in S do
    foreach tuple r in b_R do
      foreach tuple s in b_S do
        if r_i == s_j then add <r, s> to result

• For each page of R, get each page of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.

• What is the cost of this approach?

• \( M*N + M = 1000*500 + 1000 \)
  – If smaller relation (S) is outer, cost = 500*1000 + 500
Block Nested Loops Join

• Page-oriented NL doesn’t exploit extra buffers.

• **Alternative approach**: Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ‘‘block’’ of outer R.

• For each matching tuple $r$ in R-block, $s$ in S-page, add $<r, s>$ to result. Then read next R-block, scan S, etc.
Examples of Block Nested Loops

• **Cost:**
  Scan of outer + \( \# \text{outer blocks} \times \text{scan of inner} \)
  - \( \# \text{outer blocks} = \text{ceiling}(\# \text{pages of outer}/\text{blocksize}) \)

• **With Reserves (R) as outer, and 100 pages/Block:**
  - Cost of scanning R is 1000 I/Os; a total of 10 blocks.
  - Per block of R, we scan Sailors (S); 10*500 I/Os.

• **With 100-page block of Sailors as outer:**
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
Index Nested Loops Join

foreach tuple r in R do
  foreach tuple s in S where r_i == s_j do
    add <r, s> to result

- If there is an index on the join column of one relation
  (say S), can make it the inner and exploit the index.
  - Cost: $M + \left( M \times \rho_R \right) \times \text{cost of finding matching S tuples}$
- For each R tuple, cost of probing S index is about 1.2
  for hash index, 2-4 for B+ tree.
- Cost of then finding S tuples (assuming Alt. (2) or (3)
  for data entries) depends on clustering.
- Clustered index: 1 I/O per page of matching S tuples.
- Unclustered: up to 1 I/O per matching S tuple.
Examples of Index Nested Loops

• Hash-index (Alt. 2) on sid of Sailors (as inner):
  – Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  – For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. **Total:**

• Hash-index (Alt. 2) on sid of Reserves (as inner):
  – Scan Sailors: 500 page I/Os, 80*500 tuples.
  – For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. **Assuming uniform distribution,** 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.
  – **Totals:**
Sorting large files

• Before we continue, let’s think how we can sort a large file stored on Disk.
  – Have to use a block based algorithm
  – Need to bring data in RAM to do some sorting
  – What if the file fits in memory? What if it does not?
2-Way Sort

- **Pass 0:** Read a page, sort it, write it.
  - only one buffer page is used
- **Pass 1, 2, ..., etc.:**
  - requires 3 buffer pages
  - merge pairs of runs into runs twice as long
  - three buffer pages used.

![Diagram of 2-Way Sort](image)
Two-Way External Merge Sort

• Each pass we read + write each page in file.

• N pages in the file => the number of passes

\[= \lceil \log_2 N \rceil + 1\]

• So total cost is:

\[2N\left(\lceil \log_2 N \rceil + 1\right)\]

• **Idea:** Divide and conquer: sort subfiles and merge
General External Merge Sort

More than 3 buffer pages. How can we utilize them?

• To sort a file with $N$ pages using $B$ buffer pages:
  – Pass 0: use $B$ buffer pages. Produce $\lceil \frac{N}{B} \rceil$ sorted runs of $B$ pages each.
  – Pass 1, 2, ..., etc.: merge $B-1$ runs.
Cost of External Merge Sort

• Number of passes: \( 1 + \lceil \log_{B-1} \left[ \frac{N}{B} \right] \rceil \)
• Cost = \( 2N \times \) (# of passes)
• E.g., with 5 buffer pages, to sort 108 page file:
  – Pass 0: = 22 sorted runs of 5 pages each
    (last run is only 3 pages) \[108 / 5\]
• Now, do four-way (B-1) merges
  – Pass 1: = 6 sorted runs of 20 pages each (last
    run is only 8 pages) \[22 / 4\]
  – Pass 2: 2 sorted runs, 80 pages and 28 pages
  – Pass 3: Sorted file of 108 pages
### Number of Passes of External Sort

(I/O cost is $2N$ times number of passes)

<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>
Sorting in practice

• There are many other improvements of the basic algorithm
• Double buffering, etc
• In practice assuming a reasonable size buffer, sorting takes 2-3 passes.
Sort-Merge Join (R ♻ S)

• Sort R and S on the join column, then scan them to do a ``merge’’ (on join col.), and output result tuples.

• Particularly 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

• R is scanned once; each S group is scanned once per matching R tuple.
Example of Sort-Merge Join

- **Cost:** Sort S + Sort R + (M+N)
  - The cost of merging: usually M+N,
    - worst case is M*N (but very unlikely!)
- **With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.**
  (BNL cost: 2500 to 15000 I/Os)
Refinement of Sort-Merge Join

• We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
  – Pass 0 as before, but apply to both R then S before merge.
  – If \( B > \sqrt{L} \), where \( L \) is the size of the larger relation, using the sorting refinement that produces runs of length \( 2B \) in Pass 0, #runs of each relation is < \( B/2 \).
  – In “Merge” phase: Allocate 1 page per run of each relation, and ‘merge’ while checking the join condition
  – Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
    – In example, cost goes down from 7500 to 4500 I/Os.

• In practice, the I/O cost of sort-merge join, like the cost of external sorting, is *linear*. 
Impact of Buffering

• If several operations are executing concurrently, estimating the number of available buffer pages is guesswork.
• Repeated access patterns interact with buffer replacement policy.
  – e.g., Inner relation is scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold inner, replacement policy does not matter. Otherwise, MRU is best, LRU is worst (*sequential flooding*).
  – Does replacement policy matter for Block Nested Loops?
  – What about Index Nested Loops? Sort-Merge Join?
Hash-Join

- Partition both relations on the join attributes using hash function $h$.
- $R$ tuples in partition $R_i$ will only match $S$ tuples in partition $S_i$.

For $i = 1$ to $\#\text{partitions}$ {
  Read in partition $R_i$ and hash it using $h2$ (not $h$).
  Scan partition $S_i$ and probe hash table for matches.
}
Observations on Hash-Join

• \#partitions \( k < B \), and \( B-1 > \) size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing \( k \), we get:
  \[ k = B-1, \quad \text{and} \quad M/(B-1) < B-2, \quad \text{i.e.,} \quad B \text{ must be } > \sqrt{M} \]

• Since we build an in-memory hash table to speed up the matching of tuples in the second phase, a little more memory is needed.

• If the hash function does not partition uniformly, one or more \( R \) partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this \( R \)-partition with corresponding \( S \)-partition.
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.

- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory both have a cost of $3(M+N)$ I/Os. Hash Join superior if relation sizes differ greatly (e.g., if one reln fits in memory). Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.
Set Operations

- Intersection and cross-product special cases of join.
- Union (Distinct) and Except similar; we’ll do union.

- 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.

- 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 corr. R-partition and add tuples to table while discarding duplicates.
General Join Conditions

• Equalities over several attributes (e.g., \( R.sid=S.sid \text{ AND } R.rname=S.sname \)):
  – For Index NL, build index on \(<sid, sname>\) (if S is inner); or use existing indexes on \(sid\) or \(sname\).
  – For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

• Inequality conditions (e.g., \( R.rname < S.sname \)):
  – For Index NL, need (clustered!) B+ tree index.
    • Range probes on inner; \# matches likely to be much higher than for equality joins.
  – Hash Join, Sort Merge Join not applicable!
  – Block NL quite likely to be the best join method here.
Aggregation Operators
Schema for Examples

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

• Similar to old schema; *rname* added for variations.
• Reserves:
  – Each tuple is 40 bytes long, 100 tuples per page, 1000 pages. So, \( M = 1000, p_R = 100 \).
• Sailors:
  – Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  – So, \( N = 500, p_S = 80 \).
Aggregate Operations (\(\text{AVG, MIN, etc.}\))

- **Without grouping:**
  - In general, requires scanning the relation.
  - Given a tree index whose search key includes all attributes in the `SELECT` or `WHERE` clauses, can do index-only scan.

- **With grouping:**
  - Sort on group-by attributes, then scan relation and compute aggregate for each group. (Better: combine sorting and aggregate computation.)
  - Similar approach based on hashing on group-by attributes.
  - Given a tree index whose search key includes all attributes in `SELECT`, `WHERE` and `GROUP BY` clauses, can do index-only scan; if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order.
Sort GROUP BY: Naïve Solution

- The Sort iterator naturally permutes its input so that all tuples are output in sequence.
- The Aggregate iterator keeps running info ("transition values" or "transVals") on agg functions in the SELECT list, per group. Example transVals:
  - For COUNT, it keeps count-so-far.
  - For SUM, it keeps sum-so-far.
  - For AVERAGE it keeps sum-so-far and count-so-far.
- As soon as the Aggregate iterator sees a tuple from a new group:
  1. It produces an output for the old group based on the agg function.
     E.g. for AVERAGE it returns (sum-so-far/count-so-far).
  2. It resets its running info.
  3. It updates the running info with the new tuple’s info.
Sort GROUP BY: Naïve Solution

A, 3
B, 2
C, 1
D, 1
Hash GROUP BY: Naïve Solution
(similar to the Sort GROUPBY)

- The Hash iterator permutes its input so that all tuples are output in groups.
- The Aggregate iterator keeps running info ("transition values" or "transVals") on agg functions in the SELECT list, per group
  - E.g., for COUNT, it keeps count-so-far
  - For SUM, it keeps sum-so-far
  - For AVERAGE it keeps sum-so-far and count-so-far
- When the Aggregate iterator sees a tuple from a new group:
  1. It produces an output for the old group based on the agg function
     E.g. for AVERAGE it returns (sum-so-far/count-so-far)
  2. It resets its running info.
  3. It updates the running info with the new tuple’s info
External Hashing

- **Partition:**
  Each group will be in a single disk-based partition file. But those files have many groups inter-mixed.

- **Rehash:**
  For Each Partition i:
  hash i into an in-memory hash table
  Return results until records exhausted then i++
We Can Do Better!

• Put summarization into the hashing process
  – During the ReHash phase, don’t store tuples, store pairs of the form \(<\text{GroupVals}, \text{TransVals}>\)
  – When we want to insert a new tuple into the hash table
    • If we find a matching GroupVals, just update the TransVals appropriately
    • Else insert a new \(<\text{GroupVals,TransVals}>\) pair

• What’s the benefit?
  – Q: How many pairs will we have to maintain in the rehash phase?
  – A: Number of \(\textbf{distinct values}\) of GroupVals columns
    • Not the number of tuples!!
  – Also probably “narrower” than the tuples
We Can Do Even Better Than That: Hybrid Hashing

- What if the set of $<\text{GroupVals}, \text{TransVals}>$ pairs fits in memory?
  - It would be a waste to spill all the tuples to disk and read them all back back again!
  - Recall $<G,T>$ pairs may fit even if there are tons of tuples!
- Idea: keep $<G,T>$ pairs for a smaller 1st partition in memory during phase 1!
  - Output its stuff at the end of Phase 1.
  - Q: how do we choose the number of buffers ($k$) to allocate to this special partition?

![Diagram of hybrid hashing](image)

- Original Relation
- Disk
- $B$ main memory buffers
- OUTPUT
- Partitions
- Disk
A Hash Function for Hybrid Hashing

- Assume we like the hash-partition function \( h_p \).
- Define \( h_h \) operationally as follows:
  - \( h_h(x) = 1 \) if \( x \) maps to a \(<G,T>\) already in the in-memory hashtable
  - \( h_h(x) = 1 \) if in-memory hashtable is not yet full (add new \(<G,T>\))
  - \( h_h(x) = h_p(x) \) otherwise
- This ensures that:
  - Bucket 1 fits in \( k \) pages of memory
  - If the entire set of distinct hashtable entries is smaller than \( k \), we do no spilling!
Projection (DupElim)

• Issue is removing duplicates.
• Basic approach is to use sorting
  – 1. Scan R, extract only the needed attrs (why do this 1st?)
  – 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 * 2 * 250 + 250 = 2500 I/Os
• Can improve by modifying external sort algorithm:
  – Modify Pass 0 of external sort to eliminate unwanted fields.
  – Modify merging passes to eliminate duplicates.
  – Cost: for above case: read 1000 pages, write out 250 in runs of 40 pages, merge runs = 1000 + 250 + 250 = 1500.

```
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```
DupElim Based on Hashing

• Just like our discussion of GROUP BY and aggregation from before!
  – But the aggregation function is missing

  – SELECT DISTINCT R.sid, R.bid FROM Reserves R
  – SELECT R.sid, R.bid FROM Reserves R GROUP BY R.sid, R.bid

• Cost for Hashing? Without “hybrid”
  – assuming partitions fit in memory (i.e. #bufs >= square root of the #of pages of projected tuples)
  – read 1000 pages and write out partitions of projected tuples (250 pages)
  – Do dup elim on each partition (total 250 page reads)
  – Total : 1500 I/Os.

• With “hybrid hash”: subtract the I/O costs of 1st partition
DupElim & Indexes

- If an index on the relation contains all wanted attributes in its search key, can do \textit{index-only} scan.
  - Apply projection techniques to data entries (much smaller!)
- If an ordered (i.e., tree) index contains all wanted attributes as \textit{prefix} of search key, can do even better:
  - Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates.

- Same tricks apply to GROUP BY/Aggregation
Summary

• **Queries are composed of a few basic operators**;
  – The implementation of these operators can be carefully tuned (and it is important to do this!).
  – Operators are “plug-and-play” due to the *Iterator* model.

• Many alternative implementation techniques for each operator; no universally superior technique for most.

• Must consider alternatives for each operation in a query and choose best one based on statistics, etc.

• This is part of the broader task of Query Optimization, which we will cover next!