One of the advantages of being disorderly is that one is constantly making exciting discoveries.

A. A. Milne
Questions

- **We learned that the same query can be written many ways.**
  - How does DBMS decide which is best?

- **We learned about tree & hash indexes.**
  - How does DBMS know when to use them?

- **Sometimes we need to sort data.**
  - How to sort more data than will fit in memory?
Review: Query Processing

- Queries start out as SQL
- Database translates SQL to one or more Relational Algebra plans
- Plan is a tree of operations, with access path for each
- Access path is how each operator gets tuples
  - If working directly on table, can use scan, index
  - Some operators, like sort-merge join, or group-by, need tuples sorted
  - Often, operators pipelined, getting tuples that are output from earlier operators in the tree
- Database estimates cost for various plans, chooses least expensive
Review: Operator Costs

- **Selection** – either scan all tuples, or use index

- **Projection** – expensive only if duplicates eliminated, requires sorting or hashing

- **Join** – several algorithms:
  - Nested Loops
  - Indexed Nested Loops
  - Sort-Merge
  - Hash Join

```
Reserves (sid=sid) → Sailors (On-the-fly)
  bid=100 ∧ rating > 5  (On-the-fly)
   \( \prod_{\text{sname}} \) (Simple Nested Loops)
```

Review: Cost Estimation

- **To compute operator costs, DBMS needs:**
  - Sizes of relations
  - Info on indexes (type, size)
  - Info on attribute values (high, low, distr, etc.)
  - This info stored in catalogs

- **Cost often related to input size**
  - Need to compute *reduction factor*, how big output will be compared to how big it could be

- **Query costs can vary hugely between plans**
What we’ll cover:

• Efficient sorting

• Different join algorithms

• Later – developing query plans, computing costs
Why Sort?

• A classic problem in computer science!

• Database needs it in order
  – e.g., find students in increasing gpa order
  – first step in bulk loading B+ tree index.
  – eliminating duplicates
  – aggregating related groups of tuples
  – Sort-merge join algorithm involves sorting.

• Problem: sort 1Gb of data with 1Mb of RAM.
  – why not virtual memory?
Streaming Data Through RAM

- An important detail for sorting & other DB operations
- Simple case:
  - Compute $f(x)$ for each record, write out the result
  - Read a page from INPUT to Input Buffer
  - Write $f(x)$ for each item to Output Buffer
  - When Input Buffer is consumed, read another page
  - When Output Buffer fills, write it to OUTPUT
- **Reads and Writes are not coordinated**
  - E.g., if $f()$ is Compress(), you read many pages per write.
  - E.g., if $f()$ is DeCompress(), you write many pages per read.
2-Way Sort

- **Pass 0**: Read a page, sort it, write it.
  - only one buffer page is used (as in previous slide)
- **Pass 1, 2, ..., etc.**:
  - requires 3 buffer pages
  - merge pairs of runs into runs twice as long
  - three buffer pages used.
Two-Way External Merge Sort

- Each pass we read + write each page in file.
- \( N \) pages in the file \( \Rightarrow \) 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
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 $\left\lceil \frac{N}{B} \right\rceil$ sorted runs of $B$ pages each.
- Pass 1, 2, ..., etc.: merge $B-1$ runs.
Cost of External Merge Sort

- **Number of passes:**
- **Cost =** \(2N \times (\text{# of passes})\)
- **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)
  \[\lceil 108 / 5 \rceil\]
- **Now, do four-way (B-1) merges**
  - Pass 1: \(= 6\) sorted runs of 20 pages each (last run is only 8 pages)
  \[\lceil 22 / 4 \rceil\]
  - 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>
Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.
- Keep two heaps in memory, H1 and H2
  
  read B−2 pages of records, inserting into H1; 
  while (records left) {
    m = H1.removemin();  put m in output buffer;
    if (H1 is empty)
      H1 = H2;  H2.reset();  start new output run;
    else
      read in a new record r (use 1 buffer for input pages);
      if (r < m)  H2.insert(r);
      else  H1.insert(r);
  }
  H1.output();  start new run;  H2.output();
More on Heapsort

- **Fact:** average length of a run is $2(B-2)$
  - The “snowplow” analogy
- **Worst-Case:**
  - What is min length of a run?
  - How does this arise?
- **Best-Case:**
  - What is max length of a run?
  - How does this arise?
- Quicksort is faster, but ... longer runs often means fewer passes!
I/O for External Merge Sort

- Actually, doing I/O a page at a time
  - Not an I/O per record
- In fact, read a block (chunk) of pages sequentially!
- Suggests we should make each buffer (input/output) be a chunk of pages.
  - But this will reduce fan-out during merge passes!
  - In practice, most files still sorted in 2-3 passes.
<table>
<thead>
<tr>
<th>N</th>
<th>B=1,000</th>
<th>B=5,000</th>
<th>B=10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10,000</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>100,000</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1,000,000</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000,000</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

- Block size = 32, initial pass produces runs of size 2B.
Sorting Records!

- Sorting has become a competition between labs, universities, DBMS vendors
  - Parallel sorting is the name of the game ...
- **Minute Sort:** how many 100-byte records can you sort in a minute?
  - Typical DBMS: 10MB (~100,000 records)
  - Current World record: 116 GB
    - 80 Itanium2s running Linux
- **Terabyte Sort:** how fast to sort 1TB
  - Current record 7.25 minutes (80 Itanium2s)

Using B+ Trees for Sorting

- **Scenario:** Table to be sorted has B+ tree index on sorting column(s).
- **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

- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.

Better than external sorting!
Unclustered B+ Tree Used for Sorting

- Alternative (2) for data entries; each data entry contains \textit{rid} of a data record. In general, one I/O per data record!
## External Sorting vs. Unclustered Index

<table>
<thead>
<tr>
<th>N</th>
<th>Sorting</th>
<th>p=1</th>
<th>p=10</th>
<th>p=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>

- $p$: # of records per page
- $B=1,000$ and block size=32 for sorting
- $p=100$ is the more realistic value.
• External sorting is important; DBMS may dedicate part of buffer pool for sorting!
• External merge sort minimizes disk I/O cost:
  – Pass 0: Produces sorted runs of size $B$ (# buffer pages).
    Later passes: merge runs.
  – # of runs merged at a time depends on $B$, and block size.
  – Larger block size means less I/O cost per page.
  – Larger block size means smaller # runs merged.
  – In practice, # of passes rarely more than 2 or 3.
Sorting – Review (cont)

• Choice of internal sort algorithm may matter:
  – Quicksort: Quick!
  – Heap/tournament sort: slower (2x), longer runs

• The best sorts are wildly fast:
  – Despite 40+ years of research, we’re still improving!

• Clustered B+ tree is good for sorting; unclustered tree is usually very bad.
Joins

- How does DBMS join two tables?
- Sorting is one way...
- Database must choose best way for each query
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,
  - $M = 1000$ pages total.
- Sailors:
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - $N = 500$ pages total.
Equality Joins With One Join Column

\[
\text{SELECT} \quad * \\
\text{FROM} \quad \text{Reserves} \ R1, \ \text{Sailors} \ S1 \\
\text{WHERE} \quad R1.\text{sid}=S1.\text{sid}
\]

- **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.

- **Assume:** \( M \) tuples in \( R \), \( p_R \) tuples per page, \( N \) tuples in \( S \), \( p_S \) tuples per page.
  - 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.
Simple Nested Loops Join

```plaintext
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.**
  - Cost: \( M + p_R \times M \times N = 1000 + 100 \times 1000 \times 500 \) I/Os.

- **Page-oriented Nested Loops join:** 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.
  - Cost: \( M + M \times N = 1000 + 1000 \times 500 \)
  - If smaller relation (S) is outer, cost = 500 + 500 \times 1000
Block Nested Loops Join

- 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 + \#outer blocks * scan of inner
  - \#outer blocks = ⌊ # of pages of outer / blocksize ⌋

- **With Reserves (R) as outer, and 100 pages of R:**
  - 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.
  - If space for just 90 pages of R, we would scan S 12 times.

- **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.

- **With sequential reads considered, analysis changes:**
  may be best to divide buffers evenly between R and S.
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: \( M + ( (M*p_R) \times \text{cost of finding matching } S \text{ tuples}) \)
- 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 (typical), unclustered: upto 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: 220,000 I/Os.

- **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.
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.
  - Advance scan of R until current R-tuple ≥ current S tuple, then advance scan of S until current S-tuple ≥ current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (current R group) and all S tuples with same value in Sj (current S group) match; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.

- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)
Example of Sort-Merge Join

- **Cost:** $M \log M + N \log N + (M+N)$
  - The cost of scanning, $M+N$, could be $M*N$ (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)
Cost for Sorting for Sort Merge Join

**Optimization 1: Use More Buffer B**

Cost for Sorting R when Buffer B is used

\[ 2 \times (M + N) \times \text{# of Passes to sort} \]

- # of Passes to sort Bigger Table (M pages)
  \[ = 1 + \text{Ceiling} \left( \log_{ \frac{B}{M}} \left( \frac{M}{B} \right) \right) \]

Total:
\[ = 2 \times (M + N) \times \left( 1 + \text{Ceiling} \left( \log_{ \frac{B}{M}} \left( \frac{M}{B} \right) \right) \right) \]
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.
  - With $B > \sqrt{L}$, 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 $< B/2$.
  - 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, cost of sort-merge join, like the cost of external sorting, is **linear.**
Optimized Sort Merge Join Cost

- **Optimization 2:**
  If we use Buffer size $B^2 > \text{Size of bigger Table, } B > \sqrt{M}$
  Sorting both Table $M+N$ can be done in 2 passes
  1) $2 \times (M + N) \times 2$ \(\rightarrow\) to sort both tables
  2) $(M + N)$ \(\rightarrow\) to merge two sorted table
  total = $5 \times (M + N)$

- **Optimization 3:**
  If we combine the final **Merging phase of Sort** in the second pass with the **Merging phase of Join**,  
  **No need to write back** whole sorted two tables 
  \(\rightarrow\) **Save** $(M+N)$ and 
  **No need to read** two sorted tables to merge to find matching 
  \(\rightarrow\) **Save** $(M+N)$
  total = $3 \times (M + N)$
Best Case of Cost of Sort Merge Join

Best Case: When Both Tables are Sorted
- Very efficient if tables are already sorted:
  No sorting cost. Cost is Only for merging phase
  total = $M + N$
Hash-Join

- Partition both relations using hash function \( h \): R tuples in partition i will only match S tuples in partition i.

- Read in a partition of R, hash it using \( h2 \) (\( <> h \)). Scan matching partition of S, search for matches.
Observations on Hash-Join

- \#partitions \( k < B-1 \) (why?), and \( B-2 > \) size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing \( k \), we get:
  - \( k = B-1 \) for Partitioning Phase, so each \( k = \frac{M}{(B-1)} \) and
  - \( \frac{M}{(B-1)} < B-2 \) for Probing, i.e., \( B \) must be > \( \sqrt{M} \)

- If we build an in-memory hash table to speed up the matching of tuples, 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
  Total = $3(M + N)$
- In our running example, this is a total of 4500 I/Os.
- In Recursive Calls

Cost for Recursive Partitioning when Buffer $B$ is used

$$2 \times (M + N) \times (# \ of \ Recursive \ Calls \ for \ Partitioning)$$

- # of Recursive Calls to each big Partition ($B-1$ partitions) when Bigger Table is $M$ pages
  # of Recursive Calls = $1 + \text{Ceiling} \left( \log_{B-1} \frac{M}{(B-1)} \right)$

So Total Cost =

$$2 \times (M + N) \times (1 + \text{Ceiling} \left( \log_{B-1} \frac{M}{(B-1)} \right)) + (M+N)$$
Sort-Merge Join vs. Hash Join

- **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.
  - Hash Join shown to be highly parallelizable.
  - Hash Join is Bad when Data Skewed a lot (some big partitions causes *recursive partitioning*).
  - Sort-Merge Less Sensitive to Data Skew than Hash Join.
  - Result of Sort-Merge is sorted!
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.
Conclusions

• Database needs to run queries fast

• Sorting efficiently is one factor

• Choosing the right join another factor

• Next time: optimizing all parts of a query