Today

- Query execution architectures
- Data manipulation toolkit
  - “Heaps” and iteration
  - Sorting
  - Hashing
- Tables, clustering, and indices

Reminder: project proposals due Wednesday!
The Execution Engine

- The “cardiopulmonary system” of answering queries
  - Causes data to flow from sources to output
  - The optimizer is the “brain”!

- Input: physical query plan, set of sources
  - (How is physical plan different from logical?)
  - Plan is typically a tree (but can be a graph)

- Engine and phys. plan schedule execution of operators
  - What goes in sequence, what goes in parallel?
  - What operations can be distributed or parallelized?
  - “Push” vs. “pull”
Some Scheduling Possibilities

- **Series vs. parallel:**
  - Operators execute sequentially (blocking)
  - Operators are pipelined (note effect on state)

- **Scheduling:**
  - Operators are input-driven (push)
  - Operators get separate threads
  - Operators are demand-driven (iterator)
The Iterator Model in Action

- Methods: open, next, close()

```
? Merge
R1.y = R2.y

S-scan
R1(y,z)
clustered on y

I-scan
y < 5
R2(x,y)
```
Processing Data Naively

- Relation may be arbitrarily laid out on disk
  - Often called a “heap file” – but not as in the heap data structure
- How can we process it?
  - Iterate through every tuple and test
  - Iterate through every page of tuples and test (when is this different?)
- Or be smarter in laying out the data…
Sorting

- How do we do sort an R-page table, given $M$ pages’ worth of memory?
  - We all learned quicksort for in-memory sorts
    - Pick a split point, partition data above and below it
  - Can also do replacement selection
    - Heap-based – sort of like an incremental heapsort
    - Average run file is about $2M$; \( \left\lceil \frac{R}{2M} \right\rceil + 1 \) expected runs
  - But that’s only for one run… How do we combine runs efficiently?
General External Merge Sort

- To sort a file with \( R \) pages using \( M \) buffer pages:
  - Pass 0: use \( M \) buffer pages. Produce \( \left\lceil \frac{R}{M} \right\rceil \) sorted runs of \( M \) pages each.
  - Pass 2, …, etc.: merge \( M-1 \) runs.
Cost of External Merge Sort

- Number of passes: \(1 + \left\lceil\log_{M-1}\left\lfloor\frac{R}{M}\right\rfloor\right\rceil\)
- Cost = \(2R \times \# \text{ of passes}\)

With 5 buffer pages, to sort 108 page file:
  - Pass 0: \(\left\lfloor\frac{108}{5}\right\rfloor = 22\) sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: \(\left\lfloor\frac{22}{4}\right\rfloor = 6\) sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages
Hashing

- Sorting divides data using physical properties, then combines using logical properties.
- Hashing divides data using logical properties (hash key) and then chains them in buckets.
- How do we hash an R-page table, given \( M \) pages’ worth of memory?
  - Avoidance: pre-partition the data into \( R / M \) smaller units, then execute each.
  - Resolution: partition after we run against bounds.
Hybrid Hashing

- R-page table, given $M$ pages’ worth of memory in $F$ hash buckets:
  - Assign $K$ partitions, each expected to be of size $M$
  - Leaves $M - (K+1)C$ buffers for hashing
- May need to hash recursively, and skew may affect this
  - Sometimes revert to other algorithms when skew is a problem
Reading from Disk

- Simple sequential scan
- Associative access: indices
  - An index contains a collection of data entries, and supports efficient retrieval of all data entries $k^*$ with a given key value $k$.
  - Hash index – what do you think this looks like?
  - B+ tree
  - Bitmap index
Alternatives for Data Entry $k^*$ in Index

- Three alternatives:
  - Data record with key value $k$
    - ✓ Clustered -> fast lookup
    - 🔴 Index is large; only 1 can exist
  - $<k, \text{rid of data record with search key value } k>$, OR
  - $<k, \text{list of rids of data records with search key } k>$
    - ✓ Can have secondary indices
    - ✓ Smaller index may mean faster lookup
    - ▼ Often not clustered -> more expensive to use

- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value $k$. 
Classes of Indices

- **Primary vs. secondary**: primary has primary key
- **Clustered vs. unclustered**: order of records and index approximately same
  - Alternative 1 implies clustered, but not vice-versa.
  - A file can be clustered on at most one search key.
- **Dense vs. Sparse**: dense has index entry per data value; sparse may “skip” some
  - Alternative 1 always leads to dense index.
  - Every sparse index is clustered!
  - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.
Clustered vs. Unclustered Index

Suppose Index Alternative (2) used, records are stored in Heap file

- Perhaps initially sort data file, leave some gaps
- Inserts may require overflow pages

CLUSTERED

UNCLUSTERED
B+ Tree: Our Favorite Index

- Insert/delete at \( \log_F N \) cost
  - \( F = \) fanout, \( N = \# \) leaf pages
  - Keep tree *height-balanced*
- Minimum 50% occupancy (except for root).
- Each node contains \( d \leq m \leq 2d \) entries. \( d \) is called the *order* of the tree.
- Supports *equality* and *range* searches efficiently.

---

Index Entries
(Direct search)

Data Entries
("Sequence set")
Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf.
- Search for 5*, 15*, all data entries >= 24* ...

Based on the search for 15*, we know it is not in the tree!
B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133

- Typical capacities:
  - Height 4: $133^4 = 312,900,700$ records
  - Height 3: $133^3 = 2,352,637$ records

- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes
Bitmap Indices

- Primarily useful for discrete values, indices on multiple attributes
  - A bit for each possible value of an attribute
  - Example:
    \[ \text{sex} \in \{M, F\}; \quad \text{status} \in \{\text{ugrad, grad, fac}\} \]
    PennCIS(ID, name, sex, status)

<table>
<thead>
<tr>
<th>sex_bmp</th>
<th>status_bmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>MF</td>
<td>UGF</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>01</td>
<td>010</td>
</tr>
<tr>
<td>10</td>
<td>001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>sex</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Peng</td>
<td>M</td>
<td>G</td>
</tr>
<tr>
<td>50</td>
<td>Kit</td>
<td>F</td>
<td>G</td>
</tr>
<tr>
<td>99</td>
<td>Zack</td>
<td>M</td>
<td>F</td>
</tr>
</tbody>
</table>
Wrapping Up

- Today we saw a “toolkit” of techniques for query execution
  - Sorting and hashing to speed up processing of data
  - Need strategies for larger-than-memory operation
- Associative access methods for retrieving data
  - Hash indices
  - B+ trees
  - Bitmap indices
- Wednesday: putting these to use in real algorithms!