Implementation of Relational Operations

We will consider how to implement:
- **Selection** ($\sigma$) Selects a subset of rows from relation.
- **Projection** ($\pi$) Deletes unwanted columns from relation.
- **Join** ($\Join$) Allows us to combine two relations.

Since each op returns a relation, ops can be composed! After we cover the operations, we will discuss how to optimize queries formed by composing them.

Schema for Examples

- **Sailors** ($\text{id}$: integer, $\text{name}$: string, $\text{rating}$: integer, $\text{age}$: real)
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- **Reserves** ($\text{id}$: integer, $\text{bid}$: integer, $\text{day}$: dates, $\text{rname}$: string)
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

Simple Selections

- ```
  SELECT *
  FROM Reserves R
  WHERE R.rname < 'C%'
  ```

Using an Index for Selections

- Cost depends on # qualifying tuples and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering).
  - In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, up to 10000 I/Os!
- **Important refinement for unclustered indexes:**
  1. Find qualifying data entries.
  2. Sort the $\text{rid}$s of the data records to be retrieved.
  3. Fetch $\text{rid}$s in order. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).

Projection via Hashing

- ```
  SELECT DISTINCT R.sid, R.bid
  FROM Reserves R
  ```

  **Partitioning phase:** Read R using one input buffer. For each tuple, discard unwanted fields, apply hash function $h1$ to choose one of B-1 output buffers.
  - Result is B-1 partitions (of tuples with no unwanted fields).
  - 2 tuples from different partitions guaranteed to be distinct.
- **Duplicate elimination phase:** For each partition, read it and build an in-memory hash table, using hash fn $h2$ ($\Rightarrow h1$) on all fields, while discarding duplicates.
  - If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition.
- Cost: For partitioning, read R, write out each tuple, but with fewer fields. This is read in next phase.
Discussion of Projection

- Sort-based approach features better handling of skew and result is sorted.
- Hash-based approach can be faster (locally).
- If an index on the relation contains all wanted attributes in its search key, can do index-only scan.
  - Apply projection techniques to data entries (much smaller!)
- If an ordered (i.e., tree) index contains all wanted attributes as 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.

Equality Joins With One Join Column

```sql
SELECT * 
FROM Reserves R1, Sailors S1 
WHERE R1.sid = S1.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.
- Cost metric: # of I/Os. We will ignore output costs.

Simple Nested Loops Join

```plaintext
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 \( S \).
  - Cost: \( M + p_R \times M \times N = 1000 + 100 \times 1000 \) 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: \( M + M \times N = 1000 + 100 \times 500 \)
  - If smaller relation (\( S \)) is outer, cost = \( 500 + 500 \times 1000 \)

Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = \( \lceil \frac{\# of pages of outer}{blocksize} \rceil \)
- 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 (\( S \)): 100 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: 50 I/Os.
- With sequential reads considered, analysis changes: may be best to divide buffers evenly between \( R \) and \( S \).

Block Nested Loops Join

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

- 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.

Index Nested Loops Join

```plaintext
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: \( M + (M \times p_S) \times \text{cost of finding matching } S \text{ tuples} \)
- 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: 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: 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.

Example of Sort-Merge Join

```
<table>
<thead>
<tr>
<th>sid</th>
<th>name</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>
```

- sid | bid | day       | name  
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></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>
</tbody>
</table>

- 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)

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.
- 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 corresponding R-partition and add tuples to table while discarding duplicates.

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?

Summary

- A virtue of relational DBMSs: queries are composed of a few basic operators; the implementation of these operators can be carefully tuned (and it is important to do this!).
- Many alternative implementation techniques for each operator; no universally superior technique for most operators.
- Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc. This is part of the broader task of optimizing a query composed of several ops.
State of the Art (impl. algorithms)

- Approximate answers (data warehousing)
  - Too much data to find exact answer quickly
  - No need for an exact answer
- Top-K queries (K “best” matches)
  - Multimedia (fuzzy criteria), decision support
  - Approximate or exact
- Extensibility:
  - User-defined data types
  - User-defined functionality
- Improve time-to-first-result-tuple
  - Ripple Join (impl. project, part 2)