Evaluation of Relational Operations

Yanlei Diao
Overview of Query Processing
Relational Operations

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

- **Union** ($\cup$) Tuples in either reln. 1 or reln. 2.
- **Intersection** ($\cap$) Tuples in both reln. 1 and reln. 2.
- **Set-difference** ($-$) Tuples in reln. 1, but not in reln. 2.

- **GROUP BY** and **Aggregation** (SUM, MIN, etc.)

with cost estimation, which leads to **cost-based optimization**
For more details, see the textbook

Database Management Systems
3rd Edition
Ramakrishnan and Gehrke

Amazon:
- Buy new: $43-$147.09 (hardcover); paperback, $23; Kindle, rent options are also available…
Background on Algorithms

Introduction to Algorithms
3rd Edition
Thomas Cormen,
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein

Amazon:
Buy new: $66.32(hardcover);
rent options are also available…
Outline

- Selection
- Sorting routine
- Join
- Projection
- Set operators
- Group By aggregation
Schema for Examples

Sailors (\textit{sid}: integer, \textit{sname}: string, \textit{rating}: integer, \textit{age}: real)
Reserves (\textit{sid}: integer, \textit{bid}: integer, \textit{day}: date, \textit{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
Using an Index for Selections

Cost of selection includes:
1) top down search in the index
2) scan the relevant leaf nodes
3) retrieve records from file (could be large w/o clustering)

```
SELECT * 
FROM Sailors S 
WHERE S.rating > 8
```
Using an Index for Selections

- Cost of selection includes:
  1) top down search in the index
  2) scan the relevant leaf nodes
  3) retrieve records from file (could be large w/o clustering)

- Step 1) top down search: = the height of the tree \( H \) (and buffer management), \( \leq 3-4 \) I/Os

```sql
SELECT * 
FROM Sailors S 
WHERE S.rating > 8
```
Cost Factors of Steps 2 and 3

- Cost of selection includes:
  1) top down search in the index
  2) scan the relevant leaf nodes
  3) retrieve records from file (could be large w/o clustering)

Step 2 scanning leaf nodes: cost factors include

- **Num. of leaf nodes**: Either is given, or can be estimated as follows: if a data entry is 1/5 of a tuple, there are 500 / 5 = 100 leaf nodes.
- **Reduction factor (% of qualifying tuples)**: rating > 8: 20% of tuples qualify
- So, 100 * 20% = 20 I/Os.

```
SELECT * 
FROM   Sailors S
WHERE  S.rating > 8
```
Cost Factors of Selection (contd.)

- **Step 3 retrieving records from file**: cost factors include
  - *Num. of qualifying tuples*: rating > 8: 20% of tuples, 500*20%=100 data pages, 80*100=8,000 tuples
  - *Clustering*: retrieving records from file
    - **Clustered index**: 100 I/Os.
    - **Unclustered index**: worst case 1 I/O per tuple; 8,000 I/Os here!
    - **Unclustered index + Sorting or Bitmap on rid**: ≤ 500 I/Os.

(a) Retrieve matching data entries; sort by page_id; retrieve records in order of page id
(b) Bitmap Index Scan + Bitmap Heap Scan in PostgreSQL: replace sorting w. a bitmap on page ids; retrieve records from pages with the bit set to 1
General Selections

- Boolean combination of predicates using AND and OR.
  - Conjunctive Normal Form (CNF), e.g.,
    \[
    \text{pred1 AND (pred3 OR pred4)}
    \]
    \[
    (\text{pred1 OR pred2}) \text{ AND (pred3 OR pred4)}
    \]

- **File scan** always works for general selections.

- **Index scan** works when it matches a predicate that is a conjunct of CNF.
  - E.g., an index matching ‘pred1’ can be used for
    \[
    \text{pred1 AND (pred3 OR pred4)}
    \]
Outline

- Selection
  - Sorting routine
- Join
- Projection
- Set operators
- Group By aggregation
Why Sort?

- Important utility in DBMS:
  - Request data in *sorted order* (e.g., ORDER BY)
    - e.g., find students in decreasing order of *gpa*
  - *Sort-merge* join algorithm involves sorting.
  - *Eliminate duplicates* in a collection of records (e.g., SELECT DISTINCT)
  - Sorting is first step in *bulk loading* B+ tree index.

- Problem: sort 1TB of data with 1GB of RAM
  - Limited memory → key is to minimize # I/Os!
  - Methodology for algorithm design: from *simple* to *complex*
What is the **minimum memory size** (in num. of buffer pages) needed to sort a large file on disk?
2-Way Sort: Requires 3 Buffer Pages

- Pass 1: Read a page, sort it, write it as a sorted subfile
  - only one buffer page (from the memory) is used

- Pass 2, 3, …, etc.: Merge two sorted subfiles
  - three buffer pages are used
Two-Way External Merge Sort

- **Divide and conquer**: sort subfiles (runs) and then merge

A file of N pages:
- Pass 1: N sorted runs of 1 page each
- Pass 2: N/2 sorted runs of 2 pages each
- Pass 3: N/4 sorted runs of 4 pages each
- ... (omitted for brevity)
- Pass P+1: 1 sorted run of \(2^P\) pages

\[2^P \geq N \Rightarrow P \geq \log_2 N\]
Two-Way External Merge Sort

- **Divide and conquer:** sort subfiles (runs) and then merge
  - Each pass, read + write $N$ pages in file $\rightarrow 2N$.
  - Number of passes is: $\lceil \log_2 N \rceil + 1$
  - So total cost is: $2N(\lceil \log_2 N \rceil + 1)$
**General External Merge Sort**

*Given B (>3) buffer pages. How can we utilize them?*

- **Pass 1:** Use $B$ buffer pages. Produce $\lceil N/B \rceil$ sorted runs of $B$ pages each.
- **Pass 2, 3..., etc.:** Merge $B-1$ runs.

![Diagram of General External Merge Sort](image-url)
**Cost of External Merge Sort**

- E.g., with 5 (B) buffer pages, sort 108 (N) page file:

<table>
<thead>
<tr>
<th>Pass 1</th>
<th>( \lceil \frac{108}{5} \rceil = 22 ) sorted runs of 5 pages each (last run is only 3 pages)</th>
<th>( \lceil \frac{N}{B} \rceil ) sorted runs of B pages each</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass 2</td>
<td>( \lceil \frac{22}{4} \rceil = 6 ) sorted runs of 20 pages each (last run is only 8 pages)</td>
<td>( \lceil \frac{N}{B} \rceil / (B-1) ) sorted runs of ( B(B-1) ) pages each</td>
</tr>
<tr>
<td>Pass 3</td>
<td>2 sorted runs, 80 pages and 28 pages</td>
<td>( \lceil \frac{N}{B} \rceil / (B-1)^2 ) sorted runs of ( B(B-1)^2 ) pages</td>
</tr>
<tr>
<td>Pass 4</td>
<td>Sorted file of 108 pages</td>
<td>( \lceil \frac{N}{B} \rceil / (B-1)^3 ) sorted runs of ( B(B-1)^3 ) (( \geq N )) pages</td>
</tr>
</tbody>
</table>

- Number of passes = \( 1 + \lceil \log_{B-1} \lceil \frac{N}{B} \rceil \rceil \)

Cost = \( 2N \times (1 + \lceil \log_{B-1} \lceil \frac{N}{B} \rceil \rceil ) \)
## Number of Passes of External Sort

<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><strong>4</strong></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><strong>4</strong></td>
</tr>
</tbody>
</table>
Replacement Sort

- **Replacement Sort**: Produce initial sorted runs as long as possible. Used in Pass 1 of sorting.

- Organize B available buffers:
  - 1 buffer for *input*
  - B-2 buffers for *current set*
  - 1 buffer for *output*
Replacement Sort

- Pick tuple \( r \) in the current set (CS) such that \( r \) is the smallest value in CS that is \( \geq \) largest value in output, e.g. 8, to extend the current run.

- Write output buffer out if full, extending the current run.

- Fill the space in current set by adding tuples from input.

- Current run terminates if every tuple in the current set is \( < \) the largest tuple in output.
Replacement Sort

- **Good empirical results:** When used in Pass 1 for sorting, write out **sorted runs of size** \(2B\) on average.
- Affects calculation of the number of passes accordingly.
  - We will use this result in a later algorithm.
Outline

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

SELECT *  
FROM Reserves R, Sailors S  
WHERE R.sid = S.sid

- R \bowtie S, natural join. Very common operation!
- Semantics: cross product (\times) followed by selection (\sigma)
  - If R \times S is large, R \times S followed by a selection is inefficient.
  - Must be carefully optimized.
- **Cost metric**: # of I/Os. Ignore output cost in analysis.
  - R: M pages, p_R tuples per page
  - S: N pages, p_S tuples per page.
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:

```plaintext
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.)**

```plaintext
defined block in R do
    build a hash table on R-block (optional)
defined 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 + ⌈M / B - 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 + ⌈500/100⌉ * 1000 = 5,500 I/Os.
  - What is the cost if we swap R and S?
    - Cost = 1000 + ⌈1000/100⌉ * 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:

```
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** \((R \Join S)\) 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

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
<th>rname</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>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td>dustin</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>rname</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>

- **Cost**: Sorting\_cost(R) + Sorting\_cost(S) + Merging\_cost
  - Merging\_cost ∈ [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

B memory buffer pages

Run1 of R

Run2 of R

RunK of R

Run1 of S

Run2 of S

RunK of S

OUTPUT

Join Results
Merging in Two-Pass Sort-Merge

Relation R

Relation S

Join Results

OUTPUT

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)