Evaluation of Relational Operators: Chap. 14

Relational Algebra

- Relational operators:
  - Selection \( \sigma \)
  - Projection \( \pi \)
  - Join \( \bowtie \) Combines several relations using conditions
  - Set-difference \( \setminus \) Union \( \cup \) Intersection \( \cap \)
  - Aggregation and Grouping

Example Schema

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

- Similar to old schema; rname added
- Reserves:
  - 40 bytes long tuple, 100K records, 100 tuples per page, 1000 pages
- Sailors:
  - 50 bytes long tuple, 40K tuples, 80 tuples per page, 500 pages

Selections with Simple Condition

\[ \sigma_{\text{attr} \text{OP} \text{val}}(R) \]

- Case 1: No index, Unsorted data
  - Do scan
- Case 2: No Index, Sorted Data
  - Perform binary search on file (exact match or ranges)
  - \( O(\log M), M = \text{number of pages in file} \)
- Case 3: Index Available
  - Is the index B+ Tree or Hash?
  - Is it clustered or not?

Using an Index for Selections

- Cost depends on
  - Number of qualifying tuples
  - Clustering
- Cost has two components:
  - Finding qualifying data entries (typically small)
  - Retrieving records (could be large w/o clustering)
- Consider Reserves, assume 10% of tuples satisfy condition
  - Result has 10K tuples, 100 pages
  - With clustered index, cost is little more than 100 I/Os
  - If unclustered, up to 10000 I/Os!
For Unclustered Indexes

- **Important refinement:**
  1. Find qualifying data entries
  2. Sort the rid's of the data records to be retrieved
  3. Fetch rid's *in order*

- Ensures that each data page is looked at just once
  - although number of I/Os still higher than with clustering

Example from Oracle: unclustered index on K500K

```sql
SQL> select rowid from bench where k500k>=400 and k500k<403;
ROWID
--------------------
AACh1kAAJAAADVGAAU
AACh1kAAJAAAFEBABY
AACh1kAAJAAJGPOAC
AACh1kAAJAAENYAAN
AACh1kAAJAAJNXAB
AACh1kAAJAAJFBAM
AACh1kAAJAAABS

k500k=400: 2 data entries
k500k=401: 2 data entries
k500k=402: 4 data entries
```

- RIDs for a certain key are in sorted order in index.
- With 3 keys, the whole set of RIDs is not in RID order.
- This is an index-only query, no need to access heap table.

Example from Oracle: unclustered index on K500K

```sql
SQL> select kseq from bench where k500k>=400 and k500k<403;
KSEQ
-------
432909
551651
661223
801212
817431
846181
894121
985835
8 rows selected.
```

- Note that the RIDs were sorted before the KSEQ values were obtained from the heap table.

General Conditions Selections

- **Condition may be composite**
  - In *conjunctive* form: easier to deal with
  - At least one *disjunction*: less favorable case

- **Disjunctive form**
  - Only one of the conditions, if met, qualifies tuple
  - Even if some disjunct is optimized, the other(s) may require scan
  - In general, this case dealt with using set union
  - Most DBMS optimizers focus on conjunctive forms

Evaluating Conjunctive Forms (1/2)

- Find the *most selective access path*, retrieve tuples using it, and apply any remaining terms that don’t *match* the index
  - **Most selective access path:** An index or file scan that we estimate will require the fewest page I/Os
  - **Example:** `day<8/9/94 AND bid=5 AND sid=3`
  - B+ tree index on `day` can be used; then, `bid=5` and `sid=3` must be checked for each retrieved tuple
  - Similarly, a hash index on `<bid, sid>` could be used; `day<8/9/94` must then be checked.

Evaluating Conjunctive Forms (2/2)

- **Intersect rid's**
  - If we have two or more matching indexes that use Alternatives (2) or (3) for data entries:
    - Get sets of rid's of data records using each matching index
    - Then *intersect* these *sets of rid's* (we’ll discuss intersection soon!)
    - Retrieve the records and apply any remaining terms
    - **Example:** `day<8/9/94 AND bid=5 AND sid=3`
    - B+ tree index on `day` and an index on `sid`, both using Alternative (2)
    - Retrieve rid's satisfying `day<8/9/94` using the B+ tree, rid's satisfying `sid=3` using the hash, intersect, retrieve records and check `bid=5`
Intersecting RIDs via Index JOIN

- Example: \( \text{day} < 8/9/94 \) AND \( \text{bid} = 5 \) AND \( \text{sid} = 3 \)
- \( \text{B}+ \) tree index on \( \text{day} \) and an index on \( \text{sid} \), both using Alternative (2)
- Retrieve rids satisfying \( \text{day} < 8/9/94 \) using the \( \text{B}+ \) tree, rids satisfying \( \text{sid} = 3 \) using the hash, intersect, retrieve records and check \( \text{bid} = 5 \)
- Another way to achieve this: Join the two indexes
  - As tables, indexes are \( I_1 = (\text{rid}, \text{day}) \) and \( I_2 = (\text{rid}, \text{sid}) \)
  - Join them: \( I_1 \text{ where day} < 8/9/94 \) JOIN \( I_2 \text{ where sid} = 3 \)
  - Obtain \((\text{rid, day, sid})\) satisfying the two conditions and providing rids

Pg. 446: Oracle does this.

Projection

- Remove unwanted attributes
- Eliminate any duplicate tuples produced (the hard part)

Projection with Sorting

- Modify Pass 0 of external sort to eliminate unwanted fields
  - Produce runs of about 2B pages are produced
  - Tuples in runs are smaller than input tuples
  - Size ratio depends on number and size of fields that are dropped
- Modify merging passes to eliminate duplicates
  - Thus, number of result tuples smaller than input
  - Difference depends on number of duplicates

Cost

- In Pass 0, read original relation (size \( M \)), write out same number of smaller tuples
- In merging passes, fewer tuples written out in each pass. Using Reserves example, 1000 input pages reduced to 250 in Pass 0 if size ratio is 0.25

Projection with Hashing

- \( \text{Partitioning phase:} \)
  - Read \( R \) using one input buffer. For each tuple, discard unwanted fields, apply hash function \( h \) to choose one of \( B-1 \) output buffers
  - Each output buffer is feeding a run on disk
  - Result is \( B-1 \) partitions (of tuples with no unwanted fields), tuples from different parstions guaranteed to be distinct
  - See next slide for diagram
- \( \text{Duplicate elimination phase:} \) process runs from partitioning phase. Each run forms a partition of the data

Hash Projection: Partitioning Phase

- Partition \( R \) using hash function \( h \)
- Duplicates will hash to the same partition
Projection with Hashing

- **Partitioning phase**: ends up with partitions of data, each held in a run on disk
- **Duplicate elimination phase**:
  - For each partition, read it and build an in-memory hash table, using hash $h_2$ on all fields, while discarding duplicates
  - If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition
- **Cost**:
  - Read $R$, write out each tuple, but fewer fields, size $T \leq M$. Result read in next phase. Total I/O cost: $M + 2T \leq 3M$, similar to sort if it can be done in 2 passes.

Discussion of Projection

- Sort-based approach is the standard
  - better handling of skew and result is sorted.
  - Hashing is more parallelizable
- If index on relation contains all wanted attributes in its search key, 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
```

- Most frequently occurring in practice
- We will consider more complex join conditions later
- **Cost metric**: number of I/Os
  - Ignore output costs

Simple Nested Loops Join

```java
foreach tuple r in R do
  foreach tuple s in S do
    if r_i == s_i then add <r, s> to result
```

- For each tuple in the **outer** relation $R$, we scan the entire **inner** relation $S$.
  - Cost: $M + p_h \times M \times N = 1000 \times 100\times1000\times500$ I/Os

Page-oriented Nested Loops join:

- For each page of $R$, get each page of $S$, and write out matching pairs
  - Cost: $M + M^2 N = 1000 \times 100\times1000$ I/Os
  - If smaller relation ($S$) is outer, cost = $500 + 500\times1000$

Block Nested Loops Join

```java
R & S
```

- one page input buffer for scanning the inner $S$
- one page as the output buffer
- 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.

Hash Projection: Second Phase

Read in a partition of $R$, hash it using $h_2$ (<> $h$)
Discard duplicates as go along.
When partition is all read in, scan the hash table and write it out as part of the projection result

![Diagram of hash projection](image-url)
Examples of Block Nested Loops

Cost: Scan of outer + \( \text{#outer blocks} \times \text{scan of inner} \)

\( \text{#outer blocks} = \left\lceil \frac{\text{#pages of outer}}{\text{blocksize}} \right\rceil \)

With Reserves (R) as outer, and 100 pages per 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.
- Total 1000 + 10*500 = 6000 I/Os.
- Need 101 buffer pages for this.

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.
- Total 500 + 5*1000 = 5500 I/Os. Same ballpark as above.

Compare these to page-oriented NLJ: 500,000 I/O or worse!

Executing Joins: Index Nested Loops

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

Example of Index Nested Loops (1/2)

Case 1: Hash-index (Alternative 2) on sid of Sailors

- Choose Sailors as inner relation
- Scan Reserves: 100K tuples, 1000 page I/Os
- For each Reserves tuple:
  - 1.2 I/Os to get data entry in index
  - 1 I/O to get (the exactly one) matching Sailors tuple (primary key)
- Total: 221,000 I/Os

Example of Index Nested Loops (2/2)

Case 2: Hash-index (Alternative 2) on sid of Reserves

- Choose Reserves as inner
- Scan Sailors: 40K tuples, 500 page I/Os
- For each Sailors tuple:
  - 1.2 I/Os to find index page with data entries
  - Assuming uniform distribution, 2.5 matching records per sailor
  - Cost of retrieving records is nothing (Alt 1 clustered), single I/O (Alt 2 clustered index) or 2.5 I/Os (unclustered index)
- Total: 48,500 I/Os (clustered Alt 1), 88,500 I/Os (clustered Alt 2) or 148,500 I/Os (unclustered)

Sort-Merge Join

- Sort R and S on the join column (book assumes file-to-file sort, no pipelining)
- Then scan them to do a merge on join column:
  - Advance scan of R until current R-tuple >= current S tuple
  - Then, advance scan of S until current S-tuple >= current R tuple
  - Repeat until current R tuple = current S tuple
  - At this point, all R tuples with same value in \( R \) (current \( R \) group) and all S tuples with same value in \( S \) (current \( S \) group) match
  - Output \( <r, s> \) for all pairs of such tuples
  - May have to rescan part of one of the input files if have pages of duplicate join keys vs. multiple matching join keys
  - Resume scanning R and S

Sort-Merge Join Cost

- R is scanned once
- Each S group is scanned once per matching R tuple
- Multiple input-file scans per group needed only if S records with same join attribute value span multiple pages
- Multiple such scans of an S group are likely to find needed pages in buffer
- Cost: (assume B buffers)
  - Sort(R) + Sort(S) + merge
  - \( 2M (1 + \log B (M/B)) \) + \( 2N (1 + \log B (N/B)) \) + \( M+N \)
  - The cost of scanning, \( M+N \), could be \( M^2 \) worst case (very unlikely!)
  - In many cases, the join attribute is primary key in one of the tables, which means no duplicates in one merge stream.
2-Pass Sort-Merge Join

- With enough buffers, sort can be done in 2 passes
  - First pass generates \( N/B \) sorted runs of \( B \) pages each
  - If one page from each run + output buffer fits in memory, then merge can be done in one pass; denote larger relation by \( L \)
  - \( 2L/B + 1 \leq B \), holds if \( B > \sqrt{L} \)
- One optimization of sort allows runs of \( 2B \) on average
  - First pass generates \( N/2B \) sorted runs of \( 2B \) pages each
  - Condition above for 2-pass sort becomes \( B > \sqrt{L/2} \)
    - (But we're not officially covering this optimization)
- Merge can be combined with filtering of matching tuples
  - The cost of sort-merge join becomes \( 3(M+N) \)

Hash-Join: Partitioning Phase

- Partition both relations using hash function \( h \)
- \( R \) tuples in partition \( i \) will only match \( S \) tuples in partition \( i \)
- This is similar to the partitioning phase of Projection by Hashing

![Hash-Join: Partitioning Phase Diagram](image)

Hash-Join Properties

- \#partitions \( k \leq B-1 \) because one buffer is needed for scanning input
  - \( k = B-1 \), and \( M/(B-1) \leq B-2 \), i.e., \( B > \sqrt{M} \)
  - \( M \) is smaller of the two relations!
  - So best to use the smaller table’s partitions for the second-phase hash tables.
    - i.e., we can take advantage of one table being small, unlike sort-merge.
  - If the hash function does not partition uniformly, one or more second-phase partitions may not fit in memory
    - Can apply hash-join technique recursively to do the join of this \( R \)-partition with corresponding \( S \)-partition.

![Hash-Join Properties Diagram](image)

Cost of Hash-Join

- In partitioning phase, read+write both \( R \) and \( S \): \( 2(M+N) \)
- In matching phase, read both \( R \) and \( S \): \( M+N \)
  - (assumes hash tables fit in memory, \( B > \sqrt{M} \) )
- With sizes of 1000 and 500 pages, total is 4500 I/Os

Hash-Join vs Sort-Merge Join

- Given sufficient amount of memory both have a cost of \( 3(M+N) \) I/Os (with no pipelining in or out, book’s assumption)
  - Hash Join superior on this count if relation sizes differ greatly
  - Hash Join shown to be highly parallelizable
  - Sort-Merge less sensitive to data skew, and result is sorted
General Join Conditions (1/2)

- Equalities over several attributes
  - e.g., $R.sid = S.sid \land R.rname = S.sname$
  - For Index Nested Loop, 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

General Join Conditions (2/2)

- Inequality conditions
  - e.g., $R.rname < S.sname$
  - For Index Nested Loop 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 Nested Loop quite likely to be the best join method here

Set Operations

- Intersection and cross-product special cases of join
- Union (Distinct) and Except similar
  - Both hashing and sorting are possible
    - Similar in concept with projection

Union with Sorting

- Sort both relations (on combination of all attributes)
- Scan sorted relations and merge them
  - Alternative: Merge runs from Pass 0 for both relations

Union with Hashing

- Partition $R$ and $S$ using hash function $h$
  - For each $S$-partition, build in-memory hash table (using $h^2$
    - scan corresponding $R$-partition and add tuples to table while discarding duplicates

Aggregate Operations (sum, avg, count, min, max)

- Without grouping:
  - In general, requires scanning the relation
  - Given index whose search key includes all attributes in the
    SELECT or WHERE clauses, can do index-only scan
    - Example: select avg(s.age) from sailors $s$
      - With index on age, just scan it for age values, take avg on the fly
      - Select max(s.age) from sailors $s$ where age < 50;
        - Still index-only
      - Select max(s.age) from sailors $s$ where rating = 5;
        - Uses table scan unless there is an index on rating
Aggregate Operations

- With grouping:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group
  - Possible to improve upon step above by combining sorting and aggregate computation
  - Similar approach based on hashing on group-by attributes
  - Given 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

Impact of Buffering

- Repeated access patterns interact with buffer replacement policy
  - 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

- Queries are composed of a few basic operators
  - The implementation of these operators can be carefully tuned

- 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