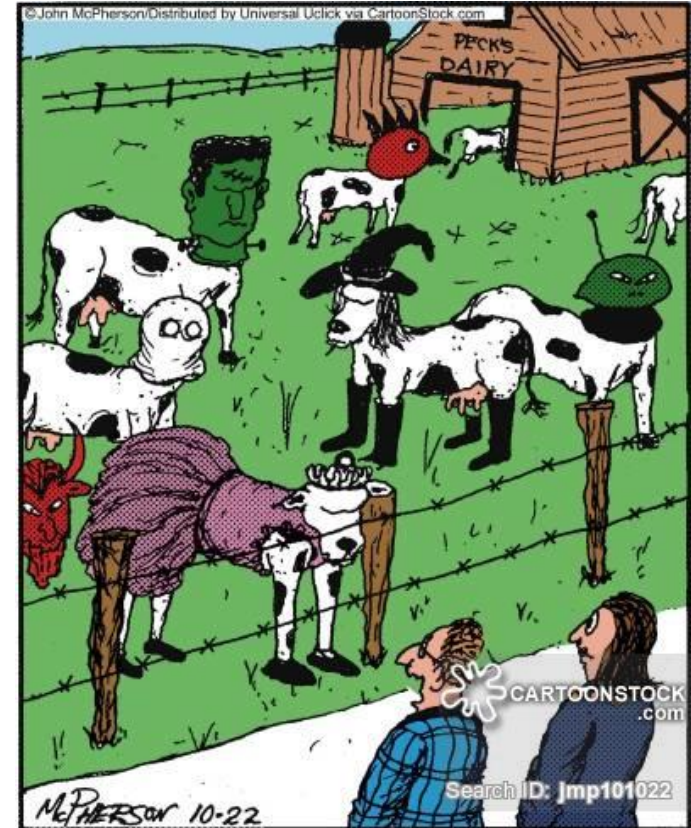




# *Evaluation of Relational Operations*

## Chapter 14



**“Ya gotta love the way this farmer gets into the Halloween spirit.”**



# Relational Operations

- ❖ We will consider in more detail how to implement:
  - Selection ( $\sigma$ ) Selects a subset of rows from relation.
  - Projection ( $\pi$ ) Deletes unwanted columns from relation.
  - Join ( $\bowtie$ ) Allows us to combine two relations.
  - Set-difference ( $-$ ) Tuples in left but not right relation.
  - Union ( $\cup$ ) Tuples in reln. 1 and in reln. 2.
  - Aggregation (SUM, MIN, etc.) and GROUP BY
- ❖ 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.



# Running Database Example

---

## ❖ Schema

Sailors (sid: integer, sname: string, rating: integer, age: real)

Reserves (sid: integer, bid: integer, day: dates, rname: string)

## ❖ ~100,000 Reserves:

- Each tuple is 40 bytes, 100 tuples per page, 1000 pages.

## ❖ ~40,000 Sailors:

- Each tuple is 50 bytes, 80 tuples per page, 500 pages.



# Selection (from Chapter 12)

(Note: we ignore “output costs”)

- ❖ No Index, Unsorted Data

- Scan the entire relation, for Reserves  $\rightarrow$  1000 I/Os

```
SELECT *  
FROM Reserves R  
WHERE R.rname like 'Joe%'
```

- ❖ No Index, Sorted by rname

- Binary search, for Reserves  $\rightarrow \log_2 1000 \sim 10$  I/Os

- ❖ B<sup>+</sup>-Tree Index, Clustered on selection attribute

- Use index to find smallest tuple satisfying selection, scan forward from there, for Reserves  $\rightarrow$  3 I/Os to find starting point + K Blocks containing 'Joe' (K  $\sim$  1-2 if op 'like' matches  $\sim$ 100 (1%))

- ❖ B<sup>+</sup>-Tree Index, Unclustered

- Discussion follows



# Using an Index for Selections

- ❖ Cost depends on #qualifying tuples, and clustering.
  - Cost of finding qualifying data entries is typically small, but the cost of retrieving records could be large w/o clustering.
  - Example, assuming uniform distribution of ratings (1-10), about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, could be upto 10000 I/Os!
- ❖ *Important refinement for unclustered indexes:*
  1. Find qualifying data entries in index.
  2. Find *distinct rids* of the pages to be retrieved. (2 ways)
    - A. Sort by *rid* while removing replicates
    - B. Hash *rids* while eliminating replicates
  3. Scan surviving *rids* while applying selection (result set will be unordered).
    - Ensures each page is considered just once (though # of pages is still likely higher than with clustering).



# General Selections

- ❖ Selections typically involve more than one attribute with logical conjuncts (and, or)
- ❖ Recall we transform to sum-of-product form
  - ANDs (products) are selective, they shrink the result set
  - ORs (sums) are applied during scan, they grow the result set
- ❖ Can be sorted or clustered by only one attribute
- ❖ Only a subset of attributes might have indices
- ❖ What order to process selection terms?
- ❖ How *selective* is a selection term?
  - rname like "Joe%" < 1% of Sailors
  - age < 20 ~ 10% of Sailors
  - Rating > 7 ~ 30 % Sailors



# Two Approaches to General Selections

- ❖ First approach: Find the *most selective access path*, retrieve tuples using it, and then apply remaining selection terms during scan:
  - *Most selective access path*: An index or file scan that we estimate will require the *fewest page I/Os*.
  - Terms that match this index reduce the number of tuples *retrieved*; other terms are used further discard retrieved tuples, but do not affect number of pages fetched.
  - Consider *day < 8/9/94 AND bid=5 AND sid=3*.
    - A B+ tree index on *day* can be used; then, *bid=5* and *sid=3* must be checked for each retrieved tuple.
    - A hash index on  $\langle bid, sid \rangle$  could be used; *day < 8/9/94* must then be checked during scan.





# Set Operation on Rids

- ❖ Second approach (query involves 2 or more indexes):
  - Get sets of *rids* of data records using each matching index.
  - *Intersect* and/or *union* these sets of *rids* (we'll see how shortly)
  - Retrieve the records and apply any remaining terms.
  - Consider *day < 8/9/94 AND bid = 5 AND sid = 3*.
    - If we have a B<sup>+</sup> tree index on *day* and an index on *sid*, both unclustered, we can retrieve *distinct rids* satisfying *day < 8/9/94* using the first, *rids* of records satisfying *sid = 3* using the second, intersect the *rid* sets, then retrieve records and check *bid = 5*.





# The Projection Operation

## ❖ Modified external sorting:

- **Modify Pass 0 of external sort to eliminate unneeded fields.** Thus, extending the run-size produced.

Tuples merged in subsequent passes are smaller than tuples of the original relation. (i.e. Instead of 40 bytes/record, perhaps 10, so 400 fit in a page. Size ratio ( $10/40 = 0.25$ ) depends on # and size of fields that are dropped.)

- **Modify merging passes to eliminate duplicates.** Thus, number of result tuples smaller than input. (Difference depends on # of duplicates.)
- **Cost:** In Pass 0, reads all original pages, but writes out fewer pages (same number of smaller tuples). In merge passes, fewer tuples are written out due to the eliminated duplicates.

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



# Projection Based on Hashing

## ❖ Modified hashing:

- *Partitioning phase*: Read R using one input buffer. For each tuple, discard unwanted fields, apply hash function  $h_1$  to direct output to one of B-1 output buffers.
  - Result is B-1 partitions (of tuples with no unwanted fields). Tuples in different partitions are guaranteed to be distinct.
- *Duplicate elimination phase*: Foreach partition either:
  - Build another “in-memory” hash table, using hash function  $h_2 (\neq h_1)$ , that discards duplicates (handled on collisions).
  - Sort partitions while eliminating duplicates
- *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 is the standard; better handles skewed attribute distributions and result is sorted.
- ❖ If an index on the relation contains the wanted projection attributes as its search key, then we can use an *index-only* scan (no fetching of the data pages).
- ❖ If an ordered (i.e., tree) index contains all wanted attributes as a *prefix* of its search key's we can
  - Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates.



# Equijoins w/one common column

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

- ❖ In relational algebra:  $R \bowtie S$  is very common! Must be carefully optimized.  $R \times S$  is very large; so,  $R \times S$  followed by a selection is inefficient.
- ❖ Assume:  $M$  tuples in  $R$ ,  $p_R$  tuples/page,  $N$  tuples in  $S$ ,  $p_S$  tuples/page.
- ❖ We will consider more complex join conditions later.
- ❖ *Cost metric*: # of I/Os. We will ignore output costs.



# Simple Nested Loops (SNL) Join

```
foreach tuple r in R:  
  foreach tuple s in S:  
    if  $r_i == s_j$  :  
      add  $\langle r, s \rangle$  to result
```

- ❖ Naïve Approach: For each tuple in the *outer* relation R, we scan the entire *inner* relation S (i.e.  $R \times S$ ).
  - Cost:  $M + (p_R * M) * N = 1000 + 100 * 1000 * 500$  I/Os.
- ❖ *Page-at-a-time* Nested Loops join: For each *page* of R, get each *page* of S, and handle all matching pairs of tuples  $\langle r, s \rangle$ , where r is in R-page and S is in S-page.
  - Cost:  $M + M * N = 1000 + 1000 * 500$
  - If smaller relation (S) is outer, cost =  $500 + 500 * 1000$



# *Index Nested Loops (INL) Join*

```
foreach tuple r in R:  
  foreach tuple s in S where  $r_i == s_j$ :  
    add  $\langle r, s \rangle$  to result
```

- ❖ If there is an index on the join column of one relation (say S), make it the inner loop, and exploit the index.
  - Cost:  $M + (M * p_R) * \text{cost of finding matching S tuples}$
- ❖ 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 depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.



# *Examples of Index Nested Loops*

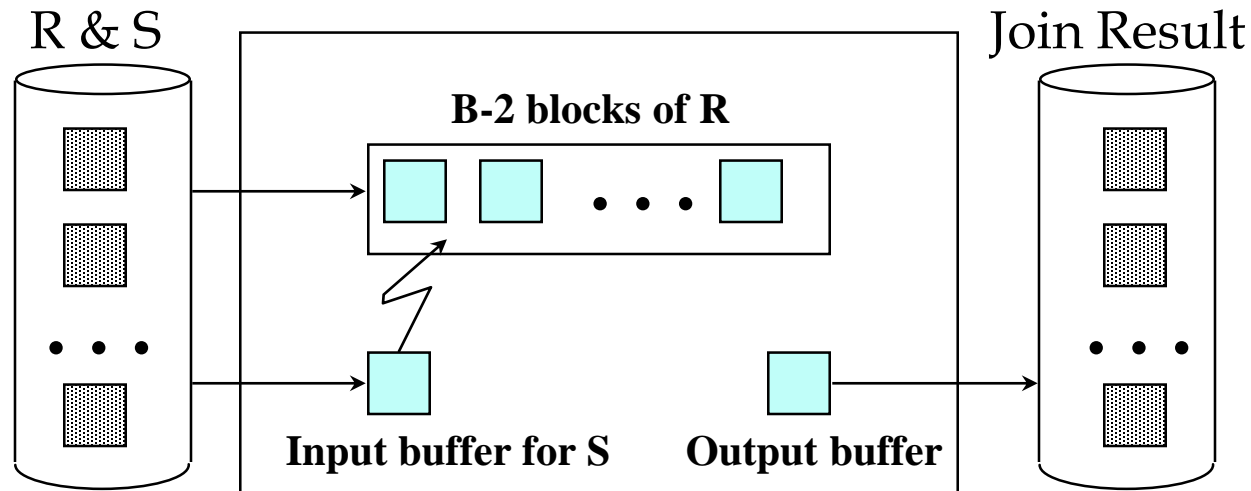
- ❖ Hash-index (Alt. 2) on *sid* of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100\*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (exactly one) matching Sailors tuple.  
Total:  $1000 + 2.2 * 100,000 = 221,000$  I/Os.
- ❖ Hash-index (Alt. 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.  
Total =  $500 + (1.2 + 2) * 40,000 = 128,500$





# Block Nested Loops (BNL) Join

- ❖ Small twist on Simple Nested Loops.
- ❖ Allocate  $B$  buffer-pool pages
- ❖ Use one page as an input buffer for scanning the inner relation,  $S$ , one page as the output buffer, and use all remaining  $B-2$  pages to hold a “block” of outer  $R$ .
  - For each matching tuple  $r$  in  $R$ -block,  $s$  in  $S$ -page, add  $\langle r, s \rangle$  to result. Then read next  $R$ -block, scan  $S$ , etc.





# *Examples of Block Nested Loops*

- ❖ **Cost:**  $M + \lceil M / (B - 2) \rceil N$
- ❖ With Reserves (R) as outer and 102 buffer pages:
  - Cost of scanning R is 1000 I/Os over 10 *passes*.
  - Per pass of R, we scan Sailors (S); 10\*500 I/Os.
  - With space for 90 pages of R, we 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.
- ❖ Better yet, double buffer with a pass size of (B-3). Fetch next block while joining current one



# Sort-Merge Join (SMJ) Review

- ❖ Sort R and S on the join column, then scan them to “merge” (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple  $\geq$  current S tuple, then advance scan of S until current S-tuple  $\geq$  current R tuple; do this until current R tuple = current S tuple.
  - At this point, one-or-more,  $\rho$ , R tuples match one-or-more,  $\sigma$ , S tuples; output  $\langle r, s \rangle$  for all pairs of such tuples ( $\rho \times \sigma$ ).
  - Then resume scanning R and S.
- ❖ Cost:  $M \log M + N \log N + (M+N)$



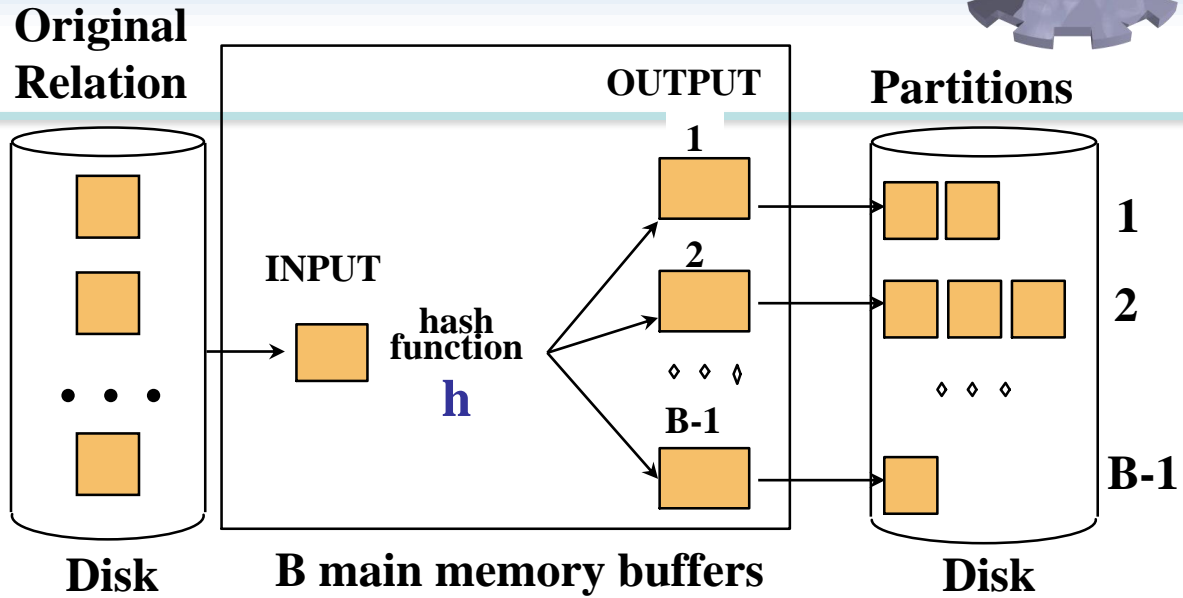
# Refinements of Sort-Merge Join

- ❖ Combine the merging phases of *external sorting* of R and S with the merging required for the join.
  - Using the sorting refinement that merges multiple runs each pass, we sort R and S up to *their last merge pass*.
  - Allocate 1 page per run of each relation, and “merge” while checking the join condition.
  - **Cost:** read+writes in (Pass 0.. Pass N-1) + read each relation in (only) merging pass (+ writing of result tuples).
  - Typically reduces I/O cost by a factor of  $\frac{1}{2}$ .
- ❖ In practice, cost of sort-merge join, like the cost of external sorting, is nearly *linear*.

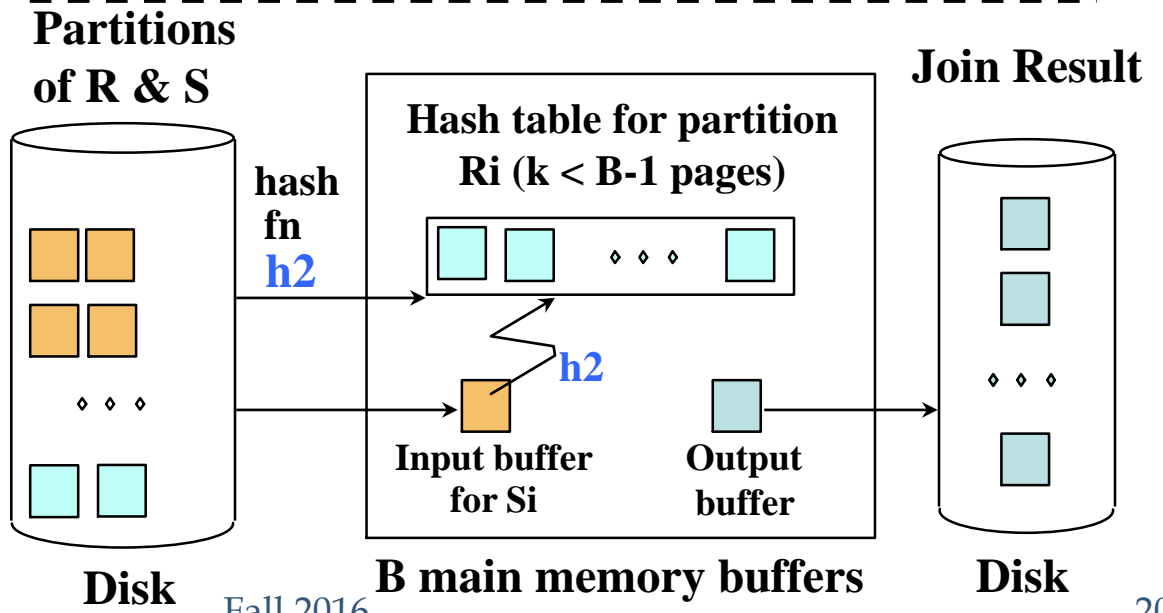


# Hash-Join

- Partition both relations using a common 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  $h_2$  ( $\neq h!$ ). Scan matching partition of  $S$ , search for matches.





# *Observations on Hash-Join*

---

- ❖ We want each partition of  $R$  to fit in  $B-2$  buffer pages, so #partitions,  $k = M / (B - 2)$ , if we assume no skew
- ❖ 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 this partition and 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.
- ❖ In our running example, this is a total of 4500 I/Os.
- ❖ Sort-Merge Join vs. Hash Join:
  - Both have a cost of  $3(M+N)$  I/Os. Hash-Join is superior if relation sizes differ greatly. Also, Hash-Join shown to be highly parallelizable.
  - Sort-Merge insensitive to data skew; and result is sorted.





# General Join Conditions

- ❖ Equalities over several attributes (e.g.,  $R.sid=S.sid$  AND  $R.rname=S.sname$ ):
  - For Index NL, build index on  $\langle sid, sname \rangle$  (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.
    - Perform 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.



# 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.
  - *Alternative*: Merge runs from final pass of *both* relations.
- ❖ Hash based approach to union:
  - Partition R and S using hash function  $h$ .
- ❖ Set Subtraction, Intersection (modified merge passes)
  - R - S Subtract – write to output if key appears in R but not S
  - $R \cap S$  Intersection – write to output if keys match



# Aggregate Operations (*AVG, MIN, etc.*)

## ❖ 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.

## ❖ With grouping:

- Sort on group-by attributes, then scan relation and compute aggregate for each group. (Can improve upon this 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*

- ❖ 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.