COURSE 7

Evaluation of Relational Operators

Relational Operators

- We will consider how to implement:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Join* ($\otimes$) Allows us to combine two relations.
  - *Set-difference* (—) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY
- Since each operation returns a relation, operations can be *composed*! After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

Why is it important?

How does the DBMS know when to use indexes?

An SQL query can be executed in many ways. Which one is best?

- Perform selection before or after join?
- Many ways of implementing a join, how to choose the right one?

The DBMS does this automatically, but we need to understand it to know what performance to expect.

Technics to Implement Operators

Iteration

- Sometimes, faster to scan all tuples even if there is an index.
- Sometimes, we can scan the data entries in an index instead of the table itself.)

Indexing

- Can use WHERE conditions to retrieve small set of tuples (selections, joins)

Partitioning

- Using sorting or hashing: partition the input tuples and replace an expensive operation by similar operations on smaller inputs

Access Path

Access path = way of retrieving tuples

- File scan or index that matches a selection (in the query)
- Cost depends heavily on access path selected

A tree index matches (a conjunction of) conditions that involve only attributes in a prefix of the search key.

A hash index matches (a conjunction of) conditions that has a term *attribute = value* for every attribute in the search key of the index

Selection conditions are first converted to conjunctive normal form (CNF)

Matching an Index

Search key <a,b,c>

	Condition	B+Tree Index	Hash Index
1	a=5 AND b=3	☑	☒
2	a>5 AND b<3	☑	☒
3	b=3	☒	☒
4	a=7 AND b=5 AND c=4 AND d>4	☑	☑
5	a=7 and c=5	☑	☒

Index matches (part of) a predicate if:

- Conjunction of terms involving only attributes (no disj)
- Hash: only equality op, predicate has all index attributes
- B+Tree: Attributes are a prefix of the search key, any ops.

Selectivity of Access Path

Selectivity = Number of pages retrieved (index+data pages)

Steps:

- Find the most selective access path,
- Retrieve tuples using it
- Apply any remaining terms that don't match the index

Most selective access path minimizes retrieval cost!

Schema for Examples

Students (*sid*: integer, *sname*: string, *age*: integer)
Courses (*cid*: integer, *name*: string, *location*: string)
Evaluations (*sid*: integer, *cid*: integer, *day*: date, *grade*: integer)

- *Students*:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
- *Courses*:
  - Each tuple is 50 bytes long, 80 tuples per page, 100 pages.
- *Evaluations*:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.

Equality Joins With One Join Column

```
SELECT  *
FROM   Evaluations R, Students S
WHERE   R.sid=S.sid
```

- In algebra: R $\otimes$ S. Must be carefully optimized. R $\times$ S is large; so, R $\times$ S followed by a selection is inefficient.
- Assume: M pages in R, p_R tuples per page, N pages in S, p_S tuples per page.
  - In our examples, R is *Evaluations* and S is *Students*.
- We will consider more complex join conditions later.
- *Cost metric*: number of I/Os.

Techniques to Implement Join

- Iteration
  - Simple/Page-Oriented Nested Loops
  - Block Nested Loops
- Indexing
  - Index Nested Loops
- Partition
  - Sort Merge Join
  - Hash

Simple Nested Loops Join

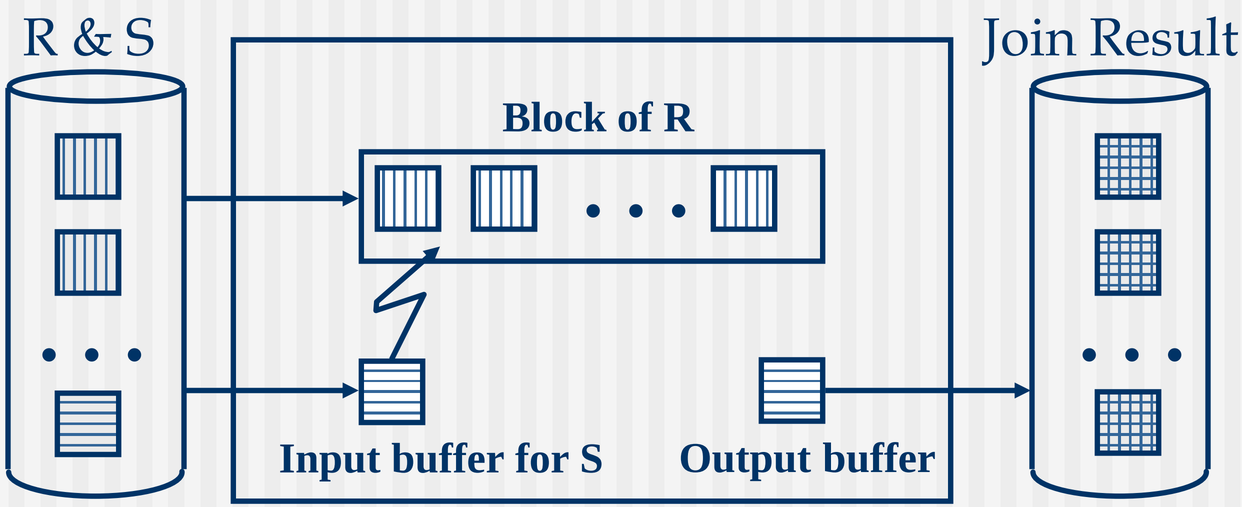
foreach tuple r in R do
foreach tuple s in S do
if $r_i == s_j$ then add <r, s> to result

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
  - Cost: $M + p_R * M * N = 1000 + 100*1000*500$ 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: $M + M*N = 1000 + 1000*500$
  - If smaller relation (S) is outer, cost = 500 + 500*1000

Block Nested Loops Join

- 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.



Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil$no of pages of outer / blocksize$\rceil$
- With *Evaluations* (R) as outer, and 100-pages block of R:
  - Cost of scanning R is 1000 I/Os; a total of 10 *blocks*.
  - Per block of R, we scan *Students* (S); 10*500 I/Os.
  - If space for just 90 pages of R, we would scan S 12 times.
- With 100-page block of *Students* as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan *Evaluations*; 5*1000 I/Os.
- With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.

Index Nested Loops Join

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 + (($M*p_R$) * 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 (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical)
  - Un-clustered: up to 1 I/O per matching S tuple.

Examples of Index Nested Loops

- Hash-index (Alt. 2) on *sid* of *Students* (as inner):
  - Scan *Evaluations*: 1000 page I/Os, 100*1000 tuples.
  - For each *Evaluations* tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching *Students* tuple ⇒ cost 220,000. Total: 221,000 I/Os.
- Hash-index (Alt. 2) on *sid* of *Evaluations* (as inner):
  - Scan *Students*: 500 page I/Os, 80*500 tuples.
  - For each *Evaluations* tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching *Evaluations* tuples. Assuming uniform distribution, 2.5 evaluations per student (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered. Total: from 88,500 to 148,500 I/Os

Sort-Merge Join $(R \otimes_{i=j} S)$

- Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple > current S tuple, then advance scan of S until current S-tuple > current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in R_i (*current R group*) and all S tuples with same value in S_j (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

Example of Sort-Merge Join

sid	*sname*	*age*
22	dustin	20
28	yuppy	21
31	johnny	20
44	guppy	22
58	rusty	21

sid	*cid*	*day*	*grade*
28	101	15/6/04	8
28	102	22/6/04	8
31	101	15/6/04	9
31	102	22/6/04	10
31	103	30/6/04	10
58	101	16/6/04	7

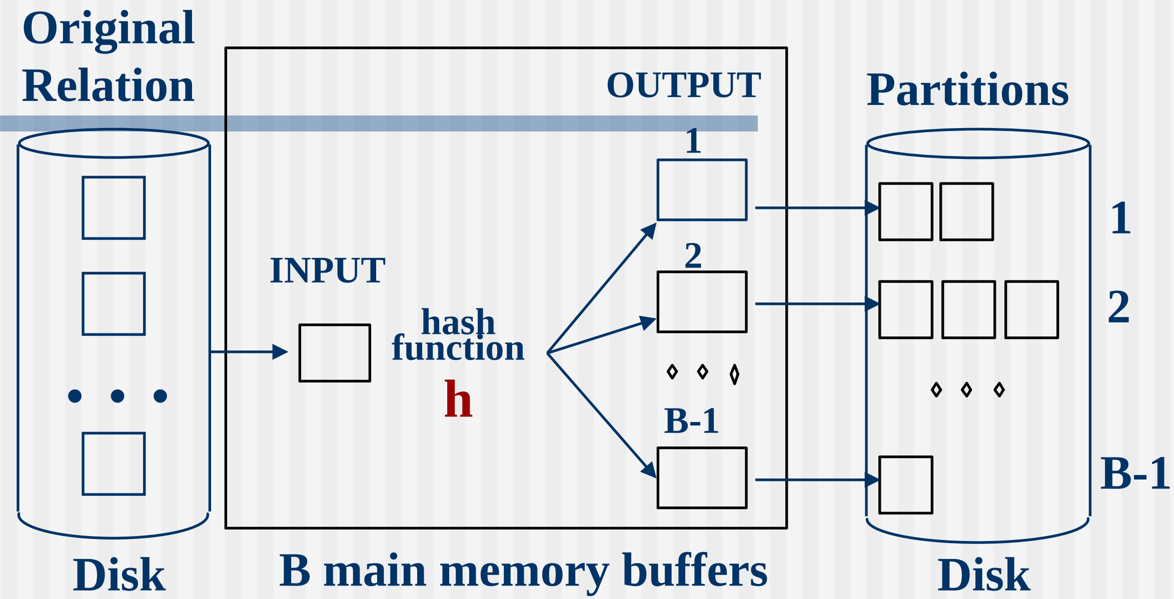
- Cost: $M \log_2 M + N \log_2 N + (M+N)$
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- With 35, 100 or 300 buffer pages, both *Evaluations* and *Students* can be sorted in 2 passes; total join cost: 7500.

Refinement of Sort-Merge Join

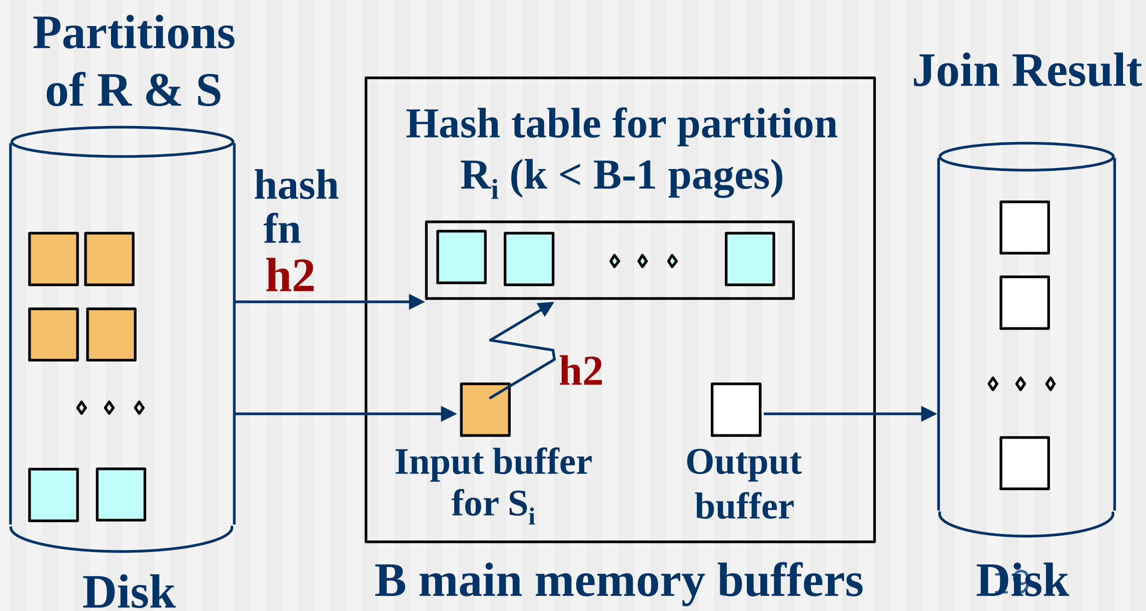
- We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
  - With B > $\sqrt{L}$, where *L* is the size of the larger relation, using the sorting refinement that produces runs of length 2B in Pass 0, number of runs of each relation is < B/2.
  - Allocate 1 page per run of each relation, and `merge' while checking the join condition.
  - Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
  - In example, cost goes down from 7500 to 4500 I/Os.
- In practice, cost of sort-merge join, like the cost of external sorting, is *linear*.

Hash-Join

- Partition both relations using hash fn **h**: R tuples in partition *i* will only match S tuples in partition *i*.



Read in a partition of R, hash it using **h2** (<> **h**!). Scan matching partition of S, search for matches.



Observations on Hash-Join

- number of partitions $k < B-1$, and $B-2 >$ size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing k, we get:
  - $k = B-1$, and $M/(B-1) < B-2$, i.e., B must be $> \sqrt{M}$
- 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 do the join of this R-partition with corresponding S-partition.

Cost of Hash-Join

- In partitioning phase, read+write both relations; 2(M+N). In matching phase, read both relations; M+N I/Os.
- In our running example, this is a total of 4500 I/Os.
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory (*what is this, for each?*) both have a cost of 3(M+N) I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; 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 <*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.
- Inequality conditions (e.g., *R.rname* < *S.sname*):
  - For Index NL, need (clustered!) B+ tree index.
    - Range probes on inner; number of 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.