Fall 2013

QUERY PROCESSING

[JOINS, SET OPERATIONS, AND AGGREGATES]

Joins

- The focus here is on “equijoins”
- These are very common, given how we design the database schemas using primary and foreign keys
- Equijoins are used to bring the tuples back together
- Example:

```
SELECT U.login AS login, COUNT(*) AS NumMsgsToday
FROM   User U, Messages M
WHERE  U.uid = M.uid
  AND  M.Date(tstamp) = CURRENT_DATE   -- select msgs posted today
GROUP BY U.login                       -- group by login
ORDER BY NumMsgsToday DESC             -- order by descending msg count
```

We look at equijoin algorithms next

Page Nested Loops Join: PNL

1. For each page in the User table, p_u
2. For each page of Message, p_m
3. Join the tuples on page p_u with tuple in p_m
4. Output matching tuples (after applying any projection)

Let |U| denote the # pages in the User table and
|M| denote the # pages in the Messages table,

Then, the IO cost of the PNL Algorithm is:

How many buffer pool pages does this algorithm use?

Can we do better if we have a larger buffer pool with B pages? Where, B >> 3

Block Nested Loops Join: BNL

1. Scan the User table B-2 pages at a time

2. For each page of Message, p_m
3. Probe the hash table with each tuple on the page p_m
4. Output matching tuples (after applying any projection)

Let |U| denote the # pages in the User table and
|M| denote the # pages in the Messages table,

Then, the IO cost of the BNL Algorithm is: O( )

What is the CPU cost for this algorithm?

Block Nested Loops Join: BNL

1. Scan the User table B-2 pages at a time
2. Insert the user tuples into an in-memory hash table on the join attribute
3. For each page of Message, p_m
4. Probe the hash table with each tuple on the page p_m
5. Output matching tuples (after applying any projection)

Let |U| denote the # pages in the User table and
|M| denote the # pages in the Messages table,

Then, the IO cost of the BNL Algorithm is: O( )

Index Nested Loops Join: INL

Can be used when there is an index on the join attribute on one of the tables

1. For each page in the User table, p_u
2. For each tuple on page p_u
3. Probe the Index on the join attribute on Messages
4. Output matching tuples (after applying any projection)

Let |U| denote the # pages in the User table, and

||U|| denote the # tuples in the User table, and

I_m denote the cost of one index probe on the Messages table

Then, the IO cost of the PNL Algorithm is:

The cost of I_m depends on the type of the index and if the index is clustered or unclustered

How many buffer pages does this use?

Blocked Index Nested Loops Join: BINL

1. Scan the User table B-2 pages at a time
2. Sort the tuples in the B-2 pages on the join key
3. For each tuple of the User table
4. Probe the Index on the join attribute on Messages
5. Output matching tuples (after applying any projection)

Why does sorting help?

Sort-Merge Join: SMJ

1. Generate sorted runs for U (Pass 0)
2. Generate sorted runs for M (Pass 0)
3. Merge the sorted runs for U and M
4. While merging check for the join condition
5. Output matching tuples

Runs of U on average are ~2B pages long (with B buffer pages)
Runs of M are also ~2B pages long

So we have |U|/2B and |M|/2B runs after Line 2
Need to hold one page from each of these runs in memory for Line 3
So, $|U|/2B + |M|/2B \leq B$
If |M| is the larger relation, then this means that a "safe" criteria is:

(Simple) Hash Join Algorithm: HJ

1. Partition U into P partitions, using a hash function h1 on the join key
2. Partition M into P partitions, using a hash function h1 on the join key
3. Join each partition of U with the corresponding partition of M
4. (using hashing as in BNL, so build a hash table on the U partition)

// Note the hash function in the second part must be different from h1

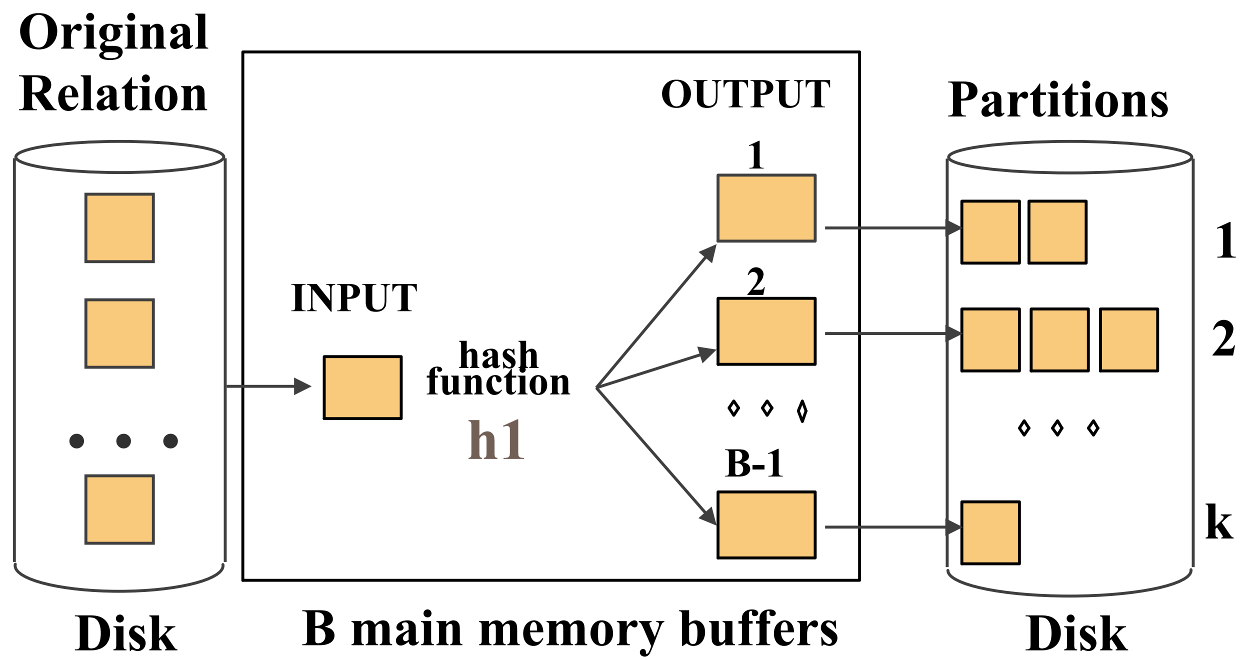
With B buffer pages, # partitions is ~B (for each U and M)

Each partition of U must fit in memory (with its hash table).

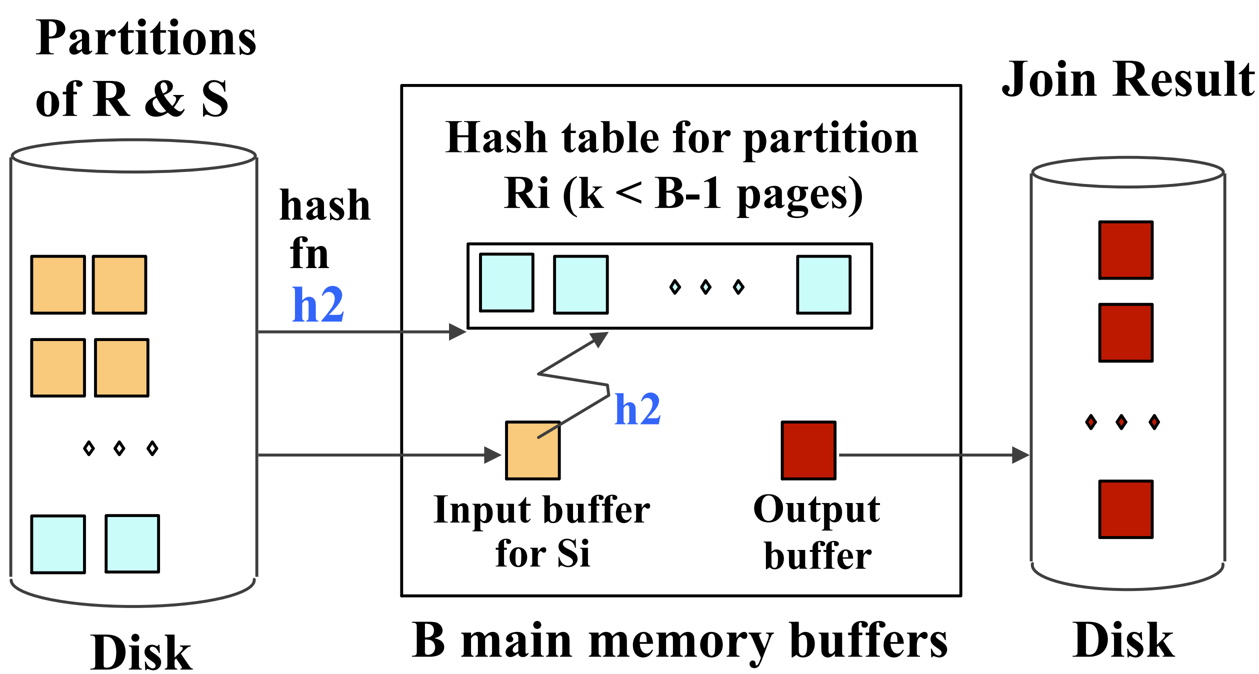
Assume that the hash table increases the space required by a factor of F.

Thus, the largest U that can be joined in two passes is constrained by:

Hash-Join



What if $f^*|U_i| > B-2$?



Hash Join versus Sort-Merge Join

- Need to join U with M, where |M| > |U|, using B buffer pages
- To do a two-pass join, SMJ needs
  - In this case the IO cost is: 3 * (|U| + |M|)
- To do a two-pass join, HJ needs
  - In this case the IO cost is: 3 * (|U| + |M|)

So HJ can sort two relations with fewer buffer pages!

General Join Conditions

- Equalities over several attributes
  e.g., *R.sid=S.sid* AND *R.rname=S.sname*:
  - Index NL
    - index on *<sid, sname>*
    - index on *sid* or *sname*.
  - SM and Hash, sort/hash on combination of join attrs
- Inequality conditions (e.g., *R.rname < S.sname*):
  - For Index NL, need (clustered!) B+ tree index.
    - Large # index matches
  - SM and Hash not applicable
  - Block NL likely to be the winner

Set Operations

- ∩ and **x** special cases of join
- ∪ and **–** similar; we'll do ∪.
  - Duplicate elimination
- Sorting:
  - Sort both relations (on all attributes).
  - Merge sorted relations eliminating duplicates.
  - *Alternative*: Merge sorted runs from *both* relations.
- Hashing:
  - Partition R and S
  - Build hash table for R_i.
  - Probe with tuples in S_i, add to table if not a duplicate

Aggregates

- Sorting
  - Sort on group by attributes (if any)
  - Scan sorted tuples, computing running aggregate
    - Max: Max
    - Average: Sum, Count
  - If the group by attribute changes, output aggregate result
- Cost: sorting cost

Aggregates

- Hashing
  - Hash on group by attributes (if any)
    - Hash entry: group attributes + running aggregate
  - Scan tuples, probe hash table, update hash entry
  - Scan hash table, and output each hash entry
- Cost: Scan relation!
- What if we have a large # groups?

Aggregates

- Index
  - Without Grouping
    - Can use B+tree on aggregate attribute(s)
    - Where clause?
  - With grouping
    - B+tree on all attributes in SELECT, WHERE and GROUP BY clauses
      - Index-only scan
      - If group-by attributes prefix of search key
        => data entries/tuples retrieved in group-by order
      - Else => get data entries and then use a sort or hash aggregate algorithm