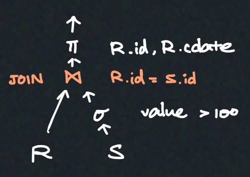
Lec 11 Join Algorithms

Put two tables together — often the most expensive operation
Most typical : join primary & foreign key

Inner equijoin ⋈

Opemisation : try to make smaller table the outter table

Dataflow :



Semantic :

```
for row r in R :
    for row s in S :
        if R.id = s.id :
            concat r and s, add to output
```

} Equivalently, do R×S, then filter

Efficiency : depends on col vs row store and OLAP vs OLTP
- row: usually more IO
- col: usually more work to track which records e.g. ~~track~~ ids, use bitmaps, ...

tradeoff btwn early vs late materialisation
- tradeoff: can hybrid
- early: easier to code
- late: hard to refer back to ids/positions

Cost analysis

Join R and S — B
- R: M pages, m tuples
- S: N pages, n tuples ← or relevant pages in col store
- B: buff pool size

Algorithms

▹ Naïve for loop

```
for r ∈ R :        ← "outer table"
    for s ∈ S :
        if r, s match , emit
```

$M + (mN)$
or $N + (nM)$
by swapping loop order

▹ Block nested loop

```
for $B_R$ in R :        ← block of multiple (or one) pages
    for $B_S$ in S :
        for r ∈ $B_R$ :
            for s ∈ $B_S$ :
                if r, s match , emit
```

$M + (MN)$
if 3 buf poll pages

sequential flooding ! if S has more blocks than buf pool size we can't keep them all pinned for the whole scan

if more pages available pin more pages in R in a time reduce S scan count

Multi-page block

```
for $B_R^{B-2}$ in R :        ← page count ; ← try use smaller table as outer
    for $B_S'$ in S :
        for r ∈ $B_R^{B-2}$ :
            for s ∈ $B_S'$ :
                if r, s match , emit
```

→ Use B-2 bufs for outer table, 1 buf to scan inner, and 1 to write output
↳ $M + (\lceil M/(B-2) \rceil N)$

if inner table fits in B-2 bufs, just keep it in mem works too

▹ Index nested loop

```
for r ∈ R :
    lookup r in index of S
```

$M + (m\overset{\uparrow}{C})$
index lookup cost

▹ Sort-Merge Join

Idea: sort both independently and traverse both with two cursors keep track of where mini scans for jumping back if hitting duplicate

Optimisation : combine the merge & traverse steps

Worse case : many duplicates — need to go back all the time

Good case : already sorted tables

▹ Hash Join

1. Build hash table for small table use hash func h_1
   - hash table: value depends on early vs late materialisation
2. Probe the hash table when traversing other table

Worse case : all duplicate

Optimisation : bloom filter on R. Probe bloom filter before hash table lookup

Try fit hash table in mem. Otherwise partition ...

▹ GRACE hash join

1. Hash R into k buckets with h_1] write to disk if necessary
2. " S " k " h_1]
3. Join each partition pair
4. Merge the results

Corner cases : everything go into same partition ... some partition too big ...
→ recursively partition by another hash func

Cost : 3(M + N)
└ read, write partitions , read partitions

Optimisation : hybrid hash join
if only need two partitions, keep 0th part. in mem
etc.

Observations
- The probe-side table can be any size