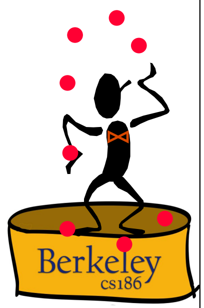
Join Algorithms

R&G 14.4



Rendezvous

- Grouping/Aggregation is one kind of rendezvous
  - groups of matching items within a single table
- Join is the other main kind of rendezvous
  - combinations of items from multiple tables

Cross Product

Given two collections R and S:

R × S => all pairs {r, s} of items in R, S
- a.k.a Cartesian product

Can always do this to create a single "supertable" and then use a single-table query
- correct but rarely best performing

"Theta" Join

- R $\bowtie_\theta$ S: all pairs {r,s} where θ(r,s)
  - e.g. FriendRequests $\bowtie_\theta$ Users
    - θ is "friendID = ID"
  - e.g. Family $\bowtie_\theta$ Family
    - θ is "age < age"
- A common case: EquiJoin
  - i.e., θ is an equality test
  - special case: one side of = is a "key"
    - E.g. Enrolled.studentID = Students.ID
    - This is like doing "lookups" into the Students table

Schema for Examples

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

- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  - [S]=500, p_S=80.
- Reserves:
  - Each tuple is 40 bytes, 100 tuples per page, 1000 pages.
  - [R]=1000, p_R=100.

Joins

```
SELECT *
FROM    Reserves R1, Sailors S1
WHERE  R1.sid=S1.sid
```

- Joins are very common.
- R × S is large
  - so, R × S followed by a "filter" is inefficient.
- Many approaches to reduce join cost.
- Join techniques we will cover today:
  - Nested-loops join
  - Index-nested loops join
  - Sort-merge join
  - Hash Joins

Some Cost Notation

- [R] : the number of pages to store R
- p_R : number of records per page of R
- |R| : the number of records in R
  - *cardinality*
- Note: p_R*[R] = |R|

Simple Nested Loops Join

R ⋈ S: foreach record r in R do
  foreach record s in S do
    if θ(r_i, s_j) then add <r, s> to result

- Cost = (p_R*[R])*[S] + [R] = 100*1000*500 + 1000 IOs
  - At 10ms/IO, Total time: ???
- What if smaller relation (S) was "outer"?
- What assumptions are being made here?
- What is cost if one relation can fit entirely in memory?

Page-Oriented NestLoop Join

R ⋈ S: foreach page b_R in R do
  foreach page b_S in S do
    foreach record r in b_R do
      foreach record s in b_S do
        if θ(r_i,s_j) then add <r, s> to result

- Cost = [R]*[S] + [R] = 1000*500 + 1000
- If smaller relation (S) is outer, cost = 500*1000 + 500
- Much better than naïve per-tuple approach!

Page-Oriented NestLoop Join

R ⋈ S: foreach page b_R in R do
  foreach page b_S in S do
    foreach record r in b_R do
      foreach record s in b_S do
        if θ(r_i,s_j) then add <r, s> to result

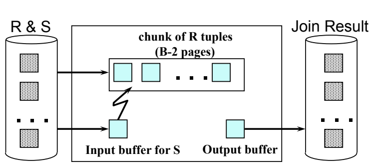
I/O for S

I/O for R

- Cost = [R]*[S] + [R] = 1000*500 + 1000
- If smaller relation (S) is outer, cost = 500*1000 + 500
- Much better than naïve per-tuple approach!

~~Block~~ Nested Loops Join
CHUNK

- Page-oriented NL doesn't exploit extra buffers :(
- Idea to use memory efficiently:



Cost: Scan outer + (#outer chunks * scan inner)

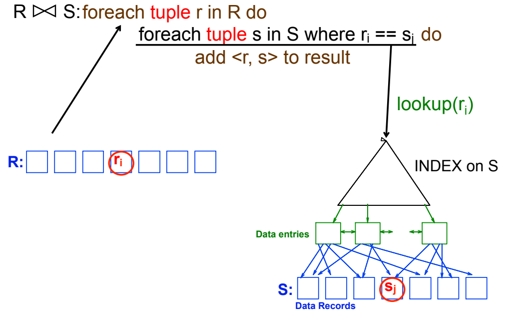
#outer chunks = $\lceil [outer] / chunksize \rceil$

~~Block~~ NestLoop Examples
CHUNK

- Say we have B = 100+2 memory buffers
- Join cost = [outer] + (outer-chunks * [inner])
  - #outer chunks = [outer] / 100
- With R as outer ([R] = 1000):
  - Scanning R costs 1000 IO's (done in 10 chunks)
  - Per chunk of R, we scan S; costs 10*500 I/Os
  - Total = 1000 + 10*500.
- With S as outer ([S] = 500):
  - Scanning S costs 500 IO's (done in 5 chunks)
  - Per chunk of S, we scan R; costs 5*1000 IO's
  - Total = 500 + 5*1000.

Index Nested Loops Join

R ⋈ S: foreach tuple r in R do
foreach tuple s in S where $r_i == s_j$ do
add <r, s> to result



Index Nested Loops Join

R ⋈ S: foreach tuple r in R do
foreach tuple s in S where $r_i == s_j$ do
add <r, s> to result

Cost = [R] + ([R]*p_R) * cost to find matching S tuples

- If index uses Alt. 1, cost = cost to traverse tree from root to leaf.
- For Alt. 2 or 3:
  1. Cost to lookup RID(s); typically 2-4 IO's for B+Tree.
  2. Cost to retrieve records from RID(s); depends on clustering.
     - Clustered index: 1 I/O per page of matching S tuples.
     - Unclustered: up to 1 I/O per matching S tuple.

Don't worry .. we will cover indexes in detail in a few weeks.

Sort-Merge Join

1. Sort R on join attr(s)
2. Sort S on join attr(s)
3. Scan sorted-R and sorted-S in tandem, to find matches

```
SELECT *
FROM   Reserves R1, Sailors S1
WHERE  R1.sid=S1.sid
```

Example:

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Cost of Sort-Merge Join

- Cost: Sort R + Sort S + ([R]+[S])
  - But in worst case, last term could be [R]*[S] (very unlikely!)
  - Q: what is worst case?
- Suppose B = 35 buffer pages:
  - Both R and S can be sorted in 2 passes
  - Total join cost = 4*1000 + 4*500 + (1000 + 500) = 7500
- Suppose B = 300 buffer pages:
  - Again, both R and S sorted in 2 passes
  - Total join cost = 7500

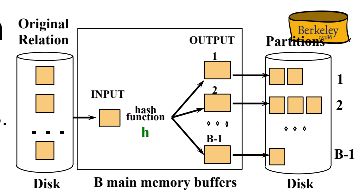
Chunk-Nested-Loop cost = 2500 … 15,000

Other Considerations …

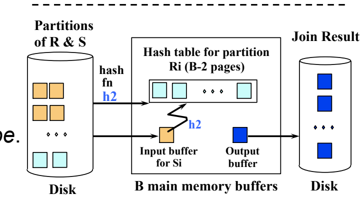
- An important refinement:
  - Do the join during the final merging pass of sort !
  - If have enough memory, can do:
    1. Read R and write out sorted runs (pass 0)
    2. Read S and write out sorted runs (pass 0)
    3. Merge R-runs and S-runs, *while* finding R ⋈ S matches
  - Cost = 3*[R] + 3*[S]
  - Q: how much memory is "enough" (remember?)
- Sort-merge join an especially good choice if:
  - one or both inputs are already sorted on join attribute(s)
  - output is required to be sorted on join attributes(s)

Hash Join

Do this for R.
Then do it for S.



Build an R H.T.
Stream S & *probe*.



Cost of Hash Join

- Partitioning phase: read+write both relations
  - ⇒ 2([R]+[S]) I/Os
- Matching phase: read both relations, write output
  - ⇒ [R]+[S] + [output] I/Os
- Total cost of 2-pass hash join = 3([R]+[S])+[output]

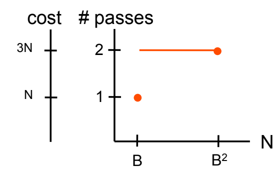
Q: what is cost of 2-pass *sort-merge join*?

Q: how much memory needed for 2-pass *sort-merge join*?

Q: how much memory needed for 2-pass *hash join*?

Exploit excess memory

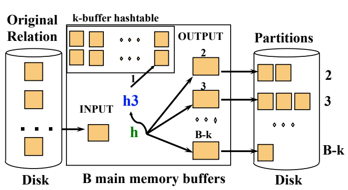
- Have B memory buffers
- Want to hash relation of size N blocks



If B < N < B², will have unused memory …

Hybrid Hashing

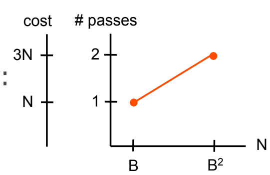
Idea: keep one of the hash buckets in memory!



Q: how do we choose the value of *k*?

Cost savings: hybrid hashing

- Now:



Hash Join vs. Sort-Merge Join

- Sorting pros:
  - Good if input already sorted, or need output sorted
  - Not sensitive to data skew or bad hash functions
- Hashing pros:
  - Can be cheaper due to hybrid hashing
  - For join: # passes depends on *size of smaller relation*
  - Good if input already hashed, or need output hashed

Recap

- Nested Loops Join
  - Works for arbitrary Θ
  - Make sure to utilize memory in "chunks"
- Index Nested Loops
  - For equi-joins
  - When you already have an index on one side
- Sort/Hash
  - For equi-joins
  - No index required
- No clear winners – may want to implement them all
- Be sure you know the cost model for each
  - You will need it for query optimization!