CSE462/562: Database Systems (Spring 22)

Lecture 14: Join algorithms

4/7/2022

University at Buffalo
Department of Computer Science
and Engineering
School of Engineering and Applied Sciences

Joins

- Joins are very common
  - need to reconstruct complete rows due to schema normalization
  - collecting correlated data (e.g., sliding window on timestamps, spatial joins, etc.)
- Joins are very expensive!
  - join results can be as large as the cartesian product
  - but they are usually far from the full cartesian product
    - can we avoid evaluating the full cartesian product?
- Many approaches to reduce join cost
  - Nested-loop join (simple/block/indexed)
  - Sort-merge join
  - Hash join (basic hash partitioning vs hybrid hashing)

Running example

- A quick recap on our running example

- Notations: for relation R
  - T_R: number of records, N_R: number of pages in its heap file, B_R: (average) number of tuples per page
  - h_I: height of a B-tree index I over the file
  - M: private workspace size in pages
- Running example
  - $t_S = 4\ ms$, $t_T = 0.1\ ms$, *4000-byte page*
  - Student: R(sid: int, name: varchar(19), login: varchar(19), major: char(2), adm_year: int)
    - 50 bytes/tuple, $B_R = 80$, $T_R = 40{,}000$, $N_R = 500$
  - Enrollment: E(sid: int, semester: char(3), cno: int, grade: double)
    - 20 bytes/tuple, $B_E = 200$, $T_E = 200{,}000$, $N_E = 1000$
  - *Consider the equi-join* $R \bowtie_{R.sid=E.sid} E$ (denote the join predicate $R.sid = E.sid$ as θ)
    - R is called the outer relation, E is called the inner relation
    - cost = #seeks × t_S + #page_transfers × t_T
    - ignoring buffer effect; not counting the final output

Simple nested-loop join

$\theta: S.sid = E.sid$

- For each tuple in the outer relation R,
  - scan the entire inner relation S

```
foreach tuple r in R do
  foreach tuple e in E do
    if (r,e) satisfies θ then
      emit r∘e as result
```

- Simple nested-loop join evaluates the full cartesian product
  - only keep those pairs that satisfy the predicate
- Cost? depends on the available memory
  - If $M = 2$, we'll have to read every pages in the inner relation once for every tuple in the outer relation
    - number of pages to read: $\mathrm{N_R} + T_R N_E$
    - number of seeks: $\mathrm{N_R} + \mathrm{T_R}$ (one seek for every page in R, and one seek for every scan of E)
    - cost = $t_T(N_R + T_R N_E) + t_S(N_R + T_R)$
    - running example: $\mathrm{cost}(\mathrm{R} \bowtie E) \approx 4162\ s \approx 1.15\ hr$!
      - What about $cost(E \bowtie R)$?
        - $t_T(N_E + T_E N_R) + t_S(N_E + T_E) \approx 10804\ s \approx 3\ hr$

Simple nested-loop join

$\theta: S.sid = E.sid$

- For each tuple in the outer relation R,
  - scan the entire inner relation S

```
foreach tuple r in R do
  foreach tuple e in E do
    if (r,e) satisfies θ then
      emit r∘e as result
```

- Simple nested-loop join evaluates the full cartesian product
  - only keep those pairs that satisfy the predicate
- Cost? depends on the available memory
  - If $M = 2$, cost = $t_T(N_R + T_R N_E) + t_S(N_R + T_R)$
  - If $M \geq N_E + 2$, we can cache the inner relation E in memory
    - number of pages to read: $N_R + N_E$
    - number of seeks: 2 (scanning E in full, followed by scan of R)
    - cost = $t_T(N_R + N_E) + 2t_S = 0.158\ s$
  - How to fully utilize the memory if $3 \leq M < N_E + 2$?

Block nested-loop join

$\theta: S.sid = E.sid$

- For each block for the outer relation S and every block of the inner relation E,
  - first assume each block is a page
  - emit the pairs of records (r, s) that satisfy the join predicate θ

```
foreach block B_S in S do
  foreach block B_E in E do
    foreach tuple r in B_S do
      foreach tuple e in B_E do
        if (r,e) satisfies θ then
          emit r∘e as result
```

- Block nested-loops only reads each page in the outer relation once
  - Cost = $t_T(N_R + N_R N_E) + 2t_S N_R$ = 54.5 s (block nested-loop) vs 1.15 hr (simple nested loop)
    - What about $E \bowtie S$?
      - cost = *58.1 s -- use smaller relation as the outer relation*

Block nested-loop join

$\theta: S.sid = E.sid$

- For each block for the outer relation S and every block of the inner relation E,
  - first assume each block is a page
  - emit the pairs of records (r, s) that satisfy the join predicate θ

```
foreach block B_S in S do
  foreach block B_E in E do
    foreach tuple r in B_S do
      foreach tuple e in B_E do
        if (r,e) satisfies θ then
          emit r∘e as result
```

- Block nested-loops only reads each page in the outer relation once
  - Cost = $t_T(N_R + N_R N_E) + 2t_S N_R = 54.5\ s$ (block nested-loop) vs 1.15 hr (simple nested loop)
  - Only uses 3 buffer frames. What about M > 3 buffer frames?
    - Read every $M - 2$ pages at a time for the outer relation, i.e., $|B_S| = M - 2$
      - cost = $t_T\left(N_R + \left\lceil\frac{N_R}{M-2}\right\rceil N_E\right) + 2t_S\left\lceil\frac{N_R}{M-2}\right\rceil$
      - $M = 12$ => cost = $5.45\ s$, $M = 102$ => cost = $0.59\ s$
        - *caveat: CPU cost may not be negligible when I/O cost is low for NL/BNL*

Index nested-loop join

$\theta: S.sid = E.sid$

- If there's an index over the inner relation's join attribute (e.g., $E.sid$)
  - only fetch records with matching values in the join attribute using the index

```
foreach block B_S in S do
  foreach tuple r in B_S do
    foreach tuple e in B_E s.t. S.sid = E.sid do
      emit r ∘ e as result
```

- Assuming heap scan over the outer relation S and block size $|B_S| = 1$
  - cost $= N_R(t_S + t_R) + T_R \times c$
    - where c is the average time for scanning all the matching record for a tuple $r \in R$
    - c depends on
      - selectivity s_E or join degree $d = s_E N_E$
        - special case foreign-key join: d = 1 or $s_E = 1/N_E$
      - clustered vs unclustered index
      - data entry alternatives

Index nested-loop join

$\theta: S.sid = E.sid$

- If there's an index over the inner relation's join attribute (e.g., $E.sid$)
  - only fetch records with matching values in the join attribute using the index

```
foreach block B_S in S do
  foreach tuple r in B_S do
    foreach tuple e in B_E s.t. S.sid = E.sid do
      emit r ∘ e as result
```

- Assuming heap scan over the outer relation S and block size $|B_S| = 1$
  - cost $= N_R(t_S + t_R) + T_R \times c$
    - where c is the average time for scanning all the matching record for a tuple $r \in R$
    - c depends on
      - selectivity s_E or join degree $d = s_E N_E$
        - special case foreign-key join: d = 1 or $s_E = 1/N_E$
      - clustered vs unclustered index
      - data entry alternatives

Index nested-loop join

$\theta: S.sid = E.sid$

- $R \bowtie_{S.sid=E.sid} E$
  - BNL cost = 54.5 s
- Example 1: E as inner, B-Tree index over $E(sid)$, alternative 2, clustered, height $h = 3$
  - assuming uniformity, average join degree $d = \frac{T_E}{T_R} = 5$
  - for each inner table scan, h random I/Os for tree search, 1 seek and $\left\lceil \frac{d}{B_E} \right\rceil = 1$ heap pages read
    - $c = h(t_S + t_T) + t_S + \left\lceil \frac{d}{B_E} \right\rceil t_T = 16.1\ ms$
  - total = $N_R(t_S + t_R) + T_R \times c = 646.05\ s$
- Example 2: E as inner, B-Tree index over $E(sid)$, alternative 2, unclustered, height $h = 3$
  - still $d = 5$
  - for each inner table scan, h random I/Os for tree search, 5 random I/Os for reading 5 heap records
    - $c = h(t_S + t_T) + d(t_S + t_T) = 32.8\ ms$
  - total = $N_R(t_S + t_R) + T_R \times c$ = 1314.05 s

Index nested-loop join

$\theta: S.sid = E.sid$

- Now consider $\sigma_{adm\_year=2021} R \bowtie_{S.sid=E.sid} E$, assuming selectivity of $adm\_year = 2021$ is $s = 0.001$
  - suppose we have an unclustered B-Tree index over $\mathrm{R}(adm\_year)$, $h_1 = 2$
    - can use the index to find all the $[sT_R] = 40$ records
    - Using nested loop for join, need to scan the inner for every $s \in \sigma_{adm\_year=2021}$
      - cost = $(h_1 + [sT_R])(t_S + t_T) + [sT_R](t_S + t_T N_E) \approx 4.33\ s$
- Example 3: E as inner, B-Tree index over $E(sid)$, alternative 2, clustered, height $h = 3$, d = 5
  - for each inner table scan, h random I/Os for tree search, 1 seek and $\left\lceil \frac{d}{B_E} \right\rceil = 1$ heap pages read
    - $c = h(t_S + t_T) + t_S + \left\lceil \frac{d}{B_E} \right\rceil t_T = 16.1\ ms$
  - total = $(h_1 + [sT_R])(t_T + t_S) + [sT_R] \times c \approx 0.82s$
- Example 4: E as inner, B-Tree index over $E(sid)$, alternative 2, unclustered, height $h = 3$, d = 5
  - for each inner table scan, h random I/Os for tree search, 5 random I/Os for reading 5 heap records
    - $c = h(t_S + t_T) + d(t_S + t_T) = 32.8\ ms$
  - total = $(h_1 + [sT_R])(t_T + t_S) + [sT_R] \times c \approx 1.48\ s$

Sort-merge join

$S \bowtie_{S.sid=E.sid} E$

- Idea: sort R on $R.sid$ and sort E on $E.sid$
  "merge" them and emit the pairs with matching values on the join colunms
- Useful if
  - One or both relations are already sorted on the join attributes
    - If not, sort them using external sorting algorithms – this may still be cheaper than BNL
  - Output should be sorted on the join attributes
    - e.g., `SELECT * from R, E WHERE R.sid = E.sid ORDER BY R.sid`
- Algorithm sketch:
  - Naïve version:

```
pr = address of first tuple in R
pe = address of first tuple in E
done = false
while (not done && pe != end && pr != end) do
  if (pe->sid != pr->sid)
     if pe->sid < pr->sid then ++pe else ++pr
     continue
  pr2 = first address after pr such that pr2 == end || pr2->sid != pr->sid
  pe2 = first address after pe such that pe2 == end || pe2->sid != pe->sid
  emit all pairs between [pr, pr2) and [pe, pe2)
  pe = pe2; pr = pr2;
```

Sort-merge join

$S \bowtie_{S.sid=E.sid} E$

- Sort-merge join: naïve version
  - Problem?

student

	sid	name	login	major	adm_year
ps	100	Alice	alicer34	CS	2021
ps2	101	Bob	bob5	CE	2020
	102	Charlie	charlie7	CS	2021
	103	David	davel	CS	2020

enrollment

	sid	semester	cno	grade
pe	100	s22	562	2.0
	100	f21	560	3.7
pe2	101	s21	560	3.3
	101	f21	560	3.3
	102	s22	562	2.3
	102	f21	560	4.0
	103	s22	460	2.7
	103	f21	250	4.0

Sort-merge join

$S \bowtie_{S.sid=E.sid} E$

- Sort-merge join: naïve version
  - Problem? each matched group is scanned for an additional pass

student

	sid	name	login	major	adm_year
	100	Alice	alicer34	CS	2021
ps	101	Bob	bob5	CE	2020
ps2	102	Charlie	charlie7	CS	2021
	103	David	davel	CS	2020

enrollment

	sid	semester	cno	grade
	100	s22	562	2.0
	100	f21	560	3.7
pe	101	s21	560	3.3
	101	f21	560	3.3
pe2	102	s22	562	2.3
	102	f21	560	4.0
	103	s22	460	2.7
	103	f21	250	4.0

Sort-merge join

$S \bowtie_{S.sid=E.sid} E$

- Idea: sort R on $R.sid$ and sort E on $E.sid$
  "merge" them and emit the pairs with matching values on the join columns
- Algorithm sketch:
  - How to ensure R is scanned once, each S group is scanned once per matching $r \in R$?

```
pr = address of first tuple in R
pe = address of first tuple in E
done = false
while (not done && pe != end && pr != end) do
  if (*pe != *pr)
     if *pe < *pr then ++pe else ++pr
     continue
  key = pr->sid
  pe0 = pe
  while pr != end && pr->sid == key
    pe = pe0
    while pe != end && pe->sid == key
       emit *pr ∘ *pe; ++pe
    pe2 = pe; ++pr
  pe = pe2
```

A few caveats in actual implementation:

1. Need to restructure the algorithm to fit into volcano model (project 5)
2. rewinding (setting to a previously saved pointer) on iterator may be expensive!
3. handling NULLs (NULLs never compare equal)

Sort-merge join

- Cost analysis: sorting cost + merge cost, let $M = 110, B = 10, \frac{M}{B} = 11$
  - Sorting cost: $2t_T N_R \left(\left\lceil log_{\lfloor \frac{M}{B} \rfloor -1} \left\lceil \frac{N_R}{M} \right\rceil \right\rceil + 1 \right) + 2t_S \left(\left\lceil \frac{N_R}{M} \right\rceil + \lceil \frac{N_R}{B} \rceil \lceil log_{\lfloor \frac{M}{B} \rfloor -1} \lceil \frac{N_R}{M} \rceil \rceil \right) +$
    $2t_T N_E \left(\left\lceil log_{\lfloor \frac{M}{B} \rfloor -1} \left\lceil \frac{N_E}{M} \right\rceil \right\rceil + 1 \right) + 2t_S \left(\left\lceil \frac{N_E}{M} \right\rceil + \lceil \frac{N_E}{B} \rceil \lceil log_{\lfloor \frac{M}{B} \rfloor -1} \lceil \frac{N_E}{M} \rceil \rceil \right)$
    - includes the cost of writing the sort results to two temporary files
    - running example: sorting cost = $0.64 + 1.28\ s = 1.92\ s$
  - Merge cost: two scans over the temporary files
    - number of pages read: $N_R + N_E = 1500$ (assuming all 5 matching tuples of S are on the same page)
      - *This could be up to $N_R + N_R N_E$ in extreme case (why?)*
    - number of seeks? (depending on the block size)
      - If we fetch one page from R and E at a time, then $N_R + N_E = 1500$
      - If we fetch b $= \left\lfloor \frac{M}{2} - 1 \right\rfloor = 54$ pages at a time for both, then $\left\lceil \frac{N_R}{b} \right\rceil + \left\lceil \frac{N_E}{b} \right\rceil = 10 + 19 = 29$
    - running example: cost = $6\ s$ (one page at a time) or $0.112\ s$ (54 pages at a time)
  - Total cost: $\approx 7.92\ s$ (one page at a time) or $2.03\ s$ (54 pages at a time)

Sort-merge join

- In practice, the cost of sort-merge join for an equi-join is usually linear to the relation sizes
  - assuming we have a large enough buffer for sorting everything in two passes
  - can even combine the merge phase of external sorting with the merge phase in sort-merge join (i.e., pipelining)
- Question: how large the tables can be in order to complete the sort-merge join in two passes? (minimal needed for sort-merge joins)
  - For simplicity, let B = 1
  - Let $N = \max(N_R, N_S)$, we need $\log_{M-1} \left\lceil \frac{N}{M} \right\rceil \leq 1$ => roughly $N \leq M^2 - M$
  - In other words, to perform a sort-merge join in two passes
    - the buffer size $M \geq 0.5 + \sqrt{N + 0.25} = O(\sqrt{N})$
      - good enough to use $\sqrt{N}$ + c for some small constant c in practice

    - Exercise: B > 1?

Hash join

- Idea: build a hash table on inner relation E over join attribute E
  - Scan the outer relation and probe the hash table

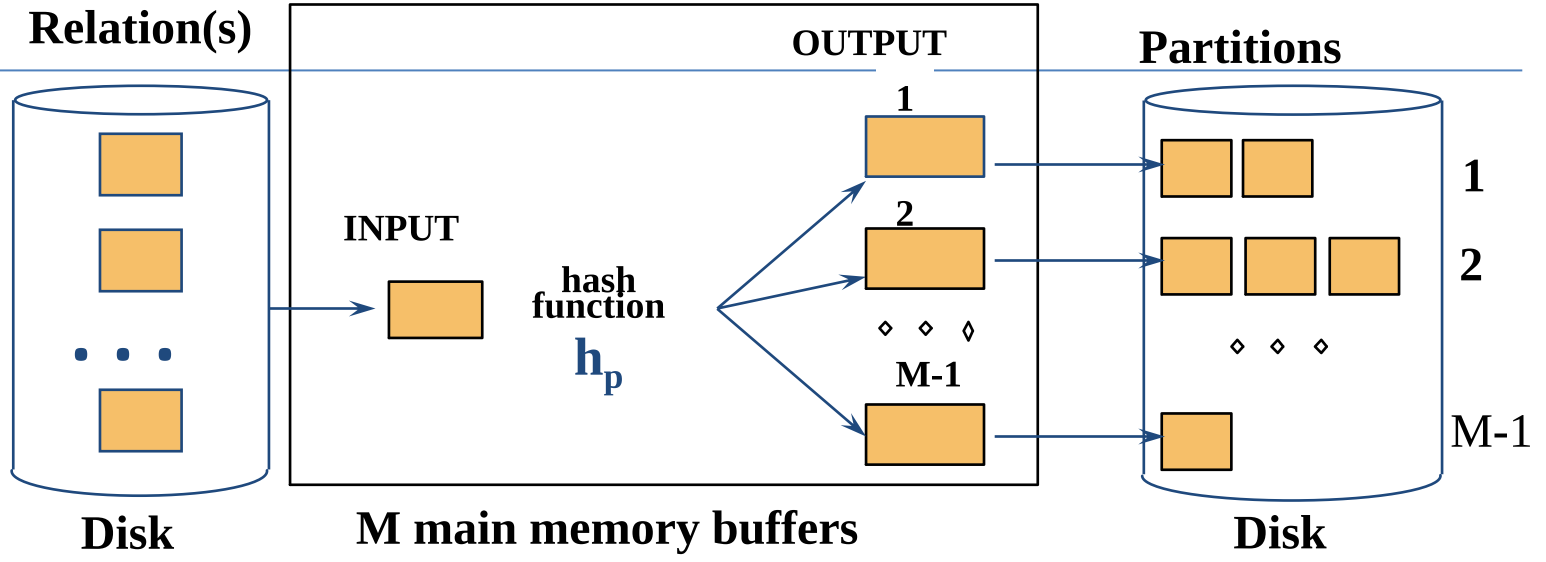
```
Build a hash table over E with hash function h_r
foreach tuple s in S do
   probe the hash table for all the matching e in E
   and emit the join results
```

  - However, the hash table might be too large to fit in memory.
  - Extendible hashing/linear hashing have overhead for dynamic updates
    - not suitable for QP purpose

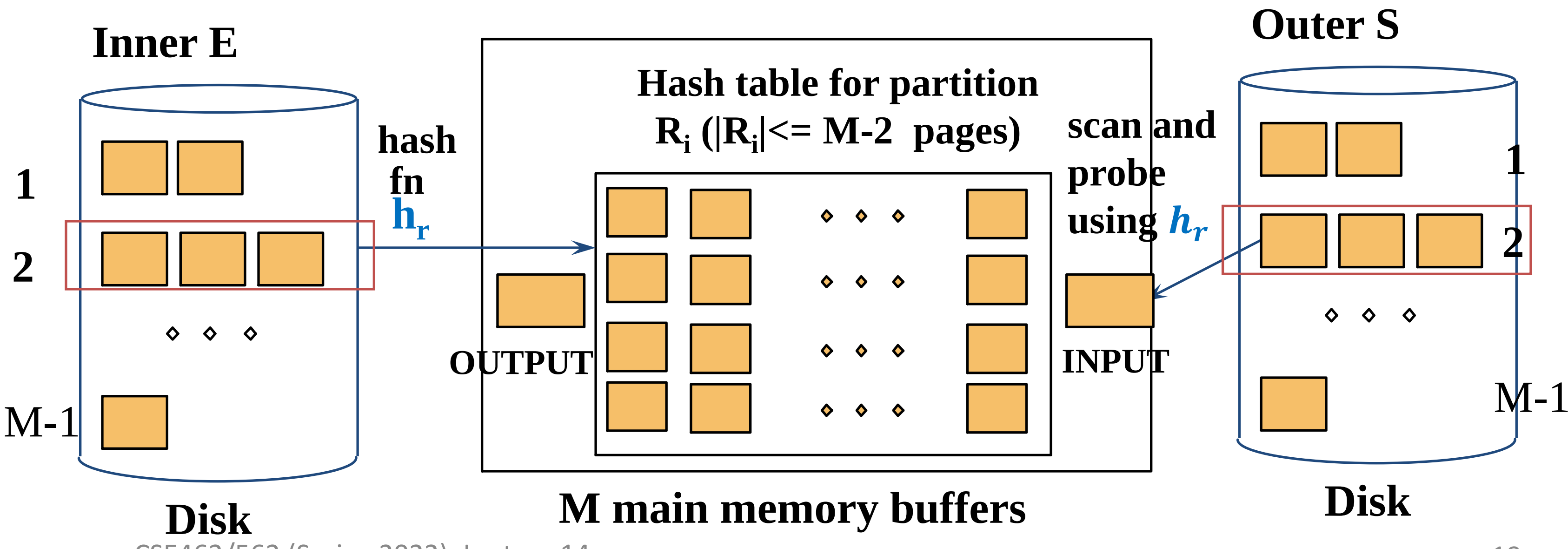
- Solution: partitioning using a hash function h_p

Hash join

- Two phases
  - Partitioning
    - Partitioning both outer S and inner E using the same hash function h_p



  - Rehashing and probing
    - load a partition for the inner E, rehash using a different hash function h_r and build a hash table
    - scan the partition of the outer S with the same hash value for h_p and probe the in-memory hash table

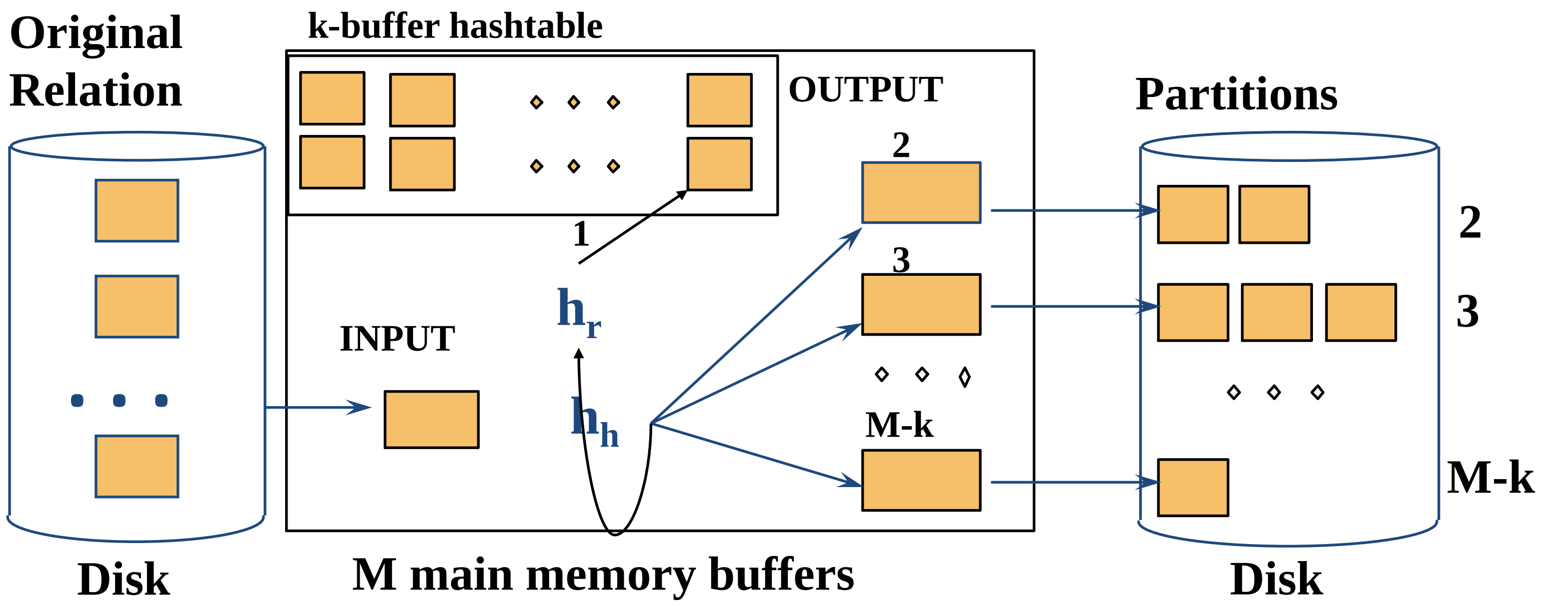


Hash join

- What if a partition won't fit into memory in the rehashing phase?
  - *Recursive partitioning!*
  - In the rehash and probe phase, if both partitions with the same hash value are larger than $M-2$
    - recursively partition them as if they were the original relations to be joined
    - use a different partitioning hash function h_p'
- Assuming there's no recursive partitioning
  - Cost of partitioning on R and E: $2t_T N_R + 2t_S N_R \; + \; 2t_T N_E + 2t_S N_E$
    - can also use larger blocks B to reduce the number of seeks to $\frac{2N_R}{B} + \frac{2N_E}{B}$
  - Cost of rehashing and probing: $t_T N_E + t_T N_R + 2t_S \left\lfloor \frac{M}{B} - 1 \right\rfloor$ => linear to relation sizes
  - total cost is roughly the cost of scanning both relations for three times
    - running example: $M = 100, B = 10$ => cost ≈ $1.72\ s$;
    - $M = 1000, B = 10 \Rightarrow cost \approx 13.2s$ (!)
- How big of a table can we hash in one pass? assuming B = 1
  - M - 1 partitions in Phase 1
  - Each should be no more than M page large
  - Answer: (M-2)(M-1) – assuming uniformity among the keys
    - i.e., we can do hash join in one pass in about $O\left(\sqrt{N_E}\right)$ space
  - Much like sorting, but only dependent on the *inner relation size (usually the smaller one)*
    - Do need to use $c\sqrt{N_E}$ in practice in case of key skews
    - Exercise: B > 1?

Hybrid Hashing

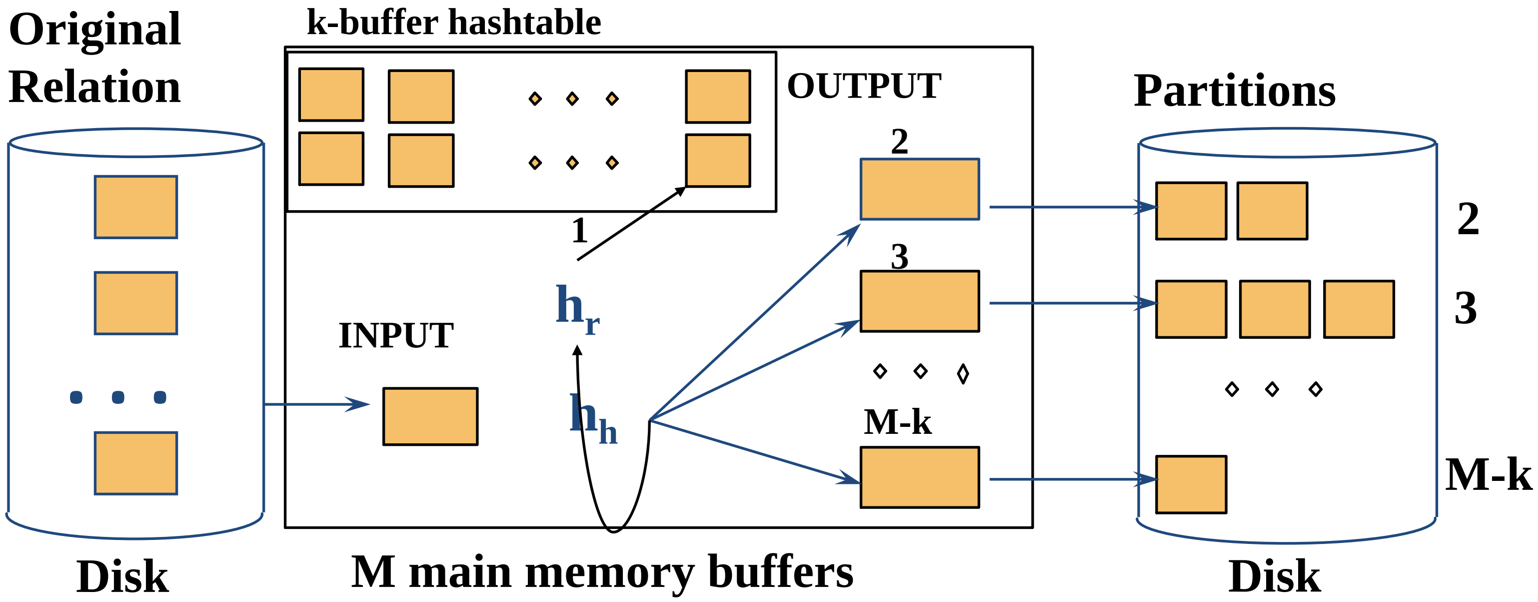
- Can we do it better when both relations fit in memory?
  - In-memory hash join can finish in 1 scan instead of 3!
- Hybrid hashing
  - Idea: keep a small 1st partition (of size k) in memory in the partitioning phase
  - directly scan and probe the keys in the 1st partition after partitioning of the inner relation finishes



Hybrid hashing

- Assume we have the hash-partition function $h_p: X \rightarrow [M-k-1]$ (X is the domain of the key, i.e., the join column)
- Define h_h as follows: (technically, it is determined by the sequence of the keys)
  - $h_h(x) = 1$ if in-memory hash table is not yet full
  - $h_h(x) = 1$ if x is already in the hash table
  - $h_h(x) = h_p(x) + 1$ otherwise
- This ensures that:
  - Bucket 1 fits in k pages of memory
  - If the entire set of distinct hash table entries is smaller than k, there is not *spilling!*

- During partitioning of the outer S
  - If $\mathrm{h_h}(s.sid) = 1$
    - probe the in-memory hash table and emit join results directly
  - Otherwise,
    - write s to its partition
- Only enter the rehashing and probing phase if there is any spill
- Running example
  - $\mathrm{M} = 1000, k = 900$
  - Cost = $2t_S + t_T(N_R + N_E) \approx 0.15s$



Hashing for single-table ops

- Recursive hashing and hybrid hashing can also be applied to aggregation and deduplication operators
  - Instead of rehashing and probing
  - We only rehash each partition and maintain aggregates/distinct values
  - Cost analysis is similar to hash joins

Summary

- This lecture
  - Join algorithms
    - Nested loop (simple/block/index)
    - Sort-merge join
    - Hash join

- Next lecture
  - Project 4 overview
    - To be released on Sunday, 4/10
  - Query optimization

  - HW4 (graded) will be released next Tuesday, 4/12
    - submission due in one week, 4/19
    - submit to UBLearns