COP 3540: Introduction to Database Structures

Fall 2017

Query Optimization III

M. Rathod

Query Processing

- **access paths** are alternative ways to retrieve tuples from a relation
- two kinds of access paths:

1. file scan
2. index plus a matching selection condition

- **selectivity** of an access path is the number of pages retrieved (index pages + data pages) if index access path is used to retrieve all desired tuples

- **most selective** access path is the one that retrieves the fewest pages

- using the most selective access path minimizes the cost of data retrieval

Query Processing

The following schema will be used:

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

rname has been added to reserves (reservation may be made by a person who is not a sailor)

Assumptions:

Reserves:

- a tuple is 40 bytes long
- a page can hold 100 tuples
- 1,000 pages

Sailors:

- a tuple is 50 bytes long
- a page can hold 80 tuples
- 500 pages

Query Processing

We will:

- consider only I/O costs
- measure I/O cost in terms of the # of page I/Os
- use big-O notation to express the complexity of an algorithm in terms of an input parameter (cost of a file scan is O(M), where M is the size of the file)

Query Processing

Selection Operation:

```
SELECT *
FROM   Reserves R
WHERE  R.rname='Joe'
```

σ *R.attr* ***op*** *value*(R)

1. No Index, Unsorted Data

- scan the entire relation
- check the condition on each tuple (*R.attr* ***op*** *value*)
- add the tuple to the result if the condition is satisfied
- cost is M 1/0s (M is # of pages) =1,000 I/Os (Reserves contains 1,000 pages)

expensive because it does not utilize the selection to reduce the number of tuples retrieved in any way!

Query Processing

Selection Operation:

2. No Index, Sorted Data (R is physically sorted on *R.attr*)

- sorted-file scan with selection condition $\sigma_{R.attr\ \mathbf{op}\ value}(R)$
- binary search to locate the first tuple that satisfies the selection condition (*R.attr1* > 5, and that R is sorted on *attr1* in ascending order)
- retrieve all tuples that satisfy the selection condition starting at this location
- cost of binary search is $O(\log_2 M)$
- cost of the scan to retrieve qualifying tuples vary from zero to M
- cost of binary search is $\log_2 1,000 \approx 10$ I/Os
-

Query Processing

Selection Operation:

3. B+ Tree Index

- cost of identifying the starting leaf page for the scan is typically 2 to 3 I/Os
- cost of scanning the leaf level page for qualifying data entries depends on the # of enteries
- cost of retrieving qualifying tuples from R depends on:
  1. # of qualifying tuples
  2. if index is clustered
     - clustered: cost is probably just one page I/O (all tuples are contained in a single page). estimating that roughly 10 percent of Reserves tuples are in the result, a clustered B+ tree index on the rname field of Reserves, would retrieve the qualifying tuples with 100 I/Os
     - unclustered: each index entry could point to a qualifying tuple on a different page resulting in 10,000 tuples, or 100 pages and cost would be 10,000 I/Os

Query Processing

Selection Operation:

4. Hash Index, Equality Selection

- cost includes 1 or 2 I/Os to retrieve the bucket page in the index + cost of retrieving qualifying tuples from R

- unclustered hash index on the *rname* attribute:
  - 10 buffer pages
  - 100 reservations made by people named Joe

- cost of retrieving the index page containing the rids is 1 or 2I/Os
- cost of retrieving the 100 Reserves tuples varies between 1 and 100
- If these 100 records are contained in 5 pages of Reserves = 5 I/Os (if rids are sorted by page component)

Query Processing

Projection Operation:

```
SELECT DISTINCT R.sid, R.bid
FROM     Reserves R
```

$\longrightarrow$ $\pi_{sid,bid} Reserves.$

To implement projection:

1. remove unwanted attributes (those not specified in the projection)
2. eliminate any duplicate tuples that are produced by
   - sorting algorithm or
   - hashing algorithm

Query Processing

Projection Operation:

1. Projection Based on Sorting

sorting algorithm has the following steps:

1. scan R and produce a set of tuples that contain only the desired attributes, cost = M I/Os (scan R) + T I/Os (write the temporary relation T is O(M)) so scan is 1,000 I/Os, assume T tuple is 10 bytes, cost of writing is 250 I/Os
2. sort this set of tuples using the combination of all its attributes as the key for sorting, cost = O(TlogT) (also O(MlogM)), if we have 20 buffer pages, we sort in two passes at a cost of $2 * 2 * 250 = 1,000$ I/Os
3. scan the sorted result, comparing adjacent tuples, and discard duplicates, cost = T (250 I/Os)

total cost is O(MlogM) = 2,500 I/Os

Query Processing

Projection Operation:

1. Projection Based on Sorting

improvements:

- project out unwanted attributes during the first pass (Pass 0) of sorting
- eliminate duplicates during the merging passes (fewer tuples are written out in each pass where most of the duplicates will be eliminated in the very first merging pass)

first pass: scan R cost =1,000 I/Os + write out 250 pages

with 20 buffer pages, the 250 pages are written out as 7 internally sorted runs each about 40 pages long

second pass: read the runs cost = 250 I/Os, and merge them

total cost = 1,500 I/Os

Query Processing

Projection Operation:

2. Projection Based on Hashing

good for large number of buffer pages (B) relative to the number of pages of R

has the following phases:

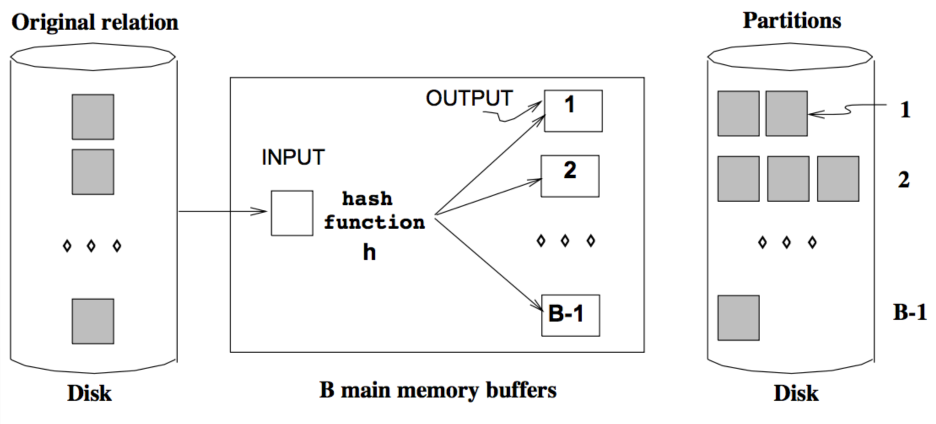
1. partitioning
2. duplicate elimination

Query Processing

Projection Operation:

2. Projection Based on Hashing

1. partitioning
   - one input buffer page and B – 1 output buffer pages
   - relation R is read into the input buffer page, one page at a time
   - project out unwanted attributes for each tuple
   - apply a hash function h to the combination of all remaining attributes
   - tuple is written to the output buffer page that it is hashed to by h
   - B – 1 partitions at the end
   - tuples that belong to different partitions are not duplicates because they have different hash values (duplicates are in the same partition)



Query Processing

Projection Operation:

2. Projection Based on Hashing

2. duplicate elimination

for each partition

1. - read in the partition one page at a time
   - hash each tuple by applying hash function h2 (≠ h!) to the combination of all fields
   - insert it into in-memory hash table
   - if new tuple hashes to the same value as existing tuple, compare the two to check whether the new tuple is a duplicate
   - discard duplicates as detected
2. -write the duplicate-free tuples in the hash table to the result file
   - clear the in-memory hash table to prepare for the next partition

- cost: read R and write T cost = M+ T I/Os, cost of hashing is a CPU cost (not considered), cost of phase 2 part 2 is M + 2T I/Os
- Total cost = $1,000 + 2 * 250 = 1,500$ I/Os

Query Processing

Join Operation

```
SELECT  *
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
```

→ $R \bowtie S.$

- join can be defined as a cross-product followed by selections and projections
- result of a cross-product is typically much larger than the result of a join
- recognize joins and implement them without materializing the underlying cross-product
- using two relations R and S, with the join condition $R_i = S_j$ we assume:
  - M pages in R with pR tuples per page
  - N pages in S with pS tuples per page

and we look at:

1. Sort-Merge Join
2. Block Nested Loops Join
3. Index Nested Loops Join
4. Sort-Merge Join
5. Hash Join

Query Processing

Join Operation

1. Sort-Merge Join

```
SELECT *
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid
```

foreach tuple $r \in R$ do
  foreach tuple $s \in S$ do
    if $r_i == s_j$ then add $\langle r, s \rangle$ to result

- cost of scanning R = M I/Os
- scan S a total of pR $*$ M times
- each scan costs N I/Os.
- total cost = M + pR $*$ M $*$ N
- M = 1,000, pR =100, and N is 500
- total cost of simple nested loops join = 1, 000 + 100 $*$ 1, 000 $*$ 500 page I/Os = 1, 000 + ($5 * 10^7$) I/Os (huge!)

Improvement - join page-at-a-time with an improvement of a factor of pR (total cost M + M $*$ N = 1, 000 + 1, 000 $*$ 500 = 501, 000 I/Os)

Query Processing

Join Operation

2. Block Nested Loops Join

```
SELECT  *
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
```

if we have enough memory to hold the smaller relation, say R, with at least two extra buffer pages left over

- each tuple $s \in S$, we check R and output a tuple $\langle r, s\rangle$ for qualifying tuples s ($r_i = s_j$), extra buffer is an output buffer
- each relation is scanned once, total I/O cost of M + N

Query Processing

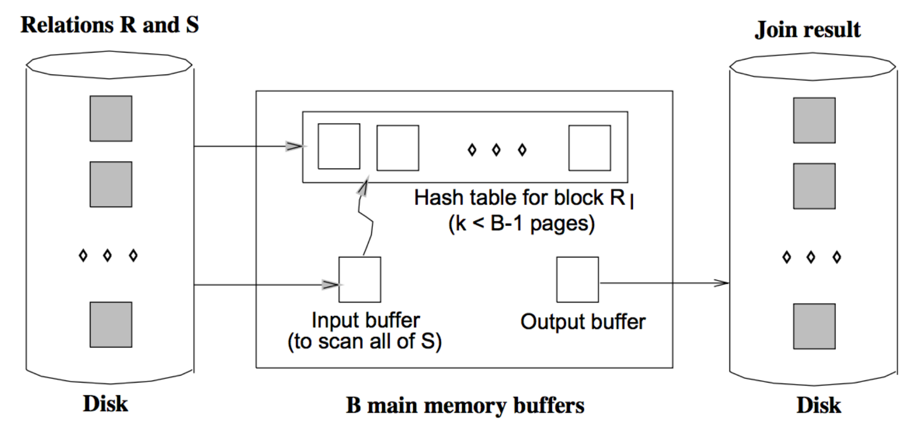
Join Operation

2. Block Nested Loops Join

if not enough memory:

- break R into blocks that can fit into the buffer pages and scan all of S for each block of R
- R is the outer relation (scanned once), S is the inner relation (scanned multiple times)
- B buffer pages, we can read in B – 2 pages of the outer relation R and scan the inner relation S using one of the two remaining pages
- write out tuples ⟨r, s⟩, where r ∈ R-block and s ∈ S-page and $r_i = s_j$, using the last buffer page for output
- a way to find matching pairs of tuples is to build a main-memory hash table for the block of R.

foreach block of $B-2$ pages of R **do**
  foreach page of S **do** {
    for all matching in-memory tuples $r \in R\text{-}block$ and $s \in S\text{-}page$,
    add $\langle r, s\rangle$ to result
  }



Query Processing

Join Operation

2. Block Nested Loops Join

```
SELECT  *
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
```

foreach block of $B-2$ pages of R **do**
  foreach page of S **do** {
    for all matching in-memory tuples $r \in R\text{-}block$ and $s \in S\text{-}page$,
    add $\langle r, s \rangle$ to result
  }

- cost = M I/Os for reading in R
- S is scanned a total of [M/B-2] times - ignoring the extra space required per page due to the in-memory hash table, costs = N I/Os
- total cost = M + N ∗ [M/B-2]

assume that we have enough buffers to hold an in-memory hash table for 100 pages of Reserve with at least two additional buffers:

- scan Reserves cost =1,000 I/Os
- for each 100-page block of Reserves, scan Sailors at 10 scans of Sailors, each costing 500 I/Os
- Total cost = 1, 000 + 10 ∗ 500 = 6, 000 I/Os
- if we have buffers to hold only 90 pages of Reserves, scan Sailors [1, 000/90] = 12 times, total cost = 1, 000 + 12 ∗ 500 = 7, 000 I/Os

Query Processing

Join Operation
2. Block Nested Loops Join

```
SELECT  *
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
```

foreach block of $B-2$ pages of R **do**
  foreach page of S **do** {
    for all matching in-memory tuples $r \in R\text{-}block$ and $s \in S\text{-}page$,
    add $\langle r, s \rangle$ to result
  }

- cost = M I/Os for reading in R
- S is scanned a total of [M/B-2] times - ignoring the extra space required per page due to the in-memory hash table, costs = N I/Os
- total cost = M + N ∗ [M/B-2]

if we choose Sailors to be the outer relation R:

- scan Sailors cost = 500 I/Os
- scan Reserves [500/100] = 5 times
- total cost is 500 + 5 ∗ 1, 000 = 5, 500 I/Os
- if we have buffers to hold only 90 pages of Sailors, scan Reserves [500/90] = 6 times, total cost = 500 + 6 ∗ 1, 000 = 6, 500 I/Os

Query Processing

Join Operation

3. Index Nested Loops Join

```
SELECT *
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid
```

foreach tuple $r \in R$ **do**
  foreach tuple $s \in S$ **where** $r_i == s_j$
    add $\langle r, s \rangle$ to result

- if there is an index on one of the relations on the join attribute(s), make the indexed relation be the inner relation
- index on S: for each tuple r ∈ R, use the index to retrieve matching tuples of S (in the same partition having the same value in the join column) cost depends on:
  1. index on S is a B+ tree index: cost to find leaf is typically 2 to 4 I/Os
     index on S is a hash index: cost to find bucket is 1 or 2 I/Os
  2. cost of retrieving matching S tuples depends on whether the index is clustered:
     clustered: cost per outer tuple r ∈ R is typically just one more I/O
     unclustered: cost could be one I/O per matching S-tuple (each can be on a different page)

Query Processing

Join Operation

3. Index Nested Loops Join

```
SELECT  *
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
```

foreach tuple $r \in R$ do
  foreach tuple $s \in S$ where $r_i == s_j$
    add $\langle r, s \rangle$ to result

- hash-based index on the sid attribute of Sailors takes about 1.2 I/Os on average to retrieve page of the index
- sid is a key for Sailors, so at most one matching tuple
- sid in Reserves is a foreign key - one matching Sailors tuple for each Reserves tuple
- cost of scanning Reserves is 1,000
- $100 * 1,000 = 100{,}000$ tuples in Reserves
- retrieve the Sailors page containing the qualifying tuple
- total cost = $100,000 * (1 + 1.2) = 221{,}000$ I/Os

Query Processing

Join Operation

3. Index Nested Loops Join

```
SELECT  *
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
```

foreach tuple $r \in R$ **do**
  foreach tuple $s \in S$ **where** $r_i == s_j$
    add $\langle r, s \rangle$ to result

- hash-based index on the sid attribute of Reserves
- scan Sailors - 500 I/Os
- 80 $*$ 500 = 40,000 Sailors tuples
- for each tuple, use the index to retrieve matching Reserves tuples (each tuple could match with either zero or more Reserves tuples) in 1.2 I/Os
- total cost = 500 + 40, 000 $*$ 1.2 = 48, 500 I/Os + cost of retrieving matching Reserves tuples

Query Processing

Join Operation

3. Index Nested Loops Join

```
SELECT *
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid
```

foreach tuple $r \in R$ **do**
  foreach tuple $s \in S$ **where** $r_i == s_j$
    add $\langle r, s\rangle$ to result

- total cost = 500 + 40, 000 $*$ 1.2 = 48, 500 I/Os + cost of retrieving matching Reserves tuples
- 100,000 reservations for 40,000 Sailors, each Sailors tuple matches with 2.5 Reserves tuples on average
- clustered index on Reserves: matching tuples are on the same page of Reserves, cost = 1 I/O per Sailor tuple = 40,000 extra I/Os and total cost = 48, 500 + 40, 000 = 88, 500 I/Os
- unclustered index on Reserves: Reserves tuple may be on a different page, cost = 2.5 $*$ 40, 000 I/Os = 100, 000 extra I/Os and total cost = 48, 500 + 100, 000 = 148, 500 I/Os

Query Processing

Join Operation

4. Sort-Merge Join

- sort both relations on the join attribute
- look for qualifying tuples r ∈ R and s ∈ S by merging the two relations
- sorting step groups all tuples with the same value in the join column (easy to identify partitions)
- compare R tuples in a partition with only the S tuples in the same partition (rather than with all S tuples)
- scan the relations R and S, look for tuples Tr in R and Ts in S such that $Tr_i = Ts_j$
- advance the scan of R as long as R tuple < current S tuple and advance the scan of S as long as the S tuple < current R tuple
- for each tuple r in the current R partition, scan all tuples s in the current S partition and output the joined tuple ⟨r,s⟩

FAU

Query Processing

Join Operation

4. Sort-Merge Join

Algorithm:

```
proc smjoin(R, S, 'R_i = S'_j)

if R not sorted on attribute i, sort it;
if S not sorted on attribute j, sort it;

Tr = first tuple in R;                                    // ranges over R
Ts = first tuple in S;                                    // ranges over S
Gs = first tuple in S;                       // start of current S-partition

while Tr ≠ eof and Gs ≠ eof do {

    while Tr_i < Gs_j do
        Tr = next tuple in R after Tr;                 // continue scan of R

    while Tr_i > Gs_j do
        Gs = next tuple in S after Gs                  // continue scan of S

    Ts = Gs;                                   // Needed in case Tr_i ≠ Gs_j
    while Tr_i == Gs_j do {                   // process current R partition
        Ts = Gs;                                  // reset S partition scan
        while Ts_j == Tr_i do {                     // process current R tuple
            add ⟨Tr, Ts⟩ to result;                  // output joined tuples
            Ts = next tuple in S after Ts;}      // advance S partition scan
        Tr = next tuple in R after Tr;                  // advance scan of R
        }                                  // done with current R partition

    Gs = Ts;                          // initialize search for next S partition

}
```

Query Processing

Join Operation

4. Sort-Merge Join

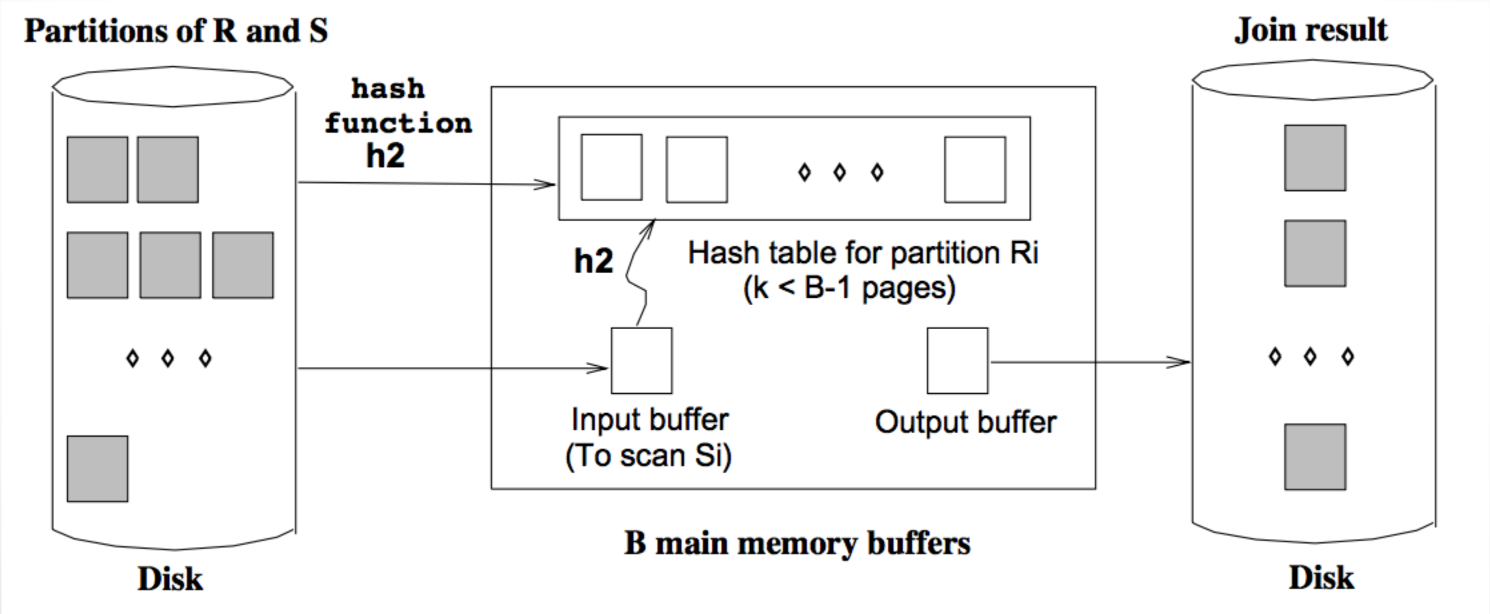
- cost of sorting R is O(M logM)
- cost of sorting S is O(N logN)
- cost of merging phase is M + N (if no S partition is scanned multiple times)
- assume that we have **100** buffer pages
- sort Reserves in 2 passes:
  - first pass produces 10 internally sorted runs of 100 pages each
  - second pass merges these 10 runs to produce the sorted relation
- read and write Reserves in each pass, sorting cost = $2 * 2 * 1000 = 4,000$ I/Os
- sort Sailors in two passes, at a cost of $2 * 2 * 500 = 2,000$ I/Os
- second phase of the sort-merge join algorithm requires an additional scan of both relations
- total cost is 4,000 + 2,000 + 1,000 + 500 = 7,500 I/Os

Query Processing

Join Operation

5. Hash Join

- **partitioning/building phase**: identify partitions in R and S
- **probing/matching phase**: compare tuples in R partition only with tuples in the corresponding S partition for testing equality join conditions
- hash both relations on the join attribute, using the same hash function h into k partitions
- R tuples in partition i can join only with S tuples in the same partition i



Query Processing

Join Operation

5. Hash Join

Algorithm:

```
// Partition R into k partitions
foreach tuple r ∈ R do
    read r and add it to buffer page h(r_i);          // flushed as page fills

// Partition S into k partitions
foreach tuple s ∈ S do
    read s and add it to buffer page h(s_j);          // flushed as page fills

// Probing Phase
for l = 1, ..., k do {

    // Build in-memory hash table for R_l, using h2
    foreach tuple r ∈ partition R_l do
        read r and insert into hash table using h2(r_i) ;

    // Scan S_l and probe for matching R_l tuples
    foreach tuple s ∈ partition S_l do {
        read s and probe table using h2(s_j);
        for matching R tuples r, output ⟨r, s⟩ };

    clear hash table to prepare for next partition;
    }
```

Query Processing

Join Operation

5. Hash Join

- partitioning phase: scan both R and S once and write them both out once, cost = 2(M + N)
- probing phase: scan each partition once, cost = M + N
- total cost = 3(M + N)
- assuming each partition fits into memory
- total cost is 3 ∗ (500 + 1, 000) = 4, 500 I/Os,

SUMMARY

- no index unsorted file - the only access path is a file scan
- no index but the file is sorted - a binary search to first tuple
- B+ tree index selectivity depends on if the index is clustered or unclustered and the # of result tuples
- hash indexes can be used only for equality selections
- projection operation - implemented by sorting and duplicate elimination during the sorting step
- nested loops join- join condition is evaluated between each pair of tuples from R and S
- block nested loops join performs pairing that minimizes the number of disk accesses
- index nested loops join fetches only matching tuples from S for each tuple of R by using an index
- sort-merge join sorts R and S on the join attributes using an external merge sort and performs pairing during the final merge step
- hash join first partitions R and S using a hash function on the join attributes, only partitions with the same hash values need to be joined in a subsequent step.