

# CS143: Query processing and join algorithms

## Book Chapters

- (4th) Chapter 13.1-6
- (5th) Chapter 13.1-6
- (6th) Chapter 12.1-6
- (7th) Chapter 15.1-6

## Things to Learn

- Join algorithms

## Motivation

Student(sid, name, addr, age, GPA)

Enroll(sid, dept, cnum, sec)

B+tree index on sid, age of Student table

- **Q:** How do we process `SELECT * FROM Student WHERE sid > 30`?
- **Q:** How do we process `SELECT * FROM Student WHERE sid > 30 AND age > 19`?
- **Q:** How do we process `SELECT * FROM Student S, Enroll E WHERE S.sid = E.sid`?
- Joins can be very expensive (maybe  $\approx |R| \times |S|$ ). How can we perform joins efficiently?

## Join algorithms

( $R$  and  $S$  example slide)

- **Q:** How to join  $R$  and  $S$ ? What is the simplest algorithm? What if we have an index? Any other ideas that we can use?
  - Four join algorithms
    - \* Nested-loop join
    - \* Index join
    - \* Sort-merge join
    - \* Hash join
  - We now learn how they work

### 1. Nested-Loop Join:

(nested-loop-join slide)

```
For each r in R do
  For each s in S do
    if r.C = s.C then output r,s pair
```

- **Q:** If  $R$  has 100,000 tuples, how many times the entire  $S$  table is scanned?
- The simplest algorithm. It works, but may not be efficient.

### 2. Index Join:

(index-join slide)

```
For each r in R do
  X <- index-lookup(S.C, r.C)
  For each s in X do
    output (r,s)
```

- Look up index to find matching tuples from  $S$ .
- **Q:** Benefit of index join compared to nested-loop join?

### 3. Sort-Merge Join:

(Sort-merge-join slide)

- Main idea: If tables have been sorted by the join attribute, we need to scan each table only once.
  - Maintain one cursor per table and move the cursor forward.
- Sort tables and join them.

(sort-merge algorithm slide)

```

(1) if R and S not sorted, sort them
(2) i <- 1; j <- 1;
    While (i <= |R|) AND (j <= |S|) do
        if R[i].C = S[j].C then outputTuples
        else if R[i].C > S[j].C then j <- j+1
        else if R[i].C < S[j].C then i <- i+1

Procedure outputTuples
    While (R[i].C = S[j].C) AND (i <= |R|) do
        k <- j;
        While (R[i].C = S[k].C) AND (k <= |S|) do
            output R[i], S[k] pair;
            k <- k + 1;
        i <- i + 1;

```

#### 4. Hash Join:

- Main idea: If hash values are different, the tuples will never join, i.e., if  $h(R.C) \neq h(S.C)$ , then  $R.C \neq S.C$ .
- Join two tuples only if their hash values are the same.

(hash-join algorithm slide)

##### (1) Hashing stage (bucketizing)

```

Hash R tuples into G1,...,Gk buckets
Hash S tuples into H1,...,Hk buckets

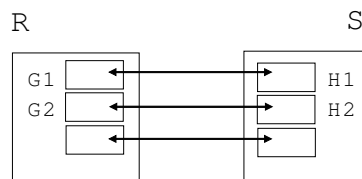
```

##### (2) Join stage

```

For i = 1 to k do
    match tuples in Gi, Hi buckets

```



## Comparison of Join Algorithms

- **Q:** Which algorithm is better?
- **Q:** What do we mean by “better”?

## Cost model

- The ultimate bottom-line:
  - How long does it take for each algorithm to finish for a particular data?
- Need of cost model
  - We need a “cost model” to estimate the performance of different algorithms
- Our cost model: Total number of disk blocks that have been read/written
  - Not very realistic
    - \* Ignore random, sequential IO issues, CPU cost, etc.
  - Yet simple to analyze and doable in class
    - \* More sophisticated models are too complex to analyze in class
  - Good approximation given that disk IOs dominate the cost
    - \* Most algorithms that we will study do mostly sequential scan
  - A better algorithm = smaller number of disk block access
  - Ignore the last IOs for result writing (the same for every algorithm)

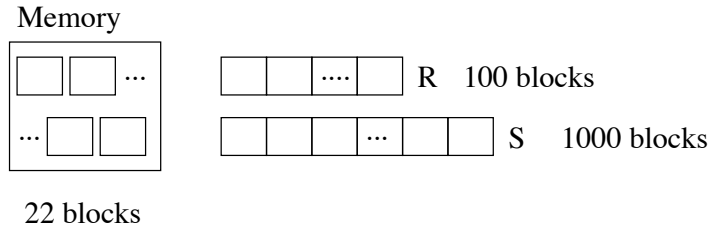
## Example to use

- Two tables  $R, S$
- $|R| = 1,000$  tuples,  $|S| = 10,000$  tuples, 10 tuples/block
- $b_R = 100$  blocks,  $b_S = 1,000$  blocks
- Memory buffer for 22 blocks

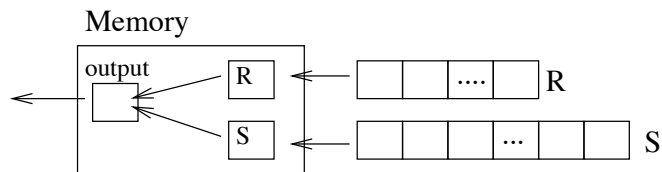
	Cost	Formula (if $b_R < b_S$ )
Nested Loop		
Sort Merge		
Hash		
Index		

## Cost of join stage of sort-merge join

- Usage of main memory blocks for join
  1. Available memory buffers. Disk blocks of each table

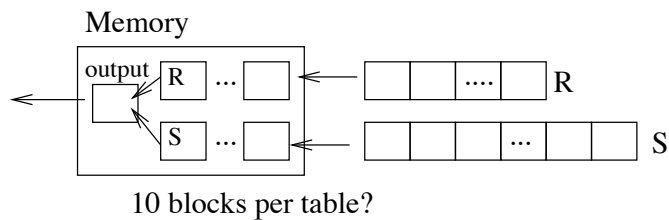


2. We need to read  $R$  table,  $S$  table and write the output.
  - Disk transfer unit is one block
  - At least one memory buffer block to read  $R$ , read  $S$  and write output.
  - Three memory blocks used for these tasks.



3. We sequentially read  $R$  and  $S$  blocks one block at a time, and join them (using the join algorithm)

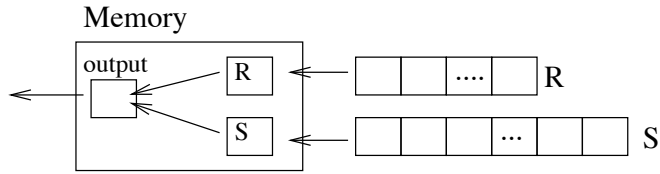
- **Q:** How many disk IOs (block reads/writes) for  $R$  and  $S$  during join stage?
- **Q:** Under our cost metric, can we make it more efficient by allocating more buffers for reading  $R$  and  $S$ ? For example,



## Nested-Loop Join

(naive nested-loop join algorithm slide for reminder)

(join diagram)



- **Q:** How many disk blocks are read?

- **Q:** Can we do any better?

### Optimization 1: Block-nested loop join

Once we read a block from  $R$ , join everything in the block in one scan of  $S$ .  
 → reduces the number of scans of  $S$  table

- **Q:** What is the cost?

- **Q:** Can we do any better?

### Optimization 2

Read as many blocks of  $R$  and join them together in one scan of  $S$   
 → reduces the number of scans of  $S$  table

- **Q:** What is the maximum # of blocks that we can read in one batch from  $R$ ?

- **Q:** What is the cost?
- **Q:** What is general cost for  $b_R$ ,  $b_S$  and  $M$ ?
- **Q:** What if we read  $S$  first? Would it be any different?

→ Use smaller table for the outer loop.

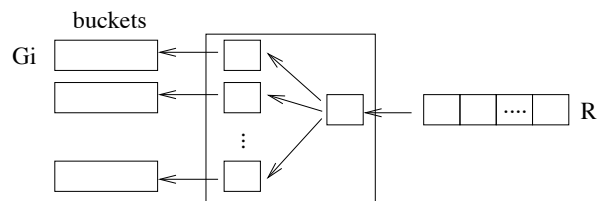
- **Summary**

- Always use block nested loop (not the naive algorithm)
- Read as many blocks as we can for the left table in one iteration
- Use the smaller table on the left (or outer loop)

## Hash Join

(hash join slide for reminder. two stages: hashing stage and join stage)

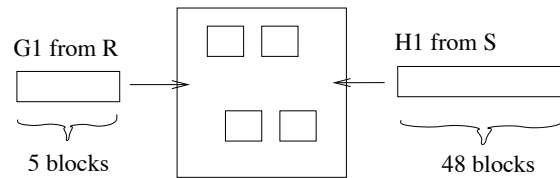
- Hashing stage: Read  $R$  (or  $S$ ) table and hash them into different buckets.



- **Q:** One block for reading  $R$ , other blocks for bucketizing. How many buckets?

- **Q:** Assuming random hashing, how many blocks per bucket?
  
- **Q:** During bucketizing,  $R$  table is read once and written once. How many disk IOs (read or write)?
  
- Repeat the same for  $S$

- Join stage: Join  $H_1$  with  $G_1$



- **Q:** 5 blocks for  $G_1$ , 48 blocks for  $H_1$ . How should we join  $G_1$  and  $H_1$ ?
  
- **Q:** How many disk IOs?
  
- **Q:** Total disk IOs?
  
- **Q:** What if  $R$  is large and  $G_1 > 20$ ?

### Recursive partitioning



\* # of bucketizing steps:  $\left\lceil \log_{M-1} \left( \frac{b_R}{M-2} \right) \right\rceil$

\* General hash join cost ( $b_R < b_S$ ):

$$2(b_R + b_S) \left\lceil \log_{M-1} \left( \frac{b_R}{M-2} \right) \right\rceil + (b_R + b_S)$$

## Index join

(index-join slide for reminder)

- **Q:** How many disk IOs?
- **Q:** What should the system do to perform index join?

Index join cost:

- IO for  $R$  scanning
- IO for index look up
- IO for tuple read from  $S$ .

- **Example 1**

- 15 blocks for index
  - \* 1 root, 14 leaf
- On average, 1 matching  $S$  tuples per an  $R$  tuple.

**Q:** How many disk IOs? How should we use memory?

**Q:** Any better way?

- **Example 2**

- 40 blocks for index
  - \* 1 root, 39 leaf
- On average, 10 matching tuples in  $S$ .

**Q:** How many disk IOs? How should we use memory?

- General cost:  $b_R + |R| \cdot (C + J)$

- $C$  average index look up cost
- $J$  matching tuples in  $S$  for every  $R$  tuple
- $|R|$  tuples in  $R$

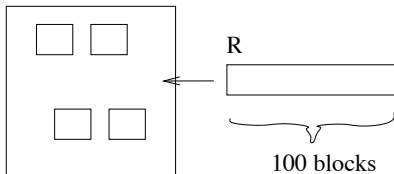
- **Q:** How can we compute  $J$ ?

- **Example:**  $R \bowtie_{R.C=S.C} S$ .  $|S| = 10$ ,  $V(C, R) = 1,000$ . Uniform distribution for  $C$  values. How many tuples in  $S$  with  $C = c$ ?

## Sort-Merge Join

- Two stage algorithm:
  1. Sort stage: Sort  $R$  and  $S$
  2. Merge stage: Merge sorted  $R$  and  $S$
- # of disk IOs during merge stage:  $b_R + b_S = 100 + 1,000 = 1,100$ .
- **Q:** How many disk IOs during sort stage?

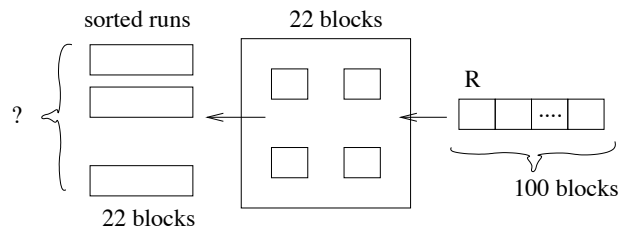
## Merge sort algorithm



- **Q:** How many blocks can we sort in main memory?

- **Q:** Do we need to allocate one block for output?

- **Q:** How many sorted runs after sorting  $R$  in chunk of 22 blocks?



- **Q:** What should we do with 5 sorted-runs?

- **Q:** How many disk IOs?

- **Q:** During first-stage sorting?

- **Q:** During second-stage merging?

Repeat it for  $S$  table of 1,000 blocks. Show that now we need three stages.

- In general, the number of passes for  $b_R$  and  $M$ :  $(\lceil \log_{M-1}(b_R/M) \rceil + 1)$ 
  - Verify it at home on your own.
  - Total # of IOs for sorting:  $2 \cdot b_R(\lceil \log_{M-1}(b_R/M) \rceil + 1)$

### Total sort-merge join cost

- In total:  $400 + 6,000 + 1,100 = 7,500$
- In general:  $2b_R(\lceil \log_{M-1}(b_R/M) \rceil + 1) + 2b_S(\lceil \log_{M-1}(b_S/M) \rceil + 1) + (b_R + b_S)$  IOs

### Summary of join algorithms

- Nested-loop join ok for “small” relations (relative to memory size)
- Hash join usually best for equi-join
  - if relations not sorted and no index
- Merge join for sorted relations
  - Sort merge join good for non-equi-join
- Consider index join if index exists
- To pick the best, DBMS maintains statistics on data

### High-level query optimization

Tables:  $R(A, B)$ ,  $S(B, C)$ ,  $T(C, D)$

- **Q:** How can we process the following query?  

```
SELECT * FROM R, S, T
WHERE R.B = S.B AND S.C = T.C AND R.A = 10 AND T.D < 30
```

  - Many different ways. (Show a couple of logical query trees)
- **Q:** For now, focus on  $R \bowtie S \bowtie T$ . How many different ways to execute it?

- In general, for  $n$  way joins,  $\frac{(2(n-1))!}{(n-1)!}$  ways.
  - Study why this is the case at home.
  - For  $n = 3$ ,  $4!/2! = 12$
  - For  $n = 5$ ,  $8!/4! = 1680$
  - For  $n = 10$ ,  $18!/9! = 17 \times 10^9$
- DBMS tries to pick the best based on statistics
  - In reality, picking the best is too difficult
    - \* For  $n = 10$ , it is clearly impossible to examine all 17 billion plans
  - DBMS tries to avoid “obvious mistakes” using a number of heuristics to examine only the ones that are likely to be reasonable
- Read the PDF file on database tuning and optimization
  - For 90% of the time, DBMS picks a good plan
  - To optimize the remaining 10%, companies pay big money to database consultants

## Statistics collection commands on DBMS

- DBMS has to collect statistics on tables/indexes for optimal performance
  - Without stats, DBMS does stupid things
- DB2
  - `RUNSTATS ON TABLE <userid>.<table> AND INDEXES ALL`
- Oracle
  - `ANALYZE TABLE <table> COMPUTE STATISTICS`
  - `ANALYZE TABLE <table> ESTIMATE STATISTICS` (cheaper than `COMPUTE`)
- Run the command after major update/index construction
- Does not matter for MySQL. No optimization based on actual data. Only rule-based optimizer.