



# CS 186 Exam Prep Section 5

## Parallel Query Processing, DB Design

### Agenda

- I. 5:10-5:30 -- Mini-lecture
- II. 5:30-6:00 -- Go over answers
- III. 6:00-7:00 -- Conceptual OH

Worksheet is available in @566 on Piazza.

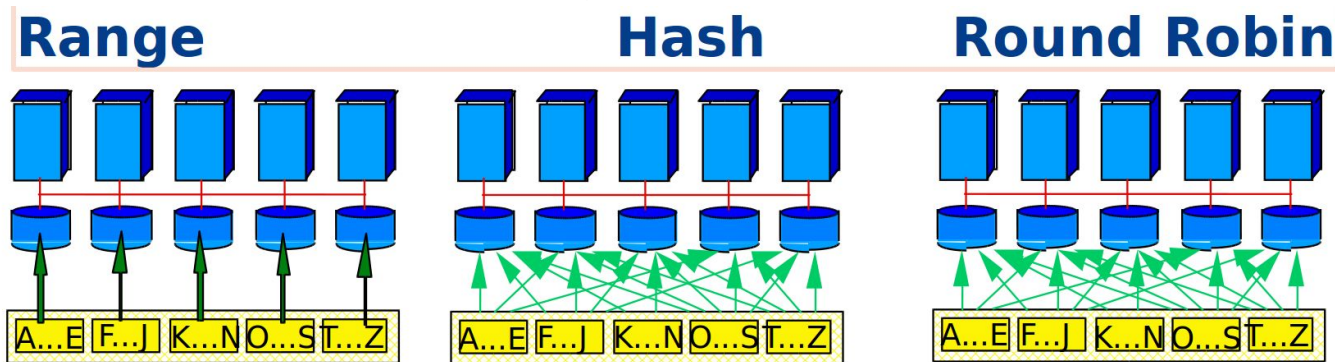
Feedback Form: <https://tinyurl.com/CS186ExamPrep5FA20>

---

# Parallel Query Processing

# Parallel Query Processing - Partitioning data

- **range partitioning** on a key divides data based on which range the key belongs to
- **hash partitioning** divides data based on a hash function
- **round robin partitioning** just cycles through the partitions in order as data come in (*not ordered based on a key*)



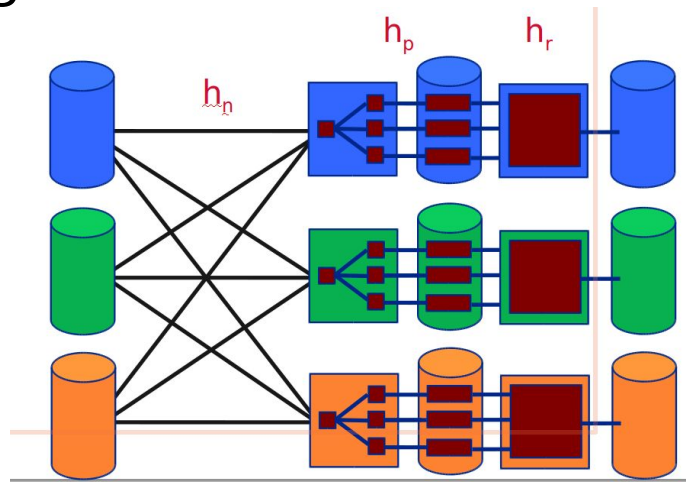
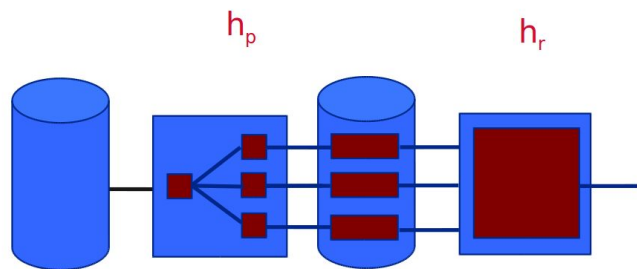
# Parallel Query Processing Notes



- Types of partitioning matter a lot. (Not an exhaustive list of pro/cons):
  - **Round-robin partitioning**
    - Strength: full scans - each machine gets the same number of records to scan (equal load)
    - Weakness: Range queries - require looking through all machines because of *uneven distribution of values*.
  - **Range partitioning**
    - Strength: Range queries - we know exactly which disks hold the values we're looking for
    - Weakness: Very susceptible to key skew (all records have the same value we're partitioning on) potentially making all records land on the same machine
  - **Hash partitioning**
    - Strength: Single value lookup - only one machine must be searched
    - Weakness: Scans on fields that the data wasn't originally hash partitioned on

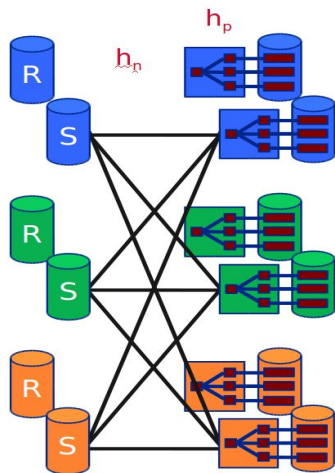
# Parallel Hashing

- Use a hash function to partition the data over all the machines (hash partitioning), then run external hashing on each machine independently
  - Similar to recursive partitioning

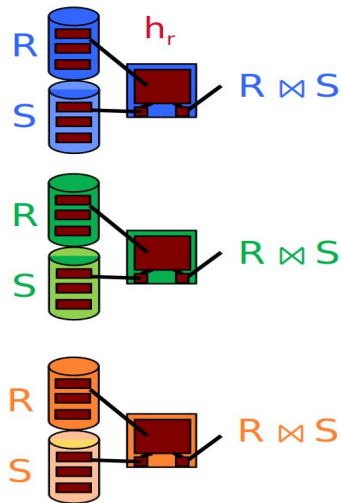


# Parallel Hash Joins

- Use hash partitioning on both relations, then perform a normal hash join on each machine independently



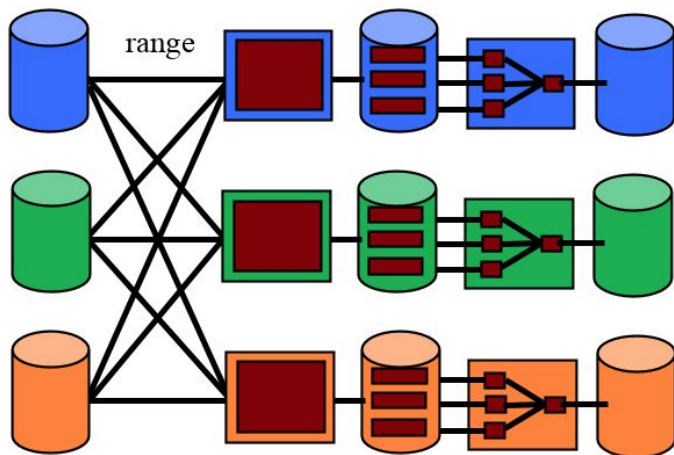
Phase 1: Partition R and S across different machines



Phase 2: Within each machine, perform local (grace) hash joins

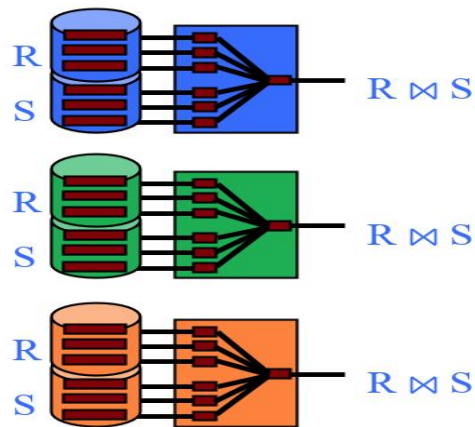
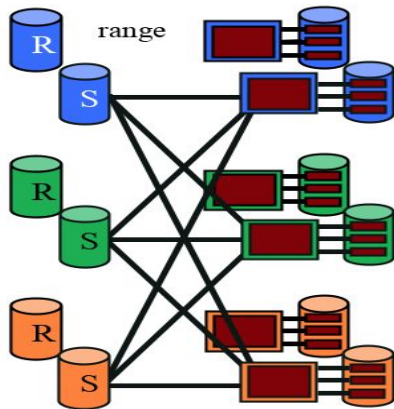
# Parallel Sorting

- Partition the data over machines with *range partitioning*
- Perform external sorting on each machine independently (each machine holds a different range of data)



# Parallel Sort Merge Join

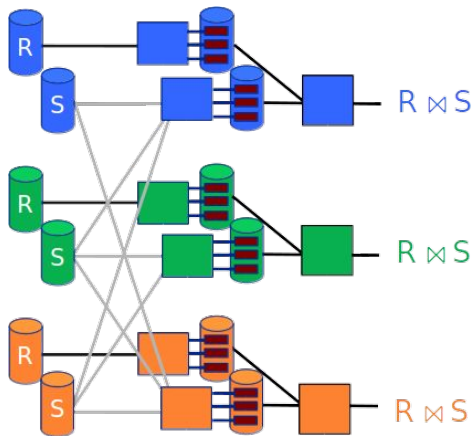
- Partition the data for both relations over machines with *range partitioning*
  - Use the same ranges for both relations
- Perform sort merge join on each machine independently





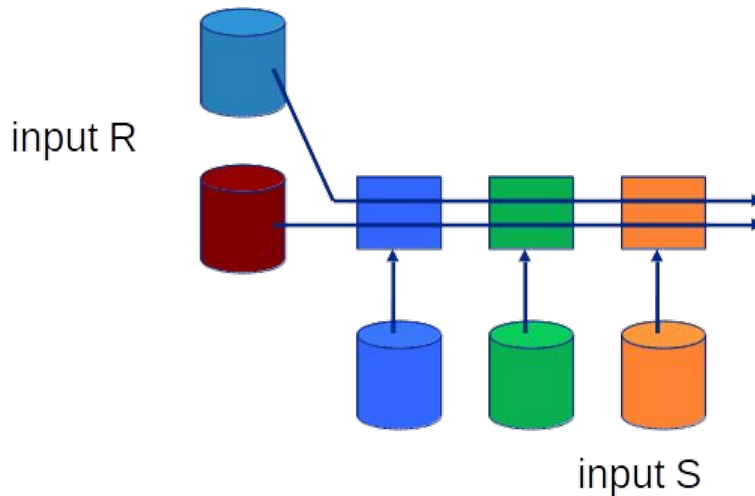
# Asymmetric Shuffles

- Sometimes, data is already partitioned the way we want
  - May already be hash partitioned on a key, or range partitioned on a key
  - Don't need to send anything across the network



# Broadcast joins

- Sometimes, one table is tiny and one is huge
  - May be much cheaper to send all of the tiny table multiple times than partition the huge table



# Parallel Query Processing Extra Things to Note



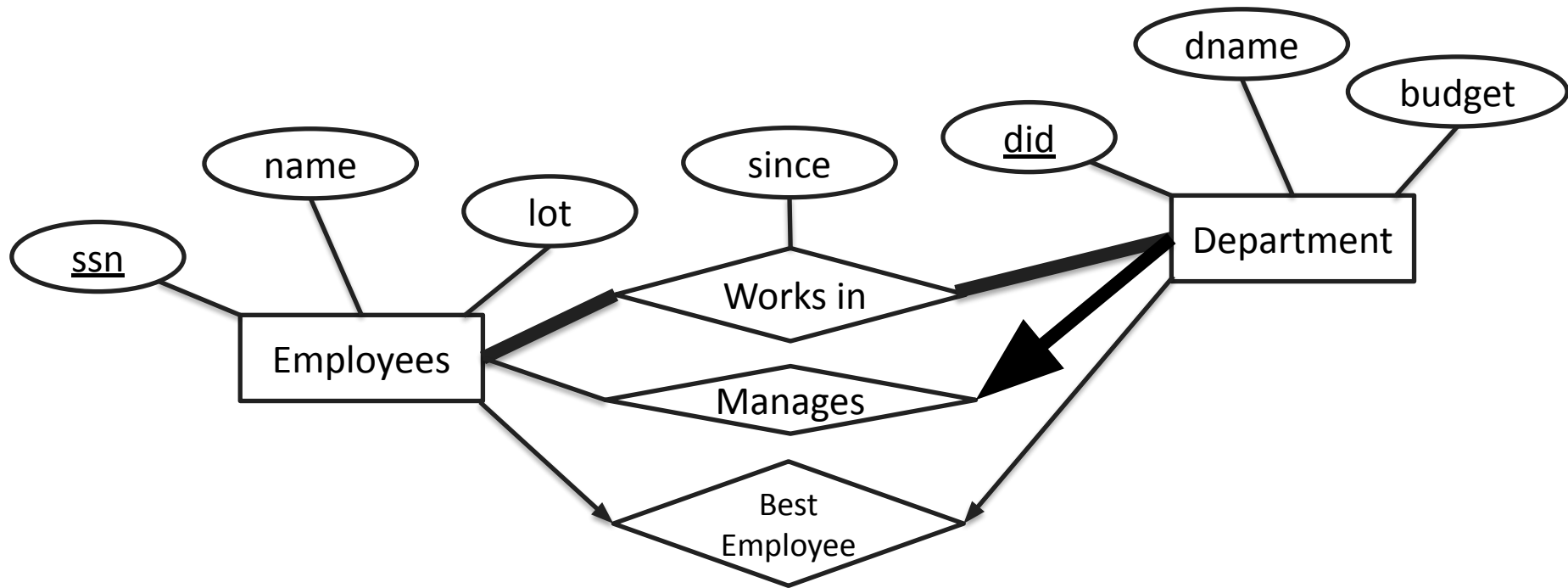
- We want to calculate the **network cost** in terms of time, or amount of data sent between machines. Less important is the number of I/Os on a given machine.
- Machines may have to receive data from other machines to start processing data if a table is sorted on only a single machine for example.
- Since we have multiple machines to use, we now care about bottlenecks.
  - Uneven number of records on each machine causes the total time spent doing operations (scanning, sorting, etc.) to be the maximum time spent of each individual machine (Machine 1 takes 500ms and Machine 2 takes 300ms, then our overall parallel query takes 500ms)

---





# Database Design

# ER Diagrams: Entities

- **Entities** are real-world objects described with **attributes**
- An **entity set** is a collection of the same type of entities
  - All entities in an entity set have the same attributes
  - All entities set have a **key** (underlined)
  - Box around entity set, ellipse around attributes
- A **relationship** is an association between 2+ entities
  - May be further described with attributes
- A **relationship set** is a collection of the same type of relationships (represented with a diamond)
  - **n-ary relationship set** = association of n entities



# ER Diagrams: Constraints

- **Key constraint** 
  - at most one (ex. every department can have at most one best employee)
- **Participation constraint** 
  - at least one (ex. every employee must work in a department )
- **Key constraint with total participation** 
  - exactly one (ex. a department can only be managed by one manager)
- **Non-key partial participation** 
  - 0 or more (no restrictions) (ex. an employee does not have to manage a department)

Note: **total participation** means everyone must participate, **optional participation** means that not everyone has to participate

# ER Diagrams: Relationships

- **Many-to-many**

- An employee may work in many departments, a department may have many employees
- Usually denoted by lines (non-key participation) on both sides

- **1-to-many/many-to-1**

- 1-to-many: An employee may manage many depts
- many-to-1: A dept may be managed by at most 1 employee
- Order depends on what you list first
- Usually denoted by line on one side and arrow (key participation) on the other

- **1-to-1**

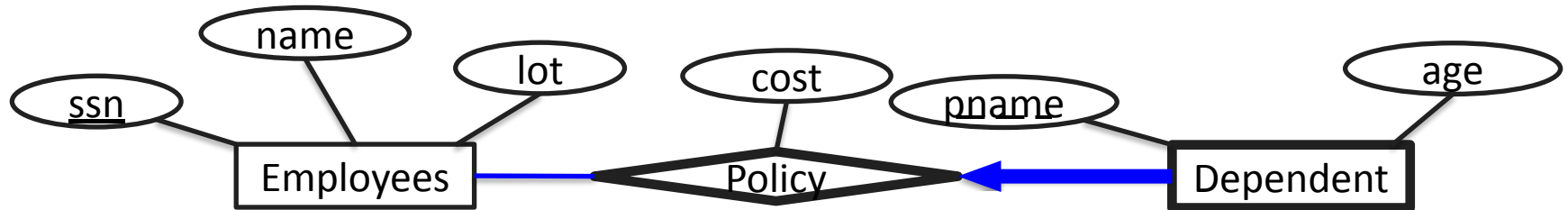
- A department can have at most one best employee, an employee can be the best employee in at most one department
- Usually denoted by arrows on both sides

- These don't denote necessity of participation (not everyone necessarily has to participate)



# ER Diagrams: Weak Entities

- A **weak entity** is an entity that can be identified uniquely *only* with the key of *another* entity (**owner entity**)
  - Weak entities have a **partial key** (dashed underline), which identifies the entity when combined with owner entity's key
  - Must be a one-to-many relationship (1 owner entity, many weak entities)
  - Weak entity must have total participation in relationship



# Redundancy and Anomalies

- Potential problems with relations
  - **Redundancy**: repeated sets of dependent values
  - Anomalies that can result from redundancy
    - Ex. Rating determines Wage, so **Wage depends on Rating**.
    - **Update anomaly**: if we change wage for one person, we have to change it for everyone
    - **Insert anomaly**: if we want to insert a person with rating 10, we have to figure out the wage associated with it
    - **Delete anomaly**: if we delete all employees with rating 8, we no longer know the wage value corresponding to rating 8 (what if we add a rating 8 person later?)

# Functional Dependencies

- **functional dependency**:  $\mathbf{X} \rightarrow \mathbf{Y}$  (*X determines Y*)
  - X, Y are sets of attributes
  - For every tuple in R, if attributes in X match, then attributes in Y must match
  - **Can *not* be inferred from the data: must come from outside of the data itself**
- **superkey**: X is a superkey of R if  $X \rightarrow [\text{all attributes of R}]$
- **candidate key**: the minimal superkey (smallest subset of attributes that is a superkey is itself; not necessarily smallest of all superkeys ever, but cannot be reduced further)

# Functional Dependencies: Inference Rules

- Armstrong's Axioms
  - **Reflexivity**: If  $Y \subseteq X$ , then  $X \rightarrow Y$
  - **Augmentation**: If  $X \rightarrow Y$ , then  $XZ \rightarrow YZ$ 
    - $XZ \rightarrow YZ$  does NOT imply  $X \rightarrow Y$
  - **Transitivity**: If  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$
- Union: If  $X \rightarrow Y$  and  $X \rightarrow Z$ , then  $X \rightarrow YZ$
- Decomposition: If  $X \rightarrow YZ$ , then  $X \rightarrow Y$  and  $X \rightarrow Z$ 
  - $XZ \rightarrow Y$  does NOT imply  $X \rightarrow Y$  and  $Z \rightarrow Y$

# Functional Dependencies: Closure

- The **closure** of a set of FDs  $F$  is  $F^+$ , the set of all FDs implied by  $F$ 
  - hard to find, exponential in # of attributes, so we use attribute closure instead
- The **attribute closure** of an attribute  $X$  given a set of FDs is  $X^+$ 
  - set of all attributes  $A$  such that  $X \rightarrow A$  is in  $F^+$  (all attributes that can be determined by just  $X$ )
  - Algorithm:
    - Closure =  $X$ ;
    - Repeat until there is no change
      - If there is an FD  $U \rightarrow V$  in  $F$  s.t.  $U \subseteq \text{closure}$ ,
        - set closure = closure  $\cup V$

# Normalization: Boyce-Codd Normal Form (BCNF)

- R is in BCNF if:
  - for every FD  $X \rightarrow A$  that holds over R, either  $A \subseteq X$  or X is a superkey of R
  - $A \subseteq X$  means the FD is trivial.
- Redundancies removed in BCNF
  - every field of every tuple contains some information that *cannot* be inferred from the FDs
- Simpler to deal with than other normal forms

# Normalization: BCNF Decomposition

- **We can decompose a relation  $R$  that is not in BCNF into multiple relations that are in BCNF**
- Algorithm:
  - Find FD  $X \rightarrow Y$  that violates BCNF
  - Decompose  $R$  into  $(R - X^+) \cup X$  and  $X^+$
  - Repeat until no FDs violate BCNF
- Heuristic: for the violating FD, make  $Y$  as big as possible (i.e. replace with  $X^+$ ; helps avoid unnecessarily fine-grained decomposition)
- What relations you get depends on what order you go in
- two attribute relations are always in BCNF

# Normalization: Lossiness

- **Lossiness**: we may not be able to reconstruct the original relation (doesn't actually lose data, it generates bad data)
  - BCNF is lossless (can still reconstruct original relation)
  - Decompose R into X and Y. Decomposition is lossless iff  $F^+$  contains:
    - $X \text{ INTERSECT } Y \rightarrow X$  or
    - $X \text{ INTERSECT } Y \rightarrow Y$
  - Alternatively, can attempt natural join between the two and manually check if the reconstruction works



# Normalization: Dependency Preservation

- **Dependency preservation**: if we can enforce  $F^+$  individually on each table; this in turn enforces the FDs on the entire database
  - BCNF is not necessarily dependency preserving (enforce FDs on each decomposed relation independently)
  - Formalism: dependency preserving iff  $F_x^+ \mathbf{U} F_y^+ = F^+$  where  $F_x$  are the FDs we can enforce just in relation X
    - For example: imagine we decomposed  $R = ABC$  into  $X=AB$ ,  $Y=BC$ . If we have an FD  $A \rightarrow C$  this is not dependency preserving because we can't enforce the dependency on either relation

---

# Worksheet

# Question 1

---



## Question 1.1

A query with a selection, followed by a projection, followed by a join, runs on a single machine with one thread.

**Answer:** No parallelism. It might look like a pipeline, but at any given point in time there is only one thing happening, since there is only one thread.

Is the query an example of:

- Inter-query parallelism,
- Intra-query, Inter-operator parallelism,
- Intra-query, Intra-operator parallelism
- No parallelism



## Question 1.2

Same as before, but there is a second machine and a second query, running independently of the first machine and the first query.

**Answer: Inter-query parallelism**

Is the query an example of:

- Inter-query parallelism,
- Intra-query, Inter-operator parallelism,
- Intra-query, Intra-operator parallelism
- No parallelism



## Question 1.3

A query with a selection, followed by a projection, runs on a single machine with multiple threads; one thread is given to the selection and one thread is given to the projection.

**Answer: Intra-query, inter-operator parallelism.**

Is the query an example of:

- Inter-query parallelism,
- Intra-query, Inter-operator parallelism,
- Intra-query, Intra-operator parallelism
- No parallelism



## Question 1.4

We have a single machine, and it runs recursive hash partitioning (for external hashing) with one thread.

**Answer: No parallelism, because there is only one machine and one thread. Don't confuse this with parallel hashing!**

Is the query an example of:

- Inter-query parallelism,
- Intra-query, Inter-operator parallelism,
- Intra-query, Intra-operator parallelism
- No parallelism



## Question 1.5

We have a multi-machine database, and we are running a join over it. For the join, we are running parallel sort-merge join.

**Answer: Intra-query, intra-operator parallelism.**  
We have a single query and a single operator, but that single operator is going to do multiple things at the same time (across different machines).

Is the query an example of:

- Inter-query parallelism,
- Intra-query, Inter-operator parallelism,
- Intra-query, Intra-operator parallelism
- No parallelism



# Question 2

---



## Question 2.1

Suppose we have a table of size 50,000 KB, and our database has 10 machines. Each machine has 100 pages of buffer, and a page is 4 KB.

We would like to perform parallel sorting on this table, so first, we perfectly range partition the data. Then on each machine, we run standard external sorting.

How many passes does this external sort on each machine take?

Answer: 2 passes

After range partitioning, each table will have 5,000 KB of data, or 1,250 pages. With 100 pages of buffer, this will take 2 passes to sort.



## Question 2.2

Suppose we were doing parallel hash join. The first step is to partition the data across the machines, and we usually use hash partitioning to do this.

Would range partitioning also work? What about round-robin partitioning?

**Answer:** Range partitioning also works, because items with the same key still end up on the same machine as required. Round-robin partitioning does not do that, so it does not work.



## Question 2.3

Suppose we have a table of 1200 rows, perfectly range-partitioned across 3 machines in order.

We just bought a 4th machine for our database, and we want to run parallel sorting using all 4 machines. The first step in parallel sorting is to repartition the data across all 4 machines, using range partitioning. (The new machine will get the last range. For each of the first 3 machines, how many rows will it send across the network during the repartitioning? (You can assume the new ranges are also perfectly uniform.)

100 rows, 200 rows, and 300 rows. The original partitions were 3 ranges of 400 rows each; the new 4 ranges will have 300 rows each. The first machine held the first 400 rows originally, and now only needs to hold the first 300. It will send the remaining 100 rows over the network (to machine 2). The second machine held rows 401-800 initially, but now needs to hold rows 301-600. It will send rows 601-800 (200 rows) to machine 3. And so on...

# Question 3

---



## Question 3.1

Suppose we have 4 machines, each with 10 buffer pages. Machine 1 has a **Students** table which consists of 100 pages. Each page is 1 KB, and it takes 1 second to send 1 KB of data across the network to another machine.

How long would it take to send the data over the network after we uniformly range partition the 100 pages? Assume that we can send data to multiple machines at the same time.

**Answer: 25 seconds**

After we uniformly partition our data, Machine 1 will send 25 pages to Machines 2, 3, and 4. It will take 25 seconds to finish sending these pages to each machine if we send the pages to each machine at the same time.



## Question 3.2

Next, imagine that there is another table, **Classes**, which is 10 pages. Using just one machine, how long would a BNLJ take if each disk access (read or write) takes 0.5 seconds?

**Answer: 105 seconds**

BNLJ will require  $10 + \text{ceil}(10/8) * 100 = 210$  I/Os, which will take 105 seconds.



## Question 3.3

Now assume that the **Students** table has already been uniformly range partitioned across the four machines, but **Classes** is only on Machine 1. How long would a broadcast join take if we perform BNLJ on each machine? Do not worry about the cost of combining the output of the machines.

**Answer: 40 seconds**

First, we must broadcast the **Classes** table to each machine, which will take 10 seconds (since we send the table to each machine at the same time). Next, we will perform BNLJ on each machine, which requires  $10 + \text{ceil}(10/8) * 25 = 60$  I/Os, or 30 seconds. In total, the time required to send the data over the network and perform the join will be 40 seconds.





## Question 3.4

Which algorithm performs better?

Broadcasting the **Classes** table and performing a parallel BNLJ runs faster than just using one machine, even with the additional time required to send the table over the network.



## Question 3.5

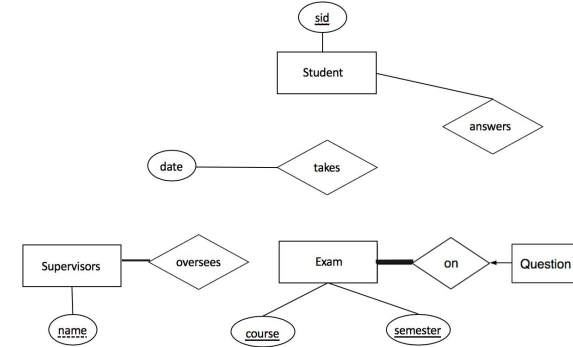
Knowing that the **Students** table was range partitioned, how can we improve the performance of the join even further?

Since we know the **Students** table was range partitioned, we can also range partition the **Classes** table on the same column and only send the corresponding partitions to each machine. This should lower the network cost and decrease the number of disk I/Os.

# Question 4

---

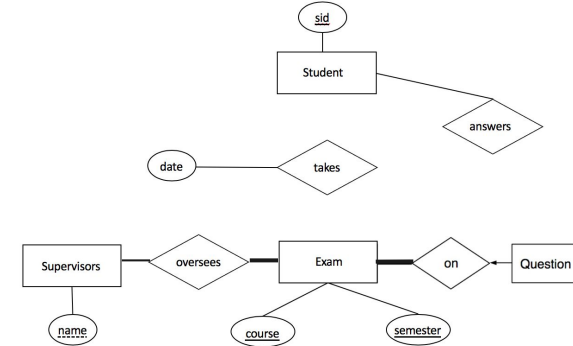
# Question 1



What type of edge should be drawn between the **Supervisors** entity and the **oversees** relationship set?

**Bold Arrow:** Because each supervisor must oversee exactly one exam, the supervisor must have a key constraint (hence the arrow) and a participation constraint (hence the bold) on their relationship with **oversees**.

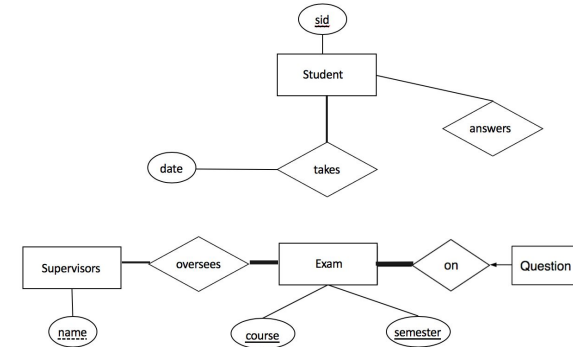
## Question 2



What type of edge should be drawn between the **Exam** entity and the **oversees** relationship set?

**Bold Line:** Because at least one supervisor oversees the exam, there is a participation constraint between the **oversees** relationship set and the **Exam**

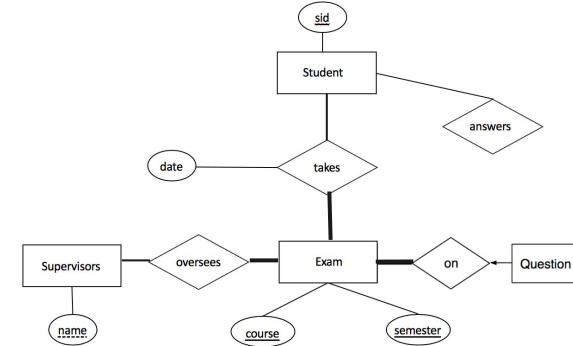
## Question 3



What type of edge should be drawn between the **Student** entity and the **takes** relationship set?

**Thin line:** Each student may take 0 or more exams on a given day (no constraints)

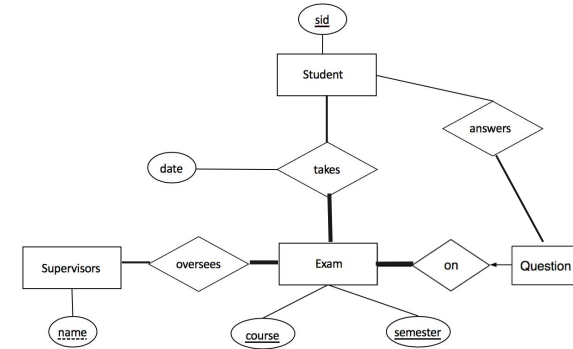
## Question 4



What type of edge should be drawn between the **Exam** entity and the **takes** relationship set?

**Bold line:** A student takes at least one exam in total, so we have a participation constraint. (If an entry appears in the **takes** relationship set, then there must be an **Exam** that is taken)

## Question 5



What type of edge should be drawn between the **Questions** entity and the **answers** relationship set?

**Thin line:** Each **student** may answer 0 or more questions (no constraints)





## Question 6

Consider the attribute set  $R = ABCDEF$  and the functional dependency set  $F = \{BE \rightarrow C, B \rightarrow F, D \rightarrow F, AEF \rightarrow B, A \rightarrow E\}$ . Which of the following are candidate keys of  $R$ ?

Compute the attribute closure of each set (using the FDs to add attributes to the closure until nothing more can be added). If the attribute closure consists of *all* attributes of a table, then the original set is a superkey. For it to be a **candidate** key, we need to ensure that none of its subsets are superkeys for the table (i.e. it contains no redundant attributes).

ACD:  $ACD \rightarrow ACDEF \rightarrow ABCDEF$ .  $ACD$  is a superkey, but is it a candidate key?

AD:  $AD \rightarrow ADEF \rightarrow ABDEF \rightarrow ABCDEF$ .  $AD$  is a superkey, and it is a subset of  $ACD$ , so  $AD$  is a candidate key and  $ACD$  is **not**

FC:  $FC \rightarrow FC$ . Not a superkey, thus not a candidate key

BF:  $BF \rightarrow BF$ . Not a superkey, thus not a candidate key



## Question 7

Given attribute set  $R = ABCDEFGH$  and FDs  $F = \{CE \rightarrow GH, F \rightarrow G, B \rightarrow CEF, H \rightarrow G\}$ , what relations are included in the final decomposition when decomposing  $R$  into BCNF in the order of FD set  $F$ ?

Start with  $CE \rightarrow GH$ . Violates BCNF! Decompose into  $ABCDEF$   $CEGH$

Next is  $F \rightarrow G$ . No relation contains  $FG$ , so skip

Next is  $B \rightarrow CEF$ . Violates BCNF! Decompose into  $ABD$   $BCEF$   $CEGH$

Next is  $H \rightarrow G$ . Violates BCNF! Decompose into  $ABD$   $BCEF$   $CEH$   $GH$ . These are the final relations



## Question 8

True or False: The decomposition of attribute set  $R=ABCDEF$  into  $ABDE \bowtie BCDF$  is lossless, given the FDs  $F = \{B \rightarrow D, E \rightarrow F, D \rightarrow E, D \rightarrow B, F \rightarrow BD\}$

**False! This is lossy.** Check the 3 conditions:

- 1) Union of attributes of tables must include all attributes of original table  
 $ABDE \cup BCDF = ABCDEF$  (passes)
- 2) Intersection must be non-empty ( $ABDE \cap BCDF \neq \text{NULL}$ ) (passes)
- 3) Common attributes to two relations must imply either of the relations. Our intersection is  $ABDE \cap BCDF = BD$ . The attribute closure of  $BD$  is  $BDEF$ , which isn't a superset of either relation (does not pass)