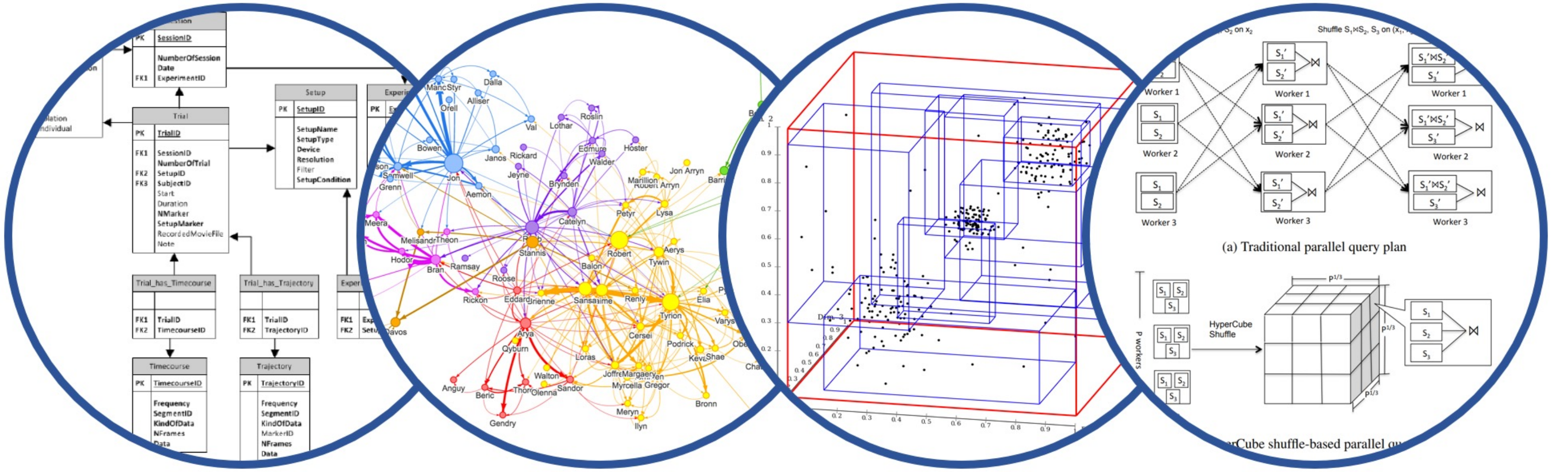


Introduction to Data Management
Review

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Course Evals

- Please take a few minutes before we start to fill out the course evals

- I read every word of your comments and make adjustments where needed based on the feedback; have done this in the past

Final Exam

- Monday, June 3rd, 8:30 – 10:20 in this room
- Comprehensive
- We will have a final review on Friday

Today

Review of this course

Relational Data Model and SQL

Relational Data Model

- Data is stored in simple, flat relations

First Normal Form
1NF

- Is retrieved via a set-at-a-time query language

- No prescription for the physical representation

Physical Data Independence

- User writes SQL query:
  - Says what they want
- System responsible for optimizing SQL query
  - How to do it

Physical Data Independence is the main reason why relational model is the most widely used

SQL

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

SQL

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

Nested Loop Semantics

SQL

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

Can use only attributes, no aggregates

SQL

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

May group by attributes, e.g. YEAR
or expressions, e.g. YEAR/10

SQL

Only attributes or exrepssions mentioned in GROUPY may be used here…

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

SQL

Only attributes or exrepssions mentioned in GROUPY may be used here…

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

Plus, any aggregates

SQL

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

We may use aggregates (same as in SELECT)

SQL

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

Finally, we can order the output

SQL Aggregates

```
count
sum
min
max
avg
```

We can apply min/max to numbers or text

SQL Aggregates

```
count
sum
min
max
avg
```

We can apply min/max to numbers or text

What does this return?

```
select min(Name), max(Name)
from Payroll;
```

Payroll

UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

SQL Aggregates

```
count
sum
min
max
avg
```

We can apply min/max to numbers or text

What does this return?

```
select min(Name), max(Name)
from Payroll;
```

min	max
Allison	Magda

Payroll

UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

SQL: NULLs

- Three-valued logic:

false = 0; unknown = 0.5; true = 1

x AND y = min(x,y);
x OR y = max(x,y);
not x = 1-x

Examples:

- true AND unknown = unknown
- true OR unknown = true
- unknown AND false = false

SQL: Outer Joins

```
SELECT
FROM Table1 LEFT OUTER JOIN Table2 ON …
```

```
INNER JOIN
LEFT OUTER JOIN
RIGHT OUTER JOIN
FULL OUTER JOIN
```

Very useful for GROUP BY queries when we need aggregates on empty groups, e.g. count(*)=0

SQL: Witness or Argmin/Argmax

- SQL has the aggregates min(…) and max(…)
- SQL does not have argmin(…) or argmax(…)

- Solution 1 using WITH:
  - Compute min or max in temporary table
  - Join main table with temp table to find argmin/argmax

- Solution 2 using self-joins:
  - Compute min/max from one copy of the table
  - Join with the other table in the HAVING clause

SQL: Subqueries

- In the FROM clause: better use WITH

- In the SELECT clause: must return single value

- In the WHERE clause:
  - EXISTS or NOT EXISTS
  - IN or NOT IN
  - ALL or ANY
  - They express mathematical quantifiers: $\forall, \exists$

- Can we avoid subquery? Non-monotone queries…

Relational Algebra

Relational Algebra

The 5 basic operations:

1. Selection $\sigma_{\text{condition}}(S)$
2. Projection $\Pi_{\text{attrs}}(S)$
3. Join $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$
4. Union $\cup$
5. Set difference $-$

Add renaming ρ, but we use variables instead

Relational Algebra

The 5 basic operations:

1. Selection $\sigma_{\text{condition}}(S)$
2. Projection $\Pi_{\text{attrs}}(S)$
3. Join $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$
4. Union $\cup$
5. Set difference $-$

Monotone (operations 1–4)

Non-monotone (operation 5)

Add renaming ρ, but we use variables instead

Relational Algebra

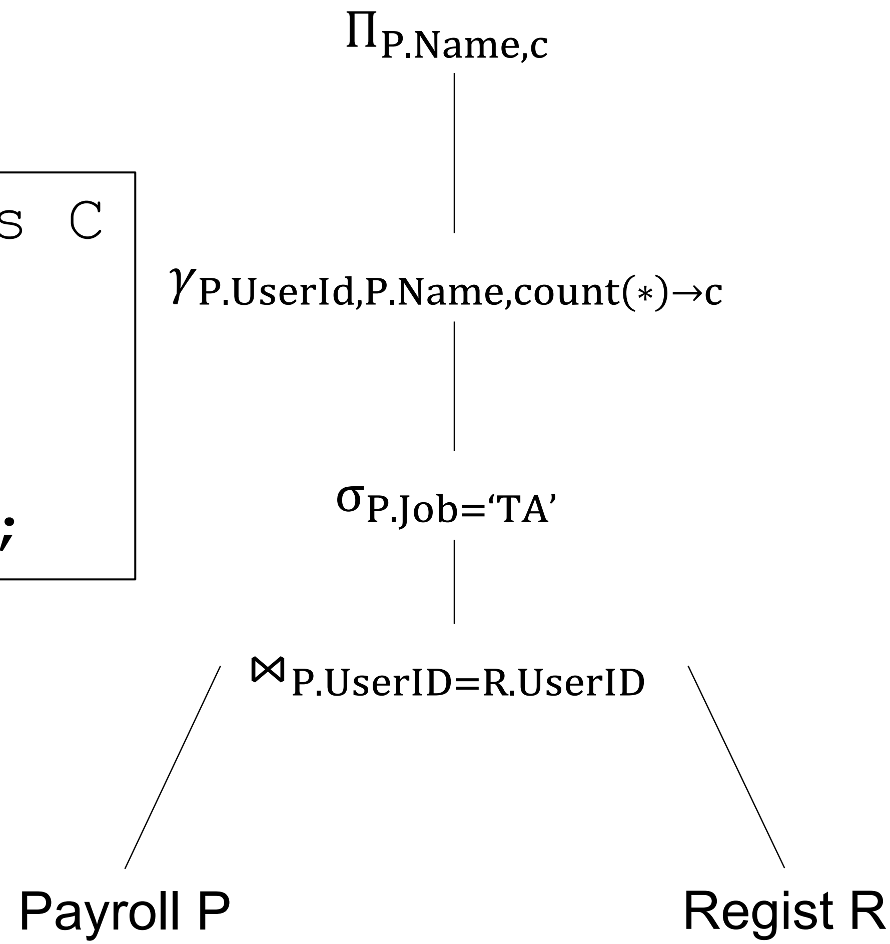
Two extended operator

- Duplicate elimination δ

- Group-by aggregate $\gamma_{\text{attr1,attr2,...,agg1,...}}$

SQL to Relational Algebra Plan

```
SELECT P.Name, count(*) as C
FROM Payroll P, Regist R
WHERE P.UserID = R.UserID
  and P.Job = 'TA'
GROUP BY P.UserID, P.Name;
```



SQL to Relational Algebra Plan

When the query has subqueries then we need to unnest first

```
SELECT P.UserID
FROM Payroll P
WHERE not exists
        (SELECT *
         FROM Regist R
         WHERE P.UserID = R.UserID);
```

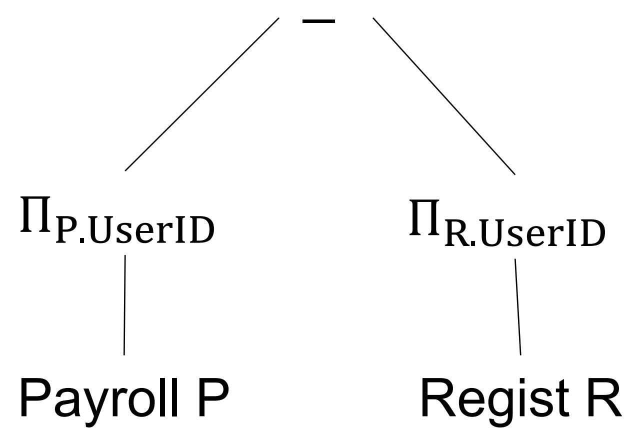
First
de-correlate

```
SELECT P.UserID
FROM Payroll P
WHERE P.UserID not in
        (SELECT R.UserID
         FROM Regist R);
```

Then unnest
using set difference

```
SELECT P.UserID
FROM Payroll P
   EXCEPT
SELECT R.UserID
FROM Regist R;
```

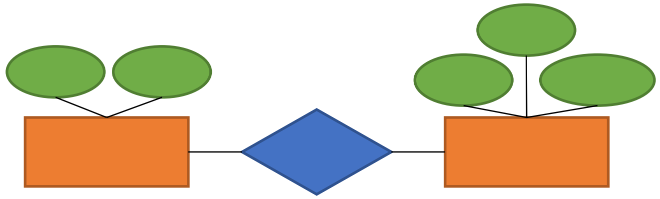
Finally,
rewrite to RA



Design Theory

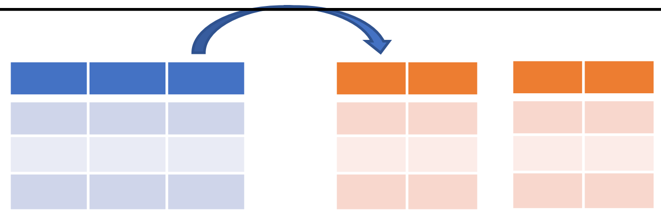
The Database Design Process

Conceptual Model

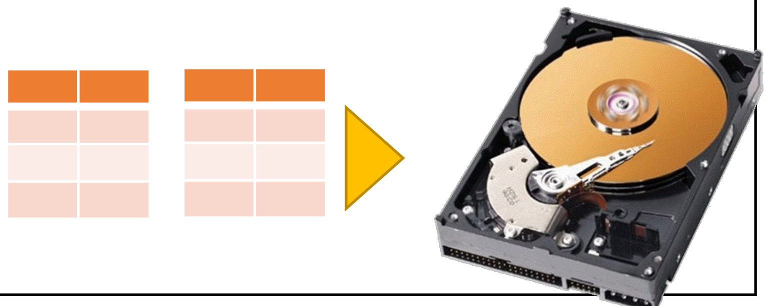


Relational Model
+ Schema
+ Constraints

Conceptual Schema
+ Normalization



Physical Schema
+ Partitioning
+ Indexing

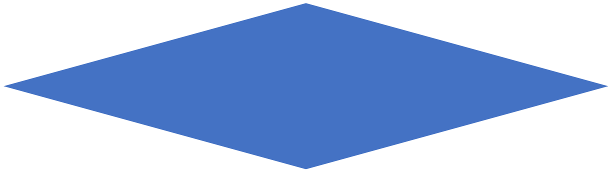


ER Diagrams

Entity set

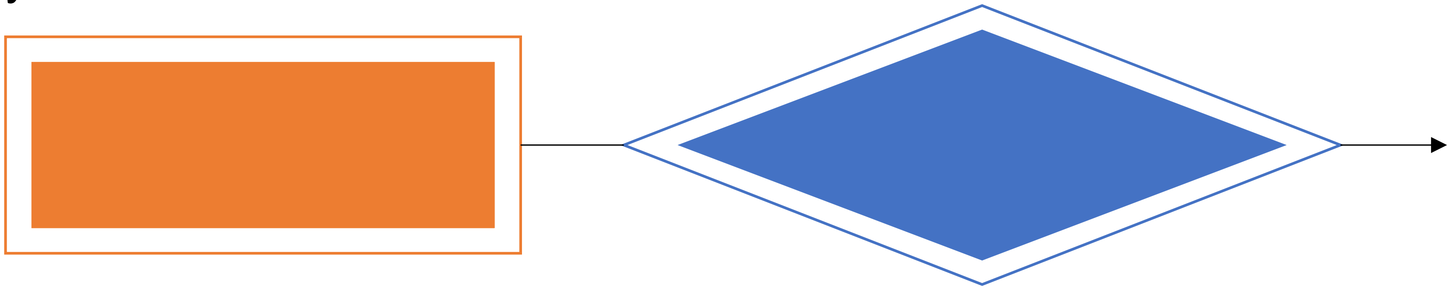
Attribute

Relationship



Subclass

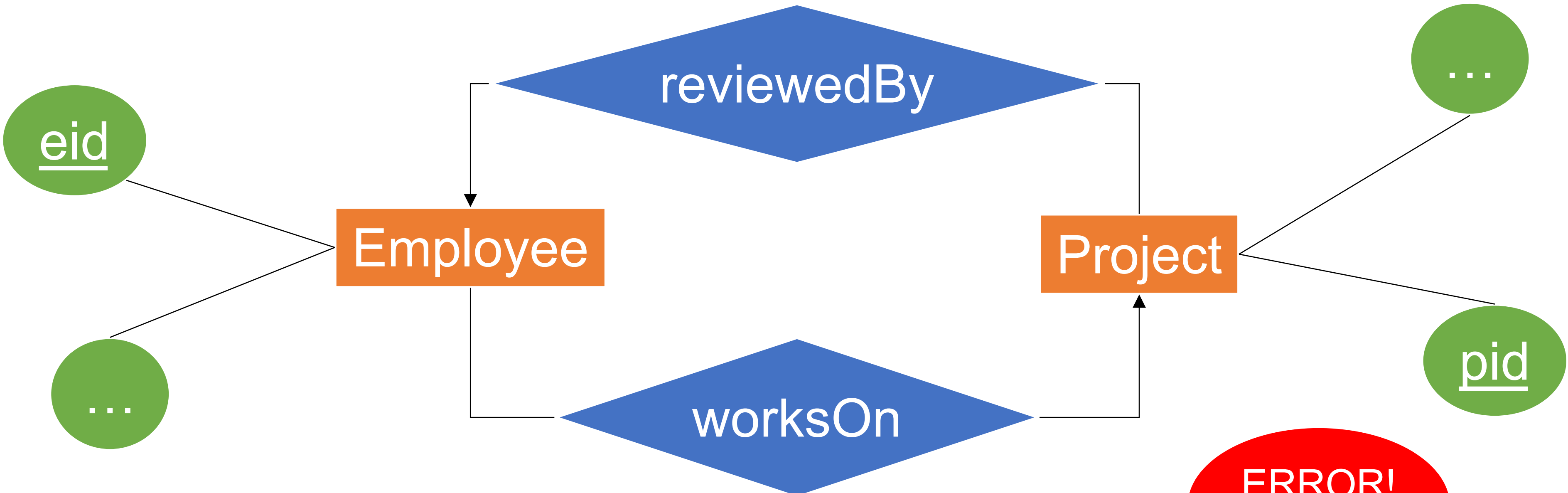
Weak Entity



ER Diagrams to Tables

- Each entity set → a table
- Each relationship → a table with two FKs
  - Except for many-one (or one-one): then add FK
- Each IS_A → a FK

ER Diagrams to Tables: A Problem

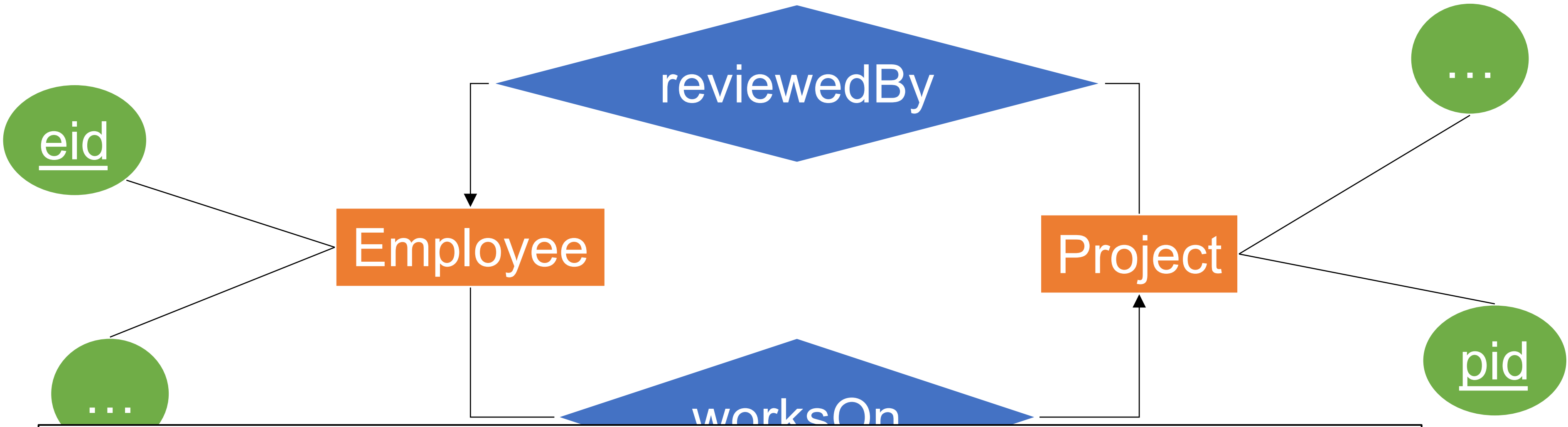



```
CREATE TABLE Empolyee
   (eid int PRIMARY KEY,
    worksOn int REFERENCES Project);
CREATE TABLE Project
    (pid int PRIMARY KEY,
     reviewedBy int REFERENCES Empolyee);
```

ERROR!
(why??)

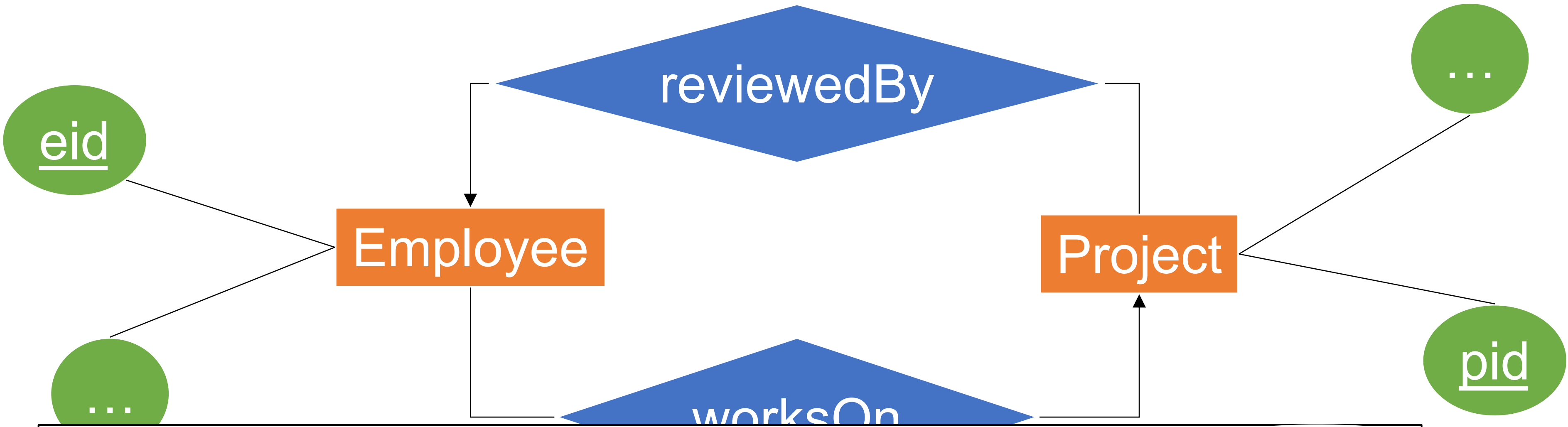
BUT OK
in our class!!!

ER Diagrams to Tables: A Problem




```
CREATE TABLE Empolyee
   (eid int PRIMARY KEY,
    worksOn int);
CREATE TABLE Project
    (pid int PRIMARY KEY,
     reviewedBy int REFERENCES Empolyee);
ALTER TABLE Employee
 ADD Constraint fk_e FOREIGN KEY (worksOn)
                     REFERENCES Project;
```

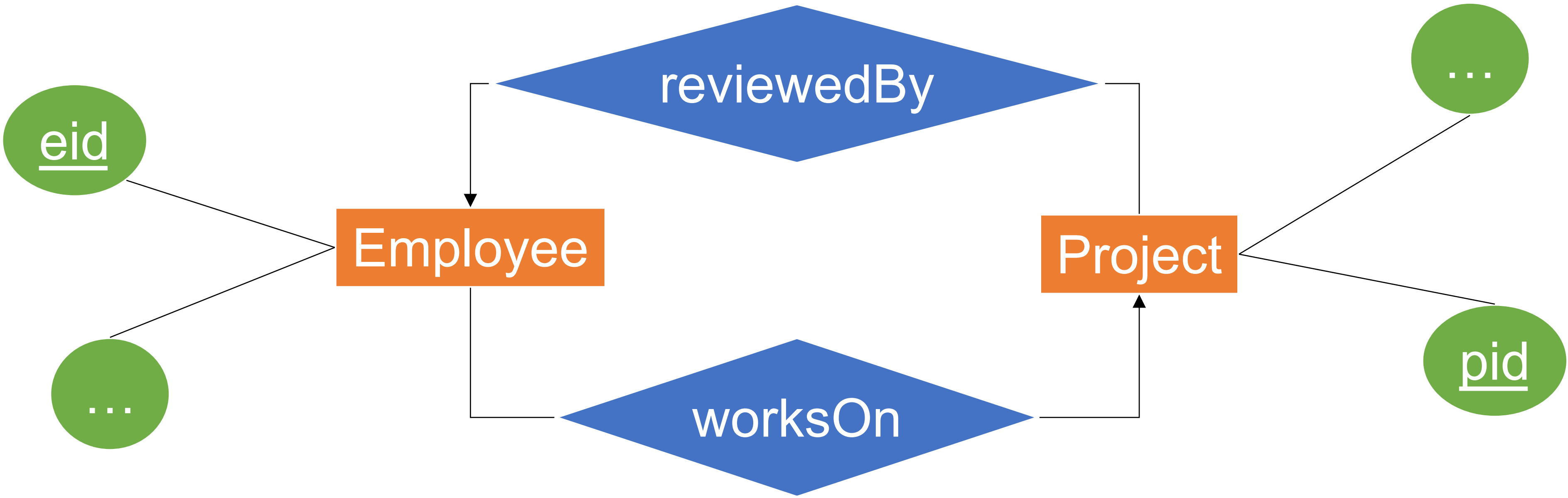
ER Diagrams to Tables: A Problem




```
CREATE TABLE Empolyee
   (eid int PRIMARY KEY,
    worksOn int);
CREATE TABLE Project
    (pid int PRIMARY KEY,
     reviewedBy int REFERENCES Empolyee);
ALTER TABLE Employee
 ADD Constraint fk_e FOREIGN KEY (worksOn)
                     REFERENCES Project;
```

We don’t ask for this on exams.

ER Diagrams to Tables: A Problem




```
CREATE TABLE Empolyee
   (eid int PRIMARY KEY,
    worksOn int REFERENCES Project);
CREATE TABLE Project
    (pid int PRIMARY KEY,
     reviewedBy int REFERENCES Empolyee);
```

This is fine on exam

ER Diagrams to Tables: A Problem

In case you missed it:

- We do not require you in this class to use the ALTER TABLE solution, but assume that mutually recursive definitions are accepted by SQL

Anomalies

The three types of anomalies

- Redundancy anomaly
- Update anomaly
- Deletion anomaly

They all happen because of some FD A → B, where A is not a super-key

Functional Dependencies

Definition:

- A→ B holds if:

  any 2 tuples that have same values of A attributes,

  also have the same values in the B attribute

- Always think about this definition!

Functional Dependencies

Payroll

UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

FD:

UserID → Name, Job, Salary

Functional Dependencies

Payroll

UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

FD:

UserID → Name, Job, Salary

```
SELECT *
FROM Payroll
WHERE Job = 'TA'
```

Functional Dependencies

Payroll

UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

FD:

UserID → Name, Job, Salary

```
SELECT *
FROM Payroll
WHERE Job = 'TA'
```

UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000

Functional Dependencies

Payroll

UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

FD:

UserID → Name, Job, Salary

```
SELECT *
FROM Payroll
WHERE Job = 'TA'
```

UserID	Name	Job	Salary
123	Jack	TA	50000
345	Allison	TA	60000

FDs:

UserID → Name, Job, Salary
Name → Job
Salary → Job

BCNF

- Goal: remove anomalies
- How:
  - Find set of attributes X such that $X \subsetneq X^+ \subsetneq [\text{all}-\text{attrs}]$
  - Split relation into two relation
  - Remember to continue with both relations!

BCNF

$UID \rightarrow Name, City$

UID	Name	Phone	City
234	Fred	206-555-9999	Seattle
234	Fred	206-555-8888	Seattle
987	Joe	415-555-7777	SF

<u>UID</u>	Name	City
234	Fred	Seattle
987	Joe	SF

UID	Phone
234	206-555-9999
234	206-555-8888
987	415-555-7777

BCNF v.s. 3NF

We do not discuss 3NF. In case you are curious:

- Problem with BCNF normalization: may lose FDs
  - R(City, State, Zip): City,State → Zip, Zip → State
  - BCNF normalization: R1(Zip, State), R2(Zip, City)
    we lost the FD City,State → Zip
  - 3NF: the relation is already in 3NF ☺

- Takeaways:
  - BCNF removes all anomalies, may lose some FDs
  - 3NF keeps all FDs, may still have some anomalies

Transactions

Transactions

Two types of SQL workloads:

- Online Analytical Processing (OLAP)
  - Lots of joins, aggregates
  - Rarely any updates
  - Great for data analysis, decision support
- Online Transaction Processing (OLTP)
  - Lots of updates
  - Usually few joins or aggregates
  - Great for data-intensive applications (banking, …)

Transactions

Problem: concurrent updates may corrupt the DB

- Transactions: introduced to ensure the DB remains consistent (assuming all applications are correct)
- ACID: A and I matter most. C is a consequence.
- Transactions slow down DBMS, but necessary for data consistency

Transactions

- Using TXNs is easy:

    BEGIN TRANSACTION
    …
    COMMIT. (or ROLLBACK)

- Implementing TXNs: must have ACID properties

Transactions

Static database:

- A fixed set of elements: A_1, A_2, …
- A TXN is a sequence of Read/Write operations

Dynamic database:

- The set of elements may increase or decrease

Schedules

Start TXN 1

Commit TXN 2

ST_1, $R_1(A)$, ST_2, $R_2(B)$, $W_2(A)$, $R_1(B)$, CO_1, $R_2(C)$, CO_2

Schedules

Start TXN 1

Commit TXN 2

$ST_1, R_1(A), ST_2, R_2(B), W_2(A), R_1(B), CO_1, R_2(C), CO_2$

Maybe this is easier to read:

$ST_1, R_1(A),$ $R_1(B), CO_1,$

$ST_2, R_2(B), W_2(A),$ $R_2(C), CO_2$

→

time

Schedules

Things to know:

- Serial Schedule
- Serializable Schedule
- Conflict Serializable Schedule
- What happens in a static v.s. a dynamic database

Locks, 2PL, Strict 2PL

ST_1, $L_1(A)$, $R_1(A)$, ST_2, $L_2(B)$, $R_2(B)$, $L_2(A)$, $L_1(C)$, $R_1(C)$, CO_1, $W_2(A)$, CO_2

Locks, 2PL, Strict 2PL

ST_1, $L_1(A)$, $R_1(A)$, $L_1(C)$, $R_1(C)$, CO_1,

ST_2, $L_2(B)$, $R_2(B)$, $L_2(A)$, $W_2(A)$, CO_2

Denied:
put on WAIT

Locks, 2PL, Strict 2PL

Release all locks: $U_1(A)$, $U_1(C)$

ST_1, $L_1(A)$, $R_1(A)$, $L_1(C)$, $R_1(C)$, CO_1,

ST_2, $L_2(B)$, $R_2(B)$, $L_2(A)$, $W_2(A)$, CO_2

Denied: put on WAIT

Locks, 2PL, Strict 2PL

Release all locks:
$U_1(A)$, $U_1(C)$

ST_1, $L_1(A)$, $R_1(A)$, ST_2, $L_2(B)$, $R_2(B)$, $L_2(A)$, $L_1(C)$, $R_1(C)$, CO_1, $W_2(A)$, CO_2

Denied:
put on WAIT

Granted:
TXN2 proceeds

Locks, 2PL, Strict 2PL

ST_1, $L_1(A)$, $R_1(A)$,

ST_2, $L_2(B)$, $R_2(B)$, $L_2(A)$,

Denied:
put on WAIT

$L_1(C)$, $R_1(C)$, CO_1,

Release all locks:
$U_1(A)$, $U_1(C)$

$W_2(A)$, CO_2

Granted:
TXN2 proceeds

Release all locks:
$U_2(A)$, $U_2(B)$

Locks, 2PL, Strict 2PL

- Strict 2PL ensures conflict serializability
  - In particular, it ensures serializability
- But only in a static database
- In a dynamic database need to handle phantoms in order to ensure serializability

Types of Locks

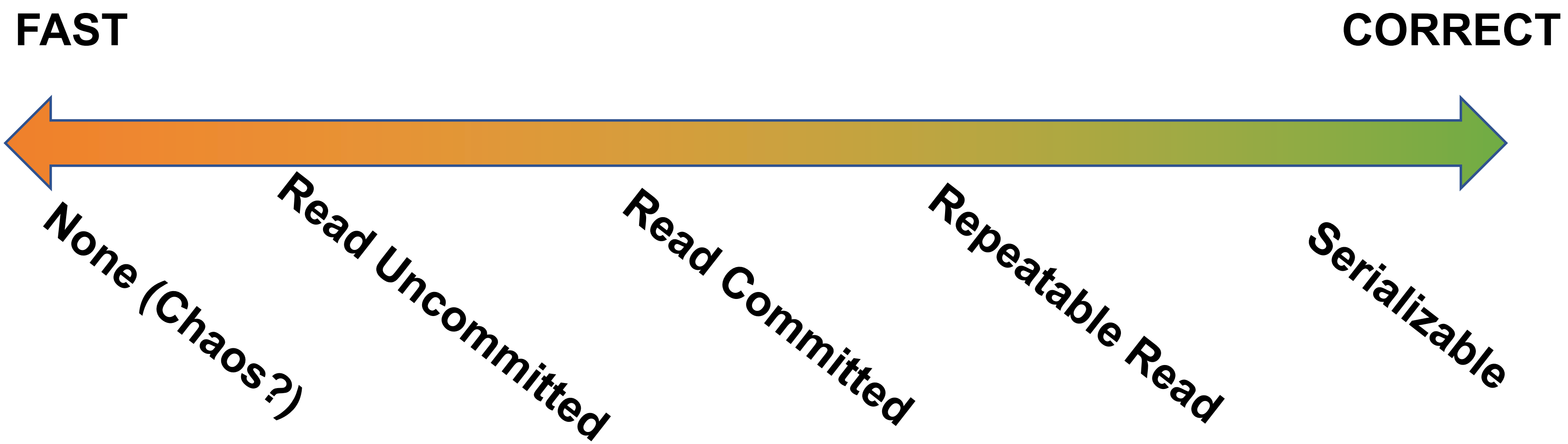
- Shared Locks, or Read Locks:
  - Many TXNs can hold a Read Lock
- Exclusive Locks, or Write Locks:
  - Only one TXN can hold a Write Lock
    No other TXN can hold either a Read or Write Lock
- L(A) replaced by either S(A) or X(A)

Weaker Isolation Levels

Running only serializable schedules is very slow

Solution: use weaker isolation level when possible

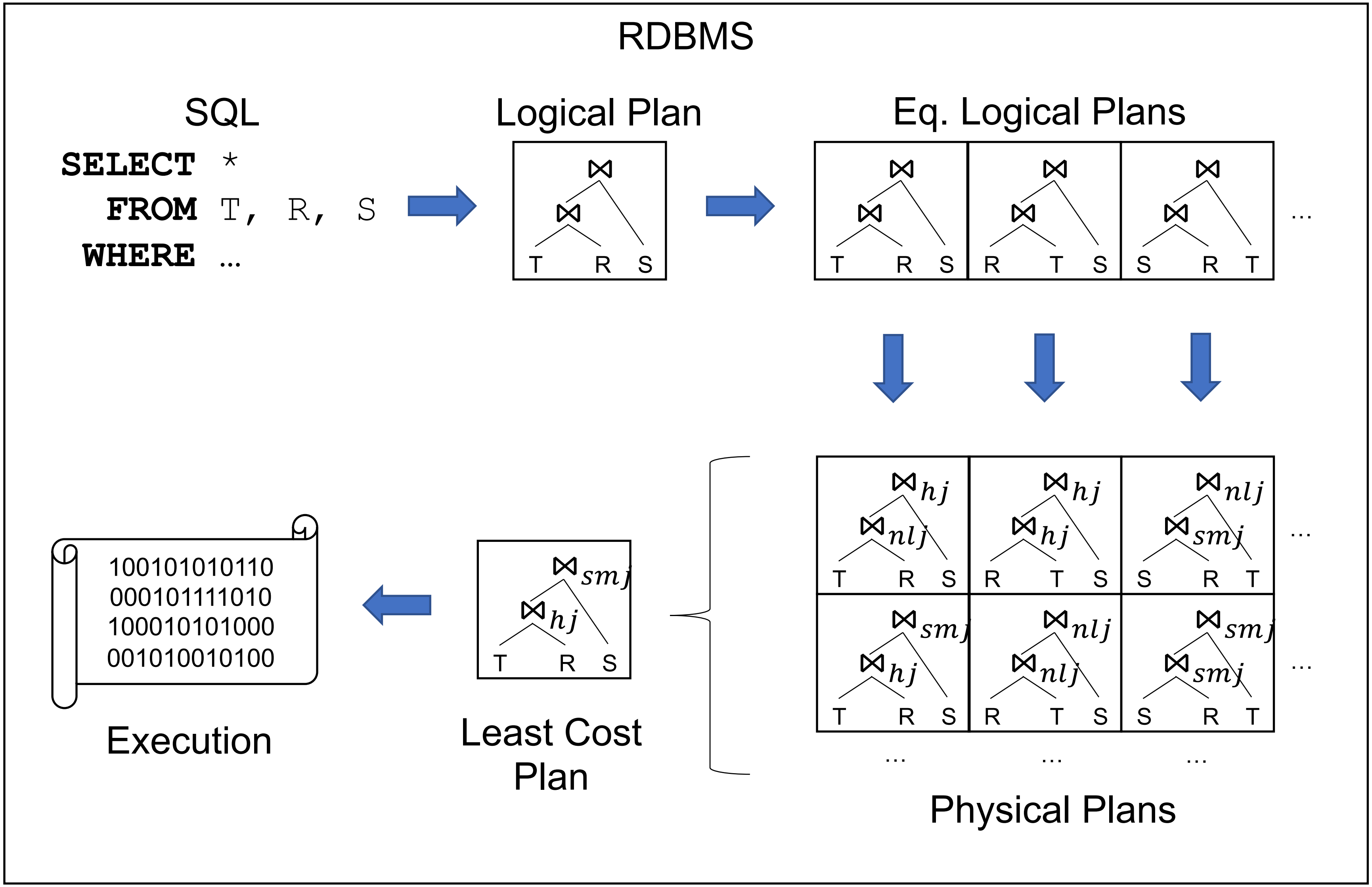
Isolation Level Design Spectrum



(Review in class what they are)

The Query Engine

The Query Engine



Physical Operators

Join $\bowtie$

- Nested loop join
- Hash-join
- Merge-join

Group-by γ

- Nested loop group-by
- Hash-based group-by
- Sort-based group-by

Selection, projection σ, Π

- On-the-fly

Optimization

Query rewrite rules:

- Selection pushdown:
  $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$
  when C refers only to R

- Join associativity:
  $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$

- Many, many others

Cardinality Estimation

Basic statistics

- T(R) = number of tuples
- V(R,A) = number of distinct values in R.A
- Histograms

Cardinality Estimation

Basic estimation formulas

$$\mathrm{EST}\left[\sigma_{A=const}(R)\right]=\theta_{A=const}\cdot T(R)=\frac{T(R)}{V(R,A)}$$

$$\theta_{C_1 \text{ and } C_2}=\theta_{C_1}\cdot\theta_{C_2}$$

$$\mathrm{EST}\left[R\bowtie_{A=B}S\right]=\frac{T(R)\cdot T(S)}{\max(V(R,A),V(S,B))}$$

Cardinality Estimation

Assumptions:

- Uniformity
- Independence
- Containment of values
- Preservation of values

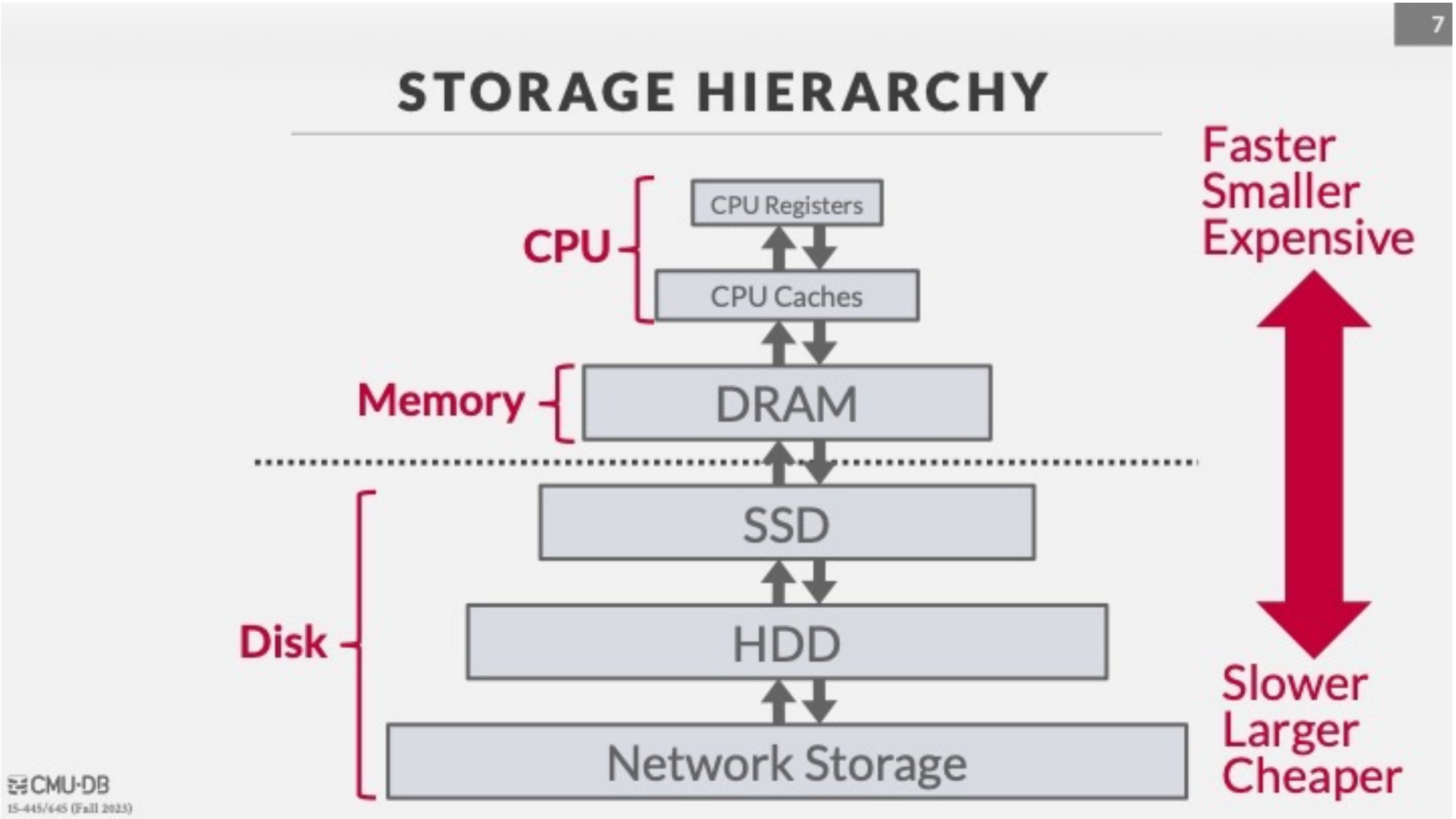
Cardinality Estimation

```
SELECT *
FROM Payroll x, Regist y, Brand z
WHERE x.UserID = y.UserID
  and y.car = z.car
  and x.Job = 'TA';
```

$$\text{Est}(Q) = \frac{T(\text{Payroll}) \cdot T(\text{Regist}) \cdot T(\text{Brand})}{\max(V(\text{Payroll}, \text{UserID}), V(\text{Regist}, \text{UserID})) \cdot \max\big(V(\text{Regist}, \text{car}), V(\text{Brand}, \text{car})\big) \cdot V(\text{Payroll}, \text{Job})}$$

(In class: discuss preservation of values)

Memory Hierarchy



Credit: https://15445.courses.cs.cmu.edu/fall2023/

Data on Disk

- The unit of disk read or write is a block
- Once in main memory, we call it a page
- Block size is fixed. Typically, 4k or 8k or 16k

- Sequential access much faster than random access

Indexes

- Index = an auxiliary file that facilitates faster access to the data

- Usually a B+ tree, but can also be a hash-table

What do these commands do?

- CREATE INDEX Idx1 ON Payroll(Name)
- CREATE INDEX Idx2 ON Payroll(Salary, Job)
- CREATE INDEX Idx3 ON Payroll(Job, Salary)

Multi-attribute Index

```
CREATE INDEX Idx1 on Payroll(job,salary);
```

```
SELECT *
FROM Payroll
WHERE job='TA';
```

```
SELECT *
FROM Payroll
WHERE salary='50000';
```

```
SELECT *
FROM Payroll
WHERE job='TA'
 and salary='50000';
```

(Discussed in class)

SQL++

(we just finished it)

The End

- We covered a lot of material this quarter!
- The details: you will show your mastery at the final
- The high-level concepts: you will remember them throughout your career
- Thanks for a great quarter!
(Come on Friday for the final review!)