WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

CS639: Data Management for Data Science

Midterm Review 1: Relational Databases and Relational Algebra

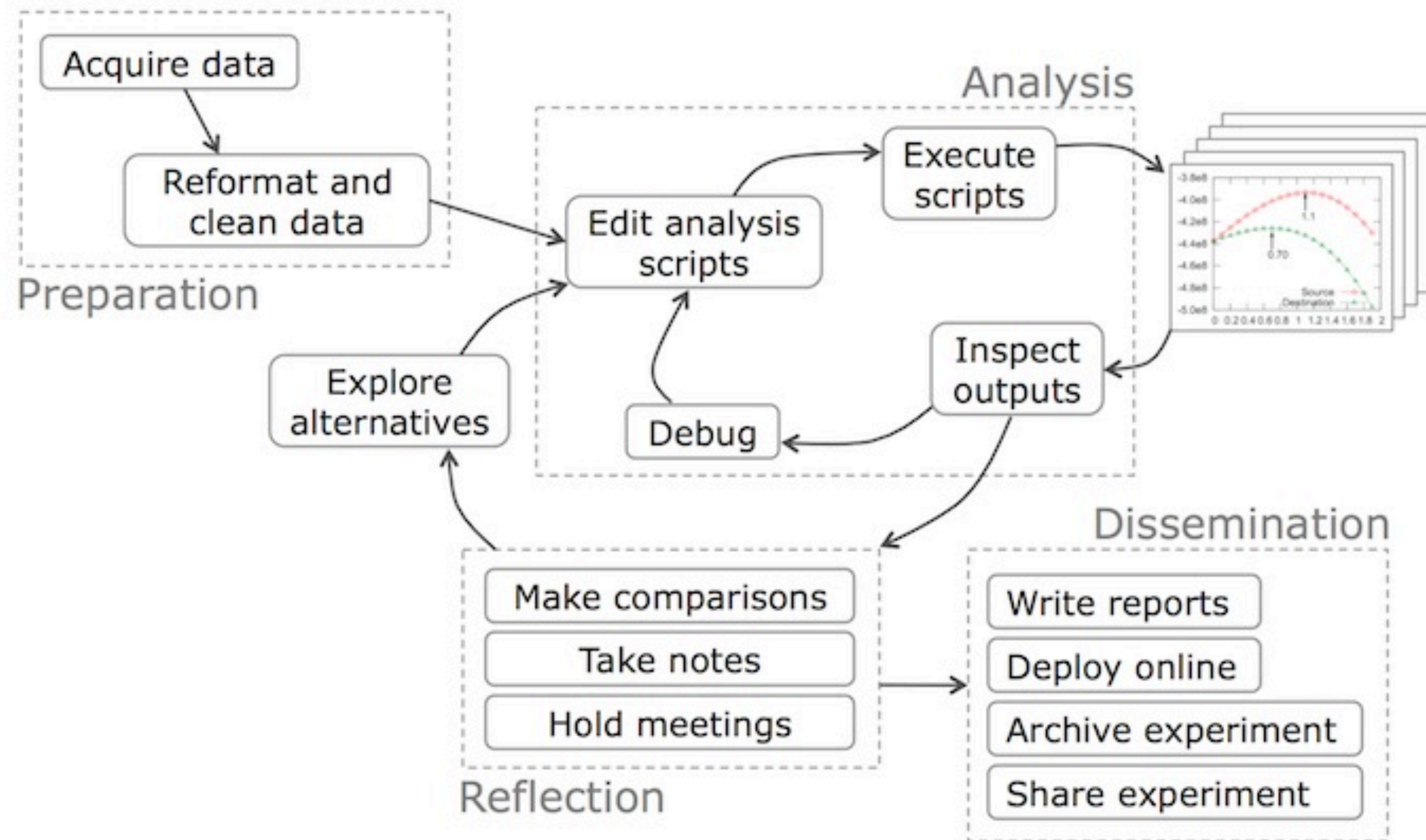
Theodoros Rekatsinas

Today's Lecture

1. Review Relational Databases and Relational Algebra

2. Next Lecture: Review MapReduce and NoSQL systems

Data science workflow



https://cacm.acm.org/blogs/blog-cacm/169199-data-science-workflow-overview-and-challenges/fulltext

Uncertainty and Randomness

- Data represents the **traces** of real-world processes.
  - The collected traces correspond to a **sample** of those processes.
- There is **randomness** and **uncertainty** in the data collection process.
- The process that generates the data is **stochastic** (random).
  - Example: Let’s toss a coin! What will the outcome be? Heads or tails? There are many factors that make a coin toss a stochastic process.
- The sampling process introduces uncertainty.
  - Example: Errors due to sensor position due to error in GPS, errors due to the angles of laser travel etc.

Models

- Data represents the **traces** of real-world processes.

- Part of the data science process: We need to **model** the real-world.

- A model is a function $f_\theta(x)$
  - x: input variables (can be a vector)
  - θ: model parameters

Modeling Uncertainty and Randomness

- Data represents the **traces** of real-world processes.
- There is **randomness** and **uncertainty** in the data collection process.
- A model is a function $f_\theta(x)$
  - x: input variables (can be a vector)
  - θ: model parameters
- Models should rely on **probability theory** to capture uncertainty and randomness!

The Relational Model: Schemata

- Relational Schema:

```
Students(sid: string, name: string, gpa: float)
```

Relation name

String, float, int, etc. are the **domains** of the attributes

Attributes

The Relational Model: Data

An **attribute** (or **column**) is a typed data entry present in each tuple in the relation

Student

sid	name	gpa
001	Bob	3.2
002	Joe	2.8
003	Mary	3.8
004	Alice	3.5

The number of attributes is the **arity** of the relation

The Relational Model: Data

Student

sid	name	gpa
001	Bob	3.2
002	Joe	2.8
003	Mary	3.8
004	Alice	3.5

The number of tuples is the **cardinality** of the relation

A **tuple** or **row** (or *record)* is a single entry in the table having the attributes specified by the schema

The Relational Model: Data

Student

sid	name	gpa
001	Bob	3.2
002	Joe	2.8
003	Mary	3.8
004	Alice	3.5

In practice DBMSs relax the set requirement, and use multisets.

A **relational instance** is a ***set*** of tuples all conforming to the same *schema*

To Reiterate

- A *relational schema* describes the data that is contained in a *relational instance*

Let $R(f_1:Dom_1,\ldots,f_m:Dom_m)$ be a *relational schema* then, an *instance* of R is a subset of $Dom_1 \times Dom_2 \times \ldots \times Dom_n$

In this way, a *relational schema* R is a **total function from attribute *names* to types**

One More Time

- A ***relational schema*** describes the data that is contained in a ***relational instance***

A relation R of arity *t* is a function:
$R : Dom_1 \times \ldots \times Dom_t \rightarrow \{0,1\}$

I.e. returns whether or not a tuple of matching types is a member of it

Then, the schema is simply the *signature* of the function

Note here that order matters, attribute name doesn't…
We'll (mostly) work with the other model (last slide) in which **attribute name matters, order doesn't!**

A relational database

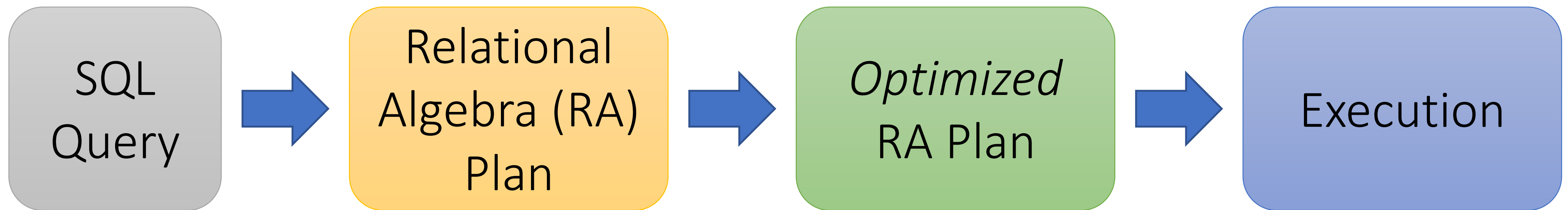
- A *relational database schema* is a set of relational schemata, one for each relation
- A *relational database instance* is a set of relational instances, one for each relation

Two conventions:

1. We call relational database instances as simply ***databases***
2. We assume all instances are valid, i.e., satisfy the *domain constraints*

RDBMS Architecture

How does a SQL engine work ?



Declarative query (from user)

Translate to relational algebra expression

Find logically equivalent- but more efficient- RA expression

Execute each operator of the optimized plan!

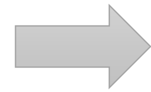
Relational Algebra (RA)

- Five **basic** operators:
  1. Selection: σ
  2. Projection: Π
  3. Cartesian Product: $\times$
  4. Union: $\cup$
  5. Difference: -
- Derived or auxiliary operators:
  - Intersection, complement
  - Joins (natural,equi-join, theta join, semi-join)
  - Renaming: ρ
  - Division

Note that RA Operators are Compositional!

```
Students(sid,sname,gpa)
```

```
SELECT DISTINCT
   sname,
   gpa
FROM Students
WHERE gpa > 3.5;
```



$$\Pi_{sname,gpa}(\sigma_{gpa>3.5}(Students))$$

$$\sigma_{gpa>3.5}(\Pi_{sname,gpa}(Students))$$

How do we represent this query in RA?

Are these logically equivalent?

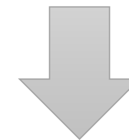
Natural Join (⋈)

- Notation: $R_1 \bowtie R_2$
- Joins R_1 and R_2 on *equality of all shared attributes*
  - If R_1 has attribute set A, and R_2 has attribute set B, and they share attributes A⋂B = C, can also be written: $R_1 \bowtie_C R_2$
- Our first example of a *derived* RA operator:
  - Meaning: $R_1 \bowtie R_2 = \Pi_{A \cup B}(\sigma_{C=D}(\rho_{C \to D}(R_1) \times R_2))$
  - Where:
    - The rename $\rho_{C \to D}$ renames the shared attributes in one of the relations
    - The selection $\sigma_{C=D}$ checks equality of the shared attributes
    - The projection $\Pi_{A \cup B}$ eliminates the duplicate common attributes

```
Students(sid,name,gpa)
People(ssn,name,address)
```

SQL:

```
SELECT DISTINCT
  ssid, S.name, gpa,
  ssn, address
FROM
  Students S,
  People P
WHERE S.name = P.name;
```



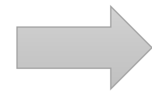
RA:

$$Students \bowtie People$$

Example: Converting SQL Query -> RA

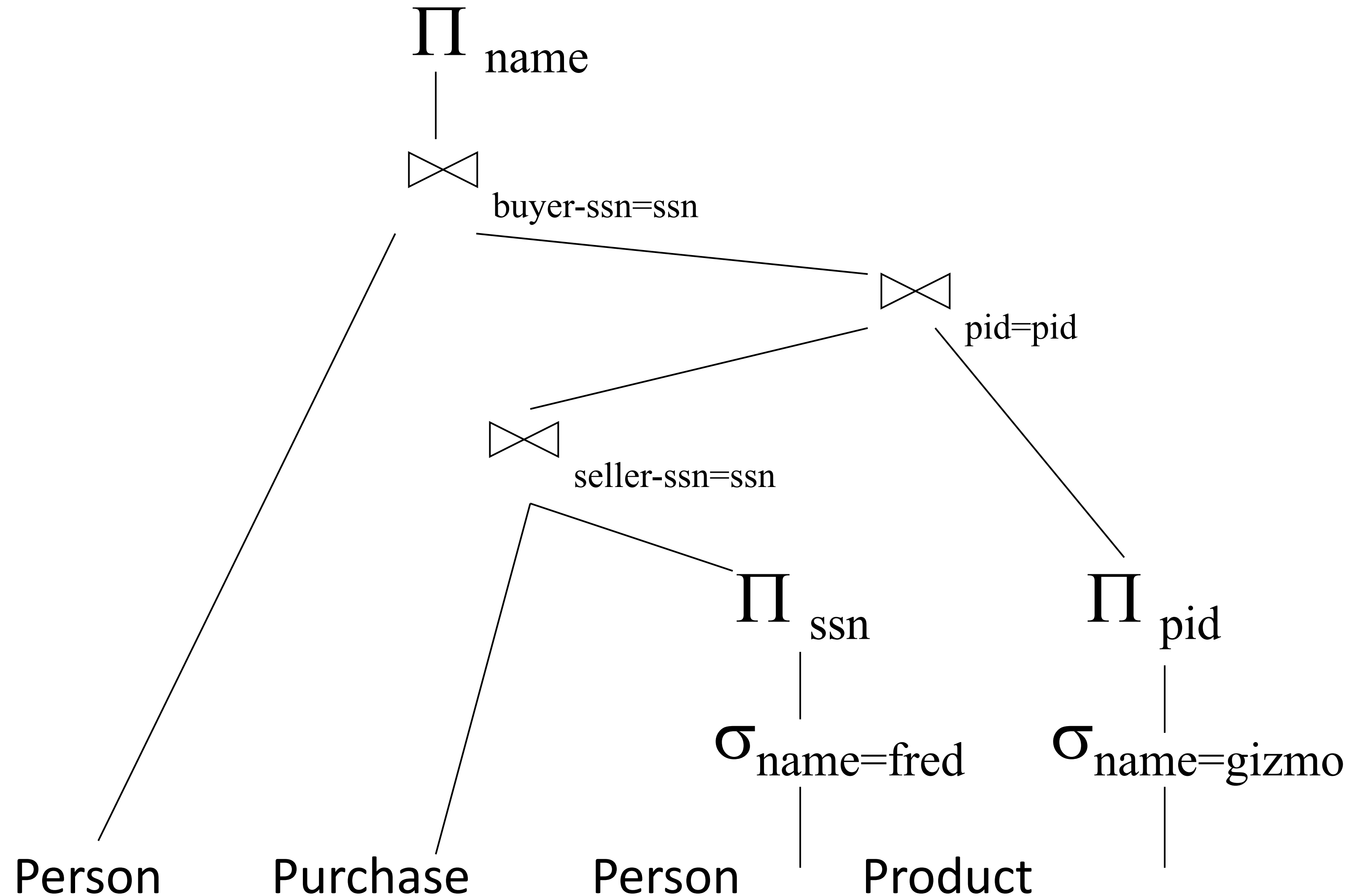
```
Students(sid,sname,gpa)
People(ssn,sname,address)
```

```
SELECT DISTINCT
   gpa,
   address
FROM Students S,
     People P
WHERE gpa > 3.5 AND
    sname = pname;
```



$$\Pi_{gpa,address}(\sigma_{gpa>3.5}(S \bowtie P))$$

RA Expressions Can Get Complex!



RA has Limitations !

- Cannot compute “transitive closure”

Name1	Name2	Relationship
Fred	Mary	Father
Mary	Joe	Cousin
Mary	Bill	Spouse
Nancy	Lou	Sister

- Find all direct and indirect relatives of Fred
- Cannot express in RA !!!
  - Need to write C program, use a graph engine, or modern SQL…

SQL Time!

```
Buyer(buyer-id,name,phone,city,is-ai)
Sale(buyer-id, seller-name,pid, number)
Product(pid, name, price, category, cid)
Company(cid, name, country, debt)
```

- The key of buyer is buyer-id. The attribute is-ai takes the value true when the buyer is an AI.
- In Sale, a buyer-id refers to Buyer.buyer-id, and seller-name is a person's name, and the pid refers to Product.pid. The number of units is the number of units of that product sold.
- A product has an id, and a price. The Product.cid refers to Company.cid
- Company has a key cid, a name, and a country in which it is based.

SQL Time!

```
Buyer(buyer-id,name,phone,city,is-ai)
Sale(buyer-id, seller-name,pid, number)
Product(pid, name, price, category, cid)
Company(cid, name, country, debt)
```

Find all the distinct names of all companies that are based in Japan.

SQL Time!

```
Buyer(buyer-id,name,phone,city,is-ai)
Sale(buyer-id, seller-name,pid, number)
Product(pid, name, price, category, cid)
Company(cid, name, country, debt)
```

Find all the distinct names of all companies that are based in Japan.

```
SELECT DISTINCT c.name
FROM Company c
WHERE c.country = 'Japan'
```

SQL Time!

```
Buyer(buyer-id,name,phone,city,is-ai)
Sale(buyer-id, seller-name,pid, number)
Product(pid, name, price, category, cid)
Company(cid, name, country, debt)
```

Find the distinct names of all companies that are based in Japan and that sold a product to an AI based in Cupertino.

SQL Time!

```
Buyer(buyer-id,name,phone,city,is-ai)
Sale(buyer-id, seller-name,pid, number)
Product(pid, name, price, category, cid)
Company(cid, name, country, debt)
```

Find the distinct names of all companies that are based in Japan and that sold a product to an AI based in Cupertino.

```
SELECT DISTINCT c.name
FROM Company c, Product p, Sale s, Buyer b
WHERE c.country = 'Japan' AND
        p.cid = c.cid AND
        s.pid = p.pid AND
        s.buyer-id = b.buyer-id AND
        b.city = 'Cupertino' AND
        b.is-ai = True;
```

SQL Time!

```
Buyer(buyer-id,name,phone,city,is-ai)
Sale(buyer-id, seller-name,pid, number)
Product(pid, name, price, category, cid)
Company(cid, name, country, debt)
```

Find the distinct names of all companies that have sold at least six distinct products.

SQL Time!

```
Buyer(buyer-id,name,phone,city,is-ai)
Sale(buyer-id, seller-name,pid, number)
Product(pid, name, price, category, cid)
Company(cid, name, country, debt)
```

Find the distinct names of all companies that have sold at least six distinct products.

```
SELECT          c.name
FROM Company c, Product p, Sale s
WHERE p.cid = c.cid AND s.pid = p.pid
GROUP BY c.name
HAVING COUNT(DISTINCT s.pid) >= 6;
```

SQL Time!

```
Buyer(buyer-id,name,phone,city,is-ai)
Sale(buyer-id, seller-name,pid, number)
Product(pid, name, price, category, cid)
Company(cid, name, country, debt)
```

Find the distinct names of all companies that have not sold even a single product.

SQL Time!

```
Buyer(buyer-id,name,phone,city,is-ai)
Sale(buyer-id, seller-name,pid, number)
Product(pid, name, price, category, cid)
Company(cid, name, country, debt)
```

Find the distinct names of all companies that have not sold even a single product.

```
SELECT DISTINCT c.name
FROM Company c
WHERE NOT EXISTS (
    SELECT *
    FROM Product p, Sale s
    WHERE p.cid = c.cid AND s.pid = p.pid
)
```

SQL Time!

```
Buyer(buyer-id,name,phone,city,is-ai)
Sale(buyer-id, seller-name,pid, number)
Product(pid, name, price, category, cid)
Company(cid, name, country, debt)
```

Find the distinct names of all companies such that every product they have ever sold costs at least 10 thousand dollars. Companies that have not sold any products should not be counted, as they are losers.

SQL Time!

```
Buyer(buyer-id,name,phone,city,is-ai)
Sale(buyer-id, seller-name,pid, number)
Product(pid, name, price, category, cid)
Company(cid, name, country, debt)
```

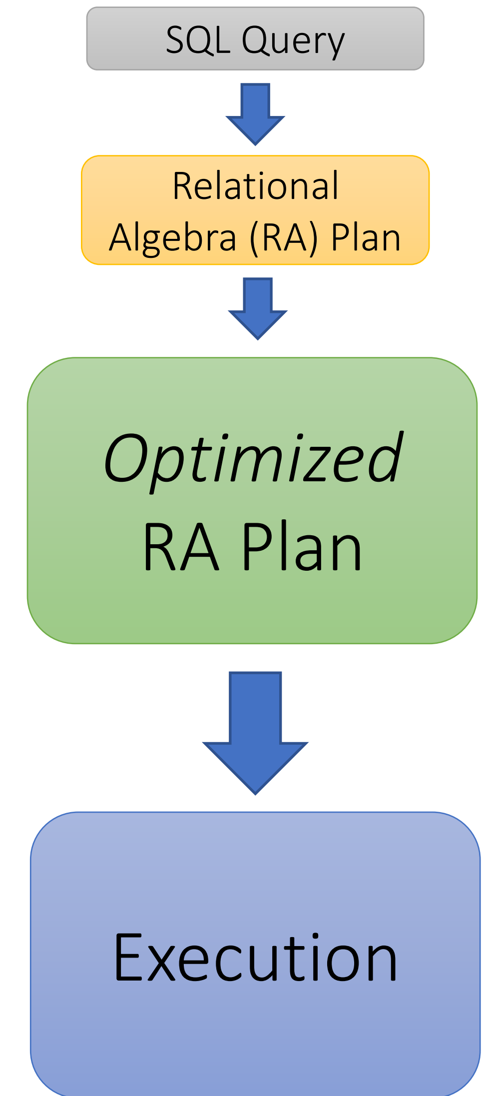
Find the distinct names of all companies such that every product they have ever sold costs at least 10 thousand dollars. Companies that have not sold any products should not be counted, as they are losers.

```
SELECT DISTINCT c.name
FROM Company c
WHERE NOT EXISTS(
    SELECT *
    FROM Product p, Sale s
    WHERE p.cid = c.cid AND s.pid = p.pid AND p.price <= 10000
) AND EXISTS(
    SELECT *
    FROM Product p2, Sale s2
    WHERE p2.cid = c.cid AND s2.pid = p2.pid
)
```

Logical vs. Physical Optimization

- **Logical optimization (we will only see this one):**
  - Find equivalent plans that are more efficient
  - *Intuition: Minimize # of tuples at each step by changing the order of RA operators*

- **Physical optimization:**
  - Find algorithm with lowest IO cost to execute our plan
  - *Intuition: Calculate based on physical parameters (buffer size, etc.) and estimates of data size (histograms)*



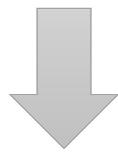
Recall: Logical Equivalence of RA Plans

- Given relations R(A,B) and S(B,C):
  - Here, projection & selection commute:
    - $\sigma_{A=5}(\Pi_A(R)) = \Pi_A(\sigma_{A=5}(R))$
  - What about here?
    - $\sigma_{A=5}(\Pi_B(R)) \,?\!= \Pi_B(\sigma_{A=5}(R))$

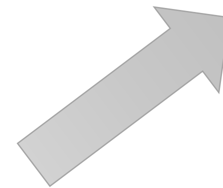
Translating to RA

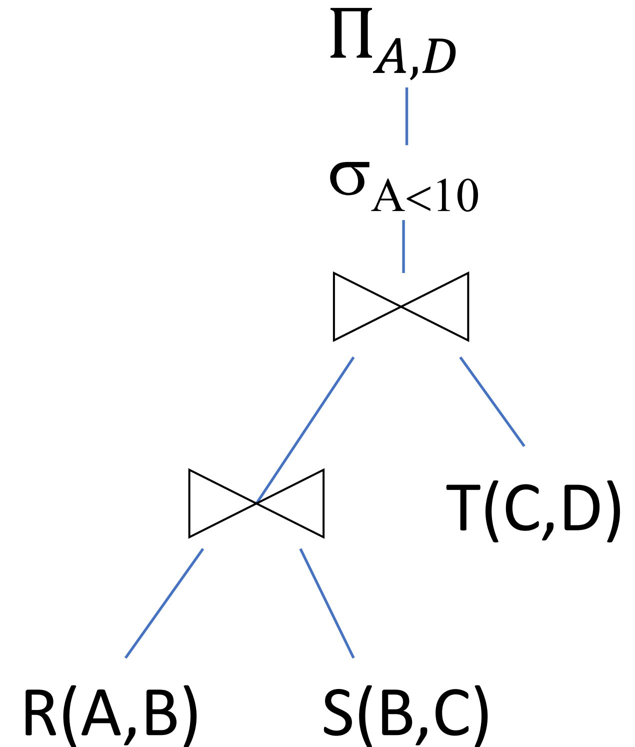
```
R(A,B)  S(B,C)  T(C,D)
```

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



$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$





Logical Optimization

- Heuristically, we want selections and projections to occur as early as possible in the plan
  - Terminology: “push down **selections**” and “pushing down **projections.”**

- **Intuition:** We will have fewer tuples in a plan.
  - Could fail if the selection condition is very expensive (say runs some image processing algorithm).
  - Projection could be a waste of effort, but more rarely.

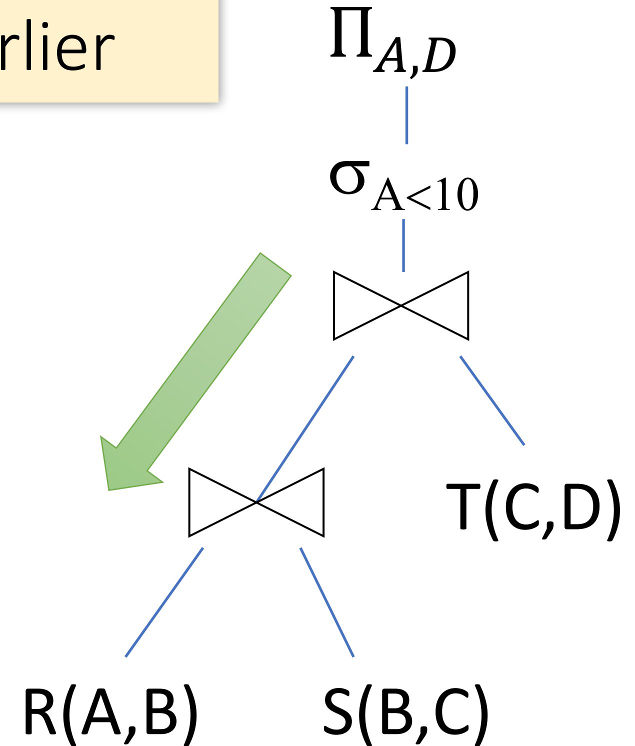
Optimizing RA Plan

```
R(A,B)  S(B,C)  T(C,D)
```

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

$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$

Push down selection on A so it occurs earlier



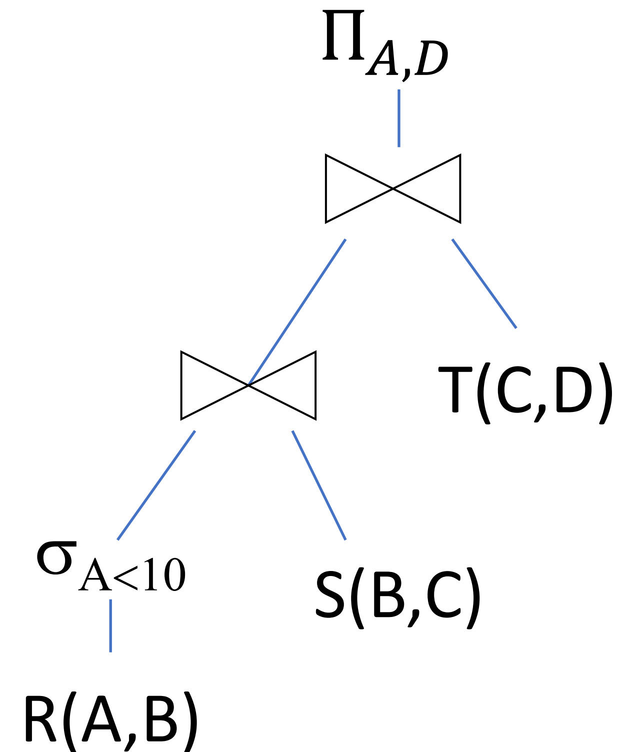
Optimizing RA Plan

```
R(A,B)  S(B,C)  T(C,D)
```

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

$$\Pi_{A,D}(T \bowtie (\sigma_{A<10}(R) \bowtie S))$$

Push down selection on A so it occurs earlier



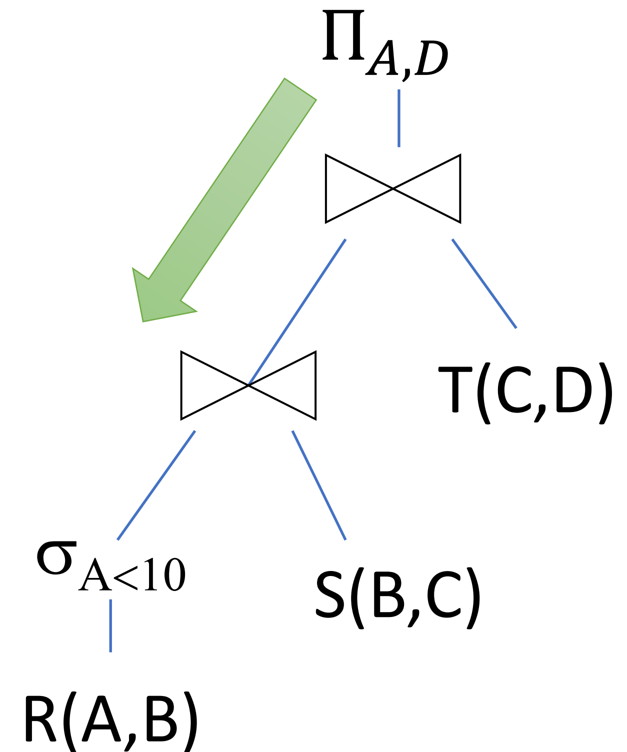
Optimizing RA Plan

```
R(A,B)  S(B,C)  T(C,D)
```

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

$$\Pi_{A,D}(T \bowtie (\sigma_{A<10}(R) \bowtie S))$$

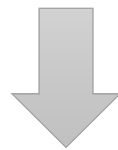
Push down projection so it occurs earlier



Optimizing RA Plan

```
R(A,B)  S(B,C)  T(C,D)
```

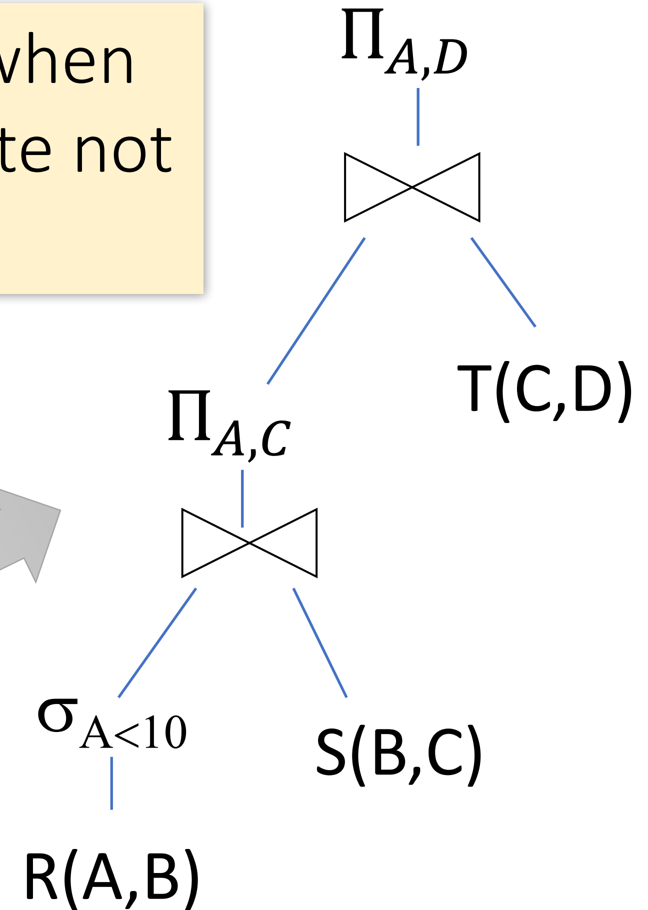
```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```



$$\Pi_{A,D}\left(T \bowtie \Pi_{A,c}(\sigma_{A<10}(R) \bowtie S)\right)$$

We eliminate B earlier!

In general, when is an attribute not needed…?



Please go over the examples here:

- https://courses.cs.washington.edu/courses/cse544/99sp/homeworks/sample/sample.html

- Only the first 4 questions!

Transaction Properties: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

ACID continues to be a source of great debate!

ACID: Atomicity

- TXN’s activities are atomic: **all or nothing**
  - Intuitively: in the real world, a transaction is something that would either occur *completely* or *not at all*
- Two possible outcomes for a TXN
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made

Transactions

- A key concept is the **transaction (TXN)**: an **atomic** sequence of db actions (reads/writes)

Atomicity: An action either completes *entirely* or *not at all*

Acct	Balance
a10	20,000
a20	15,000

Transfer $3k from a10 to a20:

1. Debit $3k from a10
2. Credit $3k to a20

Acct	Balance
a10	17,000
a20	18,000

Written naively, in which states is **atomicity** preserved?

- Crash before 1,
- After 1 but before 2,
- After 2.

DB Always preserves atomicity!

ACID: Consistency

- The tables must always satisfy user-specified ***integrity constraints***
  - *Examples:*
    - Account number is unique
    - Stock amount can’t be negative
    - Sum of *debits* and of *credits* is 0

- How consistency is achieved:
  - Programmer makes sure a txn takes a consistent state to a consistent state
  - *System* makes sure that the txn is **atomic**

ACID: Isolation

- A transaction executes concurrently with other transactions

- **Isolation**: the effect is as if each transaction executes in *isolation* of the others.

  - E.g. Should not be able to observe changes from other transactions during the run

Challenge: Scheduling Concurrent Transactions

- The DBMS ensures that the execution of $\{T_1,\ldots,T_n\}$ is equivalent to some **serial** execution

- One way to accomplish this: **Locking**
  - Before reading or writing, transaction requires a lock from DBMS, holds until the end

- **Key Idea***:* If T_i wants to write to an item x and T_j wants to read x, then T_i, T_j **conflict***.* Solution via locking:
  - only one winner gets the lock
  - loser is blocked (waits) until winner finishes

A set of TXNs is **isolated** if their effect is as if all were executed serially

What if T_i and T_j need X and Y, and T_i asks for X before T_j, and T_j asks for Y before T_i? -> *Deadlock!* One is aborted…

All concurrency issues handled by the DBMS…

ACID: Durability

- The effect of a TXN must continue to exist (***"persist"***) after the TXN
  - And after the whole program has terminated
  - And even if there are power failures, crashes, etc.
  - And etc…

- Means: Write data to **disk**

Ensuring Atomicity & Durability

- DBMS ensures **atomicity** even if a TXN crashes!

- One way to accomplish this: **Write-ahead logging (WAL)**

- **Key Idea**: Keep a log of all the writes done.
  - After a crash, the partially executed TXNs are undone using the log

Write-ahead Logging (WAL): Before any action is finalized, a corresponding log entry is forced to disk

We assume that the log is on "stable" storage

All atomicity issues also handled by the DBMS…

Challenges for ACID properties

- In spite of failures: Power failures, but not media failures
- Users may abort the program: need to "rollback the changes"
  - Need to *log* what happened
- Many users executing concurrently
  - Can be solved via locking (we'll see this next lecture!)

And all this with… Performance!!