CS 145 Midterm Review

The Best Of Collection (Master Tracks), Vol. 1

= requested on piazza (@585)
Announcements

• Thanks for doing the surveys!

• Updates from online survey in (@696)

• **Highlights:**
  • Longer activities (spotted by the Stephanie and Alex),
  • more feedback on HWs (we can do better!)
  • SCPD accessibility

• See Piazza post @728 to vote on topics to cover (in real time!!!)
High-Level: Lecture 2

• Basic terminology:
  • relation / table (+ “instance of”), row / tuple, column / attribute, multiset

• Table schemas in SQL

• Single-table queries:
  • SFW (selection + projection)
  • Basic SQL operators: LIKE, DISTINCT, ORDER BY

• Multi-table queries:
  • Foreign keys
  • JOINS:
    • Basic SQL syntax & semantics of
Tables in SQL

A **relation** or **table** is a multiset of tuples having the attributes specified by the schema.

A **multiset** is an unordered list (or: a set with multiple duplicate instances allowed).

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

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

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>
Table Schemas

• The **schema** of a table is the table name, its attributes, and their types:

```plaintext
Product(Pname: string, Price: float, Category: string, Manufacturer: string)
```

• A **key** is an attribute whose values are unique; we underline a key

```plaintext
Product(Pname: string, Price: float, Category: string, Manufacturer: string)
```
SQL Query

- Basic form (there are many many more bells and whistles)

```
SELECT <attributes>
FROM <one or more relations>
WHERE <conditions>
```

Call this a **SFW** query.
LIKE: Simple String Pattern Matching

```
SELECT * 
FROM Products 
WHERE PName LIKE '%%gizmo%%'
```

DISTINCT: Eliminating Duplicates

```
SELECT DISTINCT Category 
FROM Product 
```

ORDER BY: Sorting the Results

```
SELECT PName, Price 
FROM Product 
WHERE Category='gizmo' 
ORDER BY Price, PName 
```
Joins

Product

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19</td>
<td>Gadgets</td>
<td>GWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29</td>
<td>Gadgets</td>
<td>GWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

Company

<table>
<thead>
<tr>
<th>Cname</th>
<th>Stock</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

SELECT PName, Price
FROM Product, Company
WHERE Manufacturer = CName
AND Country='Japan'
AND Price <= 200

SingleTouch $149.99
An example of SQL semantics

```
SELECT R.A
FROM R, S
WHERE R.A = S.B
```

Output

<table>
<thead>
<tr>
<th>A</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Cross Product

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Apply Selections / Conditions

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
High-Level: Lecture 3

• Set operators
  • INTERSECT, UNION, EXCEPT, [ALL]
  • Subtleties of multiset operations

• Nested queries
  • IN, ANY, ALL, EXISTS
  • Correlated queries

• Aggregation
  • AVG, SUM, COUNT, MIN, MAX, ...

• GROUP BY

• NULLs & Outer Joins
An Unintuitive Query

```
SELECT DISTINCT R.A
FROM R, S, T
WHERE R.A=S.A OR R.A=T.A
```

Computes $R \cap (S \cup T)$

But what if $S = \phi$?

Go back to the semantics!
INTERSECT

\[
\text{SELECT } R.A \\
\text{FROM } R, S \\
\text{WHERE } R.A=S.A
\]

UNION

\[
\text{SELECT } R.A \\
\text{FROM } R, S \\
\text{WHERE } R.A=S.A
\]

EXCEPT

\[
\text{SELECT } R.A \\
\text{FROM } R, S \\
\text{WHERE } R.A=S.A
\]

\[
\text{INTERSECT} \\
\text{SELECT } R.A \\
\text{FROM } R, T \\
\text{WHERE } R.A=T.A
\]

\[
\text{UNION} \\
\text{SELECT } R.A \\
\text{FROM } R, T \\
\text{WHERE } R.A=T.A
\]

\[
\text{EXCEPT} \\
\text{SELECT } R.A \\
\text{FROM } R, T \\
\text{WHERE } R.A=T.A
\]
Nested queries: Sub-queries Returning Relations

```
SELECT c.city
FROM Company c
WHERE c.name IN (  
  SELECT pr.maker  
  FROM Purchase p, Product pr  
  WHERE p.product = pr.name  
    AND p.buyer = 'Joe Blow')
```

“Cities where one can find companies that manufacture products bought by Joe Blow”
Nested Queries: Operator Semantics

**ALL**

```sql
SELECT name
FROM Product
WHERE price > ALL(
    SELECT price
    FROM Product
    WHERE maker = 'G')
```

Find products that are more expensive than all products produced by “G”

**ANY**

```sql
SELECT name
FROM Product
WHERE price > ANY(
    SELECT price
    FROM Product
    WHERE maker = 'G')
```

Find products that are more expensive than any one product produced by “G”

**EXISTS**

```sql
SELECT name
FROM Product p1
WHERE EXISTS (  
    SELECT *
    FROM Product p2
    WHERE p2.maker = 'G'
    AND p1.price = p2.price)
```

Find products where there exists some product with the same price produced by “G”
Nested Queries: Operator Semantics

**ALL**

\[
\text{SELECT } \text{name} \text{ FROM Product WHERE price } > \text{ ALL}(X)
\]

Price must be \text{ > all} entries in multiset \text{ X}

**ANY**

\[
\text{SELECT } \text{name} \text{ FROM Product WHERE price } > \text{ ANY}(X)
\]

Price must be \text{ > at least one} entry in multiset \text{ X}

**EXISTS**

\[
\text{SELECT } \text{name} \text{ FROM Product p1 WHERE EXISTS (X)}
\]

\text{ X must be non-empty}

*Note that p1 can be referenced in X (correlated query!)*
Correlated Queries

```sql
SELECT DISTINCT title
FROM Movie AS m
WHERE year <> ANY(
    SELECT year
    FROM Movie
    WHERE title = m.title)
```

Find movies whose title appears more than once.

Note also: this can still be expressed as single SFW query...
Simple Aggregations

**Purchase**

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
</tbody>
</table>

**SQL Query**

```sql
SELECT SUM(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```

50 (= 1*20 + 1.50*20)
Grouping & Aggregations: GROUP BY

```
SELECT product, SUM(price*quantity)
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING SUM(quantity) > 10
```

Find total sales after 10/1/2005, only for products that have more than 10 total units sold

HAVING clauses contains conditions on aggregates

Whereas WHERE clauses condition on individual tuples...
GROUP BY: (1) Compute FROM-WHERE

```
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING SUM(quantity) > 10
```
GROUP BY: (2) Aggregate by the GROUP BY

SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING SUM(quantity) > 10
GROUP BY: (3) Filter by the **HAVING** clause

```
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING SUM(quantity) > 30
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Craisins</td>
<td>11/1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>11/3</td>
<td>2.5</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

**HAVING**
GROUP BY: (3) SELECT clause

```
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING SUM(quantity) > 100
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>15</td>
</tr>
</tbody>
</table>
General form of Grouping and Aggregation

```
SELECT S
FROM R₁,...,Rₙ
WHERE C₁
GROUP BY a₁,...,aₖ
HAVING C₂
```

Evaluation steps:

1. Evaluate **FROM-WHERE**: apply condition $C₁$ on the attributes in $R₁,...,Rₙ$
2. **GROUP BY** the attributes $a₁,...,aₖ$
3. Apply **HAVING** condition $C₂$ to each group (may have aggregates)
4. Compute aggregates in **SELECT**, $S$, and return the result
Null Values

• For numerical operations, NULL -> NULL:
  • If x = NULL then 4*(3-x)/7 is still NULL

• For boolean operations, in SQL there are three values:

  FALSE  =  0
  UNKNOWN = 0.5
  TRUE    =  1

  • If x= NULL then x="Joe" is UNKNOWN
Null Values

- $C_1 \text{ AND } C_2 = \min(C_1, C_2)$
- $C_1 \text{ OR } C_2 = \max(C_1, C_2)$
- $\text{NOT } C_1 = 1 - C_1$

```
SELECT * 
FROM Person 
WHERE (age < 25) 
  AND (height > 6 AND weight > 190)
```

Rule in SQL: include only tuples that yield TRUE / 1.0

Won’t return e.g.
(age=20 height=NULL weight=200)!
Null Values

Unexpected behavior:

```
SELECT *
FROM Person
WHERE age < 25
    OR age >= 25
```

Some Persons are not included!

```
SELECT *
FROM Person
WHERE age < 25
    OR age >= 25
    OR age IS NULL
```

Now it includes all Persons!

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL
RECAP: Inner Joins

By default, joins in SQL are “inner joins”:

```sql
SELECT Product.name, Purchase.store
FROM Product
    JOIN Purchase ON Product.name = Purchase.prodName
```

Both equivalent: Both INNER JOINS!
### INNER JOIN:

**Product**

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

**Purchase**

<table>
<thead>
<tr>
<th>prodName</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

**SQL Query**

```sql
SELECT Product.name, Purchase.store
FROM Product
INNER JOIN Purchase
    ON Product.name = Purchase.prodName
```

Note: another equivalent way to write an INNER JOIN!
LEFT OUTER JOIN:

<table>
<thead>
<tr>
<th>Product</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>category</td>
<td></td>
</tr>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td></td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Purchase</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>prodName</td>
<td>store</td>
<td></td>
</tr>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
<td></td>
</tr>
</tbody>
</table>

SELECT Product.name, Purchase.store
FROM Product
    LEFT OUTER JOIN Purchase
    ON Product.name = Purchase.prodName
General clarification: Sets vs. Multisets

• In theory, and in any more formal material, by definition all relations are **sets of tuples**

• In SQL, relations (i.e. tables) are **multisets**, meaning you can have duplicate tuples
  • We need this because intermediate results in SQL don’t eliminate duplicates

• If you get confused: just state your assumptions & we’ll be forgiving!
High-Level: Lecture 4

• ER diagrams!
  • Entities (vs. Entity Sets)
  • Relationships
  • Multiplicity
  • Constraints: Keys, single-value, referential, participation, etc...
Entities vs. Entity Sets

Example:

- **Product**
  - **Name**: Xbox
  - **Category**: Total Multimedia System
  - **Price**: $250

- **Product**
  - **Name**: My Little Pony Doll
  - **Category**: Toy
  - **Price**: $25

Entities are not explicitly represented in E/R diagrams!
What is a Relationship?

A relationship between entity sets $P$ and $C$ is a subset of all possible pairs of entities in $P$ and $C$, with tuples uniquely identified by $P$ and $C$’s keys.
What is a Relationship?

A **relationship** between entity sets **P** and **C** is a *subset of all possible pairs of entities in P and C*, with tuples uniquely identified by **P** and **C**’s keys.

**Company**

<table>
<thead>
<tr>
<th>name</th>
<th>Product</th>
<th>category</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>Gizmo</td>
<td>Electronics</td>
<td>$9.99</td>
</tr>
<tr>
<td></td>
<td>GizmoLite</td>
<td>Electronics</td>
<td>$7.50</td>
</tr>
<tr>
<td></td>
<td>Gadget</td>
<td>Toys</td>
<td>$5.50</td>
</tr>
</tbody>
</table>

**Product**

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Electronics</td>
<td>$9.99</td>
</tr>
<tr>
<td>GizmoLite</td>
<td>Electronics</td>
<td>$7.50</td>
</tr>
<tr>
<td>Gadget</td>
<td>Toys</td>
<td>$5.50</td>
</tr>
</tbody>
</table>

**Company C × Product P**

<table>
<thead>
<tr>
<th>C.name</th>
<th>P.name</th>
<th>P.category</th>
<th>P.price</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>Gizmo</td>
<td>Electronics</td>
<td>$9.99</td>
</tr>
<tr>
<td>GizmoWorks</td>
<td>GizmoLite</td>
<td>Electronics</td>
<td>$7.50</td>
</tr>
<tr>
<td>GizmoWorks</td>
<td>Gadget</td>
<td>Toys</td>
<td>$5.50</td>
</tr>
<tr>
<td>GadgetCorp</td>
<td>Gizmo</td>
<td>Electronics</td>
<td>$9.99</td>
</tr>
<tr>
<td>GadgetCorp</td>
<td>GizmoLite</td>
<td>Electronics</td>
<td>$7.50</td>
</tr>
<tr>
<td>GadgetCorp</td>
<td>Gadget</td>
<td>Toys</td>
<td>$5.50</td>
</tr>
</tbody>
</table>

**Makes**

<table>
<thead>
<tr>
<th>C.name</th>
<th>P.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>Gizmo</td>
</tr>
<tr>
<td>GizmoWorks</td>
<td>GizmoLite</td>
</tr>
<tr>
<td>GadgetCorp</td>
<td>Gadget</td>
</tr>
</tbody>
</table>
Multiplicity of E/R Relationships

One-to-one:

Many-to-one:

One-to-many:

Many-to-many:

Indicated using arrows

X -> Y means there exists a function mapping from X to Y (recall the definition of a function)
Constraints in E/R Diagrams

- Finding constraints is part of the E/R modeling process. Commonly used constraints are:

  - **Keys**: Implicit constraints on uniqueness of entities
    - *Ex*: An SSN uniquely identifies a person

  - **Single-value constraints**:
    - *Ex*: a person can have only one father

  - **Referential integrity constraints**: Referenced entities must exist
    - *Ex*: if you work for a company, it must exist in the database

  - **Other constraints**:
    - *Ex*: peoples’ ages are between 0 and 150

Recall FOREIGN KEYS!
RECALL: Mathematical def. of Relationship

- **A mathematical definition:**

  - Let $A$, $B$ be sets
    - $A = \{1, 2, 3\}$, $B = \{a, b, c, d\}$

  - $A \times B$ (the **cross-product**) is the set of all pairs $(a, b)$
    - $A \times B = \{(1, a), (1, b), (1, c), (1, d), (2, a), (2, b), (2, c), (2, d), (3, a), (3, b), (3, c), (3, d)\}$

- **We define a relationship to be a subset of $A \times B$**
  - $R = \{(1, a), (2, c), (2, d), (3, b)\}$
RECALL: Mathematical def. of Relationship

• **A mathematical definition:**
  • Let A, B be sets
  • A × B (the *cross-product*) is the set of all pairs
  • A relationship is a subset of A × B

• **Makes** is relationship- it is a *subset* of Product × Company:
RECALL: Mathematical def. of Relationship

• There can only be **one relationship for every unique combination of entities**

• This also means that **the relationship is uniquely determined by the keys of its entities**

• **Example:** the key for Makes (to right) is \{Product.name, Company.name\}

This follows from our mathematical definition of a relationship- it’s a SET!

Why does this make sense?
High-Level: Lecture 5

• Redundancy & data anomalies

• Functional dependencies
  • For database schema design
  • Given set of FDs, find others implied- using Armstrong’s rules

• Closures
  • Basic algorithm
  • To find all FDs

• Keys & Superkeys
Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes anomalies:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Joe</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>Sam</td>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

Similarly, we can’t reserve a room without students = an **insert anomaly**

If every course is in only one room, contains **redundant** information!

If we update the room number for one tuple, we get inconsistent data = an **update anomaly**

If everyone drops the class, we lose what room the class is in! = a **delete anomaly**

If we update the room number for one tuple, we get inconsistent data = an **update anomaly**
Constraints Prevent (some) Anomalies in the Data

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>CS145</td>
</tr>
<tr>
<td>Joe</td>
<td>CS145</td>
</tr>
<tr>
<td>Sam</td>
<td>CS145</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS145</td>
<td>B01</td>
</tr>
<tr>
<td>CS229</td>
<td>C12</td>
</tr>
</tbody>
</table>

Is this form better?
- Redundancy?
- Update anomaly?
- Delete anomaly?
- Insert anomaly?
### Defn (again):
Given attribute sets \( A = \{A_1, \ldots, A_m\} \) and \( B = \{B_1, \ldots, B_n\} \) in \( R \),

The **functional dependency** \( A \rightarrow B \) on \( R \) holds if for **any** \( t_i, t_j \) in \( R \):

- **if** \( t_i[A_1] = t_j[A_1] \) AND \( t_i[A_2] = t_j[A_2] \) AND ...
- **AND** \( t_i[A_m] = t_j[A_m] \)

**then** \( t_i[B_1] = t_j[B_1] \) AND \( t_i[B_2] = t_j[B_2] \)
- **AND** ...
- **AND** \( t_i[B_n] = t_j[B_n] \)
FDs for Relational Schema Design

• High-level idea: *why do we care about FDs?*

1. Start with some relational *schema*

2. Find out its *functional dependencies (FDs)*

3. Use these to *design a better schema*
   1. One which minimizes possibility of anomalies

(This part can be tricky!)

Midterm Review > Lecture 5
Finding Functional Dependencies

Equivalent to asking: Given a set of FDs, \( F = \{f_1, \ldots, f_n\} \), does an FD \( g \) hold?

**Inference problem**: How do we decide?

Answer: Three simple rules called **Armstrong’s Rules**.

1. Split/Combine,
2. Reduction, and
3. Transitivity... *ideas by picture*
Closure of a set of Attributes

Given a set of attributes $A_1, ..., A_n$ and a set of FDs $F$:

Then the closure, $\{A_1, ..., A_n\}^+$ is the set of attributes $B$ s.t. $\{A_1, ..., A_n\} \rightarrow B$

Example: $F =$

$\{\text{name}\} \rightarrow \{\text{color}\}$
$\{\text{category}\} \rightarrow \{\text{department}\}$
$\{\text{color, category}\} \rightarrow \{\text{price}\}$

Example Closures:

$\{\text{name}\}^+ = \{\text{name, color}\}$
$\{\text{name, category}\}^+ = \{\text{name, category, color, dept, price}\}$
$\{\text{color}\}^+ = \{\text{color}\}$
Closure Algorithm

Start with $X = \{A_1, \ldots, A_n\}$, FDs $F$.

Repeat until $X$ doesn’t change; do:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in $F$ and $\{B_1, \ldots, B_n\} \subseteq X$:
    then add $C$ to $X$.

Return $X$ as $X^+$

$F = \{
\{\text{name}\} \rightarrow \{\text{color}\},
\{\text{category}\} \rightarrow \{\text{dept}\},
\{\text{color}, \text{category}\} \rightarrow \{\text{price}\}\}$

\[
\begin{align*}
\{\text{name}, \text{category}\}^+ &= \{\text{name}, \text{category}\} \\
\{\text{name}, \text{category}\}^+ &= \{\text{name}, \text{category}, \text{color}\} \\
\{\text{name}, \text{category}\}^+ &= \{\text{name}, \text{category}, \text{color}, \text{dept}\} \\
\{\text{name}, \text{category}\}^+ &= \{\text{name}, \text{category}, \text{color}, \text{dept}, \text{price}\}\end{align*}
\]
Keys and Superkeys

A **superkey** is a set of attributes $A_1, ..., A_n$ s.t. for any other attribute $B$ in $R$, we have $\{A_1, ..., A_n\} \rightarrow B$

A **key** is a *minimal* superkey

I.e. all attributes are *functionally determined* by a superkey

Meaning that no subset of a key is also a superkey
CALCULATING Keys and Superkeys

**Superkey?**
- Compute the closure of A
- See if it = the full set of attributes

**Key?**
- Confirm that A is superkey
- Make sure that no subset of A is a superkey
  - *Only need to check one ‘level’ down!*

Let A be a set of attributes, R set of all attributes, F set of FDs:

```python
IsSuperkey(A, R, F):
A^+ = ComputeClosure(A, F)
Return (A^+==R) |
```

```python
IsKey(A, R, F):
If not IsSuperkey(A, R, F):
    return False
For B in SubsetsOf(A, size=len(A)-1):
    if IsSuperkey(B, R, F):
        return False
return True
```

Also see Lecture-5.ipynb!!!
High-Level: Lecture 7

• Conceptual design

• Boyce-Codd Normal Form (BCNF)
  • Definition
  • Algorithm

• Decompositions
  • Lossless vs. Lossy
  • A problem with BCNF

• MVDs
  • *In slightly greater depth since we skipped in lecture...*
Back to Conceptual Design

Now that we know how to find FDs, it’s a straight-forward process:

1. Search for “bad” FDs

2. If there are any, then keep decomposing the table into sub-tables until no more bad FDs

3. When done, the database schema is normalized

Recall: there are several normal forms...
Boyce-Codd Normal Form

BCNF is a simple condition for removing anomalies from relations:

A relation $R$ is in BCNF if:

if $\{A_1, \ldots, A_n\} \rightarrow B$ is a non-trivial FD in $R$
then $\{A_1, \ldots, A_n\}$ is a superkey for $R$

Equivalently: $\forall$ sets of attributes $X$, either ($X^+ = X$) or ($X^+ = \text{all attributes}$)

In other words: there are no “bad” FDs
Example

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>PhoneNumber</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>123-45-6789</td>
<td>206-555-1234</td>
<td>Seattle</td>
</tr>
<tr>
<td>Fred</td>
<td>123-45-6789</td>
<td>206-555-6543</td>
<td>Seattle</td>
</tr>
<tr>
<td>Joe</td>
<td>987-65-4321</td>
<td>908-555-2121</td>
<td>Westfield</td>
</tr>
<tr>
<td>Joe</td>
<td>987-65-4321</td>
<td>908-555-1234</td>
<td>Westfield</td>
</tr>
</tbody>
</table>

{SSN} → {Name, City}

This FD is \textit{bad} because it is \textbf{not} a superkey

What is the key?
{SSN, PhoneNumber}

⇒ \textbf{Not} in BCNF
Example

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>123-45-6789</td>
<td>Seattle</td>
</tr>
<tr>
<td>Joe</td>
<td>987-65-4321</td>
<td>Madison</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>PhoneNumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-45-6789</td>
<td>206-555-1234</td>
</tr>
<tr>
<td>123-45-6789</td>
<td>206-555-6543</td>
</tr>
<tr>
<td>987-65-4321</td>
<td>908-555-2121</td>
</tr>
<tr>
<td>987-65-4321</td>
<td>908-555-1234</td>
</tr>
</tbody>
</table>

\[ \{ \text{SSN} \} \rightarrow \{ \text{Name}, \text{City} \} \]

This FD is now **good** because it is the key.

Let’s check anomalies:
- Redundancy?
- Update?
- Delete?

Now in BCNF!
BCNF Decomposition Algorithm

BCNFDecomp(R):

Find $X$ s.t. $X + \neq X$ and $X + \neq \{\text{all attributes}\}$

if not found then return $R$

let $Y = X + X - X$, $Z = (X + X) C$

decompose $R$ into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

return $BCNFDecomp(R_1), BCNFDecomp(R_2)$
BCNF Decomposition Algorithm

BCNFDcomp(R):
   Find a set of attributes $X$ s.t.: $X^+ \neq X$ and $X^+ \neq \{\text{all attributes}\}$

Find a set of attributes $X$ which has non-trivial “bad” FDs, i.e. is not a superkey, using closures
BCNF Decomposition Algorithm

BCNFD Decomp(R):
  Find a set of attributes X s.t.: X⁺ ≠ X and X⁺ ≠ [all attributes]

  if (not found) then Return R

If no “bad” FDs found, in BCNF!
BCNF Decomposition Algorithm

BCNFDcomp(R):
Find a *set of attributes* \( X \) s.t.: \( X^+ \neq X \) and \( X^+ \neq \) [all attributes]

\[\text{if (not found)} \quad \text{then Return } R\]

**let** \( Y = X^+ - X \), \( Z = (X^+)^C \)

Let \( Y \) be the attributes that *functionally determines* \( X \) (+ that are not in \( X \))

And let \( Z \) be the other attributes that it *doesn’t*
BCNF Decomposition Algorithm

BCNFDecomp(R):
    Find a set of attributes $X$ s.t.: $X^+ \neq X$ and $X^+ \neq \{\text{all attributes}\}$

    if (not found) then Return $R$

    let $Y = X^+ - X$, $Z = (X^+)^C$
    decompose $R$ into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

Split into one relation (table) with $X$ plus the attributes that $X$ determines ($Y$)...
BCNF Decomposition Algorithm

BCNFDecomp(R):

Find a set of attributes $X$ s.t.: $X^+ \neq X$ and $X^+ \neq \text{[all attributes]}

if (not found) then Return $R$

let $Y = X^+ - X$, $Z = (X^+)^C$

decompose $R$ into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

And one relation with $X$ plus the attributes it does not determine ($Z$)
BCNF Decomposition Algorithm

BCNFDcomp(R):

Find a set of attributes X s.t.: $X^+ \neq X$ and $X^+ \neq \{\text{all attributes}\}$

if (not found) then Return R

let $Y = X^+ - X$, $Z = (X^+)^C$

decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

Return BCNFDcomp($R_1$), BCNFDcomp($R_2$)

Proceed recursively until no more “bad” FDs!
Example

BCNFDecomp(R):

Find a set of attributes X s.t.: $X^+ \neq X$ and $X^+ \neq \text{[all attributes]}$

if (not found) then Return R

let $Y = X^+ - X, \ Z = (X^+)^C$

decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

Return BCNFDecomp($R_1$), BCNFDecomp($R_2$)
Example

\[ R(A,B,C,D,E) \]
\[ \{A\}^+ = \{A,B,C,D\} \neq \{A,B,C,D,E\} \]

\[ R_1(A,B,C,D) \]
\[ \{C\}^+ = \{C,D\} \neq \{A,B,C,D\} \]

\[ R_{11}(C,D) \]
\[ R_{12}(A,B,C) \]
\[ R_2(A,E) \]

\[ R(A,B,C,D,E) \]
\[ \{A\} \rightarrow \{B,C\} \]
\[ \{C\} \rightarrow \{D\} \]
Lossless Decompositions

If \( \{A_1, ..., A_n\} \rightarrow \{B_1, ..., B_m\} \)
Then the decomposition is lossless

Note: don’t need \( \{A_1, ..., A_n\} \rightarrow \{C_1, ..., C_p\} \)

BCNF decomposition is always lossless. Why?
A Problem with BCNF

We do a BCNF decomposition on a “bad” FD:
\{Unit\}+ = \{Unit, Company\}

We lose the FD \{Company, Product\} \rightarrow \{Unit\}!!
Multiple Value Dependencies (MVDs)

Many of you asked, “what do these mean in real life?”

Grad student CA thinks: “Hmm... what is real life?? Watching a movie over the weekend?”
## MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater</th>
<th>film_name</th>
<th>snack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

Are there any functional dependencies that might hold here? **No…**

And yet it seems like there is some pattern / dependency…
# MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater</th>
<th>film_name</th>
<th>snack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

For a given movie theatre...
## MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater</th>
<th>film_name</th>
<th>snack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>
## MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater</th>
<th>film_name</th>
<th>snack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

For a given movie theatre...

Given a set of movies and snacks...

Any movie / snack combination is possible!
### MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater (A)</th>
<th>film_name (B)</th>
<th>Snack (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

More formally, we write $\{A\} \rightarrow \{B\}$ if for any tuples $t_1, t_2$ s.t. $t_1[A] = t_2[A]$.
### MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater (A)</th>
<th>film_name (B)</th>
<th>Snack (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
</tr>
<tr>
<td>t₃</td>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
</tr>
<tr>
<td></td>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
</tr>
<tr>
<td>t₂</td>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
</tr>
<tr>
<td></td>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
</tr>
<tr>
<td></td>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
</tr>
</tbody>
</table>

More formally, we write \( \{A\} \rightarrow \{B\} \) if for any tuples \( t₁, t₂ \) s.t. \( t₁[A] = t₂[A] \) there is a tuple \( t₃ \) s.t.

- \( T₃[A] = t₁[A] \)
## MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater (A)</th>
<th>film_name (B)</th>
<th>Snack (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁ Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>t₂ Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>t₃ Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td></td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td></td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td></td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

More formally, we write \( \{A\} \Rightarrow \{B\} \) if for any tuples \( t₁, t₂ \) s.t. \( t₁[A] = t₂[A] \) there is a tuple \( t₃ \) s.t.
- \( t₃[A] = t₁[A] \)
- \( t₃[B] = t₁[B] \)
**MVDs: Movie Theatre Example**

<table>
<thead>
<tr>
<th>Movie_theater (A)</th>
<th>film_name (B)</th>
<th>Snack (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp;</td>
<td>Kale Chips</td>
</tr>
<tr>
<td></td>
<td>Extended Edition</td>
<td></td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp;</td>
<td>Burrito</td>
</tr>
<tr>
<td></td>
<td>Extended Edition</td>
<td></td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

More formally, we write \( \{A\} \rightarrow \{B\} \) if for any tuples \( t_1, t_2 \) s.t. \( t_1[A] = t_2[A] \) there is a tuple \( t_3 \) s.t.
- \( t_3[A] = t_1[A] \)
- \( t_3[B] = t_1[B] \)
- and \( t_3[R\setminus B] = t_2[R\setminus B] \)

Where \( R\setminus B \) is “R minus B” i.e. the attributes of R not in B.
### MVDs: Movie Theatre Example

<table>
<thead>
<tr>
<th>Movie_theater (A)</th>
<th>film_name (B)</th>
<th>Snack (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

Note this also works!

Remember, an MVD holds over *a relation or an instance*, so defn. must hold for every applicable pair…
<table>
<thead>
<tr>
<th>Movie_theater (A)</th>
<th>film_name (B)</th>
<th>Snack (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₂ Rains 216</td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Kale Chips</td>
</tr>
<tr>
<td></td>
<td>Star Trek: The Wrath of Kahn</td>
<td>Burrito</td>
</tr>
<tr>
<td>t₃ Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Kale Chips</td>
</tr>
<tr>
<td>t₁ Rains 216</td>
<td>Lord of the Rings: Concatenated &amp; Extended Edition</td>
<td>Burrito</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Ramen</td>
</tr>
<tr>
<td>Rains 218</td>
<td>Star Wars: The Boba Fett Prequel</td>
<td>Plain Pasta</td>
</tr>
</tbody>
</table>

This expresses a sort of dependency (= data redundancy) that we can’t express with FDs.

*Actually, it expresses conditional independence (between film and snack given movie theatre)!
MVDs...

Think you can’t understand them?

YES YOU
High-Level: Lecture 8

• Our model of the computer: Disk vs. RAM, local vs. global

• Transactions (TXNs)

• ACID

• Logging for Atomicity & Durability
  • Write-ahead logging (WAL)
High-level: Disk vs. Main Memory

**Disk:**

- **Slow:** Sequential access
  - (although fast sequential reads)

- **Durable:** We will assume that once on disk, data is safe!

- **Cheap**

**Random Access Memory (RAM) or Main Memory:**

- **Fast:** Random access, byte addressable
  - ~10x faster for sequential access
  - ~100,000x faster for random access!

- **Volatile:** Data can be lost if e.g. crash occurs, power goes out, etc!

- **Expensive:** For $100, get 16GB of RAM vs. 2TB of disk!
Our model: Three Types of Regions of Memory

1. **Local**: In our model each process in a DBMS has its own local memory, where it stores values that only it “sees.”

2. **Global**: Each process can read from / write to shared data in main memory.

3. **Disk**: Global memory can read from / flush to disk.

4. **Log**: Assume on stable disk storage- spans both main memory and disk...

Log is a sequence from main memory -> disk.

“Flushing” to disk” = writing to disk + erasing (“evicting”) from main memory.
Transactions: Basic Definition

A transaction ("TXN") is a sequence of one or more operations (reads or writes) which reflects a single real-world transition.

START TRANSACTION
UPDATE Product
SET Price = Price - 1.99
WHERE pname = 'Gizmo'
COMMIT

In the real world, a TXN either happened completely or not at all.
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 is/was source of great debate!
Goal of LOGGING: Ensuring Atomicity & Durability

• **Atomicity:**
  - TXNs should either happen completely or not at all
  - If abort / crash during TXN, *no* effects should be seen

• **Durability:**
  - If DBMS stops running, changes due to completed TXNs should all persist
  - *Just store on stable disk*
Basic Idea: (Physical) Logging

- Record UNDO information for every update!
  - Sequential writes to log
  - Minimal info (diff) written to log

- The log consists of an ordered list of actions
  - Log record contains:
    <XID, location, old data, new data>

This is sufficient to UNDO any transaction!
Write-ahead Logging (WAL) Commit Protocol

This time, let’s try committing after we’ve written log to disk but before we’ve written data to disk... this is WAL!

If we crash now, is T durable?

A: 0 → 1

T: R(A), W(A)

Main Memory

A=1
B=5

Log

Data on Disk

Log on Disk

OK, Commit!
Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

This time, let’s try committing after we’ve written log to disk but before we’ve written data to disk... this is WAL!

If we crash now, is T durable?

USE THE LOG!
Write-Ahead Logging (WAL)

• DB uses **Write-Ahead Logging (WAL)** Protocol:

1. Must *force log record* for an update *before* the corresponding data page goes to storage

2. Must *write all log records* for a TX *before commit*

   - Each update is logged! Why not reads?

   → **Atomicity**

   → **Durability**
High-Level: Lecture 9

• Motivation: Concurrency with Isolation & consistency
  • Using TXNs...

• Scheduling

• Serializability

• Conflict types & classic anomalies
Concurrency: Isolation & Consistency

• The DBMS must handle concurrency such that...

1. **Isolation** is maintained: Users must be able to execute each TXN as if they were the only user
   • DBMS handles the details of *interleaving* various TXNs

2. **Consistency** is maintained: TXNs must leave the DB in a consistent state
   • DBMS handles the details of enforcing integrity constraints
Example- consider two TXNs:

The DBMS can also **interleave** the TXNs

$T_1$

$A += 100$

$B -= 100$

$T_2$

$A *= 1.06$

$B *= 1.06$

What goes / could go wrong here??
Scheduling examples

Starting Balance

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>$50</td>
<td>$200</td>
</tr>
</tbody>
</table>

Serial schedule $T_1 \rightarrow T_2$:

$T_1$

A += 100

B -= 100

$T_2$

A *= 1.06

B *= 1.06

Interleaved schedule B:

$T_1$

A += 100

$T_2$

A *= 1.06

B *= 1.06

Different result than serial $T_1 \rightarrow T_2$!

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final</td>
<td>$159</td>
<td>$106</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final</td>
<td>$159</td>
<td>$112</td>
</tr>
</tbody>
</table>
Scheduling Definitions

• A **serial schedule** is one that does not interleave the actions of different transactions

• A and B are **equivalent schedules** if, *for any database state*, the effect on DB of executing A is **identical to** the effect of executing B

• A **serializable schedule** is a schedule that is equivalent to **some** serial execution of the transactions.

The word “**some**” makes this def powerful and tricky!
Serializable?

Serial schedules:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1 \rightarrow T_2$</td>
<td>$1.06(A+100)$</td>
<td>$1.06(B-100)$</td>
</tr>
<tr>
<td>$T_2 \rightarrow T_1$</td>
<td>$1.06A + 100$</td>
<td>$1.06B - 100$</td>
</tr>
</tbody>
</table>

A $+$ = 100
B $-$ = 100
A $*$ = 1.06
B $*$ = 1.06

Same as a serial schedule
for all possible values of
A, B = serializable
The DBMS’s view of the schedule

Each action in the TXNs reads a value from global memory and then writes one back to it. Scheduling order matters!
Conflict Types

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write.

- Thus, there are three types of conflicts:
  - Read-Write conflicts (RW)
  - Write-Read conflicts (WR)
  - Write-Write conflicts (WW)

Why no “RR Conflict”?

Interleaving anomalies occur with / because of these conflicts between TXNs (*but these conflicts can occur without causing anomalies!*).
Classic Anomalies with Interleaved Execution

“Unrepeatable read”: 

“Dirty read” / Reading uncommitted data:

“Inconsistent read” / Reading partial commits:

Partially-lost update:
Notes

• Locking & etc. (all content in lecture 9 after activity 9-1) will not be covered

• PS1 review slides included as appendix (after this...)

• PS2 & additional content covered in extra review session on Sunday, in Gates 104
Linear Algebra, Declaratively

- Matrix multiplication & other operations = just *joins*!

- The shift from **procedural** to **declarative** programming

\[ C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj} \]

C = \[[0]*p for i in range(n)]

```python
for i in range(n):
    for j in range(p):
        for k in range(m):
            C[i][j] += A[i][k] * B[k][j]
```

*Proceed* through a series of instructions

```sql
SELECT A.i, B.j, SUM(A.x * B.x)
FROM A, B
WHERE A.j = B.i
GROUP BY A.i, B.j;
```

Declare a desired output set
Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```sql
SELECT station_id,
       COUNT(day) AS nbd
FROM precipitation,
    (SELECT day, MAX(precip)
     FROM precipitation
     GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```

Think about order*!

*of the semantics, not the actual execution
Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id, 
       COUNT(day) AS nbd
FROM precipitation,
   (SELECT day, MAX(precip) 
    FROM precipitation 
    GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip 
GROUP BY station_id 
HAVING COUNT(day) > 1 
ORDER BY nbd DESC;
```

Get the max precipitation by day
Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```sql
SELECT station_id, COUNT(day) AS nbd
FROM precipitation,
     (SELECT day, MAX(precip)
      FROM precipitation
      GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```

Get the station, day pairs where / when this happened

Get the max precipitation by day
Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```sql
SELECT station_id,
       COUNT(day) AS nbd
FROM precipitation,
     (SELECT day, MAX(precip)
      FROM precipitation
      GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```

Get the max precipitation by day
Get the station, day pairs where / when this happened
Group by stations
Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id, COUNT(day) AS nbd
FROM precipitation,
(SELECT day, MAX(precip)
 FROM precipitation
 GROUP BY day) AS m
WHERE day = m.day AND precip = m.precip
GROUP BY station_id
HAVING COUNT(day) > 1
ORDER BY nbd DESC;
```

Get the max precipitation by day

Get the station, day pairs where / when this happened

Group by stations

Having > 1 such day
Common SQL Query Paradigms

Complex correlated queries

```sql
SELECT x1.p AS median
FROM x AS x1
WHERE
  (SELECT COUNT(*)
   FROM x AS x2
   WHERE x2.p > x1.p)
= 
  (SELECT COUNT(*)
   FROM x AS x2
   WHERE x2.p < x1.p);
```

This was a tricky problem- but good practice in thinking about things declaratively.
Common SQL Query Paradigms

Nesting + EXISTS / ANY / ALL

```
SELECT sid, p3.precip
FROM (  
    SELECT sid, precip
    FROM precipitation AS p1
    WHERE precip > 0 AND NOT EXISTS (  
        SELECT p2.precip
        FROM precipitation AS p2
        WHERE p2.sid = p1.sid
        AND p2.precip > 0
        AND p2.precip < p1.precip)  
    ) AS p3
WHERE NOT EXISTS (  
    SELECT p4.precip
    FROM precipitation AS p4
    WHERE p4.precip - 400 > p3.precip);
```

More complex, but again just think about order!
Graph traversal & recursion

For fixed-length paths

```
SELECT A, B, d
FROM edges
UNION
SELECT e1.A, e2.B,
     e1.d + e2.d AS d
FROM edges e1, edges e2
WHERE e1.B = e2.A
     AND e2.B <> e1.A
UNION
SELECT e1.A, e3.B,
     e1.d + e2.d + e3.d AS d
FROM edges e1, edges e2, edges e3
WHERE e1.B = e2.A
     AND e2.B = e3.A
     AND e2.B <> e1.A
     AND e3.B <> e2.A
     AND e3.B <> e1.A
```
Graph traversal & recursion

For variable-length paths on trees

```
WITH RECURSIVE
paths(a, b, b_prev, d) AS (
  SELECT A, B, A
  FROM edges
  UNION
  FROM paths p, edges e
  WHERE p.b = e.A
    AND e.B <> p.b_prev
)
SELECT a, b, MAX(d)
FROM paths;
```