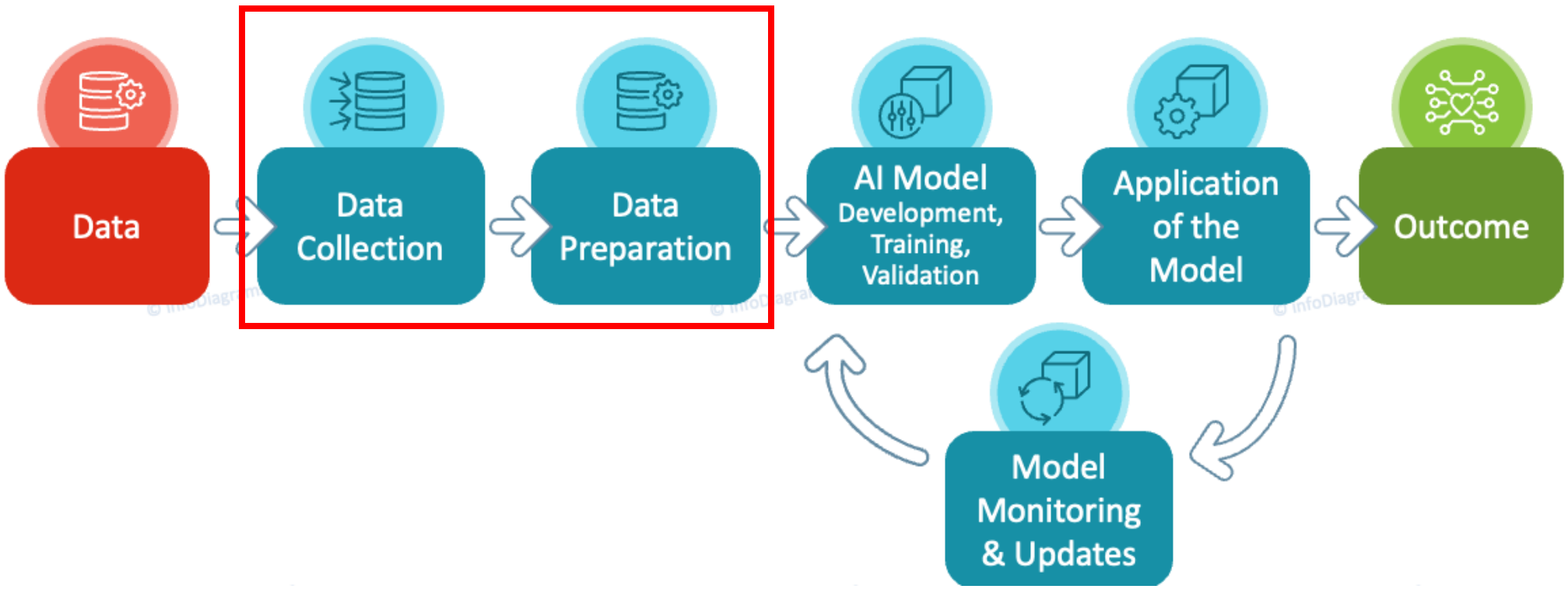
IMEN891K/IDSC501 Data Engineering

7. **SQL I**: Basics

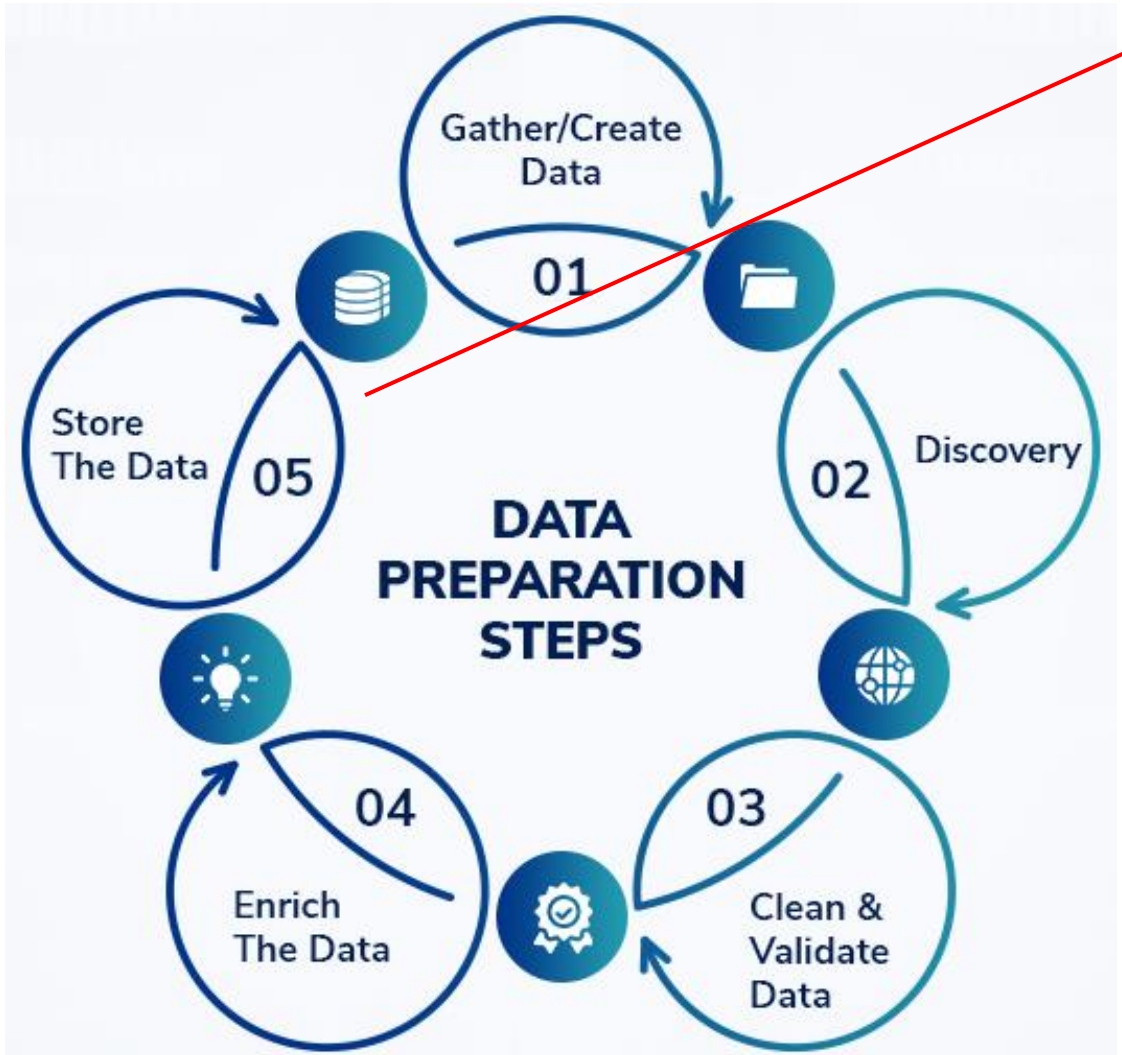
Fall 2025

Instructor: **Sangwoo Mo**

Recap: Course Overview



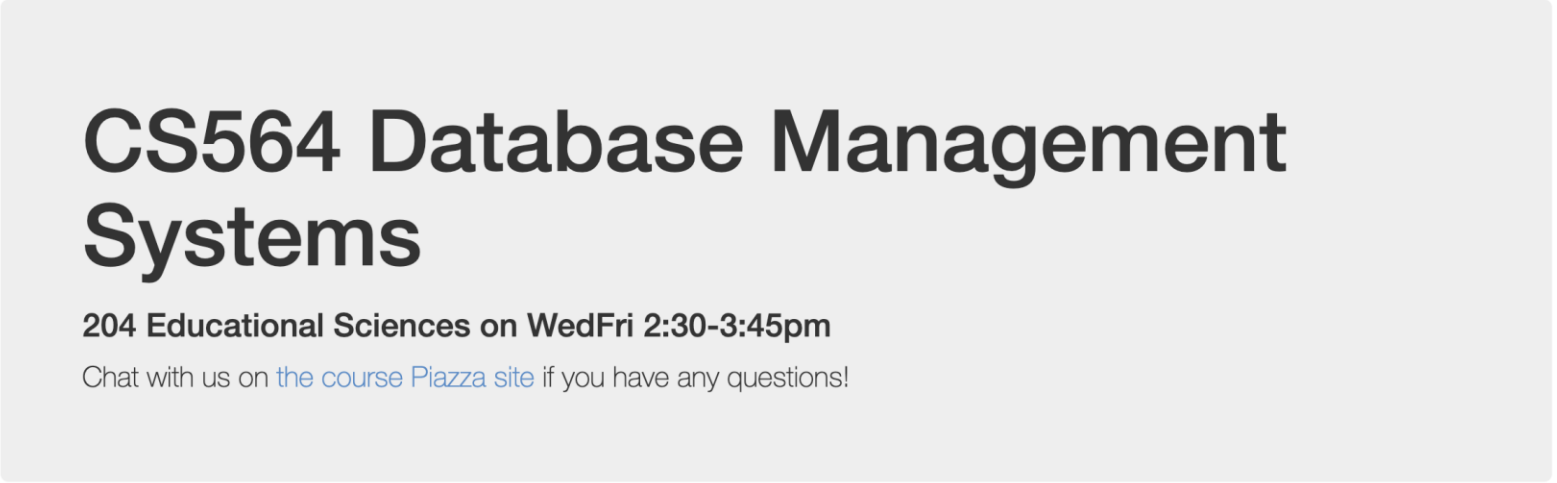
Course Overview



Lecture 7-15

Disclaimer

The SQL part was heavily copy-pasted from Wisconsin CS564



Database Management System (DBMS)

- A large, integrated collection of data

- Models a real-world *enterprise*
  - *Entities* (e.g., Students, Courses)
  - *Relationships* (e.g., Alice is enrolled in CS564)

A Database Management System (DBMS) is a piece of software designed to store and manage databases

Database Management System (DBMS)

- Consider building a course management system (**CMS**):
  - Students
  - Courses
  - Professors

  } *Entities*

  - Who takes what
  - Who teaches what

  } *Relationships*

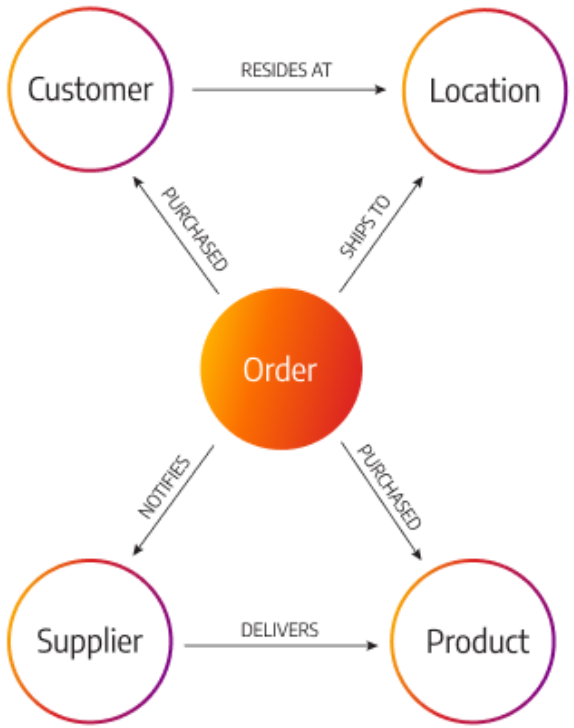
Data Models & Schema

- A **data model** is a collection of concepts for describing data
  - The relational model of data is the most widely used model today
    - Main Concept: the *relation*- essentially, a table

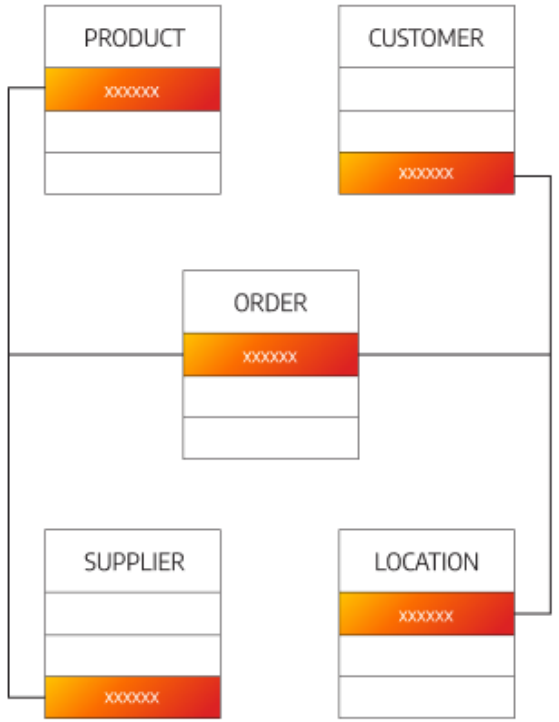
- A **schema** is a description of a particular collection of data, **using the given data model**
  - E.g. every *relation* in a relational data model has a *schema* describing types, etc.

Data Models

GRAPH DB



RELATIONAL DB



Data Models

Criteria	Relational Database	Graph Database
Data Model	Tabular with rows and columns	Graph networks with nodes and edges
Schema	Predefined schema	Flexible, schema-on-read
Relationships	Foreign keys	Nodes linked by edges
Query Language	SQL	Traversal APIs and languages
Strengths	Structured data, transactions	Interconnected data, relationships
Use Cases	OLTP, business applications	Social, recommendations, fraud detection

Schema

- *Logical Schema*
  - Students(sid: *string*, name: *string*, gpa: *float*)
  - Courses(cid: *string*, cname: *string*, credits: *int*)
  - Enrolled(sid: *string,* cid*: string,* grade*: string*)

Relations

sid	Name	Gpa
101	Bob	3.2
123	Mary	3.8

Students

cid	cname	credits
564	564-2	4
308	417	2

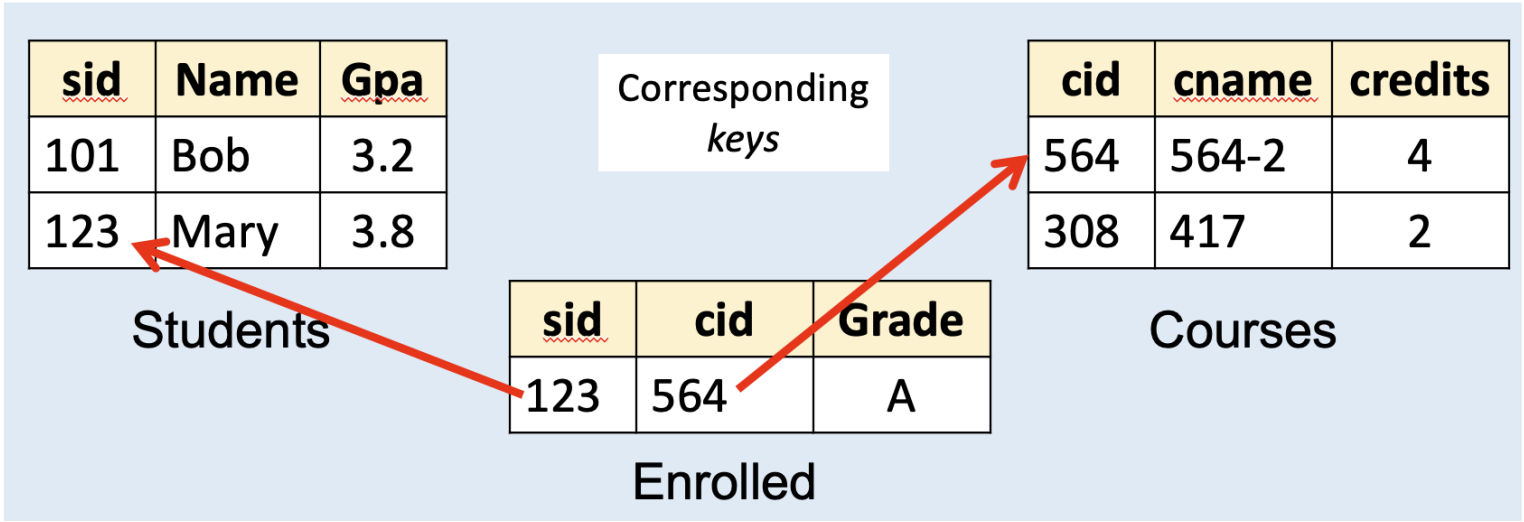
Courses

sid	cid	Grade
123	564	A

Enrolled

Schema

- *Logical Schema*
  - Students(sid: *string*, name: *string*, gpa: *float*)
  - Courses(cid: *string*, cname: *string*, credits: *int*)
  - Enrolled(sid: *string,* cid*: string,* grade*: string*)



Schema

- *Physical Schema*: describes data layout
  - Relations as unordered files
  - Some data in sorted order (index)



Administrators

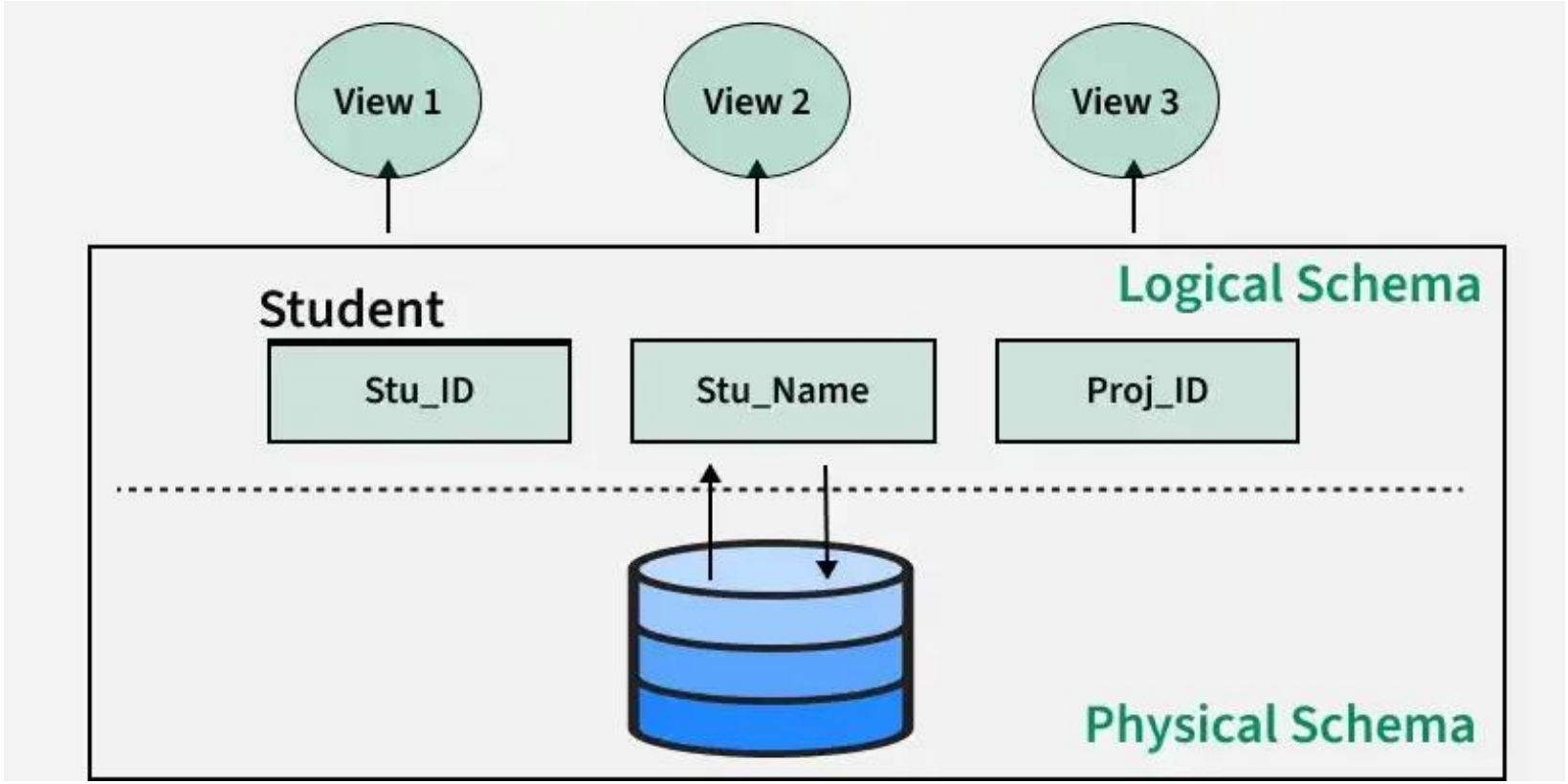
- *Logical Schema:* Previous slide



Applications

- *External Schema*: (Views)
  - Course_info(cid: *string*, enrollment: *integer*)
  - Derived from other tables

Schema


View 1
View 2
View 3
Logical Schema
Student
Stu_ID
Stu_Name
Proj_ID
Physical Schema

Data Independence

Concept: Applications do not need to worry about *how the data is structured and stored*

Logical data independence:
protection from changes in the *logical structure of the data*

I.e. should not need to ask: can we add a new entity or attribute without rewriting the application?

Physical data independence:
protection from *physical layout changes*

I.e. should not need to ask: which disks are the data stored on? Is the data indexed?

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

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

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.

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!

Transactions

- A key concept is the **transaction (TXN)**: an **atomic** sequence of db actions (reads/writes)
  - If a user cancels a TXN, it should be as if nothing happened!

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

- Transactions leave the DB in a **consistent** state
  - Users may write integrity constraints, e.g., 'each course is assigned to exactly one room'

Consistency: An action results in a state which conforms to all integrity constraints

Concurrency & Locking

- 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…

Atomicity & Logging

- 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…

Structured Query Language (SQL)

- SQL is a standard language for querying and manipulating data
- SQL is a **very high-level** programming language
  - This works because it is optimized well!
- Many standards out there:
  - ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3), ….
  - Vendors support various subsets

SQL stands for
Structured Query Language

Structured Query Language (SQL)

- Data Definition Language (DDL)
  - Define relational *schemata*
  - Create/alter/delete tables and their attributes

- Data Manipulation Language (DML)
  - Insert/delete/modify tuples in tables
  - Query one or more tables – discussed next!

SQL Dialects



Python + SQL

기존



ipython-sql 사용



Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

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

Let's break this definition down

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

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

List: [1, 1, 2, 3]
Set: {1, 2, 3}
Multiset: {1, 1, 2, 3}

i.e. no *next()*, etc. methods!

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

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

*Attributes must have an **atomic** type in standard SQL, i.e. not a list, set, etc.*

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

*Also referred to sometimes as a **record***

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

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

The number of tuples is the cardinality of the relation

The number of attributes is the arity of the relation

Data Types in SQL

- Atomic types:
  - Characters: CHAR(20), VARCHAR(50)
  - Numbers: INT, BIGINT, SMALLINT, FLOAT
  - Others: MONEY, DATETIME, …

- Every attribute must have an atomic type
  - Hence tables are flat

Table Schemas

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

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

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

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

Key Constraints

A key is a minimal subset of attributes that acts as a unique identifier for tuples in a relation

- A key is an implicit constraint on which tuples can be in the relation
  - i.e. if two tuples agree on the values of the key, then they must be the same tuple!

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

1. Which would you select as a key?
2. Is a key always guaranteed to exist?
3. Can we have more than one key?

NULL and NOT NULL

- To say “don’t know the value” we use NULL
  - NULL has (sometimes painful) semantics, more details later

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

sid	name	gpa
123	Bob	3.9
143	Jim	NULL

Say, Jim just enrolled in his first class.

In SQL, we may constrain a column to be NOT NULL, e.g., “name” in this table

General Constraints

- We can actually specify arbitrary assertions
  - E.g. “*There cannot be 25 people in the DB class*”

- In practice, we don’t specify many such constraints. Why?
  - Performance!

Whenever we do something ugly (or avoid doing something convenient) it’s for the sake of performance

Practice: Loading Tables in SQL

When we want to work with SQL easily in an Ipython notebook, we'll load the ipython-sql extension.

If you do not have this extension installed please type

```
pip install ipython-sql
```

```
%load_ext sql
```

Practice: Loading Tables in SQL

Next, we'll load an SQLite database stored as a file as follows:

NOTE: We load a file below (here, "dataset_1.db")

```
%sql sqlite:///dataset_1.db
```

Lets look at one table from the US National Oceanic and Atmospheric Administration (NOAA) Rainfall dataset- `precipitation_full` - having the following schema:

> - `state_code`
> - `station_id`
> - `year`
> - `month`
> - `day`
> - `hour`
> - `precipitation`
> - `flag_1`
> - `flag_2`

Practice: Loading Tables in SQL

Now, however, let's see how to view the **schema** of existing tables on your own; there are several ways, including but not limited to:

- DESCRIBE tablename
- SHOW CREATE TABLE tablename
- SHOW COLUMNS tablename

Unfortunately, support for these varies widely between DBMSs, and is also limited by our IPython interface (for example sqlite, which we are using, does not support the above; it does have a `.schema tablename` command, however this doesn't work in IPython notebooks...)

Practice: Creating Tables in SQL

```
%%sql
CREATE TABLE rain_corps_assignments(
    student_id INT PRIMARY KEY,
    name VARCHAR(1000),
    station_id INT,
    state_code INT,
    start_hour INT,
    end_hour INT,
    holding_cup BOOLEAN,
    cup_size FLOAT,
    off_day INT
)
```

SQL Query

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

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

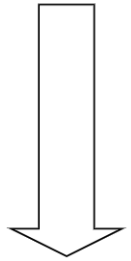
Call this a SFW query.

SQL Query: Selection

Selection is the operation of filtering a relation's tuples on some condition

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT *
FROM   Product
WHERE  Category = 'Gadgets'
```



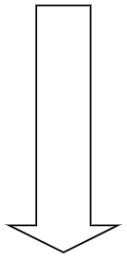
PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks

SQL Query: Projection

Projection is the operation of producing an output table with tuples that have a subset of their prior attributes

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT Pname, Price, Manufacturer
FROM   Product
WHERE  Category = 'Gadgets'
```



PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks

SQL Query: Projection

Input schema

Product(PName, Price, Category, Manfacturer)

```
SELECT Pname, Price, Manufacturer
FROM   Product
WHERE  Category = 'Gadgets'
```



Output schema

Answer(PName, Price, Manfacturer)

A Few Details

- SQL **commands** are case insensitive:
  - Same: SELECT, Select, select
  - Same: Product, product
- **Values** are **not:**
  - Different: 'Seattle', 'seattle'
- Use single quotes for constants:
  - 'abc' - yes
  - "abc" - no

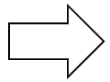
LIKE: Simple String Pattern Matching

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

- s **LIKE** p: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - _ = any single character

DISTINCT: Eliminating Duplicates

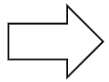
```
SELECT DISTINCT Category
FROM   Product
```



Category
Gadgets
Photography
Household

Versus

```
SELECT Category
FROM   Product
```



Category
Gadgets
Gadgets
Photography
Household

ORDER BY: Sorting the Results

```
SELECT   PName, Price, Manufacturer
FROM     Product
WHERE    Category='gizmo' AND Price > 50
ORDER BY Price, PName
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

Practice: Basic Queries

```
%%sql drop table if exists product;
create table product(
       pname        varchar primary key, -- name of the product
       price        money,               -- price of the product
       category     varchar,             -- category
       manufacturer varchar NOT NULL     -- manufacturer
);
insert into product values('Gizmo', 19.99, 'Gadgets', 'GizmoWorks');
insert into product values('PowerGizmo', 29.99, 'Gadgets', 'GizmoWorks');
insert into product values('MultiTouch', 203.99, 'Household', 'Hitachi');
insert into product values('SingleTouch', 149.99, 'Photography', 'Canon');
```

```
Done.
Done.
1 rows affected.
1 rows affected.
1 rows affected.
1 rows affected.
[]
```

Practice: Basic Queries

Let's look at the products first:

```
%sql select * from product;
```

Done.

pname	price	category	manufacturer
Gizmo	19.99	Gadgets	GizmoWorks
PowerGizmo	29.99	Gadgets	GizmoWorks
MultiTouch	203.99	Household	Hitachi
SingleTouch	149.99	Photography	Canon

Practice: Basic Queries

This is the simple SELECT-FROM-WHERE (SFW) block. Let's see some examples!

```
%%sql
SELECT * from Product
WHERE category='Gadgets' and price > 20.0;
```

```
Done.
```

pname	price	category	manufacturer
PowerGizmo	29.99	Gadgets	GizmoWorks

Practice: Basic Queries

Let's give an example of *projection*, i.e., we only retain some attributes from the query.

```
%%sql
SELECT Pname, Price, Manufacturer
FROM Product;
```

```
Done.
```

pname	price	manufacturer
Gizmo	19.99	GizmoWorks
PowerGizmo	29.99	GizmoWorks
SingleTouch	149.99	Canon
MultiTouch	203.99	Hitachi

- The output is *still* a table, and its schema is

> Answer(pname, price, manufacturer)

Practice: Basic Queries

- Of course, we can combine selection and projection.

```
%%sql
SELECT Pname, Price, Manufacturer
FROM Product
WHERE category='Gadgets';
```

Done.

pname	price	manufacturer
Gizmo	19.99	GizmoWorks
PowerGizmo	29.99	GizmoWorks

Practice: Basic Queries

The output of a query on a table is again a table

- This is because the query language is *compositional*
- The output of a query really is a table!
- look at this crazy query, what does it ask for?

```
%%sql
SELECT * FROM Product;

SELECT
    p.manufacturer, p.pname, p.price
FROM
    (SELECT distinct p0.Manufacturer
     FROM Product p0
     WHERE p0.price < 20.00) cp, -- this is a nested query!
    Product p
WHERE
    p.manufacturer = cp.manufacturer and p.price > 20.00
```

```
Done.
Done.
```

manufacturer	pname	price
GizmoWorks	PowerGizmo	29.99

Practice: Basic Queries

LIKE

The LIKE operator is to search strings, perhaps with wildcards. Format is:

> SELECT * FROM Products WHERE pname like '%gizmo%'

- % matches any number of characters
- _ matches one character
- The like operator is case sensitive

```
%%sql
SELECT *  FROM product
where pname LIKE '%Gizmo%'
```

```
Done.
```

pname	price	category	manufacturer
Gizmo	19.99	Gadgets	GizmoWorks
PowerGizmo	29.99	Gadgets	GizmoWorks

Practice: Basic Queries

Eliminating Duplicates

- duplicates can sometimes be unwelcome or suprising.
  - Recall tables are *multisets*!

```
%sql SELECT category from product;
```

```
Done.
```

category
Gadgets
Gadgets
Photography
Household

```
%%sql
-- easy to remove duplicates, use the distinct keyword
SELECT DISTINCT category from product;
```

```
Done.
```

category
Gadgets
Photography
Household

Practice: Basic Queries

Ordering the results

- Sometimes you want the results ordered, let's see some examples!

```
%%sql
-- sometimes we want to order the results.
-- order by is ascending by default!
SELECT   pname, price, manufacturer
FROM     Product
WHERE    price > 50
ORDER BY  price, pname
```

Done.

pname	price	manufacturer
SingleTouch	149.99	Canon
MultiTouch	203.99	Hitachi

Practice: Basic Queries

```
%%sql
-- sometimes we want to order the results.
-- can order like so, each component individually
SELECT   price, manufacturer
FROM     Product
-- the order is "dictionary order" in the clause.
ORDER BY   manufacturer ASC, price DESC
```

Done.

price	manufacturer
149.99	Canon
29.99	GizmoWorks
19.99	GizmoWorks
203.99	Hitachi