Week 4: Introduction to Database Systems

AF3214 Python Programming for Accounting and Finance

Vincent Y. Zhuang, Ph.D.
vincent.zhuang@polyu.edu.hk

School of Accounting and Finance
The Hong Kong Polytechnic University

R508, 8:30 am – 11:20 am, Tuesdays, Semester 2, AY 2024-25

Today's Agenda

- Database Systems Background
- Relational Model
- Relational Algebra
- Alternative Data Models (MongoDB, JSON, Vector DB)
- Modern SQL:
  - Aggregations + Group By
  - String Operations
  - Output Control + Redirection
  - Window Functions
  - Nested Queries
  - Lateral Joins
  - Common Table Expressions

What are databases?

Databases

- Organized collection of inter-related data that models some aspect of the real-world.
- Databases are the core component of most computer applications.
- Encounter Database throughout your life

feed some input → manipulate with data → **produce some output**

↑

Databases

file system, website, LLM, bank, smartphone(SQLite, DBMS)…

Databases Examples

Explain the various parts of the relational model and database management systems as we go along

- Create a database that models a digital music store (e.g., Spotify) to keep track of artists and albums.
- Information we need to keep track of in our store:
  - → Artists who putting out music
  - → The albums those artists released

Databases Examples - Flat File Strawman

A really simple implementation of the database could just be a bunch of files on disk

Store our database as CSV (comma-separated value), JSON, YAML files that we manage ourselves in application code.

→ Use a separate file per entity.
→ The application must parse the files each time they want to read/update records.

Artist(name, year, country)

```
"Wu-Tang Clan",1992,"USA"
"Notorious BIG",1992,"USA"
"GZA",1990,"USA"
```

Album(name, artist, year)

```
"Enter the Wu-Tang" , "Wu-Tang Clan",1993
"St.Ides Mix Tape" , "Wu-Tang Clan",1994
"Liquid Swords" , "GZA",1990
```

Insert a new record -> open the file and pen to the end of it
Answer a query -> scan through it until we find the answer

Databases Examples - Flat File Strawman

Example: Get the year that GZA went solo.

Artist(name, year, country)

```
"Wu-Tang Clan",1992,"USA"
"Notorious BIG",1992,"USA"
"GZA",1990,"USA"
```

Python Code

```
for line in file.readlines():
    record = parse(line)
    if record[0] == "GZA":
        print(int(record[1]))
```

Flat Files: Data Integrity

- How do we ensure that the artist is the same for each album entry?
- What if somebody overwrites the album year with an invalid string?
- What if there are multiple artists on an album?
- What happens if we delete an artist that has albums?

Flat Files: Implementation

- How do you find a particular record?
- What if we now want to create a new application that uses the same database? What if that application is running on a different machine?
- What if two threads try to write to the same file at the same time?

Flat Files: Durability

- What if the machine crashes while our program is updating a record?

- What if we want to replicate the database on multiple machines for high availability?

Database Management System (DBMS)

- A database management system (DBMS) is software that allows applications to store and analyze information in a database.
- A general-purpose DBMS supports the definition, creation, querying, update, and administration of databases in accordance with some data model (a high level representation of what is in a database).
- support any arbitrary schema; allow any arbitrary queries; and allow any kind of transformation.

Data Models

- A <u>data model</u> is a collection of concepts for describing the data in a database (a high level abstraction to represent/define a collection of data in a database).

- A <u>schema</u> is a description of a particular collection of data, using a given data model.
  - This defines the structure of data to store in a database: tables, collections of data, attributes, names, data types…
  - Otherwise, w/o schema, only random bits w/ no meaning, because schema will provide a structure to the data so that we can write queries against it.

Data Models

- Relational ←**Most DBMSs**
- Key/Value
- Graph
- Document / JSON / XML / Object
- Wide-Column / Column-family
- Array (Vector, Matrix, Tensor)
- Hierarchical
- Network
- Semantic
- Entity-Relationship

Data Models

- Relational
- $\boxed{\text{Key/Value}}$ ←Simple Apps / Caching, RocksDB, LevelDB
- Graph
- Document / JSON / XML / Object
- Wide-Column / Column-family
- Array (Vector, Matrix, Tensor)
- Hierarchical
- Network
- Semantic
- Entity-Relationship

Data Models

Relational

Key/Value

Graph

Document / JSON / XML / Object

Wide-Column / Column-family

←**NoSQL**

Array (Vector, Matrix, Tensor)

Hierarchical

Network

Semantic

Entity-Relationship

Data Models

- Relational
- Key/Value
- Graph
- Document / JSON / XML / Object
- Wide-Column / Column-family
- Array (Vector, Matrix, Tensor) ← **ML/Science**
- Hierarchical
- Network
- Semantic
- Entity-Relationship

Data Models

Relational

Key/Value

Graph

Document / JSON / XML / Object

Wide-Column / Column-family

Array (Vector, Matrix, Tensor)

Hierarchical

Network

Semantic

Entity-Relationship

70's 80's, Obsolete / Legacy / Rare

Data Models

- Relational ←this course
- Key/Value
- Graph
- Document / JSON / XML / Object
- Wide-Column / Column-family
- Array (Vector, Matrix, Tensor)
- Hierarchical
- Network
- Semantic
- Entity-Relationship

Relational Model

The **relational model** defines a database abstraction based on relations to avoid the cost of having to maintain the overhead. (or to maintain the implicit knowledge of what the data looks like in your application code)

Key ideas:

→ Store database in simple data structures (relations or tables), don't expose that information to the application code.

→ Physical storage left up to the DBMS implementation.
DB system to write down the bits of tables is up to ↑
(DB stored on disc vs DB stored on memory, having things defined in relations and interact through SQL)

→ Access data through high-level language, DBMS figures out best execution strategy.
(avoid writing procedural code like a low level for Loops to iterate one record at a time)

Relational Model

Three components of relational mode:

Structure: The definition of the database’s relations/tables and their contents independent of their physical representation (next slides).

Integrity: Ensure the database’s contents satisfy constraints. (definitions of what the data is allowed to look like in accordance to whatever you're trying to model in the real world: e.g., -20 ages)

Manipulation: Programming interface (e.g., API) for accessing and modifying a database's contents. (allow us to access and modify the contents in our database -> relational algebra)

Relational Model

Three components of relational mode:

Structure: The definition of the database's relations/tables and their contents independent of their physical representation (next slides).

Integrity: Ensure the database's contents satisfy constraints. (definitions of what the data is allowed to look like in accordance to whatever you're trying to model in the real world: e.g., -20 ages)

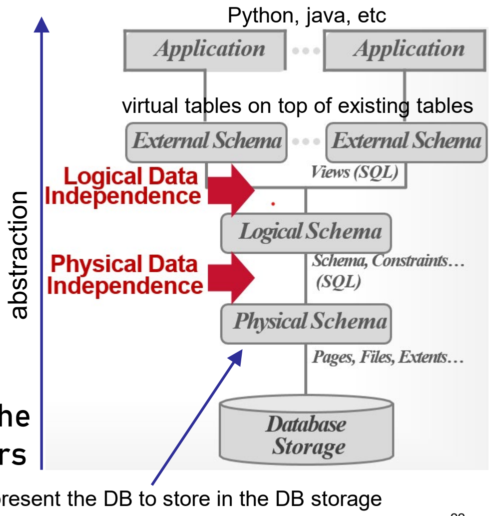
Manipulation: Programming interface (e.g., API) for accessing and modifying a database's contents. (allow us to access and modify the contents in our database -> relational algebra)

Seems obvious now but in the 1970s this was considered groundbreaking

Data Independence

The core idea of data independence is NOT to expose the physical layout of data on any storage device.

- Isolate the user/application from low-level data representation.
- The user only worries about high-level application logic.
- DBMS optimizes the layout according to operating environment, database contents, and workload.
- DBMS can then re-optimize the database if/when these factors changes



Relational Model

A relation is an unordered set of data that contain the relationship of attributes that represent some entities in the world.

A tuple (or record) is a set of attribute values (aka its domain) in the relation.

→ Values are (normally) atomic/scalar.

→ The special value NULL is a member of every domain (if allowed).

Artist(name, year, country)

name	year	country
Wu-Tang Clan	1992	USA
Notorious BIG	1992	USA
GZA	1990	USA

n-ary Relation
=
Table with *n* columns

Relational Model

A relation's **primary key** uniquely identifies a single tuple.
(set of attributes that uniquely represent one entity within our relation)

Some DBMSs automatically create an internal primary key if a table does not define one.

DBMS can auto-generation unique primary keys via an identity column:
→ IDENTITY (SQL Standard)
→ SEQUENCE (PostgreSQL / Oracle)
→ AUTO_INCREMENT (MySQL)

Artist(name, year, country)

name	year	country
Wu-Tang Clan	1992	USA
Notorious BIG	1992	USA
GZA	1990	USA

Relational Model

A relation's **primary key** uniquely identifies a single tuple.
(set of attributes that uniquely represent one entity within our relation)

Some DBMSs automatically create an internal primary key if a table does not define one.

DBMS can auto-generation unique primary keys via an identity column:
→ IDENTITY (SQL Standard)
→ SEQUENCE (PostgreSQL / Oracle)
→ AUTO_INCREMENT (MySQL)

Artist(<u>id</u>, name, year, country)

Synthetic ID

id	name	year	country
101	Wu-Tang Clan	1992	USA
102	Notorious BIG	1992	USA
103	GZA	1990	USA

Relational Model

A foreign key specifies that an attribute from one relation maps to a tuple in another relation.

ArtistAlbum(artist_id, album_id)

artist_id	album_id
101	11
101	22
103	22
102	22

Artist(id, name, year, country)

id	name	year	country
101	Wu-Tang Clan	1992	USA
102	Notorious BIG	1992	USA
103	GZA	1990	USA

Album(id, name, artists, year)

id	name	artists	year
11	Enter the Wu-Tang	101	1993
22	St.Ides Mix Tape	???	1994
33	Liquid Swords	103	1995

Can handle constraints

Relational Model

- ✓ User-defined conditions that must hold for *any* instance of the database.
  - ❑ Can validate data within a single tuple or across entire relation(s).
  - ❑ DBMS prevents modifications that violate any constraint.
- ✓ Unique key and referential (foreign key) constraints are the most common.
- ✓ SQL:92 supports global asserts but these are rarely used (too slow).

```
CREATE TABLE Artist (
  name VARCHAR NOT NULL,
  year INT,
  country CHAR(60),
  CHECK (year > 1900)
  ASSERTION myAssert
```

```
CREATE
  ASSERTION myAssert
  CHECK ( <SQL> ) ;
```

Artist(id, name, year, country)

id	name	year	country
101	Wu-Tang Clan	1992	USA
102	Notorious BIG	1992	USA
103	GZA	1990	USA

Data Manipulation Languages (Dml)

- ✓ The API that a DBMS exposes to applications to store and retrieve information from a database.

DML is the way to interact with the database.
Two category.

Procedural:

→ The query specifies the (high-level) strategy to find the desired result based on sets / bags.

defining steps in order for DB system to execute

← **Relational Algebra**

Non-Procedural (Declarative):

→ The query specifies only what data is wanted and not how to find it.

higher level form of defining operations over the relational model, e.g., specifying answers we want

← **Relational Calculus**

Relational Algebra

✓ Fundamental operations to retrieve and manipulate tuples in a relation.

→ Based on set algebra (unordered lists with no duplicates).

✓ Each operator takes one or more relations as its inputs and outputs a new relation.

→ We can "chain" operators together to create more complex operations.

Seven Primitives

σ	Select
π	Projection
$\cup$	Union
$\cap$	Intersection
$-$	Difference
$\times$	Product
$\bowtie$	Join

Idea: use them to construct any possible query we want to have on a relational database.

Relational Algebra

SELECT: **Syntax:** $\sigma_{predicate}(R)$

first order predicate logic on how to identify the tuples that I want for a given relation

Choose a subset of the tuples from a relation that satisfies a selection predicate.

→ Predicate acts as a filter to retain only tuples that fulfill its qualifying requirement.

→ Can combine multiple predicates using conjunctions / disjunctions.

$R(a\_id, b\_id)$

a_id	b_id
a1	101
a2	102
a2	103
a3	104

$\sigma_{a\_id='a2'}(R)$

a_id	b_id
a2	102
a2	103

$\sigma_{a\_id='a2' \wedge b\_id>102}(R)$

a_id	b_id
a2	103

```
SELECT * FROM R
WHERE a_id='a2' AND b_id>102;
```

where clause

Relational Algebra

PROJECTION: Syntax: $\Pi_{A1,A2,\ldots,An}(R)$

defining what the output should look like in terms of what attributes we care about from our input that we want to produce as the output, as well as how to manipulate those attributes in any given way

Generate a relation with tuples that contains only the specified attributes.
→ Rearrange attributes' ordering.
→ Remove unwanted attributes.
→ Manipulate values to create derived attributes.

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a2	103
a3	104

$\Pi_{b\_id-100,a\_id}(\sigma_{a\_id='a2'}(R))$

b_id-100	a_id
2	a2
3	a2

```
SELECT b_id-100, a_id
  FROM R WHERE a_id = 'a2';
```

UNION: Syntax: (R ∪ S)

takes all the attributes from one side and all the on the other side and combine them together;
only works if both relations have the same attributes.

Generate a relation that contains all tuples that appear in either only one or both input relations.

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id)

a_id	b_id
a3	103
a4	104
a5	105

(R ∪ S)

a_id	b_id
a1	101
a2	102
a3	103
a4	104
a5	105

```
(SELECT * FROM R)
UNION
(SELECT * FROM S);
```

union operator

INTERSECTION: Syntax: (R ∩ S)

Generate a relation that contains only the tuples that appear in both of the input relations.

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id)

a_id	b_id
a3	103
a4	104
a5	105

(R ∩ S)

a_id	b_id
a3	103

intersect operator

```
(SELECT * FROM R)
INTERSECT
(SELECT * FROM S);
```

DIFFERENCE: **Syntax: (R – S)**

finding all the tuples that appear in the first relation but not in the second relation

Generate a relation that contains only the tuples that appear in the first and not the second of the input relations.

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id)

a_id	b_id
a3	103
a4	104
a5	105

(R – S)

a_id	b_id
a1	101
a2	102

except operator

```
(SELECT * FROM R)
EXCEPT
(SELECT * FROM S);
```

Relational Algebra

PRODUCT: **Syntax: (R × S)**

Cartesian product, taking all tuples in two relations and figure out all possible combinations

Generate a relation that contains all possible combinations of tuples from the input relations.

cross join operator

```
SELECT * FROM R CROSS JOIN S;
```

```
SELECT * FROM R, S;
```

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id)

a_id	b_id
a3	103
a4	104
a5	105

(R × S)

R.a_id	R.b_id	S.a_id	S.b_id
a1	101	a3	103
a1	101	a4	104
a1	101	a5	105
a2	102	a3	103
a2	102	a4	104
a2	102	a5	105
a3	103	a3	103
a3	103	a4	104
a3	103	a5	105

JOIN: Syntax: $(R \bowtie S)$

Generate a relation that contains all tuples that are a combination of two tuples (one from each input relation) with a common value(s) for one or more attributes.

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id,val)

a_id	b_id	val
a3	103	XXX
a4	104	YYY
a5	105	ZZZ

$(R \bowtie S)$

a_id	b_id	val
a3	103	XXX

R.a_id	R.b_id	S.a_id	S.b_id	S.val
a3	103	a3	103	XXX

Relational Algebra

JOIN: Syntax: $(R \bowtie S)$

Generate a relation that contains all tuples that are a combination of two tuples (one from each input relation) with a common value(s) for one or more attributes.

```
SELECT * FROM R NATURAL JOIN S;
```

```
SELECT * FROM R JOIN S USING (a_id, b_id);
```

```
SELECT * FROM R JOIN S
    ON R.a_id = S.a_id AND R.b_id = S.b_id;
```

R(a_id,b_id)

a_id	b_id
a1	101
a2	102
a3	103

S(a_id,b_id,val)

a_id	b_id	val
a3	103	XXX
a4	104	YYY
a5	105	ZZZ

$(R \bowtie S)$

a_id	b_id	val
a3	103	XXX

R.a_id	R.b_id	~~S.a_id~~	~~S.b_id~~	S.val
a3	103	~~a3~~	~~103~~	XXX

Relational Algebra

EXTRA OPERATOR:

Rename (ρ)
Assignment ($R \leftarrow S$)
Duplicate Elimination (δ)
Aggregation (γ)
Sorting (τ)
Division ($R \div S$)

Observation

Relational algebra defines an ordering of the high-level steps of how to compute a query.
→ Example: $\sigma_{b\_id=102}(R \bowtie S)$ vs. $(R \bowtie (\sigma_{b\_id=102}(S))$

A better approach is to state the high-level answer that you want the DBMS to compute.
→ Example: Retrieve the joined tuples from R and S where b_id equals 102.

QUERIES

The relational model is independent of any query language implementation.

SQL is the *de facto* standard (many dialects).

```
for line in file.readlines():
    record = parse(line)
    if record[0] == "GZA":
        print(int(record[1]))
```

```
SELECT year FROM artists
  Where NAME = 'GZA';
```

Data Models

- Relational ←**this course**
- Key/Value
- Graph
- Document / JSON / XML / Object
- Wide-Column / Column-family
- Array (Vector, Matrix, Tensor)
- Hierarchical
- Network
- Semantic
- Entity-Relationship

Data Models

- Relational
- Key/Value
- Graph
- Document / JSON / XML / Object ←**alternatives**
- Wide-Column / Column-family
- Array (Vector, Matrix, Tensor) ←**new hotness**
- Hierarchical
- Network
- Semantic
- Entity-Relationship

Document Data Models

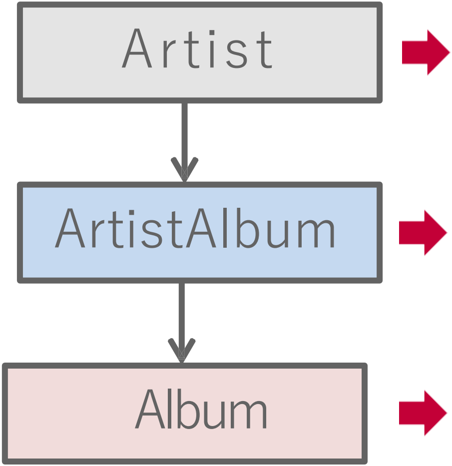
A collection of record documents containing a hierarchy of key/value pairs.

→ A field's value can be either a scalar type, an array of values, or another document.

→ Modern implementations use JSON. Older systems use XML or custom object representations.

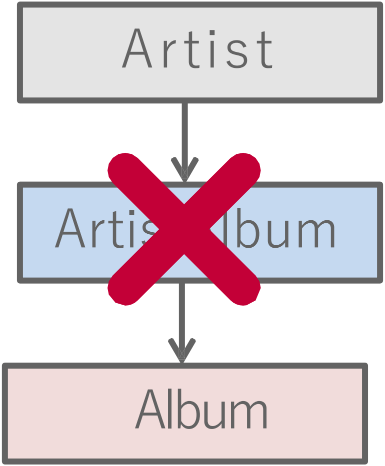
Avoid "relational-object impedance mismatch" by tightly coupling objects and database, which is cumbersome.

Document Data Models



$$R_1(id, \ldots) \bowtie R_2(artist\_id, album\_id) \bowtie R_3(id, \ldots)$$

Document Data Models



$R_1(id, \ldots)$

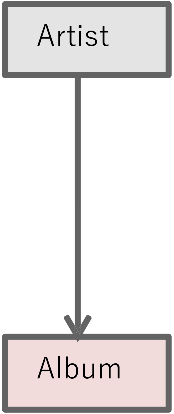
$\bowtie$

$R_2(artist\_id, album\_id)$

$\bowtie$

$R_3(id, \ldots)$

Document Data Models



Application Code

```
class Artist{
  int id;
  String name;
  int year;
  Album albums[];
}
class Album{
   int id;
   String name;
   int year;
}
```

```
{
 "name": "GZA",
 "year": 1990,
 "albums": [
  {
   "name": "Liquid Swords",
   "year": 1995
  },
  {
   "name": "Beneath the Surface",
   "year": 1999
  }
 ]
}
```

Vector Data Models

One-dimensional arrays used for nearest-neighbor search (exact or approximate).
→ Used for semantic search on embeddings generated by ML-trained transformer models (think ChatGPT).
→ Native integration with modern ML tools and APIs (e.g., LangChain, OpenAI).

At their core, these systems use specialized indexes to perform NN searches quickly.

Pretty all relational databases nowadays support Vector indexes.

Pinecone Weaviate milvus qdrant
marqo CloseVector VectorDB turbopuffer <(°O°)>

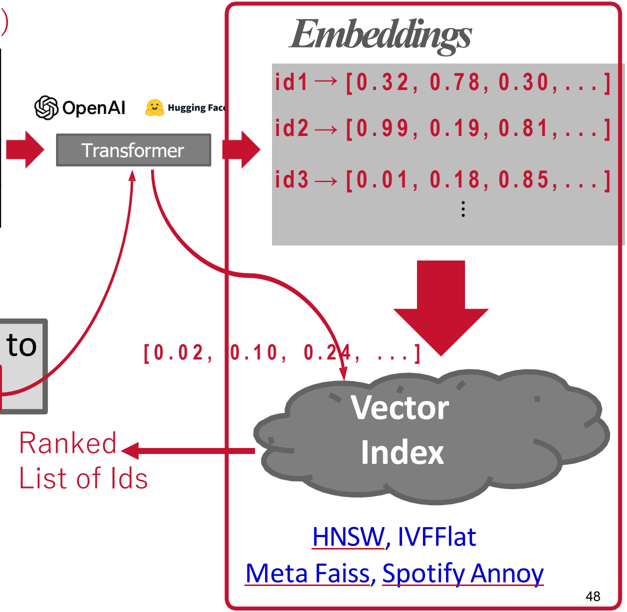
Vector Data Models

Album(<u>id</u>, name, year)

id	name	year
11	Enter the Wu-Tang	1993
22	St.Ides Mix Tape	1994
33	Liquid Swords	1995

Query

Find albums similar to "Liquid Swords"



Quick Summary

MongoDB RAVENDB amazon DynamoDB Firebase
CouchDB relax Couchbase MarkLogic fauna AEROSPIKE

- Databases are ubiquitous.
- Relational algebra defines the primitives for processing queries on a relational database.
- Relational algebra is the building blocks that will allow us to query and modify a relational database.
- Let’s continue with modern SQL.

Pinecone Weaviate milvus qdrant
marqo CloseVector VectorDB turbopuffer <(°O°)>

Modern SQL

SQL History

- In 1969, Ted Codd at IBM Research devised relational model.
- In 1971, IBM created its first relational query language SQUARE.
- IBM then created "SEQUEL" in 1972 for IBM System R prototype DBMS → Structured English Query Language
- IBM releases commercial SQL-based DBMS:
  → System/38 (1979), SQL/DS (1981), and DB2 (1983)
- ANSI Standard in 1986. ISO in 1987 → Structured Query Language
- Current standard is SQL:2023
  → SQL:2023 → Property Graph Queries, Muti-Dim. Arrays
  → SQL:2016 → JSON, Polymorphic tables
  → SQL:2011 → Temporal DBs, Pipelined DML
  → SQL:2008 → Truncation, Fancy Sorting
  → SQL:2003 → XML, Windows, Sequences, Auto-Gen IDs.
  → SQL:1999→ Regex, Triggers, OO
  → SQL:1992 → The minimum language syntax a system supports SQL is **SQL-92**

.

Example Database

student(sid, name, login, gpa)

sid	name	login	age	gpa
53666	RZA	rza@af	55	4.0
53688	Taylor	swift@af	27	3.9
53655	Tupac	shakur@af	25	3.5

enrolled(sid, cid, grade)

sid	cid	grade
53666	3212	C
53688	3215	A
53688	3426	B
53655	3212	B
53666	3215	C

course(cid, name)

cid	name
3212	Database Systems
3215	Advanced Database Systems
3426	Data Mining
3313	Special Topics in Databases

Let’s model a university.

We'll use these tables as our examples as we go along.

Aggregates

- Compute a single value that's derived from a bag of tuples or multiple tuples.

- Functions return a single value from a bag of tuples:

→ AVG(col)→ Return the average col value.
→ MIN(col)→ Return minimum col value.
→ MAX(col)→ Return maximum col value.
→ SUM(col)→ Return sum of values in col.
→ COUNT(col)→ Return # of values for col.

Aggregates

- Aggregate functions can (almost) only be used in the SELECT output list.

Get # of students with a "@af" login:

```
SELECT  COUNT(login)  AS cnt
   FROM  student  WHERE  login  LIKE  ' %@af'
```

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. Two wildcards often used in conjunction with the LIKE operator:
percent sign % (multiple character) and underscore sign _ (one character)

A percent sign stands for an unknown string of 0 or more characters.

If the percent sign starts the search string, then SQL allows 0 or more character(s) to precede the matching value in the column.

Aggregates

- Aggregate functions can (almost) only be used in the SELECT output list.

Get # of students with a "@af" login:

```
SELECT COUNT(login) AS cnt
```

```
SELECT COUNT(*) AS cnt
   FROM student WHERE login LIKE '%@af'
```

Aggregates

- Aggregate functions can (almost) only be used in the SELECT output list.

Get # of students with a "@af" login:

```
SELECT COUNT(login) AS cnt
```

```
SELECT COUNT(*) AS cnt
```

```
SELECT COUNT(1) AS cnt
  FROM student WHERE login LIKE '%@af'
```

Aggregates

- Aggregate functions can (almost) only be used in the SELECT output list.

Get # of students with a "@af" login:

```
SELECT COUNT(login) AS cnt
SELECT COUNT(*) AS cnt
SELECT COUNT(1) AS cnt
SELECT COUNT(1+1+1) AS cnt
   FROM student WHERE login LIKE '%@af'
```

Multiple Aggregates

- Aggregate functions can (almost) only be used in the SELECT output list.

Get the number of students and their average GPA that have a "@af" login.

```
SELECT AVG(gpa), COUNT(sid)
FROM student WHERE login LIKE '%@af'
```

AVG(gpa)	COUNT(sid)
3.8	3

Aggregates

- Aggregate functions can (almost) only be used in the SELECT output list.

Get the number of students and their average GPA that have a "@af" login.

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e JOIN student AS s
    ON e.sid = s.sid
```

AVG(s.gpa)	e.cid
3.86	???

Aggregates

- Output of other columns outside of an aggregate is undefined.

Get the average GPA of students enrolled in each course

```
SELECT AVG(s.gpa), e.cid
  FROM enrolled AS e JOIN student AS s
    ON e.sid = s.sid
```

AVG(s.gpa)	e.cid
3.86	???

ANY_VALUE: Returns some value of the expression from the group. It is optimized to return the first value

```
SELECT AVG(s.gpa), ANY_VALUE(e.cid),
  FROM enrolled AS e JOIN student AS s
    ON e.sid = s.sid
```

AVG(s.gpa)	e.cid
3.86	3212

Group By

➢ Project tuples into subsets and calculate aggregates against each subset.

```sql
SELECT AVG(s.gpa), e.cid
    FROM enrolled AS e JOIN student AS s
        ON e.sid = s.sid
        GROUP BY e.cid
```

e.sid	s.sid	s.gpa	e.cid
53435	53435	2.25	3215
53439	53439	2.70	3215
56023	56023	2.75	3426
59439	59439	3.90	3426
53961	53961	3.50	3426
58345	58345	1.89	3212

→

AVG(s.gpa)	e.cid
2.46	3215
3.39	3426
1.89	3212

Group By

- Non-aggregated values in SELECT output clause must appear in GROUP BY clause.

  anything that's going to be non-aggregated and produced in the output has to be in the Group By Clause

```
SELECT AVG(s.gpa), e.cid
    FROM enrolled AS e, student AS s
    WHERE e.sid = s.sid
    GROUP BY e.cid
```

Group By

- Non-aggregated values in SELECT output clause must appear in GROUP BY clause.

```
SELECT AVG(s.gpa), e.cid, s.name
    FROM enrolled AS e, student AS s
    WHERE e.sid = s.sid
    GROUP BY e.cid
```

Group By

- Non-aggregated values in SELECT output clause must appear in GROUP BY clause.

```
SELECT AVG(s.gpa), e.cid, s.name
    FROM enrolled AS e, student AS s
    WHERE e.sid = s.sid
    GROUP BY e.cid
```

Group By

- Non-aggregated values in SELECT output clause must appear in GROUP BY clause.

```
SELECT AVG(s.gpa), e.cid, s.name
    FROM enrolled AS e, student AS s
    WHERE e.sid = s.sid
    GROUP BY e.cid, s.name
```

Having

- Filters results based on aggregation computation. Like a WHERE clause for a GROUP BY

```
SELECT AVG(s.gpa) AS avg_gpa, e.cid
    FROM enrolled AS e, student AS s
    WHERE e.sid = s.sid
      AND avg_gpa > 3.9
    GROUP BY e.cid
```

Having

- Filters results based on aggregation computation. Like a WHERE clause for a GROUP BY

```
SELECT AVG(s.gpa) AS avg_gpa, e.cid
    FROM enrolled AS e, student AS s
    WHERE e.sid = s.sid
    GROUP BY e.cid
    HAVING avg_gpa > 3.9;
```

some systems won't let you even reference the Alias output in the projection output in the having Clause. MySQL allows but not Postgres

Having

- Filters results based on aggregation computation. Like a WHERE clause for a GROUP BY

```
SELECT AVG(s.gpa) AS avg_gpa, e.cid
    FROM enrolled AS e, student AS s
    WHERE e.sid = s.sid
    GROUP BY e.cid
    HAVING AVG(s.gpa) > 3.9;
```

AVG(s.gpa)	e.cid
3.75	3422
3.950000	3215
3.900000	3426

→

avg_gpa	e.cid
3.950000	3215

String Operation

	String Case	String Quotes
SQL-92	**Sensitive**	**Single Only**
Postgres	Sensitive	Single Only
MySQL	Insensitive	Single/Double
SQLite	Sensitive	Single/Double
MSSQL	Sensitive	Single Only
Oracle	Sensitive	Single Only

```
WHERE UPPER(name) = UPPER('TuPaC')
```
SQL-92

```
WHERE name = "TuPaC"
```
MySQL

String Operation

- LIKE is used for string matching. String-matching operators
  - '%' Matches any substring (including empty strings).
  - '_' Match any one character

```
SELECT * FROM enrolled AS e
WHERE e.cid LIKE '15-%'
```

```
SELECT * FROM student AS s
WHERE s.login LIKE '%@a_'
```

String Operation

- SQL-92 defines string functions.
  - ✓ Many DBMSs also have their own unique functions
- Can be used in either output and predicates:

```
SELECT SUBSTRING(name, 1, 5) AS abbrv_name
FROM student WHERE sid = 53688
```

```
SELECT * FROM student AS s
WHERE UPPER(s.name) LIKE 'KAN%'
```

String Operation

- SQL standard defines the || operator for concatenating two or more strings together.

SQL-92

```
SELECT name FROM student
WHERE  login = LOWER(name) || '@af'
```

MSSQL

```
SELECT name FROM student
WHERE  login = LOWER(name) + '@af'
```

MySQL

```
SELECT name FROM student
WHERE  login = CONCAT(LOWER(name), '@af')
```

Output Redirection

- Store query results in another table:
  - → Table must not already be defined.
  - → Table will have the same # of columns with the same types as the input.

SQL-92

```
SELECT DISTINCT cid INTO CourseIds
FROM enrolled;
```

Postgres

```
SELECT DISTINCT cid
INTO TEMPORARY CourseIds
FROM  enrolled;
```

MySQL

```
CREATE TABLE CourseIds(
SELECT DISTINCT cid FROM  enrolled);
```

DISTINCT counts the number of unique values in a specified column or expression, excluding duplicates.

Output Redirection

- Insert tuples from query into another table:
  - → Inner SELECT must generate the same columns as the target table.
  - → DBMSs have different options/syntax on what to do with integrity violations (e.g., invalid duplicates).

```
INSERT INTO CourseIds
(SELECT DISTINCT cid FROM enrolled);
```

SQL-92

Output Control

- ORDER BY <column*> [ASC|DESC]
  - Order the output tuples by the values in one or more of their columns.

```
SELECT sid, grade FROM enrolled
WHERE cid = '3215'
ORDER BY grade
```

sid	grade
53123	A
53334	A
53650	B
53666	D

Output Control

- ORDER BY <column*> [ASC|DESC]
  - Order the output tuples by the values in one or more of their columns.

```
SELECT sid, grade FROM enrolled
W
O
```

```
SELECT sid, grade FROM enrolled
WHERE cid = '3215'
ORDER BY 2
```

Output Control

- ❑ ORDER BY <column*> [ASC|DESC]
  - ➢ Order the output tuples by the values in one or more of their columns.

```
SELECT sid, grade FROM enrolled
W
O
```

```
SELECT sid, grade FROM enrolled
WHERE cid = '3215'
ORDER BY 2
```

```
SELECT sid FROM enrolled
WHERE cid = '3215'
ORDER BY grade DESC, sid ASC
```

sid
53123
53334
53650
53666

ORDER BY 1 means sorting values according to first column in the SELECT statement.

Output Control

- FETCH {FIRST|NEXT} <count> ROWS OFFSET <count> ROWS
  - Limit the # of tuples returned in output.
  - Can set an offset to return a “range”

```
SELECT sid, name FROM student
WHERE login LIKE '%@af'
FETCH FIRST 10 ROWS ONLY;
```

```
SELECT sid, name FROM student
WHERE login LIKE '%@af'
OFFSET 10 ROWS
FETCH FIRST 10 ROWS WITH TIES;
```

Window Functions

- Performs a calculation across a set of tuples that are related to the current tuple, without collapsing them into a single output tuple, to support running totals, ranks, and moving averages.
  - Like an aggregation but tuples are not grouped into a single output tuples.

How to "slice" up data
Can also sort tuples

```
SELECT  FUNC-NAME(...) OVER (...)
  FROM  tableName
```

Aggregation Functions
Special Functions

Window Functions

- Aggregation functions:
  - Anything that we discussed earlier
- Special window functions:
  - ROW_NUMBER() → # of the current row
  - RANK() → Order position of the current row.

sid	cid	grade	row_num
53666	3212	C	1
53688	3215	A	2
53688	3426	B	3
53655	3212	B	4
53666	3215	C	5

```
SELECT *, ROW_NUMBER() OVER ( ) AS row_num
  FROM enrolled
```

Window Functions

- Aggregation functions:
  - Anything that we discussed earlier
- Special window functions:
  - ROW_NUMBER()→ # of the current row
  - RANK()→ Order position of the current row.

sid	cid	grade	row_num
53666	3212	C	1
53688	3215	A	2
53688	3426	B	3
53655	3212	B	4
53666	3215	C	5

```
SELECT *, ROW_NUMBER() OVER ( ) AS row_num
   FROM enrolled
```

Window Functions

- The OVER keyword specifies how to group together tuples when computing the window function.
- Use PARTITION BY to specify group.

cid	sid	row_num
3212	53666	1
3212	53655	2
3215	53688	1
3215	53666	2
3426	53688	1

```
SELECT cid, sid,
  ROW_NUMBER() OVER (PARTITION BY cid)
  FROM enrolled
  ORDER BY cid
```

Window Functions

- The OVER keyword specifies how to group together tuples when computing the window function.
- Use PARTITION BY to specify group.

cid	sid	row_num
3212	53666	1
3212	53655	2
3215	53688	1
3215	53666	2
3426	53688	1

```
SELECT cid, sid,
  ROW_NUMBER() OVER (PARTITION BY cid)
  FROM enrolled
 ORDER BY cid
```

Window Functions

- You can also include an ORDER BY in the window grouping to sort entries in each group.

```
SELECT *,
  ROW_NUMBER() OVER (ORDER BY cid)
  FROM enrolled
 ORDER BY cid
```

Window Functions

- Find the student with the second highest grade for each course.

Group tuples by cid
Then sort by grade

```
SELECT *, FROM (
  SELECT *, RANK() OVER (PARTITION BY cid
            ORDER BY grade ASC) AS rank
  FROM enrolled) AS ranking
WHERE ranking.rank = 2
```

Nested Queries

- Invoke a query inside of another query to compose more complex computations.
  - Inner queries can appear (almost) anywhere in query.

Outer query

```
SELECT name FROM student WHERE
  sid IN (SELECT sid FROM enrolled)
```

Inner query

```
SELECT name
    (SELECT name FROM student AS s
     WHERE s.sid = e.sid) AS name
FROM enrolled AS e;
```

```
SELECT * FROM student
ORDER BY (SELECT MAX(sid) FROM student);
```

Nested Queries

- *Get the names of students in '3212'*

```
SELECT name FROM student
WHERE ...
```

sid in the set of people that take 3212

```
SELECT name FROM student
 WHERE ...
   SELECT sid FROM enrolled
   WHERE cid = '3212';
```

```
SELECT name FROM student
 WHERE sid IN (
   SELECT sid FROM enrolled
   WHERE cid = '3212'
);
```

Nested Queries

- ALL→ Must satisfy expression for all rows in the sub-query.
- ANY→ Must satisfy expression for at least one row in the sub-query.
- IN→ Equivalent to '=ANY()' .
- EXISTS→ At least one row is returned without comparing it to an attribute in outer query.

Nested Queries

- *Get the names of students in '3212'*

```
SELECT name FROM student
 WHERE sid = ANY(
    SELECT sid FROM enrolled
    WHERE cid = '3212'
)
```

Nested Queries

- *Find student record with the highest id that is enrolled in at least one course.*

```
SELECT MAX(e.sid), s.name
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid;
```

This won't work in SQL-92. It runs in SQLite, but not Postgres or MySQL (v8 with strict mode).

```
SELECT sid, name FROM student
WHERE sid = (
    SELECT MAX(sid) FROM enrolled
)
```

Nested Queries

- *Find all courses that have no students enrolled in it.*

```
SELECT * FROM course
WHERE NOT EXISTS (
    ### tuples in the enrolled table ###
    SELECT * FROM enrolled
      WHERE course.cid = enrolled.cid
)
```

"with no tuples in the enrolled table"

cid	name
3212	Database Systems
3215	Advanced Database Systems
3426	Data Mining
3313	Special Topics in Databases

sid	cid	grade
53666	3212	C
53688	3215	A
53688	3426	B
53655	3212	B
53666	3215	C

Nested Queries

- *Find all courses that have no students enrolled in it.*

```
SELECT * FROM course
WHERE NOT EXISTS (
    ### tuples in the enrolled table ###
    SELECT * FROM enrolled
      WHERE course.cid = enrolled.cid
)
```

"with no tuples in the enrolled table"

cid	name
3212	Database Systems
3215	Advanced Database Systems
3426	Data Mining
3313	Special Topics in Databases

sid	cid	grade
53666	3212	C
53688	3215	A
53688	3426	B
53655	3212	B
53666	3215	C

Lateral Join

- The LATERAL operator allows a nested query to reference attributes in other nested queries that precede it. (inner -> outer but not outer -> inner)
  - You can think of it like a for loop that allows you to invoke another query for each tuple in a table.

```
SELECT * FROM
 (SELECT 1 AS x) AS t1
   LATERAL (SELECT t1.x+1 AS y) AS t2;
```

t1.x	t2.y
1	2

Lateral Join

- *Calculate the number of students enrolled in each course and the average GPA. Sort by enrollment count in descending order.*

SELECT * **FROM** course **AS** c
For each course:
➡ Compute the # of enrolled students

For each course:
➡ Compute the average gpa of enrolled students

cid	name	cnt	avg
3212	Database Systems	2	3.75
3215	Advanced Database Systems	2	3.95
3426	Data Mining	1	3.9
3313	Special Topics in Databases	0	null

Lateral Join

- *Calculate the number of students enrolled in each course and the average GPA. Sort by enrollment count in descending order.*

```
SELECT * FROM course AS c
   LATERAL (SELECT COUNT(*) AS cnt FROM enrolled
            WHERE enrolled.cid = c.cid) AS t1,
   LATERAL (SELECT AVG(gpa) AS avg FROM student AS s
            JOIN enrolled AS e ON s.sid = e.sid
            WHERE e.cid = c.cid) AS t2;
```

cid	name	cnt	avg
3212	Database Systems	2	3.75
3215	Advanced Database Systems	2	3.95
3426	Data Mining	1	3.9
3313	Special Topics in Databases	0	null

Common Table Expressions

- Specify a temporary result set/table that can then be referenced by another part of that query.
  - Think of it like a temp table just for one query.
- Alternative to nested queries, views, and explicit temp tables.

```
WITH cteName AS (
    SELECT 1
)
SELECT * FROM cteName
```

```
WITH cteName AS (
    SELECT 1
)
SELECT * FROM cteName
```

Common Table Expressions

- You can bind/alias output columns to names before the AS keyword.

```
WITH cteName (col1, col2) AS (
    SELECT 1, 2
)
SELECT col1 + col2 FROM cteName
```

Postgres

```
WITH cteName (colxxx, colxxx) AS (
    SELECT 1, 2
)
SELECT * FROM cteName
```

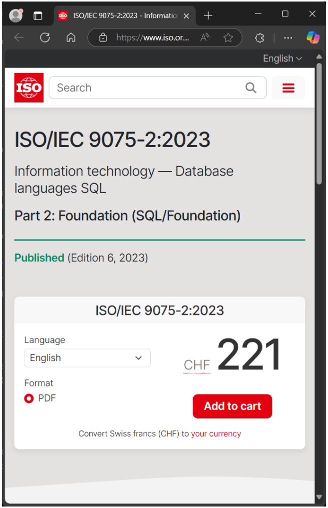
Common Table Expressions

- Find student record with the highest id that is enrolled in at least one course.

```
WITH cteSource (maxId) AS (
    SELECT MAX(sid) FROM enrolled
)
SELECT name FROM student, cteSource
   WHERE student.sid = cteSource.maxId
```

Other Important Things to Note

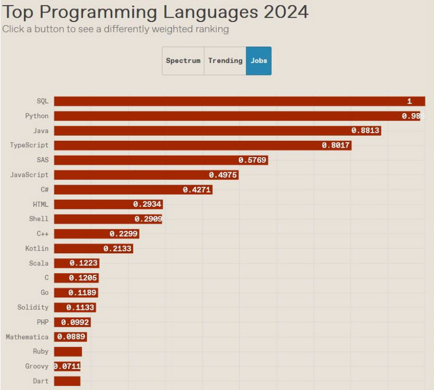
- Identifiers (e.g. table and column names) are case-insensitive.
  - Makes it harder for applications that care about case (e.g., use CamelCased names).
- One often sees quotes around names:
  - SELECT "ArtistList.firstName“
- Standard SQL documents is not free.



Conclusion

- SQL is *the* hottest language, even beating Python.
  - Lots of NL2SQL* tools, but writing SQL is not going away.

*natural language queries to structured SQL queries



Compose a single SQL statement is almost always difficult,
but you should almost always do it.