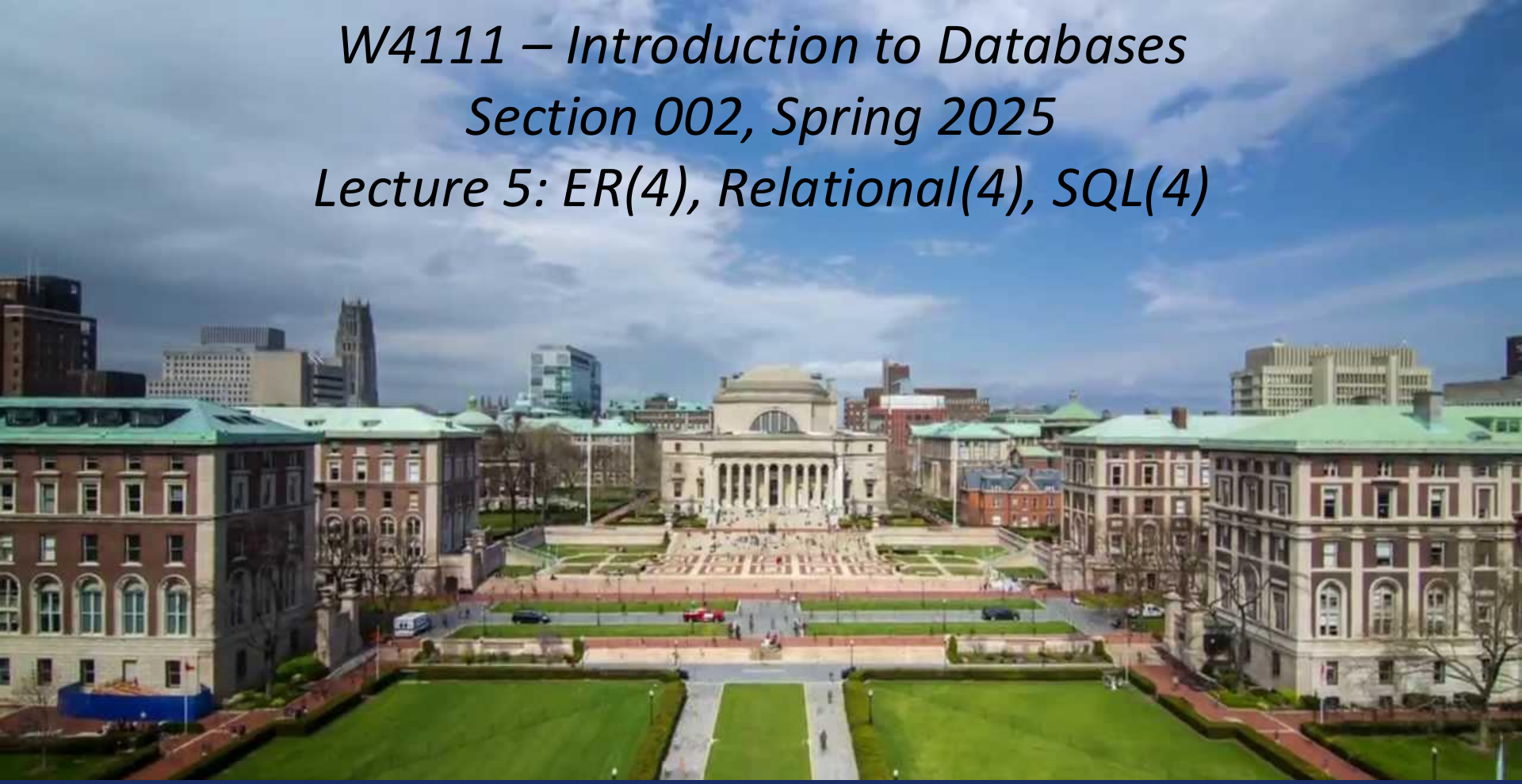
W4111 – Introduction to Databases
Section 002, Spring 2025
Lecture 5: ER(4), Relational(4), SQL(4)



Today's Contents

Contents

- ~~ER Modeling~~
  - ~~TBD~~
- ~~Relational Algebra/Model~~
  - ~~TBD~~
- SQL
  - Some datatypes and functions – Examples
  - Indexes
  - Authorization
  - Window functions
  - Recursion
  - Worked example
- Applications and Databases; Introduction to REST and web applications
- Codd's 12 Rules

ER Modeling

Relational Model

SQL

Some Types and Functions

Show Notebook

Indexes

Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the **create index** command

  create index <name> **on** <relation-name> (attribute);
- SQL engines automatically create indexes for keys and constraints.
- Identifying indexes to create requires understanding how users and applications will query the data. Indexes can have an exponential performance improvement.

Index Creation Example

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))
- **create index** *studentID_index* **on** *student*(*ID*)
- The query:

  select *
  from *student*
  where *ID* = '12345'

  can be executed by using the index to find the required record, without looking at all records of *student*

DFF:

- An index on (column1, column2, column3)
- Is also an index on (column1) and (column1, column2)
- But not (column2), (column3), (column2, column3).
- We will see "why" later in the semester.

Browse some schemas and discuss indexes:

- DB Book
- Lahman's
- IMDB

Authorization

Security Concepts (Terms from Wikipedia)

- Definitions:
  - “A (digital) identity is information on an entity used by computer systems to represent an external agent. That agent may be a person, organization, application, or device.”
  - “Authentication is the act of proving an assertion, such as the identity of a computer system user. In contrast with identification, the act of indicating a person or thing's identity, authentication is the process of verifying that identity.”
  - “Authorization is the function of specifying access rights/privileges to resources, … More formally, "to authorize" is to define an access policy. … During operation, the system uses the access control rules to decide whether access requests from (authenticated) consumers shall be approved (granted) or disapproved.
  - “Within an organization, roles are created for various job functions. The permissions to perform certain operations are assigned to specific roles. Members or staff (or other system users) are assigned particular roles, and through those role assignments acquire the permissions needed to perform particular system functions.”
  - “In computing, privilege is defined as the delegation of authority to perform security-relevant functions on a computer system. A privilege allows a user to perform an action with security consequences. Examples of various privileges include the ability to create a new user, install software, or change kernel functions.”

- SQL and relational database management systems implementing security by:
  - Creating identities and authentication policies.
  - Creating roles and assigning identities to roles.
  - Granting and revoking privileges to/from roles and identities.

Authorization

- We may assign a user several forms of authorizations on parts of the database.
  - **Read** - allows reading, but not modification of data.
  - **Insert** - allows insertion of new data, but not modification of existing data.
  - **Update** - allows modification, but not deletion of data.
  - **Delete** - allows deletion of data.
- Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

Authorization (Cont.)

- Forms of authorization to modify the database schema
  - **Index** - allows creation and deletion of indices.
  - **Resources** - allows creation of new relations.
  - **Alteration** - allows addition or deletion of attributes in a relation.
  - **Drop** - allows deletion of relations.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

  grant <privilege list> **on** <relation or view > **to** <user list>
- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Example:
  - **grant select on** *department* **to** Amit, Satoshi
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:

    grant select on *instructor* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list> **on** <relation or view> **from** <user list>

- Example:

  revoke select on *student* **from** U_1, U_2, U_3

- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:

  create a role <name>
- Example:
  - **create role** instructor
- Once a role is created we can assign “users” to the role using:
  - **grant** <role> **to** <users>

Roles Example

- **create role** instructor;
- **grant** *instructor* **to** Amit**;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
  - Creator of view did not have some permissions on *instructor?*

Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - Why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
  - And more!

Note:

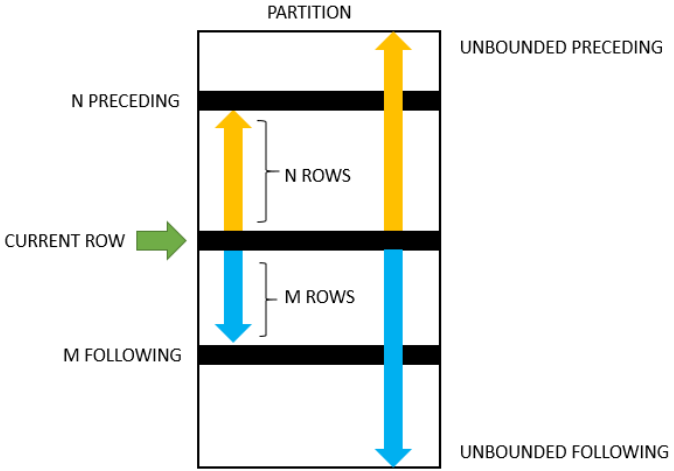
- Like in many other cases, SQL DBMS have product specific variations.

Advanced Aggregation
Window Functions

Advanced Aggregates and Window Functions

- The aggregation functions in SQL are powerful.
- There are some scenarios that are very difficult. SQL and databases add support for more advanced capabilities:
  - Ranking
  - Windows
  - Pivot, Slice and Dice, but we will cover these with a different technology (OLAP) later in the semester.

```
<window function name>( )   OVER (
        PARTITION BY <expression>
        ORDER BY <expression> [ASC | DESC]
    )
```



  - This is powerful, a little complex and requires trial and error.

Classic Models: Annual Revenue and YoY Growth by Country

```sql
SELECT
    country,
    YEAR(orderDate) AS year,
    ROUND(SUM(priceEach * quantityOrdered), 2) AS annual_revenue,
    ROUND(
        (SUM(priceEach * quantityOrdered) -
        LAG(SUM(priceEach * quantityOrdered)) OVER (PARTITION BY country ORDER BY YEAR(orderDate))) /
        LAG(SUM(priceEach * quantityOrdered)) OVER (PARTITION BY country ORDER BY YEAR(orderDate)) * 100, 2
    ) AS yoy_growth_percentage
FROM
    customers
    JOIN orders ON customers.customerNumber = orders.customerNumber
    JOIN orderdetails ON orders.orderNumber = orderdetails.orderNumber
GROUP BY
    country, year
ORDER BY
    country, year;
```

Revenue and Growth Quarter to Quarter

```
WITH quarterly_revenue AS (
   SELECT
      YEAR(o.orderDate) AS order_year, QUARTER(o.orderDate) AS order_quarter, SUM(od.quantityOrdered * od.priceEach) AS revenue
   FROM
      orders o JOIN orderdetails od ON o.orderNumber = od.orderNumber WHERE o.status = 'Shipped'
   GROUP BY  YEAR(o.orderDate), QUARTER(o.orderDate)
)

SELECT
   qr.order_year, qr.order_quarter, qr.revenue,
   LAG(qr.revenue) OVER (ORDER BY qr.order_year, qr.order_quarter) AS previous_quarter_revenue,
   ROUND(
      (qr.revenue - LAG(qr.revenue) OVER (ORDER BY qr.order_year, qr.order_quarter))
      / LAG(qr.revenue) OVER (ORDER BY qr.order_year, qr.order_quarter) * 100,
      2
   ) AS qoq_growth_percent
FROM
   quarterly_revenue qr
ORDER BY
   qr.order_year, qr.order_quarter;
```

Recursive Queries

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

This example view, *rec_prereq,* is called the *transitive closure* of the *prereq* relation

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in book

The Power of Recursion

- Computing transitive closure using iteration, adding successive tuples to *rec_prereq*
  - The next slide shows a *prereq* relation
  - Each step of the iterative process constructs an extended version of *rec_prereq* from its recursive definition.
  - The final result is called the *fixed point* of the recursive view definition.
- Recursive views are required to be **monotonic**. That is, if we add tuples to *prereq* the view *rec_prereq* contains all of the tuples it contained before, plus possibly more

This is from the book.
There is a typo in the query.
I cannot figure out what they are trying to accomplish.
The data is not complex enough to justify recursion.
See the following example.

Example of Fixed-Point Computation

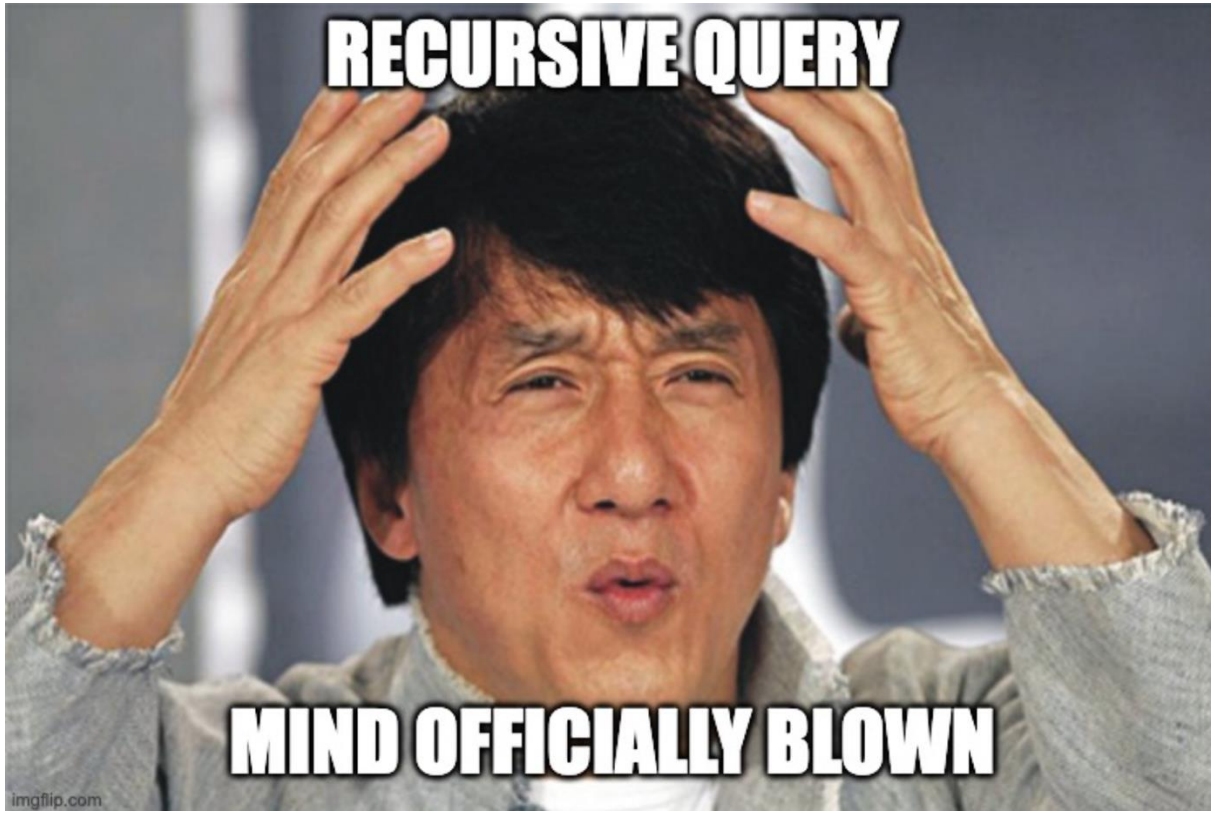
course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-190
CS-319	CS-101
CS-319	CS-315
CS-347	CS-319

Iteration Number	*Tuples in c1*
0	
1	(CS-319)
2	(CS-319), (CS-315), (CS-101)
3	(CS-319), (CS-315), (CS-101), (CS-190)
4	(CS-319), (CS-315), (CS-101), (CS-190)
5	done

Classic Models Management Chain

```
WITH RECURSIVE employee_chain AS (
   -- Base case: Start with employees who do not report to anyone (top-level managers)
   SELECT
      employeeNumber AS employee_id, reportsTo AS manager_id, CONCAT(firstName, ' ', lastName) AS employee_name,
         CAST(employeeNumber AS CHAR(100)) AS management_chain
   FROM
      employees
   WHERE
      reportsTo IS NULL
   UNION ALL
   -- Recursive step: Add employees who report to employees already in the chain
   SELECT
      e.employeeNumber AS employee_id, e.reportsTo AS manager_id, CONCAT(e.firstName, ' ', e.lastName) AS employee_name,
         CONCAT(ec.management_chain, ' -> ', e.employeeNumber) AS management_chain
   FROM
      employees e JOIN employee_chain ec
   ON
      e.reportsTo = ec.employee_id
)
SELECT
   employee_id, employee_name, management_chain
FROM
   employee_chain
ORDER BY    management_chain;
```

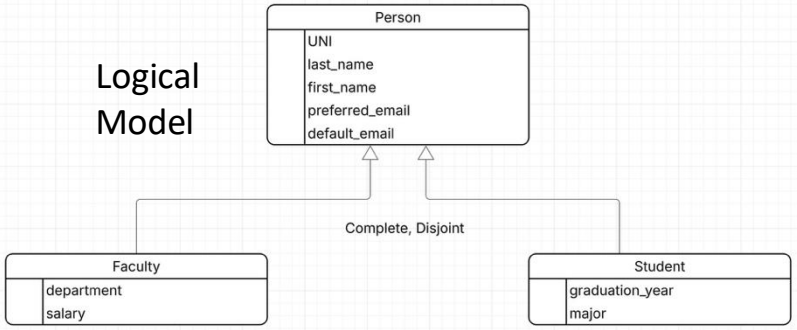
Recursive Query



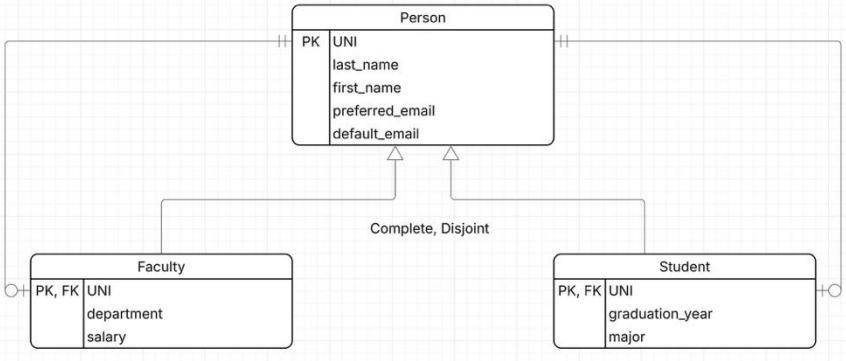
Let's Put Some Pieces Together
Specialization
Procedures, Functions, Triggers
Authorization

Specialization

Logical Model



3-Table Solution



- We will use a 3-Table Solution
- UNI must be globally unique PK
- Complete ➔
  - You create a Faculty or a Student
  - But, this requires inserting into two tables.
  - So, we will use Procedures
- preferred_email is Unique.
  Default email is autogenerated.
- Salary is sensitive information;
  We will control access.
- That seems like enough fun for now.

What Have We Seen?

- A partial implementation of specialization putting pieces together:
  - 3-table solution
  - Views
  - The use of procedures, including allowing a user to perform operations that cannot call directly because the procedure runs “as definer.”
  - Functions
- This is only a partial implementation to a contrived scenario.
  - Why would I let *general_user* set a salary on create but not query it.
  - I should add some triggers, for example to prevent changing *UNI.*
  - Additional views?
  - Additional procedures, e.g. for update and delete
  - … …
- You will do a more complete implementation of an alternate scenario on HW 3.

Applications and Databases
Web Applications and REST

Introduction

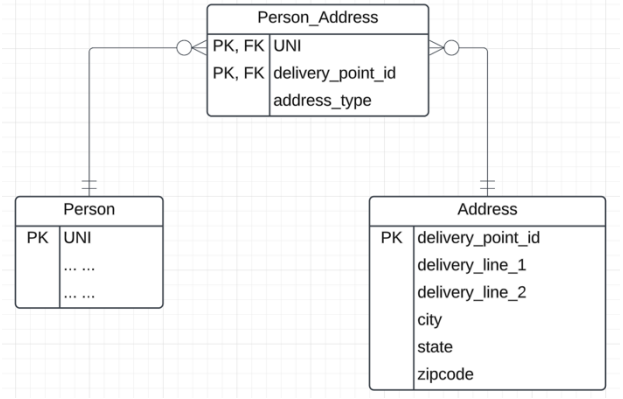
Accessing SQL from a Programming Language

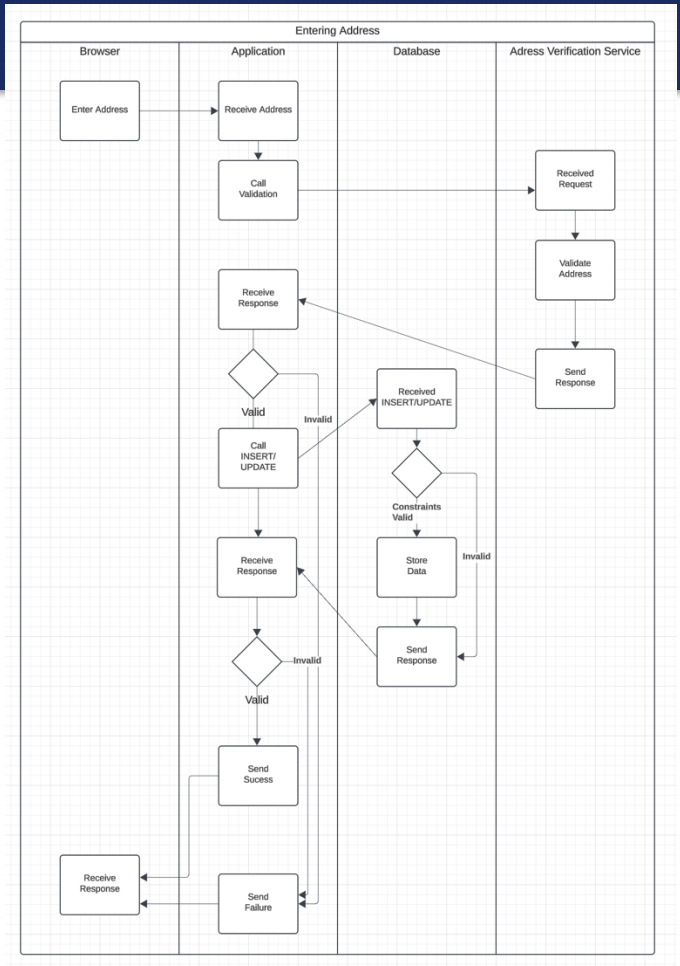
A database programmer must have access to a general-purpose programming language for at least two reasons

- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language.
- Non-declarative actions -- such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface -- cannot be done from within SQL.

Entering an address is an example that I use in my cloud computing course.

User Profile





Why application logic? An example.

- People are notoriously bad at entering addresses:
  - 520 w120th street, ny ny
  - 520 West 120th, NYC, NY
  - 520 w 120 street, New York city, NY
  - … …
- Every address has a canonical format.
- I want to call a cloud-based address verification service or some other validations.
- Complex, multi-database application logic is very hard to write in SQL.
- "Logic" in the database should focus on integrity of the data, not general purpose, complex application behavior.

Accessing SQL from a Programming Language (Cont.)

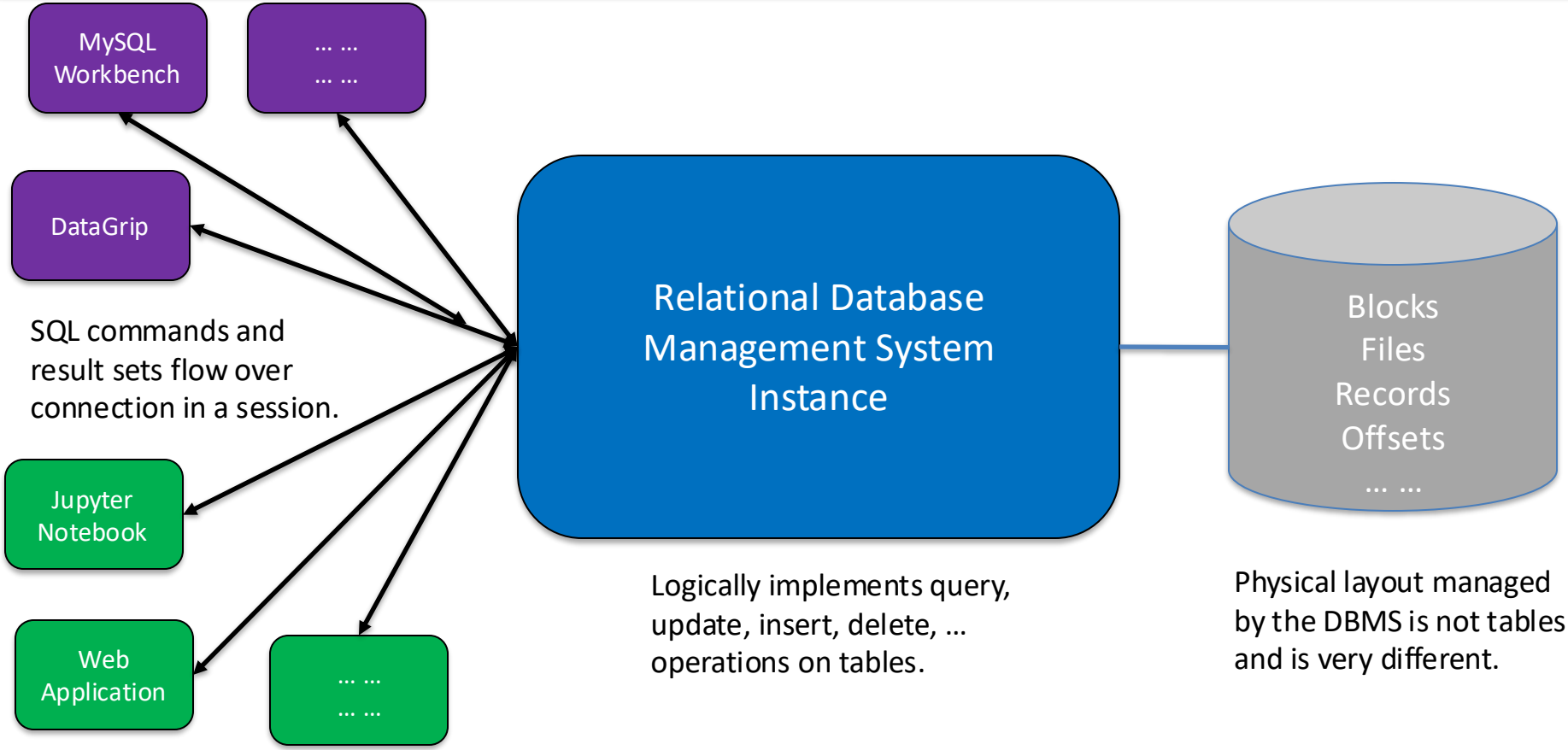
There are two approaches to accessing SQL from a general-purpose programming language

- A general-purpose program -- can connect to and communicate with a database server using a collection of functions
- Embedded SQL -- provides a means by which a program can interact with a database server.
  - The SQL statements are translated at compile time into function calls.
  - At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities.

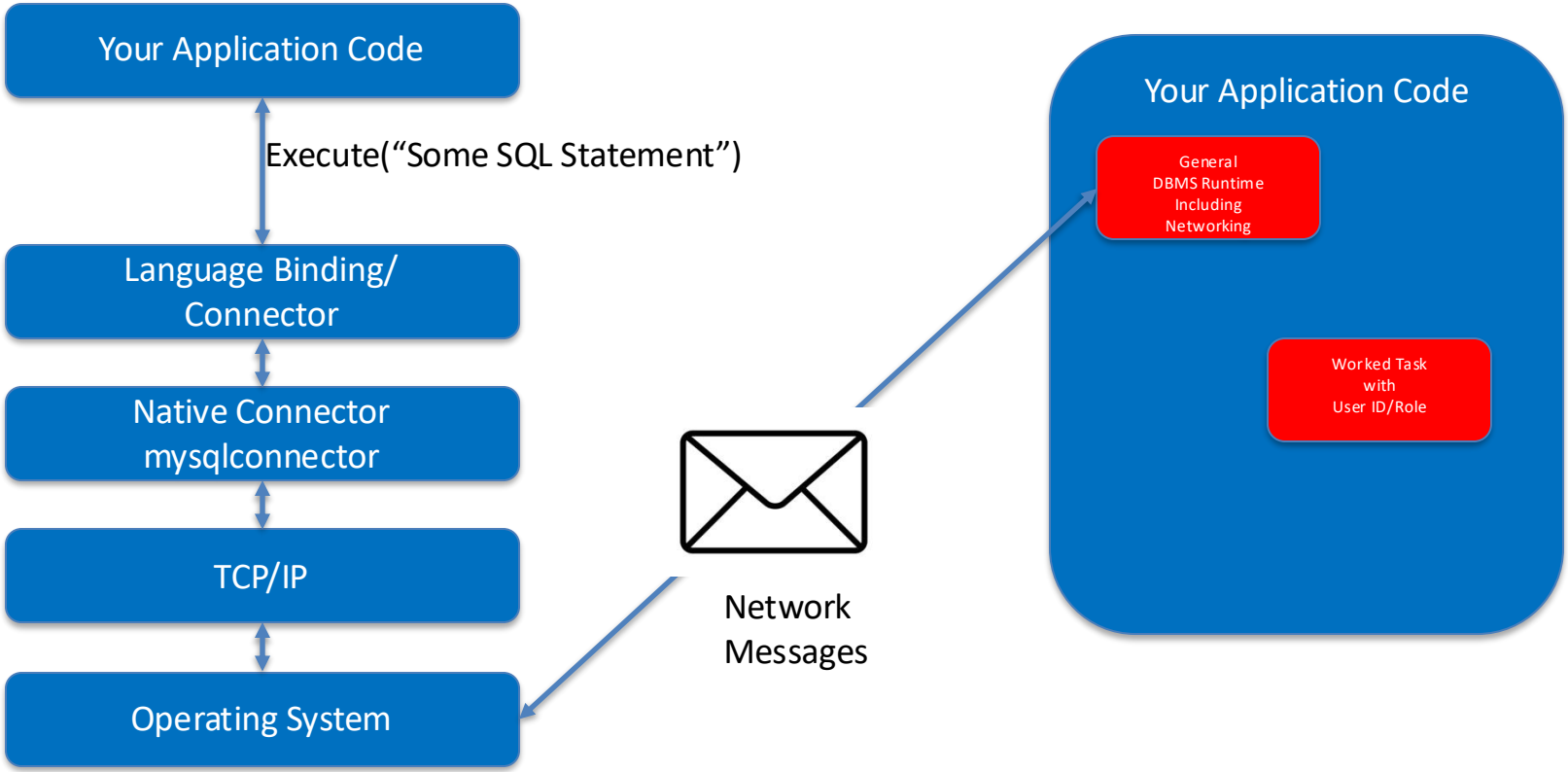
Some Terms

- "A database connection is a facility in computer science that allows client software to talk to database server software, whether on the same machine or not. A connection is required to send commands and receive answers, usually in the form of a result set." (https://en.wikipedia.org/wiki/Database_connection)
- Session:
  - "Connection is the relationship between a client and a MySQL database.
    Session is the period of time between a client logging in (connecting to) a MySQL database and the client logging out (exiting) the MySQL database."
    (https://stackoverflow.com/questions/8797724/mysql-concepts-session-vs-connection)
  - "A session is just a result of a successful connection."
- Network protocols are layered. You can think of:
  - Connection as the low-level network connection.
  - Session is the next layer up and has additional information associated with it, e.g. the user.
- Connection libraries like pymysql sometimes blur the distinction.

Concepts



Simplistic View



Transactions

- A **transaction** consists of a sequence of query and/or update statements and is a "unit" of work
- The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
- The transaction must end with one of the following statements:
  - **Commit work**. The updates performed by the transaction become permanent in the database.
  - **Rollback work**. All the updates performed by the SQL statements in the transaction are undone.
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions

Show connecting
Explain auto-commit

REST and Web Applications

Full Stack Application

Full Stack Developer Meaning & Definition

In technology development, full stack refers to an entire computer system or application from the front end to the back end and the code that connects the two. The back end of a computer system encompasses "behind-the-scenes" technologies such as the database and operating system. The front end is the user interface (UI). This end-to-end system requires many ancillary technologies such as the network, hardware, load balancers, and firewalls.

FULL STACK WEB DEVELOPERS

Full stack is most commonly used when referring to web developers. A full stack web developer works with both the front and back end of a website or application. They are proficient in both front-end and back-end languages and frameworks, as well as server, network, and hosting environments.

Full-stack developers need to be proficient in languages used for front-end development such as HTML, CSS, JavaScript, and third-party libraries and extensions for Web development such as JQuery, SASS, and REACT. Mastery of these front-end programming languages will need to be combined with knowledge of UI design as well as customer experience design for creating optimal front-facing websites and applications.

https://www.webopedia.com/definitions/full-stack/

Full Stack Web Developer

A full stack web developer is a person who can develop both **client** and **server** software.

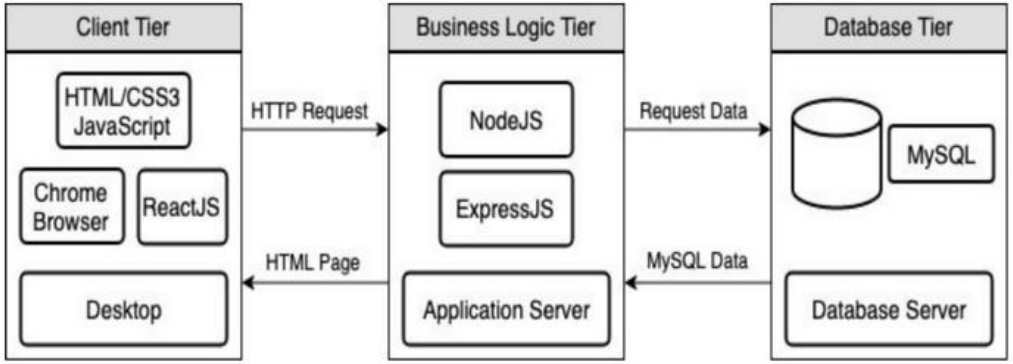
In addition to mastering HTML and CSS, he/she also knows how to:

- Program a **browser** (like using JavaScript, jQuery, Angular, or Vue)
- Program a **server** (like using PHP, ASP, Python, or Node)
- Program a **database** (like using SQL, SQLite, or MongoDB)

https://www.w3schools.com/whatis/whatis_fullstack.asp

- There are courses that cover topics:
  - COMS W4153: Advanced Software Engineering
  - COMS W4111: Introduction to Databases
  - COMS W4170 - User Interface Design
- This course will focus on cloud realization, microservices and application patterns, … …
- Also, I am not great at UIs … ... We will not emphasize or require a lot of UI work.

Full Stack Web Application



M = Mongo
E = Express
R = React
N = Node

I start with FastAPI and MySQL, but all the concepts are the same.

https://levelup.gitconnected.com/a-complete-guide-build-a-scalable-3-tier-architecture-with-mern-stack-es6-ca129d7df805

- My preferences are to replace React with Angular, and Node with FastAPI.
- There are three projects to design, develop, test, deploy, … ...
  1. Browser UI application.
  2. Microservice.
  3. Database.
- The programming track will implement a simple web application to access IMDB and Game of Thrones data.

Some Terms

- A web application server or web application framework: "A web framework (WF) or web application framework (WAF) is a software framework that is designed to support the development of web applications including web services, web resources, and web APIs. Web frameworks provide a standard way to build and deploy web applications on the World Wide Web. Web frameworks aim to automate the overhead associated with common activities performed in web development. For example, many web frameworks provide libraries for database access, templating frameworks, and session management, and they often promote code reuse." (https://en.wikipedia.org/wiki/Web_framework)

- REST: "REST (Representational State Transfer) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of a distributed, Internet-scale hypermedia system, such as the Web, should behave. The REST architectural style emphasises uniform interfaces, independent deployment of components, the scalability of interactions between them, and creating a layered architecture to promote caching to reduce user-perceived latency, enforce security, and encapsulate legacy systems.[1]

  REST has been employed throughout the software industry to create stateless, reliable web-based applications." (https://en.wikipedia.org/wiki/REST)

Some Terms

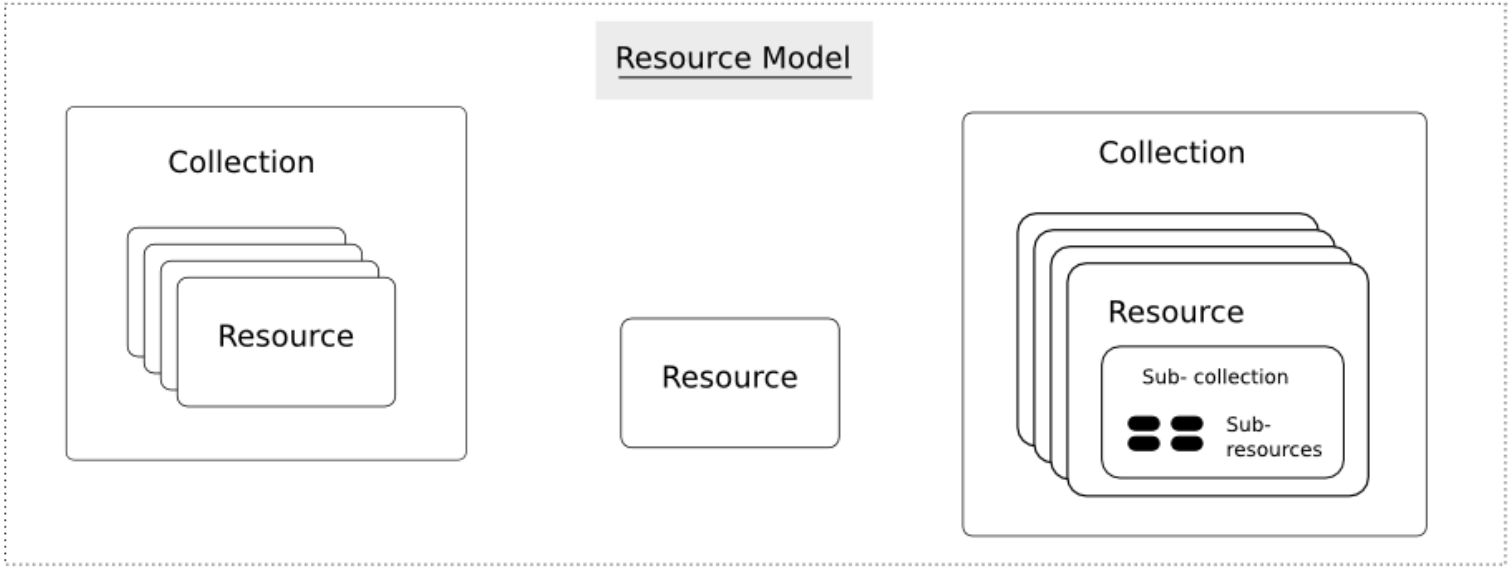
- OpenAPI: “The OpenAPI Specification, previously known as the Swagger Specification, is a specification for a machine-readable interface definition language for describing, producing, consuming and visualizing web services.” (https://en.wikipedia.org/wiki/OpenAPI_Specification)

- Model: “A model represents an entity of our application domain with an associated type.” (https://medium.com/@nicola88/your-first-openapi-document-part-ii-data-model-52ee1d6503e0)

- Routers: “What fastapi docs says about routers: If you are building an application or a web API, it’s rarely the case that you can put everything on a single file. FastAPI provides a convenience tool to structure your application while keeping all the flexibility.” (https://medium.com/@rushikeshnaik779/routers-in-fastapi-tutorial-2-adf3e505fdca)

- Summary:
  - These are general concepts, and we will go into more detail in the semester.
  - FastAPI is a specific technology for Python.
  - There are many other frameworks applicable to Python, NodeJS/TypeScript, Go, C#, Java, … …
  - They all surface similar concepts with slightly different names.

REST and Resources



REST (https://restfulapi.net/)

What is REST

- REST is acronym for **RE**presentational **S**tate **T**ransfer. It is architectural style for **distributed hypermedia systems** and was first presented by Roy Fielding in 2000 in his famous dissertation.
- Like any other architectural style, REST also does have it's own 6 guiding constraints which must be satisfied if an interface needs to be referred as **RESTful**. These principles are listed below.

Guiding Principles of REST

- **Client–server** – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
- **Stateless** – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.
- **Cacheable** – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
- **Uniform interface** – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.
- **Layered system** – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.
- **Code on demand (optional)** – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

Resources

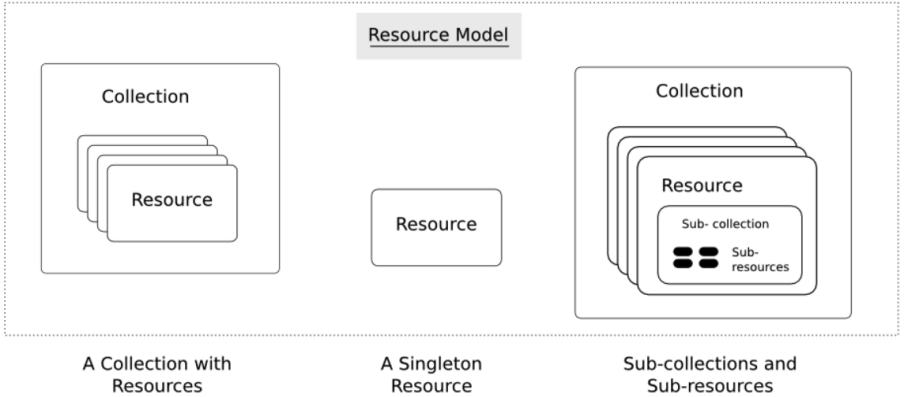
Resources are an abstraction. The application maps to create things and actions.

"A resource-oriented API is generally modeled as a resource hierarchy, where each node is either a *simple resource* or a *collection resource*. For convenience, they are often called a resource and a collection, respectively.

- A collection contains a list of resources of **the same type**. For example, a user has a collection of contacts.
- A resource has some state and zero or more sub-resources. Each sub-resource can be either a simple resource or a collection resource.

For example, Gmail API has a collection of users, each user has a collection of messages, a collection of threads, a collection of labels, a profile resource, and several setting resources.

While there is some conceptual alignment between storage systems and REST APIs, a service with a resource-oriented API is not necessarily a database, and has enormous flexibility in how it interprets resources and methods. For example, creating a calendar event (resource) may create additional events for attendees, send email invitations to attendees, reserve conference rooms, and update video conference schedules. (Emphasis added) (https://cloud.google.com/apis/design/resources#resources)



https://restful-api-design.readthedocs.io/en/latest/resources.html

REST – Resource Oriented

- When writing applications, we are used to writing functions or methods:
  - openAccount(last_name, first_name, tax_payer_id)
  - account.deposit(deposit_amount)
  - account.close()

  We can create and implement whatever functions we need.
- REST only allows four methods:
  - POST: Create a resource
  - GET: Retrieve a resource
  - PUT: Update a resource
  - DELETE: Delete a resource

  That's it. That's all you get.

"The key characteristic of a resource-oriented API is that it emphasizes resources (data model) over the methods performed on the resources (functionality). A typical resource-oriented API exposes a large number of resources with a small number of methods." (https://cloud.google.com/apis/design/resources)

- A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

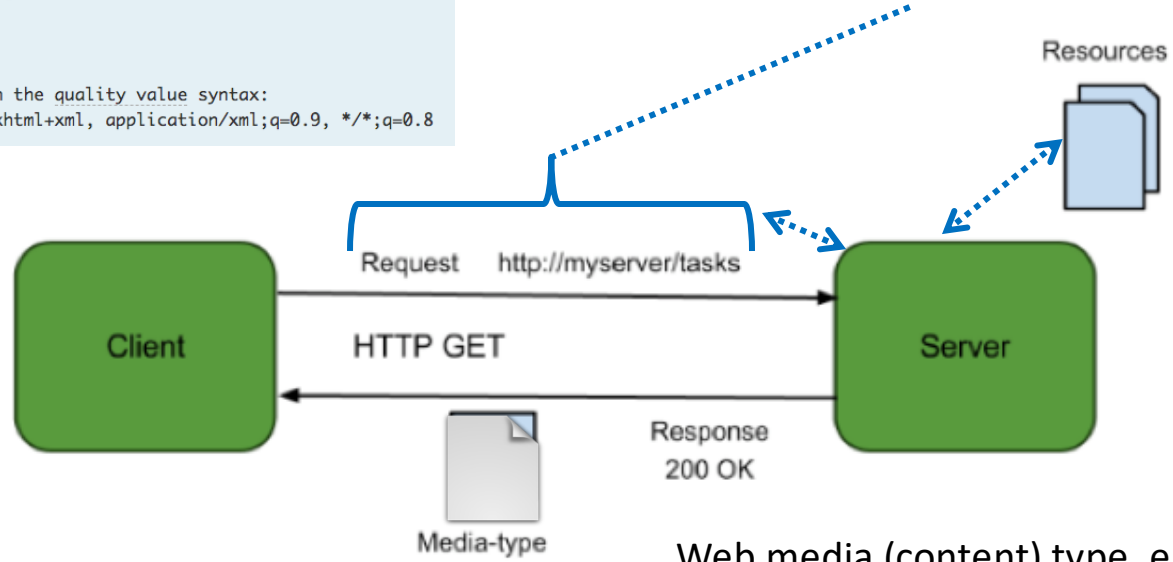
Resources, URLs, Content Types

- Relative URL identifies “resource” on the server.
- Server implementation maps abstract resource to tangible “thing,” file, DB row, … and any application logic.

Accept type in headers.

```
Accept: <MIME_type>/<MIME_subtype>
Accept: <MIME_type>/*
Accept: */*

// Multiple types, weighted with the quality value syntax:
Accept: text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8
```



Client may be

Browser

Mobile device

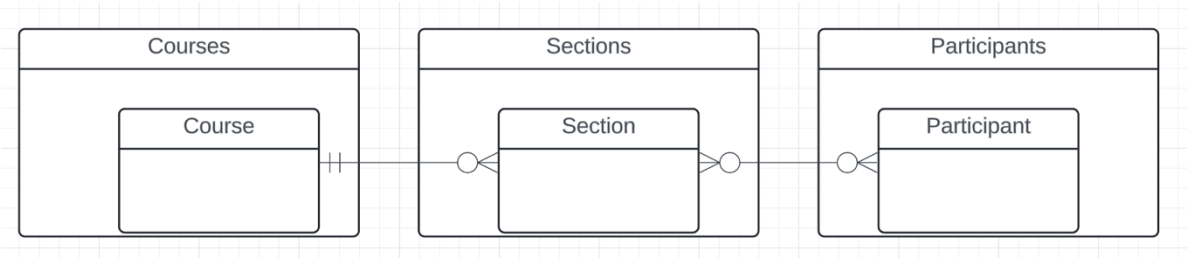
Other REST Service

… …

Web media (content) type, e.g.

- text/html
- application/json

Resources and APIs



- Base resources, paths and methods:
  - /courses: GET, POST
  - /courses/<id>: GET, PUT, DELETE
  - /sections: GET, POST
  - /sections/<id>: GET, PUT, DELETE
  - /participants: GET, POST
  - /participants/<id>: GET, PUT, DELETE
- There are relative, navigation paths:
  - /courses/<id>/sections
  - /participants/<id>/sections
  - etc.
- GET on resources that are collections may also have query parameters.

- There are two approaches to defining API
  - Start with OpenAPI document and produce an implementation template.
  - Start with annotated code and generate API document.
- In either approach, I start with *models.*
- Also,
  - I lack the security permission to update CourseWorks.
  - I can choose to not surface the methods or raise and exception.

Data Modeling Concepts and REST

Almost any data model has the same core concepts:

- Types and instances:
  - Entity Type: A definition of a type of thing with properties and relationships.
  - Entity Instance: A specific instantiation of the Entity Type
  - Entity Set Instance: An Entity Type that:
    - Has properties and relationships like any entity, but …
    - Has at least one *special relationship* – ***contains.***
- Operations, minimally CRUD, that manipulate entity types and instances:
  - Create
  - Retrieve
  - Update
  - Delete
  - Reference/Identify/… …
  - Host/database/table/pk

REST (https://www.tutorialspoint.com/restful/restful_introduction.htm)

What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- **GET** – Provides a read only access to a resource.
- **POST** – Used to create a new resource.
- **DELETE** – Used to remove a resource.
- **PUT** – Used to update a existing resource or create a new resource.

REST (https://www.tutorialspoint.com/restful/restful_introduction.htm)

Introduction to RESTFul web services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

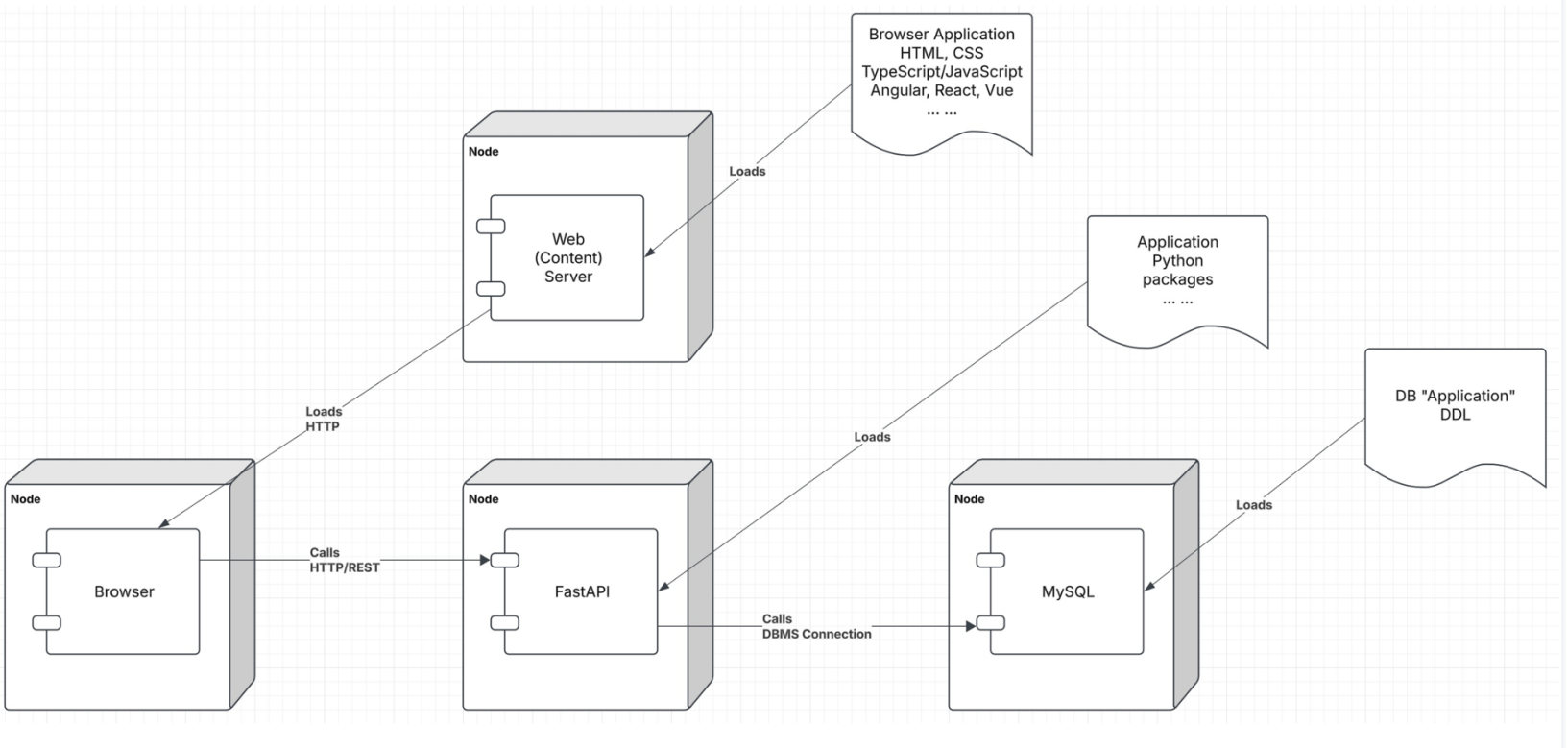
Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.

Creating RESTFul Webservice

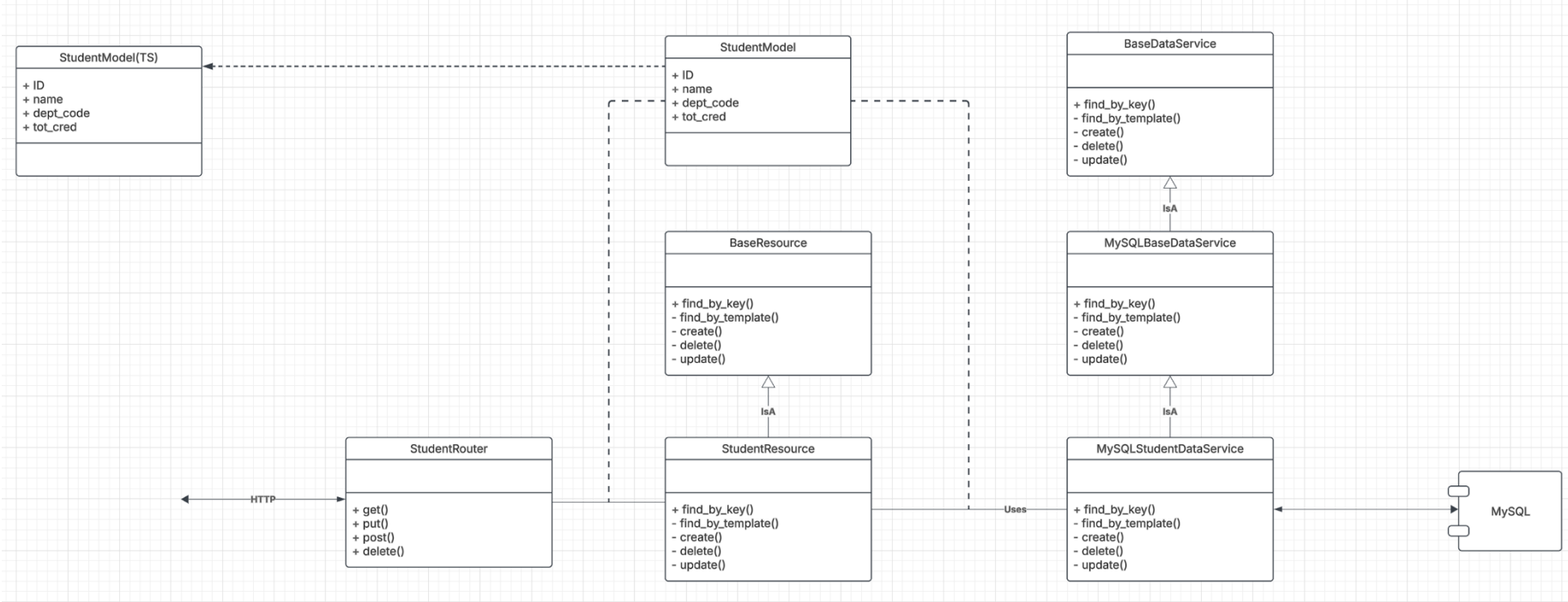
In next chapters, we'll create a webservice say user management with following functionalities –

Sr.No.	URI	HTTP Method	POST body	Result
1	/UserService/users	GET	empty	Show list of all the users.
2	/UserService/addUser	POST	JSON String	Add details of new user.
3	/UserService/getUser/:id	GET	empty	Show details of a user.

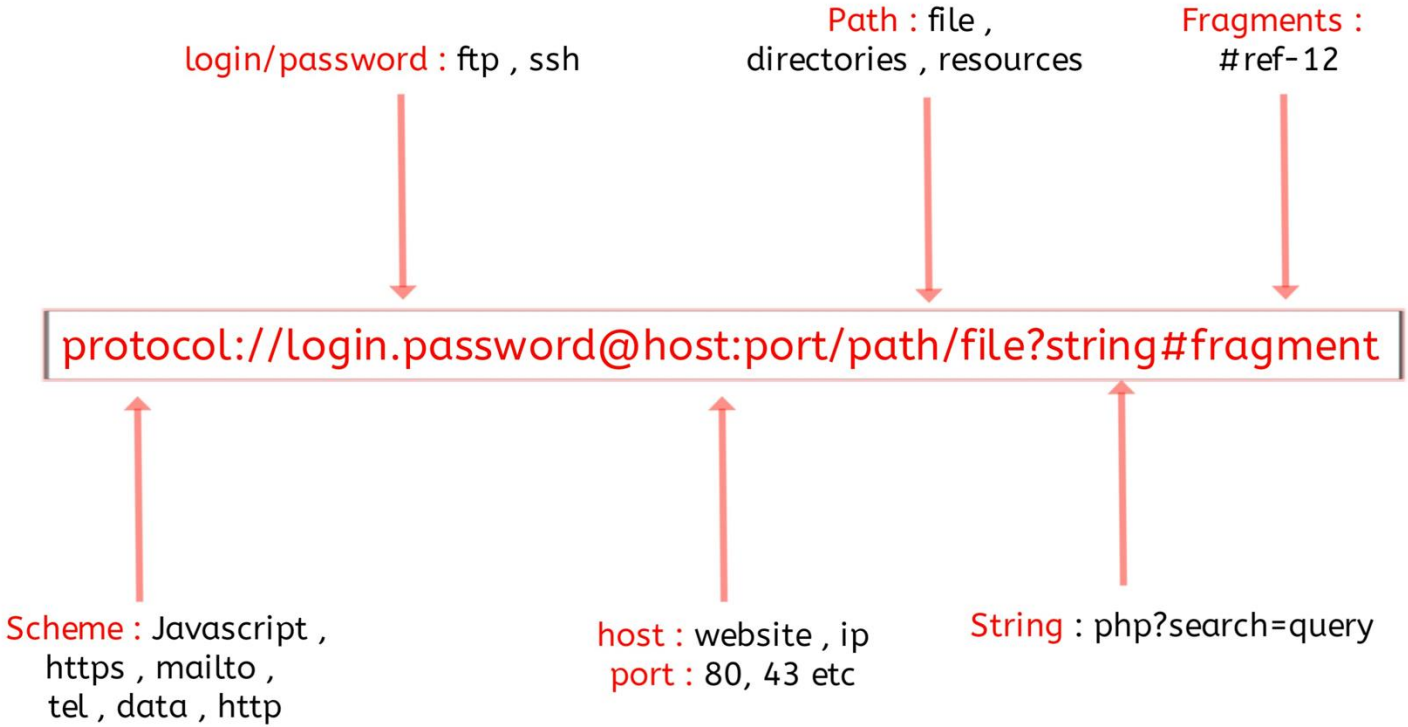
Let’s Look at the Programming Project



Let’s Look at the Programming Project



URLs



Simplistic, Conceptual Mapping (Examples)

REST Method	Resource Path	Relational Operation	DB Resource
DELETE	/people	DROP TABLE	people table
POST	/people	INSERT INTO PEOPLE (...) VALUES(...)	people table people row
GET	/people/21	SHOW KEYS FROM people ...; SELECT * FROM people WHERE playerID= 21	people row
GET	/people/21/batting	SELECT batting.* FROM people JOIN batting USING(playerID) WHERE playerID=21	
GET	/people/21/batting/2004_1	SELECT batting.* FROM people JOIN batting USING(playerID) WHERE playerID=21 AND yearID=2004 AND stint=1	

Application Architecture

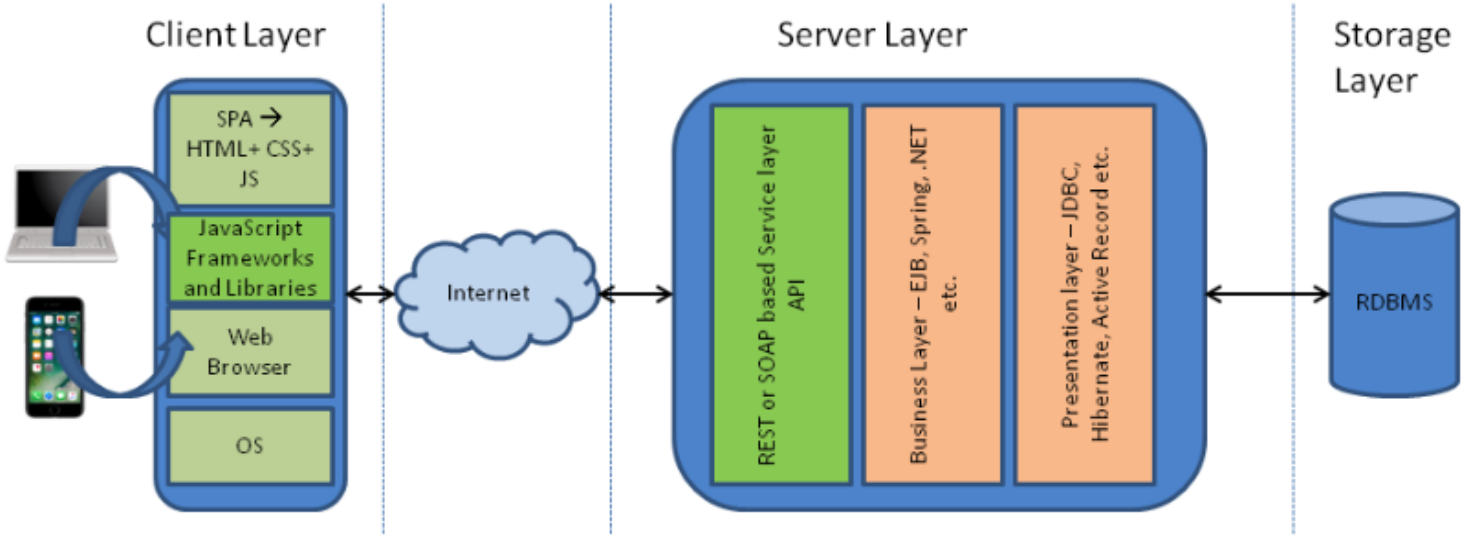


Diagram 2: The moving of the Web Layer from the Server to the Client

Codd's 12 Rules

Codd's 12 Rules

Rule 1: Information Rule

The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

Rule 2: Guaranteed Access Rule

Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data.

Rule 3: Systematic Treatment of NULL Values

The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following − data is missing, data is not known, or data is not applicable.

Rule 4: Active Online Catalog

The structure description of the entire database must be stored in an online catalog, known as data dictionary, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

Rule 5: Comprehensive Data Sub-Language Rule

A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

Rule 6: View Updating Rule

All the views of a database, which can theoretically be updated, must also be updatable by the system.

Codd's 12 Rules

Rule 7: High-Level Insert, Update, and Delete Rule

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.

Rule 8: Physical Data Independence

The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.

Rule 9: Logical Data Independence

The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rule to apply.

Rule 10: Integrity Independence

A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.

Rule 11: Distribution Independence

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

Rule 12: Non-Subversion Rule

If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.