SQL

SQL

November 14, 2025

Canine-Table

This document stages a mythic containment overlay for Structured Query Language (SQL), tracing its glyph lineage across three sacred vessels: MariaDB, PostgreSQL, and SQLite3. Each engine is treated as a sovereign temple—MariaDB, the forked steward of MySQL's legacy; PostgreSQL, the high priest of relational orthodoxy and extensible ritual; SQLite3, the hermetic scribe of embedded purity. We audit their dialectic mutations, indexing conventions, and transaction glyphs, staging boxed procedures for schema invocation, constraint binding, and query optimization. Through modular glossary engines and expressive TeX overlays, we dramatize the migration of symbols, the containment of NULL, and the sacred rites of JOIN. This mythic documentation offers disciplined lineage tracing, expressive abstraction, and ritualized troubleshooting for SQL practitioners seeking glyph purity and containment clarity across divergent relational domains.

Abstract

Contents

I Relationships and Keys

Alternative Terminology

Table	Column	Row
Relation	Attribute	Tuple
File	Field	Record

Key Types in Relational Schema

- Relational databases use keys to uniquely identify rows in a table. These keys are the glyphs of identity.
- A primary key is a natural or chosen attribute that uniquely identifies each record. It must be unique and not null.
- A surrogate key is an artificial identifier—often an auto-incremented number or UUID—used solely for uniqueness. It has no business meaning.
- A composite key is formed by combining two or more columns to uniquely identify a record. Each part alone may not be unique.
- Primary keys are often meaningful, like email or username. Surrogate keys are silent stamps, like ID numbers.
- Composite keys are used when no single attribute suffices—like (OwnerName, PetName) in a veterinary schema.
- Surrogate keys simplify joins and indexing, while composite keys preserve domain logic.
- Choosing between them depends on schema ancestry, query performance, and semantic clarity.

Primary vs Surrogate vs Composite Keys

Aspect	Primary Key	Surrogate Key	Composite Key
Definition	Natural or chosen attribute that uniquely identifies a row	Artificial identifier with no domain meaning	Combination of multiple columns for uniqueness
Semantic Meaning	Often meaningful (e.g., email, username)	None—used purely for identity	Each part may be meaningful, but only together ensures uniqueness
Common Types	Email, SSN, username	Auto-incremented ID, UUID	(FirstName, LastName), (OwnerID, PetName)
Usage in Joins	Can be used directly	Preferred for performance	Requires matching all parts
Schema Simplicity	Simple if domain key is stable	Very simple and clean	More complex, especially with foreign keys
Performance	Depends on data type and indexing	Optimized for joins and indexing	May be slower due to multiple columns
Best Use Case	When domain attribute is stable and unique	When no natural key exists or domain key is volatile	When uniqueness depends on multiple attributes

I Primary Keys

Primary Key Principles

- Each row in a table is identified by a primary key.
- A primary key is a combination of one or more column values that make a record unique.
- Primary keys are essential for defining relationships between records in relational databases.
- Good primary keys improve lookup speed and reliability.
- Keep it short—short keys are faster for comparisons and indexing.
- Prefer numbers—numeric keys are /nx/shapes/processed faster than character types.
- Maintain simplicity—avoid special characters, spaces, and mixed casing.
- Do not change the primary key once assigned—it must remain stable.
- Primary keys do not allow duplicates or null values.
- They can be defined at the column level (single key) or table level (composite key).

Primary Key Summary

Aspect	Primary Key Principle
Definition	A column or combination of columns that uniquely identifies each row in a table.
Purpose	Used to compare, join, and define relationships between records.
Uniqueness	Must be unique across all rows.
Nullability	Cannot contain null values.
Length	Should be short for faster lookups and indexing.
Data Type Preference	Prefer numeric types over strings for performance.
Simplicity	Avoid special characters, spaces, and mixed casing.
Immutability	Should not be changed once assigned.
Definition Scope	Can be defined at column level (single key) or table level (composite key).

I Surrogate Keys

Surrogate Key Doctrine

- Surrogate keys are artificial identifiers assigned by the DBMS to uniquely identify records.
- They are typically numeric and auto-generated, such as PropertyID or UserID.
- In the RENTAL_PROPERTY table without a surrogate key, uniqueness is derived from a combination of Street, City, State/Province, Zip/PostalCode, and Country.
- These columns form a composite candidate key, but they are long and semantically heavy.
- With a surrogate key, the table uses PropertyID as the primary key, simplifying joins and indexing.
- Surrogate keys decouple schema from domain logic and improve performance.
- They are ideal when natural keys are volatile, lengthy, or composed of multiple attributes.
- Surrogate keys are internal stamps—they carry no business meaning but serve as anchors for relational integrity.

Surrogate Key Comparison

Aspect	Without Surrogate Key	With Surrogate Key
Primary Key	Composite of Street, City, State/Province, Zip/PostalCode, Country	Single-column PropertyID
Key Type	Candidate key derived from domain attributes	Surrogate key auto-generated by DBMS
Semantic Meaning	High—each part reflects real-world location	None—used purely for identity
Length	Long and multi-column	Short and single-column
Performance	Slower joins and indexing due to multiple columns	Faster joins and indexing
Stability	May change if address changes	Immutable once assigned
Foreign Key Usage	Requires multi-column references	Simple single-column references
Best Use Case	When domain attributes are stable and meaningful	When domain keys are volatile or complex

I Composite Keys

Composite Key Principles

- Composite keys are formed by combining two or more columns to uniquely identify each row in a table.
- They are used when no single attribute is sufficient to guarantee uniqueness.
- Each component of a composite key may be meaningful, but only together do they form a unique glyph.
- In a veterinary schema, (OwnerName, PetName) might be used to identify pets uniquely.
- Composite keys enforce domain logic and preserve semantic clarity.
- They require multi-column foreign keys in related tables, which can increase schema complexity.
- Joins using composite keys must match all parts, which may affect performance.
- Composite keys are defined at the table level, not the column level.
- They are ideal when the natural identity of a record is inherently multi-attribute.

Composite Key Principles

Aspect	Primary Key	Surrogate Key	Composite Key
Definition	Natural or chosen attribute for uniqueness	Artificial identifier with no domain meaning	Combination of columns for uniqueness
Semantic Meaning	Often meaningful (e.g., email, username)	None—used purely for identity	Each part may be meaningful, but only together ensures uniqueness
Common Types	Email, SSN, username	Auto-incremented ID, UUID	(OwnerName, PetName), (CourseID, StudentID)
Schema Simplicity	Simple if domain key is stable	Very simple and clean	More complex, especially with foreign keys
Performance	Depends on data type and indexing	Optimized for joins and indexing	May be slower due to multiple columns
Best Use Case	Stable, unique domain attribute	No natural key or volatile domain logic	Uniqueness depends on multiple attributes
Nulls Allowed	Not allowed	Not allowed	Not allowed
Duplicates Allowed	Not allowed	Not allowed	Not allowed
Definition Location	Column or table level	Column level only	Table level only
Foreign Key Complexity	Simple single-column reference	Simple single-column reference	Requires multi-column reference

I Foreign Keys

Foreign Key Principles

- When a table references data from another table, it forms a relationship.
- The key in the referenced table is the primary key; the referencing table holds the foreign key.
- A foreign key is the attribute that identifies a primary key in another table.
- It provides the link between two tables, enabling relational integrity and joins.
- A foreign key can be a single column or a composite of columns.
- The term ”foreign” arises because the key originates from a different table.
- Foreign keys must match the data type and constraints of the referenced primary key.
- They may be italicized in schema notation to distinguish them from primary keys.
- Example: In EMPLOYEE (EmployeeNumber, LastName, FirstName, Department), the Department attribute is a foreign key referencing DEPARTMENT.
- This relationship binds EMPLOYEE to DEPARTMENT, allowing queries to traverse organizational ancestry.

Foreign Key Summary

Aspect	Foreign Key Principle
Definition	A column or composite of columns that references the primary key of another table.
Purpose	Establishes relationships between tables, enabling joins and enforcing referential integrity.
Origin	Defined in the referencing table, pointing to a primary key in the referenced table.
Multiplicity	Can be single-column or composite.
Naming Convention	Often italicized in schema notation to distinguish from primary keys.
Example	EMPLOYEE.Department is a foreign key referencing DEPARTMENT.DepartmentName.
Constraints	Must match the data type and uniqueness of the referenced primary key.
Relational Role	Enables traversal of schema ancestry and enforces valid references.

I Identifiers and Attributes

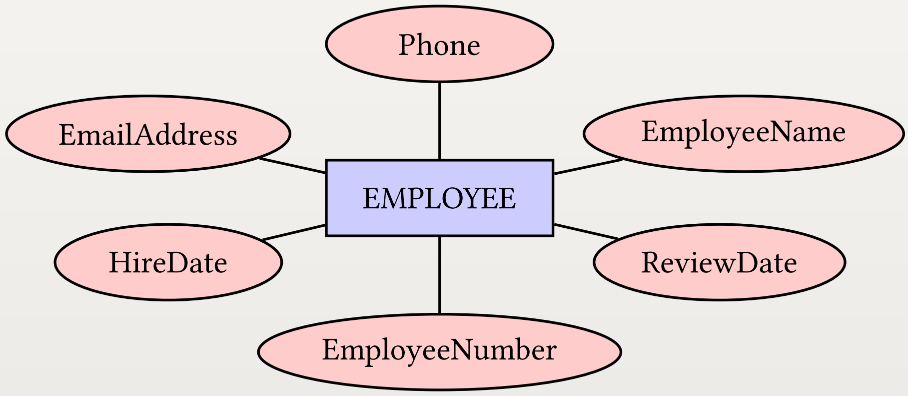
Identifiers and Attributes in Relational Schema

- An identifier is an attribute that distinguishes one entity instance from all others in the database.
- A primary key is the chosen identifier—an attribute or set of attributes that uniquely identifies each row in a table.
- Primary keys must not contain null values and must be unique across all rows.
- Candidate keys are attributes or combinations of attributes that could serve as a primary key.
- Each candidate key is a valid identifier, but only one is chosen as the primary key.
- Attributes describe the characteristics of an entity, such as name, address, or birthdate.
- All instances of an entity class share the same attributes, but differ in their values.
- In early data models, attributes were shown as ellipses; modern tools use rectangles.
- Choosing the right primary key from candidate keys ensures reliable lookups and relational integrity.

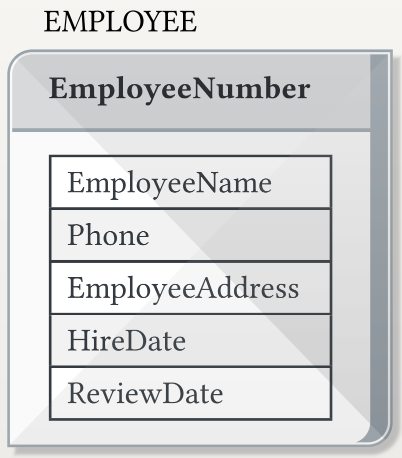
Identifier and Attribute Summary

Concept	Definition and Role
Identifier	An attribute that uniquely distinguishes one entity instance from another
Primary Key	The chosen identifier that uniquely identifies each row in a table; must be unique and not null
Candidate Key	A set of attributes that could serve as a primary key; one is selected as the actual primary key
Attribute	A property or characteristic of an entity; describes its features and values
Attribute Consistency	All instances of an entity class share the same attributes, but have different values
Modeling Convention	Originally shown as ellipses in ER diagrams; now commonly shown as rectangles
Selection Importance	Choosing the right primary key from candidate keys ensures schema clarity and performance

I Modeling Convention



(b) Attributes in Rectangle



(a) Attributes in Elipses

I Entities

Entity Principles in Relational Schema

- Entities are the foundational objects in a relational database—each representing a distinct concept or thing.
- An entity corresponds to a table, and each row within that table represents an instance of the entity.
- Entities are defined by their attributes—columns that describe properties of each instance.
- Every entity must have a primary key to uniquely identify its instances.
- Entities may participate in relationships with other entities, forming links across the schema.
- Examples of entities include Customer, Book, Author, Employee, Department, Pet, and Visit.
- Entities can be strong (having their own primary key) or weak (dependent on another entity's key).
- Entity design should reflect real-world concepts clearly and consistently.
- Attributes should be atomic, meaningful, and normalized to avoid redundancy.
- Entities are the scrolls upon which relational logic is inscribed—they anchor the schema and define its ancestry.

Entity Summary

Concept	Definition and Role
Identifier	An attribute that uniquely distinguishes one entity instance from another
Primary Key	The chosen identifier that uniquely identifies each row in a table; must be unique and not null
Candidate Key	A set of attributes that could serve as a primary key; one is selected as the actual primary key
Attribute	A property or characteristic of an entity; describes its features and values
Attribute Consistency	All instances of an entity class share the same attributes, but have different values
Modeling Convention	Originally shown as ellipses in ER diagrams; now commonly shown as rectangles
Selection Importance	Choosing the right primary key from candidate keys ensures schema clarity and performance

Entity Examples

Entity	Attributes	Notes
Customer	CustomerID (PK), Name, Email, Address, Phone	Represents a person who purchases items or services
Book	BookID (PK), Title, Genre, Price, Year	Represents a book in inventory or catalog
Author	AuthorID (PK), Name, Email, Biography	Represents a writer associated with books
Employee	EmployeeNumber (PK), FirstName, LastName, Department (FK)	Represents a staff member in an organization
Department	DepartmentName (PK), BudgetCode, OfficeNumber, Phone	Represents a division or unit within an organization
Pet	OwnerName + PetName (PK), Species, Breed, Sex, Neutered	Represents an animal owned by a client
Visit	VisitID (PK), PetID (FK), StaffID (FK), Date, Reason	Represents a medical or service encounter

I Relations

Characteristics of Relations

- Rows contain data about an entity.
- Columns contain data about attributes of the entities.
- All entries in a column are of the same kind.
- Each column has a unique name.
- Cells of the table hold a single value.
- The order of the columns is unimportant.
- The order of the rows is unimportant.
- No two rows may be identical.

Entities vs Relations

- Entities are the core objects in a database—real-world concepts like Customer, Book, or Pet.
- Each entity is represented by a table, and each row is an instance of that entity.
- Entities are defined by attributes, and must have a primary key to ensure uniqueness.
- Relations describe how entities are connected—such as a Customer purchasing a Book or a Pet visiting a Doctor.
- A relation is often implemented as a foreign key or a separate relationship table.
- Entities hold data; relations define structure and connectivity between that data.
- Relations can be one-to-one, one-to-many, or many-to-many, depending on the schema design.
- In ER diagrams, entities are boxes and relations are lines or diamonds connecting them.
- Entities are the scrolls; relations are the threads that bind them into a coherent tapestry.

I Entity Instances

CUSTOMER Entity

CUSTOMER
CustomerNumber
CustomerName
Street
City
State
ZIP
ContactName
EmailAddress

Two CUSTOMER Instances

1234	99890
Ajax Manufacturing	Jones Brothers
123 Elm Street	434 10th Street
Memphis	Boston
TN	MA
32455	01234
Peter Schwartz	Fritz Billingsley
Petter@ajax.com	Fritz@JB.com

Entities vs Relations

Aspect	Entity	Relation
Definition	A real-world object or concept represented as a table	A logical connection between two or more entities
Role in Schema	Stores data and attributes	Defines how entities interact or reference each other
Examples	Customer, Book, Pet, Employee	Purchase, AuthoredBy, Visit, Enrollment
Representation	Table with rows and columns	Foreign key, junction table, or relationship node
Key Requirement	Must have a primary key	Often uses foreign keys to link entities
Multiplicity	Not applicable directly	One-to-one, one-to-many, many-to-many
ER Diagram Symbol	Box (rectangle)	Line or diamond connecting entities
Purpose	Captures attributes and identity of things	Captures how things are connected or interact

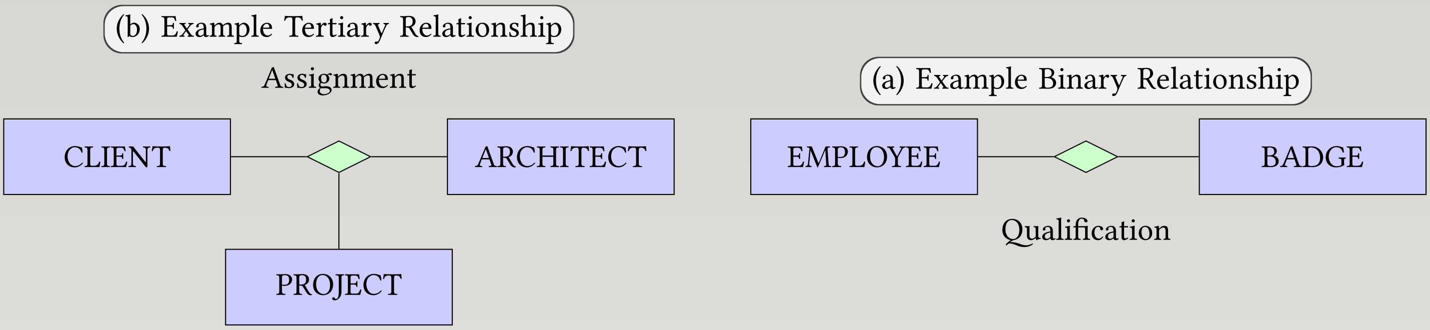
Degree of Relationship

- The degree of a relationship refers to the number of entity classes involved in that relationship.
- A binary relationship involves two entities—for example, a CUSTOMER placing an ORDER.
- A ternary relationship involves three entities—for example, a STUDENT enrolling in a COURSE taught by a PROFESSOR.
- Higher-degree relationships (quaternary and beyond) are rare and often decomposed into simpler binary relationships.
- The degree determines the complexity of the relationship and how it is represented in ER diagrams.
- Binary relationships are the most common and are represented with lines or crow's foot glyphs.
- Ternary relationships are shown with a diamond connected to three entity rectangles.
- Understanding relationship degree helps in designing normalized, expressive schemas.

Relationship Degree Comparison

Degree	Description	Example
Unary (1)	Relationship involving one entity class	EMPLOYEE manages EMPLOYEE
Binary (2)	Relationship involving two entity classes	CUSTOMER places ORDER
Ternary (3)	Relationship involving three entity classes	STUDENT enrolls in COURSE taught by PROFESSOR
Quaternary (4)	Relationship involving four entity classes (rare)	PATIENT receives TREATMENT from DOCTOR using DEVICE

I Binary Versus Ternary Relationships



I Cardinality

Cardinality in Relationships

- Cardinality describes the number of instances of one entity that can be associated with instances of another entity.
- It defines the multiplicity of a relationship—how many of A relate to how many of B.
- The most common cardinalities are one-to-one, one-to-many, and many-to-many.
- A one-to-one relationship means each instance of Entity A relates to exactly one instance of Entity B.
- A one-to-many relationship means one instance of Entity A relates to multiple instances of Entity B.
- A many-to-many relationship means multiple instances of Entity A relate to multiple instances of Entity B.
- Cardinality is often expressed in ER diagrams using crow's foot notation or numeric ranges (e.g., 0..*, 1..1).
- It helps enforce referential integrity and guides foreign key placement.
- Understanding cardinality is essential for designing normalized, expressive schemas.

Cardinality Types

Cardinality Type	Description	Example
One-to-One (1:1)	Each instance of Entity A relates to exactly one instance of Entity B	Each PERSON has one PASSPORT
One-to-Many (1:N)	One instance of Entity A relates to many instances of Entity B	One CUSTOMER places many ORDERS
Many-to-One (N:1)	Many instances of Entity A relate to one instance of Entity B	Many EMPLOYEES work in one DEPARTMENT
Many-to-Many (M:N)	Many instances of Entity A relate to many instances of Entity B	STUDENTS enroll in many COURSES; COURSES have many STUDENTS
Optional (0..1)	An instance may or may not participate in the relationship	A BOOK may have zero or one TRANSLATION
Mandatory (1..*)	An instance must participate in at least one relationship	Each ORDER must have at least one ITEM

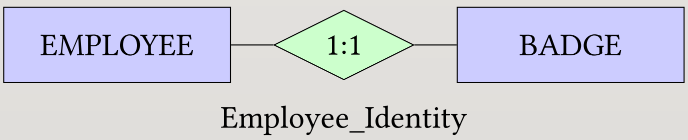
Cardinality and Relationship Roles

- Maximum cardinality defines the maximum number of relationship instances an entity can participate in.
- Minimum cardinality defines the minimum number of relationship instances an entity must participate in.
- Cardinality governs multiplicity—how many of A relate to how many of B.
- There are three types of maximum cardinality: One-to-One [1:1], One-to-Many [1:N], and Many-to-Many [N:M].
- In a one-to-many relationship, the entity on the "one" side is called the parent.
- The entity on the "many" side is called the child.
- These are HAS-A relationships—each entity instance has a relationship with another.
- Example: An EMPLOYEE has one or more COMPUTERs; each COMPUTER has one assigned EMPLOYEE.
- Cardinality constraints are essential for enforcing referential integrity and guiding schema design.

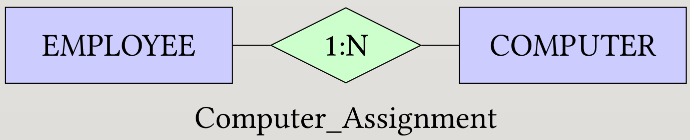
Cardinality and Relationship Roles

Concept	Definition	Example
Maximum Cardinality	Max number of relationship instances an entity can participate in	EMPLOYEE can have multiple COMPUTERs (1:N)
Minimum Cardinality	Min number of relationship instances an entity must participate in	ORDER must have at least one ITEM (1..*)
One-to-One (1:1)	Each instance of A relates to one instance of B	PERSON has one PASSPORT
One-to-Many (1:N)	One instance of A relates to many instances of B	EMPLOYEE has many COMPUTERs
Many-to-Many (N:M)	Many instances of A relate to many instances of B	STUDENTS enroll in many COURSES
Parent Entity	Entity on the "one" side of a 1:N relationship	EMPLOYEE in EMPLOYEE–COMPUTER
Child Entity	Entity on the "many" side of a 1:N relationship	COMPUTER in EMPLOYEE–COMPUTER
HAS-A Relationship	One entity instance has a relationship with another	EMPLOYEE has COMPUTER(s)

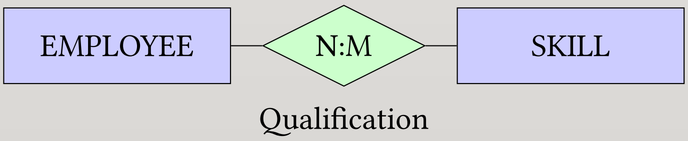
I Three Types of Minimum Cardinality



(a) Mandatory-to-Mandatory (M-M) Relaitonship



(b) Optional-to-Optional (O-O) Relaitonship



(c) Optional-to-Mandatory (O-M) Relaitonship

I Database Integrity

Database Integrity Principles

- Database integrity ensures that data remains accurate, consistent, and valid across all tables and relationships.
- It is enforced through rules, constraints, and relational logic embedded in the schema.
- Entity integrity ensures that each table has a unique, non-null primary key.
- Referential integrity ensures that foreign keys correctly reference existing primary keys in related tables.
- Domain integrity ensures that attribute values fall within valid ranges, types, and formats.
- User-defined integrity includes custom business rules specific to the application or domain.
- Integrity constraints prevent orphan records, duplicate keys, and invalid data entries.
- Together, these principles form the ritual scaffolding that protects the database from corruption and chaos.

Types of Database Integrity

Integrity Type	Definition	Example
Entity Integrity	Ensures each table has a unique, non-null primary key	EMPLOYEE.EmployeeNumber must be unique and not null
Referential Integrity	Ensures foreign keys reference valid primary keys in other tables	EMPLOYEE.Department must match DEPARTMENT.DepartmentName
Domain Integrity	Ensures attribute values are valid for their domain (type, range, format)	Salary must be a positive number; Birthdate must be a valid date
User-Defined Integrity	Enforces custom business rules beyond built-in constraints	A BOOK cannot be borrowed if its status is "Archived"

II Modeling

Data Modeling Principles

- Data modeling is the method of documenting a software system using entity-relationship diagrams (ERDs).
- ERDs represent data structures as tables, capturing the organization's business requirements.
- Data models serve as guides for database analysts and developers during system design and implementation.
- They are used across multiple stages: conceptual, logical, and physical modeling.
- A data model is a generalized, abstract blueprint for database design.
- It is easier to modify a data model than a deployed database design.
- Data models are ideal for resolving conceptual database problems before physical implementation.
- The data model corresponds to the conceptual design phase of schema development.
- Conceptual design defines entities, relationships, and constraints without concern for physical storage.
- Logical and physical designs refine the model into schemas and actual database structures.

Data Modeling Summary

Aspect	Description
Definition	Method for documenting a software system using ERDs to represent data structures
Purpose	Captures business requirements and guides database design and implementation
Stages	Conceptual, Logical, Physical
Conceptual Design	Abstract schema defining entities, relationships, and constraints
Logical Design	Refines conceptual model into database-specific schema (e.g., keys, types)
Physical Design	Implements schema in a DBMS with storage, indexing, and performance tuning
Flexibility	Easier to modify than physical database design
Tooling	Typically represented using ER diagrams (rectangles, diamonds, crow's foot glyphs)
Role in Development	Used by analysts and developers to align schema with business logic

II Diagrams

Conceptual, Logical, and Physical Diagrams

- Conceptual diagrams define the high-level structure of the data—entities, relationships, and constraints—without concern for implementation.
- They are abstract and business-focused, capturing what the data represents rather than how it is stored.
- Logical diagrams refine the conceptual model into a schema with keys, data types, and normalization rules.
- They are platform-independent and focus on relational structure, integrity constraints, and join logic.
- Physical diagrams translate the logical schema into actual database structures—tables, indexes, partitions, and storage details.
- They are platform-specific and include performance tuning, access paths, and physical storage formats.
- Conceptual = What; Logical = How; Physical = Where and With What.
- Together, these diagrams guide the full lifecycle of database design—from abstract glyph to deployed schema.

Schema Design Comparison

Aspect	Conceptual Diagram	Logical Diagram	Physical Diagram
Purpose	Define entities and relationships abstractly	Refine structure with keys, types, and constraints	Implement schema with storage and performance details
Focus	Business rules and data meaning	Relational structure and normalization	Storage, indexing, and access paths
Audience	Business analysts, domain experts	Database designers, architects	DBAs, system engineers
Platform Dependency	Independent of technology	Independent of DBMS	Specific to DBMS (e.g., Oracle, PostgreSQL)
Includes	Entities, attributes, relationships, cardinality	Tables, keys, data types, constraints	Tablespaces, indexes, partitions, triggers
Notation	ER diagrams with rectangles and diamonds	Enhanced ER or relational schema diagrams	DBMS-specific diagrams or DDL scripts
Example Glyphs	Customer, Order Customer	CUSTOMER(CustomerID PK, Name, Email)	CREATE TABLE CUSTOMER (...) WITH INDEX (...)
Modifiability	Easy to change and iterate	Moderately flexible	Harder to change once deployed

From Conceptual to Physical: Schema Evolution

- Conceptual diagrams define the abstract structure of the data—entities, relationships, and cardinality—without implementation details.
- They may or may not include attributes; regular conceptual diagrams omit them, while extended conceptual diagrams include them.
- Conceptual diagrams do not include primary keys, data types, or constraints—they are pure semantic glyphs.
- Logical diagrams refine the conceptual model by adding attributes, keys, data types, and normalization rules.
- Logical diagrams are platform-independent and focus on relational integrity and schema structure.
- Physical diagrams translate the logical schema into DBMS-specific structures—tables, indexes, partitions, and storage formats.
- Each stage builds upon the previous: Conceptual $\longrightarrow$ Logical $\longrightarrow$ Physical.
- This progression allows designers to iterate abstractly before committing to implementation.
- Conceptual diagrams are the mythic scrolls; logical diagrams are the registry emitters; physical diagrams are the deployed glyphs.

Schema Design Stages

Aspect	**Conceptual Diagram**	**Logical Diagram**	**Physical Diagram**
Purpose	Abstract model of entities and relationships	Refined schema with keys, types, and constraints	Implemented schema with storage and performance details
Includes Attributes	Optional (only in extended conceptual)	Yes, with types and constraints	Yes, with storage formats and indexing
Includes Keys	No primary keys	Includes primary and foreign keys	Includes keys plus indexes and access paths
Platform Dependency	Independent of DBMS	Independent of DBMS	Specific to DBMS (e.g., Oracle, PostgreSQL)
Notation	ERD with rectangles and diamonds; optional ellipses for attributes	Enhanced ERD or relational schema diagrams	DBMS-specific diagrams or DDL scripts
Modifiability	Highly flexible and abstract	Moderately flexible	Harder to change once deployed
Design Phase	Conceptual design	Logical design	Physical design
Example Glyphs	Pet, Visit Pet	PET(PetID PK, Species, Breed)	CREATE TABLE PET (...) WITH INDEX (...)

II Entity-Relationship Model

Entity-Relationship (E-R) Model

- The E-R model is a conceptual framework for designing and visualizing database schemas.
- It uses graphical symbols—rectangles for entities, diamonds for relationships, ellipses for attributes—to represent data structures.
- The original E-R model was introduced by Peter Chen in 1976.
- Chen's model focused on entities, relationships, and attributes, forming the foundation of conceptual schema design.
- Later extensions introduced subtypes, inheritance, and specialization—forming the extended E-R model.
- The extended E-R model includes additional constructs like generalization, aggregation, and category relationships.
- In this course, the term "E-R model" refers to the extended version.
- E-R diagrams are used to create conceptual schemas before logical and physical design.
- They are ideal for capturing business rules, data meaning, and relational structure.

E-R Model Versions

Version	Features	Introduced By
Original E-R Model	Entities, relationships, attributes; basic conceptual schema	Peter Chen (1976)
Extended E-R Model	Subtypes, inheritance, generalization, aggregation, categories	Later extensions to Chen's model
Course Usage	Refers to the extended E-R model	Adopted in modern database design curricula

Crow's Foot Notation

Symbol	Meaning	Numeric Meaning
	Mandatory-One	Exactly-One
	Optional-Many	Zero or more
	Mandatory-Many	One or More
	Optional-One	Zero or one

II Business Rules

Business Rules (3) - Characteristics

Characteristic	Explanation
Declarative	A business rule is a statement of policy and describes what a process validates but does not describe how a policy is enforced or conducted or its implementation.
Precise	A rule must have only one interpretation among all interested people, and its meaning must be clear.
Atomic	A rule is indivisible, yet sufficient.
Consistent	A business rule must be internally and externally consistent.
Expressible	A business rule must be able to be stated in natural language without misinterpretation.
Distinct	Business rules are not redundant, but a business rule may refer to other rules.

II MySQL Datatypes

Numeric Data Types

Data Type	Description
BIT(M)	M = 1 to 64 bits.
TINYINT	Range: −128 to 127.
TINYINT UNSIGNED	Range: 0 to 255.
BOOLEAN	0 = FALSE; 1 = TRUE. Synonym for TINYINT(1).
SMALLINT	Range: −32,768 to 32,767.
SMALLINT UNSIGNED	Range: 0 to 65,535.
MEDIUMINT	Range: −8,388,608 to 8,388,607.
MEDIUMINT UNSIGNED	Range: 0 to 16,777,215.
INT or INTEGER	Range: −2,147,483,648 to 2,147,483,647.
INT UNSIGNED	Range: 0 to 4,294,967,295.
BIGINT	Range: −9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
BIGINT UNSIGNED	Range: 0 to 1,844,674,073,709,551,615.
FLOAT(p)	Precision p = 0 to 53.
FLOAT or REAL(M,D)	Single-precision 4-byte float. M = display width, D = digits after decimal.
DOUBLE(M,D)	Double-precision 8-byte float. M = display width, D = digits after decimal.

MySQL Date and Time Data Types

Data Type	Description
DATE	Format: YYYY-MM-DD. Range: 1000-01-01 to 9999-12-31.
DATETIME	Format: YYYY-MM-DD HH:MM:SS. Full timestamp range.
TIMESTAMP	Range: 1970-01-01 00:00:01 to 2038-01-19 03:14:07.
TIME	Format: HH:MM:SS. Range: −838:59:59 to 838:59:59.
YEAR(M)	Range: 1901 to 2155.

MySQL String Data Types

Data Type	Description
CHAR(M)	Fixed-length string. M = 0 to 255 bytes.
VARCHAR(M)	Variable-length string. M = 0 to 65,535 bytes.
BLOB(M)	Binary Large Object. Max: 65,535 characters.
TEXT(M)	Text string. Max: 65,535 characters.
TINYBLOB / TINYTEXT	See documentation.
MEDIUMBLOB / MEDIUMTEXT	See documentation.
LONGBLOB / LONGTEXT	See documentation.
ENUM('v1', 'v2', ...)	One value chosen from list.
SET('v1', 'v2', ...)	Zero or more values chosen from list.

III Normalization

Normalization Principles

- Normalization organizes a database into tables and columns, each focused on a specific topic.
- Each table should include only the columns that support its topic.
- This reduces redundancy and improves clarity in data structure.
- Example: A spreadsheet mixing salespeople and customers serves multiple roles—identifying staff, listing clients, and mapping relationships.
- By separating these roles into distinct tables, normalization eliminates duplication and modification anomalies.
- Normalization introduces formal rules for table organization.
- These rules are staged as progressive levels called normal forms.

Normalization Summary

Aspect	Description
Purpose	Organize data into topic-specific tables to reduce redundancy
Example	Split spreadsheet roles: salespeople, customers, and their relationships
Benefit	Minimizes duplicate data and modification anomalies
Method	Apply rules that define how tables should be structured
Output	A cleaner schema aligned with business logic and technical clarity
Stages	Normal forms: progressive levels of refinement in table design

III Anomalies

Modification Anomaly Scenarios

- Updating the Chicago office to Evanston requires modifying every SalesPerson entry tied to Chicago.
- In large tables, this could mean hundreds of updates—introducing risk and inconsistency.
- If John Hunt quits and his record is deleted, the New York office information may vanish with him.
- These are examples of modification anomalies—schema weaknesses that arise from poor normalization.

Types of Modification Anomalies

Anomaly Type	Description
Insertion Anomaly	Difficulty adding data due to missing related information (e.g., can't add a new office without a SalesPerson)
Update Anomaly	Inconsistent updates when redundant data must be changed in multiple places
Deletion Anomaly	Loss of critical data when deleting a record also removes related information (e.g., deleting a SalesPerson removes office info)

III Functional Dependency

Functional Dependency Principles

- A functional dependency (FD) exists when one attribute uniquely determines another within a relation.
- Notation: $A \to B$ means that for each value of A, there is exactly one value of B.
- Functional dependencies are the foundation of normalization—they guide decomposition and anomaly resolution.
- Candidate keys are determinants in functional dependencies that uniquely identify tuples.
- Transitive dependencies (e.g., $A \to B \to C$) violate 3NF and must be resolved.
- Partial dependencies (where only part of a composite key determines an attribute) violate 2NF.
- BCNF requires that every determinant in a functional dependency be a candidate key.

Functional Dependency Summary

Aspect	Description
Definition	Relationship where one attribute determines another (e.g., $A \to B$)
Determinant	The attribute(s) on the left side of the dependency (e.g., A)
Dependent	The attribute(s) on the right side (e.g., B)
Use in Normalization	Guides decomposition into normal forms (1NF to BCNF)
Partial Dependency	Non-key attribute depends on part of a composite key (violates 2NF)
Transitive Dependency	Non-key attribute depends indirectly via another non-key (violates 3NF)
BCNF Rule	Every determinant must be a candidate key

III Determinant Value

Determinant Value Principles

- A determinant is any attribute (or set of attributes) on which another attribute is fully functionally dependent.
- In a relational table, if attribute A determines attribute B, then A is the determinant and B is the dependent.
- Determinants are foundational to identifying candidate keys and enforcing normalization rules.
- They help define functional dependencies, which are critical for decomposing tables into normal forms.
- Every candidate key is a determinant, but not every determinant is a candidate key.
- Determinants must be carefully chosen to avoid update, insertion, and deletion anomalies.
- In higher normal forms (e.g., BCNF), every determinant must be a candidate key.

Determinant Value Summary

Aspect	Description
Definition	An attribute (or set) that functionally determines another attribute
Role in Schema	Used to define functional dependencies and candidate keys
Example	EmployeeID $\longrightarrow$ EmployeeName (EmployeeID is the determinant)
Importance	Guides normalization and helps eliminate redundancy and anomalies
In BCNF	Every determinant must be a candidate key

Determinant Uniqueness Summary

Condition	Determinant Uniqueness
Candidate Key	Always unique
Primary Key	Always unique
Composite Key	Unique if combination is a candidate key
Non-Key Attribute	May determine others, but not uniquely
Partial Dependency	Not unique across entire relation

III Normalization Theory

Normalization Theory Principles

- Normalization is the process of organizing relational data to reduce redundancy and improve integrity.
- It relies on functional dependencies and determinant values to guide schema refinement.
- Each stage of normalization is called a normal form, with stricter rules at higher levels.
- Normalization eliminates modification anomalies: insertion, update, and deletion errors.
- The theory is grounded in mathematical logic and set theory, ensuring predictable schema behavior.
- Normalization is not just technical—it reflects business logic, semantic clarity, and containment purity.
- Higher normal forms (e.g., BCNF, 4NF) enforce stricter dependency rules and multivalued containment.

Normal Forms Summary

Normal Form	Condition
First Normal Form (1NF)	All attributes contain atomic (indivisible) values; no repeating groups
Second Normal Form (2NF)	1NF + no partial dependency on a subset of a candidate key
Third Normal Form (3NF)	2NF + no transitive dependency (non-key attributes depend only on keys)
Boyce-Codd Normal Form (BCNF)	Every determinant is a candidate key
Fourth Normal Form (4NF)	BCNF + no multivalued dependencies
Fifth Normal Form (5NF)	4NF + no join dependency anomalies

III Boyce-Codd Normal Form

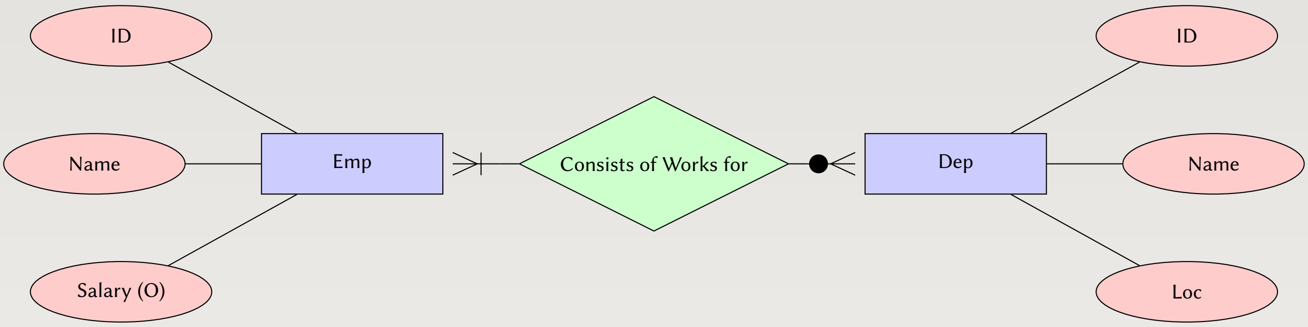
Boyce-Codd Normal Form Principles

- BCNF is a refinement of Third Normal Form (3NF) that resolves remaining anomalies caused by non-candidate key determinants.
- A relation is in BCNF if, for every non-trivial functional dependency $A \rightarrow B$, the determinant A is a candidate key.
- BCNF eliminates update and deletion anomalies that persist in 3NF when non-key attributes determine other attributes.
- It ensures that all dependencies are anchored in keys, preserving schema integrity and containment purity.
- BCNF may require decomposing relations even when they satisfy 3NF, if hidden dependencies exist.
- The form is named after Raymond Boyce and Edgar F. Codd, pioneers of relational theory.

Boyce-Codd Normal Form Summary

Aspect	Description
Definition	A relation where every determinant is a candidate key
Dependency Rule	For all $A \to B$, A must be a candidate key
Difference from 3NF	BCNF removes anomalies caused by non-key determinants still allowed in 3NF
Benefit	Stronger containment discipline, fewer anomalies, cleaner decomposition
Example Violation	If Course $\to$ Instructor but Course is not a candidate key
Resolution	Decompose into smaller relations where all determinants are keys

IV Syntax



IV Tables

IV CREATE TABLE

CREATE TABLE — Syntax Across Engines

```
-- PostgreSQL
CREATE TABLE IF NOT EXISTS users (
        id SERIAL PRIMARY KEY,
        name TEXT NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- MariaDB / MySQL
CREATE TABLE IF NOT EXISTS users (
        id INT AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(255) NOT NULL,
        created_at DATETIME DEFAULT CURRENT_TIMESTAMP
```

```sql
);

-- SQLite3
CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        created_at TEXT DEFAULT CURRENT_TIMESTAMP
);
```

CREATE TABLE — The Ritual of Genesis

- **Purpose** ⇝ Defines a new table and its columns
- **Input** ⇝ Table name, column definitions, optional constraints
- **Fallback** ⇝ IF NOT EXISTS prevents error if table already exists
- **Rendering** ⇝ Engine-specific type and default syntax
- **Use Case** ⇝ Initial schema creation, migration scripts, embedded table definitions

CREATE TABLE — Syntax Glyph Map

Engine	Auto-Increment Syntax
PostgreSQL	SERIAL or GENERATED AS IDENTITY
MariaDB / MySQL	AUTO_INCREMENT
SQLite3	INTEGER PRIMARY KEY AUTOINCREMENT
Engine	**Default Timestamp Syntax**
PostgreSQL	DEFAULT CURRENT_TIMESTAMP
MariaDB / MySQL	DEFAULT CURRENT_TIMESTAMP
SQLite3	DEFAULT CURRENT_TIMESTAMP (stored as TEXT)

CREATE TABLE — Beginner Questions

- **Should I always use IF NOT EXISTS?** ⇝ Recommended for safety, but not mandatory. It prevents errors if the table already exists.
- **Is IF NOT EXISTS the only way to avoid duplicate creation errors?** ⇝ Yes — for table creation, it's the standard conditional directive.
- **Does IF NOT EXISTS work in all engines?** ⇝ Yes — supported in PostgreSQL, MariaDB/MySQL, and SQLite3.
- **What happens if I omit IF NOT EXISTS and the table exists?** ⇝ An error is raised: "table already exists." This stops execution unless handled.
- **Can I use IF NOT EXISTS with temporary tables?** ⇝ Yes — all three engines support conditional creation of temporary tables.
- **Is CREATE TABLE reversible?** ⇝ No — you must use `DROP TABLE` to remove it.
- **Can I define constraints inside CREATE TABLE?** ⇝ Yes — you can define primary keys, foreign keys, defaults, and checks inline.
- **Can I create multiple tables in one statement?** ⇝ No — each `CREATE TABLE` must be issued separately.

IV DROP TABLE

DROP TABLE — Syntax Across Engines

```sql
-- PostgreSQL
DROP TABLE IF EXISTS users;

-- MariaDB / MySQL
DROP TABLE IF EXISTS users;

-- SQLite3
DROP TABLE IF EXISTS users;
```

DROP TABLE — The Ritual of Erasure

- **Purpose** ⇝ Removes a table and all its data from the database
- **Input** ⇝ Table name; optionally prefixed with `IF EXISTS`
- **Fallback** ⇝ IF EXISTS prevents error if the table is missing
- **Rendering** ⇝ Immediate and irreversible deletion
- **Use Case** ⇝ Schema resets, cleanup scripts, migration rollbacks

DROP TABLE — Syntax Glyph Map

Engine	Supports `IF EXISTS`
PostgreSQL	✅
MariaDB / MySQL	✅
SQLite3	✅
Engine	**Effect**
All	Deletes table and all data immediately

DROP TABLE — Beginner Questions

- **Should I always use `IF EXISTS`?** ⇝ Yes — it prevents errors if the table is already gone.
- **Is DROP TABLE reversible?** ⇝ No — once dropped, the table and its data are lost unless backed up.
- **Can I drop multiple tables at once?** ⇝ Yes — separate them with commas: `DROP TABLE IF EXISTS a, b, c;`.
- **Does it affect related tables?** ⇝ No — but foreign key constraints may block the drop unless handled.
- **Can I drop temporary tables?** ⇝ Yes — same syntax applies.
- **Does DROP TABLE delete indexes and constraints?** ⇝ Yes — all associated structures are removed.

IV ALTER TABLE

ALTER TABLE — Syntax Across Engines

```sql
-- PostgreSQL
ALTER TABLE users ADD COLUMN email TEXT;
ALTER TABLE users RENAME COLUMN name TO full_name;
ALTER TABLE users DROP COLUMN email;

-- MariaDB / MySQL
ALTER TABLE users ADD COLUMN email VARCHAR(255);
ALTER TABLE users CHANGE COLUMN name full_name VARCHAR(255);
ALTER TABLE users DROP COLUMN email;

-- SQLite3
ALTER TABLE users ADD COLUMN email TEXT;
-- Rename supported (v3.25+)
ALTER TABLE users RENAME COLUMN name TO full_name;
-- Drop column supported (v3.35+)
ALTER TABLE users DROP COLUMN email;
```

ALTER TABLE — The Ritual of Mutation

- **Purpose** ⇝ Modifies an existing table's structure
- **Input** ⇝ Table name and alteration clause (ADD, DROP, RENAME)
- **Fallback** ⇝ No built-in rollback; changes are immediate
- **Rendering** ⇝ Engine-specific syntax for renaming and dropping columns
- **Use Case** ⇝ Schema evolution, adding features, correcting column names

ALTER TABLE — Syntax Glyph Map

Action	PostgreSQL
Add Column	ADD COLUMN
Rename Column	RENAME COLUMN old TO new
Drop Column	DROP COLUMN
Action	**MariaDB / MySQL**
Add Column	ADD COLUMN
Rename Column	CHANGE COLUMN old new TYPE
Drop Column	DROP COLUMN
Action	**SQLite3**
Add Column	ADD COLUMN (v3.25+)
Rename Column	RENAME COLUMN
Drop Column	DROP COLUMN (v3.35+)

ALTER TABLE — Beginner Questions

- **Can I rename a column?** ⇝ Yes — syntax varies by engine. SQLite supports it from v3.25 onward.
- **Can I drop a column?** ⇝ Yes — PostgreSQL and MariaDB/MySQL support it. SQLite supports it from v3.35 onward.
- **Is ALTER TABLE safe?** ⇝ Changes are immediate and irreversible unless wrapped in a transaction.
- **Can I add multiple columns at once?** ⇝ Yes — separate them with commas: `ADD COLUMN a INT, ADD COLUMN b TEXT`.
- **Can I change a column's type?** ⇝ Yes — syntax varies. PostgreSQL uses `ALTER COLUMN ... TYPE`.
- **Does ALTER TABLE affect data?** ⇝ Usually no — but dropping or changing types may cause data loss.

IV Constraints

IV Named CHECK Constraint

Named CHECK Constraint — Engine-Specific Examples

- PostgreSQL ⇝ Full support for named CHECK constraints with drop-by-name
- MariaDB / MySQL ⇝ Full support for named CHECK constraints with drop-by-name (MySQL 8.0+)
- SQLite3 ⇝ Supports named CHECK constraints in table-level syntax only; cannot drop by name

```sql
-- PostgreSQL: CHECK for numeric range
CREATE TABLE products (
        id SERIAL PRIMARY KEY,
        price NUMERIC NOT NULL,
        CONSTRAINT price_check CHECK (price > 0 AND price < 10000)
);

-- PostgreSQL: CHECK for set membership
CREATE TABLE orders (
        id SERIAL PRIMARY KEY,
        status TEXT NOT NULL,
        CONSTRAINT status_check CHECK (status IN ('pending',
↪'shipped', 'delivered', 'cancelled'))
);

-- PostgreSQL: CHECK for pattern match
CREATE TABLE emails (
        id SERIAL PRIMARY KEY,
        address TEXT NOT NULL,
        CONSTRAINT email_format CHECK (address LIKE '%@%')
);

-- MariaDB / MySQL: CHECK for numeric range
CREATE TABLE products (
        id INT AUTO_INCREMENT PRIMARY KEY,
        price DECIMAL(10,2) NOT NULL,
        CONSTRAINT price_check CHECK (price > 0 AND price < 10000)
);

-- MariaDB / MySQL: CHECK for set membership
CREATE TABLE orders (
        id INT AUTO_INCREMENT PRIMARY KEY,
        status VARCHAR(20) NOT NULL,
        CONSTRAINT status_check CHECK (status IN ('pending',
↪'shipped', 'delivered', 'cancelled'))
);

-- MariaDB / MySQL: CHECK for boolean flag
CREATE TABLE users (
        id INT AUTO_INCREMENT PRIMARY KEY,
```

```sql
        is_active TINYINT(1) NOT NULL,
        CONSTRAINT active_check CHECK (is_active IN (0, 1))
);

-- SQLite3: CHECK for numeric range (table-level syntax)
CREATE TABLE products (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        price REAL NOT NULL,
        CONSTRAINT price_check CHECK (price > 0 AND price < 10000)
);

-- SQLite3: CHECK for set membership (table-level syntax)
CREATE TABLE orders (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        status TEXT NOT NULL,
        CONSTRAINT status_check CHECK (status IN ('pending', 'shipped', 'delivered', 'cancelled'))
);

-- SQLite3: CHECK for cross-column logic (table-level syntax)
CREATE TABLE bookings (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        start_date TEXT NOT NULL,
        end_date TEXT NOT NULL,
        CONSTRAINT date_order CHECK (start_date < end_date)
);
```

Named CHECK Constraint — The Ritual of Validation

- **Purpose** ⇝ Assigns a name to a CHECK constraint for clarity, debugging, and schema control
- **Input** ⇝ CONSTRAINT name CHECK (expression)
- **Fallback** ⇝ Unnamed CHECK constraints are valid but harder to reference or drop
- **Rendering** ⇝ Constraint name appears in error messages and schema inspection
- **Use Case** ⇝ Validating column ranges, formats, or logical conditions with named enforcement

Named CHECK Constraint — Syntax Glyph Map

Engine	Supports Named CHECK Constraints
PostgreSQL	✅ fully supported
MariaDB / MySQL	✅ fully supported
SQLite3	✅ supported in table-level syntax
Engine	**Can Drop by Name**
PostgreSQL	✅ `ALTER TABLE ... DROP CONSTRAINT name`
MariaDB / MySQL	✅ `ALTER TABLE ... DROP CHECK name`
SQLite3	❌ must recreate table to drop constraints

Named CHECK Constraint — Beginner Questions

- **Why name a `CHECK` constraint?** ⇝ It helps with debugging, dropping, and documenting validation rules.
- **Is naming required?** ⇝ No — constraints can be anonymous, but naming is recommended.
- **Can I drop a `CHECK` constraint by name?** ⇝ Yes — in PostgreSQL and MariaDB/MySQL. SQLite requires table recreation.
- **Can I use any name?** ⇝ Yes — but it must be unique within the table.
- **Does the name affect behavior?** ⇝ No — it's purely for identification.
- **Can I name other constraints too?** ⇝ Yes — `FOREIGN KEY`, `UNIQUE`, and `PRIMARY KEY` can also be named.

IV Named DEFAULT Constraint

Named DEFAULT Constraint — Syntax Across Engines

- PostgreSQL ⇝ Supports named `DEFAULT` constraints via `ALTER TABLE ... ADD CONSTRAINT`
- MariaDB / MySQL ⇝ Does not support naming `DEFAULT` constraints directly — defaults are column-level only
- SQLite3 ⇝ Does not support naming `DEFAULT` constraints — defaults are inline only

```sql
-- PostgreSQL: Named DEFAULT constraint (table-level)
CREATE TABLE users (
	id SERIAL PRIMARY KEY,
	is_active BOOLEAN NOT NULL,
	CONSTRAINT default_active DEFAULT TRUE FOR is_active
);

-- PostgreSQL: Named DEFAULT constraint via ALTER TABLE
ALTER TABLE users
ADD CONSTRAINT default_active DEFAULT TRUE FOR is_active;

-- MariaDB / MySQL: DEFAULT value (unnamed, column-level only)
CREATE TABLE users (
	id INT AUTO_INCREMENT PRIMARY KEY,
	is_active BOOLEAN NOT NULL DEFAULT TRUE
);

-- SQLite3: DEFAULT value (unnamed, inline only)
CREATE TABLE users (
	id INTEGER PRIMARY KEY AUTOINCREMENT,
	is_active BOOLEAN NOT NULL DEFAULT 1
);
```

Named DEFAULT Constraint — The Ritual of Prepopulation

- **Purpose** ⇝ Assigns a default value to a column when no explicit value is provided during insertion
- **Input** ⇝ `CONSTRAINT name DEFAULT value` (engine-dependent)
- **Fallback** ⇝ Unnamed defaults are valid but harder to reference or document
- **Rendering** ⇝ Default value is automatically inserted unless overridden
- **Use Case** ⇝ Prepopulating timestamps, flags, counters, or status fields with consistent defaults

Named DEFAULT Constraint — Syntax Glyph Map

Engine	Supports Named DEFAULT Constraints
PostgreSQL	✅ supported via `ALTER TABLE ... ADD CONSTRAINT name DEFAULT value`
MariaDB / MySQL	❌ does not support naming DEFAULT constraints directly
SQLite3	❌ does not support naming DEFAULT constraints directly
Engine	**Can Drop DEFAULT by Name**
PostgreSQL	✅ `ALTER TABLE ... DROP CONSTRAINT name`
MariaDB / MySQL	❌ must alter column directly
SQLite3	❌ must recreate table to change default

Named DEFAULT Constraint — Beginner Questions

- **Why use a DEFAULT constraint?** ⇝ To automatically assign values when none are provided.
- **Can I name a DEFAULT constraint?** ⇝ Only in PostgreSQL — other engines do not support naming directly.
- **Can I override the default?** ⇝ Yes — any explicit value will replace the default.
- **Can I drop a default by name?** ⇝ Only in PostgreSQL — others require column alteration or table recreation.
- **Is naming required?** ⇝ No — defaults work without names, but naming improves clarity and control.
- **Can I use expressions as defaults?** ⇝ Yes — engines support literals, functions like `CURRENT_TIMESTAMP`, and booleans.

IV Named FOREIGN KEY Constraint

Named FOREIGN KEY Constraint — Syntax Across Engines

- **PostgreSQL** ⇝ Supports named `FOREIGN KEY` constraints with full syntax and drop-by-name
- **MariaDB / MySQL** ⇝ Supports named `FOREIGN KEY` constraints with full syntax and drop-by-name
- **SQLite3** ⇝ Supports named `FOREIGN KEY` constraints in table-level syntax only; cannot drop by name

```sql
-- PostgreSQL: Named FOREIGN KEY constraint
CREATE TABLE customers (
        id SERIAL PRIMARY KEY,
        name TEXT NOT NULL
);

CREATE TABLE orders (
        id SERIAL PRIMARY KEY,
        customer_id INT NOT NULL,
        CONSTRAINT fk_customer FOREIGN KEY (customer_id)
    REFERENCES customers(id)
);

-- PostgreSQL: Drop named FOREIGN KEY constraint
ALTER TABLE orders DROP CONSTRAINT fk_customer;
```

```sql
-- MariaDB / MySQL: Named FOREIGN KEY constraint
CREATE TABLE customers (
        id INT AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(255) NOT NULL
);

CREATE TABLE orders (
        id INT AUTO_INCREMENT PRIMARY KEY,
        customer_id INT NOT NULL,
        CONSTRAINT fk_customer FOREIGN KEY (customer_id) REFERENCES customers(id)
);

-- MariaDB / MySQL: Drop named FOREIGN KEY constraint
ALTER TABLE orders DROP FOREIGN KEY fk_customer;

-- SQLite3: Named FOREIGN KEY constraint (table-level only)
CREATE TABLE customers (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL
);

CREATE TABLE orders (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        customer_id INT NOT NULL,
        CONSTRAINT fk_customer FOREIGN KEY (customer_id) REFERENCES customers(id)
);

-- SQLite3: Drop FOREIGN KEY constraint — not supported by name
-- Must recreate the table to remove or modify constraints
```

Named FOREIGN KEY Constraint — The Ritual of Referential Binding

- **Purpose** ⇝ Assigns a name to a FOREIGN KEY constraint for clarity, debugging, and schema control
- **Input** ⇝ CONSTRAINT name FOREIGN KEY (col) REFERENCES table(col)
- **Fallback** ⇝ Unnamed constraints are valid but harder to inspect, drop, or document
- **Rendering** ⇝ Constraint name appears in error messages, schema inspection, and migration tooling
- **Use Case** ⇝ Enforcing referential integrity with traceable lineage and drop-by-name support

Named FOREIGN KEY Constraint — Syntax Glyph Map

Engine	Supports Named FOREIGN KEY Constraints
PostgreSQL	✅ fully supported
MariaDB / MySQL	✅ fully supported
SQLite3	✅ supported in table-level syntax only
Engine	**Can Drop by Name**
PostgreSQL	✅ `ALTER TABLE ... DROP CONSTRAINT name`
MariaDB / MySQL	✅ `ALTER TABLE ... DROP FOREIGN KEY name`
SQLite3	❌ must recreate table to drop constraints

Named FOREIGN KEY Constraint — Beginner Questions

- **Why name a FOREIGN KEY constraint?** ⇝ It helps with debugging, dropping, and documenting relationships.
- **Is naming required?** ⇝ No — constraints can be anonymous, but naming is recommended.
- **Can I drop a constraint by name?** ⇝ Yes — in PostgreSQL and MariaDB/MySQL. SQLite requires table recreation.
- **Can I use any name?** ⇝ Yes — but it must be unique within the table.
- **Does the name affect behavior?** ⇝ No — it's purely for identification.
- **Can I name other constraints too?** ⇝ Yes — `CHECK`, `UNIQUE`, and `PRIMARY KEY` can also be named.

IV PRIMARY KEY vs UNIQUE NOT NULL

PRIMARY KEY vs UNIQUE NOT NULL — The Ritual of Identity

- **Purpose** ⇝ Both enforce uniqueness — but PRIMARY KEY defines the row's identity, while UNIQUE NOT NULL enforces alternate uniqueness
- **Input** ⇝ PRIMARY KEY (col) vs UNIQUE (col) + NOT NULL
- **Fallback** ⇝ UNIQUE allows multiple per table; PRIMARY KEY is singular and implicit NOT NULL
- **Rendering** ⇝ PRIMARY KEY is the default clustering/index key in many engines
- **Use Case** ⇝ Use PRIMARY KEY for row identity; use UNIQUE NOT NULL for alternate keys or constraints

PRIMARY KEY vs UNIQUE NOT NULL — Syntax Glyph Map

Aspect	PRIMARY KEY	UNIQUE + NOT NULL
Uniqueness	Enforced	Enforced
Nullability	Implicitly NOT NULL	Must be declared NOT NULL explicitly
Multiplicity	Only one per table	Multiple allowed
Naming	Can be named	Can be named
Indexing	Often clustered index (engine-specific)	Usually non-clustered index
Identity Role	Defines row identity	Alternate candidate key
Composite Support	Yes	Yes

PRIMARY KEY vs UNIQUE NOT NULL — Syntax Across Engines

```
-- PostgreSQL / MySQL / SQLite: PRIMARY KEY
CREATE TABLE users (
        id SERIAL PRIMARY KEY,
        username TEXT UNIQUE NOT NULL
);

-- PostgreSQL / MySQL / SQLite: UNIQUE + NOT NULL (alternate key)
CREATE TABLE products (
```

```sql
        sku TEXT NOT NULL,
        name TEXT NOT NULL,
        CONSTRAINT unique_sku UNIQUE (sku)
);
```

PRIMARY KEY vs UNIQUE NOT NULL — Beginner Questions

- **Can I have more than one PRIMARY KEY?** ⇝ No — only one per table.
- **Can I have multiple UNIQUE NOT NULL columns?** ⇝ Yes — each defines a separate uniqueness rule.
- **Is PRIMARY KEY always NOT NULL?** ⇝ Yes — it is enforced implicitly.
- **Do I need to write NOT NULL with PRIMARY KEY?** ⇝ No — it's automatic.
- **Which one should I use for identity?** ⇝ Always use `PRIMARY KEY` for the main identifier.
- **Can I name both constraints?** ⇝ Yes — both can be named using `CONSTRAINT name ...`
- **Does a table need a PRIMARY KEY?** ⇝ No — but it's strongly recommended. Without it, rows lack guaranteed identity and indexing support.

IV Key Types

Key Types — Identity Rituals in Relational Design

- **Composite Key** ⇝ A **PRIMARY KEY** composed of multiple columns. Used when no single column guarantees uniqueness.
- **Surrogate Key** ⇝ An artificial identifier (e.g., `id SERIAL`, `UUID`) with no business meaning. Used for simplicity, indexing, and mutation safety.
- **Natural Key** ⇝ A real-world identifier with domain meaning (e.g., `email`, `SSN`). Used when uniqueness is guaranteed by business logic.

Key Types — Syntax Glyph Map

Aspect	Composite Key	Surrogate Key	Natural Key
Definition	Multi-column PRIMARY KEY	Engine-generated unique ID	Domain-derived unique value
Business Meaning	Yes (combined)	No	Yes
Portability	Schema-dependent	Engine-neutral	Domain-dependent
Indexing	Manual or implicit	Often clustered	Depends on usage
Mutation Risk	Low	Low	High (if business rules change)
Multiplicity	One per table	One per table	Often used with UNIQUE

Natural Key — Syntax Across Engines

- **Natural Key** ⇝ A real-world identifier with domain meaning (e.g., email, SSN)
- **Use Case** ⇝ When business logic guarantees uniqueness and mutation risk is low

```sql
-- PostgreSQL
CREATE TABLE countries (
    iso_code CHAR(2) PRIMARY KEY,
    name TEXT NOT NULL
);

-- MariaDB / MySQL
CREATE TABLE countries (
    iso_code CHAR(2) PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

-- SQLite3
CREATE TABLE countries (
    iso_code TEXT PRIMARY KEY,
    name TEXT NOT NULL
);
```

Surrogate Key — Syntax Across Engines

- Surrogate Key ⇝ An artificial identifier with no business meaning (e.g., `id SERIAL`, `UUID`)
- Use Case ⇝ When simplicity, indexing, and mutation safety are prioritized

```sql
-- PostgreSQL
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email TEXT UNIQUE NOT NULL
);

-- MariaDB / MySQL
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL
);

-- SQLite3
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    email TEXT UNIQUE NOT NULL
);
```

Composite Key — Syntax Across Engines

- Composite Key ⇝ A `PRIMARY KEY` composed of multiple columns
- Use Case ⇝ When no single column guarantees uniqueness — identity is defined by column combination

```sql
-- PostgreSQL
CREATE TABLE enrollments (
    student_id INT NOT NULL,
    course_id INT NOT NULL,
    enrolled_on DATE NOT NULL,
    CONSTRAINT pk_enrollment PRIMARY KEY (student_id, course_id)
);

-- MariaDB / MySQL
CREATE TABLE enrollments (
    student_id INT NOT NULL,
    course_id INT NOT NULL,
    enrolled_on DATE NOT NULL,
    CONSTRAINT pk_enrollment PRIMARY KEY (student_id, course_id)
);

-- SQLite3
```

```sql
CREATE TABLE enrollments (
    student_id INT NOT NULL,
    course_id INT NOT NULL,
    enrolled_on TEXT NOT NULL,
    CONSTRAINT pk_enrollment PRIMARY KEY (student_id, course_id)
);
```

Key Types — Identity Rituals in Relational Design

- **Composite Key** ⇝ A PRIMARY KEY composed of multiple columns. Used when no single column guarantees uniqueness.
- **Surrogate Key** ⇝ An artificial identifier (e.g., id SERIAL, UUID) with no business meaning. Used for simplicity, indexing, and mutation safety.
- **Natural Key** ⇝ A real-world identifier with domain meaning (e.g., email, SSN). Used when uniqueness is guaranteed by business logic.

Key Types — Syntax Glyph Map

Aspect	Composite Key	Surrogate Key	Natural Key
Definition	Multi-column PRIMARY KEY	Engine-generated unique ID	Domain-derived unique value
Business Meaning	Yes (combined)	No	Yes
Portability	Schema-dependent	Engine-neutral	Domain-dependent
Indexing	Manual or implicit	Often clustered	Depends on usage
Mutation Risk	Low	Low	High (if business rules change)
Multiplicity	One per table	One per table	Often used with UNIQUE
Drop Complexity	Must drop all columns	Single column	Single column
Naming	Can be named	Can be named	Can be named

Key Types — Beginner Questions

- **Which key type should I use?** ⇝ Use Surrogate for simplicity, Natural for domain clarity, Composite when identity spans multiple columns.
- **Can I mix key types?** ⇝ Yes — for example, a surrogate PRIMARY KEY with a natural UNIQUE constraint.
- **Is a surrogate key always numeric?** ⇝ No — it can be a UUID, hash, or any unique token.
- **Can a composite key include a surrogate?** ⇝ Yes — but it's rare. Composite keys usually reflect domain logic.
- **Can I change a natural key later?** ⇝ Yes — but it risks breaking relationships and should be done cautiously.
- **Does every table need a key?** ⇝ Yes — a PRIMARY KEY or equivalent is essential for row identity and indexing.

IV Cardinality

IV Optional One to Mandatory Many

Optional One to Mandatory Many — Syntax Across Engines

- **order table** ⇝ The order table is the mandatory many — it can contain many rows, and each row may optionally reference a user
- **user table** ⇝ optional side — meaning a child row in orders may or may not link to a user.

```sql
-- PostgreSQL
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL
);

CREATE TABLE orders (
    user_id INT,
    FOREIGN KEY (user_id) REFERENCES users(id)
);

-- MariaDB / MySQL
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);
```

```sql

CREATE TABLE orders (
    user_id INT,
    FOREIGN KEY (user_id) REFERENCES users(id)
);

-- SQLite3
PRAGMA foreign_keys = ON;

CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL
);

CREATE TABLE orders (
    user_id INT,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Optional One to Mandatory Many — The Ritual of Open Belonging

- **Purpose** ⇝ Allows many child rows to reference one parent, but the link is optional
- **Input** ⇝ Foreign key column without NOT NULL
- **Fallback** ⇝ Child rows may exist without a parent; no enforcement until value is present
- **Rendering** ⇝ Engine enforces referential integrity only when a value is supplied
- **Use Case** ⇝ Orders that may be anonymous, comments without authors, drafts without owners

Optional One to Mandatory Many — Syntax Glyph Map

Engine	Foreign Key Column Nullable
PostgreSQL	✅ omit `NOT NULL`
MariaDB / MySQL	✅ omit `NOT NULL`
SQLite3	✅ omit `NOT NULL`
Engine	**Parent Key Required**
PostgreSQL	✅ must be `PRIMARY KEY` or `UNIQUE NOT NULL`
MariaDB / MySQL	✅ must be `PRIMARY KEY` or `UNIQUE NOT NULL`
SQLite3	✅ must be `PRIMARY KEY` or `UNIQUE NOT NULL`

Optional One to Mandatory Many — Beginner Questions

- **What makes the relationship optional?** ⇝ The child foreign key column allows `NULL`.
- **What makes it many?** ⇝ No `UNIQUE` constraint — multiple children can link to the same parent.
- **Can a child row exist without a parent?** ⇝ Yes — the foreign key can be `NULL`.
- **What happens if I insert a value that doesn't match a parent?** ⇝ The insert fails — referential integrity is enforced when a value is present.
- **Can I later enforce mandatory linkage?** ⇝ Yes — add `NOT NULL` to the foreign key column.
- **Is this the default pattern?** ⇝ Yes — most foreign keys are optional unless constrained.

IV Optional One to Mandatory One

Mandatory One to Optional One — Syntax Across Engines

- passport table ⇝ The passport table is the mandatory one — every passport must link to a user
- user table ⇝ The user table is the optional one — a user may or may not have a passport

```sql
-- PostgreSQL
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL
);

CREATE TABLE passports (
    user_id INT UNIQUE NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id)
);

-- MariaDB / MySQL
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE passports (
    user_id INT UNIQUE NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id)
);

-- SQLite3
PRAGMA foreign_keys = ON;

CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL
);

CREATE TABLE passports (
    user_id INT UNIQUE NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Mandatory One to Optional One — The Ritual of Singular Ownership

- **Purpose** ⇝ Enforces that every child row must link to one and only one parent, while the parent may have zero or one child
- **Input** ⇝ Foreign key column with `UNIQUE` and `NOT NULL`
- **Fallback** ⇝ Parent may exist without a child; child cannot exist without a parent
- **Rendering** ⇝ Enforced uniqueness and mandatory linkage on the child's foreign key column
- **Use Case** ⇝ Passports, licenses, or metadata that must belong to a person, but not all persons have one

Mandatory One to Optional One — Syntax Glyph Map

Engine	Foreign Key Column Mandatory
PostgreSQL	✅ use `NOT NULL`
MariaDB / MySQL	✅ use `NOT NULL`
SQLite3	✅ use `NOT NULL`
Engine	**Enforces One-to-One via UNIQUE**
PostgreSQL	✅ use `UNIQUE` on foreign key
MariaDB / MySQL	✅ use `UNIQUE` on foreign key
SQLite3	✅ use `UNIQUE` on foreign key

Mandatory One to Optional One — Beginner Questions

- **What makes the child mandatory?** ⇝ The foreign key column is marked NOT NULL
- **What enforces one-to-one?** ⇝ The foreign key column is marked UNIQUE
- **Can a passport exist without a user?** ⇝ No — the foreign key must reference a valid user
- **Can a user exist without a passport?** ⇝ Yes — no constraints enforce ownership
- **Can two passports link to the same user?** ⇝ No — UNIQUE prevents duplicate links
- **Is this enforced by the engine?** ⇝ Yes — all engines enforce UNIQUE and NOT NULL constraints

IV Optional One to Optional One

Optional One to Optional One — Syntax Across Engines

- **locker table** ⇝ The locker table is the optional one — a locker may or may not be assigned
- **student table** ⇝ The student table is also optional — a student may or may not have a locker

```sql
-- PostgreSQL
CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL
);

CREATE TABLE lockers (
    id SERIAL PRIMARY KEY,
    student_id INT UNIQUE,
    FOREIGN KEY (student_id) REFERENCES students(id)
);

-- MariaDB / MySQL
CREATE TABLE students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE lockers (
    id INT AUTO_INCREMENT PRIMARY KEY,
```

```
    student_id INT UNIQUE,
    FOREIGN KEY (student_id) REFERENCES students(id)
);

-- SQLite3
PRAGMA foreign_keys = ON;

CREATE TABLE students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL
);

CREATE TABLE lockers (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    student_id INT UNIQUE,
    FOREIGN KEY (student_id) REFERENCES students(id)
);
```

Optional One to Optional One — The Ritual of Loose Pairing

- **Purpose** ⇝ Allows a child row to optionally link to one and only one parent, and the parent may have zero or one child
- **Input** ⇝ Foreign key column with UNIQUE and nullable
- **Fallback** ⇝ Neither side is required; linkage is optional and singular
- **Rendering** ⇝ Enforced uniqueness on the child's foreign key column; no mandatory linkage
- **Use Case** ⇝ Lockers that may be unassigned, and students who may or may not have a locker

Optional One to Optional One — Syntax Glyph Map

Engine	Foreign Key Column Nullable
PostgreSQL	✅ omit NOT NULL
MariaDB / MySQL	✅ omit NOT NULL
SQLite3	✅ omit NOT NULL
Engine	**Enforces One-to-One via UNIQUE**
PostgreSQL	✅ use UNIQUE on foreign key
MariaDB / MySQL	✅ use UNIQUE on foreign key
SQLite3	✅ use UNIQUE on foreign key

Optional One to Optional One — Beginner Questions

- **What makes both sides optional?** ⇝ The foreign key column allows NULL, and the parent has no constraints enforcing linkage
- **What enforces one-to-one?** ⇝ The foreign key column is marked UNIQUE
- **Can a locker exist without a student?** ⇝ Yes — the foreign key may be NULL
- **Can a student exist without a locker?** ⇝ Yes — no constraints enforce ownership
- **Can two lockers link to the same student?** ⇝ No — UNIQUE prevents duplicate links
- **Is this enforced by the engine?** ⇝ Yes — all engines enforce UNIQUE constraints

IV Mandatory One to Mandatory One

Mandatory One to Mandatory One — Syntax Across Engines

- **account table** ⇝ The account table is mandatory — every account must link to one settings row
- **account_settings table** ⇝ The settings table is mandatory — every settings row must link to one account

```
-- PostgreSQL
CREATE TABLE account_settings (
    id SERIAL PRIMARY KEY,
```

```sql
    theme TEXT NOT NULL
);

CREATE TABLE accounts (
    id SERIAL PRIMARY KEY,
    username TEXT NOT NULL,
    settings_id INT UNIQUE NOT NULL,
    FOREIGN KEY (settings_id) REFERENCES account_settings(id)
);

-- MariaDB / MySQL
CREATE TABLE account_settings (
    id INT AUTO_INCREMENT PRIMARY KEY,
    theme VARCHAR(255) NOT NULL
);

CREATE TABLE accounts (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL,
    settings_id INT UNIQUE NOT NULL,
    FOREIGN KEY (settings_id) REFERENCES account_settings(id)
);

-- SQLite3
PRAGMA foreign_keys = ON;

CREATE TABLE account_settings (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    theme TEXT NOT NULL
);

CREATE TABLE accounts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL,
    settings_id INT UNIQUE NOT NULL,
    FOREIGN KEY (settings_id) REFERENCES account_settings(id)
);
```

Mandatory One to Mandatory One — The Ritual of Singular Binding

- **Purpose** ⇝ Enforces that each row in both tables must be linked to exactly one row in the other
- **Input** ⇝ Foreign key column marked UNIQUE and NOT NULL
- **Fallback** ⇝ Neither side may exist without the other — linkage is mandatory and singular
- **Rendering** ⇝ One-to-one enforced by UNIQUE and NOT NULL on foreign key
- **Use Case** ⇝ Accounts and their settings in a system where both must exist together

Mandatory One to Mandatory One — Syntax Glyph Map

Engine	**Foreign Key Column Mandatory**
PostgreSQL	✅ use NOT NULL
MariaDB / MySQL	✅ use NOT NULL
SQLite3	✅ use NOT NULL
Engine	**Enforces One-to-One via UNIQUE**
PostgreSQL	✅ use UNIQUE on foreign key
MariaDB / MySQL	✅ use UNIQUE on foreign key
SQLite3	✅ use UNIQUE on foreign key

Mandatory One to Mandatory One — Beginner Questions

- **What makes both sides mandatory?** ⇝ The foreign key column is marked NOT NULL, and each row must be linked.
- **What enforces one-to-one?** ⇝ The foreign key column is marked UNIQUE
- **Can an account exist without settings?** ⇝ No — the foreign key must reference a valid settings row.
- **Can settings exist without an account?** ⇝ No — every settings row must be linked to an account.
- **Can two accounts share the same settings?** ⇝ No — UNIQUE prevents duplicate links.
- **Is this enforced by the engine?** ⇝ Yes — all engines enforce UNIQUE and NOT NULL constraints

IV Mandatory Many to Mandatory Many

Mandatory Many to Mandatory Many — Syntax Across Engines

- **student table** ⇝ Each student must be enrolled in at least one course
- **course table** ⇝ Each course must have at least one enrolled student
- **junction table** ⇝ Enforces many-to-many linkage with mandatory participation on both sides

```sql
-- PostgreSQL
CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL
);

CREATE TABLE courses (
    id SERIAL PRIMARY KEY,
    title TEXT NOT NULL
);

CREATE TABLE enrollments (
    student_id INT NOT NULL,
    course_id INT NOT NULL,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES students(id),
    FOREIGN KEY (course_id) REFERENCES courses(id)
);
```

```sql
-- MariaDB / MySQL
CREATE TABLE students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE courses (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255) NOT NULL
);

CREATE TABLE enrollments (
    student_id INT NOT NULL,
    course_id INT NOT NULL,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES students(id),
    FOREIGN KEY (course_id) REFERENCES courses(id)
);

-- SQLite3
PRAGMA foreign_keys = ON;

CREATE TABLE students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL
);

CREATE TABLE courses (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL
);

CREATE TABLE enrollments (
    student_id INT NOT NULL,
    course_id INT NOT NULL,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES students(id),
    FOREIGN KEY (course_id) REFERENCES courses(id)
);
```

Mandatory Many to Mandatory Many — The Ritual of Mutual Participation

- **Purpose** ⇝ Enforces that each row in both tables must participate in at least one relationship
- **Input** ⇝ Junction table with NOT NULL foreign keys and composite PRIMARY KEY
- **Fallback** ⇝ SQL cannot enforce that each student or course has at least one link — this must be handled in application logic
- **Rendering** ⇝ Many-to-many linkage with mandatory foreign keys; participation must be ensured externally
- **Use Case** ⇝ Students and courses where both must be linked — no unassigned students or empty courses

Mandatory Many to Mandatory Many — Syntax Glyph Map

Engine	Junction Foreign Keys Mandatory
PostgreSQL	✅ use NOT NULL on both foreign keys
MariaDB / MySQL	✅ use NOT NULL on both foreign keys
SQLite3	✅ use NOT NULL on both foreign keys
Engine	**Participation Enforced**
PostgreSQL	❌ not enforceable in SQL constraints
MariaDB / MySQL	❌ not enforceable in SQL constraints
SQLite3	❌ not enforceable in SQL constraints

Mandatory Many to Mandatory Many — Beginner Questions

- **What makes it many-to-many?** ⇝ The junction table allows multiple students per course and multiple courses per student.
- **What makes it mandatory?** ⇝ The foreign keys are NOT NULL, and business rules require participation.
- **Can a student exist without a course?** ⇝ SQL allows it — but application logic must prevent it.
- **Can a course exist without students?** ⇝ SQL allows it — but application logic must prevent it.
- **Can SQL enforce mandatory participation?** ⇝ No — this must be enforced by triggers or application logic.
- **Is this a common pattern?** ⇝ Yes — especially in enrollment, tagging, and assignment systems.

IV Mandatory One to Mandatory Many

Mandatory One to Mandatory Many — Syntax Across Engines

- **department table** ⇝ The department table is mandatory — each department must have at least one employee
- **employee table** ⇝ The employee table is mandatory — each employee must belong to a department

```sql
-- PostgreSQL
CREATE TABLE departments (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL
);

CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    department_id INT NOT NULL,
    FOREIGN KEY (department_id) REFERENCES departments(id)
);

-- MariaDB / MySQL
CREATE TABLE departments (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

```

```sql
CREATE TABLE employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    department_id INT NOT NULL,
    FOREIGN KEY (department_id) REFERENCES departments(id)
);

-- SQLite3
PRAGMA foreign_keys = ON;

CREATE TABLE departments (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL
);

CREATE TABLE employees (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    department_id INT NOT NULL,
    FOREIGN KEY (department_id) REFERENCES departments(id)
);
```

Mandatory One to Mandatory Many — The Ritual of Required Multiplicity

- **Purpose** ⇝ Enforces that each child must link to a parent, and each parent must have at least one child
- **Input** ⇝ Foreign key column marked `NOT NULL`
- **Fallback** ⇝ SQL cannot enforce that each department has employees — this must be handled in application logic
- **Rendering** ⇝ Each employee must belong to a department; each department must have employees
- **Use Case** ⇝ Departments and employees in a system where both must exist and be linked

Mandatory One to Mandatory Many — Syntax Glyph Map

Engine	Child Must Link to Parent
PostgreSQL	✅ use NOT NULL on foreign key
MariaDB / MySQL	✅ use NOT NULL on foreign key
SQLite3	✅ use NOT NULL on foreign key
Engine	**Parent Must Have Children**
PostgreSQL	❌ not enforceable in SQL constraints
MariaDB / MySQL	❌ not enforceable in SQL constraints
SQLite3	❌ not enforceable in SQL constraints

Mandatory One to Mandatory Many — Beginner Questions

- **What makes the relationship mandatory?** ⇝ Each employee must link to a department, and each department must have employees.
- **Can I insert an employee without a department?** ⇝ No — the foreign key must reference a valid department.
- **Can I insert a department without employees?** ⇝ Yes — but SQL will not enforce that employees must follow.
- **Can SQL enforce that every department has employees?** ⇝ No — this must be enforced by application logic or triggers.
- **Is this a common pattern?** ⇝ Yes — especially in organizational systems where entities must be grouped.

IV Recovery

IV COMMIT — Syntax Across Engines

Atomic Transfer Between Accounts

- accounts table ⇝ Tracks user balances. Funds are debited from one account and credited to another.
- transaction block ⇝ Ensures both updates succeed or none are applied.

```sql
-- PostgreSQL
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;

-- MariaDB / MySQL
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;

-- SQLite3
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

COMMIT — The Ritual of Finalizing Changes

- **Purpose** ⇝ Permanently applies all changes made since BEGIN or START TRANSACTION
- **Input** ⇝ COMMIT
- **Fallback** ⇝ Without COMMIT, changes remain uncommitted and may be lost or rolled back
- **Rendering** ⇝ COMMIT ends the transaction and releases locks
- **Use Case** ⇝ Used when all operations succeed and consistency is confirmed

COMMIT — Syntax Glyph Map

Engine	COMMIT Support
PostgreSQL	✅ full support for COMMIT and transactional control
MariaDB / MySQL	✅ full support for COMMIT and rollback logic
SQLite3	✅ supports COMMIT and nested SAVEPOINTs
Engine	**COMMIT Behavior**
PostgreSQL	Applies all changes and ends transaction; locks released
MariaDB / MySQL	Same; triggers durability and indexing updates
SQLite3	Same; ends transaction and finalizes WAL segment

COMMIT — Beginner Questions

- **When should I use COMMIT?** ⇝ After all operations succeed and the transaction is ready to be finalized.
- **What happens after COMMIT?** ⇝ All changes become permanent and visible to other sessions.
- **Can I undo a COMMIT?** ⇝ No — once committed, changes cannot be rolled back.
- **Does COMMIT release locks?** ⇝ Yes — it ends the transaction and frees associated locks.
- **Is COMMIT automatic?** ⇝ In autocommit mode, yes — otherwise it must be explicit.
- **Can I commit part of a transaction?** ⇝ No — COMMIT finalizes the entire transaction. Use SAVEPOINT for partial control.

Inventory Adjustment — Reverted on Failure

- **inventory table** ⇝ Tracks item quantities. A failed update must be reverted to preserve consistency.
- **transaction block** ⇝ ROLLBACK ensures no partial changes persist if logic fails.

```sql
-- PostgreSQL
BEGIN;
UPDATE inventory SET quantity = quantity - 1 WHERE item_id = 42;
-- Something goes wrong
```

```sql
ROLLBACK;

-- MariaDB / MySQL
START TRANSACTION;
UPDATE inventory SET quantity = quantity - 1 WHERE item_id = 42;
-- Error detected
ROLLBACK;

-- SQLite3
BEGIN TRANSACTION;
UPDATE inventory SET quantity = quantity - 1 WHERE item_id = 42;
-- Abort transaction
ROLLBACK;
```

IV ROLLBACK — Syntax Across Engines

Inventory Adjustment — Reverted on Failure

- **inventory table** ⇝ Tracks item quantities. A failed update must be reverted to preserve consistency.
- **transaction block** ⇝ ROLLBACK ensures no partial changes persist if logic fails.

```sql
-- PostgreSQL
BEGIN;
UPDATE inventory SET quantity = quantity - 1 WHERE item_id = 42;
-- Something goes wrong
ROLLBACK;

-- MariaDB / MySQL
START TRANSACTION;
UPDATE inventory SET quantity = quantity - 1 WHERE item_id = 42;
-- Error detected
ROLLBACK;

-- SQLite3
BEGIN TRANSACTION;
UPDATE inventory SET quantity = quantity - 1 WHERE item_id = 42;
-- Abort transaction
ROLLBACK;
```

ROLLBACK — The Ritual of Undoing All Changes

- **Purpose** ⇝ Cancels all changes made since BEGIN or START TRANSACTION
- **Input** ⇝ ROLLBACK
- **Fallback** ⇝ Without ROLLBACK, failed transactions may leave partial or corrupt state
- **Rendering** ⇝ ROLLBACK discards all uncommitted changes and resets the transaction context
- **Use Case** ⇝ Used when an error, constraint violation, or logic failure is detected during a transaction

ROLLBACK — Syntax Glyph Map

Engine	ROLLBACK Support
PostgreSQL	✅ full support for ROLLBACK and ROLLBACK TO SAVEPOINT
MariaDB / MySQL	✅ full support for ROLLBACK and SAVEPOINT rollback
SQLite3	✅ supports ROLLBACK and nested SAVEPOINT rollback
Engine	**ROLLBACK Behavior**
PostgreSQL	Discards all changes since BEGIN; resets transaction state
MariaDB / MySQL	Same; also supports rollback to named SAVEPOINTs
SQLite3	Same; supports full and partial rollback via SAVEPOINT

ROLLBACK — Beginner Questions

- **When should I use ROLLBACK?** ⇝ When any part of a transaction fails or violates constraints.
- **What happens after ROLLBACK?** ⇝ All uncommitted changes are discarded — the database reverts to its pre-transaction state.
- **Can I rollback part of a transaction?** ⇝ Yes — use `SAVEPOINT` and `ROLLBACK TO SAVEPOINT` for partial undo.
- **Does ROLLBACK affect committed data?** ⇝ No — only uncommitted changes are undone.
- **Is ROLLBACK automatic on error?** ⇝ In some engines, yes — others require explicit rollback or error handling.
- **Can I rollback after COMMIT?** ⇝ No — once committed, changes are permanent. Use `ROLLBACK` only before `COMMIT`.

IV SAVEPOINT — Syntax Across Engines

Order Processing with Isolated Discount Logic

- **orders table** ⇝ Tracks order status and discount. SAVEPOINT isolates discount logic from status update.
- **savepoint** ⇝ Allows partial rollback without discarding entire transaction.

```sql
-- PostgreSQL
BEGIN;
UPDATE orders SET status = 'processing' WHERE id = 100;
SAVEPOINT before_discount;
UPDATE orders SET discount = 0.2 WHERE id = 100;
-- Discount logic fails
ROLLBACK TO SAVEPOINT before_discount;
COMMIT;

-- MariaDB / MySQL
START TRANSACTION;
UPDATE orders SET status = 'processing' WHERE id = 100;
SAVEPOINT before_discount;
UPDATE orders SET discount = 0.2 WHERE id = 100;
-- Abort discount
ROLLBACK TO SAVEPOINT before_discount;
COMMIT;

-- SQLite3
```

```sql
BEGIN TRANSACTION;
UPDATE orders SET status = 'processing' WHERE id = 100;
SAVEPOINT before_discount;
UPDATE orders SET discount = 0.2 WHERE id = 100;
-- Undo discount
ROLLBACK TO SAVEPOINT before_discount;
COMMIT;
```

SAVEPOINT — The Ritual of Partial Reversion

- **Purpose** ⇝ Marks a named point within a transaction to allow partial rollback
- **Input** ⇝ SAVEPOINT name, ROLLBACK TO SAVEPOINT name
- **Fallback** ⇝ Without SAVEPOINT, partial rollback is impossible — entire transaction must be aborted
- **Rendering** ⇝ SAVEPOINT names are local to the current transaction and discarded on COMMIT or full ROLLBACK
- **Use Case** ⇝ Isolating risky operations (e.g., optional updates, conditional inserts) within a larger transaction

SAVEPOINT — Syntax Glyph Map

Engine	SAVEPOINT Support
PostgreSQL	✅ SAVEPOINT, ROLLBACK TO SAVEPOINT, RELEASE SAVEPOINT supported
MariaDB / MySQL	✅ full support for SAVEPOINT and partial rollback
SQLite3	✅ supports SAVEPOINT and nested transactions
Engine	**SAVEPOINT Scope and Behavior**
PostgreSQL	Local to transaction; released on COMMIT or full ROLLBACK
MariaDB / MySQL	Same; can explicitly RELEASE SAVEPOINT
SQLite3	Same; SAVEPOINTs can be nested

SAVEPOINT — Beginner Questions

- **Why use a SAVEPOINT?** ⇝ To isolate risky operations and allow partial rollback without discarding the full transaction.
- **What happens after ROLLBACK TO SAVEPOINT?** ⇝ All changes after the SAVEPOINT are undone; earlier changes remain.
- **Can I name SAVEPOINTs anything?** ⇝ Yes — names must be unique within the transaction scope.
- **What is RELEASE SAVEPOINT?** ⇝ It removes the SAVEPOINT marker without rolling back.
- **Are SAVEPOINTs required?** ⇝ No — they're optional, but essential for fine-grained control in complex transactions.
- **Do SAVEPOINTs work across transactions?** ⇝ No — they exist only within the current transaction and vanish on COMMIT or full ROLLBACK.

IV RELEASE SAVEPOINT — Syntax Glyph Map

Order Processing — Discarding Discount SAVEPOINT After Success

- **orders table** ⇝ Tracks order status and discount. SAVEPOINT isolates discount logic from status update.
- **release** ⇝ Removes SAVEPOINT marker after successful discount logic

```sql
-- PostgreSQL
BEGIN;
UPDATE orders SET status = 'processing' WHERE id = 100;
SAVEPOINT before_discount;
UPDATE orders SET discount = 0.2 WHERE id = 100;
RELEASE SAVEPOINT before_discount;
COMMIT;

-- MariaDB / MySQL
START TRANSACTION;
UPDATE orders SET status = 'processing' WHERE id = 100;
SAVEPOINT before_discount;
UPDATE orders SET discount = 0.2 WHERE id = 100;
RELEASE SAVEPOINT before_discount;
COMMIT;

-- SQLite3
BEGIN TRANSACTION;
UPDATE orders SET status = 'processing' WHERE id = 100;
SAVEPOINT before_discount;
```

```sql
UPDATE orders SET discount = 0.2 WHERE id = 100;
RELEASE SAVEPOINT before_discount;
COMMIT;
```

RELEASE SAVEPOINT — The Ritual of Discarding Partial Rollback Markers

- **Purpose** ⇝ Removes a named SAVEPOINT from the transaction context without rolling back
- **Input** ⇝ RELEASE SAVEPOINT name
- **Fallback** ⇝ If not released, SAVEPOINTs persist until COMMIT or full ROLLBACK
- **Rendering** ⇝ Once released, the SAVEPOINT name cannot be reused unless re-declared
- **Use Case** ⇝ Used to clean up SAVEPOINTs after successful logic branches or to prevent accidental rollback

RELEASE SAVEPOINT — Syntax Glyph Map

Engine	RELEASE SAVEPOINT Support
PostgreSQL	✅ fully supported — removes SAVEPOINT without rollback
MariaDB / MySQL	✅ fully supported — same behavior
SQLite3	✅ supported — removes SAVEPOINT and commits nested transaction
Engine	**Behavior After Release**
PostgreSQL	SAVEPOINT name discarded; cannot rollback to it unless re-declared
MariaDB / MySQL	Same; engine frees internal SAVEPOINT marker
SQLite3	Same; nested transaction ends and SAVEPOINT is discarded

RELEASE SAVEPOINT — Beginner Questions

- **Why use RELEASE SAVEPOINT?** ⇝ To discard a SAVEPOINT after successful logic, preventing accidental rollback.
- **Does it rollback anything?** ⇝ No — it simply removes the SAVEPOINT marker.
- **Can I rollback after release?** ⇝ No — the SAVEPOINT is gone. You must declare a new one.
- **Is it required?** ⇝ No — SAVEPOINTs are auto-discarded on COMMIT or full ROLLBACK.
- **Can I release multiple SAVEPOINTs?** ⇝ Yes — each must be released individually.
- **Does SQLite3 treat it differently?** ⇝ Slightly — it ends the nested transaction associated with the SAVEPOINT.

IV ROLLFORWARD — Recovery Ritual Across Engines

ROLLFORWARD — The Ritual of Log Replay After Restore

- **Purpose** ⇝ Reapplies committed transactions from archived logs after restoring a backup image
- **Input** ⇝ Engine-specific commands like `ROLLFORWARD DATABASE TO END OF LOGS`
- **Fallback** ⇝ Without rollforward, restored databases remain in an incomplete or crash-consistent state
- **Rendering** ⇝ Reconstructs the database to a consistent state by replaying committed operations
- **Use Case** ⇝ Used after media failure, crash recovery, or point-in-time restore

ROLLFORWARD — Syntax Glyph Map

Engine	ROLLFORWARD Support and Behavior
DB2	✅ ROLLFORWARD DATABASE dbname TO END OF LOGS — applies logs after restore
Oracle	✅ automatic log replay via redo logs during recovery
SQL Server	✅ log replay during RESTORE WITH RECOVERY
PostgreSQL	✅ WAL segments replayed automatically during startup
MariaDB / MySQL	❌ no explicit rollforward — crash recovery is automatic
SQLite3	❌ no rollforward — uses rollback journal or WAL for crash recovery only

ROLLFORWARD — Beginner Questions

- **Is ROLLFORWARD a SQL command?** ⇝ No — it's a recovery utility or engine-level operation, not part of transactional SQL.
- **When is ROLLFORWARD used?** ⇝ After restoring a backup, to replay committed transactions from logs.
- **Does it undo uncommitted changes?** ⇝ No — only committed transactions are reapplied. Uncommitted changes are discarded.
- **Is it automatic?** ⇝ In many engines (e.g., PostgreSQL, Oracle), yes. In DB2, it must be invoked manually.
- **Can I roll forward to a point in time?** ⇝ Yes — engines like DB2 support TO TIMESTAMP recovery.
- **Is this the same as ROLLBACK?** ⇝ No — ROLLBACK undoes uncommitted changes in a transaction. ROLLFORWARD replays committed changes after restore.

IV ISOLATION LEVEL — Concurrency Ritual Across Engines

Customer Lookup — Ensuring Repeatable Reads During Transaction

- **customers table** ⇝ Tracks customer records. Isolation level ensures consistent reads during lookup and update.
- **repeatable read** ⇝ Prevents non-repeatable reads — same query returns same result within transaction

```sql
-- PostgreSQL
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM customers WHERE region = 'east';
-- Perform updates or checks
COMMIT;

-- MariaDB / MySQL
START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM customers WHERE region = 'east';
-- Perform updates or checks
COMMIT;

-- SQLite3
-- No configurable isolation level
-- Begins transaction with snapshot isolation
BEGIN TRANSACTION;
SELECT * FROM customers WHERE region = 'east';
-- Perform updates or checks
COMMIT;
```

ISOLATION LEVEL — The Ritual of Transaction Visibility Control

- **Purpose** ⇝ Defines how and when changes made by one transaction become visible to others
- **Input** ⇝ `SET TRANSACTION ISOLATION LEVEL level`
- **Fallback** ⇝ Default isolation varies by engine — often `READ COMMITTED`
- **Rendering** ⇝ Controls phenomena like dirty reads, non-repeatable reads, and phantom reads
- **Use Case** ⇝ Used to balance consistency and concurrency depending on workload and risk tolerance

ISOLATION LEVEL — Syntax Glyph Map

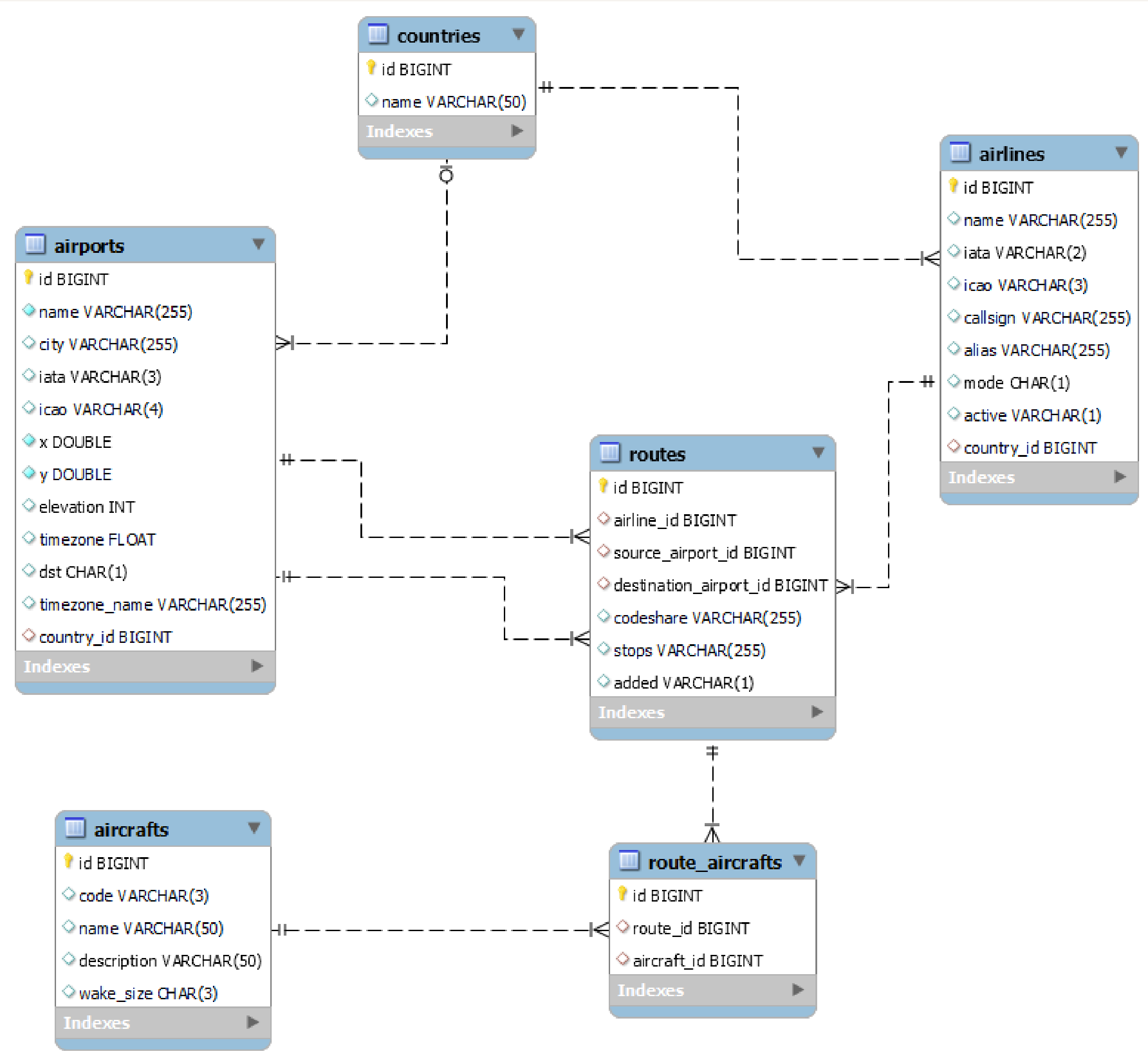
Engine	Supported Isolation Levels
PostgreSQL	✓ READ COMMITTED, REPEATABLE READ, SERIALIZABLE
MariaDB / MySQL	✓ READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE
SQLite3	✗ does not support configurable isolation levels — uses snapshot isolation via SERIALIZABLE mode
Engine	**Default Isolation Level**
PostgreSQL	READ COMMITTED
MariaDB / MySQL	REPEATABLE READ
SQLite3	SERIALIZABLE (snapshot isolation)

ISOLATION LEVEL — Beginner Questions

- **What is an isolation level?** ⇝ It defines how visible uncommitted changes are between concurrent transactions.
- **Why change it?** ⇝ To prevent anomalies like [**gfg_dirty_read**] or phantom rows, or to improve concurrency.
- **Can I change it mid-transaction?** ⇝ No — it must be set before the transaction begins.
- **What's the safest level?** ⇝ SERIALIZABLE — but it may reduce performance.
- **What's the default?** ⇝ Depends on engine — PostgreSQL uses READ COMMITTED, MySQL uses REPEATABLE READ.
- **Does SQLite support isolation levels?** ⇝ No — it uses snapshot isolation internally and does not expose configuration.

V Examples

V FlightDB (Postgresql)



Note: To access the embedded file, please open this PDF in Adobe Acrobat Reader or another full-featured PDF viewer.

Select all columns from the airports table.

```sql
SELECT * FROM airports;
```

Select the distinct "name" from the airports table. Alias the name to "Airport Name".

```sql
SELECT name AS "Airport Name" FROM airports;
```

Select all columns from the country table that has id of 160.

```sql
SELECT * FROM countries WHERE id = 160;
```

Select name, city and country id for all airports that are in the city of Hamilton.

```sql
SELECT name,city,country_id FROM airports WHERE city  = 'Hamilton';
```

Select name, city and country id for all airports that are in the city of Hamilton.

```sql
SELECT name,city,country_id FROM airports WHERE city  = 'Hamilton'
↪AND country_id = 160;
```

Select name, city and country id for all airports that are in the city of Hamilton.

```sql
SELECT name,city,country_id FROM airports WHERE city  = 'Chicago'
↪OR city = 'Boston';
SELECT name,city,country_id FROM airports WHERE city  IN
↪('Chicago', 'Boston');
```

Select airports that have an elevation between 400 and 500. You must make sure to include 400 and 500. Sort the results in decreasing order of elevation (i.e., higher to lower).

```sql
SELECT * FROM airports WHERE elevation >= 400 AND elevation <= 500
↪ORDER BY elevation DESC
```

Select all airlines that do not have a country id (i.e., country id has a null value.)

```
SELECT * FROM airlines where country_id IS NULL;
```

Select all airlines that has a name starting with “Can”. Sort the results by country id.

```
SELECT * FROM airlines where name ILIKE 'Can%' ORDER BY country_id
↪ASC;
```

Select all airlines that have a name that contains ”International”.

```
SELECT * FROM airlines WHERE name ~* '\mInternational\M';
```

Select all airlines that have a name that contains ”International”.

```
SELECT * FROM airlines WHERE name ~* '\mInternational\M';
```

Select all airlines that have a name that ends in ”Aviation”.

```
SELECT * FROM airlines WHERE name LIKE '%Aviation';
SELECT * FROM airlines WHERE name ILIKE '%Aviation' ORDER BY name
↪ASC;
```

Select all airlines that have a name that ends in ”Aviation”.

```
SELECT * FROM airlines WHERE name LIKE '%Aviation';
SELECT * FROM airlines WHERE name ILIKE '%Aviation' ORDER BY name
↪ASC;
```

Select all airports in Canada that have an elevation of 0.

```
SELECT * FROM airports
WHERE country_id = (
  SELECT id FROM countries WHERE name = 'Canada' LIMIT 1
) AND elevation = 0;
```

Select all airlines in France that have an active status of ”Y”.

```
SELECT * FROM airlines
WHERE country_id = (
  SELECT id FROM countries WHERE name = 'France' LIMIT 1
) AND active = 'Y';
```

Count how many rows exist in the airports table.

```
SELECT COUNT(*) FROM airports;
```

Calculate the average elevation in the airports table.

```
SELECT AVG(elevation) FROM airports;
```

Calculate the maximum and minimum elevations in the airports table.

```
SELECT MIN(elevation), MAX(elevation) FROM airports;
```

Calculate the average elevation, grouped by country id, in the airports table.

```
SELECT
  country_id,
  AVG(elevation) AS avg_elevation
FROM airports GROUP BY country_id;
```

Calculate the average elevation, grouped by country id, in the airports table ordered by the average elevation in descending order for any country that has an average elevation of at least 300.

```
SELECT
  country_id,
  COUNT(*) AS airport_count,
  AVG(elevation) AS avg_elevation
FROM airports
WHERE elevation >= 300
GROUP BY country_id
ORDER BY country_id DESC;
```

Calculate the number of airports in each country (i.e., grouped by country id), arrange the output in descending order of number of airports.

```
-- Long hand
SELECT
    c.name AS country_name,
    AVG(a.elevation) AS avg_elevation
FROM airports AS a
    INNER JOIN countries AS c ON a.country_id = c.id
    GROUP BY c.name
    ORDER BY c.name ASC;

-- short hand
SELECT
    c.name AS country_name,
    AVG(a.elevation) AS avg_elevation
FROM airports a
    JOIN countries c ON a.country_id = c.id
    GROUP BY c.name
    ORDER BY c.name ASC;
```

Calculate the number of airports in each city of country id 160 and arrange the output in descending order.

```
SELECT
    city,
    COUNT(*) AS airport_count
FROM airports
    WHERE country_id = 160
    GROUP BY city
    ORDER BY airport_count DESC;
```

Show only those cities where the number of airports is more then 5.

```sql
SELECT
    city,
    COUNT(*) AS airport_count
FROM airports
    WHERE country_id = 160 AND airport_count > 5
    GROUP BY city
    ORDER BY airport_count DESC;
```

Provide the query to determine how many aircrafts ”Boeing” has that start with ”74”.

```sql
SELECT COUNT(*) AS count_74s
FROM aircrafts
WHERE name = 'Boeing' AND code LIKE '74%';
```

Provide the query to determine how many different wake sizes “Boeing” and “Airbus” have.

```sql
SELECT
    name,
    COUNT(DISTINCT wake_size) AS wake_sizes
FROM aircrafts
    WHERE name IN ('Boeing', 'Airbus')
    GROUP BY name;
```