

SQL 101

SQL 101



November 3, 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.

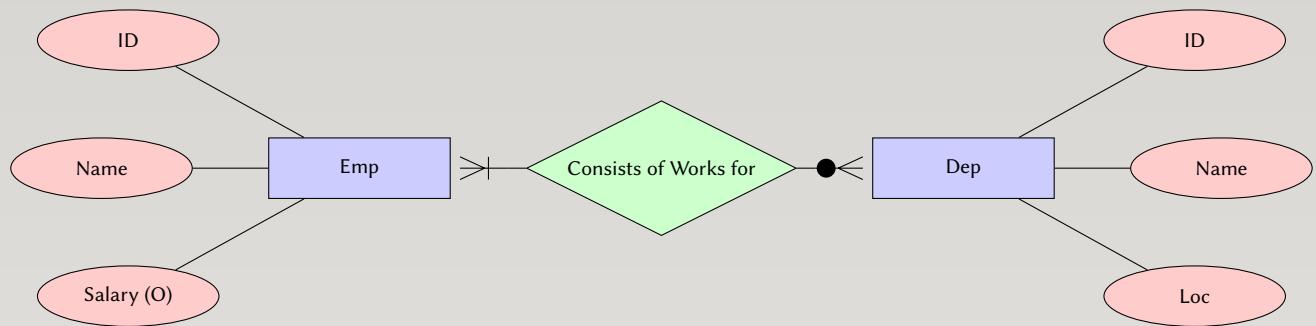
Abstract

Contents

I	Syntax	II
I	Tables	II
I	CREATE TABLE	II
I	DROP TABLE	IV
I	ALTER TABLE	VI
I	Constraints	VIII
I	Named CHECK Constraint	VIII
I	Named DEFAULT Constraint	XI
I	Named FOREIGN KEY Constraint	XIII
I	PRIMARY KEY vs UNIQUE NOT NULL	XVI
I	Key Types	XVII
I	Cardinality	XXI
I	Optional One to Mandatory Many	XXI
I	Optional One to Mandatory One	XXIV
I	Optional One to Optional One	XXVI
I	Mandatory One to Mandatory One	XXVIII
I	Mandatory Many to Mandatory Many	XXXI
I	Mandatory One to Mandatory Many	XXXIV
I	Recovery	XXXVII
I	COMMIT – Syntax Across Engines	XXXVII
I	ROLLBACK – Syntax Across Engines	XXXIX
I	SAVEPOINT – Syntax Across Engines	XLI
I	RELEASE SAVEPOINT – Syntax Glyph Map	XLIII
I	ROLLFORWARD – Recovery Ritual Across Engines	XLV
I	ISOLATION LEVEL – Concurrency Ritual Across Engines	XLVII



I Syntax



I Tables

I CREATE TABLE

CREATE TABLE – Syntax Across Engines

```

1  -- PostgreSQL
2  CREATE TABLE IF NOT EXISTS users (
3      id SERIAL PRIMARY KEY,
4      name TEXT NOT NULL,
5      created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
6  );
7
8  -- MariaDB / MySQL
9  CREATE TABLE IF NOT EXISTS users (
10     id INT AUTO_INCREMENT PRIMARY KEY,
11     name VARCHAR(255) NOT NULL,
12     created_at DATETIME DEFAULT CURRENT_TIMESTAMP
13 );
14
15 -- SQLite3
16 CREATE TABLE IF NOT EXISTS users (
17     id INTEGER PRIMARY KEY AUTOINCREMENT,
18     name TEXT NOT NULL,
19     created_at TEXT DEFAULT CURRENT_TIMESTAMP
20 );

```



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.

I DROP TABLE

DROP TABLE – Syntax Across Engines

```

1 -- PostgreSQL
2 DROP TABLE IF EXISTS users;
3
4 -- MariaDB / MySQL
5 DROP TABLE IF EXISTS users;
6
7 -- SQLite3
8 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 <code>IF EXISTS</code>
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.

I ALTER TABLE

ALTER TABLE – Syntax Across Engines

```

1  -- PostgreSQL
2  ALTER TABLE users ADD COLUMN email TEXT;
3  ALTER TABLE users RENAME COLUMN name TO full_name;
4  ALTER TABLE users DROP COLUMN email;

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

10 -- SQLite3
11 ALTER TABLE users ADD COLUMN email TEXT;
12 -- Rename supported (v3.25+)
13 ALTER TABLE users RENAME COLUMN name TO full_name;
14 -- Drop column supported (v3.35+)
15 ALTER TABLE users DROP COLUMN email;
16

```

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

I Constraints

I 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

```

1   -- PostgreSQL: CHECK for numeric range
2   CREATE TABLE products (
3       id SERIAL PRIMARY KEY,
4       price NUMERIC NOT NULL,
5       CONSTRAINT price_check CHECK (price > 0 AND price < 10000)
6   );
7
8   -- PostgreSQL: CHECK for set membership
9   CREATE TABLE orders (
10      id SERIAL PRIMARY KEY,
11      status TEXT NOT NULL,
12      CONSTRAINT status_check CHECK (status IN ('pending',
13      'shipped', 'delivered', 'cancelled'))
14  );
15
16  -- PostgreSQL: CHECK for pattern match
17  CREATE TABLE emails (
18      id SERIAL PRIMARY KEY,
19      address TEXT NOT NULL,
20      CONSTRAINT email_format CHECK (address LIKE '%@%')
21  );
22
23  -- MariaDB / MySQL: CHECK for numeric range
24  CREATE TABLE products (
25      id INT AUTO_INCREMENT PRIMARY KEY,
26      price DECIMAL(10,2) NOT NULL,
27      CONSTRAINT price_check CHECK (price > 0 AND price < 10000)
28  );
29
30  -- MariaDB / MySQL: CHECK for set membership
31  CREATE TABLE orders (
32      id INT AUTO_INCREMENT PRIMARY KEY,
33      status VARCHAR(20) NOT NULL,
34      CONSTRAINT status_check CHECK (status IN ('pending',
35      'shipped', 'delivered', 'cancelled'))
36  );
37
38  -- MariaDB / MySQL: CHECK for boolean flag
39  CREATE TABLE users (
40      id INT AUTO_INCREMENT PRIMARY KEY,

```



```

39      is_active TINYINT(1) NOT NULL,
40      CONSTRAINT active_check CHECK (is_active IN (0, 1))
41  );
42
43  -- SQLite3: CHECK for numeric range (table-level syntax)
44  CREATE TABLE products (
45      id INTEGER PRIMARY KEY AUTOINCREMENT,
46      price REAL NOT NULL,
47      CONSTRAINT price_check CHECK (price > 0 AND price < 10000)
48  );
49
50  -- SQLite3: CHECK for set membership (table-level syntax)
51  CREATE TABLE orders (
52      id INTEGER PRIMARY KEY AUTOINCREMENT,
53      status TEXT NOT NULL,
54      CONSTRAINT status_check CHECK (status IN ('pending',
55      'shipped', 'delivered', 'cancelled'))
56  );
57
58  -- SQLite3: CHECK for cross-column logic (table-level syntax)
59  CREATE TABLE bookings (
60      id INTEGER PRIMARY KEY AUTOINCREMENT,
61      start_date TEXT NOT NULL,
62      end_date TEXT NOT NULL,
63      CONSTRAINT date_order CHECK (start_date < end_date)
64  );

```

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	✓ <code>ALTER TABLE ... DROP CONSTRAINT name</code>
MariaDB / MySQL	✓ <code>ALTER TABLE ... DROP CHECK name</code>
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.



I 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

```

1  -- PostgreSQL: Named DEFAULT constraint (table-level)
2  CREATE TABLE users (
3      id SERIAL PRIMARY KEY,
4      is_active BOOLEAN NOT NULL,
5      CONSTRAINT default_active DEFAULT TRUE FOR is_active
6  );
7
8  -- PostgreSQL: Named DEFAULT constraint via ALTER TABLE
9  ALTER TABLE users
10 ADD CONSTRAINT default_active DEFAULT TRUE FOR is_active;
11
12 -- MariaDB / MySQL: DEFAULT value (unnamed, column-level only)
13 CREATE TABLE users (
14     id INT AUTO_INCREMENT PRIMARY KEY,
15     is_active BOOLEAN NOT NULL DEFAULT TRUE
16 );
17
18 -- SQLite3: DEFAULT value (unnamed, inline only)
19 CREATE TABLE users (
20     id INTEGER PRIMARY KEY AUTOINCREMENT,
21     is_active BOOLEAN NOT NULL DEFAULT 1
22 );

```

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 <code>ALTER TABLE ... ADD CONSTRAINT name DEFAULT value</code>
MariaDB / MySQL	✗ does not support naming DEFAULT constraints directly
SQLite3	✗ does not support naming DEFAULT constraints directly
Engine	Can Drop DEFAULT by Name
PostgreSQL	✓ <code>ALTER TABLE ... DROP CONSTRAINT name</code>
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.

I 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

```

1  -- PostgreSQL: Named FOREIGN KEY constraint
2  CREATE TABLE customers (
3      id SERIAL PRIMARY KEY,
4      name TEXT NOT NULL
5  );
6
7  CREATE TABLE orders (
8      id SERIAL PRIMARY KEY,
9      customer_id INT NOT NULL,
10     CONSTRAINT fk_customer FOREIGN KEY (customer_id)
11    REFERENCES customers(id)
12  );
13
14  -- PostgreSQL: Drop named FOREIGN KEY constraint
15  ALTER TABLE orders DROP CONSTRAINT fk_customer;

```



```

16  -- MariaDB / MySQL: Named FOREIGN KEY constraint
17  CREATE TABLE customers (
18      id INT AUTO_INCREMENT PRIMARY KEY,
19      name VARCHAR(255) NOT NULL
20  );
21
22  CREATE TABLE orders (
23      id INT AUTO_INCREMENT PRIMARY KEY,
24      customer_id INT NOT NULL,
25      CONSTRAINT fk_customer FOREIGN KEY (customer_id)
26      REFERENCES customers(id)
27  );
28
29  -- MariaDB / MySQL: Drop named FOREIGN KEY constraint
30  ALTER TABLE orders DROP FOREIGN KEY fk_customer;
31
32  -- SQLite3: Named FOREIGN KEY constraint (table-level only)
33  CREATE TABLE customers (
34      id INTEGER PRIMARY KEY AUTOINCREMENT,
35      name TEXT NOT NULL
36  );
37
38  CREATE TABLE orders (
39      id INTEGER PRIMARY KEY AUTOINCREMENT,
40      customer_id INT NOT NULL,
41      CONSTRAINT fk_customer FOREIGN KEY (customer_id)
42      REFERENCES customers(id)
43  );
44
45  -- SQLite3: Drop FOREIGN KEY constraint – not supported by name
46  -- 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	✓ <code>(ALTER TABLE ... DROP CONSTRAINT name)</code>
MariaDB / MySQL	✓ <code>(ALTER TABLE ... DROP FOREIGN KEY name)</code>
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.



I 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

```

1  -- PostgreSQL / MySQL / SQLite: PRIMARY KEY
2  CREATE TABLE users (
3      id SERIAL PRIMARY KEY,
4      username TEXT UNIQUE NOT NULL
5  );
6
7  -- PostgreSQL / MySQL / SQLite: UNIQUE + NOT NULL (alternate key)
8  CREATE TABLE products (

```



```

9      sku TEXT NOT NULL,
10     name TEXT NOT NULL,
11     CONSTRAINT unique_sku UNIQUE (sku)
12 );

```

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.

I 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

```

1  -- PostgreSQL
2  CREATE TABLE countries (
3      iso_code CHAR(2) PRIMARY KEY,
4      name TEXT NOT NULL
5  );
6
7  -- MariaDB / MySQL
8  CREATE TABLE countries (
9      iso_code CHAR(2) PRIMARY KEY,
10     name VARCHAR(255) NOT NULL
11 );
12
13 -- SQLite3
14 CREATE TABLE countries (
15     iso_code TEXT PRIMARY KEY,
16     name TEXT NOT NULL
17 );

```



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

```

1  -- PostgreSQL
2  CREATE TABLE users (
3      id SERIAL PRIMARY KEY,
4      email TEXT UNIQUE NOT NULL
5  );
6
7  -- MariaDB / MySQL
8  CREATE TABLE users (
9      id INT AUTO_INCREMENT PRIMARY KEY,
10     email VARCHAR(255) UNIQUE NOT NULL
11 );
12
13 -- SQLite3
14 CREATE TABLE users (
15     id INTEGER PRIMARY KEY AUTOINCREMENT,
16     email TEXT UNIQUE NOT NULL
17 );

```

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

```

1  -- PostgreSQL
2  CREATE TABLE enrollments (
3      student_id INT NOT NULL,
4      course_id INT NOT NULL,
5      enrolled_on DATE NOT NULL,
6      CONSTRAINT pk_enrollment PRIMARY KEY (student_id, course_id)
7  );
8
9  -- MariaDB / MySQL
10 CREATE TABLE enrollments (
11     student_id INT NOT NULL,
12     course_id INT NOT NULL,
13     enrolled_on DATE NOT NULL,
14     CONSTRAINT pk_enrollment PRIMARY KEY (student_id, course_id)
15 );
16
17 -- SQLite3

```

```

18  CREATE TABLE enrollments (
19      student_id INT NOT NULL,
20      course_id INT NOT NULL,
21      enrolled_on TEXT NOT NULL,
22      CONSTRAINT pk_enrollment PRIMARY KEY (student_id, course_id)
23  );

```

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.

I Cardinality

I 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.

```

1 -- PostgreSQL
2 CREATE TABLE users (
3     id SERIAL PRIMARY KEY,
4     name TEXT NOT NULL
5 );
6
7 CREATE TABLE orders (
8     user_id INT,
9     FOREIGN KEY (user_id) REFERENCES users(id)
10);
11
12 -- MariaDB / MySQL
13 CREATE TABLE users (
14     id INT AUTO_INCREMENT PRIMARY KEY,
15     name VARCHAR(255) NOT NULL
16 );

```



```

17
18   CREATE TABLE orders (
19     user_id INT,
20     FOREIGN KEY (user_id) REFERENCES users(id)
21   );
22
23 -- SQLite3
24 PRAGMA foreign_keys = ON;
25
26 CREATE TABLE users (
27   id INTEGER PRIMARY KEY AUTOINCREMENT,
28   name TEXT NOT NULL
29 );
30
31 CREATE TABLE orders (
32   user_id INT,
33   FOREIGN KEY (user_id) REFERENCES users(id)
34 );

```

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.

I 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

```

1  -- PostgreSQL
2  CREATE TABLE users (
3      id SERIAL PRIMARY KEY,
4      name TEXT NOT NULL
5  );
6
7  CREATE TABLE passports (
8      user_id INT UNIQUE NOT NULL,
9      FOREIGN KEY (user_id) REFERENCES users(id)
10 );
11
12 -- MariaDB / MySQL
13 CREATE TABLE users (
14     id INT AUTO_INCREMENT PRIMARY KEY,
15     name VARCHAR(255) NOT NULL
16 );
17
18 CREATE TABLE passports (
19     user_id INT UNIQUE NOT NULL,
20     FOREIGN KEY (user_id) REFERENCES users(id)
21 );
22
23 -- SQLite3
24 PRAGMA foreign_keys = ON;
25
26 CREATE TABLE users (
27     id INTEGER PRIMARY KEY AUTOINCREMENT,
28     name TEXT NOT NULL
29 );
30
31 CREATE TABLE passports (
32     user_id INT UNIQUE NOT NULL,
33     FOREIGN KEY (user_id) REFERENCES users(id)
34 );

```



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 <code>NOT NULL</code>
MariaDB / MySQL	✓ use <code>NOT NULL</code>
SQLite3	✓ use <code>NOT NULL</code>
Engine	Enforces One-to-One via UNIQUE
PostgreSQL	✓ use <code>UNIQUE</code> on foreign key
MariaDB / MySQL	✓ use <code>UNIQUE</code> on foreign key
SQLite3	✓ use <code>UNIQUE</code> 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

I 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

```

1  -- PostgreSQL
2  CREATE TABLE students (
3      id SERIAL PRIMARY KEY,
4      name TEXT NOT NULL
5  );
6
7  CREATE TABLE lockers (
8      id SERIAL PRIMARY KEY,
9      student_id INT UNIQUE,
10     FOREIGN KEY (student_id) REFERENCES students(id)
11 );
12
13  -- MariaDB / MySQL
14  CREATE TABLE students (
15      id INT AUTO_INCREMENT PRIMARY KEY,
16      name VARCHAR(255) NOT NULL
17  );
18
19  CREATE TABLE lockers (
20      id INT AUTO_INCREMENT PRIMARY KEY,

```



```

21     student_id INT UNIQUE,
22     FOREIGN KEY (student_id) REFERENCES students(id)
23 );
24
25 -- SQLite3
26 PRAGMA foreign_keys = ON;
27
28 CREATE TABLE students (
29     id INTEGER PRIMARY KEY AUTOINCREMENT,
30     name TEXT NOT NULL
31 );
32
33 CREATE TABLE lockers (
34     id INTEGER PRIMARY KEY AUTOINCREMENT,
35     student_id INT UNIQUE,
36     FOREIGN KEY (student_id) REFERENCES students(id)
37 );

```

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

I 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

```

1 -- PostgreSQL
2 CREATE TABLE account_settings (
    id SERIAL PRIMARY KEY,

```



```
4      theme TEXT NOT NULL
5  );
6
7  CREATE TABLE accounts (
8      id SERIAL PRIMARY KEY,
9      username TEXT NOT NULL,
10     settings_id INT UNIQUE NOT NULL,
11     FOREIGN KEY (settings_id) REFERENCES account_settings(id)
12  );
13
14  -- MariaDB / MySQL
15  CREATE TABLE account_settings (
16      id INT AUTO_INCREMENT PRIMARY KEY,
17      theme VARCHAR(255) NOT NULL
18  );
19
20  CREATE TABLE accounts (
21      id INT AUTO_INCREMENT PRIMARY KEY,
22      username VARCHAR(255) NOT NULL,
23      settings_id INT UNIQUE NOT NULL,
24      FOREIGN KEY (settings_id) REFERENCES account_settings(id)
25  );
26
27  -- SQLite3
28  PRAGMA foreign_keys = ON;
29
30  CREATE TABLE account_settings (
31      id INTEGER PRIMARY KEY AUTOINCREMENT,
32      theme TEXT NOT NULL
33  );
34
35  CREATE TABLE accounts (
36      id INTEGER PRIMARY KEY AUTOINCREMENT,
37      username TEXT NOT NULL,
38      settings_id INT UNIQUE NOT NULL,
39      FOREIGN KEY (settings_id) REFERENCES account_settings(id)
40  );
```

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 <code>(NOT NULL)</code>
MariaDB / MySQL	✓ use <code>(NOT NULL)</code>
SQLite3	✓ use <code>(NOT NULL)</code>
Engine	Enforces One-to-One via UNIQUE
PostgreSQL	✓ use <code>(UNIQUE)</code> on foreign key
MariaDB / MySQL	✓ use <code>(UNIQUE)</code> on foreign key
SQLite3	✓ use <code>(UNIQUE)</code> 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

I 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

```

1 -- PostgreSQL
2 CREATE TABLE students (
3     id SERIAL PRIMARY KEY,
4     name TEXT NOT NULL
5 );
6
7 CREATE TABLE courses (
8     id SERIAL PRIMARY KEY,
9     title TEXT NOT NULL
10);
11
12 CREATE TABLE enrollments (
13     student_id INT NOT NULL,
14     course_id INT NOT NULL,
15     PRIMARY KEY (student_id, course_id),
16     FOREIGN KEY (student_id) REFERENCES students(id),
17     FOREIGN KEY (course_id) REFERENCES courses(id)
18 );

```

```

19   -- MariaDB / MySQL
20   CREATE TABLE students (
21       id INT AUTO_INCREMENT PRIMARY KEY,
22       name VARCHAR(255) NOT NULL
23   );
24
25   CREATE TABLE courses (
26       id INT AUTO_INCREMENT PRIMARY KEY,
27       title VARCHAR(255) NOT NULL
28   );
29
30   CREATE TABLE enrollments (
31       student_id INT NOT NULL,
32       course_id INT NOT NULL,
33       PRIMARY KEY (student_id, course_id),
34       FOREIGN KEY (student_id) REFERENCES students(id),
35       FOREIGN KEY (course_id) REFERENCES courses(id)
36   );
37
38   -- SQLite3
39   PRAGMA foreign_keys = ON;
40
41   CREATE TABLE students (
42       id INTEGER PRIMARY KEY AUTOINCREMENT,
43       name TEXT NOT NULL
44   );
45
46   CREATE TABLE courses (
47       id INTEGER PRIMARY KEY AUTOINCREMENT,
48       title TEXT NOT NULL
49   );
50
51   CREATE TABLE enrollments (
52       student_id INT NOT NULL,
53       course_id INT NOT NULL,
54       PRIMARY KEY (student_id, course_id),
55       FOREIGN KEY (student_id) REFERENCES students(id),
56       FOREIGN KEY (course_id) REFERENCES courses(id)
57   );
58

```



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.

I 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

```

1  -- PostgreSQL
2  CREATE TABLE departments (
3      id SERIAL PRIMARY KEY,
4      name TEXT NOT NULL
5  );
6
7  CREATE TABLE employees (
8      id SERIAL PRIMARY KEY,
9      name TEXT NOT NULL,
10     department_id INT NOT NULL,
11     FOREIGN KEY (department_id) REFERENCES departments(id)
12 );
13
14  -- MariaDB / MySQL
15  CREATE TABLE departments (
16      id INT AUTO_INCREMENT PRIMARY KEY,
17      name VARCHAR(255) NOT NULL
18  );
19

```



```

20  CREATE TABLE employees (
21      id INT AUTO_INCREMENT PRIMARY KEY,
22      name VARCHAR(255) NOT NULL,
23      department_id INT NOT NULL,
24      FOREIGN KEY (department_id) REFERENCES departments(id)
25  );
26
27  -- SQLite3
28  PRAGMA foreign_keys = ON;
29
30  CREATE TABLE departments (
31      id INTEGER PRIMARY KEY AUTOINCREMENT,
32      name TEXT NOT NULL
33  );
34
35  CREATE TABLE employees (
36      id INTEGER PRIMARY KEY AUTOINCREMENT,
37      name TEXT NOT NULL,
38      department_id INT NOT NULL,
39      FOREIGN KEY (department_id) REFERENCES departments(id)
40  );

```

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.



I Recovery

I 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.

```

1  -- PostgreSQL
2  BEGIN;
3  UPDATE accounts SET balance = balance - 100 WHERE id = 1;
4  UPDATE accounts SET balance = balance + 100 WHERE id = 2;
5  COMMIT;
6
7  -- MariaDB / MySQL
8  START TRANSACTION;
9  UPDATE accounts SET balance = balance - 100 WHERE id = 1;
10 UPDATE accounts SET balance = balance + 100 WHERE id = 2;
11 COMMIT;
12
13 -- SQLite3
14 BEGIN TRANSACTION;
15 UPDATE accounts SET balance = balance - 100 WHERE id = 1;
16 UPDATE accounts SET balance = balance + 100 WHERE id = 2;
17 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 <code>(COMMIT)</code> and transactional control
MariaDB / MySQL	✓ full support for <code>(COMMIT)</code> and rollback logic
SQLite3	✓ supports <code>(COMMIT)</code> 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.

```

1 -- PostgreSQL
2 BEGIN;
3 UPDATE inventory SET quantity = quantity - 1 WHERE item_id = 42;
4 -- Something goes wrong

```



```

5   ROLLBACK;
6
7   -- MariaDB / MySQL
8   START TRANSACTION;
9   UPDATE inventory SET quantity = quantity - 1 WHERE item_id = 42;
10  -- Error detected
11  ROLLBACK;
12
13  -- SQLite3
14  BEGIN TRANSACTION;
15  UPDATE inventory SET quantity = quantity - 1 WHERE item_id = 42;
16  -- Abort transaction
17  ROLLBACK;

```

I 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.

```

1   -- PostgreSQL
2   BEGIN;
3   UPDATE inventory SET quantity = quantity - 1 WHERE item_id = 42;
4   -- Something goes wrong
5   ROLLBACK;
6
7   -- MariaDB / MySQL
8   START TRANSACTION;
9   UPDATE inventory SET quantity = quantity - 1 WHERE item_id = 42;
10  -- Error detected
11  ROLLBACK;
12
13  -- SQLite3
14  BEGIN TRANSACTION;
15  UPDATE inventory SET quantity = quantity - 1 WHERE item_id = 42;
16  -- Abort transaction
17  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 <code>(ROLLBACK)</code> and <code>(ROLLBACK TO SAVEPOINT)</code>
MariaDB / MySQL	✓ full support for <code>(ROLLBACK)</code> and <code>SAVEPOINT</code> rollback
SQLite3	✓ supports <code>(ROLLBACK)</code> and nested <code>SAVEPOINT</code> rollback
Engine	ROLLBACK Behavior
PostgreSQL	Discards all changes since <code>(BEGIN)</code> ; resets transaction state
MariaDB / MySQL	Same; also supports rollback to named <code>SAVEPOINTS</code>
SQLite3	Same; supports full and partial rollback via <code>SAVEPOINT</code>



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)`.

I 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.

```

1  -- PostgreSQL
2  BEGIN;
3  UPDATE orders SET status = 'processing' WHERE id = 100;
4  SAVEPOINT before_discount;
5  UPDATE orders SET discount = 0.2 WHERE id = 100;
6  -- Discount logic fails
7  ROLLBACK TO SAVEPOINT before_discount;
8  COMMIT;

9
10 -- MariaDB / MySQL
11 START TRANSACTION;
12 UPDATE orders SET status = 'processing' WHERE id = 100;
13 SAVEPOINT before_discount;
14 UPDATE orders SET discount = 0.2 WHERE id = 100;
15 -- Abort discount
16 ROLLBACK TO SAVEPOINT before_discount;
17 COMMIT;

18
19 -- SQLite3

```



```

20 BEGIN TRANSACTION;
21 UPDATE orders SET status = 'processing' WHERE id = 100;
22 SAVEPOINT before_discount;
23 UPDATE orders SET discount = 0.2 WHERE id = 100;
24 -- Undo discount
25 ROLLBACK TO SAVEPOINT before_discount;
26 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.

I 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

```

1  -- PostgreSQL
2  BEGIN;
3  UPDATE orders SET status = 'processing' WHERE id = 100;
4  SAVEPOINT before_discount;
5  UPDATE orders SET discount = 0.2 WHERE id = 100;
6  RELEASE SAVEPOINT before_discount;
7  COMMIT;
8
9  -- MariaDB / MySQL
10 START TRANSACTION;
11 UPDATE orders SET status = 'processing' WHERE id = 100;
12 SAVEPOINT before_discount;
13 UPDATE orders SET discount = 0.2 WHERE id = 100;
14 RELEASE SAVEPOINT before_discount;
15 COMMIT;
16
17 -- SQLite3
18 BEGIN TRANSACTION;
19 UPDATE orders SET status = 'processing' WHERE id = 100;
20 SAVEPOINT before_discount;
```



```

21 UPDATE orders SET discount = 0.2 WHERE id = 100;
22 RELEASE SAVEPOINT before_discount;
23 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 <code>SAVEPOINT</code> without rollback
MariaDB / MySQL	✓ fully supported – same behavior
SQLite3	✓ supported – removes <code>SAVEPOINT</code> and commits nested transaction
Engine	Behavior After Release
PostgreSQL	<code>SAVEPOINT</code> name discarded; cannot rollback to it unless re-declared
MariaDB / MySQL	Same; engine frees internal <code>SAVEPOINT</code> marker
SQLite3	Same; nested transaction ends and <code>SAVEPOINT</code> 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.

I 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	✓ <code>(ROLLFORWARD DATABASE dbname TO END OF LOGS)</code> – applies logs after restore
Oracle	✓ automatic log replay via redo logs during recovery
SQL Server	✓ log replay during <code>(RESTORE WITH RECOVERY)</code>
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.



I 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

```

1  -- PostgreSQL
2  BEGIN;
3  SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
4  SELECT * FROM customers WHERE region = 'east';
5  -- Perform updates or checks
6  COMMIT;

7
8  -- MariaDB / MySQL
9  START TRANSACTION;
10 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
11 SELECT * FROM customers WHERE region = 'east';
12 -- Perform updates or checks
13 COMMIT;

14
15 -- SQLite3
16 -- No configurable isolation level
17 -- Begins transaction with snapshot isolation
18 BEGIN TRANSACTION;
19 SELECT * FROM customers WHERE region = 'east';
20 -- Perform updates or checks
21 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 [fgf_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.