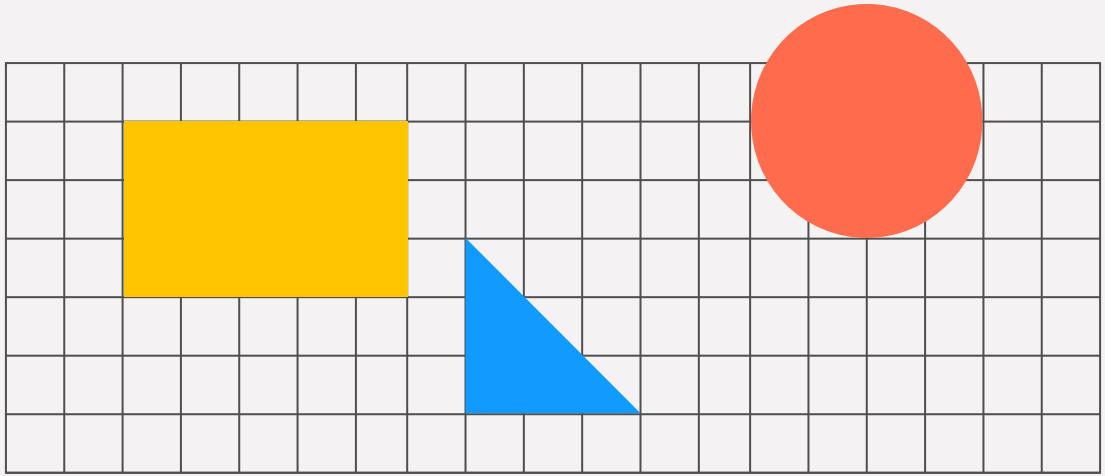
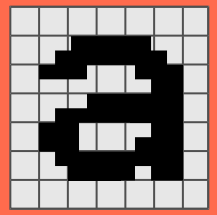
Database Management!

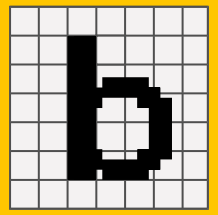
Ali Abrishami



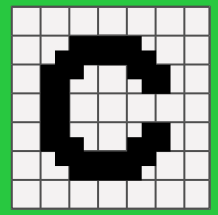
In This Lecture You Will...



Learn what a DBMS is



Send queries to a DB and get proper responses



Know when to use SQL and NoSQL Databases

How Did We Get Here?

- As you may remember from your FOP and AP courses, a software interacts with two types of data.
  - **Temporary data:** the data program requires in its runtime.
  - **Permanent data:** the data that is used throughout different runtimes and sessions.

- For example, the variables required to calculate the 10th number in fibonacci series are temporary data.

- However, your Dark Souls III save data should be permanent and you expect them to be there the next time you launch the game (or you will be really angry!)

Storing Permanent Data

- There are several ways to store permanent data.
- You COULD store your data on a **file** (or many files). It's simple and doesn't really require any technical knowledge in most use cases.
- But it might not be efficient, as there may be techniques to speed-up the future operations on said data.
- Take hash-sets, for instance. If we could store a hashset of our data, next to our data, could it bring some benefits?
- It could speed up the process of checking if an element exists in our dataset or not.

Class Activity Time!

Suppose we're storing some user data, alongside their favorite movies, on a file. It is said that the emails, phone numbers, and IDs of the users are unique. Movies have unique IDs as well.

- Name some ways you could speed up the process of fetching a specific user with their respective favorite movies.
- Now, let's discuss how storing additional data helps storing and retrieval of our original data faster.

Isn’t That Complicated?

- Providing solutions for said questions require a lot of thought and may complicate the process of software development.
- Luckily, there are some common solutions to our storage problems.
- Some tech people have gathered around and created solutions and services that provide more structured and efficient ways for us to store data.
- These solutions are called **Database Management Systems** (DBMS).

History of DBMS

- The first database management systems (DBMS) were created to handle complex data for businesses in the 1960s. These systems included Charles Bachman's Integrated Data Store (IDS) and IBM's Information Management System (IMS). Databases were first organized into tree-like structures using hierarchical and network models.

- Edgar F. Codd popularized the relational model in the 1970s, transforming database management systems (DBMS) with the concept of arranging data in tables, or relations and utilizing SQL for queries. As a result, contemporary DBMS systems like Oracle and MySQL were established. These systems are still developing today, incorporating newer technologies like NoSQL databases to handle unstructured data.

To Query & Mutate

- Usually, DBMS communicates with the database using **Structured Query Language** (**SQL**).
- A database management system (DBMS) is primarily used to manage **large amounts** of data in an organized manner while maintaining data security, consistency, and integrity.
- Structured Query Language is a standard Database language that is used to create, maintain, and retrieve the relational database. In this article, we will discuss this in detail about SQL. Following are some interesting facts about SQL. Let's focus on that.
- SQL is **case insensitive**. But it is a recommended practice to use keywords (like `SELECT`, `UPDATE`, `CREATE`, etc.) in capital letters and use user-defined things (like table name, column name, etc.) in small letters.

Who Said Relational?

- A relational database means the data is stored as well as retrieved in the form of **relations** (**tables**).
- The most common definition of an RDBMS is a product that presents a view of data as a collection of **rows and columns**, even if it is not based strictly upon relational theory, which was defined by *E. F. Codd* at IBM in 1970.
- At minimum, a relational database should Present the data to the user as relations (a presentation in tabular form, i.e. as a collection of tables with each table consisting of a set of rows and columns) and provide relational operators to manipulate the data in **tabular form**.

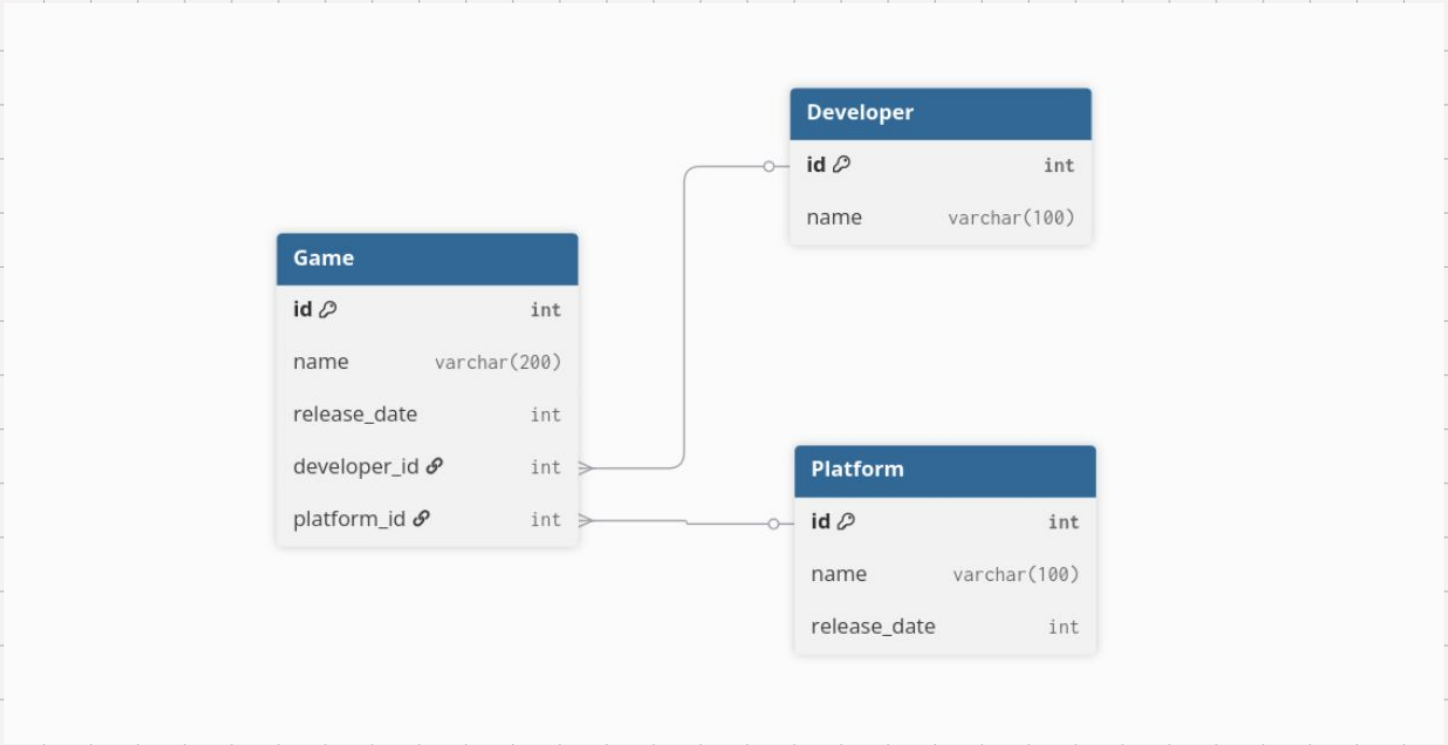
Relations Cardinality

- One-to-One (1:1): Each record in A maps to at most one record in B; used for splitting rarely-used or sensitive attributes (e.g., Users ↔ Passports).
- One-to-Many (1:N): A single record in A relates to multiple in B, but each B relates to exactly one A (e.g., Departments → Employees).
- Many-to-Many (M:N): Records on both sides relate to multiple counterparts; requires a junction table (e.g., Students ↔ Courses via Enrollment).

Diagrams!

- An **entity-relation diagram** (ERD) is a visual modeling of data entities, their attributes, and relationships before designing tables.
- Highlights cardinality constraints (1:1, 1:N, M:N) and participation (mandatory/optional).
- Entities become **tables**; attributes become **columns**; relationships may become **foreign keys** or **junction tables**.
- Used to catch logical design flaws early, before committing to schema or normalization.
- Use pen and paper, draw.io, or dbdiagram.io to create ERDs.

Sample ERD

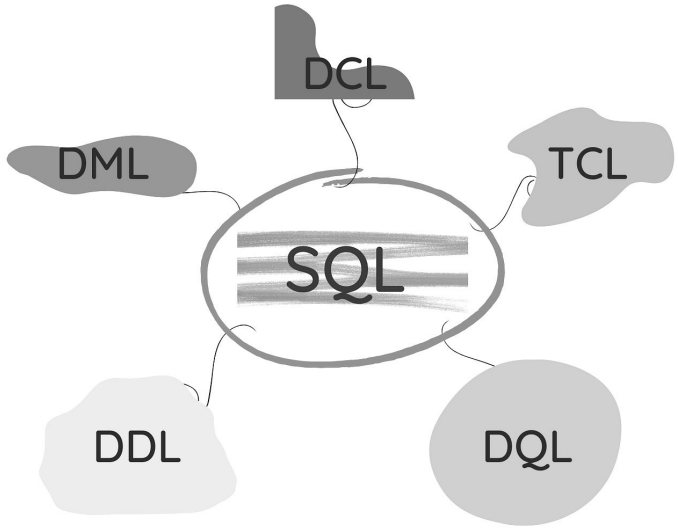

Developer
id int
name varchar(100)
Game
id int
name varchar(200)
release_date int
developer_id int
platform_id int
Platform
id int
name varchar(100)
release_date int

RDBMS Terminologies

- **Attribute**: Attributes are the properties that define a relation.
- **Tuple**: Each row in the relation is known as tuple.
- **Degree**: The number of attributes in the relation is known as degree of the relation.
- **Cardinality**: The number of tuples in a relation is known as cardinality.
- **Column**: Column represents the set of values for a particular attribute.

Query Languages

- **Data Definition Language**: used to create new relations.
- **Data Manipulation Language:** used to manipulate (i.e. update or delete) existing relations.
- **Data Query Language:** used to fetch already existing data.
- There are also **“Data Control Language”** and **“Transaction Control Language”**.

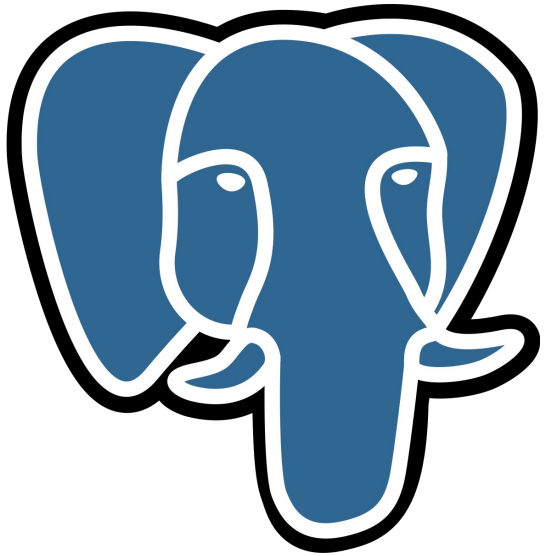


SQL

- Declarative language for **querying** and **managing** relational data.
- Core verbs: `SELECT`, `INSERT`, `UPDATE`, `DELETE`.
- Schema defined via `CREATE TABLE`, constraints enforce integrity.
- Supports joins to combine data across related tables.
- Strong consistency and optimized querying through indexes.
- Less flexible schema; scaling horizontally requires deliberate design.

PostgreSQL

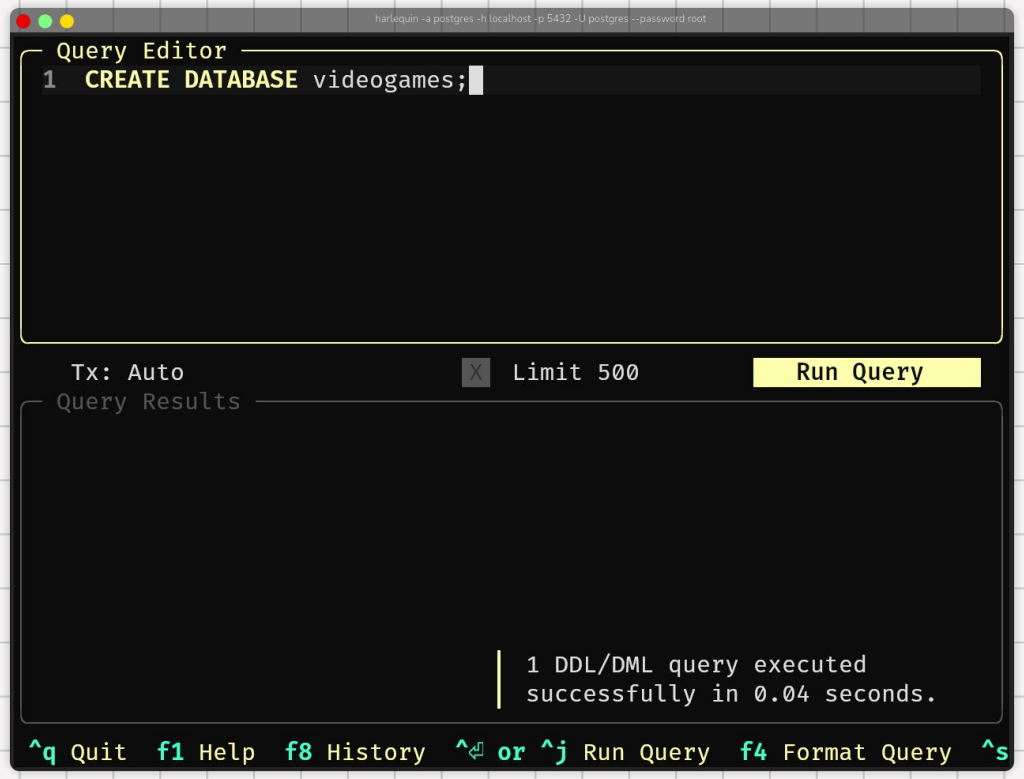
- We'll use PostgreSQL for the rest of this lecture; examples assume Postgres syntax.
- **Open-source relational** database with strong standards and feature coverage.
- Supports advanced features: JSONB, extensions (PostGIS), custom types, window functions.
- Prioritizes correctness and consistency of behavior over raw performance.



CREATE DATABASE Query

- Used to create a **new database** instance within the PostgreSQL server.
- Syntax: `CREATE DATABASE dbname;` (run from a superuser or a role with createdb privilege).
- Optional parameters: owner, encoding, template, locale, tablespace.
- Databases are **isolated** namespaces, schemas, tables, etc. live inside them.
- Typically executed from psql, admin tools, or migrations before schema creation.

CREATE DATABASE Query



CREATE TABLE Query

- Defines a new table by specifying column names, data types, and optional default values.
- **Constraints** enforce correctness: `PRIMARY KEY`, `NOT NULL`, `UNIQUE`, `CHECK`, `FOREIGN KEY`.
- Data types differ across systems, but most support **integers, decimals, text, timestamps, and booleans.**
- Auto-generated values use sequences or identity mechanisms (`GENERATED` or equivalent), avoiding manual IDs.
- Table definition determines performance and relationships, schema design is not a cosmetic choice.

CREATE TABLE Query

harlequin -a postgres -h localhost -p 5432 -U postgres --password root -d videogames

Query Editor

```
1  CREATE TABLE platforms (
2      id SERIAL PRIMARY KEY,
3      name VARCHAR(100) NOT NULL UNIQUE,
4      manufacturer VARCHAR(100),
5      release_year INT
6  );
```

CREATE TABLE Query

harlequin -a postgres -h localhost -p 5432 -U postgres --password root -d videogames

Query Editor

```
1  CREATE TABLE developers (
2      id SERIAL PRIMARY KEY,
3      name VARCHAR(150) NOT NULL UNIQUE,
4      country VARCHAR(100)
5  );
6
```

CREATE TABLE Query

harlequin -a postgres -h localhost -p 5432 -U postgres --password root -d videogames

Query Editor

```sql
CREATE TABLE games (
    id SERIAL PRIMARY KEY,
    title VARCHAR(150) NOT NULL,
    release_year INT,
    platform_id INT NOT NULL,
    developer_id INT NOT NULL,
    FOREIGN KEY (platform_id) REFERENCES platforms(id),
    FOREIGN KEY (developer_id) REFERENCES developers(id)
);
```

INSERT Queries

- **Add** new rows to a table, specifying values in column order or by named columns.
- Explicit column lists prevent breakage if the table schema changes.
- Default values fill unspecified columns when defined in the schema.
- Multi-row inserts allow efficient bulk loading rather than repeated single statements.
- Integrity constraints apply on insert, violations **raise errors** rather than silently ignoring problems.
- Auto-generated keys are typically returned via dialect-specific methods, not by assuming a value.

INSERT Queries

harlequin -a postgres -h localhost -p 5432 -U postgres --password root -d videogames

Query Editor

```
1  INSERT INTO platforms (name, manufacturer, release_year)
2  VALUES
3      ('PlayStation 5', 'Sony', 2020),
4      ('Nintendo Switch', 'Nintendo', 2017),
5      ('Xbox Series X', 'Microsoft', 2020);
```

INSERT Queries

harlequin -a postgres -h localhost -p 5432 -U postgres --password root -d videogames

Query Editor

```
1  INSERT INTO developers (name, country)
2  VALUES
3      ('FromSoftware', 'Japan'),
4      ('CD Projekt Red', 'Poland'),
5      ('Santa Monica Studio', 'USA');
```

INSERT Queries

harlequin -a postgres -h localhost -p 5432 -U postgres --password root -d videogames

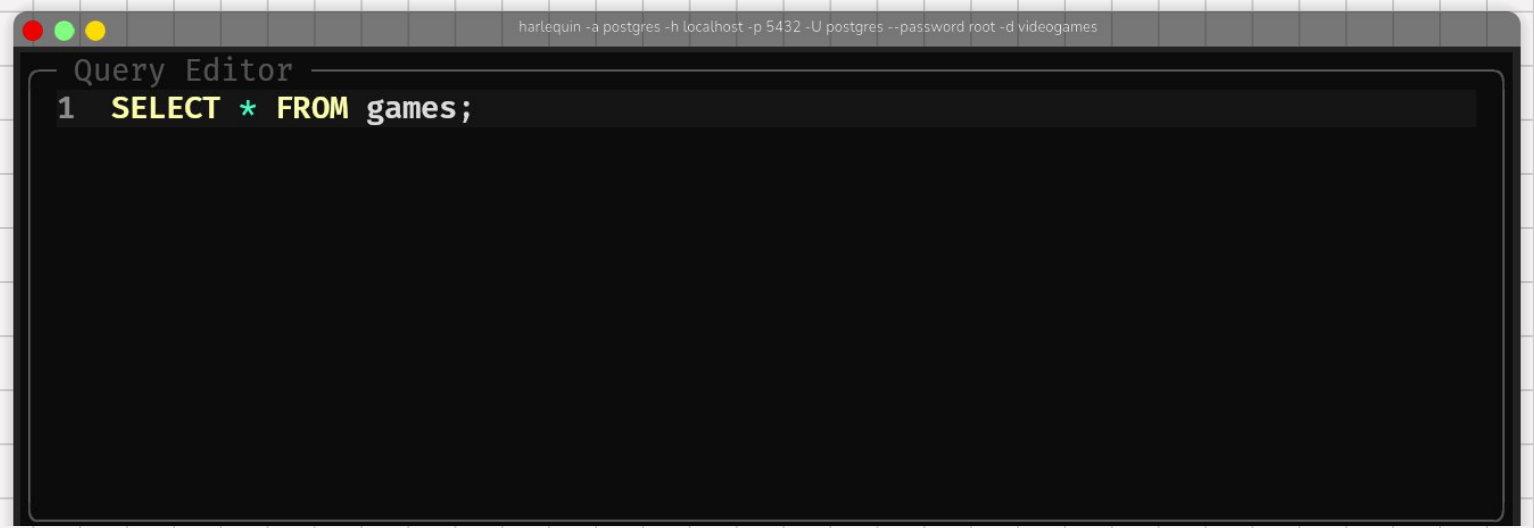
Query Editor

```
1  INSERT INTO games (title, release_year, platform_id, developer_id)
2  VALUES
3      ('Elden Ring', 2022, 1, 1),
4      ('God of War: Ragnarök', 2022, 1, 3),
5      ('Cyberpunk 2077', 2020, 3, 2),
6      ('Zelda: Breath of the Wild', 2017, 2, 2);
```

SELECT Queries

- Used to **retrieve rows** from one or more tables based on specified conditions.
- Columns selected explicitly (e.g., name, price) or via wildcard (*), the latter is lazy and discouraged.
- **Filtering** done with `WHERE` clauses, supporting predicates, functions, and subqueries.
- **Sorting** controlled with `ORDER BY`; deterministic ordering requires explicit columns.
- **Aggregation** via `GROUP BY`, `COUNT`, `SUM`, `AVG`, `HAVING` for filtered aggregates.
- **Multi-table retrieval** achieved through `JOIN` operations rather than manual merging.

SELECT Queries



Result

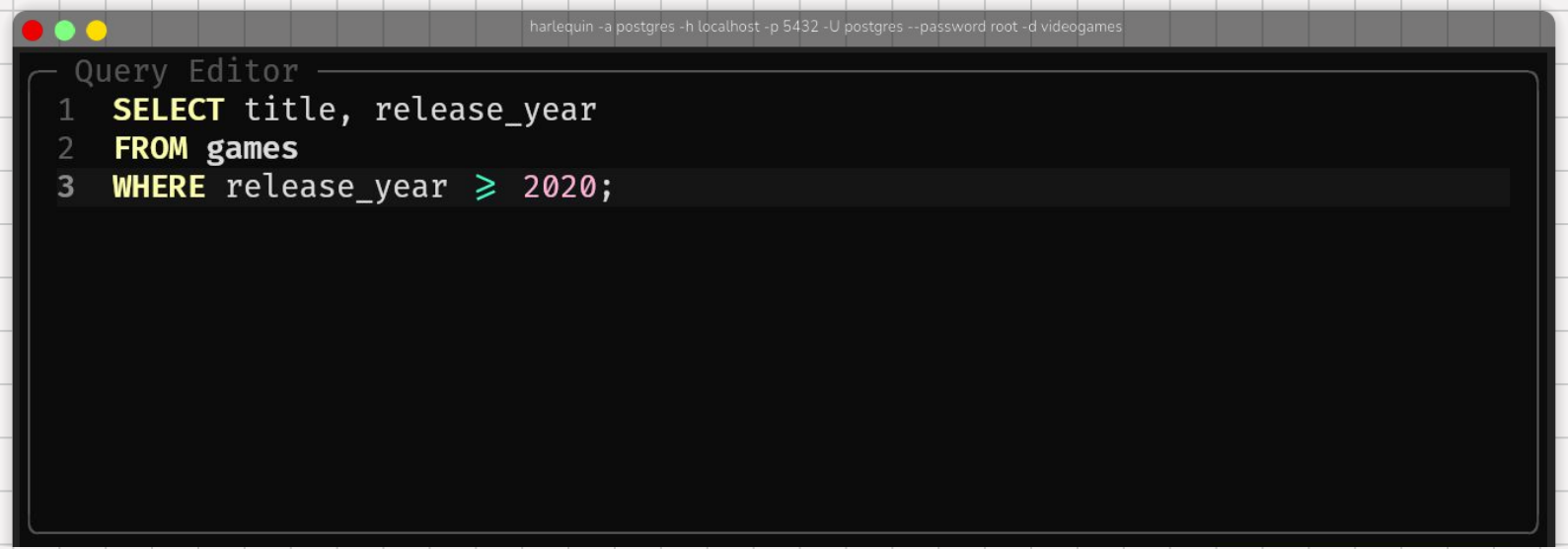

Tx: Auto
X Limit 500
Run Query
Query Results (4 Records)
id # title s release_year # platform_id # developer_id #
1 Elden Ring 2,022 1 1
2 God of War: Ragnarök 2,022 1 3
3 Cyberpunk 2077 2,020 3 2
4 Zelda: Breath of the Wild 2,017 2 2
^q Quit f1 Help f8 History

SELECT Queries

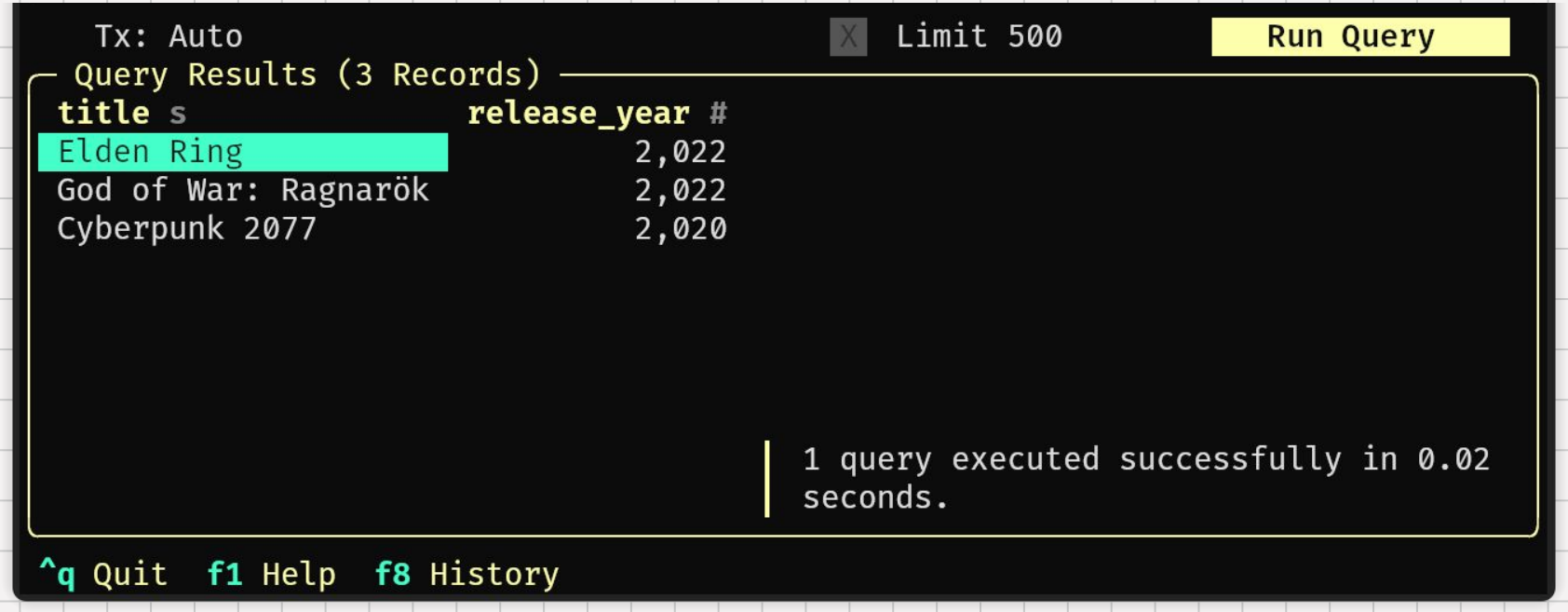
Result


Tx: Auto
X Limit 500
Run Query
Query Results (4 Records)
title s
release_year #
Elden Ring
2,022
God of War: Ragnarök
2,022
Cyberpunk 2077
2,020
Zelda: Breath of the Wild
2,017
1 query executed successfully in 0.02 seconds.
^q Quit f1 Help f8 History

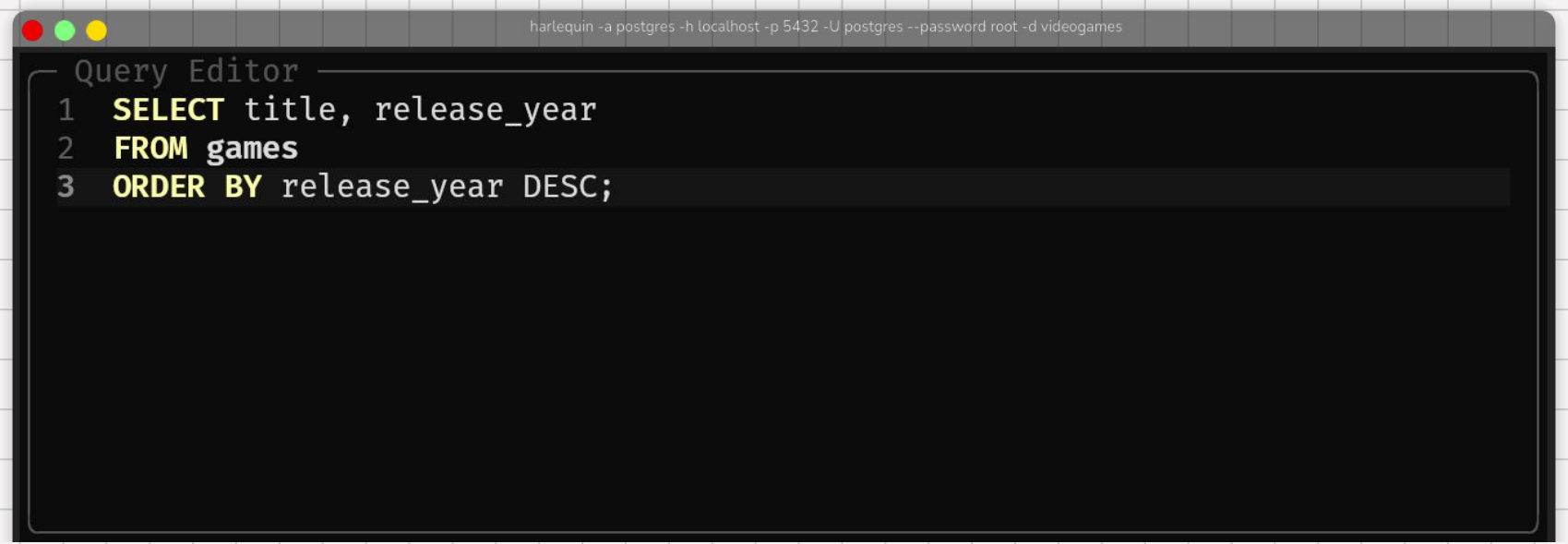
SELECT Queries



Result


Tx: Auto
X Limit 500
Run Query
Query Results (3 Records)
title s
release_year #
Elden Ring
2,022
God of War: Ragnarök
2,022
Cyberpunk 2077
2,020
1 query executed successfully in 0.02 seconds.
^q Quit f1 Help f8 History

SELECT Queries



Result


Tx: Auto
X Limit 500
Run Query
Query Results (4 Records)
title s
release_year #
Elden Ring
2,022
God of War: Ragnarök
2,022
Cyberpunk 2077
2,020
Zelda: Breath of the Wild
2,017
1 query executed successfully in 0.02 seconds.
^q Quit f1 Help f8 History

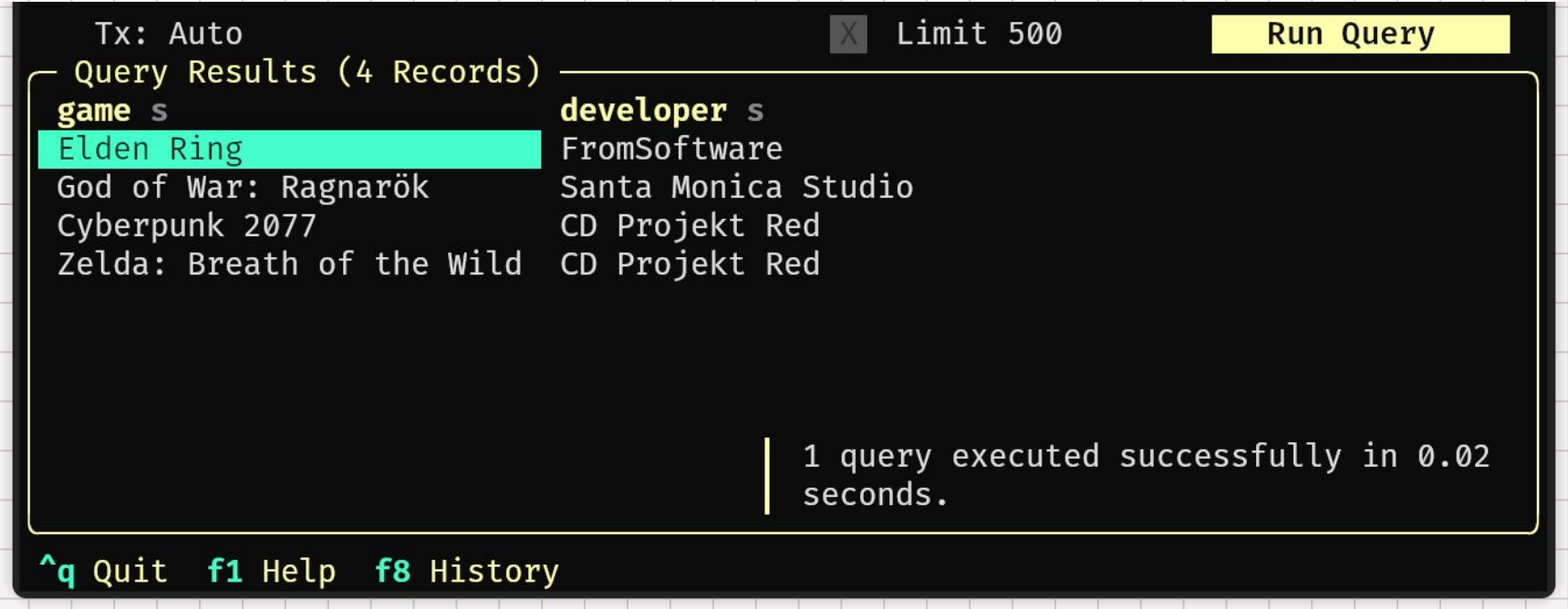
SELECT Queries

harlequin -a postgres -h localhost -p 5432 -U postgres --password root -d videogames

Query Editor

```
1  SELECT g.title AS game, d.name AS developer
2  FROM games AS g
3  JOIN developers AS d ON g.developer_id = d.id;
```

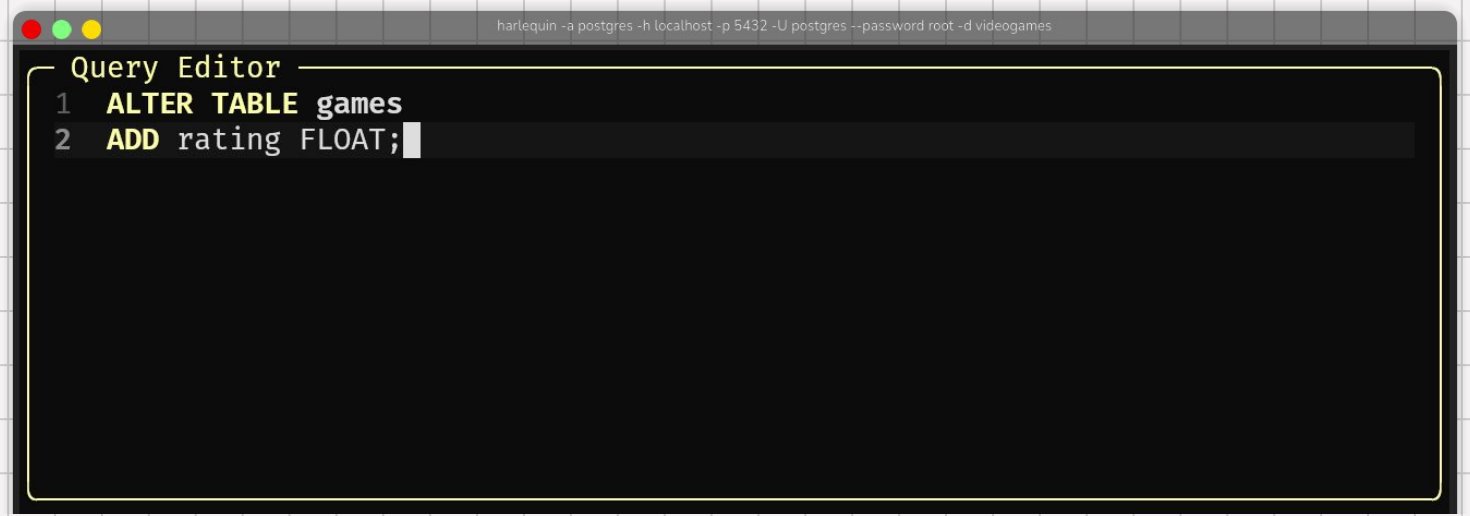
Result


Tx: Auto
X
Limit 500
Run Query
Query Results (4 Records)
game s
developer s
Elden Ring
FromSoftware
God of War: Ragnarök
Santa Monica Studio
Cyberpunk 2077
CD Projekt Red
Zelda: Breath of the Wild
CD Projekt Red
1 query executed successfully in 0.02 seconds.
^q Quit
f1 Help
f8 History

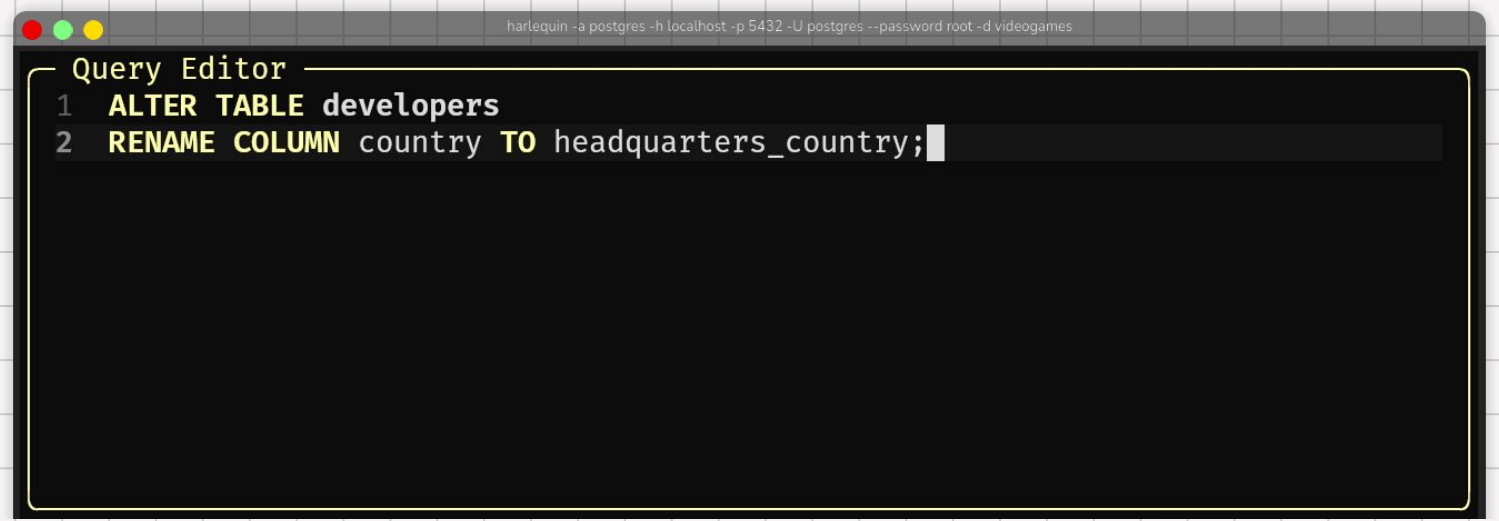
ALTER TABLE Queries

- **Modifies** an existing table's structure without recreating it from scratch.
- Supports adding, dropping, or renaming **columns** and **constraints**.
- Changing data types may require casting or fail if existing data cannot convert safely.
- Removing or tightening constraints can break dependent data, cleanup often required first.
- Renaming tables or columns preserves data but can break queries, applications, and views.
- Structural changes in live systems may require locks or migrations to avoid downtime.

ALTER TABLE Queries



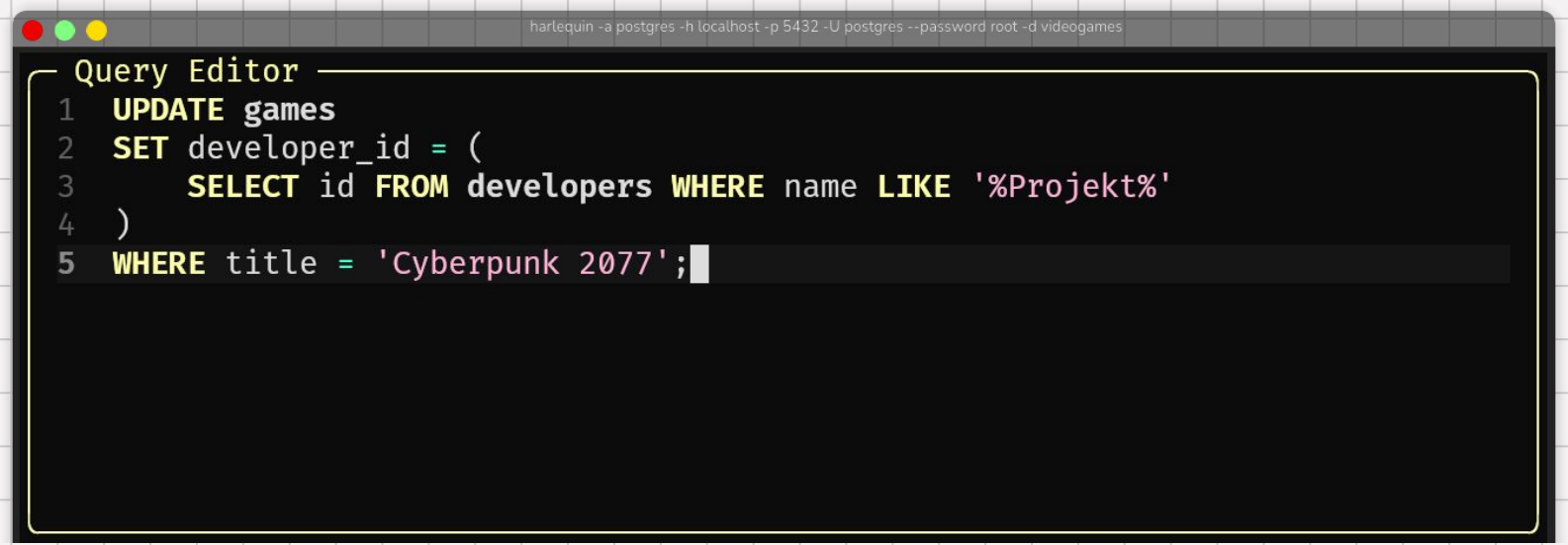
ALTER TABLE Queries



UPDATE Queries

- Modifies **existing** rows in a table according to a condition (if any).
- Syntax: `UPDATE table_name SET column = value, ... WHERE condition;`
- Without a `WHERE` clause, all rows get updated.
- Can use expressions, functions, and subqueries in the `SET` clause.
- Returns the number of affected rows in PostgreSQL.

UPDATE Queries



DELETE Queries

- Remove rows from a table based on a `WHERE` condition, omitting it removes all rows.
- Deletes obey constraints; dependent rows may block deletion unless cascading rules exist.
- `DELETE` targets rows, not structure, table and schema remain intact.
- `RETURNING` (when supported) can output deleted rows for auditing or application logic.
- Bulk deletes can be slow; filtering and indexing reduce full-table scans.
- Irreversible without backups or transaction rollback, treat with production-level caution.

DELETE Queries

harlequin -a postgres -h localhost -p 5432 -U postgres --password root -d videogames

Query Editor

```
1  DELETE FROM games
2  WHERE developer_id = (
3      SELECT id FROM developers WHERE name LIKE '%Projekt%'
4  );
5
```

DROP Query

- Permanently removes a database object (a table, for example).
- Deletes both structure and all contained data, cannot be reversed without backups.
- Often supports `IF EXISTS` to avoid errors when objects are missing.
- Cascading options remove dependent objects; use minimal scope to avoid accidental destruction.
- Dropping a table breaks references from queries, foreign keys, and application code.
- Typically used for schema cleanup or irreversible migrations, not routine data management.

READ MORE!

Entity Design

- Entity design defines what your system “is”, not what SQL syntax you happen to remember. Each entity must represent a real, meaningful concept in the domain.
- A good entity has well-defined attributes, each with a single, clear purpose — no duplicated fields, no vague “misc_data” placeholders.
- Proper design avoids mixing responsibilities: an entity should model one thing, not three different business ideas bundled for convenience.
- Entity relationships (1-1, 1-M, M-N) must reflect actual business rules, not whatever schema layout happens to be the easiest to code.

Identifiers in Entity Design

- A primary key must uniquely identify each entity instance; it should be stable, minimal, and never carry business meaning that might change.
- Surrogate keys (like auto-increment IDs or UUIDs) simplify design and avoid future inconsistencies, whereas natural keys must be used with caution.
- Foreign keys enforce real relationships between entities, ensuring referential integrity instead of relying on developer discipline.
- The choice of keys influences indexing, join costs, and the overall structure of the schema. A sloppy key today becomes a performance curse tomorrow.

Weak Entities

- A weak entity has no independent primary key; it depends on a strong (owner) entity for identification and existence.
- It uses a partial key combined with the owner's key to form a composite primary key alone, the partial key is meaningless.
- Weak entities model real-world dependents: a `LineItem` cannot exist without its `Invoice`, a `Room` cannot exist without its `Building`.
- Deleting the owner typically cascades to the weak entity, reflecting true dependency in both structure and business logic.

Normalization

- Normalization is the disciplined process of structuring tables to reduce redundancy and prevent inconsistent or duplicated data.
- It breaks large, messy tables into smaller, well-defined relations, each representing one concept with minimal overlap.
- The goal is to enforce data integrity by ensuring every fact lives in exactly one place. No *“update one row, forget the other”* disasters!
- Normalization also helps the query optimizer by clarifying relationships, but yes, over-normalization can hurt performance if you misuse it.

1st Normal Form (1NF)

- Every attribute holds atomic (indivisible) values; no lists, sets, or nested structures hiding inside cells.
- Each row-column intersection contains exactly one value, not multiple.
- All records in a table follow a consistent structure. Same columns, same meaning per column.
- Eliminates repeating groups and multi-valued attributes, forcing clean, tabular data.

2nd Normal Form (2NF)

- **Must already satisfy 1NF.**
- Every non-key attribute must depend on the whole primary key, not just part of a composite key.
- Removes partial dependencies, which happen when fields depend only on one component of a multi-column PK.
- Typical fix: split the table so each meaningful dependency lives in its rightful relation.

3rd Normal Form (3NF)

- Must already satisfy 2NF.
- No transitive dependencies: non-key attributes must depend only on the key, not on each other.
- Ensures non-key columns don’t act like secondary keys storing derived or inferable data.
- Results in cleaner schema boundaries and avoids redundant, inconsistent facts.

Boyce-Codd Normal Form

- BCNF is a stronger form of 3NF. For every functional dependency, the left side must be a superkey.
- Eliminates tricky anomalies that 3NF fails to catch when multiple candidate keys exist.
- Prevents situations where non-key fields can “control” other fields.
- More rigorous but sometimes requires splitting tables further (correctness first, convenience later.)

ACID?

- Guarantees reliable transactions even when everything else around them misbehaves.
- Ensures databases remain correct under concurrency, crashes, or user stupidity :)
- Core foundation behind serious systems: banking, inventory, reservations, anything where wrong means disaster.
- ACID isn’t really optional! it’s what separates a real DBMS from a glorified Excel sheet.

ACID!

- Atomicity: The transaction is all-or-nothing. Half-done work is for amateurs.
- Consistency: Every transaction moves the DB from one valid state to another; no illegal states escape.
- Isolation: Concurrent transactions behave as if each runs alone; no messy interference.
- Durability: Once committed, the data survives crashes, power loss, and your OS having a panic attack.

Commit/Rollback

- Commit finalizes a transaction: all changes become permanent, visible to others, and guaranteed by durability.
- Rollback aborts a transaction: every change since the start (or a savepoint) is undone as if nothing ever happened.
- Both rely on the transaction log, the DBMS keeps enough history to reverse or confirm your actions safely.
- Used to maintain integrity under errors, failed constraints, deadlocks, or even your own bad decisions :)

Indexing

- Indexes are auxiliary data structures (usually B-Trees or Hashes) that let the DBMS locate rows without scanning the entire table.
- They work like a sorted lookup directory: fast reads, but every insert/update/delete must also update the index.
- Great for speeding up WHERE, JOIN, ORDER BY, and UNIQUE constraints, unless you misuse them and slow everything down.
- Choosing the right columns matters: high-selectivity, frequently searched columns benefit; low-selectivity ones usually waste space.

Types of Indexes

- B-Tree Index: The default workhorse. Balanced tree structure, great for range queries, sorting, and general-purpose lookups.
- Hash Index: Lightning-fast equality lookups (=), but useless for ranges or ordering. Think exact key to exact value.
- Bitmap Index: Efficient for low-cardinality columns (e.g., gender, status flags), mainly in analytical workloads.
- Composite/Multicolumn Index: Indexes multiple columns in a defined order; powerful when queries filter on that prefix pattern.

When Not to Use Indexes

- Avoid indexing columns with very low selectivity (e.g., boolean, tiny enums) unless you're using a bitmap index in analytics.
- Don't index columns that are constantly updated, like counters or timestamps; you're just increasing write cost for no real gain.
- Skip indexes on small tables. A full table scan is faster than maintaining an index nobody needs.
- Don't create indexes you never query on (common beginner mistake): an index without matching WHERE/JOIN patterns is wasted I/O.

Clustered vs. Non-clustered

- A clustered index defines the physical order of rows on disk. The table is the index. Only one allowed per table because you can't reorder data twice.
- A non-clustered index is a separate structure pointing to row locations; you can have many of these for different access patterns.
- Clustered indexes excel at range queries and sequential reads. Non-clustered ones work for targeted lookups on frequently filtered columns.
- Picking a bad clustered key (wide, random, frequently updated) punishes the entire table. Choose narrow, stable, increasing keys.