Relation: consists of a *set* of tuples (records). Each tuple is a row and has n attributes or columns. Each tuple contains the exact same attributes in the same order.
Superkey: a set of $k \leq n$ attributes that uniquely identifies a tuple. There are at most $2^n - 1$ superkeys for an n-attribute relation.
Candidate Key: is a minimal superkey s.t. no subset of its attributes form a superkey itself. A candidate key may be `null`.
Primay Key: is a candidate key chosen by the DB designer to enforce uniqueness based on use case. A primary key may not be `null`. If a primary key is composite, no component can be `null`
Foreign Key: in S points to a primary key in R. FK's need not be unique in S, but must be unique (by def.) in R. FK's are primarily used for referential integrity. Further, the FK $\in S$ need not have the same name as the PK $\in R$.

Selection: $\sigma_\psi(R) = \{t \in R : \psi(t)\}$. $\sigma_\psi(R) \approx$ `SELECT * FROM` R `WHERE` $\psi(t)$. It filters on ***tuples*** using: $=, \neq, <, >, \leq, \geq, \neg, \vee, \wedge$.
Projection: $\Pi_{a_i}(R) = \{t[a_i] : t \in R, i \leq n\}$. $\Pi \approx$ `SELECT` $a_1, \ldots, a_n$ `FROM` R. Also, $\Pi_{f(a_i) \to a'}$ where f is any reasonable function.
Cartesian Product: $R \times S = \{(r, s) : r \in R, s \in S\}$. They are very bad and inefficient.
Natural Join: $R \bowtie S = \Pi_{R \cup S}(\sigma_{R.k=S.k}(R \times S)) = \{(r, s) : r \in R, s \in S, r[k] = s[k]\}$. Only to be used in relational algebra.
Natural Join Edge Cases: If $k = \emptyset$, $R \bowtie S = R \times S$. **If** $\forall r \in R, s \in S, r[k] \neq s[k]$, $R \bowtie S = \emptyset$.
Join Key: is the set of $k \leq n$ attributes that we join R, S on. All conditions are equality $\implies$ equijoin. Otherwise, non-equijoin.
Theta Join: $R \bowtie_\theta S = \sigma_\theta(R \times S) = \{(r, s) : r \in R, s \in S, \theta((r, s))\}$. Name clash $\to$ alias. We choose the join key.
Inner Join: Include all rows that satisfy $\theta((r, s))$. Throw out all rows that don't satisfy $\theta((r, s))$.
Aggregation: ${}_{\texttt{group}}\gamma_{f(a_i)}(R)$ where f is an aggregation function. Some include `SUM, AVG, MIN, MAX, DISTINCT-COUNT`.
Rename: $\rho_S(R)$ renames a ***relation*** $R \to S$. $\rho_{a/b}(R)$ renames an ***attribute*** $a \to b \in R$. Usually used in $rho_S(R) \times R$.
Union: $R \cup S = \{r_1, \ldots, r_{|R|}, s_1, \ldots, s_{|S|} : r_i \in R, s_j \in S\}$. R, S must have the same set of attributes for this to work.
Set Difference: $R - S = \{t : t \in R, t \notin S\}$. **Note: Division $\div$ is not implemented in SQL.**
Intersection: $R \cap S = \{t : t \in R, S\}$. R, S must have the same set of attributes for this to work. Note: $R \cap S = R - (R - S)$.
Order of Operations: $\sigma, \Pi, \rho \to \times, \bowtie \to \cap \to \cup, -$.

`ENUM`: Order of defined when type is constructed. Values are case sensitive, whitespace matters. ***Can:*** add, rename values. ***Cannot:*** delete, reorder values. 4 bytes.
Create Enum/Table:

```
CREATE TYPE enum_name AS ENUM ('value_1', ..., 'value_n');
CREATE TABLE table_name (
    column_1     type OPTIONS,
    ...
    column_n     type OPTIONS
    );
```

where `type` is a data type and `OPTIONS` can be none or more of: `NOT NULL, DEFAULT [DEFAULT VALUE], UNIQUE, PRIMARY KEY, FOREIGN KEY REFERENCES other_table(other_table_ukey) ON DELETE/UPDATE CASCADE/RESTRICT/SET NULL`. We can set the PK/FK inline or at the bottom using `PRIMARY KEY (column_i)` and `FOREIGN KEY (column_j) REFERENCES other_table(other_table_ukey)`.
Changing Schema: Don't lmao. Use `extra` (if you were smart enough to think ahead) or create another table with a join key.
Alter Table: add/drop columns, constraints (e.g. PK/FK), rename tables/columns, change data types of columns.

```
ALTER TABLE table_name
    DROP col_i,                        -- delete column
    ALTER COLUMN col_j TYPE new_type,  -- changes type of col_j to new_type
    ADD col_k type,                    -- adds col_k
    DROP CONSTRAINT table_name_pkey,   -- drops PK constraint
    ADD PRIMARY KEY col_l,             -- adds PK constraint to col_l
    RENAME COLUMN col_m TO new_col_name, -- renames col_m to new_col_name
    RENAME TO new_table_name;          -- renames table_name to new_table_name
```

Drop, Truncate, Delete: `DROP [TABLE/SCHEMA/DATABASE] table_name/schema_name/db_name;` deletes the table/schema/db. If inside a script, use `IF EXISTS`. `TRUNCATE table_name` will delete all of the data inside `table_name`, but will preserve the schema. This is the same as `DELETE FROM table_name WHERE 1=1`.
Select: `SELECT col_1, ..., col_n FROM table_name WHERE condition;`.
Where: pre-filters ***rows*** in a table. It acts on values in columns and transformation functions applied on rows independently (***NOT*** aggregation functions). Note: `WHERE c BETWEEN x AND y` $\simeq$ `WHERE c <= y AND c >= x`.
Query Order: `SELECT` $\to$ `FROM` $\to$ `JOIN` $\to$ `ON(s)` $\to$ `WHERE` $\to$ `GROUP BY` $\to$ `HAVING` $\to$ `ORDER BY` $\to$ `LIMIT` $\to$ `OFFSET`
Execution Order: `FROM` $\to$ `ON` $\to$ `JOIN` $\to$ `WHERE` $\to$ `GROUP BY` $\to$ `HAVING` $\to$ `SELECT` $\to$ `DISTINCT` $\to$ `ORDER BY`

Aggregation/Group By: Aggregations over a relation does not need a `GROUP BY`. Aggregations over groups requires a `GROUP BY`. For example: `SELECT AVG(one) AS avg FROM table_name;` and `SELECT one, AVG(two) AS avg FROM table_name GROUP BY one;`
Having: post-filters result of an aggregation. `SELECT one AVG(two) AS avg FROM r_name GROUP BY one HAVING AVG(two) < 100;`
Outer Join: keep rows that don't have a match, replacing the "other side" as `null`. We use `LEFT/RIGHT/FULL OUTER JOIN` where `OUTER` is optional.
Left Join: keeps all rows in the LHS of the join.
Right Join: keeps all rows in the RHS of the join.
Full Join: keeps rows from both sides of the join.
Coalesce: `COALESCE(expr, replacement value)` where `expr` may return `null`. It can take multiple arguments and returns the first that is not `null`.

Nested Query/Subquery: Innermost query gets evaluated first.
Derived Table Subquery: returns a table.

```
SELECT uid, last, first, mi, scores.career, midterm, (midterm - mean) / sd AS z_score
FROM (
    SELECT career, AVG(midterm) AS mean, STDDEV(midterm) AS sd
    FROM midterm_scores
    GROUP BY career
) aggregated
JOIN midterm_scores scores
ON scores.career = aggregated.career;
```

Scalar Subquery: returns a scalar.

```
SELECT uid, last, first, mi, midterm
FROM midterm_scores
WHERE midterm > (
    SELECT AVG(midterm) + 0.5 * STDDEV(midterm)
    FROM midterm_scores
);
```

```
SELECT uid, last, first, mi, midterm,
  (midterm - (SELECT AVG(midterm) FROM midterm_scores))
  / (SELECT STDDEV(midterm) FROM midterm_scores)
  AS zscore
FROM midterm_scores;
```

Filter Subquery: using `IN/NOT IN` is a semijoin if we project out all of the columns from the `flights` table.

```
SELECT flights.*
FROM flights
    WHERE flights.tail IN (
    SELECT tail FROM airtran_aircraft
);
```

Correlated Subquery: They suck, lol. This reexecutes the subquery for every row in the outer query.

```
SELECT uid, last, first, mi, midterm
FROM midterm_scores m1
WHERE midterm > (
    SELECT AVG(midterm) + 0.5 * STDDEV(midterm)
    FROM midterm_scores m2
    WHERE m1.career = m2.career
);
```

Subqueries v. Joins: Subqueries are typically faster. Joins are slow so we want to filter as much as possible before joining.
Adding Rows: `INSERT INTO table_name VALUES ('val11', ..., 'val1n'), ('val21', ..., 'val2n'),...;` requires us to know the schema. Order matters, and all values must be specified. Another way is:
`INSERT INTO table_name (col1_name, ..., colk_name) VALUES ('val11', ..., 'val1k'), ('val21', ..., 'val2k'), ...;`
We just specify the names of the columns we insert into. Order doesn't matter but we need to be consistent.
Modifying Rows: `UPDATE table_name SET column_name = new_value WHERE condition;`
Check Constraint: `CONSTRAINT Constraint_Name CHECK (condition);` is put at the end of a `CREATE TABLE`. They can be added using `ALTER TABLE`. We can only use check constraints on rows.
Casting: Cast with `column_name::new_type`.
NullIf: `NULLIF(var, replacement)`. If `var` is `null`, replace with `replacement`.
Control Flow: Case and Searched Case statements:

```
SELECT ...,
    CASE column_name
        WHEN condition_1 THEN result_1
        ...
        WHEN condition_n THEN result_n
        ELSE default_result
    END AS new_column_name
FROM midterm_scores;
```

```
SELECT ...,
    CASE
        WHEN column_name = condition_1 THEN result_1
        ...
        WHEN column_name = condition_n THEN result_n
        ELSE default_result
    END AS new_column_name
FROM midterm_scores;
```

SQL Injection: If we don't use a prepared query, consider `SELECT uid FROM bruinbase WHERE uid='{}'`. In place of "{}", we can inject `'; DROP DATABASE students; --` to drop the `students` database.
Caching: Caching is fast and decreases the workload on the DB. We can either talk to the cache and DB directly or have a broker/proxy talk to the DB and cache.
Logging: is important, so do it lmao. But, minimize the amount of private data.

Salt and Pepper: A string (salt) is randomly chosen to be affixed to the data before it is hashed. This hash and salt are stored. Peppering is similar, but is stored in a separate table. This makes it more difficult to steal than salting. Peppering is not widely implemented.

Normalization: Normalization is the process of refactoring tables to reduce redundancy in a relation. It involves splitting a table with redundant data into two or more non-redundant tables. Tables without redundancies are called ***normalized***. When there are redundancies, we can ***decompose*** the table using ***functional dependencies***.

Problems with Deormalized Tables: Redundancy, data integrity issues (update/insert), delay in creating new records. Normalized tables allow for separation of concerns.

Functional Dependency: $X \to Y$: X functionally determines Y if every $x \in X$ is associated with exactly one $y \in Y$. If there exists $X \to Y$, we can decompose the table into two: $R(X, Y)$ and $R(X, Z)$ where $Z := R \setminus Y$. For example:

X	Y	A	B
α	β	σ	π
α	β	γ	Δ
γ	η	π	Δ

Here, $X \to Y$ since $\alpha \mapsto \beta, \gamma \mapsto \eta$, so we can decompose the relation into $R_1 :=$

X	Y
α	β
α	β
γ	η

and $R_2 :=$

X	A	B
α	σ	π
α	γ	Δ
γ	π	Δ

Functional Dependency Properties (Armstrong's Axioms [1-3] and Corollaries [4-7]): $\alpha, \beta, \gamma \in r(R)$.

(1) Reflexivity: If $\beta \subseteq \alpha$, then $\alpha \to \beta$. **Ex:** $A \subseteq A \implies A \to A, A \subseteq AB \implies AB \to A$.

(2) Augmentation: If $\alpha \to \beta$, then $\alpha\gamma \to \beta\gamma$. **Ex:** $\{uid\} \to \{name\} \implies \{uid, major\} \to \{name, major\}$.

(3) Transitivity: If $\alpha \to \beta$ and $\beta \to \gamma$, then $\alpha \to \gamma$. **Ex:** $\{uid\} \to \{room\ \#\}, \{room\ \#\} \to \{room\ type\} \implies \{uid\} \to \{room\ type\}$.

(4) Union: If $\alpha \to \beta$ and $\alpha \to \gamma$, then $\alpha \to \beta\gamma$. **Pf.** $(\alpha \to \gamma \implies \alpha\alpha \to \alpha\gamma \iff \alpha \to \alpha\gamma), (\alpha \to \beta \implies \alpha\gamma \to \beta\gamma) \implies \alpha \to \alpha\gamma \to \beta\gamma$.

(5) Composition: If $\alpha \to \beta, \gamma \to \Delta$, then $\alpha\gamma \to \beta\Delta$.

(6) Decomposition: If $\alpha \to \beta\gamma$, then $\alpha \to \beta$ and $\alpha \to \gamma$.

(7) Pseudotransitivity: If $\alpha \to beta$, $\Delta\beta \to \gamma$, then $\Delta\alpha \to \gamma$.

Canonical Cover: $F_c \subseteq F^+$ is the basis set of the set of all functional dependencies F^+. It is ***not*** unique.

Finding F_c: (1) Decompose RHS: ($X \to YZA$ becomes $X \to Y, X \to Z, X \to A$). **(2)** Remove extraneous attributes: ($AB \to C, B \to C$, $AB \to C$ is extraneous). **(3)** Remove trivial, duplicate, inferred FD's (by transitivity). **(4)** Union and repeat until set doesn't change.

Example: Given $\{B \to D, C \to D, AB \to C, B \to E, C \to F, A \to BCDEF, AB \to D, AB \to F\}$,
After **(1)**, we get $\{A \to B, A \to C, A \to D, A \to E, A \to F, B \to D, C \to D, AB \to C, B \to E, C \to F, AB \to D, AB \to F\}$.
After **(2)**, we get $\{A \to B, A \to C, A \to D, A \to E, A \to F, B \to D, C \to D, B \to E, C \to F\}$.
After **(3)**, we get $\{A \to B, A \to C, B \to D, C \to D, B \to E, C \to F\}$.
After **(4)**, we get $F_c := \{A \to BC, B \to DE, C \to DF\}$. Then we have $R_1(A, B, C), R_2(B, D, E), R_3(C, D, F)$.

Normal Forms: There are 8 normal forms, but we discuss 1NF, 2NF, 3NF, and BCNF (3.5NF).

First Normal Form (1NF): Atomic attributes (flat, no nesting/collections), no repeated groups, there is a unique key, no `null` values.

Second Normal Form (2NF): R is 1NF and does not contain any composite keys. More generally, R is 2NF $\iff \forall a \in R$, ***either* (1)** $a \in$ CK or **(2)** $a \in R$ depends on an ***entire*** key; i.e. it is not partially dependent on ***any*** composite candidate key.

Third Normal Form (3NF): All non-prime $a \in R$ depend directly on a CK (no transitivity); i.e. if all $a \in R$ are part of a candidate key, R is 3NF. **Zaniolo's 3NF:** $\forall f \in F$, at least one is true: **(1)** $a \to \beta$ is trivial. **(2)** $\alpha \in R$ is SK. **(3)** $\beta \in$ CK.