COMP 421 cheat sheet (Yingjie Xu)

Entity-Relationship Model (ER): representation of the data model of the application

Entity (instance in OOP): an entity is described using a set of **attributes.**

Entity Set (class in OOP) → rectangle in ER: All entities in an **entity set → (oval in ER)** have the same attributes. (An entity set must have a **Key** → underlined)

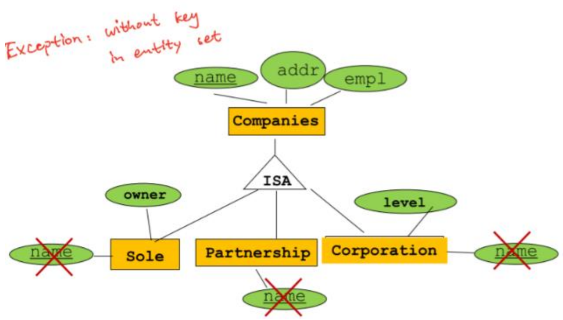
Entity → Rows; attributes → Cols

ISA ("is a") Hierarchies → Subclasses: A ISA B, then every A entity is also a B entity – Key only in B.

Reason for ISA: 1. Additional descriptive attributes specific for a subclass 2. Identification of a subclass that participates in a relationship.

ISA ("is a") Hierarchies: Keys

- Only the highest entity sets has the key attribute!!



Overlap Constraint: Can an entity be in more than one subclass? (allowed/disallowed)

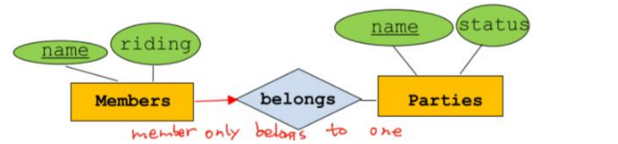
Covering Constraint: Must every entity of the superclass be in one of the subclasses? (yes/no)

Relationship: Association among two or more entities.

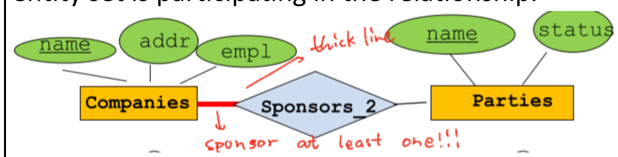
Relationship Set: Collection of similar relationships.

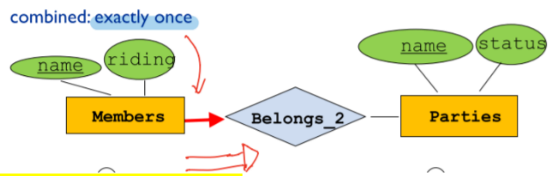
Many-to-Many: …

Key-constraints (one-many, many-one): we must have **at most** one entity set is participating in the relationship.



Participation constraints: we must have **at least** one entity set is participating in the relationship.

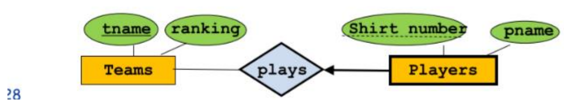




Ternary Relationship is relationships involving 3 entity sets. Keep in mind that a ternary relationship database entry MUST include all 3 entities.

Weak Entity: a weak entity can be identified uniquely only by considering the primary key of another (owner) entity.

E/R:
- Weak entity set in **bold**
- Relationship set to supporting entity set with key and participating constraint (**bold and arrowed**)
- Relationship set in **bold**
- Partial key in weak entity set with **dashed line**



Don't keep redundant information.

- **Relational Database:** a set of relations
- **Relation:** Consists of two parts:
  - **Schema:** specifies name of relation, plus a set of attributes, plus the domain/type of each attribute column
    - E.g., Students(sid:*int*, name:*string*, login:*string*, faculty:string, major:*string*)
  - **Instance:** set of tuples (all tuples are distinct). row
  - Compare with entity set and entity; or with object class and object instance
- A relation can be seen as a table:
  - Column headers = attribute names, rows = tuples/records, columns/fields = attribute values

Schema = column header + table name

Column header = attribute names

Row = tuple / record

Columns / fields = attribute values

All rows are distinct in DB

Database Schema: collection of relation schemas

Data Definition Language (DDL): defines the schema of a database.

Data Manipulation Language (DML): manipulates the data

Integrity Constraints must be true for any instance of the database

- **Not Null:** requires an attribute to always have a proper value

- **Primary Key Constraints:** No two distinct tuples can have same values in all key fields. (**unique**) PK should be **not null**.

```
CREATE TABLE Location
   (building VARCHAR(20),
    roomNo   INT,
   capacity INT,
   PRIMARY KEY(building, roomNo)
)
```

- **Candidate Key Constraints: (UNIQUE)**
  - Each student has a unique id.
  - Each student has a unique login

candidate key.

```
CREATE TABLE Students
   (sid CHAR(9) PRIMARY KEY,
    login VARCHAR(30) NOT NULL UNIQUE,
    name VARCHAR(20),
    ...)
```

- **Foreign Key Constraint:** Set of attributes in one relation R that is used to "refer" to a tuple in another relation Q. the foreign key value of a tuple must represent an existing tuple in the referred relation.

delete enroll first, then delete student. ← sid in enroll must be in the student table.

```
PRIMARY KEY (sid,cid),
FOREIGN KEY (sid) REFERENCES Students,
FOREIGN KEY (cid) REFERENCES Courses
```

If all foreign key constraints are enforced, **referential integrity** is achieved, i.e., no dangling references.

ER-Relational Translation

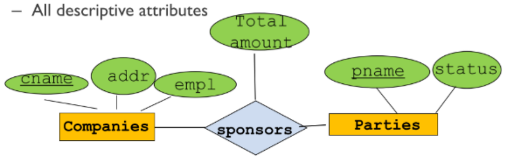
- Entity Sets to Relations:

Companies(name, address, empl)

```
PostgreSQL:
CREATE TABLE Companies
     (name VARCHAR(30),
      addr VARCHAR(50),
      empl INTEGER,
      PRIMARY KEY (name))
```

- Many-many Relationship Sets: A many-to-many relationship set is **ALWAYS** translated as an individual table.

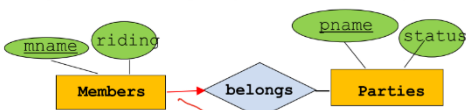
Attributes of the table are
- Keys for each participating entity set (as foreign keys)
  - This set of attributes forms the key for the relation
- All descriptive attributes



Companies(cname,addr,empl)
Parties(pname, status)
Sponsorship(cname,pname,tamount) → Relationship many-to-many
cname references Companies
pname references Parties

- Relationships Sets with Key Constraints:

- Alternative 1: map relationship set to table
- Many-one from entity set E1 to entity set E2: key of E1
  - i.e., key of entity-set with the key constraint is the key for the new relationship table (**mname** is now the key)
- One-one: key of either entity set
- Separate tables for entity sets (**Members** and **Parties**)



look at arrow

Members(mname,riding)

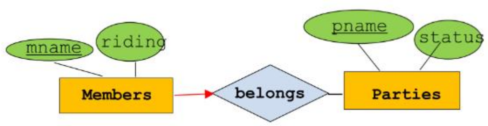
Parties(pname,status)

Membership(mname,pname)

key constraint

```
CREATE TABLE Membership
  (mname VARCHAR(30),
   pname VARCHAR(20),
   PRIMARY KEY (mname),
   FOREIGN KEY (mname)
   REFERENCES Members,
   FOREIGN KEY (pname)
   REFERENCES Parties)
```

- Alternative 2: include relationship set in table of the entity set with the key constraint



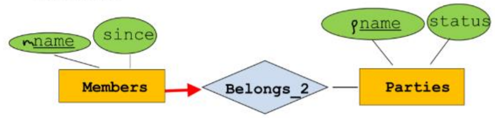
Members(mname,riding,pname)

Parties(pname,status)

```
CREATE TABLE Member
  (mname VARCHAR(30),
   riding VARCHAR(30),
   pname VARCHAR(20),
   PRIMARY KEY (mname),
   FOREIGN KEY (pname)
       REFERENCES Parties)
```

- Key and Participation Constraints

- Include relationship set in table of the entity set with the key constraint



not null

Members(mname,since,pname)

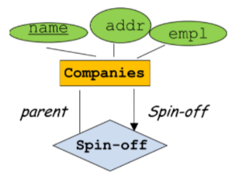
Parties(pname,status)

```
CREATE TABLE Member
  (mname VARCHAR(30),
   since DATE,
   pname VARCHAR(20) NOT NULL,
   PRIMARY KEY (mname),
   FOREIGN KEY (pname)
       REFERENCES Parties)
```

- Participation Constraints: cannot be reflected usually, except for both key and participation.

Renaming

In the case the keys of the participating entity sets have the same names we must rename attributes accordingly



- Companies(name, addr, empl)
- SpinOff(spinoffcompany, parentcompany)

```
CREATE TABLE SpinOff
  (spinoffcompany VARCHAR(30) PRIMARY
KEY,
   parentcompany VARCHAR(30),
   FOREIGN KEY (spinoffcompany)
     REFERENCES Companies(name),
   FOREIGN KEY (parentcompany)
     REFERENCES Companies(name))
```

Renaming can also occur for foreign keys, etc.

- Weak Entity Sets

- Weak entity set and identifying relationship set are translated into a single table

Teams(tname,ranking)
Players(tname,shirtN,pname)

```
CREATE TABLE Players
  (tname VARCHAR(30),
   shirtN INT,
   pname VARCHAR(30,) NOT NULL
   PRIMARY KEY (tname,shirtN),
   FOREIGN KEY (tname)
     REFERENCES Teams)
```

- Translating ISA Hierarchies

- General Approach: distribute information among relations. Relation of superclass stores the general attributes and defines key. Relations of subclasses have key of superclass and additional attributes.

- Companies(name, addr, empl)
- Sole(name, owner)
- Partnership(name)
- Corporation(name, level)

- Object-oriented approach: Sub-classes have all attributes; if an entity is in a sub-class it does not appear in the super-class relation.

- One big relation: Create only one relation for the root entity set with all attributes found anywhere in its network of subclasses. Put NULL in attributes not relevant to a given entity.

Relational Algebra

Relational Algebra: Operations

- Single relation as input
  - **Selection σ:** Selects a subset of tuples from a relation
  - **Projection ∏:** projects to a subset of attributes from a relation
  - **Renaming ρ:** of relations or attributes; useful when combining several operators
- Two relations as input
  - **Cross Product X:** Combines two relations
  - **Join⋈ :** Combination of Cross product and selection
  - **(Division):** not covered in class
- Set operators with two relations as input
  - **Intersection (∩)**
  - **Union (∪)**
  - **Set Difference –:** Tuples that are in the first but not the second relation

Output will not have any duplicated results

Relational algebra doesn't care about keys

- Notation:
  - $R_{in1} \cup R_{in2}$ (Union),
  - $R_{in1} \cap R_{in2}$ (Intersection),
  - R_{in1} - R_{in2} (Difference),

same # of cols && same type of attributes

- Usual operations on sets
- R_{in1} and R_{in2} must be set-compatible,
  - same number of attributes
  - corresponding attributes must have the same type
  - no need for same name
- Result schema
  - same as the schema of the input relations
  - possibly renamed attributes

peration

start from inside

$$\Pi_{sname,rating}(\sigma_{rating > 8}(Skaters))$$

A – B: everything in A and not in B

Cross-Product: Each row of first table is paired with each row in second table |A| X |B| = |A X B|

Joins = cross-product + selection

Condition Join (Theta-Join): $R_{out} = R_{in1} \bowtie_c R_{in2} = \sigma_C(R_{in1} \times R_{in2})$

`Skaters` $\bowtie_{\texttt{Skaters.rating > OurSkaters.rating}}$ `OurSkaters`

Equi-Join: $R_{out} = R_{in1} \bowtie_{Rin1.a1 = Rin2.b1 \wedge \ldots Rin1.an = Rin2.bn} R_{in2}$

A special case of condition join where the condition C contains only *equalities*.

`Skaters` $\bowtie_{\texttt{Skaters.rating = OurSkaters.rating}}$ `OurSkaters`

Natural Join: Equijoin on all common attributes, i.e., on all attributes with the same name

– *Attributes do not need to be indicated in index of join symbol*

`Skaters` $\bowtie$ `Participates`

- ***Renaming :*** $\rho(R_{out}(B1,\ldots Bn), R_{in}(A1,\ldots An))$
  – Produces a relation identical to R_{in}
  – Output relation is named R_{out}
  – Attributes A1, … An of R_{in} renamed to B1, … Bn

Rules:

Equivalence: Let R, S, T be relations; C, C1, C2 conditions; L projection lists of the relations R and S

– Commutativity: → select all C needed.
  - $\prod_L(\sigma_C(R)) = \sigma_C(\prod_L(R))$
    – But only if C only considers attributes of L
  - $R1 \bowtie R2 = R2 \bowtie R1$

– Associativity:
  - $R1 \bowtie (R2 \bowtie R3) = (R1 \bowtie R2) \bowtie R3$

– Idempotence:
  - $\prod_{L2}(\prod_{L1}(R)) = \prod_{L2}(R)$
    – Only if $L2 \subseteq L1$
  - $\sigma_{C2}(\sigma_{C1}(R)) = \sigma_{C1 \wedge C2}(R))$

Integrity Constraints (CHECK)

❑ Problem of previous examples:
☆ what if constraints change (e.g., we want to increase rating constraint to (rating <=5 OR age > 5)

❑ Solution: name constraints:

Alter constraint (But already one violates the constraint → won't allow you alter constraint)

```
CREATE TABLE Skaters (
  sid INT NOT NULL,
  sname VARCHAR(20),
  rating INT CONSTRAINT rat CHECK
                (rating > 0 AND rating < 11),
  age INT,
  CONSTRAINT pk PRIMARY KEY (sid),
  CONSTRAINT ratage CHECK
              (rating <= 4 OR age > 5))
```

❑ This allows us to drop and recreate them later on

```
ALTER TABLE Skaters DROP CONSTRAINT ratage
ALTER TABLE Skaters ADD CONSTRAINT ratage
                        CHECK (rating <=5 OR age > 5)
```

Internal of DB

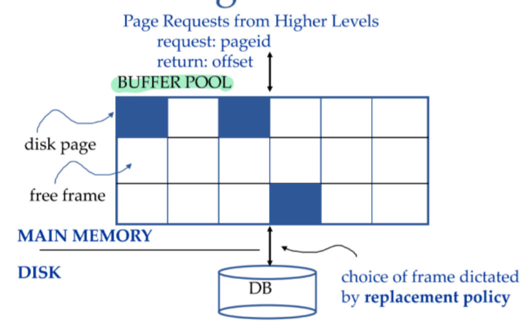
Query optimization and execution → relational operators → file and access methods → buffer management → disk space management

Structure: register → cache → main memory → disk

block = unit of transfer for disk r/w = page = frame

To change a page, bring the page from the disk to memory and then change.

Buffer Management in a DBMS



❑ Data must be in RAM for DBMS to operate on it!

❑ Table of <frame#, pageid> pairs is maintained.

❑ Some more information about each page in buffer is maintained

Buffer pool stores <frame #, page id>

Buffer manager will load up data to buffer pool when upper layer sends a request.

Loading a page from disk:

If requested page is not in pool: If there is an empty frame → Choose empty frame.

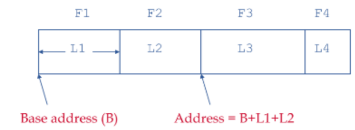
Else (no empty frame) → Choose a frame for replacement → If frame is **dirty** (page was modified), write it to disk → Read requested page into chosen frame.

Page Pin: replacement frame has pin counter 0.

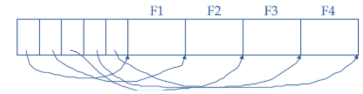
When requesting a page that is in the buffer → increment the pin counter. After finishing the operation → decrement pin counter (set dirty bit if page has been modified). Replacement policy → only pin counter = 0 could be chosen to replace.

Record Format

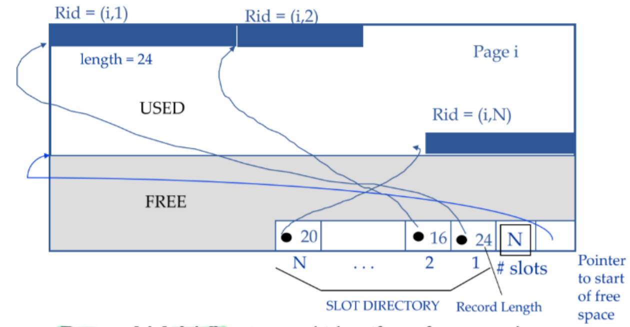
1. Fixed length (Works with fixed-length types)



2. Variable length (efficient storage of nulls)



Page Formats: Variable Length Records



☛ **Record id (rid)** = internal identifier of a record: <page id, slot #>.

☛ Can move records on page without changing rid;

(record id) Rid = <page id, slot #>

File = collection of pages

Unordered (Heap) File: Suitable when typical access is a full scan of all records

Sorted file: Records are sorted by one of the attributes (e.g., name).

Indexes: We call the collection of attributes over which the index is built the ***search key*** **attributes** for the index.

indirect Indexing:

- Indirect Indexing I: on non-primary key search key: (2015, rid1), (2015, rid2), (2015, rid3), … → several entries with the same search key side by side

- Indirect indexing II: on non-primary key search key: (2015, (rid1, rid2, rid3,…))

Direct Indexing: store record directly instead of rid

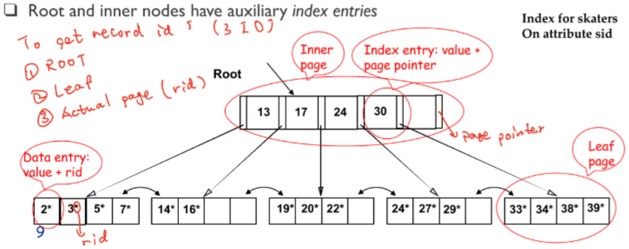
Primary vs. secondary: If **search key** contains primary key, then called primary index. (unique)

Clustered vs. unclustered: Def of **clustered**: Relation in file sorted by the search key attributes of the index. (A file can be clustered on **at most one** search key.) Cost of retrieving data records through index varies greatly based on whether index is clustered or not!

B+ Tree: The Most Widely Used Index

Each node is one page

- ❑ Each node/leaf represents one page
  - ☆ Since the page is the transfer unit to disk
- ❑ Leafs contain *data entries* (denoted as k*)
  - ☆ For now, assume each data entry *represents* one tuple. The data entry consists of two parts
    - Value of the search key (k)
    - Record identifier (rid = (page-id, slot))
  - ☆ That is: data entry is NOT a tuple but a pointer to a tuple
- ❑ Root and inner nodes have auxiliary *index entries*



Node = Page; Inner page + Leaf page

Data Entry = Leaf = value of search key + rid

Index Entry = page pointer (point to the next level of B+ tree) + value = Root + Inner nodes (min 50% occupancy)

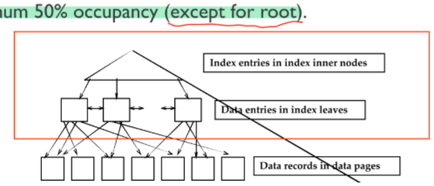
Height-balanced

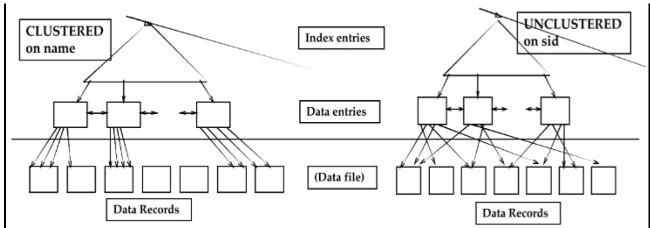
Fanout = F = number of children for each node

N = number of leaf pages

Cost of insert/delete = logF(N)

- ❑ *height-balanced.*
  - ☆Each path from root to tree has the same height
- ❑ F = fanout = number of children for each node (~ number of index entries stored in node)
- ❑ N = # leaf pages
- ❑ Insert/delete at $\log_F N$ cost;
- ❑ Minimum 50% occupancy (except for root).





Cost Model for Execution: number of I/Os

assumption that the root and all intermediate nodes of the B+ are in main memory: only leaf pages may not be in main memory!

Selection
Selectivity / Reduction Factor

- Reduction Factor of a condition is defined as
  - – $Red(\sigma_{condition}(R)) = |\sigma_{condition}(R)| / |R|$ # of qualifed / # of total
  - – $Red(\sigma_{experience=5}(Users)) = |\sigma_{experience=5}(Users)| / |Users| =$
    e.g., 10.000/40.000 = 0.25 (assuming 10,000 user have experience of 5)
- If not known, DBMS makes simple assumptions
  - – $Red(\sigma_{experience=5}(R)) = 1/|\text{different experience levels}| = 0.1$
    - Uniform distribution assumed
  - – $Red(\sigma_{age<=16}(R)) =$
    $(16 - \min(age)+1) / (\max(age) - \min(age)+1) = (16 - 11)/(61 - 12+1) = 5/50 = 0.1$
  - – $Red(\sigma_{experience=5 \text{ and } age <= 16}(R)) = ?$ → DB assumes independent!
- Result sizes: number of input tuples * reduction factor
- How to know number of different values, how many of a certain value, max, min…
  - through indices, heuristics, separate statistics (histograms)

10

Indices usually only useful with very **small reduction factors**

Naming constraints

- ❑ Problem of previous examples:
  - ☆ what if constraints change (e.g., we want to increase rating constraint to (rating <=5 OR age > 5)
- ❑ Solution: name constraints:

Alter constraint
(But already one violates the constraint → won't allow you alter constraint)

```
CREATE TABLE Skaters (
  sid INT NOT NULL,
  sname VARCHAR(20),
  rating INT CONSTRAINT rat CHECK
                    (rating > 0 AND rating < 11),
  age INT,
  CONSTRAINT pk PRIMARY KEY (sid),
  CONSTRAINT ratage CHECK
               (rating <= 4 OR age > 5))
```

- ❑ This allows us to drop and recreate them later on

```
ALTER TABLE Skaters DROP CONSTRAINT ratage
ALTER TABLE Skaters ADD CONSTRAINT ratage
                        CHECK (rating <=5 OR age > 5)
```

  - ❑ what if there is already a record with rating = 11 and age = 2 ?


```
SELECT sname, rating
, COALESCE(rating, 0) modrating,
COALESCE(rating, 0)+1 newrating
FROM skaters;

CREATE VIEW activeSkaters (sid,sname) AS
SELECT DISTINCT s.sid, s.sname
FROM  skaters s, participates p
WHERE s.sid = p.sid
;

SELECT *
FROM activeSkaters
;

DROP VIEW activeSkaters;

-- ERROR !! aggregate functions are not allowed in WHERE
SELECT sname
FROM skaters
WHERE rating = MAX(rating)
;

-- smaller or equal to all of the result
SELECT *
FROM skaters
WHERE rating <= ALL (SELECT rating FROM skaters)
;

-- large than any one of the result
SELECT *
FROM skaters
WHERE rating > ANY (SELECT rating FROM skaters)
;
```

```sql
SELECT DISTINCT column, AGG_FUNC(column_or_expression), …
FROM mytable
    JOIN another_table
      ON mytable.column = another_table.column
    WHERE constraint_expression
    GROUP BY column
    HAVING constraint_expression
    ORDER BY column ASC/DESC
    LIMIT num_limit OFFSET num_offset;
```

Query order of execution

1. **`FROM` and `JOIN`s:** determine the total working set of data that is being queried.
2. **`WHERE`:** constraints are applied to the individual rows, and rows that do not satisfy the constraint are discarded. Aliases in the `SELECT` part of the query are not accessible
3. **`GROUP BY`:** As a result of the grouping, there will only be unique values in that column. Implicitly, this means that you should only need to use this when you have *aggregate functions* in your query.
4. **`HAVING`:** If the query has a `GROUP BY` clause, then the constraints in the `HAVING` clause are then applied to the grouped rows, discard the grouped rows that don't satisfy the constraint.
5. **`SELECT`:** select required columns from table
6. **`DISTINCT`:** remove duplicated rows
7. **`UNION` \ `INTERSECTION` \ `EXCEPT` \ `UNION ALL`:** `ALL` keyword is used to allow duplicates
8. **`ORDER BY`:** order by ...
9. **`LIMIT` \ `OFFSET`:** limit to show only few rows (could also apply the offset)

Constraints

Operator	Condition	SQL Example
=, !=, < <=, >, >=	Standard numerical operators	col_name **!=** 4
BETWEEN … AND …	Number is within range of two values (inclusive)	col_name **BETWEEN** 1.5 **AND** 10.5
NOT BETWEEN … AND …	Number is not within range of two values (inclusive)	col_name **NOT BETWEEN** 1 **AND** 10
IN (…)	Number exists in a list	col_name **IN** (2, 4, 6)
NOT IN (…)	Number does not exist in a list	col_name **NOT IN** (1, 3, 5)

NULL

Sometimes, it's also not possible to avoid `NULL` values, as we saw in the last lesson when outer-joining two tables with asymmetric data. In these cases, you can test a column for `NULL` values in a `WHERE` clause by using either the `IS NULL` or `IS NOT NULL` constraint.

Operator	Condition	Example
=	***Case sensitive*** exact string comparison (*notice the single equals*)	col_name **=** "abc"
!= or <>	***Case sensitive*** exact string inequality comparison	col_name **!=** "abcd"
LIKE	Case insensitive exact string comparison	col_name **LIKE** "ABC"
NOT LIKE	Case insensitive exact string inequality comparison	col_name **NOT LIKE** "ABCD"
%	Used anywhere in a string to match a sequence of zero or more characters (only with LIKE or NOT LIKE)	col_name **LIKE** "%AT%" (matches "AT", "ATTIC", "CAT" or even "BATS")
_	Used anywhere in a string to match a single character (only with LIKE or NOT LIKE)	col_name **LIKE** "AN_" (matches "AND", but not "AN")
IN (…)	String exists in a list	col_name **IN** ("A", "B", "C")
NOT IN (…)	String does not exist in a list	col_name **NOT IN** ("D", "E", "F")

JOIN

- The `INNER JOIN` is a process that matches rows from the first table and the second table which have the same key (as defined by the `ON` constraint) to create a result row with the combined columns from both tables.
- When joining table A to table B, a `LEFT JOIN` simply includes rows from A regardless of whether a matching row is found in B. The `RIGHT JOIN` is the same, but reversed, keeping rows in B regardless of whether a match is found in A. Finally, a `FULL JOIN` simply means that rows from both tables are kept, regardless of whether a matching row exists in the other table.

Expressions

```sql
-- rename with `AS`
SELECT particle_speed / 2.0 AS half_particle_speed
FROM physics_data
WHERE ABS(particle_position) * 10.0 > 500;

SELECT col_expression AS expr_description, …
FROM mytable;
```

```sql
-- EXISTS, corelated subquery
SELECT s.sname
FROM skaters  s
WHERE EXISTS (
                SELECT *
                FROM participates p
                WHERE p.cid = 101 AND p.sid = s.sid
              )
;
```

Aggregate functions

Function	Description
COUNT(*), **COUNT(**column**)**	A common function used to counts the number of rows in the group if no column name is specified. Otherwise, count the number of rows in the group with non-NULL values in the specified column.
MIN(column**)**	Finds the smallest numerical value in the specified column for all rows in the group.
MAX(column**)**	Finds the largest numerical value in the specified column for all rows in the group.
AVG(column**)**	Finds the average numerical value in the specified column for all rows in the group.
SUM(column**)**	Finds the sum of all numerical values in the specified column for the rows in the group.

Operations

```sql
-- insert rows
INSERT INTO mytable
VALUES (value_or_expr, another_value_or_expr, …);

INSERT INTO mytable
(column, another_column, …)
VALUES (value_or_expr, another_value_or_expr, …);

-- update rows
UPDATE mytable
SET column = value_or_expr,
    other_column = another_value_or_expr,
    …
WHERE condition;

-- delete rows
DELETE FROM mytable
WHERE condition;

-- create table
CREATE TABLE mytable (
    column DataType TableConstraint DEFAULT default_value,
    another_column DataType TableConstraint DEFAULT default_value,
    …
);

-- altering table
ALTER TABLE mytable
ADD column DataType OptionalTableConstraint
    DEFAULT default_value;

ALTER TABLE mytable
DROP column_to_be_deleted;

ALTER TABLE mytable
RENAME TO new_table_name;

-- drop
DROP TABLE mytable;
```