CS2102 Cheatsheet AY22/23 S2 —— @JasonYapzx

- Atomicity - all/none of T reflected in the database
- Consistency - T guarantees the correct state
- Isolation - T isolated from effects of concurrent transactions
- Durability - after T, effects are permanent
- *Data model:* collection of concepts for describing data.
- *Schema:* desc. of structure of a database using a data model.
- **superkey**: subset of attributes that uniquely identifies a tuple
- **key**: is a ***superkey*** that is also minimal → no proper subset of the key is a superkey. (cannot be made smaller) ≠ smallest
- **candidate keys**: set of all **keys** of a given relation
- **primary key**: selected **candidate keys**, and they CANNOT be NULL. To simplify our notation, we underline our primary keys in the schema notation.
- **Foreign Keys:** subset of attributes of relation R_1 that refers to the **PK** of relation R_2. → R_1: referencing, R_2: referenced relation
- **Requirements**: each FK in R_1 must appear as a PK in R_2 **OR** be NULL value (in a tuple, contain at least 1 NULL value)

Data Types in SQL boolean: `true/false` (`null` == unknown) — **integer:** signed 4 byte integer — **float8:** double-precision floating point number (8 bytes) — **numeric:** arbitrary precision floating point number — **numeric(p, s):** max total of p digits with max of s decimals — **char(n):** fixed-length string consisting of n characters — **varchar(n):** variable-length string up to n characters — **text:** variable-length character string — **date:** calender date (year, month, day) — **timestamp:** date and time

Is Null Predicate: `x IS NULL`

Is Distinct from Predicate: `x IS DISTINCT FROM y`

non-null: `name varchar(100) NOT NULL,`

unique: `studentId INT UNIQUE,` or `unique (city, state) -- at bottom`

primary key: `studentId INT PRIMARY KEY` or `PRIMARY KEY (eid, pname), -- at bottom`

foreign key: `studentId INT REFERENCES Student (id)` or `FOREIGN KEY (a, b) REFERENCES Other (a, b) -- at bottom`

Foreign Key Constraints Violations

No Action: Rejects action if violates constraint (default) — *Restrict:* Same as No action but constraint checking is not deffered — *Cascade:* Propagates delete/update to referencing tuples — *Set NULL:* Updates foreign keys to NULL — *Set Default:* Updates foreign keys to some value, need to speicfy this value as the referecing column and must meet foreign key constraints otherwise fails

CREATE/ALTER/DROP Table

```sql
ALTER TABLE <table_name> [ALTER / ADD / DROP]
    [COLUMN / CONSTRAINT] <name> <changes>;
DROP TABLE [IF EXISTS] // no error if does not exist
<table_name>[, <table_name> [, <table_name> [...]]]
    [CASCADE]; // also delete referencing table
```

Entity-Relationship (ER) Model

Entity: Real-world object distinguishable from other objects
Attribute: Specific information describing an entity, ovals
Entity set: Collection of similar entities, rectangles
Key: Represented as underlined attributes
Relationship: Association among 2 or more entries *Relationship set*: Collection of similar relationships, represented by diamonds. Attributes used to describe information about relationships.

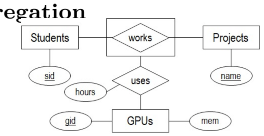
Relationship Constraints

- *Key Constraint*: E **at most one** of R (→)
- *Total Part Constraint*: E **at least 1** of R (=)
- *Key & Total Part Constraint*: E **exactly 1** of R (⇒)
- *Weak Entity Set*: E's identifying owner is EE, identifying relationship set: R. E does not have own key to be uniquely indentified.
- *Partial Key*: Set of attributes of a weak entity set that uniquely identifies a weak entity for a given owner entity.

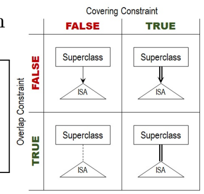
1. Can **PK** be used to uniquely identify other attributes
2. Is the LOWER BOUND satisfied in the schema?
3. Is the UPPER BOUND satisfied in the schema?

Extended Notations - Aggregation

Abstraction that treats relationships as higher-level entities. Treat it as a relation class in OOP.



ISA: Every entity subclass is an entity in its superclass.



Overlap: Can a superclass belong to multiple subclasses?
Covering: Must a superclass belong to at least one subclass?

Subqueries

- WHERE - Pattern Matching: `_` matches any single character, `%` matches any sequence of 0 or more characters.
  - start w/ 'Ma', end with 'a': `WHERE pizza LIKE 'Ma%a'`
  - starts w/ 'A' and ≥ 5 chars: `WHERE name LIKE 'A____%'`
- SET OPERATIONS `Q1 UNION/INTERSECT/EXCEPT [ALL] Q2` `ALL` does not remove duplicates
- JOIN OPERATIONS: Most common in practice: cross product + selection condition + attribute selection

Scalar Subqueries: A query that returns at most a single value.

[NOT] IN: returns exactly one column.
- `IN` returns `TRUE` if `<expr>` matches any subquery row
- `NOT IN` returns `TRUE` if `<expr>` matches no subquery row

ANY/SOME: Expression `<expr>` is compared to each row from subquery using the operator `<op>`, returns `TRUE` if comparison evaluates to `TRUE` for **at least one row** in the subquery

ALL: same as `ANY` except needs **ALL** rows

[NOT] EXISTS: May return any number of columns
- `EXISTS` returns `TRUE` if the subquery returns at least one row
- `NOT EXISTS` returns `TRUE` if the subquery returns no row

⋆ Not all constructs required: IN ≡ ANY ≡ EXIST

Order By: `ORDER BY <attribute> ASC/DESC`: (default for SQL, ASC can be removed). If duplicate removal needed, attribute being sorted **must** appear in `SELECT`. Sorting w.r.t multiple attributes / differing order supported

LIMIT k: Return the first* `k` rows of the result table

OFFSET i: Specify the position of the first row to be considered

⋆ `LIMIT` and `OFFSET` typically meaningful only in with `ORDER BY`

Aggregate Functions

Function	Input Type	Output Type
MIN	*any comparable type*	*same as input*
MAX	*any comparable type*	*same as input*
SUM	Numeric data *(e.g., INT, BIGINT, REAL, etc)*	SUM(INT) → BIGINT; SUM(REAL) → REAL
COUNT	*any data*	BIGINT

Aggregate functions compute a ***single value*** from a set of tuples.

If `R` is empty set with `B` attribtue, `COUNT(*) = 0, COUNT(B) = 0, SELECT aggrFn(B) FROM R = NULL`. If `S` is non-empty relation with n-rows of attribute `A` with only `NULL` values in table `COUNT(*) = n, COUNT(A) = 0` and `SELECT aggrFn(A) = NULL`

Grouping (GROUP BY):

⋆ f column A_i appears in SELECT, one condition must hold
- A_i appears in the GROUP BY clause
- A_i appears as input of aggregation function in SELECT clause
- The pkey of R appears in the GROUP BY clause

HAVING: `WHERE` clauses only works on each row. If we were to use `WHERE p < AVG(p)`, any p that does not fit this condition will be removed, resulting in `AVG(p)` to change. `HAVING` instead aggregates the condition for each group defined by `GROUP BY`.

⋆ If column A_i appears in HAVING, one condition must hold
- A_i appears in the GROUP BY clause
- A_i appears as input of aggregation function in HAVING clause
- The pkey of R appears in the GROUP BY clause

Conceptual Evaulation of Queries 1. `FROM`: Compute cross-product of all tables in `FROM`. 2. `WHERE`: Filter (keep) tuples evaulate to `TRUE` on `WHERE`. 3. `GROUP BY`: Partition tables into groups wrt to grouping attributes. 4. `HAVING`: Filter groups evaulate to `TRUE` on `HAVING`. 5. `SELECT`: Remove all attributes not specified in `SELECT` (remove dup if `DISTINCT`). 6. `ORDER BY`: Sort tables based on specified attribute. 7. `LIMIT/OFFSET`: Filter tuples based on their order in the table.

Conditional Expressions: `CASE` Only one of the results will be returned, `ELSE` optional, `NULL` returned if no condition matched.
`CASE WHEN <condition_1> THEN <result_1> ... ELSE result_0 END`

`COALESCE` Returns first non-NULL value in list of input arguments (order of `order` matters!)
`COALESCE(<value_1>, <value_2>, <value_3>, ...)`

`NULLIF` Returns NULL if `<value_1>` = `<value_2>`, else `<value_1>`

Common Table Expression

```sql
WITH
CTE_1 AS ( Q_1 ),
...
CTE_n AS ( Q_n )
Q_0 -- main SELECT ↩
    statement:
```

- Each CTE_i can reference any other CTE_j that has been declared before (i.e., j < i)
- Main SELECT statement Q_0 can reference any possible subset of all CTE_i → Any CTE_i not referenced can be deleted

Views Usually need parts of the table, restrict access to table for certain users, and often use the same queries and subqueries frequently. `CREATE OR REPLACE VIEW <name> AS <query>`

Functions

```sql
CREATE OR REPLACE FUNCTION
    markCnt(OUT TopMark INT, OUT Cnt INT)
RETURNS SETOF RECORD Scores AS $$
    SELECT Mark, COUNT(*) FROM Scores GROUP BY Mark;
$$ LANGUAGES sql;
```

`SETOF` used when want to return multiple records. `RECORDS` used for only 1 record, and not in the current schema. `SETOF RECORDS` is used for multiple tuples, not in the current schema. `TABLE` is similar to `SETOF RECORDS`.

Procedures No return value/tuple needed, may use **SQL procedures**. `CALL transfer('Alice', 'Bob', 100);`
`CREATE OR REPLACE PROCEDURE g(x int) as $$`

Recursive Queries

```
WITH RECURSIVE
    CTE_name AS (
        Q_1
        UNION [ ALL ]
        Q_2 ( CTE_name )
    )
Q_0 ( CTE_name )
```

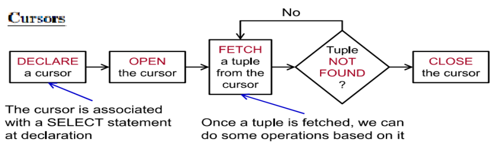
```
WITH RECURSIVE
Linker(to_stn, stops) AS (
    SELECT to_stn, 0
    FROM MRT
    WHERE fr_stn = 'NS1'
    UNION ALL
    SELECT M.to_stn, L.stops + 1
    FROM Linker L, MRT M
    WHERE L.to_stn = M.fr_stn
        AND L.stops < 2
)
SELECT DISTINCT (to_stn)
FROM Linker;
```

- Q_1 is non-recursive
- Q_2 is recursive and can reference CTE_{name}
- Query is evaluated 'lazily', stops when a fixxed-point is reached

Variables and Control Structures

```
AS $$ DECLARE
    temp_val INTEGER;
BEGIN -- functinon body
END;
$$ language plpgsql;
_________________________
LOOP
    EXIT WHEN condition;
    -- loop body;
END LOOP;
```

```
IF condition1 THEN
    statement1;
ELSEIF condition2 then
    statement2;
ELSE
    else-statement;
END IF;
```




```
CREATE OR REPLACE FUNCTION score_gap()
RETURNS TABLE(name TEXT, mark INT, gap INT) AS $$
DECLARE // Declares cursor variable
    curs CURSOR FOR (SELECT * FROM Scores ORDER BY Mark DESC);
    r RECORD;
    prv_mark INT;
BEGIN
    prv_mark := -1;
    OPEN curs;
    LOOP
        FETCH curs INTO r; EXIT WHEN NOT FOUND;
        name := r.Name; mark := r.Mark;
        IF prv_mark >= 0 THEN gap := prv_mark - mark;
        ELSE gap := NULL; END IF;
        RETURN NEXT; prv_mark := r.Mark;
    END LOOP;
    CLOSE curs;
END;
$$ LANGUAGE plpgsql;
```

- `FETCH PRIOR/FIRST/LAST FROM cur INTO r` → fetch the prior/first/last row
- `FETCH ABSOLUTE 3 FROM cur INTO r` → fetch the 3^{rd} tuple

Triggers ↔ Trigger Functions

```
CREATE TRIGGER trigger AFTER INSERT ON Table
FOR EACH ROW EXECUTE FUNCTION trigger_fn();
CREATE OR REPLACE FUNCTION trigger_fn()
RETURNS TRIGGER AS $$
BEGIN /*Trigger Logic*/  END; $$ LANGUAGE plpgsql;
```

- `RETURNS TRIGGER`' indicates that this is a trigger function, only `RETURNS TRIGGER` allowed as only `TRIGGER` has access to the `NEW`.
- `NEW` refers to new row inserted into Scores
- `CURRENT_DATE` returns the current date
- `TG_OP`: operation that activates trigger: `INSERT`, `UPDATE`, `DELETE`
- `TG_TABLE_NAME`: name of table that cause trigger invocation
- `OLD`: old tuple being updated/deleted

Return Values of Trigger Function

- `BEFORE INSERT`: non-null `t` — `t` inserted, null `t` — no insertion
- `BEFORE UPDATE`: non-null `t` — `t` updated, null `t` — no update
- `BEFORE DELETE`: non-null `t` — `t` deleted, null `t` — no deletion
- `AFTER INSERT/UPDATE/DELETE` return value does not matter
- `INSTEAD OF` non-null `t` — proceed, null `t` — ignore operations on current row. If a `BEFORE` row-level trigger returns `NULL`, then all subsequent triggers on same row are omitted. (`VIEW`s only)

Trigger Levels — `FOR EACH ROW`: **row-level** trigger that executes trigger function for every tuple encountered. `STATEMENT`: **statement-level** trigger that executes the trigger function only once. `EXCEPTION`s raised to ignore operations for statement-level triggers. `NOTICE`s does not stop operations

Trigger Condition Has the following rules: e.g.
`FOR EACH ROW WHEN (NEW.Name = 'Hello') EXECUTE`

- No `SELECT` in `WHEN()`
- No `NEW` in `WHEN()` for `DELETE`
- No `OLD` in `WHEN()` for `INSERT`
- No `WHEN` for `INSTEAD OF`

Deffered Triggers

```
CREATE CONSTRAINT TRIGGER bal_check_trigger
AFTER INSERT OR UPDATE OR DELETE ON Account
DEFFERABLE INITIALLY DEFERRED
FOR EACH ROW EXECUTE FUNCTION bal_check_func();
```

- `CONSTRAINT` + `DEFERRABLE`: trigger can be deffered
- `INITIALLY DEFFERED`: by default, trigger is deffered. `INITIALLY IMMEDIATE` trigger not deffered by default.
- DEFFERED Triggers only work for `AFTER` and `FOR EACH ROW`, `AFTER` so that it can be executed after main operation.
- Order of trigger activation: `BEFORE` statement-level triggers → `BEFORE` row-level triggers → `AFTER` row-level triggers `AFTER` statement-level triggers
- Within each category, triggers activated in alphabetic order

Functional Dependencies: $A \to B$ means A decides B, if 2 rows have same A, then they have same B also.

Armstrong's Axioms

1. **Reflexivity:** AB → A
2. **Augmentation:** If A → B, then AC → BC
3. **Transitivity:** If A → B, and B → C then A → C
4. **Decomposition:** If A → BC then A→ B and A → C
5. **Union:** If A → B and A → C then A → BC

Closure: Four attributes *A, B, C, D*. Given that B → D, DB → A, AD → C, check if B → C holds. { B }$^+$ = { B } = { B, D }, since B → D = { B, D, A }, since DB → A = { B, D, A, C }, since AD → C. { B, D, A, C } reffered to as *closure* of B, which is the set of componenets that can be 'activated' by B

Superkeys: set of attributes that decides all other attributes. **Keys:** Superkey that is minimal. Whether or not a table has *redundancy* or *anomalies* depend on keys. **Prime Attribute:** Attribute that appears in a key. **Finding Keys:** any attribute that *does not appear in RHS*, it **must be in every key**. If key: e.g. { A, C } able to activate all other attributes, do no need to check supersets for *keys* as keys are minimal

Normal Forms BCNF requires that if there is any **non-trivial** and **decomposed** FD $A_1A_2...A_n$ → B, then $A_1A_2...A_n$ must be a superkey. More simply, BCNF requires LHS to be super keys, prevent redundancy.
Decomposed FD: FD whose RHS only has 1 attribute
Non-Trivial: FD whose RHS does not appear in LHS
Completely Non-Trivial: $LHS \cap RHS = \emptyset$ 1. Compute closure of each attribute subset, 2. Derive keys of R (using closures) 3. Derive non-trivial and decomposed FDs from each closure, 4. For each of the FD, if all LHS is super key then R satisfy BCNF
Simplified BCNF: Violation of BCNF *iff* we have a closure that satisfies the **more but not all** condition or some attribute not in the schema's attributes { $A_1A_2...A_n$ }.

Normalization Algorithm

1. Find subset X of attributes in R, such that its closure $\{X\}^+$ (i) does not contain more attributes than X, but (ii) does not contain all attributes in R
2. Decompose R to form 2 tables R_1 and R_2 such that
   - R_1 contains all attributes in $\{X\}^+$
   - R_2 contains all attributes in X and attributes not in $\{X\}^+$
3. If R_1 not in BCNF, further decompose R_1, same for R_2

BCNF decomposition not unique. If table only has 2 attributes, it MUST BE in BCNF: Check ∅ set, A→B, B→A or both.

Projection of Closures/FDs: If we derive closures on table R_i decomposed from table R, we can enumerate attribute subsets of R. For each subset, derive its closure on R, and project each closure onto R by removing attributes that do not appear in R_i.
Properties of BCNFs: No `UPDATE/DELETE/INSERT` anomalies. Small redundancy. Original table can be reconstructed from decomposed table. *Lossless join:* no extra/loss of tuples.

- BCNF gurantees lossless join whenever common attributes in R_1 and R_2 contain a superkey of R_1 or R_2
- Decomposing R, R_1 contains all attributes in $\{X\}^+$, R_2 contains all attributes in X as well as those not in $\{X\}^+$
- Thus X will be the common attributes between both R_1 and R_2, X will be a superkey of R_1

However, dependencies may not be preserved.

3NF: *iff* for every non-trivial and decomposed FD: 1. Either the LHS is **superkey**, 2. OR RHS is a **prime attribute** (appears in a key) ⋆ 3NF more lenient than BCNF. **FD Equivalence:** Prove that Given *S, S'* can be derived, and vice versa.

3NF Decomposition Algorithm

1. Given table R and set S of FDs, Derive minimal basis of S
2. In minimal basis, combine FDs whose LHS are the same
3. Create a table for each FD remained
4. If non of the tables contains a key of original table R, create table that contains a key of R (any key of R is ok)
5. Remove redundant tables

Minimal Basis: Given set S of FDs, minimal basis of S, M is a simplified version of S, such that M equivalent to original set S.

1. every FD in S can be derived from M, and vice versa
2. every FD in M is a non-trivial and decomposed FD
3. if any FD removed from M, some FD in S ×-derived from M
4. for any FD in M, if we remove an attribute from its left hand side, then the FD cannot be derived from S

Find Minimal Basis: 1. Transform FDs so RHS 1 attr only 2. Remove redundant attr on LHS. 3. Remove redundant FDs
Adding Key for Lossless Join: $R(A,B,C,D)$ $A \to B, C \to D$

- Minimal basis: $A \to B, C \to D$; tables: $R_1(A,B), R_2(C,D)$
- R_1 and R_2 cannot be used to reconstruct R, require that the tables contain a key of R, if not then create one: AC is the only key of R, need add table $R_3(A,C)$