1 Creating and Populating Tables

Basic types: `int | numeric(precision, scale) | varchar(n) | text | date | boolean`
Integrity Constraints: `not null, primary key, unique, foreign key, check`
Good Practices: Avoid not null, justify cascade, Defer constraint checks until end of transaction

2 Simple Queries

Cases: `case when ... then ... else ... end;`
Null: has the value of 'unknown', comparator (`< > <> = >= <=`) and arithmetic operations return null. `null is null` is true. `1 is null` is false. `null is not null` is false. `1 is not null` is true. `coalesce(x1, x2, ...)` returns the first non-null of its arguments.
Count: `count(*)` counts null values. `count(col)` does not count null values.
Cartesian Product: `cross join`

3 Algebraic and Aggregate Queries

Aggregation Functions: `count(), sum(), max(), min(), avg(), stddev()`. Use `distinct` to select distinct rows. Aggregation functions can be used in conditions, but not in the `where` clause. Aggregation functions can be evaluated after groups are formed (i.e. after the `where` clause) in the `having` clause.
Truncate: `trunc(..., dp)` truncates a value to decimal places.
Renamed Columns: `extract(col from t.c) as ...`

4 Entity Relation Model

Aggregation: Associating an entity set with a relationship set.
Look here convention: cardinality describes participation of entity set it is attached to.
Convention: If cardinality is omitted, many-to-many i.e. (0,n) is assumed
Weak entity: can only be defined for a participation constrained by (1,1) cardinalities, a.k.a mandatory one-to-many relationships
Design advice: avoid null values, create a separate entity set
Schema Translation Rule 1: Every attribute must have a meaningful type
Rule 2: Every entity set becomes a relation (i.e. table). Candidate keys should be unique not null. Candidate keys are all sets of attributes that uniquely identify the entries. There must be at least one primary key.
Rule 3: Relationship sets are mapped to relations. The attributes of the relation consist of the attributes of the relationship set. The keys are the keys of the participating entities.
Exception 1: One-to-many. Enforce (0,1) cardinality (upper bound). Restrict the primary key.
Exception 2: (1,1) (one to one) cardinality (must participate) → Merge tables.
Exception 3: Weak entity set. Merge tables, add primary key of dominant entity set.
Limitation: Not all cardinalities can be enforced such as (1,n), but still try to enforce as much as possible.

5 Nested Queries

Permanent Copy: `create table <name> as (...)`
Temporary Copy: `create temporary table <name> as (...)`
View: `create view <viewname> as (select ...)`
Common Table Expression: `with <table> as (select ...) select ...`
Subquery in From: `select ... from (select ...) as ...`
Scalar Subquery: Returns only one row and one column.
Where In, Any, All: `where ... in | all | any (...)`
Where Exists: Evaluates to true if subquery has some result, false if no result
Correlated Subquery: `.. from t1 ... where (.. where t1... = )` . You can always use column from an outer table in an inner query. You can use subquery in select clause but it must be a scalar subquery. `select (select .. where t1... =) from t1`
Nested Queries: `where ... ALL | ANY | EXISTS | NOT EXISTS | IN | NOT IN (...)`
Nested Having: If you use aggregation functions on two different groupings. `... group by ... having count(*) >= all(...)`

6 Relational Algebra

Conjunction: $\wedge$, **Disjunction**: $\vee$
Negation: $\neg$
Selection: $\sigma[c](R)$ selects all rows from the relation R that satisfies the condition c (similar to `where` clause)
Projection: $\pi[l](R)$ keeps only the columns specified in the ordered list l and in the same order (similar to `select` clause)
Renaming: $\rho(R_1, R_2)$ renames R_1 to R_2. $\rho(R_1, R_2(A_1 \to A_2))$ renames the relation and some of its attributes
Set Operators: $\cup$ `union`, $\cap$ `intersect`, $-$ `except`. The two relations must be union-compatible (have the same column types)
Product: $R_1 \times R_2$ (cartesian product)
Join: $R_1 \bowtie [c] R_2$ is defined as $\sigma[c](R_1 \times R_2)$, i.e. only the tuples that satisfies the condition after the cross product. Omit condition = natural join. Left outer: $\bowtie$ right outer: $\ltimes$ full outer: $\bowtie$

7 Stored Procedures

`create [or replace] function <fn_name> [ returns <rettype> | returns table(col_name col_type, ...) ] language plpgsql as $$ declare begin end; $$;`
Return single row (all columns): `returns <table_name>`
Return multiple rows: `returns setof <table_name>`
Return new tuple: `...(out var1 type, out var2 type) returns record as ...` Must have at least two out parameters.
No return value: `return void` (Procedure). Need to `call` a procedure
Procedure vs Function: Function must return something (special case is void), procedure has no return, but can use out parameter. Transaction: procedure can commit or rollback, function cannot
Variables: `temp := val1` (Walrus operator)
Selection: `if ... then ... else if ... else ... endif;`
Repetition: `while ... loop ... end loop;` or `loop exit when ... end loop;` (forever loop)
Cursor: Access individual rows returned by `select`. 1. Declare cursor, 2. open cursor, 3. fetch tuple from cursor, perform some operations, go to next tuple. 4. if tuple not found, close the cursor. Store tuples of each row in a record.
`declare curs1 cursor for (select ...); record r; begin open curs; loop fetch curs into r; exit when not found; ...return next; end loop; close curs;`
Cursor Movement: `fetch prior | first | last | absolute <row number> from <cursor> into <var>`
SQL Injection Attack: Malicious user inserts their own code into the generated query. Prepared statements protect against this: sql statement is compiled when it is prepared. At runtime, anything is treated as a string. Same for stored functions and procedures

8 Triggers

`create [or replace] [constraint] trigger <name> {before|after|instead of } {<event> [or...]} on <table_name> [not deferrable | [deferrable] [initially immediate | initially deferred]] [for [each] {row | statement}] [when ( condition )] execute {function | procedure} <function_name> (arguments)`
Event can be one of: `insert | update [of <column_name> [, ... ]] | delete | truncate`
Rules: `instead of` only allowed with `for each row` for views. `before` and `after` can be used for rows/statements on tables, but must be `for each statement` for views. `truncate` only allowed with `for each statement`
Granularity: `for each statement` ignores return values by trigger functions. `return null` would not make the database omit the subsequent operation. `raise exception` to abort current transaction
Conditions: No `select` in `when()`, No `old` in `when()` for `insert`, No `new` in `when()` for `delete`, No `when()` for `instead of`
Deferred Trigger: Check constraint at the end of a transaction. `constraint` must be used with `deferrable`. Defaults to `initially immediate`. Only works with `after` and `for each row`
Order of activation: before statement → before row → after row → after statement. Within each category, alphabetical based on function name. If before row-level trigger returns `NULL`, then subsequent triggers **on the same row** are omitted (after statement not affected, other rows not affected unless they also return `NULL`)
Trigger Function: For row-level statement, return `NULL` to ignore all operations on current row. Non-`NULL` signals database to proceed as normal. Returning `NULL` skips the triggering operation but any statements executed inside may still take effect. `after` not affected by return

values. `old` not available for `insert`. `new` not available for `delete`. Both available for `update`.
Syntax: `create or replace function <fn_name>() returns trigger language plpgsql as $$ ... $$`
Views: Virtual table to hide complexity from users
Trigger Variables: `OLD, NEW, TG_OP, TG_NAME, TG_LEVEL, TG_TABLE_NAME, TG_NARGS, TG_ARGV`. Before triggers typically used for checking/modifying data. After triggers usually used to propagate updates to other tables or make consistency checks.

9 Normal Forms

Definition: A definition of minimum requirements to reduce redundancy, and improve data integrity
Purpose: to decompose (normalize) a table to prevent anomalies (insertion, deletion, update) and to remove redundancy. When a field needs to be modified independently of other fields, it should be split into another table
Functional Dependency: When two objects have the same values on certain fields $A_1, \ldots, A_n$, they always have the same values on $B_1, \ldots, B_n$
A FD may hold on one table but not on another
Axiom of Reflexivity: $AB \to A$
Axiom of Augmentation: If $A \to B$ then $AC \to BC$
Axiom of Transitivity: If $A \to B$ and $B \to C$ then $A \to C$
Rule of Decomposition: If $A \to BC$, then $A \to B$ and $A \to C$
Rule of Union: If $A \to B$ and $A \to C$, then $A \to BC$
Closure: the set of all attributes which a certain set of attributes implies
To prove that $X \to Y$, we only need to prove $\{X\}^+$ contains Y
To prove that $X \to Y$ does not hold, we only need to show that $\{X\}^+$ does not contain Y
Superkey: A set of attributes in a table that decides all other attributes (All attributes must appear in its closure)
Key: A superkey that is minimal: if we remove any attribute, it will not be a superkey anymore. A table may have multiple keys. Whether a table has redundancy depends partially on the keys
Algorithm for finding keys: 1. Consider every attribute subset, 2. Derive the closure of each subset, 3. Identify all superkeys, 4. Identify keys.
Trick 1: Check smaller attribute sets first. If you find a key, all supersets containing it will not be keys since they are not minimal)
Trick 2: Any attribute that does not appear on the right of any FD must be in every key
Prime Attribute: If an attribute appears in a key, it is a prime attribute. Otherwise, it is non-prime.

10 Boyce-Codd NF (BCNF)

1st-6th NF is increasing in strictness. 3rd NF and BCNF are always possible to satisfy and get rid of the most redundancies.
Decomposed FD: an FD whose right hand side has only one attribute. A non-decomposed FD can always be transformed into an equivalent set of decomposed FD
Non-trivial FD: e.g. $BC \to D$. $BC \to C$ is trivial since it is implied by reflexivity.
Algorithm to find non-trivial and decomposed FD: 1. Consider all attribute subsets, 2. Compute their closures, 3. Remove trivial attributes, 4. Derive FDs
BCNF: A table is in BCNF, if every non-trivial and decomposed FD has a superkey on its left hand side. For all FDs $A_1 A_2 \ldots A_n \to B$, B can depend only on superkeys. Any dependency on non-superkeys is prohibited by BCNF. Since if B depended on a non-superkey, every time that non-superkey appears, there could be duplicates of that value with B and hence redundancy
Algorithm to check BCNF: 1. Compute the FDs for all attribute subsets, 2. Derive the keys, 3. Derive the non-trivial and decomposed FDs, 4. For each non-trivial and decomposed FD, check if its left hand side is a superkey
Simplified Check: Compute the closure for each subset, then check for "more but not all" FDs (i.e. not a superkey). If such a closure exists, then the table is not in BCNF.
BCNF Decomposition: If a table is not in BCNF, we can decompose (normalize) it into smaller tables. The decomposition may not be unique. If a table only has two attributes, it must be in BCNF, so you don't need to check them
Algorithm for BCNF Decomposition: 1. Find a subset X such that $\{X\}^+$ contains more attributes than X but not all attributes, 2. Decompose the table in R_1 and R_2 such that R_1 has all the attributes in $\{X\}^+$, R_2 has all the attributes in X and all the attributes not in $\{X\}^+$, 3. Repeat on R_1 and R_2 until they are in BCNF. In general, each decomposition removes at least one violation of BCNF.
Projecting FDs: to derive closures on a table R_i that is decomposed from R, 1. enumerate the attribute subsets of R_i, 2. For each subset, derive its closure on R, 3. Project each closure onto R_i by removing those attributes that do not appear in R_i. Use the projected closures to check BCNF
Good properties of BCNF: No update/insertion/deletion anomalies, Small redundancy, Lossless join (original table can be reconstructed from decomposed tables without losing information)
Bad properties of BCNF: Functional dependencies may not be preserved. Need to join back tables to check for constraint integrity
Lossless: $R_1 \bowtie R_2 = R$
Lossy: $R_1 \bowtie R_2 \supset R$
Binary Lossless Decomposition: $(R_1 \cap R_2) \to R_1$, or $(R_1 \cap R_2) \to R_2$ i.e. the common attribute(s) is a superkey of either fragment.
Lossless Decomposition: If there is a sequence of lossless binary decompositions

11 3NF

Dependency Preservation: Decomposition preserves all FDs iff S (original set of FDs) and S' (set of FDs on decomposed tables) are equivalent i.e. they can be derived from each other
Definition: A table satisfies 3NF iff for every non-trivial and decomposed FD, either the left hand side is a superkey, or the right hand side is a prime attribute (appears in a key)
3NF is more lenient than BCNF. Satisfying BCNF = satisfying 3NF, but not vice versa. Violating 3ND = violating BCNF, but not vice versa. 3NF preserves all FDs, but can have update and deletion anomalies
3NF Check: 1. Compute the closure of all subsets, 2. Derive keys of R, 3. For each FD, check if left hand side is a superkey, or right hand side is a prime attribute.
BCNF vs 3NF Decomposition: BCNF performs one or more binary splits, 3NF makes only one split into two or more tables
3NF Decomposition: 1. Derive a *minimal basis* of S, 2. In the minimal basis, combine the FDs whose left hand sides are the same (Rule of Union), 3. Create a table for each FD remaining, 4. If none of the tables contains a key of the original table R, create a table that contains a key of R (to ensure lossless join), 5. Remove redundant (subsumed) tables
Minimal basis: A.k.a minimal cover.
1. Every FD in the minimal basis can be derived from S, and vice versa.
2. Every FD in the minimal basis is non-trivial and decomposed.
3. No FD in the minimal basis is redundant (no FD in the minimal basis can be derived from other FDs in the minimal basis. If any is removed, S cannot be derived).
4. No redundant attribute on the left hand side for each FD in the minimal basis (if an attribute is removed, then S cannot be derived)
Algorithm for Minimal Basis: 1. Transform FDs so that each right hand side contains only one attribute, 2. Remove redundant attributes on the left hand side of each FD, 3. Remove redundant FDs. (Minimal basis may not be unique)
BCNF vs 3NF: Go for BCNF if you can find a BCNF decomposition that preserves all FDs, or if preserving FDs is not important. Else, go for 3NF.