- Order of queries

```
select      distinct select-list
from        from-list
where       where-condition
group by    groupby-list
having      having-condition
order by    orderby-list
limit       limit-specification
offset      offset-specification
```

- Insert:

```
insert into Students
values      (12345, 'Alice', '1999-12-25', 'Maths')
```

- Delete

```
-- Remove all students
delete from Students;

-- Remove all students from Maths department
delete from Students
where       dept = 'Maths';
```

- Add/modify/delete columns:

```
alter table Students alter column dept drop default;
alter table Students drop column  dept;
alter table Students add column faculty varchar(20);
```

- Add/remove constraints:

```
alter table Students add constraint fk_grade foreign key
(grade) references Grades;
```

- IS NULL: true if x is NULL, false otherwise

- IS DISTINCT FROM: x<>y if both non-null, true if one null,

- Conceptual evaluation: WITH, FROM, WHERE, GROUP BY, HAVING, SELECT, DISTINCT, UNION/INTERSECT/EXCEPT, ORDER BY,LIMIT + OFFSET

- Non scalar queries: FROM, WHERE, HAVING

- Aggregate/Scalar queries: SELECT, HAVING, ORDER BY

- Aggregate functions

```
select min(A) from R
select max(A) from R
select avg(A) from R
select sum(A) from R
select count(A) from R
select count(*) from R
```

```
select avg(distinct A) from R
select sum(distinct A) from R
select count(distinct A) from R
```

- Each column in SELECT must (i) in GROUP BY (ii) an aggregate function (iii) primary key in GROUP BY

- Aggregate function in SELECT and no GROUP BY, SELECT must not have any column not in aggregate function

- Each column in HAVING must (i) in GROUP BY (ii) an aggregate function (iii) primary key in GROUP BY

- WITH

```
with
    R1 as (Q1),
    R2 as (Q2),
    ...,
    Rn as (Qn)
select/insert/update/delete statement S;
```

- CASE

```
select name, case
      when marks >= 70 then 'A'
      when marks >= 60 then 'B'
      when marks >= 50 then 'C'
      else 'D'
end as grade
from  Scores;
```

```
case expression
 when value1 then result1
 ...
 when valuen then resultn
 else result0
end
```

```
update Sells S
set price =
    case (select area from Restaurants where rname = S.rname)
    when 'Central' then price + 3
    when 'East' then price +2
    else price + 1
    end;
```

- COALESCE: returns first non-null value in arguments

```
select name, coalesce(third,second,first) as result
from   Tests;
```

- NULLIF: nullif(value1, value2) returns null if value1 = value2

```
select name, nullif(result,'absent') as status
from   Tests;
```

- LIKE: underscore matches single char, % matches >= 0 char

- PLPGSQL example

```
CREATE OR REPLACE FUNCTION convert (mark INT)
RETURNS char(1) AS $$
        SELECT  CASE
                WHEN mark >=70 THEN 'A'
                WHEN mark >= 60 THEN 'B'
                WHEN mark >= 50 THEN 'C'
                ELSE  'D'
                END;
$$ LANGUAGE sql;
```

- Returning an existing tuple (same columns, 1 row):

```
CREATE OR REPLACE FUNCTION topStudent ()
RETURNS Scores AS $$
        SELECT      *
        FROM        Scores
        ORDER BY    Mark  DESC LIMIT 1;
$$ LANGUAGE sql;
```

- Returning an existing set of tuple (same columns, multiple rows):

```
CREATE OR REPLACE FUNCTION topStudents ()
RETURNS SETOF Scores AS $$
        SELECT      *
        FROM        Scores
        WHERE       Mark =
                (SELECT MAX(Mark) FROM Scores);
$$ LANGUAGE sql;
```

- Returning a new tuple (diff columns, 1 row):

```
CREATE OR REPLACE FUNCTION topMarkCnt (OUT TopMark INT,
        OUT Cnt INT)
RETURNS RECORD AS $$
        SELECT Mark, COUNT(*)
        FROM   Scores
        WHERE Mark = (SELECT MAX(Mark) FROM Scores)
        GROUP BY      Mark ;
$$ LANGUAGE sql;
```

- Returning new set of tuples (diff columns, multiple row):

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

- Alternative:

```
CREATE OR REPLACE FUNCTION MarkCnt ()
RETURNS TABLE(Mark INT, Cnt INT) AS $$
        SELECT Mark, COUNT(*)
        FROM   Scores
        GROUP BY    Mark ;
$$ LANGUAGE sql;
```

- Use procedures when not returning anything:

```
CREATE OR REPLACE PROCEDURE transfer (fromAcct TEXT,
                toAcct TEXT, toAcct TEXT, amount INT)
AS $$
        UPDATE Accounts
        SET balance = balance - amount
        WHERE name = fromAcct;

        UPDATE Accounts
        SET balance = balance + amount
        WHERE name = toAcct;
$$ LANGUAGE SQL
```

- Use variables:

```
CREATE OR REPLACE FUNCTION swap (INOUT val1 INT, INOUT val2 INT)
RETURNS RECORD AS $$
DECLARE
        temp_val INTEGER;
BEGIN
        temp_val := val1;
        val1 := val2;
        val2 := temp_val;
END;
$$ LANGUAGE plpgsql;
```

- IF THEN, ELSIF THEN, ELSE, END IF:

```
CREATE OR REPLACE FUNCTION swap (INOUT val1 INT, INOUT val2 INT)
RETURNS RECORD AS $$
DECLARE
        temp_val INTEGER;
BEGIN
        IF val1 > val2 THEN
                temp_val := val1;
                val1 := val2;
                val2 := temp_val;
        END IF;
END;
$$ LANGUAGE plpgsql;
```

- LOOP, END LOOP:

```
CREATE OR REPLACE FUNCTION sum_to_x (IN x INT, OUT s INT)
RETURNS INTEGER AS $$
DECLARE
        temp_val INTEGER;
BEGIN
        s := 0;
        temp_val := 1;
        LOOP
                EXIT WHEN temp_val > x;
                s := s + temp_val;
                temp_val := temp_val + 1;
        END LOOP;
END;
$$ LANGUAGE plpgsql;
```

- CURSOR:

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

- FOR LOOP

```
CREATE OR REPLACE FUNCTION getAllFoo() RETURNS SETOF foo AS
$BODY$
DECLARE
  r foo%rowtype;
BEGIN
  FOR r IN SELECT * FROM foo
  WHERE fooid > 0
  LOOP
    -- can do some processing here
    RETURN NEXT r; -- return current row of SELECT
  END LOOP;
  RETURN;
END
$BODY$
LANGUAGE plpgsql;
```

- More FOR LOOP:
    1. FOR varname IN 1..10 LOOP [PLPGSQL] END LOOP;
    2. FOREACH varname IN ARRAY arrname LOOP [PLPGSQL] END LOOP;
    3. FOR record in WITH tablename AS ([QUERY]) SELECT [QUERY] LOOP [PLPGSQL] END LOOP;
- Cursor movement: FETCH PRIOR FROM curs INTO r, FETCH FIRST FROM curs INTO r, FETCH LAST FROM curs INTO r, FETCH ABSOLUTE n FROM CUR INTO r

-Trigger:

```
CREATE TRIGGER scores_log_trigger
AFTER INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION score_log_func();
```

```
CREATE OR REPLACE FUNCTION scores_log_func() RETURNS TRIGGER
AS $$
BEGIN
        INSERT INTO Scores_Log(Name, EntryDate)
                VALUES (NEW.Name, CURRENT_DATE);
        RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

- Assess operations:

```
CREATE OR REPLACE FUNCTION scores_log2_func() RETURNS TRIGGER AS $$
BEGIN
    IF (TG_OP = 'INSERT') THEN
        INSERT INTO Scores_Log2 SELECT NEW.Name, 'Insert', CURRENT_DATE;
        RETURN NEW;
    ELSEIF (TG_OP = 'DELETE') THEN
        INSERT INTO Scores_Log2 SELECT OLD.Name, 'Delete', CURRENT_DATE;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO Scores_Log2 SELECT NEW.Name, 'Update', CURRENT_DATE;
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

- Return values for trigger:

	non-null	null
BEFORE INSERT	Insert returned	no insertion
BEFORE UPDATE	update returned	no update
BEFORE DELETE	delete proceeds	no delete
AFTER INSERT	does not matter	does not matter
AFTER DELETE	does not matter	does not matter
AFTER UPDATE	does not matter	does not matter

- INSTEAD OF trigger: only on views
- Statement level trigger executes once, ignores return value, RAISE EXCEPTION / NOTICE
- Trigger condition: no SELECT, no OLD for INSERT, no NEW for DELETE, no WHEN for INSTEAD OF

```
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW
WHEN (NEW.Name = 'Elise')
EXECUTE FUNCTION for_Elise_func();
```

- Deferred Trigger: must be AFTER, FOR EACH ROW

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

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

```
BEGIN TRANSACTION;
SET CONSTRAINTS bal_check_trigger DEFERRED;
UPDATE Account SET Bal = Bal – 100 WHERE AID = 1;
UPDATE Account SET Bal = Bal + 100 WHERE AID = 2;
COMMIT;
```

- Order of activation: BEFORE statement, BEFORE row, AFTER statement, AFTER row
- FD: $A_1..A_n$-> $B_1B_2..B_m$ if one value of $A_1..A_n$ maps to one value of $B_1...B_m$
- Reflexivity: ABCD -> A
- Augmentation: if A -> B, AC -> BC
- Transitivity: if A -> B and B -> C, A -> C
- Decomposition: if A -> BC, A -> B and A -> C
- Union: if A -> B and A -> C, A -> BC
- Find key of table: use closures to find minimal subset that decides all attributes
- Prime attribute: attribute in a key
- Decomposed FD: RHS only one attribute
- Non-trivial: RHS not in LHS
- BCNF: all non-trivial and decomposed FD has superkey as LHS
- Algorithm for checking BCNF:
    1. Find closure for each subset
    2. Derive the keys of table
    3. Derive all non-trivial, decomposed FD
    4. Check all non-trivial, decomposed FD satisfy BCNF
- Simplified algorithm for BCNF:
    1. Find closure for each attribute subset
    2. Check if exist a closure with “more but not all”
    3. If exists, not in BCNF
- BCNF decomposition:
    1. Find a subset X that violates “more but not all”
    2. R1 = all attributes in $\{X\}^+$, R2 = X $\cup$ remaining attributes
    3. Find closures for attribute subsets of R
    4. Remove irrelevant attributes depending on R1 or R2
    5. Check if R1 or R2 in BCNF
- Lossless join: common attributes of R1 and R2 is a superkey of R1 or R2
- BCNF pros: no update/delete/insert anomalies, small redundancy, original table can be reconstructed (guaranteed lossless)
- BCNF cons: dependencies not preserved
- Dependency preserving: Calculate all closure of subsets of R1 and R2 with FDs of R, remove attributes not in R1 or R2, see if all dependency preserved
- 3NF: all non-trivial and decomposed FD has superkey as LHS or RHS is prime attribute (has anomalies in rare cases, but preserves FD)
- Algorithm for checking 3NF:
    1. Find closure for each subset
    2. Derive the keys of table
    3. For each closure, check if RHS is not all and RHS has attribute that is not prime attribute
    4. If exist such closure, not in 3NF
- 3NF decomposition:
    1. Find minimal basis for FD
    2. Combine FDs in minimal basis with same LHS
    3. Create table for each FD remaining
    4. If no table has a key, create a table with any key
- Minimal basis condition:
    1. Every FD can be derived from original FD, vice versa
    2. Every FD is non-trivial and decomposed
    3. No FD in minimal basis is redundant
    4. No attribute in LHS is redundant
- Algorithm for minimal basis:
    1. Decompose FD such that RHS only one attribute
    2. Remove redundant attributes at LHS (IS A redundant? If remove A, AB -> C becomes B -> C, check B closure contains C
    3. Remove redundant FDs (For each FD, find closure without that FD, if result same then the FD is redundant)
- Null values operation

x	y	x AND y	x OR y	NOT x
FALSE	FALSE	FALSE	FALSE	
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	TRUE	FALSE	TRUE	
UNKNOWN	FALSE	FALSE	UNKNOWN	
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	TRUE	
TRUE	FALSE	FALSE	TRUE	
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	TRUE	TRUE	TRUE	

- Data types

boolean	false/true (null represents unknown)
integer	signed four-byte integer
float8	double-precision floating point number (8 bytes)
numeric	arbitrary precision floating point number number
numeric(p,s)	maximum total of p digits with maximum of s digits in fractional part
char(n)	fixed-length string consisting of n characters
varchar(n)	variable-length string up to n characters
text	variable-length character string
date	calendar date (year, month, day)
timestamp	date and time

- Transactions: start with begin, end with commit/rollback

```
begin;
update Accounts
set balance = balance + 1000
where accountId = 2;

update Accounts
set balance = balance - 1000
where accountId = 1;
commit;
```