Relational Model

- **Domain**: Set of atomic values
- **Relation**: Set of tuples
- **Superkey**: Subset of attributes that uniquely identifies one tuple
- **Key**: Minimal superkey
- **Candidate Keys**: Set of all <u>keys</u>
- **Primary Key**: Chosen candidate key; cannot be NULL
- **Foreign Key**: Refers to a primary key in another relation; must apear as a primary key of another relation or have NULL for an attribute

Relational Algebra

$\sigma_{[\sigma]}(R)$: Select
- Select rows that satisfy condition *c*
- Principle of Acceptance

$\pi_{[l]}(R)$: Project
- Select columns listed in *l*

$\rho_{[B_1 \leftarrow A_1, B_2 \leftarrow A_2]}(R)$: Rename
- Order does not matter
- No two attributes can be renamed to the same name
- No attributes can be renamed more than once in a single operation

Set Operations
- Relations must be union compatible:
  1. Have same number of attributes
  2. Attributes have same or compatible domains

Cross Product
- Set of attributes must be <u>disjoint</u>
- |R × S| = |R| × |S|
- If either R or S is empty, result is an empty relation

Join
- **Inner Joins**
  - $\bowtie_\theta$: choose if tuples satisfy the condition
  - $\bowtie_=$: choose if tuples satisfy the condition; condition only uses =
  - $\bowtie$: choose if all common attributes between two tuples are equal. Becomes cross product if no common attributes
  - Common attributes (i.e. columns) can can appear twice in output relation, unless natural join is used (then only appear once)
- **Outer Joins**
  - ⟕: $R \bowtie$ **S**, plus dangling tuples in R. Dangling tuples have values NULL for attributes from S
  - ⟖: $R \bowtie$ **S**, plus dangling tuples in S. Dangling tuples have values NULL for attributes from R
  - ⟗: $R \bowtie$ **S**, plus dangling tuples in R and S

Equivalence
- Strong equivalence: Both queries produce error or both queries always produce the same results
- Weak equivalence: Both queries always produce the same results if neither queries produce an error

Joins	Select and Project
$R \times S \not\equiv S \times R$ $R \bowtie S \not\equiv S \bowtie R$ $(R \times S) \times T \equiv R \times (S \times T)$ $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$ $(R \bowtie_\theta S) \bowtie_\theta T \not\equiv R \bowtie_\theta (S \bowtie_\theta T)$ (weak)	$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$ $\sigma_{c_1}(\sigma_{c_1}(R)) \equiv \sigma_{c_1 \wedge c_2}(R)$ $\pi_{\ell_1}(\pi_{\ell_2}(R)) \not\equiv \pi_{\ell_1}$(weak, unless $\ell_1 \subseteq \ell_2$) $\pi_\ell(\sigma_\theta(R)) \not\equiv \sigma_\theta(\pi_\ell(R))$ (weak) $\sigma_\theta(R \times S) \not\equiv \sigma_\theta(R) \times S$ (weak)
Set operations	
$R \cup S \equiv S \cup R$ $R \cap S \equiv S \cap R$ $(R \cup S) \cup T \equiv R \cup (S \cup T)$ $(R \cap S) \cap T \equiv R \cap (S \cap T)$ $R - S \not\equiv S - R$ $\sigma_c(R - S) \equiv \sigma_c(R) - S \not\equiv \sigma_c(R) - \sigma_c(S)$ (unless R and S have some common attribute names used in c) $\pi_\ell(R \cup S) \equiv \pi_\ell(R) \cup \pi_\ell(S)$	

SQL DDL

CREATE

```
CREATE TABLE <table_name> (
   <attr1> <type> [<col_constraint>],
   ...
   [<table_constraints]
);
```

INSERT

```
INSERT INTO <table_name> [(attr1, attr2,...)]
VALUES (val1, val2,...), ...;
```

- Either all are inserted, or none inserted

DELETE

```
DELETE FROM <table_name>
[WHERE <conditions>];
```

- Delete all tuples that match given condition; if no condition, all tuples are deleted
- Principle of ACCEPTANCE

Integrity Contraints
- Principle of **REJECTION**

Common constraints		
	Column	Table
Primary Key	PRIMARY KEY	PRIMARY KEY (attr1, attr2)
Unique	UNIQUE	UNIQUE (attr1, attr2)
Foreign Key	REFERENCES R(attr1, attr2)	FOREIGN KEY (attr1, attr2) REFERENCES R(attr3, attr4)
CHECK	CHECK(attr <op>)	CHECK(attr <op>)

- Foreign Keys `ON UPDATE/ON DELETE` specifies behaviour of referencing table when data in referenced table is deleted/updated
  - `NO ACTION` : reject update/delete if violates constraint
  - `RESTRICT` : NO ACTION, but not deferrable
  - `CASCADE` : propagate delete/update to referencing tuples
  - `SET DEFAULT` : Set FK in referencing tuples to default values; default values must be PK in the referenced table
  - `SET NULL` : Set FK in referencing tuples to NULL; affected columns must no have NOT NULL constraint
- UNIQUE constraints check individual attributes using <>; 2 tuples are unique if either one contains NULL

ALTER
- Useful for circular references ($FK_R \rightarrow PK_S$, $FK_S \rightarrow PK_R$)

```
ALTER TABLE <table_name>
[ALTER/ADD/DROP] [COLUMN/CONSTRAINT] <name>
<changes>
```

DROP TABLE

```
DROP TABLE [IF EXISTS]
<table_name> [, <table_name2>, ...]
[CASCADE]
```

DEFFERABLE CONSTRAINTS
- `NOT DEFERRABLE` : (default). Constraints checked at end of SQL statement and aborts if violated
- `DEFERRABLE INITIALLY DEFERRED` : Constraints checked on COMMIT, can be temporarily violated in transaction
- `DEFERRABLE INITIALLY DEFERRED` : Constraints initially not deferrable, but can be set to deferrable later with `SET CONTRAINTS <name> DEFERRED`
- Transaction: `BEGIN; ... COMMIT;`

SQL DQL

```
SELECT [DISTINCT] <attrs>
FROM <relations>
WHERE <conditions>
```

- Aliasing: `column AS alias`
- Operations
  - Maths: +-*/, |/, ^ %, etc.
  - String: || (concatenate), LOWER(s), UPPER(s), etc.
  - Date Time: +, NOW(), etc.
- Principle of ACCEPTANCE
- = and <> can be safely used if you do not want NULL values, else use `IS NULL`
- Regex `LIKE <regex>`
  - _: Any <u>single</u> character
  - %: Any sequence of 0 or more characters

UNION/INTERSECT/EXCEPT
- Must be union compatible
- `UNION ALL` : # dups = #dup in R + #dup in S
- `INTERSECT ALL` : # dups = min{#dup in R, #dup in S}
- `EXCEPT ALL` : # dups = #dup in R - #dup in S

JOIN
- Cross product `FROM R1 [AS][Alias], R2, R3,…`
  - Set of attributes need not be disjoint
- JOIN `R JOIN S ON <cond>`
- NATURAL JOIN `R NATURAL JOIN S`
  - Becomes cross product if no common attributes
- OUTER JOIN `R LEFT/RIGHT/FULL [OUTER] JOIN S ON <cond>`

ORDER
- Default: ASC
- `ORDER BY <attr1> DESC/ASC [, <attr2> DESC/ASC]`
- Stable sort from rightmost to left

LIMIT & OFFSET

```
LIMIT <j>
```

CASE

```
SELECT (
   CASE
      WHEN <cond> THEN <result>
      ...
      ELSE <result>
   END
) FROM ...
```

```
SELECT (
   CASE <expr>
      WHEN <value> THEN <result>
      ...
      ELSE <result>
   END
) FROM ...
```

; returns NULL if no conditions matched

COASLESCE

```
SELECT (
   COALESCE(<value1>, <value2>, ...)
) FROM ...
```

n-NULL value, or NULL if all values are NULL

NULLIF

```
SELECT (
   NULLIF(<value1>, <value2>)
) FROM ...
```

<value1> = <value2>, otherwise returns <value1>

SQL Subqueries

Scalar Subqueries

```
SELECT (
   SELECT <attr> FROM <table_name> WHERE <cond>
) FROM ...
```

- Query returning at most one row and one column (i.e. single value) or NULL

Operation	Syntax	Subquery Behaviour
IN	`WHERE <expr> IN <subquery | tuple>`	Returns 1 column; Empty → vacuously false
NOT IN	`WHERE <expr> IN <subquery | tuple>`	Returns 1 column; Empty → vacuously true
ANY	`WHERE <expr> <op> ANY <subquery>`	Returns 1 column; Empty → vacuously false
ALL	`WHERE <expr> <op> ALL <subquery>`	Returns 1 column; Empty → vacuously true
EXISTS	`WHERE EXISTS <subquery>`	
NOT EXISTS	`WHERE NOT EXISTS <subquery>`	

SQL Aggregates

- Queries with aggregate return a single row

Operation	Behaviour	Empty/All NULL table
MIN(attr)	min. non-NULL value in attr	NULL
MAX(attr)	max. non-NULL value in attr	NULL
AVG(attr)	avg of all. non-NULL values in attr	NULL
SUM(attr)	sum of all. non-NULL values in attr	NULL
COUNT(attr)	# of non-NULL values in attr	0
COUNT(*)	# of rows in table	# of rows in table

- Use DISTINCT (e.g. SUM(DISTINCT attr)) to ignore duplicates

GROUP BY & HAVING

```
GROUP BY attr1, attr2
HAVING <condition>
```

- Treat tuples with same values for the listed attributes "as one group"
- Aggregate functions now apply to each group (vs entire table)
- HAVING is a WHERE clause that applies to an entire group (vs individual rows)
- Column X can appear in SELECT / HAVING if:
  - X appears in GROUP BY clause, or
  - X appears as an input to an aggregate function, or
  - PK of table X belongs to appears in GROUP BY clause (UNIQUE constraint is insufficient)

SQL MISC.

Common Table Expressions

```
WITH
   <table_name> AS <query>,
   …
<main_query>
```

- CTEs can reference any CTEs declared before it

Recursive Queries

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

- Q_1 is non-recursive; Only Q_2 and Q_0 can reference the CTE

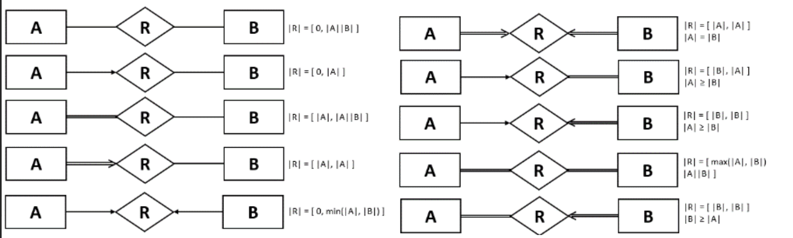
Universal Quantification

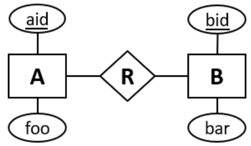
1. Split query into two sets: R and S
2. Figure out cardinality:
   - If R⊇S, then |R ∪ S| = |R|
   - If R⊆S, then |R ∩ S| = |R|
3. Craft 2 scalar subqueries queries
   - Query 1: Count # entries in R ∪ S or R ∩ S
   - Query 2: Count # entries in R
   - Check count is the same in both queries

SQL Order of Execution

ORDER	CLAUSE	FUNCTION
1	from	Choose and join tables to get base data.
2	where	Filters the base data.
3	group by	Aggregates the base data.
4	having	Filters the aggregated data.
5	select	Returns the final data.
6	order by	Sorts the final data.
7	limit	Limits the returned data to a row count.

ER Model

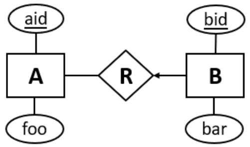





```
CREATE TABLE R (
   aid INT REFERENCES A,
   bid INT REFERENCES B,
   PRIMARY KEY(aid, bid)
);
```

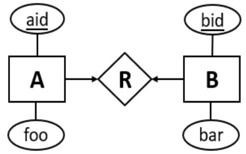
```
CREATE TABLE A (
   aid INT PRIMARY KEY,
   foo INT
);

CREATE TABLE B (
   bid INT PRIMARY KEY,
   bar INT
);
```



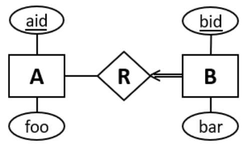

```
CREATE TABLE R (
   bid INT PRIMARY KEY,
   aid INT REFERENCES A,
   bar INT
);
```

```
CREATE TABLE A (
   aid INT PRIMARY KEY,
   foo INT
);
```



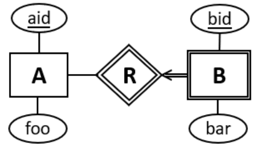

```
CREATE TABLE R (
   bid INT PRIMARY KEY,
   aid INT UNIQUE REFERENCES A,
   bar INT
);
```

```
CREATE TABLE A (
   aid INT PRIMARY KEY,
   foo INT
);
```



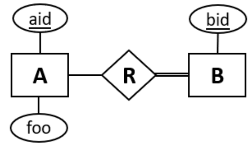

```
CREATE TABLE R (
   bid INT PRIMARY KEY,
   aid INT NOT NULL REFERENCES A,
   bar INT
);
```

```
CREATE TABLE A (
   aid INT PRIMARY KEY,
   foo INT
);
```



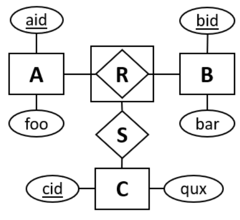

```
CREATE TABLE A (
   aid INT PRIMARY KEY,
   foo INT
);
```

```
CREATE TABLE B (
   bid INT,
   bar INT,
   aid INT REFERENCES A
      ON DELETE CASCADE
      ON UPDATE CASCADE,
   PRIMARY KEY (aid, bid)
);
```




```
CREATE TABLE R (
   bid INT,
   aid INT UNIQUE REFERENCES A,
   PRIMARY KEY (bid, aid)
);
```

```
CREATE TABLE A (
   aid INT PRIMARY KEY,
   foo INT
);
```

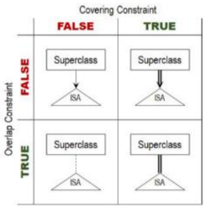



```
CREATE TABLE R (
   aid INT REFERENCES A,
   bid INT REFERENCES B,
   PRIMARY KEY (aid, bid)
);
```

```
CREATE TABLE S (
   aid INT,
   bid INT,
   cid INT REFERENCES C
   PRIMARY KEY (aid, bid, cid),
   FOREIGN KEY (aid bid)
      REFERENCES R(aid, bid)
);
```

ISA Hierarchies

- Inherited primary key of child references the primary key of the direct parent
- Specify ON DELETE/ON UPDATE on child foreign keys



PL/pgSQL

Functions

```
CREATE OR REPLACE
FUNCTION <fn_name> ([IN/OUT/INOUT] <param> <type> …)
RETURNS <return_type> AS $$
DECLARE   --plpgsql
  ...
BEGIN     --plpgsql
  ...
END;
$$ LANGUAGE <sql/plpgsql>;
```

- IN/OUT/INOUT :
  - IN: (default) Parameter is an input. A constant; cannot be reassigned
  - OUT: Parameter is return value. An uninitialised variable; must be assigned a value later
  - INOUT: Parameter is both an input and return value. An initialised variable; should be but not value later
- Return types
  - <table_name>: Returns one existing tuple from the table
  - SETOF <table_name>: Returns one or more existing tuples from the table
  - RECORD: Returns one new tuple from the table containing the OUT/INOUT attributes specified in the parameters list
  - SETOF RECORD: Returns one or more new tuples containing the OUT/INOUT attributes specified in the parameters list
  - TABLE(<attr> <type>, …): Returns a new table with the specified schema. Parameters list should not contain OUT/INOUT
    - plpgsql: Function should call one or more RETURN NEXT to populate the table

Procedures

```
CREATE OR REPLACE
PROCEDURE <procedure_name> (<param> <type> …)
AS $$
...
$$ LANGUAGE <sql/plpgsql>;

CALL procedure_name(params…)
```

IF ELSE & Control Flow

```
IF <cond> THEN
  ...
ELSE
  ...
END IF;
```

```
LOOP
  EXIT WHEN <cond>
  ...
END LOOP
```

Cursor

```
DECLARE
  curs CURSOR FOR <table_name>;
  r RECORD
  ...
BEGIN
  OPEN curs;
  LOOP
    FETCH curs INTO r; --get current tuple
    EXIT WHEN NOT FOUND; --exit when end of table
    ...
    RETURN NEXT; --insert tuple into table
  END LOOP;
  CLOSE curs;
  ...
```

Triggers

```
CREATE [CONSTRAINT] TRIGGER <trigger_name>
AFTER/BEFORE INSERT/UPDATE/DELETE OR [...] ON <table>
[DEFERRABLE INITIALLY DEFERRED/IMMEDIATE]
FOR EACH ROW/STATEMENT
[WHEN <cond>]
EXECUTE FUNCTION <fn_name>();

CREATE OR REPLACE FUNCTION <fn_name>
RETURNS TRIGGER ...
```

- Special variables
  - NEW: INSERT/UPDATE: the new tuple; DELETE: NULL
  - OLD: DELETE/UPDATE: the old tuple; INSERT: NULL
  - TG_OP: INSERT/UPDATE/DELETE
  - TG_TABLE_NAME: Table associated with the trigger
- Only AFTER and FOR EACH ROW triggers can be deferred

Return Types

- **BEFORE**
  - INSERT: Null → nothing inserted; Non-null *t* → insert *t*
  - UPDATE: Null → not updated; Non-null *t* → updated to *t*
  - DELETE: Null → not deleted; Non-null *t* → delete (not nec. *t*)
- **AFTER**
  - Does not matter; just return NULL for convenience
- **INSTEAD OF**
  - NULL → ugnore rest of the operation on current row; Non-null → proceed as normal
  - INSTEAD OF is only defined on VIEWS and ROW-LEVEL

RAISE

- RAISE NOTICE '…': Prints warning message, but does not prevent the operation
- RAISE EXCEPTION '…': Prints warning message and prevents the operation

WHEN

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

Order of Execution

- BEFORE statement > BEFORE row > INSTEAD OF > AFTER row > AFTER statement
- Within each category triggers are activated in alphabetical order of their names (A → Z)
- If a BEFORE row-level trigger returns NULL, then subsequent triggers on the same row are omitted

Functional Dependencies

Armstrong's Axioms

- **Reflexivity**: AB → A
- **Augmentation**: If A → B, then AC → BC
- **Transitivity**: If A → B and B → C, then A → C
- Decomposition: If A → BC, then A → B and A → C
- Union: If A → B and A → C, then A → BC

Closure

1. Initialise closure of X to contain X i.e. $X^+ = \{ X \}$
2. If there is a FD X → Y s.t. X is in the closure, put Y into closure
3. Repeat 2 until no more attributes can be added
4. If Y is in X^+, then X → Y holds

Superkeys and Keys

1. Find all possible subset of attributes of R
2. Find the closure of each subset
   - Start with the smallest sets first
   - If an attribute A does not appear in RHS of any FD, then A must be in every key
3. Find all superkeys. $\{X_1, X_2, \ldots\}$ is a superkey if $\{X_1, X_2, \ldots\}^+$ contains all attributes in R
4. Find all keys. A key is a superkey that cannot be minimised further

BCNF

- Condition: Every non-trivial and decomposed FD has a superkey as its LHS
- Decomposition guarantees lossless join, but may not preserve all FDs

Checking

1. For each attribute set in R, find its closure
2. If a closure satisfies "more but not all" condition, then R is not in BCNF
   - "more but not all": Closure containing more attributes than the attribute set but not all attributes in the table

BCNF Decomposition

1. Choose any closure that does not satisfy the "more but not all" condition
2. Based on the selected closure, split the table R into 2:
   - Table R_1: Contains all attributes in the selected closure
   - Table R_2: Contains all attributes in R, but not in the selected closure + attribute set of closure
3. Check if both tables are in BCNF
   - Enumerate all attribute subsets in R, then chose only relevant ones for each table (relevant: attribute sets containing only attributes in the new table)
   - Derive the closures of the selected attribute sets. If the closure contains an attribute not belonging to the new table, ignore that attribute
   - A table with only 2 attributes will always be in BCNF

Lossless Join

1. Derive $S = R_1 \cap R_2$
2. Using the original FDs, check if S is a superkey of either R_1 or R_2 (i.e. if S^+ contains all attributes in that table)
3. Repeat for each level

3NF

- Condition: Every <u>non-trivial</u> and <u>decomposed</u> FD has a superkey as its LHS OR a prime attribute on its RHS
- Preserves all FDs

Checking

1. Find all keys of R
2. For each non-trivial and decomposed FD, check if LHS is superkey or RHS is a prime attribute

3NF Decomposition

1. Find minimal basis of the set of FDs.
   Minimal basis:
   (i). every FD in MB can be derived from original set of FDs, vice versa;
   (ii). every FD in MB must be non-trivial and decomposed;
   (iii). no redundant FDs in MB
   (iv). none of the attributes on the LHS of each FD in MB is redundant
   1.1. Check if set of FDs is already a minimal basis. If yes, skip to 2.
   1.2. Decompose FDs (satisfy (ii))
   1.3. Remove redundant attributes on LHS of FDs (satisfy (iv))
      1.3.1. Given AB → C, is A → C ok? If B → C ok?
      1.3.2. A → C is ok if $\{A\}^+$ contains C. Same for B
   1.4. Remove redundant FDs (satisfy (iii))
      1.4.1. Remove one FD from MB e.g. A → B. Can A → B be derived from the remaining FDs?
      1.4.2. Can, if $\{A\}^+$ contains B, using the remaining FDs
2. In minimal basis, combine FDs whos LHS are the same (rule of union)
3. Convert each FD in minimal basis into a table
4. If no tables contain a key of R, create a table using a key (any arbitrary one) as the attributes
5. Remove redundant tables (R_2 is redundant if $attrSet_{R2} \subseteq attrSet_{R1}$)