CS2102 Cheatsheet

Selection – Select all tuples from relation that satisfy condition **iff** evaluates to **true**

$$\sigma_{manager='Judy'}(\text{Projects})$$

Selection Conditions

attribute op constant	$\sigma_{start\_year=2020}(\text{Projects})$
$attribute_1$ op $attribute_2$	$\sigma_{start\_year=end\_year}(\text{Projects})$
$expr_1 \wedge expr_2$	$\sigma_{start\_year=2020 \wedge manager='Judy'}(\text{Projects})$
$expr_1 \vee expr_2$	$\sigma_{start\_year=2020 \vee manager='Judy'}(\text{Projects})$
¬ expr	$\sigma_{\neg(start\_year=2020)}(\text{Projects})$

→ Comparison operation with null = unknown
→ Arithmetic operation with null = null

Projection

$$\pi_{pname,cname}(\text{Teams})$$

→Order matters, **duplicates removed**

Renaming

$$\rho_{name \leftarrow ename, title \leftarrow pname}(\text{Teams})$$

$B_i \leftarrow A_i, \ldots, B_k \leftarrow A_k$ final ← original

Set Operators

Union Intersection Set Difference

$R \cup S$ $R \cap S$ $R - S$

→ R and S must be union compatible, i.e. they have the same number of column attributes and must be compatible type (no need same name)

Cross Product X

→A X B returns {I,J,K,Y,Z} for columns I, J, K belonging to A and Y, Z belonging to B

Inner Joins $R \bowtie_\theta S = \sigma_\theta(R \times S)$

→ Join based on condition θ specified

$\text{Projects} \bowtie_{manager=name} \text{Managers}$

→ Manager belongs to Projects, name belongs to Managers. Output has **both** manager & name

→ Above is also an equi join

→ Inner Joins allow for arbitrary comparison operators (e.g., =, <>, <, ⩽, ⩾, >)

Equi Joins

Special case of inner join – only for '=' operator

Natural Join

- Over **ALL** attributes R and S have in common, **output contains common attributes only once**

$$R \bowtie S = \pi_\ell(R \bowtie_c \rho_{b_i \leftarrow a_i, \ldots, b_k \leftarrow a_k}(S))$$

Outer Join

- Includes dangling tuples + inner join

Perform inner join $M = R \bowtie_\theta S$

To M, add dangling tuples to result of

R	in case of a **left outer join** $⟕_\theta$
S	in case of a **right outer join** $⟖_\theta$
R and S	in case of a **full outer join** $⟗_\theta$

$\text{Employees} ⟕_{name=ename} (\pi_{ename}(\text{Projects}))$

name	age	role	ename
...	...	...	...
Jack	40	dev	Jack
Bill	45	dev	Bill
Marie	36	hr	*null*
Bernie	19	*null*	*null*

inner join result

dangling tuples with *null* padding

→ Basically match all employees based on condition to projects table, and include those who ended up not matching to anything

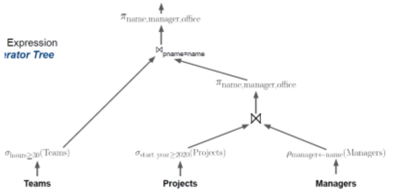
→ Full outer joins have dangling tuples padded with null for both tables R and S.

Natural Outer Join

Outer join that outputs common attributes of R and S once, similar to Natural Join

Natural left outer join	$R ⟕ S$
Natural right outer join	$R ⟖ S$
Natural full outer join	$R ⟗ S$

Relational Operator Trees



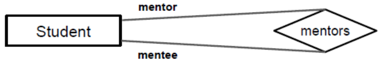
ER MODEL - Attributes

Specific info describing an entity – represented by oval in ER diagrams. 4 types:

Key attribute (uniquely identifies entity), **Composite attribute** (comprised of multiple attributes, oval branched to other ovals), **Multivalued attribute**(consists of more than one value, double-lined oval),

Derived attribute(derived from other attributes, dashed oval)

Relationship / Relationship Sets (Roles)



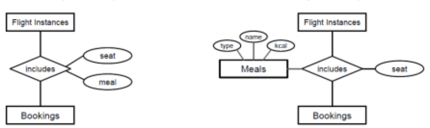
→ Descriptor of entity set's participation in a rs, explicit role label only common in ambiguity or if same entity set participates in relation > 1 time

→ Rectangle = object, Diamond = relationship

n-ary Relationship Set

n = degree of relationship set, involves n entity roles (> 3 are very rare)

n = 2 → binary relationship set n = 3 → ternary relationship set

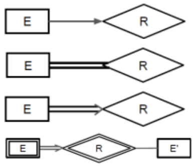


Cardinality Constraints

- Many to Many (m:n), Many to one (m:1), One to one (1:1)

Participation Constraints

- **Partial** participation constraint (default)
  - Participation of an entity in a relationship is not mandatory
  - Example: A user made 0 or more bookings

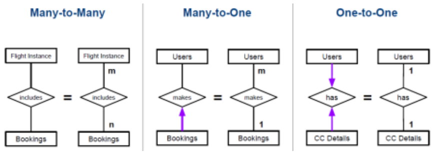


Each instance of E participates in at most one instance of R.

Each instance of E participates in at least one instance of R.

Each instance of E participates in exactly one instance of R.

E is a weak entity set with identifying owner E' and identifying relationship set R.



Weak Entity Sets


```
CREATE TABLE Flights (
    fnr         VARCHAR(10),
    from        VARCHAR(10),
    to          VARCHAR(10),
    PRIMARY KEY (fnr)
);
```

```
CREATE TABLE FlightInstances (
    fnr         VARCHAR(10),
    date        DATE,
    aircraft    VARCHAR(10),
    PRIMARY KEY (fnr, date),
    FOREIGN KEY (fnr) REFERENCES Flights (fnr)
        ON DELETE CASCADE
);
```

→ Do not have own key, can only be uniquely identified by considering pk of owner entity

→ Weak entity's existence depends on existence of owner entity

→ Many to one relationship from weak entity set to owner entity set

→ **Partial Key** – Set of attributes of weak entity that uniquely identifies a weak entity for a given owner entity

Modelling Cardinality Constraints → Schema

M:N → Create table with pk as all the attributes of the entity sets participating, have fks pointing back to the entity sets

1:M → Create table with the entity participating once as pk, have fk pointing to both entities

1:M → Combine relationship set and entity set participating once, point FK to parent

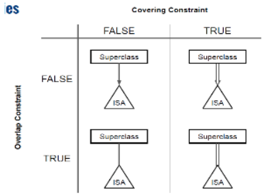
1:1 → Put just them all in the same table, or if you want to split them, have FKs pointing to each other

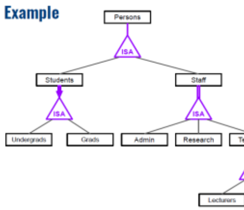
ISA Hierarchies - Overlap Constraint

→ Can a superclass entity belong to multiple subclasses (can be both vs either student/staff)

ISA Hierarchies - Covering Constraint

→ Does superclass entity have to belong to a subclass (Every student must be a grad / undergrad vs not every person is a student)





- A person can be both a student and staff
- A person can be neither a student or staff
- Each student is either a undergraduate or a graduate student
- Each staff is either admin, research, teaching or any combination of these three roles
- A teaching staff cannot be a lecturer and TA
- A teaching staff can be neither lecturer or TA

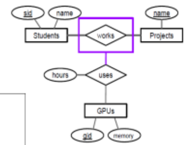
→ Use ON DELETE CASCADES / FKs etc to model

Aggregation

→ Treat relationships as higher level entities

Schema definition of "uses"

- Primary key of aggregation relationship → (sid, pname)
- Primary key of associated entity set "GPUs" → gid
- Descriptive attributes of "uses" → hours




```
CREATE TABLE Uses (
    gid     INTEGER,
    sid     CHAR(20),
    pname   VARCHAR(50),
    hours   NUMERIC,
    PRIMARY KEY (gid, sid, pname),
    FOREIGN KEY (gid) REFERENCES GPUs (gid),
    FOREIGN KEY (sid, pname) REFERENCES works (sid, pname)
);
```

Functional Dependencies

Armstrong Axioms

1. Axiom of Reflexivity (Set of attributes → subset of attributes)

- {NRIC, Name} → {NRIC}
- {StudentID, Name, Age} → {Name, Age}

2. Axiom of Augmentation (If A → B then AC → BC)

- **Example:** if {NRIC} → {Name} then
  - {NRIC, Age} → {Name, Age}
  - {NRIC, Salary, Weight} → {Name, Salary, Weight}
  - {NRIC, Address, Postal} → {Name, Address, Postal}

3. Axiom of Transitivity

- If {A} → {B} and {B} → {C}
- Then {A} → {C}

- **Example:**
  - if {NRIC} → {Address}
  - and {Address} → {Postal}
  - then {NRIC} → {Postal}

Extended Armstrong's Axioms

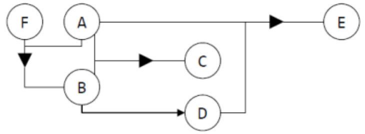
4. Rule of Decomposition

- If {A} → {BC}
- Then {A} → {B} and {A} → {C}

5. Rule of Union

- If {A} → {B} and {A} → {C}
- Then {A} → {BC}

→ Can use circuit diagram to compute closures



Boyce-Codd Normal Form (BCNF)

- Non-trivial means when your FD/closure doesn't just point to itself (e.g. A->A)

- Decomposed means the RHS only has 1 attribute (e.g. AB->C, A->D)

- BCNF definition: Every non-trivial and decomposed FD has a superkey as its LHS

BCNF Decomposition

- Compute closure for all possible combinations of attributes, find one closure where RHS is not trivial and not all attributes

- If schema is (ABCD) and FD violation is A->ABC, split tables into ABC and AD

→ Take the 3 violating attributes put in one table, the other table has the LHS attribute + the attribute(s) that didn't appear

→ Keep splitting until every single table has either 2 attributes or is in BCNF

→ If the new table you split don't have relevant FDs, just use it as if its there and remove it after

iii. Check if R1(A, B) and R2(A, C, D, E) are in BCNF

YES! ???

Do we even know what are the FDs that holds on R2?

{A} → {B}? But no B {BC} → {D}? Also no B

a. Compute closure of each subset of attributes

b. Remove attributes not in the current table (projection)

- Now we can find the violation

$\{A\}^+ = \{AB\}$	$\{C\}^+ = \{C\}$	$\{D\}^+ = \{D\}$	$\{E\}^+ = \{E\}$
$\{AC\}^+ = \{ABCD\}$	$\{AD\}^+ = \{ABD\}$	$\{AE\}^+ = \{AE\}$	
$\{CD\}^+ = \{CD\}$	$\{CE\}^+ = \{CE\}$	$\{DE\}^+ = \{DE\}$	
$\{ACD\}^+ = \{ABCD\}$	$\{ACE\}^+ = \{ABCDE\}$	$\{ADE\}^+ = \{ABDE\}$	$\{CDE\}^+ = \{CDE\}$

→ From the image, we split ACDE into ACD and ACE

- **Properties of BCNF**
  - Good properties
    - No update or deletion or insertion anomalies
    - Small redundancies (very hard to completely remove redundancies)
    - The original table can always be reconstructed from the decomposed tables
  - Bad properties
    - Dependencies may not be preserved (non dependency-preserving decomposition)
      - *Topic for next week lecture*

3NF

→ BCNF does not preserve dependencies

→ A table is in 3NF is every non-trivial and decomposed FD either has LHS as superkey or RHS is a prime attribute

Find the keys of R:

1. Look at the RHS, if the attribute not there then it will definitely be part of key

2. Try use FDs, start from the smallest ones, get the closures, get minimal ones that give us closure with all attributes (if we can find 2 attributes to give us the key then 2 is the key size, anyth else is superkey)

3. All attributes in key are prime attributes

(a) Find all the keys of R. You do not have to show your step. Write the answer in the following format:

{A,B,C,D}, {B,C,D,E}, {C,D,E,A}

Solution: All the keys are

{A,C,E}, {A,C,D}

Minimal Basis:

1. See if we can form all the FDs that are defined in the question based on the few that we picked

2. Try finding as little as possible

(c) Find a minimal basis of the set of functional dependencies. You do not have to show your step. Write the answer in the following format:

{A,B,C} -> {C,D,E}, {B,C,D} -> {D,E,A}, {C,D,E} -> {E,A,B}

Solution: One possible minimal basis is:

{{B,D} → {E}, {C,D} → {B}, {C,E} → {D}}

Lossless join and dependency preserving 3NF decomposition of R

1. Combine the FD we got from the minimal basis. Check the keys we calculated, there must be **at least one key** from the schema and FDs in the 3NF decomposition, so if its not present, add it in as a new table.

2. Remember when combine must create **canonical cover**

(d) Using the minimal basis you computed in part (c), find a lossless-join and dependency-preserving 3NF decomposition of R. Note that you have to first ensure that your answer to part (c) is a minimal basis first. Write the answer in the same format as part (b).

Solution: Using the previous minimal basis, we arrive at the following decomposition:

{R1(B,D,E), R2(B,C,D), R3(C,D,E)}

The keys are not in the decomposition, so we add the key R4(A,C,E). And voila, we have exactly the same decomposition as our BCNF decomposition.

Lossless join and dependency preserving BCNF decomposition of R

→ Typically this is by luck, may not always get one. We get it by checking our 3NF decomposition, make sure every table is in BCNF.

Deriving preserved FDs from split tables

- Compute closure on each table w.r.t original fds!!! → get the new fds that are captured by the new tables. Use them to check if they can derive the original FDs.

(a) Consider the following ER diagram:

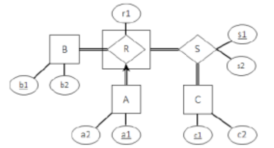


Figure 1: ER Diagram

→ Qn: Find min set of attr that can uniquely identify all other attr

Solution: $A = \{a1, s1, c1\}$.

Due to the key constraint from A to R, we actually have $\{a1\} \to \{r1\}$.

→ Notice that we have a participation and cardinality constraint from A to R, so there is only one unique a1, which means a1 → r1, b1, so we don't need b1 to uniquely identify the other attributes in R

→ double line – need both keys. If have arrow and double line then only need that key. If question says ignore NULL value then I think arrow itself can imply FD alrd

A lossless join decomposition is a decomposition of a relation into relations. such that a natural join of the two smaller relations yields back the original relation

To check for lossless join decomposition using FD set, e.g R(ABCD) splits into R1(ABC), R2(AD), R3(CD). For R1 and R2, common attribute is A. Check if A -> ABC or A -> AD can be found using the FDs from the original table ABCD. If can compute with the closure → it is lossless

SQL Data Types

boolean	logical Boolean (true/false)
integer	signed four-byte integer
float8	double precision floating-point number (8 bytes)
numeric [(p,s)]	exact numeric of selectable precision
char(n)	fixed-length character string
varchar(n)	variable-length character string
text	variable-length character string
date	calendar date (year, month, day)
timestamp	date and time

Numeric(10,3) – 10 digits, 3 d.p. If u put 1d.p pgsql will auto pad 0s until 10dp

Inserting, Deleting, Update-ing Data:

```
INSERT INTO Employees (id, name) VALUES (102, 'Judy'), (103, 'Max');
```

```
-- Delete selected tuples
DELETE FROM Employees
WHERE role = 'dev';
```

```
UPDATE Sells s
SET price = CASE
    WHEN r.area = 'Central' THEN price + 3
    WHEN r.area = 'EAST' THEN price + 2
    ELSE price + 1 END
FROM Restaurants r
where s.rname = r.rname;
```

SQL NULL Values

Use IS NULL to check if SQL value is null. For non null, can use x IS NOT NULL or NOT (x IS NULL)

IS DISTINCT FROM

Equivalent to x <> y. For x is null and y is null, x IS DISTINCT FROM y returns false, for x null and y non-null, x IS DISTINCT FROM y returns true

NOT NULL Constraint

```
CREATE TABLE Employees (
    id    VARCHAR(50) CONSTRAINT nn_id NOT NULL,
    name  VARCHAR(50) CONSTRAINT nn_name NOT NULL,
    age   INTEGER,
    role  VARCHAR(50),
);
```

UNIQUE Constraint

can be used across multiple columns in table constraint

unnamed column constraint

```
CREATE TABLE Employees (
    id    INTEGER UNIQUE,
    name  VARCHAR(50),
    age   INTEGER,
    role  VARCHAR(50)
);
```

named column constraint

```
CREATE TABLE Employees (
    id    INTEGER CONSTRAINT u_id UNIQUE,
    name  VARCHAR(50),
    age   INTEGER,
    role  VARCHAR(50)
);
```

unnamed table constraint

```
CREATE TABLE Employees (
    id    INTEGER,
    name  VARCHAR(50),
    age   INTEGER,
    role  VARCHAR(50),
    UNIQUE (id)
);
```

named table constraint

```
CREATE TABLE Employees (
    id    INTEGER,
    name  VARCHAR(50),
    age   INTEGER,
    role  VARCHAR(50),
    CONSTRAINT u_id UNIQUE (id)
);
```

For unique to be false:

"$(t_i.eid <> t_k.eid)$ or $(t_i.pname <> t_k.pname)$" evaluates to ***false***

i.e. both eid and pname for 2 tuples exactly same

Note

- Column and table constraints can be combined, even w.r.t to same column
- All can be column/table constraint except NULL

Foreign Key Constraints

- Reference pkey of another table

```
CREATE TABLE Teams (
    eid     INTEGER,
    pname   VARCHAR(100),
    hours   INTEGER,
    PRIMARY KEY (eid, pname),
    FOREIGN KEY (eid) REFERENCES Employees (id) ON DELETE NO ACTION ON UPDATE CASCADE,
    FOREIGN KEY (pname) REFERENCES Projects (name) ON DELETE SET NULL ON UPDATE CASCADE
);
```

optional since it is the default action

→ Might not need to specify id and name in employees and projects, DBMS know which is pk

→ Either primary key in referenced relation OR NULL value

→ Deletion/Update of referenced tuple can lead to error thrown/cascade etc

FK Violation Actions (ON DELETE/UPDATE)

NO ACTION	rejects delete/update if it violates constraint (default value)
RESTRICT	similar to "no action" except that check of constraint cannot be deferred (deferrable constraints are discussed in a bit)
CASCADE	propagates delete/update to referencing tuples
SET DEFAULT	updates foreign keys of referencing tuples to some default value (important: default value must be a primary key in the referenced table!)
SET NULL	updates foreign keys of referencing tuples to *null* (important: corresponding column must allowed to contain *null* values!)

```
CREATE TABLE Teams (
    eid     INTEGER,
    pname   VARCHAR(100) DEFAULT 'FastCash',   -- default value must be primary key in "Projects"!
    hours   INTEGER,
    PRIMARY KEY (eid, pname),
    FOREIGN KEY (eid) REFERENCES Employees (id) ON UPDATE CASCADE,
    FOREIGN KEY (pname) REFERENCES Projects (name) ON UPDATE CASCADE ON DELETE SET NULL
);
```

→ Special example: when project is deleted, set null, but default fastcash so final value is fastcash

→ ON DELETE CASCADE can have bad conseq, and affect performance

CHECK Constraints – CHECK(Condition)

→ can make condition as complex as needed

```
CREATE TABLE Projects (
    name        VARCHAR(50) PRIMARY KEY,
    start_year  INTEGER,
    end_year    INTEGER,
    -- CHECK (start_year <= end_year),
    CONSTRAINT valid_lifetime CHECK (start_year <= end_year)
);
```

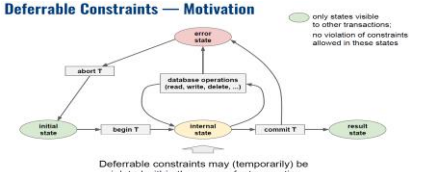
CREATE ASSERTIONS

Not really used in practice, but triggers are.

DEFERRABLE CONSTRAINTS

- Constraint check deferred to end of tranxctn

Deferrable Constraints — Motivation



Deferrable constraints may (temporarily) be violated within the scope of a transaction

→ NOT DEFERRABLE, check constraint after each statement in a transaction

```
CONSTRAINT manager_fkey FOREIGN KEY (manager) REFERENCES Employees (id)
    NOT DEFERRABLE -- default value (optional), check if constraint is immediate and cannot be changed
```

→ DEFERRABLE INITIALLY DEFERRED, only check constraint after transaction ends

```
CONSTRAINT manager_fkey FOREIGN KEY (manager) REFERENCES Employees (id)
    DEFERRABLE INITIALLY DEFERRED -- check of constraint deferred by default
```

→ DEFERRABLE INITIALLY IMMEDIATE – DB checks if violation occurs after every statement, but we can switch it off on demand, unlike NOT DEFERRABLE, can never be switched off

```
CREATE TABLE Employees (
    id        INTEGER PRIMARY KEY,
    name      VARCHAR(50),
    manager   INTEGER,
    CONSTRAINT manager_fkey FOREIGN KEY (manager) REFERENCES Employees (id)
        DEFERRABLE INITIALLY IMMEDIATE -- check of constraint immediate by default, but can be cha
);

INSERT INTO Employees VALUES (101, 'Sarah', null), (102, 'Judy', 101), (103, 'Max', 102);

BEGIN;
SET CONSTRAINT manager_fkey DEFERRED;           -- Set check of constraint from "immediate" to "defe
DELETE FROM Employees WHERE id = 102;           -- Judy got fired → constraint violated buy not che
UPDATE Employees SET manager = 101 WHERE id = 103;  -- Max gets a new manager → constraint re-estab
COMMIT;
```

+ No need to care abt order of statements within txn, allow for cyclic fk constraints, less checks higher performance

- Harder to debug, data defintn not unambiguous

Modify Single Column in DB

```
ALTER TABLE Projects ALTER COLUMN name TYPE VARCHAR(200);    -- change data type to VARCHAR(200)
ALTER TABLE Projects ALTER COLUMN start_year SET DEFAULT 2021;  -- set default value of column "start_year"
ALTER TABLE Projects ALTER COLUMN start_year DROP DEFAULT;   -- drop default value of column "start_year"
```

Add/Drop Columns

```
ALTER TABLE Projects ADD COLUMN budget NUMERIC DEFAULT 0.0;   -- add new column with a default value
ALTER TABLE Projects DROP COLUMN budget;                      -- drop column from table
```

Add/Drop Constraints

```
ALTER TABLE Teams ADD CONSTRAINT eid_fkey FOREIGN KEY (eid) REFERENCES Employees (id);
    -- add foreign key constraint

ALTER TABLE Teams DROP CONSTRAINT eid_fkey;
    -- drop foreign key constraint (name of constraint might be retrieved from metadata)
```

Dropping Tables

- DROP TABLE (IF EXISTS) Projects;

- With dependent objects (assume foreign key constraint Teams.pname → Projects.name)

```
DROP TABLE Projects;           -- will throw an error because of foreign key constraint

DROP TABLE Projects CASCADE;   -- will delete table "Projects" and foreign key constraint
                               -- (will not delete table "Teams"!)
```

→ Drop foreign key constraint in Teams, does not delete teams!

SELECT CLAUSE – DOESN'T ELIMINATE DUPS

SELECT [DISTINCT] target-list **FROM** relation-list

[WHERE conditions **] [AND** condition **]**

→ combine, process attributes, rename columns

Find the name and the all countries the GDP per capita in SGD rounded to the nearest dollar for all countries.

"||" concatenates strings — "AS" is optional

```
SELECT name, 'S$ ' || ROUND((gdp / population)*1.35) AS gdp_per_capita
FROM countries;
```

Convert from USD to SGD

name	gdp_per_capita
Denmark	S$ 63955
Finland	S$ 55490
Kyrgyzstan	S$ 4952
Norway	S$ 93586
Singapore	S$ 113944
Slovakia	S$ 46848
Turkmenistan	S$ 24166
United Arab Emirates	S$ 151910
...	...

→ Use **DISTINCT** to enforce dup elimination

WHERE Clause

→ DO NOT use col = NULL as condition, returns unknown → no tuples returned (no err thrown); use column IS NULL / column IS NOT NULL

Find all countries in Asia and Europe with a population between 5 and 6 Million.

```
SELECT *
FROM countries
WHERE (continent = 'Asia' OR continent = 'Europe')
  AND (population BETWEEN 5000000 AND 6000000);
```

→ **Pattern Matching**

- "_" matches any single character

- "%" matches any seq of 0 / more characters

'abc' LIKE 'abc'	→	true
'abc' LIKE 'a%'	→	true
'abc' LIKE '_b_'	→	true
'abc' LIKE '_c'	→	false

Find all cities that start with "Si" and end with "re".

```
SELECT name
FROM cities
WHERE name LIKE 'Si%re';
```

name
Singapore
Sierre
Sierra Madre

Set Operation Queries

→ Any SQL queries that yield union-compatible tables can do:

- Q1 UNION (ALL) Q2=Q1∪Q2

- Q1 INTERSECT (ALL) Q2=Q1 ∩ Q2

- Q1 EXCEPT (ALL) Q2 = Q1 – Q2

Find all names that refer to both a city and a country.

```
(SELECT name FROM cities)
INTERSECT ALL
(SELECT name FROM countries);
```

name
Singapore
Mexico

→ Eliminates duplicates from result

→ Add ALL behind set operation if we do not want to eliminate tuples from result

Complex JOIN query example

Find all airports in European countries without a land border which cannot be reached by plane given the existing routes in the database.

```
SELECT t1.country, t1.city, t1.airport
FROM
  (SELECT n.name AS country, c.name AS city,
          a.name AS airport, a.code
   FROM borders b, countries n, cities c, airports a
   WHERE b.country1_iso2 = n.iso2
     AND n.iso2 = c.country_iso2
     AND c.name = a.city
     AND b.country2_iso2 IS NULL
     AND n.continent = 'Europe') t1
LEFT OUTER JOIN
  routes r
ON t1.code = r.to_code
WHERE r.to_code IS NULL;
```

subquery → table

4 tables ⇓ 3 distinct selections

attribute selections for join operations

All airports in European countries without a land border (13 tuples)

AS optional

country	city	airport
Iceland	Hofn	Hornafjörður Airport
Malta	Pembroke	Pembroke Airport
Malta	Victoria	Victoria Airport

Subqueries

→ Must be enclosed in parentheses, table alias mandatory, column aliases optional

→ Subquery must return exactly 1 col (not row)

→ Expression compared to every subquery row

→ Subquery can be correlated to outer query

→ Can contain multiple nested subqueries

IN / NOT IN Subqueries

→ **Can replace IN with inner joins, NOT IN with outer joins**

Find all countries in Asia and Europe with a population between 5 and 6 Million.

```
SELECT *
FROM countries
WHERE continent IN ('Asia', 'Europe')
  AND population BETWEEN 5000000 AND 6000000;
```

Find all names that refer to both a city and a country.

outer query →

```
SELECT name
FROM countries
WHERE name IN (SELECT name
               FROM cities);
```

← inner query

population

ANY / SOME Subqueries

→ Just need to be true for 1 val in subquery

Find all countries with a population size smaller than any city called "London" (there are actually 4 cities called "London" on the database).

```
SELECT name, population
FROM countries
WHERE population < ANY (SELECT population
                        FROM cities
                        WHERE name = 'London');
```

145 tuples

name	population
Singapore	5781728
Portugal	10833816
Sweden	9880604
Brunei	436620
Bhutan	750125
Austria	8711770
Jamaica	2970340
Maldives	392960
...	

ALL Subqueries

→ Must be true for ALL values in subquery

For each continent, find the country with the highest GDP.

```
SELECT name, continent, gdp
FROM countries c1
WHERE gdp >= ALL (SELECT gdp
                  FROM countries c2
                  WHERE c2.continent = c1.continent);
```

"Europe"

GERMANY

name	continent	gdp
Australia	Oceania	1190000000000
Brazil	South America	3080000000000
China	Asia	21100000000000
Egypt	Africa	1110000000000
Germany	Europe	3980000000000
United States	North America	18600000000000

Correlated Subqueries

→ Naming ambiguities: Use table aliases, but scoping rules go from inner to outer scope

→ Outer scope will not know about table alias declaration in inner scope

(NOT) EXISTS Subqueries

→ **EXISTS** – As long as one tuple match, **NOT EXISTS** – NO tuples can match

→ **NOT EXIST** subqueries should be correlated - good tool to filter tuples, similar to EXCEPT??

Find the all the countries for which there is not city in the database.

```
SELECT n.name
FROM countries n
WHERE NOT EXISTS (SELECT *
                  FROM cities c
                  WHERE c.country_iso2 = n.iso2);
```

Scalar Subqueries

→ Subquery that returns 1 single value

→ DBMS looks at key constraints to know that subquery is scalar

→ Can use as SELECT list, WHERE, JOIN

For all cities, find their names together with the names of the countries they are located in.

```
SELECT name AS city,
       (SELECT name AS country
        FROM countries n
        WHERE n.iso2 = c.country_iso2)
FROM cities c;
```

Sorting – ORDER BY

→ ORDER BY column DESC / ASC

→ Can sort w.r.t multiple attributes

Find all cities sorted by country (ascending from A to Z) and for each country with respect to the cities' population size in descending order.

```
SELECT n.name AS country, c.name AS city, c.population
FROM cities c, countries n
WHERE c.country_iso2 = n.iso2
ORDER BY n.name ASC, c.population DESC;
```

The 2nd sorting criteria only affects result if 1st sorting criteria does not yield an unambiguous order already!

country	city	population
Afghanistan	Kabul	3678034
Afghanistan	Kandahar	491500
Afghanistan	Herat	436300
...	...	...
Albania	Tirana	418495
Albania	Durres	142432
Albania	Vlore	130827
...	...	...
Zimbabwe	Chivhu	10263
Zimbabwe	Mazoe	9966
Zimbabwe	Plumtree	2148

ROW Constructors

- Combine cols to compare > 1 col in subquery

Find all countries with a higher population or higher gdp than France or Germany

```
SELECT name, population, gdp
FROM countries
WHERE ROW(population, gdp) > ANY (SELECT population, gdp
                                  FROM countries
                                  WHERE name IN ('Germany', 'France'));
```

LIMIT / OFFSET (Usually used with ORDER BY)

→ Limit k – Return first k

→ OFFSET i – consider i^{th} tuple onwards

Find the "second" top-5 countries regarding their GDP per capita for all countries.

```
SELECT name, (gdp/population) AS gdp_per_capita
FROM countries
ORDER BY gdp_per_capita DESC
OFFSET 5
LIMIT 5;
```

AGGREGATION FUNCTIONS

- Let *R*, *S* be two relations with an attribute *A*
  - Let R be an empty relation
  - Let S be a non-empty relation with *n* tuples but only null values for *A*

Query	Result
SELECT MIN(A) FROM R;	null
SELECT MAX(A) FROM R;	null
SELECT AVG(A) FROM R;	null
SELECT SUM(A) FROM R;	null
SELECT COUNT(A) FROM R;	0
SELECT COUNT(*) FROM R;	0

Query	Result
SELECT MIN(A) FROM S;	null
SELECT MAX(A) FROM S;	null
SELECT AVG(A) FROM S;	null
SELECT SUM(A) FROM S;	null
SELECT COUNT(A) FROM S;	0
SELECT COUNT(*) FROM S;	*n*

[MIN, MAX, AVG, COUNT, SUM(attr)]

- All except COUNT will ignore null values

→ Return type dependent on column data type

- Examples
  - MIN(), MAX() defined for all data types; return date type same as input data type
  - SUM() defined for all numeric data types; SUM(INTEGER)→BIGINT, SUM(REAL)→REAL, ...
  - COUNT() defined for all data types; COUNT(...)→BIGINT

GROUP BY

→ Always applied with aggregation

- Example:
  - Table *R* with three attributes *A*, *B*, *C*

A	B	C
null	4	19
6	1	null
20	2	10
1	1	2
1	18	2
null	21	19
6	20	null

```
SELECT ...
FROM R
GROUP BY A, C;
```

A	B	C
null	4	19
null	21	19
6	1	null
6	20	null
20	2	10
1	1	2
1	18	2

GROUP BY Restrictions to SELECT clause

- If column A_i of table *R* appears in the **SELECT** clause, one of the following conditions must hold:
  - A_i appears in the **GROUP BY** clause
  - A_i appears as input of an aggregation function in the **SELECT** clause
  - The primary key ~~or a candidate key~~ of R appears in the **GROUP BY** clause

HAVING CLAUSE (GROUP BY)

- **HAVING conditions**
  - Conditions check for each group defined by **GROUP BY** clause
  - **HAVING** clause cannot be used without a **GROUP BY** clause
  - Conditions typically involve aggregate functions

Find all countries that have at least one city with a population size large than the average population size of all European countries

```
SELECT n.name, n.continent
FROM cities c, countries n
WHERE c.country_iso2 = n.iso2
GROUP BY n.name, n.continent
HAVING MAX(c.population) > (SELECT AVG(population)
                            FROM countries
                            WHERE continent = 'Europe');
```

scalar subquery

name	continent
China	Asia
Mexico	North America
India	Asia
Egypt	Africa
Philippines	Asia
Russia	Europe
Thailand	Asia
Brazil	South America
South Korea	Asia
Indonesia	Asia
United States	North America

If column A_i of table *R* appears in the **HAVING** clause, one of the following conditions must hold:
- A_i appears in the **GROUP BY** clause
- A_i appears as input of an aggregation function in the **HAVING** clause
- The primary key ~~or a candidate key~~ of R appears in the **GROUP BY** clause

→ Same logic as aggregation

CASE – Conditional Expressions

```
SELECT class, COUNT(*) AS city_count
FROM
    (SELECT name, CASE
         WHEN population > 10000000 THEN 'Super City'
         WHEN population > 5000000 THEN 'Mega City'
         WHEN population > 1000000 THEN 'Large City'
         WHEN population > 500000 THEN 'Medium City'
         ELSE 'Small City' END AS class
     FROM cities) t
GROUP BY class;
```

subquery

→ Transform column based on conditions

COALESCE – Conditional Expr. for NULL Vals

- COALESCE(value1, value2, value3, ...)
  - Returns the first non-NULL value in the list of input arguments
  - Returns NULL if all values in the list of input arguments are NULL
  - Example: **SELECT COALESCE**(*null*, *null*, 1, *null*, 2) →

val
1

Find the number of cities for each city type; consider cities with NULL for column "capital" as "other".

```
SELECT capital, COUNT(*) AS city_count
FROM
    (SELECT COALESCE(capital, 'other') AS capital
     FROM cities) t
GROUP BY capital;
```

capital	city_count
primary	202
other	17147
admin	3531
minor	3687

NULLIF – Conditional Expr. for NULL Values

NULLIF(v1, v2)

- Returns NULL if v1=v2; else return v1

→ Convert zero, empty str to NULL values

```
SELECT MIN(NULLIF(gdp, 0)) AS min_gdp,
       ROUND(AVG(NULLIF(gdp, 0))) AS avg_gdp
FROM countries;
```

min_gdp	avg_gdp
1500000	549329956636

Universal Quantification

→ Concept of a 'FORALL' operator – person X has visited ALL countries (for e.g.)

→ Use 'not exists (X except Y) or below:

"Find the names of all users that have visited all countries."

```
SELECT u.user_id, u.name
FROM users u, visited v
WHERE u.user_id = v.user_id
GROUP BY u.user_id
HAVING COUNT(*) = (SELECT COUNT(*) FROM countries);
```

user_id	name
103	Max
107	Emma

Recursive Queries

Find all airports that can be reached from SIN with 0..2 stops. (limitation to max. 2 stops purely for performance reasons)

```
WITH RECURSIVE flight_path AS (
    SELECT from_code, to_code, 0 AS stops
    FROM connections
    WHERE from_code = 'SIN'
    UNION ALL
    SELECT c.from_code, c.to_code, p.stops+1
    FROM flight_path p, connections c
    WHERE p.to_code = c.from_code
    AND p.stops < 2
)
SELECT DISTINCT to_code, stops
FROM flight_path
ORDER BY stops ASC;
```

to_code	stops
PEK	0
BKK	0
FRA	0
...	...
KUA	0
DUB	1
...	...
PEK	1
SIN	1
...	...
MME	1
AMS	2
BKK	2
...	...
PER	2
...	...
ZYL	2

103 tuples

927 tuples

1,725 tuples

Deriving preserved FDs from split tables

- Compute closure on each table w.r.t original fds!!! → get the new fds that are captured by the new tables. Use them to check if they can