# CS2102

## DBMS

### Challenges

Want

- (Efficiency) Fast access to information in huge volumes of data
- (Transactions) "All-or-nothing" changes to data
- (Data integrity) Parallel access and changes to data
- (Recovery) Fast and reliable handling of failures
  - HDD/SSD/system crash | Power outage | Network disruption
- (Security) Fine-grained data access rights

### File-based data management

- Complex, low-level code
- Often similar requirements across different programs

**Problems** High development effort | Long development times | Higher risk of (critical) errors

### Transaction

- Finite sequence of database operations (reads and/or writes)
- Smallest logical unit of work from an application perspective

Each transaction T satisfies the following ACID properties:

- Atomicity: either all effects of T are reflected in the database or none
- Consistency: Execution of T guarantees to yield a correct state of the database
- Isolation: Execution of T is isolated from the effects of concurrent transactions
- Durability: After the commit of T, its effects are permanent even in case of failures

### Concurrent Execution

#### Common problems

| $T_1$(B, 500) | $T_2$(B, 100) |
|---|---|
| begin | |
| read(B) | |
| B = B + 500 | |
| | begin |
| | read(B) 1,000 |
| | B = B + 100 1,100 |
| write(B) 1,500 | |
| commit | |
| | write(B) 1,100 |
| | commit |

Final balance B = 1,100 (effect of $T_1$ overwritten)

➜ **Lost Update**

| $T_1$(B, ~~100~~ 500) | $T_2$(B, ~~500~~ 100) |
|---|---|
| begin | |
| read(B) 1000 | |
| B = B + 500 1,500 | |
| write(B) | |
| | begin |
| | read(B) 1,500 |
| | B = B + 100 1,600 |
| | write(B) |
| | commit |
| abort | |

Final balance B = 1,600 (when it should be 1,100)

➜ **Dirty Read**

| $T_1$(B, 100) | $T_2$(B, 500) |
|---|---|
| begin | |
| read(B) | |
| | begin |
| | read(B) 1000 |
| | B = B + 500 1,500 |
| | write(B) 1,500 |
| | commit |
| read(B) | |
| ... | |

Balance B is retrieved twice but the values differ

➜ **Unrepeatable Read**

#### Requirements

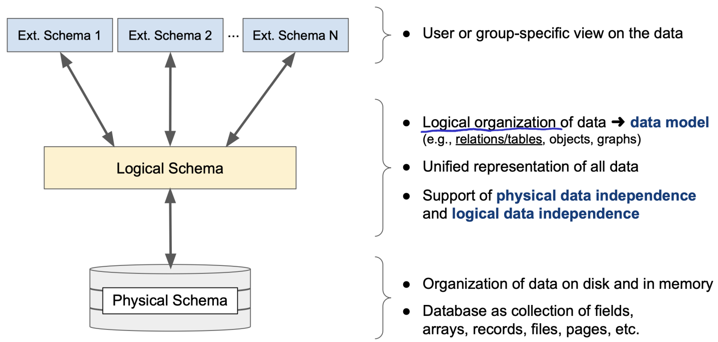
- Want serializable transaction execution
  - A concurrent execution of a set of transactions is **serializable** if this execution is equivalent to some serial execution of the same set of instructions
  - Two executions are equivalent if they have the same effect on the data
- Hence, DBMS
  - Supports concurrent executions of transactions to optimize performance
  - Enforces serializability of concurrent executions to ensure integrity of data

### DBMS

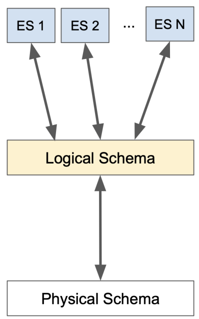
- Is a set of universal and powerful functionalities for data management
- Complex, low-level code moved from application logic to DBMS

**Benefits** Faster application development | Increased productivity | Higher stability / less errors

#### Data abstraction




#### Data independence




**Logical data independence**

- Ability to change logical schema without affecting external schemas (e.g., adding/deleting/updating attributes, changing data types, changing data model)

**Physical data independence**

- Representation of data independent from physical scheme
- Physical schema can be changed without affecting logical schema (e.g., creating indexes, new caching strategies, different storage devices)

#### Terminology

**Data model**

- Set of concepts for describing data
- Framework to specify structure of a DB
- e.g. Relational model, where everything is a table

**Schema** Description of structure of a DB, using the concepts provided by data model

**Schema instance** Content of a DB at a particular time

### Relational Model

#### Terminology

**Relation**

- Set of tuples (or records)
- $R(A_1, A_2, \cdots, A_n)$: relation schema with name $R$, and $n$ attributes $A_1, A_2, \cdots, A_n$
- Each instance of schema $R$ is a relation which is a subset of
$$\{(a_1, a_2, \cdots, a_n) \mid a_i \in dom(A_i) \cup \{null\}\}$$

**Relation schema**

- Definition of a relation
- Specifies relation name, attributes (columns) and data constraints (e.g. domain constraints)
  - Employees (id: **integer**, name: **text**, dob: **date**, salary: **numeric**)

**Domain** Set of atomic values (e.g. integer, numeric, text)

- Domain of attribute $A_i$, $dom(A_i)$ = set of possible values of $A_i$
- Each value $v$ of attribute $A_i$: either $v \in dom(A_i)$ or $v =$ **null**
- **null** - special value indicating the $v$ is not known or not specified

**Relational database schema** Set of relation schemas + data constraints

**Relational database** Collection of tables

**DB vs DBS vs DBMS**
$$DBS = DBMS + n \times DB, \quad \text{where } n > 0$$

### Integrity constraints

- Condition that restricts what constitutes valid data
- 3 main structural integrity constarints of the Relation Model
  - Domain constraints | Key constraints | Foreign key (referential integrity) constraints

#### Domain constraints

- Data type
- NOT NULL / UNIQUE / PRIMARY KEY / FOREIGN KEY / CHECK / DEFAULT

## Key constraints

**Superkey** Subset of attributes that **uniquely** identifies a tuple in a relation

**Key** Superkey that is also minimal (i.e. no proper subset of the key is a superkey)

**Candidate keys** Set of all possible keys for a relation

**Primary key**

- Chosen candidate key for a relation
- Cannot be **null** (entity integrity constraint)
- Underlined in relation schema
  - Employees (id: **integer**, name: **text**, dob: **date**, salary: **numeric**)

**Prime attribute** Attribute of a primary key (cannot be null)

## Foreign key constraints

**Foreign key**

- Subset of attributes of relation A that refer to the primary key in a relation B
- Each foreign key in referencing relation must either
  - Appear as primary key in referenced relation, or
  - Be **null**

**Properties**

- Specified by DB designer to define what constitutes valid data
- Referencing and referenced relation can be the same relation (e.g. each employee has at most one manager)
- Relation can be referencing and referenced relation for different relations

## Limitations

- Covers application-independent constraints (e.g. limit domain to valid values)
- Does not cover application-dependent constraints derived from deeper semantics of the data

## Practical considerations

- Optional, not mandatory
- May affect performance, since checking constraints require additional processing

# RELATIONAL ALGEBRA

- Relations are closed under the Relational Algebra

## Unary operators

### Selection $\sigma_c$

- For each tuple $t \in R$, $t \in \sigma_c(R) \iff$ selection condition $c$ evaluates to true for tuple $t$.
- Input and output have same schema
- e.g. Find all projects where Judy is the manager:

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

**Selection condition** is a boolean expression of one of the following forms:

| expression | example |
|---|---|
| attribute **op** constant | $\sigma_{\text{start}=2020}(\text{Projects})$ |
| $attr_1$ **op** $attr_2$ | $\sigma_{\text{start}=\text{end}}(\text{Projects})$ |
| $expr_1 \land expr_2$ | $\sigma_{\text{start}=2020 \land \text{end}=2021}(\text{Projects})$ |
| $expr_1 \lor expr_2$ | $\sigma_{\text{start}=2020 \lor \text{end}=2021}(\text{Projects})$ |
| $\neg\, expr$ | $\sigma_{\neg(\text{start}=2020)}(\text{Projects})$ |
| $(expr)$ | - |

where

- **op** $\in \{=, <>, <, \leq, \geq, >\}$
- Precedence: $()$, **op**, $\neg, \land, \lor$
- Comparision with **null** is **unknown**, arithmetic with **null** is **null**

In boolean expressions, treat unknown as literally unknown. e.g.

- false $\land$ unknown = false
- false $\lor$ unknown = unknown
- $\neg$ unknown = unknown
- true $\land$ unknown = unknown
- true $\lor$ unknown = true

### Projection $\pi_l$

- Projects columns of a table specified in list $l$
- Order of attributes in $l$ matters
- Duplicates are removed, because a relation is a set of tuples

**Example**

Teams

| ename | pname | hours |
|---|---|---|
| Sarah | BigAI | 10 |
| Sam | BigAI | 5 |
| Sam | BigAI | 3 |

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

| pname | ename |
|---|---|
| BigAI | Sarah |
| BigAI | Sam |

### Renaming $\rho_l$

- Renames attributes of a relation

Consider $R$(ename, pname, hours). Rename ename to name, pname to title. Can either specify

- list of all attr.: $\rho_{(\text{name, title, hours})}(R)$
- or list of renames:

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

## Set operations

- Union, Intersection, Set difference (all obvious)
- Note: intersection can be expressed with union and set difference:

$$R \cap S = (R \cup S) - ((R - S) \cup (S - R))$$

- The two relations must be union-compatible

### Union compatability

Two relations are union-compatible if

- Same number of attributes
- Corresponding attributes have same or compatible domains (different attribute names are ok)

**Example** The following are union-compatible.

- Employees(name: **text**, role: **text**, age: **integer**)
- Teams(ename: **text**, pname: **text**, hours: **integer**)

### Cross product

Forms all possible pairs of tuples from the two relations

## Division operator

1. This question considers a binary relational algebra operator called the **division operator** denoted by /[1].

   Consider two relations $R$ and $S$ where the set of attributes in the schema of $R$ and $S$ are $(A_1, \cdots, A_m, B_1, \cdots, B_n)$ and $(B_1, \cdots, B_n)$ respectively where $m \geq 1$ and $n \geq 1$. That is, the set of attributes in $S$ is a *proper* subset of the set of attributes in $R$.

   Assume that the attributes that are in $R$ but not in $S$ are ordered as $(A_1, \cdots, A_m)$ in the schema of $R$ and the schema of $S$ is $(B_1, \cdots, B_n)$. Let $L$ denote the list of attributes in the schema of $R$.

   The division of $R$ by $S$ (denoted by $R/S$) computes the largest set of tuples $Q \subseteq \pi_{A1,\cdots,A_m}(R)$ such that for every tuple $(a_1, \cdots, a_m) \in Q$,

$$\pi_L(\{(a_1, \cdots, a_m)\} \times S) \subseteq R$$

   $Q$ is also referred to as the **quotient** of $R/S$ and its schema is $(A_1, \cdots, A_m)$. The following example illustrates $R/S$ given two relations $R(A, B)$ and $S(B)$.

R

| A | B |
|---|---|
| a | 1 |
| a | 2 |
| b | 1 |
| c | 1 |
| c | 2 |
| c | 3 |
| d | 2 |
| d | 3 |

S

| B |
|---|
| 1 |
| 2 |

R/S

| A |
|---|
| a |
| c |

## Join operations

- Combines $\times, \sigma_c, \pi_l$ into a single op
- Simple relational algebra expressions

## Inner joins

- Eliminates tuples that do not satisfy matching criteria (i.e. selection)
- Is a selection from cross product

### $\theta$-Join

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

### Equi Join

Like $\theta$-Join, but $\theta$ must only involve $=$

### Natural Join

Like equi join (i.e. only equality operator), but

- Join is performed over common attributes of $R$ and $S$
- If there are no common attributes, acts like a cross product, since selection condition $c$ is vacuously true
- Output relation keeps one copy of common attributes

Formally,

$$R \bowtie S = \pi_l(R \bowtie_c \rho_{b_i \leftarrow a_i, \cdots, b_k \leftarrow a_k}(S))$$

where

- $A = \{a_i, \cdots, a_k\}$ is the set of common attributes of $R$ and $S$
- $c = (a_i = b_i) \wedge \cdots \wedge (a_k = b_k)$
- $l =$ list of (attr. of $R$ + attr. of $S$ not in $A$)

## Outer joins

- Inner join + dangling tuples
- A **dangling tuple** is a tuple that doesn't satisfy the inner join condition, i.e. foreign key not referenced in the relation.

### Steps

- Perform inner join $M = R \bowtie_\theta S$
- To $M$, add dangling tuples from

$$\begin{cases} R & \text{in left outer join} ⟕_\theta \\ S & \text{in right outer join} ⟖_\theta \\ R \text{ and } S & \text{in full outer join} ⟗_\theta \end{cases}$$

- Pad missing attribute values with **null**

### Formal definitions

- Set of dangling tuples in $R$, with respect to $R \bowtie_\theta S$
$$\text{dangle}(R \bowtie_\theta S) \subseteq R$$
- $null(R)$ is a $n$-compoennt tuple of **null** values, where $n$ is the number of attributes in $R$
- Left outer join ($R ⟕_\theta S$)
$$= (R \bowtie_\theta S) \cup (\text{dangle}(R \bowtie_\theta S) \times \{\text{null}(S)\})$$
- Right outer join ($R ⟖_\theta S$)
$$= (R \bowtie_\theta S) \cup (\{\text{null}(R)\} \times \text{dangle}(S \bowtie_\theta R))$$
- Full outer join ($R ⟗_\theta S$)
$$= (R \bowtie_\theta S) \cup \Big((\text{dangle}(R \bowtie_\theta S) \times \{\text{null}(S)\}) \cup (\{\text{null}(R)\} \times \text{dangle}(S \bowtie_\theta R))\Big)$$

### Natural outer joins

- Like natural inner joins
- Only equality operator used for condition
- Join is performed over common atributes of $R$ and $S$
- Output relation keeps one copy of common attributes

## Complex expressions

There are multiple ways to formulate a query to get the same result, e.g.

- Order of joins
- Order of selection (before/after join)
- Additional projections to minimize intermediate results

## Invalid expressions

- Attribute no longer available after projection
$$\sigma_{\text{role='dev'}}(\pi_{\text{name,age}}(Employees))$$
- Attribute no longer available after renaming
$$\sigma_{\text{role='dev'}}(\rho_{\text{position} \leftarrow \text{role}}(Employees))$$
- Incompatible attribute types
$$\sigma_{\text{age=role}}(Employees)$$

# SQL (DEFINE & MANIPULATE)

- Declarative language: focus on what to compute, not on how to compute
- Statement Level Interface - app is a mixture of host language statements and SQL statements
- Call Level Interface - app is written in host language, SQL statements passed as arguments

## Data types

- `CAST(x AS NUMERIC)` for typecasting
  - Useful to force floating point division

| type | description |
|---|---|
| boolean | logical Boolean (true/false) |
| integer | signed 4-byte integer |
| float8 | double precision floating-point number (8 bytes) |
| numeric(p, s) | number with $p$ significant digits and $s$ decimal places |
| 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 |

Other extended types:

- Document types: XML, JSON
- Spatial types: point, line, polygon, circle, box, path
- Special types: money/currency, MAC/IP address
- User defined types

## Data definition

### Creating tables

Employees (id: **integer**, name: **text**, dob: **date**, salary: **numeric**)

```sql
CREATE TABLE Employees(
  id    INTEGER,
  name  VARCHAR(50),
  age   INTEGER,
  role  VARCHAR(50) DEFAULT 'sales'
);
```

## Modifying a schema

```sql
-- change data type
ALTER TABLE Projects ALTER COLUMN name TYPE VARCHAR(200);

-- set default value
ALTER TABLE Projects ALTER COLUMN start_year SET DEFAULT
    2021;

-- drop default value
ALTER TABLE Projects ALTER COLUMN start_year DROP
    DEFAULT;

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

-- add FK constraint
ALTER TABLE Teams ADD CONSTRAINT eid_fkey FOREIGN KEY
    (eid) REFERENCES Employees(id);
-- drop FK constraint
ALTER TABLE Teams DROP CONSTRAINT eid_fkey;

-- drop table
DROP TABLE Projects;
DROP TABLE IF EXISTS Projects; -- avoids throwing error
    if does not exist

-- drop table with dependent objects
DROP TABLE Projects CASCADE; -- deletes Projects and FK
    constraint
```

# Data manipulation

## Insert data

```sql
-- Specify all attribute values
INSERT INTO Employees VALUES (101, 'Sarah', 25, 'dev');
-- Specify selected attribute values
-- role: sales, age: NULL
INSERT INTO Employees (id, name) VALUES (102, 'Judy');
```

## Deleting data

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

## Updating data

```sql
-- Update with where clause
UPDATE Employees SET age = age+1 WHERE name='Sarah';
-- Set all age to 0
UPDATE Employees SET age = 0;
-- Uppercase ALL strings
UPDATE Employees SET name=UPPER(name), role=UPPER(role);
```

# Handling NULL

- Comparison with **null** is unknown
- Arithmetic opertaion with **null** is **null**

## IS (NOT) NULL

- value is **null** $\iff$ evaluates to true
- `x IS NOT NULL` equivalent to `NOT (x IS NULL)`

## IS (NOT) DISTINCT FROM

- Equivalent to `x <> y` if $x$ and $y$ both non-null
- Both null $\implies$ return false
- One null $\implies$ return true

# Constraints

- All constraints can be named or unnamed (unnamed constraints still get named by DBMS)
- All column constraints can be specified as table constraints (except not null)
- Table constraints referring to a single column can be specified as column constraint
- Column and table constraints can be combined, e.g.

```sql
CREATE TABLE Employees (
  -- id specified twice
  id   INTEGER NOT NULL,
  name VARCHAR(50),
       UNIQUE(id)
);
```

## Not-null constraints

- Violation: $\exists t \in$ Employees where `t.id IS NOT NULL` evaluates to false

```sql
CREATE TABLE Employees(
  -- unnamed (name assigned by DBMS)
  id   INTEGER NOT NULL,
  -- named (easier bookkeeping)
  name VARCHAR(50) CONSTRAINT nn_name NOT NULL
);
```

## Unique constraints

- Violation: For any two tuples $x, y \in$ Teams, `(x.id <> y.id)` or `(x.name <> y.name)` evaluates to false
- Since `(null <> null)` evaluates to unknown, the tuples `(101, null)` and `(101, null)` are considered unique
- Unique constraint involving multiple attributes is specified using table constraints

```sql
CREATE TABLE Employees (
  -- column constraint
  id INTEGER UNIQUE,
  name VARCHAR(50),
  -- table constraint
  UNIQUE(id),       -- single attribute
  UNIQUE(id, name), -- multi attribute
  CONSTRAINT unique_id UNIQUE(id) -- named
);
```

## PK constraints

- `PRIMARY KEY` and `UNIQUE NOT NULL` have the same effect
- Only 1 `PRIMARY KEY`, but can have multiple `UNIQUE NOT NULL`

```sql
CREATE TABLE Employees (
  -- These 2 statements have the same effect
  id   INTEGER PRIMARY KEY,
  id   INTEGER UNIQUE NOT NULL,
  --
  name VARCHAR(50),
       PRIMARY KEY (id, name) -- multiple attributes
);
```

## FK constraints

The foreign key must either

- exist in the other table, or
- contain NULL in at least one of its attributes

```sql
CREATE TABLE Employees (
  id     INTEGER PRIMARY KEY,
  name   VARCHAR(50),
  age    INTEGER
);
```

```sql
CREATE TABLE Projects (
  name       VARCHAR(50) PRIMARY KEY,
  start_year INTEGER,
  end_year   INTEGER
);
CREATE TABLE Teams (
  eid    INTEGER,      -- eid -> Employee.name
  pname  VARCHAR(100), -- pid -> Project.name
  hours  INTEGER,
  PRIMARY KEY (ename, pname),
  FOREIGN KEY (eid) REFERENCES Employees (id),
  FOREIGN KEY (pname) REFERENCES Projects (name)
);
```

## FK constraints ON action

- Specify action in case a FK constraint is violated
- `ON DELETE/UPDATE <action>`
- Both specifications are optional

| case | action |
|---|---|
| NO ACTION (default) | rejects delete/update if it violates constraint |
| RESTRICT | similar to NO ACTION, but check of constraint cannot be **deferred** |
| CASCADE | propagates delete/update to referencing tuples (can significantly affect performance) |
| SET DEFAULT | updates FKs of referencing tuples to some default value (default value must be a PK in referenced table) |
| SET NULL | updates FKs of referencing tuples to null (corresponding column must be allowed to contain null values) |

```sql
CREATE TABLE Teams (
  eid    INTEGER,
  pname  VARCHAR(100),
  hours  INTEGER,
  PRIMARY KEY (ename, pname),
  FOREIGN KEY (eid) REFERENCES Employees (id) ON DELETE
      <action> ON UPDATE <action>,
  FOREIGN KEY (pname) REFERENCES Projects (name) ON
      DELETE <action> ON UPDATE <action>
);
```

## Check constraints

- Specify that column values must satisfy a Boolean expression
- Scope: one table, single row

```sql
-- column constraint
CREATE TABLE Teams (
  eid    INTEGER PRIMARY KEY,
  pname  VARCHAR(100),
  -- unnamed
  hours  INTEGER check (hours > 0),
  -- named
  hours  INTEGER constraint positive_hours check (hours
       > 0)
);

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

Can be complex Boolean expressions

```sql
CREATE TABLE Teams (
  eid    INTEGER PRIMARY KEY,
  pname  VARCHAR(100),
  hours  INTEGER,
  CHECK (
    (pname = 'CoreOS' AND hours >= 30)
    OR
    (pname <> 'CoreOS' AND hours >= 0)
  )
);
```

### Deferrable constraints

- Default behavior for constraints: check immediately at the end of SQL statement execution, even for transaction with multiple SQL statements
  - Violation causes statement to be rolled back

## Deferrable constraints

- Check can be deferred for some constraints to the end of a transaction
- Allows violation of constraints temporarily within the scope of a transaction
- Constraint still needs to be resolved at the end of the transaction
- Can be used for `UNIQUE`, `PRIMARY KEY`, `FOREIGN KEY`

**Benefits**

- No need to care about order of SQL statements within a transaction
- Allows for cyclic FK constraints
- Performance boost when constraint checks are bottleneck

**Downsides**

- Difficult to troubleshoot
- Data definition no longer unambiguous
- Performance penalty when performing queries

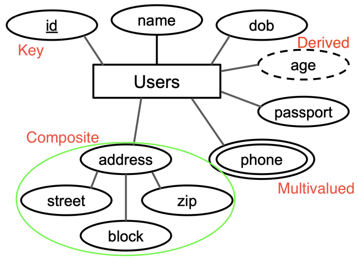
# ER MODEL

### Entity

- Objects that are distinguishable from other objects
- **Entity set:** Collection of entities of the same type

### Attribute

- Specific information describing an entity
- **Key attribute(s)** uniquely identifies each entity
- **Composite attribute** composed of multiple other attributes
- **Multivalued attribute** may consist of more than one value for a given entity
- **Derived attribute** derived from other attributes




### Relationship

Association among two or more entities

## Relationship set

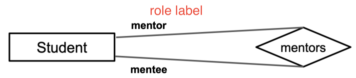
- Collection of relationships of the same type
- Can have their own attributes that further describe the relationship
- $Key(E_i)$ is the attributes of the selected key of entity set $E_i$

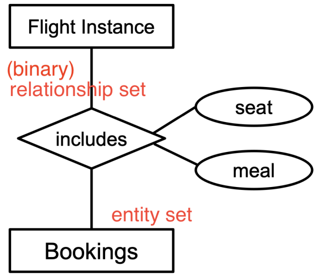
**Role**

- Describes an entity set's participation in a relationship
- Explicit role label only in case of ambiguities (e.g. same entity set participates in same relationship more than once)

**Degree**

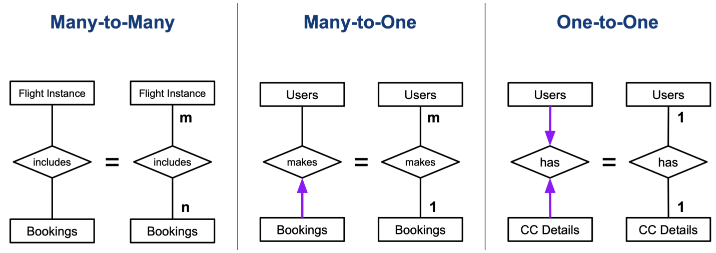
- An $n$-ary relationship set involves $n$ entity roles, where $n$ is the degree of the relationship set
- Typically binary or ternary



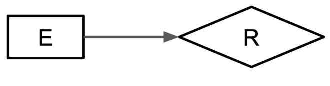

### Cardinality constraints

- **Upper bound** for entity's participation

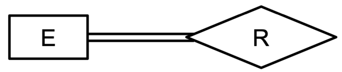



### Participation constraints

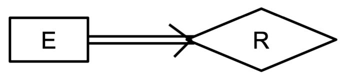
- **Lower bound** for entity's participation
- Partial (default): participation not mandatory
- Total: mandatory (at least 1)




At most one

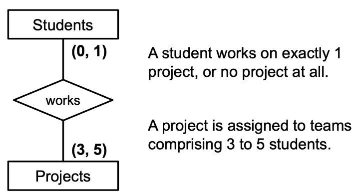



At least one




Exactly one

## Alternative



## Implementation

**Many-to-Many** Represent relationship set with a table

**Many-to-One**

1. Represent relationship set with a table
2. Combine relationship set and total participiation entity set into one table

**One-to-One**

1. Represent relationship set with a table
2. Combine relationship set and either entity set into one table

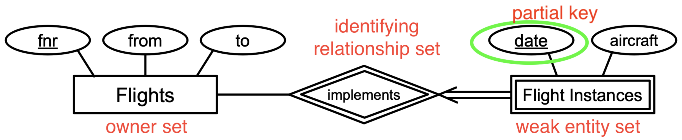
## Dependency constraints

### Weak entity sets

- Entity set that does not have its own key
- Can only be uniquely identified by considering primary key of owner entity
- Existence depends on existence of owner entity

**Partial key**

- Set of attributes of weak entity set that uniquely identifies a weak entity, for a given owner entity




**Requirements**

- Many-to-one relationship from weak entity set to owner entity set
- Weak entity set must have total participation in identifying relationship

## Relational mapping

- Entity set → table
- Composite/multivalued attributes:
  1. Convert to single-valued attributes
  2. Additional table with FK constraint
  3. Convert to a single-valued attribute (e.g. comma separated string)

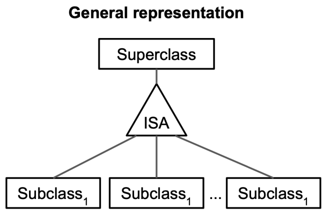
## ISA Hierarchies

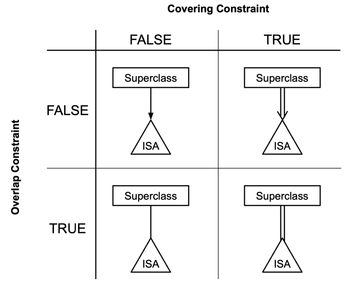
- "Is a" relationship - used to model generalization/specialization of entity sets

### Constraints

**Overlap** Can a superclass entity belong to multiple subclasses?

**Covering** Does a superclass entity have to belong to a subclass?



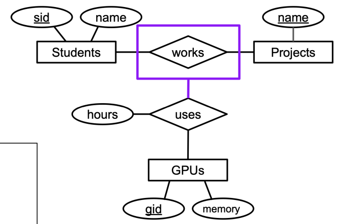

## Aggregation

- Abstraction that treats 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)
);
```




# SQL (QUERIES)

## Basic form

```
SELECT DISTINCT a1, a2, ... am
FROM r1, r2, ... rm
WHERE c
```

corresponds to

$$\pi_{a_1,a_2,\cdots,a_m}(\sigma_c(r_1 \times r_2 \times \cdots \times r_n))$$

## SELECT

- Wildcard '*' to include all attributes
- `expr BETWEEN <lower> AND <upper>` for basic value range conditions

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

- || for string concatenation

```
-- AS is optional
SELECT name, 'S$ ' || ROUND((gdp/population)*1.38)
AS gdp_per_capita FROM countries;
```

- `SELECT DISTINCT` to eliminate duplicates
- Two tuples $(n_1, c_1)$ and $(n_2, c_2)$ are distinct if `(n1 IS DISTINCT FROM n2)or (c1 IS DISTINCT FROM c2)` evaluates to true

## WHERE

- Returns rows that evaluate to true
- Does not return rows that evaluate to unknown/null!
- Use `IS (NOT) NULL` for comparison with null
- `IS (NOT) LIKE`
  - '_' matches single char
  - '%' matches any sequence of zero or more chars

## Set operations

- Eliminate duplicates: `UNION`, `INTERSECT`, `EXCEPT`
- Keep duplicates: `UNION ALL`, `INTERSECT ALL`, `EXCEPT ALL`

## Join operations

- `JOIN` is interpreted as `INNER JOIN`
- `NATURAL JOIN`
- `LEFT OUTER JOIN` interpreted as `LEFT JOIN`
  - Use `WHERE ... IS NULL` to get only dangling tuples

## Subqueries

### In FROM clause

- Must be enclosed in parenthesis
- Table alias mandatory
- Column aliases optional

```
SELECT * FROM (
  SELECT n.iso2, n.name
  FROM countries n, borders b
  WHERE n.iso2 = b.country1_iso2
  AND country2_iso2 IS NULL
) AS LandborderfreeCountries;
-- column aliases optional
-- ) AS LandborderfreeCountries(code, name);
```

### IN subquery

- `expr IN (subquery)`
- Subquery must return exactly one column
- Returns true if expr matches with any subquery row
- `IN` can typically be replaced with inner joins
- `NOT IN` can typically be replaced with outer joins

```
-- subquery is a SELECT
SELECT * FROM countries WHERE name IN (SELECT name FROM
    cities);
-- subquery is a manually specified result of a subquery
SELECT * FROM countries WHERE continent IN ('Asia',
    'Europe');
```

## ANY/SOME subquery

- `expr op ANY (subquery)`
- Subquery must return exactly one column
- Expression is compared to each subquery row using operator `op`
- Returns true if comparison evaluates to TRUE for at least one subquery row
- `.. < ANY(..)` ⇒ Not maximum
- `.. > ANY(..)` ⇒ Not minimum

## ALL subquery

- `expr op ANY (subquery)`
- Subquery must return exactly one column
- Expression is compared to each subquery row using operator `op`
- Returns true if comparison evaluates to TRUE for all subquery rows
- `.. <= ALL(..)` ⇒ Minimum
- `.. >= ALL(..)` ⇒ Maximum

## EXISTS subqueries

- `EXISTS (subquery)`
- Returns TRUE if subquery returns at least one tuple
- Generally correlated. If uncorrelated, then likely either wrong or unnecessary

## Correlated subqueries

- Uses value from outer query
- Result of subquery depends on value of outer query
  - Potentially slow performance
  - For `ALL` condition, problematic if subquery contains NULL value, since condition never evaluates to TRUE
  - Potential naming ambiguities - use table aliases

## Scoping rules

- (Scope applies inwards) Table alias decalred in subquery Q can only be used in Q, or subqueries nested within Q
- If same table alias is declared in subquery Q and in an outer query, then use the declaration in Q

## Scalar subqueries

- Occurs when subquery returns a single value (1 row, 1 column)
- Can be used as expression in queries

## Row constructors

- Allow subqueries to return more than one attribute
- Number of attributes in row constructor must match the one of the subquery

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

- If comparison op is `=` or `<>`, then tuple comparison works as expected
- However, for `<`, `<=`, `>`, `>=`, the row elements are compared left-to-right, stopping as soon as an unequal or null pair of elements is found
  - If either of this pair of elements is null, the result of the row comparison is unknown (null)
  - Otherwise comparison of this pair of elements determines the result
- Can use `SELECT ROW(...)op ROW(...)as res` to test the functionality

## Equivalent subqueries

- `expr IN (subquery)` ≡ `expr = ANY (subquery)`
- `expr1 op ANY(SELECT expr2 FROM ... WHERE ...)` ≡ `EXISTS(SELECT * FROM ... WHERE ... AND expr1 op expr2)`

## Sorting

- `ORDER BY attribute ASC (DESC)`
- `ORDER BY attr1 ASC, attr2 DESC` - 2nd sorting criteria only affects result if 1st sorting criteria has ambiguity

## Ranking

- `LIMIT k`: return first $k$ tuples of the result table
- `OFFSET k`: specify the position of the first tuple to be considered
- Used with `ORDER BY`
- `ORDER BY ... OFFSET 5 LIMIT 5` - get 6th to 10th tuples in sorting criteria

# SQL (AGGREGATION)

- Computes a single value, given a set of tuples
- e.g. `MIN()`, `MAX()`, `AVG()`, `COUNT()`, `SUM()`
- `MIN(A)` does not consider null values for attribute $A$

## Signatures

- `MIN`, `MAX` defined for all data types, returns same type as input
- `SUM` defined for all numeric data types
  - `SUM(INTEGER)-> BIGINT` | `SUM(REAL)-> REAL`
- `COUNT` defined for all data types; returns `BIGINT`

## Handling NULL

Let $R$ be a non-empty relation with attribute $A$.

| Query | Interpretation | Result |
|---|---|---|
| SELECT MIN(A) FROM R; | Minimum non-null value in A | 0 |
| SELECT MAX(A) FROM R; | Maximum non-null value in A | 42 |
| SELECT AVG(A) FROM R; | Average of non-null values in A | 12 |
| SELECT SUM(A) FROM R; | Sum of non-null values in A | 48 |
| SELECT COUNT(A) FROM R; | Count of non-null values in A | 4 |
| SELECT COUNT(*) FROM R; | Count of rows in R | 5 |
| SELECT AVG(DISTINCT A) FROM R; | Average of distinct non-null values in A | 15 |
| SELECT SUM(DISTINCT A) FROM R; | Sum of distinct non-null values in A | 45 |
| SELECT COUNT(DISTINCT A) FROM R; | Count of distinct non-null values in A | 3 |

Let $R, S$ be two relations with an attribute $A$,

- where $R$ is an empty relation, and
- $S$ is a non-empty relation with $n$ tuples, 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* |

## Grouping

- Logical partition of relation into groups, based on values for specified attributes
- Aggregate function applied over group, so 1 result tuple for each group
- Given `GROUP BY a1, a2, .. an`, 2 tuples $x$ and $y$ belong to the same group if for all $k \in \{1..n\}$, `x.ak IS NOT DISTINCT FROM y.ak` evaluates to TRUE
  - i.e. NULL is treated as a value
- If column $A$ of table $R$ appears in `SELECT` clause, one of the following conditions must hold:
  1. $A$ appears in the `GROUP BY` clause
  2. $A$ appears as input of an aggregation function in the `SELECT` clause
  3. Primary key of $R$ appears in the `GROUP BY` clause

```
-- find lowest, highest, overall population size for
   each continent
SELECT continent,
  MIN(population) AS lowest,
  MAX(population) AS highest,
  SUM(population) AS overall
FROM countries
GROUP BY continent;
```

## Having

- Conditions check for each group defined by GROUP BY clause
- Must be used with GROUP BY
- If column $A$ of table $R$ appears in HAVING clause, one of the following conditions must hold:
  1. $A$ appears in the GROUP BY clause
  2. $A$ appears as input of an aggregation function in the HAVING clause
  3. Primary key of $R$ appears in the GROUP BY clause

```sql
-- Find all routes served by >12 airlines
SELECT from_code, to_code, COUNT(*) AS num_airlines
FROM routes GROUP BY from_code, to_code HAVING COUNT(*)
    > 12;
```

## Conceptual evaluation of queries

FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT/OFFSET

# SQL (CONDITIONALS)

## CASE expression

- Generic conditional, like if/else statements
- Used in SELECT, ORDER BY, etc

Regular if-else

```sql
CASE
  WHEN cond1 then res1
  WHEN cond2 then res2
  ...
  WHEN condN then resN
  ELSE res0
END
```

Switch-like statement

```sql
CASE expression
  WHEN val1 then res1
  WHEN val2 then res2
  ...
  WHEN valN then resN
  ELSE res0
END
```

## COALESCE

- Returns first non-NULL value in list of input args
- Returns NULL if lit of input args are NULL

```sql
-- if NULL, then set type to other
SELECT type, COUNT(*) AS city_ount
FROM
  (SELECT COALESCE(type, 'other') AS type
   FROM cities) t
GROUP BY type;
```

## NULLIF

- NULLIF(v1, v2) returns NULL if $v_1 = v_2$, otherwise returns $v_1$
- Common use case: convert special values (zero, empty string) to NULL values

```sql
-- Unknown values are represented as 0
-- so set them to NULL for aggregation to ignore
SELECT MIN(NULLIF(gini, 0)) AS min_gini,
       AVG(NULLIF(gini, 0)) AS avg_gini,
FROM countries;
```

# SQL (STRUCTURING QUERIES)

## Common Table Expressions (CTEs)

- Temporarily named query
- Multiple CTEs can be used within an SQL statement
- CTEs can refer previous CTEs, or can be not referred at all
- Improves readabiliity, debugging, maintenance

General syntax

```sql
WITH
  C1 AS (Q1),
  C2 AS (Q2),
  ...
  CN as (QN),
SQL statement S;
```

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

- Each $C_i$ is the name of a temporary table defined by $Q_i$
- Each $C_i$ can reference any other $C_j$ that has been declared before $C_i$
- SQL statement $S$ can reference any possible subset of all $C_i$

## Views

- Permanently named query (virtual table)
  - Query is stored (not result), and re-executed each time view is used
- Can be used like normal tables
  - No restriction when used in SELECT statements
  - Restrictions for INSERT, UPDATE, DELETE (updatable view)

### Updatable view requirements

Satisfy all of

1. Must have exactly 1 entry in its FROM list, which must be a table, or another updatable view
2. Can only update one attribute of a particular row at a time
3. No WITH, DISTINCT, GROUP BY, HAVING, LIMIT, OFFSET
4. No set operations UNION, INTERSECT, EXCEPT
5. No aggregate functions
6. No constraint violations

```sql
CREATE VIEW view_name AS
SELECT .. FROM .. WHERE .. GROUP BY ..;
```

## Universal quantification

- No support for universal quantification (e.g. find names of all users that have visited all countries)
- Transform query using logical equivalences
  - There does not exist a country that the user has not visited
- Alternative interpretation
  - Number of tuples in Visited for that user must match total number of countries

### Logical equivalences

- Can be used for other types of queries
- e.g. $p \implies q \equiv \sim p \lor q$

## Recursive queries

- Using CTEs

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

```sql
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 | 90 tuples |
| ... | ... | |
| KUA | 0 | |
| DUB | 1 | |
| PEK | 1 | |
| SIN | 1 | 825 tuples |
| ... | ... | |
| MME | 1 | |
| AMS | 2 | |
| BKK | 2 | |
| PER | 2 | 1,561 tuples |
| ... | .... | |
| ZYL | 2 | |