CS2102

AY21/22 SEM 1

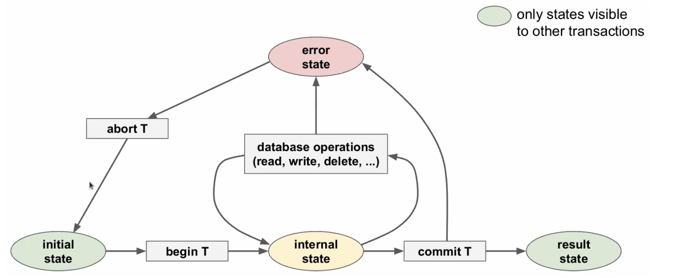
github/jovyntls

01. DBMS: DATABASE MANAGEMENT SYSTEMS

- set of universal and powerful **functionalities** for data management
- **database system** → DBMS (functionality) supporting several databases
  - DBS = DMBS + n*DB
- **data model** → framework to specify the structure of a DB
- **schema** → describes the DB structure using concepts provided by the data model
- **schema instance** → content of a DB at a particular time

Transactions

- **transaction,** T → a finite sequence of database operations
  - smallest logical unit of work from an application perspective
- guarantees the **ACID** properties



ACID properties

1. **Atomicity** → either all effects of T are reflected in the database, or none
2. **Consistency** → the execution of T guarantees to yield a *correct state* of the DB
3. **Isolation** → execution of T is *isolated* from the effects of concurrent transactions
4. **Durability** → after the commit of T, its effects are *permanent* in case of failures

Serial vs Concurrent Execution

Serial Execution

- ✓ *correct* final result
- × less (unoptimised) resource utilisation; low throughput

Concurrent Execution

- × potential issues: lost update / dirty read / unrepeatable read

Serializability

- Requirement for Concurrent Execution: **serializable transaction execution**
- (concurrent execution of a set of transactions is) **serializable** → execution is equivalent to some serial execution of the same set of transactions
  - **equivalent** → → they have the same *effect* on the data

Core tasks of DBMS

- Support *concurrent executions* of transactions - to optimise performance
- enforce *serializability* of concurrent executions - to ensure integrity of data

01-1. RELATIONAL MODEL

- **relation schema** → defines a relation
  - specifies the **attributes** (columns) and data constraints
  - **data constraints** → limits the kind of data you can put into the database
- **relational database schema** → set of relation schemas + data constraints
  - TableName(col_1, col_2, col_3) with dom(col_1) = {x, y, z}, ...
- **relational database** → collection of tables
- **domain** → a set of *atomic* values
  - domain of attribute A_i, $dom(A_i)$ = set of possible values for A_i
  - for each value v of attribute A_i, $v \in dom(A_i)$ or $v =$ `null`
    - `null`: special value indicating that v is not known or specified
  - e.g. dom(course) = {cs2102, cs2030, cs2040}
- **relation** → a set of *tuples*
  - $R(A_1, A_2, \ldots, A_n)$: relation schema with name R and n attributes $A_1, A_2, \ldots, A_n$
  - each instance of schema R is a relation which is a subset of $\{(a_1, a_2, \ldots, a_n) \mid a_i \in dom(A_i) \cup \{null\}\}$

Data Integrity

- **integrity constraint** → condition that restricts what constitutes valid data
  - DBMS will check that tables only ever contain valid data
- **structural** → (integrity) inherent to the data model
- 3 main strucutral integrity constraints of the Relation Model
  1. Domain constraints
  2. Key constraints
  3. Foreign key constraints

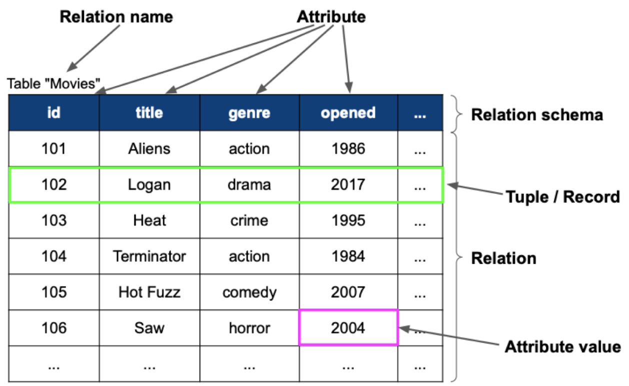
Key Constraints

- **superkey** → subset of attributes that *uniquely* identifies a tuple in a relation
  - e.g. {id, title}
- **key** → superkey that is also **minimal**
  - no proper subset of the key is a superkey
  - e.g. {id}
- **candidate keys** → set of all keys for a relation
- **primary key** → selected candidate key for a relation
  - *cannot* be `null` ⇒ **entity integrity constraint**

Foreign Key Constraints

- **foreign key** → subset of attributes of relation A if it refers to the *primary key* in a relation B.
- each foreign key in a relation must:
  1. appear as a **primary key** in the referenced relation, OR:
  2. be a `null` value

01-2. SUMMARY



id	title	genre	opened	...
101	Aliens	action	1986	...
102	Logan	drama	2017	...
103	Heat	crime	1995	...
104	Terminator	action	1984	...
105	Hot Fuzz	comedy	2007	...
106	Saw	horror	2004	...
...	...	...	...	...

02. RELATIONAL ALGEBRA

- **algebra** → mathematical system of operands and operators
  - **operands**: variables or values from which new values can be constructed
  - **operators**: symbols denoting procedures that construct new values from given values
- **relation algebra** → procedural query language
  - **operands**: relations or variables representing relations
  - **operators**: transform one or more input relations into one output relation

Closure Property

- **closure** → relations are *closed* under relational algebra
  - all input operands and outputs of all operators are *relations*
  - the output of one operator can serve as input for subsequent operators
- allows for nesting of relational operators ⇒ **relational algebra expressions**

02-1. BASIC OPERATORS

UNARY OPERATORS

Selection, σ_c

- $\sigma_c(R)$ → selects all tuples from a relation R that satisfy condition c.
  - for each tuple $t \in R,\ t \in \sigma_c(R) \iff c$ evaluates to true on t
  - input and output relation have the same schema
- **selection condition** →
  - a *boolean expression* of one of the following forms:
    - constant selection - attribute **op** constant
    - attribute selection - attribute$_1$ **op** attribute$_2$
    - expr$_1 \wedge$ expr$_2$;   expr$_1 \vee$ expr$_2$;   item $\neg$ expr ;   (expr)
  - with **op** $\in \{=, <>, <, \leq, \geq, >\}$
    - **operator precedence**: (), **op**, $\neg$, $\wedge$, $\vee$
  - handling `null` values
    - comparison operation with `null` ⇒ **unknown**
    - arithmetic operation with `null` ⇒ `null`

Projection, π_ℓ

- $\pi_\ell(R)$ → projects all attributes of a given **relation** specified in list ℓ
  - *relation* = set of tuples ⇒ duplicates removed from output relation!
  - **order** of attributes matters!

Renaming, ρ_ℓ

- $\rho_\ell(R)$ → renames the attributes of a relation R (schema $R(A_1, A_2, \ldots, A_n)$)
- 2 possible formats for ℓ
  - ℓ is the new *schema* in terms of the new attribute names
    - $\ell = (B_1, B_2, \ldots, B_n)$; $B_i = A_i$ if attribute A_i does not get renamed
  - ℓ is a list of attribute renamings of the form: $B_i \leftarrow A_i, \ldots, B_k \leftarrow A_k$
    - each renaming $B_j \leftarrow A_j$ renames attribute A_j to attribute B_j
    - order of renaming doesn't matter

SET OPERATORS

- **union** → $R \cup S$ returns a relation with all tuples in both R **or** S
- **intersection** → $R \cap S$ ... all tuples in both R **and** S
- **set difference** → $R - S$ ... all the tuples in R **but not in** S

! requirement for all set operators: R and S must be **union-compatible**

Union Compatibility

- two relations R and S are **union-compatible** → if
  - R and S have the same number of attributes; and
  - the corresponding attributes have the *same or compatible domains*
- note: R and S do not have to use the same attribute names

CROSS PRODUCT

- **cross product** → given two relations $R(A, B, C)$ and $S(X, Y)$, $R \times S$ returns a relation with schema (A, B, C, X, Y) defined as $R \times S = \{(a, b, c, x, y) \mid (a, b, c) \in R, (x, y) \in S\}$
  - combines two relations R and S by forming all pairs of tuples from the relations
- **size** of cross product $= |R| * |S|$

02-2. JOIN OPERATORS

Inner Joins

- eliminate all tuples that do not satisfy a matching criteria (i.e. **attribute selection**)

θ-join

- **θ-join** $\rightarrow$ (of two relations R and S) $R \bowtie_\theta S = \sigma_\theta(R \times S)$

Equi Join $\bowtie$

- special case of θ-join defined over the **equality** operator ($=$) only

Natural Join $\bowtie$

- the **natural join** $\rightarrow$ (of two relations R and S) is defined as
  $R \bowtie S = \pi_\ell(R \bowtie_c \rho_{b_i \leftarrow a_i, \dots, b_k \leftarrow a_k}(S))$
  - $A = \{a_i, \dots, a_k\}$ is the set of attributes that R and S have in common
  - $c = ((a_i = b_i) \wedge \dots \wedge (a_k = b_k))$
  - $\ell =$ list of all attributes of R + list of all attributes in S that are **not in** A
- performed over all attributes that R and S have in common
  - no explicit matching criteria has to be specified
- output relation contains the common attributes of R and S only *once*

Outer Joins

- **dangling tuples** $\rightarrow$ tuples in R or S that do not match with tuples in the other relation
  - ***dangle***$(R \bowtie_\theta S) \rightarrow$ set of dangling tuples in R wrt to $R \bowtie_\theta S$
    - *dangle*$(R \bowtie_\theta S) \subseteq R$
  - always removed by inner joins, kept by outer joins
  - missing attribute values are padded with `null`
- $null(R) \rightarrow$ n-component **tuple** of `null` values where n is the number of attributes of R

Definitions

- **left outer join** $\rightarrow R ⟕_\theta S = R \bowtie_\theta S \cup (dangle(R \bowtie_\theta S) \times \{null(S)\})$
- **right outer join** $\rightarrow R ⟖_\theta S = R \bowtie_\theta S \cup (\{null(R)\} \times dangle(S \bowtie_\theta R))$
- **full outer join** $\rightarrow R ⟗_\theta S$
  $= R \bowtie_\theta S \cup (dangle(R \bowtie_\theta S) \times \{null(S)\}) \cup (\{null(R)\} \times dangle(S \bowtie_\theta R))$

Natural Outer Joins

- natural left/right/full outer join: $R ⟕ S$ / $R ⟖ S$ / $R ⟗ S$
- only equality operator is used for the join condition
- join is performed over all attributes that R and S have in common
- output relation contains the common attributes of R and S only once

03. SQL

Overview

- **domain-specific language** - used for relational databases
- **declarative language** - focuses on *what* to compute, not *how* to compute
- built on top of RA
- query = `SELECT` statement

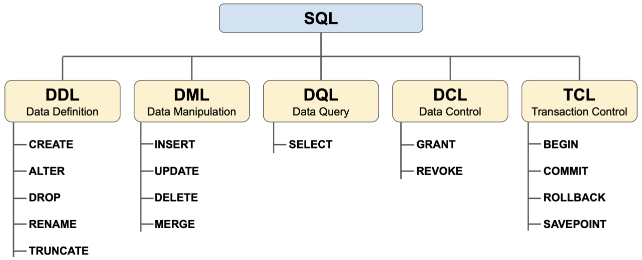
Data Types (psql)

- user-defined types
- basic data types

type	description
boolean	logical Boolean
integer	signed 4-byte integer
float8	double precision floating point number (8 bytes)
numeric[(p, s)]	exact numeric of selectable preciison
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

- char, varchar, text: different sizes to optimise storage
  - `varchar(n)` - n is the maximum length
  - `char(n)` - storage size = maximum size = n (will be padded up to n bytes)
  - `text` - usually for very long strings

Types of Commands/Statements



DDL (Data Definition)

Create Tables

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

Insert Data

```
-- specifying all attribute values
INSERT INTO Employees VALUES (101, 'John', 25, 'developer');
-- specifying selected attribute values
INSERT INTO Employees (id, name) VALUES (102, 'Smith');
```

Modify Schema

```
-- 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 a default value
ALTER TABLE Projects ADD COLUMN budget NUMERIC DEFAULT 0.0;
-- drop column from table
ALTER TABLE Projects DROP COLUMN budget;
-- add constraint
ALTER TABLE Teams ADD CONSTRAINT eid_fkey FOREIGN KEY (eid)
    REFERENCES Employees (id);
-- drop constraint
ALTER TABLE Teams DROP CONSTRAINT eid_fkey; /* eid_fkey = name
     of constraint */
```

Drop Tables

```
DROP TABLE Projects;
-- check first if table exists; avoids throwing an error
DROP TABLE IF EXISTS Projects;
-- will also delete FK constraint (but not referencing tables)
DROP TABLE Projects CASCADE;
```

DML (Data Manipulation)

Delete Data

```
-- deletes all tuples
DELETE FROM Employees;
-- deletes selected tuples
DELETE FROM Employees WHERE role='developer';
```

Update Data

```
UPDATE Employees
SET age = age + 1
WHERE name = 'John';

-- updates all values
UPDATE Employees
SET name=UPPER(name),
    job=UPPER(job);

-- updates all values
UPDATE Employees
SET age = 0;
```

Handling NULLs

- prerequisite for integrity constraints
- **comparison** operation with `null` $\Rightarrow$ *unknown*
- **arithmetic** operation with `null` $\Rightarrow$ `null`

IS (NOT) NULL comparison predicate

- checks if values are equal to `null`
  - evaluates to true $\iff$ x is `null`
- `x IS NOT NULL` $\equiv$ `NOT (x IS NULL)`

IS (NOT) NOT DISTINCT comparison predicate

- equivalent to x `<>` y if x and y are non-`null` values
  - x and y both `null` $\Rightarrow$ *false*
  - only one value is `null` $\Rightarrow$ *true*
- `x IS NOT DISTINCT FROM y` $\equiv$ `NOT (x IS DISTINCT FROM y)`

x	y	x<>y	x IS DISTINCT FROM y
1	1	FALSE	FALSE
1	2	TRUE	TRUE
null	1	*null*	TRUE
null	*null*	*null*	FALSE

03-1. CONSTRAINTS

- **unnamed**: name assigned by DBMS
- **named**: name is specified - easier bookkeeping
- all column constraints can be specified as table constraints, except `NOT NULL`
  - table constraints referring to a single column can be writen as column constraints
  - column and table constraints can be combined

```
... id INTEGER NOT NULL,
...
UNIQUE(id)
```

Not-Null Constraints

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

```
CREATE TABLE Employees (
  id    INTEGER NOT NULL, /* unnamed */
  name  VARCHAR(50) CONSTRAINT nn_name NOT NULL, /* named */
  age   INTEGER,
  job   VARCHAR(50),
);
```

Unique Constraints

- **violation** (of a unique constraint defined on attributes A and B):
  - For any two tuples $t_i, t_k \in$ R, $(t_i \cdot A <> t_k \cdot A)$ or $(t_i \cdot B <> t_k \cdot B)$ evaluates to **false**
  - !!! `null` rows will NOT violate unique key constraints

```
-- column constraint
CREATE TABLE Employees (
  id   INTEGER UNIQUE, /* unnamed */
  pid  INTEGER CONSTRAINT u_id UNIQUE, /* named */
  name VARCHAR(50), age INTEGER,
  role VARCHAR(50)
);
```

```
-- table constraint
CREATE TABLE Employees (
  id   INTEGER,
  name VARCHAR(50),
  UNIQUE(id),  /* unnamed */,
  CONTRAINT u_name UNIQUE (name) /* named */
);
```

- unique constraints for **multiple attributes** can only be specified using **table** constraints

```
CREATE TABLE Employees (
  id     INTEGER,
  name   VARCHAR(50),
  UNIQUE (id, name), /* unnamed */
  CONSTRAINT u_allocation UNIQUE (id, name)  /* named */
)
```

Primary Key Constraints

- **prime attributes** → attributes of the primary key
  - cannot be null
- primary key vs `UNIQUE NOT NULL`
  - `UNIQUE NOT NULL` is a candidate key
  - max 1 primary key, but any number of `UNIQUE NOT NULL` constraints
  - FK constraints are only applicable to PKs in referenced table

```
-- PK constraint for one attribute
CREATE TABLE Teams (
  eid INTEGER PRIMARY KEY,
  ...
);
```

```
-- PK constraint for multiple attributes
CREATE TABLE Teams (
  eid INTEGER,
  pname VARCHAR(100),
  PRIMARY KEY (ename, pname), /* unnamed */
  CONSTRAINT pk_alloc PRIMARY KEY (eid, pname) /* named */
);
```

Foreign Key Constraints

- each FK in the referencing relation **must**:
  - appear as a PK in the referenced relation, OR
  - be a `null` value
- referenced columns must be a PK or declared to be **unique**
- `R.sid` → `S.id`: `R.sid` is a FK referencing PK `id` in `S`

```
CREATE TABLE Teams (
  eid INTEGER,
  pname VARCHAR(100),
  hours INTEGER,
  PRIMARY KEY (ename, pname),
  /* Teams.eid -> Employees.id */
  FOREIGN KEY (eid) REFERENCES Employees (id),
  /* Teams.pname -> Projects.name */
  FOREIGN KEY (pname) REFERENCES Projects (name)
);
```

specifications for table changes

- `ON DELETE/UPDATE`: Specify action in case of the violation of a foreign key constraint
  - attempting to delete primary key will throw error if ON DELETE not specified
  - specify behavior when data in referenced table changes
- possible actions:
  - `NO ACTION`: (**default value**) - rejects the delete/update if it violates constraint
    - `ON DELETE NO ACTION` - will raise error if key is referenced elsewhere
  - `RESTRICT`: similar to `NO ACTION`; checks that constraint cannot be deferred
  - `CASCADE`: propagates delete/update to referencing tuples
  - `SET DEFAULT`: updates FKs of referencing tuples to a specified default value
    - !! default value must be a PK in the referenced table !!
    - e.g. `... pid INTEGER DEFAULT 1, ...`
  - `SET NULL`: update FKs of referencing tuples to `null`
    - be careful for primary attributes
    - corresponding column must be allowed to contain `null` values!

```
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 NO
       ACTION ON UPDATE CASCADE
  /* 'NO ACTION' is optional since it's default */
);
```

Check Constraint

- specify that column values must satisfy a boolean expression
- scope: one table, single row
- not a structural integrity constraint

```
-- column constraint
CREATE TABLE Teams (
  eid INTEGER,
  hours INTEGER check (hours > 0), /* unnamed */
  minutes INTEGER constraint positive_hours check (hours > 0)
       /* named */
);
```

```
-- table constraint
CREATE TABLE Teams (
  eid INTEGER,
  ...
  CHECK (hours <= end_year), /* unnamed table */
  CONSTRAINT valid_lifetime CHECK (start_year <= end_year) /*
       named table */
);
```

- `CHECK` constraints can be complex boolean expressions:

```
CREATE TABLE Teams (
  ...
  CHECK (
    (pname = 'Hello' AND hours >= 30)
    OR
    (panme <> 'Hello' AND hours > 0)
  )
);
```

Deferrable Constraints

- default behaviour for constraints: checked immediately at the end of SQL statement execution
  - violation causes statement to be rolled back
- **deferrable constraints**: relaxed constraint checks
  - check will be deferred to the *end* of the transaction
  - available for: `UNIQUE`, `PRIMARY KEY`, `FOREIGN KEY`
- advantages
  - no need to care about order of SQL statements within a transaction
  - allows for cyclic FK constraints
  - performance boost (when constraint checks are bottleneck)
- disadvantages
  - harder to troubleshoot
  - data definition is no longer unambiguous
  - performance penalty when performing queries

syntax

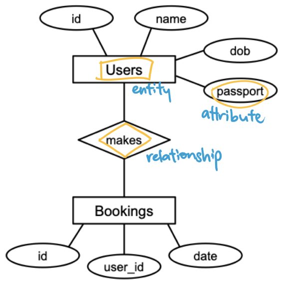
- `NOT DEFERRABLE` - (**default**) immediate check of constraint, cannot be changed
- `DEFERRABLE INITIALLY IMMEDIATE` - immediate check of constraint by default, but can be changed
- `DEFERRABLE INITIALLY DEFERRED` - deferred check of constraint by default, but can be changed

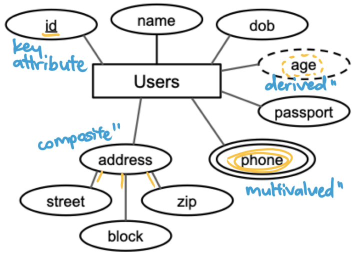
```
CREATE TABLE Employees (
  id INTEGER PRIMARY KEY,
  name VARCHAR(50),
  manager INTEGER,
  CONSTRAINT manager_fkey FOREIGN KEY (manager) REFERENCES
       Employees (id) DEFERRABLE INITIALLY IMMEDIATE
);

BEGIN;
SET CONSTRAINT manager_fkey DEFERRED;
  -- set check of constraint from "immediate" to "deferred"
DELETE FROM Employees WHERE id = 102;
  -- constraint violated but not checked
UPDATE Employees SET manager = 101 WHERE id = 103;
  -- constraint re-established
COMMIT;
```

04. ENTITY RELATIONSHIP MODEL

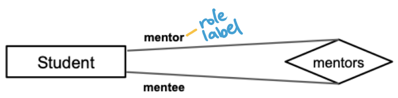
- all data is described in terms of **entities** and their **relationships**
- **entity** → objects that are distinguishable from other objects
  - **entity set** → collection of entities of the same type
- **attribute** → specific information describing an entity
  - **key attribute** → uniquely identifies each entity (underline)
  - **composite attribute** → composed of multiple other attributes (oval of ovals)
  - **multivalued attribute** → may comprise more than one value for a given entity (double-lined oval)
  - **derived attribute** → derived from other attributes (dashed oval)
- **relationship** → association among two or more entities
  - **relationship set** → collection of relationships of the same type
    - may have their own attributes that describe the relationship

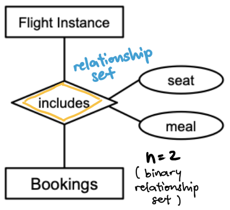




relationship sets

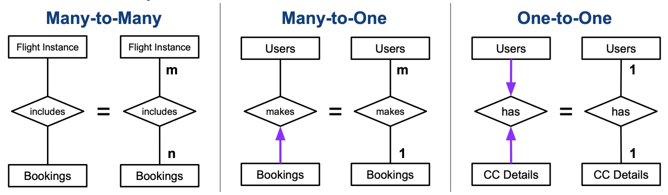
- **role** $\rightarrow$ descriptor of an entity set's participation in a relationship
  - explicit role labels
- **degree** $\rightarrow$ number of entity roles participating in a relationship
  - an n-ary relationship set involves n entity roles (where n is the degree of the relationship set)
  - binary/ternary relationship set
  - general n-ary relation:
    - n participating entity sets $E_1, E_2, \ldots, E_n$
    - k relationship attributes $A_1, A_2, \ldots, A_k$
    - $Key(E_i) \rightarrow$ the attributes of the selected key of entity set E_i





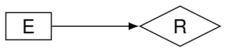
Cardinality Constraints

- describes how often an entity can participate in a relationship **at most**
- 3 basic cardinality constraints:

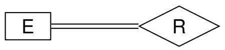


Participation Constraints

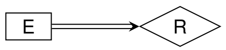
- specifies if an entity has to participate in a relationship (lower bound)
- **partial participation constraint** $\rightarrow$ participation (of an entity in a relationship) is not mandatory (0 or more)
- **total participation constraint** $\rightarrow$ participation is mandatory (1 or more)



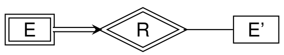
each instance of E participates in **at most one** instance of R



each instance of E participates in **at least one** instance of R



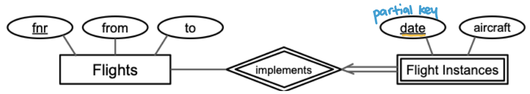
each instance of E participates in **exactly one** instance of R



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

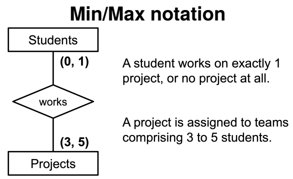
Dependency Constraints

- **weak entity sets** $\rightarrow$ entity set that does not have its own key
  - can only be uniquely identified through the primary key of its **owner entity**
  - existence depends on the existence of its owner entity
  - **partial key** $\rightarrow$ set of attributes of a weak entity set that uniquely identifies a weak entity for a given owner entity
    - identifies the exact instance of a weak entity



- requirements
  1. many-to-one relationship (identifying relationship) from weak entity set to owner entity set
  2. weak entity set must have **total participation** in identifying relationship

Alternative Representations



04-1. RELATIONAL MAPPING

- entity set $\rightarrow$ table
- handling composite/multivalued attributes
  1. convert to a set of single-valued attributes (e.g. phone $\rightarrow$ phone1, phone2)
  2. additional table with FK constraint (e.g. PhoneNumbers with user_id, phone)
  3. convert to one single-valued attribute (e.g. string containing everything)

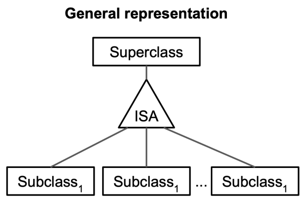
04-2. EXTENDED CONCEPTS

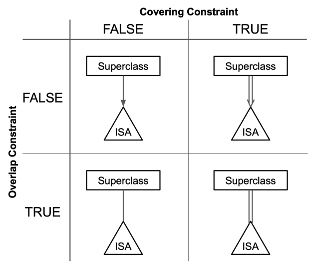
ISA Hierarchy

- "is a" relationship used to model generalisation/specialisation of entity sets
- every entity in a subclass is an entity in its superclass
  - each subclass has specific attributes and/or relationships

constraints

- **overlap contraint** $\rightarrow$ a superclass entity can belong to **multiple** subclasses
- **covering constraint** $\rightarrow$ a superclass entity **has to** belong to a subclass

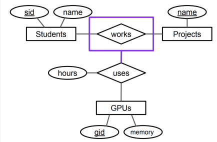




Aggregation

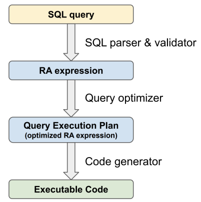
- abstraction that treats relationships as higher-level entities
  - e.g. treating 2 entities + 1 relationship as an entity set

```
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)
);
```



05. SQL (QUERYING A DATABASE)

- **DQL** $\rightarrow$ data query language
- duplicate tuples are allowed!
  - use `DISTINCT` to eliminate duplicates



SELECT DISTINCT $a_1, a_2, \ldots, a_m$
FROM $r_1, r_2, \ldots, r_n$
WHERE c

$\Rightarrow$ corresponds to:

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

SELECT clause

- wildcard `*` - include all attributes
- `expr BETWEEN <lower> AND <upper>` - basic value range conditions

```
SELECT * FROM countries
WHERE (continent = 'Asia' OR continent = 'Europe')
  AND (population BETWEEN 500 AND 600);
```

- `||` - concatenate strings

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

- `SELECT DISTINCT` - remove duplicates
  - tuples (n_1, c_1) and (n_2, c_2) are considered distinct
  $\Longleftrightarrow$ n_1 **IS DISTINCT FROM** $n_2 \vee c_1$ **IS DISTINCT FROM** c_2

WHERE clause

- `IS (NOT) NULL`
  - evaluates to true: $null$ `IS NULL`
  - evaluates false: $null$ `= NULL` (unknown), $null$ `<> NULL`
- `(NOT) LIKE` - pattern matching
  - `_` - match any single character
  - `%` - match any sequence of zero or more characters

SET Operations

- `UNION`, `INTERSECT`, `EXCEPT`
  - will eliminate duplicate tuples from result
- `UNION ALL`, `INTERSECT ALL`, `EXCEPT ALL`
  - will NOT eliminate duplicate tuples from result
- no ordering of tuples

```
(SELECT value FROM R)
UNION
(SELECT value FROM S);
```

JOIN Queries

- `JOIN` - interpreted as `INNER JOIN` by default
- `NATURAL JOIN` - joins based on attribute names
  - identical attribute names can be reinforced with renaming
- `LEFT OUTER JOIN` - same as `LEFT JOIN`
  - keep only dangling tuples: `... WHERE c.country_iso2 IS NULL;`
- complex join queries
- equivalent queries:

```sql
SELECT c.name, n.name
FROM cities AS c, countries AS n
WHERE c.country_iso2 = n.iso2;

SELECT c.name, n.name
FROM cities c INNER JOIN countries n
  ON c.country_iso2 = n.iso2;

SELECT c.name, n.name
FROM cities c JOIN countries n
  ON c.country_iso2 = n.iso2;
```

Subqueries

- **table alias** $\rightarrow$ every subquery has to have a name (mandatory **table alias**) to uniquely identify its attributes
  - **column alias** is optional - `AS optional`
  - must be enclosed in parentheses
- `(NOT) IN` - returns true if *expr* matches **any** subquery row
  - syntax: `expr IN (subquery)`, `expr NOT IN (subquery)`
  - subquery must return *exactly one* column
  - `IN` can typically be replaced with (inner) joins
  - `NOT IN` can typically be replaced with (outer) joins

```sql
SELECT name FROM COUNTRIES
WHERE name IN (SELECT name FROM cities)
   OR name IN ('Singapore', 'Hong_Kong');
```

- `ANY` - returns true if comparison evaluates to true for *at least one* subquery row
  - syntax: `expr op ANY (subquery)`
  - subquery must return *exactly one* column
  - expression is compared to each subquery row using `op`

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

- `ALL` - returns true if comparison evaluates to true for *all* subquery rows
  - syntax: `expr op ALL (subquery)`

```sql
SELECT name, continent, gdp FROM countries c1
WHERE gdp >= ALL(SELECT gdp FROM countries c2
                 WHERE c2.continent = c1.continent);
                 -- c1 from outer query
```

- `EXISTS` - returns true if the subquery returns *at least one* tuple
  - syntax: `EXISTS (subquery)`, `NOT EXISTS (subquery)`
  - `(NOT) EXISTS` subqueries are generally **correlated**
    - uncorrelated $\Rightarrow$ will always give the same result $\Rightarrow$ redundant

```sql
SELECT n.name FROM countries n
WHERE NOT EXISTS (SELECT * FROM cities c
                  where c.country_iso2 = n.iso2);
                  -- n from outer query
```

correlated subquery

- **correlated subquery** $\rightarrow$ relies on value(s) from outer query
- result of subquery depends on value of outer query
  - potential performance issues (slow)
  - potential naming ambiguity - use table aliases
- **scoping rules**
  - a table alias declared in subquery Q can only be used in Q or subqueries nested within Q
  - if the same table alias is declared both in Q and in an outer query (or undeclared), the declaration in Q is applied.
    - aka when unsure, apply the smallest scope ("inner to outer")

scalar subqueries

- **scalar subquery** $\rightarrow$ returns a **single** value (1 row 1 column)
- can be used as an expression in queries

row constructors

- allow subqueries to return more than one attribute/column
  - number of attributes/columns in the row constructor must match that of the subquery
- e.g. find all countries with higher population *or* gdp than France *or* Germany

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

equivalent subqueries

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

Sorting

- `ORDER BY` - sort by attribute(s), `ASC`/`DESC`
- e.g. `ORDER BY n.name ASC, c.population DESC`
  - second criteria only affects result if first criteria has ambiguity

Rank-based Selection

- `LIMIT k` - return the first k tuples of the result table
- `OFFSET i` - specify the position of the "first" tuple to be considered

```sql
-- e.g. find the "second" top 5 countries by GDP per capita
SELECT name, (gdp/population) AS gdp_per_capita FROM countries
ORDER BY gdp_per_capita DESC
OFFSET 5 LIMIT 5;
```

06-1. SQL (AGGREGATION)

- compute a single value from a set of tuples
- NOT allowed in WHERE
- e.g. `MIN()`, `MAX()`, `AVG()`, `COUNT()`, `SUM()`

```sql
SELECT MIN(population) as lowest,
       MAX(population) as highest,
       SUM(population) as world
FROM countries;
```

handling null values

Query	Interpretation
SELECT MIN(A) **FROM** R;	Minimum non-null value in A
SELECT MAX(A) **FROM** R;	Maximum non-null value in A
SELECT AVG(A) **FROM** R;	Average of non-null values in A
SELECT SUM(A) **FROM** R;	Sum of non-null values in A
SELECT COUNT(A) **FROM** R;	Count of non-null values in A
SELECT COUNT(*) **FROM** R;	Count of rows in R
SELECT AVG(DISTINCT A) **FROM** R;	Average of distinct non-null values in A
SELECT SUM(DISTINCT A) **FROM** R;	Sum of distinct non-null values in A
SELECT COUNT(DISTINCT A) **FROM** R;	Count of distinct non-null values in 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*

signatures

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

GROUP BY

- given `GROUP BY` $a_1, a_2, \ldots, a_n$, 2 tuples t and t' belong to the same group if $\forall k \in (1, n)$, `(t.`a_k `IS NOT DISTINCT FROM t'.`a_k`)` evaluates to `TRUE`.
- logical partition of relation into groups based on values for specified attributes
- one result tuple for each group
- if column A_i or table R appears in the SELECT clause, one of the following conditions must hold:
  1. A_i appears in the GROUP BY clause
  2. A_i appears as input of an aggregation function in the SELECT clause
  3. the primary key of R appears in the GROUP BY clause

```sql
-- for each continent, find the lowest, highest and total
      country population and number of countries
SELECT continent,
  MIN(population) AS lowest,
  MAX(population) AS highest,
  SUM(population) AS overall,
  COUNT(*)        AS number_of_countries,
FROM countries
GROUP BY continent;
```

HAVING

- conditions check for each group defined by `GROUP BY` clause
  - cannot be used without a `GROUP BY` clause
- if column A_i of table R apears in the `HAVING` clause, one of the following conditions must hold:
  1. A_i appears in the GROUP BY clause
  2. A_i appears as input of an aggregation function in the HAVING clause
  3. the 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;
```

06-2. SQL (CONDITIONAL EXPRESSION)

CASE

- generic conditional expression, similar to if/else
- two basic ways of formulating CASE expressions:

```sql
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ...
  WHEN condition_n THEN
       result_n
  ELSE result0
END
```

```sql
CASE expression
  WHEN value1 THEN result1
  WHEN value2 THEN result2
  ...
  WHEN condition_n THEN
       result_n
  ELSE result0
END
```

COALESCE

- `COALESCE(val1, val2, ...)` 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
- e.g. `SELECT COALESCE(null, null, 1, null, 2) -> 1`

```
-- find the number of cities for each city type;
-- consider cities with NULL for capital as 'other'
SELECT capital, COUNT(*) AS city_count FROM
  (SELECT COALESCE(capital, 'other') AS capital FROM cities) t
GROUP BY capital;
```

NULLIF

- `NULLIF(val1, val2)` returns NULL if `val1 = val2`; otherwise return `val1`
- common use case: convert special values to NULL

```
-- select min and average GDP
-- from a table where unknown GDP is represented as 0
SELECT MIN(NULLIF(gdp, 0)) AS min_gdp,
       ROUND(AVG(NULLIF(gdp, 0))) AS avg_gdp
FROM countries;
```

06-3. SQL (STRUCTURING QUERIES)

Common Table Expressions (CTEs)

- *temporarily* named query

general syntax:
WITH
C_1 **AS** (Q_1)
C_2 **AS** (Q_2)
. . . , ...,
C_n **AS** (Q_n)

```
WITH CTE_name AS
  -- <CTE BODY>
  (SELECT n.name AS country, ...
   FROM ...
   WHERE ...)
  -- </CTE BODY>
SELECT i.country, ...
FROM CTE_name i    /* CTE usage */
  LEFT OUTER JOIN routes r N i.code = r.to_code
...
```

- general syntax
  - each C_i is the name of a temporary table defined by query 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 relation)
  - query is stored (not the query result) $\Rightarrow$ re-executed whenever it is used
- can be used like normal tables
  - no restriction when used in SELECT statements
  - restrictions when using INSERT/UPDATE/DELETE

```
CREATE VIEW ViewName AS
  SELECT ...
  FROM ...
  WHERE ... ;
```

RECURSIVE QUERIES

- using CTEs and `RECURSIVE`

```
WITH RECURSIVE CTE_name (col_a, col_b, col_c) AS (
  SELECT ... , 0 as counter
  FROM ...  WHERE ...
  UNION ALL
  SELECT ..., cte.counter + 1
  FROM CTE_name cte, ... WHERE ...
  AND cte.counter < 3  /* base case */
)
SELECT DISTINCT counter, ...
FROM CTE_name
ORDER BY counter ASC;
```

UNIVERSAL QUANTIFICATION

- no direct support for universal quantification (e.g. find users who have visited *all* countries)
  - SQL only supports *existential* quantification (EXISTS)
- possible workarounds:

```
SELECT n.iso2
FROM countries n
WHERE NOT EXISTS (SELECT 1
 FROM visited v
 WHERE v.iso2 = n.iso2
AND v.user_id = x);
```

```
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);
```

07. PL/pgSQL

- SQL-based Procedural Language implemented by PostgreSQL

Functions

- `CREATE OR REPLACE` - best practice (just `CREATE` is possible as well)
- `<type>`: all data types in SQL, a tuple, a set of tuples, custom tuples, triggers
  - to return nothing: `RETURNS VOID AS ...`
- main body of function is enclosed within `$$`
- functions are compiled $\rightarrow$ validity is not checked whenever you call the function
  - vs CTE views - validity is checked by the engine every time it's called

```
-- define function
CREATE OR REPLACE FUNCTION function_name(x INT, y INT)
RETURNS CHAR(1) AS $$
  SELECT CASE
    WHEN x >= 70 THEN 'A'
    WHEN y >= 60 THEN 'B'
    ELSE 'C'
  END
$$ LANGUAGE sql;

-- call function
SELECT function_name(65, 67);  /* returns a composite type, tuple */
SELECT * FROM function_name(66, 68);  /* returns a table */
SELECT name, function_name(marks1, marks2) AS grade FROM Scores;
```

returning a set of tuples

- by default, a function will only return the first tuple (similar to using `LIMIT 1`)
- to return a set of tuples: `SETOF`

```
-- returns a set of tuples
CREATE OR REPLACE FUNCTION GradeStudents
  (Grade CHAR(1))
RETURNS SETOF Scores AS $$
  SELECT *
  FROM Scores
  WHERE convert(Mark) = Grade;
$$ LANGUAGE sql;
```

returning a custom tuple

- specify `IN` and `OUT` for input and output parameters

```
-- returns a tuple (Mark, Count)
CREATE OR REPLACE FUNCTION CountGradeStudents
  (IN Grade CHAR(1) OUT Mark CHAR(1), OUT Count INT)
RETURNS RECORD AS $$  /* use SETOF to return all tuples */
  SELECT Grade, COUNT(*)
  FROM Scores
  WHERE convert(Mark) = Grade;
$$ LANGUAGE sql;

-- call the function
SELECT CountGradeStudents('C');
```

Procedures

- no return value

```
-- define a procedure
CREATE OR REPLACE PROCEDURE UpdateMark
  (IN amount INTEGER)
AS $$
  UPDATE Scores SET Mark = Mark + amount;
  ALTER TABLE Scores ADD COLUMN IF NOT EXISTS
    Grade CHAR(1) DEFAULT NULL;
  SELECT * FROM Scores;
$$ LANGUAGE sql;

-- call the procedure
CALL UpdateMark(1);
```

Variables

```
CREATE OR REPLACE FUNCTION splitMarks  /* same for PROCEDURE */
  (IN name1 VARCHAR(20), IN name2 VARCHAR(20), OUT mark1 INT,
      OUT mark2 INT)
RETURNS TABLE(mark1 INT, mark2 INT) AS $$  /* return multiple tuples */
DECLARE
  newmark INT := 0;
BEGIN
  SELECT mark INTO mark1 FROM Scores WHERE name = name1;  /* selects into a mark1 variable */
  SELECT mark INTO mark2 FROM Scores WHERE name = name2;  /* selects into a mark2 variable */

  newmark := (mark1 + mark2) / 2;

  UPDATE Scores SER mark = newmark WHERE name = name1 OR name = name2;
  -- does NOT exit the function:
  RETURN QUERY SELECT mark1, mark2;  /* returns values */
  -- does NOT exit the function:
  RETURN NEXT;  /* returns the defined output parameters */
  -- returns output params and EXITS the function:
  RETURN;  /* optional */
END;
$$ LANGUAGE plpgsql;  /* not sql */
```

Return statements

- without `RETURN`, function will end naturally and return output params
- `RETURN;` - returns output params and **exits** the function
- `RETURN QUERY <SELECT ...>;` - appends queried tuples to output table; does not exit function
- `RETURN NEXT;` - appends output params to output table; does not exit function

Control Flow

Conditionals

```
CREATE OR REPLACE FUNCTION splitMarks  /* same for PROCEDURE */
  (IN name1 VARCHAR(20), IN name2 VARCHAR(20), OUT mark1 INT,
      OUT mark2 INT)
RETURNS TABLE(mark1 INT, mark2 INT) AS $$
DECLARE
  newmark INT := 0;
BEGIN
  -- <SELECT statements />
  newmark := (mark1 + mark2) / 2;
```

```
  IF x > 60 THEN      x := x / 2;
  ELSIF x > 50 THEN   x := x - 20;
  ELSE                x := x - 10;
  END IF;
  -- <UPDATE statements />
  RETURN QUERY SELECT mark1, mark2;  /* returns values */
END;
```

While

```
BEGIN
  -- <SELECT statements />
  x := (mark1 + mark2) / 2;
  WHILE x > 30 LOOP
    x := x / 2;
  END LOOP;
  -- <UPDATE statements />
  RETURN QUERY SELECT mark1, mark2;
END;
```

Exit When

- equivalent to `while True: if (cond): break;`

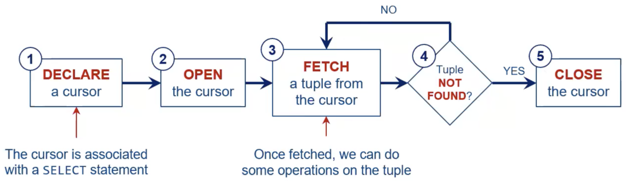
```
BEGIN
  -- <SELECT statements />
  LOOP
    EXIT WHEN x < 30;
    x := x / 2;
  END LOOP;
  -- <UPDATE statements />
  RETURN QUERY SELECT mark1, mark2;
END;
```

Foreach

```
DECLARE
  x INT := 0; d INT; denoms INT[] := ARRAY[1, 2, 3];
BEGIN
  -- <SELECT statements />
  FOREACH d IN ARRAY denoms LOOP
    x := x / d;
  END LOOP;
  -- <UPDATE statements />
  RETURN QUERY SELECT mark1, mark2;
END;
```

Cursor

- allows us to access each individual row returned by a SELECT statement



- e.g. Based on this ranking system of cryptocurrencies, I want to have daily record of first three coins from the TOP 10 cryptocurrencies that are down by more than 5% in the past 7 days and are within 2 ranks apart from each other.

```
CREATE OR REPLACE FUNCTION consCryptosDown
(IN num INT)
RETURNS TABLE(rank INT, sym CHAR(4)) AS $$
DECLARE
  curs CURSOR FOR (SELECT * FROM cryptosRank
                  WHERE changes < -5);
  r1 RECORD;
  r2 RECORD;
BEGIN
  OPEN curs;
  LOOP

  -- move cursor
  FETCH curs INTO r1;       /* loads tuples into r1 */
  EXIT WHEN NOT FOUND;      /* check for end of table */
  FETCH RELATIVE (num-1) FROM curs INTO r2;
  EXIT WHEN NOT FOUND;

  -- compare rows
  IF r2.rank - r1.rank = 2 THEN
    MOVE RELATIVE -(num) FROM curs;
      FOR c IN 1..num LOOP
      FETCH curs INTO r1;
      rank := r1.rank;
      sym := r1.symbol;
      RETURN NEXT;
    END LOOP;
    CLOSE curs;
    RETURN;
  END IF;

  MOVE RELATIVE -(num - 1) FROM curs;

  END LOOP;
  CLOSE curs;
END;
$$ LANGUAGE plpgsql;
```

SUMMARY: RELATIONAL MODEL

Term	Description
attribute	column of a table
domain	set of possible values for an attribute
attribute value	element of a domain
relation schema	set of attributes (with their data types + relation name)
relation	set of tuples
tuple	roles of a table
database schema	set of relation schemas
database	set of relations / tables
key	minimal set of attributes uniquely identifying a tuple in a relation
primary key	selected key (in case of multiple candidate keys)
foreign key	set of attributes that is a key in referenced relation
prime attribute	attribute of a key

CONCEPTUAL EVALUATION OF QUERIES

FROM	Compute cross-product of all tables in FROM clause
WHERE	Filter tuples that evaluate to true on the WHERE condition(s)
GROUP BY	Partition table into groups w.r.t. to the grouping attribute(s)
HAVING	Filter groups that evaluate to true on the HAVING condition(s)
SELECT	Remove all attributes no specified in SELECT clause
ORDER BY	Sort tables based on specified attribute(s)
LIMIT/OFFSET	Filter tuples based on their order in the table