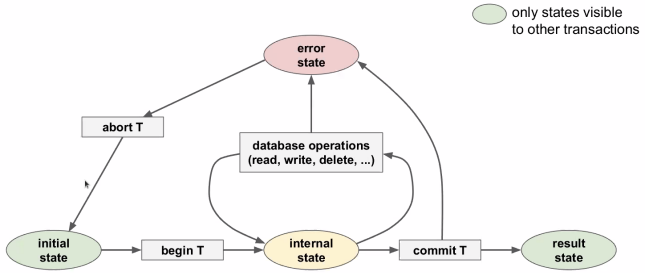
CS2102
AY21/22 SEM 1
github/jovyntls

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

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 $\{(a_1, a_2, \ldots, a_n) \mid a_i \in dom(A_i) \cup \{null\}\}$

01-2. ENSURING 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

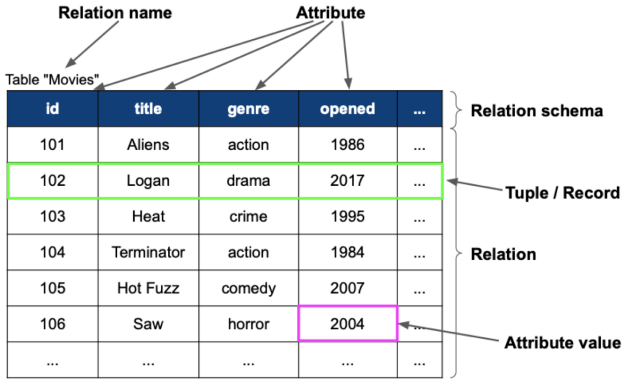
Key Constraints

- **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

01-3. SUMMARY



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 (i.e. rows from a table) 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!
  - i.e. projects all columns of a table specified in list ℓ

Renaming, ρ_ℓ

- $\rho_\ell(R)$ → renames the attributes of a relation R
  R is a relation with 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 that are in both R **or** S
- **intersection** → $R \cap S$ ... all tuples that are in both R **and** S
- **set difference** → $R - S$ ... all the tuples that are 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*
  - BUT R and S do not have to use the same attribute names

CROSS PRODUCT

- **cross product** → combines two relations R and S by forming all pairs of tuples from the two relations
  - 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
    $R \times S = \{(a, b, c, x, y) \mid (a, b, c) \in R, (x, y) \in S\}$
- **size** of cross product $= |R| * |S|$

02-2. JOIN OPERATORS

Inner Joins θ-join

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

θ-join

- the θ-join $R \bowtie_\theta S$ of two relations R and S is defined as

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

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 $R \bowtie S = \pi_\ell(R \bowtie_c \rho_{b_i \leftarrow a_i, \ldots, b_k \leftarrow a_k}(S))$
  - $A = \{a_i, \ldots, a_k\}$ is the set of attributes that R and S have in common
  - $c = ((a_i = b_i) \wedge \cdots \wedge (a_k = b_k))$
  - $\ell =$ 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** $\rightarrow R \bowtie\!\!\!\!\!\!\bowtie_\theta S = R \bowtie_\theta S \cup (dangle(R \bowtie_\theta S) \times \{null(S)\})$
- **right outer join** $\rightarrow R \bowtie\!\!\!\!\!\!\bowtie_\theta S = R \bowtie_\theta S \cup (\{null(R)\} \times dangle(S \bowtie_\theta R))$
- **full outer join** $\rightarrow R \bowtie\!\!\!\!\!\!\bowtie_\theta S = R \bowtie_\theta S \cup (dangle(R \bowtie_\theta S) \times \{null(S)\}) \cup (\{null(R)\} \times dangle(S \bowtie_\theta R))$

Natural Outer Joins

- 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

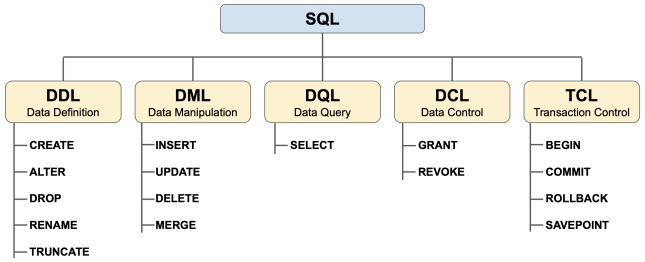
Data Types (psql)

- 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    VARCHAR(50),
age     INTEGER,
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';

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	xy	x IS DISTINCT FROM y
1	1	FALSE	FALSE
1	2	TRUE	TRUE
null	1	*null*	TRUE
null	*null*	*null*	FALSE

03-1. CONSTRAINTS

- **named**: name assigned by DBMS
- **unnamed**: 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

```
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
- (un)named column constraint

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

- (un)named 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 (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 contraint 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

```
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
  - `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 !!
  - `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

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