Database Systems

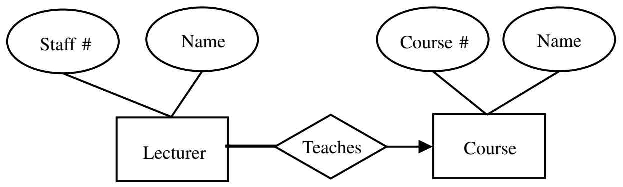
Haeohreum Kim

I. Data Modelling

Data Modelling is a useful technique that is applied to organise database schemas *before* the implementation of them. There are multiple ways to do data modelling, but we generally follow the Object Oriented $\rightarrow$ Record Based or ER Diagram $\rightarrow$ Relational.

A. ER Diagram

An ER diagram is a diagram with well defined sets of rules and principles, which describes relations as objects with attributes and relationships with other objects.



In the above case, one or more lecturers *must* teach a course - and both courses and lecturers have an identify number, as well as a name.

B. Relational and ER $\rightarrow$ Relational mapping

Relational data models reflect SQL tables, and hence, are the last step before we translate into code.

- Strong entities $\rightarrow$ relations
- Weak entities $\rightarrow$ relations with foreign keys to parent entity
- n : m relationships $\rightarrow$ three relations - two for the entities, one for the relationship with foreign keys
- 1 : n relationships $\rightarrow$ two relations for the entities, and the "child" in the relationship should hold a foreign key
- 1 : 1 relationships $\rightarrow$ same as above - but with unique foreign keys to establish 1 : 1
- Multi-valued attributes $\rightarrow$ separate relations with foreign keys to entities
- Sub-classes $\rightarrow$ multiple methods; just use foreign keys

Lecturer

staff num	name	courses

Course

course num	name

II. PostgreSQL

PostgreSQL is a SQL extension language that adds more functionalities, such as the ability to create functions, and use object-oriented features such as inheritance.

A. Syntax

The syntax in PostgreSQL is simple and ordered.

Query

- `SELECT column1, column2, ...`
  `FROM table1, table2, ...;`
  Selects a column of information from a table and returns it. Can use * in column, to return the entire table. Note that you can use the same table twice in `FROM` to do comparative methods, such as:
  ```
  SELECT t1.id
  FROM table t1, table t2
  WHERE t1.id > t2.id
  ```
  Note that this example also shows *aliasing*, where we define different references to the same table as an alias after the declaration. You can also define elements from the same table, and multiple of the same element, to do comparisons between rows.
- `LEFT/RIGHT/INNER/FULL JOIN left_table`
  `ON left_table.id = right_table.id`
  Joins two columns from two tables based on the condition detailed after `ON`. The join statement is appended after a select statement. In the select statement, your `FROM` statement should be the opposing table to the joining table.
- `CROSS JOIN table`
  Cross join can be appended with another join statement (as above), to create a 3 table join in one statement. Cross join statements are essentially full joins, but can be controlled by the following join statements "on" conditions.
- `aggregate(column) OVER (PARTITION BY column)`
  The aggregate function is applied across every single value present within the column. For example, if there are multiple data points for a single column, the partition will apply the aggregate to every single column individually.
- `GROUP BY column`
  When using select statements with `COUNT`, or any operation like `COUNT`, you should specify which columns to collect by - in this case, you are collecting by "column".
- `HAVING condition`
  When using a group by statement, you can create an extra constraint such that only groups that have a specific condition will be included.

- `WITH RECURSIVE R(atrributes)`
  `AS (setup query UNION recursive query)`
  Recursive queries are used for querys that require a recursive behaviour - such as recursively finding children of a parent attribute.

Conditional

- `WHERE condition;`
  Can be appended to many statements (including queries), to force some condition.

-
  ```
  CASE
    WHEN condition THEN return
    WHEN condition THEN return
            ...            ...
  END
  ```
  If style statement that returns or does some action when a condition is first met.
- `COALESCE(val, val, ...)`
  Returns the first non-NULL value - if all values are NULL, returns NULL.
- `NULLIF(value1, value2)`
  Returns null if `value1 == value2` else, returns `value1`.

Constraints

- `DOMAIN attribute CHECK (condition)`
  The domain is a ”type” - the check statement checks if a condition regarding the attribute (or something) is true, and enforces it.
- `LIKE '[0-9]{3}'`
  The like statement is a pattern matching statement. A useful property of pattern matching is shown above, with the use of ranges to enforce a pattern.
- `CONSTRAINT name CHECK (condition)`
  You should give constraints names for clarity, and also reusability across relations when you reference primary keys or other attributes with a constraint.
- `FOREIGN KEY (key) REFERENCES table(key)`
  Enforces referential integrity - or in other words, makes sure that the foreign key is the same as a key in another table.
- `UNIQUE`
  Forces the uniqueness of an attribute in a column
- `NOT NULL`
  Forces the attribute in a column to never be NULL.

Views

Views are ”pointers” to existing relations that allow complex select statements to be truncated and reduced. If views are updated, their related tables are also updated - however if they are dropped, the data is not also deleted on the related table.

1) `CREATE VIEW view AS [SELECT STATEMENT]`
   Creates a unmaterialised view - is not stored in memory. Smaller space footprint, but larger time cost when searching.
2) `CREATE MATERIALIZED VIEW view`
   `AS [SELECT STATEMENT]`
   Creates a materialised view, which is stored as a table in memory. Much faster lookup, but more space used.

Relation management

-
  ```
  CREATE TABLE table (
    DOMAIN column1,
    DOMAIN column2,
    ...
  )
  ```
  Creates a new table
- `DROP TABLE table`
  Deletes a table
- `TRUNCATE table`
  Deletes table information, but retains table
- `ALTER table ADD/RENAME/... COLUMN/ column`
  Adds a new column

Database management

- `createdb database`
  Creates a new database
- `dropdb database`
  Drops a database if it exists
- `psql db -f file.sql`
  Executes a file into a PostgreSQL database.
- `\i file.sql`
  Executes a file into a database when in psql CLI mode.

Logic Abstraction

-
  ```
  SELECT course, student, mark
  FROM (SELECT course, student, mark,
  avg(mark) OVER (PARTITION BY course)
  FROM Enrolments) AS CourseMarksWithAvg
  WHERE mark < avg;
  ```
  An example of using a subquery to create a temporary table, and then using that table to do some logic.

-
  ```
  CREATE VIEW
  CourseMarksWithAvg(course,student,mark,avg)
  AS
  SELECT course, student, mark,
  avg(mark) OVER (PARTITION BY course)
  FROM Enrolments;

  SELECT course, student, mark
  FROM CourseMarksWithAvg
  WHERE mark < avg;
  ```
  An example of using a view to create a temporary table, and then using that table to do some logic.

-
  ```
  WITH CourseMarksWithAvg AS
  (SELECT course, student, mark,
  avg(mark) OVER (PARTITION BY course)
  FROM Enrolments)
  SELECT course, student, mark, avg
  FROM CourseMarksWithAvg
  WHERE mark < avg;
  ```
  An example of using a common table expression to create a temporary table, and then using that table to do some logic.

Set Operations

- `R1 UNION R2`
  Returns all rows from both tables, and removes duplicates.
- `R1 UNION ALL R2`
  Returns all rows from both tables, and retains duplicates.
- `R1 INTERSECT R2`
  Returns rows that are present in both tables.
- `R1 EXCEPT R2`
  Returns rows that are present in the first table, but not the second.
- `select distinct column from table`
  Returns a proper set - such that there are no duplicates.

SQL/pgSQL Functions

-
  ```
  CREATE OR REPLACE FUNCTION
  funcName(arg1 type, arg2 type, ...)
  RETURNS type
  AS $$
   SQL logic
  $$ LANGUAGE sql;
  ```

  Creates an SQL function. Function arguments may be accessed by $1, $2, ... Return type can be defined as `TABLE(name type, name type, ...)`, or any other PostgreSQL type.

-
  ```
  CREATE PROCEDURE
  procName(arg1 type, arg2 type, ...) $$
  SQL logic
  $$ LANGUAGE sql;
  ```

  Procedures don't return any value and rather only serve as to update some table, or do other logic.

-
  ```
  CREATE OR REPLACE FUNCTION
  funcName(arg1 type, arg2 type, ...)
  RETURNS rettype
  AS $$
  DECLARE
    var1 type,
    var2 type, ...
  BEGIN
    PGpGSQL/SQL Logic
  END
  $$ LANGUAGE sql;
  ```
  PLpgSQL functions - some important things:
  - PLpgSQL variables cannot be used in SQL function; use execute format instead
  - `EXECUTE FORMAT('SELECT * FROM %L', variable)`

User defined aggregates

-
  ```
  CREATE AGGREGATE agg(input type)
  (
    sfunc = function for aggregate,
    stype = type of aggregate,
    initcond = initial condition
    finalfunc = finalising function
  )
  ```

Triggers

```
CREATE TRIGGER trigger
AFTER|BEFORE|INSTEAD OF Event1 OR ...
ON table
WHEN condition
FOR EACH ROW|STATEMENT
EXECUTE FUNCTION procedure(args)
```

- AFTER triggers apply a procedure AFTER an event has occurred.
- BEFORE triggers apply a procedure BEFORE an event has occurred.
- INSTEAD OF triggers apply a procedure INSTEAD OF the action the triggered.
- If an exception is raised in any of the above trigger events, the original event is cancelled. This means that even if you run an after trigger on an insertion, you can run error checks and remove the insertion if necessary.
- A function should return a trigger
- Within a function, special variables NEW and OLD can be accessed. In an INSERT trigger, we only have a NEW variable - since we don't have an old variable. In an UPDATE trigger, we have NEW and OLD variables, and in a DELETE, we only have OLD variables.
- RETURN NEW to apply changes in BEFORE and INSTEAD OF triggers for INSERT and UPDATE
- RETURN OLD to access the state of a row before an UPDATE or DELETE operation
- An event can be defined as UPDATE, INSERT, DELETE

III. PsycoPG2

PsycoPG2 is a Python package that allows us to interact with psql databases in a user-friendly way. **Syntax**

- `conn = psycopg2.connect(connect)` connect string should include the following:
  - `dbname`: name of database
  - `user`: user name
  - `password`: password
  - `host`: host of db server
  - `port`: port of db server
- `conn.close()` close the current connection
- `cur = conn.cursor()` is a cursor, which allows us to perform queries/updates on a database. All db actions should be undertaken in the cursor.
- `conn.commit()` commits changes made to the database from the last commit.
- `cur.execute("query", (arguments,))` can perform queries on the database, with arguments denoted by `%s` within the query.
- `cur.fetchall()` fetches all tuples generated by a previous query.
- `cur.fetchone()` fetches one tuple generated by a previous query.
- `cur.fetchmany(nTuples)` fetches n many tuples.

IV. Redundancies

Redundancies can occur in relations that aren't strongly defined, or defined in an appropiate matter. Redundant data means that we have multiple records of the same instance of data. Redundant data can be reduced by using good primary keys - which can be found using functional dependencies.

V. Functional dependencies

Notation and terminology

- Attributes: uppercase letters from start of alphabet (e.g A, B, C)
- Sets of attributes: concatenation of attribute letters (e.g $X = ABCD$)
- Relation schemas: upper case letters, denoting set of all attributes (e.g R)
- Relation instance: lowercase letter responding to schema (e.g $r(R)$)
- Tuples: lower-case letters (e.g $t, t', \ldots$)
- Attributes in tuples: `tuple[attrSet]`

Definition
We say that Y is functionally dependent on X if for any $t, u \in r$, $t[X] = u[X] \implies t[Y] = u[Y]$, where t and u are some tuples in a relation instance. We denote this as $X \to Y$, or that X determines Y.

Inference rules

1) Reflexivity: $X \to X$
2) Augmentation: if a dependency holds, then we can apply new attributes to both sides (e.g $X \to Y \implies XZ \to YZ$)
3) Transitivity: $X \to Y, Y \to Z \implies X \to Z$
4) Additivity: $X \to Y, X \to Z \implies X \to YZ$
5) Projectivity: $X \to YZ \implies X \to Y, X \to Z$
6) Pseudotransitivity: $X \to Y, YZ \to W \implies XZ \to W$

VI. Closures, Normal Forms and Normalisation

A. Closures

Closures can be defined as the functional dependencies derivable from a set of other functional dependencies. Clearly, this becomes quite exponential, due to the recursive nature of the inference rules.

Thus, we define closures based on the set of attributes. We denote the closure of a set of functional dependencies X as X^+, and for some $X = \{X \to Y, Y \to Z\}$, we have that $X^+ = XYZ$. We can determine an appropiate primary key through finding some attribute subset $K \subseteq R$ such that $K^+ = R$.

For example, in $R = ABCD, F = \{A \to B, B \to C, C \to D\}$, $A^+ = ABCD$, and hence, works as a primary key.

B. Minimal covers

A minimal cover is the smallest set of functional dependencies that is a closure. Minimal covers has a few rules. For the minimal cover F_c of some functional dependencies F, we need:

1) $F_c \implies F$
2) All functional dependencies $\in F_c$ are of the form $X \to A$, where A is a single attribute.
3) F_c could not possible made smaller, by deleting a functional dependency or attribute.

A functional dependency d is "redundant" if $(F - \{d\})^+ = F^+$. An attribute is redundant if $(F - \{d\} \cup \{d'\})+ = F^+$, for some set d where d' does not contain a.

Consider the following relation and it's functional dependencies:

$$R = ABC, F = \{A \to BC, B \to C, A \to B, AB \to C\}$$

We can derive the following, through consideration of each "factor":

- Canonical functional dependencies: $\{A \to B, A \to C, B \to C, AB \to C\}$ (projectivity)
- Redundant attributes: $\{A \to B, A \to C, B \to C\}$ (Both A and B in $AB \to C$ are redundant)
- Redundant functional dependencies: $\{A \to B, B \to C\}$. (C is already reachable through B from A)

C. Normal forms

Normal forms dictate the levels of redundancies that can occur. There are 6 levels of normal forms, being 1NF, 2NF, 3NF, BCNF, 4NF, 5NF. The higher the level the stricter the properties are. In 3311, we will largely focus on BCNF and 3NF, which are very similar.

BCNF has the following properties:

- Either $X \to Y$ is trivial ($Y \subseteq X$)
- Or, X is a superkey

3NF has the following properties:

- Either $X \to Y$ is trivial
- Or, X is a superkey
- Or, Y is an attribute from the key X

D. Normalisation

The algorithm for BCNF normalisaiton is as follows, starting with original set of functional dependencies F and the relational attributes A:

- Test $x \to y \in F$, if $x \to y$ does not hold for BCNF, then create two relations - $R(x^+)$ and $R(A - x^+ + x)$. For any $f \in F$ with an element of x^+ on the RHS of the dependency, remove it from the RHS. For any f with an element of x^+ on the LHS of the dependency, remove the dependency.
- Then, continue this with $R(A - x^+ + x)$; testing for each functional dependency.

This can be fairly easily done on paper, and may be expected in the exam.
The algorithm for 3NF Normalisation is as follows:

- Get the functional dependencies, and compute the minimal cover
- Collect the minimal covers using additivity and other inference rules.
- For each collected cover, create a relation containing the attributes.
- If there exists an attribute not represented in the decomposed form, create a new relation with the unincluded attribute(s) in form of the candidate key (since the attribute is unincluded in the functional dependencies, it must be presented in the candidate key).

VII. Relational Algebra

Relational algebra can be seen as the mathematical language for relations (with regards to tables).

- `Sel[condition](Relation)`
  A selection chooses every single column - and hence is less like a SQL select. Full tuples will be selected for tuples that match the condition given. Note: Selects don't return a relation, and hence cannot use joins on them.
- `Proj[A, B, C, ...](Relation)`
  A projection is more like a SQL statement - the A, B, C's are the attributes that the projection selects. Note: projections do not return a relation and hence cannot be used in conjuction with joins, etc.
- `Rename[S(A1, A2, A3, ...)](R)`
  Renames the relation given in R and it's column names, but keeps the same data.
- `intersect, union, -`
  Placed betwen two relations to apply the set operations intersection, union and difference.
- `R x S`
  A cross join - all the tuples in R get a combination of all the tuples in R - ergo, for each tuple in R, create a new tuple join for all tuples in S. If there are commom attributes, rename to be R.A, S.A, etc.
- `R join S`
  Joins two tables (must have a common attribute), where the common attributes are equal.
- `R join[condition] S`
  Joins two tables' tuples, where some condition holds. If there are duplicate attributes, name them R.A, S.A, etc.
- `R Left/Right Outer Join[condition] S`
  Joins two tables' tuples, where some condition holds, as well as the tuples that don't hold on the left/right tuple, filled with NULL values on the joined attributes.
- `R / S`
  For any common attributes in R and S - where the attributes match, remove the attribute, and return the remaining attributes.

VIII. Performance

Database performance can be optimised through the use of **indexation** and avoiding certain SQL query structures. *Indexes* create a fast way to look up specific tables - but can come at a cost. We generally don't want to use indexes on tables that are constantly being updated - but they are useful for constantly read tables.

We generally want to:

- Avoid functions in where/group-by clauses
- Filter first, then join
- Joins are *generally* faster than subquery

A. Performance analysis

Performance can be measured using the following command:

```
EXPLAIN [ANALYSE] Query
```

With analyse, it runs the query and gives us a real runtime. Without it, the DBMS estimates the time taken.

IX. Transactions and concurrency

DBMS's, are in reality **shared** and **unstable**. However, we want to provide for the users, an API that is **unshared** and **stable**. Thereby, we want the databases to be safe from concurrent *transactions.*

A *transaction* is:

- an atomic "unit of work" in an application
- which may require multiple database changes

To maintain integrity of the data, transactions must be atomic, consistent, isolated and durable.

A. Transaction scheduling

When we talk about database transactions, we consider:

- READ - transfer data item from database to memory
- WRITE - transfer data item from memory to database
- BEGIN - start a transaction
- COMMIT - sucessfully complete a transaction
- ABORT - fail a transaction

We generally only really consider read and write. To read an element x, we denote `R(x)`, and to write an element x, we denote `W(x)`. We use these notations to consider transaction schedules to consider if concurrent transactions are valid and don't conflict with eachother. We express schedules like below:

```
T1: R(X) W(X)    R(X)    W(Y)
T2:            R(X)    W(X)
```

B. Conflict serialisability

A serial schedule is one where the transactions are not concurrent - where all the work for one transaction is finished before beginning another. We want to test if a concurrent schedule is serialisable (aka, no concurrency), so that we can test if the above conditions hold. There are two types of serialisability, conflict and view. Conflict serialisability can be found by using a graph representation.

Consider some schedule:

```
R(A)                         R(C) W(A) W(C)
            R(B) R(C)   W(C)
   R(B) R(A)     W(B)
```

Here, we have the following conflicts:

- `T3.R(A) -> T1.W(A)`
- `T2.R(B) -> T3.W(B)`
- `T2.W(C) -> T1.R(C), T1.W(C)`

This graph does not contain any cycles for the vertices T1, T2 and T3, and thereby, is conflict serialisable.

C. View serialisability

- The same transaction reads the same initial value
- The intermediary transactions read the same values
- The final state is the same - ergo, the same transaction performs the final write.

```
T1: R(A) W(A)      R(B)    W(B)
T2:           R(A)    W(A)    R(B) W(B)

T1: R(A)W(A)R(B)W(B)
T2:                 R(A)W(A)R(B)W(B)

T1:                 R(A)W(A)R(B)W(B)
T2: R(A)W(A)R(B)W(B)
```

In the concurrent schedule:

- T1 reads initial, T2 reads T1's write, T2 writes final.

In the two combinations:

- T1 reads initial, T2 reads T1's write, T2 writes final.