Relational Design Theory

NOTE: A candidate key is a set of attributes such that it's closure covers all of the attributes in a relation.

Inference Rules

1. Reflexivity: `X -> X`
2. Augmentation: `X -> Y => XZ -> YZ`
3. Transitivity: `X -> Y, Y -> Z => X -> Z`
4. Additivity: `X -> Y, X -> Z => X -> YZ`
5. Projectivity: `X -> YZ => X -> Y, X -> Z`
6. Pseudo-transitivity: `X -> Y, YZ -> W => XZ -> W`

Closures

Closures are the set of functional dependencies you can derive from a single functional dependency. Since this can get very big, we denote a functional dependency by the *attributes* we can derive. For example, if we have:

```
A -> B, B -> C, C -> D
```

We have `A+ = ABCD`

Minimal Covers

A minimal cover F_c for some set of functional dependencies F, is the smallest set of functional dependencies that satisfy:

- $F_c \equiv F$
- All functional dependencies are canonical
- F_c is not reducible

A functional dependency f is redundant if $(F - \{d\} \cup \{d'\})^+ = F^+$.

An attribute a is redundant if $(F - \{a\} \cup \{a'\})^+ = F^+$. Basically, if the FD is valid when a is removed, it's redundant.

Normal Forms

Boyce-Codd Normal Form

The Boyce-Codd Normal Form (BCNF) is defined, such that for all functional dependencies $a \to b$:

1. $b = a$, thus, is trivial.
2. a is a candidate key.

3rd Normal Form

The 3rd Normal Form (3NF) is less strict then BCNF. It is defined such that for all functional dependencies $a \to b$:

1. $b = a$, thus, is trivial.
2. a is a candidate key
3. b is consisted of attribute(s) from the candidate key

Decomposition

BCNF Decomposition

For a relation R and some functional dependency f:

- Check if f is BCNF compliant.
- If f is not BCNF compliant, partition into two tables, R(f+) and R(R – f+ + f)
- Remove any functional dependencies in R(R – f+ + f) with f+ in the LHS, and remove any attributes in f+ on the RHS.
- Continue until all relations are BCNF compliant.

3NF Decomposition

For a relation R and it's functional dependencies f:

- Compute the minimal cover f_c
- Combine the functional dependencies in f_c, using the inference rules.
- Create relations for each functional dependency, plus a relation of the candidate key if there is a missing attribute.

Relational Algebra

- Sel[condition](Relation/Set)
- Proj[attributes](Relation/Set)
- Rename[attrs](Relation)

R Union S: Takes the union of the two relations/sets.
R Intersection S: Takes the intersection of the two relations/sets.
R Difference S: Removes common attributes between the left-hand and right-hand relation from the left hand relation.
R Division S: Returns the remaining columns of the left-hand relation after the removal of common columns where the attributes match. Where the attributes don't match, the tuples are removed entirely.
R Join[conditions] S: Joins R and S on their conditions – if no conditions are given, it assumes there are common attributes between R and S.
R Left/Right Outer Join[conditions] S: Joins R and S on their conditions AND includes non-conforming tuples from the left or right relations.
R × S: Creates very possible tuple combination across R and S. There will be |R| × |S| amount of tuples.

Performance and Transactions

Performance

Indexes

Indexes provide faster lookup times in a database. They are good when attributes are used frequently, or in expensive queries.

Tuning/Analysis

Using the command `EXPLAIN [ANALYZE] Query`, you are able to analyse theoretical or real (using the analyze keyword), execution time of the query.

Transactions

Transactions are an atomic operation on a database, which may require multiple database changes. We want transactions to be Atomic, Consistent, Isolated and Durable.

Transaction Schedules

Transaction schedules detail how a "thread" might act on some database(s). We generally only care about two operations, read (denoted by R(X) on some relation X), and write (denoted by W(X)).

We are interested in ensuring stability and consistency in concurrent schedules.

Conflict Serialisability

Any sequence that is not read to read, is not consistent. Hence, we use a graph that maps inconsistent schedules, to see if we can serialise the schedule. For example:

```
T1: R(A)                         R(C) W(A) W(C)
T2:               R(B)R(C)   W(C)
T3:     R(B) R(A)       W(B)

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

There is no cycle in the graph representation. Thus,
conflict serializable.
```

View Serialisability

Serialise the concurrent threads into just serial transaction threads. Check every combination of serial transactions.

Check for each variable:

- Who does initial read?
- Who does initial write?

If this is the same in the serial transaction as the concurrent one, then view serializable.

PostgreSQL, PLpgSQL and PsycoPG2

Constraints

Constraints are held after an attribute, using the keyword CHECK. For example:

```
attr int CHECK (attr LIKE '[0-9]{5}')
```

Useful aggregates, functions and features

- **COALESCE(variables)**: Returns the first non-null value; if all values are null, returns null. Very useful when we might have null values, that we want to replace with something more sensible like 0, or ' '.
- **STRING_AGG([DISTINCT] attribute, separator [ORDER BY])**: Aggregates strings by some separator that is given. Very useful.
- **MAX/MIN(attribute)**: Finds the maximum/minimum attribute value that exists within the column.
- **TRUNC(attribute)**: Truncates the attribute towards zero.
- **SUBSTR(string, position, extract)**: Extracts a given number of characters (extract) starting from a position, from a given string.
- **ROUND/FLOOR/CEIL(number)**: Round/Floors/Ceils the attribute
- **UPPER/LOWER(string)**: Returns given string as upper case or lower case

- **E'\n'**: E infront of a string escapes a character – here, a new line will be printed.
- **POSITION(substring IN string)**: Finds the first position of a string
- **string LIKE [%]string**: Pattern matching. If you include a % in front of the comparator, it checks if the word ends in the comparator.

Case statements and PLpgSQL control structures

Case statements

Case statements can be supremely useful for select statements that have some branching aspect. Case statement returns can be used as a column. Case statements follow the syntax:

```
CASE
        WHEN condition THEN return
        …
        ELSE …
END;
```

Control structures

```
IF condition THEN
…
ELSIF condition THEN
…
ELSE
…
END IF;

FOR element IN [REVERSE] set/expression LOOP
…
END LOOP;

WHILE condition LOOP;
…
END LOOP;
```

Function definitions

PLpgSQL

```
create or replace f(x)
returns return_type
as $$
declare
…
begin
…
end
…
$$ language plpgsql;
```

SQL

```
create or replace f(x)
returns return_type
as $$
…
$$ language sql;
```

Return Semantics

PLpgSQL

- RETURN NEXT lets you store a return result; you must include a RETURN at the end to return the dump
- RETURN QUERY returns the results of a query; you must include a RETURN at the end to return the query
- RETURN is a normal return.

SQL

- A SELECT statement serve as the return.

Custom Types

```
CREATE TYPE type_name AS type + constraints…
```

Can be useful if you want to return a set of a custom type.

Variables and arguments semantics

- You can access *function parameters* using $1, $2, …, on both PLpgSQL and SQL functions.
- You can *sometimes* use parameter and variable names in PLpgSQL. I can't exactly pinpoint when you can't; but when you aren't allowed to, you can use:

```
EXECUTE FORMAT('SELECT * FROM %L', variable)
```

- Overall, the variables/parameter system seems a little fiddley, so work around it by using execute format where necessary, etc.

Aggregates

```
CREATE OR REPLACE aggregate(type) (
        stype = type of aggregation,
        sfunc = initial function,
        finalfunc = finalising function,
        initcond = initial value
);
```

You generally want to keep track of some state in your stype/your overall answer.

Triggers

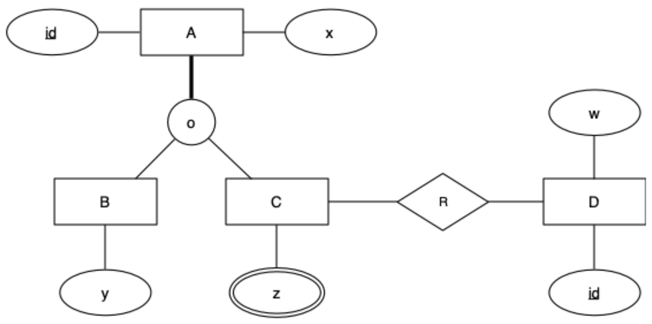
```
CREATE TRIGGER trigger
AFTER|BEFORE|INSTEAD OF Event OR …
ON table
WHEN condition
FOR EACH ROW|STATEMENT
EXECUTE FUNCTION procedure(args)
```

- Events include UPDATE, DELETE, INSERT
- With INSERT we have access to NEW variable, with UPDATE we have access to NEW and OLD, with DELETE we only have OLD variables.

PsycoPG2

- `conn = psycopg2.connect(dbname)`
- `conn.close()`
- `cur = conn.cursor()`
- `conn.commit()`
- `cur.execute('query', (arguments))`
- `cur.fetchall()`
- `cur.fetchone()`
- `cur.fetchmany(nTuples)`

ER Mapping



```
CREATE TABLE A (
        id SERIAL PRIMARY KEY,
        x text
);
CREATE TABLE B (
        a PRIMARY KEY References A(id),
        y text
);
CREATE TABLE C (
        a PRIMARY KEY References A(id),
);
CREATE TABLE Z (
        c References C(id),
        z text,
        PRIMARY KEY (c, z)
);
CREATE TABLE D (
        id SERIAL PRIMARY KEY,
        w text
);
CREATE TABLE R (
        c References C(a),
        d References D(id)
);
```

In an OOP mapping, B and C "inherit" from A – thus, B and C would also have x as an attribute.

In a single-table mapping, you would fuse B and C into A (since they are properties of A). R would now reference A, since C exists in A.