CS2102 Cheatsheet

SQL Data Types

boolean, integer, float8, numeric, numeric(p, s), char(n), varchar(n), text, date, timestamp

Constraint Names

bDate date constraint bdate check (bdate is not null) or constraint obj_pri_key primary key (name, day, hour)

Insert: insert into Students (name, studentId) values ('Bob', 67890), ('Carol', 11122);

Delete: delete from Students where dept = 'Maths';

Update: update Accounts set balance = balance + 500, name = 'Alice' where accountId = 12345;

Foreign Key Constraints Violations

- *No Action*: Rejects action if it violates constraint (default).
- *Restrict:* Same as No Action but constraint checking is not deferred.
- *Cascade*: Propagates delete/update to referencing tuples.
- *Set Null:* Updates foreign keys to null.
- *Set Default:* Updates foreign keys to some default value. We will need to specify this value at the referencing column, and it must exist

Transactions

Starts with begin; and ends with commit; or rollback;. Can contain multiple SQL statements.

Deferrable Constraints

unique, PK and FK can be deferred using deferrable initially deferred or deferrable initially immediate We can change this using set constraints, e.g. set constraints FK deferred; or set constraints FK immediate; (retroactive)

Set Operations

union, intersect and except are same as ∪, ∩, –. Eliminates duplicates. union all, intersect all and except all preserve duplicate records.

Scalar Subqueries

A subquery that returns at most one tuple with one column. If empty, then null is returned. It can be then used as a scalar expression.

Order By: order by area asc, price desc; order by area, price desc; **Limit:** limit 3: Show top 3.

Offset: offset 3: Show 4th and onwards.

Aggregate Functions

min(A), max(A), avg(A), sum(A), count(A) (non-null), count(*) (all rows, may be null), avg(distinct A) (non-null), sum(distinct A) (non-null), count(distinct A) (non-null).

- Performed on an empty relation OR nulls, all functions except count will return null. count(R) will return 0, count(*) return n
- Can be used in select, having and order by.
- If a select clause contains an aggregated function and there is no group by clause, then the select clause must not contain any column that is not in an aggregated expression, i.e. all or nothing.
- To use agg in ORDER BY, agg need appear in GROUP BY

Group By

Conditions for a group by clause on relation R:

- Output column A must appear in the group by clause, OR
- A appears in an aggregated expression in the select clause.
- The primary key of R appears in the group by clause.

Having

- Output column A must appear in the group by clause, OR
- A appears in an aggregated expression in the having clause.
- The primary key of R appears in the group by clause.

Conceptual Evaluation of Queries

select
from
where
group by
having
order by
offset
limit

1. Compute cross-product of tables in from-list
2. Select tuples that evaluate to true for the where-condition
3. Partition selected tuples into groups using groupby-list
4. Select the groups that evaluate to true for the having-condition
5. For each group, generate output tuple based on select-list
6. Remove duplicate output tuples because of distinct
7. Sort the output tuples based on orderby-list
8. Offset-specification then limit-specification

Coalesce: select coalesce(third, second, first) as result Returns the first non-null value in its arguments.

Nullif: select name, nullif(result, 'absent') as status Returns null if result is equal to 'absent', otherwise result.

Like / Similar To: where cname like '___%e';

Underscore matches a single character

- % matches any sequence of 0 or more characters
- More complex regex will need similar to.

Relational Algebra

A *query* is composed of a collection of operators called *relational operators*. Relations are *closed* under relational operators.

Operator precedence: (), op, not, and, or → op ∈ {=,<>,<,≤,>,≥}

Null vals: Result of a comparison operation involving null value is unknown Result of an arithmetic operation involving null value is null

x	y	x AND y	x OR y	NOT x
FALSE	FALSE	FALSE	FALSE	
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	TRUE	FALSE	TRUE	
UNKNOWN	FALSE	FALSE	UNKNOWN	
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	TRUE	
TRUE	FALSE	FALSE	TRUE	
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	TRUE	TRUE	TRUE	

Select:

σprice<20(Sells) → will output the rows only if eval to TRUE (not unknown)

Projection:

πrname,pizza(Sells) → outputs the columns and remove dupes

Rename:

ρarea:region,rname:sname(Restaurants) → changes the column header from oldname:newname, order doesn't matter

Union compatible:

same no of attributes (columns), each attribute same domains (datatype)

Cross-Product:

πcname(σarea='Central' (Customers)) × πrname(σarea='Central' (Restaurants)). The output can be smaller cardinality

Joins:

inner c, natural , left outer →c, right outer ←c, full outer ↔c, natural left/right/full outer

Outer joins:

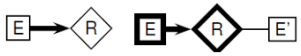
preserves dangling of the left/right/both tables, not associative with inner joins

Relationship Constraints

Key Constraint: Each instance of *E* can participate in *at most one* instance of *R*. Represented by an arrow. Allows for *one-to-many* if one entity has an constraint but the other doesn't, or *one-to-one* if both entities have the constraint

Total Participation Constraint: Each instance of *E* must participate in *at least one* instance of *R*. Represented by a double line.
A single line is a *partial participation constraint*, i.e. 0 or more.

Key & Total Participation Constraint: Each instance of *E* participates in exactly one instance of *R*. Represented by double line arrow.



Weak Entity Set: *E* is a weak entity set with identifying owner *E'* & identifying relationship set *R*. *E* only has partial key and requires the PK of E' be uniquely identified.

Database Design (aka converting to CREATE TABLE)

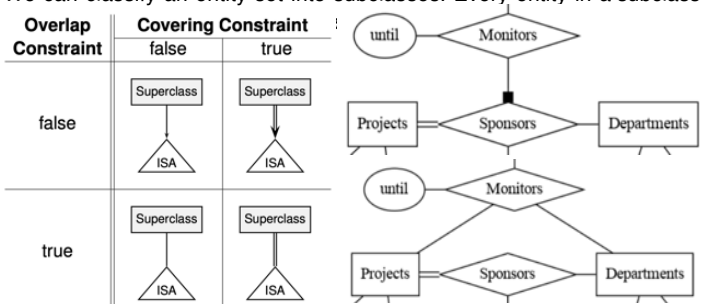
- We can represent primary key. **Cannot** represent unique, not null.
- Without constraints, we generally represent relationship sets as tables with foreign keys that form part of its primary key, i.e. association class.
- With constraints, we can choose to represent them as a separate table or as an attribute in an entity set's table (the one that's many in many-to-one, or any in an one-to-one).
- Use separate tables for key constraint only, and combined for key + total constraints (if we use separate for key + total, need Trigger to enforce)
- To represent roles for the same entity, need a check that they not same Weak entity set & its identifying relationship set can be represented as a single relation, which has a foreign key that **deletes on cascade** and forms part of its primary key.

Aggregation (put black square at r/s to be treated as entity)

When a relation between two entities is treated as an entity. The two diagrams below are not the same, since the bottom does not enforce a relationship between Projects and Departments.

ISA Hierarchies

We can classify an entity set into subclasses. Every entity in a subclass



- *Overlap Constraint*: Can an entity belong to multiple subclasses?
- *Covering Constraint*: Does an entity in a superclass have to belong to some subclass?

We can create a relation per subclass/superclass, with the subclass' tables having a foreign key referencing the superclass' table, along with additional attributes. Need ON CASCADE DETETE

SQL Functions

```
create or replace function a(x int) returns char(1) as $$
    -- select a single character using x
$$ language sql;
returns R
returns setof R
d(out x int, out y int) returns record as $$ -- called
using SELECT d(); returns single tuple
e(out x int, out y int) returns setof record as $$
f() returns table(x int, y int) as $$ -- same as e
```

You can use inout for the parameters if the input "schema" and the output "schema" share columns.

Procedures

create or replace procedure g(x int) as $$ -- CALL g(x);

Variables and Control Structures

```
as $$
declare
   temp_val integer;
begin
  -- function body
end;
$$ language plpgsql;
loop
    exit when condition;
    statements;
end loop;
```

```
if condition1 then
       statement1;
elseif condition2 then
    statement2;
else
    else-statement;
end if;
```

Returning custom tuples

```
returns table(mark1 int mark2 int) as $$
 -- function body
return query select mark1, mark2;
return next -- both are the same, DOES NOT EXIT FUNC
```

Cursor

```
declare
   curs cursor for (select * from R order by A desc);
   r record; -- used to store cursor outputs
   begin
   open curs;
   loop
      fetch curs into r;
      exit when not found; -- terminating condition
     return next; -- insert tuple to func output
end loop;
close curs;
end;
```

Fetch types

```
- fetch cur into r -- will fetch and move curs down
- fetch prior from cur into r; -- fetch the prev row
- fetch first from cur into r;
- fetch last from cur into r;
- fetch absolute 3 from cur into r; -- fetch 3rd tuple
```

SQL Injection

Normally, can use SQL syntax like `--` , `' '` and ;

By using functions or procedure every input is compiled and treated as string, cant do injection attack. Sanitize input by disallowing special chars

Triggers

```
create or replace function f_name() returns trigger as $$
create trigger t_name [before / after]
[insert/update/delete] on Table for each [row / statement]
execute function f_name();
```

Order of triggers: before statement, before row, after row, after statement

Special Values in Triggers

- TG_OP: Operation that activates trigger, i.e. 'INSERT', 'UPDATE', 'DELETE'
- TG_TABLE_NAME: Name of table causing the invocation
- OLD: Old tuple being updated or deleted. **null** for insertion.
- NEW: New tuple being inserted or updated. **null** for deletion.

Instead Of Triggers (Only Row-Level)

You can also define instead of [insert / update / delete] triggers for views (only), e.g. to update the actual tables instead.

Return Values for Row-Level Triggers

- before insert: non-null tuple t → t will be inserted, null → no insertion
- before update: non-null tuple t → t will be the updated tuple, null → no update
- before delete: non-null tuple t → deletion happens, null → no deletion
- after [insert/update/delete]: return value **does not matter**
- instead of: non-null tuple t → proceed, null → ignore operations on current row

If a BEFORE FOR EACH ROW trigger returns null, then all subsequent triggers on the same row are ignored.

Statement-Level Triggers

- Return values are ignored. Raise exceptions if to ignore operations.
- Only can use before and after triggers, not instead of

Trigger Condition

Example: `for each row when (NEW.Name = 'Sherman Ho') execute`

No SELECT in WHEN(cond), no OLD in WHEN(cond) for INSERT, no NEW in WHEN(cond) for DELETE, no WHEN(cond) for INSTEAD OF.

Deferred Triggers

`create constraint trigger t_name after … on R deferrable initially [deferred / immediate] for each row …`

Only works for AFTER and FOR EACH ROW.

If initially immediate, we can change on the fly

`begin transaction; set constraints t_name deferred; --cont here`

Functional Dependencies

A → B means A decides B, i.e. if two rows have the same A, then they have the same B.

Armstrong's Axioms

- Reflexivity: ABC → A (set to subset)
- Augmentation: If A → B, then AC → BC for any C
- Transitivity: If A → B and B → C then A → C

Extended Axioms

- Decomposition: If A → BC then A → B and A → C
- Union: If A → B and A → C then A → BC

Closures

$\{A\}^+$ denotes the set of attributes that can be directly or indirectly decided by A, also called the closure of A.

To compute: start with {A}; for all FDs such that the LHS can be found in the current closure, put the RHS into the closure as well; repeat until no more new attributes can be added.

To prove X → Y, we just need to show $\{X\}^+$ contains Y. Opposite is true.

Keys, Superkeys and Prime Attributes

Superkey: A set of attributes in a table that decides all other attributes.

Key: A superkey that is minimal.

Prime attribute: Attribute that appears in a key.

To find keys of T: consider all subsets of T; derive the closure of each subset; identify the superkeys; identify the keys.

Shortcut: Check smaller attribute sets first.

Shortcut: If attribute does not appear in RHS of any FD, it must be in key.

Non-Trivial and Decomposed FDs (NTD)

Decomposed FD: RHS only has one attribute.

Non-trivial and decomposed (NTD): RHS does not appear in LHS.

To find such FDs, simply compute all closures, remove trivial attributes, then separate them into decomposed FDs.

Boyce-Codd Normal Form (BCNF)

Normal form: Definition of minimum requirements in terms of redundancy.

A table R is in BCNF if every NTD has a **superkey** as its LHS.

To check if R is in BCNF: compute all closures; find the keys from the closures; derive NTD FDs from the closures; check the BCNF requirement.

Shortcut: Check if there's a closure that contains "more but not all", i.e. the RHS has more attributes than the LHS, but does not contain all attributes.

BCNF Decomposition

Find any "more but not all" closure $\{X\}^+$, then decompose R into R_1 and R_2:

- R_1 contains all attributes in $\{X\}^+$ (i.e. RHS) and
- R_2 contains all attributes in X and attributes not in $\{X\}^+$ (i.e. LHS + remaining attributes).

Then, project the closures from R onto R_1 or R_2, then repeat the process.

To project: enumerate all attribute subsets in R_1 (WLOG); derive their closures on R; remove irrelevant attributes.

If a table only has two attributes, then it **must** be in BCNF.

BCNF Properties

- No update or deletion or insertion anomalies, small redundancy and you can always reconstruct the original table.
- A **lossless join** is guaranteed whenever the common attributes in R_1 and R_2 constitute a superkey of R_1 or R_2.
- But it may not guarantee **dependency preservation**.

Dependency Preservation

Let S be the given set of FDs on the original table, and S' be the set of FDs on the decomposed tables.

A decomposition preserves all FDs **iff** S and S' are equivalent, i.e. every FD in S' can be derived from S, and vice versa (calc closures for both ways) Preserving FDs allow us to prevent inappropriate updates.

3NF

A table satisfies 3NF iff for every NTD, either the (for each table):

i) LHS is a superkey or ii) the RHS is a prime attribute.

satisfying BCNF → satisfying 3NF. Violating 3NF → violating BCNF

Shortcut: "more but not all" then check if the more is a prime attribute

3NF Decomposition (a single split into multiple tables)

1. Derive the minimal basis of the FDs (see below);
2. Combine the FDs whose LHS are same, i.e. A → B, A → C == A → BC;
3. Create a table for each remaining FD, e.g. R1(A, B, C);
4. If none of the tables contain key of original table R, create table w/ key
5. If there is any redundant table (is subset of another table), remove it

Minimal Basis / Cover (MB)

We simplify a set S of FDs using the four conditions:

1. Every FD in the MB can be derived from S and vice versa.
2. Every FD in the MB is a NTD.
3. No FD is redundant, i.e. can be derived from other FDs in the MB.
4. For each FD, none of the LHS attributes is redundant, i.e. if we remove it, it cannot be derived from the original set of FDs.

Algorithm for MB

1. Transform the FDs so that each RHS only contains one attribute;
2. remove redundant attributes on the LHS of each FD, if LHS has >1 elem, check if any can be removed
3. remove redundant FDs, check if FD can be implied by other FD. Once we remove an FD, future FDs only check remaining (not removed) FDs