CS2102 CheatSheet

by Zachary Chua

Transactions

Abstraction for logical unit of work

ACID properties

Atomicity: Either all effects of transaction are reflected or none

Consistency: execution of transaction in isolation preserves DB consistency

Isolation: execution of transaction is isolated from effects of other concurrent transaction executions

Durability: effects of commited transaction persists even if system failure

Relational Algebra

Selection: σ_c

$\sigma_c(R)$: selects tuples from relations R that satisfies the selection condition c

Result of comparison operation involving **null** is unknown

Result of arithmetic operation involving **null** is **null**

Projection: π_l

π_l projects attribute given by a list l of attributes from relation R

Duplicate records are removed in output relation

Renaming: ρ_l

ρ_l renames attributes in R based on list of attribute renamings l

l: list of attribute renamings $a_i : b_i$ (renames attirbute a_i to b_i)

Set Operators

Set Operators require input relations to be union compatible

Schema of output of R op S is **identical to schema of R**

Union Compatibility:

1. have same number of attributes, and
2. correesponding attributes have same domains

Dont't need to have same attribute names

Cross Product, $\times$: cartesian product

Inner Join, $\bowtie_c$

$R \bowtie_c S = \sigma_c(R \times S)$

Might have to involve renaming attributes if same name

Natural Join, $\bowtie$

$R \bowtie S = \pi_l(R \bowtie_c \rho_{a1:b1,\ldots,an:bn}(S))$ where,

A = common attributes

$c = (a1 = b1)$ and ... and $(an = bn)$

l = attributes in A, then attributes in R - A, then attributes in S - A

natural join removes duplicate columns, joins on attributes w same name

Dangling Tuples - tuple in join operand that does not take part in join operation

Preserved using outer joins

Natural outer joins: no need state join condition, remove dup column

ER Model

```
create table Profs (
    pid   char(7),
    name  char(30),
    room  char(6),
    primary key (pid)
);
```

```
create table Students (
    sid   char(20),
    name  char(30),
    dob   date,
    pid   char(7),
    since date,
    primary key (sid),
    foreign key (pid) references Profs
);
```

Key from Student → Supervises (alternative: separate Supervises table)

```
create table Profs (
    pid   char(7),
    name  char(30),
    room  char(6),
    primary key (pid)
);
```

```
create table Students (
    sid   char(20),
    name  char(30),
    dob   date,
    pid   char(7) not null,
    since date,
    primary key (sid),
    foreign key (pid) references Profs
);
```

Key and Total Constraint from Students to Supervises R/S

Separate supervises table needs triggers to enforce Total participation

```
create table Books (
    isbn   char(30),
    title  char(50),
    author char(60),
    primary key (isbn)
);
```

```
create table Chapters (
    number integer,
    title  char(50),
    isbn   char(30),
    primary key (isbn, number),
    foreign key (isbn) references Books
        on delete cascade
);
```

Coexistence of weak

Weak Entity Relationship from Chapters to Books

ISA hierarchies:

1. Overlap Constraint: Can entity belong to multiple subclasses?
   yes: undirected edge from superclass to triangle, **no**: Directed edge
2. Covering Constraint: Does entity have to be one of the subclasses?
   yes: thick edge, **no**: thin edge

SQL

Foreign Key Constraints

Delete/ Update tuple may violate FK constraint, specify behaviour

1. NO ACTION: rejects if violates FK constraint
2. Restrict: similar to NO ACTION but constraint check cannot be deferred
3. CASCADE: propagate delete or update to referenced tuple
4. SET DEFAULT: set to default value (must exist in referenced table)
5. SET NULL

Syntax: `on delete / update cascade`

Deferrable Constraints

Deferrable Constraints: unique, PK, FK

1. `deferrable initially deferred`: checked at end of transaction
2. `deferrable initially immediate`: checked at end of statement (default)

Set Operations

`union` $\cup$, `intersect` $\cap$, `except` $-$

Note: they eliminate duplicate records (to retain use all eg. `union all`)

Note: `intersect` has higher precedence than `union` and `except`

Subquery Expressions

`exists, in, any / some, all`

- evaluates inner query for every tuple in table from outer query

Note: For `ALL` if one v_j is **null** then op evaluates to **null**, entire result becomes **null**

Aggregate Functions - `min, max, count, sum, avg`

Note `count(A)` counts non-null values in A, `count(*)` counts rows in table

`avg(distinct A)`: average of distinct non null values, `count(distinct A)`: counts distinct non null values etc

Note: Aggregate functions can only be used in SELECT, HAVING, ORDER BY (NOT WHERE)

`where price = (select max(price) from Sells);` not where price = max(price)

GROUP BY

Each output tuple corresponds to one group

If A in R appearing in SELECT clause, one must hold

- A in group by, OR A in aggregated expression OR grouped by PK

Note: if aggregate function in SELECT clause, and no GROUP BY clause, SELECT clause cannot contain any column not in aggregated expression

`HAVING`

having applied to every group, so attributes in `HAVING` have same restrictions

Conceptual Evaluation of Queries

1. Compute tables in FROM
2. Select tuples that evaluate to true for WHERE
3. Partition by WHERE
4. Select groups evaluate to true for HAVING
5. Generate output tuple by selecting attributes in SELECT
6. Remove duplicate output tuples (if DISTINCT)
7. Sort by ORDER-by
8. Remove based on OFFSET and LIMIT

NULLIF

Syntax: `nullif(value1, value2)`

Returns null if value1 = value2; otherwise returns value1

Universal Quantification

```
-- Names of students who have enrolled in all CS Courses
select name
from Students S
where not exists (
    select courseId
    from Courses C
    where dept = 'CS'
    and not exists (
        select 1
        from Enrolls E
        where E.cid = C.courseId and E.sid = S.studentId
    )
);
```

PL/PGSQL

Functions and Procedures

```
CREATE OR REPLACE FUNCTION <name>
    (IN / OUT <param> <type>, ...)
RETURNS <type> AS $$
<code>
$$ LANGUAGE sql;
<name>(params);

CREATE OR REPLACE PROCEDURE <name>
    (<param> <type>, ...)
AS $$
<code>
$$ LANGUAGE sql;
CALL <name>(params);
```

Return type (for functions):

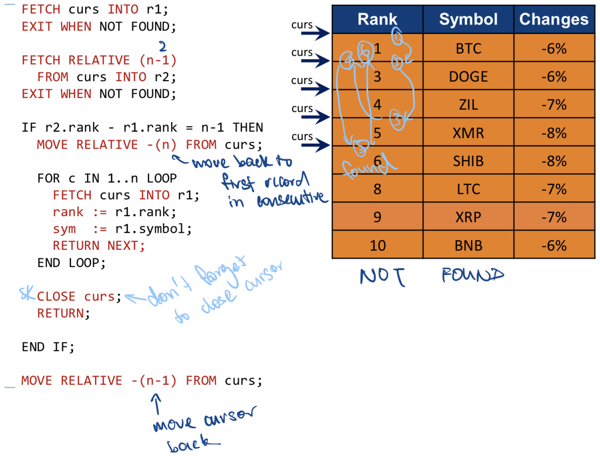
- `VOID`: nothing
- `RECORD` or table name: first (one) tuple, if record specify out params
- `TABLE(<param> <type>, ...)`: same as `RECORD`

Control Flow

```
If <case> THEN <statement> ;
ELSIF <case> THEN <statement> ;
ELSE <statement> ;
END IF;
WHILE <condition> LOOP
    <code>
END LOOP;
LOOP
    EXIT WHEN <condition> ;
<code>
END LOOP;
array INT[] := ARRAY[1,2,3]; -- index starts at 1
FOREACH d IN ARRAY array LOOP
    <code>
END LOOP;
```

Cursor: access each indiv row returned by `SELECT`

Declare cursor → Open cursor → fetch tuple from cursor → close cursor



Cursor Movement:

- `FETCH curs INTO r;` - fetch then move
- `FETCH NEXT FROM curs INTO r;` - move then fetch
- `FETCH PRIOR FROM curs INTO r;`
- `FETCH FIRST FROM curs INTO r;`
- `FETCH LAST FROM curs INTO r;`
- `FETCH ABSOLUTE X FROM curs INTO r;` - Fetch Xth tuple
- `FETCH RELATIVE X FROM curs INTO r;`
- `MOVE RELATIVE X FROM curs;` - Moves cursor X rows away

Triggers

Trigger Function

```
CREATE OR REPLACE FUNCTION <name>()
RETURNS TRIGGER AS $$
BEGIN
    <code>
END;
$$ LANGUAGE plpgsql;
```

Must return `TRIGGER` to access:

`NEW`: pointer to the tuple to be inserted

`OLD`: old tuple being updated / deleted

`TG_OP`, `TG_TABLE_NAME`

Trigger Timing

After: Executed after insertion (return value not important)

Before: Executed before insertion

Instead Of: function run instead of insertion, can only be defined on views

- typically used to do something on table instead of on view

Return Value

BEFORE INSERT: non-null t: t is inserted, NULL: nothing inserted

BEFORE UPDATE: non-null t: t is updated tuple, NULL: no tuple updated

BEFORE DELETE: non-null t: delete (even if t non-existent), NULL: no delete

Note: Before row-level triggers return `NULL`, subsequent triggers ignored

AFTER: return value does not matter

INSTEAD OF: `NULL`: ignore rest of operations on row, non null: proceed

Statement Level

Return Values: statement level triggers ignore the values returned by trigger functions

Note: RETURN NULL would not make the database omit the subsequent operations, raise exception to do this

Note: `INSTEAD OF` only allowed on row-level, `BEFORE, AFTER` allowed on both

Deferred Trigger

To defer trigger to end of transaction

```
CREATE CONSTRAINT TRIGGER <name> -- instead of CREATE TRIGGER <name>
AFTER INSERT OR UPDATE OR DELETE ON <table>
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW EXECUTE FUNCTION <name>;
```

- Constraint and deferrable together indicate trigger can be deferred

NOTE: Deferred triggers only work with AFTER and FOR EACH ROW

Order of Activation

BEFORE statement → BEFORE row → AFTER row → AFTER statement

Within category, triggers activated in alphabetical order

Functional Dependency

Definition: $A_1A_2\dots A_m \to B_1B_2\dots B_n$ if

1. 2 rows have same values on $A_1, A_2, \dots, A_m$ and
2. they always have the same values on $B_1, B_2, \dots, B_n$

If 2 tuples have same NRIC, they have same NAME

Closure

1. Initialise the closure to $\{A_1, \dots A_n\}$
2. If there is an FD: $\{A_1, \dots, A_m\} \to B$, such that $A_1, \dots, A_m$ are all in closure, then put B into closure
3. Repeat 2 until no new attributes

To prove that $X \to Y$ holds, show that $\{X\}^+$ contains Y. Inverse is true

Superkeys, Keys

Closure of Superkeys contain all columns in the table

Note: if attribute is not in the RHS of any FDs, then must be in key

Prime Attributes: attribute that appears in keys

BCNF

In BCNF if every non-trivial decomposed FD has superkey as its LHS

BCNF check: No closure violates "more but not all" condition

BCNF decomposition: Binary split until subtables are in BCNF

1. Find subset X of attributes in R that violates "more but not all"
2. Decompose R into R_1 and R_2, such that
   - R_1 contains all attributes in $\{X\}^+$
   - R_2 contains all attributes in X and attributes not in $\{X\}^+$
3. if R_1 or R_2 not in BCNF, further decompose

NOTE: BCNF decomposition is not unique, table with 2 columns is in BCNF

Projection of FDs

1. For each attribute subset of R_i derive closure on R
2. Project closure onto R_i by removing attributes not in R_i

Lossless decomposition

Table decomposed by BCNF can be reconstructed from sub tables.

BCNF decomposition is lossless because X is key of R_1, so for each row in R_2 there is a unique row in R_1 to join to

Dependency Preservation: to avoid making "inappropriate" updates

BCNF may not preserve dependencies.

eg $AB \to C, B \to C$, decomposes to $R_1(A, C), R_2(B, C)$, $AB \to C$ is lost

S = set of FDs on original table

S' = set of FDs on decomposed table

Decomposition preserves all FCs iff S and S' are equivalent (use closures)

- Every FD in S' can be derived from S and vice versa

3NF

Every non-trivial decomposed FD, LHS superkey, or RHS prime attribute

More lenient than BCNF, satisfy BCNF → 3NF, converse may not be true

violate 3NF → violate BCNF, converse may not be true

3NF Check

"more but not all", and not all is not prime attribute → violate 3NF

3NF decomposition: n-ary split into tables in 3NF

1. Derive minimal basis of S
2. In minimal Basis, combine FDs with same LHS
3. Create table for each FD remaining
4. Make table with Key if none contain key. (To ensure lossless join)
5. Remove redundant tables

Minimal Basis: Simplified version of S, the set of FDs

Conditions for minimal basis:

1. Every FD can be derived from S and vice versa
2. Every FD is a non-trivial, decomposed FD
3. No FD in minimal basis is redundant
4. For each FD in minimal basis, none of the attributes on LHS is redundant.

ie. if remove an attribute from LHS, FD cannot be derived from S

Algo to find minimal basis:

1. Decompose FDs
2. Remove redundant attributes on LHS
   - Remove one, check remaining attributes closure same as before removing.
   eg. remove A from $AB \to C$, check if $\{B\}^+$ is same as before removing A
3. Remove redundant FDs
   - Remove FD and check if closure of attributes on FD LHS is same
   eg. remove $AB \to C$, check $\{AB\}^+$ is same as before removing $AB \to C$

Tip: $AB \to C$ is redundant if $B \to C$ exists