Z5488317: Felix Lin

ERD and SQL :D

mapping ERD rels:
N:M 3rd table w FKs
1:N '1' table has FK
1:1 **total** table has FK

Mapping ERD subclasses
ER-style parent table, subclass tables w FK to parent
OO-style table for parent and subclass w all attr; subclasses must reference parent
Single Table 1 table w attrs of all classes; can represent disjoint inheritance by using an attr to signify its subclass

Treat **MVAs and weak entities** as rels. wk PK = FK + discrim, MVA PK = FK + attrs

```
CASE WHEN cond1 THEN res1
     WHEN cond2 THEN res2
     …
     ELSE default
END
```
equiv to if-else; returns single val

attr CHECK(cond); cond involves attr

```
CREATE TYPE name
AS ENUM(val1, val2, …)
```

```
attr type PRIMARY KEY
REFERENCES table(X)
```

Can use CASE for just one attr in query

```
IF cond THEN
    … code …
ELSIF cond THEN
    … code …
ELSE
    … code …
END IF
```
plpgsql

```
CREATE TABLE name (
    attr1    type    special constraint CHECK(condition),
    …
    CONSTRAINT name CHECK(condition),
    PRIMARY KEY (PK attrs),
    FOREIGN KEY (FK attrs) REFERENCES r(PK)
);
```

Filter groups using HAVING instead of WHERE

```
CREATE OR REPLACE FUNCTION
    funcName(arg1 type1, …)
RETURNS retType
AS $$
    … sql code …
$$ LANGUAGE sql;
```

```
INSERT INTO table (attr1, attr2, …)
VALUES (val1, val2, …)
```

If we want to find (a,b) pairs and exclude (b,a) pairs, we can sort by PK and put a WHERE pk1 > pk2

```
CREATE OR REPLACE FUNCTION
    funcName(arg1 type, …)
RETURNS retType
AS $$
DECLARE
    variable1 type;
    …
BEGIN
    … plpgsql code …
END
$$ LANGUAGE plpgsql;
```

plpgsql funcs returning a SETOF can use RETURN NEXT to append to the result, and RETURN to return all

```
DELETE FROM table
WHERE cond
```

```
UPDATE table
SET attr_i = val1, attr_j = val2, …
WHERE cond
```

```
FOR record IN
    sql select query
LOOP
    … plpgsql code …
END LOOP;
```

```
CREATE ASSERTION name
check(condition);
```
SQL only
Checked before and after all ops on the db

```
CREATE AGGREGATE name(intype) (
    sfunc = name of transition func
    stype = type of intermediate states
    initcond = initial value of starting state
    finalfunc = name of finalisation func
)
```

sfunc and finalfunc are plpgsql functions. finalfunc is optional

Append 'RETURNING x' to INSERT if inserting into a table w a serial attr x without specifying a value for x

Arrow points at the '1' entity in the relationship

Not null and check() constraints are row level, unique and PK are table level, FK is inter-table

Triggers :0

Trigger funcs have access to **TG_OP** = 'INSERT', 'DELETE', 'UPDATE'

```
CREATE OR REPLACE TRIGGER name
AFTER/BEFORE operations ON table
FOR EACH ROW
EXECUTE FUNCTION funcName();
```

If operation is UPDATE, we can specify BEFORE/AFTER UPDATE OF attr

plpgsql trigger functions have return type TRIGGER, take no args, and have access to NEW and OLD which have type RECORD

For each statement executes once for the entire statement

Modifying and returning NEW in BEFORE trigger funcs impact the operation

Insert: NEW
Update: NEW + OLD
Delete: OLD

Returning OLD or NULL in a BEFORE trigger function or raising an exception aborts the operation

Modifying OLD does nothing.

For each row runs funcName on each row impacted by the triggering event

STRING_AGG(expr, 'split' ORDER BY …) will create a 'split' separated string of the expr (optionally) ordered by …

AGGREGATE(expr) FILTER(WHERE cond) allows a where to be applied only on the aggregate

RAISE EXCEPTION str;

def funcName(args):

PSYCOPG2 :a

RA note: theta joins will include both joined attr
R Join[R.c=S.c] C will have a column for R.c and S.c

a**b = a^b
a // b = floor(a / b)
exit(0) success, (1) failure
Strings are similar to C; str[0] is first letter

Parse strings using int(str) or float(str). round(num, x) will round to x dp
len(list) = length of list

cur.execute(query_string, [flags])

Use this notation to avoid SQL injections

sys.argv to access command-line args

Python uses None instead of NULL; is None, is not None

input("prompt") to print prompt and take in user input from terminal

Redundancy and FDs :O

Reflexivity
$X \rightarrow X$

Transitivity
$X \rightarrow Y, Y \rightarrow Z \Rightarrow X \rightarrow Z$

Additivity
$X \rightarrow Y, X \rightarrow Z \Rightarrow X \rightarrow YZ$

Projectivity
$X \rightarrow YZ \Rightarrow X \rightarrow Y,\ X \rightarrow Z$

Closure F^+ is the largest set of FDs derivable from F

Closure X^+ is the largest set of attributes derivable from X using F

If $Y \subset X^+$, then $X \rightarrow Y$

To find the **attribute closure** of set X, start with $closure = X$, then extend $closure$ by adding B from $A \rightarrow B \in F$ where $A \subset closure$ until you can't add anything more

$X \rightarrow Y$: if $X \subseteq Y$, the FD is trivial; i.e. X is all attrs in schema

To check if F and G are equivalent:
1. For each FD in G, check it is derivable from F, and vice versa
2. If all true, then $F = G$

The key of R implied by F is the smallest subset of attributes $K \subset R$ such that $K^+ = R$

Pseudo-Transitivity
$X \rightarrow Y, YZ \rightarrow W \Rightarrow XZ \rightarrow W$

In order of most to least redundancy 1,2,3NF, BCNF, 4, 5 NF

BCNF differs from 3NF in that it may not preserve all FDs

Decompose R into S and T such that $R = S \cup T,\ S \cap T \neq \emptyset$

A schema R is in BCNF iff all FDs $(X \rightarrow Y) \in F^+$ are trivial or $X^+ = R$ (is CK)

The reduced minimal cover of F is the minimal cover with common LHSs recombined through additivity: $A \rightarrow B,\ A \rightarrow C$ becomes $A \rightarrow BC$

$X \rightarrow Y$ can be left reduced if there exists $Z \subset X$ such that $Z \rightarrow A$ can replace $X \rightarrow A$ without changing F^+

A decomp is lossy if we lose critical connection info needed for joins

$r(R) \neq s(S)\ Join\ t(T)$

When finding FDs, just find the ones on the schema, don't look at FKs

To **find the minimal cover** of F, make FDs canonical (1 attr RHS), left-reduce FDs (remove redundant attrs from X), then remove redundant (derivable) FDs

Note that minimal covers are rarely unique

Check each schema $S \in Res$, and choose a relevant FD $X \rightarrow Y$ on S that violates BCNF ($X\ != key(S)$) and FD is not trivial) to do decomp with

Compute F^+ (impossible, just use what they give)

BCNF Decomp

$Res = \{R\}$
while (any schema $S \in Res$ is not in BCNF)
choose any FD $X \rightarrow Y$ on S that violates BCNF
$R1 = S - Y$
$R2 = XY$
$Res = (Res - S) \cup R1 \cup R2$

3NF Decomp

Compute **reduced** minimal cover F_c
$Res = \{\}$
for each FD $X \rightarrow Y \in F_c$
if (no $S \in Res$ contains XY)
$Res = Res \cup XY$

if (no $S \in Res$ contains $key(R)$)
$K = any\ key(R)$
$Res = Res \cup K$

EXPLAIN ANALYZE sql_query;
Shows query evaluation tree in preorder traversal: read from the deepest indented leaves out

ANALYZE is optional (runs and times the query instead of just estimating)

\timing in psql cli toggles timing for queries

CK is any attr set X such that $X^+ = R$, and there is no $Y \subset X$ such that $Y^+ = R$. often multi CKs

Rename[S(attr1, attr2, …)]R
Renames R S, and the attrs of r to attr1, attr2, …

Rename[attr1, …]R
Renames the attrs of R to attr1, …

NOTE: RA ops return sets, not bags
NO DUPLICATES → impacts aggregates

Is in 3NF if all $X \rightarrow Y$: X is a CK or Y is a single attr in a CK; or if in BCNF

Is in BCNF if all LHS of all relevant FDs are CKs

Good strat for BCNF decomp is to state S, state relevant FDs on S, state $key(S)$ then determine if S is in BCNF, decomposing if not

Performance Tuning and Transactions ö

Sorting	$O(n \log_B n)$	External merge sort using B memory buffers
Selection – Sequential Scan	$O(n)$	
Selection – Index-Based	$O(n \log n)$	Uses B-trees with pages as nodes. Finds the page a tuple is in then searches it. Good for max/min
Selection – Hash-Based	$O(1)$	Can only be used for equality (=) tests
Nested-Loop Join	$O(nm)$	Reducible to $O(n + m)$ with buffering. Optimal for joining large relations with small ones
Sort-Merge Join	$O(n \log n + m \log m)$	Optimal for joining two large relations that are similar in size
Hash-Join	$O(n + m \bullet \frac{n}{B})$	Uses B buffers. Optimal for two large relations that are not similar in size. Hashed indexes may lose value after a join, so don't use multiple hash-joins together

To **encourage efficient choices by DBMS**:

Use joins instead of subqueries, especially if subqueries are correlated

Filter before combining data (join, prod, div)

Avoid applying functions in WHERE/GROUP BY clauses

Indexes provide efficient content-based access to tuples, speeding up filtering/searching

Create and use indexes on tables; only if the table is queried much more than it is updated, or the queries are very expensive

Rearrange math to have indexed attr as subject, and use UNIONs of select statements with indexed attrs not OR, otherwise indexes will not be used

CREATE INDEX name ON table(attr1, …) USING method

If indexing on unique attrs for equality tests, use hash method, else for non-unique range tests use btree method

Rearrange math to have indexed attr as subject

Use UNIONs of select statements with indexed attrs instead of OR

Transactions are
Atomic
Consistent
Isolated
Durable

The first query on some data will be slower as it reads from disk to memory, but subsequent queries can just use the data read into memory by the first query

Conflict serialisable → schedule can execute R and W ops in the 'right' order

Conflict if $R_i(X)\ W_j(X)$, $W_i(X)\ W_j(X)$, or $W_i(X)\ R_j(X)$

Build precedence graph by drawing edge from T_i to T_j for each conflicting op between the transactions where the op occurs in T_i before T_j

Two schedules are view equivalent if every R views the same version of a data item, and the final W for each data item is the same

A schedule is safe if it is view equivalent to a serial schedule

All conflict serialisable schedules are view serialisable, but not the other way around

To check **view serializability**, find a view equivalent serial schedule from among the n! serial schedules

Serializability tests are theoretically useful, but don't help make schedules in the first place
Computationally cost $O(n!)$

Div is good for finding "all" conditions; Div(ppl, xyz) finds all ppl with x and y and z.

when considering relevant FDs, break existing RHS to get more relevant FDs; **A->BCD gives A->BC on R(ABC)**

'' to print apostrophe, E'\n'

sfunc(sType, baseType) returns sType

REMEMBER: MVA PK = FK + all attrs

Find the ids of employees who make the highest salary.
Approach: find employees who do not have the highest salary (via the join), and then subtract them from the set of all employees, thus leaving just the highest paid employees.
E1 = Employees
E2 = Employees
Employees = Proj[eid](Employees)
LowerPaidEmployees = Proj[E2.eid](E1 Join[E1.salary>E2.salary] E2)
Answer = Employees Minus LowerPaidEmployees

$R = ABCDEFGH$
$F = \{ABH \rightarrow C,\ A \rightarrow D,\ C \rightarrow E,\ BGH \rightarrow F,\ F \rightarrow AD,\ E \rightarrow F,\ BH \rightarrow E\}$
$F_c = \{BH \rightarrow C,\ A \rightarrow D,\ C \rightarrow E,\ F \rightarrow A,\ E \rightarrow F,\ BH \rightarrow E\}$
Produce a 3NF decomposition of R.
3NF is constructed directly from the minimal cover, after combining dependencies with a common right hand side where possible.

F_c gives the following tables (with table keys in bold):
BHCE AD CE FA EF

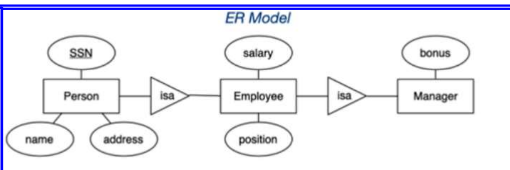
A key for R is BHG; G must be included because it appears in no functional dependency. Since no table contains the whole key for R, we must add a table containing just the key, giving: 3NF = *BHCE AD CE FA EF BGH*

Compute a minimal cover for:
$F = \{B \rightarrow A,\ D \rightarrow A,\ AB \rightarrow D\}$
[hide answer]
Steps in converting to a minimal cover:

1. put FDs into canonical form:
   $B \rightarrow A$, $D \rightarrow A$, $AB \rightarrow D$ is already in canonical form.
2. eliminate redundant attributes:
   The only possible redundant attributes are A or B in $AB \rightarrow D$.
   We can prove that A is redundant as follows:
   $B \rightarrow A \Rightarrow BB \rightarrow AB \Rightarrow B \rightarrow AB$
   $AB \rightarrow D + B \rightarrow AB \Rightarrow B \rightarrow D$
   Since we have $AB \rightarrow D$ and $B \rightarrow D$, A is redundant.
3. eliminate redundant dependencies:
   The above elimination leaves: $B \rightarrow A$, $D \rightarrow A$, $B \rightarrow D$
   But $D \rightarrow A$, $B \rightarrow D \Rightarrow B \rightarrow A$

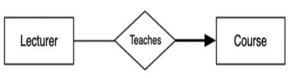
So, the minimal cover is: $B \rightarrow D$, $D \rightarrow A$

ER Model

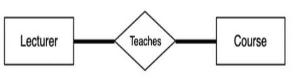


OO-Style *Relational Version*

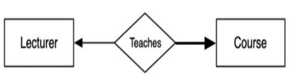
Person	SSN	name	address			
Employee	SSN	name	address	salary	position	
Manager	SSN	name	address	salary	position	bonus



A lecturer may teach one course
Every course is taught by 1 or more lecturers



Every lecturer teaches one or more courses
Every course is taught by 1 or more lecturers



Every lecturer may teach one course
Every course is taught by one lecturer

$R = ABCDEFGH$
$F = \{ABH \rightarrow C,\ A \rightarrow DE,\ BGH \rightarrow F,\ F \rightarrow ADH,\ BH \rightarrow GE\}$

- We start from a schema: *ABCDEFGH*, with key *BH* (work it out from FDs).
- The FD $A \rightarrow DE$ violates BCNF (FD with non key on LHS).
- To fix, we need to decompose into tables: *ADE* and *ABCFGH*.
- FDs for *ADE* are $\{A \rightarrow DE\}$, therefore key is *A*, therefore BCNF.
- FDs for *ABCFGH* are $\{ABH \rightarrow C,\ BGH \rightarrow F,\ F \rightarrow AH,\ BH \rightarrow G\}$
- Key for *ABCFGH* is *BH*, and FD $F \rightarrow AH$ violates BCNF (FD with non key on LHS)
- To fix, we need to dcompose into tables: *AFH* and *BCFG*.
- FDs for *AFH* are $\{F \rightarrow AH\}$, therefore key is *F*, therefore BCNF.
- FDs for *BCFG* are $\{\}$, so key is *BCFG* and table is BCNF.
- Final schema (with keys boldened): ***ADE***, ***FAH***, ***BCFG***