Normalization – achieve better designed relational database schemas using FDs and PKs

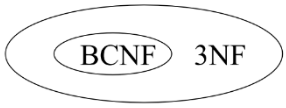
- **1NF** – only have single value as attribute (BUT have redundancy OR too many NULL value)
- **2NF** – FD: X→Y, where X is minimal key and Y is non-prime attribute, then no proper subset of X determines Y
- **Boyce-Codd (BCNF)** – for every FD: if X → b (non-trivial: b is not the subset of X), then X is a Superkey for R 【LHS is Superkey, RHS is not part of LHS】

  Decomposing into BCNF losslessly (3 steps):

  1.. Add implicit FDs to your list of FDs 2. Pick FDs that violates BCNF of the form $X \rightarrow b$ 3. Decompose R: $R_1(X \cup \text{others})$ & $R_2(X \cup b)$ [Note: table with only 2 attributes is in BCNF]

  √ : no redundancy of data/ no anomalies & ×: cannot preserve all dependencies

  Dependency preservation: FD $X \rightarrow Y$ is preserved, if R contains all of the attributes of X and Y (having all attributes of FD in one table) → hard to check violation of FD without join back into a single table
- **Third normal form (3NF)** – (RHS is not part of LHS) LHS is Superkey OR RHS is prime attribute (keep some redundancy/anomalies BUT preserve all FDs)



  Minimal cover G for a set of FDs F 【Closure of F = closure of G】

  · RHS is a single attribute & cannot delete any FD or attribute

  Finding minimal covers of FDs (do in order):

  1. Put FDs in standard form (have only one attribute on RHS) 2. Minimize LHS of each FD

  3. Delete redundant FDs (the closures are the same with and without the redundant FD)

  Decomposition into 3NF (using minimal cover):

  1. Get the minimal cover F' & find all the candidate keys
  2. Decompose using F' if violating 3NF (same step as BCNF include decompose the implicit FDs)
  3. Identify FDs in F' that are not preserved → add back [e.g. $X \rightarrow a$ in N create a relation $R_n(X \cup a)$]

Denormalization – violate normal form to gain performance improvements: fewer joins & reduces number of FKs

SQL (structured query language) – declarative, based on relational algebra【result of SQL query is table/relation】

Data definition language (DDL) – define database schema/ structure – e.g. CREATE, ALTER and DROP TABLE

CREATE TABLE: create a new relation (specifying table name, attributes and constraints)

StarsIn[MovieID, StarID, Role]
StarsIn.StarID → MovieStar.StarID
StarsIn.MovieID → Movies.MovieID

```
CREATE TABLE StarsIn(
  StarID      INTEGER,
  MovieID     INTEGER,
  Role        CHAR(20),
  PRIMARY KEY (StarID, MovieID),
  FOREIGN KEY (MovieID) REFERENCES (Movies)
  FOREIGN KEY (StarID) REFERENCES MovieStar,
    ON DELETE CASCADE
    ON UPDATE CASCADE)
```

Domain constraint specified for each attribute

Can specified directly or by CREATE DOMAIN

Varchar (variable character) V.S. character (fixed length character)

Enforcing referential integrity constraint

Referential triggered actions (4) when referenced tuple is deleted or updated:

- NO ACTION (default): the delete/update is rejected
- SET NULL
- SET DEFAULT
- CASCADE: delete/update all roles that refer to

ALTER TABLE: alter the structure of tables

Possible actions: add/ drop column OR change column definition/domain OR add/ drop constraints

ALTER TABLE <table name>
ADD <column name> <column type> [<attribute constraint>] {, <column name> <column type> [<attribute constraint>] }
| **DROP** <column name> [CASCADE]
| **ALTER** <column name> <column-options>
| **ADD** <constraint name> <constraint-options>
| **DROP** <constraint name> [CASCADE];

DROP TABLE: delete/remove/drop tables/objects from the database:

Deletes all tuples within the table

Removes the table definition from the system catalog

Drop all constraints defined on table includes FK constraints in others table

CASCADE: drop things related to that table

RESTRICT: disallow dropping the table, if that table got somethings related to it

DROP TABLE [IF EXISTS] *tbl_name* [, *tbl_name*] ... [RESTRICT | CASCADE]

Data manipulation language (DML) – manipulate/ managing data

SELECT: retrieve data from a database

INSERT: insert data into a table

UPDATE: updates existing data within a table

DELETE: deletes records from a table

> **INSERT INTO <table name>**
> [(<column name> {, <column name> })]
> (**VALUES** (<constant value>, {,<constant value> })
> | **<select statement>**);

INSERT INTO Student (<u>sID</u>, sName, GPA, sizeHS)
VALUES (53688, 'Smith', 3.2, 200)

> **UPDATE** <table name>
> **SET** <column name> = <value expression>
> {, <column name> = <value expression>}
> [**WHERE** <select condition>];

> **DELETE FROM** <table name>
> [WHERE <select condition>];

Relational algebra:

> SELECT [DISTINCT] (attribute/expression list | *)
> FROM <table list>
> [WHERE [join condition and] search_condition]
> **[ORDER BY column_name [ASC|DESC] {, column-name [ASC|DESC]}];**

- **Projection (SELECT)**: vertically select attributes (columns wanted to return)
  Can evaluate expressions (+, -, *, /) on numeric values or attributes with numeric domains (e.g. SELECT Year+2)
  Renaming attributes using AS clause: *old-name* AS *new-name* (e.g. SELECT Role AS Role1)
- **Selection (WHERE: join condition & search condition)**: horizontal scanner (select tuples)
  Complex WHERE conditions:
  - Substring comparisons (LIKE, IN, IS)
    LIKE is used for string matching:
    %: stands for 0 or more arbitrary characters
    [e.g. WHERE Title like "%sin%"]
    _ (underscore): stands for any one character
    IN [e.g. WHERE LName IN ('Jones', 'Wong')]
    IS [e.g. WHERE No IS NULL]
  - Arithmetic operators and functions
    +, -, *, /, date, time, year etc
    e.g. WHERE Salary*2 > 50000 & WHERE Year(Sys_Date – Bdate) > 55
    BETWEEN … AND …
    e.g. WHERE Salary BETWEEN 1000 AND 3000
- **Join (FROM & WHERE join condition)** (e.g. FROM StarsIn S) – join 2 tables: R1, R2 on their shared attribute
  Based on Cartesian product (give all the combination) – n rows in R_1, m rows in R_2 $\rightarrow$ R with $m * n$ rows
  Types of join operations (3):
  Theta-join: with logical operators $(=, \neq, <, \leq, >, \geq)$ in WHERE clause
  Equi-join: with $=$ in WHERE clause (special case of Theta join)
  Natural join (Equi-join): columns with same name of associate tables will appear once only)
  [Natural join of tables with no pairs of identically named columns/ no common attribute will return the cross product of the 2 tables]
  Inner join (default) – includes the tuple only if matching tuples exist in both relations
  Outer join – include tuples that do not satisfy the join condition [set NULL value to tuple that does not match]
  Full outer join – includes all rows from both tables & Left outer join – includes all rows from first table & Right outer join – includes all rows from second table

> **SELECT** <attribute list>
> **FROM** *table_reference* {LEFT|RIGHT} [OUTER] JOIN *table_reference ON* < search_condition>

R

A	B
1	2
3	3

S

B	C
2	4
4	6

Natural Inner Join

A	B	C
1	2	4

Natural Left outer Join

A	B	C
1	2	4
3	3	Null

Natural Right outer Join

A	B	C
1	2	4
Null	4	6

Natural outer Join

A	B	C
1	2	4
3	3	Null
Null	4	6

> Outer join (without the Natural) will use the key word on for specifying the condition of the join.

> Outer join not implemented in MYSQL
> Outer join is implemented in Oracle

- **Sorting (ORDER clause)**: order the resulting tuples according to the given sort key
  ASC (ascending): small to large (default) OR DESC (descending): large to small

<u>Set operations</u> – relation is set of tuples (no duplicates) – to retain duplicates use multiset versions: UNION ALL, INTERSECT ALL, EXCEPT ALL [have m, n duplicates in relation r, s, then the number of duplicates will be: $m + n$ times in r UNION ALL s & $\min(m, n)$ times in r INTERSECT ALL s & $\max(0, m - n)$ times in r EXCEPT ALL s)]

【Relations must be union compatible to do the operations

Union compatibility: iff 2 relations $R_1(A_1, A_2, \ldots, A_n)$ and $R_2(B_1, B_2, \ldots, B_n)$: have the same degree n (number of columns) & their columns have corresponding domains ($\mathrm{domain}(A_i) = \mathrm{domain}(B_i)$ for $1 \leq i \leq n$)

[Note: the corresponding columns do not have to have the same column name]】

- **UNION (∪)**: SELECT … UNION [ALL] SELECT … [UNION [ALL] SELECT …] same as OR using in WHERE clause
- **INTERSECTION (∩)**: SELECT … INTERSECT [ALL] SELECT … [INTERSECT [ALL] SELECT …]

【Note: INTERSECT is part of the SQL standard, BUT is not implemented in MySQL → do it in 2 table】

```
WHERE
        M1.MovieID = S1.MovieID AND M1.year = 1944 AND
        M2.MovieID = S2.MovieID AND M2.year = 1974 AND
        S2.StarID = S1.StarID
```

- **DIFFERENCE/EXCEPT/MINUS (−)**: includes all tuples in R_1, but not in R_2 【Note: EXCEPT is part of SQL standard, BUT cannot use in MySQL → nested queries can work as EXCEPT】

  SELECT ⋯ EXCEPT [ALL] SELECT ⋯ [EXCEPT [ALL] SELECT ⋯]

Aggregation: produce summary values on SELECT clause (cannot have nested aggregation)

Aggregation functions: SUM/ AVG ([DISTINCT]) expression), COUNT ([DISTINCT] expression), COUNT(*), MAX/ MIN(expression) (Note: the domain of values can be non-numeric)

GROUP BY and HAVING – divide tuples into groups (GROUP BY) and apply conditions to each group (HAVING)

```
SELECT      [DISTINCT] target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
ORDER BY    target-list
```

Target-list contains:

(i) Attribute names (must also be in *grouping-list*)

Each answer tuple corresponds to a *group*

Group = a set of tuples with same value for all attributes in *grouping-list*

Selected attributes must have a single value per group

(ii) Terms with aggregate operations

Group-qualification: attributes in *group-qualification* are either in *grouping-list* or are arguments to an aggregate operator

Having: similar to WHERE clause, BUT HAVING clause can also include aggregates

Conceptual evaluation of a query:

1. Compute the cross-product of relation-list (FROM clause)
2. Keep only tuples that satisfy qualification (WHERE clause)
3. Groups remaining tuples by where attributes in grouping-list (GROUP BY clause)
4. Keep only the groups that satisfy group-qualification (expressions in group-qualification have a single value per group) (HAVING clause)
5. Delete fields that are not in target-list (SELECT clause)
   → Generate one answer tuple per qualifying group
6. If DISTINCT is specified, eliminate duplicate rows
7. If ORDER BY is specified, sort the results

```
SELECT ... FROM ... WHERE
    {expression {[NOT] IN |
    comparison-operator [ANY|ALL]}
| [NOT] EXISTS}
(SELECT ... FROM ... WHERE ...);
```

Outer Query

Nested/Sub-Query

Nested queries

- **Nested query/ sub-query**: query within another query

  Useful for expressing queries where data must be fetched and used in a comparison condition

  · Inside the WHERE clause of another SELECT statement (other query search conditions, including joins, can also appear in the outer query WHERE clause, either before or after the inner query)

  · Inside an INSERT, UPDATE or DELETE statement

  · Sub-query cannot include the ORDER BY clause BUT DISTINCT may effectively order the results of a sub-query, since most systems eliminate duplicates by first ordering the results

- **Correlated and non-correlated variants**

  Non-correlated nested queries – inner query, outer query run independently (don't need to do join), so it could be computed just once (sub-queries are evaluated from the 'inside out')

  Correlated nested queries – inner query depended on outer query (need to do join in inner query WHERE) & compute many times

- **Sub-query operators** (expression and attribute list in sub-query SELECT clause must have same domain)

  · [NOT] IN (sub-query)

  · Comparison-operator (>, <, =, ≥, ≤, <>) [ANY|ALL] (sub-query)

```
SELECT cName, state
FROM College C1
WHERE exists (SELECT *
              FROM College C2
              WHERE C2.state = C1.state AND
              C2.cName <> C1.cName);
```

Think of this as passing parameters

[sub-query return a single value can only use comparison-operator]

ANY: evaluates to true if one comparison is true [e.g. > ANY X → return the value greater than MIN(X)]

ALL: evaluates to true if all comparisons are true [e.g. > ALL X → return the value greater than MAX(X)]

Equivalence: 'IN' and '= ANY' & 'NOT IN' and '<> ALL' & Non-equivalence: 'NOT IN' and '<> ANY'

Sub-queries V.S. set operations V.S. joins (can be used to write the same query)

- Use joins when you are displaying results from multiple tables
- Use sub-queries when you need to compare aggregates to other values

EXISTS – check (non)existence of data → return T or F (won't do join → always use correlated subquery)

WHERE EXISTS/ NOT EXIST (sub-query) – T if the result of the correlated sub-query is not-empty/ empty set

Division – answer queries include 'for all'/ 'for every' [BUT no direct way to write division in SQL →double negation]

Views – types: virtual tables (not physically exist) & materialized (physically exist & need to update)

- Defining a view:
  CREATE VIEW <view name> (<column name> {,<column name>}) AS SELECT …
- Dropping views:
  DROP VIEW [IF EXISTS] <view name> [,<view name>]… [RESTRICT | CASCADE]
  RESTRICT: drops the table, unless there is a view on it
  CASCADE: drops the table, and recursively drops any view referencing it

Retrieve the names of corals which are in **ALL** reefs

→ using double negation

→ the names of corals which have no reefs they are not in

REEF[reefname, latitude, longitude, 2006_bleachedarea, summer_maximum_mounthly_mean_temperature]

CORAL[coralname, coralcode, thermalthreshold]

CORALSAMPLING[sampleno, coralcode, reefname, dateofsampling, bleachpercent]

```
SELECT coralname
FROM CORAL A
WHERE NOT EXIST (
    SELECT reefname
    FROM REEF
    WHERE reefname NOT IN (
        SELECT reefname
        FROM CORALSAMPLING B
        WHERE B.coralcode = A.coralcode))
```

1. What reefs is coral__(a random coral) in?
   SELECT reefname
   FROM CORALSAMPLING B
   ~~WHERE coralcode = X~~
   → do the join:
   WHERE B.coralcode = A.coralcode
2. What reefs is coral__(a random coral) not in?
   SELECT reefname
   FROM REEF
   WHERE reefname NOT IN (part 1)
3. What did we get answer in 2?
   SELECT coralname
   FROM CORAL A
   WHERE NOT EXIST (part 2)
   → as NOT EXIST do not do the join, so need to do the join in part 1

```
Create View Temp(major, average) as
    SELECT S.major, AVG(S.age) AS average
    FROM Student S
    GROUP BY  S.major;
```

NULL value – indicates the value is unknown

- IS NULL (IS NOT NULL): used to check whether the value is known (not known)
- Operations on NULL value
  NULL requires a 3-valued (true, unknown, false) logic using the truth value unknown
  - OR: (unknown OR true) = true
    (unknown OR false) = unknown
    (unknown OR unknown) = unknown
  - AND:(true AND unknown) = unknown
    (false AND unknown) = false
    (unknown AND unknown) = unknown
  - NOT: (NOT unknown) = unknown
  - Comparisons between 2 null values, or between a null and any other value → return unknown
  - All aggregate operations except COUNT(*) ignore tuples with null values on the aggregated attributes

Semantic constraints can be specified using CHECK and ASSERTION statements

- **CHECK**– constraints write at the end of CREATE TABLE statement (check when tuples are inserted or modified)
  CHECK (condition)
- **ASSERTION** (similar to CHECK) – check multiple tables
  CREATE ASSERTION <assertion name> CHECK (NOT EXISTS (SELECT …))

Add a constraint that no more than 3 ships of the same model can participate in a single battle (5 marks).

```
CREATE ASSERTION battleModelCap as CHECK
(NOT EXISTS
  (SELECT *
  FROM Ships S, Outcomes O
  WHERE S.shipName = O.shipName
  GROUP BY O.battleName, S.model
  HAVING COUNT(*) > 3));
```