Introduction

Relation: Set of tuples; Relational database scheme: set of schemas; Relational database: collection of tables

Integrity Constraints

Domain constraints, key constraints, foreign key constraints, other general constraints

Key Constraints

Superkey: subset of attributes in a relation that uniquely identifies its tuples

Key: a superkey that satisfies the additional property → *not null* & *no proper subset of a key is a superkey*

- ➔ Minimal subset of attributes that uniquely identifies its tuples
- ➔ Can have multiple in a relation (candidate keys) but one is selected as the primary key

Foreign key: refers to the primary key of a second relation

- ➔ Each foreign key value in referencing relation must either *appear as primary key* value in referenced relation or be a null value
  * Referencing and referenced relations could be the same relation

Relational Algebra

Unary operators (input: one relation)

Closure of relation: unary operator takes in a relation as input and gives a relation as output

Selection σ	Projection π	Renaming ρ
Selects tuples from relation R that satisfies condition C * *won't affect columns* * *may remove rows* * *won't add rows* * *won't reorder/rename columns*	Projects attributes given by a list L of attributes from relation R * *may remove columns, rows* * *won't add columns, rows* * *may reorder columns* * *won't rename columns* * *o/p is a set (no duplicates)*	$\rho_s(B_1, B_2 \ldots B_n)(R)$ renames $R(A_1 \ldots A_n)$ to $S(B_1, B_2 \ldots B_n)$ * *won't add/remove rows, columns* * *won't reorder columns* * *may rename columns*

Binary operators (input: two relations)

Closure of relation: binary operators takes in two relations as inputs and gives a relation as output

Cross-product ×	Union ∪	Intersection ∩	Set-difference −
Returns a relation with schema (A, B, C, X, Y)	Returns a relation containing all tuples that occur in R, S or both	Returns a relation containing all tuples that occur in R & S	Returns a relation containing all tuples that occur in R but not S

SQL

Null values

→ Result of comparison operations involving NULL is UNKNOWN (eg. ≤ ≥ =)

→ Result of arithmetic operations involving NULL is NULL (eg. + / - *)

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

x	y	x IS DISTINCT FROM y
null	null	FALSE
non-null	null	TRUE
non-null	non-null	x <> y

* use IS NULL to check for NULL

$\pi_{\text{select-list}}(\sigma_{\text{condition}}(r_1 \times r_2 \times \ldots \times r_n))$

projection → SELECT; selection → WHERE; cross product → FROM

SQL Syntax

Create table

```
CREATE TABLE [IF NOT EXISTS] table_name ( [
    { column_name data_type
            [ column_constraints[…]]
            | table_constraints }
    [, …]
]);
```

Drop table

DROP TABLE [IF EXISTS] table_name

Table modification

INSERT INTO table_name [(column)] VALUES ( )

DELETE FROM table_name [WHERE …]

UPDATE table_name SET … [WHERE …]

Alter table

ALTER TABLE table_name ALTER COLUMN column_name DROP DEFAULT;

ALTER TABLE table_name DROP COLUMN column_name;

ALTER TABLE table_name ADD COLUMN column_name column_type;

Constraints

→ PRIMARY KEY

→ REFERENCES … [ON DELETE action] [ON UPDATE action]

action → NO ACTION / RESTRICT / CASCADE / SET DEFAULT / SET NULL

→ NOT NULL

→ UNIQUE

→ CHECK

→ DEFAULT

Query

SELECT [DISTINCT] select_list

FROM from_list

[WHERE condition]

Renaming Column

SELECT 'Price of' || pizza || ' is' || round(price/1.3) || 'USD' AS menu → Price of Diavola is 18 USD

Pattern Matching

attr LIKE pattern

→ underscore (_) : match any single character

→ percent (%) : match sequence of 0 or more characters

Conditional Expressions

```
→ CASE [expression]
    WHEN condition THEN result
    [ WHEN … ]
    [ ELSE result ]
  END
```

→ NULLIF (result, 'absent')

→ COALESCE (arg1, arg2, … argn)

Returns the first non-null value in its argument, & returns null if all null

Multi-Relation Queries

→ Set operations ['ALL' preserves duplicate records]

- $Q_1 \cup Q_2$ — Q_1 UNION Q_2 — Q_1 UNION ALL Q_2
- $Q_1 \cap Q_2$ — Q_1 INTERSECT Q_2 — Q_1 INTERSECT ALL Q_2
- $Q_1 - Q_2$ — Q_1 EXCEPT Q_2 — Q_1 EXCEPT ALL Q_2

→ Join

- Inner Join — aka join — *eliminates all with no match (null)
- Left Join — *all rows/values from left table preserved
- Right Join — *all rows/values from right table preserved
- Outer Join — aka full join — * gets all dangling tuples
- Natural Join — can be put with left/right

Views (a virtual relation that can be used for querying)

CREATE VIEW view_name [(column1, column2, …)] AS

Aggregate Functions (computes a single value from a set of tuples)

→ MIN(_), MAX(_), AVG(_), SUM(_), COUNT(_)

* take note that COUNT(_) counts null values too, COUNT(*) is to count number of rows

→ ORDER BY column1 [ASC | DESC]

[, column2 [ASC | DESC] […]]

→ LIMIT{ number | ALL } → top n rows

→ OFFSET number → removes top n rows

→ GROUP BY column1 [, column2[…]]

Divides the rows into groups such that aggregate functions can be applied to each group

* In a query, two tuples belong to the same group if the values are NOT DISTINCT

Remember that 2 null values are non-distinct!

* For each column A in relation R that appears in SELECT, one of the following conditions must hold:

1. Column A appears in the GROUP BY clause
2. Column A appears in aggregated expression in SELECT
3. The primary/candidate key of R appears in the GROUP BY clause

→ HAVING

Replaces "WHERE" for aggregated functions

Condition is same as GROUP BY but SELECT is replaced by HAVING

Subqueries (inner/nested queries)

* a tuple variable declared in a subquery/query Q can be used only in Q & any subquery nested in Q

* if a tuple variable is declared both locally as well as in an outer query, the local declaration applies

→ Scalar subqueries return at most one tuple with one column

→ Common Table Expressions (a temporary named result set that can be queried)

```
WITH
    cte1 AS (subquery1) [,
    cte2 AS (subquery2) […] ]
query
```

→ Types of subqueries

- EXISTS
  Returns true if result subquery is non-empty
- IN
  Subquery must return exactly one column, else if empty, false
- ANY / SOME
- ALL

Universal Quantification

→ ∀f ⟹ ~~(∀f) ⟹ ~(∃~f)

SELECT _ FROM _

WHERE NOT EXISTS (

SELECT _ FROM _

WHERE _

AND NOT EXISTS (_insert subquery_)

);

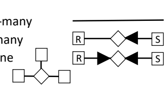
Entity Relationship Data Model

→ Entity, Attribute, Entity Set, Relationship, Relationship Sets

Keys

→ Each entity set has a key, attributes that form a primary key are underlined

Key Constraints



- Many-to-many — ≥ 0
- One-to-many — Each S ≤ 1 R, each R ≥ 0 S
- One-to-one — Each S ≤ 1 R, each R ≤ 1 S
- N-ary

Participation Constraints

- Partial participation constraints
- Total participation constraints

* if => means = 1

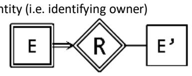
Roles

→ Used when one entity set appears ≥ 2 times in a relationship set

Weak Entity Sets

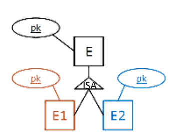
→ An entity set that does not have its own key (i.e. its existence is dependent on owner entity's existence)

→ Can only be uniquely identified by considering the primary key of another entity (i.e. identifying owner)



- Must be many-to-one relationship from WES to owner ES
- WES must have total participation in identifying relationship
- Partial key of a WES is a set of attributes of weak entity sets that uniquely identifies a weak entity for owner

IS-A Hierarchies

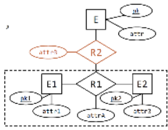


→ Subclass-superclass relationship

- Overlap constraints: satisfied if entity in superclass could belong to multiple subclasses
- Covering constraints: satisfied if every entity in a superclass has to belong to some subclass

* typically not reflected in ER diagram

Aggregation



→ When a relationship with corresponding entities is aggregated into a higher level entity

CREATE TABLE R2 (pk type REFERENCES E, pk1 type, pk2 type, attr type,

PRIMARY KEY (pk, pk1, pk2)

FOREIGN KEY (pk1, pk2) REFERENCES R1(pk1, pk2)

);

Stored Procedure, Functions

Transaction → consists of one or more update/retrieval operations

BEGIN code { COMMIT | ROLLBACK } (must end with either)

→ COMMIT — success → update database

→ ROLLBACK — failure → restore database to state before BEGIN

ACID Properties

- Atomicity → either all or none of the effects of the transactions are reflected in the DB
- Consistency → user-defined property should be preserved (constraints)
- Isolation → isolated other concurrent transaction executions, can run concurrently
- Durability → commit is permanent

Constraint Check

By default, constraints are checked at the end of each SQL statement execution

→ A violation will cause the statement to be rollbacked which can be deferred to the end of the transaction

- ➔ Default: NOT DEFERRABLE / DEFERRABLE INITIALLY IMMEDIATE
- ➔ Wait till transaction completes before checking constraint: DEFERRABLE INITIALLY DEFERRED

Stored Procedures and Function

A procedure/function/subroutine that is available to applications that access a DBMS and is stored in the DB

Create Function

➔ Stored functions may have return types

CREATE [OR REPLACE] FUNCTION

func_name ([**arg1 type1** [, **arg2 type 2** […]]])

RETURNS **ret_type** AS **func_def**

LANGUAGE **lang**;

➔ *Eg. CREATE OR REPLACE FUNCTION hello_world()*

RETURNS CHAR(11) AS

$$ BEGIN RETURN 'Hello World'; END; $$

LANGUAGE plpgsql;

Create Procedure

➔ Stored procedures may not have return type (mainly deals with side-effects)

CREATE [OR REPLACE] PROCEDURE

proc_name ([**arg1 type1** [, **arg2 type2** […]]])

AS **proc_def**

LANGUAGE **lang**;

Remove Function/Procedure

DROP { FUNCTION | PROCEDURE } [IF EXISTS] **name**;

Declaration (before BEGIN)

DECLARE **var type**;

(below are for PLPGSQL)

Assignment

var := **expr**;

Selection (IF statement)

IF **cond** THEN **stmt;**

[ELSIF **cond** THEN **stmt** […]]

[ELSE **stmt**];

Iteration

➔ WHILE **cond** LOOP **stmt**; END LOOP;

➔ FOR **var** IN (**expr …**) LOOP

{ **stmt;** | EXIT; | EXIT WHEN cond; } […]

END LOOP;

➔ LOOP { **stmt;** | EXIT; | EXIT WHEN **cond;** } […]

END LOOP;

Operations

Arithmetic and bitwise

- ➔ *Simple* +, -, *, /, %, ^
- ➔ *Bitwise* &, |, # (xor), ~(not), <<, >>
- ➔ *Others* |/ (square root), ||/ (cube root), @ (absolute value), ! (factorial postfix), !! (factorial prefix)

Comparison

- ➔ *Simple* <, >, <=, >=, =, <>

Functional Dependencies

Constraints on schemas that specify that the values for certain set of attributes determine unique values for another set of attributes (i.e. uniquely identifies)

Notations

We use $R(A_1, A_2, \ldots, A_n)$ to denote relation schema with n attributes

We use lowercase letter a, b, … except r to denote subsets of attributes in R

- Let $a, b \subseteq R$ and $A_i, A_j \in R$
  - We use ab — to denote $a \cup b$ — union
  - We use A_iA_j — to denote $\{A_i, A_j\}$ — set
  - We use A_ib — to denote $\{A_i\} \cup b$ — union
  - We use $b - A_i$ — to denote $b - \{A_i\}$ — set diff

Definition

Let r be a relation instance of relation schema R

- ➔ r satisfies FD a → b if for every pair of tuples t_1 and t_2 in r such that $\pi_a(t_1) = \pi_a(t_2)$, it is also true that $\pi_b(t_1) = \pi_b(t_2)$
- ➔ an FD *f* holds on R if and only if for any relation instance r of R, r satisfies R
- ➔ r is a legal instance of R if r satisfies all FDs that holds on R

Trivial vs Non-Trivial

For FD a → b,

Trivial: b is a subset of a

Non-Trivial: b is not a subset of a

Completely Non-Trivial: a and b have completely different attributes
* completely non-trivial implies non-trivial
* an empty set is a subset of everything → trivial

Closure

- → F ⊨ G if F ⊨ g for all g ∈ G
- → The closure of F (F^+) is the set of all FDs implied by F

F is equivalent to G if $F^+ = G^+$, i.e. F ≡ G, F ⊨ G and G ⊨ F → closures are the same
* Anything trivial is always true/is implied

Armstrong's Axioms

- Reflexivity — if b ⊆ a then a → b
- Augmentation — if a → b then ac → bc
- Transitivity — if a → b then b → c then a → c

Extension

- Union — if a → b then a → c then a → bc
- Decomposition — if a → b then a → b' where b' ⊆ b
  - Specific case — if a → bc then a → b and a → c

Superkeys, keys and prime attributes

Superkey: A set of attributes *a* is a superkey of schema R (with FDs F) if F ⊨ a → R

Prime Attributes: An attribute A ∈ R is a prime attribute if A is contained in some key of R

Attribute Closure

Given a set of attribute *a*, other attributes that we can know is called *attribute closure* of *a*

- → The closure of a (wrt F) is a^+ = {A ∈ R | F ⊨ a → A}

Algorithm 1 (get attribute closure)

Input A set of attributes a ⊆ R and a set of FDs **F** on **R**

Output a^+ (wrt F)

1. initialize θ = *a*
2. while (there exists some FD b → c ∈ F such that b ⊆ θ and c ⊊ θ)
3. θ = θ ∪ c
4. return θ

Example: let $F = \{A \to C, B \to C, CD \to E\}$
- Show that $F \vdash AD \to E$
1. initialize $\Rightarrow \theta = AD$
2. with $A \to C$ $\Rightarrow \theta = ACD$
3. with $CD \to E$ $\Rightarrow \theta = ACDE$
4. therefore $AD^+ = ACDE$
   - thus $F \vdash AD \to E$

Minimal Covers

Some FDs are *redundant* → can be removed.

Smallest set of FDs is called minimal cover (may have ≥ 1 unique minimal covers)

A minimal cover:

- Every FD is of the form a → A (single attribute on the right)
- For each FD a → A in G, a has no redundant attributes
- There are no redundant FDs in G
- G and F are equivalent

* Each set of FDs has at least one minimal cover (trivially, it is itself!)

1. Given an FD a → b, an attribute A ∈ a is a redundant attribute in FD if:
   - (F – {a → B}) ∪ {(a – A) → B} is equivalent to F
   - i.e. having (a – A) → B instead of a → B does not change F^+
2. Given an FD *f* ∈ F, *f* is a redundant FD if
   - F – *f* is equivalent to F

Algorithm 2 (get a minimal cover)

Input A set of FDs **F**

Output A minimal cover for F

initialize $G = \emptyset$
for each (FD $a \to B_1 \ldots B_n$ in F)
  $G = G \cup \{a \to B_i \mid i \in [1, n]\}$ — Decompose
for each (FD $a \to B$ in G)
  initialize $a' = a$
  for each ($A \subset a$) do
    if (B in $(a' - A)^+$ w.r.t. G) then
      replace $a' \to B$ in G by $(a' - A) \to B$
      $a' = a' - A$ — Remove redundant attribute
for each (FD $a \to B$ in G)
  if (B in a^+ w.r.t. $G - \{a \to B\}$) then
    remove $a \to B$ from G — Remove redundant FDs
return G

- Example: $F = \{ABCD \to E, E \to D, A \to B, AC \to D\}$
  - Find a minimal cover of F
  - Steps
    - Decompose FDs: already decomposed
    - Remove redundant attributes: start with $G = \{ABCD \to E, E \to D, A \to B, AC \to D\}$
      1. A in $ABCD \to E$ is non-redundant — $BCD^+ = BCD$ w.r.t. G
      2. B in $ABCD \to E$ is redundant — $ACD^+ = ABCDE$ w.r.t. G
         - $G = \{ACD \to E, E \to D, A \to B, AC \to D\}$
      3. C in $ABCD \to E$ is non-redundant — $AD^+ = ABD$ w.r.t. G
      4. D in $ABCD \to E$ is redundant — $AC^+ = ABCDE$ w.r.t. G
         - $G = \{AC \to E, E \to D, A \to B, AC \to D\}$
      5. A in $AC \to D$ is non-redundant — $C^+ = C$ w.r.t. G
      6. C in $AC \to D$ is non-redundant — $A^+ = AB$ w.r.t. G
    - Remove redundant FDs: start with $G = \{AC \to E, E \to D, A \to B, AC \to D\}$
      1. $AC \to E$ is non-redundant — $AC^+ = ABCD$ w.r.t. $G - \{AC \to E\}$
      2. $E \to D$ is non-redundant — $E^+ = E$ w.r.t. $G - \{E \to D\}$
      3. $A \to B$ is non-redundant — $A^+ = A$ w.r.t. $G - \{A \to B\}$
      4. $AC \to D$ is redundant — $AC^+ = ABCDE$ w.r.t. $G - \{AC \to D\}$
    - Minimal cover is $G = \{AC \to E, E \to D, A \to B\}$

Decomposition

The decomposition of schema R is a set of schemas {$R_1, R_2, \ldots, R_n$} (called fragments) such that

- R_i ⊆ R for each R_i (each fragment is simpler than the original schema)
  * need not be a proper subset
- R = $R_1 \cup R_2 \cup \ldots \cup R_n$ (no attributes are missing)

Lossless-join Decomposition

{$R_1, R_2, \ldots, R_n$} is a lossless-join decomposition wrt F if no information is lost by performing a join

Lemma 1

If {$R_1, R_2, \ldots, R_n$} is a decomposition of R, then for any relation r or R

A natural join of the decomposition of R will lead to a superset of R

Lossy-join decomposition will produce more tuples! → gain data, lose information

Theorem 1

The decomposition of R with FDs F into {R_1, R_2} is a lossless-join decomposition wrt F if

- F ⊨ $R_1 \cap R_2$ → R_1

  OR
- F ⊨ $R_1 \cap R_2$ → R_2

i.e. when $R_1 \cap R_2$ is a superkey for either R_1 or R_2

Corollary 1

If a → b is a completely non-trivial FD that holds on R, then the decomposition of R into {R – b, ab} is a lossless-join decomposition

- Since (R – b) ∩ ab = a and a → b so a → ab

Theorem 2

If {$R_1, R_2, \ldots, R_n$} is a lossless-join decomposition of R, and {$R_{1,1}, R_{1,2}$} is a lossless-join decomposition of R_1m then {$R_{1,1}, R_{1,2}, R_2, \ldots, R_n$} is a lossless-join decomposition of R
* only works on splits into 2

To check for lossless-join, must check for lossless-join of all the combinations of the decompositions. If any of them are not lossless-join, can conclude it is not lossless-join.

Dependency-preserving Decomposition

Only use attributes that appear in the decomposition

Decomposition {$R_1, R_2, \ldots, R_n$} of R is dependency-preserving if

- ($F_{R1} \cup F_{R2} \cup \ldots \cup F_{Rn}$) is equivalent to F
  - ($F_{R1} \cup F_{R2} \cup \ldots \cup F_{Rn}$) ≡ F
  - $(F_{R1} \cup F_{R2} \cup \ldots \cup F_{Rn})^+ = F^+$
  - ($F_{R1} \cup F_{R2} \cup \ldots \cup F_{Rn}$) ⊨ F ^ F ⊨ ($F_{R1} \cup F_{R2} \cup \ldots \cup F_{Rn}$)
- Guarantees that for each update to a decomposed relation, FD violations can be detected w/o computing joins

Algorithm 3 (find FDs of a decomposition)

Input A set of attributes a ⊆ R and a set of FDs **F** on **R**

Output F_a

1. Initialise θ = *a*
2. for each (b ⊆ a such that b ≠ ∅)
3. θ = θ ∪ {b → (b^+ ∩ a)} // w.r.t F
4. return θ

Example: Let $R(A, B, C)$ with FDs $F = \{A \to B, B \to C, C \to B\}$
- Compute F_{BC}
  - initialize $\to \theta = \emptyset$
  - let $b = B$, $B^+ = BC$ $\Rightarrow \theta = \{B \to BC\}$
  - let $b = C$, $C^+ = BC$ $\Rightarrow \theta = \{B \to BC, C \to BC\}$
  - let $b = BC$, $BC^+ = BC$ $\to \theta = \{B \to BC, C \to BC, BC \to BC\}$

Lemma 2

For every decomposition {$R_1, R_2, \ldots, R_n$} of R,

- F ⊨ {$F_{R1} \cup F_{R2} \cup \ldots \cup F_{Rn}$}
- By definition, F_{Ri} = { b → c ∈ F^+ | bc ⊆ R}
- For all b → c in F_{Ri}, we also have b → c in F^+

Hence, we only need to check if ($F_{R1} \cup F_{R2} \cup \ldots \cup F_{Rn}$) ⊨ F

Algorithm 4 (check if decomposition is dependency-preserving)

Input A decomposition $\{R_1, \ldots, R_n\}$ of R with FDs F

Output *YES* (if dependency-preserving) or *NO* (otherwise)

for each ($R_i \in \{R_1, \ldots, R_n\}$)
  compute F_{R_i}
let $G = F_{R_1} \cup \cdots \cup F_{R_n}$
for each (FD $a \to b \in F$)
  compute a^+ w.r.t. G
  if ($b \not\subseteq a^+$) then return *NO*
return *YES*

Normal Forms

Restricts the set of data dependencies that are allowed to hold on a schema to avoid certain undesirable redundancy and update problems in database

Boyce-Codd normal form (BCNF)

R is in BCNF if for every FD a → A in F either

1. a → A is trivial *(a ∈A)* OR
2. a is a superkey of R

R violates BCNF if both are not satisfied

A decomposition of R is in BCNF if all of its decompositions is in BCNF

- → if every decomposed element is a single element, i.e. fulfills BCNF, then it fulfills BCNF by default (everything is trivial). BUT it might not be lossless-join/dependency-preserving

Checking if a relation schema R_i is in BCNF, check if there exists some non-trivial FD *f* which holds on R_i that violates BCNF

1. If F is the set of FDs that hold on R_i (i.e. R_i is not from a decomposition)
   - → check for any violating non-trivial FD in F
   - → * if there are no non-trivial FD, R_i is in BCNF
2. If F is the set of FDs that hold on R, and R_i is a decomposed relation schema of R
   - → check for any violating non-trivial FD in F_{Ri} (check if any FD in the ***projection*** of R_i violates BCNF too)

Lemma 3

For any relation schema R with exactly two attributes, R is in BCNF

Algorithm 5 (checks if a schema violates BCNF)

Input F is a set of FDs that hold on schema R and R_i is either R or a decomposed schema of R

Output A completely non-trivial FD that violates BCNF if R_i is not in BCNF; otherwise *null*

if (R_i has exactly 2 attributes)
  return null
for each ($a \subseteq R$ such that $a \neq \emptyset$)
  let $X = a^+ \cap R_i$ w.r.t. F // compute $a \to X$
  if ($a \subset X \subset R_i$)
    return $a \to (X - a)$
return null

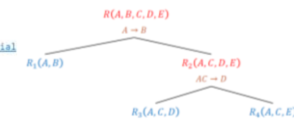
$a \subset X$ implies non-trivial (since trivial FD means $a = X$)

$X \subset R_i$ implies non-superkey (superkey means $X = R_i$)

Algorithm 6 (*lossless-join, non dependency-preserving* BCNF decomposition)

- Input Schema R with FDs F
- Output A lossless BCNF decomposition of R

initialize $\delta = \emptyset$; $i = 1$; $\theta = \{R\}$
while ($\theta \neq \emptyset$)
  remove some R' from θ
  let f = Algorithm#5(F, R')
  if (f = null) then $\delta = \delta \cup \{R'\}$ // in BCNF
  else
    let f be $a \to b$ // completely non-trivial
    let $c = R' - b$
    $\theta = \theta \cup \{R_i(ab), R_{i+1}(c)\}$ // decompose
    $i = i + 2$
return δ



At the worst-case, each fragment will have exactly 2 attributes

3rd normal form (3NF)

R is in 3NF if for every FD a → A in F either

1. a → A is trivial *(a ∈A)* OR
2. a is a superkey of R OR
3. A is a prime attribute

R violates 3NF if all 3 are not satisfied

A decomposition of R is in 3NF if all of its decompositions is in 3NF

Every decomposition in BCNF is definitely in 3NF

Algorithm 7 (lossless-join, dependency-preserving decompositions in 3NF)

Minimal cover input as it guarantees all FD to have no redundancy → all FDs are completely non-trivial

- Input Schema R with FDs F which is a *minimal cover*
- Output A lossless and dependency-preserving 3NF decomposition of R

initialize $\delta = \emptyset$
apply union rule to combine FDs in F
let $G = \{f_1, f_2, \ldots, f_n\}$ be the resultant set of FDs
for each (FD f_i of the form $a_i \to b_i$ in G)
  create a relation schema $R_i(a_i b_i)$ for FD f_i
  insert $R_i(a_i b_i)$ into δ
choose a key K of R and insert $R_{n+1}(K)$ into δ
remove redundant relation schema from δ
  $\Rightarrow$ delete R_i from δ if $\exists R_j \in \delta \cdot i \neq j \wedge R_i \subseteq R_j$
return δ

Since minimal cover is non-deterministic, 3NF is non-deterministic as well
* but always try to start with BCNF first and if can't, then use 3NF