Relational Design Theory

Functional Dependencies

Generalization of notion of keys

Relational design by decomposition

- To design a better schema, which minimizes the possibility of anomalies

Data storage

- Compression schemes based on functional dependencies can be used

Reasoning about queries

- Optimization of queries

Example:

- Student(SSN, sName, address, HScode, HSname, HScity, GPA, priority)
- Suppose priority is determined by **GPA**.
- Two tuples with the same GPA have the same priority.
- ∀ t, u ∈ Student: t.GPA = u.GPA => t.priority = u.priority

Definition

A and B are attributes of a relation R A -> B

- A functionally determines B
- ∀ t, u ∈ R: t.A= u.A=> t.B= u.B

A1, A2, ..., An and B1, B2, ..., Bn are attributes of relation R A1, A2, ..., An -> B1, B2, ..., Bn

- ∀ t, u ∈ R: t.[A1, A2, ..., An] = u.[A1, A2, ..., An]=> t.[B1, B2, ..., Bn]= u.[B1, B2, ..., Bn]

Identifying FDs

- Based on knowledge of real world
- All instances of relation must adhere
- You can check if an FD is violated by examining a single instance
- However, you cannot prove that an FD is part of the schema by examining a single instance
    - This would require checking every valid instance

Trivial Functional Dependecy

A -> B is a trivial dependency if B is inside A. Otherwise, it is a non-trivial functional dependency.

Completely Non-trivial Functional Dependecy

A non-trivial functional dependency A -> B, where the intersection of A and B is empty. These are the ones we are most interested in.

Splitting Rule

A -> B1, B2, ..., Bn => A -> B1, A -> B2, ..., A -> Bn

Combining Rule

A -> B1, A -> B2, ..., A -> Bn => A -> B1, B2, ..., Bn

Trivial-Dependency Rules

A -> B => A -> A $\cup$ B

- We can add to the right side what's already on the left side

A -> B => A -> A $\cap$ B

- Implied by the splitting rule

Transitive Rule

A -> B $\wedge$ B -> C => A -> C

Closure, Superkeys and Keys

Closure of attributes

Given a relation, FDs, set of attributes A, find all B such that A -> B.

A^+ is the closure of A.

- Finding the set of attributes functionally determined by $\{A1, ..., An\}^+$.

Algorithm

- Start with {A1, ..., An}
- Repeat until no change
    - If X -> Y and X in set, add Y to set.

Closure and Keys

If A^+ contains all attributes of R, the A is a key of R.

How can we find all keys given a set of FDs?

- Consider every subset of attributes and compute its closure to see if it determines all attributes
- To increase efficiency, consider substets in increasing order. (if A is a key, all supersets of A will be a key)

Superkeys and keys

A superkey is a set of attributes such that for any other attribute, we can determine it through that set. All attributes are determined by a superkey.

A key is a minimal superkey

- Meaning that no subset of a key is also a superkey
- Also named candidate key

Primary key is one and **only one** of the keys.

Inferring FDs

S1 and S2 are sets of FDs.

S2 "follows from" S1 if every relation instance satisfying S1 also satisfies S2.

Example:

- S2: {SSN -> priority}
- S1: {SSN -> GPA, GPA -> priority}

How to test if A -> B follows from S?

- Compute A^+ based on S and check if B is in the set.
- Armstrong's Axioms
    - Set of rules that are what's called complete.
    - If one thing about functional dependecies can be proved from another, then it can proved using the Armstrong's Axioms

Goal: Find minimal set of completely nontrivial FDs such that all FDs that hold on the relation follow from the dependencies in this set.

Normal Forms

1st Normal Form (1NF)

The domain of each attribute contains only atomic values and the value of each attribute contains only a single value from that domain.

2nd Normal Form (2NF)

1NF and no attribute not prime is functionally dependent on a proper subset of a

candidate key. An attribute that is member of some key is prime.

Boyce-Codd Normal Form (BCNF)

Relation R with FDs is in BCNF if for each nontrivial A->B, A is a (super)key.

3rd Normal Form (3NF)

2NF and all non-prime attributes are functionally dependent of every candidate key in a non-transitive way

OR

Relation R is in 3NF if, for each nontrivial A->B,

- A is a (super)key or
- B consists of prime attributes only

Decomposition

R(A1, ..., An) = A

- R1(B1, ..., Bn) = B
- R2(C1, ..., Cn) = C

R1 and R2 are a decomposition of R if

- B united with C is equal to A
- R1 joined with R2 is equal to R

BCNF decomposition algorithm

Input: relation R + FDs for R

Output: decomposition of R into BCNF relations with "lossless join"

Compute keys for R

Repeat until all relations are in BCNF:

- Pick any R' with A->B that violates BCNF
- Decompose R' into R1(A, B) and R2(A, rest)
- Compute FDs for R1 and R2
- Compute keys for R1 and R2

BCNF shortcomings

Dependency preservation is not guaranteed

- No guarantee that all original dependencies can be checked on decomposed relations

- This may require joins of those relations in order to check them Various ways to handle so that decompositions are all lossless/no FDs lost
- For example, 3NF Usually a tradeoff between redundancy / data anomalies and FD preservation BCNF still most common
- With extra steps to keep track of lost FDs

3NF Decomposition Algorithm

Input: Relation R + set F of DFs for R

Output: decomposition of R into 3NF relations with "lossless join" and "dependency preservation"

1. For each FD X -> A in G, create a relation R'(X, A)
    - Previously, merge FDs with equal left sides
2. If none of the relations of step 2 is a superkey for R, add another relation for a key for R

BCNF and 3NF decomposition

BNCF decomposition

- Assures lossless joins
- Dependency preservation is not always possible 3NF decomposition
- Assures lossless joins and dependency preservation

Summary

Designing a database schema

- Usually many designs possible
- Some are (much) better than others
- How do we choose? Very nice theory for relational database design
- Normal forms - "good" relations
- Design by decomposition
- Usually intuitive and works well
- Some shortcomings
    - Dependency enforcement
    - Query workload
    - Over-decomposition