1 Relational Databases

1.1 Creating relations (tables) in SQL

1.1.1 Create Table

```
Create Table Student(ID, name, GPA, photo)
```

```
Create Table College(
        name string,
        state char(2),
        enrollment integer
)
```

1.1.2 Querying Relational Databases

Steps in creating and using a (relational) database

1. Design schema; create using DDL
2. "Bulk load" initial data
3. Repeat: execute queries and modifications

Databases support ad-hoc queries in high-level language

Queries return relations:

- compositional
- closed - closure - when you get back the same type as you queried (table)

Query languages:

- Relational Algebra - formal
- SQL - actual/implemented

2 SQL

3 Relational Design Theory

3.1 Relational design overview

3.1.1 Motivation & overview

Example: College application info

- SSN and name
- Colleges applying to
- High schools attended (with city)
- Hobbies

```
Apply(SSN, sName, cName, HS, HScity, hobby
```

Populating:

```
123 Ann Stanford PAHS P.A. tennis
123 Ann Berkeley PAHS P.A. tennis
123 Ann Stanford PAHS P.A. trumpet
```

need 12 tupples to represent this

Design "anomalies"

- Redundancy
- Update anomaly - can update facts differently in different places, trumpet can be named cornet
- Deletion anomaly - can do complete deletion of somebody in db. If we delete all people surfing

Better alternative

```
Student(SSN, sName)
Apply(SSN, cName)
HighSchool(SSN, HS)
Located(HS, HScity)
Hobbies(SSN, hobby)
```

No anomalies, we can reconstruct original data.

One possible problem is that **HS** alone is not a key, so if we brake up **HS** and **HScity**, we can no longer identify the high school, we can do:

```
Student(SSN, sName)
Apply(SSN, cName)
HighSchool(SSN, HS, HScity)
Hobbies(SSN, hobby)
```

Suppose student doesn't want to reveal hobbies to all of the colleges, we do this:

```
Student(SSN, sName)
Apply(SSN, cName, hobby)
HighSchool(SSN, HS, HScity)
```

Design by decomposition

- Start with "mega" relations containing everything
- Decompose into smaller, better relations with same info.
- Can do decomposition automatically

Automatic decomposition

- "Mega" relations + properties of the data
- System decomposes based on properties
- Final set of relations satisfies **normal form**
  - no anomalies, no lost information

3.1.2 Properties and Normal Forms

For specification of properties:

- Functional dependencies⇒Boyce-Codd Normal Form
  - Stricter than 4NF
- Multivalued dependencies⇒Fourth Normal Form - adds to BCNF
  - 1NF - Relations are real relations with atomic values in each cell
  - 2NF - Specifies something about the way relations are structured with respect to their keys
  - 3NF - Slight weakening of BCNF
  - **BCNF** - most common NF used if we have **functional dependencies only**
  - **4NF** - most common NF used if we have **functional dependencies and multivalued dependencies**

3.1.3 Functional Dependencies and BCNF

Apply(SSN, sName, cName)

This relation has all:

- Redundancy; Update & Deletion Anomalies
- Storing SSN-sName pair once for each college

Functional Dependency SSN→sName

- Same **SSN** always has same **sName**
- Should store each **SSN**'s **sName** only once

BCNF

- If **A**→**B** then **A** is a key
  - meaning we have one tuple for each attribute

The functional dependency would tell us to:

Decompose:

Student(SSN, sName)
Apply(SSN, cName)

3.1.4 Multivalued Dependencies and 4NF

Apply(SSN, cName, HS)

- Redundancy; Update & Deletion Anomalies
- Multiplicative effect
  - C colleges, H highschools⇒$C \cdot H$ tupples, we'd like to have $C + H$, because then we'd capture each piece of info only once.
- Not addressed by BCNF: No functional dependencies, if there are no functional dependencies, then the relation is automatically in BCNF. It's not necessary that every instance of **SSN** is associated with a single **cName** or a single **HS**.

Multivalued Dependency SSN↠cName

- Given **SSN** has every combination of **cName** with **HS** that's associated with that **SSN**.
- Should store each **cName** and each **HS** for an **SSN** once

That's what 4NF will solve. If $\mathbf{A} \twoheadrightarrow \mathbf{B}$ then **A** is a key.

Decompose:

Apply(SSN, cName)
HighSchool(SSN, HS)

Consider the relation

StudentInfo(sID, dorm, courseNum)

Multivalued dependencies:

- sID↠dorm
- sID↠courseNum

4NF requires the left-hand side of multivalued dependencies to be a key for the relation. **sID** is not a key for **StudentInfo**.

3.2 Functional dependencies

Functional dependencies are generally useful concept

- Data storage -compression
- Reasoning about queries - optimization

Example: College application info.

Student(SSN, sName, address, HScode, HSname, HScity, GPA, priority)
Apply(SSN, cName, state, date, major)

Suppose priority is determined by GPA

say, if

$GPA > 3.8$ then $priority = 1$

$3.3 < GPA \leq 3.8$ then $priority = 2$

$GPA \leq 3.3$ then $priority = 3$

If this is guaranteed in our data, then we can say that two tuples with same **GPA** have the same **priority**.

Formally

$$\forall_{t,u \in Student} t.GPA = u.GPA \Rightarrow t.priority = u.priority$$

$$GPA \rightarrow priority$$

meaning that **GPA** determines **priority**.

Generally, for a relation $\mathcal{R}$:

$$\forall_{t,u \in R} t.A = u.A \Rightarrow t.B = u.B$$

$$\mathcal{R} \qquad A \rightarrow B$$

more generally, for a set of attributes of $\mathcal{R}$

$$\forall_{t,u \in R} t\left[A_1, A_2, \ldots, A_n\right] = u\left[A_1, A_2, \ldots, A_n\right] \Rightarrow t\left[B_1, B_2, \ldots, B_m\right] = u\left[B_1, B_2, \ldots, B_m\right]$$

$$\text{®} \qquad A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m$$

The two tuples **t** and **u** have the same values for all attributes $A_1, A_2, \ldots, A_n$ and if they do then they will also have the same values for $B_1, B_2, \ldots, B_m$.

For simplicity, the notation for $B_1, B_2, \ldots, B_m$ will be $\overline{B}$, same for A.

Functional Dependency

- Basend on knowledge of real world data that's being captured.
- All instances of a relation must adhere.

Say we have attributes $\bar{A} \rightarrow \bar{B}$ and $R(\bar{A}, \bar{B}, \bar{C})$

If we have 2 tuples who's values are the same then:

$\bar{A}$	$\bar{B}$	$\bar{C}$
$\bar{a}$	$\bar{b}$	$\bar{c_1}$
$\bar{a}$	$\bar{b}$	$\bar{c_2}$

values $\bar{C}$ don't have to be the same

For the above example

SSN functionally determines sName

$SSN \rightarrow sName$

Student doesn't change address

$SSN \rightarrow address$

Same HSname and HScity for every HScode

$HScode \rightarrow HSname, HScity$

There's no two high schools with the same name and city

$HSname, HScity \rightarrow HScode$

$SSN \rightarrow GPA$

$GPA \rightarrow priority$

$SSN \rightarrow priority$

For Apply relation, it may depend on real world constraints:

If every college has a particular single date on which it receives its application. College names are unique

$cName \rightarrow date$

Students are allowed to apply to a single major at each college they apply to

$SSN, cName \rightarrow major$

Students only allowed to colleges in one state

$SSN \rightarrow state$

3.2.1 Functional Dependencies and Keys

- Relation with no duplicates $R(\bar{A}, \bar{B})$
- Suppose $\bar{A} \rightarrow$all attributes

$\bar{A}$	$\bar{B}$
$\bar{a}$	$\bar{b}$
$\bar{a}$	$\bar{b}$
$\vdots$	$\vdots$

We cannot have the same two tuples with the same A values is what it means for A to be a **key**. That's for the case when we have no duplicates in R.

Trivial Functional Dependency

$\bar{A} \rightarrow \bar{B}$ if $\bar{B} \subseteq \bar{A}$

Nontrivial FD

$\bar{A} \rightarrow \bar{B}$ if $\bar{B} \not\subseteq \bar{A}$

If we have some values agreeing in $\bar{A}$ then they will also agree in $\bar{B}$, there are some attributes in $\bar{B}$ that are not a part of $\bar{A}$.

Completely nontrivial FD

$\bar{A} \rightarrow \bar{B}$ if $\bar{B} \cap \bar{A} = \emptyset$

$\bar{B}$ attributes are going to be some part of remaining portion of attributes from $\bar{A}$. We're saying if $\bar{a}$ values are the same then $\bar{b}$ values will be the same.

Rules for Funcional Dependencies

- Splitting Rule
  If we have a set of attributes determining another set of attributes
  $\bar{A} \rightarrow B_1, B_2, \ldots, B_m$
  then we also have this implication that we have A determines $B_1, B_2, \ldots$
  $\bar{A} \rightarrow B_1, \bar{A} \rightarrow B_2, \ldots$
  Can we also split left-hand side?
  $A_1, A_2, \ldots, A_n \rightarrow \bar{B}$
  $A_1 \rightarrow \bar{B}, A_2 \rightarrow \bar{B}, \ldots$
  NO
  say if we have:
  HSname, HScity→HScode
  it is likely not the case that we have:
  HSname→HScode
  or
  HScity→HScode
- Combining rule - an inverse of the splitting rule
  If we have:
  $\bar{A} \rightarrow B_1$
  $\bar{A} \rightarrow B_2$
  $\vdots$
  $\bar{A} \rightarrow B_n$
  $\Rightarrow \bar{A} \rightarrow B_1, \ldots, B_n$
- Trivial-dependency rules
  $\bar{A} \rightarrow \bar{B}$ then $\bar{A} \rightarrow \bar{A} \cup \bar{B}$
  and
  $\bar{A} \rightarrow \bar{B}$ then $\bar{A} \rightarrow \bar{A} \cap \bar{B}$
  which is also implied by the splitting rule.

- Transitive rule
  If $\bar{A} \rightarrow \bar{B}$ and $\bar{B} \rightarrow \bar{C}$ then
  $\bar{A} \rightarrow \bar{C}$

A	B	C	C
$\bar{a}$	b	$\bar{c}$	
$\bar{a}$	b	$\bar{c}$	
$\vdots$	$\vdots$	$\vdots$	

  If we know that the two $\bar{a}$ values are the same, then the two $\bar{b}$ values are the same and if the two $\bar{b}$ values are the same then the two $\bar{c}$ values are the same.

3.2.2 Closure of Attributes

- Given relation, FDs, set of attributes $\bar{A}$
- Find all **B** such that $\bar{A} \rightarrow \bar{B}$
- Notation: $\bar{A}^+$

Algorithm:

Start with $\{A_1, A_2, \ldots, A_n\}$

repeat adding attributes to the closure until we get to the closure:

repeat if $\bar{A} \rightarrow \bar{B}$ and all of $\bar{A}$ is in the set, add $\bar{B}$ to the set.

Closure Example

Student(SSN, sName, address, HScode, HSname, HScity, GPA, priority)

$$\begin{aligned} SSN &\rightarrow sName, address, GPA \\ GPA &\rightarrow priority \\ HScode &\rightarrow HSname, HScity \end{aligned}$$

Suppose that we're interesting in computing the closure of $\{SSN, HScode\}^+$, so in other words we want to find all attributes in the student relation that are determined by these two attributes.

Start with:

$\{\mathbf{SSN}, \mathbf{HScode}\}$

Add attributes that are functionally determined by ones in the set from $SSN \rightarrow sName, address, GPA$:

$\{SSN, HScode, \mathbf{sName}, \mathbf{address}, \mathbf{GPA}\}$

Repeat with $GPA \rightarrow priority$:

$\{SSN, HScode, sName, address, GPA, \mathbf{priority}\}$

Repeat with $HScode \rightarrow HSname, HScity$:

$\{SSN, HScode, sName, address, GPA, priority, \mathbf{HSname}, \mathbf{HScity}\}$

The two attributes: **SSN, HScode**, functionally determine all of the attributes of the Student relation and therefore are the key.

3.2.3 Closure and Keys

Is $\bar{A}$ a key for R?

Compute $\bar{A}^+$, if it's equal to all attributes, then $\bar{A}$ is a key.

In general, how can we find all keys given a set of FDs?

Consider every subset, $\bar{A}$, of attributes

$\bar{A}^+ \rightarrow$all attributes then it's a key. To be more efficient, we can consider these subsets in an increasing order. So an actual algorithm would start with a single attribute.

Example:

Consider the relation R(A, B, C, D, E) and suppose we have the functional dependencies:

AB→C

AE→D

D→B

the key for R is AE

$AB^+ = ABC$

$AC^+ = AC$

$AD^+ = ABCD$

$AE^+ = ABCDE$

3.2.4 Specifying FDs for a relation

- S_1 and S_2 sets of FDs
- S_2 "follows from" S_1if every relation instance satisfying S_1 also satisfies S_2.

Example:

Suppose

$S_2 : \{SSN \rightarrow priority\}$

$S_1 : \{SSN \rightarrow GPA, GPA \rightarrow priority\}$

then S_2 follows from S_1.

How to test?

Does $\bar{A} \rightarrow \bar{B}$ follow from S?

- Compute $\bar{A}^+$ based on functional dependencies that are in S and check if $\bar{B}$ is in the set.
  Example:
  Consider the relation R(A, B, C, D, E) and the set of functional dependencies S_1= {AB→C, AE→D, D→B}
  Which of the following sets S_2 of FDs does NOT follow from S_1?
  S_2={ABC→D, D→B}
  Using the FDs in S1:
  AD^+ = {ABCD}
  AE^+= {ABCDE}
  ABC^+= {ABC}
  D^+= {B}
  ADE^+= {ABCDE}
- Armstrong's Axioms

Want: Minimal set of completely nontrivial FDs such that all FDs that hold on the relation follow from the dependencies in this set.

3.3 Boyce-Codd normal form

3.3.1 Decomposition of a relational schema

Say we have a relation $R(A_1, A_2, \ldots, A_n)$, we can decompose it into $R_1(B_1, \ldots, B_k)$ and $R_2(C_1, \ldots, C_m)$, call those $\bar{A}$, $\bar{B}$ and $\bar{C}$ respectively. Then $\bar{B} \cup \bar{C} = \bar{A}$ and $R_1 \bowtie R_2 = R$. R_1 and R_2 can also be defined as:

$R_1 = \Pi_{\bar{B}}(R)$

$R_2 = \Pi_{\bar{C}}(R)$

Example:

```
Student(SSN, sName, address, HScode, HSname, HScity, GPA, priority)
```

Decomposition 1

```
S1(SSN, sName, address, HScode, GPA, priority)
S2(HScode, HSname, HScity)
```

Is this a correct decomposition in a sense that$\bar{B} \cup \bar{C} = \bar{A}$ and will $S_1 \bowtie S_2 = Student$ (yes)?

Decomposition 2

```
S1(SSN, sName, address, HScode, HSname, HScity)
S2(HScode, GPA, priority)
```

Is this a correct decomposition in a sense that$\bar{B} \cup \bar{C} = \bar{A}$ and will $S_1 \bowtie S_2 = Student$ (no)? We'd be joining on sName and HSname and likely those are not unique, so we could be getting information back that doesn't belong together.

3.3.2 Rational design by decomposition

- "Mega" relations + properties of the data
- System decomposes based on properties
- "Good" decompositions only
  - reassembly produces original
  - lossless join property
- Into "good" relations - BCNF

3.3.3 Boyce-Codd Normal Form

A relation **R** with FDs is in BCNF if:

For each $\bar{A} \rightarrow B$, $\bar{A}$ is a key (or contains a key, like a superset of attributes/key)

Example: Determine if a relation is in BCNF

We need a relation and a set of functional dependencies

```
Student(SSN, sName, address, HScode, HSname, HScity, GPA, priority)
```

$$\begin{aligned} SSN &\rightarrow sName, address, GPA \\ GPA &\rightarrow priority \\ HScode &\rightarrow HSname, HScity \end{aligned}$$

To determine if this R is in BCNF, we need a key or set of keys. Use closure idea - {SSN, HScode} determins all other attributes.

Does every FD have a key on left hand side?

NO

In this case none does.

We would have to use these FDs to brake this R down into one that's a better design.

Example: Determine if a relation is in BCNF

Apply(SSN, cName, state, date, major)

$$SSN, cName, state \rightarrow date, major$$

Note: since we have only this one FD, here we are not assuming cName is a unique college name, but that colleges are identified by cName+state.

The three attributes on LHS form a key, because they determine the rest of the attributes. Furthermore the only (all) FD has a key on LHS, so this relation is already in BCNF and there's no way to decompose it into a better design.

Example:

For the relation

Apply(SSN, cName, state, date, major)

Suppose college names are unique and students may apply to each only once, so we have two FDs:

$cName \rightarrow state$

$SSN, cName \rightarrow date, major$

Is Apply in BCNF?

{SSN, cName} is a key so only **cName** $\rightarrow$ **state** is a BCNF violation. Based on this violation we decompose into

A1(cName, state)
A2(SSN, cName, date, major)

Now both FDs have keys on their left-hand-side so we're done.

3.3.4 BCNF decomposition algorithm

Input: relation **R**+**FD**s for **R**

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

- Compute keys for R - using FDs
- Repeat until all relations are in BCNF:
  - Pick any R' with $\bar{A} \rightarrow \bar{B}$ that violates BCNF
  - Decompose R' into $R_1(A, B)$ and $R_2(A, rest)$
  - Compute FDs for $R_1$and R_2
  - Compute keys for $R_1$and R_2

R'

A	B	rest

↓

R_1

A	B

R_2

A	rest

$R_1 \bowtie R_2 \rightarrow R'$

Example:

Student(SSN, sName, address, HScode, HSname, HScity, GPA, priority)

SSN→sName, address, GPA

GPA→priority

HScode→HSname, HScity

key: {SSN, HScode}

Goal is to brake down the relation until it's in BCNF.

Pick some FD that violates BCNF and use it to start decomposition. All of the FDs in this case violate BCNF, because none have a key on LHS. Pick HScode→HSname, HScity

S1(HScode, HSname, HScity) ✓

S2(SSN, sName, address, HScode, GPA, priority)

For relation S1, the only relational dependency is HScode→HSname, HScity; that tells us that HScode is the key for S1, the only FD for S1 has a key on LHS and so it's in BCNF.

For S2, SSN and HScode are the key and we still have the two FDs that are BCNF violations. Take GPA→priority, now we have:

S1(HScode, HSname, HScity) ✓

~~S2(SSN, sName, address, HScode, GPA, priority)~~

S3(GPA, priority) ✓

S4(SSN, sName, address, HScode, GPA)

S4 still has SSN and HScode as its key, so we decompose further.

S1(HScode, HSname, HScity) ✓

~~S2(SSN, sName, address, HScode, GPA, priority)~~

S3(GPA, priority) ✓

~~S4(SSN, sName, address, HScode, GPA)~~

S5(SSN, sName, address, GPA) ✓

S6(SSN, HScode) ✓

Example:

Consider relation

Apply(SSN, cName, state, date, major)

with FDs

cName→state

SSN, cName→date, major

What schema would be produced by the BCNF decomposition algorithm?

A1(cName, state)
A2(SSN, cName, date, major)

{SSN, cName} is a key so only cName → state is a BCNF violation. Based on this violation we decompose into A1(cName,state), A2(SSN, cName, date, major). Now both FDs have keys on their left-hand-side so we're done.

To compute

- Compute FDs for $R_1$and R_2
- Compute keys for $R_1$and R_2

We use closure to compute **Implied FDs**.

Picking any R' introduces non-determinism to the algorithm. So you can get a different answer depending what you choose, but the schema will be in BCNF.

Some decomposition algos have another step after the above one. We can extend the FD used for the decomposition. If we have $A \to B$, we can also have $A \to BA^+$. We'll have relations that are bigger. In some cases that is good, because you don't need to join them back when doing queries. [shortcomings of BCNF]

3.4 Multivalued dependencies, 4th normal form

Example: College application info.

Apply(SSN, cName, hobby)

FDs? No

Keys? All attributes

BCNF? Yes, no FDs.

Good design? No, if you apply to 5 colleges and have 6 hobbies, we have 30 tuples

3.4.1 Multivalued Dependency

- Based on knowledge of real world
- All instances of relation must adhere to the dependency

For a relation R, $\bar{A} \twoheadrightarrow \bar{B}$

$\twoheadrightarrow$means multi-determines

$$\begin{aligned} \forall_{t,u \in R} : t\left[\bar{A}\right] &= u\left[\bar{A}\right] \text{ then} \\ \exists_{v \in R} : v\left[\bar{A}\right] &= t\left[\bar{A}\right] \text{ and} \\ v\left[\bar{B}\right] &= t\left[\bar{B}\right] \text{ and} \\ v\left[rest\right] &= u\left[rest\right] \end{aligned}$$

For all tuples t, u that are in a relation R, if t with the attributes $\bar{A}$ equals u with the attributes $\bar{A}$ (tuples agree on their $\bar{A}$ values), then there exists a third tuple, v, in R where v has the same values, $\bar{A}$ as t and u, furthermore, it has $\bar{B}$ value from t and rest from u[rest].

	A	B	$rest$
t	$\bar{a}$	b_1	r_1
u	$\bar{a}$	b_2	r_2
v	$\bar{a}$	b_1	r_1
w	$\bar{a}$	b_2	r_2

w comes from swapping the values ot t and u

b and r values are independent

FD are sometimes called tuple-generating dependencies

Example:

Consider a relation R(A, B, C) with multivalued dependecy $\bar{A} \twoheadrightarrow \bar{B}$. Suppose there are at least 3 different values for A, and each value of A is associated with at least 4 different B values and at least 5 different C values.

The minimum number of tuples in R is 60.

Multivalued dependency $\bar{A} \twoheadrightarrow \bar{B}$ says that for each value of A, we must have every combination of B and C values. So for each of the 3 values of A we must have at least 4*5=20 different tuples.

Example:

Apply(SSN, cName, hobby)

$SSN \twoheadrightarrow cName$

(and also $SSN \twoheadrightarrow hobby$)

	SSN	cName	hobby
t	123	Stanford	trumpet
u	123	Berkeley	tennis
v	123	Stanford	tennis
w	123	Berkeley	trumpet

Modified example

Apply(SSN, cName, hobby)

Reveal hobbies to colleges selectively

MVDs? None

Good design? Yes, no multiplicative effect in the relation

Expanded example

Apply(SSN, cName, date, major, hobby)

Reveal hobbies to colleges selectively

Apply once to each college

May apply to multiple majors

SSN, cName→date

SSN, cName, date →major

Example: For the relation Apply(SSN, cName, date, major) with functional dependency SSN, cName→date, what real-world constraint is captured by SSN→cName, date?

A sutdent must apply to the same set of majors at all colleges.

SSN→cName, date says that (cName, date) and major are independent, i.e., all combinations of (cName, date) and major must be present for any college or major a student applies for.

Trivial Multivalued Dependency

$\bar{A} \twoheadrightarrow \bar{B}$ when $\bar{B} \subseteq \bar{A}$ or $\bar{A} \cup \bar{B}$ =all attributes

Nontrivial MVD

otherwise

3.4.2 Rules for Multivalued Dependencies

FD-is-an-MVD rule

$\bar{A} \rightarrow \bar{B}$ then $\bar{A} \twoheadrightarrow \bar{B}$

	A	B	rest
t	$\bar{a}$	$\bar{b}_1$	$\bar{r}_1$
u	$\bar{a}$	$\bar{b}_2$	$\bar{r}_2$
→v	$\bar{a}$	$\bar{b}_1 = \bar{b}_2$	$\bar{r}_2$
	⋮	⋮	⋮

Intersection rule

$\bar{A} \twoheadrightarrow \bar{B}$ and $\bar{A} \twoheadrightarrow \bar{C}$ then $\bar{A} \twoheadrightarrow \bar{B} \cap \bar{C}$

Transitive rule

$\bar{A} \twoheadrightarrow \bar{B}$ and $\bar{A} \twoheadrightarrow \bar{C}$ then $\bar{A} \twoheadrightarrow \bar{B} - \bar{C}$

Every rule for MVD is a rule for FD, not necessarily the other way around, example is splitting rule.

3.4.3 Fourth Normal Form

Relation **R** with MVDs is in 4NF if for each non-trivial $A \twoheadrightarrow B$, **A is a key**.

	$\bar{A}$	$\bar{B}$	rest
t	$\bar{a}$	$\bar{b}_1$	$\bar{r}_1$
u	$\bar{a}$	$\bar{b}_2$	$\bar{r}_2$
	$\vdots$	$\vdots$	$\vdots$

3.4.4 4NF Decomposition Algorithm

Input: relation R + FDs for R + MVDs for R

Output: decomposition of R into 4NF relations with "lossless join"

- Compute keys for R
- Repeat until all relations are in 4NF:
  - Pick any R' with nontrivial A↠B that violates 4NF
  - Decompose R' into $R_1(A, B)$ and $R_2(A, rest)$
  - Compute FDs and MVDs for R_1 and R_2
  - Compute keys for $R_1$and R_2

Example 1: 4NF Decomposition

Apply(SSN, cName, hobby)

SSN↠cName, no keys

This is a violating MVD, so decompose into

A_1(SSN, cName)

A_2(SSN, hobby)

no FDs and no MVDs, so done.

Example 2: 4NF Decomposition

Apply(SSN, cName, date, major, hobby)

SSN, cName→date

SSN, cName, date↠major

no keys

A_1(SSN, cName, date, major) - contains all attributes of our MVD

A_2(SSN, cName, date, hobby) - contains all the remaining attributes along with the left hand side of our MVD. Now take a look at the decomposed relations and see what we have in terms of FDs and MVDs for them. We have no more MVDs but FD applies to both of the decomposed relations and we still don't have a key on LHS, so we need to decompose further based on the first FD. Take A_1 and turn it into:

~~(SSN, cName, date, major)~~

A_2(SSN, cName, date, hobby)

A_3(SSN, cName, date)

A_4(SSN, cName, major)

We have a similar problem with A_2, so decompose it:

~~(SSN, cName, date, major)~~

~~(SSN, cName, date, hobby)~~

A_3(SSN, cName, date)

A_4(SSN, cName, major)

we'll discover that A_3 is the same relation in the decomposition of A_2 as we got with A_1, so we only add one relation.

A_5(SSN, cName, hobby)

Now the only MVDs or FDs we have left do have a key on the LHS. This is in 4NF.

Example:

Consider relation StudentInfo(sID, name, dorm, major) with functional dependency sID→name and MVD sID↠dorm. What schema would be produced by the 4NF decomposition algorithm?

There is no key for the relation. Decomposing on the violating FD separates (sID, name) from (sID, dorm, major). Decomposing on the violating MVD separates (sID, dorm) from (sID, major). Now there are no violating FDs or nontrivial MVDs.

S_1(sID, name)

S_2(sID, dorm)

S_3(sID, major)

3.4.5 Summary

Functional dependencies & BCNF

If we have R(A, B, C), a FD $A \to B$ tells us that when we have the same A values, we have the same B values and BCNF tells us to factor those attributes into their own relation so we don't repeat that relationship over and over

MVDs & 4NF

Say we have a relation R(A, B, C, D) and if we have the MVD $\bar{A} \twoheadrightarrow B$, what that tells us is that we have every combination of for a given A of B values and CD values and if we have that multiplicative effect, we take the A and B attributes and we put them in a separate relation so that we can separate those facts from the independent fact of A and its CD values.

4NF is a stronger form than BCNF.

3.5 Shortcomings of BCNF/4NF

3rd normal form may be a better design if we need to join needlessly.

After decomposition, there is no guarantee dependencies can be checked on decomposed relations, need joins.

Denormalized relation - when query needs to reassemble relation, it may be preferable to use this, not in NF.

Too decomposed relations

Shortcomings:

- Dependency enforcement
- Query workload
- Over-decomposition

4 UML:

4.1 Unified Data Modeling

4.1.1 Higher level Database Design Models

- Entity-Relationship Model (E/R)
- Unified Modeling Language (UML)
  - data modeling subset

4.1.2 UML Data Modeling: 5 concepts

1. Classes
2. Associations
3. Association Classes
4. Subclasses
5. Composition & Aggregation

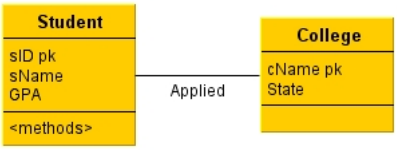
4.1.3 Classes

Name, attributes, methods

For data modeling: add "pk", drop methods

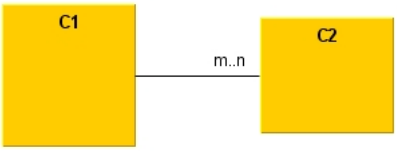
4.1.4 Associations

Capture relationships between objects of two classes.



Multiplicity of Associations

Each object of class C_1 is related to at least m and at most n objects of class C_2.



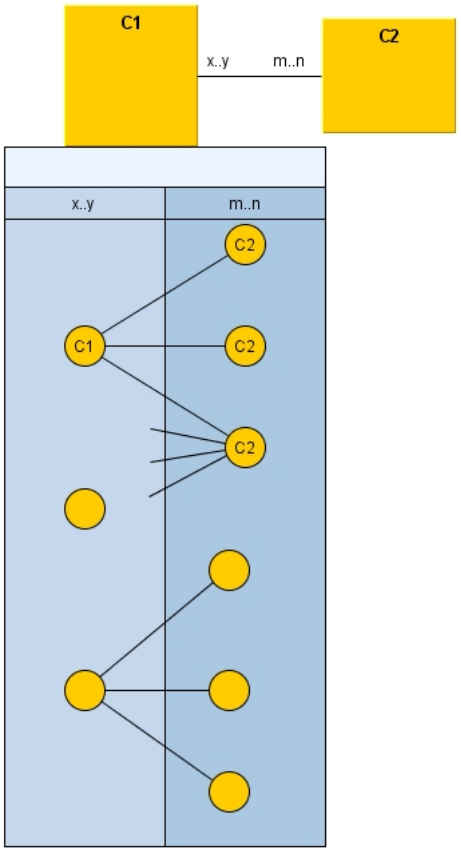
every object in C_1 will be related to between m and n objects of C_2.

$m..*$ ←any number

$0..n$

0..*

1..1 ←default



1..1 → 1

0..* → *

Example:

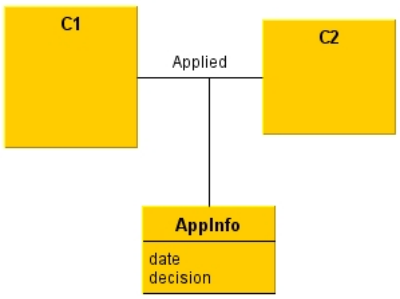
Students must apply somewhere and may not apply to more than 5 colleges. No college takes more than 20000 applications.

4.1.5 Types of Relationships

- One-to-One
  - 0..1 0..1
- Many-to-One
  - * 0..1
- Many-to-Many
  - * *
- Complete
  - default
  - 1..1 1..1
  - 1..* 1..*

4.1.6 Association Classes

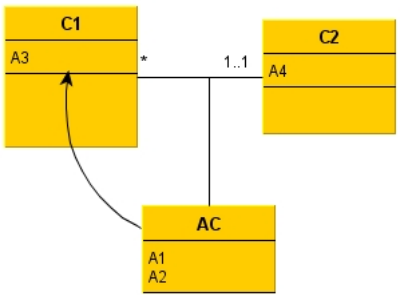
Relationships between objects of two classes, with attributes on relationships.



4.1.7 Eliminating Association Classes

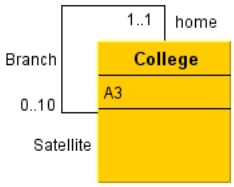
Unnecessary if 0..1 or 1..1 multiplicity

Suppose that multiplicity on left is * and right 1..1, so each object of C1 is related to exactly one object of C2, so we know there's gonna be just one association for each object of C1. We can take attributes from the association and put them in C1.

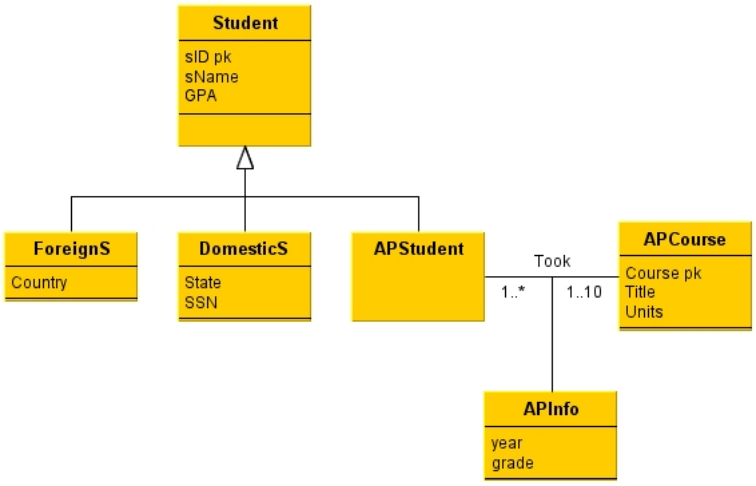


4.1.8 Self-Association

Associations between a class and itself



4.1.9 Subclasses



Student is a superclass

Rest are subclasses

Superclass = Generalization

Subclass = Specialization

Incpmplete (partial) vs Complete (every obj in at least one subclass)

Disjoint (exclusive, every obj in at most one subclass) vs. Overlapping

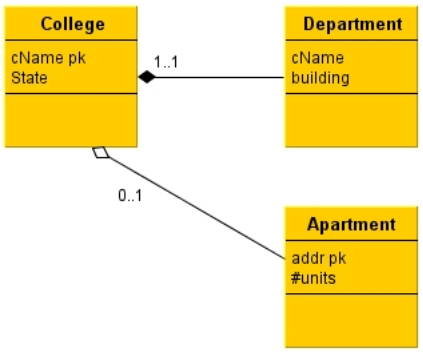
We can have any combination of the above

For above example - {complete, overlapping}, every student in at least one subclass, some students are both ap and domestic student.

If we didn't have the ap section, it would be disjoint.

4.1.10 Composition & Aggregationp

Objects of one class belong to objects of another class.



Composition - Each department belongs to one college. Aggregation - Some apartment buildings are owned by the college but not all of them.

4.2 UML to relations

4.2.1 Classes

Every class becomes a relation; pk→primary key

Student(sID, sName, GPA)

College(cName, state, enrollment)

4.2.2 Associations

Relation with key from each side

Student(sID, sName, GPA)

College(cName, state, enrollment)

Applied(sID, cName)

4.2.3 Keys for Association Relations

Depends on multiplicity

Suppose:

C1(k1, O1)

C2(k2, O2)

A(k1, k2)

Suppose on left it's 0..1 and * on right.

C1 can be related to many objects of C2 but each object of C2 can be related to at most 1 object of C1. From this, we conclude that k2 is the key for A. So when we have 0..1 on the left side, then the key attribute from the other side is a key for the association.

For the student example

So according to the above, sID would be the key for applied.

4.2.4 Association Relation Always Needed?

Depends on multiplicity

C1(k1, O1)

C2(k2, O2)

A(k1, k2)

C1 1..1 —— * C2

We can take the related element from the left and add it to right.

C1(k1, O1)

C2(k2, O2, k1)

If left side was 0..1, this would still be ok as long as Nulls are allowed for k1.

Student example:

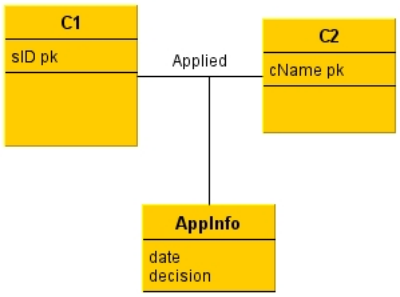
Student(sID, sName, GPA, cName)

College(cName, state, enrollment)

This works for 0..n or m..n multiplicity, although the bigger n is, the less sense this translation makes.

4.2.5 Association Classes

Add attributes to relation for association



Student(sID, ...)

College(cName, ...)

Applied(sID, cName, date, decision)

4.2.6 Self-Associations

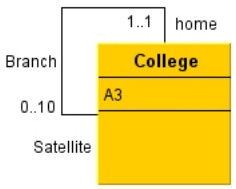
Student(sID pk, sName, GPA)

Student * –(sibling)– * Student

Student(sID, sName, GPA)

Sibling(sID1, sID2)

Example:



College(cName, state, enrollment)

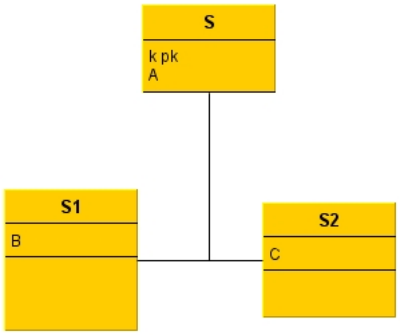
Branch(home, satellite)

Key for branch is satellite - if we have 1..1 on one side then the other side is a key in the association relation.

4.2.7 Subclasses

1. Subclass relation contan superclass key + specialized attributes
2. Subclass relations contain all attributes

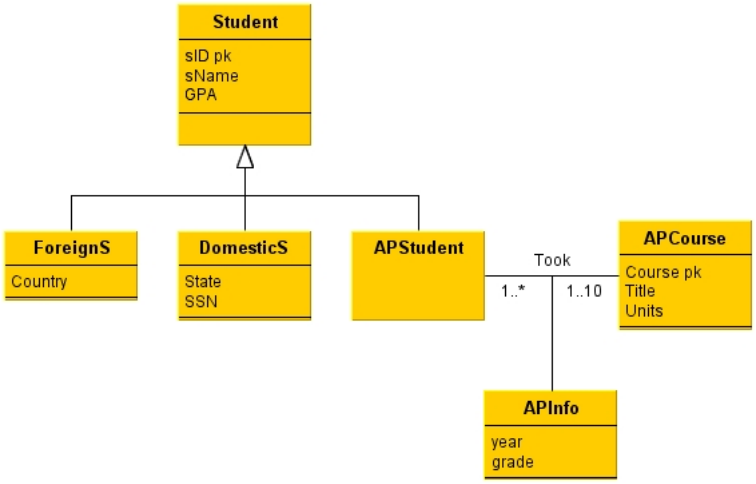
3. One relation containing all superclass + subclass attributes.



1. S(K, A) S1(K, B) S2(K, C)
2. S(K, A) S1(K, A, B) S2(K, A, C)
3. S((K, A, B, C)

Heavily overlapping⇒design 3

Disjoint/complete⇒design 2, we could get rid of S



Student(sID, sName)

ForeignS(sID, country)

DomesticS(sID, state, SSN)

APStudent(sID)

APCourse(Course#, title)

Took(sID, course#, year, grade)

Comment: if every student takes at least one course, then we can eliminate this relation, because ever sID in APStudent will also appear in the Took relation, so it's redundant.

Keys for "regular" classes - subclasses don't need to have keys and we can still have automatic translation.

4.2.8 Composition & Aggregation

College(cName pk, state) ◆——-Department(dName, building)

College(cName, state)

Department(dName, building, cName)

With aggregation, empty diamond, we need for cName to have the ability to be Null.

5 Indexes

- Primary mechanism to get improved performance on database
- Persistent data structure, stored in database
- Many interesting implementation issues

T	A	B	C
1	cat	2	...
2	dog	5	...
3	cow	1	...
4	dog	9	...
5	cat	2	...
6	cat	8	...
7	cow	6	...
	...	...	...

Users don't access indexes; they're used underneath by the query execution engine.

5.1 Utility

- Index = difference between full table scans and immediate location of tuples
  - Order of magnitude performance difference
- Underlying data structures
  - Balanced trees (B trees, B+ trees)
    - for $=, <, \leq, \geq$
    - $\log(n)$
  - Hash tables
    - only for=
    - constant running time

5.2 Examples

If Index on sID

```
Select sName
From Student
Where sID=18942
```

Many DBMS's build indexes automatically on PRIMARY KEY (and sometimes UNIQUE) attributes.

```
Select sID
From Student
Where sName='Mary' and GPA>3.9
```

Index on sName - find Mary and then check each of the found rows to find if any has GPA>3.9 - tree or hash based index

Index on GPA - find students with GPA>3.9 and then check if their name is Mary - tree based index

Index on (sName, GPA) - find simultaneously for both conditions

```
Select sName, cName
From Student, Apply
Where Student.sID = Apply.sID
```

Let's say we have an index on Apply.sID. In this case the query engine can scan Student relation and for each Student.sID quickly match the sID in Apply relation.

In some cases it's possible to use indexes on both indexed columns. Indexes often allow relations to be accessed in sorted order of the index attributes. There may be other conditions that allow more choices - Query planning and optimization.

5.3 Downsides of Indexes

1. Extra space - persistent data structure in db, marginal downside
2. Index creation overhead - medium
3. Index maintenance - when db values change, indexes have to be updated, can offset benefits if db not queried as often as updated

5.4 Picking which indexes to create

Benefit of an idex depends on:

- Size of table (and possibly layout)
- Data distributions
- Query vs. update load

"Physical design advisor" - software to determine what to index

Input: database (statistics) and worklad

Output: recommended indexes

Relies on Query Optimizer, which takes a query and figures out how to execute it.

It will take statistics on the database, query to be executed and the set of indexes that exist and explore ways of executing the query, estimates the cost of each one and will output the best execution plan with estimated cost.

5.5 SQL Syntax

```
Create Index IndexName on T(A)
Create Index IndexName on T(A1, A2, ..., An)
Create Unique Index IndexName on T(A)
Drop Index IndexName
```

Unique will check if all values of **A** are unique otherwise will throw an error.

6 Constraints and Triggers

6.1 Motivation and Overview

- For relational databases
- SQL standard; systems vary considerably in support

(Integrity) Constraints: constrain allowable database states. static concept

Triggers: monitor database changes, check conditions and initiate actions. dynamic concept

6.1.1 Integrity Constraints

Impose restrictions on allowable data, beyond those imposed by structure and types

Example:

$0.0 < GPA \leq 4.0$ GPA must be between 0 and 4

decision: 'y', 'n', null

major='cs'⇒decision = null

6.1.2 Why use them?

- Data-entry errors (inserts)
- Correctness criteria (updates)
- Enforce consistency
- Tell system about the data (key constraints, unique...) - store, query processing

6.1.3 Classification

- Non-null
- Key
- Referential integrity (foreign key)
- Attribute-based
- Tuple-based
- General assertion

6.1.4 Declaring and enforcing constraings

- Declaration
  - With original schema - checked after bulk loading, error raised if bad
  - Or later - checked on current DB
- Enforcement
  - Check after every "dangerous" modification
  - Deferred constraing checking - sometimes we want to do a whole bunch of operations that violate constraings, but after we're done, they won't be violated - **transaction**.

6.1.5 Triggers

"Event-Condition-Action Rules": When even occurs check condition; if true, do action.

Examples:

If enrollment $> 35000 \Rightarrow$reject all applicants

Insert app with GPA $> 3.95 \Rightarrow$accept automatically.

Update sizeHS to be $> 7000 \Rightarrow$change to "wrong" or raise an error

Why use them?

Original motivation was to move logic appearing in application to the DBMS.

To enforce constraints - because most constraint features across databases are limited.

- expressiveness
- constraint "repair" logic

6.1.6 Triggers in SQL

```
Create Trigger name
Before|After|Instead Of events
[referencing-variables]
[For Each Row]
When (condition)
action
```

6.2 Constraints of several types

Constraints Demo

```
College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)
```

```
create table Student(
        sID int,
        sName text,
        GPA real not null,
        sizeHS int);
```

only GPA is allowed to be null

```
update Student set GPA = null where sID=123
```

will be updated if there is a student with sID = 123

Defered constraint checking - check if there are constraint violations after all changes are made

Immediate constraint checking - check constraint violations after each change

Only **one primary** key is **allowed** per table, but more keys allowed with **unique**.

```
create table Student(
        sID int primary key,
        sName text unique,
        GPA real,
        sizeHS int);
```

We can have **keys that span several attributes** - combination of attributes must be unique for each tuple.

```
create table College(
        cName text,
        state text,
        enrollment int,
        primary key(cName, state));

insert into College values('Mason', 'CA', 123454)
insert into College values('Mason', 'NY', 123454)
insert into College values('Mason', 'CA', 123454)
```

Last one will generate an error because there can only be one 'Mason', 'CA' pair.

Two key constraints

Each student can apply to each college once and each major once.

```
create table Apply(
        sID int,
        cName text,
        major text,
        decision text,
        unique(sID, cName),
        unique(sID, major)
);
```

Changing values may violate constraints as well.

Null values with keys

Null values are allowed, but not repeated for the same key.

6.2.1 Attribute based check constraints

```
create table Student(
        sID int,
        sName text,
        GPA real check(GPA <= 4.0 and GPA > 0.0),
        sizeHS int check(sizeHS < 5000));
```

6.2.2 Tuple based check constraints

Have boolean expression in check condition.

The above attribute and tuple based constraints IN MySQL ARE ACCEPTED BUT **NOT ENFORCED**, but are enforced in sqlite and postgres.

No subqueries or aggregation in check constraints (in SQL standard but not supported)

Make sure that sID in the Apply table is in the Student table. This will not work because of the subquery - but demonstrates referential integrity.

```
create table Student(
        sID int,
        sName text,
        GPA real
        sizeHS int
);

create table Apply(
        sid int,
        cName text,
        major text,
        decision text,
        check(
                sid in(
                        select sID from Student
                )
        )
);
```

6.2.3 General Assertions

Not supported by any database.

6.3 Referential integrity

Impose restrictions on allowable data, beyond those imposed by structure and types.

Referential integrity = integrity of references = no "dangling pointers"

Student

sID	sName	GPA	HS
123	Mary	-	-

Apply

sID	sName	major	dec
123	Stanford	CS	Y

College

cName	state	enr
Stanford		

Referential integrity from R.A to S.B

Each value in column A of tabe R must appear in column B of table S

- A is called the "foreign key"
- B is usually required to be the primary key for the table S or at least unique
- Multi-attribute foreign keys are allowed

Referential Integrity Enforcement (R.A to S.B)

Potential violating modifications:

- Insert into R - if violation→ *error*
- Delete from S - possible to modify the referencing table so there's no violation
- Update R.A - if violation→ *error*
- Update S.B - possible to modify the referencing table so there's no violation

Special actions:

- Delete from S (we do a deletion from a referenced table)
  - Restrict(default): generates error if constraint violated
  - Set Null - if we delete a tuple in the referenced table, we take the referencing tuples and replace them with null
  - Cascade - delete tuple that has a referencing value, can setup a chain of referential integrity constraints
- Update S.B
  - Restrict(default)
  - Set Null
  - Cascade: if we're updating a referenced value, we'll update the referencing value too

```
create table College(
        cName text primary key,
        state text,
        enrollment int
);

create table Student(
        sID int primary key,
        sName text,
        GPA real,
        sizeHS int
);

create table Apply(
        sID int references Student(sID),
        cName text references College(cName),
        major text,
        decision text
);
```

Need to first insert tuples for Student and College, Apply last.

Automatic referential integrity management

```
create table Apply(
        sID int references Student(sID) on delete set null,
        cName text references College(cName) on update cascade,
        major text,
        decision text
);
```

Example

```
create table T(
        A int,
        B int,
        C int,
        primary key (A, B),
        foreign key (B, C) references T(A, B) on delete cascade);

insert into T values (1, 1, 1);
insert into T values (2, 1, 1)
insert into T values (3, 2, 1)
...
```

Demonstrates

- Referential integrity within a single table
- Referential integrity involving multiple attributes, foreign keys and primary keys
- Real cascading

6.4 Triggers introduction

"Even-Condition-Action Rules"

When event occurs, check condition; if true, do action

1. Move monitoring logic from apps into DBMS
2. Enforce constraints
   (a) beyond what constraint system supports
   (b) automatic constraint "repair"

```
Create Trigger name
Before|After|Instead Of events
[referencing-variables]
[For Each Row]
When (condition)
action
```

events:

- insert on T (referencing-variable: new)
- delete on T (referencing-variable: old)
- update of [C1, ..., Cn] on T (referencing-variable: old, new)

[For Each Row]: Run the trigger per each modified tuple, if missing, it will execute trigger once

[referencing-variables]: give us a way to reference the data that was modified that caused the trigger to be activated. Once declared, these variables can be referenced in the trigger condition and action. keywords:

- old row as var
- new row as var
- old table as var
- new table as var

Row level trigger - old row will refer to the specific tuple that the trigger is activated for, while old table will refere to all of the (deleted) tuples. It is refering not to the old state of db but the specific tuples that were, in this case, deleted.

If the trigger is statement level, then we can not refer to row variables, just table.

The last (condition) is like SQL where →general assertion

action - SQL statement

Example: Referential integrity

R.A references S.B, implement cascaded delete

```
Create Trigger Cascade
After Delete on S
Referencing Old Row as O
For Each Row
Delete From R Where A=O.B
```

After we delete on S, activate trigger for each deleted row, call row o, delete from R all values where A value = O.B of the deleted tuple from S.

Example: Same as a statement level

```
Create Trigger Cascade
After Delete on S
Referencing Old Table as OT

Delete From R Where A in (select B from OT)
```

May be more efficient, also **not all systems support both.**

6.4.1 Tricky Issues

- Row-level vs. Statement-level
  - New/Old Row and New/Old Table
  - Before, Instead Of
- Multiple triggers activated at same time
- Trigger actions activating other triggers (chaining)
  - also self-triggering, cycles, nested invocations
- Conditions in when vs. as part of action
- Can affect performance

6.5 Triggers demo I

- Postgres
  - Expressiveness/behavior = full standard row-level + statement-level, old/new row & table
  - Cumbersome & awkward syntax
- SQLite
  - Row-level only, immediate activation⇒no old/new table
- MySQL
  - Row-level only, immediate activation⇒no old/new table
  - Only one trigger per event type
  - Limited trigger chaining

6.6 Triggers demo II