

UML Data Modelling

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Agenda

Introduction to Database Design Composition & Aggregation

Classes

Constraints

Associations

Derived Elements

Association Classes

Generalizations

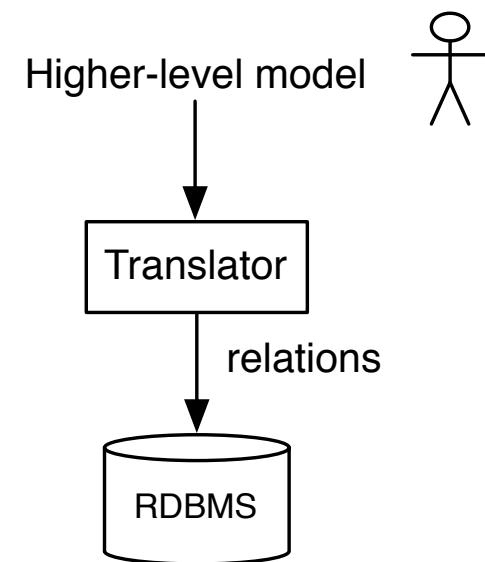
Data modelling

How to represent data for application

Database design model

Not implemented by system

Translated into model of DBMS



Higher-Level Database Design Models

Entity-Relationship Model (E/R)

Unified Modeling Language (UML)

Data modeling subset

Both are graphical

Both can be translated to relations automatically

Or semi-automatically

Key concepts

Classes

Constraints

Associations

Derived Elements

Association Classes

Generalizations

Composition & Aggregation

Classes

Descriptor of a set of objects that share the same properties (semantics, attributes, and relationships)

Concrete things

person, book, car, ...

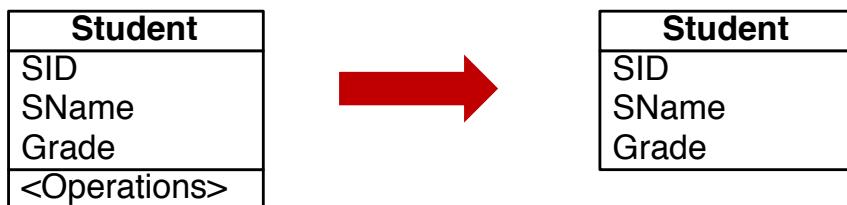
Conceptual things

class, course, profession, ...

Classes

Characterized by name, attributes and operations

For data modeling: drop operations



The class name is usually written in the singular, with the first letter in uppercase

Attributes

Attributes are defined in terms of class, while values of the attributes are defined at the instance level

A student has the attributes: identifier, name and admission grade

John is a student with the id *123*, name *John Smith* and an admission grade of *180*

A class should not have two attributes with the same name

Attributes can be associated with types

Not predefined in UML

Use the ones of the DBMS

Student
SID: integer
SName: string
Grade: integer

In data modelling, we can also specify a primary key

Student
SID <u>pk</u>
SName
Grade

Exercise

Imagine the SIGARRA database.

List 3 classes with attributes that might be in this database.

Key concepts

Classes

Constraints

Associations

Derived Elements

Association Classes

Generalizations

Composition & Aggregation

Associations

Relationships between objects of two classes



As an object is an instance of a class, a **link** is an instance of an association

The name is optional

There may be more than one association between the same pair of classes

Having different names

Associations

Imagine a scenario where students apply to colleges.

What classes would we create? And associations?



Multiplicity of Associations



Each object of Class 1 is related to at least m and at most n objects of Class 2

A * in place of n stands for *no upper limit*

Abbreviations

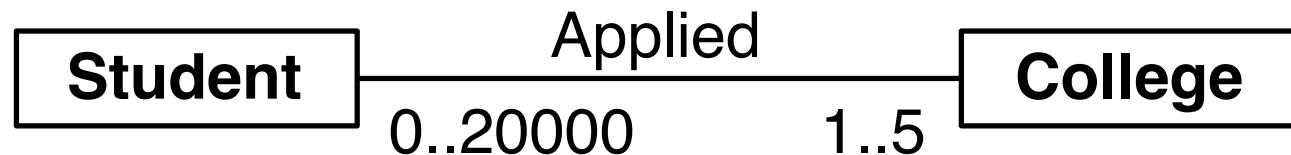
* stands for 0..*, that is, no restrictions

1 stands for 1..1

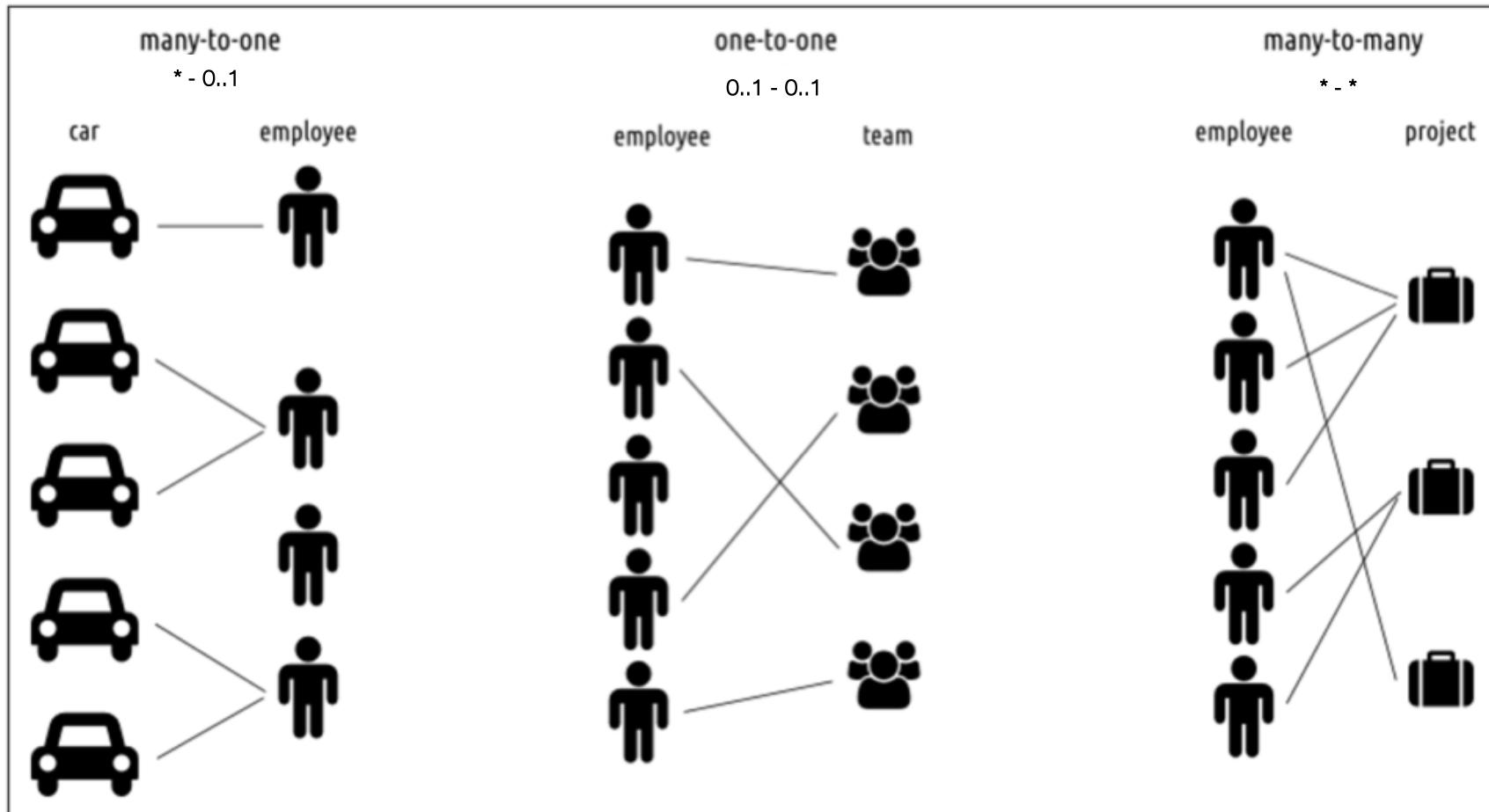
Default: 1..1

Multiplicity of Associations

Students must apply somewhere and may not apply to more than 5 colleges. No college takes more than 20,000 applications.



Multiplicity of Associations



Credits: André Restivo

Complete Associations

Every object must participate in the association

Complete many-to-one

$1..* - 1..1$

Complete one-to-one

$1..1 - 1..1$

Default association

Complete many-to-many

$1..* - 1..*$

Kahoot time

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in
Database Systems 3rd Edition

Section 4.7 - Unified Modeling Language

UML Data Modelling

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Key concepts

Classes

Constraints

Associations

Derived Elements

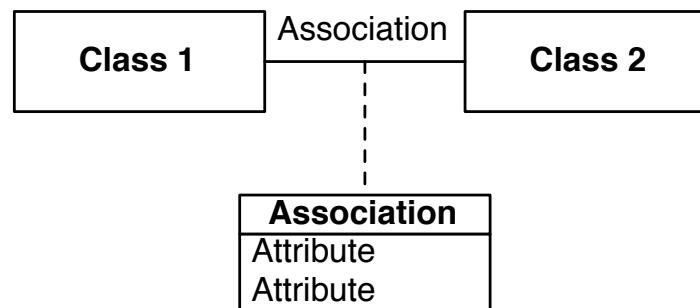
Association Classes

Generalizations

Composition & Aggregation

Association Classes

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

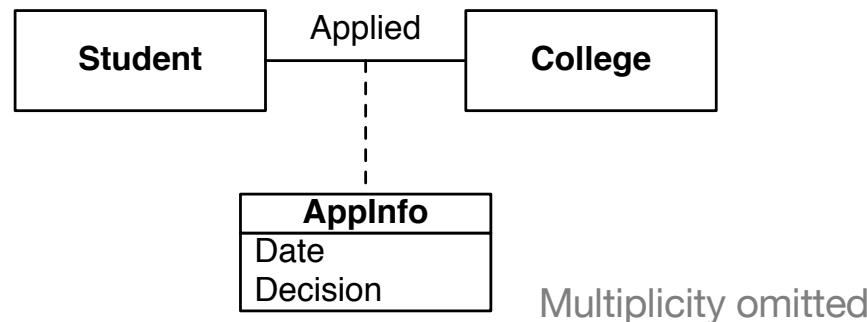


The name may be placed in the association, in the class or in both

It only captures **one** relationship between the two specific objects across the two classes

Association Classes

Suppose that, in the previous example, we also want to have the date the students applied to college and the decision.

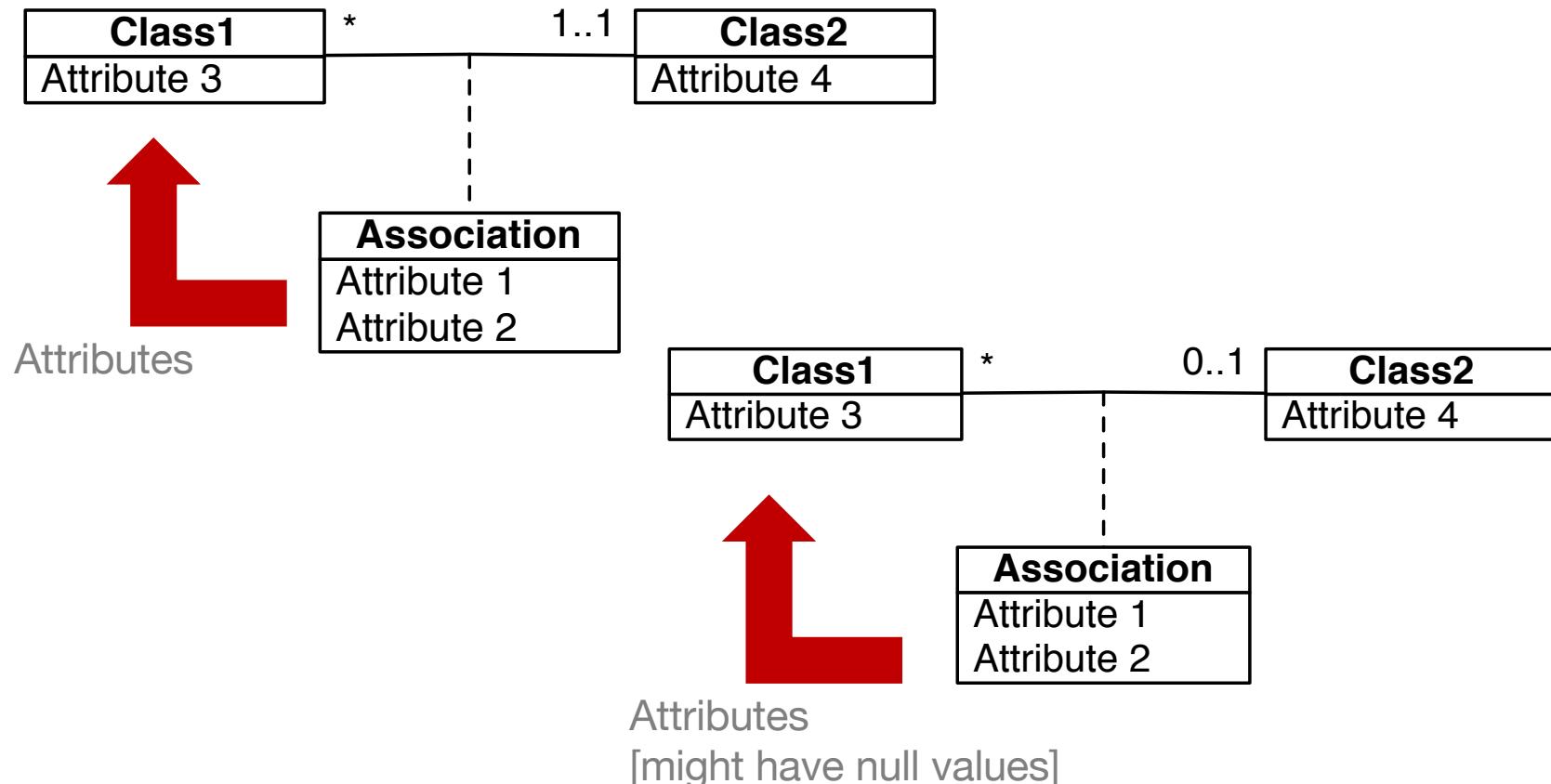


Doesn't allow students to apply multiple times to the same college.

How could we model this situation?

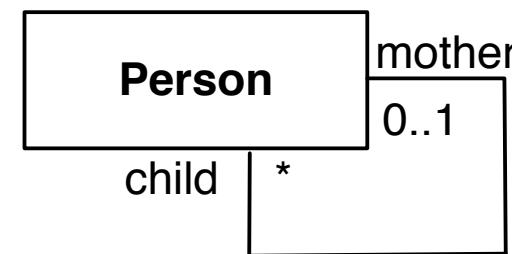
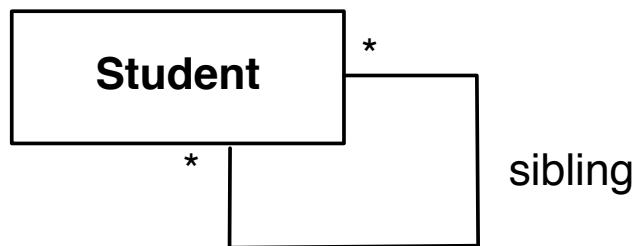
Eliminating Association Classes

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

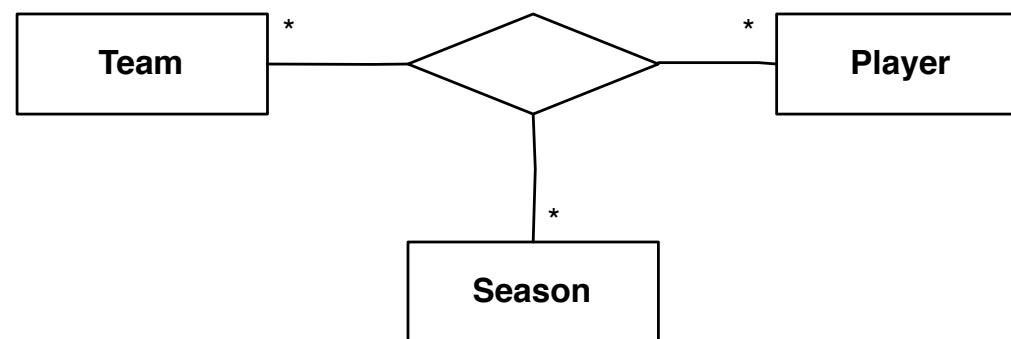
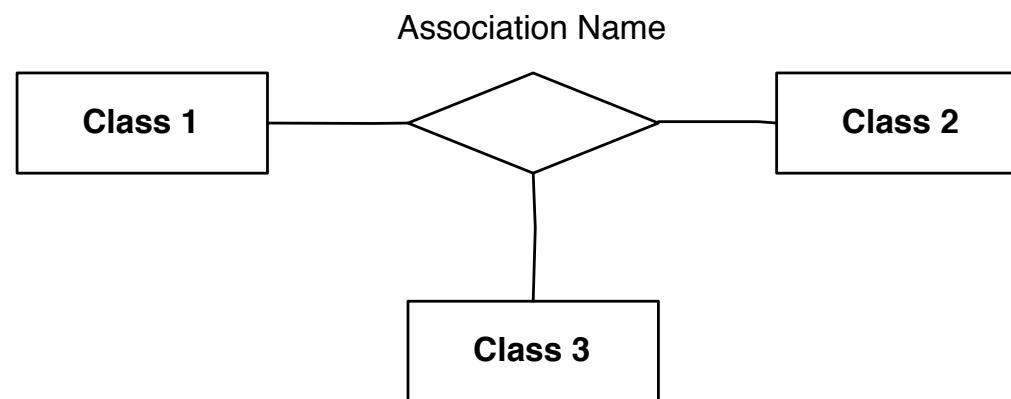


Self-Associations

Associations between a class and itself

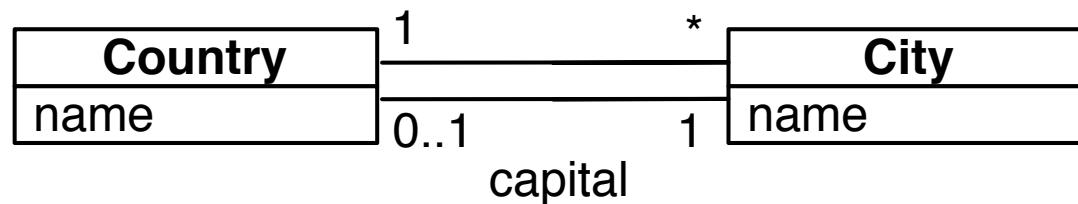
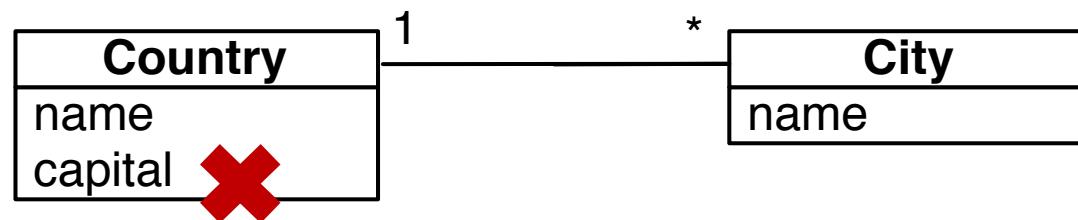


Associations n-arys

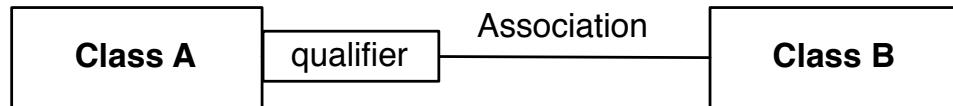


Association versus attribute

An attribute should never be a reference to a class



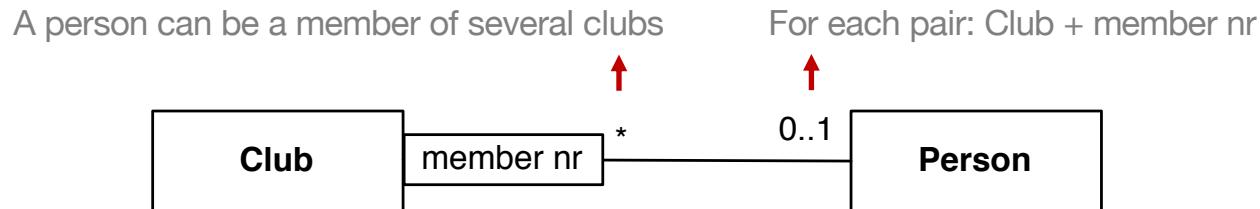
Qualified associations



Qualifier

One or more attributes of an association used to navigate from A to B

“Access key” to B from an object of A



Exercise

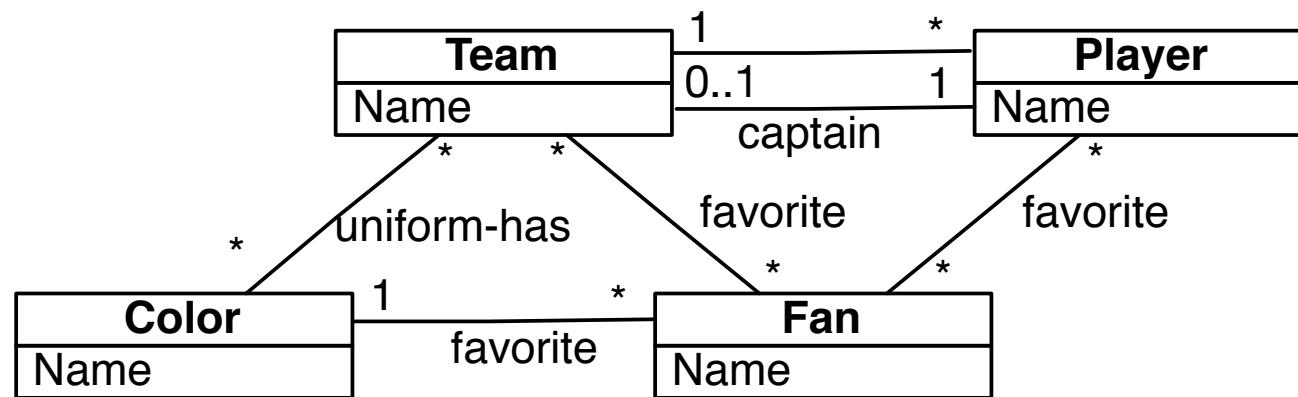
Draw a UML diagram for a database recording information about teams, players, and their fans, including:

For each team, its name, its players, its team captain (one of its players), and the colors of its uniform

For each player, his/her name

For each fan, his/her name, favorite teams, favorite players, and favorite color.

Exercise



Key concepts

Classes

Constraints

Associations

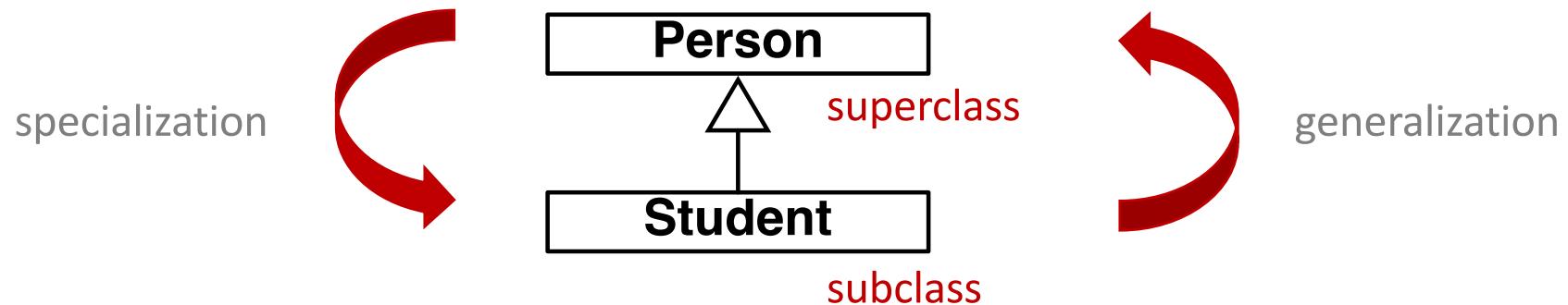
Derived Elements

Association Classes

Generalizations

Composition & Aggregation

Generalizations

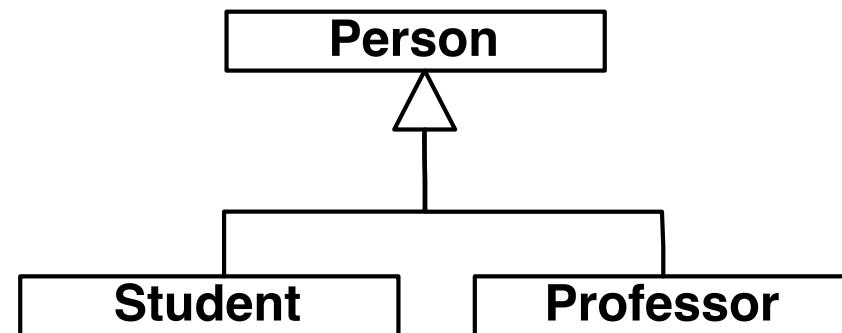
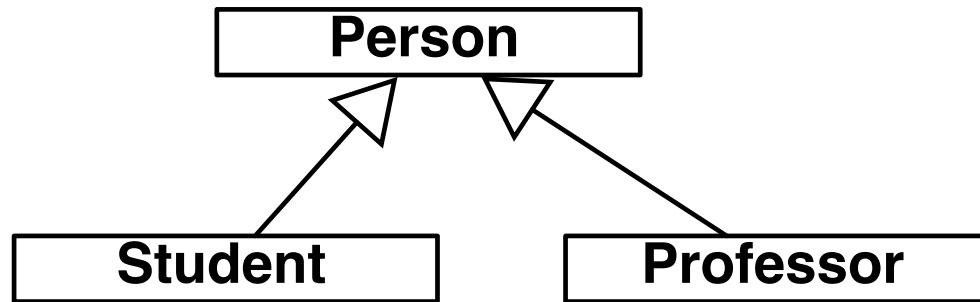


“is a” semantic relationship

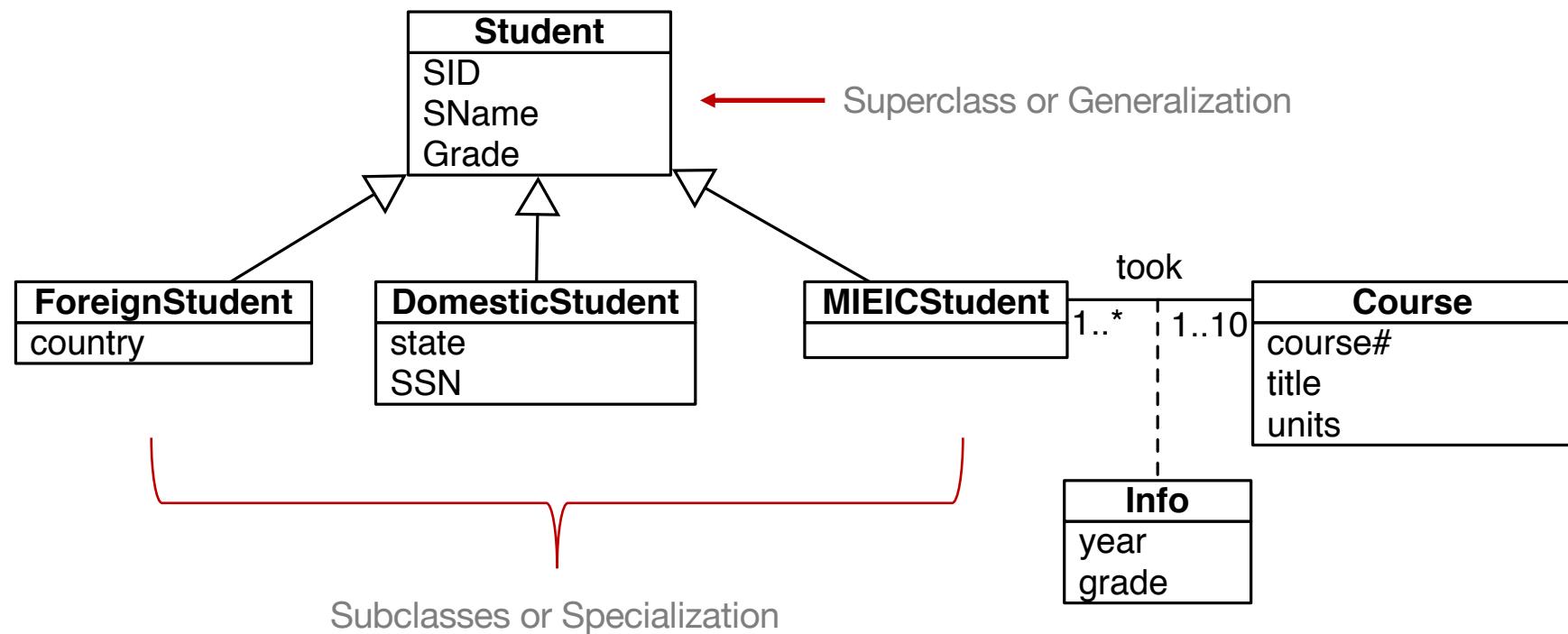
A student “is a” person

The subclass inherits the properties (attributes and relationships) of the superclass and may add other

Generalizations – Alternative Notations



Generalizations Example



Generalizations Properties

Complete

If every object in the super class is in at least one subclass

Incomplete or partial

If it's not complete

We can have any combination of the first two with the second two.

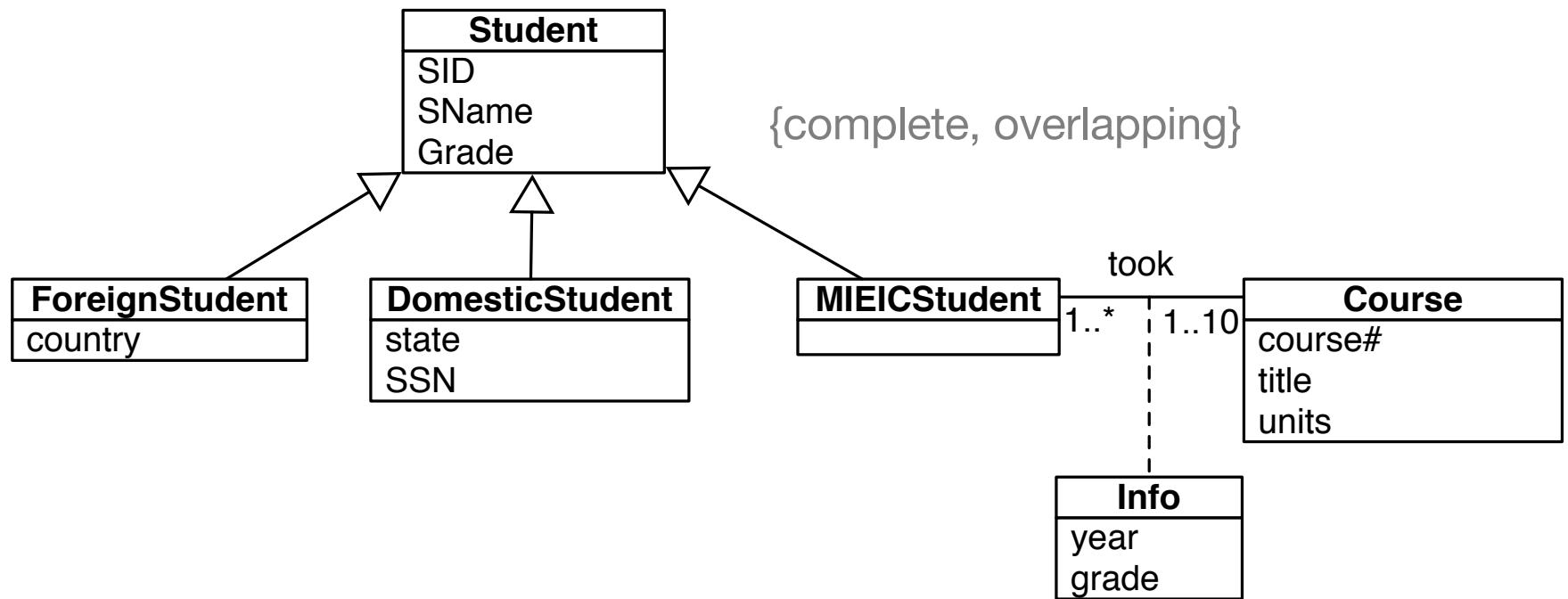
Disjoint or Exclusive

If every object is on at most one subclass

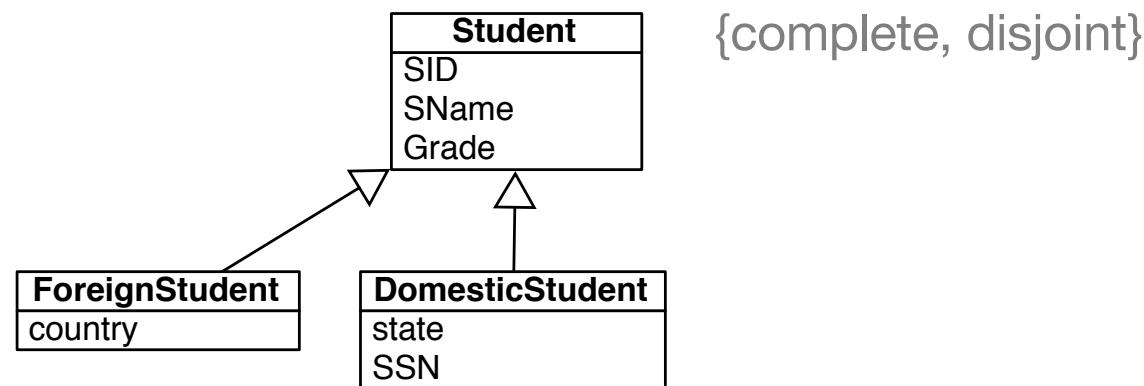
Overlapping

If it's not disjoint

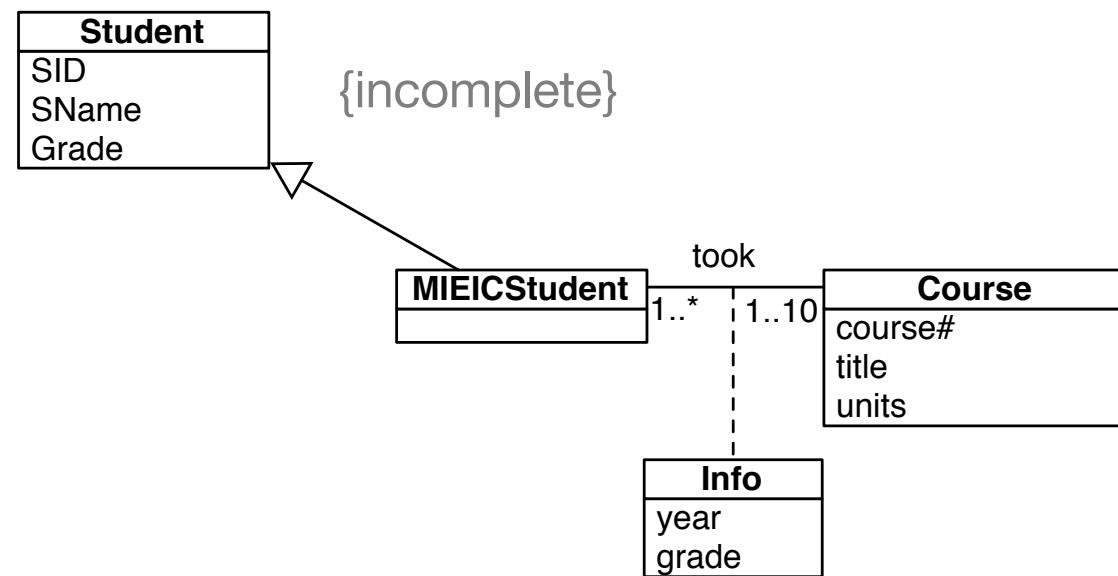
Generalizations Properties



Generalizations Properties



Generalizations Properties



Key concepts

Classes

Constraints

Associations

Derived Elements

Association Classes

Generalizations

Composition & Aggregation

Aggregation

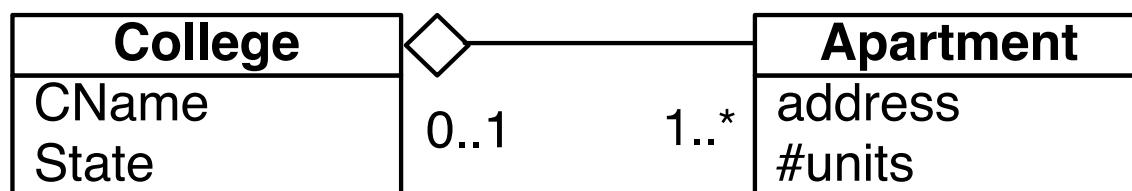
Special type of association

Models a “whole/part” relationship

Represents a “has-a” relationship

Does not link the lifetimes of the whole and its parts

0..1 is implicit



Aggregation

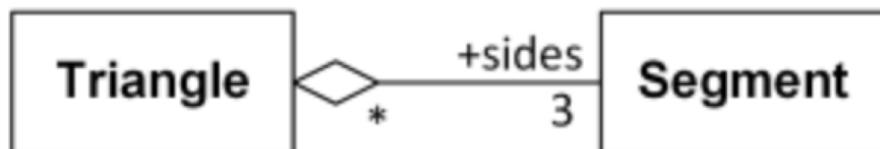
Binary association

Asymmetric

Only one end of association can be an aggregation

Shared part can be included in several composites

If some or all of the composites are deleted, shared part may still exist



Triangle has 'sides' collection of three line Segments.

Each line Segment could be part of none, one, or several triangles.

Composition

Strongest form of aggregation

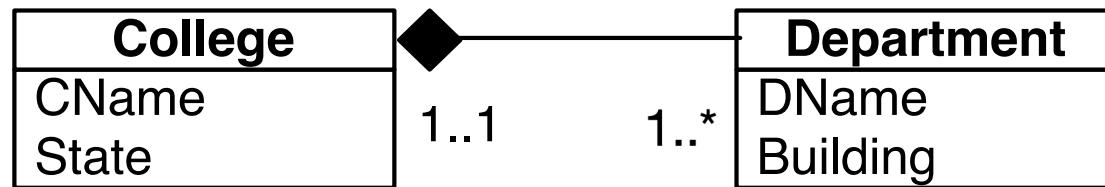
Strong ownership and coincident lifetime as part of the whole

An object may be a part of only one composite at a time

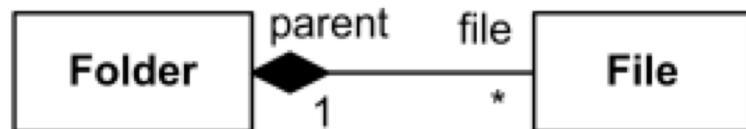
The whole is responsible for managing the creation and destruction of its parts

1..1 is implicit

Composition



College has 1 or more Departments and each Department belongs to exactly one College.
If College is closed, so are all of its Departments.



Folder can contain many files, while each File has exactly one Folder parent.
If Folder is deleted, all contained Files are deleted as well.

Key concepts

Classes

Constraints

Associations

Derived Elements

Association Classes

Generalizations

Composition & Aggregation

Constraints

Specifies a condition that has to be present in the system

It is indicated by

an expression or text between brackets

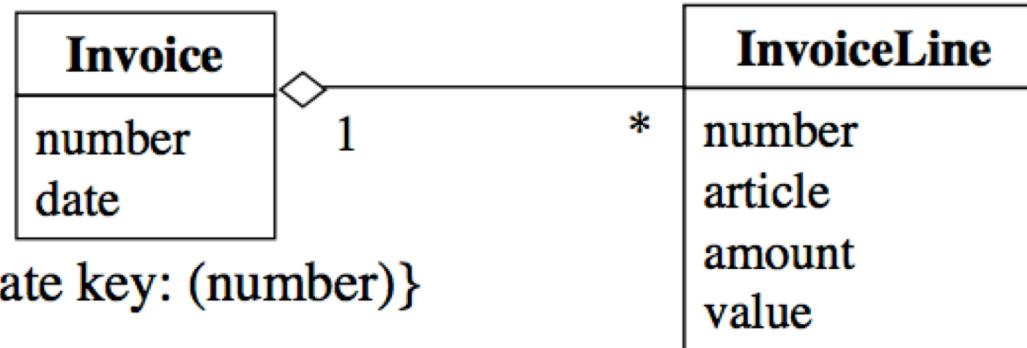
note placed near (or connected by dotted lines to) the elements to which it relates

Constraints in classes

Person
name
birthDate
birthPlace
deathDate

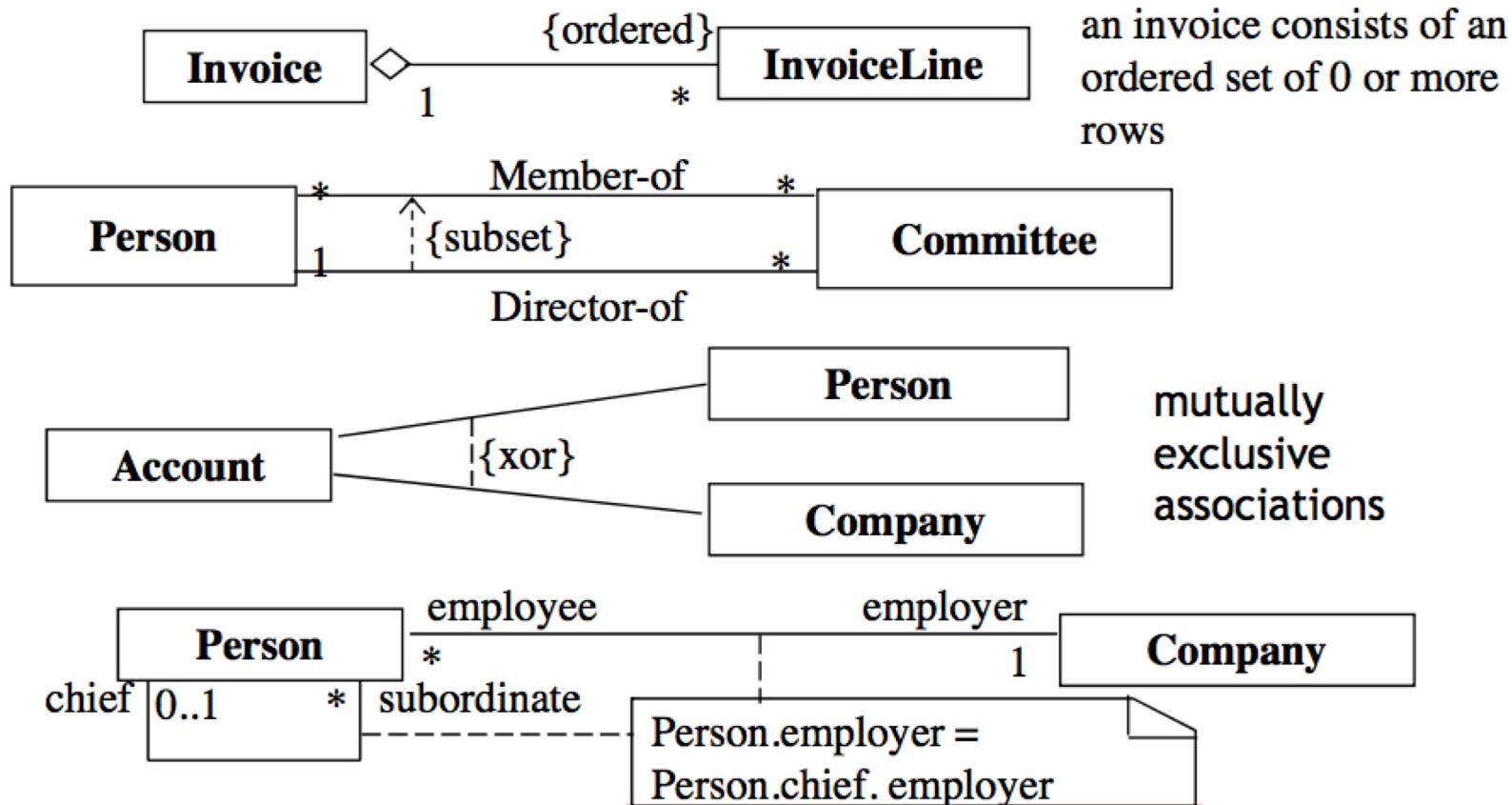
{candidate key: (name, birthDate, birthPlace)}

{deathDate > birthDate}



Credits: João Moreira

Constraints in associations



Credits: João Moreira

Key concepts

Classes

Constraints

Associations

Derived Elements

Association Classes

Generalizations

Composition & Aggregation

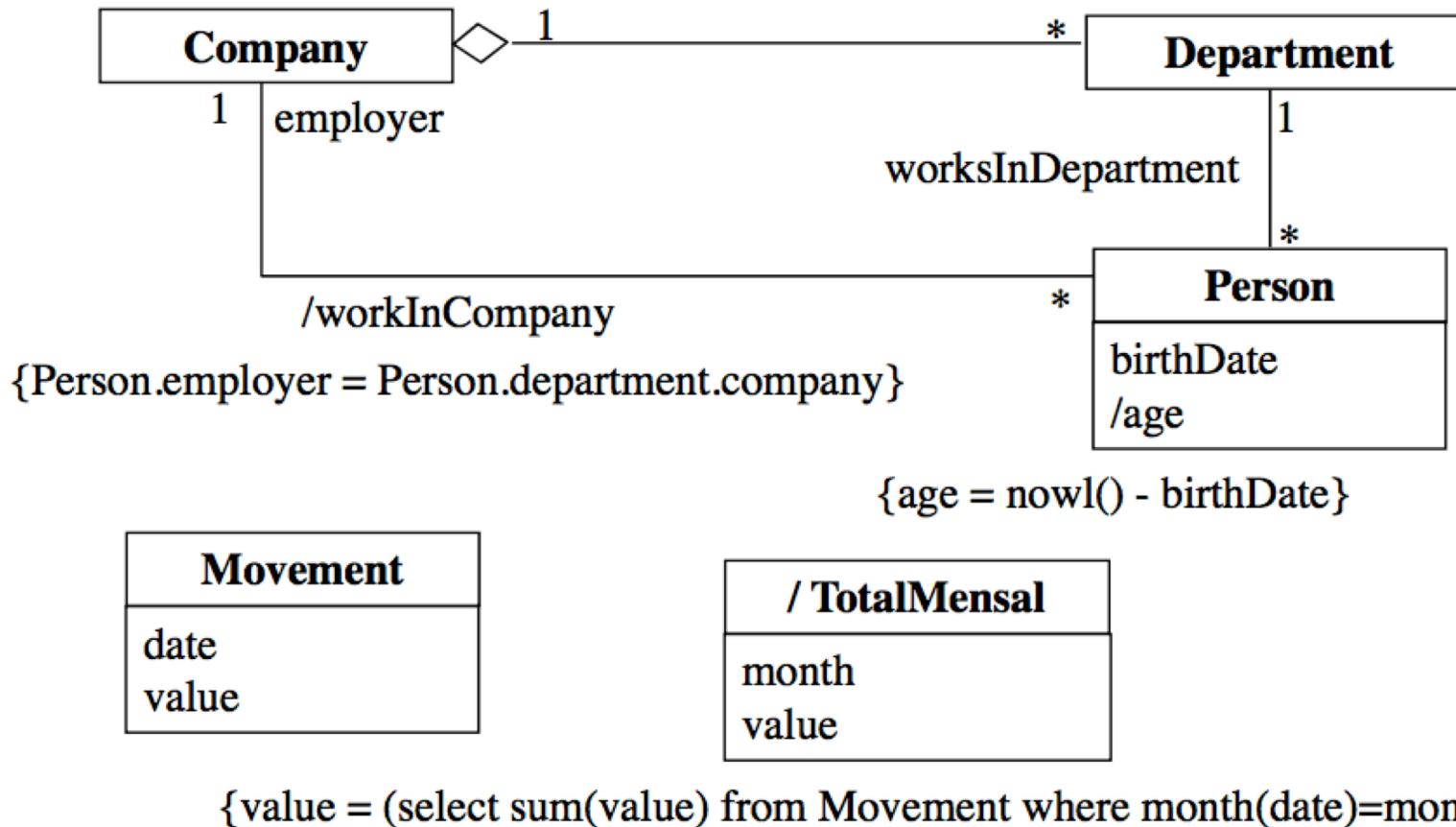
Derived Elements

Element (class, attribute or association) computed using other elements in the model

Notation: ‘/’ before the name of the derived element

Usually have an associated constraint that relates them with other elements

Derived Elements



Credits: João Moreira

Higher-Level Database Design

Unified Modeling Language (UML)

 Data modeling subset

Graphical

Key concepts

 Classes

 Associations

 Association Classes

 Generalizations

 Composition & Aggregation

 Constraints

 Derived Elements

Can be translated to relations automatically

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in
Database Systems 3rd Edition

Section 4.7 - Unified Modeling Language

From UML to Relations

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

UML key concepts

Classes

Constraints

Associations

Derived Elements

Association Classes

Generalizations

Composition & Aggregation

Classes

Every class becomes a relation

Student
SID
SName
Grade

College
CName
State
Enrollment

Student (SID, SName, Grade)

College (CName, State, Enrollment)

UML key concepts

Classes

Constraints

Associations

Derived Elements

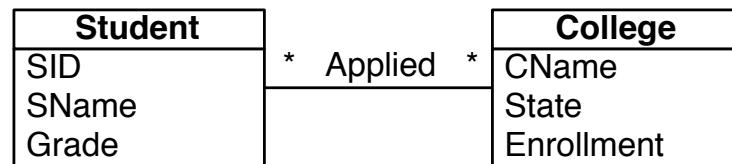
Association Classes

Generalizations

Composition & Aggregation

Many-to-many associations

Add a relation with key from each side



Student (SID, SName, Grade)

College (CName, State, Enrollment)

Applied (SID->Student, Cname->College)

Many-to-one associations

Add a foreign key to the **many** side of the relationship to the relation in the one side



Student (SID, SName, Grade, College->College)

College (CName, State, Enrollment)

Many-to-one associations

Add a relation with key from the many side



Student (SID, SName, Grade)

College (CName, State, Enrollment)

Applied (SID->Student, Cname->College)

Many-to-one associations

Add a foreign key to the many side of the relationship to the relation in the one side

- Most common

- Less relations in the schema

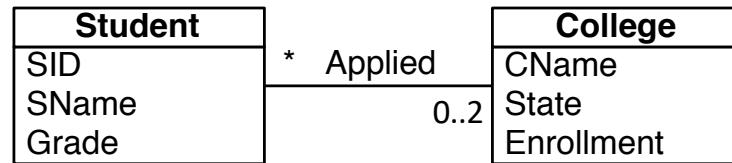
- Increased performance due to a smaller number of relations

Add a relation with key from the many side

- Increased rigour of the schema

- Increased extensibility

Question



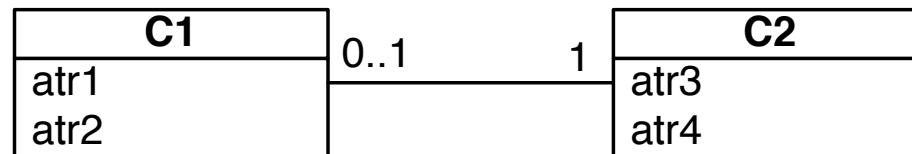
Suppose we had **0..2** on the right-hand side, so students can apply to up to 2 colleges. Is there still a way to "fold in" the association relation in this case, or must we have a separate **Applied** relation?

Yes there is a way

No, if it's not **0..1** or **1..1** then **Applied** is required

One-to-one associations

Add a foreign key from one of the relations to the other



C1 (atr1, atr2, c2_id->C2)

C2 (atr3, atr4)

Add the foreign key to the relation that is expected to have less tuples

Add a unique key constraint to the foreign key

UML key concepts

Classes

Constraints

Associations

Derived Elements

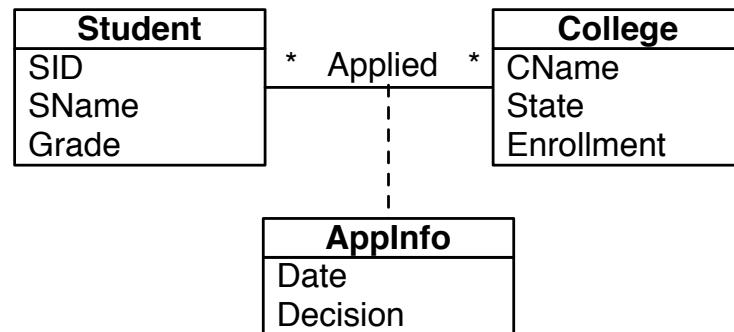
Association Classes

Generalizations

Composition & Aggregation

Association classes

Add attributes to relation for association



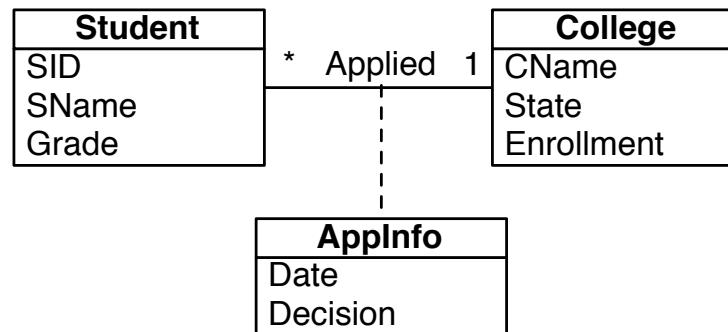
Student (SID, SName, Grade)

College (CName, State, Enrollment)

Applied (SID->Student, Cname->College, Date, Decision)

Association classes

Add attributes to relation for association

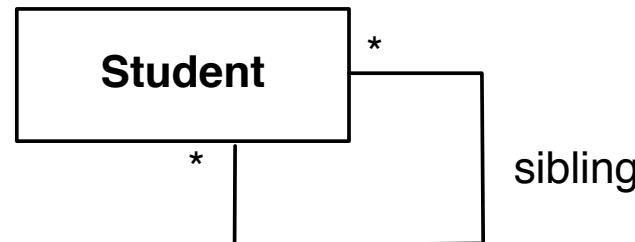


Student (SID, SName, Grade)

College (CName, State, Enrollment)

Applied (SID->Student, Cname->College, Date, Decision)

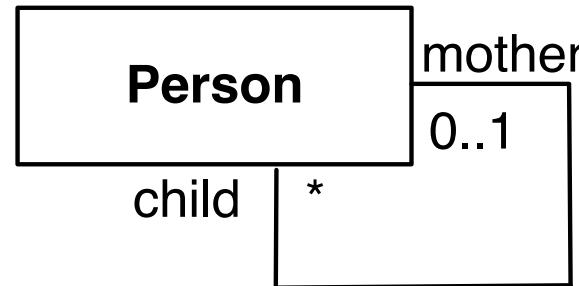
Self associations



Student (id, ...)

Sibling (sid1->Student, sid2->Student)

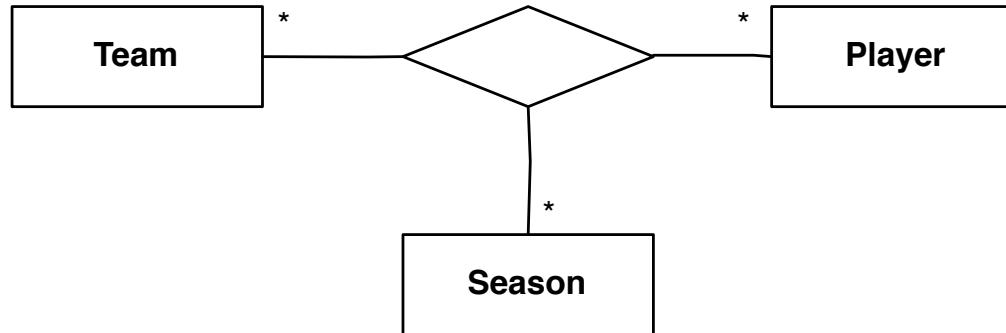
Self associations



Person (id, ...)

Relationship (mother->Person, child->Person)

Associations n-ary



Relation with key from each side

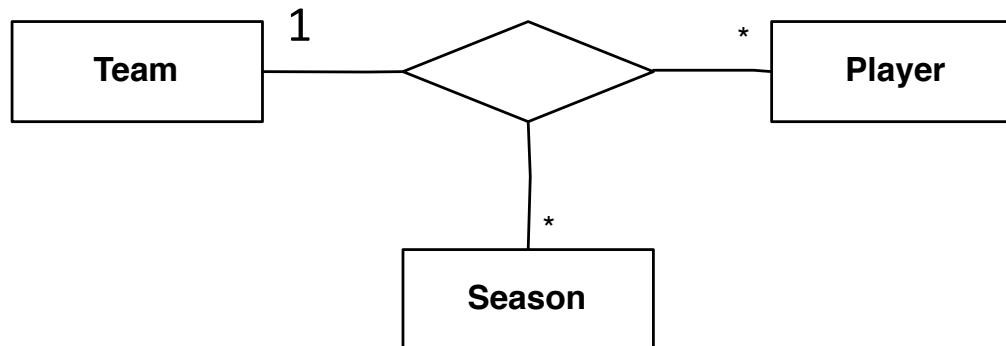
Team (ID, ...)

Player (ID, ...)

Season (ID, ...)

PlayerSeasonTeam (PlayerID->Player, SeasonID->Season, TeamID->Team)

Associations n-ary



Relation with key from each side

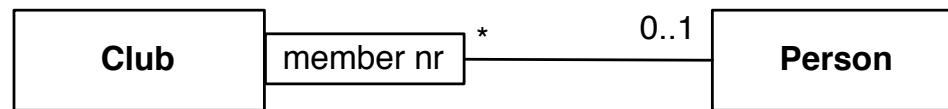
Team (ID, ...)

Player (ID, ...)

Season (ID, ...)

PlayerSeasonTeam (PlayerID->Player, SeasonID->Season, TeamID->Team)

Qualified associations



Club (ClubID, ...)

Person (PersonID, ...)

Member (ClubID->Club, PersonID->Person, MemberNr)

{ClubID, MemberNr} UK

UML key concepts

Classes

Constraints

Associations

Derived Elements

Association Classes

Generalizations

Composition & Aggregation

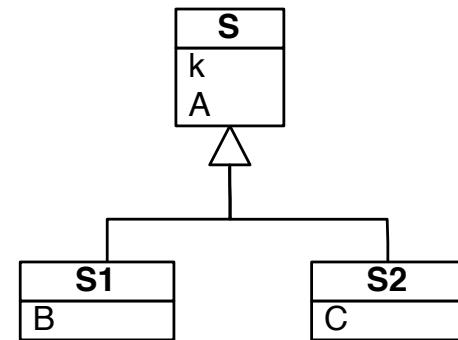
Generalizations

3 conversion strategies

E/R style

Object-oriented

Use nulls



Best translation may depend on the properties of the generalization

Generalizations – E/R style

A relation per each class

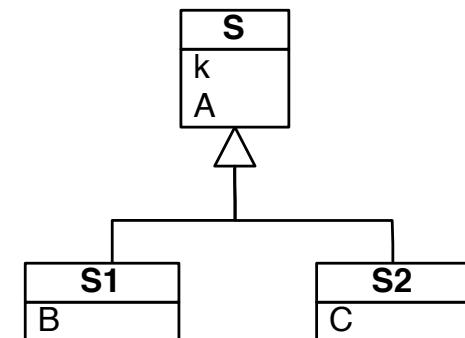
Subclass relations contain superclass key + specialized attributes

Good for overlapping generalizations with a large number of subclasses

$S(\underline{k}, A)$

$S1(\underline{k} \rightarrow S, B)$

$S2(\underline{k} \rightarrow S, C)$



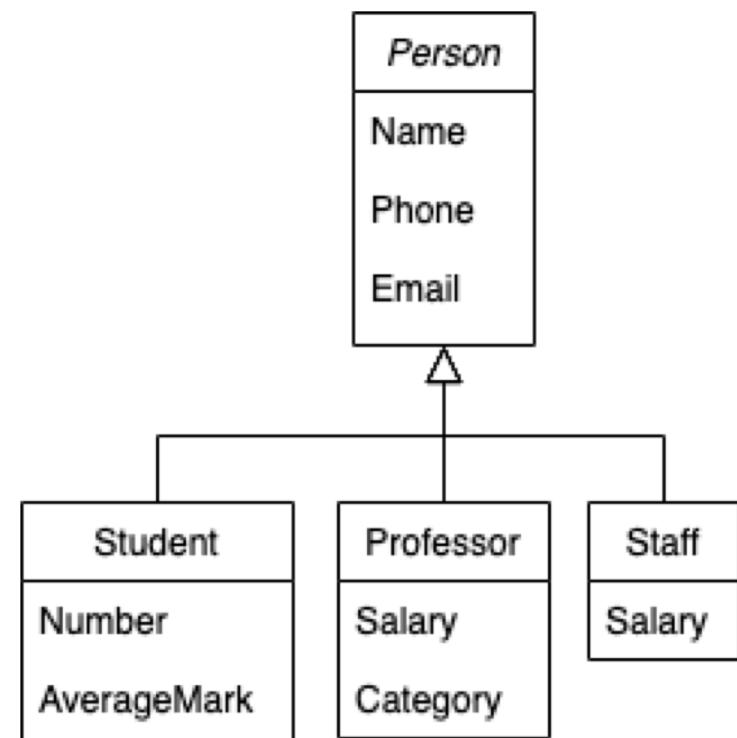
Generalizations – E/R style

Person (id, Name, Phone, Email)

Student (id->Person, Number, AverageMark)

Professor (id->Person, Salary, Category)

Staff (id->Person, Salary)



Generalizations – Object-oriented

Subclass relations contain all attributes

In complete generalizations, the relation for the superclass may be eliminated

Cannot guarantee the uniqueness of the values of the superclass

Good for

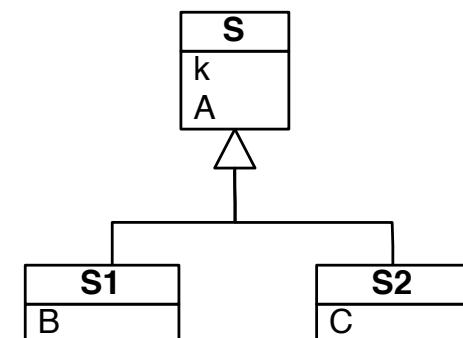
disjoint generalizations

superclass has few attributes and subclasses many attributes

S (k, A)

S1(k ->S, A, B)

S2(k->S, A, C)

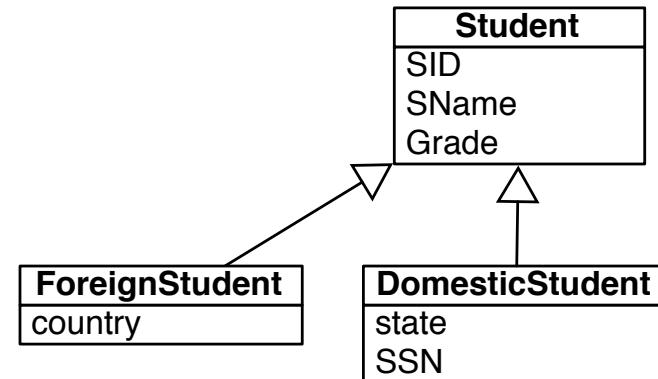


Generalizations – Object-oriented

Student (SID, SName, Grade)

ForeignStudent (SID ->Student, SName, Grade, country)

DomesticStudent (SID->Student, SName, Grade, state, SSN)



Or, because it is complete:

ForeignStudent (SID, SName, Grade, country)

DomesticStudent (SID, SName, Grade, state, SSN)

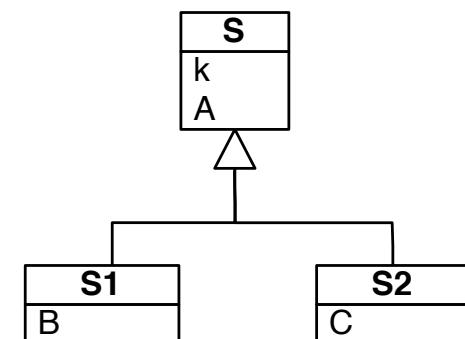
Generalizations – Use nulls

One relation with all the attributes of all the classes

NULL values on non-existing attributes for a specific object

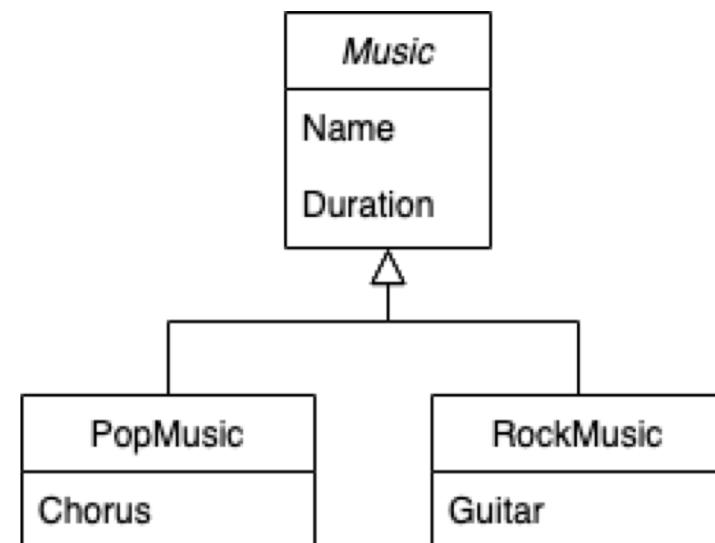
Good for heavily overlapping generalizations with a small number of subclasses

$S(k, A, B, C)$

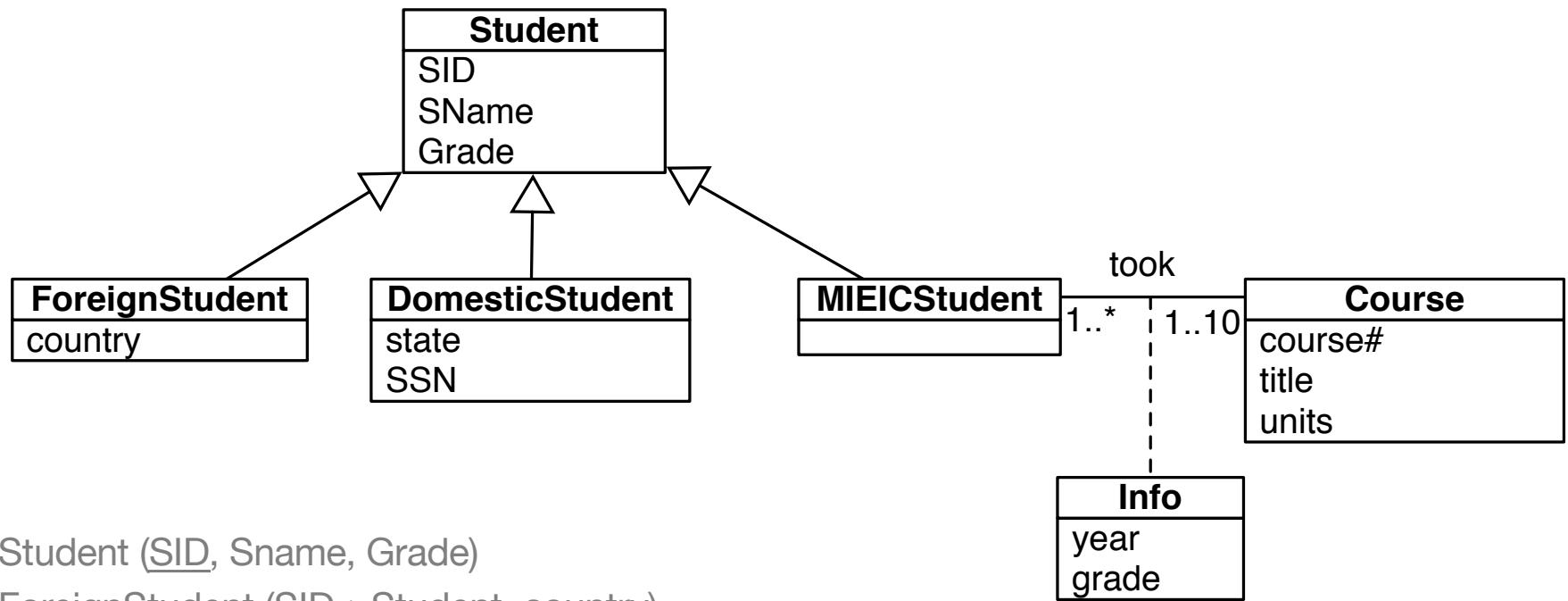


Generalizations – Use nulls

Music (id, Name, Duration, Chorus, Guitar)



Generalizations – Example



Student (SID, Sname, Grade)

ForeignStudent (SID->Student, country)

DomesticStudent (SID->Student, state, SSN)

MIEICStudent (SID->Student)

Course (course#, title, units)

Took (SID->MIEICStudent, course#->Course, year, grade)

Can we simplify the schema?

UML key concepts

Classes

Constraints

Associations

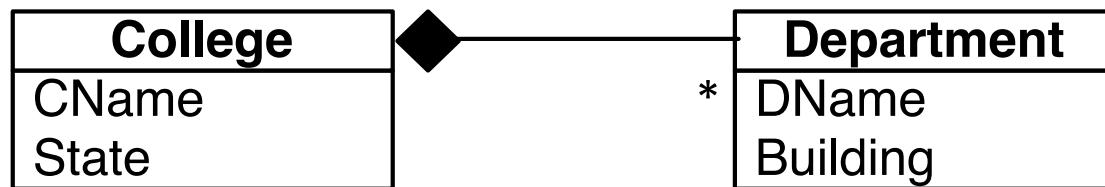
Derived Elements

Association Classes

Generalizations

Composition & Aggregation

Composition

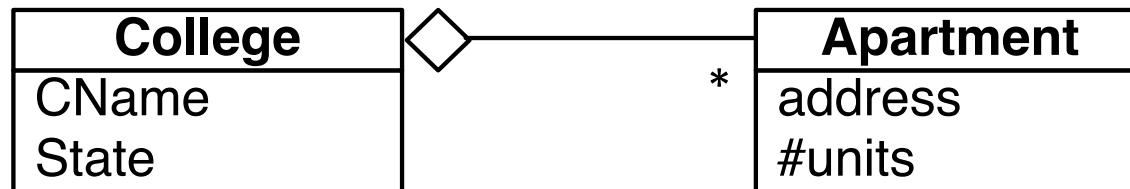


Treat it as a regular association

College (CName, State)

Department (DName, Building, CName->College)

Aggregation



Treat it as a regular association

College (CName, State)

Apartment (address, #units, CName->College)



NULL

UML key concepts

Classes

Constraints

Associations

Derived Elements

Association Classes

Generalizations

Composition & Aggregation

Constraints and Derived Elements

Constraints

NOT NULL

UNIQUE

PRIMARY KEY

FOREIGN KEY

CHECK

Ensures that the value in a column meets a specific condition

DEFAULT

Specifies a default value for a column

Derived Elements

Treat them as regular elements

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 2.1 – Basics of the Relational Model

Section 4.8 – From UML Diagrams to Relations

Section 4.6 – Converting Subclass Structures to Relations

Relational Design Theory

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom and Christopher Ré slides

Agenda

Relational Design Overview

Functional Dependencies

Closures, Superkeys and Keys

Inferring Functional Dependencies

Normal Forms

Decompositions

Relational Design

Designing a database schema

Usually many designs possible

Some are (much) better than others!

How do we choose?

Often use higher-level design tools, but ...

Some designers go straight to relations

Useful to understand why tools produce certain schemas

Design theory is about how to represent your data to avoid anomalies.

Example

College application info

SSN and name

Colleges applying to

High schools attended (with city)

Hobbies

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

Example

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

123 Ann from Palo Alto High School (PAHS) and Gunn High School (GHS) also in Palo Alto plays tennis and trumpet and applied to Stanford, Berkeley, and MIT

Apply

SSN	sName	cName	HS	HScity	hobby
123	Ann	Stanford	PAHS	Palo Alto	tennis
123	Ann	Stanford	PAHS	Palo Alto	trumpet
123	Ann	Berkeley	PAHS	Palo Alto	trumpet
.
.

12 tuples

Design Anomalies: Redundancy

Capture information multiple times

How many times do we capture the fact that

123 is the social security number of Ann?

she plays tennis?

she applied to MIT?

Apply

SSN	sName	cName	HS	HScity	hobby
123	Ann	Stanford	PAHS	Palo Alto	tennis
123	Ann	Stanford	PAHS	Palo Alto	trumpet
123	Ann	Berkeley	PAHS	Palo Alto	trumpet
.
.

Design Anomalies: Update Anomaly

Direct effect of redundancy

Can update facts in some places but not all or differently in different places

~~trumpet~~ -> piano

Apply

SSN	sName	cName	HS	HScity	hobby
123	Ann	Stanford	PAHS	Palo Alto	tennis
123	Ann	Stanford	PAHS	Palo Alto	piano
123	Ann	Berkeley	PAHS	Palo Alto	trumpet
.
.



Inconsistent
database

Design Anomalies: Deletion Anomaly

Inadvertently deletion

Example

1 new tuple about John

Someone decides surfing is an unacceptable hobby and delete the tuples about surfing

Students with surfing as their only hobby will be deleted completed

Apply

SSN	sName	cName	HS	HScity	hobby
123	Ann	Stanford	PAHS	Palo Alto	trumpet
123	Ann	Berkeley	PAHS	Palo Alto	trumpet
.
234	John	MIT	PAHS	Palo Alto	surfing

Example: New Design

College application info

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

What about this design?

Student (SSN, sName)

Apply (SSN, cName)

HighSchool (SSN, HS)

Located (HS, HScity)

Hobbies (SSN, hobby)



No redundancy

No update or deletion anomalies

Reconstruction of original data

We'll understand why this design is better and learn how to find this decomposition

Example: Modifications to the New Design

What if the high school name alone is not a key?

Student (SSN, sName)

Apply (SSN, cName)

HighSchool (SSN, HS)

Located (HS, HScity)

Hobbies (SSN, hobby)



Student (SSN, sName)

Apply (SSN, cName)

HighSchool (SSN, HS, **HScity**)

~~Located (HS, HScity)~~

Hobbies (SSN, hobby)

Example: Modifications to the New Design

What if students don't want all of their hobbies revealed to all of the colleges?

Student (SSN, sName)

Apply (SSN, cName)

HighSchool (SSN, HS, HScity)

~~Located (HS, HScity)~~

Hobbies (SSN, hobby)



Student (SSN, sName)

Apply (SSN, cName, **hobby**)

HighSchool (SSN, HS, HScity)

~~Located (HS, HScity)~~

~~Hobbies (SSN, hobby)~~

The best design also depends in what the data is representing in the real world

Design by decomposition

Start with “mega” relations containing everything

Decompose into smaller, better relations with same information

Decomposition can be done automatically

“Mega” relations + **properties of the data** →

Functional
dependencies

System decomposes based on properties

Final set of relations satisfies **normal form** →

No anomalies, no
lost information

Agenda

~~Relational Design Overview~~

Functional Dependencies

Closures, Superkeys and Keys

Inferring Functional Dependencies

Normal Forms

Decompositions

Functional Dependencies (FD)

Generalization of the notion of keys

Relational design by decomposition

To design a better schema, one 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**

$\text{GPA} > 3.8 \rightarrow \text{priority} = 1$

$3.3 < \text{GPA} \leq 3.8 \rightarrow \text{priority} = 2$

$\text{GPA} \leq 3.3 \rightarrow \text{priority} = 3$

Two tuples with same GPA have same priority

Example

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

Two tuples with same GPA have same priority

$$\forall t, u \in Student: t.GPA = u.GPA \Rightarrow t.priority = u.priority$$

GPA \rightarrow priority

Definition

A and B are attributes of a relation R

$A \rightarrow B$

A functionally determines B

$\forall t, u \in R: t.A = u.A \Rightarrow t.B = u.B$

Definition

A_1, A_2, \dots, A_n and B_1, B_2, \dots, B_m are attributes of a relation R

$$A_1, A_2, \dots, A_n \rightarrow B_1, B_2, \dots, B_m$$

The diagram illustrates a function mapping from a set of attributes A to another set of attributes B . The set A is represented by the expression A_1, A_2, \dots, A_n , which is grouped by a red bracket underneath. This bracket points to a red bar over the expression \bar{A} . Similarly, the set B is represented by the expression B_1, B_2, \dots, B_m , which is also grouped by a red bracket underneath. This bracket points to a red bar over the expression \bar{B} .

$$\forall t, u \in R: t[A_1, \dots, A_n] = u[A_1, \dots, A_n] \Rightarrow t.[B_1, \dots, B_m] = u.[B_1, \dots, B_m]$$

A Picture of FDs

$$\forall t, u \in R: t[A_1, \dots, A_n] = u[A_1, \dots, A_n] \Rightarrow t.[B_1, \dots, B_m] = u.[B_1, \dots, B_m]$$

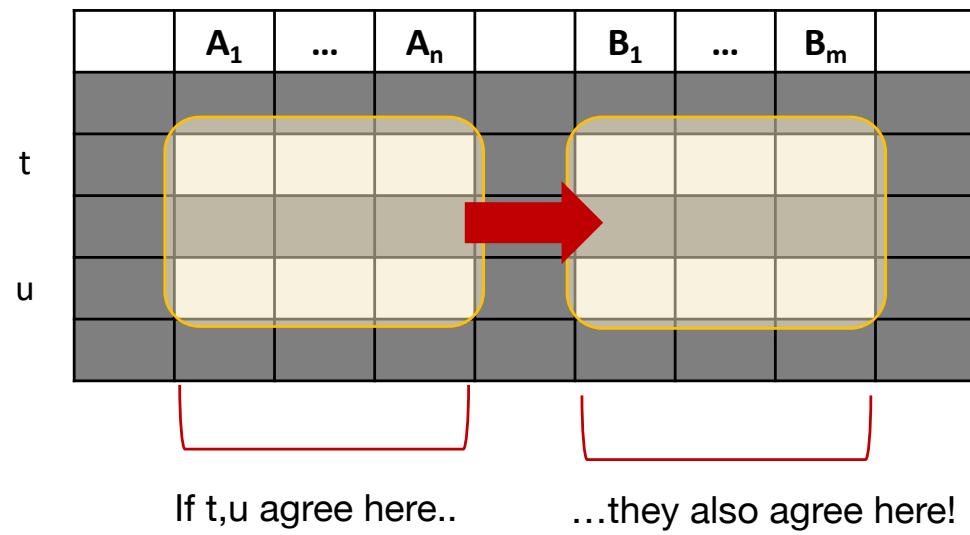
	A₁	...	A_n		B₁	...	B_m	
t								
u								

A diagram illustrating functional dependencies in a relational database. It shows two rows, *t* and *u*, of a table with columns labeled **A₁**, ..., **A_n**, **B₁**, ..., **B_m**. The first three columns (**A₁** to **A_n**) represent the primary key, while the remaining columns represent attributes. The first three columns of both rows *t* and *u* are highlighted with a yellow box, indicating they agree. A red bracket below the table spans the width of the **B₁** to **B_m** columns, indicating that if *t* and *u* agree on the primary key, they must also agree on all other attributes.

If *t,u* agree here..

A Picture of FDs

$$\forall t, u \in R: t[A_1, \dots, A_n] = u[A_1, \dots, A_n] \Rightarrow t.[B_1, \dots, B_m] = u.[B_1, \dots, B_m]$$



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

Example 1

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

What functional dependencies do you identify?

Example 1

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876 ←	Salesrep
E1111	Smith	9876 ←	Salesrep
E9999	Mary	1234	Lawyer

$\{\text{Position}\} \rightarrow \{\text{Phone}\}$

Example 2

EmpID	Name	Phone	Position
E0045	Smith	1234 →	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234 →	Lawyer

but not $\{\text{Phone}\} \rightarrow \{\text{Position}\}$

Example 3

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

SSN → sName

SSN → address



Assuming the student
doesn't move

HScode → HSname, HScity

HSname, HScity → HScode



No two high schools with the
same name in the same city

SSN → GPA

GPA → priority

SSN → priority

Example 4

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

cName \rightarrow date



Assuming every college has
a single date to receive
applications

SSN, cName \rightarrow major



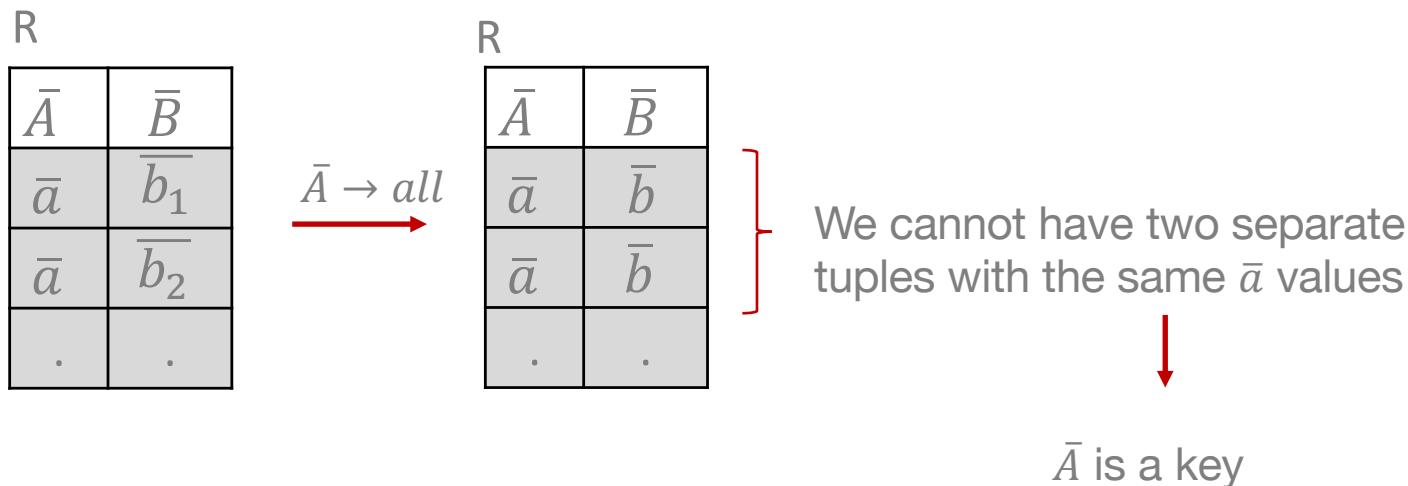
Assuming students are only
allowed to apply to a single
major at each college

Depends on the real world constraints

Keys

Relation with no duplicates: $R(\bar{A}, \bar{B})$

If $\bar{A} \rightarrow \text{all}$ attributes then \bar{A} is a key



Trivial Functional Dependency

$\bar{A} \rightarrow \bar{B}$ is a trivial dependency if $\bar{B} \subseteq \bar{A}$

R

\bar{A}	
.	.
.	.
.	.
.	.

\bar{B}

It's obvious that, if \bar{A} has the same values, then \bar{B} will have the same values

Nontrivial Functional Dependency

A functional dependency that's not a trivial one

$\bar{A} \rightarrow \bar{B}$ is a nontrivial dependency if $\bar{B} \not\subseteq \bar{A}$

R

\bar{A}	
.	.
.	.
.	.
.	.



Now the FD is saying something

Completely Nontrivial Functional Dependency

A functional dependency that's not a trivial one

$\bar{A} \rightarrow \bar{B}$ is a completely nontrivial dependency if $\bar{A} \cap \bar{B} = \emptyset$

R	
\bar{A}	\bar{B}
.	
.	
.	

These are the ones we're most interested in

Finding Functional Dependencies

Given a set of FDs, $F = \{f_1, \dots, f_n\}$, does an FD g hold?

How do we decide?

Using rules

Splitting/Combining, Trivial and Transitivity

Splitting Rule

We can split the right side of the FD

$$\bar{A} \rightarrow B_1, B_2, \dots, B_n \Rightarrow \bar{A} \rightarrow B_1, \bar{A} \rightarrow B_2, \dots, \bar{A} \rightarrow B_n$$

Can we also split left-hand-side?

$$A_1, A_2, \dots, A_n \rightarrow \bar{B} \Rightarrow A_1 \rightarrow \bar{B}, A_2 \rightarrow \bar{B}, \dots, A_n \rightarrow \bar{B} ?$$

No, for example

~~HName, HScity~~ \rightarrow HScode

~~HName~~ \rightarrow HScode

~~HScity~~ \rightarrow HScode

Combining Rule

Inverse of the splitting rule

$$\bar{A} \rightarrow B_1, \bar{A} \rightarrow B_2, \dots, \bar{A} \rightarrow B_n \Rightarrow \bar{A} \rightarrow B_1, B_2, \dots, B_n$$

Trivial-dependency rules

Reminder: $\bar{A} \rightarrow \bar{B}$ is a trivial dependency if $\bar{B} \subseteq \bar{A}$

Every left hand side determines itself or any subset of itself

$$\bar{A} \rightarrow \bar{B} \Rightarrow \bar{A} \rightarrow \bar{A} \cup \bar{B}$$

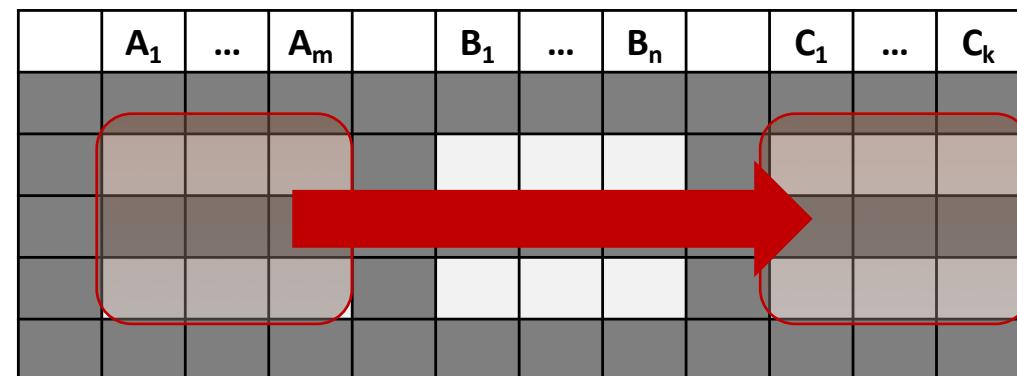
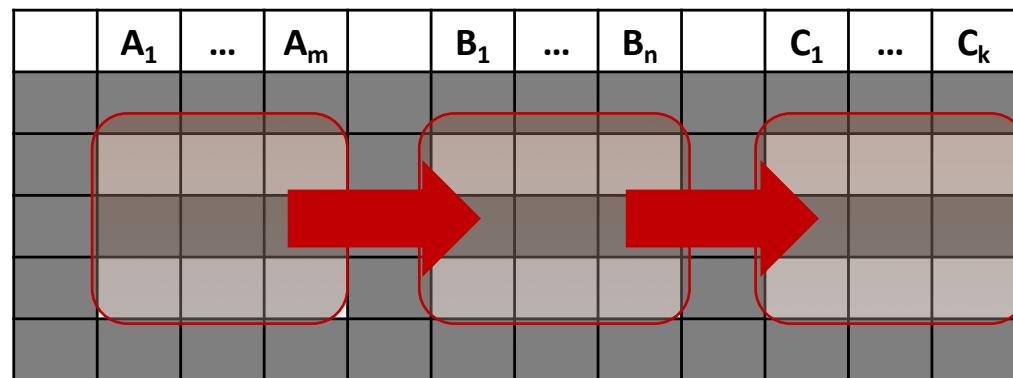
We can add to the right side of every FD what's already on the left hand side

$$\bar{A} \rightarrow \bar{B} \Rightarrow \bar{A} \rightarrow \bar{A} \cap \bar{B}$$

Also implied by the splitting rule

Transitive Rule

$$\bar{A} \rightarrow \bar{B} \wedge \bar{B} \rightarrow \bar{C} \Rightarrow \bar{A} \rightarrow \bar{C}$$



Transitive Rule

$$\bar{A} \rightarrow \bar{B} \wedge \bar{B} \rightarrow \bar{C} \Rightarrow \bar{A} \rightarrow \bar{C}$$

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



\bar{A}	\bar{B}	\bar{C}	\bar{D}
\bar{a}	\bar{b}		
\bar{a}	\bar{b}		

$$\bar{B} \rightarrow \bar{C}$$

\bar{A}	\bar{B}	\bar{C}	\bar{D}
\bar{a}	\bar{b}	\bar{c}	
\bar{a}	\bar{b}	\bar{c}	



Shows that $\bar{A} \rightarrow \bar{C}$ holds

Finding Functional Dependencies

Products

Name	Color	Category	Dep	Price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

Provided FDs

1. $\{Name\} \rightarrow \{Color\}$
2. $\{Category\} \rightarrow \{Department\}$
3. $\{Color, Category\} \rightarrow \{Price\}$

Inferred FD	Rule used
4. $\{Name, Category\} \rightarrow \{Name\}$?
5. $\{Name, Category\} \rightarrow \{Color\}$?
6. $\{Name, Category\} \rightarrow \{Category\}$?
7. $\{Name, Category\} \rightarrow \{Color, Category\}$?
8. $\{Name, Category\} \rightarrow \{Price\}$?

Finding Functional Dependencies

Products

Name	Color	Category	Dep	Price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

Provided FDs

1. $\{\text{Name}\} \rightarrow \{\text{Color}\}$
2. $\{\text{Category}\} \rightarrow \{\text{Department}\}$
3. $\{\text{Color}, \text{Category}\} \rightarrow \{\text{Price}\}$

Inferred FD	Rule used
4. $\{\text{Name}, \text{Category}\} \rightarrow \{\text{Name}\}$	Trivial
5. $\{\text{Name}, \text{Category}\} \rightarrow \{\text{Color}\}$	Transitive (4, 1)
6. $\{\text{Name}, \text{Category}\} \rightarrow \{\text{Category}\}$	Trivial
7. $\{\text{Name}, \text{Category}\} \rightarrow \{\text{Color}, \text{Category}\}$	Combining rule (5, 6)
8. $\{\text{Name}, \text{Category}\} \rightarrow \{\text{Price}\}$	Transitive (7, 3)

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 3.1 – Functional Dependencies

Section 3.2 – Rules About Functional Dependencies

Section 3.3 – Design of Relational Database Schemas

Section 3.4 – Decomposition: The Good, Bad, and Ugly

Section 3.5 – Third Normal Form

Relational Design Theory

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom and Christopher Ré slides

Agenda

~~Relational Design Overview~~

~~Functional Dependencies~~

Closures, Superkeys and Keys

Inferring Functional Dependencies

Normal Forms

Decompositions

Closure of attributes

Given a relation, FDs, set of attributes \bar{A} , find all B such that $\bar{A} \rightarrow B$

\bar{A}^+ is the closure of \bar{A}

Finding the set of attributes functionally determined by $\{A_1, \dots, A_n\}^+$

Algorithm

Start with $\{A_1, \dots, A_n\}$

Repeat until no change:

If $\bar{A} \rightarrow \bar{B}$ and \bar{A} in set, add \bar{B} to set

Applying combining
and transitive rules

Closure example

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

FD1. SSN \rightarrow sName, address, GPA

FD2. GPA \rightarrow priority

FD3. HScode \rightarrow HSname, HScity

Compute $\{SSN, HScode\}^+$

$\{SSN, HScode\}$

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

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

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



FD1



FD2



FD3

Closure example

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

FD1. SSN → sName, address, GPA

FD2. GPA → priority

FD3. HScode → HSname, HScity

$\{SSN, HScode\}^+$

{SSN, HScode, sName, address, GPA, priority, HSname, HScity}

All attributes of Student

Key for the relation

Exercise

$R(A, B, C, D, E, F)$

$\{A, B\} \rightarrow \{C\}$

$\{A, D\} \rightarrow \{E\}$

$\{B\} \rightarrow \{D\}$

$\{A, F\} \rightarrow \{B\}$

Compute $\{A, B\}^+$

Compute $\{A, F\}^+$

Exercise

$R(A, B, C, D, E, F)$

$\{A, B\} \rightarrow \{C\}$

$\{A, D\} \rightarrow \{E\}$

$\{B\} \rightarrow \{D\}$

$\{A, F\} \rightarrow \{B\}$

Compute $\{A, B\}^+ = \{A, B\}$

Compute $\{A, F\}^+ = \{A, F\}$

Exercise

$R(A, B, C, D, E, F)$

$\{A, B\} \rightarrow \{C\}$

$\{A, D\} \rightarrow \{E\}$

$\{B\} \rightarrow \{D\}$

$\{A, F\} \rightarrow \{B\}$

Compute $\{A, B\}^+ = \{A, B, C, D\}$

Compute $\{A, F\}^+ = \{A, F, B\}$

Exercise

$R(A, B, C, D, E, F)$

$\{A, B\} \rightarrow \{C\}$

$\{A, D\} \rightarrow \{E\}$

$\{B\} \rightarrow \{D\}$

$\{A, F\} \rightarrow \{B\}$

Compute $\{A, B\}^+ = \{A, B, C, D, E\}$

Compute $\{A, F\}^+ = \{A, F, B, C, D\}$

Exercise

$R(A, B, C, D, E, F)$

$\{A, B\} \rightarrow \{C\}$

$\{A, D\} \rightarrow \{E\}$

$\{B\} \rightarrow \{D\}$

$\{A, F\} \rightarrow \{B\}$

Compute $\{A, B\}^+ = \{A, B, C, D, E\}$

Compute $\{A, F\}^+ = \{A, F, B, C, D, E\}$

Closure and keys

Is \bar{A} a key for a relation R with a set of FDs?

If \bar{A}^+ contains all attributes of R, then \bar{A} is a key

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 the subsets in increasing order

If AB is a key, $AB \rightarrow$ all attributes, every superset of AB is also a key

Start with single attributes, then go to pairs and so on

Find all keys - Example

Compute X^+ , for every set of attributes X in R (A,B,C,D)

$\{A, B\} \rightarrow \{C\}$

$\{A, D\} \rightarrow \{B\}$

$\{B\} \rightarrow \{D\}$

$$\{A\}^+ = \{A\}$$

$$\{B\}^+ = \{B, D\}$$

$$\{C\}^+ = \{C\}$$

$$\{D\}^+ = \{D\}$$

Find all keys - Example

Compute X^+ , for every set of attributes X in R (A,B,C,D)

$\{A, B\} \rightarrow \{C\}$

$\{A, D\} \rightarrow \{B\}$

$\{B\} \rightarrow \{D\}$

$\{A\}^+ = \{A\}$

$\{B\}^+ = \{B, D\}$

$\{C\}^+ = \{C\}$

$\{D\}^+ = \{D\}$

$\{A, B\}^+ = \{A, B, C, D\}$

$\{A, C\}^+ = \{A, C\}$

$\{A, D\}^+ = \{A, B, C, D\}$

$\{B, C\}^+ = \{B, C, D\}$

$\{B, D\}^+ = \{B, D\}$

$\{C, D\}^+ = \{C, D\}$

Find all keys - Example

Compute X^+ , for every set of attributes X in R (A,B,C,D)

$$\{A, B\} \rightarrow \{C\}$$

$$\{A, D\} \rightarrow \{B\}$$

$$\{B\} \rightarrow \{D\}$$

$\{A\}^+ = \{A\}$
$\{B\}^+ = \{B, D\}$
$\{C\}^+ = \{C\}$
$\{D\}^+ = \{D\}$
$\{A, B\}^+ = \{A, B, C, D\}$
$\{A, C\}^+ = \{A, C\}$
$\{A, D\}^+ = \{A, B, C, D\}$
$\{B, C\}^+ = \{B, C, D\}$
$\{B, D\}^+ = \{B, D\}$
$\{C, D\}^+ = \{C, D\}$
$\{A, B, C\}^+ = \{A, B, D\}^+ = \{A, C, D\}^+ = \{A, B, C, D\}$
$\{B, C, D\}^+ = \{B, C, D\}$

→ Don't need to compute

Find all keys - Example

Compute X^+ , for every set of attributes X in R (A,B,C,D)

$$\{A, B\} \rightarrow \{C\}$$

$$\{A, D\} \rightarrow \{B\}$$

$$\{B\} \rightarrow \{D\}$$

$$\{A\}^+ = \{A\}$$

$$\{B\}^+ = \{B, D\}$$

$$\{C\}^+ = \{C\}$$

$$\{D\}^+ = \{D\}$$

$$\{A, B\}^+ = \{A, B, C, D\}$$

$$\{A, C\}^+ = \{A, C\}$$

$$\{A, D\}^+ = \{A, B, C, D\}$$

$$\{B, C\}^+ = \{B, C, D\}$$

$$\{B, D\}^+ = \{B, D\}$$

$$\{C, D\}^+ = \{C, D\}$$

$$\{A, B, C\}^+ = \{A, B, D\}^+ = \{A, C, D\}^+ = \{A, B, C, D\}$$

$$\{B, C, D\}^+ = \{B, C, D\}$$

$$\{A, B, C, D\}^+ = \{A, B, C, D\} \longrightarrow \text{Don't need to compute}$$

Find all keys - Example

Compute X^+ , for every set of attributes X in R (A,B,C,D)

$\{A,B\} \rightarrow \{C\}$

$\{A, D\} \rightarrow \{B\}$

$\{B\} \rightarrow \{D\}$

$\{A\}^+ = \{A\}$
 $\{B\}^+ = \{B, D\}$
 $\{C\}^+ = \{C\}$
 $\{D\}^+ = \{D\}$
 $\{A, B\}^+ = \{A, B, C, D\}$
 $\{A, C\}^+ = \{A, C\}$
 $\{A, D\}^+ = \{A, B, C, D\}$
 $\{B, C\}^+ = \{B, C, D\}$
 $\{B, D\}^+ = \{B, D\}$
 $\{C, D\}^+ = \{C, D\}$
 $\{A, B, C\}^+ = \{A, B, D\}^+ = \{A, C, D\}^+ = \{A, B, C, D\}$
 $\{B, C, D\}^+ = \{B, C, D\}$
 $\{A, B, C, D\}^+ = \{A, B, C, D\}$

(Super)keys

$\{A, B\}$
 $\{A, D\}$
 $\{A, B, C\}$
 $\{A, B, D\}$
 $\{A, C, D\}$
 $\{A, B, C, D\}$

Superkeys and keys

A superkey is a set of attributes A_1, \dots, A_n such that for *any other* attribute B in R , we have $\{A_1, \dots, A_n\} \rightarrow B$
all attributes are functionally 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

Example of finding keys

Product (name, price, category, color)

{name, category} → price

{category} → color

What is a key?

Example of finding keys

Product (name, price, category, color)

$\{\text{name}, \text{category}\} \rightarrow \text{price}$

$\{\text{category}\} \rightarrow \text{color}$

$\{\text{name}, \text{category}\}^+ = \{\text{name}, \text{price}, \text{category}, \text{color}\}$

= the set of all attributes

this is a **superkey**

this is a **key**, since neither *name* nor *category* alone is a superkey

Agenda

~~Relational Design Overview~~

~~Functional Dependencies~~

~~Closures, Superkeys and Keys~~

Inferring Functional Dependencies

Normal Forms

Decompositions

Inferring FDs

S1 and S2 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}

Inferring FDs

How to test if $\bar{A} \rightarrow \bar{B}$ follows from S ?

Compute \bar{A}^+ based on S and check if \bar{B} is in set

Armstrong's Axioms

Set of rules that are what's called complete

If one thing about functional dependencies can be proved from another, then it can be 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

Projecting a set of FDs

Input: relation R; FDs for R; relation $R_1 = \pi_L(R)$

Output: T, the set of FDs that hold in R_1

For each set of attributes \bar{X} of R_1 , compute \bar{X}^+

With respect to the FDs for R. These FDs may involve attributes that are in R and not in R_1

$$\bar{X} \cap \bar{A} = \emptyset$$



Add to T all nontrivial FDs $\bar{X} \rightarrow \bar{A}$, such that $\bar{X}^+ \supseteq \bar{A}$ and \bar{A} contains attributes of R_1

For minimal base, repeat until no changes

Remove FDs from T that follow from the other FDs in T

Replace $\bar{Y} \rightarrow \bar{B}$ by $\bar{Z} \rightarrow \bar{B}$ if \bar{Z} is \bar{Y} with one of its attributes removed

Projecting a set of FDs Example

Input: R (A, B, C, D); FDs: A->B, B->C, C->D; R₁= (A, C, D)

Compute the closure for all the subsets of {A, C, D}

{A}⁺={A, B, C, D} thus **A->C** and **A->D** hold in R₁

No need to consider any superset of {A}, every FD would follow an FD with only A on the left side (e.g.: AC->D follows from A->D)

{C}⁺={C, D} thus **C->D** holds in R₁

{D}⁺={D}

{C,D}⁺={C,D}

If \bar{X} is a key of R₁,
no need to close
supersets of \bar{X}

No need to close the empty set and the set of all attributes

Cannot yield a nontrivial FD

Minimal base: A->C and C->D

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 3.1 – Functional Dependencies

Section 3.2 – Rules About Functional Dependencies

Section 3.3 – Design of Relational Database Schemas

Section 3.4 – Decomposition: The Good, Bad, and Ugly

Section 3.5 – Third Normal Form

Relational Design Theory

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom and Christopher Ré slides

Agenda

~~Relational Design Overview~~

~~Functional Dependencies~~

~~Closures, Superkeys and Keys~~

~~Inferring Functional Dependencies~~

Normal Forms

Decompositions

Normal Forms

1st Normal Form (1NF)

All tables are flat

2nd Normal Form

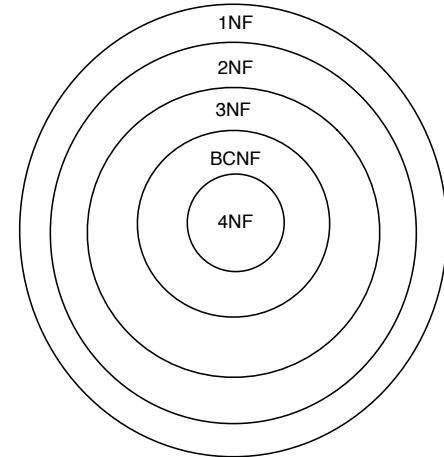
disused

Boyce-Codd Normal Form (BCNF)

3rd Normal Form (3NF)

4th and 5th Normal Forms

see text books



DB designs based on functional dependencies, intended to prevent data anomalies

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

Student	Courses
Mary	{CS145,CS229}
Joe	{CS145,CS106}
...	...



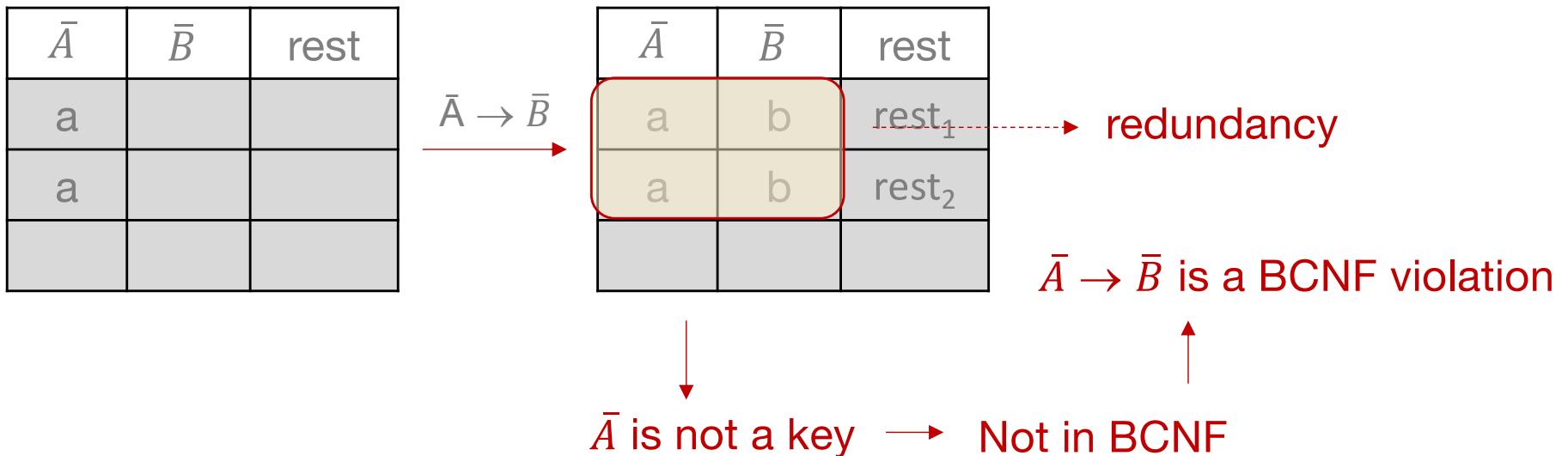
Student	Course
Mary	CS145
Mary	CS229
Joe	CS145
Joe	CS106

Boyce-Codd Normal Form

Relation R with FDs is in BCNF if

For each nontrivial $\bar{A} \rightarrow \bar{B}$, \bar{A} is a (super)key

Why do we have a bad design when this doesn't happen?



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*

Student-Professor

<u>SID</u>	<u>PID</u>	PName
1	3	Smith
2	2	Bayer

PID->PName



Student-Professor

<u>SID</u>	<u>PID</u>
1	3
2	2

Professor

<u>PID</u>	PName
3	Smith
2	Bayer

BCNF? Example #1

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

$\text{SSN} \rightarrow \text{sName, address, GPA}$

$\text{GPA} \rightarrow \text{priority}$

$\text{HScode} \rightarrow \text{HSname, HScity}$

Keys of the relation?

$\{\text{SSN, HScode}\}$

Does every FD have a key on its left-hand side?

No, none.

BCNF? Example #2

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

SSN, cName, state → date, major

Keys of the relation?

{SSN, cName, state}

Does every FD have a key on its left-hand side?

Yes.

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 $\bar{A} \rightarrow \bar{B}$,
 \bar{A} is a (super)key **or**
 \bar{B} consists of prime attributes only

3NF Example

Bookings (title, theater, city)

$\text{theater} \rightarrow \text{city}$

$\text{title, city} \rightarrow \text{theater}$



No booking of a movie
in two theaters of the
same city

Keys of the relation?

$\{\text{title, city}\}, \{\text{theater, title}\}$

BCNF?

FD $\text{theater} \rightarrow \text{city}$ is a BCNF violation

3NF?

FD $\text{theater} \rightarrow \text{city}$ has only prime attributes on its right-side

FD $\text{title, city} \rightarrow \text{theater}$ has a key on its left-hand side and only prime attributes on its right-side

Agenda

~~Relational Design Overview~~

~~Functional Dependencies~~

~~Closures, Superkeys and Keys~~

~~Inferring Functional Dependencies~~

~~Normal Forms~~

Decompositions

Decomposition of a relational schema

R1 and R2 are a decomposition of R (A_1, \dots, A_n) if

$$R_1 = \pi_{B_1, \dots, B_n}(R)$$

$$R_2 = \pi_{C_1, \dots, C_n}(R)$$

$$\{B_1, \dots, B_n\} \cup \{C_1, \dots, C_n\} = \{A_1, \dots, A_n\}$$

\bar{B}

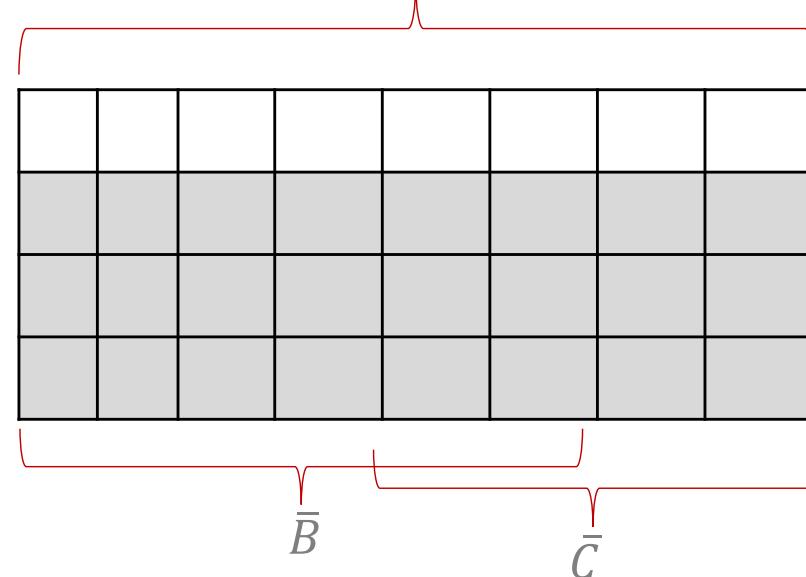
\bar{C}

\bar{A}

\bar{A}

If: $R_1 \bowtie R_2 = R$

Lossless join property



Natural Join (\bowtie)

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	John	3.8	50

Apply

sID	cName	major	dec
12	Stanford	CS	Y
23	MIT	CS	N



Student \bowtie Apply

sID	sName	GPA	HS	cName	major	dec
12	Mary	3.5	90	Stanford	CS	Y
23	John	3.8	50	MIT	CS	N

Decomposition Example #1

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

S_1 (SSN, sName, address, **HScode**, GPA, priority)

S_2 (**HScode**, HSname, HScity)

Is it a correct decomposition?

$$\bar{B} \cup \bar{C} = \bar{A}$$

$$S_1 \bowtie S_2 = Student$$

Decomposition Example #2

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

S_1 (SSN, SName, address, HScode, HSname, HScity)

S_2 (SName, HSname, GPA, priority)

Is it a correct decomposition?

$$\bar{B} \cup \bar{C} = \bar{A}$$

$$S_1 \bowtie S_2 = Student ?$$

SName and HSname
may not be unique

BCNF decomposition algorithm

Input: relation R + FDs for R

Output: decomposition of R into BCNF relations with “lossless join”

Compute keys for $\textcolor{red}{R}$

Repeat until all relations are in BCNF:

Pick any $\textcolor{red}{R}'$ with $\bar{A} \rightarrow \bar{B}$ that violates BCNF

Decompose $\textcolor{red}{R}'$ into $\textcolor{red}{R}_1(\bar{A}, \bar{B})$ and $\textcolor{red}{R}_2(\bar{A}, \text{rest})$

Compute FDs for $\textcolor{red}{R}_1$ and $\textcolor{red}{R}_2$

Compute keys for $\textcolor{red}{R}_1$ and $\textcolor{red}{R}_2$

R'		
\bar{A}	\bar{B}	rest



R_1	
\bar{A}	\bar{B}

R_2	
\bar{A}	rest

BCNF Decomposition Example

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

$\text{SSN} \rightarrow \text{sName, address, GPA}$; $\text{GPA} \rightarrow \text{priority}$; $\text{HScode} \rightarrow \text{HSname, HScity}$

Key: {SSN, HScode}

Pick a BCNF violation

$\text{HScode} \rightarrow \text{HSname, HScity}$

Decompose Student

S1 (HScode, HSname, HScity)

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

Compute FDs and keys for S1

$\text{HScode} \rightarrow \text{HSname, HScity}$

Key: {HScode}

S1 is in BCNF

Compute FDs and keys for S2

$\text{SSN} \rightarrow \text{sName, address, GPA}$

$\text{GPA} \rightarrow \text{priority}$

Key: {SSN, HScode}

S2 is not in BCNF

} BCNF violations

BCNF Decomposition Example

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

$\text{SSN} \rightarrow \text{sName, address, GPA}$; $\text{GPA} \rightarrow \text{priority}$; $\text{HScode} \rightarrow \text{HSname, HScity}$

Key: {SSN, HScode}

Pick a BCNF violation

$\text{GPA} \rightarrow \text{priority}$

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

S3 (GPA, priority)

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

Compute FDs and keys for S3

$\text{GPA} \rightarrow \text{priority}$

Key: {GPA}

S3 is in BCNF

Compute FDs and keys for S4

$\text{SSN} \rightarrow \text{sName, address, GPA}$ } BCNF violation

Key: {SSN, HScode}

S4 is not in BCNF

BCNF Decomposition Example

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

$\text{SSN} \rightarrow \text{sName, address, GPA}$; $\text{GPA} \rightarrow \text{priority}$; $\text{HScode} \rightarrow \text{HSname, HScity}$

Key: {SSN, HScode}

Pick a BCNF violation

$\text{SSN} \rightarrow \text{sName, address, GPA}$

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

S5 (SSN, sName, address, GPA)

S6 (SSN, HScode)

Compute FDs and keys for S5

$\text{SSN} \rightarrow \text{sName, address, GPA}$

Key: {SSN}

S5 is in BCNF

Compute FDs and keys for S6

Key: {SSN, HScode}

S6 is in BCNF

BCNF Decomposition Example

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

$\text{SSN} \rightarrow \text{sName, address, GPA}$; $\text{GPA} \rightarrow \text{priority}$; $\text{HScode} \rightarrow \text{HSname, HScity}$

Key: {SSN, HScode}

S1 (HScode, HSname, HScity) → Information about high schools

S3 (GPA, priority) → Information about GPA and priorities

S5 (SSN, sName, address, GPA) → Information about students

S6 (SSN, HScode) → Information about the high schools students went

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 $\bar{A} \rightarrow \bar{B}$ that violates BCNF

Different answers depending on the chosen R'

Extend FD that is used for decomposition (if $A \rightarrow B$ then $A \rightarrow BA^+$)

Decompose R' into $R_1(\bar{A}, \bar{B})$ and $R_2(\bar{A}, \text{rest})$

Compute FDs for R_1 and R_2

See “Projecting a set of FDs” slides

Compute keys for R_1 and R_2

Exercise

Consider the following relation and FDs

Movie (title, year, studioName, president, presAddr)

title, year \rightarrow studioName

studioName \rightarrow president

president \rightarrow presAddr

Decompose into BCNF relations.

Exercise

Movie (title, year, studioName, president, presAddr)

title, year \rightarrow studioName

studioName \rightarrow president

president \rightarrow presAddr

Key: {title, year}

Pick a BCNF violation

studioName \rightarrow president

Decompose Student

S1 (studioName, president)

S2 (studioName, title, year,
presAddr)

Compute FDs and keys for S1

studioName \rightarrow president

Key: {studioName}

S1 is in BCNF

Compute FDs and keys for S2

title, year \rightarrow studioName

studioName \rightarrow presAddr

Key: {title, year}

S2 is not in BCNF

BCNF
violation



Exercise

Movie (title, year, studioName, president, presAddr)

title, year \rightarrow studioName

studioName \rightarrow president

president \rightarrow presAddr

Key: {title, year}

Pick a BCNF violation

studioName \rightarrow presAddr

Compute FDs and keys for S3

studioName \rightarrow presAddr

Key: {studioName}

S3 is in BCNF

Decompose S2 (studioName, title, year, presAddr)

S3 (studioName, presAddr)

S4 (studioName, title, year)

Compute FDs and keys for S4

title, year \rightarrow studioName

Key: {title, year}

S4 is in BCNF

Exercise

Movie (title, year, studioName, president, presAddr)

title, year -> studioName

studioName -> president

president -> presAddr

Key: {title, year}

S1 (studioName, president)

S3 (studioName, presAddr)

S4 (studioName, title, year)

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 3.1 – Functional Dependencies

Section 3.2 – Rules About Functional Dependencies

Section 3.3 – Design of Relational Database Schemas

Section 3.4 – Decomposition: The Good, Bad, and Ugly

Section 3.5 – Third Normal Form

Relational Design Theory

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom and Christopher Ré slides

Does BCNF guarantee a good decomposition?

Remove anomalies?

Yes

Can logically reconstruct original relation?

$$R_1 \bowtie R_2 = R ?$$

Too few or too many tuples?

R	A	B	C
	1	2	3
	4	2	5

R ₁	A	B
	1	2
	4	2

R ₂	B	C
	2	3
	2	5

$$R_1 \bowtie R_2 =$$

123
125
423
425

Does BCNF guarantee a good decomposition?

R	A	B	C
1	1	2	3
4	4	2	5

R_1	A	B
1	1	2
4	4	2

R_2	B	C
2	2	3
2	2	5

$$R_1 \bowtie R_2 = \begin{matrix} 123 \\ 125 \\ 423 \\ 425 \end{matrix}$$

What happened?

Not a BCNF decomposition

R_1 and R_2 would demand $B \rightarrow A$ or $B \rightarrow C$ and none hold

BCNF always lossless

BCNF decomposition is standard practice - very powerful & widely used!

The Chase Test for Lossless Join

S (A, B, C, D) decomposed in
S1 (A, D), S2 (A,C) and S3 (B, C, D)

DFs

$A \rightarrow B$; $A \rightarrow C$; $CD \rightarrow A$

Does this decomposition ensure lossless joins?

The Chase Test for Lossless Join

$S(A, B, C, D)$ decomposed in
 $S1(A, D)$, $S2(A, C)$ and $S3(B, C, D)$
 $A \rightarrow B$; $A \rightarrow C$; $CD \rightarrow A$

Build the tableau

One line per decomposed relation

A letter per each attribute in the decomposed relation

Subscript the letter with i , if the attribute is not in S_i

A	B	C	D



A	B	C	D
a			d
a		c	
	b	c	d



A	B	C	D
a	b_1	c_1	d
a	b_2	c	d_2
a_3	b	c	d

$S1(A, D)$

$S2(A, C)$

$S3(B, C, D)$

The Chase Test for Lossless Join

S (A, B, C, D) decomposed in
S1 (A, D), S2 (A,C) and S3 (B, C, D)
 $A \rightarrow B$; $B \rightarrow C$; $CD \rightarrow A$

A	B	C	D
a	b_1	c_1	d
a	b_2	c	d_2
a_3	b	c	d

Iterations

$A \rightarrow B$ tells us that the first two rows must agree in the B attribute, that is, $b_1 = b_2$
From $B \rightarrow C$, we know that $c_1 = c$
From $CD \rightarrow A$, we know that $a = a_3$

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2

$CD \rightarrow A$

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2

A	B	C	D
a	b_1	c_1	d
a	b_1	c	d_2

$B \rightarrow C$

$A \rightarrow B$

The Chase Test for Lossless Join

S (A, B, C, D) decomposed in
S1 (A, D), S2 (A,C) and S3 (B, C, D)
 $A \rightarrow B$; $B \rightarrow C$; $CD \rightarrow A$

Conclusion

If the final table has a line without subscripts, it is a lossless join decomposition

A	B	C	D
a	b_1	c	d
a	b_1	c	d_2
a	b	c	d

Exercise

Consider the following relation and FDs

Movie (title, year, studioName, president, presAddr)

title, year \rightarrow studioName

studioName \rightarrow president

president \rightarrow presAddr

Test if the following decomposition is lossless:

S1 (studioName, president)

S3 (studioName, presAddr)

S4 (studioName, title, year)

Exercise

Movie (title, year, studioName, president, presAddr) decomposed in
S1 (studioName, president); S3 (studioName, presAddr);
S4 (studioName, title, year)

title, year -> studioName
studioName -> president
president -> presAddr

Build the tableau

A (title)	B (year)	C (studioName)	D (president)	E (presAddr)	
a ₁	b ₁	c	d	e ₁	S1
a ₂	b ₂	c	d ₂	e	S3
a	b	c	d ₃	e ₃	S4

Exercise

Movie (title, year, studioName, president, presAddr) decomposed in

S1 (studioName, president); S3 (studioName, presAddr); S4 (studioName, title, year)

title, year \rightarrow studioName; studioName \rightarrow president; president \rightarrow presAddr

title	year	studio Name	president	pres Addr
a ₁	b ₁	c	d	e ₁
a ₂	b ₂	c	d ₂	e
a	b	c	d ₃	e ₃

studioName \rightarrow president

title	year	studio Name	president	pres Addr
a ₁	b ₁	c	d	e ₁
a ₂	b ₂	c	d	e
a	b	c	d	e ₃

lossless join
decomposition

title	year	studio Name	president	pres Addr
a ₁	b ₁	c	d	e
a ₂	b ₂	c	d	e
a	b	c	d	e

president \rightarrow presAddr

BCNF shortcomings – Example 1

Apply(SSN, cName, date, major)

Can apply to each college once and for one major
Colleges have non-overlapping application dates

FDs: SSN, cName \rightarrow date, major; date \rightarrow cName

Keys: {SSN, cName}

BCNF?

No.

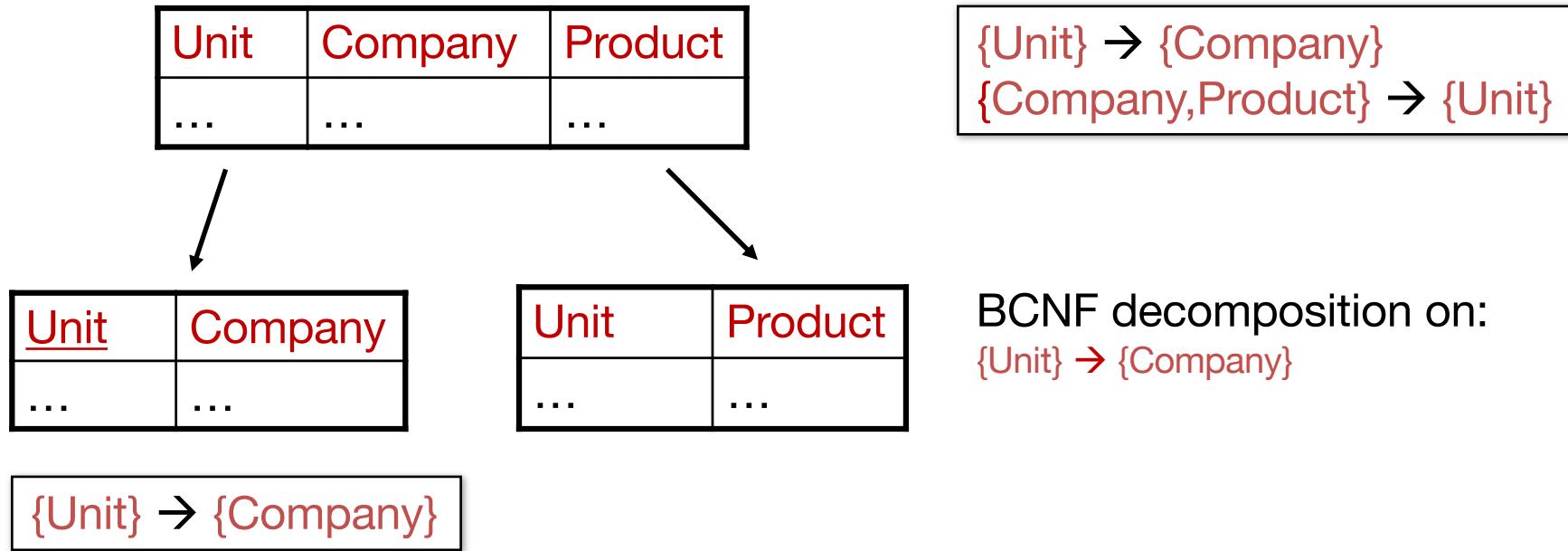
A1 (date, cName)
A2 (date, SSN, major)

Good design? Not necessarily.

Student's application separated from the college
Checking the first DF would require a join
We might just prefer to keep everything together

Apply is in 3NF

BCNF shortcomings – Example 2



We lose the FD $\{Company, Product\} \rightarrow \{Unit\}$

BCNF shortcomings – Example 2

<u>Unit</u>	Company
Galaga99	UW
Bingo	UW

<u>Unit</u>	Product
Galaga99	Databases
Bingo	Databases

$\{Unit\} \rightarrow \{Company\}$

Unit	Company	Product
Galaga99	UW	Databases
Bingo	UW	Databases

No problem so far.
All *local* FDs are satisfied.

Let's put all the data back into a single table again

Violates the FD $\{Company, Product\} \rightarrow \{Unit\}$

BCNF shortcomings – Example 3

College (cName, state)

CollegeSize (cName, enrollment)

CollegeScores (cName, avgSAT)

CollegeGrades (cName, avgGPA)

Too decomposed

We could capture all of the information in one relation or a couple and still be in BCNF

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 additional steps to keep track of lost FDs

3NF Decomposition Algorithm

Input: relation R + set F of FDs for R

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

1. Find a minimal basis for F, say G

Right sides with only 1 attribute

No redundant FDs

For each DF $\bar{X} \rightarrow \bar{A}$, compute \bar{X}^+ using the other DFs. If $A \subseteq \bar{X}^+$, the DF $\bar{X} \rightarrow \bar{A}$ is redundant

No redundant attributes on the left sides

Remove 1 attribute from left side and compute closure of the remaining attributes with the **original** DFs. If closure includes the right side, the attribute can be removed

3NF Decomposition Algorithm

Input: relation R + set F of FDs for R

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

2. For each DF $\bar{X} \rightarrow \bar{A}$ in G , create a relation R' (\bar{X} , \bar{A})
Previously, merge DFs with equal left sides
3. If none of the relations of step 2 is a superkey for R ,
add another relation for a key for R

3NF Decomposition Example

R (A, B, C, D, E); $AB \rightarrow C$, $C \rightarrow B$ and $A \rightarrow D$ Minimal base

1. Find a minimal basis for DFs

Right sides with only 1 attribute?

No redundant DFs?

$\{A, B\}^+ = \{A, B, D\}$ → It does not contain C thus the DF is essential

$\{C\}^+ = \{C\}$ → It does not contain B thus the DF is essential

$\{A\}^+ = \{A\}$ → It does not contain D thus the DF is essential

No redundant attributes on left side?

On $AB \rightarrow C$, remove A, getting $B \rightarrow C$. $\{B\}^+ = \{B\}$. Since it does not contain C, the attribute A is essential

On $AB \rightarrow C$, remove B, getting $A \rightarrow C$. $\{A\}^+ = \{AD\}$. Since it does not contain C, the attribute B is essential

3NF Decomposition Example

$R(A, B, C, D, E)$; $AB \rightarrow C$, $C \rightarrow B$ and $A \rightarrow D$

2. For each DF $\bar{X} \rightarrow \bar{A}$ in G, create a relation $R'(X, A)$

$R_1(A, B, C)$

$R_2(C, B)$

$R_3(A, D)$

3. If none of the relations of step 2 is a superkey for R, add another relation for a key for R

Keys: $\{A, B, E\}$, $\{A, C, E\}$

R_4 should be one of them

Exercise

Consider the following relation and FDs

Movie (title, year, studioName, president, presAddr)

title, year \rightarrow studioName

studioName \rightarrow president

president \rightarrow presAddr

Decompose into 3NF relations.

Any advantages over the BCNF decomposition?

BCNF and 3NF decomposition

BCNF 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

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 3.1 – Functional Dependencies

Section 3.2 – Rules About Functional Dependencies

Section 3.3 – Design of Relational Database Schemas

Section 3.4 – Decomposition: The Good, Bad, and Ugly

Section 3.5 – Third Normal Form

SQL – Data Definition Language

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Agenda

SQL Introduction

Defining a Relation Schema in SQL

Constraints

SQL

Stands for Structured Query Language

Pronounced “sequel”

Supported by all major commercial database systems

Standardized – many features over time

Interactive via GUI or prompt, or embedded in programs

Declarative, based on relational algebra

SQL History

1970 “A Relational Model of Data for Large Shared Data Banks” by Edgar Codd

Early **70's** SEQUEL Developed at IBM by Donald Chamberlin e Raymond Boyce

1979 First commercial version by Relational Software (now Oracle)

1986 SQL-86 and SQL-87. Ratified by ANSI and ISO

1989 SQL-89¹⁹⁹² SQL-92. Also known as SQL2

1999 SQL:1999. Also known as SQL3. Includes regular expressions, recursive queries, triggers, non-scalar data types and some object-oriented expressions

2003 SQL:2003 XML support and auto-generated values

2006 SQL:2006 XQuery support

2008 SQL:2008

2011 SQL:2011

SQL is a ...

Data Definition Language (DDL)

Define relational schemata

Create/alter/delete tables and their attributes

Data Manipulation Language (DML)

Insert/delete/modify tuples in tables

Query one or more tables

Standard

Many standards out there

Database management systems implement something similar, but not identical to the standard for SQL

These slides will try to adhere to the standard as much as possible

Primarily the SQL2 standard and some constructs from the SQL3 standard

Sometimes we'll talk specifically about SQL as understood by SQLite

Agenda

SQL Introduction

Defining a Relation Schema in SQL

Constraints

Relations in SQL

Tables

Kind of relation we deal with ordinarily

Exists in the database and can be modified as well as queried

Views



We'll see this in
another class

Relations defined by computation

Not stored, constructed when needed

Temporary tables

Constructed by the SQL processor during query execution and data modification

Not stored

Data Types in SQL - Text

CHAR(n)

Stores fixed-length string of up to n characters

Normally, shorter strings are padded by trailing blanks to make n characters

VARCHAR(n)

Stores variable-length string of up to n characters

Data Types in SQL – Numeric values

INT (or INTEGER)

For whole numbers

SHORTINT

Denotes whole numbers but the bits permitted may be less
(depends on the implementation)

FLOAT (or REAL)

For floating-point numbers

DECIMAL(n,d)

Values that consist of n decimal digits, with the decimal point assumed to be d positions from the right

Data Types in SQL – Boolean values

Denotes an attribute whose value is logic

Possible values: TRUE, FALSE and UNKNOWN

Data Types in SQL – Dates and Times

DATE

DATE ‘1948-05-14’

TIME

TIME ‘15:00:02.5’

Two and a half seconds past three o'clock

Essentially character strings of a special form

We may coerce dates and times to string types and do the reverse if the string “makes sense” as a data or time

Different implementations may provide different representations

Storage Classes and Data Types in SQLite

NULL

The value is a NULL value.

INTEGER

The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.

REAL

The value is a floating point value, stored as an 8-byte IEEE floating point number.

TEXT

The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).

BLOB

The value is a binary large object of data, stored exactly as it was input.

Boolean Data Type in SQLite

SQLite does not have a separate Boolean storage class

Boolean values are stored as integers

0 (false) and 1 (true)

Dates and Times in SQLite

SQLite does not have a storage class for storing dates and/or times

Dates and times can be stored as:

- TEXT as ISO8601 strings ("YYYY-MM-DD HH:MM:SS.SSS").

- REAL as Julian day numbers, the number of days since noon in Greenwich on November 24, 4714 B.C.

- INTEGER as Unix Time, the number of seconds since 1970-01-01 00:00:00 UTC.

SQLite has built-in date and time functions to convert between formats

Type Affinities in SQLite

To maximize compatibility between SQLite and other database engines, SQLite supports the concept of "type affinity"

The type affinity of a column is the recommended type for data stored in that column

Each column has one of the following type affinities:

TEXT, NUMERIC, INTEGER, REAL, BLOB

Determination of Type Affinities in SQLite

If the declared type contains

“INT”: INTEGER affinity

“CHAR”, “CLOB”, “TEXT”: TEXT affinity

“BLOB” or no type specified: BLOB affinity

“REAL”, “FLOA”, “DOUB”: REAL affinity

Otherwise: NUMERIC affinity

Rules should be assessed by the above order.

What is the affinity of a CHARINT declared type?

Integer

Type Affinities in SQLite

Text affinity

Storage classes: NULL, TEXT or BLOB

Numerical data is converted into TEXT when inserted into a column with text affinity

Numeric affinity

All storage classes

Text data is converted into INTEGER or REAL if conversion is lossless and to TEXT otherwise

Integer affinity

Similar to the numeric affinity

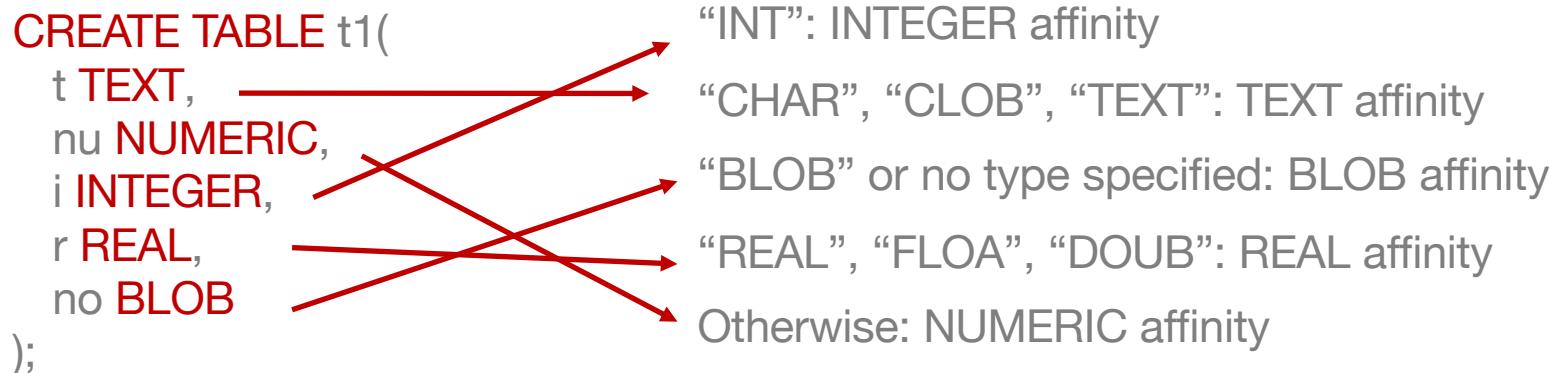
Real affinity

Similar to numeric affinity but forcing integer values into floating point representation

BLOB affinity

No attempt to coerce data from one storage class into another

Determination of Type Affinities in SQLite



`INSERT INTO t1 VALUES('500.0', '500.0', '500.0', '500.0', '500.0');`
text | integer | integer | real | text

`INSERT INTO t1 VALUES(500.0, 500.0, 500.0, 500.0, 500.0);`
text | integer | integer | real | real

`INSERT INTO t1 VALUES(500, 500, 500, 500, 500);`
text | integer | integer | real | integer

BLOBS are always stored as BLOBS regardless of column affinity

NULLs are also unaffected by affinity

Simple Table Declarations

```
CREATE TABLE <table_name> (
    <column_name> <data_type>,
    <column_name> <data_type>,
    ...
    <column_name> <data_type>
);
```

Example

```
CREATE TABLE MovieStar (
    id          INTEGER,
    name        CHAR(30),
    address     VARCHAR(255),
    gender      CHAR(1),
    birthdate   DATE
);
```

Modifying Relation Schemas

To remove an entire table and all its tuples

DROP TABLE <table_name>;

To modify the schema of an existing relation

ALTER TABLE <table_name> **ADD** <column_name> <data_type>;

ALTER TABLE <table_name> **DROP** <column_name>;

Default values

For each column we can define its default value

```
CREATE TABLE <table_name> (  
    <column_name> <data_type> DEFAULT <default_value>,  
    ...  
    <column_name> <data_type>  
);
```

The default default value is NULL

Example

```
CREATE TABLE MovieStar (
    id          INTEGER,
    name        CHAR(30),
    address     VARCHAR(255),
    gender      CHAR(1) DEFAULT '?',
    birthdate   DATE
);
```

```
ALTER TABLE MovieStar ADD phone CHAR(16) DEFAULT
'unlisted';
```

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 2.3 – Defining a Relation Schema in SQL

Section 2.5 – Constraints on Relations

Section 7.1 – Keys and Foreign Keys

Section 7.2 – Constraints on Attributes and Tuples

Section 7.3 – Modification if Constraints

Section 7.4 - Assertions

SQL – Data Definition Language

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Agenda

~~SQL Introduction~~

~~Defining a Relation Schema in SQL~~

Constraints

(Integrity) Constraints

Part of the SQL standard but systems vary considerable

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

Examples

$0.0 < GPA \leq 4.0$

$\text{enrolment} < 50,000$

Decision: ‘Y’, ‘N’ or NULL

$\text{Major} = 'CS' \Rightarrow \text{decision} = \text{NULL}$

$\text{sizeHS} < 200 \Rightarrow \text{not admitted in colleges with enroll} > 30,000$

Why use constraints?

Data-entry errors (inserts)

Correctness criteria (updates)

Enforce consistency

Tell system about data – store, query processing

Classification of constraints

Non-null constraints

Key constraints

Attribute-based and tuple-based constraints

Referential integrity (foreign key)

General assertions

Declaring and enforcing constraints

Declaration

With original schema

Checked after bulk loading

Or later

Checked at the time it's declared on the current state of the DB

Enforcement

Check after every modification

Check only the dangerous ones

If there is a constraint on one table, there is no need to check updates on another tables

Deferred constraint checking

Instead of checking after every modification, checking is done after every transaction

Non-null constraint

Defines that a column does not have NULL values

```
CREATE TABLE <table_name> (
    <column_name> <data_type> NOT NULL,
    ...
    <column_name> <data_type>
);
```

Example

```
CREATE TABLE MovieStar (
    id          INTEGER,
    name        CHAR(30) NOT NULL,
    address     VARCHAR(255),
    gender      CHAR(1) DEFAULT '?',
    birthdate   DATE
);
```

Primary key (PK) constraint

We can define one, and only one, primary key for a table

```
CREATE TABLE <table_name> (
    <column_name> <data_type> PRIMARY KEY,
    ...
    <column_name> <data_type>
);
```

A PK cannot be NULL and cannot have repeated values

In SQLite, INTEGER PK will auto increment if a NULL value is inserted in the PK column

Example

```
CREATE TABLE MovieStar (
    id      INTEGER PRIMARY KEY,
    name    CHAR(30),
    address VARCHAR(255),
    gender   CHAR(1) DEFAULT '?',
    birthdate DATE
);
```

Multiple Column PK

If a primary key is composed by more than one column

```
CREATE TABLE <table_name> (
    <column_name> <data_type>,
    ...
    <column_name> <data_type>,
PRIMARY KEY (<column_name>, <column_name>)
);
```

Example

```
CREATE TABLE MovieStar (
    name      CHAR(30),
    address   VARCHAR(255),
    gender    CHAR(1) DEFAULT '?',
    birthdate DATE,
    PRIMARY KEY (name, birthdate)
);
```

ROWID in SQLite

If a table is created without specifying the WITHOUT ROWID option, SQLite adds an implicit column called rowid that stores 64-bit signed integer

The rowid is a key that uniquely identifies the row in its table

Can be accessed using ROWID, _ROWID_, or OID (except if ordinary columns use these names)

On an INSERT, if the ROWID is not explicitly given a value, then it will be filled automatically with an unused integer greater than 0, usually one more than the largest ROWID currently in use.

SQLite PK and ROWID

If a table has a one-column PK defined as INTEGER,
this PK column becomes an alias for the rowid column

Tables with rowid are stored as a B-Tree using rowid as
the key

Retrieving and sorting by rowid are very fast

Faster than using any other PK or indexed value

SQLite Autoincrement

Imposes extra CPU, memory, disk space, and disk I/O overhead

If an AUTOINCREMENT keyword appears after INTEGER PRIMARY KEY, that changes the ROWID assignment algorithm to prevent the reuse of ROWIDs

The purpose of AUTOINCREMENT is to prevent the reuse of ROWIDs from previously deleted rows

Should be avoided if not strictly needed

Unique key constraint

We can define multiple unique keys for a table

```
CREATE TABLE <table_name> (
    <column_name> <data_type> UNIQUE,
    ...
    <column_name> <data_type>
);
```

Unique Keys allow NULL values

The SQL standard and most DBMS do allow repeated NULL values in UNIQUE keys

Example

```
CREATE TABLE MovieStar (
    id      INTEGER PRIMARY KEY,
    name    CHAR(30),
    address VARCHAR(255) UNIQUE,
    gender   CHAR(1) DEFAULT '?',
    birthdate DATE,
    phone   CHAR(16) UNIQUE
);
```

Address cannot have repeated values

Phone cannot have repeated values

Multiple Column Unique Key

If a unique key is composed by more than one column

```
CREATE TABLE <table_name> (
    <column_name> <data_type>,
    ...
    <column_name> <data_type>,
    UNIQUE (<column_name>, <column_name>)
);
```

Example

```
CREATE TABLE MovieStar (
    id      INTEGER PRIMARY KEY,
    name    CHAR(30),
    address VARCHAR(255),
    gender   CHAR(1) DEFAULT '?',
    birthdate DATE,
    UNIQUE (name, birthdate),
    UNIQUE (name, address)
);
```

Attribute-based check constraint

Constrain the value of a particular attribute

```
CREATE TABLE <table_name> (  
    <column_name> <data_type> CHECK <check_expression>,  
    ...  
    <column_name> <data_type>  
);
```

Checked whenever we insert or update a tuple

Example

```
CREATE TABLE Student (
    sID      INTEGER PRIMARY KEY,
    sName    TEXT,
    GPA      REAL CHECK (GPA<=4.0 and GPA>0.0),
    sizeHS   INTEGER CHECK (sizeHS < 5000),
);
```

GPs must be less than or equal to 4.0 and greater than zero

The size of the high school must be less than five thousand

Tuple-based check constraint

Constrain relationships between different values in each tuple

```
CREATE TABLE <table_name> (
    <column_name> <data_type>,
    ...
    <column_name> <data_type>,
    CHECK (<check_expression>)
);
```

Checked whenever we insert or update a tuple

Example

```
CREATE TABLE Apply (
    sID      INTEGER,
    cName    TEXT,
    major    TEXT,
    decision TEXT,
    PRIMARY KEY (sID, cName, major),
    CHECK (decision='N' or cName <>'Stanford' or major <>'CS')
);
```

Either the decision is null or the college name is not Stanford or the major is not CS



There are no people who have applied to Stanford and been admitted to CS at Stanford

Example

```
CREATE TABLE person (
    id      INTEGER PRIMARY KEY,
    name    TEXT,
    salary  TEXT,
    taxes   TEXT,
    CHECK(taxes<salary)
);
```

Taxes have to be lower than salary

Subqueries and check constraints

```
CREATE TABLE Student (sID INTEGER PRIMARY KEY, sName  
TEXT, GPA REAL, sizeHS INTEGER);
```

```
CREATE TABLE Apply (  
    sID      INTEGER,  
    cName    TEXT,  
    major    TEXT,  
    decision TEXT,  
    PRIMARY KEY (sID, cName, major),  
    CHECK (sID in (select sID from Student))  
);
```

Syntactically valid in the SQL standard but no SQL systems supports subqueries and check constraints.

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 2.3 – Defining a Relation Schema in SQL

Section 2.5 – Constraints on Relations

Section 7.1 – Keys and Foreign Keys

Section 7.2 – Constraints on Attributes and Tuples

Section 7.3 – Modification if Constraints

Section 7.4 - Assertions

SQL – Data Definition Language

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Referential Integrity

Integrity of references

No “dangling pointers”

Referential integrity from R.A to S.B

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

Student

sID	sName	GPA	HS
123	Mary	3.5	Palo Alto

Apply

sID	cName	major	dec
123	Stanford	CS	Y

College

cName	state	enr
Stanford	California	10000

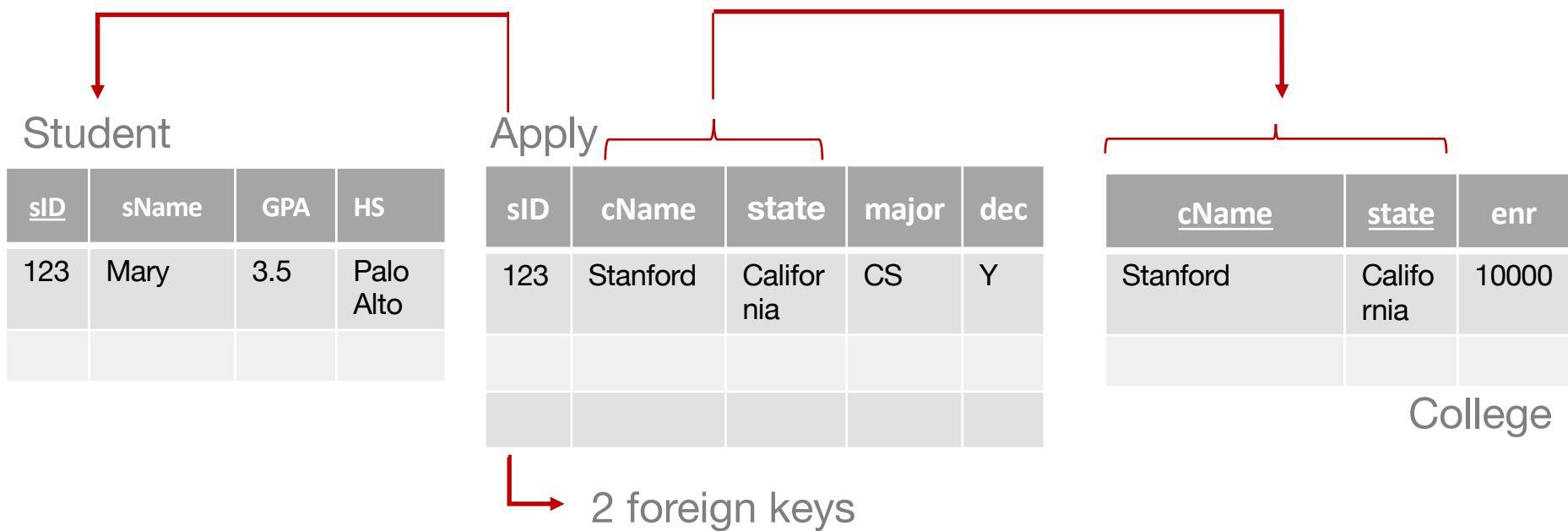
Referential Integrity

Referential integrity from R.A to S.B

A is called the “foreign key”

B is usually required to be the primary key for table S or at least unique

Multi-attribute foreign keys are allowed



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

Potentially violating modifications

Insert into R

Delete from S

Update R.A

Update S.B

If violation -> error

Depends on Foreign key definition

Student			
sID	sName	GPA	HS
123	Mary	3.5	Palo Alto

Apply			
sID	cName	major	dec
123	Stanford	CS	Y

College		
cName	state	enr
Stanford	Califor	10000
	nia	

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

Delete from S

Restrict (default)

Generate an error, modification disallowed

Set Null

Replace R.A by NULL

Cascade

Delete tuples having a referencing value

Student

sID	sName	GPA	HS
123	Mary	3.5	Palo Alto
234	Louis	3.8	Palo Alto

Apply

sID	cName	major	dec
123	Stanford	CS	Y
234	MIT	CS	Y

College

cName	state	enr
Stanford	California	10000
MIT	Massachusetts	15000

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

Update S.B

Restrict (default)

Generate an error, modification disallowed

Set Null

Replace R.A by NULL

Cascade

Do the same update to R.A

Student

sID	sName	GPA	HS
123	Mary	3.5	Palo Alto
456			

456

Apply

sID	cName	major	dec
NULL	Stanford	CS	Y
234	Standford MIT	CS	Y

College

cName	state	enr
Stanford	California	10000
Standford		

Foreign Key Declaration

```
CREATE TABLE <table_A> (
    <column_A> <data_type> PRIMARY KEY,
    <column_B> <data_type>,
    ...
    <column_C> <data_type>
);
```

```
CREATE TABLE <table_B> (
    <column_X> <data_type> PRIMARY KEY,
    <column_Y> <data_type>,
    ...
    <column_Z> <data_type> REFERENCES <table_A>(<column_A>)
);
```

Example

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 **REFERENCES** Student(sID),
 cName text **REFERENCES** College (cName),
 major text,
 decision text,
 PRIMARY KEY(sID, cName)
);

Foreign Key to Primary Key

If the referenced column is the primary key of the other table, we can omit the name of the column

```
CREATE TABLE <table_A> (
    <column_A> <data_type> PRIMARY KEY,
    <column_B> <data_type>,
    ...
    <column_C> <data_type>
);
```

```
CREATE TABLE <table_B> (
    <column_X> <data_type> PRIMARY KEY,
    <column_Y> <data_type>,
    ...
    <column_Z> <data_type> REFERENCES <table_A>
);
```

Example

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,
 cName text **REFERENCES** College,
 major text,
 decision text,
 PRIMARY KEY(sID, cName)
);

Multiple Column Foreign Key Declaration

```
CREATE TABLE <table_A> (
    <column_A> <data_type>,
    <column_B> <data_type>,
    ...
    <column_C> <data_type>,
    PRIMARY KEY (<column_A>, <column_B>)
);
```

```
CREATE TABLE <table_B> (
    <column_X> <data_type> PRIMARY KEY,
    <column_Y> <data_type>,
    ...
    <column_Z> <data_type>,
    FOREIGN KEY (<column_X>, <column_Y>) REFERENCES <table_A>(<column_A>, <column_B>)
);
```

Example

```
CREATE TABLE College (cName text, state text, enrollment int, PRIMARY KEY (cName, state));
```

```
CREATE TABLE Student (sID int PRIMARY KEY, sName text, GPA real, sizeHS int);
```

```
CREATE TABLE Apply (
    sID REFERENCES Student,
    collegeName text,
    collegeState text,
    major text,
    decision text,
    FOREIGN KEY (collegeName, collegeState) REFERENCES College(cName, state),
    PRIMARY KEY(sID, collegeName, collegeState)
);
```

We can omit the referenced columns if they are primary keys

On Delete and On Update Actions

Define actions that take place when deleting or modifying parent key values.

Use the ON DELETE and ON UPDATE clauses with one of three possible values

RESTRICT - prohibit operation on a parent key when there are child keys mapped to it

SET NULL – child key columns are set to NULL

CASCADE – propagates the operation on the parent key to each dependent child key

Example

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 **REFERENCES** Student(ID),

cName text **REFERENCES** College (cName) **ON DELETE SET NULL ON UPDATE CASCADE**,

major text,

decision text,

PRIMARY KEY(sID, cName)

);

Enabling Foreign Key Support in SQLite

Foreign key constraints are disabled by default

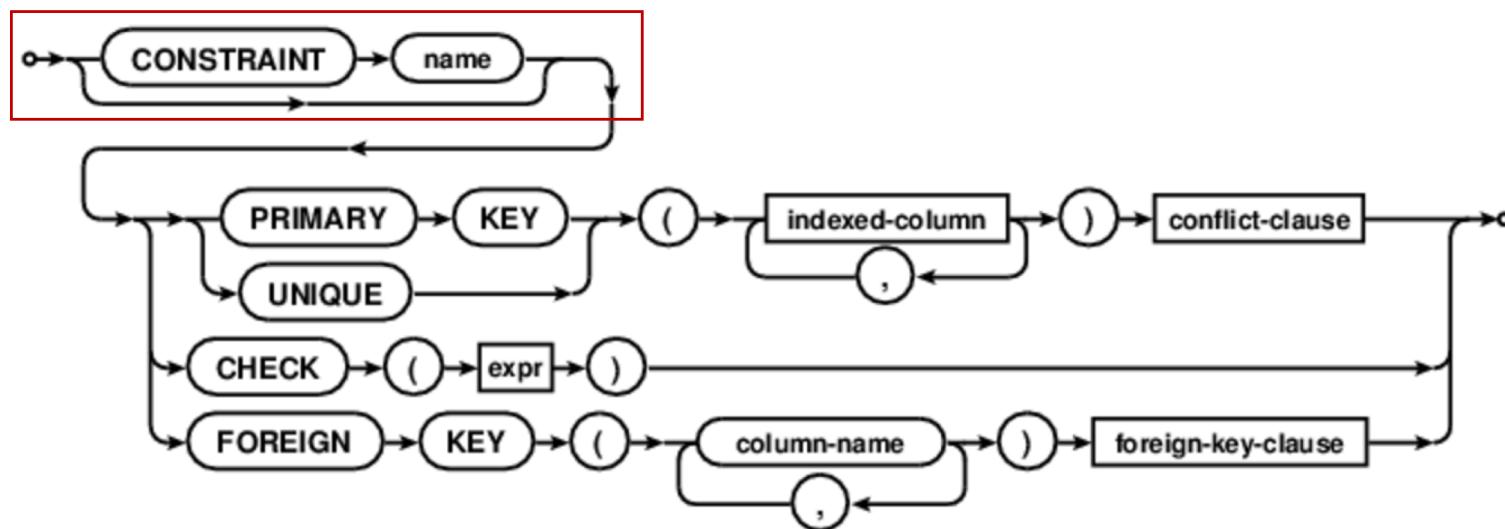
Must be enabled separately for each database connection

`PRAGMA foreign_keys = ON;`

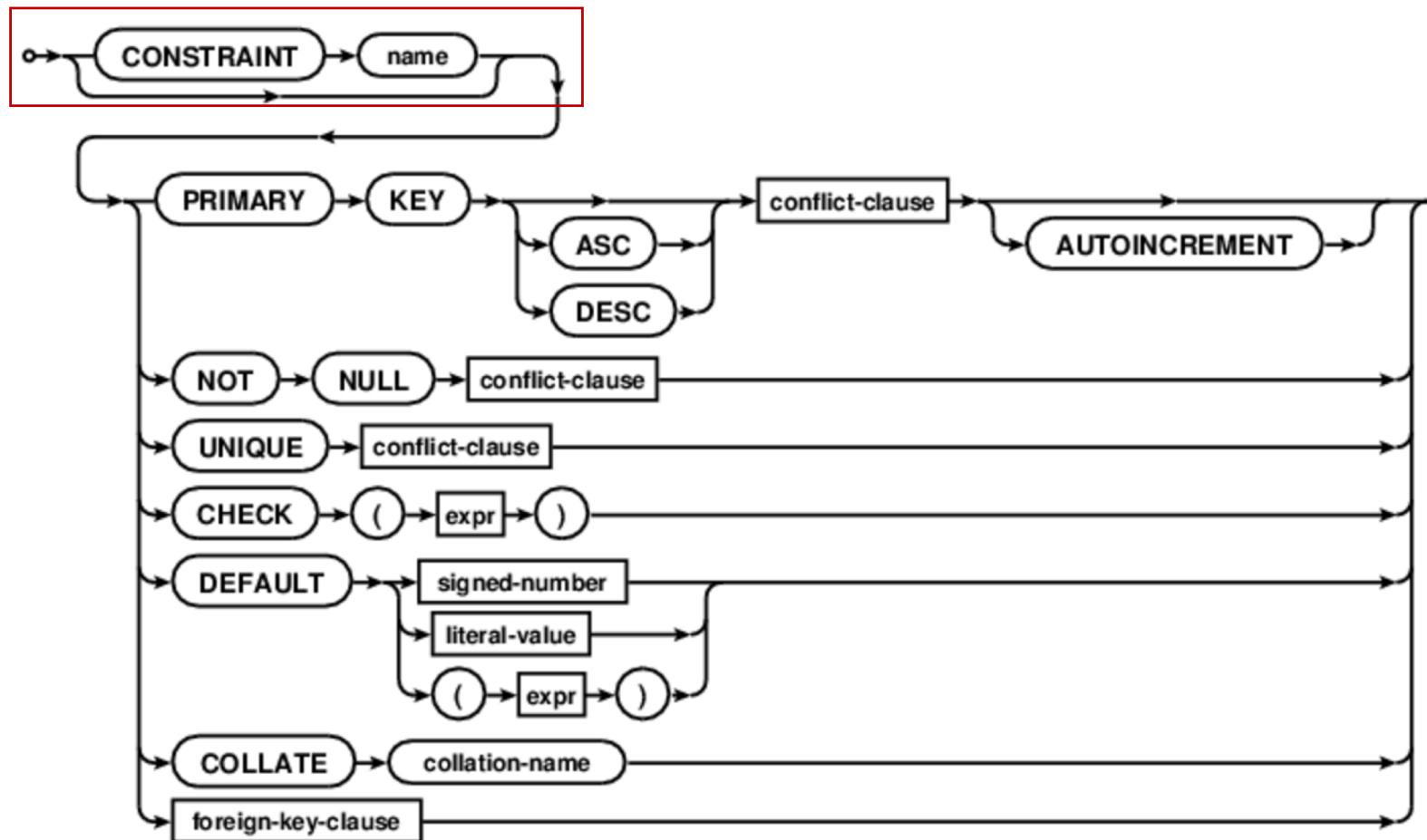
Constraint Naming

Naming constraints is optional but is a good practice

It makes it easier to identify the constraints when errors occur and to refer to them



Constraint Naming



Example

```
CREATE TABLE Student (
    sID      INTEGER,
    sName    TEXT,
    GPA      REAL CONSTRAINT GPARange CHECK (GPA<=4.0),
    sizeHS   INTEGER CONSTRAINT maxSizeHS CHECK (sizeHS < 5000),
    CONSTRAINT StudentPK PRIMARY KEY (sID)
);
```

Assertions

Constraints on entire relation or entire database

Are in the SQL standard but are not supported by any database system

CREATE ASSERTION <assertion_name> **CHECK** (<condition>);

Example

CREATE ASSERTION Key **CHECK**(
(select count(distinct A) from T) = (select count(*) from T));

CREATE ASSERTION ReferentialIntegrity **CHECK**(
not exists (**SELECT * from Apply**
where sID not in (select sID from Student)));

CREATE ASSERTION AvgAccept **CHECK**(
3.0 < **(select avg(GPA) from Student**
where sID in
(select sID from Apply where decision = 'Y')));

Assertion checking

```
CREATE ASSERTION AvgAccept CHECK(  
    3.0 < (select avg(GPA) from Student  
        where sID in  
            (select sID from Apply where decision = 'Y')));
```

Determine every possible change that could violate the assertion

What changes are these in the above assertion?

Modifying a GPA, student ID and decision

Inserting or deleting from students or apply

After those modifications

check the constraint,

make sure they it's still satisfied and, if not, generate an error and disallow the database change

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 2.3 – Defining a Relation Schema in SQL

Section 2.5 – Constraints on Relations

Section 7.1 – Keys and Foreign Keys

Section 7.2 – Constraints on Attributes and Tuples

Section 7.3 – Modification if Constraints

Section 7.4 - Assertions

Relational Algebra

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Agenda

Introduction to Relational Algebra

Operators

Alternate notations

Extensions to Relational Algebra

What is Algebra?

Mathematical system consisting of:

Operands - variables or values from which new values can be constructed.

Operators - symbols denoting procedures that construct new values from given values.

What is Relational Algebra?

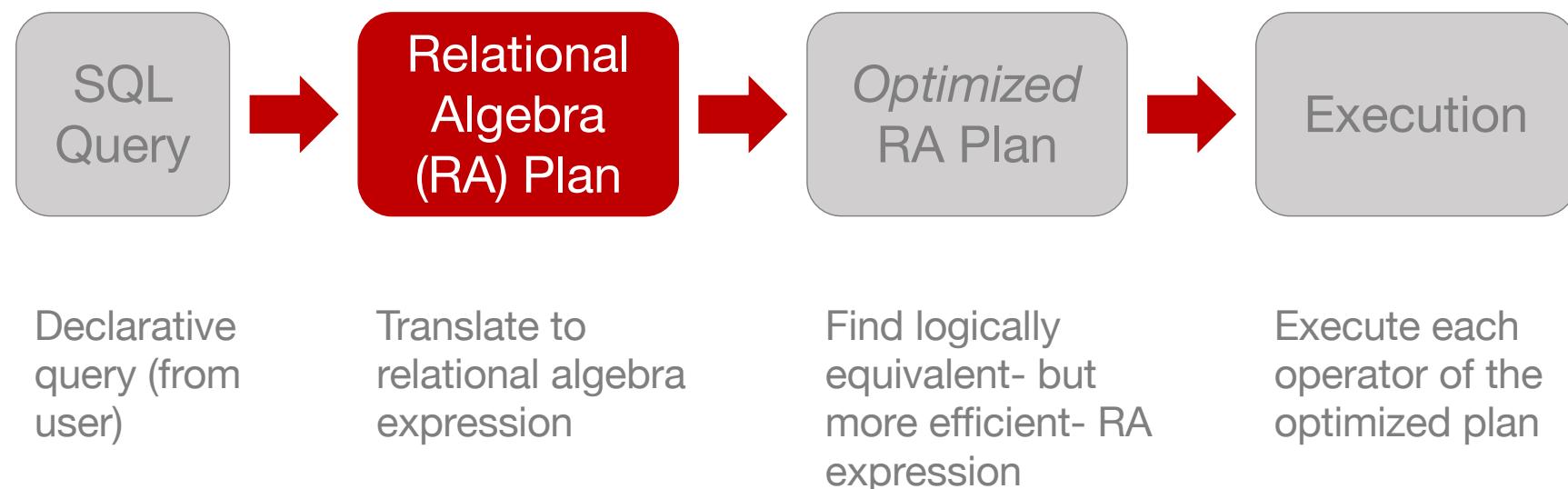
An algebra whose operands are relations or variables that represent relations.

Operators are designed to do the most common things that we need to do with relations in a database.

The result is an algebra that can be used as a query language for relations.

RDBMS Architecture

How does a SQL engine work?

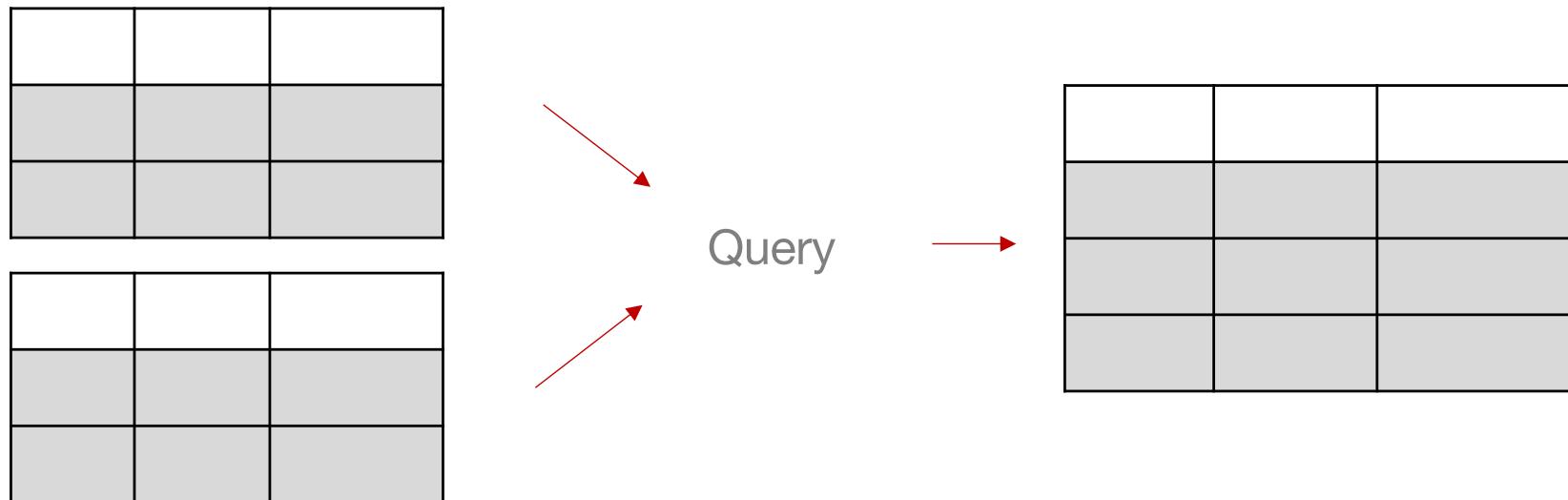


Relational Algebra

Formal language

Operates on relations and produce relations as a result

Operators are used to filter, slice and combine



Agenda

Introduction to Relational Algebra

Operators

Alternate notations

Extensions to Relational Algebra

College Admission Database

College (cName, state, enr)

Student (sID, sName, GPA, HS)

Apply (sID, cName, major, dec)

Demo in Relax: <https://dbis-uibk.github.io/relax/>

College

<u>cName</u>	state	enr

Student

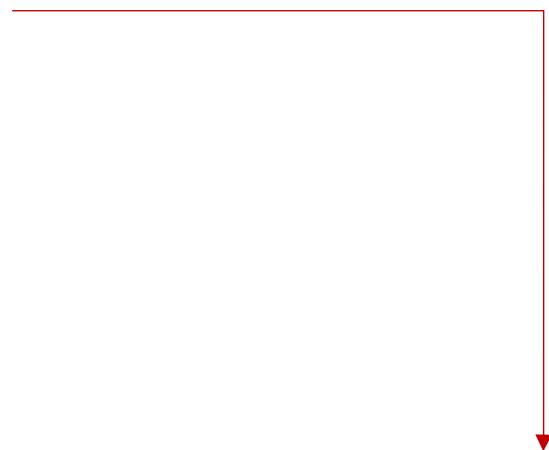
<u>sID</u>	sName	GPA	HS

Apply

<u>sID</u>	<u>cName</u>	<u>major</u>	dec

Simplest query: relation name

Student



Student

sID	sName	GPA	HS

Select operator (σ)

Returns all tuples which satisfy a condition

Notation: $\sigma_{condition} Relation$

The condition can involve $=, <, \leq, >, \geq, \neq$

Examples

Students with $\text{GPA} > 3.7$

$\sigma_{\text{GPA} > 3.7} \text{ Student}$

Students with $\text{GPA} > 3.7$ and $\text{HS} < 1000$

$\sigma_{\text{GPA} > 3.7 \wedge \text{HS} < 1000} \text{ Student}$

Applications to Stanford CS major

$\sigma_{\text{cName} = 'Stanford' \wedge \text{major} = 'CS'} \text{ Apply}$

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	John	3.8	500
31	Jane	3.9	1000

Apply

sID	cName	major	dec
12	Stanford	CS	Y
23	MIT	CS	N
12	MIT	CS	N

Project operator (π)

Picks certain columns

Notation: $\pi_{A_1, \dots, A_n} \text{ Relation}$

sID and decision of all applications

$\pi_{sID, dec} \text{ Apply}$

Apply

sID	cName	major	dec
12	Stanford	CS	Y
23	MIT	CS	N
12	MIT	CS	N



sID	dec
12	Y
23	N
12	N

Combining the Select and Project Operators

ID and name of students with GPA>3.7

$$\pi_{sID, sName} (\sigma_{GPA > 3.7} Student)$$

Redefinition of operators

$$\sigma_{condition} (Expression)$$

$$\pi_{A_1, \dots, A_n} (Expression)$$

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	John	3.8	50
31	Jane	3.9	1000

Sets, Bags and Lists

Sets

Only one occurrence of each element

Unordered elements

Bags (or multisets)

More than one occurrence of an element

Unordered elements and their occurrences

Lists

More than one occurrence of an element

Occurrences are ordered

Duplicates

Relational Algebra

Eliminates duplicates

Based on sets (although there is also a multiset relation algebra)

SQL

Does not eliminate duplicates

Based on multisets or bags

List of application majors and decisions

$\pi_{major,dec}$ Apply

Apply

sID	cName	major	dec
12	Stanford	CS	Y
23	MIT	CS	N
12	MIT	CS	N

$\pi_{major,dec}$ Apply

major	dec
CS	Y
CS	N

No duplicates



Cross-product

Also known as Cartesian product

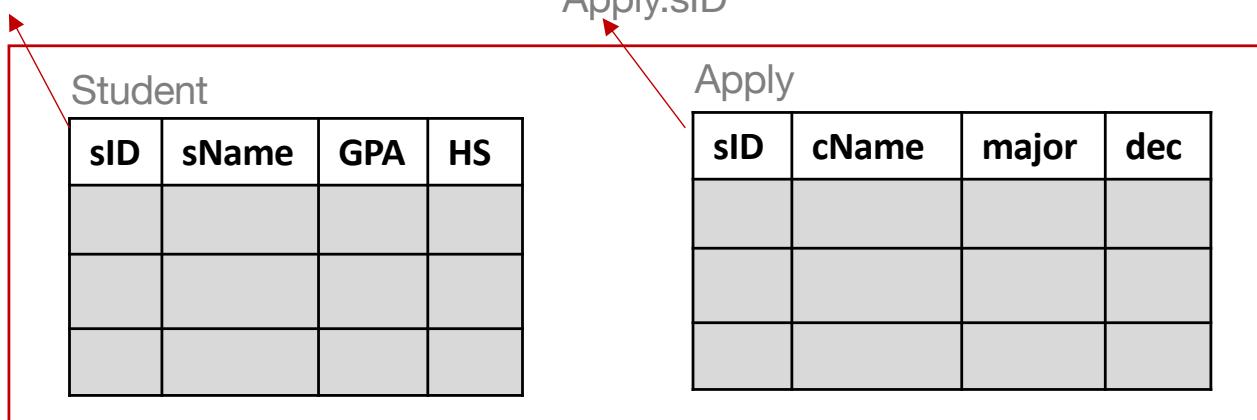
Notation: Rel1 x Rel2

Student x Apply

Attributes with the same name are prefaced with the name of the relation

Student.sID

Apply.sID



Cross-product

One tuple for every combination of tuples from the student and apply relations

Student

sID	sName	GPA	HS

S tuples

Student x Apply



$S \times A$ tuples

Apply

sID	cName	major	dec

A tuples

Example 1

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	John	3.8	50

Apply

sID	cName	major	dec
12	Stanford	CS	Y
23	MIT	CS	N



Student x Apply

Student.sID	sName	GPA	HS	Apply.sID	cName	major	dec
12	Mary	3.5	90	12	Stanford	CS	Y
12	Mary	3.5	90	23	MIT	CS	N
23	John	3.8	50	12	Stanford	CS	Y
23	John	3.8	50	23	MIT	CS	N

Example 2

Names and GPAs of students with HS>100 who applied to CS and were rejected

Student x Apply

All combinations

$\sigma_{Student.sID=Apply.sID}(Student \times Apply)$

Combinations that make sense

$\sigma_{Student.sID=Apply.sID \wedge HS>100 \wedge major='CS' \wedge dec='N'}(Student \times Apply)$

Additional filtering

$\pi_{sName, GPA}(\sigma_{Student.sID=Apply.sID \wedge HS>100 \wedge major='CS' \wedge dec='N'}(Student \times Apply))$

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	John	3.8	5000

Apply

sID	cName	major	dec
12	Stanford	CS	Y
23	MIT	CS	N

Natural Join

Operator: \bowtie

Cross product enforcing equality on all attributes with same name

Eliminate one copy of duplicate attributes

College

cName	state	enr

Student

sID	sName	GPA	HS

Apply

sID	cName	major	dec

Example 1

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	John	3.8	50

Apply

sID	cName	major	dec
12	Stanford	CS	Y
23	MIT	CS	N



Student ⚡ Apply

sID	sName	GPA	HS	cName	major	dec
12	Mary	3.5	90	Stanford	CS	Y
23	John	3.8	50	MIT	CS	N

Example 2

Names and GPAs of students with HS>100 who applied to CS and were rejected

Student \bowtie *Apply*

$\sigma_{HS>100 \wedge major='CS' \wedge dec='N'}(Student \bowtie Apply)$

$\pi_{sName, GPA}(\sigma_{HS>100 \wedge major='CS' \wedge dec='N'}(Student \bowtie Apply))$

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	John	3.8	5000

Apply

sID	cName	major	dec
12	Stanford	CS	Y
23	MIT	CS	N

Example 3

Names and GPAs of students with HS>100 who applied to CS at college with enr>10,000 and were rejected

Student \bowtie (Apply \bowtie College)

$\sigma_{HS>100 \wedge major='CS' \wedge dec='N' \wedge enr>10,000} (Student \bowtie (Apply \bowtie College))$

$\pi_{sName, GPA} (\sigma_{HS>100 \wedge major='CS' \wedge dec='N' \wedge enr>10,000} (Student \bowtie (Apply \bowtie College)))$

College

cName	state	enr
MIT	NULL	30000
Stanford	NULL	20000

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	John	3.8	5000

Apply

sID	cName	major	dec
12	Stanford	CS	Y
23	MIT	CS	N

Natural Join

Given $R(A, B, C, D)$, $S(A, C, E)$, what is the schema of $R \bowtie S$?

Given $R(A, B, C)$, $S(D, E)$, what is $R \bowtie S$?

Given $R(A, B)$, $S(A, B)$, what is $R \bowtie S$?

Natural Join does not add expressive power

Can be rewritten using the cross-product

$$Exp1 \bowtie Exp2 \equiv \pi_{schema(E1) \cup schema(E2)}(\sigma_{E1.A1=E2.A1 \wedge E1.A2=E2.A2 \wedge \dots}(Exp1 \times Exp2))$$

It is convenient in terms of notation

Theta Join

A join that involves a predicate

Notation: \bowtie_θ

$$Exp_1 \bowtie_\theta Exp_2 \equiv \sigma_\theta(Exp_1 \times Exp_2)$$

Basic operation implemented in DBMS

Term “join” often means theta join

θ can be any condition

If θ is an equality, the join is called an equi-join

Example

Student

ID	sName	GPA	HS
12	Mary	3.5	90
23	John	3.8	50

Apply

sID	cName	major	dec
12	Stanford	CS	Y
23	MIT	CS	N



Student $\bowtie_{ID=sID}$ Apply

ID	sName	GPA	HS	sID	cName	major	dec
12	Mary	3.5	90	12	Stanford	CS	Y
23	John	3.8	50	23	MIT	CS	N

Semijoin

Notation: \bowtie

$$Exp_1 \bowtie Exp_2 \equiv \pi_{A_1, \dots, A_n} (Exp_1 \bowtie Exp_2)$$

Where A_1, \dots, A_n are attributes in Exp_1

Returns the tuples of Exp_1 with a pair in Exp_2

Example

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	John	3.8	50
35	Jane	3.9	60

Apply

sID	cName	major	dec
12	Stanford	CS	Y
23	MIT	CS	N

Student \bowtie Apply



sID	sName	GPA	HS
12	Mary	3.5	90
23	John	3.8	50

Union operator

Operator: U

List of college and student names

Can we do it using previous operators?

$\pi_{cName} College \cup \pi_{sName} Student$

Combines information vertically

College

cName	state	enr
MIT	NULL	NULL
Washington	NULL	NULL

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	Washington	3.8	50

Technically, the two operands have to have the same schema

Not the case in the example above, but we'll correct it later

Example

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	Washington	3.8	50

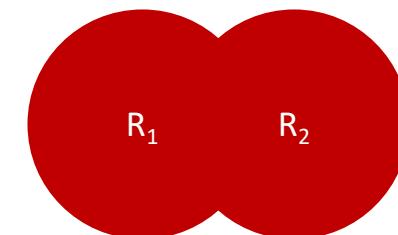
College

cName	state	enr
MIT	NULL	NULL
Washington	NULL	NULL



$\pi_{cName} College \cup \pi_{sName} Student$

cName
Mary
Washington
MIT



Difference operator

Operator: –

IDs of students who didn't apply anywhere

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	Washington	3.8	50

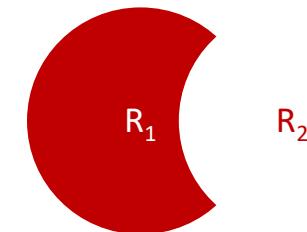
Apply

sID	cName	major	dec
12	Stanford	CS	Y



$$\pi_{sID} \text{Student} - \pi_{sID} \text{Apply}$$

sID
23



Example

Names of students who didn't apply anywhere

$\pi_{sName} Student - \pi_{sID} Apply ?$

$\pi_{sName} ((\pi_{sID} Student - \pi_{sID} Apply) \bowtie Student)$

Schema equal to the student relation

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	Washington	3.8	50

Apply

$\pi_{sID} Student - \pi_{sID} Apply$



sID
23

Intersection operator

Operator: \cap

Names that are both a college name and a student name

$$\pi_{cName} College \cap \pi_{sName} Student$$

Technically, the two operands have to have the same schema
Not the case in the example above, but we'll correct it later

College

cName	state	enr
MIT	NULL	NULL
Washington	NULL	NULL

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	Washington	3.8	50

Example

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	Washington	3.8	50

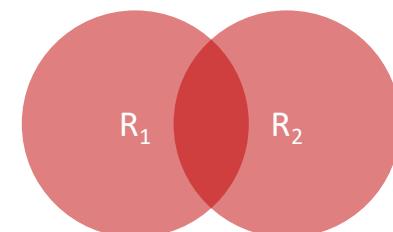
College

cName	state	enr
MIT	NULL	NULL
Washington	NULL	NULL



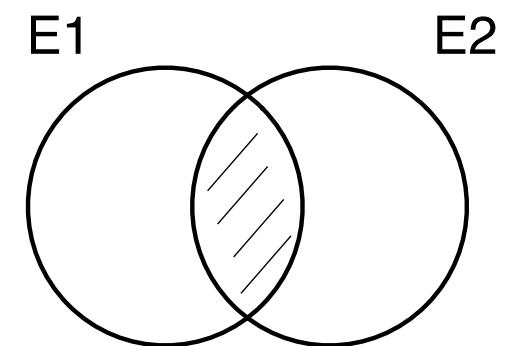
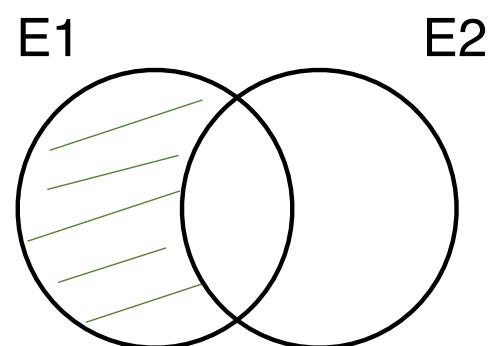
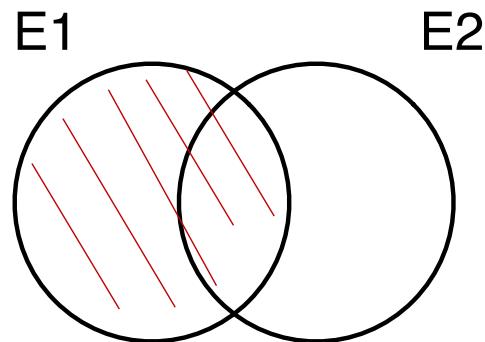
$\pi_{cName} College \cap \pi_{sName} Student$

cName
Washington



Intersection doesn't add expressive power

$$E_1 \cap E_2 \equiv \textcolor{red}{E_1} - (\textcolor{green}{E_1} - E_2)$$



Intersection doesn't add expressive power

$$E_1 \cap E_2 \equiv E_1 \bowtie E_2$$



Identical schema

Nevertheless, the intersection can be very useful in queries

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 2.4 – An Algebraic Query Language

Section 5.2 – Extended Operators of Relational Algebra

Relational Algebra

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Division operator

Operator: /

Identifies the attribute values from a relation that are paired with **all** of the values from another relation

The division is to the Cartesian product (\times) what the division is to multiplication in arithmetic

Necessary to answer queries with “all”

Cross-product and division

Division is the opposite of the cross-product

R1

A
4
8

R2

B
3
1
7

$$R3 = R1 \times R2$$


R3

A	B
4	3
4	1
4	7
8	3
8	1
8	7

$$R3 / R2 = R1$$

$$R3 / R1 = R2$$

How to divide

R	A	B	C	D
a	b	c	d	
a	b	e	f	
b	c	e	f	
e	d	c	d	
e	d	e	f	
a	b	d	e	

/

S	C	D
c		
d		

= ?

Reorder the columns in R so the last ones are the ones in S

Order tuples in R by the first columns

Each R sub-tuple is part of the result if the sub-tuple of the last columns contains the divisor

How to divide

Reorder the columns in R so the last ones are the ones in S

$$\begin{matrix} R \\ \begin{array}{|c|c|c|c|} \hline A & B & C & D \\ \hline a & b & c & d \\ \hline a & b & e & f \\ \hline b & c & e & f \\ \hline e & d & c & d \\ \hline e & d & e & f \\ \hline a & b & d & e \\ \hline \end{array} \end{matrix} / \begin{matrix} S \\ \begin{array}{|c|c|} \hline C & D \\ \hline c & d \\ \hline e & f \\ \hline \end{array} \end{matrix} = ?$$

How to divide

Order R by the **first columns**

R	A	B	C	D
a	b	c	d	
a	b	e	f	
b	c	e	f	
e	d	c	d	
e	d	e	f	
a	b	d	e	

/

S	C	D
c		
c	d	
e	f	

= ?

→

R	A	B	C	D
a	b	c	d	
a	b	e	f	
a	b	d	e	
b	c	e	f	
e	d	c	d	
e	d	e	f	

How to divide

Each R sub-tuple is part of the result if the sub-tuple of the last columns contains the divisor

R	S	R/S																																		
<table border="1"><thead><tr><th>A</th><th>B</th><th>C</th><th>D</th></tr></thead><tbody><tr><td>a</td><td>b</td><td>c</td><td>d</td></tr><tr><td>a</td><td>b</td><td>e</td><td>f</td></tr><tr><td>a</td><td>b</td><td>d</td><td>e</td></tr><tr><td>b</td><td>c</td><td>e</td><td>f</td></tr><tr><td>e</td><td>d</td><td>c</td><td>d</td></tr><tr><td>e</td><td>d</td><td>e</td><td>f</td></tr></tbody></table>	A	B	C	D	a	b	c	d	a	b	e	f	a	b	d	e	b	c	e	f	e	d	c	d	e	d	e	f	/	<table border="1"><thead><tr><th>C</th><th>D</th></tr></thead><tbody><tr><td>c</td><td>d</td></tr><tr><td>e</td><td>f</td></tr></tbody></table>	C	D	c	d	e	f
A	B	C	D																																	
a	b	c	d																																	
a	b	e	f																																	
a	b	d	e																																	
b	c	e	f																																	
e	d	c	d																																	
e	d	e	f																																	
C	D																																			
c	d																																			
e	f																																			
	=	<table border="1"><thead><tr><th>A</th><th>B</th></tr></thead><tbody><tr><td>a</td><td>b</td></tr><tr><td>e</td><td>d</td></tr></tbody></table>	A	B	a	b	e	d																												
A	B																																			
a	b																																			
e	d																																			

Example

Which members are enrolled in all sports?

EnrolledIn

Member	Sport	Payment
6078	GM	25
5819	KB	30
4526	KB	30
4526	SW	20
3955	KB	30
3955	SW	20
3955	GM	25
9876	KB	0

Sports

ID	Name
KB	Kickbox
SW	Swimming
GM	Gimnastics

Example

Which members are enrolled in all sports?

Member	Sport	Payment
6078	GM	25
5819	KB	30
4526	KB	30
4526	SW	20
3955	KB	30
3955	SW	20
3955	GM	25
9876	KB	0

EnrolledIn

ID	Name
KB	Kickbox
SW	Swimming
GM	Gimnastics

Sports

$$A = \pi_{Member, Sport} \text{ EnrolledIn}$$

Member	Sport
6078	GM
5819	KB
4526	KB
4526	SW
3955	KB
3955	SW
3955	GM
9876	KB

$$B = \pi_{ID} \text{ Sports}$$

ID
KB
SW
GM

A/B

Member
3955

Division doesn't add expressive power

$R(a_1, \dots, a_n, b_1, \dots, b_m)$

$S(b_1, \dots, b_m)$

$$R/S = \Pi_{a_1, \dots, a_n} (R) - \Pi_{a_1, \dots, a_n} [(\Pi_{a_1, \dots, a_n} (R) \times S) - R]$$

Division doesn't add expressive power

$$R/S = \Pi_{a_1, \dots, a_n}(R) - \Pi_{a_1, \dots, a_n}[(\Pi_{a_1, \dots, a_n}(R) \times S) - R]$$

R	
Member	Sport
5819	KB
4526	KB
4526	SW
3955	KB
3955	SW
3955	GM

S
ID
KB
SW
GM

$\Pi_{a_1, \dots, a_n}(R)$
Member
5819
4526
3955

All tuples from the first
n attributes of R

Division doesn't add expressive power

$$R/S = \Pi_{a_1, \dots, a_n}(R) - \Pi_{a_1, \dots, a_n} [(\Pi_{a_1, \dots, a_n}(R) \times S) - R]$$

$\Pi_{a_1, \dots, a_n}(R)$
Member
5819
4526
3955

S
ID
KB
SW
GM

$\Pi_{a_1, \dots, a_n}(R) \times S$	
Member	ID
5819	KB
5819	SW
5819	GM
4526	KB
4526	SW
4526	GM
3955	KB
3955	SW
3955	GM

All combinations of the first n attributes of R with the tuples of S

Division doesn't add expressive power

$$R/S = \Pi_{a_1, \dots, a_n}(R) - \Pi_{a_1, \dots, a_n} [(\Pi_{a_1, \dots, a_n}(R) \times S) - R]$$

$\Pi_{a_1, \dots, a_n}(R) \times S$

Member	ID
5819	KB
5819	SW
5819	GM
4526	KB
4526	SW
4526	GM
3955	KB
3955	SW
3955	GM

R

Member	Sport
5819	KB
4526	KB
4526	SW
3955	KB
3955	SW
3955	GM

$\Pi_{a_1, \dots, a_n}(R) \times S - R$

Member	ID
5819	SW
5819	GM
4526	GM

All combinations of the first n attributes of R with the tuples of S **excluding the tuples that are present in R**

Division doesn't add expressive power

$$R/S = \Pi_{a_1, \dots, a_n} (R) - \Pi_{a_1, \dots, a_n} [(\Pi_{a_1, \dots, a_n} (R) \times S) - R]$$

$\Pi_{a_1, \dots, a_n} (R) \times S - R$

Member	ID
5819	SW
5819	GM
4526	GM

$\Pi_{a_1, \dots, a_n} [(\Pi_{a_1, \dots, a_n} (R) \times S) - R]$

Member
5819
4526

First n attributes of all combinations of the first n attributes of R with the tuples of S excluding the tuples that are present in R

R

Member	Sport
5819	KB
4526	KB
4526	SW
3955	KB
3955	SW
3955	GM

S

ID
KB
SW
GM

Division doesn't add expressive power

$$R/S = \Pi_{a_1, \dots, a_n} (R) - \Pi_{a_1, \dots, a_n} [(\Pi_{a_1, \dots, a_n} (R) \times S) - R]$$

Member	Sport
5819	KB
4526	KB
4526	SW
3955	KB
3955	SW
3955	GM

Member
5819
4526

Member
3955

Rename operator

Changes the schema, not the instance

General Notation

$$\rho_{R(A_1, \dots, A_n)}(E)$$

Abbreviated notation to only change the relation name

$$\rho_R(E)$$

Abbreviated notation to only change attribute names

$$\rho_{A_1, \dots, A_n}(E)$$

Example

Student

sID	sName	GPA	HS

$\rho_{Student2(ID, Name, Grade, HighSchool)}(Student)$



Student2

ID	Name	Grade	HighSchool

Rename operator in use

To unify schemas for set operators

List of colleges and student names

$$\rho_{C(name)}(\pi_{cName} College) \cup \rho_{C(name)}(\pi_{sName} Student)$$

College

cName	state	enr

Student

sID	sName	GPA	HS

Rename operator in use

For disambiguation in “self-joins”

Pairs of colleges in the same state

$$\sigma_{state=state}(College \times College) ?$$
$$\sigma_{s1=s2}(\rho_{c_1(n_1,s_1,e_1)}(College) \times \rho_{c_2(n_2,s_2,e_2)}(College))$$
$$\rho_{c_1(n_1,s_1,e_1)}(College) \bowtie \rho_{c_2(n_2,s_2,e_2)}(College) ?$$
$$\rho_{c_1(n_1,s,e_1)}(College) \bowtie \rho_{c_2(n_2,s,e_2)}(College)$$

College		
cName	state	enr

Rename operator in use

Pairs of **different** colleges in the same state

$$\sigma_{n_1 \neq n_2} (\rho_{c_1(n_1, s, e_1)}(\text{College}) \bowtie \rho_{c_2(n_2, s, e_2)}(\text{College}))$$

~~MIT MIT~~

Pairs of **different** colleges in the same state without repeated pairs

$$\sigma_{n_1 > n_2} (\rho_{c_1(n_1, s, e_1)}(\text{College}) \bowtie \rho_{c_2(n_2, s, e_2)}(\text{College}))$$

~~Stanford Berkeley~~
~~Berkeley Stanford~~

College

cName	state	enr
MIT	Massachusetts	NULL
Stanford	California	NULL
Berkeley	California	NULL

Agenda

Introduction to Relational Algebra

Operators

Alternate notations

Extensions to Relational Algebra

Assignment statements

Pairs of different colleges in the same state

$$\sigma_{n_1 < n_2}(\rho_{c_1(n_1, s, e_1)}(\text{College}) \bowtie \rho_{c_2(n_2, s, e_2)}(\text{College}))$$

Alternative notation

$$C1 := \rho_{c_1(n_1, s, e_1)}(\text{College})$$

$$C2 := \rho_{c_2(n_2, s, e_2)}(\text{College})$$

$$CP := C1 \bowtie C2$$

$$AnsW := \sigma_{n_1 < n_2} CP$$

College		
cName	state	enr

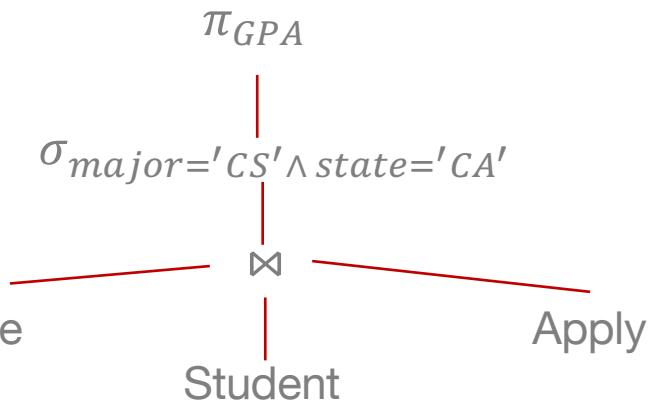
Expression trees

GPA of students applying to CS in CA

$$\pi_{GPA}(\sigma_{major='CS' \wedge state='CA'}(Student \bowtie Apply \bowtie College))$$

Alternative notation

The leaves are always relation names



College

cName	state	enr

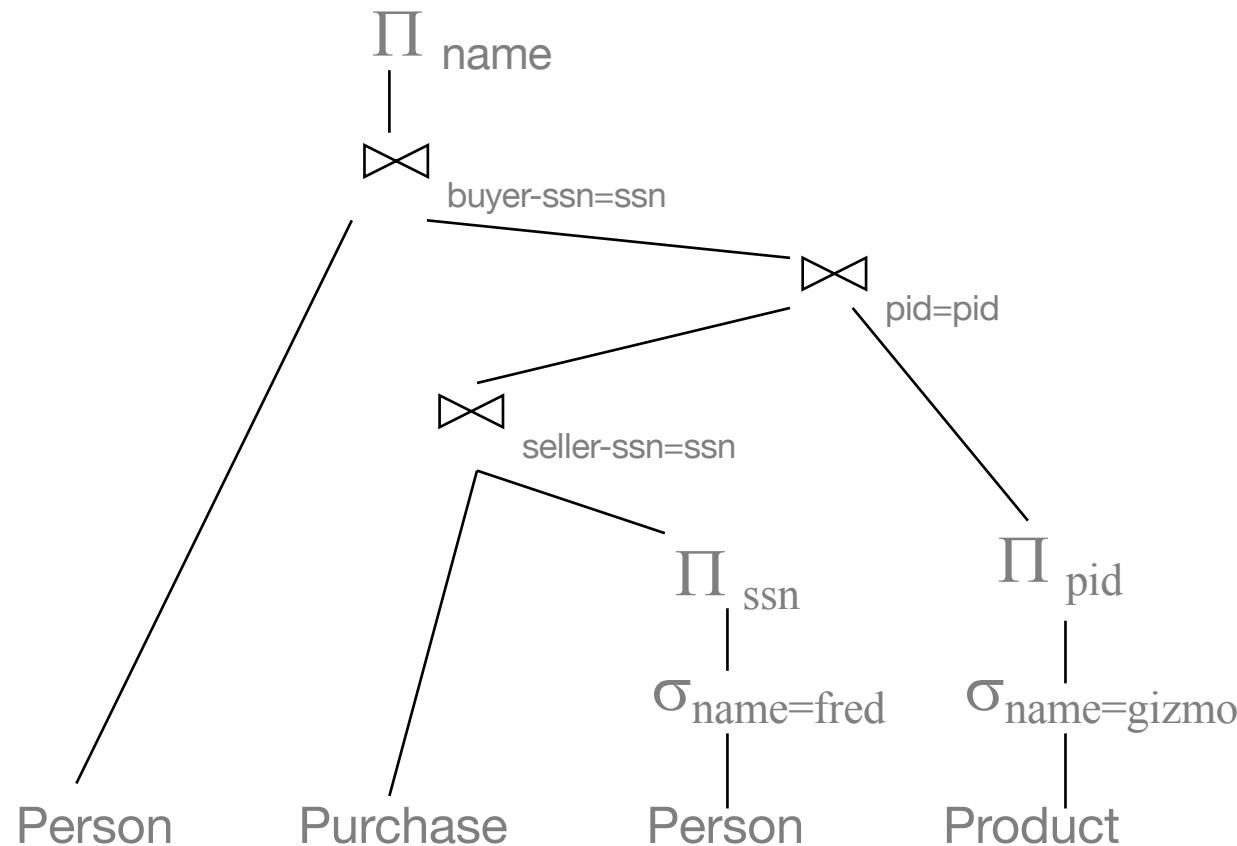
Student

sID	sName	GPA	HS

Apply

sID	cName	major	dec

Expressions can get complex



Agenda

Introduction to Relational Algebra

Operators

Alternate notations

Extensions to Relational Algebra

Extensions to Relational Algebra

Arithmetic expressions in projections

Student

sID	sName	GPA	HS
12	Mary	3.5	90
23	John	3.8	50
31	Jane	3.9	1000

List the student ids with the GPA increased in 0.2

$\pi_{sID, new=GPA+0.2} Student$

In the parameters of the projection

Left side can only be the name of a new attribute

Right side can only involve attributes of the involved relation

Extensions to Relational Algebra

Aggregation operators
cnt, sum, avg, max, min

$\pi_{cnt(*)} R$

Relation with 1 tuple and 1 attribute containing the number of tuples in R

College

cName	state	enr
MIT	Massachusetts	30000
Stanford	California	20000
Berkeley	California	10000
Harvard	Massachusetts	NULL



$\pi_{cnt(*)} \text{ College}$
cnt(*)
4

Extensions to Relational Algebra

$$\pi_{cnt(B)} R$$

Relation with 1 tuple and 1 attribute containing the number of tuples in R with non-null values in B

College

cName	state	enr
MIT	Massachusetts	30000
Stanford	California	20000
Berkeley	California	10000
Harvard	Massachusetts	NULL



$\pi_{cnt(enr)} \text{ College}$

cnt(enr)
3

Extensions to Relational Algebra

 $\pi_{\max(B)} R$

Relation with 1 tuple and 1 attribute containing the maximum value in B

College

cName	state	enr
MIT	Massachusetts	30000
Stanford	California	20000
Berkeley	California	10000
Harvard	Massachusetts	NULL



$\pi_{\max(enr)} \text{ College}$

max(enr)
30000

Extensions to Relational Algebra

$$\pi_{A,\max(B)} R$$

Relation with 1 tuple per each value of A and 2 attributes: the value of A and the maximum value of B for that value of A

College

cName	state	enr
MIT	Massachusetts	30000
Stanford	California	20000
Berkeley	California	10000
Harvard	Massachusetts	NULL



$\pi_{state,\max(enr)} College$

state	max(enr)
Massachusetts	30000
California	20000

Relational Algebra Operators Summary

Core operators

R

$\sigma_{condition} E$

$\pi_{A_1, \dots, A_n} E$

$E_1 \times E_2$

$E_1 \cup E_2$

$E_1 - E_2$

$\rho_{R(A_1, \dots, A_n)}(E)$

Derived operators

$E_1 \bowtie E_2$

$E_1 \bowtie_\theta E_2$

$E_1 \cap E_2$

$E_1 \ltimes E_2$

E_1 / E_2

Parentheses are used for disambiguation

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 2.4 – An Algebraic Query Language

Section 5.2 – Extended Operators of Relational Algebra

SQL – Data Manipulation Language

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Agenda

Introduction

The JOIN family of operators

Basic SQL Statement

Aggregation

Table Variables and Set
Operators

Null values

Subqueries in WHERE clauses

Data Modification statements

Subqueries in FROM and
SELECT clauses

SQL

Stands for Structured Query Language

Pronounced “sequel”

Supported by all major commercial database systems

Standardized – many features over time

Interactive via GUI or prompt, or embedded in programs

Declarative, based on relational algebra

SQL History

1970 “A Relational Model of Data for Large Shared Data Banks” by Edgar Codd

Early 70's SEQUEL Developed at IBM by Donald Chamberlin e Raymond Boyce

1979 First commercial version by Relational Software (now Oracle)

1986 SQL-86 and SQL-87. Ratified by ANSI and ISO

1989 SQL-891992 SQL-92. Also known as SQL2

1999 SQL:1999. Also known as SQL3. Includes regular expressions, recursive queries, triggers, non-scalar data types and some object-oriented expressions

2003 SQL:2003 XML support and auto-generated values

2006 SQL:2006 XQuery support

2008 SQL:2008

2011 SQL:2011

SQL is a ...

Data Definition Language (DDL)

Define relational schemata

Create/alter/delete tables and their attributes

Data Manipulation Language (DML)

Insert/delete/modify tuples in tables

Query one or more tables

Standard

Many standards out there

Database management systems implement something similar, but not identical to the standard for SQL

These slides will try to adhere to the standard as much as possible

Primarily the SQL2 standard and some constructs from the SQL3 standard

Sometimes we'll talk specifically about SQL as understood by SQLite

Agenda

Introduction

Basic SELECT Statement

Table Variables and Set Operators

Subqueries in WHERE clauses

Subqueries in FROM and SELECT clauses

The JOIN family of operators

Aggregation

Null values

Data Modification statements

The Basic SELECT Statement (SFW)

SELECT A_1, A_2, \dots, A_n  What to return

FROM R_1, R_2, \dots, R_m  Identifies the relations to query

WHERE condition  Combines and filters relations

What's the equivalent in Relational Algebra?

$$\pi_{A_1, \dots, A_n} (\sigma_{\text{condition}} (R_1 \times \dots \times R_m))$$

The result is a relation with the schema A_1, A_2, \dots, A_n

SQL is compositional

Relational query languages are compositional

When a query is run over relations, the result is a relation

The schema of the obtained relation is the set of attributes that are returned

The output of one query can be used as the input to another (nesting)

This is extremely powerful

College Admission Database

Apply

sID	cName	major	dec
123	Stanford	CS	Y
123	Stanford	EE	N
123	Berkeley	CS	Y
123	Cornell	EE	Y
234	Berkeley	biology	N
345	MIT	bioengineering	Y
345	Cornell	bioengineering	N
345	Cornell	CS	Y
345	Cornell	EE	N
678	Stanford	history	Y
987	Stanford	CS	Y
987	Berkeley	CS	Y
876	Stanford	CS	Y
876	MIT	biology	Y
876	MIT	marine biology	N
765	Stanford	history	Y
765	Cornell	history	N
765	Cornell	psychology	Y
543	MIT	CS	N

College

cName	state	enr
Stanford	CA	15000
Berkeley	CA	36000
MIT	MA	10000
Cornell	NY	21000

Student

sID	sName	GPA	HS
123	Amy	3.9	1000
234	Bob	3.6	1500
345	Craig	3.5	500
456	Doris	3.9	1000
567	Edward	2.9	2000
678	Fay	3.8	200
789	Gary	3.4	800
987	Helen	3.7	800
876	Irene	3.9	400
765	Jay	2.9	1500
654	Amy	3.9	1000
543	Craig	3.4	2000

SQL query with one relation

```
SELECT sID, sName, GPA  
FROM Student  
WHERE GPA > 3.6;
```

sID	sName	GPA
123	Amy	3.9
456	Doris	3.9
678	Fay	3.8
987	Helen	3.7
876	Irene	3.9
654	Amy	3.9

Not necessary to include GPA in the result even if we filter on the GPA

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

A Few Details

SQL commands are case insensitive

Same: SELECT, Select, select

Same: Student, student

Values are not

Different: ‘Stanford’, ‘stanford’

Use single quotes for constants

‘abc’ - yes

“abc” - no

SQL query combining two relations

```
SELECT sName, major  
FROM Student, Apply  
WHERE Student.sID=Apply.sID;
```



This would happen automatically in a natural join of RA

What does it computes?

Duplicate values



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

sName	major
Amy	CS
Amy	EE
Amy	CS
Amy	EE
Bob	biology
Craig	bioengineering
Craig	bioengineering
Craig	CS
Craig	EE
Fay	history
Helen	CS
Helen	CS
Irene	CS
Irene	biology
Irene	marine biology
Jay	history
Jay	history
Jay	psychology
Craig	CS

SQL query excluding duplicate values

```
SELECT DISTINCT sName, major  
FROM Student, Apply  
WHERE Student.sID=Apply.sID;
```

No duplicate values



sName	major
Amy	CS
Amy	EE
Bob	biology
Craig	bioengineering
Craig	CS
Craig	EE
Fay	history
Helen	CS
Irene	CS
Irene	biology
Irene	marine biology
Jay	history
Jay	psychology

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

Another SQL query combining two relations

```
SELECT sName, GPA, decision  
FROM Student, Apply  
WHERE Student.sID=Apply.sID AND sizeHS<1000 AND  
      major='CS' AND cName='Stanford';
```

What does it compute?

sName	GPA	decision
Helen	3.7	Y
Irene	3.9	N

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

One more SQL query combining two relations

```
SELECT cName
```

```
FROM College, Apply
```

```
WHERE College.cName=Apply.cName AND  
enr>20000 AND major='CS';
```

What does it compute?

SQLite Error: ambiguous
column name: cName

How can we correct it?

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

One more SQL query combining two relations

```
SELECT College.cName
```

```
FROM College, Apply
```

```
WHERE College.cName=Apply.cName AND  
enr>20000 AND major='CS';
```

cName
Berkeley
Berkeley
Cornell



How can we eliminate duplicates?

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

Order of the results

SQL is based on an unordered model

The order of the results may change each time we run a query

We can ask for a result to be sorted, in ascending or descending order, by an attribute or set of attributes

ORDER BY <attributes> ASC

ORDER BY <attributes> DESC

Ordering is ascending, unless you specify the DESC keyword

Order of the results

College(cName, state, enr)

Student(sID, sName, GPA, sizeHS)

Apply(sID, cName, major, decision)

```
SELECT Student.sID, sName, GPA,  
       Apply.cName, enr  
FROM   Student, College, Apply  
WHERE  Student.sID=Apply.sID AND  
       Apply.cName=College.cName;
```

sID	sName	GPA	cName	enr
123	Amy	3.9	Stanford	15000
123	Amy	3.9	Stanford	15000
123	Amy	3.9	Berkeley	36000
123	Amy	3.9	Cornell	21000
234	Bob	3.6	Berkeley	36000
345	Craig	3.5	MIT	10000
345	Craig	3.5	Cornell	21000
345	Craig	3.5	Cornell	21000
345	Craig	3.5	Cornell	21000
678	Fay	3.8	Stanford	15000

```
SELECT Student.sID, sName, GPA,  
       Apply.cName, enr  
FROM   Student, College, Apply  
WHERE  Student.sID=Apply.sID AND  
       Apply.cName=College.cName  
ORDER BY GPA DESC;
```

sID	sName	GPA	cName	enr
123	Amy	3.9	Stanford	15000
123	Amy	3.9	Stanford	15000
123	Amy	3.9	Berkeley	36000
123	Amy	3.9	Cornell	21000
876	Irene	3.9	Stanford	15000
876	Irene	3.9	MIT	10000
876	Irene	3.9	MIT	10000
678	Fay	3.8	Stanford	15000
987	Helen	3.7	Stanford	15000
987	Helen	3.7	Berkeley	36000

Order of the results

```
SELECT Student.sID, sName, GPA, Apply.cName, enr  
FROM Student, College, Apply  
WHERE Student.sID=Apply.sID AND Apply.cName=College.cName  
ORDER BY GPA DESC, enr;
```



Descending GPA as primary sort order and, within each of those, ascending enrollment

sID	sName	GPA	cName	enr
876	Irene	3.9	MIT	10000
876	Irene	3.9	MIT	10000
123	Amy	3.9	Stanford	15000
123	Amy	3.9	Stanford	15000
876	Irene	3.9	Stanford	15000
123	Amy	3.9	Cornell	21000
123	Amy	3.9	Berkeley	36000
678	Fay	3.8	Stanford	15000
987	Helen	3.7	Stanford	15000
987	Helen	3.7	Berkeley	36000

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

Like operator: pattern matching on strings

Built-in operator that allows string matching on attribute values

SELECT sID, major

FROM Apply

WHERE major like '%bio%';



Match any major containing bio

sID	major
234	biology
345	bioengineering
345	bioengineering
876	biology
876	marine biology

% = any sequence of characters

_ = any single character

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

Selecting all attributes

SELECT *

FROM Apply

WHERE major like '%bio%';

sID	cName	major	decision
234	Berkeley	biology	N
345	MIT	bioengineering	Y
345	Cornell	bioengineering	N
876	MIT	biology	Y
876	MIT	marine biology	N

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

Arithmetic within SQL clauses

```
SELECT sID, sName, GPA, HS, GPA*(HS/1000)  
FROM Student;
```

sID	sName	GPA	HS	GPA*(HS/1000)
123	Amy	3.9	1000	3.9
234	Bob	3.6	1500	5.4
345	Craig	3.5	500	1.75
456	Doris	3.9	1000	3.9
567	Edward	2.9	2000	5.8
678	Fay	3.8	200	0.76
789	Gary	3.4	800	2.72
987	Helen	3.7	800	2.96
876	Irene	3.9	400	1.56
765	Jay	2.9	1500	4.35
654	Amy	3.9	1000	3.9
543	Craig	3.4	2000	6.8

Boosts GPA if student is from a big high school and reduces it, if he is from a small one

→ Can we improve this result?

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

Renaming columns

```
SELECT sID, sName, GPA, HS, GPA*(HS/1000) AS scaledGPA  
FROM Student;
```

College(<u>cName</u> , state, enr)
Student(<u>sID</u> , sName, GPA, sizeHS)
Apply(<u>sID</u> , <u>cName</u> , <u>major</u> , decision)

sID	sName	GPA	HS	scaledGPA
123	Amy	3.9	1000	3.9
234	Bob	3.6	1500	5.4
345	Craig	3.5	500	1.75
456	Doris	3.9	1000	3.9
567	Edward	2.9	2000	5.8
678	Fay	3.8	200	0.76
789	Gary	3.4	800	2.72
987	Helen	3.7	800	2.96
876	Irene	3.9	400	1.56
765	Jay	2.9	1500	4.35
654	Amy	3.9	1000	3.9
543	Craig	3.4	2000	6.8

Agenda

Introduction

The JOIN family of operators

Basic SQL Statement

Aggregation

Table Variables and Set
Operators

Null values

Subqueries in WHERE clauses

Data Modification statements

Subqueries in FROM and
SELECT clauses

Table variables

Used in the FROM clause for two purposes

Make queries more readable

Rename relations when we have two instances of the same relation

```
SELECT Student.sID, sName, GPA, Apply.cName, enr  
FROM Student, College, Apply  
WHERE Student.sID=Apply.sID AND Apply.cName=College.cName;
```

```
SELECT S.sID, sName, GPA, A.cName, enr  
FROM Student S, College C, Apply A  
WHERE S.sID=A.sID AND A.cName=C.cName;
```



Not changing the result, only making the query more readable

Table variables

How to list the pairs of students who have the same GPA?

Student(sID,sName,GPA,sizeHS)

```
SELECT S1.sID, S1.sName, S1.GPA, S2.sID, S2.sName, S2.GPA  
FROM Student S1, Student S2  
WHERE S1.GPA=S2.GPA;
```

How to list only pairs
of different students?



sID	sName	GPA	sID1	sName1	GPA1
123	Amy	3.9	123	Amy	3.9
123	Amy	3.9	456	Doris	3.9
123	Amy	3.9	876	Irene	3.9
123	Amy	3.9	654	Amy	3.9
234	Bob	3.6	234	Bob	3.6
345	Craig	3.5	345	Craig	3.5
456	Doris	3.9	123	Amy	3.9
456	Doris	3.9	456	Doris	3.9
...

Table variables

```
SELECT S1.sID, S1.sName, S1.GPA, S2.sID, S2.sName, S2.GPA  
FROM Student S1, Student S2  
WHERE S1.GPA=S2.GPA AND S1.sID <> S2.sID;
```

How to exclude the same pairs in different order?



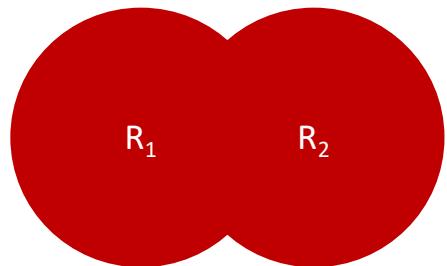
sID	sName	GPA	sID1	sName1	GPA1
123	Amy	3.9	456	Doris	3.9
123	Amy	3.9	876	Irene	3.9
123	Amy	3.9	654	Amy	3.9
456	Doris	3.9	123	Amy	3.9
456	Doris	3.9	876	Irene	3.9
456	Doris	3.9	654	Amy	3.9
567	Edward	2.9	765	Jay	2.9
...

Table variables

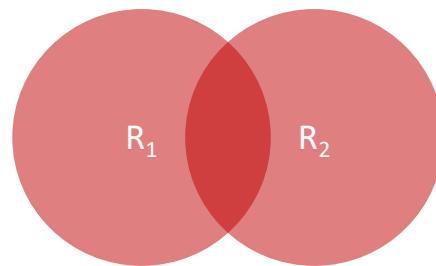
```
SELECT S1.sID, S1.sName, S1.GPA, S2.sID, S2.sName, S2.GPA  
FROM Student S1, Student S2  
WHERE S1.GPA=S2.GPA AND S1.sID < S2.sID;
```

sID	sName	GPA	sID1	sName1	GPA1
123	Amy	3.9	456	Doris	3.9
123	Amy	3.9	876	Irene	3.9
123	Amy	3.9	654	Amy	3.9
456	Doris	3.9	876	Irene	3.9
456	Doris	3.9	654	Amy	3.9
567	Edward	2.9	765	Jay	2.9
654	Amy	3.9	876	Irene	3.9
543	Craig	3.4	789	Gary	3.4

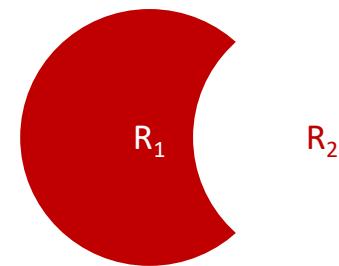
Set operators



Union



Intersect



Except

Sets, Bags and Lists

Sets

Only one occurrence of each element

Unordered elements

Bags (or multisets)

More than one occurrence of an element

Unordered elements and their occurrences

Lists

More than one occurrence of an element

Occurrences are ordered

Recall Bags

$\lambda(X)$ = “Count of tuple in X”

Multiset X

Tuple
(1, a)
(1, a)
(1, b)
(2, c)
(2, c)
(2, c)
(1, d)
(1, d)



Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	1
(2, c)	3
(1, d)	2

In a set all counts are (0,1).

Union as a bag operation

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

U

Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

Multiset Z

Tuple	$\lambda(Z)$
(1, a)	7
(1, b)	1
(2, c)	5
(1, d)	2

=

$$\lambda(Z) = \lambda(X) + \lambda(Y)$$

Intersect as a bag operation

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0



Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

Multiset Z

Tuple	$\lambda(Z)$
(1, a)	2
(1, b)	0
(2, c)	2
(1, d)	0



$$\lambda(Z) = \min(\lambda(X), \lambda(Y))$$

Except as a bag operation

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

—

Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

==

Multiset Z

Tuple	$\lambda(Z)$
(1, a)	0
(1, b)	0
(2, c)	1
(1, d)	0

If $\lambda(X) > \lambda(Y)$

$$\lambda(Z) = \lambda(X) - \lambda(Y)$$

Else

0

Union operator

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

```
SELECT cName FROM College  
UNION  
SELECT sName FROM Student;
```

In SQL, the two sides of the union don't have to be the same

How to unify the two schemas?

```
SELECT cName AS name FROM College  
UNION  
SELECT sName AS name FROM Student;
```

cName
Amy
Berkeley
Bob
Cornell
Craig
Doris
Edward
Fay
Gary
Helen
Irene
Jay
MIT
Stanford

Union operator

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

By default, in SQL, the union operator eliminates duplicates

If we want to have duplicates in our result

```
SELECT cName AS name FROM College
```

```
UNION ALL
```

```
SELECT sName AS name FROM Student;
```

Result is not sorted anymore

Why?

How can we sort the result?

```
SELECT cName AS name FROM College
```

```
UNION ALL
```

```
SELECT sName AS name FROM Student
```

```
ORDER BY name;
```

name
Stanford
Berkeley
MIT
Cornell
Amy
Bob
Craig
Doris
Edward
Fay
Gary
Helen
Irene
Jay
Amy
Craig

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 6.1 – Simple Queries in SQL

Section 6.2 – Queries Involving More Than One Relation

Section 6.3 - Subqueries

Section 6.4 – Full-Relation Operations

Section 6.5 – Database Modifications

Philip Greenspun, SQL for Web Nerds,
<http://philip.greenspun.com/sql/>

SQL – Data Manipulation Language

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Intersect operator

```
SELECT sID FROM Apply WHERE major='CS'
```

```
INTERSECT
```

```
SELECT sID FROM Apply WHERE major='EE';
```

sID
123
345

Some DBMS don't support the intersect operator

How to rewrite this query without the intersect operator?

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

Intersect operator

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

```
SELECT A1.sID
FROM Apply A1, Apply A2
WHERE A1.sID = A2.sID AND A1.major = 'CS' AND A2.major = 'EE';
```

Duplicates are not eliminated now

How to eliminate them?

```
SELECT DISTINCT A1.sID
FROM Apply A1, Apply A2
WHERE A1.sID = A2.sID AND A1.major = 'CS' AND A2.major = 'EE';
```

sID
123
123
123
123
345

Except operator

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

It's called *difference* in Relational Algebra

SELECT sID FROM Apply WHERE major='CS'

EXCEPT

SELECT sID FROM Apply WHERE major='EE';

sID
543
876
987

Some DBMS don't support the except operator

How to rewrite this query without the except operator?

Except operator

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

```
SELECT DISTINCT A1.sID
FROM Apply A1, Apply A2
WHERE A1.sID=A2.sID AND A1.major='CS' AND A2.major<>'EE';
```

Finding one pair that satisfies this doesn't mean there's not another pair with the same student where he applied to 'CS' and 'EE'

This query is finding students who applied to 'CS'

Not possible to rewrite the except query with the operators seen until now

sID
123
345
543
876
987

Agenda

Introduction

The JOIN family of operators

Basic SQL Statement

Aggregation

Table Variables and Set
Operators

Null values

Subqueries in WHERE clauses

Data Modification statements

Subqueries in FROM and
SELECT clauses

Subqueries in WHERE

SELECT A₁, A₂, ..., A_n

FROM R₁, R₂, ..., R_m

WHERE condition



Expressions involving
subqueries

Subqueries are nested SELECT statements

Subqueries generate sets that are used for comparison

A first example

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

```
SELECT sID, sName  
FROM Student  
WHERE sID in  
(SELECT sID FROM Apply WHERE major='CS');
```

sID	sName
123	Amy
345	Craig
987	Helen
876	Irene
543	Craig

Subquery that finds the IDs of students who have applied to a major in CS

IDs and names of students who have applied to a major in CS at some college

How can we do this query without the subquery?

A first example

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

```
SELECT DISTINCT Student.sID, sName
FROM Student, Apply
WHERE Student.sID = Apply.sID and major='CS';
```

sID	sName
123	Amy
345	Craig
987	Helen
876	Irene
543	Craig

Could we write sID instead of Student.sID?

Why is DISTINCT necessary here and not in the previous query?

A second example - duplicates

```
SELECT sName  
FROM Student  
WHERE sID in
```

```
(SELECT sID FROM Apply WHERE major='CS');
```

sName
Amy
Craig
Helen
Irene
Craig

Names of students who have applied to a major in CS at some college

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

A second example - duplicates

```
SELECT DISTINCT sName  
FROM Student, Apply  
WHERE Student.sID = Apply.sID and major='CS';
```

sName
Amy
Craig
Helen
Irene

Why is the result different from the one in the previous query?

The two different Craigs turned in one result

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

Third example - duplicates

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

Can we list the GPA of students who have applied to a major in CS at some college using a subquery?

```
SELECT GPA
FROM Student
WHERE sID in
      (SELECT sID FROM Apply WHERE major='CS');
```

GPA
3.9
3.5
3.7
3.9
3.4

Third example - duplicates

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

Can we list the GPA of students who have applied to a major in CS at some college **without** using a subquery?

No

```
SELECT DISTINCT GPA
FROM Student, Apply
WHERE Student.sID = Apply.sID and major='CS';
```

GPA
3.9
3.5
3.7
3.4

The same problem as in previous example

Fourth example

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

Can we now write a query to list the students that have applied to a major in CS but have not applied to a major in EE, without using the except operator?

```
SELECT sID, sName  
FROM Student
```

```
WHERE sID in (select sID from Apply where major='CS') AND  
      sID not in (select sID from Apply where major='EE');
```

↓
Equivalent

```
SELECT sID, sName  
FROM Student
```

```
WHERE sID in (select sID from Apply where major='CS') AND  
      not sID in (select sID from Apply where major='EE');
```

sID	sName
987	Helen
876	Irene
543	Craig

Exists operator

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

Checks whether a subquery is empty or not

Write a query that finds all colleges, such that there's some other college that is in the same state

cName	state	enr
Stanford	CA	15000
Berkeley	CA	36000
MIT	MA	10000
Cornell	NY	21000



cName	state
Stanford	CA
Berkeley	CA

Exists operator

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

```
SELECT cName, state
```

```
FROM College C1
```

```
WHERE exists (select * from College C2  
where C2.state=C1.state);
```

Correlated reference

cName	state
Stanford	CA
Berkeley	CA
MIT	MA
Cornell	NY



What's the problem?



Every college is in the same state as itself

Exists operator

```
SELECT cName, state  
FROM College C1  
WHERE exists (select * from College C2  
              where C2.state=C1.state and C1.cName<>C2.cName);
```

cName	state
Stanford	CA
Berkeley	CA

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

Getting the largest value

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

Find the college that has the largest enrollment

```
SELECT cName  
FROM College C1  
WHERE not exists (select * from College C2  
                  where C2.enr > C1.enr);
```

Find all colleges where there does not exist another college whose enrollment is higher than the first college

cName
Berkeley

To look for something that's the largest or the smallest

Getting the largest value – example 2

Find the student with the highest GPA

```
SELECT sName  
FROM Student C1  
WHERE not exists (select * from Student C2  
                  where C2.GPA > C1.GPA);
```

sName
Amy
Doris
Irene
Amy

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

Getting the largest value – example 2

Can we find the student with the highest GPA without subqueries?

```
SELECT S1.sName, S1.GPA  
FROM Student S1, Student S2  
WHERE S1.GPA > S2.GPA;
```

→ Does this work?

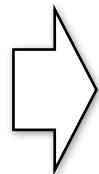
↓
Cannot do it using joins

↓
Finds all students such that there is some other student whose GPA is lower, that is, all students except those who have the lowest GPA

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

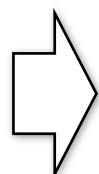
Alternatives to Intersect and Except

```
(SELECT R.A, R.B  
FROM R)  
  
INTERSECT  
  
(SELECT S.A, S.B  
FROM S)
```



```
SELECT R.A, R.B  
FROM R  
WHERE EXISTS(  
    SELECT *  
    FROM S  
    WHERE R.A=S.A AND R.B=S.B)
```

```
(SELECT R.A, R.B  
FROM R)  
  
EXCEPT  
  
(SELECT S.A, S.B  
FROM S)
```



```
SELECT R.A, R.B  
FROM R  
WHERE NOT EXISTS(  
    SELECT *  
    FROM S  
    WHERE R.A=S.A AND R.B=S.B)
```

All operator

Checks whether the value has a certain relationship with **all** the results of the subquery

```
SELECT sName  
FROM Student  
WHERE GPA >= all (select GPA from Student);
```

sName
Amy
Doris
Irene
Amy

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

All operator

```
SELECT sName  
FROM Student S1  
WHERE GPA > all (select GPA from Student S2  
                   where S2.sID <> S1.sID);
```

sName

Does this work?

The query would be correct if we knew that every student's GPA was unique

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

Any operator

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

Checks whether the value has a certain relationship with **at least one** element of the results of the subquery

```
SELECT sName
FROM Student
WHERE not GPA < any (select GPA from Student);
```

```
SELECT sName
FROM Student
WHERE GPA>= all (select GPA from Student);
```



$$\forall_{x \in X} \neg P(x) \equiv \neg \exists_{x \in X} P(x)$$

sName
Amy
Doris
Irene
Amy

Any operator – example 2

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

Find all students who are not from the smallest high school in the database

```
SELECT sID, sName, sizeHS  
FROM Student  
WHERE sizeHS > any (select sizeHS from Student);
```

SQLite does not support the any and all operators

Have to rewrite the queries using exists and not exists

How can we rewrite this query using exists?

sID	sName	HS
123	Amy	1000
234	Doris	1500
345	Irene	500
456	Amy	1000
567	Edward	2000
789	Gary	800
987	Helen	800
876	Irene	400
765	Jay	1500
654	Amy	1000
543	Craig	2000

Any operator – example 2

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

```
SELECT sID, sName, sizeHS  
FROM Student S1  
WHERE exists (select * from Student S2  
              where S2.sizeHS < S1.sizeHS);
```

An *any* or *all* query can always be written using *exists* or *not exists*

sID	sName	HS
123	Amy	1000
234	Doris	1500
345	Irene	500
456	Amy	1000
567	Edward	2000
789	Gary	800
987	Helen	800
876	Irene	400
765	Jay	1500
654	Amy	1000
543	Craig	2000

Any operator – example 3

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

Can we rewrite the query that finds students who have applied to a major in CS and have not applied to a major in EE?

```
SELECT sID, sName  
FROM Student  
WHERE sID in (select sID from Apply where major='CS') AND  
      sID not in (select sID from Apply where major='EE');
```

sID	sName
987	Helen
876	Irene
543	Craig

```
SELECT sID, sName  
FROM Student  
WHERE sID = any (select sID from Apply where major = 'CS')  
AND sID <> any (select sID from Apply where major = 'EE');
```

Satisfied as long as there's anybody who applied to EE
that is not the same as the student we're looking at

sID	sName
123	Amy
345	Craig
987	Helen
876	Irene
543	Craig

Any operator – example 3

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

SELECT sID, sName

FROM Student

WHERE sID = any (select sID from Apply where major = 'CS')
AND sID <> any (select sID from Apply where major = 'EE');

sID	sName
123	Amy
345	Craig
987	Helen
876	Irene
543	Craig

How can we correct it?

SELECT sID, sName

FROM Student

WHERE sID = any (select sID from Apply where major = 'CS')
AND not sID = any (select sID from Apply where major = 'EE');

sID	sName
987	Helen
876	Irene
543	Craig

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 6.1 – Simple Queries in SQL

Section 6.2 – Queries Involving More Than One Relation

Section 6.3 - Subqueries

Section 6.4 – Full-Relation Operations

Section 6.5 – Database Modifications

Philip Greenspun, SQL for Web Nerds,
<http://philip.greenspun.com/sql/>

SQL – Data Manipulation Language

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Operators summary

attribute [**NOT**] **IN** Relation

[**NOT**] attribute **IN** Relation

[**NOT**] **EXISTS** Relation

attribute <comparison> **ALL** Relation

ALL and ANY not supported by SQLite

attribute <comparison> **ANY** Relation

Comparison may be: <;>; <=;>=; =; <>

Existential quantifier

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

An existential quantifier (\exists) is a quantifier of the form “there exists”

Find colleges with some applications of students with a GPA higher than 3.8

```
SELECT DISTINCT cName
FROM Apply, Student
WHERE Apply.sID=Student.sID AND GPA>3.8;
```

Existential is easy!

cName
Stanford
Berkeley
MIT
Cornell

Universal quantifier

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

A universal quantifier (\forall) is a quantifier of the form “for all”

Find colleges with applications of students all having a GPA higher than 3.8

↑
Equivalent
↓

Find colleges that only have applications of students with a GPA higher than 3.8

cName

```
SELECT DISTINCT cName
```

```
FROM Apply
```

```
WHERE College.cName NOT IN (
```

```
    SELECT Apply.cName FROM Apply, Student
```

```
    WHERE Apply.sID=Student.sID AND GPA<=3.8);
```

Agenda

Introduction

The JOIN family of operators

Basic SQL Statement

Aggregation

Table Variables and Set
Operators

Null values

Subqueries in WHERE clauses

Data Modification statements

Subqueries in FROM and
SELECT clauses

Subqueries in FROM and SELECT

SELECT A_1, A_2, \dots, A_n



Expressions involving
subqueries

FROM R_1, R_2, \dots, R_m



WHERE condition

Subqueries are nested SELECT statements

Subqueries in FROM generate a table to be used in the query

Subqueries in SELECT produce a value that comes out of the query

Subqueries in the FROM clause

```
SELECT sID, sName, GPA, GPA*(HS/1000) as scaledGPA  
FROM Student  
WHERE GPA*(HS/1000)-GPA>1.0 OR GPA - GPA*(HS/1000) > 1.0;
```

Return all students whose scaledGPA changes GPA by more than 1

Can we simplify this query?

College(<u>cName</u> , state, enr)
Student(<u>sID</u> , sName, GPA, sizeHS)
Apply(<u>sID</u> , <u>cName</u> , <u>major</u> , decision)

sID	sName	GPA	scaledGPA
234	Bob	3.6	5.4
345	Craig	3.5	1.75
567	Edward	2.9	5.8
678	Fay	3.8	0.76
876	Irene	3.9	1.56
765	Jay	2.9	4.35
543	Craig	3.4	6.8

Subqueries in the FROM clause

```
SELECT sID, sName, GPA, GPA*(HS/1000) as scaledGPA  
FROM Student  
WHERE GPA*(HS/1000)-GPA>1.0 OR GPA – GPA*(HS/1000) > 1.0;
```



```
SELECT sID, sName, GPA, GPA*(HS/1000) as scaledGPA  
FROM Student  
WHERE abs(GPA*(HS/1000)-GPA)>1.0;
```

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

sID	sName	GPA	scaledGPA
234	Bob	3.6	5.4
345	Craig	3.5	1.75
567	Edward	2.9	5.8
678	Fay	3.8	0.76
876	Irene	3.9	1.56
765	Jay	2.9	4.35
543	Craig	3.4	6.8

Subqueries in the FROM clause

```
SELECT sID, sName, GPA, GPA*(HS/1000) as scaledGPA  
FROM Student  
WHERE abs(GPA*(HS/1000)-GPA) > 1.0;
```



```
SELECT *  
FROM (select sID, sName, GPA, GPA*(HS/1000) as scaledGPA  
      from Student) G  
WHERE abs(scaledGPA - GPA) > 1.0;
```

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

sID	sName	GPA	scaledGPA
234	Bob	3.6	5.4
345	Craig	3.5	1.75
567	Edward	2.9	5.8
678	Fay	3.8	0.76
876	Irene	3.9	1.56
765	Jay	2.9	4.35
543	Craig	3.4	6.8

Subqueries in the SELECT clause

```
SELECT DISTINCT College.cName, state, GPA  
FROM College, Apply, Student  
WHERE College.cName = Apply.cName  
      AND Apply.sID = Student.sID  
      AND GPA >= all  
(select GPA from Student, Apply  
where Student.sID = Apply.sID  
and Apply.cName = College.cName);
```

cName	state	GPA
Stanford	CA	3.9
Berkeley	CA	3.9
Cornell	NY	3.9
MIT	MA	3.9

→ How to rewrite it
using a subquery
in SELECT?

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

Return colleges paired with the highest GPA of their applicants

Subqueries in the SELECT clause

```
SELECT cName, state,  
       (SELECT DISTINCT GPA  
        FROM Apply, Student  
       WHERE College.cName = Apply.cName  
             AND Apply.sID = Student.sID  
             AND GPA >= all  
             (select GPA from Student, Apply  
              where Student.sID = Apply.sID  
              and Apply.cName = College.cName)) AS GPA  
  FROM College;
```

Computes the highest GPA for the college

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

Subqueries in the SELECT clause

Query that returns colleges paired with the name of the applicants

```
SELECT cName, state,  
       (SELECT DISTINCT sName  
        FROM Apply, Student  
        WHERE College.cName = Apply.cName  
              AND Apply.sID = Student.sID) AS sName  
     FROM College;
```

→ Error. Why?

↓

↓

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

subquery returns more than 1 row

subquery in SELECT can only return 1 column of 1 tuple

Agenda

Introduction

The JOIN family of operators

Basic SQL Statement

Aggregation

Table Variables and Set
Operators

Null values

Subqueries in WHERE clauses

Data Modification statements

Subqueries in FROM and
SELECT clauses

The JOIN family of operators

SELECT A_1, A_2, \dots, A_n

FROM R_1, R_2, \dots, R_m

WHERE condition



Implicit join

Explicit Joins

Inner join on *condition*

\bowtie_θ

Natural join

\bowtie

Inner join using (*attrs*)

\bowtie explicitly listing the attributes to be equated

Left | Right | Full Outer Join

Combines tuples as in \bowtie_θ but when they don't match they are added to the result with NULL values

None of these operators adds expressive power to SQL

Inner join on condition

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

```
SELECT DISTINCT sName, major  
FROM Student, Apply  
WHERE Student.sID = Apply.sID;
```

Equivalent to:

```
SELECT DISTINCT sName, major  
FROM Student INNER JOIN Apply  
ON Student.sID = Apply.sID;
```

INNER JOIN is the DEFAULT JOIN operator in SQL

```
SELECT DISTINCT sName, major  
FROM Student JOIN Apply  
ON Student.sID = Apply.sID;
```

sName	major
Amy	CS
Amy	EE
Bob	biology
Craig	bioengineering
Craig	CS
Craig	EE
Fay	history
Helen	CS
Irene	CS
Irene	biology
Irene	marine biology
Jay	history
Jay	psychology

Inner join on condition: example 2

```
SELECT sName, GPA  
FROM Student, Apply  
WHERE Student.sID=Apply.sID AND HS<1000 AND major='CS' AND cName='Stanford';
```

↓
Equivalent to:

```
SELECT sName, GPA  
FROM Student JOIN Apply  
ON Student.sID=Apply.sID  
WHERE HS<1000 AND major='CS' AND cName='Stanford';
```

sName	GPA
Helen	3.7
Irene	3.9

↓
Equivalent to:

```
SELECT sName, GPA  
FROM Student JOIN Apply  
ON Student.sID=Apply.sID AND HS<1000 AND major='CS' AND cName='Stanford';
```

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

Inner join on *condition* with 3 relations

```
SELECT Apply.sID, sName, GPA, Apply.cName, enr  
FROM   Apply, Student, College  
  
WHERE Apply.sID = Student.sID AND Apply.cName = College.cName;
```

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

```
SELECT Apply.sID, sName, GPA, Apply.cName, enr  
FROM   Apply JOIN Student JOIN College  
  
ON     Apply.sID = Student.sID AND Apply.cName = College.cName;
```

Runs on SQLite but
not in every system

```
SELECT Apply.sID, sName, GPA, Apply.cName, enr  
FROM   (Apply JOIN Student ON Apply.sID = Student.sID) JOIN College  
ON     Apply.cName = College.cName;
```

Some systems
(e.g.: Postgres)
requires the join
operators to be
binary

Interaction between query and processor

SQL systems tend to follow the structure that is provided by the JOIN operators and parenthesis

```
SELECT Apply.sID, sName, GPA, Apply.cName, enr  
FROM   (Apply JOIN Student ON Apply.sID = Student.sID) JOIN College  
ON Apply.cName = College.cName;
```



Typically, Apply will be joined with Student first

The order by things are done affects query performance

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

Natural join

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

It joins tables based on the attributes with the same name and keeps only one of those attributes

```
SELECT DISTINCT sName, major  
FROM Student, Apply  
WHERE Student.sID = Apply.sID;
```

↓
Equivalent to:

```
SELECT DISTINCT sName, major  
FROM Student NATURAL JOIN Apply;
```

sName	major
Amy	CS
Amy	EE
Bob	biology
Craig	bioengineering
Craig	CS
Craig	EE
Fay	history
Helen	CS
Irene	CS
Irene	biology
Irene	marine biology
Jay	history
Jay	psychology

Natural join with additional conditions

```
SELECT sName, GPA  
FROM Student JOIN Apply  
ON Student.sID=Apply.sID  
WHERE HS<1000 AND major='CS' AND cName='Stanford';
```

↓ Equivalent to:

sName	GPA
Helen	3.7
Irene	3.9

```
SELECT sName, GPA  
FROM Student NATURAL JOIN Apply  
WHERE HS<1000 AND major='CS' AND cName='Stanford';
```

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

Inner join using (attrs)

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

Using lists the attributes that should be acquainted across the 2 relations

Only attributes that appear in both relations

```
SELECT sName, GPA
```

```
FROM Student NATURAL JOIN Apply
```

```
WHERE HS<1000 AND major='CS' AND cName='Stanford';
```

↓ Equivalent to:

sName	GPA
Helen	3.7
Irene	3.9

```
SELECT sName, GPA
```

```
FROM Student JOIN Apply using(sID)
```

```
WHERE HS<1000 AND major='CS' AND cName='Stanford';
```

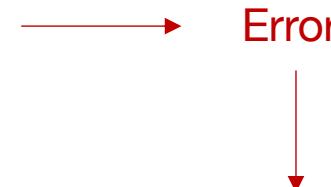
Better practice than using the natural join

Join with more than 1 instance of a relation

```
SELECT S1.sID, S1.sName, S1.GPA, S2.sID, S2.sName, S2.GPA  
FROM Student S1, Student S2  
WHERE S1.GPA=S2.GPA AND S1.sID < S2.sID;
```

Finds pairs of students with the same GPA

```
SELECT S1.sID, S1.sName, S1.GPA, S2.sID, S2.sName, S2.GPA  
FROM Student S1 join Student S2 using(GPA)  
ON S1.sID < S2.sID;
```



College(<u>cName</u> , state, enr)
Student(<u>sID</u> , sName, GPA, sizeHS)
Apply(<u>sID</u> , <u>cName</u> , <u>major</u> , decision)

Error
Using and ON
cannot be used
in combination

Join with more than 1 instance of a relation

```
SELECT S1.sID, S1.sName, S1.GPA, S2.sID, S2.sName, S2.GPA  
FROM Student S1 join Student S2 using(GPA)  
WHERE S1.sID < S2.sID;
```

sID	sName	GPA	sID1	sName1	GPA1
123	Amy	3.9	456	Doris	3.9
123	Amy	3.9	876	Irene	3.9
123	Amy	3.9	654	Amy	3.9
456	Doris	3.9	876	Irene	3.9
456	Doris	3.9	654	Amy	3.9
567	Edward	2.9	765	Jay	2.9
654	Amy	3.9	876	Irene	3.9
543	Craig	3.4	789	Gary	3.4

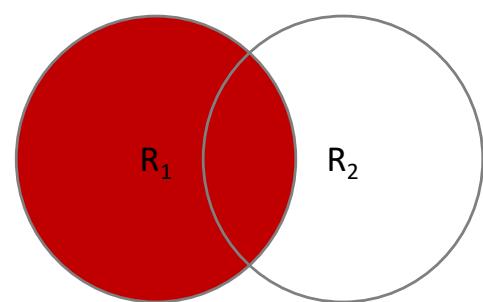
College(cName, state, enr)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

Left outer join

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

Takes any tuples on the left side and if they don't have a match on a tuple from the right, it is still added to the result and padded with NULL values

Tuples with no matches are *dangling tuples*



```
SELECT sName, sID, cName, major  
FROM Student LEFT OUTER JOIN Apply using(sID);
```

LEFT JOIN

abbreviation

Left outer join

```
SELECT sName, sID, cName, major  
FROM Student INNER JOIN Apply  
using(sID);
```

sName	sID	cName	major
Amy	123	Cornell	EE
Amy	123	Berkeley	CS
Amy	123	Stanford	EE
Amy	123	Stanford	CS
Bob	234	Berkeley	biology
Craig	345	Cornell	EE
Craig	345	Cornell	CS
Craig	345	Cornell	bioengineering
Craig	345	MIT	bioengineering
Fay	678	Stanford	history
...

Students who have applied somewhere

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

```
SELECT sName, sID, cName, major  
FROM Student LEFT JOIN Apply  
using(sID);
```

sName	sID	cName	major
Amy	123	Cornell	EE
Amy	123	Berkeley	CS
Amy	123	Stanford	EE
Amy	123	Stanford	CS
Bob	234	Berkeley	biology
Craig	345	Cornell	EE
Craig	345	Cornell	CS
Craig	345	Cornell	bioengineering
Craig	345	MIT	bioengineering
Doris	456	NULL	NULL
...

Students who have applied somewhere plus students who haven't yet applied anywhere

Natural left outer join

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

```
SELECT sName, sID, cName, major
FROM Student NATURAL LEFT JOIN Apply;
```

sName	sID	cName	major
Amy	123	Cornell	EE
Amy	123	Berkeley	CS
Amy	123	Stanford	EE
Amy	123	Stanford	CS
Bob	234	Berkeley	biology
Craig	345	Cornell	EE
Craig	345	Cornell	CS
Craig	345	Cornell	bioengineering
Craig	345	MIT	bioengineering
Doris	456	NULL	NULL
...

Left outer join

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

How to rewrite the left outer join without using it?

```
SELECT sName, Student.sID, cName, major  
FROM Student, Apply  
Where Student.sID=Apply.sID  
UNION  
SELECT sName, sID, NULL, NULL  
FROM Student  
WHERE sID NOT IN (select sID from Apply)
```

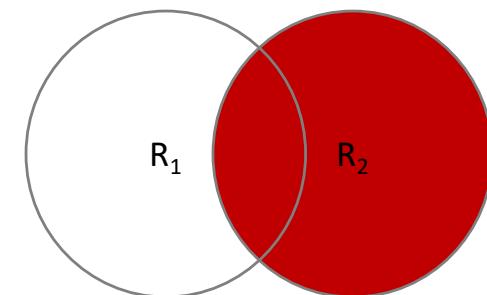
sName	sID	cName	major
Amy	123	Cornell	EE
Amy	123	Berkeley	CS
Amy	123	Stanford	EE
Amy	123	Stanford	CS
Bob	234	Berkeley	biology
Craig	345	Cornell	EE
Craig	345	Cornell	CS
Craig	345	Cornell	bioengineering
Craig	345	MIT	bioengineering
Doris	456	NULL	NULL
...

Right outer join

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

Takes any tuples on the right side and if they don't have a match on a tuple from the left, it is still added to the result and padded with NULL values

We can also use the left outer join for the same effect swapping the order of the relations



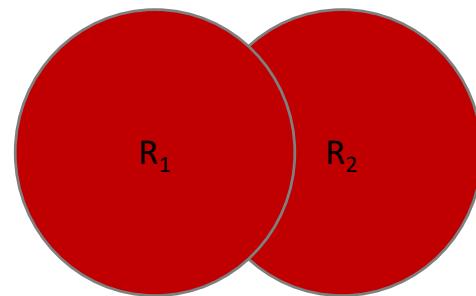
```
SELECT sName, sID, cName, major
FROM Student RIGHT OUTER JOIN Apply using(sID);
```

Full outer join

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

To include unmatched tuples from both sides of a join

```
SELECT sName, sID, cName, major
FROM Student FULL OUTER JOIN Apply using(sID);
```



Full outer join

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

How can we express a full outer join without using it?

```
SELECT sName, Student.sID, cName, major
```

```
FROM Student LEFT JOIN Apply using(sID)
```

```
UNION
```



Automatically
eliminates duplicates

```
SELECT sName, Student.sID, cName, major
```

```
FROM Student RIGHT JOIN Apply using(sID);
```

Full outer join

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

How can we express a full outer join without using joins?

```
SELECT sName, Student.sID, cName, major
```

```
FROM Student, Apply
```

```
Where Student.sID=Apply.sID
```

```
UNION
```

```
SELECT sName, sID, NULL, NULL
```

```
FROM Student
```

```
WHERE sID NOT IN (select sID from Apply)
```

```
UNION
```

```
SELECT NULL, sID, cName, major
```

```
FROM Apply
```

```
WHERE sID NOT IN (select sID from Student)
```

Outer joins and commutativity

Commutativity $(A \text{ op } B) = (B \text{ op } A)$

Left and Right Outer Joins are not commutative

Full Outer Join is commutative

Outer joins and associativity

Associativity $(A \text{ op } B) \text{ op } C = A \text{ op } (B \text{ op } C)$

T1	A	B	T2	B	C
	1	2		2	3

SELECT A, B, C

FROM (T1 natural full outer join T2) natural full outer join T3;



A	B	C
1	2	3
4	NULL	5

SELECT A, B, C

FROM T1 natural full outer join (T2 natural full outer join T3);



A	B	C
4	NULL	5
NULL	2	3
1	2	NULL

Left and right outer joins are not associative either

Outer joins summary

Left outer join

Include the left tuple even if there's no match

Right outer join

Include the right tuple even if there's no match

Full outer join

Include the both left and right tuples even if there's no match

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 6.1 – Simple Queries in SQL

Section 6.2 – Queries Involving More Than One Relation

Section 6.3 - Subqueries

Section 6.4 – Full-Relation Operations

Section 6.5 – Database Modifications

Philip Greenspun, SQL for Web Nerds,
<http://philip.greenspun.com/sql/>

SQL – Data Manipulation Language

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Agenda

~~Introduction~~

~~The JOIN family of operators~~

~~Basic SQL Statement~~

~~Aggregation~~

~~Table Variables and Set Operators~~

~~Null values~~

~~Subqueries in WHERE clauses~~

~~Data Modification statements~~

~~Subqueries in FROM and SELECT clauses~~

Aggregation

Perform aggregation over sets of values in multiple rows

Basic functions

min, max, sum, avg, count

Except *count*, all aggregations apply to a single attribute

SELECT A₁, A₂, ..., A_n

FROM R₁, R₂, ..., R_m

WHERE condition

GROUP BY columns

HAVING condition

New clauses

A first aggregation query

```
SELECT *
FROM Student;
```

sID	sName	GPA	HS
123	Amy	3.9	1000
234	Bob	3.6	1500
345	Craig	3.5	500
456	Doris	3.9	1000
567	Edward	2.9	2000
678	Fay	3.8	200
789	Gary	3.4	800
987	Helen	3.7	800
876	Irene	3.9	400
765	Jay	2.9	1500
654	Amy	3.9	1000
543	Craig	3.4	2000

```
SELECT avg(GPA)
FROM Student;
```

avg(GPA)
3.566

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

A second aggregation query

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

```
SELECT *
FROM Student, Apply
WHERE Student.sID =
Apply.sID and major='CS';
```



sID	sName	GPA	HS	sID1	cName	major	dec
123	Amy	3.9	1000	123	Stanford	CS	Y
123	Amy	3.9	1000	123	Berkeley	CS	Y
345	Craig	3.5	500	345	Cornell	CS	Y
987	Helen	3.7	800	987	Stanford	CS	Y
987	Helen	3.7	800	987	Berkeley	CS	Y
876	Irene	3.9	400	876	Stanford	CS	N
543	Craig	3.4	2000	543	MIT	CS	N

```
SELECT min(GPA)
FROM Student, Apply
WHERE Student.sID = Apply.sID
and major='CS';
```



min(GPA)
3.4

Lowest GPA of
students applying
to CS

A third aggregation query

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

```
SELECT *  
FROM Student, Apply  
WHERE Student.sID =  
Apply.sID and major='CS';
```



sID	sName	GPA	HS	sID1	cName	major	dec
123	Amy	3.9	1000	123	Stanford	CS	Y
123	Amy	3.9	1000	123	Berkeley	CS	Y
345	Craig	3.5	500	345	Cornell	CS	Y
987	Helen	3.7	800	987	Stanford	CS	Y
987	Helen	3.7	800	987	Berkeley	CS	Y
876	Irene	3.9	400	876	Stanford	CS	N
543	Craig	3.4	2000	543	MIT	CS	N

```
SELECT avg(GPA)  
FROM Student, Apply  
WHERE Student.sID = Apply.sID  
and major='CS';
```



avg(GPA)
3.71

Average GPA of
students applying
to CS?

A third aggregation query

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

```
SELECT *  
FROM Student  
WHERE sID in (select sID from  
Apply where major='CS');
```

sID	sName	GPA	HS
123	Amy	3.9	1000
345	Craig	3.5	500
987	Helen	3.7	800
876	Irene	3.9	400
543	Craig	3.4	2000

```
SELECT avg(GPA)  
FROM Student  
WHERE sID in (select sID from  
Apply where major='CS');
```

avg(GPA)
3.68

Average GPA of students
applying to CS

A first count query

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

```
SELECT *
FROM College
WHERE enr > 15000;
```

```
SELECT count(*)
FROM College
WHERE enr > 15000;
```

cName	state	enr
Berkeley	CA	36000
Cornell	NY	21000

count(*)
2

Number of colleges
bigger than 15000

A second count query

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

```
SELECT count(sID)  
FROM Apply  
WHERE cName = 'Cornell';
```

```
SELECT count(distinct sID)  
FROM Apply  
WHERE cName = 'Cornell';
```

count(*)

6

Number of students
applying to Cornell?



Number of applications
to Cornell

count(distinct sID)

3

Number of students
applying to Cornell



COUNT applies to duplicates,
unless otherwise stated

A third count query

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

```
SELECT *
FROM   Student S1
WHERE (select count(*) from Student S2
       where S2.sID<>S1.sID and S2.GPA = S1.GPA) =
       (select count(*) from Student S2
       where S2.sID <> S1.sID and S2.sizeHS = S1.sizeHS);
```

What does it compute?

Students such that number of other students with same GPA is equal to number of other students with same HS size

A fourth count query

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

```
SELECT CS.avgGPA – NonCS.avgGPA
FROM   (select avg(GPA) as avgGPA from Student
        where sID in (
          select sID from Apply where major = 'CS')) as CS,
        (select avg(GPA) as avgGPA from Student
        where sID not in (
          select sID from Apply where major = 'CS')) as nonCS;
```

What does it compute?

**CS.avgGPA-
NonCS.avgGPA**

0.19

Amount by which average GPA of students applying to CS exceeds average of students not applying to CS

A fourth count query

Compute the “amount by which average GPA of students applying to CS exceeds average of students not applying to CS” using subqueries in the select clause

```
SELECT (select avg(GPA) from Student  
        where sID in (  
                      select sID from Apply where major = 'CS')) -  
       (select avg(GPA) from Student  
        where sID not in (  
                      select sID from Apply where major = 'CS')) as d  
FROM Student;
```

d
0.19
0.19
0.19
0.19
...



Duplicates:
difference is
computed for
each student

A fourth count query

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

Compute the “amount by which average GPA of students applying to CS exceeds average of students not applying to CS” using subqueries in the select clause

```
SELECT DISTINCT (select avg(GPA) as avgGPA from Student
```

```
where sID in (
```

```
    select sID from Apply where major = 'CS')) -
```

```
(select avg(GPA) as avgGPA from Student
```

d

0.19

```
where sID not in (
```

```
    select sID from Apply where major = 'CS')) as d
```

```
FROM Student;
```

Another example

```
SELECT SUM(price * quantity)  
FROM Purchase  
WHERE product = 'cake'
```



Purchase

Product	Date	Price	Quantity
cake	21/10	1	20
banana	3/10	0.5	10
banana	10/10	1	10
cake	25/10	1.50	20

Sum(price*quantity)

50

(= 1*20 + 1.50*20)

College(cName, state, enr)

Student(sID, sName, GPA, sizeHS)

Apply(sID, cName, major, decision)

Group by clause

Only used in conjunction with aggregation

It partitions a relation by values of a given attribute or set of attributes

Semantics of the query

1. Compute the FROM and WHERE clauses
2. Group by the attributes in the GROUP BY
3. Compute the SELECT clause: grouped attributes and aggregates

SELECT	S
FROM	R_1, \dots, R_n
WHERE	C_1
GROUP BY	a_1, \dots, a_k

Group by clause

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

```
SELECT *
FROM Apply
ORDER BY cName;
```

```
SELECT cName, count(*)
FROM Apply
GROUP BY cName;
```

1. FROM and WHERE

sID	cName	major	dec
123	Berkeley	CS	Y
234	Berkeley	biology	N
987	Berkeley	CS	Y

2. Group By



3. Select

cName	count(*)
Berkeley	3
Cornell	6
MIT	4
Stanford	6

sID	cName	major	dec
123	Berkeley	EE	Y
345	Cornell	bio	N
...
123	Berkeley	CS	Y
234	Berkeley	biology	N
987	Berkeley	CS	Y
123	Cornell	EE	Y
345	Cornell	bioengineering	N
...

A second query with group by

```
SELECT *  
FROM College  
ORDER BY state;
```

```
SELECT state, sum(enr)  
FROM College  
GROUP BY state;
```

cName	state	enr
Stanford	CA	15000
Berkeley	CA	36000
MIT	MA	10000
Cornell	NY	21000

state	sum(enr)
CA	51000
MA	10000
NY	21000

College enrollments by state

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

A third query with group by

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

```
SELECT cName, major, GPA
FROM Student, Apply
WHERE Student.sID = Apply.sID
ORDER BY cName, major;
```

```
SELECT cName, major, min(GPA),
max(GPA)
FROM Student, Apply
WHERE Student.sID = Apply.sID
GROUP BY cName, major;
```

cName	major	GPA
Berkeley	biology	3.6
Berkeley	CS	3.9
Berkeley	CS	3.7
Cornell	bioengineering	3.5
...

cName	major	min(GPA)	max(GPA)
Berkeley	biology	3.6	3.6
Berkeley	CS	3.7	3.9
Cornell	bioengineering	3.5	3.5
...

Minimum and maximum GPAs of applicants to each college and major

A fourth query with group by

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

Find the largest spread of GPAs in colleges and majors

Step 1: Find the spread of GPAs of applicants for each college and major

```
SELECT mx-mn
FROM ( SELECT cName, major, min(GPA) as mn, max(GPA) as mx
        FROM Student, Apply
       WHERE Student.sID = Apply.sID
      GROUP BY cName, major ) M;
```

mx-mn
0
0.19
0
...

A fourth query with group by

Find the largest spread of GPAs in colleges and majors

Step 2: Find the largest spread

```
SELECT max(mx-mn)
FROM ( SELECT cName, major, min(GPA) as mn, max(GPA) as mx
      FROM Student, Apply
      WHERE Student.sID = Apply.sID
      GROUP BY cName, major ) M;
```

max(mx-mn)

0.899

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

A fifth query with group by

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

```
SELECT Student.sID, cName  
FROM Student, Apply  
WHERE Student.sID = Apply.sID  
ORDER BY Student.sID;
```

```
SELECT Student.sID, count(distinct  
cName)  
FROM Student, Apply  
WHERE Student.sID = Apply.sID  
GROUP BY Student.sID;
```

sID	cName
123	Stanford
123	Stanford
123	Berkeley
123	Cornell
234	Berkeley
...	...

sID	Count(distinct cName)
123	3
234	1
345	2
...	...

Number of colleges applied to
by each student

A fifth query with group by

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

What if we also want the student's name?

```
SELECT Student.sID, sName, count(distinct cName)  
FROM Student, Apply  
WHERE Student.sID = Apply.sID  
GROUP BY Student.sID;
```

It only worked because for
each sID we only have one
sName



sID	sName	count(distinct cName)
123	Amy	3
234	Bob	1
345	Craig	2
...		...

A fifth query with group by

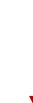
```
College(cName, state, enr)  
Student(sID, sName, GPA, sizeHS)  
Apply(sID, cName, major, decision)
```

```
SELECT Student.sID, sName, count(distinct cName), cName  
FROM Student, Apply  
WHERE Student.sID = Apply.sID  
GROUP BY Student.sID;
```

sID	sName	count(distinct cName)	cName
123	Amy	3	Stanford
234	Bob	1	Berkeley
345	Craig	2	MIT
...	



It chooses a random value from the group to include



Some systems throw an error in this situation

A fifth query with group by

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

```
SELECT Student.sID, count(distinct cName)
```

```
FROM Student, Apply
```

```
WHERE Student.sID = Apply.sID
```

```
GROUP BY Student.sID;
```

sID	Count(distinct cName)
123	3
234	1
345	2
...	...



What if want to list students who haven't applied anywhere with a 0 in the count?

A fifth query with group by

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

SELECT Student.sID, count(distinct cName)

FROM Student, Apply

WHERE Student.sID = Apply.sID

GROUP BY Student.sID

UNION

SELECT sID, 0

FROM Student

WHERE sID not in (select sID from Apply);

sID	Count(distinct cName)
876	2
987	2
456	0
567	0
...	...

Having clause

Applies conditions to the results of aggregate functions

Check conditions that involve the entire group

Where clause applies to one tuple at a time

Applied after the GROUP BY clause

SELECT	S	→ attributes a_1, \dots, a_k and/or aggregates over other attributes
FROM	R_1, \dots, R_n	
WHERE	C_1	→ any condition on the attributes in R_1, \dots, R_n
GROUP BY	a_1, \dots, a_k	
HAVING	C_2	→ any condition on the aggregate expressions

Having clause

List the colleges with fewer than 5 applications

```
SELECT cName  
FROM Apply  
GROUP BY cName  
HAVING count(*) < 5;
```

cName
Berkeley
MIT

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

Having clause

What if we're interested in the colleges that have fewer than 5 applicants

```
SELECT cName  
FROM Apply  
GROUP BY cName  
HAVING count(distinct sID) < 5;
```

cName
Berkeley
Cornell
MIT

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

A final having clause

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

```
SELECT major
FROM Student, Apply
WHERE Student.sID = Apply.sID
GROUP BY major
HAVING max(GPA) < (SELECT avg(GPA) FROM Student);
```

major
bioengineering
psychology

What does it compute?

Majors whose applicant's maximum GPA is below the average

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 6.1 – Simple Queries in SQL

Section 6.2 – Queries Involving More Than One Relation

Section 6.3 - Subqueries

Section 6.4 – Full-Relation Operations

Section 6.5 – Database Modifications

Philip Greenspun, SQL for Web Nerds,
<http://philip.greenspun.com/sql/>

SQL – Data Manipulation Language

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Group-by versus subqueries

```
SELECT cName, count(*) AS cnt  
FROM Apply  
GROUP BY cName;
```

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

```
SELECT DISTINCT cName,  
          (SELECT count(*))  
          FROM Apply A2  
          WHERE A2.cName = A1.cName) AS cnt  
          FROM Apply A1;
```

cName	cnt
Berkeley	3
Cornell	6
MIT	4
Stanford	6

Group-by versus subqueries

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

```
SELECT cName  
FROM Apply
```



No need for DISTINCT: automatically
from GROUP BY

```
GROUP BY cName  
HAVING count(*) < 5;
```

cName
Berkeley
MIT

```
SELECT DISTINCT cName  
FROM Apply A1  
WHERE 5 > (SELECT count(*) FROM Apply A2 WHERE A2.cName =  
A1.cName);
```

Every group by and having query can be written without using those clauses

Group-by versus subqueries

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

```
SELECT cName
FROM Apply
GROUP BY cName
HAVING count(*) < 5;
```

```
SELECT DISTINCT cName
FROM Apply A1
WHERE 5 > (SELECT count(*)
             FROM Apply A2
             WHERE A2.cName = A1.cName);
```

This is SQL by an expert

This is SQL by a novice

Which way is more efficient?

How many times do we do a SFW query in each case?
With GROUP BY can be much more efficient!

Aggregation summary

SELECT	S	→ attributes a_1, \dots, a_k and/or aggregates over other attributes
FROM	R_1, \dots, R_n	
WHERE	C_1	→ any condition on the attributes in R_1, \dots, R_n
GROUP BY	a_1, \dots, a_k	
HAVING	C_2	→ any condition on the aggregate expressions

Evaluation steps

1. Evaluate FROM-WHERE: apply condition C_1 on the attributes in R_1, \dots, R_n
2. GROUP BY the attributes a_1, \dots, a_k
3. Apply condition C_2 to each group
4. Compute aggregates in S and return the result

Agenda

~~Introduction~~

~~The JOIN family of operators~~

~~Basic SQL Statement~~

~~Aggregation~~

~~Table Variables and Set Operators~~

~~Null values~~

~~Subqueries in WHERE clauses~~

~~Data Modification statements~~

~~Subqueries in FROM and SELECT clauses~~

NULL values

Unless specified otherwise, any value in an attribute can take on the special value NULL

NULL usually means that the value is undefined or unknown or not applicable

We will see what happens when we have NULL values and we run queries over the database

NULL values for numerical operations

NULL -> NULL

If $x = \text{NULL}$ then

$4^*(3-x)/7$ is NULL

NULL values for Boolean operations

Conditions are evaluated using a three value logic:
TRUE, FALSE, UNKNOWN

If $x = \text{NULL}$ then $x = \text{'Joe'}$ is UNKNOWN

Considering TRUE is 1.0, UNKNOWN is 0.5 and FALSE is 0.0

$C_1 \text{ AND } C_2 = \min(C_1, C_2)$

$C_1 \text{ OR } C_2 = \max(C_1, C_2)$

$\text{NOT } C_1 = 1 - C_1$

Rule in SQL: include only tuples that yield TRUE (1.0)

A first query with NULL values

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

```
SELECT sID, sName, GPA  
FROM Student  
WHERE GPA > 3.5;
```



sID	sName	GPA
123	Amy	3.9
234	Bob	3.6
456	Doris	3.9
678	Fay	3.8
987	Helen	3.7
876	Irene	3.9
654	Amy	3.9

2 additional students

Student			
sID	sName	GPA	HS
123	Amy	3.9	1000
234	Bob	3.6	1500
345	Craig	3.5	500
456	Doris	3.9	1000
567	Edward	2.9	2000
678	Fay	3.8	200
789	Gary	3.4	800
987	Helen	3.7	800
876	Irene	3.9	400
765	Jay	2.9	1500
654	Amy	3.9	1000
543	Craig	3.4	2000
432	Kevin	NULL	1500
321	Lori	NULL	2500

A second query with NULL values

```
SELECT sID, sName, GPA  
FROM Student  
WHERE GPA <= 3.5;
```

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

sID	sName	GPA
345	Craig	3.5
567	Edward	2.9
789	Gary	3.4
765	Jay	2.9
543	Craig	3.4

College(cName, state, enr)

Student(sID, sName, GPA, sizeHS)

Apply(sID, cName, major, decision)

A third query with NULL values

SELECT sID, sName, GPA

FROM Student

WHERE GPA > 3.5 OR GPA <= 3.5;



Tautology

Even with a tautology, we might not get all the data

<u>sID</u>	<u>sName</u>	GPA
123	Amy	3.9
234	Bob	3.6
345	Craig	3.5
456	Doris	3.9
567	Edward	2.9
678	Fay	3.8
789	Gary	3.4
987	Helen	3.7
876	Irene	3.9
765	Jay	2.9
654	Amy	3.9
543	Craig	3.4

College(cName, state, enr)

Student(sID, sName, GPA, sizeHS)

Apply(sID, cName, major, decision)

A fourth query with NULL values

To make this query return all the students

```
SELECT sID, sName, GPA  
FROM Student  
WHERE GPA > 3.5 OR GPA <= 3.5 OR  
GPA IS NULL;
```

Can test for NULL explicitly

attribute IS NULL

attribute IS NOT NULL

sID	sName	GPA
123	Amy	3.9
234	Bob	3.6
345	Craig	3.5
456	Doris	3.9
567	Edward	2.9
678	Fay	3.8
789	Gary	3.4
987	Helen	3.7
876	Irene	3.9
765	Jay	2.9
654	Amy	3.9
543	Craig	3.4
432	Kevin	
321	Lori	

A fifth query with NULL values

```
SELECT sID, sName, GPA, HS  
FROM Student  
WHERE GPA > 3.5 OR HS < 1600;
```

Although Kevin has a NULL GPA, he is retrieved because it has a HS<1600

sID	sName	GPA	HS
123	Amy	3.9	1000
234	Bob	3.6	1500
345	Craig	3.5	500
456	Doris	3.9	1000
678	Fay	3.8	200
789	Gary	3.4	800
987	Helen	3.7	800
876	Irene	3.9	400
765	Jay	2.9	1500
654	Amy	3.9	1000
432	Kevin		1500

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

NULL values and aggregate functions

```
SELECT distinct GPA  
FROM Student;
```

GPA
NULL
2.9
3.4
3.5
3.6
3.7
3.8
3.9

8 tuples

```
SELECT count(distinct GPA)  
FROM Student;
```

count(distinct GPA)
7

When counting (the distinct) values, NULLs are not included

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

NULL values

When using a database with NULL values

Be careful when writing queries

Understand how the NULL values are going to influence the result

Agenda

~~Introduction~~

~~The JOIN family of operators~~

~~Basic SQL Statement~~

~~Aggregation~~

~~Table Variables and Set Operators~~

~~Null values~~

~~Subqueries in WHERE clauses~~

Data Modification statements

~~Subqueries in FROM and SELECT clauses~~

Inserting new data

Insert Into Table (A_1, A_2, \dots, A_n)

Values (v_1, v_2, \dots, v_n)



If we provide values for all attributes, we may omit the list of attributes.

Insert Into Table

Select-Statement



Same schema as the table

Inserting new data

Insert Into College

Values ('Carnegie Mellon', 'PA', 11500);

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

College

cName	state	enr
Stanford	CA	15000
Berkeley	CA	36000
MIT	MA	10000
Cornell	NY	21000
Carnegie Mellon	PA	11500

Inserting new data

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

Have all students who didn't apply anywhere apply to CS at Carnegie Mellon

```
SELECT *  
FROM Student  
WHERE sID not in (select sID FROM Apply);
```

sID	sName	GPA	HS
456	Doris	3.9	1000
567	Edward	2.9	2000
789	Gary	3.4	800
654	Amy	3.9	1000

```
INSERT INTO Apply  
SELECT sID, 'Carnegie Mellon', 'CS', NULL  
FROM Student  
WHERE sID not in (select sID FROM Apply);
```

Apply			
sID	cName	major	dec
...
543	MIT	CS	N
456	Carnegie Mellon	CS	NULL
567	Carnegie Mellon	CS	NULL
789	Carnegie Mellon	CS	NULL
654	Carnegie Mellon	CS	NULL

Inserting new data

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

Admit to Carnegie Mellon EE all students who were turned down in EE elsewhere

```
SELECT *  
FROM Student
```

sID	sName	GPA	HS
123	Amy	3.9	1000
345	Craig	3.5	500

```
WHERE sID in (select sID FROM Apply WHERE major='EE' AND  
decision='N' AND cName<>'Carnegie Mellon');
```

```
INSERT INTO Apply
```

```
SELECT sID, 'Carnegie Mellon', 'EE', 'Y'  
FROM Student
```

Apply

sID	cName	major	dec
...
654	Carnegie Mellon	CS	NULL
123	Carnegie Mellon	EE	Y
345	Carnegie Mellon	EE	Y

```
WHERE sID in (select sID FROM Apply WHERE major='EE' AND  
decision='N' AND cName<>'Carnegie Mellon');
```

Deleting existing data

Delete From **Table**

Where **Condition**

Condition can be complicated

Can include subqueries and aggregation over other tables

Deleting existing data

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

Delete all students who applied to more than two different majors

```
SELECT sID
```

```
FROM Apply
```

```
GROUP BY sID
```

```
HAVING count(distinct major) > 2;
```

```
DELETE FROM Student
```

```
WHERE sID in (
```

```
SELECT sID FROM Apply GROUP BY sID
```

```
HAVING count(distinct major) > 2);
```

sID
345
876

Student

sID	sName	GPA	HS
123	Amy	3.9	1000
234	Bob	3.6	1500
456	Doris	3.9	1000
567	Edward	2.9	2000
678	Fay	3.8	200
789	Gary	3.4	800
987	Helen	3.7	800
765	Jay	2.9	1500
654	Amy	3.9	1000
543	Craig	3.4	2000

Deleting existing data

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

Delete those students also from Apply

```
DELETE FROM Apply
```

```
WHERE sID in (
```

```
    SELECT sID FROM Apply GROUP BY sID
```

```
    HAVING count(distinct major) > 2);
```

Not all database systems allow deletion commands where the subquery includes the same relation that you're deleting from

In systems that don't allow, a temporary table would have to be created

Deleting existing data

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

Delete colleges with no CS applicants

```
SELECT *
```

```
FROM College
```

```
WHERE cName not in
```

```
(SELECT cName from Apply where major='CS');
```

cName	state	enr
Cornell	NY	21000

```
DELETE FROM College
```

```
WHERE cName not in
```

```
(SELECT cName from Apply where major='CS');
```

cName	state	enr
Stanford	CA	15000
Berkeley	CA	36000
MIT	MA	10000
Carnegie Mellon	PA	11500

College

Updating existing data

Update **Table**

Set **Attr = Expression**

Where Condition

Update **Table**

Set **A₁ = Expr₁, A₂ = Expr₂, ..., A_n = Expr_n**

Where Condition

Conditions and expressions can include subqueries and queries over other tables or the same table

Updating existing data

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

Accept applicants to Carnegie Mellon with GPA<3.6 but turn them into economics majors

```
SELECT *
```

```
FROM Apply
```

```
WHERE cName = 'Carnegie Mellon' and sID in  
(SELECT sID from Student where GPA<3.6);
```

sID	cName	major	dec
567	Carnegie Mellon	CS	
789	Carnegie Mellon	CS	

```
UPDATE Apply
```

```
SET decision = 'Y', major='economics'
```

```
WHERE cName = 'Carnegie Mellon' and sID in  
(SELECT sID from Student where GPA<3.6);
```

Apply

sID	cName	major	dec
...
654	Carnegie Mellon	CS	NULL
123	Carnegie Mellon	EE	Y
567	Carnegie Mellon	economics	Y
789	Carnegie Mellon	economics	Y

Updating existing data

Turn the highest GPA EE applicant into a CSE applicant

SELECT *

FROM Apply

WHERE major='EE' AND sID in

(select sID from Student where GPA>=all

(select GPA from Student where sID in

(select sID from Apply where major='EE'));

sID	cName	major	dec
123	Stanford	EE	N
123	Cornell	EE	Y
123	Carnegie Mellon	EE	Y

College(cName, state, enr)

Student(sID, sName, GPA, sizeHS)

Apply(sID, cName, major, decision)

Updating existing data

Turn the highest GPA EE applicant into a CSE applicant

UPDATE Apply

SET major = 'CSE'

WHERE major='EE' AND sID in

(select sID from Student where GPA>=all

(select GPA from Student where sID in

(select sID from Apply where major='EE'));

sID	cName	major	dec
...
789	Carnegie Mellon	economics	Y
123	Stanford	CSE	N
123	Cornell	CSE	Y
123	Carnegie Mellon	CSE	Y

Apply

College(cName, state, enr)

Student(sID, sName, GPA, sizeHS)

Apply(sID, cName, major, decision)

Updating existing data

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

Give every student the highest GPA and the smallest high school in the database

UPDATE Student

SET GPA = (select max(GPA) from Student),
sizeHS = (select min(sizeHS) from
Student);

In the SET command, the right-hand side of
the equals can itself be a subquery

Student

sID	sName	GPA	HS
123	Amy	3.9	200
234	Bob	3.9	200
456	Doris	3.9	200
567	Edward	3.9	200
678	Fay	3.9	200
789	Gary	3.9	200
987	Helen	3.9	200
765	Jay	3.9	200
654	Amy	3.9	200
543	Craig	3.9	200

Summary

SQL is a rich programming language that handles the way data is processed declaratively

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 6.1 – Simple Queries in SQL

Section 6.2 – Queries Involving More Than One Relation

Section 6.3 - Subqueries

Section 6.4 – Full-Relation Operations

Section 6.5 – Database Modifications

Philip Greenspun, SQL for Web Nerds,
<http://philip.greenspun.com/sql/>

SQL – Views

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

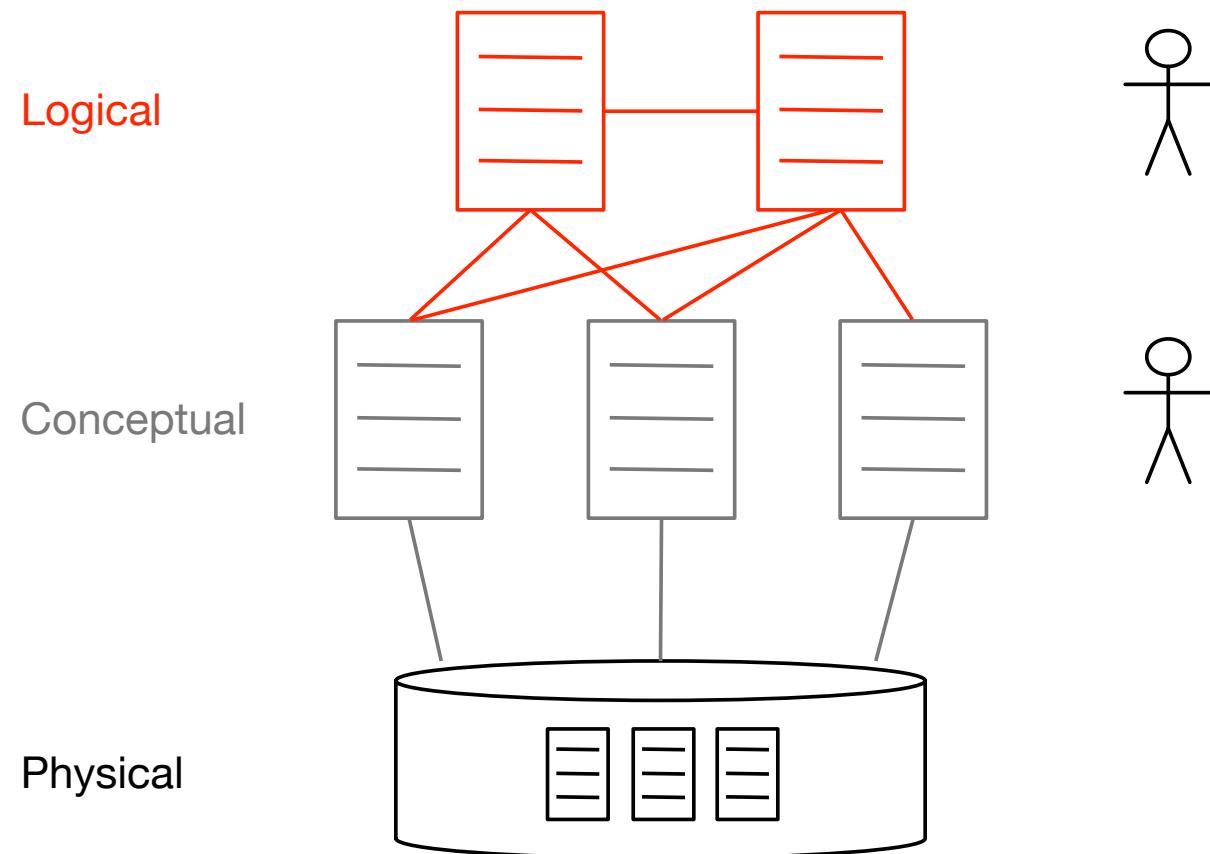
Agenda

Defining and Using Views

Views Modifications Introduction

Views Modification Using Triggers

Three-level vision of database



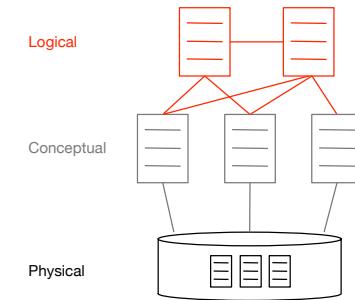
Why use views?

Hide some data from some users

Make some queries easier / more natural

Modularity of database access

Real applications tend to use lots and lots of views



Defining and using views

View $V = \text{ViewQuery}(R_1, R_2, \dots, R_n)$

Schema of V is schema of query result

Query Q involving V , conceptually:

$V := \text{ViewQuery}(R_1, R_2, \dots, R_n)$

Evaluate Q

In reality, Q rewritten to use R_1, \dots, R_n instead of V



Performed
automatically
by the DBMS

R_i could itself be a view

SQL Syntax

Create View **Vname** As

<Query>

Create View **Vname(A₁,A₂,...,A_n)** As

<Query>

Once the view is created, it can be used as if it is a regular table

College Admission Database

Apply

sID	cName	major	dec
123	Stanford	CS	Y
123	Stanford	EE	N
123	Berkeley	CS	Y
123	Cornell	EE	Y
234	Berkeley	biology	N
345	MIT	bioengineering	Y
345	Cornell	bioengineering	N
345	Cornell	CS	Y
345	Cornell	EE	N
678	Stanford	history	Y
987	Stanford	CS	Y
987	Berkeley	CS	Y
876	Stanford	CS	Y
876	MIT	biology	Y
876	MIT	marine biology	N
765	Stanford	history	Y
765	Cornell	history	N
765	Cornell	psychology	Y
543	MIT	CS	N

College

cName	state	enr
Stanford	CA	15000
Berkeley	CA	36000
MIT	MA	10000
Cornell	NY	21000

Student

sID	sName	GPA	HS
123	Amy	3.9	1000
234	Bob	3.6	1500
345	Craig	3.5	500
456	Doris	3.9	1000
567	Edward	2.9	2000
678	Fay	3.8	200
789	Gary	3.4	800
987	Helen	3.7	800
876	Irene	3.9	400
765	Jay	2.9	1500
654	Amy	3.9	1000
543	Craig	3.4	2000

View 1

Create View CSaccept As

Select sID, cName

From Apply

Where major = 'CS' and dec = 'Y';

Select *
From CSaccept

The diagram illustrates the execution flow of a database query. It starts with the view definition 'Create View CSaccept As' (in red), followed by its components: 'Select sID, cName', 'From Apply', and 'Where major = 'CS' and dec = 'Y''. An arrow points from this definition to a table showing student IDs and their corresponding university names. Another arrow points from the table to the final result, which is a query 'Select * From CSaccept'. This visualizes how the database system rewrites the query based on the view's definition.

sID	cName
123	Stanford
123	Berkeley
345	Cornell
987	Stanford
987	Berkeley

View is actually not stored, query is rewritten based on the view definition

A query using View 1

Select Student.sID, sName, GPA

From Student, CSaccept

Where Student.sID = Csaccept.sID and cName = 'Stanford' and
GPA < 3.8;

slD	sName	gpa
987	Helen	3.7

What happens when we run a query that refers to a view?

A query using View 1 – conceptual analysis

Create temporary **table T** as

Select sID, cName

From Apply

Where major = ‘CS’ and dec= ‘Y’;

sID	sName	gpa
987	Helen	3.7

Select Student.sID, sName, GPA

From Student, T

Where Student.sID = T.sID and cName = ‘Stanford’ and GPA < 3.8;

Drop table T;

A query using View 1 – practical analysis

Select Student.sID, sName, GPA

From Student,

(Select sID, cName

From Apply

Where major = 'CS' and dec= 'Y') as CSaccept

Where Student.sID = CSaccept.sID and cName = 'Stanford' and
GPA < 3.8;



Not how the systems
tend to do it either

sID	sName	gpa
987	Helen	3.7

A query using View 1 – practical analysis

Select Student.sID, sName, GPA

From Student, Apply

Where major = 'CS' and dec= 'Y' and Student.sID = Apply.sID and cName = 'Stanford' and GPA < 3.8;

sID	sName	gpa
987	Helen	3.7

View 2 – a view using other view

Create View CSberk As

Select Student.sID, sName, GPA

From Student, CSaccept

Where Student.sID = Csaccept.sID and cName = 'Berkeley' and HS > 500;

Select *

From CSberk



sID	sName	gpa
123	Amy	3.9
987	Helen	3.7

A query using View 2

Select *

From CSberk

Where GPA > 3.8;

sID	sName	gpa
123	Amy	3.9

What happens when we run a query that refers to a view?

A query using View 2 – rewrite process

select * from

(select Student.sID, sName, GPA

from Student, (select sID, cName from Apply

where major = 'CS' and dec = 'Y') as CSaccept

where Student.sID = CSaccept.sID and cName = 'Berkeley' and HS
> 500) Csberk

where GPA > 3.8;



What would be the
flattened rewrite of this
query?

sID	sName	gpa
123	Amy	3.9

View 2 – dropping View 1

drop view CSaccept;

What happens?

Postgres: Error - other objects depend on it

SQLite/MySQL: no error on drop; error when we refer to CSberk

View 3

Create View Mega As

Select College.cName, state, enr, Student.sID, sName, GPA, HS,
major, dec

From College, Student, Apply

Where College.cName = Apply.cName and Student.sID = Apply.sID;

Select *
From Mega;

A query using View 3

Select sID, sName, GPA, cName

From Mega

Where GPA > 3.5 and major = 'CS' and enr > 15000;

sID	sName	gpa	cName
123	Amy	3.9	Berkeley
987	Helen	3.7	Berkeley

View 3 rewritten

Select Student.sID, sName, GPA, College.cName

From College, Student, Apply

Where College.cName = Apply.cName and Student.sID = Apply.sID
and GPA > 3.5 and major = 'CS' and enr > 15000;

sID	sName	gpa	cName
123	Amy	3.9	Berkeley
987	Helen	3.7	Berkeley

Agenda

~~D~~efining and Using Views

Views Modifications Introduction

Views Modification Using Triggers

Modifying views

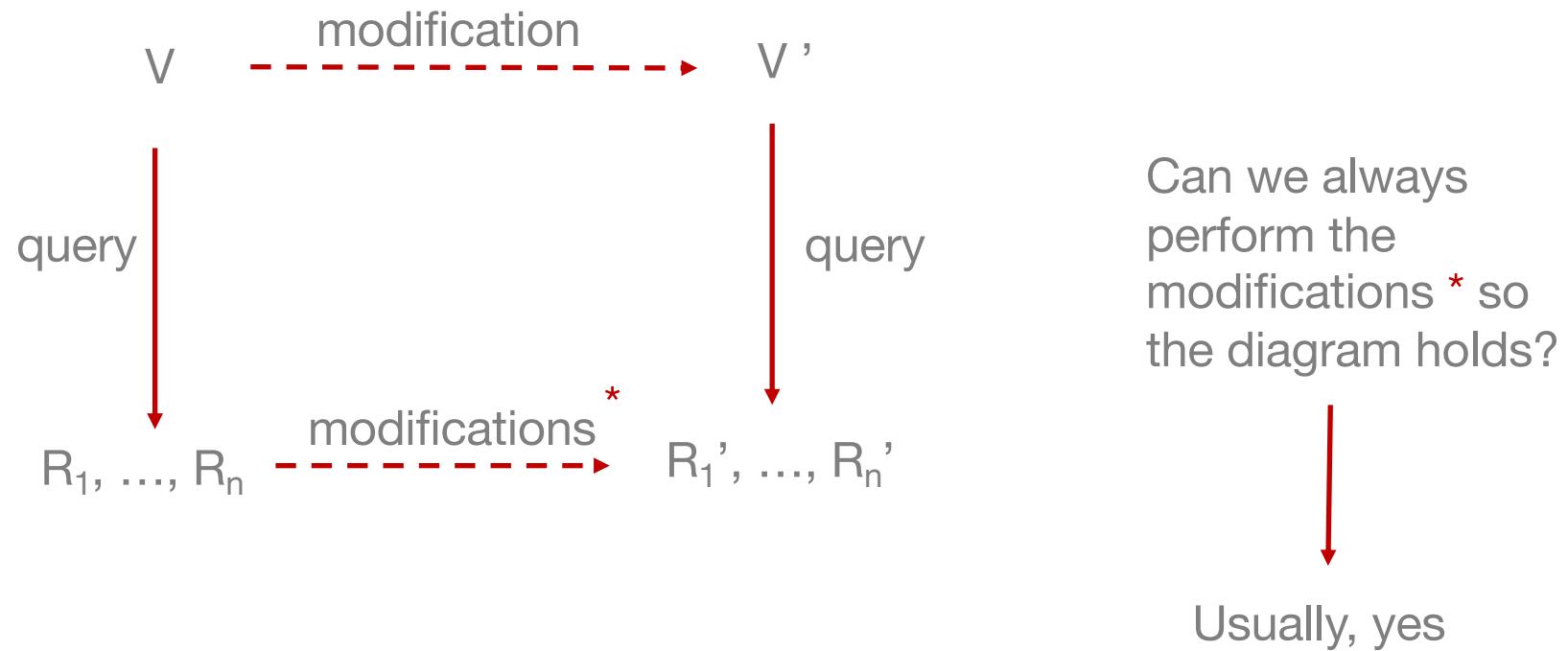
Once view V defined, can we modify V like any table ?

Doesn't make sense: V is not stored

Has to make sense: views are some users' entire “view” of the database

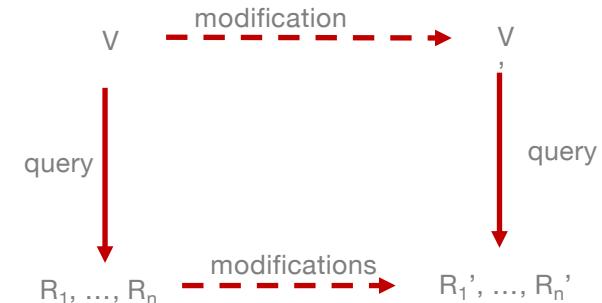
Solution: Modifications to V rewritten to modify base tables

Modifying base tables



Problems with modifying base tables

Often, there are many possible modifications



$R(A,B)$

(1,2)

$V = \pi_A(R)$

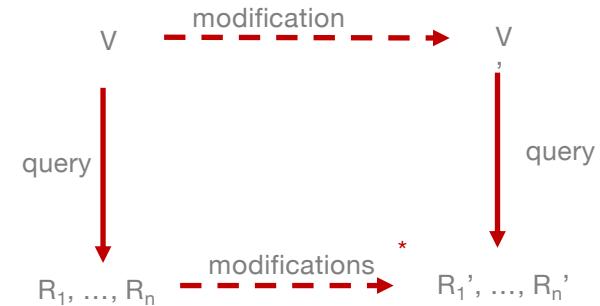
(1)

(3) \leftarrow insert

What modifications will be done on R ? What do we insert on B ?

Problems with modifying base tables

Often, there are many possible modifications



$R(N)$

$$V = avg(N)$$

(1)

(3) ← update to 7

(3)

(5)

What modifications will be done on R ? There are many options.

Unlike queries,
modifications cannot be automated in general

Modifying views

Rewriting process specified explicitly by view creator

- + Can handle all modifications Instead-of triggers
- No guarantee of correctness (or meaningful)

Restrict views + modifications so that translation to base table modifications is meaningful and unambiguous

- + No user intervention SQL standard
- Restrictions are significant

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 8.1 – Virtual Views

Section 8.2 – Modifying Views

SQL – Triggers

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Agenda

Introduction

Examples

Before and After

Insert, Delete, and Update

New and Old

Conditions and actions

Triggers enforcing constraints

Triggers

“Event-Condition-Action Rules”

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

Move monitoring logic from apps into DBMS

Enforce constraints

Beyond what constraint system supports

Automatic constraint “repair”

Implementations vary significantly

Introduction: SQL standard

Examples: SQLite

Triggers in SQL

Create Trigger **name**

Before | After | Instead Of **events**

[**referencing-variables**]

[For Each Row]

When (**condition**)

action

Triggers in SQL - events

insert on T

delete on T

update [C₁, ..., C_n] on T



columns are optional

Create Trigger **name**

Before | After | Instead Of **events**

[**referencing-variables**]

[For Each Row]

When (**condition**)

action

Triggers in SQL – [For Each Row]

Optional clause

States that the trigger is activated once for each modified tuple

Create Trigger **name**

Before | After | Instead Of **events**

[**referencing-variables**]

[For Each Row]

When (**condition**)

action

If, for example, a delete command deletes 10 tuples

With “for each row” -> trigger will run 10 times, once for each deleted tuple

Without “for each row” -> trigger runs once for the entire statement

The trigger is always activated at the end of the statement

Either x times or once if “for each row” is not present

Triggers in SQL – Referencing variables

To reference the data that was modified and caused the trigger to be activated

old row as var

new row as var

old table as var

new table as var

Create Trigger **name**

Before | After | Instead Of **events**

[**referencing-variables**]

[For Each Row]

When (**condition**)

action

New variables for inserts and updates

Old variables for deletes and updates

Triggers in SQL – Referencing variables

old row as var / new row as var
only for row-level triggers
refer to the specific affected tuple

old table as var / new table as var
for row-level or statement-level triggers
refer to the set of all the affected tuples

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

What referencing variables can we use in these cases?
Row-level delete
Statement-level insert
Row-level update

Triggers in SQL – condition and action

Condition

Tests the condition, if it is true, action will be performed

Like a general assertion

Action

SQL statement

Systems vary much here

set of simple statements + begin-end bracket

stored procedures

set of simple statements + begin/end

Create Trigger **name**

Before | After | Instead Of **events**

[**referencing-variables**]

[For Each Row]

When (**condition**)

action

Referential integrity – row-level trigger

R.A references S.B, cascaded delete

Create Trigger **Cascade**

After Delete On **S**

Referencing Old Row As **O**

For Each Row

[no condition]

Delete From **R** Where **A = O.B**

Referential integrity – statement-level trigger

R.A references S.B, cascaded delete

Create Trigger **Cascade**

After Delete On **S**

Referencing Old Row As **O** —————→

[For Each Row] —————→

[no condition]

Delete From **R** Where **A = O.B** —————→

Not the old state of the table, just the values of the tuples that have been deleted



Referencing **Old Table** As OT

Eliminate [For Each Row]

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

Row-level versus statement-level triggers

Create Trigger **Cascade**

After Delete On **S**

Referencing Old Row As **O**

For Each Row

[no condition]

Delete From **R** Where **A = O.B**

Create Trigger **Cascade**

After Delete On **S**

Referencing Old Table As **OT**

[For Each Row]

[no condition]

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

Which version to use?

In this case, probably the statement-level trigger would be more efficient

Some systems don't support both types of triggers -> no choice

Tricky issues

Row-level vs. Statement-level

New/Old Row and New/Old Table

Before, Instead Of

Multiple triggers activated at same time → Which goes first?

Trigger actions activating other triggers (chaining)

Also self-triggering, cycles, nested invocations

Conditions in When vs. as part of action

Implementations vary significantly

Row-level versus statement-level triggers

Create Trigger **IncreaseInserts**

$T(K, V)$ – K key, V value

After Insert On **T**

stable value, NT
is always the set
of inserted tuples

Referencing New Row As **NR**, New Table As **NT**

For Each Row

When (Select Avg(V) From T) < (Select Avg(V) From NT)

Update T set V=V+10 where K=NR.K

No statement-level equivalent

Nondeterministic final state

Agenda

Introduction

Examples

Before and After

Insert, Delete, and Update

New and Old

Conditions and actions

Triggers enforcing constraints

SQL standard

Previous slides used SQL standard

No DBMS implements exact standard

Some deviate considerably in syntax and behavior

SQL standard

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

Triggers in SQLite

Row-level triggers, immediate activation

For Each Row implicit if not specified

No Old Table or New Table

No Referencing clause

Old and New predefined for Old Row and New Row

Trigger action: SQL statements in begin-end block

College Admission Database

Apply

sID	cName	major	dec
123	Stanford	CS	Y
123	Stanford	EE	N
123	Berkeley	CS	Y
123	Cornell	EE	Y
234	Berkeley	biology	N
345	MIT	bioengineering	Y
345	Cornell	bioengineering	N
345	Cornell	CS	Y
345	Cornell	EE	N
678	Stanford	history	Y
987	Stanford	CS	Y
987	Berkeley	CS	Y
876	Stanford	CS	Y
876	MIT	biology	Y
876	MIT	marine biology	N
765	Stanford	history	Y
765	Cornell	history	N
765	Cornell	psychology	Y
543	MIT	CS	N

College

cName	state	enr
Stanford	CA	15000
Berkeley	CA	36000
MIT	MA	10000
Cornell	NY	21000

Student

sID	sName	GPA	sizeHS
123	Amy	3.9	1000
234	Bob	3.6	1500
345	Craig	3.5	500
456	Doris	3.9	1000
567	Edward	2.9	2000
678	Fay	3.8	200
789	Gary	3.4	800
987	Helen	3.7	800
876	Irene	3.9	400
765	Jay	2.9	1500
654	Amy	3.9	1000
543	Craig	3.4	2000

Trigger 1 – after insert

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

Create Trigger R1

After Insert On Student

For Each Row

When New.GPA > 3.3 AND New.GPA <= 3.6

Begin

Insert into Apply values (New.sID, 'Stanford', 'geology', null);

Insert into Apply values (New.sID, 'MIT', 'biology', null);

End;

A first test to trigger 1

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

Insert into Student values ('111', 'Kevin', 3.5, 1000);

Insert into Student values ('222', 'Lori', 3.8, 1000);

Student

sID	sName	GPA	sizeHS
...
765	Jay	2.9	1500
654	Amy	3.9	1000
543	Craig	3.4	2000
111	Kevin	3.5	1000
222	Lori	3.8	1000

Apply

sID	cName	major	dec
...
765	Cornell	history	N
765	Cornell	psychology	Y
543	MIT	CS	N
111	Stanford	geology	NULL
111	MIT	biology	NULL

A second test to trigger 1

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

Insert into Student

```
select sID+1, sName, GPA, sizeHS from Student;
```

Student

<u>sID</u>	<u>sName</u>	GPA	sizeHS
...
111	Kevin	3.5	1000
222	Lori	3.8	1000
...
766	Jay	2.9	1500
655	Amy	3.9	1000
544	Craig	3.4	2000
112	Kevin	3.5	1000
223	Lori	3.8	1000

Apply

<u>sID</u>	<u>cName</u>	<u>major</u>	<u>dec</u>
...
111	Stanford	geology	NULL
111	MIT	biology	NULL
...
112	Stanford	geology	NULL
112	MIT	biology	NULL

Trigger 2 – after delete

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

Create Trigger R2

After Delete On Student

For Each Row

Begin

Delete from Apply where sID = Old.sID;

End;

What does it do?

A test to trigger 2

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

Delete from Student where sID > 500;

Student

<u>sID</u>	<u>sName</u>	GPA	sizeHS
...
111	Kevin	3.5	1000
222	Lori	3.8	1000
...
457	Doris	3.9	1000
112	Kevin	3.5	1000
223	Lori	3.8	1000

12 tuples

Apply

<u>sID</u>	<u>cName</u>	<u>major</u>	dec
...
111	Stanford	geology	NULL
111	MIT	biology	NULL
...
112	Stanford	geology	NULL
112	MIT	biology	NULL

17 tuples

Trigger 3 – after update

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

Create Trigger R3

After **Update Of** cName on College

For Each Row

Begin

Update Apply

Set cName = New.cName

Where cName = Old.cName;

End;



If we left out cName then
any update to College
would activate this trigger

What does it do?

A test to trigger 3

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

Update College **set** cName = 'The Farm' **where** cName = 'Stanford';

Update College **set** cName = 'Bezerkeley' **where** cName = 'Berkeley';

College

cName	state	enr
The Farm	CA	15000
Bezerkeley	CA	36000
MIT	MA	10000
Cornell	NY	21000

Apply

sID	cName	major	dec
123	The Farm	CS	Y
123	The Farm	EE	N
123	Bezerkeley	CS	Y
123	Cornell	EE	Y
234	Bezerkeley	biology	N
345	MIT	bioengineering	Y
345	Cornell	bioengineering	N
345	Cornell	CS	Y
345	Cornell	EE	N
678	The Farm	history	Y
987	The Farm	CS	Y
987	Bezerkeley	CS	Y

Trigger 4 – Simulating key constraints

Create Trigger R4

Before Insert on College

For Each Row

When exists (select * from College where cName = New.cName)

Begin

Select raise(ignore);



College(cName, state, enr)

Student(sID, sName, GPA, sizeHS)

Apply(sID, cName, major, decision)

End;

Ignores the operation
that's underway

What does it do?

Trigger 5 – Simulating key constraints

Create Trigger R4

Before Update of cName on College

For Each Row

When exists (select * from College where cName = New.cName)

Begin

Select raise(ignore);

End;

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

A test to trigger 4

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

Insert into College values ('Stanford', 'CA', 15000);

Insert into College values ('MIT', 'hello', 10000);

What should happen?

College

cName	state	enr
The Farm	CA	15000
Bezerkeley	CA	36000
MIT	MA	10000
Cornell	NY	21000



College

cName	state	enr
The Farm	CA	15000
Bezerkeley	CA	36000
MIT	MA	10000
Cornell	NY	21000
Stanford	CA	15000

A test to trigger 5

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

Update College **set** cName = 'Berkeley' **where** cName = 'Bezerkeley';

Update College **set** cName = 'Stanford' **where** cName = 'The Farm';

Update College **set** cName = 'Standford' **where** cName = 'The Farm';

What should happen?

College

<u>cName</u>	state	enr
The Farm	CA	15000
Bezerkeley	CA	36000
MIT	MA	10000
Cornell	NY	21000
Stanford	CA	15000



College

<u>cName</u>	state	enr
Standford	CA	15000
Berkeley	CA	36000
MIT	MA	10000
Cornell	NY	21000
Stanford	CA	15000

A test to trigger 5

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

Anything happened behind the scenes?

Apply

<u>sID</u>	<u>cName</u>	<u>major</u>	<u>dec</u>
123	The Farm	CS	Y
123	The Farm	EE	N
123	Bezerkeley	CS	Y
123	Cornell	EE	Y
234	Bezerkeley	biology	N
345	MIT	bioengineering	Y
345	Cornell	bioengineering	N
345	Cornell	CS	Y
345	Cornell	EE	N
678	The Farm	history	Y
987	The Farm	CS	Y
987	Bezerkeley	CS	Y
978	Stanford	CS	Y

Apply

<u>sID</u>	<u>cName</u>	<u>major</u>	<u>dec</u>
123	Standford	CS	Y
123	Standford	EE	N
123	Berkeley	CS	Y
123	Cornell	EE	Y
234	Berkeley	biology	N
345	MIT	bioengineering	Y
345	Cornell	bioengineering	N
345	Cornell	CS	Y
345	Cornell	EE	N
678	Standford	history	Y
987	Standford	CS	Y
987	Berkeley	CS	Y
978	Stanford	CS	Y

trigger 3



Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in
Database Systems 3rd Edition

Section 7.5 – Triggers

SQL – Views

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Agenda

~~D~~efining and Using Views

~~V~~iews Modifications Introduction

Views Modification Using Triggers

View CSaccept

Create View CSaccept As

Select sID, cName

From Apply

Where major = 'CS' and dec = 'Y';

Select *
From CSaccept



sID	cName
123	Stanford
123	Berkeley
345	Cornell
987	Stanford
987	Berkeley

Delete from CSaccept

Delete from CSaccept

Where sID = 123;



Error: SQLite does not
allow to modify views

Create trigger CSacceptDelete

instead of delete on Csaccept

For each row

Begin

Delete from Apply

Where sID = Old.sID and cName = Old.cName and
major = 'CS' and dec = 'Y';

End;

sID	cName
123	Stanford
123	Berkeley
345	Cornell
987	Stanford
987	Berkeley

Delete from CSaccept

CSaccept

sID	cName
123	Stanford
123	Berkeley
345	Cornell
987	Stanford
987	Berkeley

Create trigger CSacceptDelete

instead of delete on Csaccept

For each row

Begin

 Delete from Apply

 Where sID = Old.sID and

 cName = Old.cName and

 major = 'CS' and dec = 'Y';

End;

Apply

sID	cName	major	dec
123	Stanford	CS	Y
123	Stanford	EE	N
123	Berkeley	CS	Y
123	Cornell	EE	Y
234	Berkeley	biology	N
345	MIT	bioengineering	Y
345	Cornell	bioengineering	N
345	Cornell	CS	Y
345	Cornell	EE	N
678	Stanford	history	Y
987	Stanford	CS	Y
987	Berkeley	CS	Y
876	Stanford	CS	Y
876	MIT	biology	Y
876	MIT	marine biology	N
765	Stanford	history	Y
765	Cornell	history	N

Delete from CSaccept

Delete from CSaccept

Where sID = 123;



sID	cName
345	Cornell
987	Stanford
987	Berkeley



Apply

sID	cName	major	dec
123	Stanford	EE	N
123	Cornell	EE	Y
234	Berkeley	biology	N
345	MIT	bioengineering	Y
345	Cornell	bioengineering	N
345	Cornell	CS	Y
345	Cornell	EE	N
678	Stanford	history	Y
987	Stanford	CS	Y

Update CSaccept

sID	cName
345	Cornell
987	Stanford
987	Berkeley



Change this acceptance to
Carnegie Melon

Update CSaccept

Set cName = 'CMU'



Error: SQLite does not
allow to modify views

Where sID = 345;

Update CSaccept

Create trigger CSacceptUpdate

instead of update of cName on Csaccept

For each row

Begin

 Update Apply

 Set cName = New.cName

 Where sID = Old.sID and cName = Old.cName and

 major = 'EE' and dec = 'N';

End;

Update CSaccept

CSaccept

sID	cName
345	Cornell
987	Stanford
987	Berkeley

Create trigger CSacceptUpdate

instead of update of cName on Csaccept

For each row

Begin

 Update Apply

 Set cName = New.cName

 Where sID = Old.sID and cName = Old.cName
 and major = 'EE' and dec = 'N';

End;

Apply

sID	cName	major	dec
123	Stanford	CS	Y
123	Stanford	EE	N
123	Berkeley	CS	Y
123	Cornell	EE	Y
234	Berkeley	biology	N
345	MIT	bioengineering	Y
345	Cornell	bioengineering	N
345	Cornell	CS	Y
345	Cornell	EE	N
678	Stanford	history	Y
987	Stanford	CS	Y
987	Berkeley	CS	Y
876	Stanford	CS	Y
876	MIT	biology	Y
876	MIT	marine biology	N
765	Stanford	history	Y
765	Cornell	history	N

Update CSaccept

Update CSaccept

Set cName = 'CMU'

Where sID = 345;

Nothing in the system is checking the correctness of the translation



CSaccept

sID	cName
345	Cornell
987	Stanford
987	Berkeley



Apply

sID	cName	major	dec
123	Stanford	EE	N
123	Cornell	EE	Y
234	Berkeley	biology	N
345	MIT	bioengineering	Y
345	Cornell	bioengineering	N
345	Cornell	CS	Y
345	CMU	EE	N
678	Stanford	history	Y
987	Stanford	CS	Y



Update CSaccept

Create trigger CSacceptUpdate

instead of update of cName on Csaccept

For each row

Begin

 Update Apply

 Set cName = New.cName

 Where sID = Old.sID and cName = Old.cName and

 major = 'CS' and dec = 'Y';

End;

View CSEE

Create View CSEE As

Select sID, cName, major

From Apply

Where major = 'CS' or major = 'EE';

Select *

From CSEE



sID	cName	major
123	Stanford	EE
123	Berkeley	EE
345	Cornell	CS
345	CMU	EE
987	Stanford	CS
987	Berkeley	CS
876	Stanford	CS
543	MIT	CS

Insert on CSEE

Insert into CSEE values (111, 'Berkeley', 'CS'); → Error: SQLite does not allow to modify views

Create trigger CSEEinser

Instead of insert on CSEE

For each row

Begin

Insert into Apply values (New.sID, New.cName, New.major, null);

End;

Insert on CSEE

Insert into CSEE values
(111, 'Berkeley', 'CS');



CSEE

sID	cName	major
123	Stanford	EE
123	Berkeley	EE
345	Cornell	CS
345	CMU	EE
987	Stanford	CS
987	Berkeley	CS
876	Stanford	CS
543	MIT	CS
111	Berkeley	CS

Apply

sID	cName	major	dec
123	Stanford	EE	N
...
543	MIT	CS	N
111	Berkeley	CS	Null

Insert on CSEE

Insert into CSEE values

(222, 'Berkeley', 'biology');



CSEE

sID	cName	major
123	Stanford	EE
123	Berkeley	EE
345	Cornell	CS
345	CMU	EE
987	Stanford	CS
987	Berkeley	CS
876	Stanford	CS
543	MIT	CS
111	Berkeley	CS

Apply

sID	cName	major	dec
123	Stanford	EE	N
...
111	Berkeley	CS	Null
222	Berkeley	biology	Null

Don't want insertions
on a view that affect
the base tables and
not the view



Insert on CSEE

Drop trigger CSEEEinsert;

Create trigger CSEEEinsert

Instead of insert on CSEE

For each row

When New.major = 'CS' or New.major = 'EE'

Begin

 Insert into Apply values (New.sID, New.cName, New.major, null);

End;

Insert on CSEE

Insert into CSEE values

(333, 'Berkeley', 'biology');



CSEE

sID	cName	major
123	Stanford	EE
123	Berkeley	EE
345	Cornell	CS
345	CMU	EE
987	Stanford	CS
987	Berkeley	CS
876	Stanford	CS
543	MIT	CS
111	Berkeley	CS

Apply

sID	cName	major	dec
123	Stanford	EE	N
...
111	Berkeley	CS	Null
222	Berkeley	biology	Null

Insert on CSEE

Insert into CSEE values
(333, 'Berkeley', 'EE');



CSEE

sID	cName	major
123	Stanford	EE
123	Berkeley	EE
345	Cornell	CS
345	CMU	EE
987	Stanford	CS
987	Berkeley	CS
876	Stanford	CS
543	MIT	CS
111	Berkeley	CS
333	Berkeley	EE

Apply

sID	cName	major	dec
123	Stanford	EE	N
...
222	Berkeley	biology	Null
333	Berkeley	EE	Null

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 8.1 – Virtual Views

Section 8.2 – Modifying Views

SQL – Indexes

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Indexes

Primary mechanism to get improved performance on a database

Persistent data structure, stored in database

Many interesting implementation issues

But we are focusing on user/application perspective

Functionality



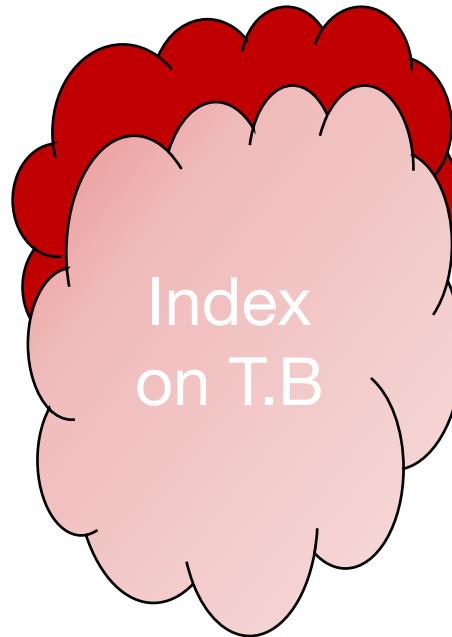
If we want tuples with $T.A = \text{'cow'}$,
DBMS doesn't need to scan the entire
table

Users don't access indexes

Indexes are used underneath by the
query execution engine

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	...

Functionality



Useful for queries involving T.B like:

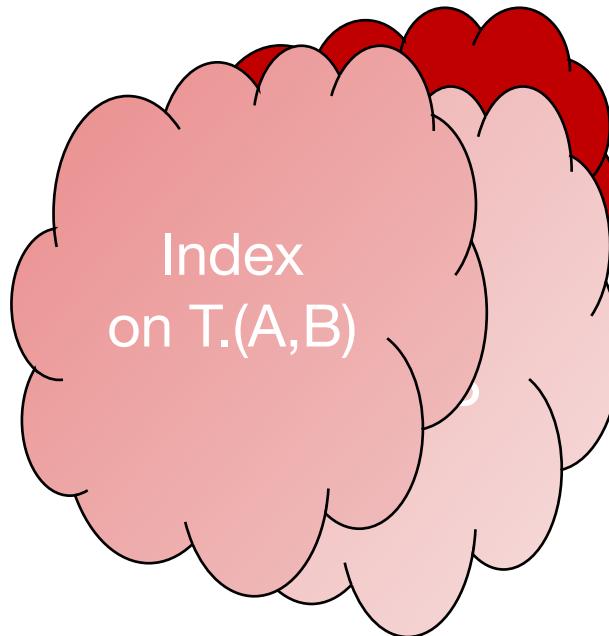
T.B = 2

T.B < 6

$4 < T.B \leq 8$

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	...

Functionality



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	...

Useful for queries involving T.A and T.B like:

T.A = 'cat' AND T.B > 5

T.A < 'd' AND T.B = 1

Utility

Index = difference between full table scans and immediate location of tuples

Orders of magnitude performance difference

Underlying data structures

Balanced trees (B trees, B+ trees)

For equalities or inequalities conditions

Operations running time tend to be logarithmic

Hash tables

Only for equality conditions

Operations running time is more or less constant

Where should indexes be created?

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

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

Where should indexes be created?

Select sID

From Student

Where sName = 'Mary' and GPA>3.9

Index on sName

Hash or tree-based

Index on GPA

Tree-based

Index on (sName, GPA)

Where should indexes be created?

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

More attributes →
Query planning &
optimization

Index on Apply.sID

Scans Student relation and, for each student, finds the matching SID in the Apply relation

Index on Student.sID

Scans Apply relation and, for each student, finds the matching SID in the Student relation

Index on (Student.sID, Apply.sID)

Downside of Indexes

Extra space

Marginal downside

Index creation

Medium downside

Index maintenance

Can offset benefits

Picking which indexes to create

Benefit of an index depends on:

- Size of table (and possibly layout)

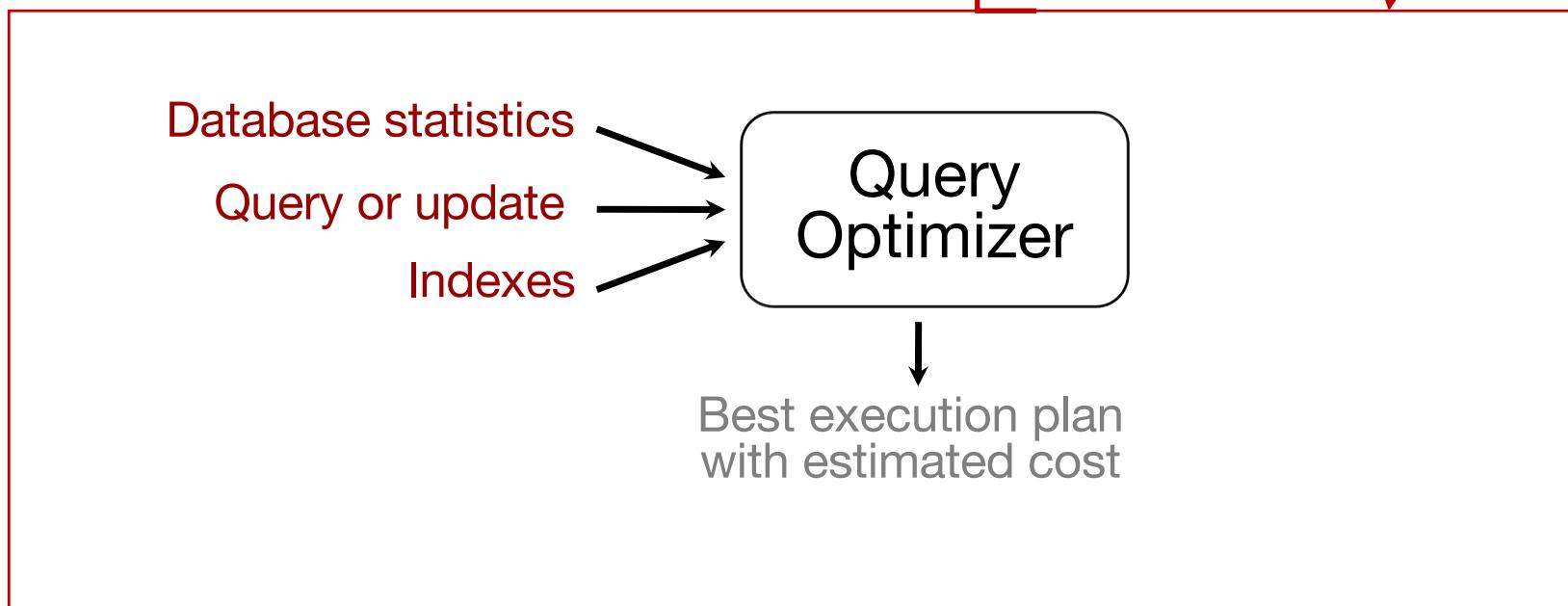
- Data distributions

- Query vs. update load

“Physical design advisors”

Input: database (statistics) and workload

Output: recommended indexes



Selects indexes whose
benefits outweigh
drawbacks

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

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in Database Systems 3rd Edition

Section 8.3 – Indexes in SQL

Section 8.4 – Selection of Indexes

SQL – Transactions

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Based on Jennifer Widom slides

Agenda

Introduction

Properties

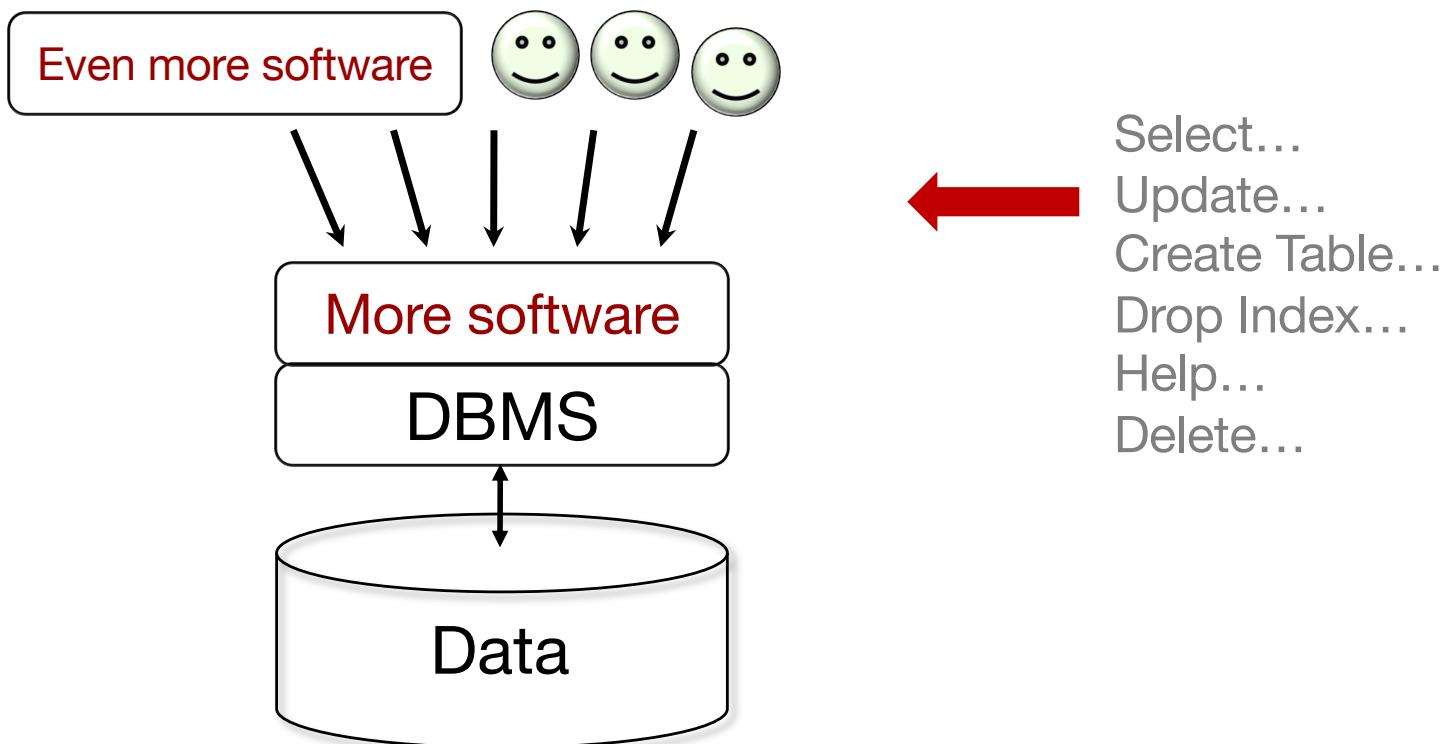
Isolation levels

Motivation for transactions

Concurrent database access

Resilience to system failures

Concurrent Database Access



Attribute-level Inconsistency

Update College Set enr = enr + 1000 **Where** cName = ‘Stanford’

concurrent with ...

Update College Set enr = enr + 1500 **Where** cName = ‘Stanford’

	15000	

get; modify; put

$$15\ 000 + 2\ 500 = 17\ 500$$

$$15\ 000 + 1\ 000 = 16\ 000$$

$$15\ 000 + 1\ 500 = 16\ 500$$

Tuple-level Inconsistency

Update Apply Set major = 'CS' Where sID = 123

concurrent with ...

Update Apply Set dec = 'Y' Where sID = 123



sID	major	dec
123		

get; modify; put

both changes

One of the two changes

Table-level Inconsistency

Update Apply Set decision = 'Y'

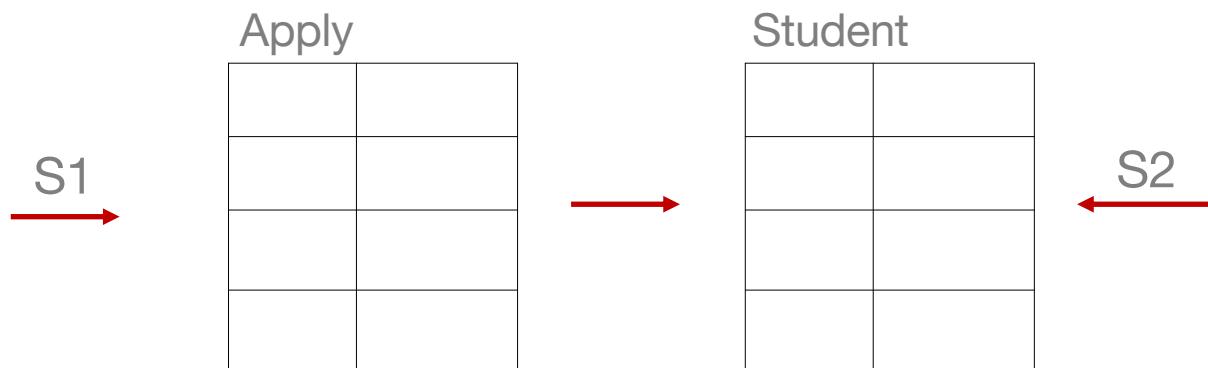
Where sID In (Select sID From Student Where GPA > 3.9)

} S1

concurrent with ...

Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500 }

S2



Multi-statement Inconsistency

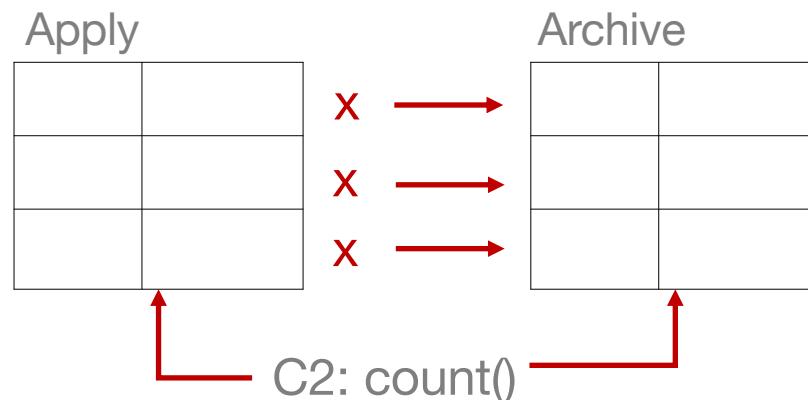
Insert Into Archive Select * From Apply Where decision = 'N';
Delete From Apply Where decision = 'N';

C1

concurrent with ...

Select Count(*) From Apply;
Select Count(*) From Archive;

C2



Concurrency goal

Execute sequence of SQL statements so they appear to be running in isolation

Simple solution: execute them in isolation

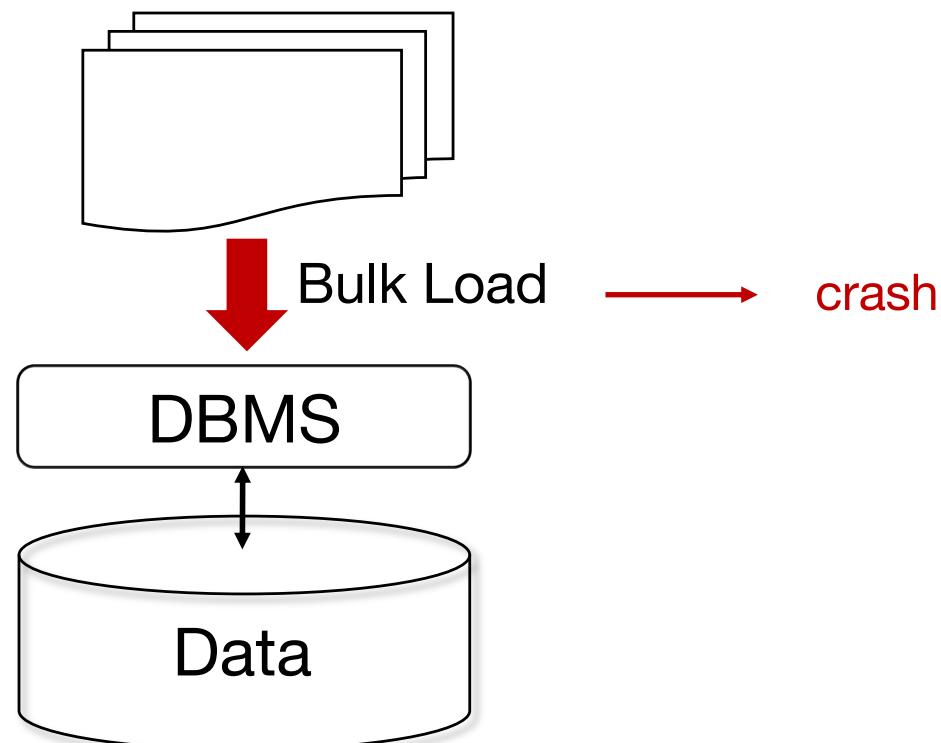
But want to enable concurrency whenever safe to do so

Multiprocessor system

Multithreaded system

Asynchronous I/O

Resilience to System Failures

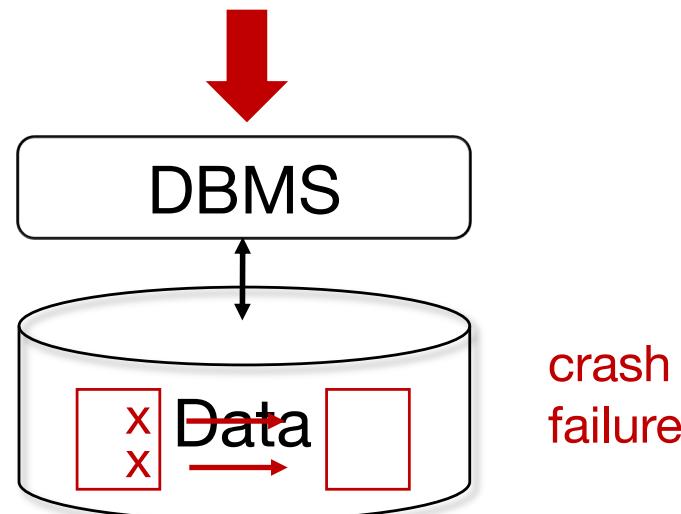


Resilience to System Failures

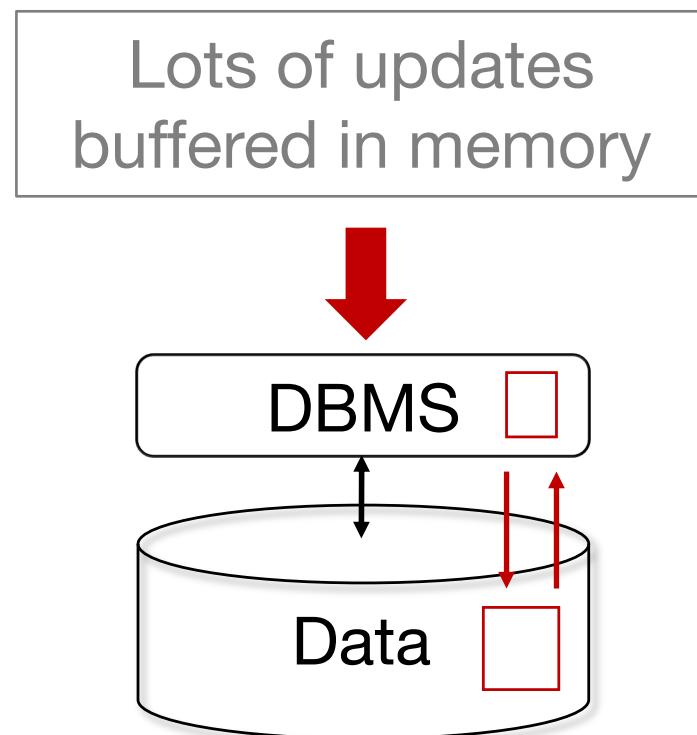
Insert Into Archive

Select * From Apply Where decision = 'N';

Delete From Apply Where decision = 'N';

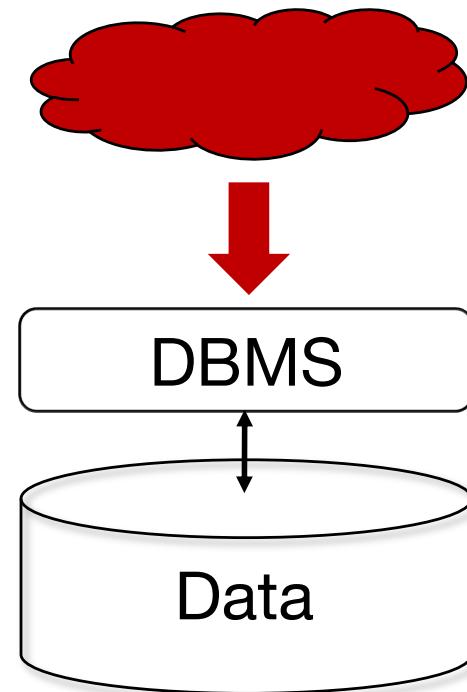


Resilience to System Failures



System-Failure Goal

Guarantee all-or-nothing execution, regardless of failures



Transactions

Solution for both concurrency and failures

A transaction is a sequence of one or more SQL operations treated as a unit

Transactions appear to run in isolation

If the system fails, each transaction's changes are reflected either entirely or not at all

Transactions: SQL standard

Transaction begins automatically on first SQL statement

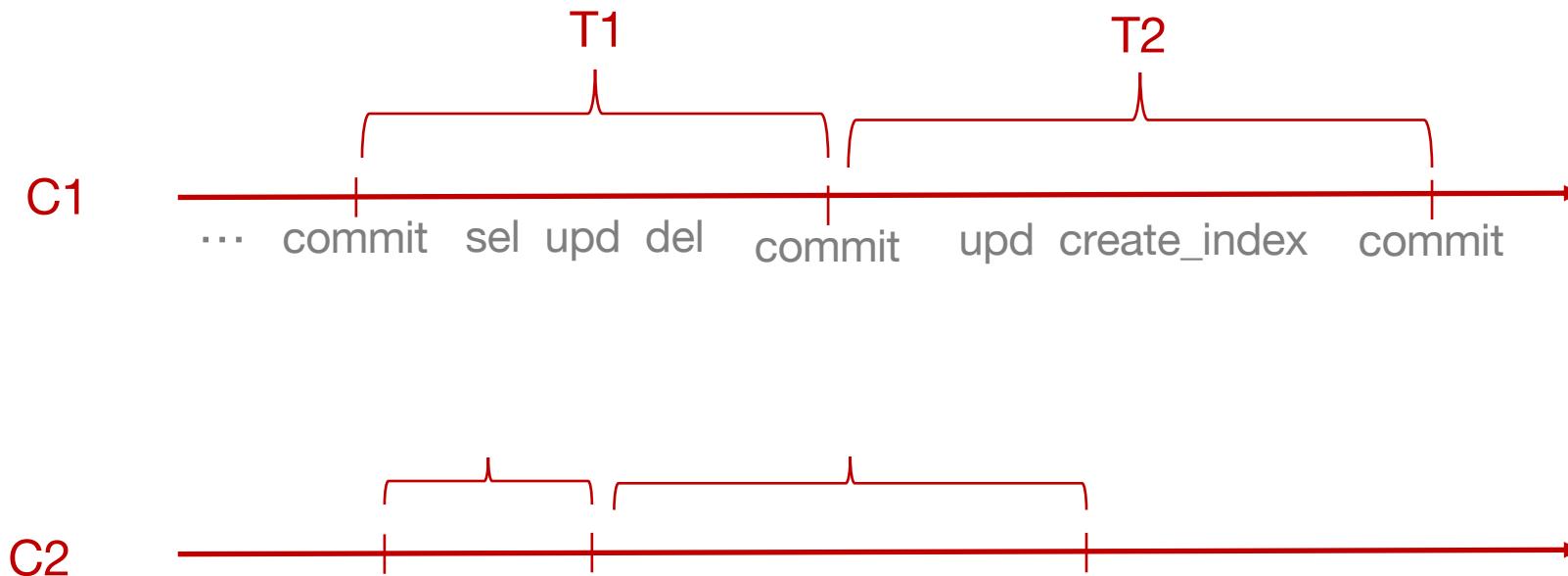
On “**commit**” transaction ends and new one begins

Current transaction ends on session termination

“**Autocommit**” turns each statement into transaction

Transactions

A transaction is a sequence of one or more SQL operations treated as a unit



Agenda

Introduction

Properties

Isolation levels

ACID Properties

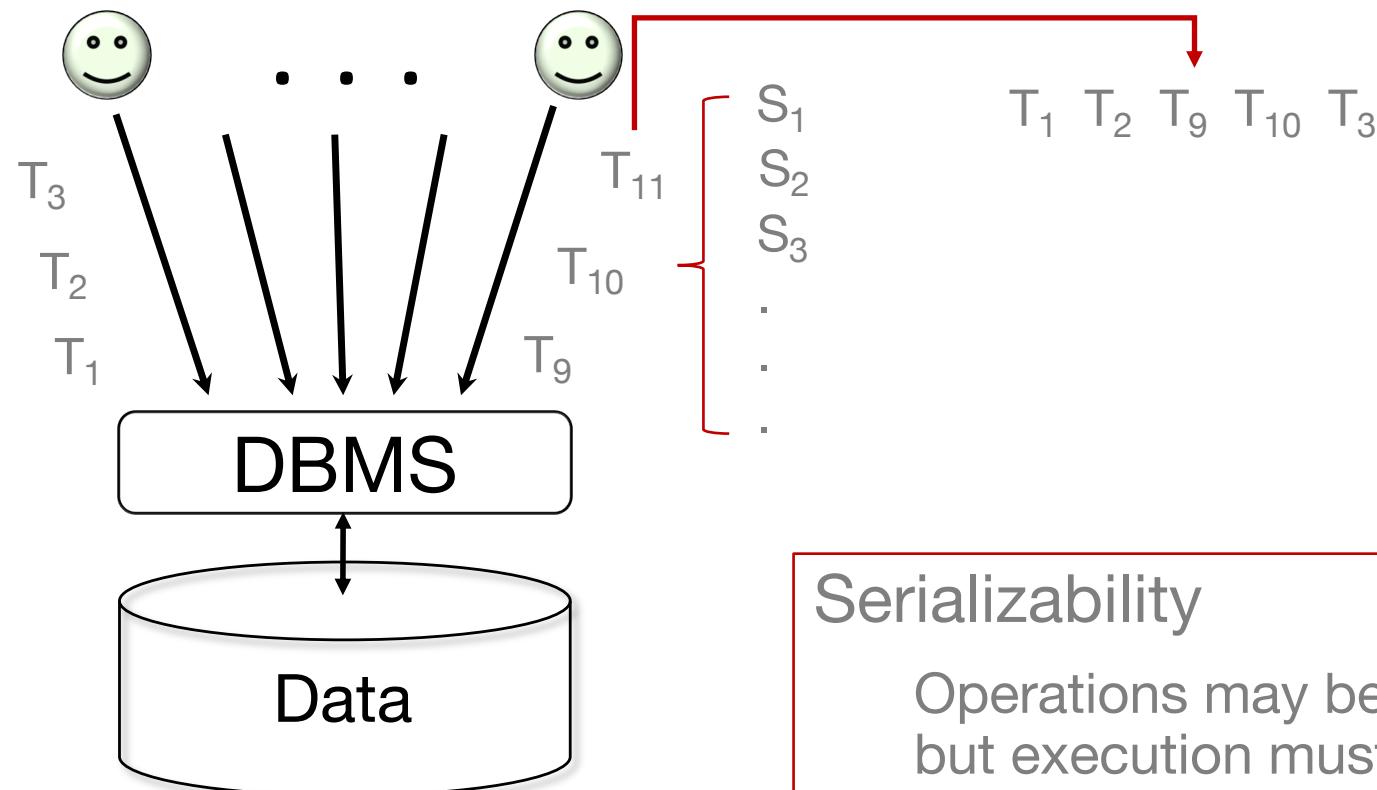
Atomicity 3

Consistency 4

Isolation 1

Durability 2

ACID Properties: Isolation



Serializability

Operations may be interleaved,
but execution must be
equivalent to some sequential
(serial) order of all transactions

Attribute-level Inconsistency

Update College Set enr = enr + 1000 **Where** cName = ‘Stanford’ T_1

concurrent with ...

Update College Set enr = enr + 1500 **Where** cName = ‘Stanford’ T_2

If serializability is guaranteed

$T_1; T_2$ \longrightarrow 15 000 \rightarrow 17 500
 $T_2; T_1$

Tuple-level Inconsistency

Update Apply Set major = 'CS' Where sID = 123

T₁

concurrent with ...

Update Apply Set dec = 'Y' Where sID = 123

T₂

If serializability is guaranteed

T₁; T₂
T₂; T₁

→ Both changes

Table-level Inconsistency

Update Apply Set decision = 'Y'

Where sID In (Select sID From Student Where GPA > 3.9)

} T₁

concurrent with ...

Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500

} T₂

If serializability is guaranteed

T₁; T₂
T₂; T₁



Order
matters



DBMS don't guarantee the exact sequential order if the transactions are being issued at the same time

Multi-statement Inconsistency

Insert Into Archive Select * From Apply Where decision = 'N';

Delete From Apply Where decision = 'N';

T₁

concurrent with ...

Select Count(*) From Apply;

Select Count(*) From Archive;

T₂

If serializability is guaranteed

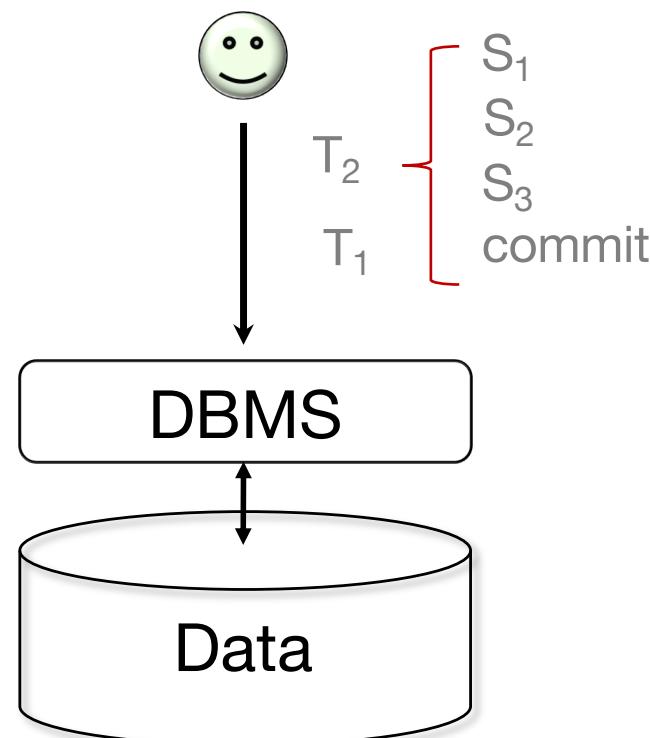
T₁; T₂
T₂; T₁



Order matters

ACID Properties: Durability

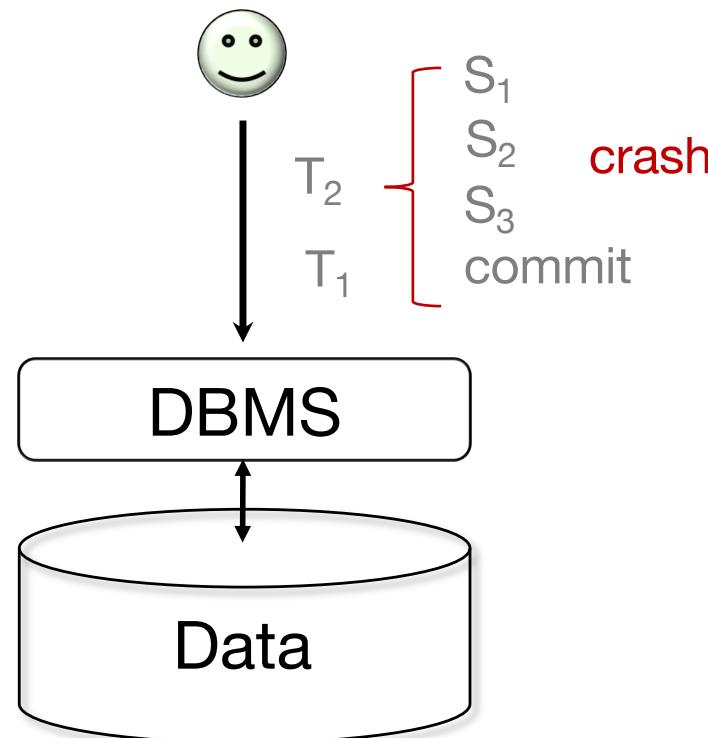
If system crashes after transaction commits, all effects of transaction remain in database



ACID Properties: Atomicity

Each transaction is “all-or-nothing”, never left half done

Using a logging mechanism, partial effects of transactions at the time of crash are undone



Transaction Rollback (= Abort)

Undoes partial effects of transaction

Can be system- or client-initiated

```
Begin Transaction;  
<get input from user>  
SQL commands based on input  
<confirm results with user>  
If ans='ok' Then Commit; Else Rollback;
```

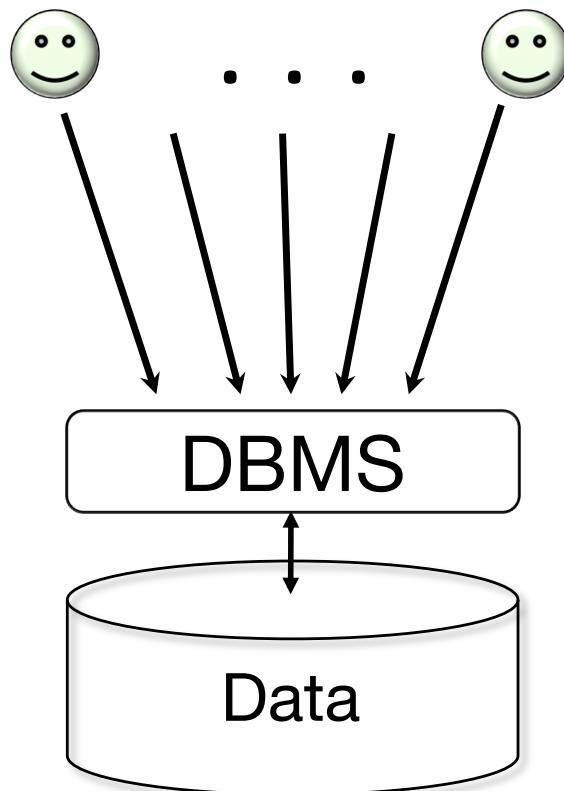
Transactions should
be constructed to run
quickly

→ Not wait arbitrary
amounts of time

Locking

Only undoes effects on the data itself

ACID Properties: Consistency



Each client, each transaction:

Can assume all constraints hold when transaction begins

Must guarantee all constraints hold when transaction ends

Serializability

Constraints always hold

T_1 T_2 T_3

Three red arrows originate from below the text and point upwards towards the transaction labels T_1 , T_2 , and T_3 , emphasizing the concept of serializability.

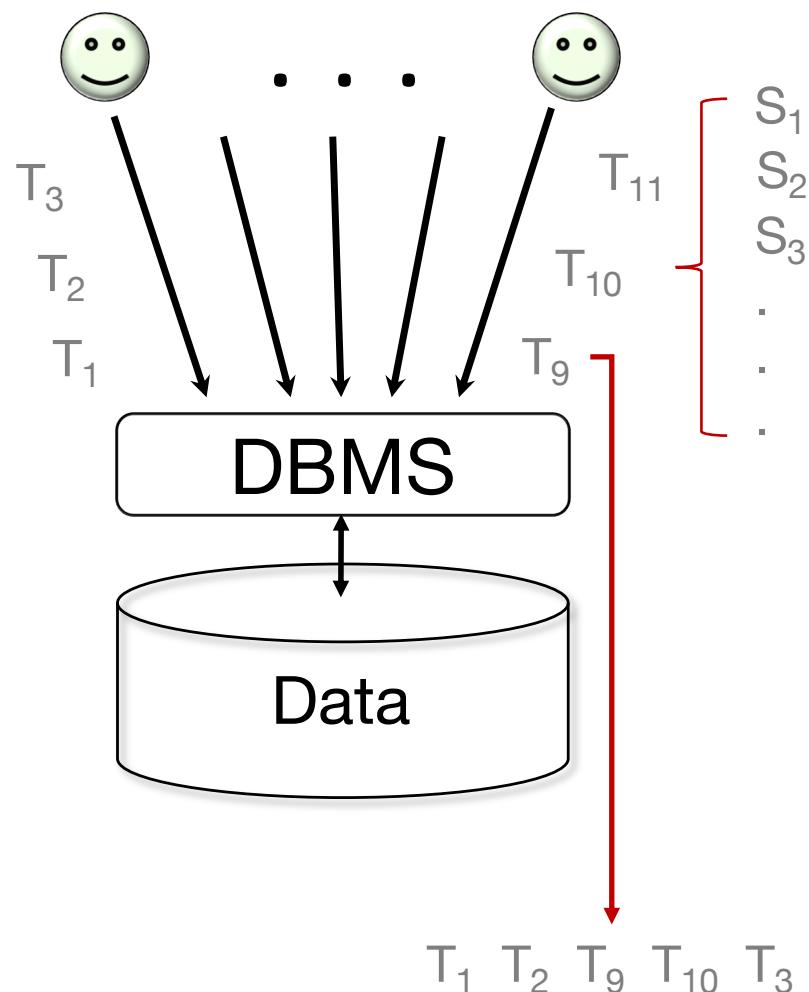
Agenda

Introduction

Properties

Isolation levels

ACID Properties: Isolation



Serializability

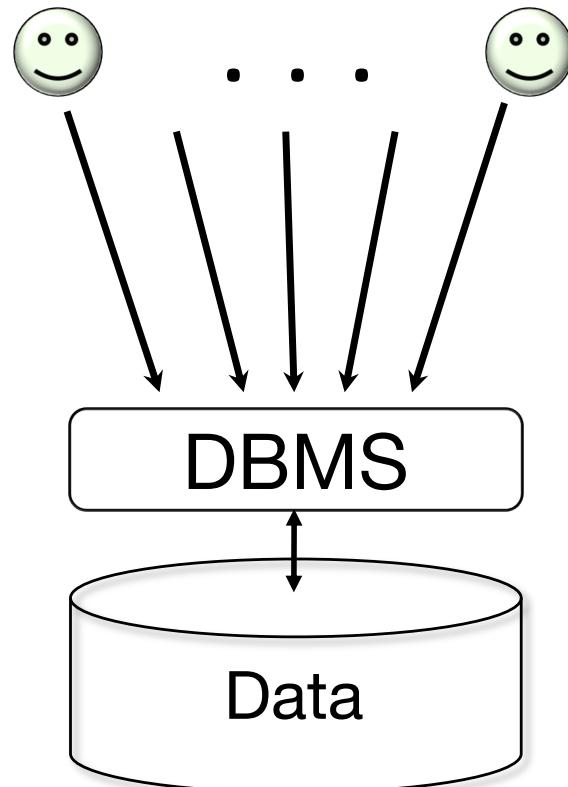
Operations may be interleaved, but execution must be equivalent to some sequential (serial) order of all transactions

Disadvantages

Overhead in locking

Reduction in concurrency

ACID Properties: Isolation



Weaker “Isolation Levels”

Read Uncommitted

Read Committed

Repeatable Read

Serializable

Weak
↓
Strong

↓Overhead in locking

↑Concurrency

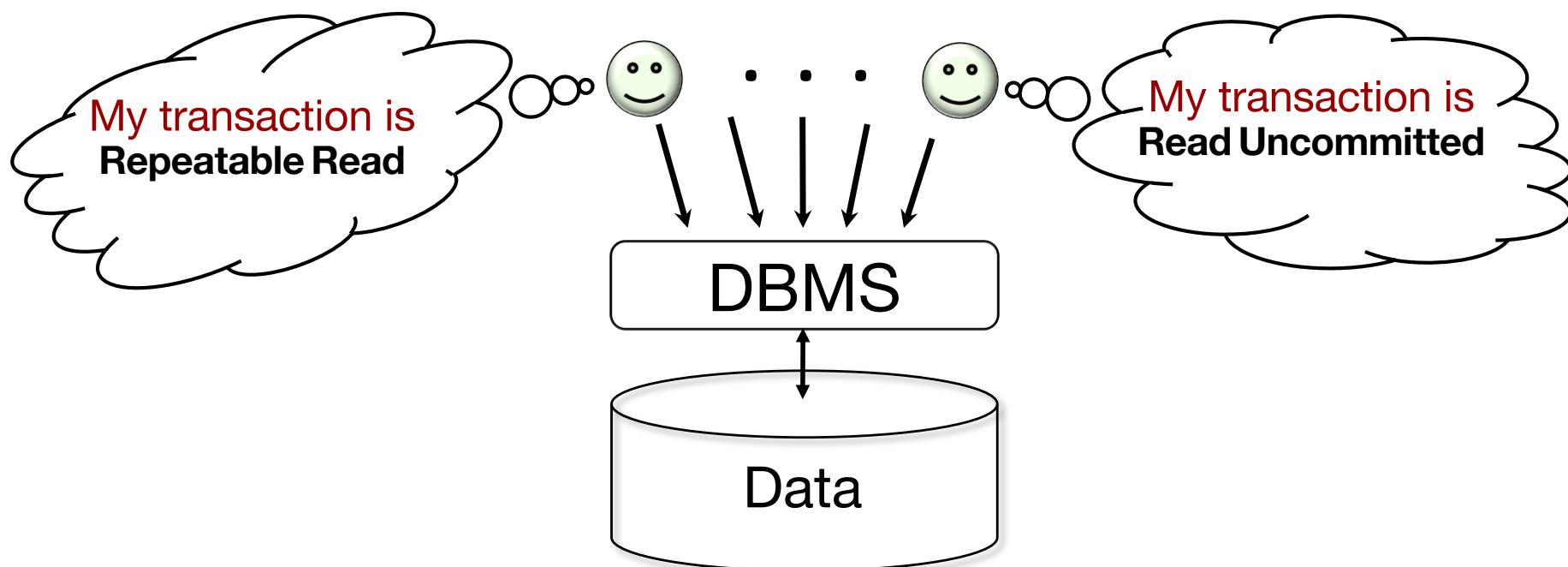
↓Consistency Guarantees

Isolation Levels

Per transaction

It does not affect the behaviour of any other transaction

Specific to Reads



Dirty Reads

“Dirty” data item: written by an uncommitted transaction

Update College Set enr = enr + 1000 Where cName = ‘Stanford’

T₁

concurrent with ...

Select avg(enr) From College

T₂



If read before T1 commits, this value is known as dirty

Assume there is a commit at the end of each box

Dirty Reads – Example 2

Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500

T₁

concurrent with ...

Select GPA From Student Where sID=123

T₂

concurrent with ...

Update Student Set sizeHS=2600 Where sID=234

T₃

Where can we have dirty data items?

There are no
dirty reads
within the same
transaction

Read Uncommitted

A transaction may perform dirty reads

Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500

T₁

concurrent with ...

Select avg(GPA) From Student

T₂

If transactions are serializable

T1; T2 or

T2; T1

Read Uncommitted

Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500

T₁

concurrent with ...

Set Transaction Isolation Level Read Uncommitted;

Select avg(GPA) From Student;

T₂

We don't have serializable behaviour

We might don't care that much about consistency

Read Committed

A transaction may **not** perform dirty reads

Still does not guarantee global serializability

Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500

T₁

concurrent with ...

Set Transaction Isolation Level Read Committed;

Select avg(GPA) From Student

Select max(GPA) From Student

T₂

Repeatable Read

A transaction may **not** perform dirty reads

An item read multiple times cannot change value

Still does not guarantee global serializability

Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500;

Update Student Set sizeHS=1500 Where sID = 123;

T₁

concurrent with ...

Set Transaction Isolation Level Repeatable Read;

Select avg(GPA) From Student

Select avg(sizeHS) From Student

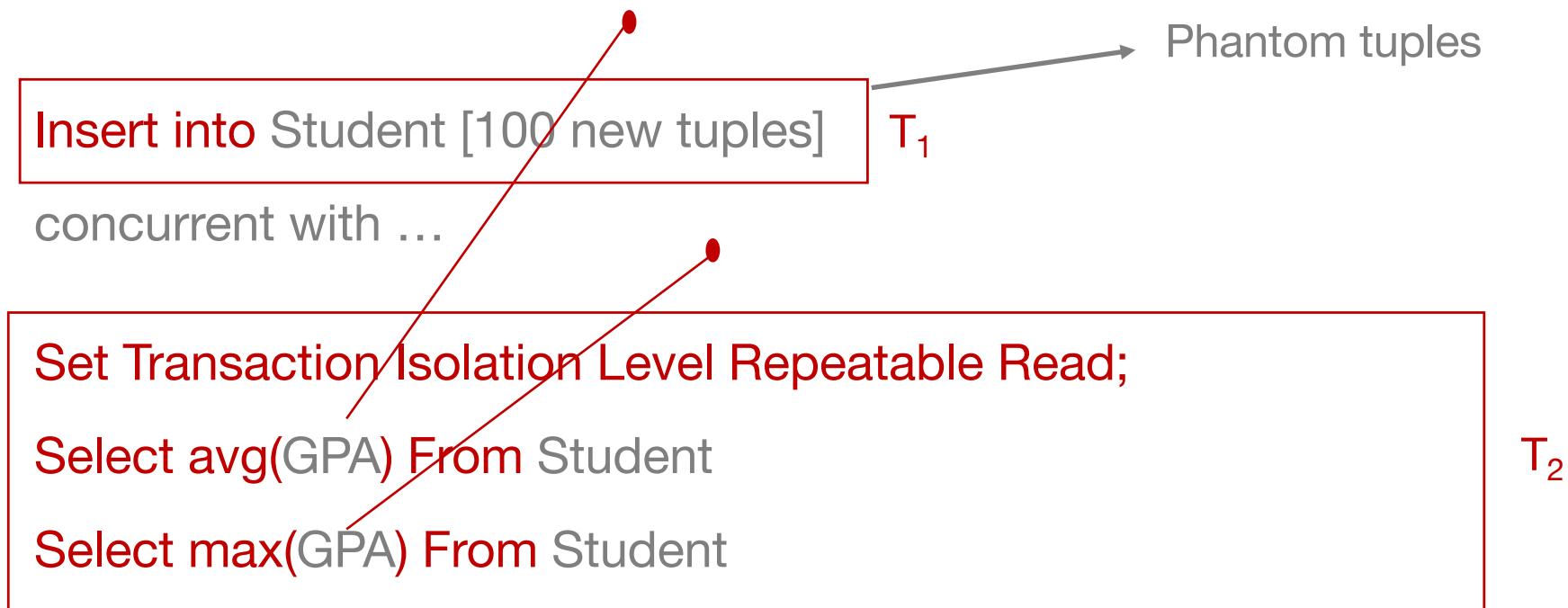
T₂

Repeatable Read

A transaction may **not** perform dirty reads

An item read multiple times cannot change value

But a relation *can* change: “phantom” tuples



Repeatable Read

A transaction may **not** perform dirty reads

An item read multiple times cannot change value

But a relation *can* change: “phantom” tuples

Delete from Student [100 new tuples]

T₁

concurrent with ...

Set Transaction Isolation Level Repeatable Read;

Select avg(GPA) From Student

T₂

Select max(GPA) From Student

Once read, values get locked and deletion is not possible in the middle of T₂

Read Only Transactions

Helps system optimize performance

Independent of isolation level

Not going to perform modifications to
the database within the transaction

Set Transaction Read Only;

Set Transaction Isolation Level Repeatable Read;

Select avg(GPA) From Student

Select max(GPA) From Student

Isolation Levels: Summary

	dirty reads	nonrepeatable reads	phantoms
weak			
Read Uncommitted	Y	Y	Y
Read Committed	N	Y	Y
Repeatable Read	N	N	Y
Serializable	N	N	N
strong			

Isolation Levels: Summary

Standard default: Serializable

Weaker isolation levels

Increased concurrency + decreased overhead = increased performance

Weaker consistency guarantees

Some systems have default Repeatable Read

Isolation level per transaction

Each transaction's reads must conform to its isolation level

Kahoot time!

Any doubts?

Readings

Jeffrey Ullman, Jennifer Widom, A first course in
Database Systems 3rd Edition

Section 6.6 – Transactions in SQL

NoSQL

Carla Teixeira Lopes

Bases de Dados

Mestrado Integrado em Engenharia Informática e Computação, FEUP

Lemahieu, Broucke and Baesens slides

Agenda

The NoSQL movement

Key-Value stores

Tuple and Document stores

Column-oriented databases

Graph based databases

The NoSQL movement

RDBMSs put a lot of emphasis on keeping data consistent.

- Entire database is consistent at all times (ACID)

Focus on consistency may hamper flexibility and scalability

As the data volumes or number of parallel transactions increase, capacity can be increased by

- Vertical scaling: extending storage capacity and/or CPU power of the database server
- Horizontal scaling: multiple DBMS servers being arranged in a cluster

The NoSQL movement

RDBMSs are not good at extensive horizontal scaling

- Coordination overhead because of focus on consistency
- Rigid database schemas

Other types of DBMSs needed for situations with massive volumes, flexible data structures and where scalability and availability are more important -> NoSQL databases

The NoSQL movement

NoSQL databases

- Describes databases that store and manipulate data in other formats than tabular relations, i.e. non-relational databases (NoREL)

NoSQL databases aim at near linear horizontal scalability, by distributing data over a cluster of database nodes for the sake of performance as well as availability

Eventual consistency: the data (and its replicas) will become consistent at some point in time after each transaction

The NoSQL movement

	Relational databases	NoSQL databases
Data paradigm	Relational tables	Key-value (tuple) based Document based Column based Graph based XML, object based Others: time series, probabilistic, etc.
Distribution	Single-node and distributed	Mainly distributed
Scalability	Vertical scaling, harder to scale horizontally	Easy to scale horizontally, easy data replication
Openness	Closed and open source	Mainly open source
Schema role	Schema-driven	Mainly schema-free or flexible schema
Query language	SQL as query language	No or simple querying facilities, or special-purpose languages
Transaction mechanism	ACID: Atomicity, Consistency, Isolation, Durability	BASE: Basically available, Soft state, Eventual consistency
Feature set	Many features (triggers, views, stored procedures, etc.)	Simple API
Data volume	Capable of handling normal-sized data sets	Capable of handling huge amounts of data and/or very high frequencies of read/write requests

Agenda

The NoSQL movement

Key-Value stores

Tuple and Document stores

Column-oriented databases

Graph based databases

Key-value Stores

Key-value based database stores data as (key, value) pairs

- Keys are unique
- Hash map, or hash table or dictionary

Key-value Stores

```
import java.util.HashMap;
import java.util.Map;
public class KeyValueStoreExample {
    public static void main(String... args) {
        // Keep track of age based on name
        Map<String, Integer> age_by_name = new HashMap<>();

        // Store some entries
        age_by_name.put("wilfried", 34);
        age_by_name.put("seppe", 30);
        age_by_name.put("bart", 46);
        age_by_name.put("jeanne", 19);

        // Get an entry
        int age_of_wilfried = age_by_name.get("wilfried");
        System.out.println("Wilfried's age: " + age_of_wilfried);

        // Keys are unique
        age_by_name.put("seppe", 50); // Overrides previous entry
    }
}
```

Key-value Stores

Keys (e.g., “bart”, “seppe”) are hashed by means of a so-called hash function

- A hash function takes an arbitrary value of arbitrary size and maps it to a key with a fixed size, which is called the hash value.
- Each hash can be mapped to a space in computer memory

Key
wilfried
seppe
bart
jeanne



Hash	Key
01	(wilfried, 34)
03	(seppe, 30)
07	(bart, 46)
08	(jeanne, 19)

Key-value Stores

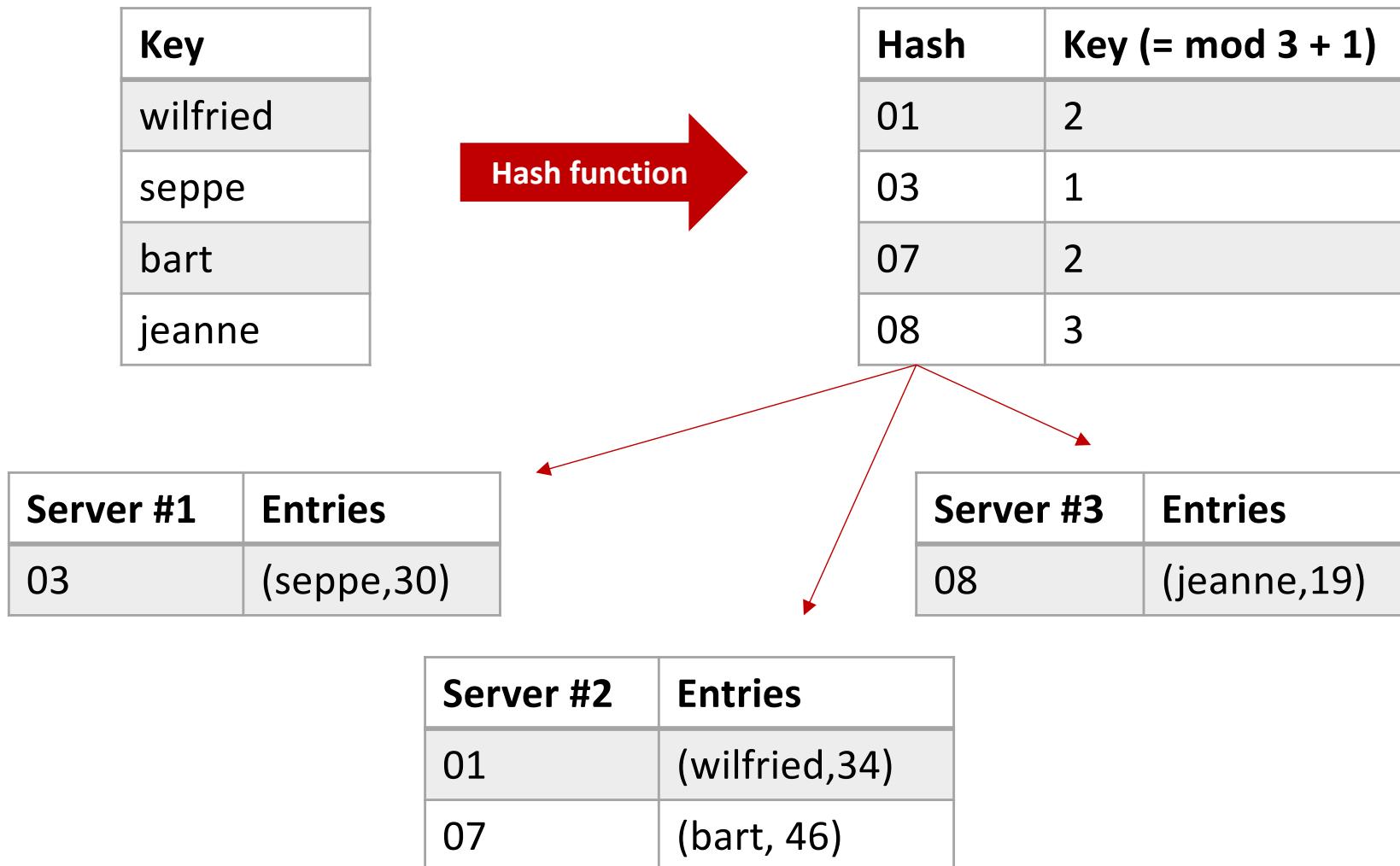
NoSQL databases are built with horizontal scalability support in mind

Distribute hash table over different locations

Assume we need to spread our hashes over three servers

- Hash every key (“wilfried”, “seppe”) to a server identifier
- $\text{index}(\text{hash}) = \text{mod}(\text{hash}, \text{nrServers}) + 1$

Sharding



Key-value Stores

Example: Memcached

- Implements a distributed memory-driven hash table (i.e. a key-value store), which is put in front of a traditional database to speed up queries by caching recently accessed objects in RAM
- Caching solution

Key-value Stores

Request Coordination

Consistent Hashing

Replication and Redundancy

Eventual Consistency

Stabilization

Integrity Constraints and Querying

Request Coordination

In many NoSQL implementations (e.g. Cassandra, Google's BigTable, Amazon's DynamoDB) all nodes implement the same functionality and are all able to perform the role of request coordinator

Need for membership protocol

- Dissemination
 - Based on periodic, pairwise communication
- Failure detection

Consistent Hashing

Consistent hashing schemes are often used, which avoid having to remap each key to a new node when nodes are added or removed

Suppose we have a situation where 10 keys are distributed over 3 servers ($n = 3$) with the following hash function

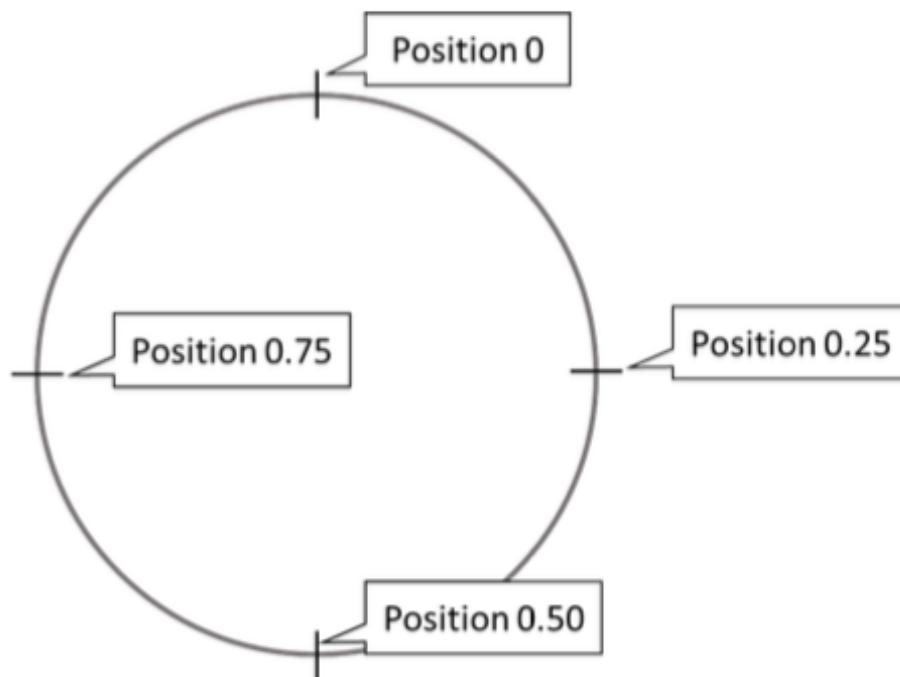
- $h(\text{key}) = \text{key} \bmod n$

Consistent Hashing

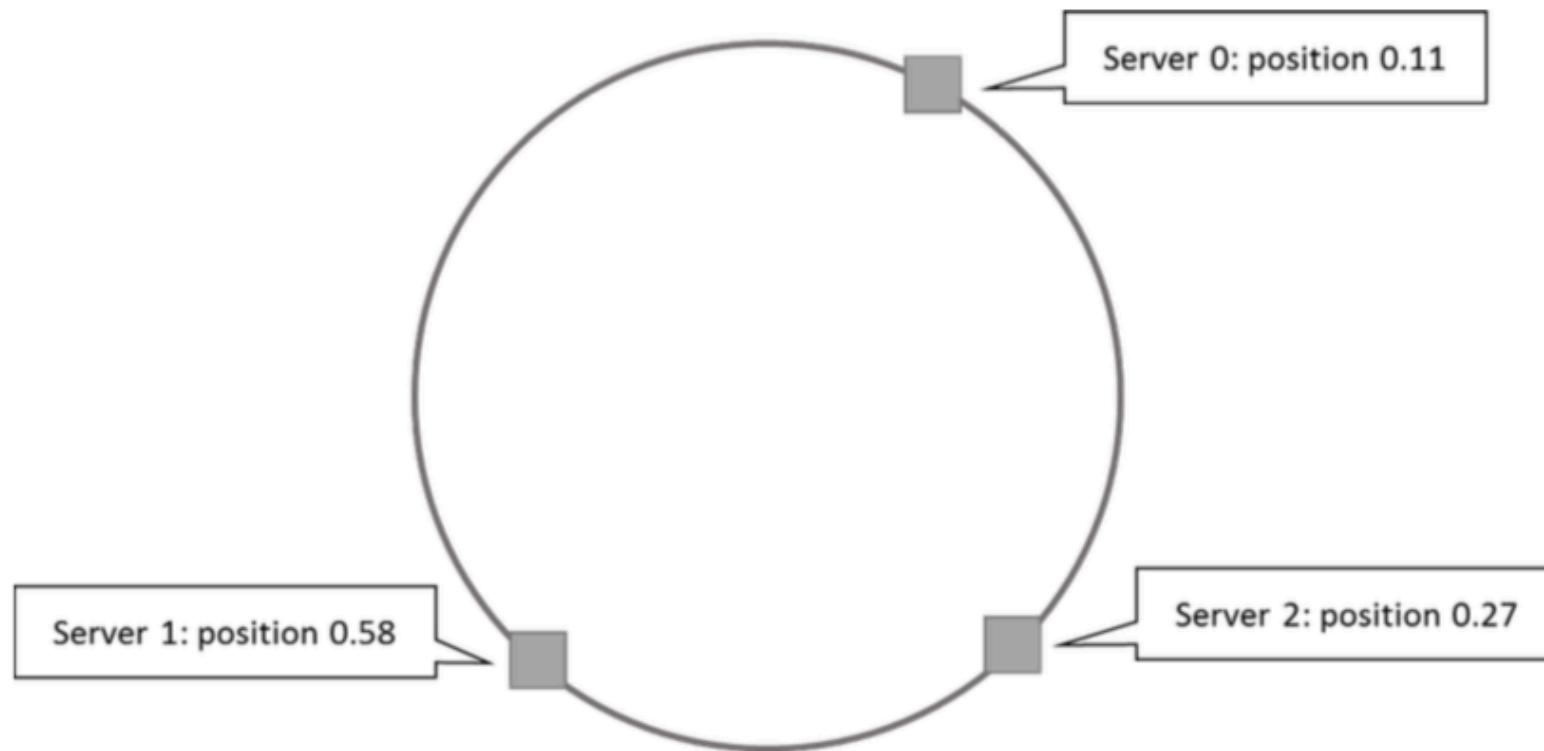
	n		
key	3	2	4
0	0	0	0
1	1	1	1
2	2	0	2
3	0	1	3
4	1	0	0
5	2	1	1
6	0	0	2
7	1	1	3
8	2	0	0
9	0	1	1

Consistent Hashing

At the core of a consistent hashing setup is a so called “ring”-topology, which is basically a representation of the number range $[0,1]$:

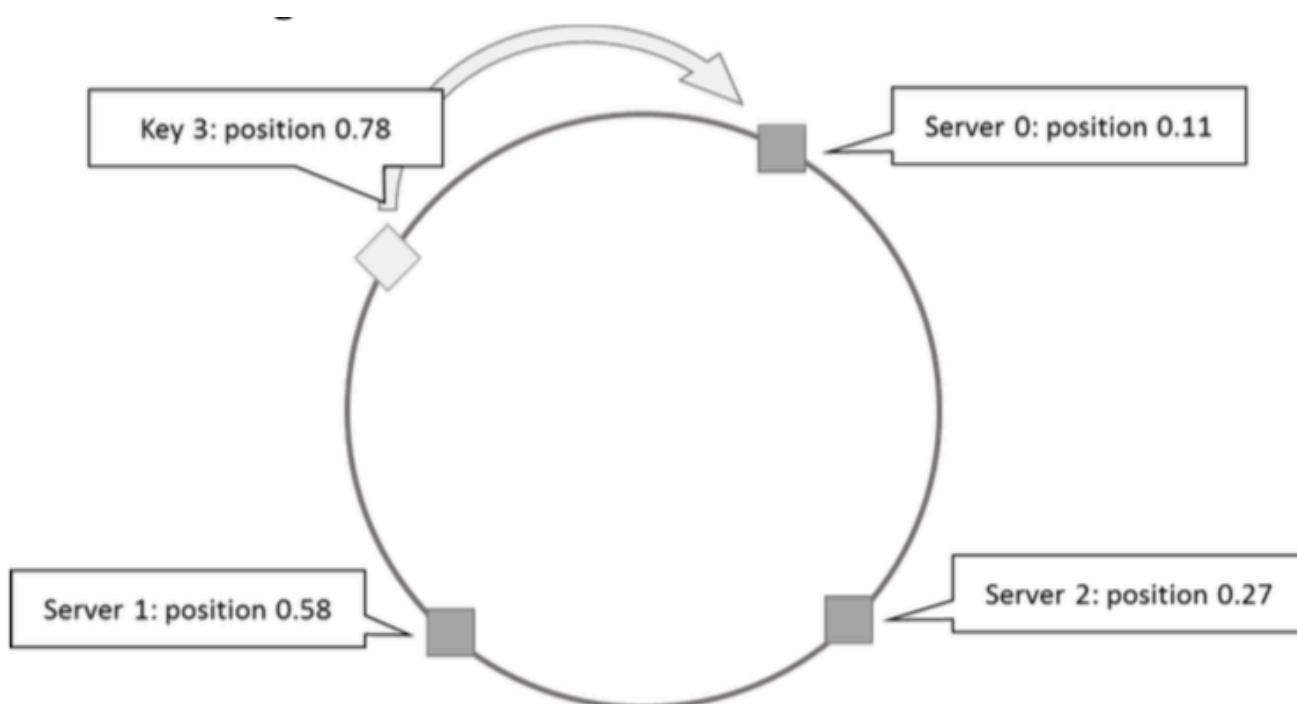


Consistent Hashing



Consistent Hashing

Hash each key to a position on the ring, and store the actual key-value pair on the first server that appears clockwise of the hashed point on the ring

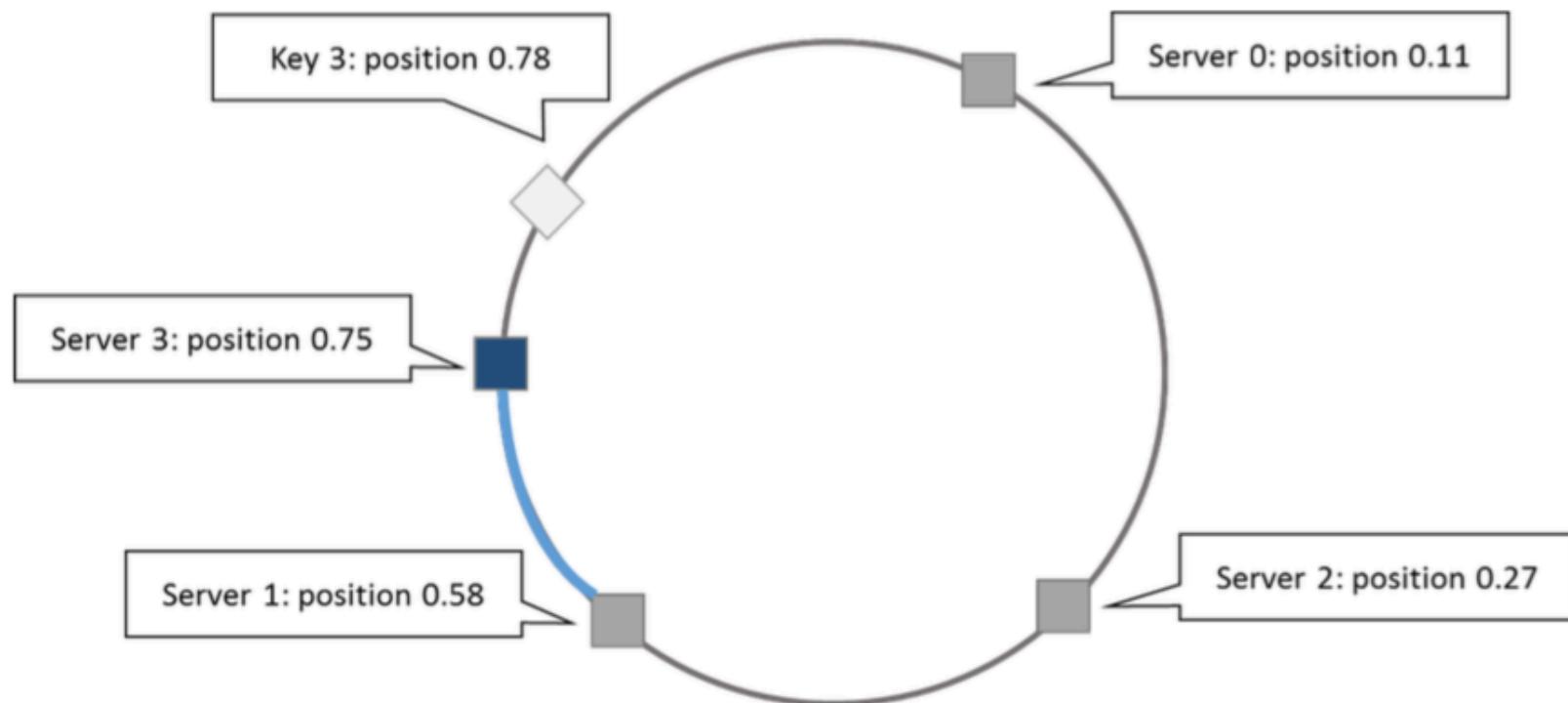


Consistent Hashing

Because of the uniformity property of a “good” hash function, roughly $1/n$ of key-value pairs will end up being stored on each server

Most of the key-value pairs will remain unaffected in case a machine is added or removed

Consistent Hashing



Replication and Redundancy

Problems with consistent hashing:

- If 2 servers end up being mapped close to one another, one of these nodes will end up with few keys to store
- In case a server is added, all of the keys moved to this new node originate from just one other server

Instead of mapping a server s to a single point on our ring, we map it multiple positions, called **replicas**

For each physical server s , we hence end up with r (the number of replicas) points on the ring

Note: each of the replicas still represents the same physical instance (<-> redundancy)

- Virtual nodes

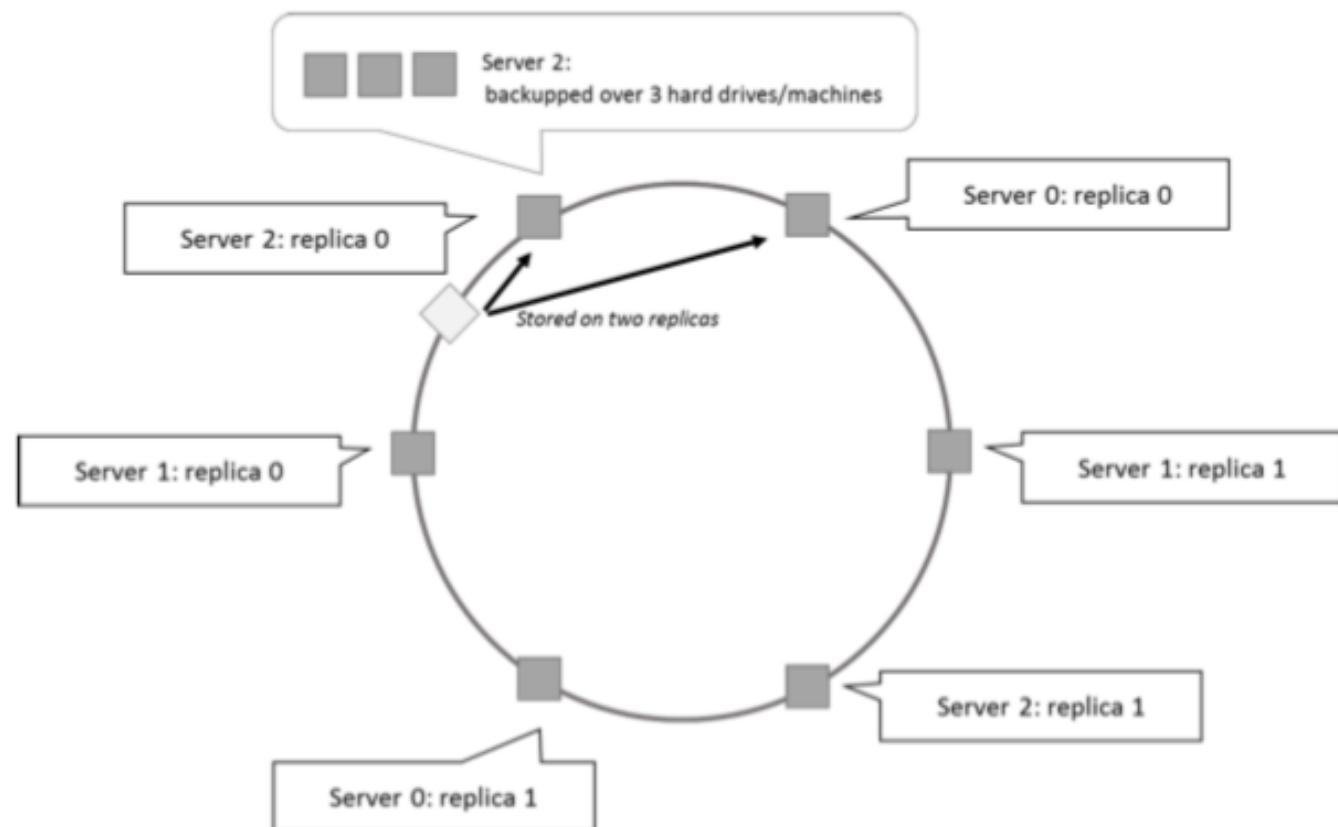
Replication and Redundancy

To handle data replication or redundancy, many vendors extend the consistent hashing mechanism so that key-value pairs are duplicated across multiple nodes

- E.g., by storing the key-value pair on two or more nodes clockwise from the key's position on the ring

Replication and Redundancy

It is also possible to set up a full redundancy scheme where each node itself corresponds to multiple physical machines each storing a fully redundant copy of the data



Eventual Consistency

Membership protocol does not guarantee that every node is aware of every other node *at all times*

- It will reach a consistent state over time

State of the network might not be perfectly consistent at any moment in time, though will become eventually consistent at a future point in time

Many NoSQL databases guarantee so called **eventual consistency**

Eventual Consistency

Most NoSQL databases follow the BASE principle

- Basically available, Soft state, Eventual consistency

CAP theorem states that a distributed computer system cannot guarantee the following three properties at the same time:

- Consistency (all nodes see the same data at the same time)
- Availability (guarantees that every request receives a response indicating a success or failure result)
- Partition tolerance (the system continues to work even if nodes go down or are added).

Eventual Consistency

Most NoSQL databases sacrifice the consistency part of CAP in their setup, instead striving for eventual consistency

The full BASE acronym stands for:

- Basically available: NoSQL databases adhere to the availability guarantee of the CAP theorem
- Soft state: the system can change over time, even without receiving input
- Eventual consistency: the system will become consistent over time

Stabilization

The operation which repartitions hashes over nodes in case nodes are added or removed is called **stabilization**

If a consistent hashing scheme being applied, the number of fluctuations in the hash-node mappings will be minimized.

Integrity Constraints and Querying

Key value stores represent a very diverse gamut of systems

Full blown DBMSs versus caches

Only limited query facilities are offered

- E.g. put and set

Limited to no means to enforce structural constraints

- DBMS remains agnostic to the internal structure

No relationships, referential integrity constraints or database schema, can be defined

Agenda

The NoSQL movement

Key-Value stores

Tuple and Document stores

Column-oriented databases

Graph based databases

Tuple and Document Stores

A tuple store is similar to a key-value store, with the difference that it does not store pairwise combinations of a key and a value, but instead stores a unique key together with a vector of data

Example:

- marc -> ("Marc", "McLast Name", 25, "Germany")

No requirement to have the same length or semantic ordering (schema-less!)

Tuple and Document Stores

Various NoSQL implementations do, however, permit organizing entries in semantical groups, (aka collections or tables)

Examples:

- Person:marc -> ("Marc", "McLast Name", 25, "Germany")
- Person:harry -> ("Harry", "Smith", 29, "Belgium")

Tuple and Document Stores

Document stores store a collection of attributes that are labeled and unordered, representing items that are semi-structured

Example:

```
{  
    Title = "Harry Potter"  
    ISBN = "111-11111111"  
    Authors = [ "J.K. Rowling" ]  
    Price = 32  
    Dimensions = "8.5 x 11.0 x 0.5"  
    PageCount = 234  
    Genre      = "Fantasy"  
}
```

Tuple and Document Stores

Most modern NoSQL databases choose to represent documents using JSON

```
{  
    "title": "Harry Potter",  
    "authors": ["J.K. Rowling", "R.J. Kowling"],  
    "price": 32.00,  
    "genres": ["fantasy"],  
    "dimensions": {  
        "width": 8.5,  
        "height": 11.0,  
        "depth": 0.5  
    },  
    "pages": 234,  
    "in_publication": true,  
    "subtitle": null  
}
```

Agenda

The NoSQL movement

Key-Value stores

Tuple and Document stores

Column-oriented databases

Graph based databases

Column-oriented Databases

A column-oriented DBMS is a database management system that stores data tables as sections of columns of data

Useful if

- aggregates are regularly computed over large numbers of similar data items
- data is sparse, i.e. columns with many null values

Can also be an RDBMS, key-value or document store

Column-oriented Databases

Example

id	Genre	Title	Price	Audiobook price
1	fantasy	My first book	20	30
2	education	Beginners guide	10	null
3	education	SQL strikes back	40	null
4	fantasy	The rise of SQL	10	null

Row based databases are not efficient at performing operations that apply to the entire data set

- Need indexes which add overhead

Column-oriented Databases

In a column-oriented database, all values of a column are placed together on disk

Genre: fantasy:1,4 education:2,3

Title: My first book:1 Beginners guide:2 SQL strikes back:3 The rise of SQL:4

Price: 20:1 10:2,4 40:3

Audiobook price: 30:1

A column matches the structure of a normal index in a row-based system

Operations such as: find all records with price equal to 10 can now be executed directly

Null values do not take up storage space anymore

Column-oriented Databases

Disadvantages

- Retrieving all attributes pertaining to a single entity becomes less efficient
- Join operations will be slowed down

Examples

- Google BigTable, Cassandra, HBase, and Parquet

Agenda

The NoSQL movement

Key-Value stores

Tuple and Document stores

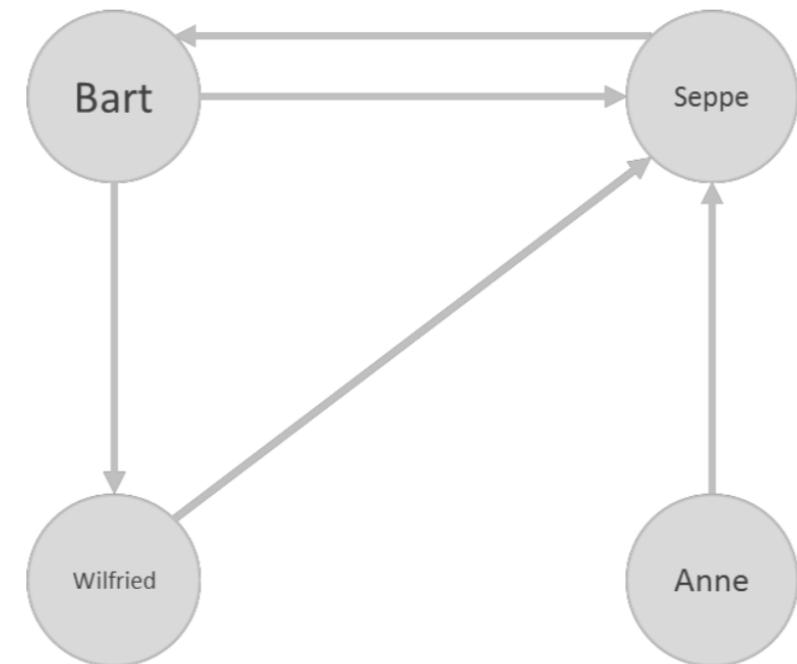
Column-oriented databases

Graph based databases

Graph based Databases

Graph databases apply graph theory to the storage of information of records

Graphs consist of nodes and edges



Graph based Databases

One-to-one, one-to-many, and many-to-many structures can easily be modeled in a graph

Consider N-M relationship between books and authors

RDBMS needs 3 tables: Book, Author and Books_Authors

SQL query to return all book titles for books written by a particular author would look like follows

```
SELECT title  
FROM books, authors, books_authors  
WHERE author.id = books_authors.author_id  
AND books.id = books_authors.book_id AND author.name = "Bart Baesens"
```

Graph based Databases

In a graph database (using **Cypher query language** from Neo4j)



```
MATCH (b:Book) <- [ :WRITTEN_BY ] - (a:Author)
```

```
WHERE a.name = "Bart Baesens"
```

```
RETURN b.title
```

Graph based Databases

A graph database is a hyper-relational database, where JOIN tables are replaced by more interesting and semantically meaningful relationships that can be navigated and/or queried using graph traversal based on graph pattern matching.

Graph databases

Location-based services

Recommender systems

Social media (e.g. Twitter and FlockDB)

Knowledge based systems

NoSQL DBMS

	RDBMS	NoSQL
Relational	Yes	No
SQL	Yes	No
Column stores	No	Yes
Scalability	Limited	Yes
Eventually consistent	Yes	Yes
BASE	No	Yes
Big volumes of data	No	Yes
Schema-less	No	Yes

Kahoot time!

Any doubts?