

# Database Systems

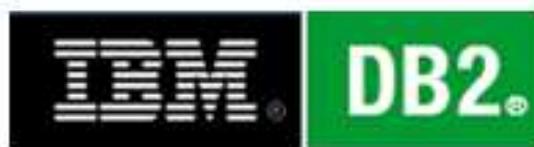
Sven Helmer

# Chapter 1

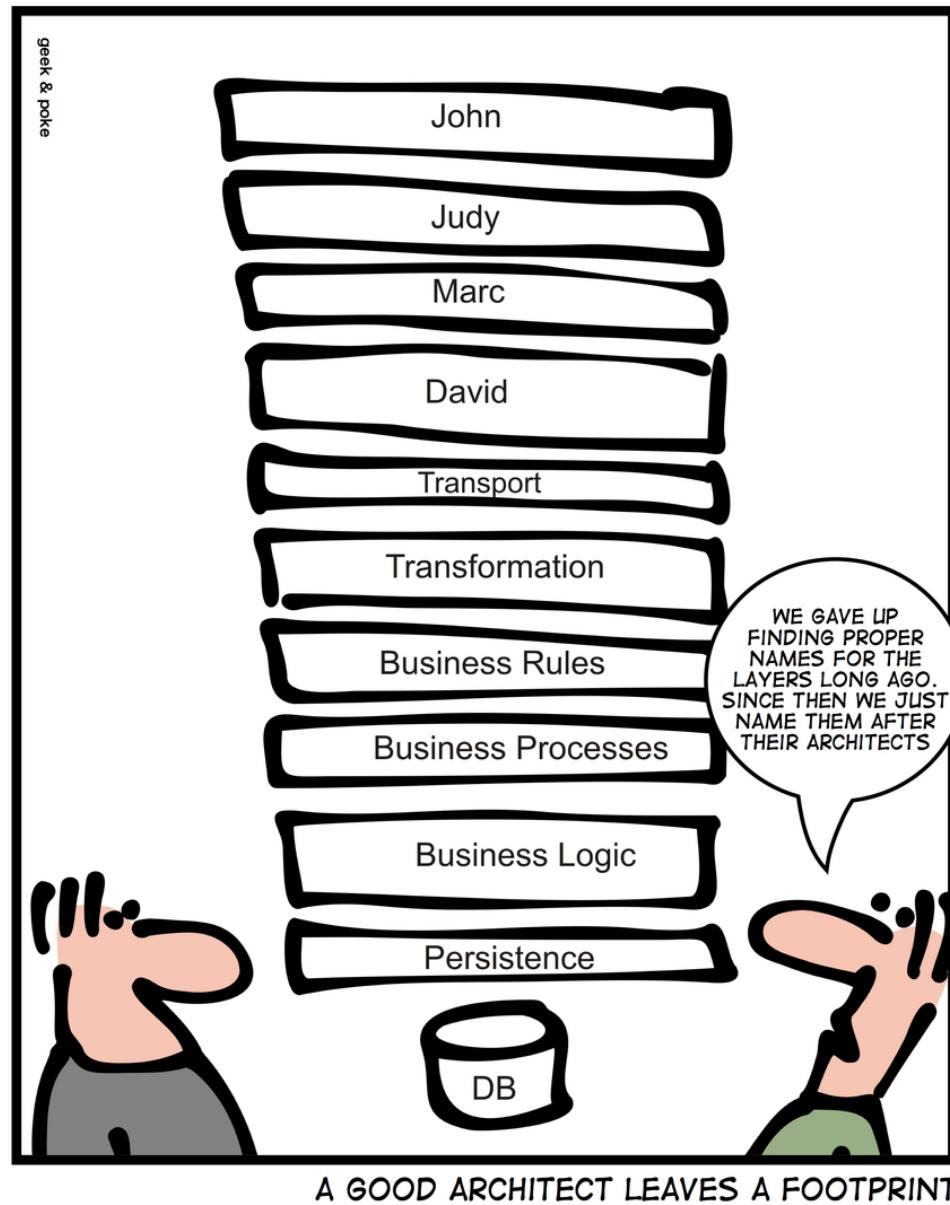
## Introduction and Motivation

# What?

- What is a database system (DBS)?
  - Obviously a system for storing and managing data
- You have probably heard about some of the big players:

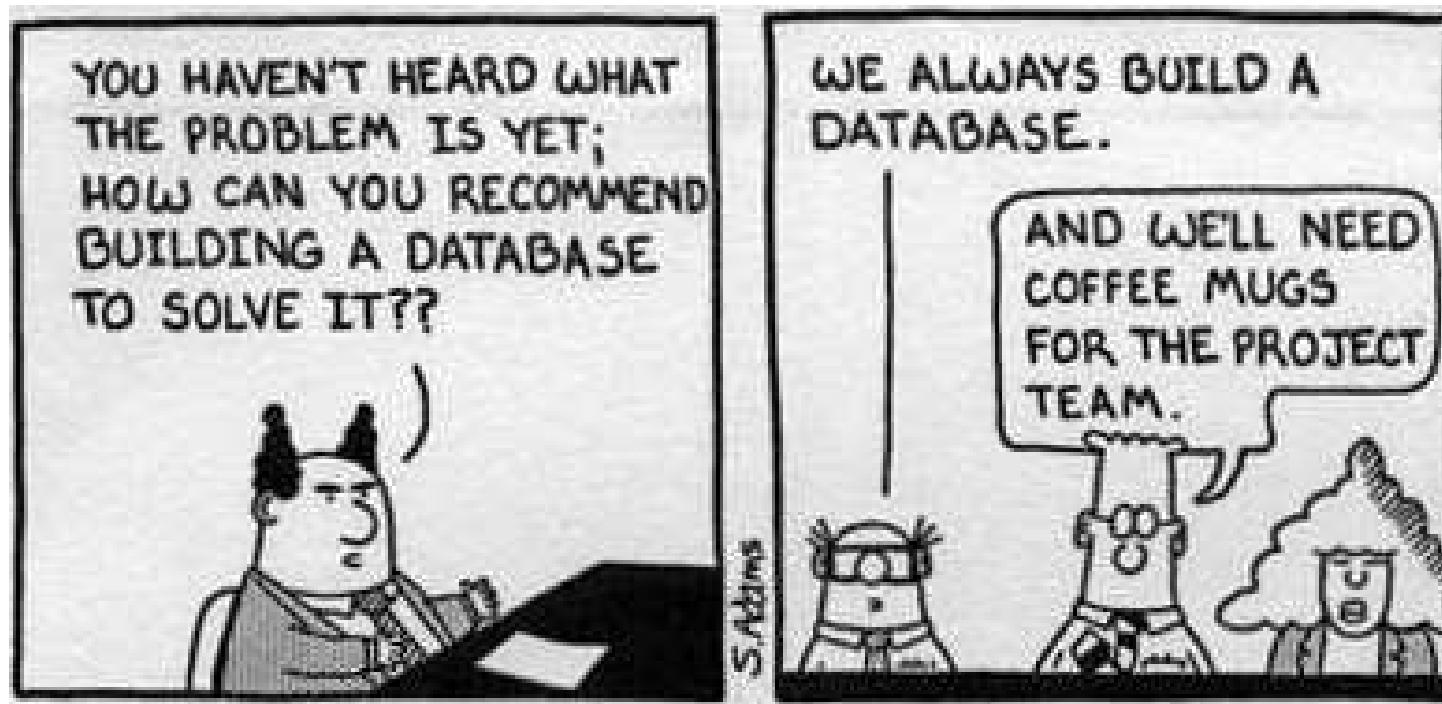


# Where?



# Why?

- But why use a database system?
- Why not just use a file system?



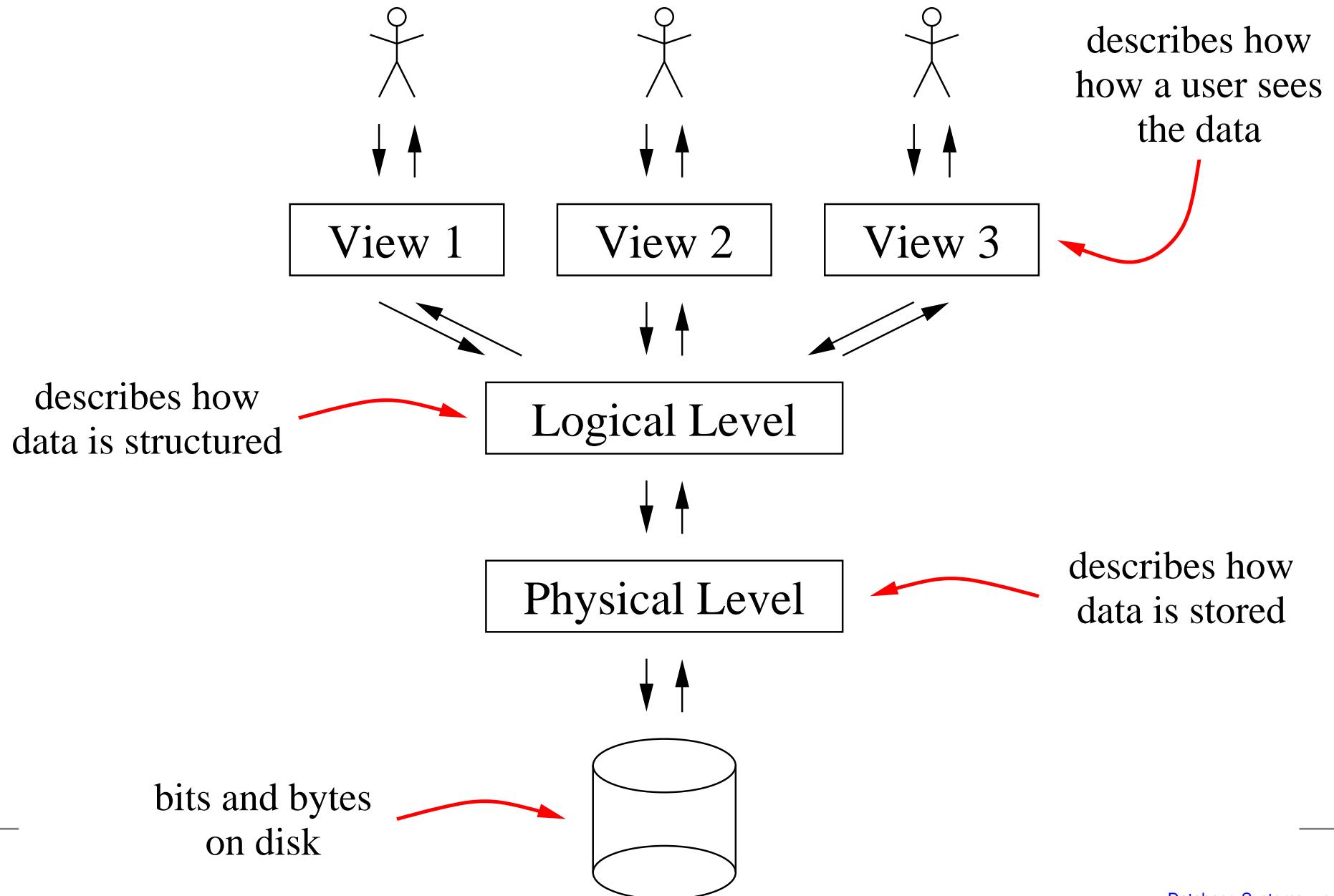
# Typical Problems without a DBS

- Redundancy and inconsistency
- Various different file formats
- Issues with synchronizing concurrent access
- Data loss
- Violation of data integrity
- Problems with security
- Costs for developing applications

# Advantages of Using a DBS

- Data independence
- Declarative query languages
- Multi-user synchronization
- Recovery
- Safeguarding data integrity
- Efficiency and scalability

# Data Independence



# Data Independence(2)

- DBS decouples applications from the structure and storage schema of the data
- Logical data independence
  - Changes on the logical level have no impact on the application
- Physical data independence
  - Changes on the physical level have no impact on the logical level
  - This is implemented in (almost) all database systems

# Declarative Query Languages

- Users tell DBS *what* data they want ...

```
select name, address  
from persons;
```

- ... and not *how* the data should be fetched

```
[MV_UI4_C_C 1 0  
LOAD_SF8_C 4 1 6  
LOAD_SF8_C 5 1 7  
LOAD_SF8_C 6 1 8  
...]
```

- Less error-prone, as a user does not need to know about physical layer

# Multi-user Synchronization

- Multiple users modifying the same data without any controls can lead to big problems



- DBS puts mechanisms in place to synchronize concurrent access ...
  - ... so that there will be no unwanted side effects

# Recovery

- Sometimes database systems crash



- DBS can reconstruct state immediately before crash
- DBS creates and maintains log files to do this

# Data Integrity



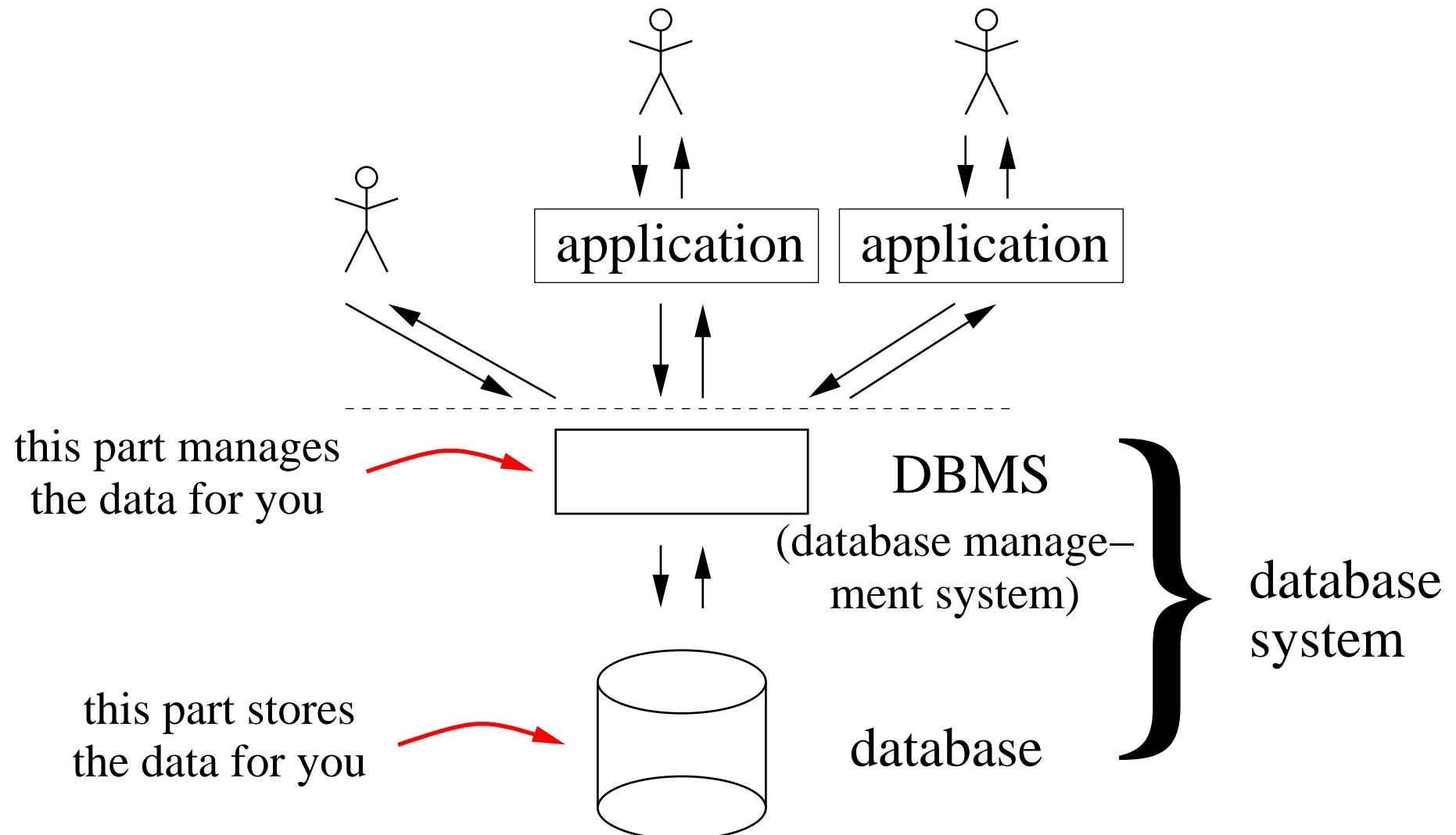
- Data processing should not be “anything goes”, but should follow certain procedures
- Administrators (and users) can set up constraints that DBS will follow
- This helps in avoiding
  - user mistakes
  - programming errors

# Efficiency and Scalability

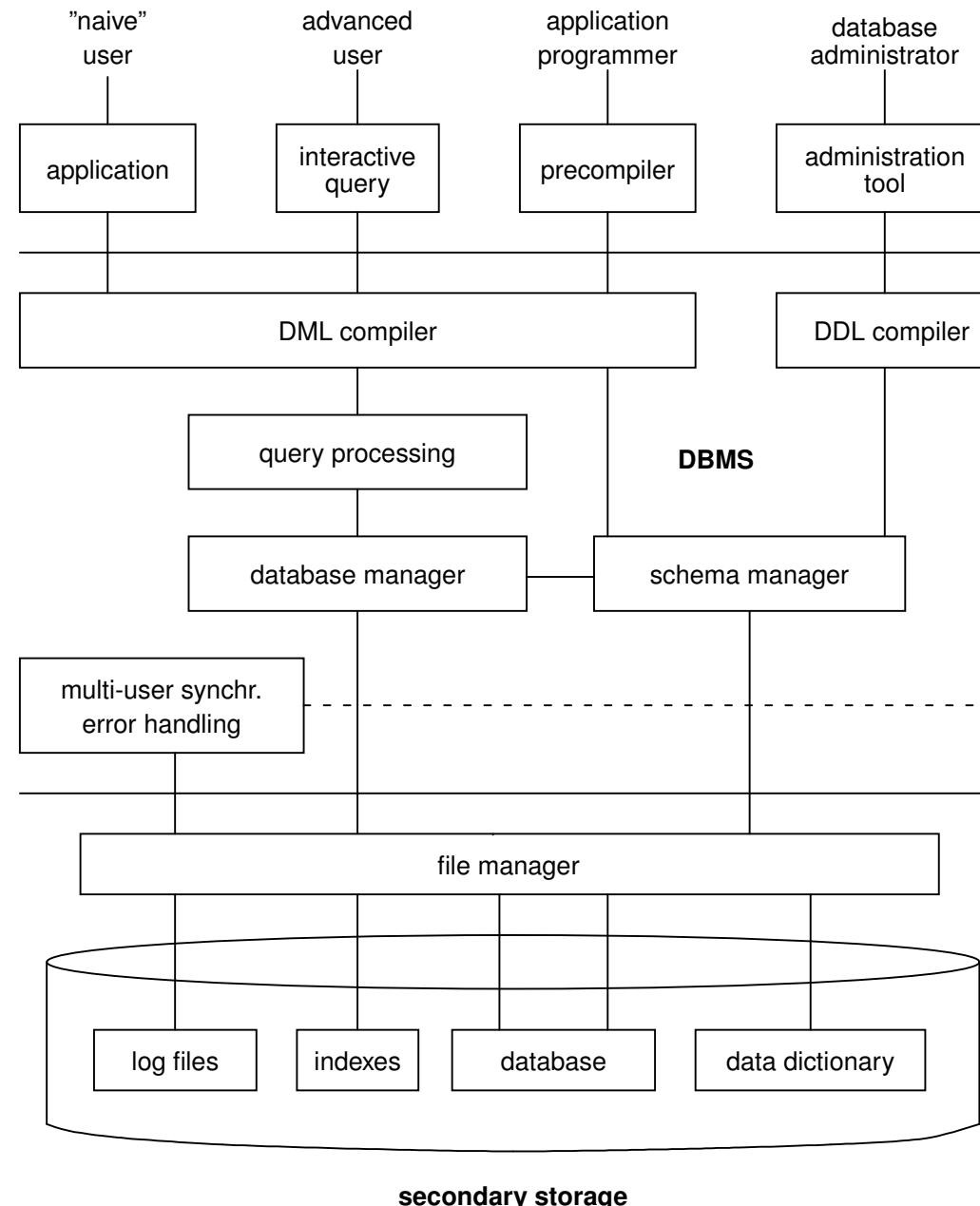
- DBSs are built for large-scale applications:
  - that means, they can handle large volumes of data



# (Rough) Architecture of a DBS



# More Details



secondary storage

# Data Modeling

- Even though a DBS can do a lot, it won't
  - wash the family dog
  - cook breakfast
- More specifically:
  - A user has to come up with the requirements of an application ...
  - ... and specify the data that is to be stored

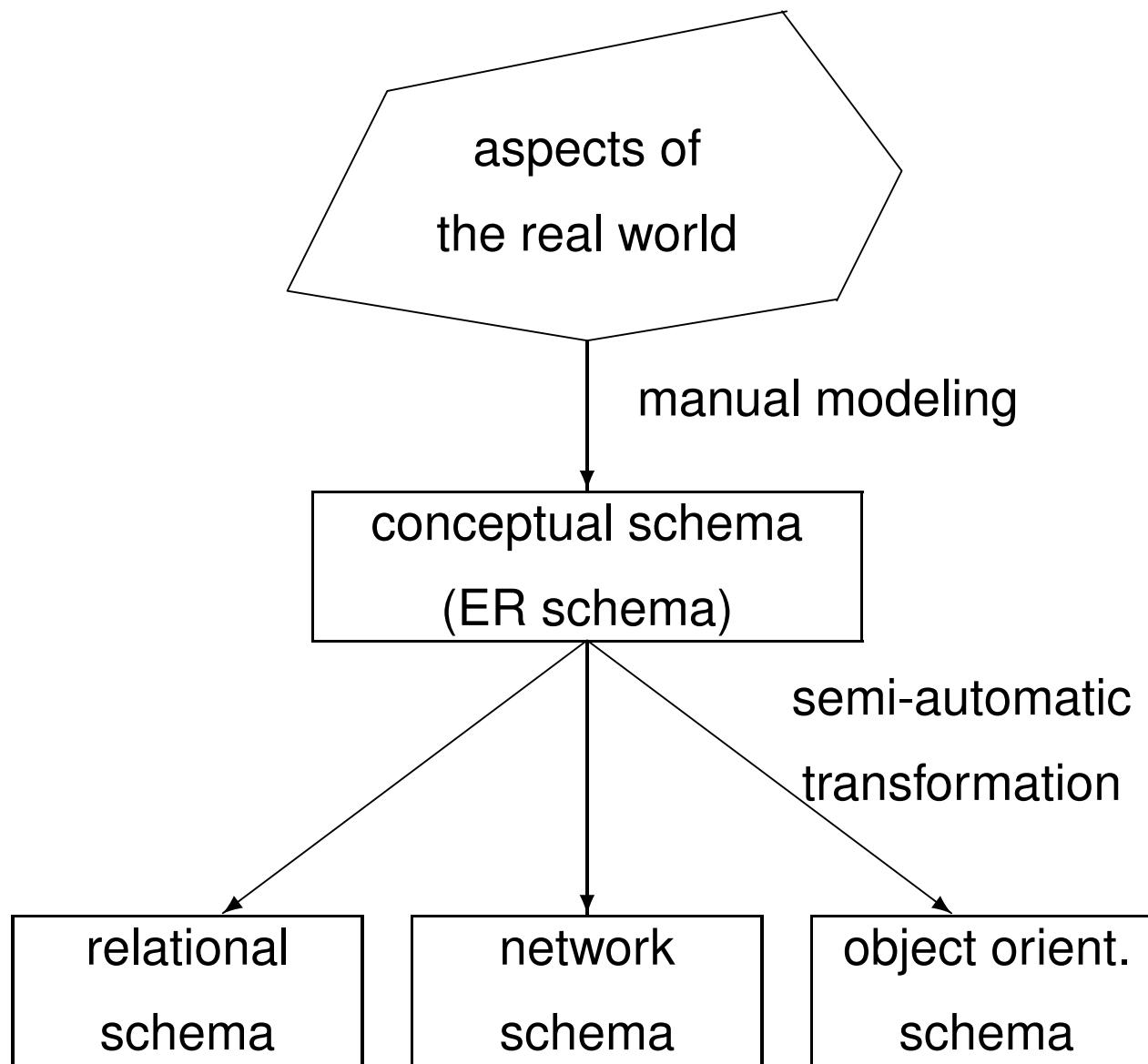
# Data Modeling(2)

- Two important concepts for designing database applications
  - Data model: determines which constructs are available for describing data
    - Provides a language in which to describe the structure of data
  - Schema: a concrete description of a particular data collection
    - Uses a particular data model for the description

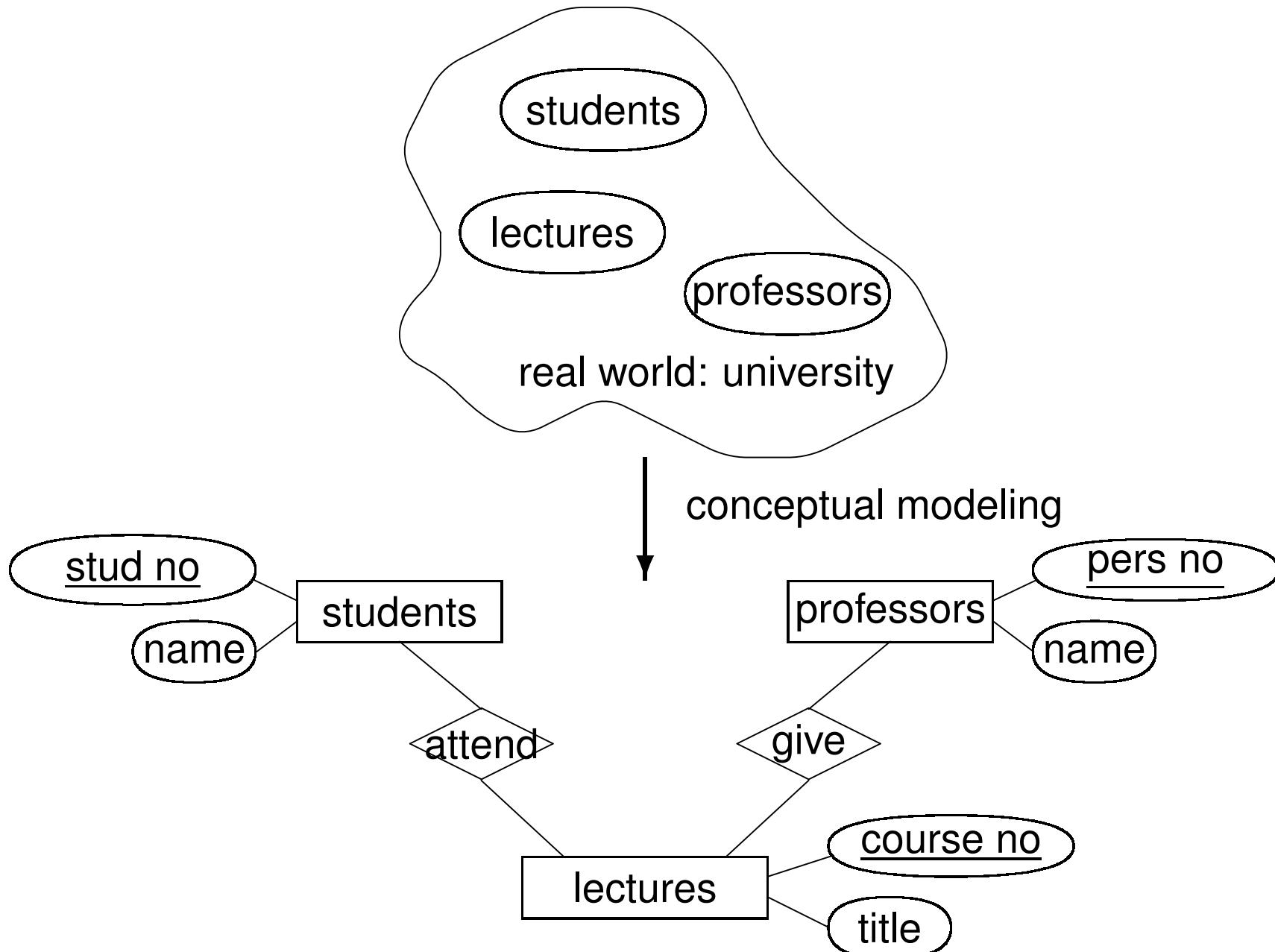
# Data Models

- Conceptual models:
  - Entity Relationship Model (ER Model)
  - Unified Modeling Language (UML)
- Logical Model:
  - Hierarchical model
  - Network model
  - Relational model
  - Object-oriented model
  - Object-relational model
  - Semistructured (XML) model

# Modeling Steps



# Example



# Example(2)

relational schema

students		attend	
stud_no	name	stud_no	course_no
26120	Smith	25403	5022
25403	Jones	26120	5001
....	....	....	....

lectures	
course_no	title
5001	Introduction to Prog.
5022	Algorithms
....	....

# Summary

- Many applications have similar requirements in terms of data storage
- A DBS provides an infrastructure supporting these applications



# Overview of Course

- We cover two aspects:
  - How do you use a DBS and design/develop applications for DBSs?
  - What happens behind the scenes in a DBS?

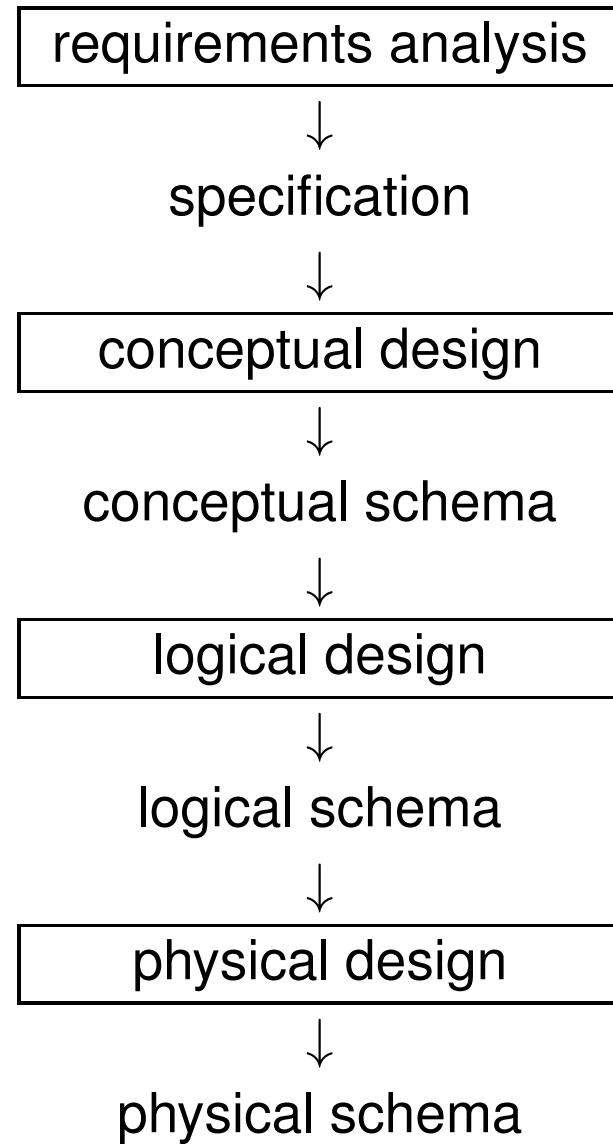
# Overview of Course(2)

- Content:
  - Database design (chapter 2)
  - Relational model (chapter 3)
  - SQL (chapter 4)
  - Active Databases (chapter 5)
  - Relational design theory (chapter 6)
  - Physical data storage (chapter 7)
  - Query processing (chapter 8)
  - Transaction management (chapter 9)
  - Recovery (chapter 10)
  - Multi-user synchronization (chapter 11)
  - Databases and security (chapter 12)

# Chapter 2

## Database Design

# Database Design Phases



# Requirements Analysis

- Not that much different from what you learn in software engineering courses
- So we only briefly touch on this topic here
- One of the most important points: users, developers, and other stakeholders communicate effectively
- Unfortunately, reality often looks different . . .

# Real-world Development



How the customer explained it



How the Project Leader understood it



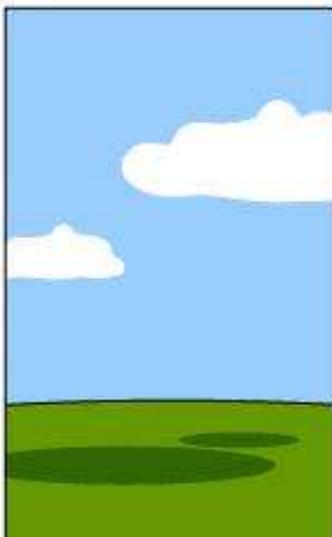
How the Analyst designed it



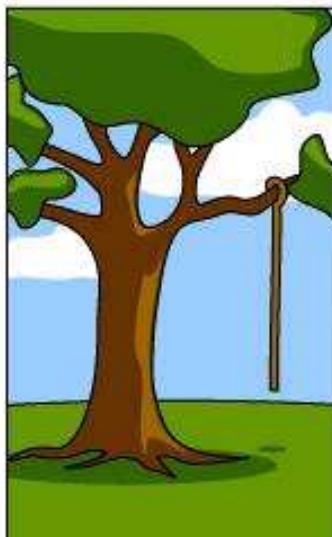
How the Programmer wrote it



How the Business Consultant described it



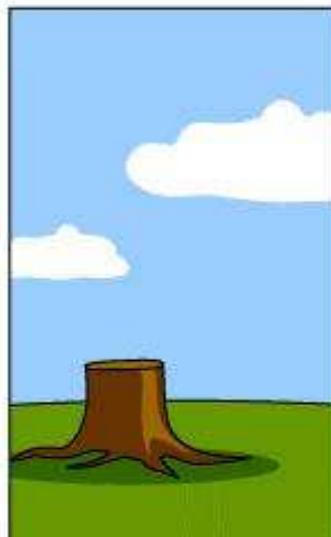
How the project was documented



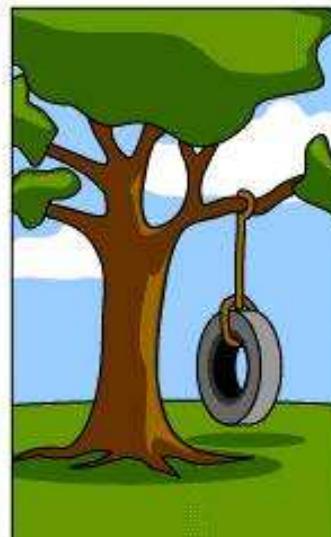
What operations installed



How the customer was billed



How it was supported



What the customer really needed

# Specification

- Here the *what* is defined (not the *how*)
- Important aspects of specifications include
  - Process descriptions
  - Use cases
  - User interfaces
  - Interfaces to other systems
  - ...

# Specification(2)

- The ideal specification is
  - unambiguous
  - complete
  - understandable (by all involved parties)
  - without redundancies
  - ... and in reality not achievable

# Methodologies

- A large selection of different methodologies is available:
  - Structured Analysis
  - Object-oriented Analysis/Object-oriented Design
  - Prototyping
  - ...
- It depends on the project which methodology makes most sense
- Coming up with requirements is often an iterative (and political) process

# Data Modeling

- Once the general requirements are clear, we have to look at the data that will be stored in the database
- There are special ways of doing this, starting with a conceptual design



# Conceptual Design

- The conceptual design is often done using the Entity-Relationship Model (ER-Model)
- An ER-schema is a graphical representation of the conceptual data modeling
- It identifies *entities* and their *relationships*



# Conceptual Design(2)

- An ER-schema
  - describes all the data we need in the database
  - is simple enough to show to users (to check that it is correct and complete)
  - can be transformed semi-automatically into a logical schema

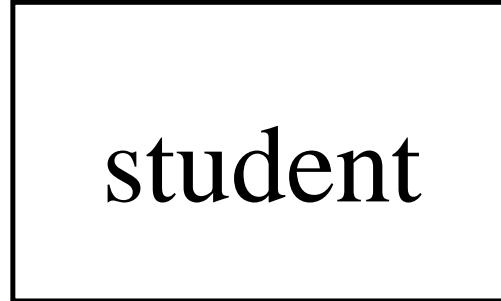
# Basic Building Blocks

- Entities
- Attributes
- Keys
- Relationships
- Roles



# Entities

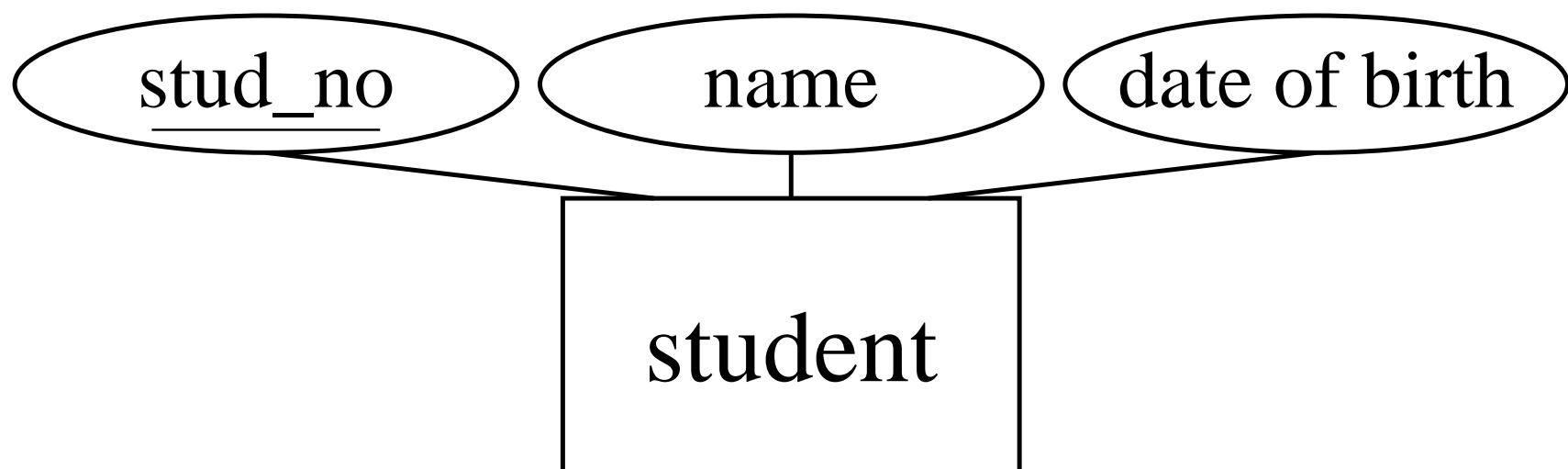
- An *entity* is an abstract object of some sort (e.g. a thing, a person, a place)
- A collection of similar entities forms an *entity set*
- Entity sets are represented by rectangles:



student

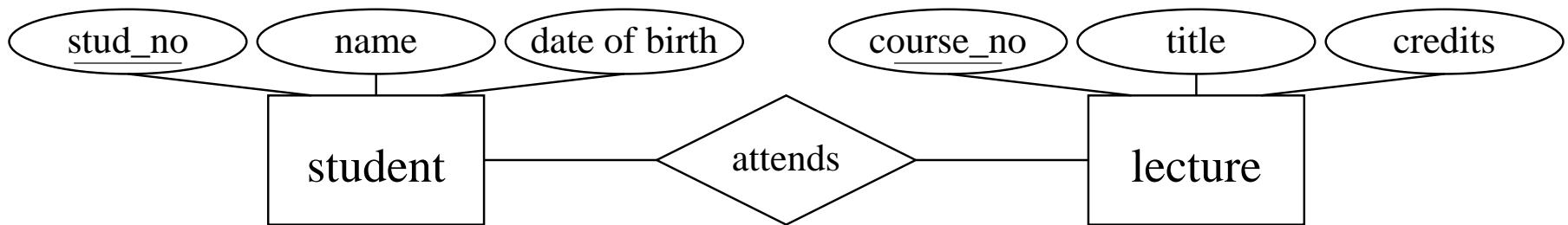
# Attributes

- Common properties of entities in an entity set are described by *attributes*
- Every attribute has a *domain*, which defines the possible values of this attribute (e.g. integer, string)
- An attribute that uniquely identifies an entity is called a *key*
- Attributes are represented by ellipses:



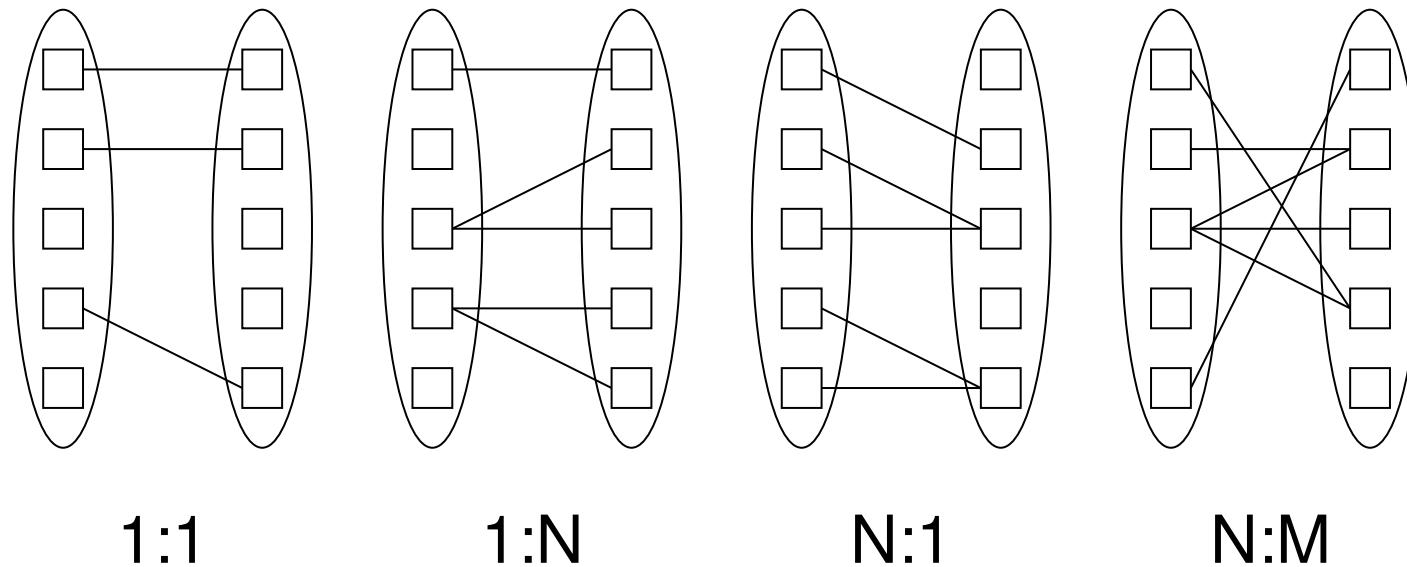
# Relationships

- A *relationship* connects two or more entity sets
- Represented by a diamond
- Example of a binary relationship:



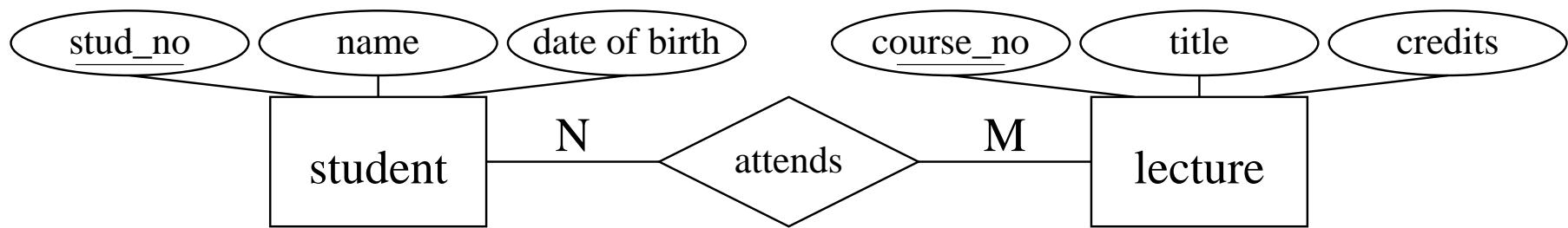
# Multiplicities

- Theoretically, an entity can be connected to an arbitrary number of entities from another entity set
- However, there are often restrictions, which are described by *multiplicities*



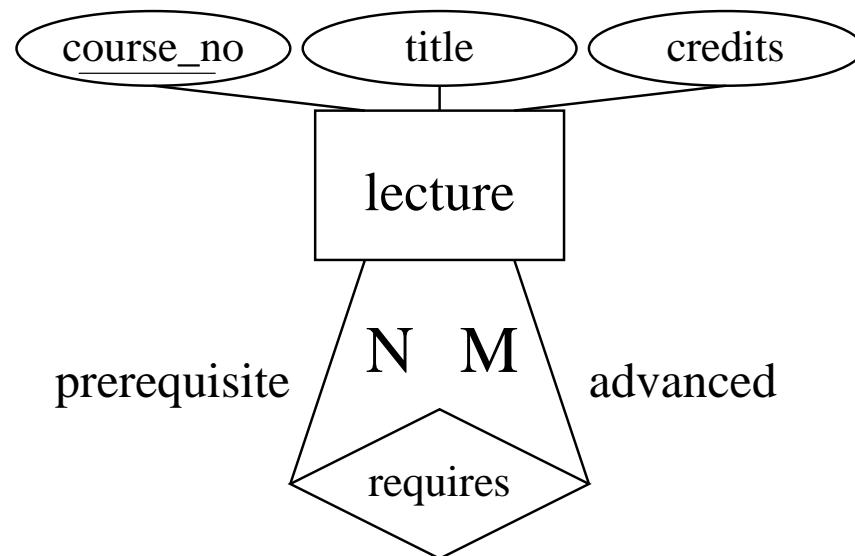
# Multiplicities(2)

- attends is an N:M relationship:

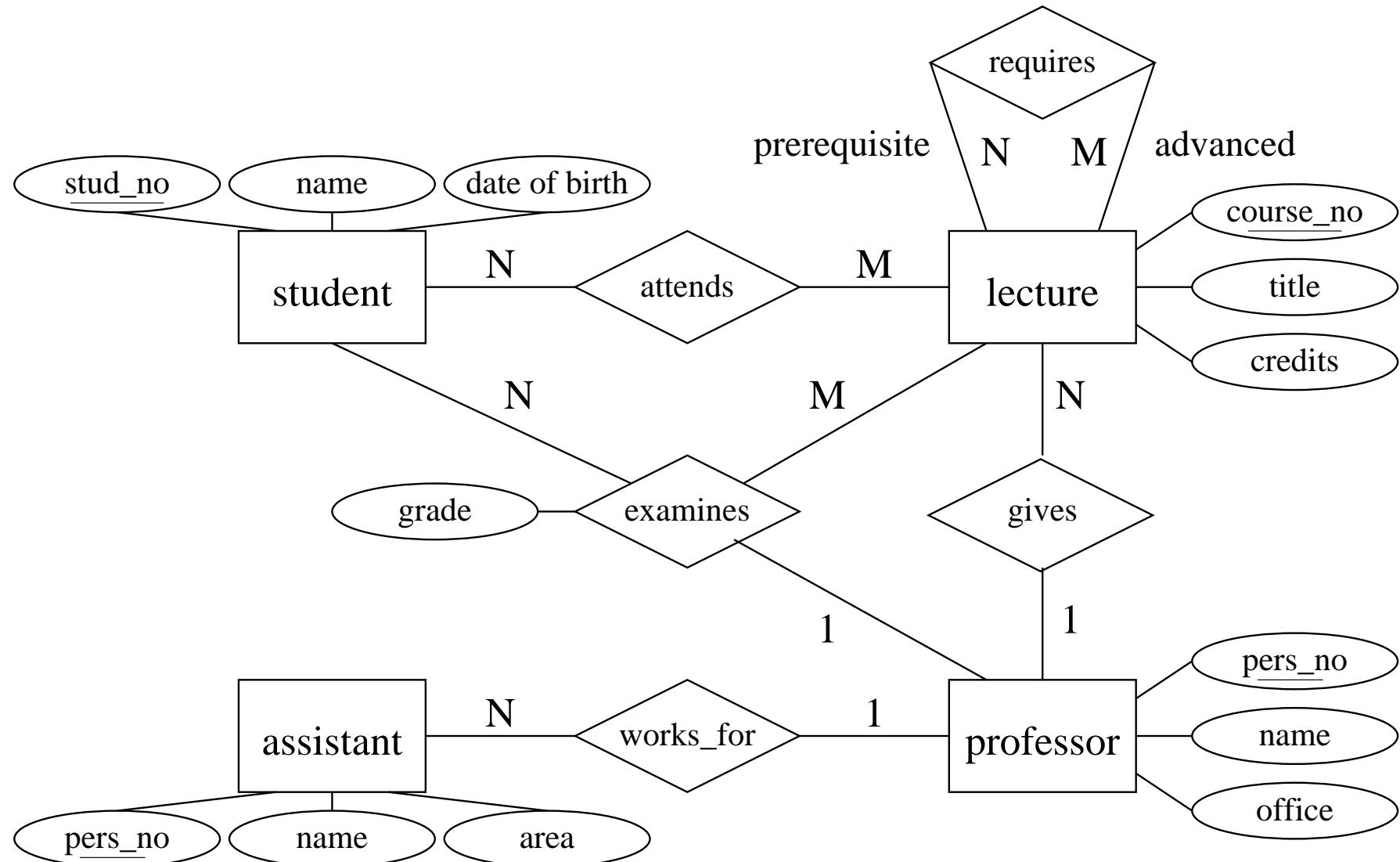


# Roles

- It is also possible to describe which *role* an entity set plays in a relationship
- Useful when an entity set appears more than once in a relationship
  - For example, in recursive relationships:



# The Full ER Schema



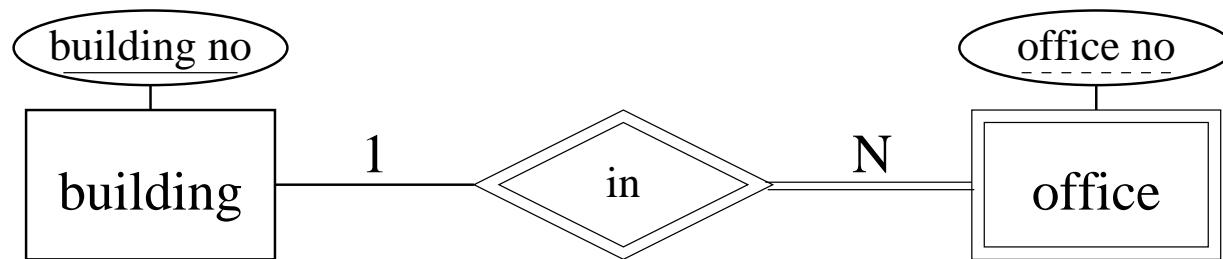
# Advanced Concepts

- Weak Entities
- Generalizations
- Aggregations



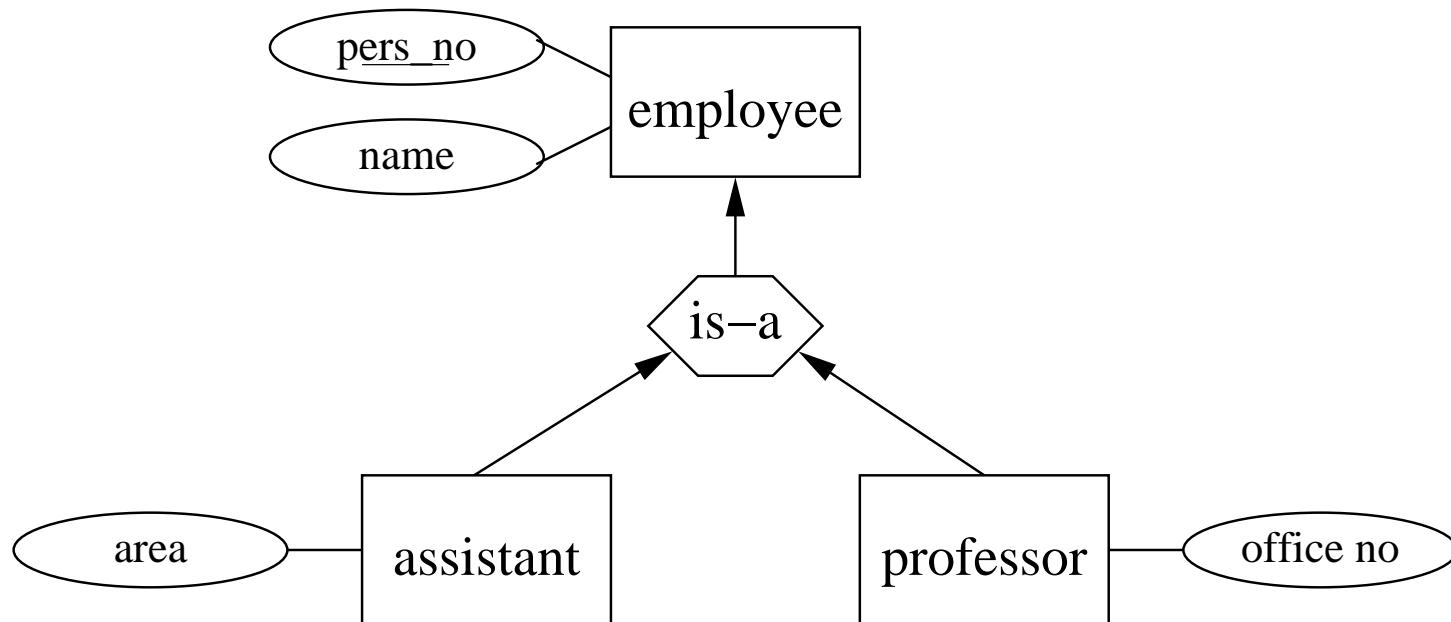
# Weak Entities

- *Weak entities* are entities that cannot exist on their own
- They are associated with a “strong” entity
- A weak and the corresponding strong entity enter into a 1:N (or, sometimes, 1:1) relationship



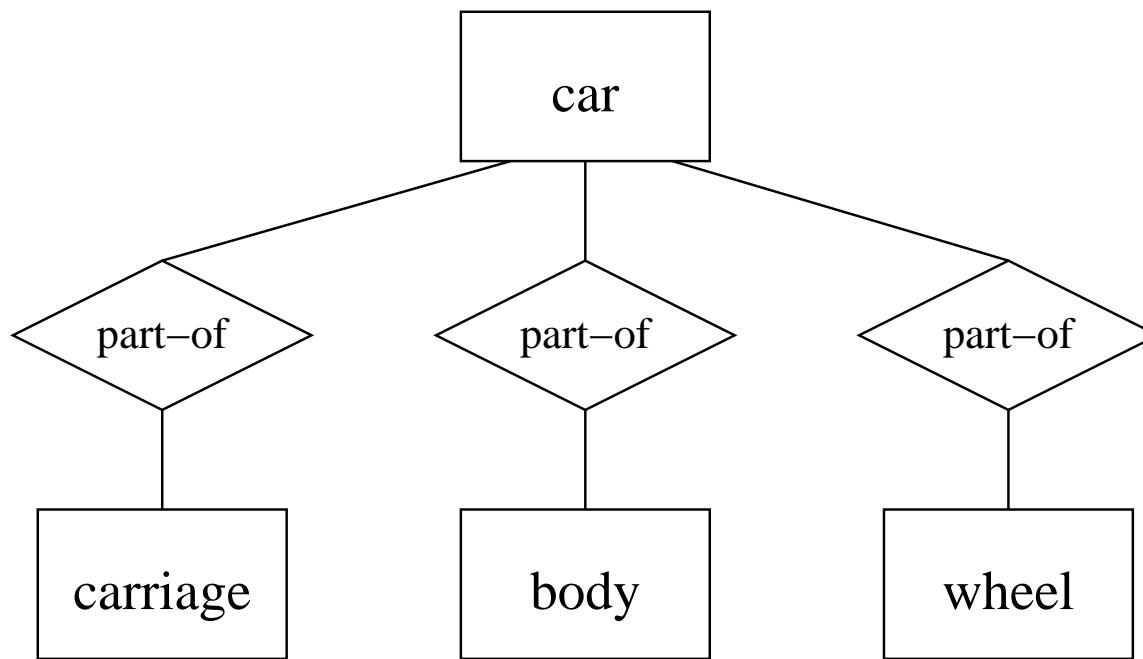
# Generalizations

- A *generalization* is a concept taken from the object-oriented world
- An entity subclass can inherit attributes from another entity set



# Aggregations

- *Aggregations* model a part-of relationship



# Design Decisions

- There is often more than one way to model something
- A designer is often confronted with the following situations:
  - Entities vs attributes
  - Entities vs relationships
  - Binary vs ternary relationships
  - Connection traps

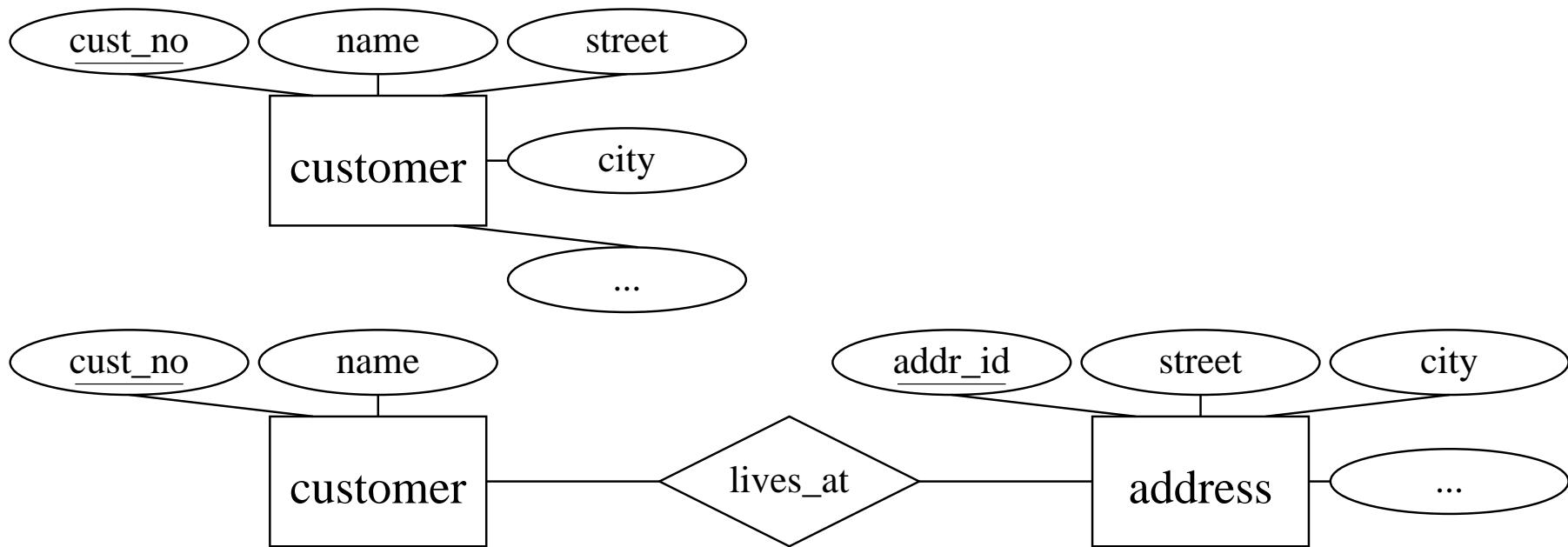


# Entities vs Attributes

- In general, modeling something as an
  - attribute is simpler
  - entity is more flexible
- Example: storing the address of a customer
  - First of all, the components of an address should be split up into street, house number, postcode, city, etc.

# Entities vs Attributes(2)

- However, this still leaves us with two options:
  - Attach parts of address as attributes directly to customer entity set
  - Create an entity set address and connect attributes to it



# Entities vs Attributes(3)

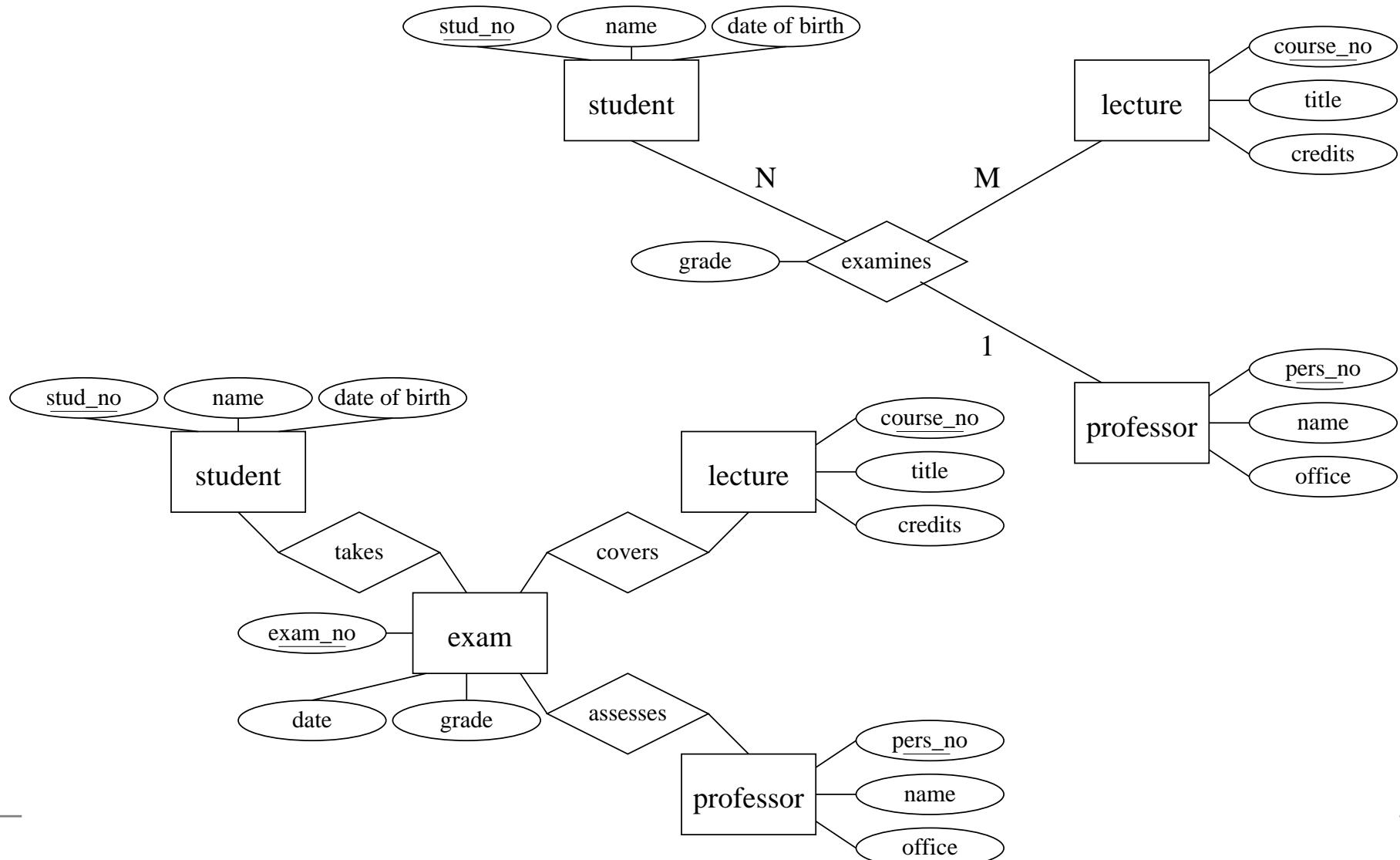
- If a customer has exactly one unique address, then attaching it as attributes is fine
- If a customer can have more than one address, then it has to be modeled using an entity
  - Attributes of an entity may only have one value (they cannot be set-valued)
  - Addresses as entities can also be re-used for other parts of the ER diagram, e.g. addresses of suppliers

# Entities vs Relationships

- Again, in general, modeling something as
  - a relationship is simpler
  - an entity is more flexible
- Example: receiving more than one grade in an examination for the same course

# Entities vs Relationships(2)

- The two options are:

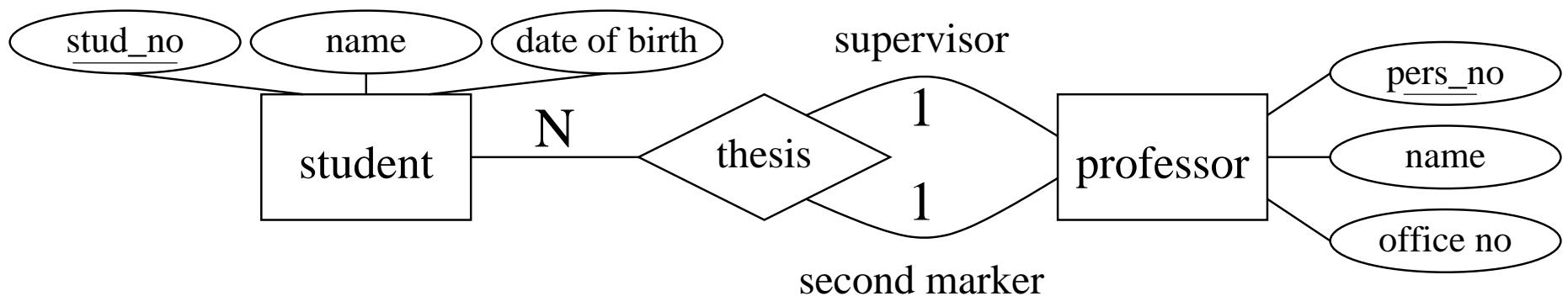


# Entities vs Relationships(3)

- Relationships do not have any identity:
  - Two entities cannot enter the same relationship more than once
- Modeling this as an entity:
  - We can have the same student/lecture/professor combination more than once
  - An exam can connect to other entity sets via relationships as well (not possible for a relationship)

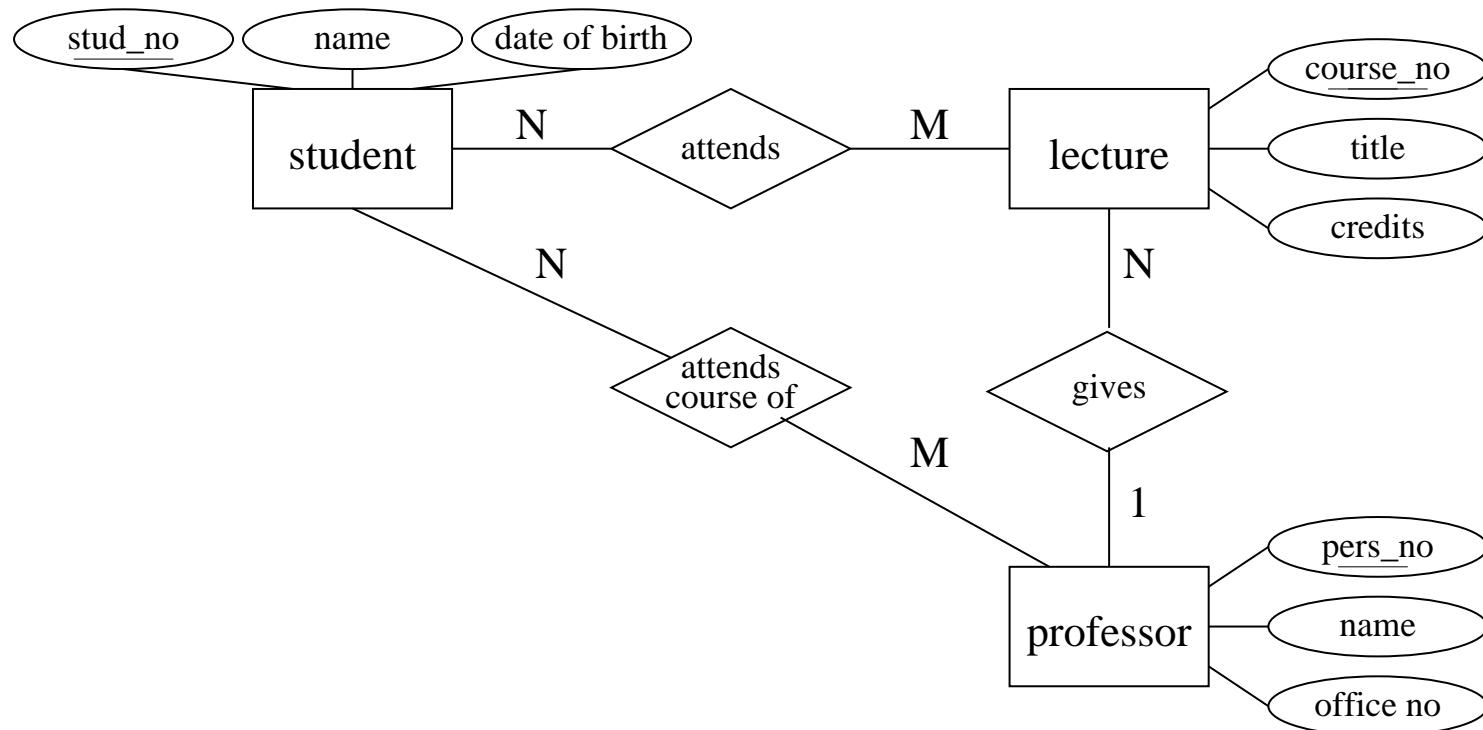
# Binary vs Ternary Relationships

- In real ternary relationships all participating entities are needed to describe the relationship
- The relationship `examines` is ternary
  - If student, lecture, or professor is missing, it's not a complete description
- The following relationship is not truly ternary:



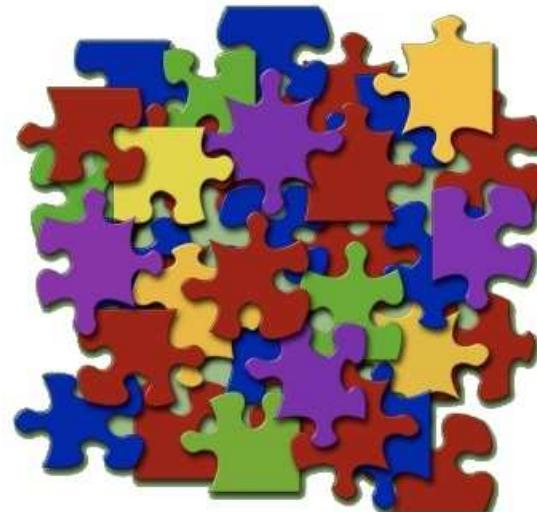
# Connection Trap

- A *connection trap* is an overspecification, i.e., adding redundant, cyclical relationships
- attends course of is redundant, this is already expressed (transitively) by attends and gives

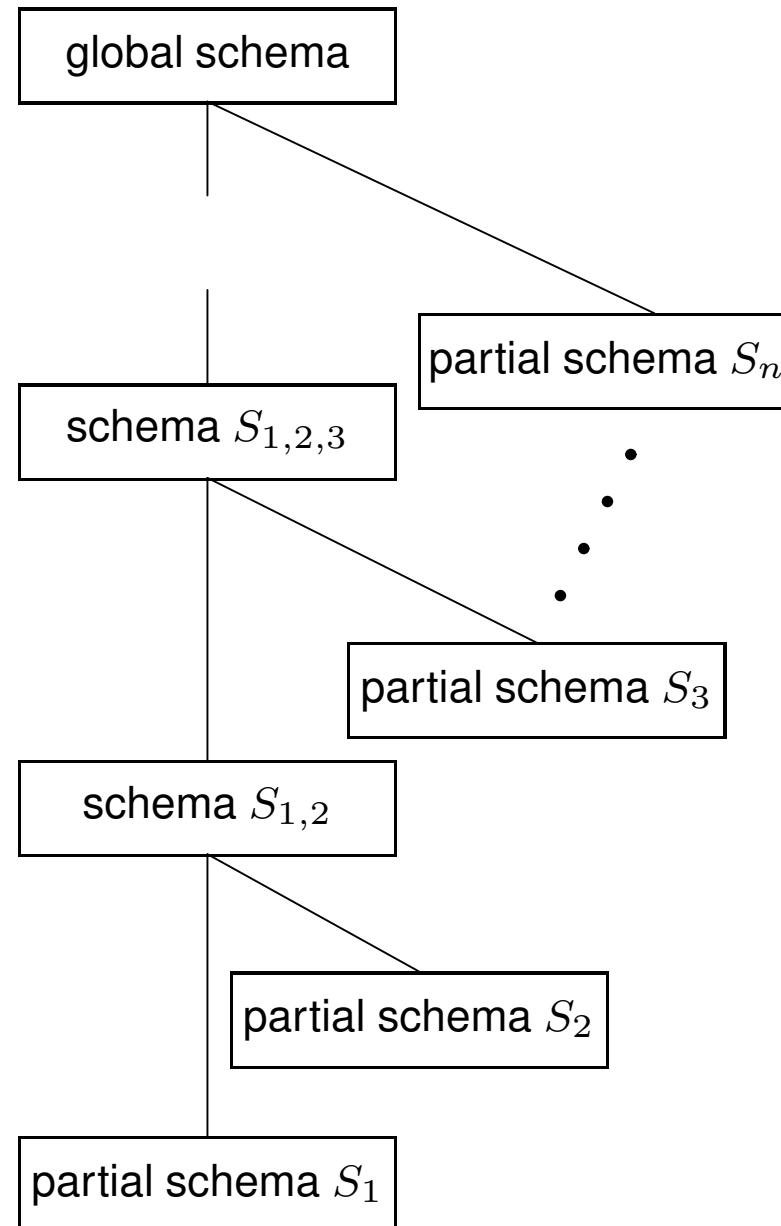


# Schema Consolidation

- Usually there are several stakeholders in medium-sized and large projects
- These different groups may have a different point of view when it comes to modeling the data
- Often, different groups develop partial schemas, which have to be integrated into a single, overall schema
- The process of integrating the schemas is called *schema consolidation*



# Schema Consolidation(2)

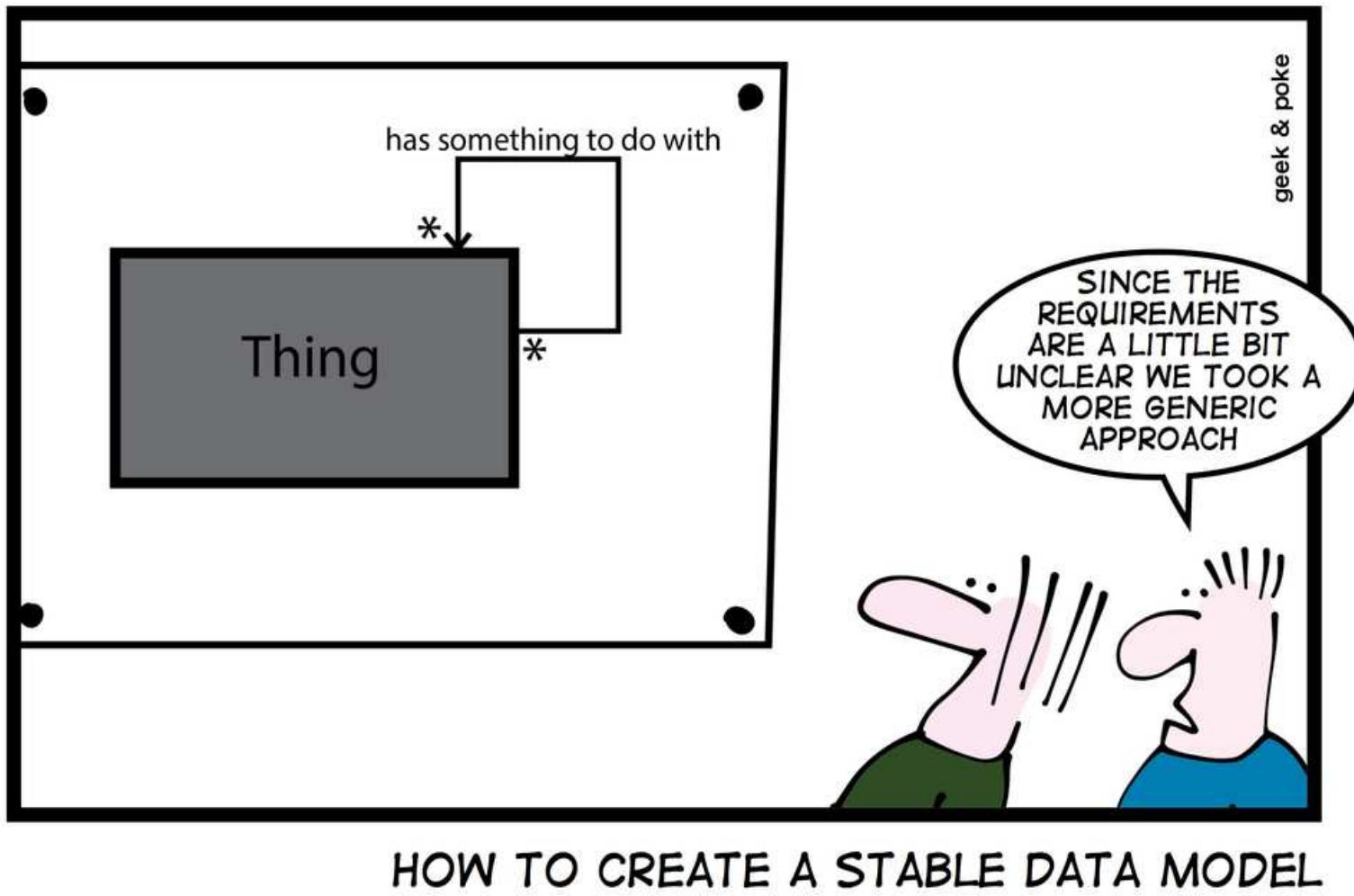


# Summary

- ER diagrams are a well-known tool for conceptual data modeling
  - They are similar to the structural part of UML
- While the individual constructs are not hard to understand, modeling a complex application can be hard
  - It may be worth modeling even simpler applications, to make sure nothing was overlooked

# Summary (2)

- And then there's always the generic approach:

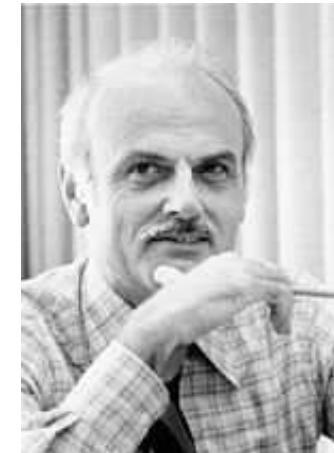


# Chapter 3

## The Relational Model

# Introduction

- The relational model was developed by E.F. Codd in the 1970s (he received the Turing award for it)



- One of the most widely-used data models
- Superseded the hierarchical and network models
- It was/is being extended with object-oriented and semistructured (XML) concepts

# Basic Definition

- A relational database contains a set of relations
- A *relation R* consists of two parts:
  - An *instance R*, which is a table with rows and columns
    - Represents the current content of this table
  - A *schema R* which specifies the name of the relation and the names and data types of the columns
    - Defines the structure of the table

# Example of an Instance

- Assume we want to keep information on students in the relation `student`:

attribute  
↓

stud_no	name	date_of_birth
1	Smith, John	1990-10-12
2	Miller, Anne	1992-07-30
3	Jones, Betty	1991-03-24
...	...	...

- We call the rows *tuples*
- We call the columns *attributes*

# Schema of the Relation Student

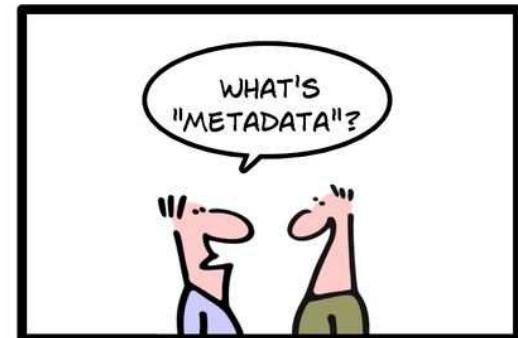
- student has three attributes: stud\_no, name, and date\_of\_birth
- We associate a domain, or data type, with each attribute
  - $D_{stud\_no}$  = integer
  - $D_{name}$  = string
  - $D_{date\_of\_birth}$  = date
- So the complete schema looks like this:

```
student(stud_no:integer,  
        name:string,  
        date_of_birth:date)
```

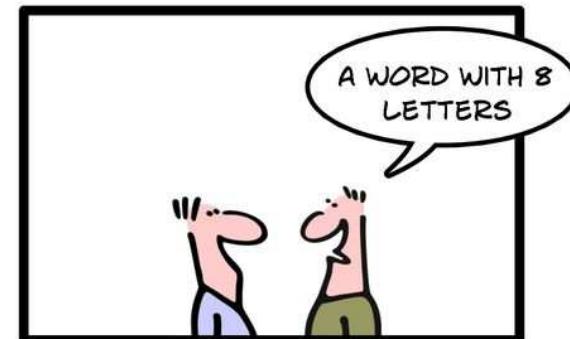
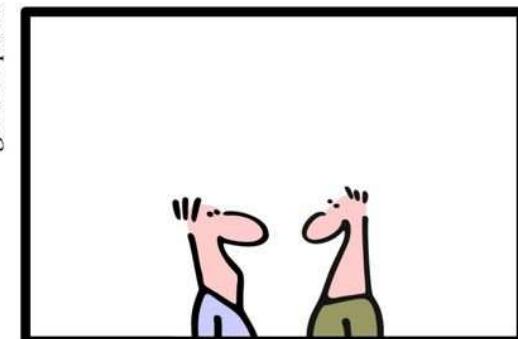
# Schema Information

- Schema information is also called *metadata*
  - It is not the actual data...
  - ...but a description of what the data looks like

SIMPLY EXPLAINED:  
METADATA

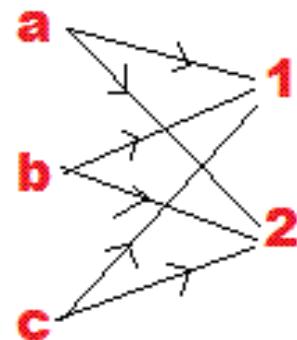


geek & poke



# More Definitions

- A relation (instance) is a subset of the Cartesian product of the domains



- $R \subseteq D_1 \times D_2 \times \dots \times D_n$
- For example,  
student  $\subseteq D_{stud\_no} \times D_{name} \times D_{date\_of\_birth} =$   
student  $\subseteq \text{integer} \times \text{string} \times \text{date}$

# Even More Definitions

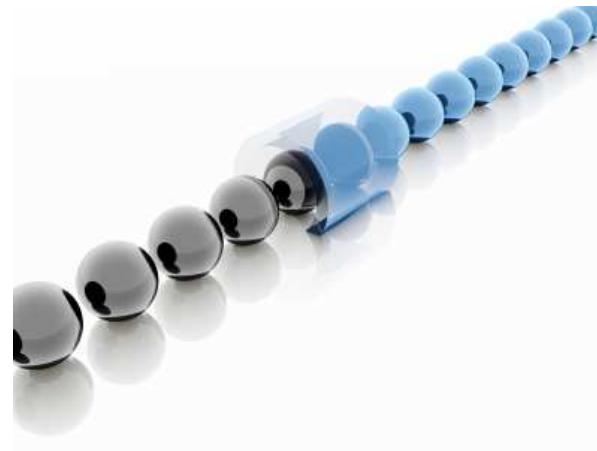
- The *cardinality* of a relation is its size in number of tuples
- A minimal set of attributes whose values uniquely identify a tuple is called a *key*
- Usually, the *primary key* is underlined

student(stud\_no, name, date\_of\_birth)



# Converting ER → Relational

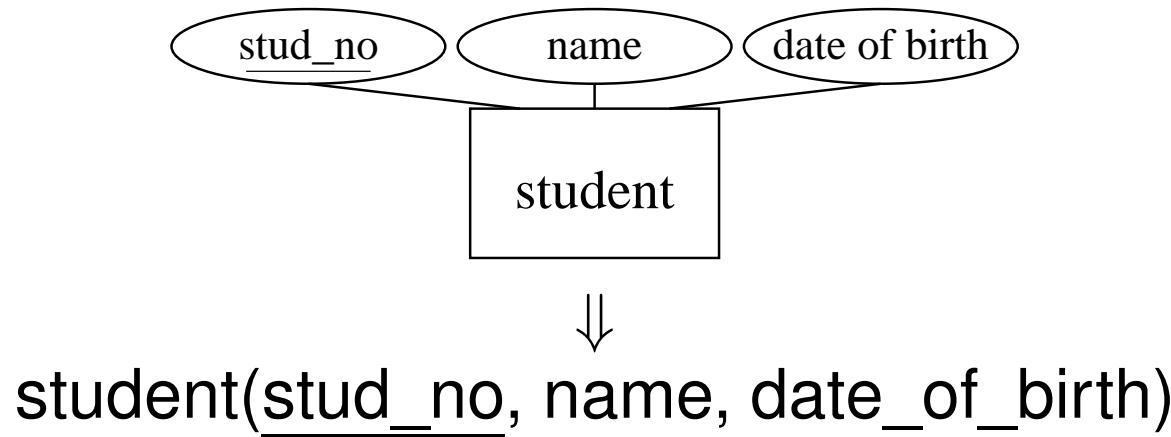
- If you want to implement your database using a relational system and you have an ER diagram...
- ...you can convert this diagram into a relational schema!



- This is done in three steps:
  - Converting entities
  - Converting relationships
  - Simplifying the schema

# Converting Entities

- Every entity is transformed into a separate relation
- Every attribute of the entity becomes an attribute in the relation
  - Choose the key attribute(s) of the entity as primary key of the relation



# Converting Relationships

- Every relationship is transformed into a separate relation
- Attributes of this relation are made up of the keys of the participating entity relations
  - + any attributes of the relationship (e.g. grade of the relationship examines)
- What is the key of this relation?

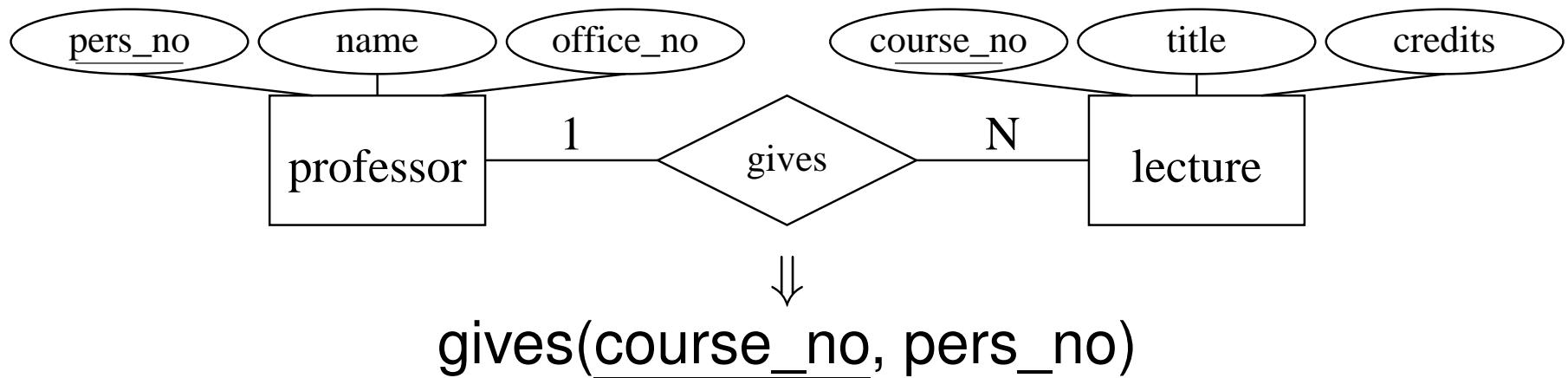
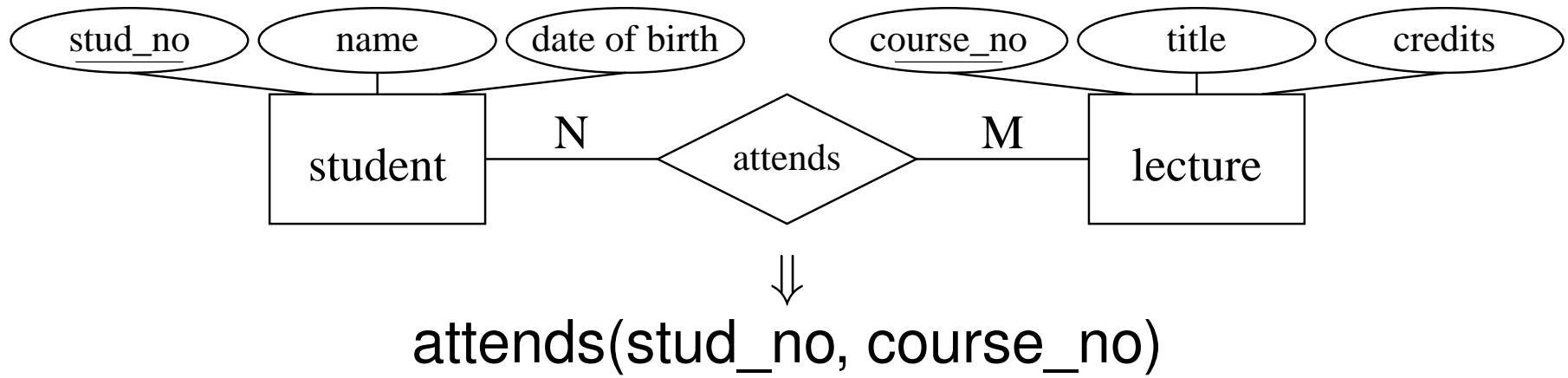


# Converting Relationships(2)

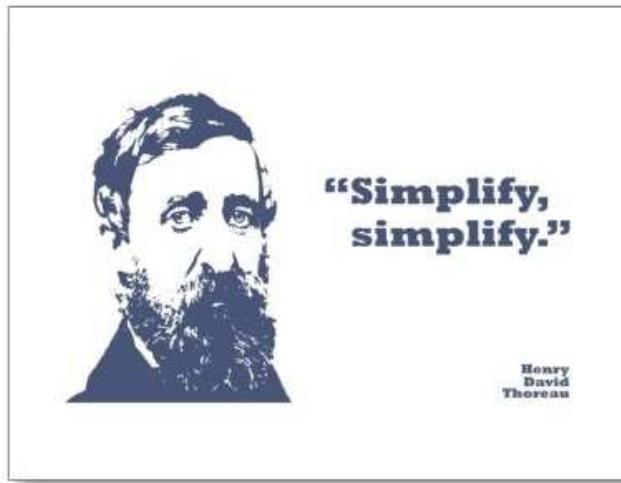
- The multiplicity of a relationship determines the key
  - N:M
    - the key is a combination of both entity keys
  - 1:N, N:1
    - key of the entity on the N-side becomes key
  - 1:1
    - choose one of the entity keys (arbitrarily)



# Examples



# Simplifying the Schema



- There may be relations with the same key after the second step
- Relations with the same key are merged into a single relation
- This happens in the case of 1:N, N:1, and 1:1 multiplicities
  - The relationship relation has the same key as one of the entity relations

# Example

- lecture and gives have the same key:

lecture(course\_no, title, credits)  
gives(course\_no, pers\_no)



lecture(course\_no, title, credits, pers\_no)

# Example Continued

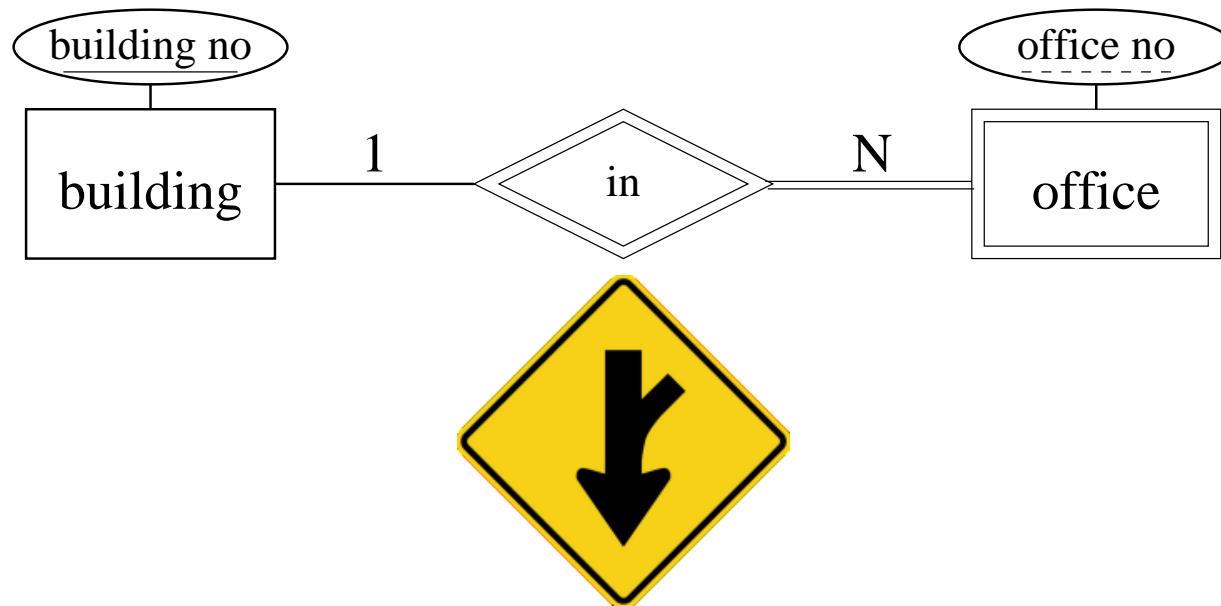
- pers\_no is called a *foreign key* in lecture
- It references the key in professor (and also contains the same values)
- Sometimes foreign keys are renamed to make it clear where they come from

lecture(course\_no, title, credits, prof\_pers\_no)



# Weak Entities

- Basically, weak entities are converted like 1:N relationships
- Major difference: the weak entity has a combined key

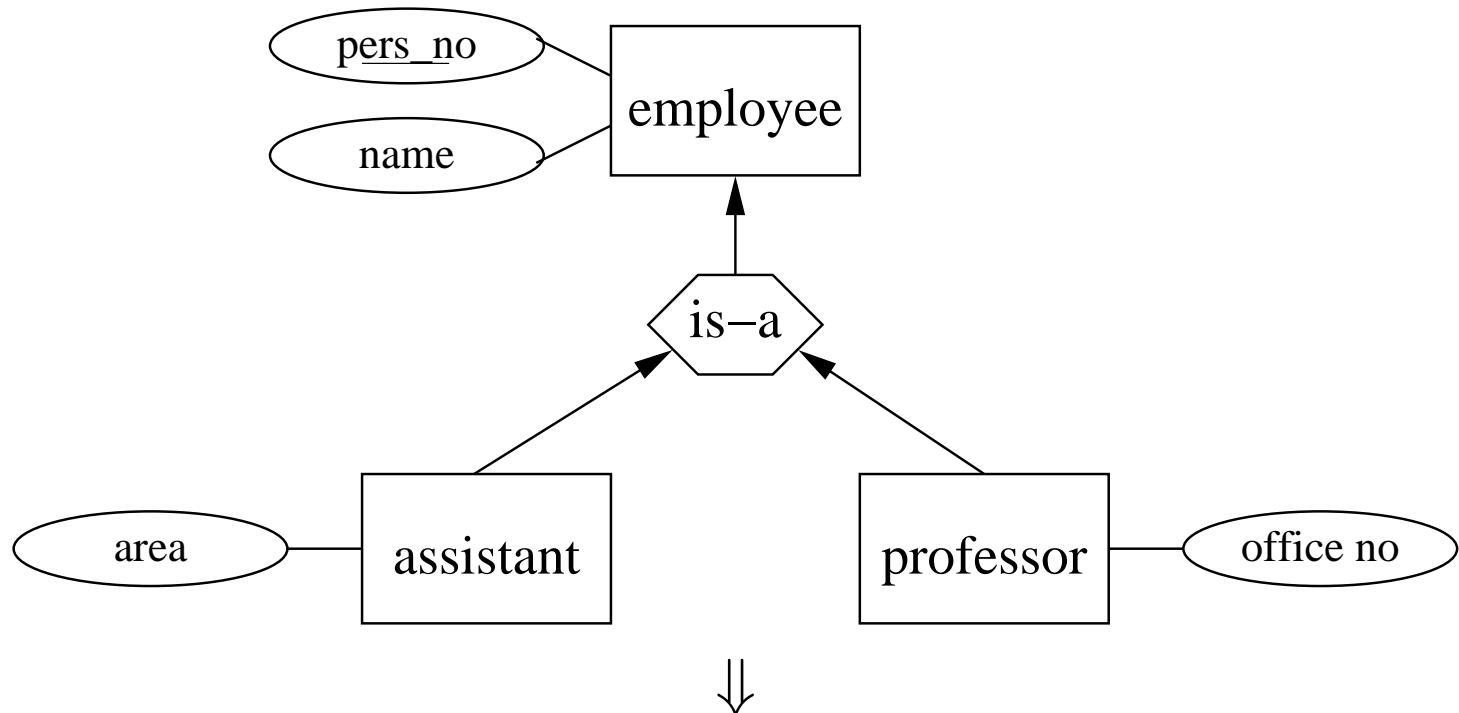


building(building\_no, ...)

office(building\_no, office\_no, ...)

# Generalizations

- Generalizations are converted in three different ways
  - There is no direct equivalent translation into the relational model



(1) only super-type, (2) only sub-types, (3) all types

# Generalizations(2)

- Only super-type:  
 $\text{employee}(\underline{\text{pers\_no}}, \text{name}, \text{area}, \text{office\_no})$   
Issue: NULL values
- Only sub-types:  
 $\text{assistant}(\underline{\text{pers\_no}}, \text{name}, \text{area})$   
 $\text{professor}(\underline{\text{pers\_no}}, \text{name}, \text{office\_no})$   
Issue: employees who are neither assistants nor professors
- All types:  
 $\text{employee}(\underline{\text{pers\_no}}, \text{name})$   
 $\text{assistant}(\underline{\text{pers\_no}}, \text{area})$   
 $\text{professor}(\underline{\text{pers\_no}}, \text{office\_no})$   
Issue: information is distributed among three relations

# Generalizations(3)

- There is no silver bullet
- You have to make a decision depending on the requirements of the application



# Query Languages

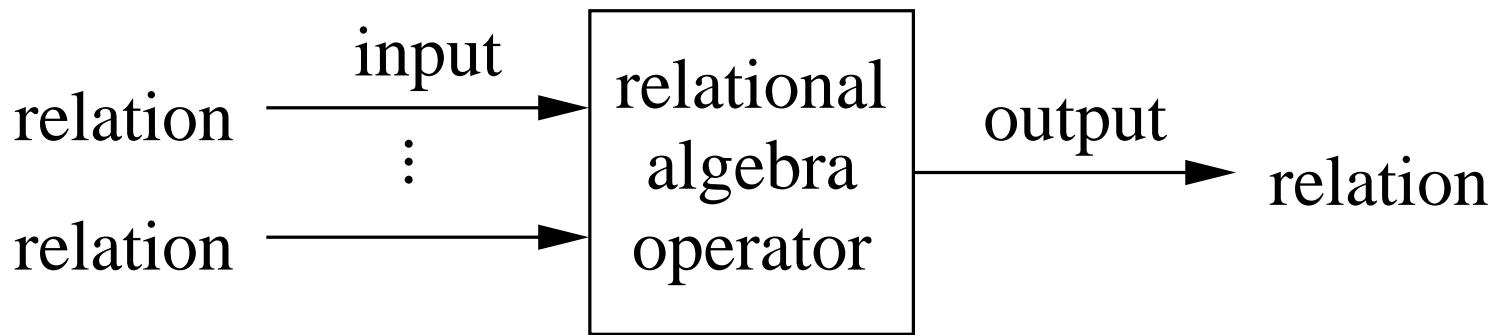
- So far we have only looked at how to describe the stored data
- We haven't covered how to retrieve the data from the database



- In order to access the data we need a *query language*
- The relational model comes with the *relational algebra* as query language

# Relational Algebra

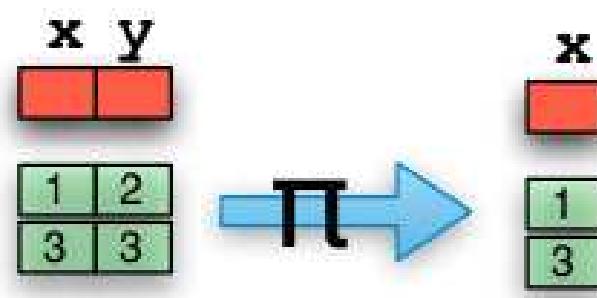
- All the operators of the relational algebra are set-oriented and closed:
  - Each operator gets as input one or more sets of tuples...
  - ... and outputs a set of tuples



- Let us have a closer look at the operators

# Projection

- Chooses a subset of attributes (columns)  $A_1, \dots, A_n$  from a relation  $R$
- ... and throws away the other attributes:



- Has the following syntax:  $\pi_{A_1, \dots, A_n}(R)$

# Example

student

stud_no	name	date_of_birth
1	Smith, John	1990-10-12
2	Miller, Anne	1992-07-30
3	Jones, Betty	1991-03-24
4	Johnson, Boris	1992-07-30

Query: “Output the names of all students”

Algebra:  $\pi_{name}(\text{student})$

name
Smith, John
Miller, Anne
Jones, Betty
Johnson, Boris

# Projection(2)

- What happens with duplicates?
- The (basic) relational model is set-oriented: a relation is a set of tuples
- That means, duplicates are removed



# Example

student

stud_no	name	date_of_birth
1	Smith, John	1990-10-12
2	Miller, Anne	1992-07-30
3	Jones, Betty	1991-03-24
4	Johnson, Boris	1992-07-30

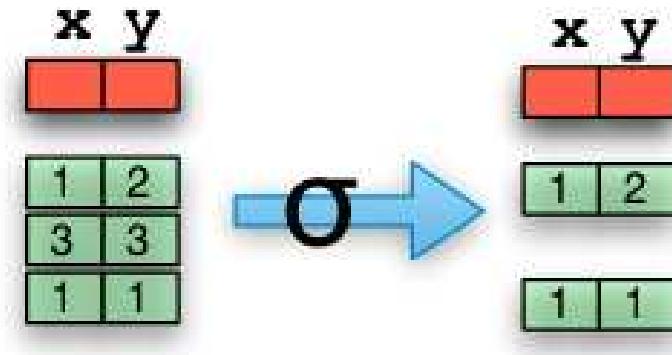
Query: “Output the birthdays of all students”

Algebra:  $\pi_{date\_of\_birth}(\text{student})$

date_of_birth
1990-10-12
1992-07-30
1991-03-24

# Selection

- Chooses a subset of tuples (rows)  $t_1, \dots, t_n$  from  $R$
- ... and throws away the other tuples:



- Has the following syntax:  $\sigma_p(R)$ 
  - A tuple has to satisfy the predicate  $p$  to stay in
  - The usual comparison operators are used in predicates:  $=, <, \leq, >, \geq$
  - Predicates can be combined via  $\wedge, \vee, \neg$

# Example

student

stud_no	name	date_of_birth
1	Smith, John	1990-10-12
2	Miller, Anne	1992-07-30
3	Jones, Betty	1991-03-24
4	Johnson, Boris	1992-07-30

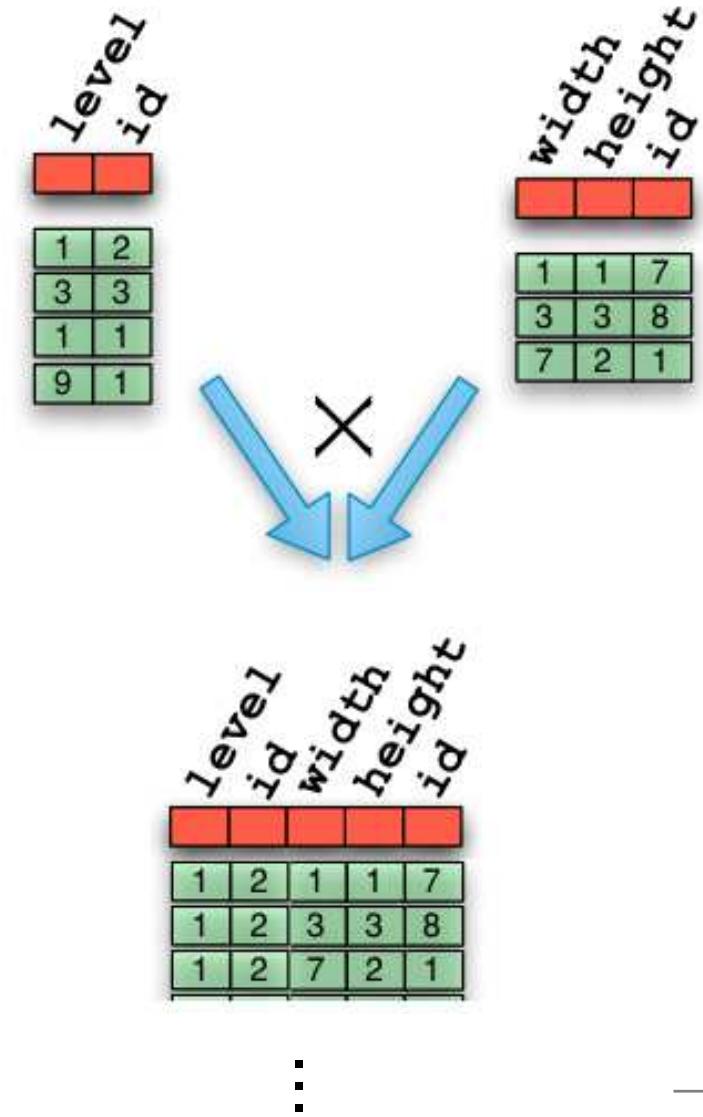
Query: “Get all students with a student number less or equal than 3 and a date of birth after 1.1.1991”

Algebra:  $\sigma_{stud\_no \leq 3 \wedge date\_of\_birth > 1991-01-01}(\text{student})$

stud_no	name	date_of_birth
2	Miller, Anne	1992-07-30
3	Jones, Betty	1991-03-24

# Cartesian Product

- Up to now, all operators had one relation as input parameter
- What do we do if we need information from more than one table?
- The Cartesian product (or cross product) combines every tuple of one relation with every tuple of another
- Syntax:  $R_1 \times R_2$



# Example

lecture

course_no	...	prof_pers_no
1	...	1
2	...	1
3	...	2
4	...	2

professor

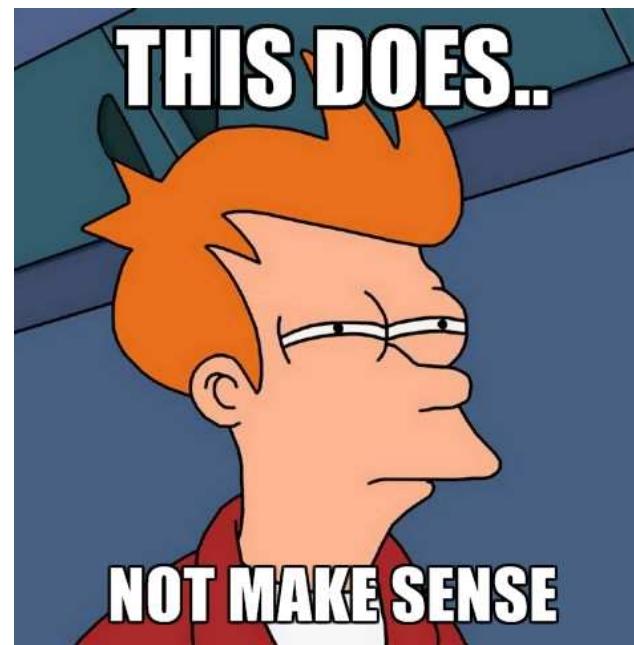
pers_no	name	...
1	Gamper	...
2	Helmer	...

Query: “Output all lectures and professors”

Algebra:  $\text{lecture} \times \text{professor}$

# Result

course_no	...	prof_pers_no	pers_no	name	...
1	...	1	1	Gamper	...
1	...	1	2	Helmer	...
2	...	1	1	Gamper	...
2	...	1	2	Helmer	...
3	...	2	1	Gamper	...
3	...	2	2	Helmer	...
4	...	2	1	Gamper	...
4	...	2	2	Helmer	...



# Filtering Out Useless Combinations

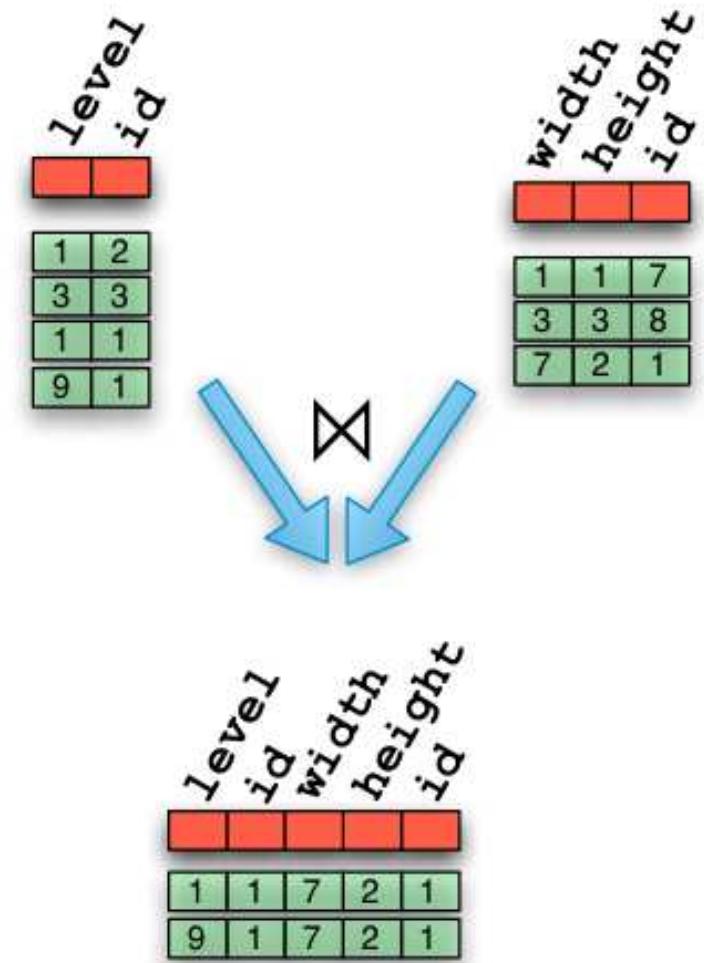
- Many of the tuples on the previous slide merge lectures with professors who have nothing to do with this lecture
- Useless combinations can be filtered out with a selection



- For our example, we could rewrite the query to  $\sigma_{prof\_pers\_no=pers\_no}(\text{lecture} \times \text{professor})$

# Join

- As cross product/filtering is needed very often, there is a shortcut:  
a *join operator* ( $\bowtie$ )
- $R_1 \bowtie_{R_1.A_i=R_2.A_j} R_2 = \sigma_{R_1.A_i=R_2.A_j}(R_1 \times R_2)$
- Joins can also be implemented more efficiently than Cartesian products



# Example

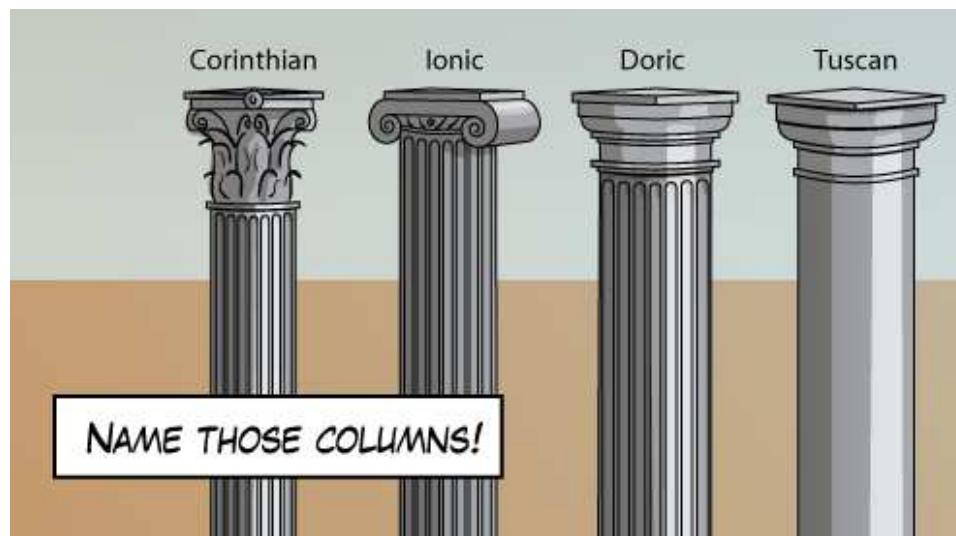
Query: “Output all lectures together with the responsible professors”

Algebra:  $\text{lecture} \bowtie_{\text{prof\_pers\_no} = \text{pers\_no}} \text{professor}$

course_no	...	prof_pers_no	pers_no	name	...
1	...	1	1	Gamper	...
2	...	1	1	Gamper	...
3	...	2	2	Helmer	...
4	...	2	2	Helmer	...

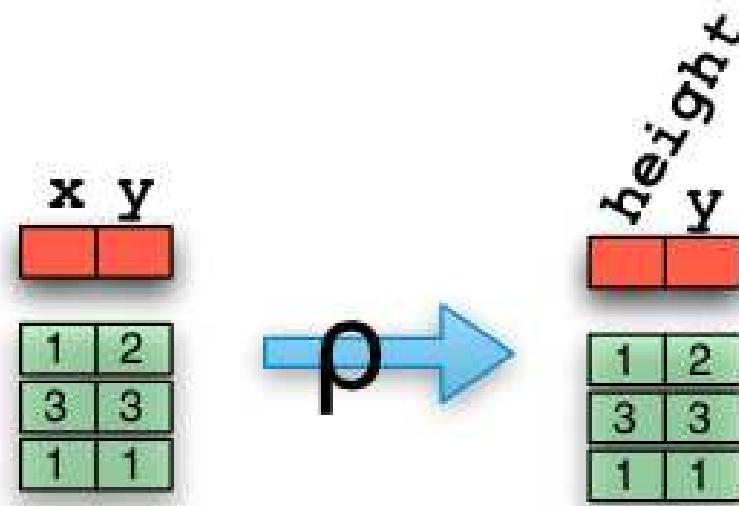
# Combining Relations

- The names of attributes in a relation need to be unique
- Important when combining different relations
  - Example:  
assistant(pers\_no, name, area, mgr)  
professor(pers\_no, name, office\_no)  
**assistant  $\bowtie_{mgr=pers\_no}$  professor**
  - We have two different personnel numbers: one for assistants, another for professors



# Renaming

- In order to keep names unique, we can do renaming:
  - Chooses a subset of attributes and gives them new names



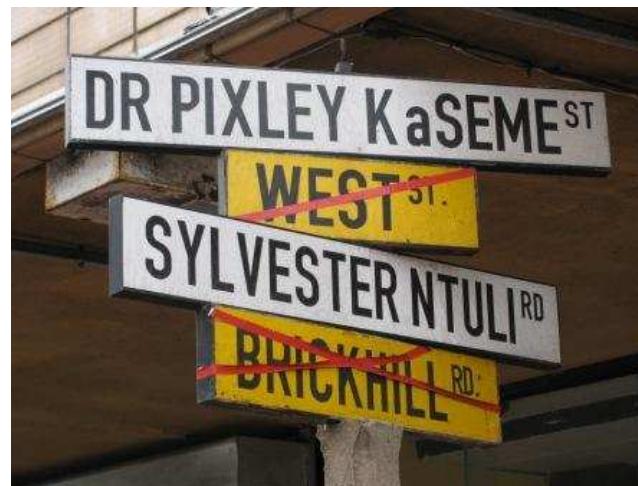
- Has the following syntax:  $\rho_{B_1 \leftarrow A_1, \dots, B_n \leftarrow A_n}(R)$

# Example

- So to avoid clashes between attribute names, we could rename attributes before joining:
- In relational algebra:

$$\rho_{a\_no \leftarrow pers\_no, a\_name \leftarrow name}(\text{assistant})$$
$$\bowtie_{mgr = p\_pers\_no}$$
$$\rho_{p\_pers\_no \leftarrow pers\_no, p\_name \leftarrow name}(\text{professor})$$

- This helps us in distinguishing which attributes come from which relation



# Joins Revisited

- There are different variants of joins, classified after the join predicate
  - Theta-join ( $\theta$ -join): the most general join, the predicate may include arbitrary comparison operators
    - That is the one we have looked at so far
  - Equi-join: the join predicate only checks for equality (subset of  $\theta$ -joins)
  - Natural join: a special version of an equi-join
    - We will now look at this operator in more detail

# Natural Join

- A natural join
  - does an equi-join on attributes with the same names
  - and projects away the redundant columns
  - does not have a join predicate (it's implicit)

**Query:** “Output all lectures together with the responsible professors”

**Algebra:**  $(\rho_{pers\_no \leftarrow prof\_pers\_no}(\text{lecture})) \bowtie \text{professor}$

course_no	...	pers_no	name	...
1	...	1	Gamper	...
2	...	1	Gamper	...
3	...	2	Helmer	...
4	...	2	Helmer	...

# Joins Revisited(2)

- You can join an arbitrary number of relations in a single relational algebra expression
- The order does not matter, you'll always get the same result
  - Join operators are commutative and associative



# Example

- Combine students with the lectures they attend

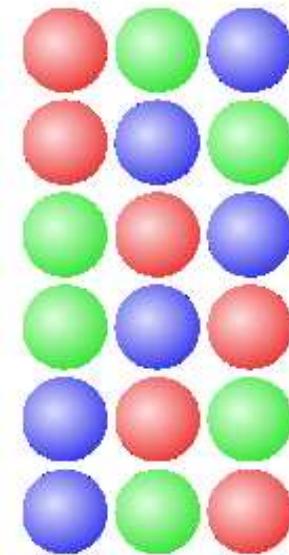
student		
stud_no	name	...
1	Smith	...
2	Miller	...
...	...	...

lecture		
course_no	title	...
1	databases	...
2	networks	...
...	...	...

attends	
stud_no	course_no
1	1
1	2
2	1
...	...

# Example(2)

- So we could join the three relations in any of the following ways:
  - student  $\bowtie$  attends  $\bowtie$  lecture
  - student  $\bowtie$  lecture  $\bowtie$  attends
  - attends  $\bowtie$  student  $\bowtie$  lecture
  - attends  $\bowtie$  lecture  $\bowtie$  student
  - lecture  $\bowtie$  student  $\bowtie$  attends
  - lecture  $\bowtie$  attends  $\bowtie$  student
- You'll notice that the 2nd and 5th permutation joins two relations that do no have a common attribute
  - In that case, the natural join computes a Cartesian product



# Even More Joins

- If a tuple does not find a “join partner” in the other relation, it is dropped



- If you want to keep these tuples, you need to use an *outer join*

# Full Outer Join

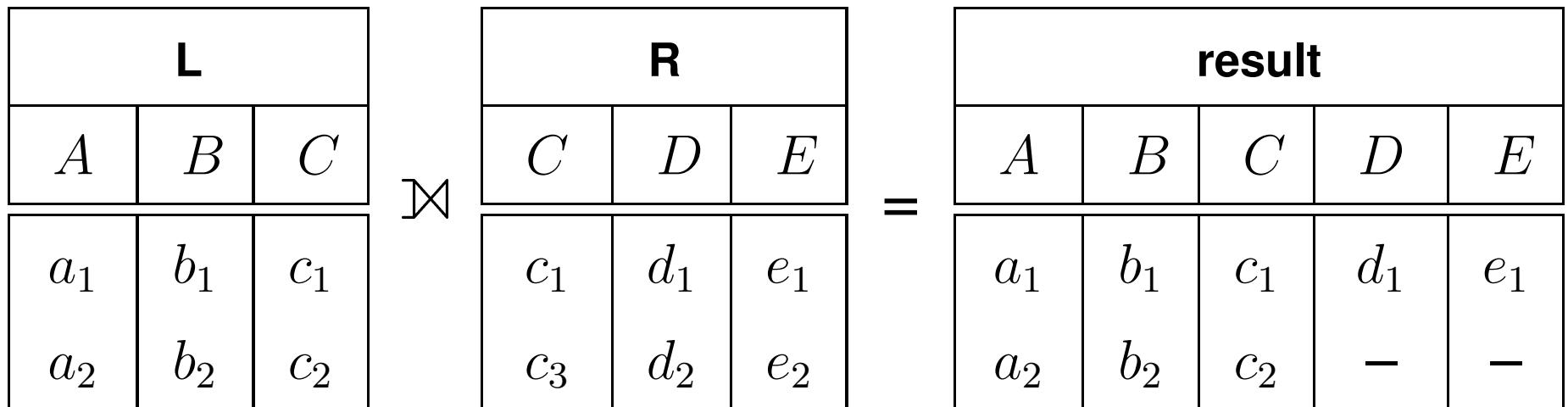
- In a *full outer join* ( $\bowtie$ ) all tuples from both relations are kept

L			R			result		
A	B	C	C	D	E	A	B	C
$a_1$	$b_1$	$c_1$	$c_1$	$d_1$	$e_1$	$a_1$	$b_1$	$c_1$
$a_2$	$b_2$	$c_2$	$c_3$	$d_2$	$e_2$	$a_2$	$b_2$	$c_2$
—	—	—	—	—	—	—	—	$e_2$

- A natural join would only return the first tuple

# Left Outer Join

- In a *left outer join* this is only true for tuples from the relation on the left-hand side



# Right Outer Join

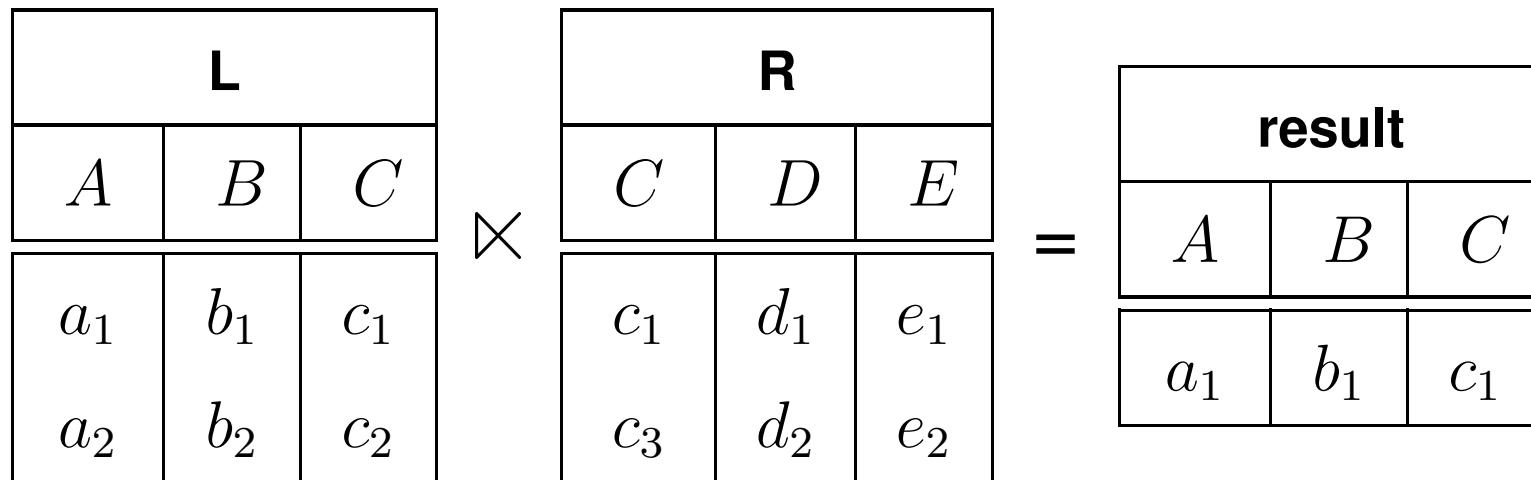
- In a *right outer join* this is only true for tuples from the relation on the right-hand side

L			R			result				
A	B	C	C	D	E	A	B	C	D	E
$a_1$	$b_1$	$c_1$	$c_1$	$d_1$	$e_1$	$a_1$	$b_1$	$c_1$	$d_1$	$e_1$
$a_2$	$b_2$	$c_2$	$c_3$	$d_2$	$e_2$	—	—	$c_3$	$d_2$	$e_2$



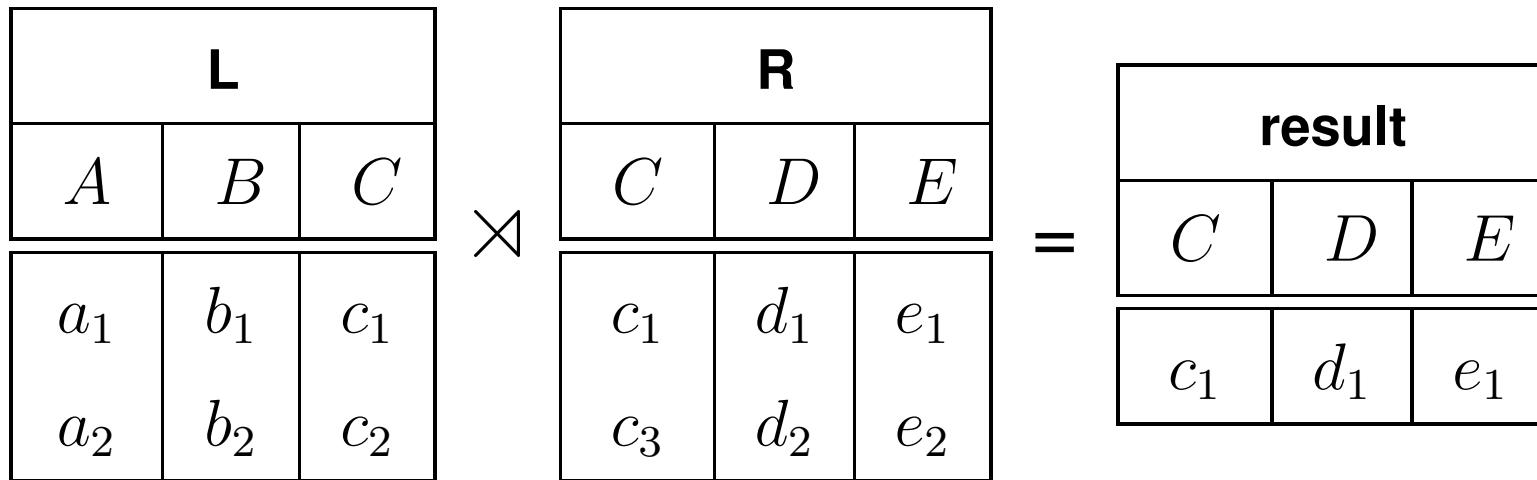
# Semi-Join

- A *semi-join* does not really join the relations
- It checks which of the tuples in the relation on the left-hand side satisfies the join predicate
- ... and then keeps those



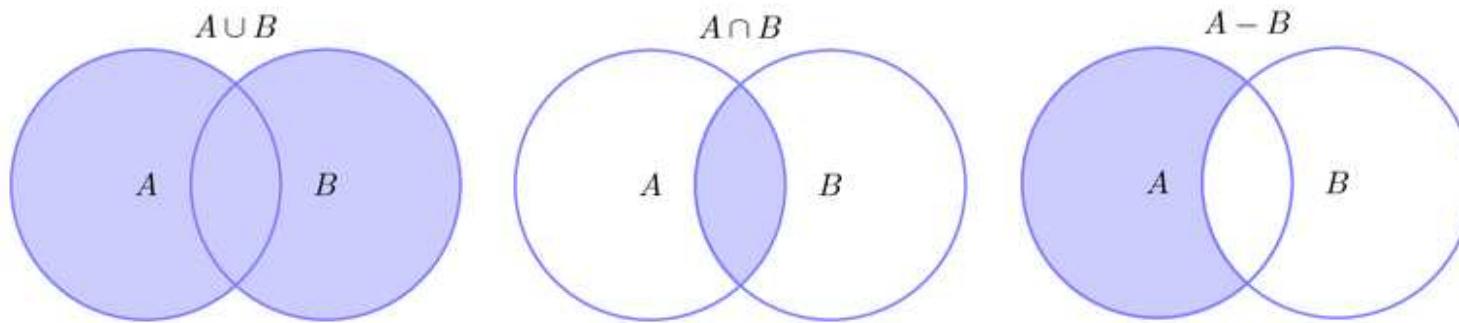
# Semi-Join(2)

- There is also the right-handed version:



# Set Operations

- Set operations such as union, intersection, and difference can also be applied to relations
- ... provided that both relations have the same schema, meaning:
  - same number of attributes (with the same names)
  - corresponding attributes have the same data type



# Union

prof1	
pers_no	name
1	Gamper
2	Helmer

prof2	
pers_no	name
2	Helmer
3	Wood

Query: “Create the union of the relation prof1 and prof2”

Algebra:  $\text{prof1} \cup \text{prof2}$

pers_no	name
1	Gamper
2	Helmer
3	Wood

# Intersection

prof1		prof2	
pers_no	name	pers_no	name
1	Gamper	2	Helmer
2	Helmer	3	Wood

Query: “Which professors are in both relations?”

Algebra:  $\text{prof1} \cap \text{prof2}$

pers_no	name
2	Helmer

# Set Difference

prof1	
pers_no	name
1	Gamper
2	Helmer

prof2	
pers_no	name
2	Helmer
3	Wood

Query: “Which professors are in the first but not the second relation?”

Algebra:  $\text{prof1} \setminus \text{prof2}$

pers_no	name
1	Gamper

# Relational Division

- Relational division is the most complicated operator found in relational algebra
- Let's start nice and simple:
  - In arithmetic, division is the opposite of multiplication
  - The Cartesian product can be seen as “multiplying” two relations
  - So relational division can be interpreted as the opposite of the Cartesian product



# Relational Division(2)

- Let's look at a concrete example:

$$\begin{array}{c} R \\ \hline \hline A \\ \hline \hline a \\ b \\ c \end{array}$$
$$\begin{array}{c} S \\ \hline \hline B \\ \hline \hline 1 \\ 2 \end{array}$$

$R \times S$	
A	B
a	1
a	2
b	1
b	2
c	1
c	2

- Dividing  $R \times S$  by  $R$  or  $S$  gives us the following results:
  - $(R \times S) \div R = S$
  - $(R \times S) \div S = R$

# Relational Division(3)

- What happens if we apply relational division  $T \div R$  to an arbitrary relation  $T$ ?
  - That means,  $T$  is not the immediate result of a Cartesian product  $R \times X$
  - Can't be that arbitrary:  $\mathcal{R} \subset \mathcal{T}$  has to hold
- Roughly similar to dividing 8 by 3 (integer division)
- 8 is not divisible by 3, but we can say that  $8 \div 3 = 2$  remainder 2
- When computing  $T \div R$ , we try to find a relation  $R'$  such that
  - $R \times R'$  covers as much as possible of  $T$

# Relational Division(4)

- Let's look at a concrete example again:

$R$	$S$
$\overline{A}$	$\overline{B}$
a	1
b	2
c	

$T$	
$A$	$B$
a	2
b	1
b	2
c	1
c	2

$(S' \times S) \rightarrow$   $\leftarrow (R \times R')$

$(S' \times S) \rightarrow$   $\leftarrow (R \times R')$

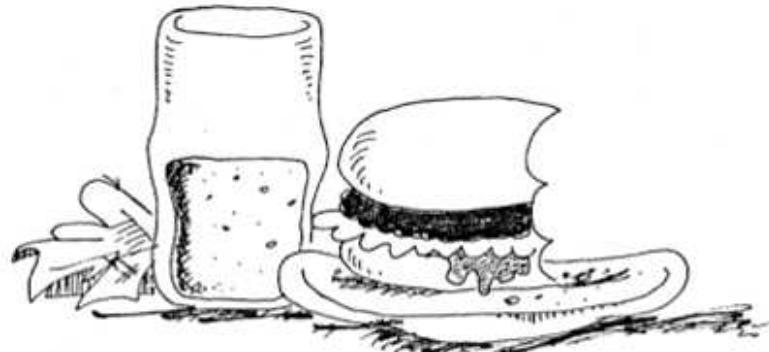
$(S' \times S) \rightarrow$   $\leftarrow (R \times R')$

- $T \div S = S'$
- $T \div R = R'$

$S'$	$R'$
$\overline{A}$	$\overline{B}$
b	
c	2

# Relational Division(5)

- $S'$  does not include  $a$ , as this would produce  $[a,1]$ , which is not in  $T$ 
  - $S' \times S$  creates four tuples and we have the “remainder”  $[a,2]$
- $R'$  does not include  $1$ , as this would again produce  $[a,1]$ , which is not in  $T$ 
  - $R \times R'$  creates three tuples and we have the “remainder”  $[b,1], [c,1]$
- We have some “leftover” tuples not covered by the Cartesian product



# What Do We Need It For?

- Relational division is used to answer queries involving universal quantification:

attends	
stud_no	course_no
1	1
1	2
2	1
2	3
3	1
3	2
3	3

lect1	lect2	lect3
course_no	course_no	course_no
1	1	1
2	2	3

Query: “List the `stud_no` of *all* students who attend *all* lectures in `lect1/lect2/lect3`”

# Result

attends ÷ lect1

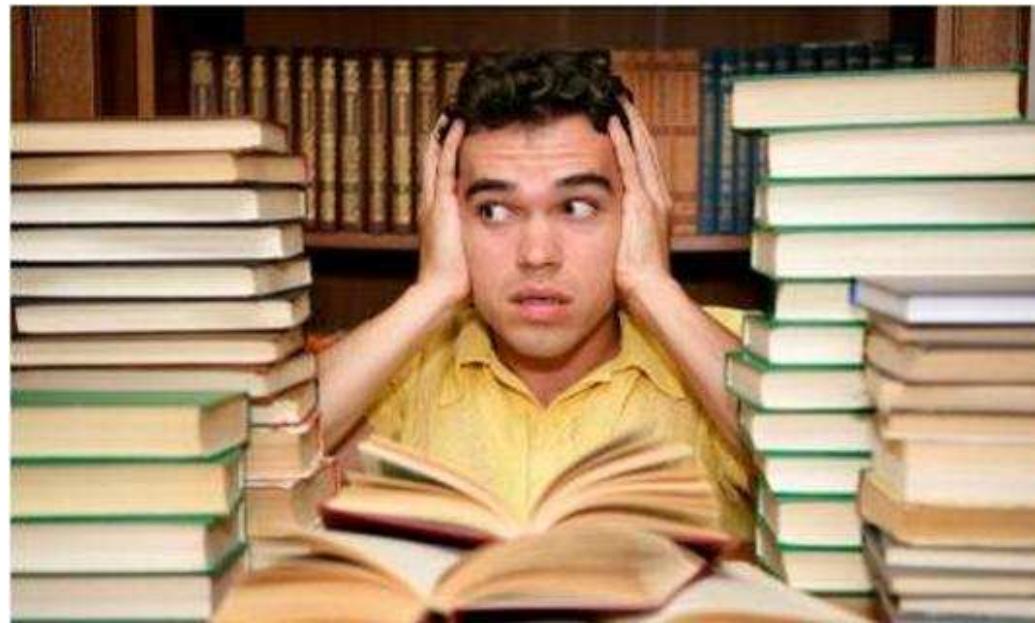
attends ÷ lect2

attends ÷ lect3

<u>stud_no</u>
1
3

<u>stud_no</u>
2
3

<u>stud_no</u>
3



# Formal Definition

- There is also a formal definition using other operators:

$$T \div R = \pi_{(\mathcal{T} \setminus \mathcal{R})}(T) \setminus \pi_{(\mathcal{T} \setminus \mathcal{R})}((\underbrace{\pi_{(\mathcal{T} \setminus \mathcal{R})}(T) \times R}_{2}) \setminus \underbrace{T}_{3})$$

1

2

3

4

1. Take the values in  $T$  of attributes not shared with  $R$
2. Combine them with  $R$  in a Cartesian product
3. Subtract all the stuff that is in  $T$   
(Now we are left with what's missing in  $T$  from the “full” Cartesian product)
4. Subtract this from 1. and we are now left with the values participating in a product

# Relational Calculus

- In relational algebra a user has to specify *how* to compute the result
  - This is called *procedural* or *imperative*
- In relational calculus a user specifies *what* data they want
  - This is called *declarative*



# Relational Calculus (2)

- An expression in relational calculus creates a new relation
- This relation is specified with variables ranging over
  - tuples of relations (tuple relational calculus (TRC))
  - the domains of attributes of relations (domain relational calculus (DRC))
- TRC, DRC, and relational algebra all have the same expressive power
- We focus on DRC

# Domain Relational Calculus

- Relational calculus (TRC and DRC) are based on *first-order logic* (also called *predicate logic*)
- An expression in DRC is of the form:

$$\{\langle x_1, x_2, \dots, x_n \rangle | P(x_1, x_2, \dots, x_n)\}$$

- where  $x_1, x_2, \dots, x_n$  represent domain variables
- and  $P$  represents a formula composed of atoms ( $P$  is also called a *predicate*)

# Atoms

- An atom in DRC has one of the following forms
  - $\langle x_1, x_2, \dots, x_n \rangle \in r$ , where  $r$  is a relation with  $n$  attributes
    - $x_1, x_2, \dots, x_n$  are variables taking on values from the domains of the attributes of  $r$
  - $x\theta y$ , where  $x$  and  $y$  are domain variables and  $\theta$  is a comparison operator ( $<$ ,  $\leq$ ,  $=$ ,  $\neq$ ,  $>$ ,  $\geq$ )
    - $x$  and  $y$  have to have domains that can be compared by  $\theta$
  - $x\theta c$ , where  $x$  is a domain variable,  $\theta$  is a comparison operator, and  $c$  is a constant
    - $c$  is a value from the domain of  $x$

# Formulae

- We build formulae from atoms using the following rules:
  - An atom is a formula
  - If  $P_1$  is a formula, then so are  $\neg P_1$  and  $(P_1)$
  - If  $P_1$  and  $P_2$  are formulae, then so are  $P_1 \vee P_2$ ,  $P_1 \wedge P_2$ , and  $P_1 \Rightarrow P_2$
  - If  $P_1(x)$  is a formula in  $x$  ( $x$  is a domain variable), then
    - $\exists x(P_1(x))$  and
    - $\forall x(P_1(x))$are also formulae
  - A variable is called a *free variable* unless it is quantified by  $\exists$  or  $\forall$
  - Quantified variables are called *bound variables*

# Example Queries

- Let's start with something simple (basically the equivalent of a selection with a projection)
- This refers to the relation  
`student(stud_no, name, date_of_birth)`

Query: “Get the names and dates of birth for all students with a date of birth after 1.1.1991”

DRC:  $\{\langle n, d \rangle | \exists s (\langle s, n, d \rangle \in \text{student} \wedge d > 1991-01-01)\}$

- Only variables on the left of | are allowed to be free

# Example Queries (2)

- Joins are done implicitly by using the same variable
- Here we use the relations:  
`lecture(course_no, title, credits, pers_no)`  
`professor(pers_no, name, office_no)`

Query: “Output the titles of all lectures together  
with the names of the responsible professors”

DRC:  $\{\langle t, n \rangle | \exists c, e, p (\langle c, t, e, p \rangle \in \text{lecture} \wedge \exists o (\langle p, n, o \rangle \in \text{professor}))\}$

# Example Queries (3)

- Now let's do something fancier
- We use the following relations:  
`student(stud_no, name, date_of_birth)`  
`professor(pers_no, name, office_no)`  
`exam(stud_no, course_no, pers_no, grade)`

Query: “Find the `stud_no` and `name` of all students who have taken at least one exam with professor Curie”

DRC:  $\{ \langle s, n \rangle |$   
 $\exists d(\langle s, n, d \rangle \in \text{student} \wedge$   
 $\exists c, p, g(\langle s, c, p, g \rangle \in \text{exam} \wedge$   
 $\exists a, o(\langle p, a, o \rangle \in \text{professor} \wedge a = \text{"Curie"})))\}$

# Safety of Expressions

- We can write expressions generating an infinite relation
- For example,

$$\{\langle s, n, d \rangle | \neg(\langle s, n, d \rangle \in \text{student})\}$$

is unsafe, as it generates all tuples not contained in the relation student (which is an infinite number)



# Safety of Expressions (2)

- We also have to be careful with quantifiers
- For example, given the expression

$$\{\langle x \rangle | \exists y (\langle x, y \rangle \in r) \wedge \exists z (\neg(\langle x, z \rangle \in r)) \wedge P(x, z)\}$$

- we can easily test the first part  $\exists y (\langle x, y \rangle \in r)$  by checking the values in  $r$
- However, for the second part we have to consider all values  $z$  not appearing in  $r$ 
  - Since this is an infinite number, we potentially have to evaluate  $P$  an infinite number of times

# Safety of Expressions (3)

- What does it mean for a DRC expression to be safe?
- For that, we define the *domain of a formula*,  $\text{dom}(P)$
- Basically,  $\text{dom}(P)$  is the set of all values referenced by  $P$
- This includes
  - all constants that appear in  $P$
  - all values that appear in a tuple of a relation in  $P$
- For example, for

$$\{\langle t \rangle \mid \exists c, e (\langle c, t, e \rangle \in \text{lecture} \wedge e > 2)\}$$

its domain includes the value 2 and all values appearing in tuples in the relation lecture

# Safety of Expressions (4)

- A DRC expression

$$\{\langle x_1, x_2, \dots, x_n \rangle | P(x_1, x_2, \dots, x_n)\}$$

is safe if all of the following hold

- All values that appear in tuples of the expression are values from  $\text{dom}(P)$
- For every subformula  $\exists x(P_1(x))$ , the subformula is true, iff there is an  $x \in \text{dom}(P_1)$  such that  $P_1(x)$  is true
- For every subformula  $\forall x(P_1(x))$ , the subformula is true, iff  $P_1(x)$  is true for all values  $x \in \text{dom}(P_1)$

# Summary

- The relational model and its variants are the most common data models used in DBMSs
- In this chapter we covered the (more theoretical) basics of the model:
  - Definition of the model
  - Transforming an ER diagram into a relational schema
  - Two basic query languages:
    - Relational algebra
    - Domain relational calculus

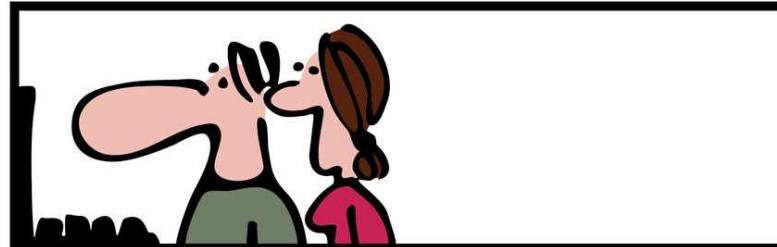
# Chapter 4

SQL

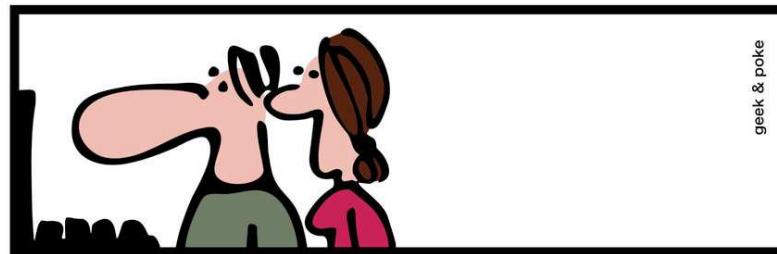
# General Remarks

- SQL stands for *Structured Query Language*
- Formerly known as SEQUEL: *Structured English Query Language*
- Standardized query language for relational DBMSs: SQL-86, SQL-89, SQL-92, SQL-1999, SQL-2003, SQL-2008, SQL-2011, SQL-2016

SIMPLY EXPLAINED - VINTAGE EDITION



geek & poke



SQL

# Major Parts of SQL

- At its heart, SQL is a *declarative* query language
- It's divided into four major parts:
  - DRL: Data Retrieval Language
  - DML: Data Manipulation Language
  - DDL: Data Definition Language
  - DCL: Data Control Language

# DRL

- The DRL contains commands to retrieve data from the database



- So its goal is very similar to that of the relational algebra: get data out of the database...

# DRL(2)

- A simple SQL-query consists of the three clauses **select**, **from** and **where**
- This is also called a select-from-where-block or sfw-block

```
select  list of attributes
from   list of relations
where   predicate;
```

# A Simple Example

student

stud_no	name	date_of_birth
1	Smith, John	1990-10-12
2	Miller, Anne	1992-07-30
3	Jones, Betty	1991-03-24
4	Johnson, Boris	1992-07-30

Query: “Give me all available data about all students”

```
select      *  
from      student;
```

- This returns the whole table `student` as shown above

# Selecting Attributes

Query: “Output student number and name of all students”

```
select stud_no, name  
from student;
```

student	
stud_no	name
1	Smith, John
2	Miller, Anne
3	Jones, Betty
4	Johnson, Boris

# Comparison With Projection

- This is similar to the relational projection operator
- However, there is one major difference:
  - By default SQL does *not* remove duplicates
  - SQL uses a *multi-set* or *bag* semantic
- If you want to remove duplicates, you have to use the keyword **distinct**



# Example

```
select date_of_birth  
from student;
```

date_of_birth
1990-10-12
1992-07-30
1991-03-24
1992-07-30

```
select distinct date_of_birth  
from student;
```

date_of_birth
1990-10-12
1992-07-30
1991-03-24

# Where Clause

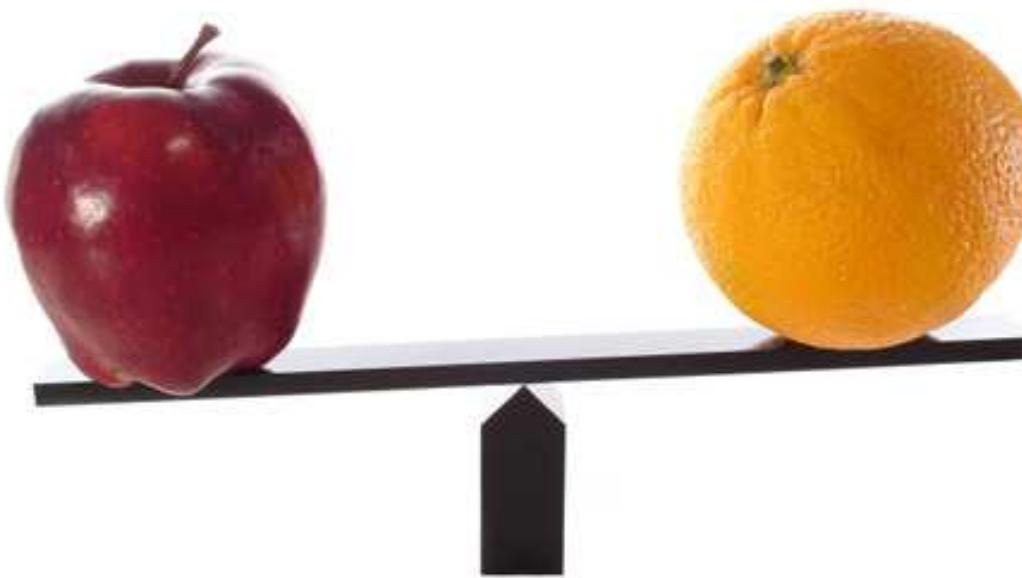
Query: “Give me all available data about students with a student number less than 3”

```
select      *
from        student
where       stud_no < 3;
```

student		
stud_no	name	date_of_birth
1	Smith, John	1990-10-12
2	Miller, Anne	1992-07-30

# Predicates

- In a predicate you can use comparison operators such as  $=$ ,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ , between, like
- Predicates can be combined in a where clause using AND, OR, NOT



# Example For Between

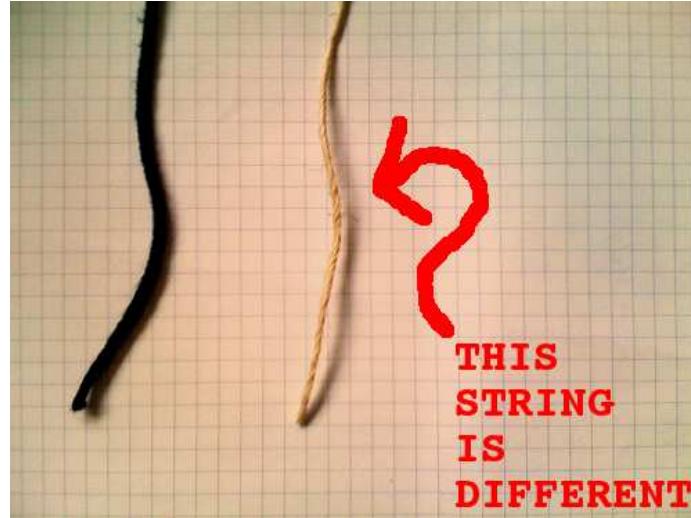
Query: “Give me the names of all students who were born between 1992-01-01 and 1994-01-01”

```
select name  
from student  
where date_of_birth between 1992-01-01 and 1994-01-01;
```

this is equivalent with

```
select name  
from student  
where date_of_birth  $\geq$  1992-01-01  
and date_of_birth  $\leq$  1994-01-01;
```

# String Comparisons

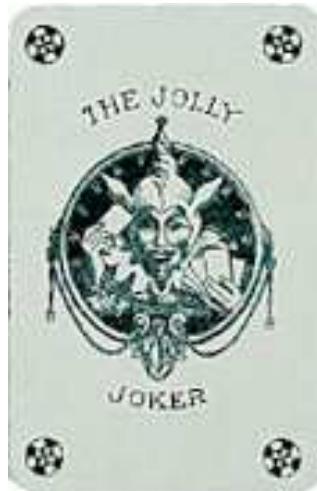


- Literals (string constants) have to be enclosed in single quotes

Query: “Give me all the information about the student named Smith, John”

```
select *
from student
where name = 'Smith, John';
```

# Searching with Wildcards



Query: “Give me all the information about the students whose names start with a J”

```
select *
from student
where name like 'J%';
```

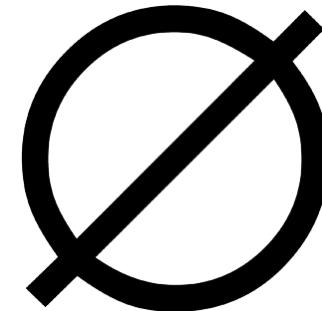
# Two Different Wildcards



- % stands for an arbitrary string of characters (can be empty)
  - Examples: %mit%, Smi%, %ith, Smith%
- \_ stands for a single character
  - Examples: \_mith, Sm\_th, Smit\_, S\_i\_h

# Null Values

- There is a special value in SQL called **NULL**, which is used whenever the value of an attribute is
  - unknown
  - unavailable
  - inapplicable
- To check whether an attribute has a value of **NULL** use the keyword **is**:



```
select      *
from        student
where       date_of_birth is NULL;
```

# Null Values(2)

- If at least one of the operands in an arithmetic expression is **NULL**, then the result is **NULL**
- SQL uses a three-valued logic: true (t), false (f), and unknown (u)



not		and			or		
		t	u	f	t	u	f
t	f	t	u	f	t	t	t
u	u	u	u	f	u	t	u
f	t	f	f	f	f	t	u

- The result of an SQL query contains only tuples for which the **where** clause is true!

# Accessing More Than One Relation

- If there is more than one relation in the **from** clause, they are combined using a Cartesian product
- Example:

Query: “Output all lectures and professors”

```
select *
  from lecture, professor;
```

- This gives the same result as  $(\text{lecture} \times \text{professor})$  in relational algebra

# Joins

- As Cartesian products usually don't make a lot of sense, we want to use joins



- You formulate join predicates in the **where** clause:

```
select *
  from lecture, professor
 where prof_pers_no = pers_no;
```

# Joins(2)

- The different join variants you know from the relational algebra are also available in SQL:

```
select      *
from        R [natural|left outer|right outer|full outer]
                join S [on R.A = S.B];
```

# Joins(3)

- You may have an arbitrary number of relations in the **from** clause
- If you don't want Cartesian products, you have to formulate the join predicates in the **where** clause
- Similar problem as in relational algebra: name collision of attribute names
- Example: joining
  - student (stud\_no, name, date\_of\_birth)
  - attends (stud\_no, course\_no)
  - lecture (course\_no, title, credits)



# Qualified Attribute Names

- In this query you have to specify which `stud_no` and which `course_no` you are referring to
- You can use fully qualified attribute names by prepending them with the name of the relation they belong to:

```
select      *
from        student, attends, lecture
where       student.stud_no = attends.stud_no
and         attends.course_no = lecture.course_no;
```

# Short Form

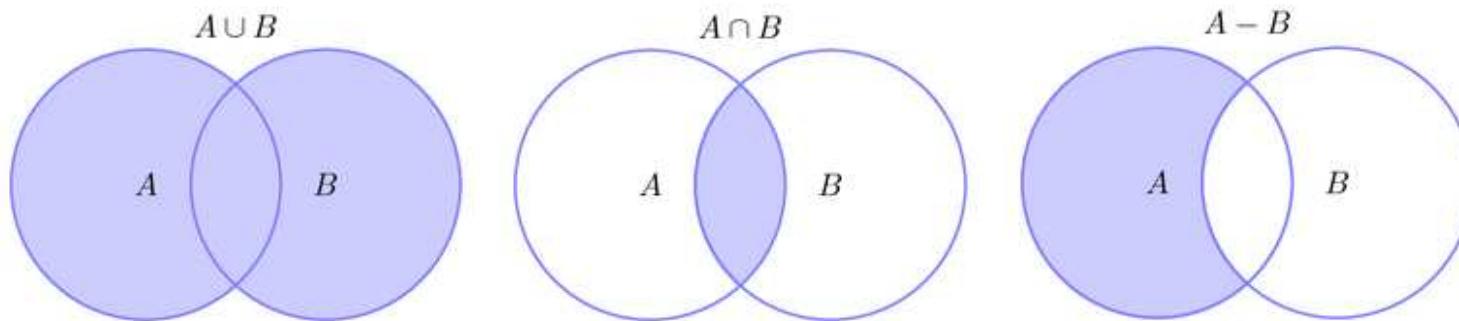
- As this involves lots of typing there is a quicker way
- You can rename the relations

```
select *
  from student S, attends A, lecture L
 where S.stud_no = A.stud_no
   and A.course_no = L.course_no;
```

1.  $\rightarrow$   $S, A, L$   
2.  $\downarrow$   $S$   $A$   $L$   
3.  $\{$   $S$   $A$   $L$   $\}$   
4.  $\{$   $S$   $A$   $L$   $\}$   $\rightarrow$   $S, A, L$ .

# Set Operations

- The usual set operations are also available in SQL: union, intersection, and difference
- Assume (like in relational algebra) that the combined relations follow the same schema



# Union

prof1	
pers_no	name
1	Gamper
2	Helmer

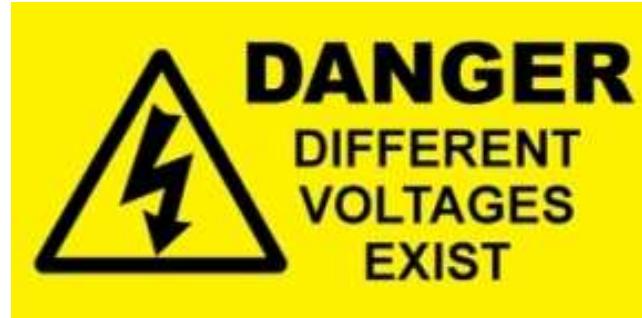
prof2	
pers_no	name
2	Helmer
3	Wood

Query: “Create the union of the two relations”

```
select * from prof1
union
select * from prof2;
```

pers_no	name
1	Gamper
2	Helmer
3	Wood

# Removing Duplicates



- In contrast to `select`, `union` does remove duplicates automatically
- If you want to keep the duplicates, you have to use the `union all` operator

# Intersection

prof1	
pers_no	name
1	Gamper
2	Helmer

prof2	
pers_no	name
2	Helmer
3	Wood

Query: “Which professors can be found in both relations?”

```
select * from prof1  
intersect  
select * from prof2;
```

pers_no	name
2	Helmer

# Set Difference

prof1		prof2	
pers_no	name	pers_no	name
1	Gamper	2	Helmer
2	Helmer	3	Wood

Query: “Which professors are in the first relation but not the second one?”

```
select * from prof1  
except  
select * from prof2;
```

pers_no	name
1	Gamper

# Sorting

- Relations are not sorted by default
- If you need to sort them, you can use the **order by** clause
- Tuples can be sorted in ascending or descending order (default is ascending)



# Example

```
select *  
from student  
order by date_of_birth desc, name;
```

student

stud_no	name	date_of_birth
4	Johnson, Boris	1992-07-30
2	Miller, Anne	1992-07-30
3	Jones, Betty	1991-03-24
1	Smith, John	1990-10-12

# Nested Queries

- SQL allows nested queries, i.e., there may be more than one `sfw-block`
- This nested `sfw-block` can appear in the `where`, `from`, and even the `select` clause
- Basically, the “inner” query computes some kind of intermediate result that is used in the “outer” query

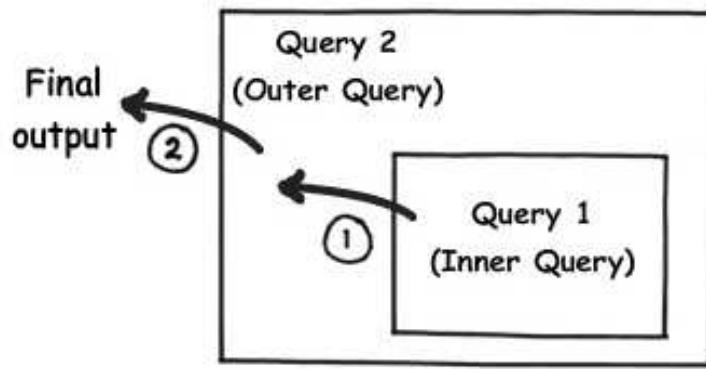


# SFW-Block in a Where Clause

- We have to distinguish two different types of subqueries: *correlated* and *uncorrelated* ones

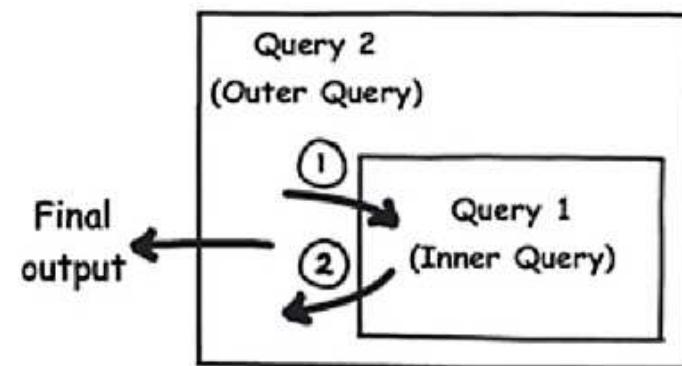
Uncorrelated subquery:

inner query does not refer to attributes of the outer one



Correlated subquery:

inner query references attributes used in the outer query



# Uncorrelated Subquery

Query: “Return the names of all students who attend the lecture with course number 5”

```
select S.name
  from student S
 where S.stud_no in
       (select A.stud_no
        from attends A
       where A.course_no = 5);
```

A blue arrow points from the `S.stud_no` in the `where` clause to the `A.stud_no` in the inner query, and a blue arrow points from the `inner query` label up to the inner query itself.

# Uncorrelated Subquery(2)

- The data only flows from the inner query to the outer query
- Inner query can be evaluated independently of the outer query, i.e., we only run it once
- Then we check for every tuple of the outer query, if the `stud_no` appears in the result of the inner query

# Correlated Subquery

Query: “Find all professors with assistants who work in different areas”

```
select distinct P.name  
from professor P, assistant A  
where A.mgr = P.pers_no  
and exists  
      (select *  
       from assistant B  
       where B.mgr = P.pers_no  
       and not (A.area = B.area));
```

A.area                          ↗ any tuple?  
P.pers\_no

↑  
inner query

# Correlated Subquery(2)

- This time we cannot just run the inner query independently of the outer query
- We need the attribute value of `A.area` and `P.pers_no` from the outer query
- For every tuple of the outer query, we have to run the inner query
- The **exists** predicate is true if the subquery contains at least one tuple

# Other Nested Queries



- Nested queries in **select** and **from** clauses are similar
- Most important differences:
  - If nesting a query in a **select** clause, the tuples returned from inner query may have only one attribute
  - Many DBMSs do not allow correlated nested queries in the **from** clause

# Aggregate Functions

- Attribute values (or whole tuples) can be summarized with the help of *aggregate functions*

**count()** counts tuples

**sum()** sums up attribute values

**avg()** finds the average value

**max()** finds the maximum

**min()** finds the minimum



# Example

student

stud_no	name	date_of_birth
1	Smith, John	1990-10-12
2	Miller, Anne	1992-07-30
3	Jones, Betty	1991-03-24
4	Johnson, Boris	1992-07-30

```
select count(*)  
from student;
```

$$\frac{1}{4}$$

# Example(2)

student

stud_no	name	date_of_birth
1	Smith, John	1990-10-12
2	Miller, Anne	1992-07-30
3	Jones, Betty	1991-03-24
4	Johnson, Boris	1992-07-30

```
select count(distinct date_of_birth)  
from student;
```

$$\frac{1}{3}$$

# Min/Max

Query: “Find the student with the largest stud\_no”

```
select    name, max(stud_no)  
from      Student;
```



- DBMSs won't like this SQL query...

# Min/Max(2)

- Aggregate functions summarize all values in one column to a *single* value
- For the attribute `stud_no` we tell the DBMS that we want the maximum
- However, we don't say anything about the attribute name
  - DBMS has no idea how to merge all these names into one value



# Min/Max(3)

- What's the correct query?
- Has to be done via a nested query

```
select stud_no, name  
from student  
where stud_no =  
      (select max(stud_no)  
       from student);
```

- Inner query finds maximum student number
- Outer query checks every number against that value

# Grouping

- A common query type involves grouping:
  - Divide up tuples into groups
  - Aggregate every group separately



Query: “Determine the number of students attending each lecture”

```
select course_no, count(*) as no_of_students  
from attends  
group by course_no;
```

# Result

attends

stud_no	course_no
1	1
2	1
4	1
1	2
4	2
2	3
4	3

→

course_no	no_of_students
1	3
2	2
3	2

# Grouping(2)

- All attributes *not* appearing in the **group by** clause
  - may only appear in an aggregated form in the **select** clause
- The following query is not correct (same reason as earlier incorrect max query)

```
select prof_pers_no, title, count(*) as number  
from lecture  
group by prof_pers_no;
```



# Having

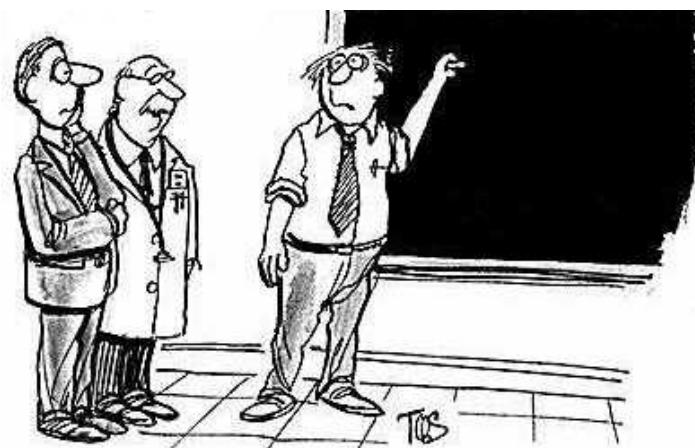
- What if, after grouping/aggregating, we are not interested in all the groups?
- We can't put a predicate into the **where** clause, as it is evaluated before the grouping
- SQL has a **having** clause to do filtering after the grouping/aggregating



# Example

Query: “Find all the professors who give more than three lectures”

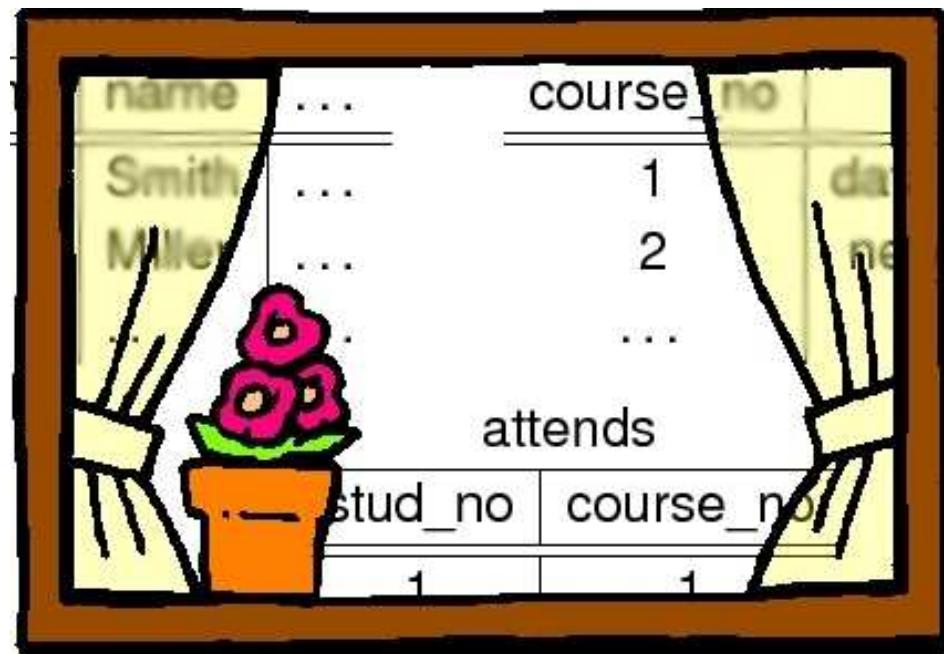
```
select prof_pers_no, count(course_no) as no_of_lect  
from lecture  
group by prof_pers_no  
having count(*) > 3;
```



"It's a clear case of RLS:  
Repetitive Lecture Syndrome."

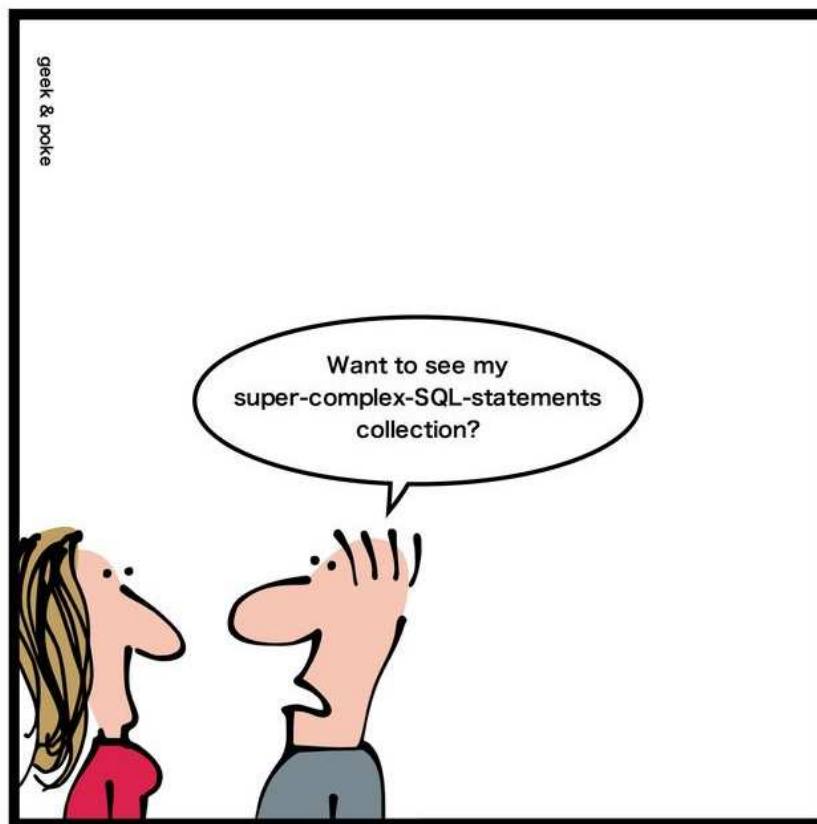
# Views

- Strictly speaking, views belong to the DDL
- However, they are often used to simplify queries, that's why we are covering them here
- Open a “window” into the database in form of a virtual relation



# Complex Query

Query: “Output the names of all professors who give at least one lecture that is worth more than the average number of credits and who have more than three assistants”



How to tell a DB geek

# Complex Query(2)

- We are not formulating this query in one go...
- ...but break it down with the help of views



# Complex Query(3)

- We need all lectures that are worth more than the average number of credits

```
create view aboveAvgCredits as
select course_no, prof_pers_no
from lecture
where credits >
    (select avg (credits)
     from lecture);
```

- We have created a “virtual” relation

aboveAvgCredits (course\_no, prof\_pers\_no)

- Contains the result of the above query
- Can be used in the **from** clause like any other relation

# Complex Query(4)

- We also need to find the professors with more than three assistants

```
create view manyAssistants as
select    mgr
from      assistant
group by  mgr
having    count(*) > 3;
```



# Putting It All Together

- Now we are ready to formulate the main query:

```
select    name  
from      professor  
where     pers_no in  
          (select prof_pers_no  
           from   aboveAvgCredits)  
and pers_no in  
          (select mgr  
           from   manyAssistants);
```



# View Summary

- Further aspects of views:
  - Advantages
    - Can simplify the access to certain tables
    - Can also be used to restrict access (users may only access views, not base tables)
  - Disadvantages
    - Not all views support update operations

# Temporary Views

- Often we need a view for only one query
- Creating the view, running the query, deleting the view results in too much overhead
- For this reason, there are *temporary views*
  - They are specified via the `with` clause
  - Only exist while the associated query is executed

# Example

- For example, if we want to create `aboveAvgCredits` only temporarily, we would write

```
with aboveAvgCredits(cno, pno) as (
    select course_no, prof_pers_no
    from lecture
    where credits >
        (select avg (credits)
         from lecture))
    select *
    from aboveAvgCredits;
```

- The `select * from aboveAvgCredits` is the actual query

# Multiple Temporary Views

- It is also possible to define multiple temporary views
- They are separated with a comma
- So, to formulate the previous complex query completely:

```
with aboveAvgCredits(cno, pno) as (...),
manyAssistants(mgr) as (...)

select name
from professor
where pers_no in
      (select prof_pers_no
       from aboveAvgCredits)
and pers_no in
      (select mgr
       from manyAssistants);
```

# Recursive SQL

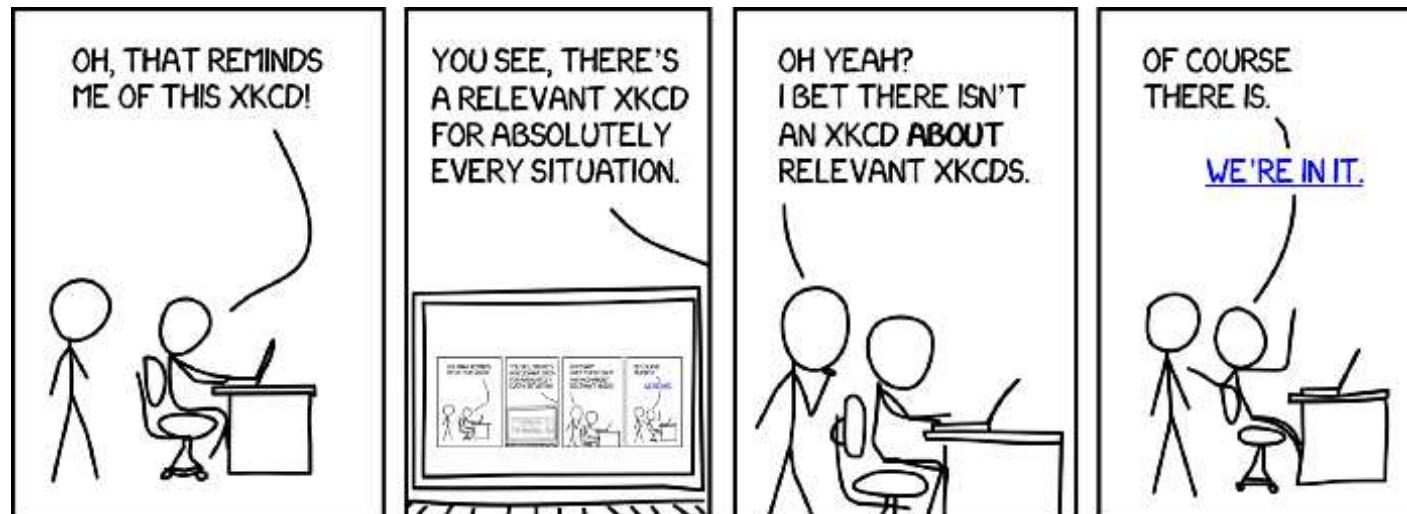
- In our university schema, we have the table requires (prerequisite, advanced) which tells us which lecture is a prerequisite of another
- If we want to know which lectures are required to attend the course 5259, we could ask

```
select    l.course_no, l.title  
from      lecture l, requires r  
where     r.advanced = 5259  
and       l.course_no = r.prerequisite;
```

- But that would only give us the direct requirements
- We are missing the requirements of the requirements
- ... and their requirements and so on

# Recursive SQL (2)

- We could add another join of requires
  - But we would need one for every level of indirection
  - and we would need to know when to stop
- This is very tedious, so SQL-1999 added recursion
- It allows us to compute *transitive closures*
- SQL uses the with-syntax to specify recursive queries



# Recursive SQL (3)

- A recursive query consists of three parts:
  - an initial (non-recursive) subquery
  - a recursive subquery
  - a final query (on the transitive closure)

```
with recursive r( $a_1, a_2, \dots, a_n$ ) as (
    ⟨ initial subquery ⟩
    union
        ⟨ recursive subquery ⟩
)
select *
from    r;
```

# Example

- So, to get all the required courses for course 5259, we need to run the following query

```
with recursive req(courseno) as (
    select prerequisite
    from requires
    where advanced = 5259
    union
    select r.prerequisite
    from requires r, req q
    where q.courseno = r.advanced
)
select *
from req;
```

# Processing a Recursive Query

- A recursive SQL query is processed in the following way
  - The initial subquery is executed once
    - This initializes the content of the temporary view
  - Each time the recursive subquery is executed, it sees only the rows added in the previous iteration
  - The recursive subquery is evaluated until no more rows are added to the temporary view

# A Word of Caution

- We have to make sure we do not enter an infinite loop
- In our example, the requirements form a hierarchy
  - Eventually, we reach the leaves and the query stops
- However, this is not always the case
  - On a general graph we could go round and round
  - In that case, we need to add a stopping condition



# DML



- The Data Manipulation Language (DML) contains operations to
  - insert data
  - delete data
  - change data

# Inserting Data

- The `insert` statement adds (new) tuples to a table
- Adding tuples with literals (constant values) works like this:
  - Assuming we have a value for every attribute:

```
insert into professor  
values(123456, 'Kossmann', 012);
```

- Adds the newly created tuple  
(123456, 'Kossmann', 012)  
to the relation `professor`
- If there is already a tuple with the key 123456 in the relation, this will raise an error

# Inserting Data(2)

- We can insert tuples into a relation that are only partially specified:

```
insert into professor(pers_no, name)  
values(123456, 'Kossmann');
```

- Adds the newly created tuple  
(123456, 'Kossmann', NULL)  
to the relation professor
- A value always has to be specified for the key

# Inserting Data(3)

- It is also possible to copy data from other relations:

```
insert into professor(pers_no, name)
select pers_no, name
from assistant
where pers_no = 111111;
```



# Deleting Data



- The **delete** statement removes tuples from a relation

```
delete from professor  
where pers_no = 123456;
```
- Removes all the tuples that satisfy the predicate in the **where** clause
- **Warning!** If no **where** clause is specified, every tuple in the relation is deleted:delete from professor;

# Changing Data

- In principle data could be changed by deleting the old tuples and inserting new ones
- Can also be done in one go with an **update** statement

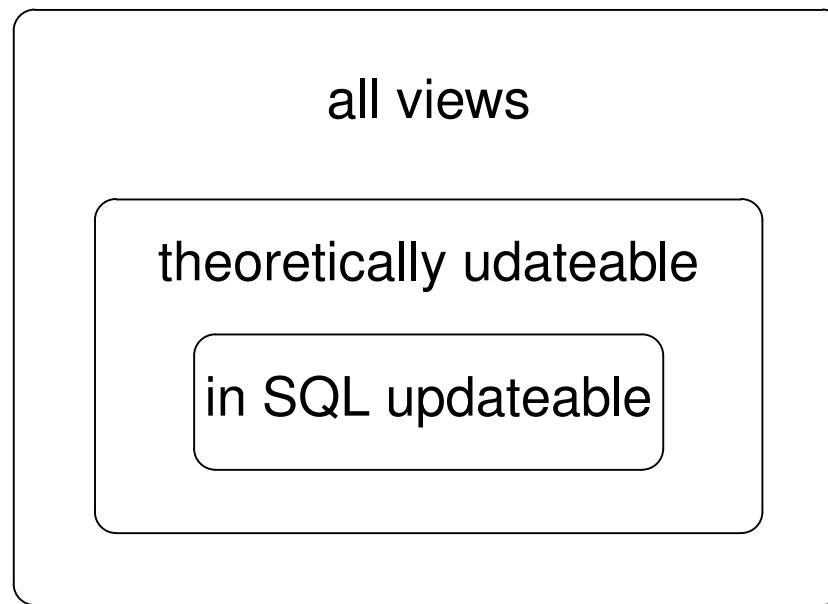
```
update professor  
set     office_no = 121  
where   pers_no = 123456;
```

- The **where** clause identifies the tuples to be changed
- The **set** clause determines the new attribute value



# Updating Views

- A view (in SQL) can only be updated if
  - it contains only one base relation
  - the key of this relation is present
  - there is no aggregation, grouping, or removal of duplicates
- SQL is a bit too strict about this:



# DDL

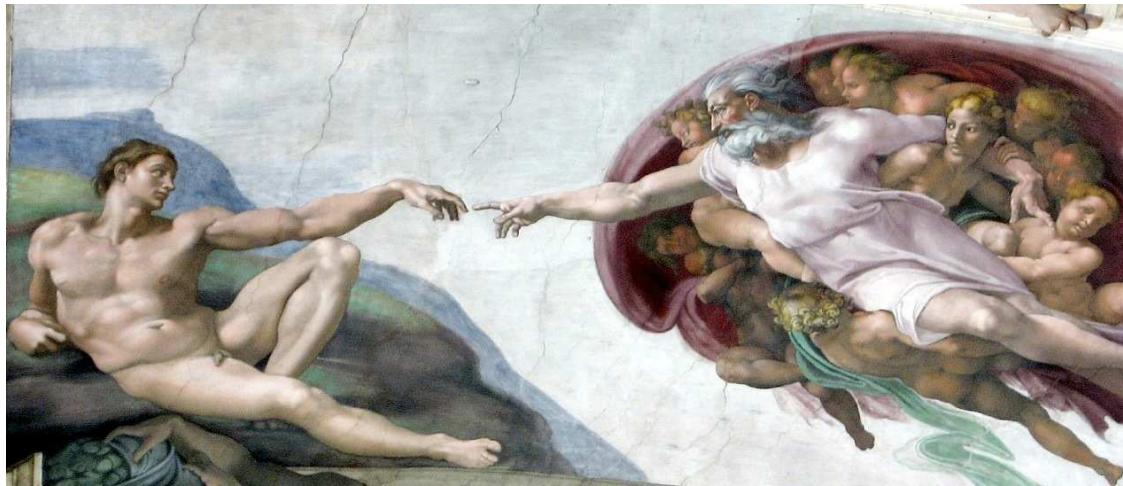


- The Data Definition Language (DDL) is used to define the schema of a database
- This also includes commands to control data access

# Creating Relations

- New relations are created with a **create table** statement

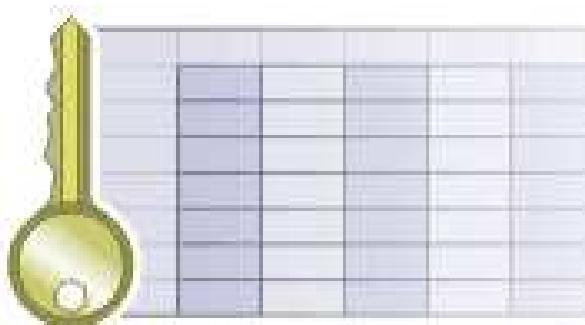
```
create table professor (
    pers_no    integer,
    name       varchar(80),
    office_no  integer
);
```



# Specifying Keys

- The primary key of a relation can be defined in the **create table** statement

```
create table professor (
    pers_no      integer,
    name         varchar(80),
    office_no   integer,
    primary key (pers_no)
);
```



# Enforcing Integrity

- One of the tasks of a DBMS is to keep data consistent
- In SQL integrity requirements are formulated using *constraints*



# Constraints

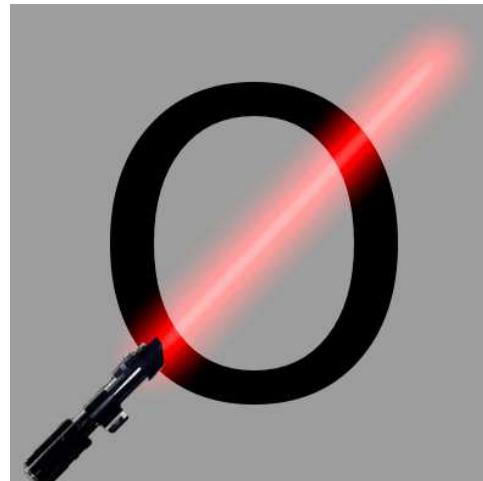
- In addition to primary keys, there is a whole range of other constraints in SQL
- The most important ones are:
  - not null
  - unique
  - check clauses
  - referential integrity
  - default values



# Not Null Constraint

- An attribute declared as **not null** has to have a value when inserting a tuple
- A key attribute is always **not null** (may be done implicitly by a DBMS)

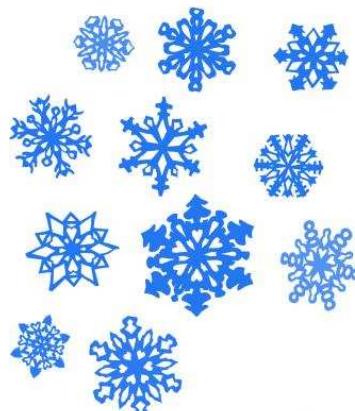
```
create table professor (
    pers_no      integer not null primary key,
    name         varchar(80) not null,
    office_no    integer
);
```



# Unique Constraint

- An attribute declared as **unique** may not contain duplicate values
- A key attribute is always **unique** (may be done implicitly by a DBMS)

```
create table professor (
    pers_no    integer not null unique primary key,
    name       varchar(80) not null,
    office_no  integer unique
);
```



# Check Clauses

- check clauses can be used for various things
- For example, the range of a domain can be restricted

```
create table professor (
    pers_no    integer not null unique primary key,
    name       varchar(80) not null,
    office_no  integer unique
    check (office_no > 0 and office_no < 99999),
);
```



# Check Clauses(2)

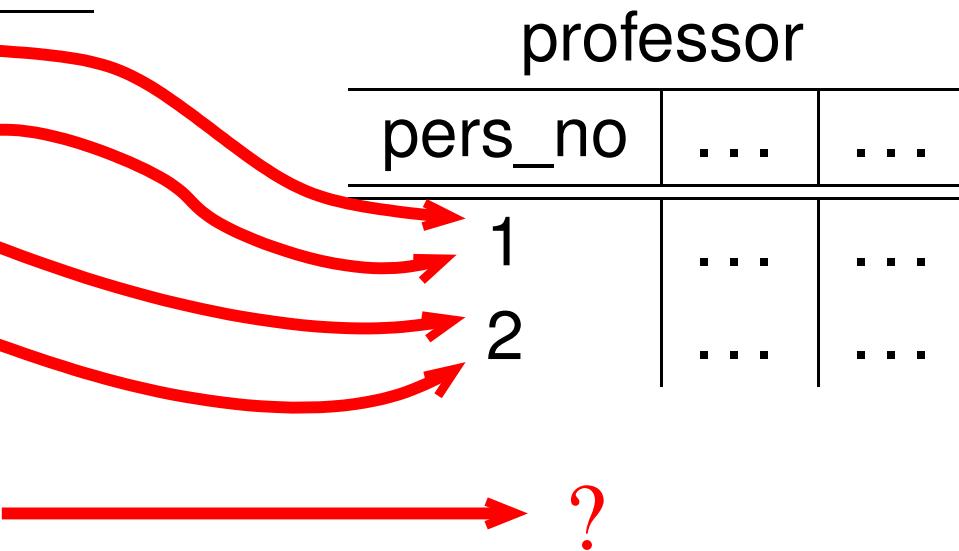
- A **check** clause can contain a full-fledged SQL query:

```
create table attends (
    stud_no      integer,
    course_no   integer,
    check        (stud_no not in
                  (select e.stud_no
                   from examines e
                   where e.course_no = attends.course_no
                     and grade >= 18)),
    primary key (stud_no, course_no)
);
```

# Referential Integrity

lecture

...	...	prof_pers_no
...	...	1
...	...	1
...	...	2
...	...	2
...	...	...
...	...	3982



- Every value used as a foreign key needs to show up as a key value in the referenced relation
- This constraint is called *referential integrity*

# Referential Integrity(2)

- Referential integrity can be enforced in SQL:

```
create table professor (
    pers_no integer primary key,
    ...
);
create table lecture (
    course_no      integer primary key,
    ...
    prof_pers_no  integer not null,
    foreign key (prof_pers_no)
        references professor(pers_no)
);
```

# Referential Integrity(3)

- If a key attribute changes values, this can be propagated to the foreign keys
- **set null:** if a key value is updated or deleted, the corresponding foreign key values are set to **NULL**

lecture		
...	...	prof_pers_no
...	...	1
...	...	1
...	...	NULL
...	...	NULL

professor		
pers_no	...	...
1	...	...
2	...	...

# Referential Integrity(4)

- **cascade:**

- If a key value is updated, the corresponding foreign key values are set to the new value
- if a key value is deleted, the corresponding foreign key values are set to **NULL**

lecture		
...	...	prof_pers_no
...	...	1
...	...	1
...	...	3
...	...	3

professor		
pers_no	...	...
1	...	...
2 → 3	...	...

# Referential Integrity(5)

- SQL syntax:

```
create table lecture (
    course_no      integer primary key,
    ...
    prof_pers_no   integer not null,
    foreign key (prof_pers_no)
        references professor(pers_no)
            [on delete {set null | cascade}]
            [on update {set null | cascade}]
);
```

# Referential Integrity(6)

- Formal definition:
  - $R$  and  $S$  are two relations with schemas  $\mathcal{R}$  and  $\mathcal{S}$
  - $\kappa$  is primary key of  $R$
  - $\alpha \subset \mathcal{S}$  is a foreign key obeying referential integrity, if
    - $s.\alpha$  is either equal to **NULL**
    - or there exists a tuple  $r \in R$  such that  $s.\alpha = r.\kappa$

# Default Values

- If an attribute value is not specified in an insertion statement, the database uses a *default value*
- If no default value has been specified, then **NULL** is used

```
create table assistant (
    pers_no      integer not null primary key,
    name         varchar(80) not null,
    area         varchar(200) default 'computer science'
);
```

# Index Structures

- An *index* speeds up the retrieval (but, unfortunately, slows down update operations)
- Most DBMS automatically create an index on the primary key
  - To quickly check uniqueness of the key values
- More details on indexes follow later, the SQL syntax for creating an index is:

```
create [unique] index index name
on table relation (attribute [asc | desc],
                    attribute [asc | desc], ...)
```

# Removing Objects

- Relations, views, and indexes can be removed with a **drop** statement:
  - **drop table *relation name*;**
  - **drop view *view name*;**
  - **drop index *index name*;**



# DCL



- The Data Control Language (DCL) contains operations to control the flow of transactions
- A transaction is a sequence of interactions between an application/user and the DBMS
- This is an important (and complex) topic and will be covered later

# Variants of SQL

- Most databases have an interactive user interface:
  - User types in an SQL query, DBMS executes it and returns the result (on screen)

The screenshot shows the AdventNet SwisSQL Console 4.1 application window. The interface has three main panes:

- Type SQL:** The top pane contains the original SQL query:

```
SELECT
    employeeid,
    lastname,
    firstname
FROM employees
WHERE rownum < 11
```
- Converted SQL:** The middle pane shows the converted query for Oracle:

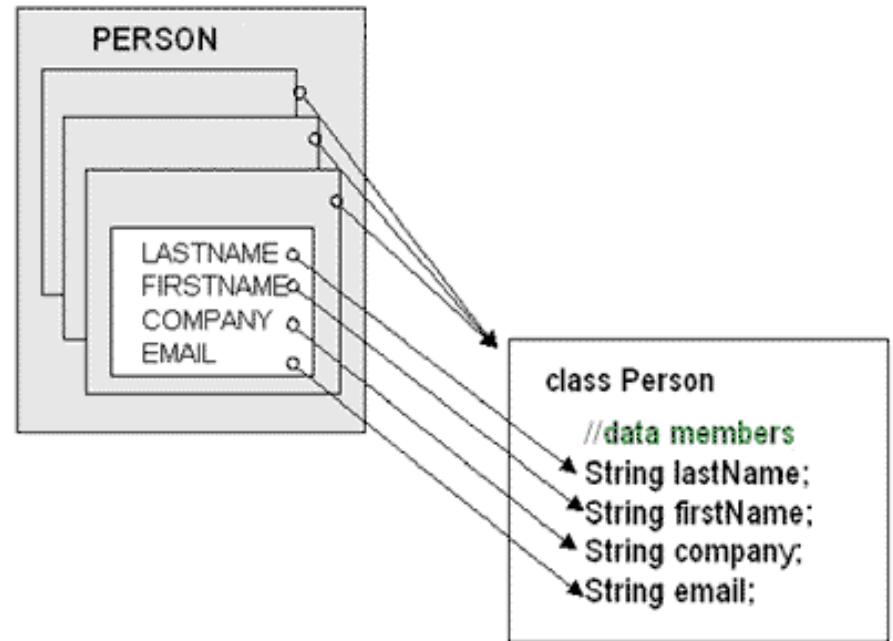
```
SELECT TOP 10
    employeeid,
    lastname,
    firstname
FROM employees
```
- Result Set:** The bottom pane displays the resulting data from the executed query:

employeeid	lastname	firstname
1	HAAS	CHRISTINE
2	THOMPSON	MICHAEL
3	KYAN	SALLY
4	GEYER	JOHN
5	STERN	IRVING
6	PULASKI	EVA

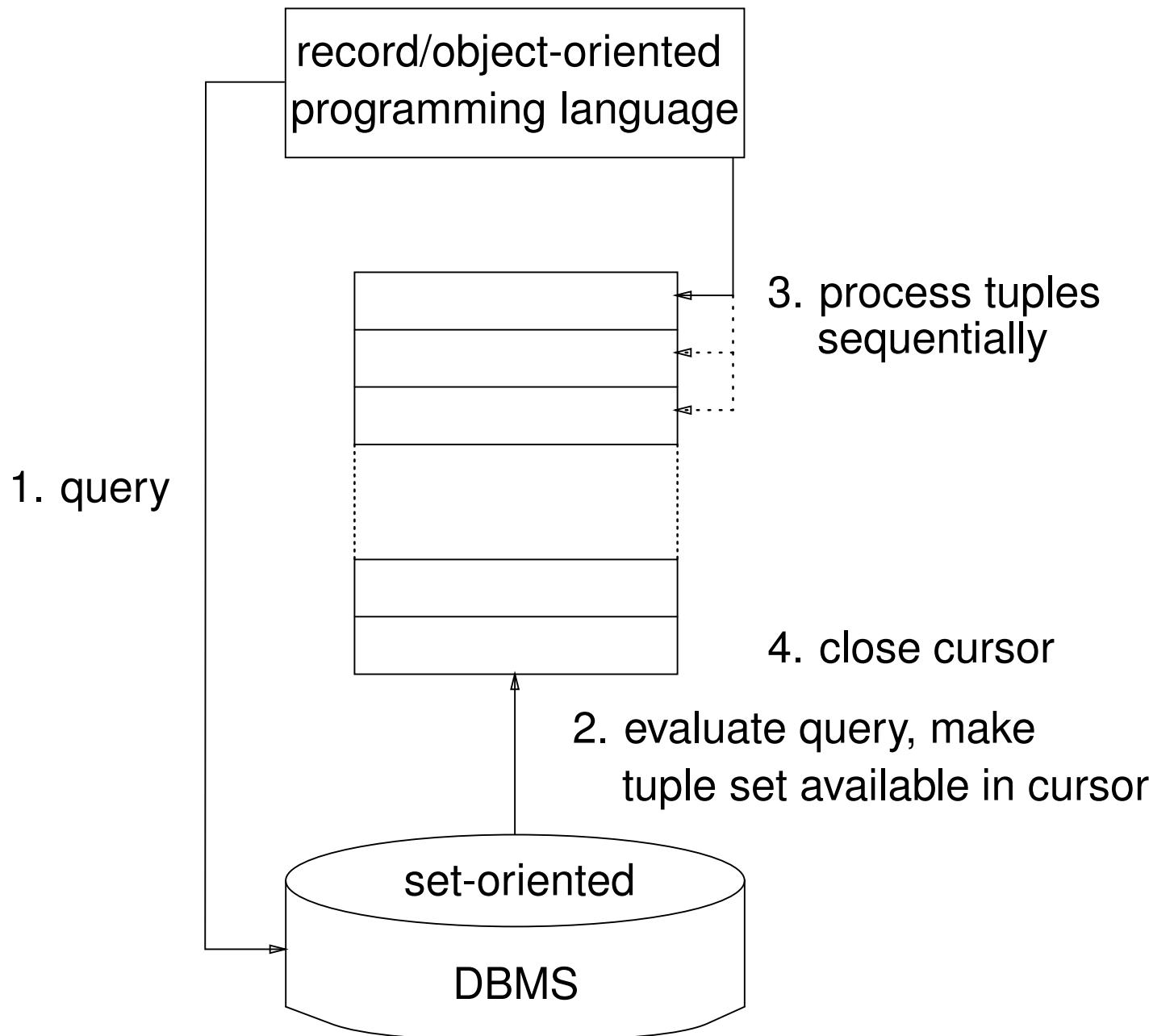
At the bottom of the window, status messages indicate "Query executed successfully!" and "This copy is licensed to swisql".

# Variants of SQL(2)

- However, that's not the only way to query a database
- SQL can be embedded into other (multi-purpose) programming languages
- There's one problem, though: SQL is set-oriented
- Returns a whole set, which has to be mapped to individual records or objects



# SQL Queries in Applications



# Embedded SQL

- SQL commands are embedded directly into the host language (e.g. C, C++, Java)
- SQL commands are identified by a **EXEC SQL** prefix
- A preprocessor replaces them with host language instructions
- A snippet of example code is given below:

```
...
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT name, DOUBLE(population), DOUBLE(product)
    FROM country;

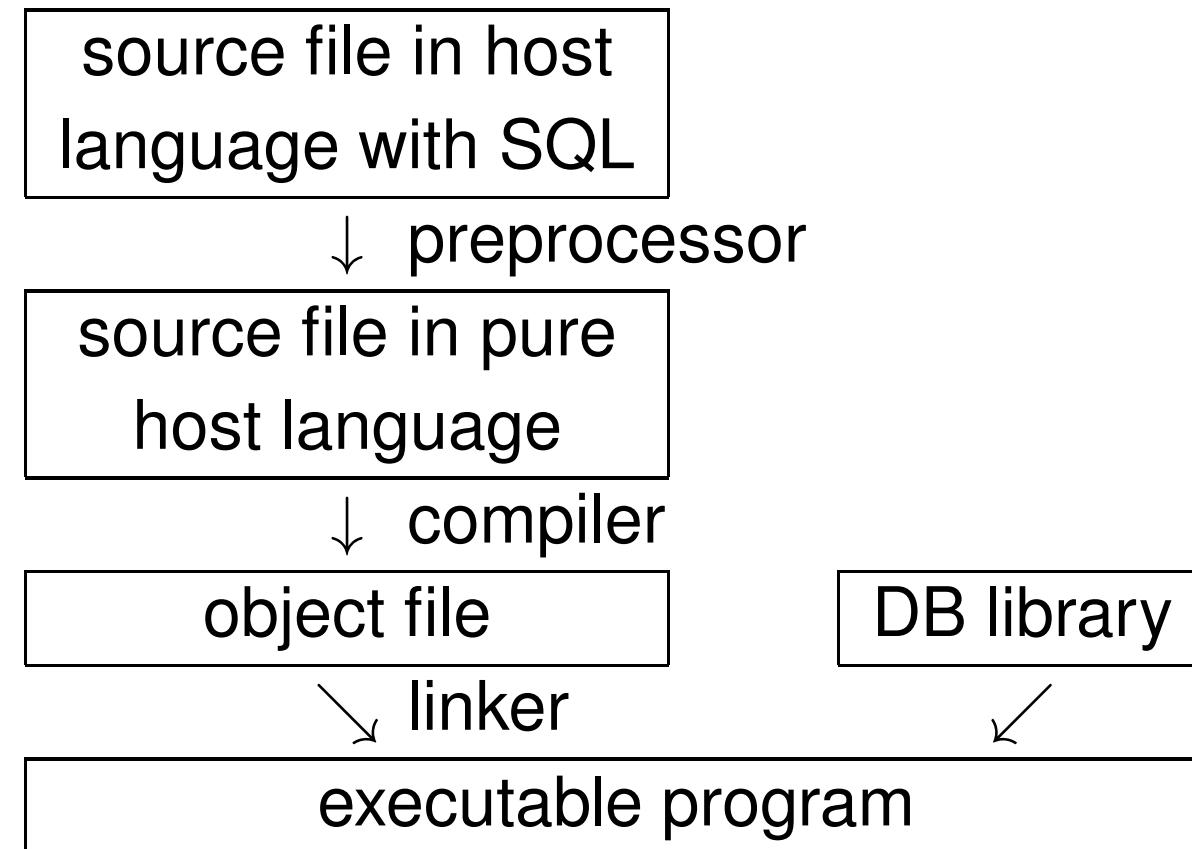
EXEC SQL CONNECT TO GEODB;

EXEC SQL OPEN C1;

counter = 0;
while(strncmp(sqlca.sqlstate, "02000", 5)) {
  ...
}
```

# Embedded SQL(2)

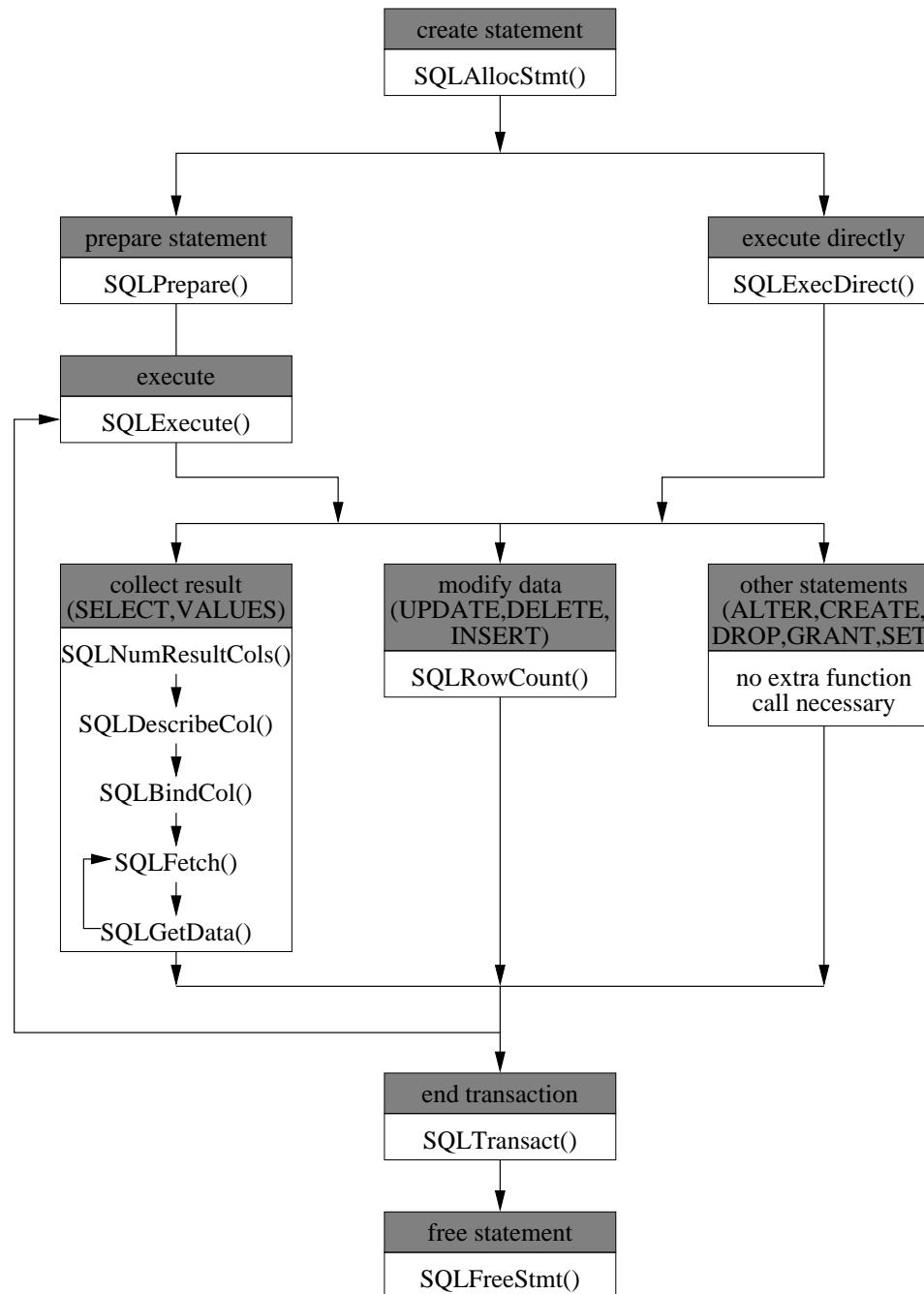
- Steps to create an embedded SQL program:



# Dynamic SQL

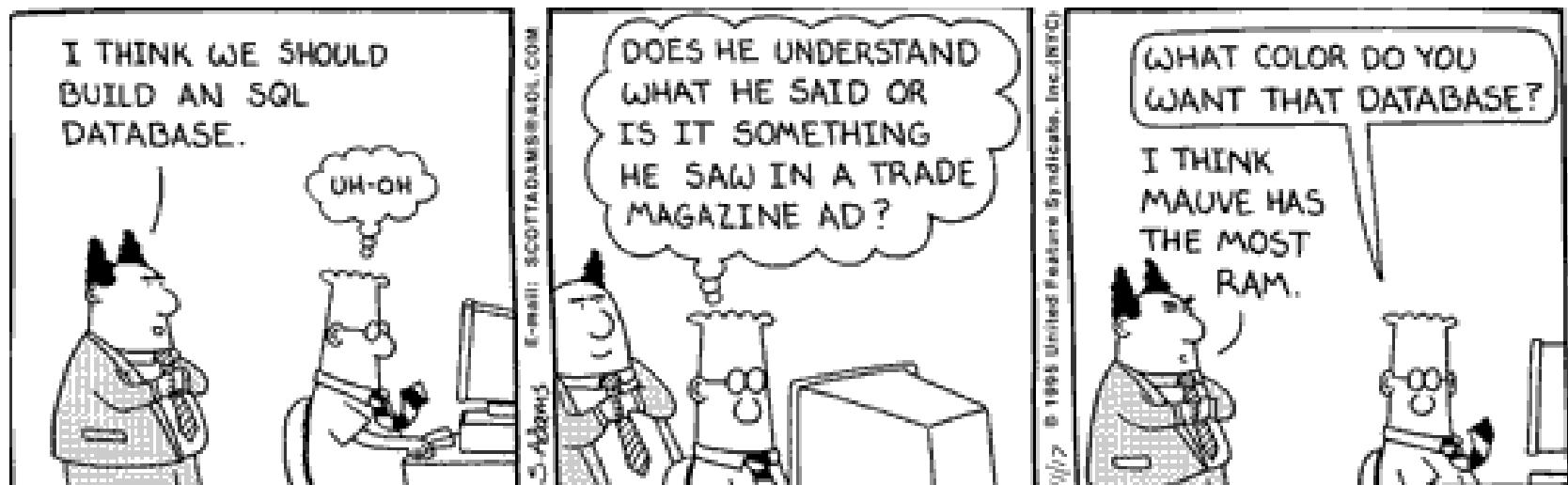
- Dynamic SQL is used when the queries are not yet known during compile time
- SQL queries can be created and evaluated on the fly during program execution
- Relies on standardized interfaces
  - ODBC (Open Database Connectivity)
  - JDBC (for Java)
- More flexible than Embedded SQL; comes at a price:
  - Slower execution
  - Security concerns (SQL injection)

# Dynamic SQL(2)



# Summary

- SQL is *the* standard query language when it comes to relational systems



- SQL contains all the commands and operations necessary to query data and control the DBMS

# Chapter 5

## Active Databases

# Active Data?

- With the help of constraints (primary key, foreign key, check clauses) we increased the data semantics
  - Anything that the database can take care of, does not have to be replicated in each application
- This can be pushed further with the use of *triggers*
  - Triggers are general-purpose actions that are automatically invoked by certain events



# Triggers

- A trigger executes a series of SQL statements whenever data is
  - inserted,
  - deleted,
  - or updatedin a specific table
- This goes much further than a simple constraint
- The activation of a trigger can affect any table or even the world outside the DBMS (e.g. sending an e-mail)
- Triggers can be used for
  - keeping audit trails
  - detecting exceptional conditions
  - maintaining relationships in DBMSs

# Triggers (2)

- Although triggers are part of the SQL standard, there are variations
- An implementation close to the standard is the one used by DB2
- We will be using the DB2 syntax on these slides

# Triggers (3)

- When defining a trigger, you need to specify a couple of things:
- Like a relation or view, it has to have a unique name
- The event activating the trigger has to be specified:
  - INSERT ON **table-name**
  - DELETE ON **table-name**
  - UPDATE (OF **column-name**) ON **table-name**
- A trigger is always attached to a real table (not a view)

# Activation Time

- In addition to the event there is an activation time
- The timing is defined in relation to the event taking place
- Trigger execution takes place either BEFORE or AFTER
- Some examples are

BEFORE INSERT ON R

AFTER DELETE ON S

# Granularity

- There is also the granularity of a trigger
- When an SQL statement activates the trigger,
  - this can be done once for the whole statement
  - or once for each row that is modified
- If an SQL statement does not modify any tuples,
  - it will not activate a FOR EACH ROW trigger
  - but it will still trigger FOR EACH STATEMENT

# Transition Variables

- When executing a trigger, we often need to know the state of the database before and after the event
- For example, when updating tuples, we may need to see which concrete changes have been made
- Information is made available via *transition variables*:
  - Old tuple variable: tuple values before change
  - New tuple variable: tuple values after change
  - Old table variable: table before change
  - New table variable: table after change



# Trigger Condition

- In the presence of a trigger condition a trigger is only activated if this condition is true
- Similar to a WHERE clause, but uses the keyword WHEN
- Some examples:

WHEN (newrow.salary < oldrow.salary)

WHEN (SELECT count(\*) FROM oldtable) > 100

# Trigger Body

- Finally, there is the trigger body consisting of one or more SQL statements
- The trigger body is executed atomically (i.e., it's all or nothing)
- Other important building blocks are
  - Assignments: can be used to modify values to be inserted or updated
  - Signals: can be used to raise errors

# Before/After Triggers

- Although syntactically before and after triggers look similar, they are used differently
- Before triggers act as constraints, checking conditions before allowing certain operations to go ahead
- After triggers usually contain more general application logic



# Before Triggers

- Let's have a look at some examples
- Assume there is a general policy on salaries paid to new employees
- This could be implemented in a before trigger

```
CREATE TRIGGER emp_start_salary
    NO CASCADE BEFORE INSERT ON emp
    REFERENCING NEW AS newrow
    FOR EACH ROW
    SET (salary, bonus) =
        (SELECT salary, bonus
        FROM startingPay
        WHERE jobcode = newrow.jobcode);
```

# Before Triggers (2)

- A trigger can be used to override values provided by the triggering SQL statement
- This trigger limits salary increases:

```
CREATE TRIGGER emp_inc_salary
    NO CASCADE BEFORE UPDATE OF salary ON emp
    REFERENCING OLD AS oldrow NEW AS newrow
    FOR EACH ROW
    WHEN (newrow.salary > 1.5 * oldrow.salary)
    SET newrow.salary = 1.5 * oldrow.salary;
```

# Before Triggers (3)

- Triggers can be used to detect exceptional conditions and roll back SQL statements
- Offer more flexibility than check constraints:

```
CREATE TRIGGER emp_non_fire
    NO CASCADE BEFORE DELETE ON emp
    REFERENCING OLD AS oldrow
    FOR EACH ROW
    WHEN (importance(oldrow.jobcode) > 20)
    SIGNAL SQLSTATE '...' ('Important Person');
```

# After Triggers

- Let's have a look at some examples for after triggers
- This trigger checks for a decreasing salary and interacts with the outside world (via a log file):

```
CREATE TRIGGER emp_dec_salary
  AFTER UPDATE OF salary ON emp
  REFERENCING OLD AS oldrow NEW AS newrow
  FOR EACH ROW
  WHEN (newrow.salary < oldrow.salary)
  BEGIN ATOMIC
    VALUES(logEvent('Salary decrease',
                    CURRENT_TIMESTAMP,
                    oldrow.empno));
    SIGNAL SQLSTATE '...' ('Salary decrease');
  END;
```

# After Triggers (2)

- Next up is a trigger for a meteorological database keeping a table with highest temperatures up-to-date:

```
CREATE TRIGGER extreme_temp
  AFTER UPDATE ON temperatures
  REFERENCING NEW AS newrow
  FOR EACH ROW
  WHEN (newrow.temp >
        (SELECT hightemp
         FROM   extremes
         WHERE  place = newrow.place) )
  UPDATE extremes
    SET hightemp = newrow.temp,
        highdate = CURRENT DATE
    WHERE place = newrow.place;
```

# Recursive Triggers

- The body of a trigger may apply some updates to a database
- These updates may in turn cause other triggers to activate
- It can even happen that the changes made by a trigger activates the same trigger again (directly or indirectly)
  - This is called a *recursive trigger*
- You have to be very careful when writing recursive triggers
  - Can lead to a situation that overwhelms a system's resources

# Recursive Triggers (2)

- Let's look at an example:

```
CREATE TRIGGER add_sales_tax
    AFTER UPDATE ON invoice
    REFERENCING NEW AS newrow
    FOR EACH ROW
    UPDATE invoice
        SET newrow.total_price =
            1.08 * newrow.total_price;
```

- This trigger will activate itself over and over again...

# Constraints vs. Triggers

- Although Triggers are more flexible than constraints, constraints should be preferred
  - Constraints are written in a more declarative way ⇒ DBMS has more opportunities to optimize
  - When created, constraints are enforced for *all existing* data in the database
  - Constraints apply to all kinds of statements, not just specific ones such as insert, delete, and update

# Further Thoughts

- When designing active databases, two aspects have to be balanced out
  - Integrating business logic into a database via triggers
  - Keeping the database maintainable
- Databases tend to have long lives, so any business logic integrated into it makes sure that it is not forgotten
- Having a large number of triggers in a database is hard to maintain and debug

# Summary

- Triggers are a mechanism for formulating complex constraints in a DBMS
- They offer a lot of flexibility
- However, they can be a mixed blessing, as they can be hard to debug

# Chapter 6

## Relational Design Theory

# Relational Design Theory



- A badly designed schema can cause a lot of trouble
- Database designers talk of *anomalies* and distinguish three different types:
  - Update anomaly
  - Insertion anomaly
  - Deletion anomaly

# Relational Design Theory(2)

- Update anomaly:
  - Changing a single fact requires you to touch multiple tuples
- Insertion anomaly:
  - When inserting a tuple not all the required information is available, resulting in lots of **NULL** values
  - In the worst case, value for a key attribute is missing  
⇒ tuple cannot be inserted
- Deletion anomaly:
  - Deleting a tuple removes more information than intended

# Example



- A financial consultant stores data in a relational DBMS
- There are (B)rokers, their (O)ffices, (I)nvestors, (S)hares, (D)ividends, (N)umber of shares an investor holds
- Everything is kept in one big relation:

finance(B, O, I, S, D, N)

# Example(2)

finance

B	O	I	S	D	N
Gekko	103	Trump	IBM	1.20	50
Gekko	103	Trump	Fiat	1.00	30
Gekko	103	Trump	Coca Cola	0.00	100
Fox	214	Smith	BASF	0.80	80
Fox	214	Smith	Fiat	1.00	140
Fox	214	Smith	IBM	1.20	30
....	....	....	....	....	....

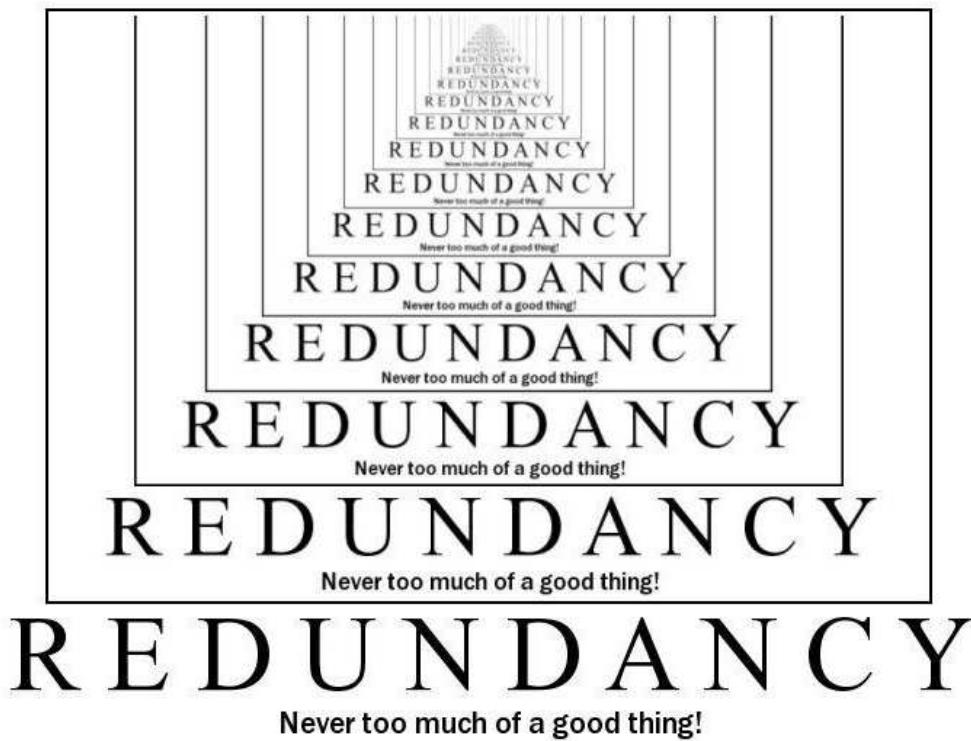
# Example(3)

- What happens, when Gekko moves from 103 to 145?
  - We have to change three (or even more) tuples
- What happens, when Lynch starts working as a broker, but doesn't have any clients yet?
  - We insert [Lynch, 210, NULL, NULL, NULL, NULL]
- What happens, when Smith sells all his shares?
  - He vanishes from the database...



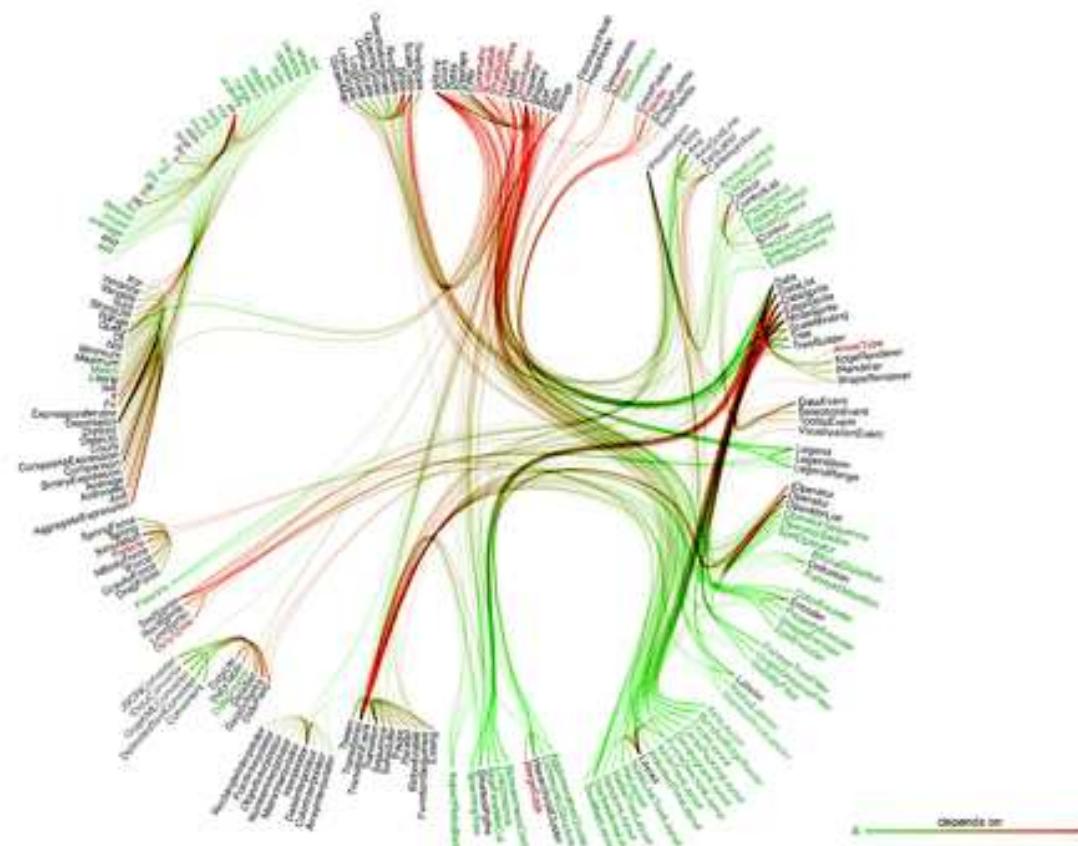
# Taking a Closer Look

- The relation `finance` seems to contain a lot of redundant data:
  - 103 is Gekko's office
  - IBM pays a dividend of 1.20 per share
  - Fox is acting as broker for Smith



# Taking a Closer Look(2)

- Where does this redundancy come from?
  - Certain attribute values determine other attribute values
  - There are *functional dependencies* between attributes



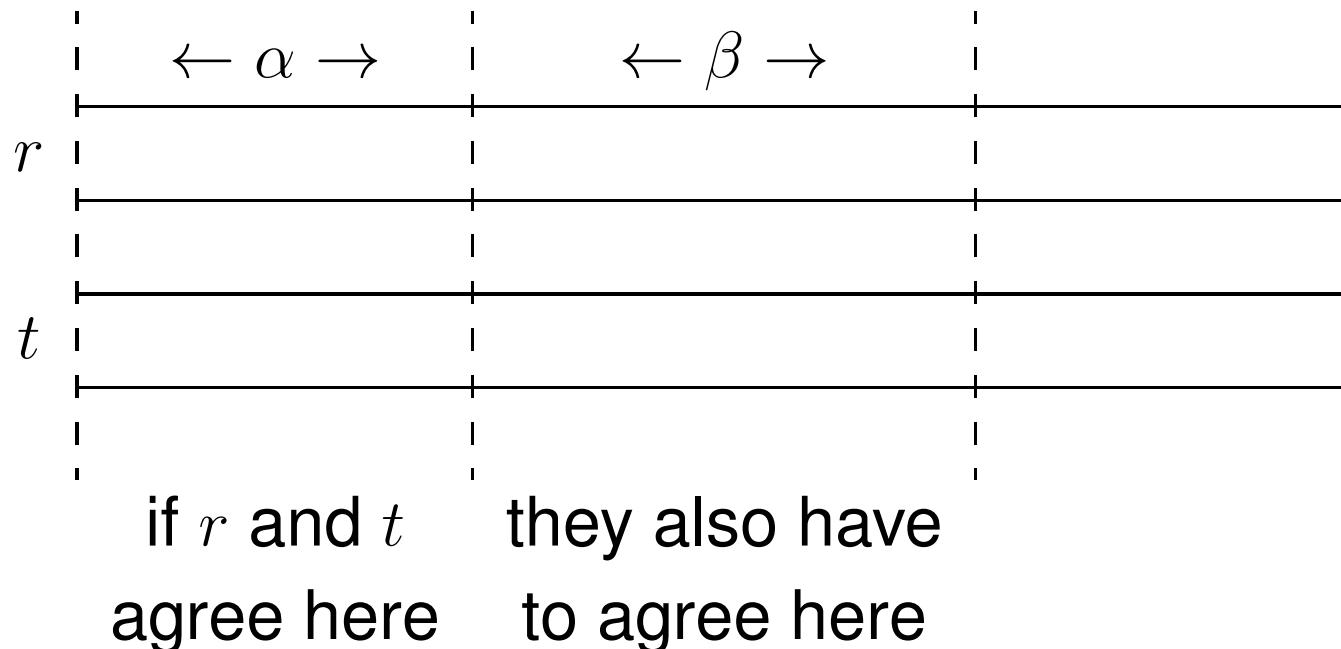
# Functional Dependencies

- For example, there is a functional dependency (FD) between Broker and Office:  
 $\text{Broker} \rightarrow \text{Office}$ 
  - Assuming every broker has one office, the value for  $\text{Broker}$  uniquely determines the value for  $\text{Office}$
- As far as we can see, looks good:
  - Gekko always shows up with office 103, Fox with office 214



# Functional Dependencies(2)

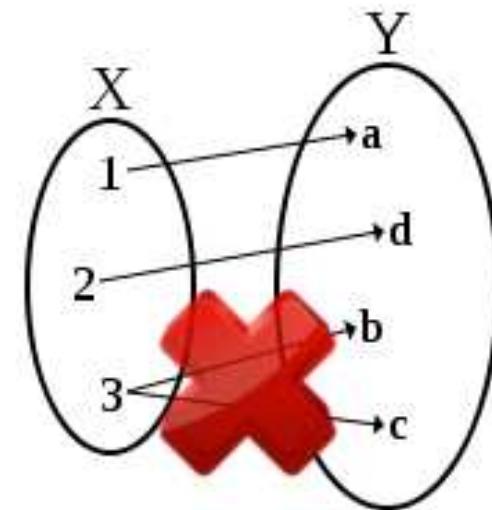
- Formal definition of an FD:
  - $\alpha$  and  $\beta$  are attribute sets of a relational schema  $\mathcal{R}$
  - There is an FD  $\alpha \rightarrow \beta$ , if for all pairs of tuples  $r, t \in R: r.\alpha = t.\alpha \Rightarrow r.\beta = t.\beta$
- Let's say  $\alpha = \{A_1, \dots, A_n\}$  and  $\beta = \{B_1, \dots, B_m\}$



# Functional Dependencies(3)

- A functional dependency has to hold for *all* possible instances of  $\mathcal{R}$
- Using the current instance  $R$ , we can only disprove FDs:

If you find this,  
it's not an FD



- FDs have to be derived from background knowledge of the application
- Can you find other FDs in finance?

# Functional Dependencies(4)

- $\mathcal{F}_R = \{B \rightarrow O, S \rightarrow D, I \rightarrow B, IS \rightarrow N\}$
- What do we do with these FDs?
- First of all, we can now *compute* a key for  $R$



# Properties of a Key

- A key  $\kappa$  has the following properties:
  1.  $\kappa \subseteq \mathcal{R}$
  2.  $\kappa \rightarrow \mathcal{R}$
  3. There is no  $\kappa' \subset \kappa$  such that  $\kappa' \rightarrow \mathcal{R}$
- Property 2 means the key is *complete*
- property 3 means the key is *minimal*

# Properties of a Key (2)

- If  $\kappa$  satisfies all the properties, it's called a *candidate key*
  - A relation may have more than one candidate key
- Choose one of candidate keys as the *primary key*
- If  $\kappa$  only satisfies properties 1 and 2, it's called a *superkey*
  - (candidate) key  $\subseteq$  superkey

# Computing a Key

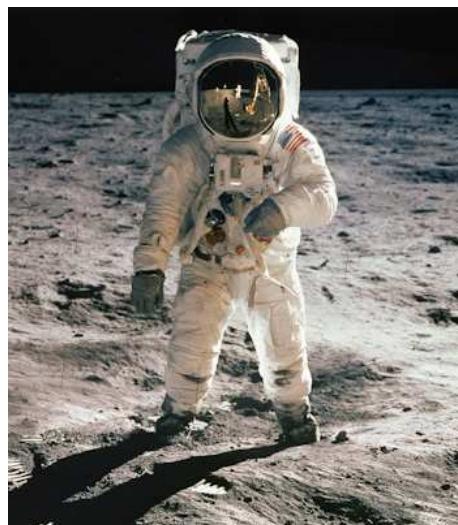
- Is IS a key of finance?
- Property 1: ✓
- Checking properties 2 and 3 requires a few more concepts of relational design theory

*The  
Theory  
Of  
Everything*



# Reasoning about FDs

- We can infer additional FDs from a set of given  $\mathcal{F}$
- The *closure*  $\mathcal{F}^+$  is the set of all FDs that can be inferred from  $\mathcal{F}$
- There are rules, called *Armstrong's Axioms*, to do this



# Armstrong's Axioms

- Let  $\alpha, \beta$  and  $\gamma$  be subsets of  $\mathcal{R}$
- We have three axioms:
  - Reflexivity:  $\beta \subseteq \alpha \Rightarrow \alpha \rightarrow \beta$
  - Augmentation: If  $\alpha \rightarrow \beta$ , then  $\alpha\gamma \rightarrow \beta\gamma$
  - Transitivity:  $\alpha \rightarrow \beta$  and  $\beta \rightarrow \gamma \Rightarrow \alpha \rightarrow \gamma$
- Try this out on the relational schema for finance:  
 $\mathcal{R} = (B, O, I, S, D, N)$   
 $\mathcal{F}_{\mathcal{R}} = \{B \rightarrow O, S \rightarrow D, I \rightarrow B, IS \rightarrow N\}$   
Which attributes are functionally determined by  $IS$ ?

# Closure of a Set of Attributes

- $AC(\alpha)$  is the closure of a set of attributes  $\alpha$ 
  - $AC(\alpha)$  contains all the attributes functionally determined by  $\alpha$  (given  $\mathcal{F}_R$ )
- Figuring out  $AC(\alpha)$  by using Armstrong's Axioms can be a bit tiresome
- We don't have to, there's a simple algorithm:

```
 $AC := \alpha$ 
while ( $AC$  still changes) do
    for each FD  $\beta \rightarrow \gamma$  in  $\mathcal{F}_R$  do
        if ( $\beta \subseteq AC$ )
            then  $AC := AC \cup \gamma$ 
```

# Closure of a Set of Attributes(2)

- We can use this to check properties 2 and 3 of a key:
  - $AC(IS) = BOISDN$   
 $\Rightarrow IS$  is a superkey
  - $AC(I) = BOI$   
 $AC(S) = SD$   
 $\Rightarrow IS$  is also a candidate key



# The Next Step

- Now that we know how to find keys, what do we do next?
- Design theory can be used to check the quality of a schema



- This is done via *normal forms* (NFs): 1NF, 2NF, 3NF, BCNF, 4NF
  - (Usually) the higher, the better

# First Normal Form (1NF)

- Assume we want to store parent/child relationships in a relation:

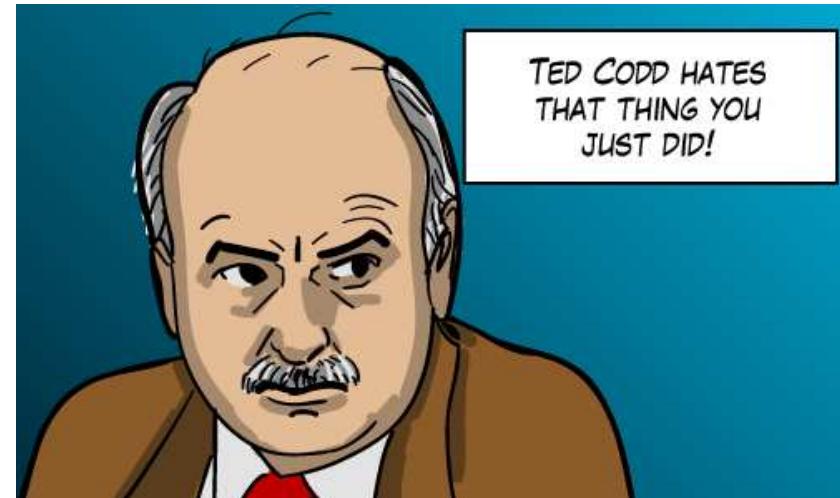
parents		
mother	father	child
Mary	John	Dave
Sue	Mike	Kate
...	...	...

- What do we do if a couple has more than one child?
- We have to get this information into the relational table structure

# First Normal Form(2)

- We could store the names of the children in a tuple or add columns:

parents		
mother	father	children
Mary	John	[Jen, Dave]
...	...	...



parents					
mother	father	child1	child2	child3	...
Mary	John	Jen	Dave	NULL	...
Sue	Mike	Kate	NULL	NULL	...
...	...	...	...	...	...

# First Normal Form(3)

- The attribute `children` has a domain, e.g. `char(30)`
- First solution takes more than one value from that domain and assigns it to an attribute
  - Breaks the relational table structure
- We could change the domain: have a list of `char(30)`
  - Adding constraints becomes more difficult
  - Formulating queries such as “list all parents who have children with the same name” also becomes harder

# First Normal Form(4)



- Second solution seems to follow relational table structure, but has problems as well
  - How many columns do we choose? Every additional column adds lots of NULL values
  - Querying for children with the same name needs to compare every column with every other column

# First Normal Form(5)

- A relational schema is in 1NF, if all attribute values are atomic:
  - an attribute may take on only one value from its domain

parents (in 1NF)		
mother	father	child
Mary	John	Jen
Mary	John	Dave
Sue	Mike	Kate
...	...	...

# Second Normal Form (2NF)

- A relational schema is in 2NF, if
  - it is in 1NF
  - and every non-key attribute (NKA) is fully functional dependent on every key
- What's a *full functional dependency*?
- $\beta$  is fully functional dependent on  $\alpha$  ( $\alpha \xrightarrow{*} \beta$ ), if
  - $\alpha \rightarrow \beta$  and
  - there is no  $\alpha' \subset \alpha$  such that  $\alpha' \rightarrow \beta$

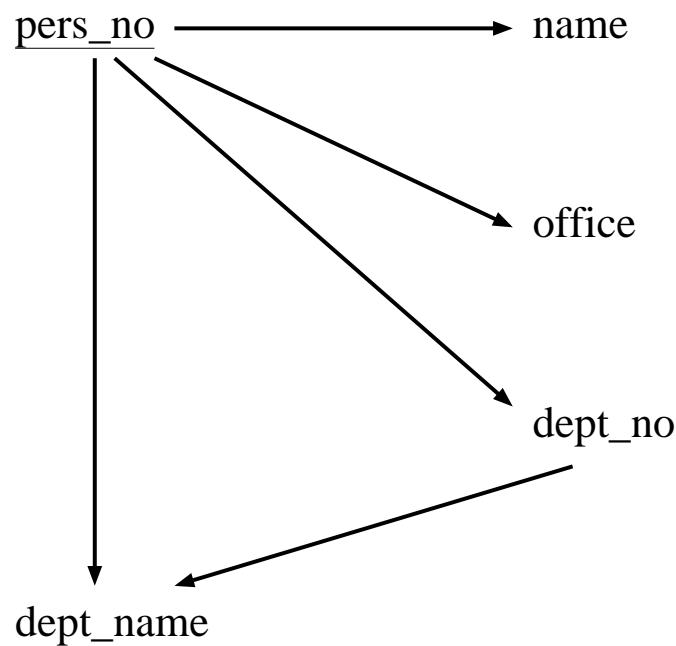
# Second Normal Form(2)

- For example, finance is not in 2NF because
  - $D$  is a non-key attribute
  - It is dependent on  $S \Rightarrow$  not fully functional dependent on key  $IS$
- If we violate 2NF, that means we combine different entity relationships in one table
  - In the case of finance, the relationships between shares and dividends & investors and brokers
  - Causes redundancy



# Third Normal Form (3NF)

- Even if a relational schema is in 2NF, there may still be redundancies in it
- These are caused by transitive dependencies
- For example, adding data about departments to the relation `professor` may result in one

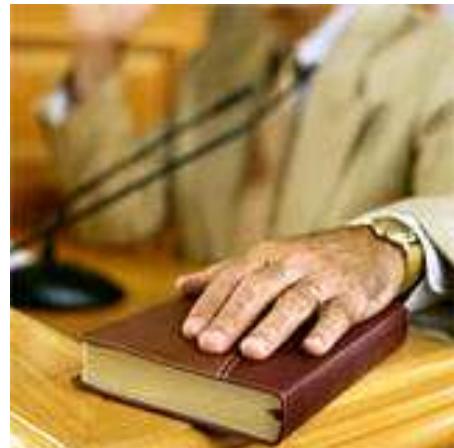


- `dept_name` is a non-key attribute
- There is a transitive dependency between the key `pers_no` and another non-key attribute, `dept_no`:

$\text{pers\_no} \rightarrow \text{dept\_no} \rightarrow \text{dept\_name}$

# Third Normal Form(2)

- A schema is in 3NF, if for every FD  $\alpha \rightarrow \beta$  at least one of the following conditions holds
  - $\alpha \rightarrow \beta$  is trivial, i.e.,  $\beta \subseteq \alpha$
  - $\alpha$  is a superkey
  - Every attribute in  $\beta$  is part of a candidate key



“Every non-key attribute has to depend on the key, the whole key, and nothing but the key, so help me Codd”

# Boyce-Codd Normalform (BCNF)

- Usually, designers are already quite happy with 3NF
- Only in rare cases does a schema in 3NF still contain redundancies
- For example, assume a restaurant offers a “meal deal”:
  - There are starters, main dishes, and drinks
  - For a meal you choose one of each type



# Boyce-Codd Normalform(2)

- The ordered meals are stored in a table:

meals		
(N)umber	(D)ish	(T)ype
1	bruschetta	starter
1	penne con salmone	main
1	white wine	drink
2	soup of the day	starter
2	pizza funghi	main
2	white wine	drink
...	...	...

- We have the following FDs:  
 $ND \rightarrow T$ ,  $NT \rightarrow D$ ,  $D \rightarrow T$
- That means,  $ND$  and  $NT$  are candidate keys

# Boyce-Codd Normalform(3)

- This schema is in 3NF:
  - *Every* attribute in this schema is part of some candidate key
  - Third condition of 3NF always holds
- However, there is still redundancy: we already know that white wine is a drink



# Boyce-Codd Normalform(4)

- We need to tighten the conditions
- A schema is in BCNF, if for every FD  $\alpha \rightarrow \beta$  at least one of the following conditions holds
  - $\alpha \rightarrow \beta$  is trivial, i.e.,  $\beta \subseteq \alpha$
  - $\alpha$  is a superkey

“*Every attribute (including key attributes) has to depend on the key, the whole key, and nothing but the key*”

- A schema in BCNF does not have any redundancies caused by *functional* dependencies
- Unfortunately, this is →



# Multi-valued Dependencies

- Let's look at the following relation:

skills		
pers_no	lang	prog_lang
3002	English	C
3002	Italian	C
3002	English	Java
3002	Italian	Java
3005	German	C
3005	Italian	C

- We have no FDs whatsoever, but there are still redundancies

# Multi-valued Dependencies(2)

- Someone speaking four languages and knowing five programming language
  - needs twenty tuples in this relation!
- We are throwing together two completely independent aspects
- Can be stored more compactly in two relations:

skills	
pers_no	lang
3002	English
3002	Italian
3005	German
3005	Italian

skills	
pers_no	proglang
3002	C
3002	Java
3005	C

# Multi-valued Dependencies(3)

- We have something called multi-valued dependencies (MVDs) here:  
 $\text{pers\_no} \rightarrow\!\!> \text{lang}$  and  $\text{pers\_no} \rightarrow\!\!> \text{proglang}$
- Because the languages and programming languages a person knows are completely independent
  - every possible combination has to show up for a particular person



# Multi-valued Dependencies(4)

- There's also a formal definition:
- Let  $\alpha, \beta, \gamma \subseteq \mathcal{R}$  (with  $\alpha \cup \beta \cup \gamma = \mathcal{R}$ )
- $\alpha \rightarrow\!\!\! \rightarrow \beta$  holds if for every instance  $R$ :  
for every pair of tuples  $t_1, t_2$  in  $R$  with  $t_1.\alpha = t_2.\alpha$   
exists a tuple  $t_3 \in R$  with
  - $t_3.\alpha = t_1.\alpha (= t_2.\alpha)$
  - $t_3.\beta = t_1.\beta$
  - $t_3.\gamma = t_2.\gamma$
- Formulated differently: for all tuples with the same value  
for  $\alpha$ , all possible  $\beta, \gamma$ -combinations have to appear

# Multi-valued Dependencies(5)

- MVDs are a generalization of FDs, i.e., every FD is an MVD (but not necessarily the other way around)
- Similar to the Armstrong Axioms for FDs, there are also inference rules for MVDs
  - We are not going to go into details here...
- One important property is the following, though:  
 $\alpha \rightarrow\!\!\! \rightarrow \beta \Rightarrow \alpha \rightarrow\!\!\! \rightarrow \gamma$  for  $\gamma = \mathcal{R} - \alpha - \beta$

# Fourth Normal Form (4NF)

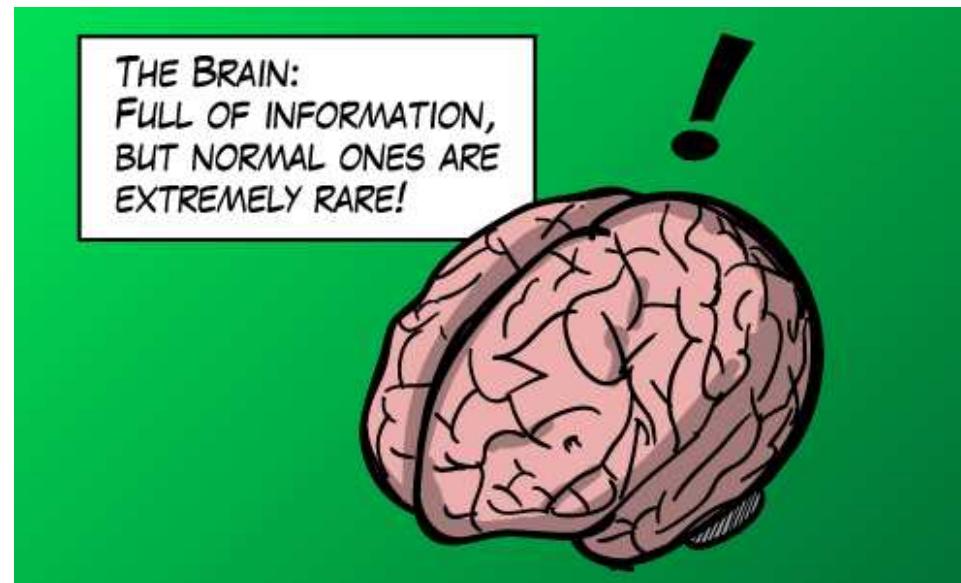
- 4NF is a generalization of BCNF to MVDs
- A relational schema is in 4NF, if for every MVD  $\alpha \rightarrow\!\!\!\rightarrow \beta$  at least one of the following conditions holds
  - $\alpha \rightarrow\!\!\!\rightarrow \beta$  is trivial, i.e.,  $\beta \subseteq \alpha$  or  $\alpha \cup \beta = \mathcal{R}$
  - $\alpha$  is a superkey
- There are even higher normal forms than 4NF...



- Mainly of academic interest, not covered here

# Data Quality

- A higher normal form includes all the lower ones:  
 $4NF \subset BCNF \subset 3NF \subset 2NF \subset 1NF$
- The higher the normal form, the better (should be at least 3NF)
- What do you do if you're not satisfied with the normal form of your current schema?



# Decomposition of Relations

- A relational schema can be transformed into a higher normal form by taking it apart
  - This can be done by *decomposing* a schema  $\mathcal{R}$  into subschemas  $\mathcal{R}_1, \dots, \mathcal{R}_n$  with  $\mathcal{R}_i \subset \mathcal{R}$  for  $1 \leq i \leq n$



finance					
B	O	I	S	D	N
Gekko	103	Trump	IBM	1.20	50
Gekko	103	Trump	Fiat	1.00	30
Gekko	103	Trump	Coca Cola	0.00	100
Fox	214	Smith	BASF	0.80	80
Fox	214	Smith	Fiat	1.00	140
Fox	214	Smith	IBM	1.20	30
...	...	...	...	...	...

# Decomposition of Relations(2)

- A decomposition needs to have two properties to be useful
  - Recoverability of information:  
we are able to reconstruct the original instance  $R$  from the instances  $R_1, \dots, R_n$  (for all possible instances  $R$ )
  - Preservation of dependencies:  
all FDs in  $\mathcal{F}_R$  are also found in  $\mathcal{F}_{R_1}, \dots, \mathcal{F}_{R_n}$



# Recoverability of Information

- Assume we have a decomposition of  $\mathcal{R}$  into  $\mathcal{R}_1$  and  $\mathcal{R}_2$ 
  - with  $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$  and  $R_1 = \pi_{\mathcal{R}_1}(R)$ ,  $R_2 = \pi_{\mathcal{R}_2}(R)$
- This decomposition recovers all information, if for every possible instance of  $R$  the following holds

$$R = R_1 \bowtie R_2$$

- How do we check this for *every* instance?



# Recoverability of Information(2)

- Luckily we don't have to!



- It's fine if at least one of the following conditions hold:
  - $(\mathcal{R}_1 \cap \mathcal{R}_2) \rightarrow \mathcal{R}_1 \in \mathcal{F}_{\mathcal{R}}^+$
  - $(\mathcal{R}_1 \cap \mathcal{R}_2) \rightarrow \mathcal{R}_2 \in \mathcal{F}_{\mathcal{R}}^+$

# What Happens If We Get It Wrong?

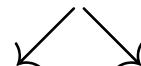
$$\mathcal{F}_R = \{B \rightarrow O, S \rightarrow D, I \rightarrow B, IS \rightarrow N\}$$

finance

B	O	I	S	D	N
Gekko	103	Trump	IBM	1.20	50
Gekko	103	Trump	Fiat	1.00	30
Fox	214	Smith	Fiat	1.00	140
Fox	214	Smith	IBM	1.20	30

finance1

finance2



B	O	S	I	S	D	N
Gekko	103	IBM	Trump	IBM	1.20	50
Gekko	103	Fiat	Trump	Fiat	1.00	30
Fox	214	Fiat	Smith	Fiat	1.00	140
Fox	214	IBM	Smith	IBM	1.20	30

# What Happens If We Get It Wrong?(2)

finance1  $\bowtie$  finance2

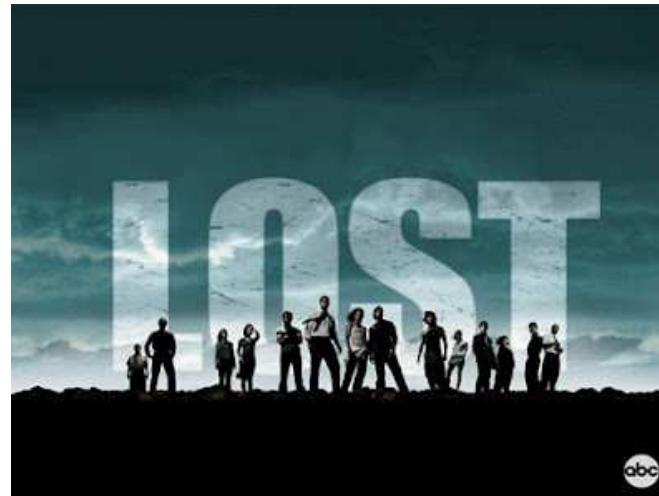
finance (reconstructed)

B	O	I	S	D	N
Gekko	103	Trump	IBM	1.20	50
Gekko	103	Trump	Fiat	1.00	30
Fox	214	Smith	Fiat	1.00	140
Fox	214	Smith	IBM	1.20	30
Gekko	103	Smith	Fiat	1.00	140
Gekko	103	Smith	IBM	1.20	30
Fox	214	Trump	IBM	1.20	50
Fox	214	Trump	Fiat	1.00	30

We end up with more tuples than before!

# Dependency Preservation

- Given a decomposition of  $\mathcal{R}$  into  $\mathcal{R}_1, \dots, \mathcal{R}_n$
- Decomposition preserves dependencies, if  
 $\mathcal{F}_{\mathcal{R}} \equiv (\mathcal{F}_{\mathcal{R}_1} \cup \dots \cup \mathcal{F}_{\mathcal{R}_n})$  or  
 $\mathcal{F}_{\mathcal{R}}^+ = (\mathcal{F}_{\mathcal{R}_1} \cup \dots \cup \mathcal{F}_{\mathcal{R}_n})^+$ , respectively
- In the previous (non-recoverable) decomposition we also lose dependencies:
  - $I \rightarrow B$  is not included anymore



# Decomposition Algorithms

- Decomposing relations on a trial-and-error basis and then checking them would be very tedious
  - There are algorithms to do this systematically



- Most important one: 3NF Synthesis Algorithm
  - Decomposes a schema  $\mathcal{R}$  into 3NF guaranteeing recoverability and dependency preservation
  - Prerequisite: a minimal basis for the set of FDs  $\mathcal{F}_{\mathcal{R}}$

# Minimal Basis

- $\mathcal{F}_c$  is a *minimal basis* or *canonical cover* of  $\mathcal{F}$  if the following three properties hold:
  - $\mathcal{F}_c \equiv \mathcal{F}$ , i.e.,  $\mathcal{F}_c^+ = \mathcal{F}^+$
  - There are no FDs  $\alpha \rightarrow \beta$  in  $\mathcal{F}_c$  in which  $\alpha$  or  $\beta$  contain unnecessary attributes
  - The left-hand side of every FD in  $\mathcal{F}_c$  is unique
    - This can be achieved by applying the union rule:  
$$\alpha \rightarrow \beta, \alpha \rightarrow \gamma \Rightarrow \alpha \rightarrow \beta\gamma$$



# Minimal Basis(2)



- How do we check for “unnecessary” attributes?
  - $\forall A \in \alpha :$   
 $(\mathcal{F}_c - \{\alpha \rightarrow \beta\} \cup \{(\alpha - A) \rightarrow \beta\}) \not\equiv \mathcal{F}_c$
  - $\forall B \in \beta :$   
 $(\mathcal{F}_c - \{\alpha \rightarrow \beta\} \cup \{\alpha \rightarrow (\beta - B)\}) \not\equiv \mathcal{F}_c$
- We do this in two steps: first looking at the left-hand side (LHS) and then at the right-hand side (RHS)

# Minimizing LHS

- For the LHS of every FD  $\alpha \rightarrow \beta \in \mathcal{F}$  check
  - for all  $A \in \alpha$  if  $A$  is unnecessary, that means:  
 $\beta \subseteq AC(\mathcal{F}, \alpha - A)$
  - If yes, replace  $\alpha \rightarrow \beta$  by  $(\alpha - A) \rightarrow \beta$ .

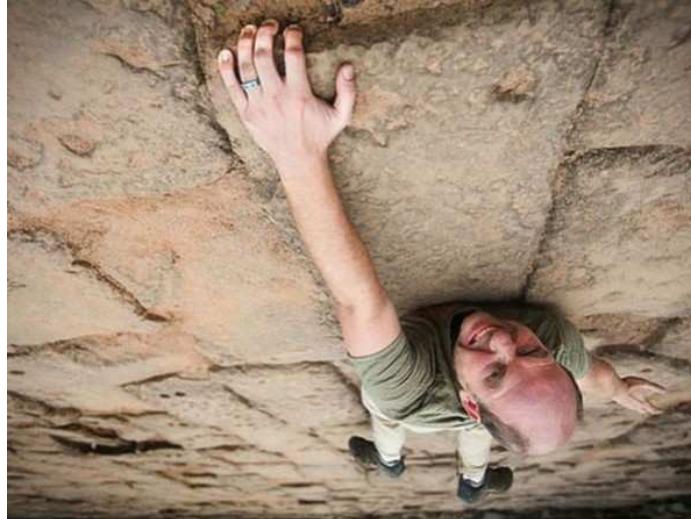


# Minimizing RHS

- For the RHS of every (remaining) FD  $\alpha \rightarrow \beta \in \mathcal{F}$  check
  - for all  $B \in \beta$  if  $B$  is unnecessary, that means:  
$$B \in AC(\mathcal{F} - \{\alpha \rightarrow \beta\} \cup \{\alpha \rightarrow (\beta - B)\}, \alpha)$$
  - If yes, replace  $\alpha \rightarrow \beta$  by  $\alpha \rightarrow (\beta - B)$



# Only Two More Steps...



- If present, remove all FDs of the form  $\alpha \rightarrow \emptyset$
- Merge all FDs that have the same LHS (union rule):  
 $\alpha \rightarrow \beta_1, \dots, \alpha \rightarrow \beta_n$  become  $\alpha \rightarrow \beta_1 \cup \dots \cup \beta_n$
- That's it, you now have a minimal basis!
- Depending on the order you process the FDs, you may end up with a slightly different result
  - However, does not matter, no solution will contain any redundancy

# 3NF Synthesis Algorithm

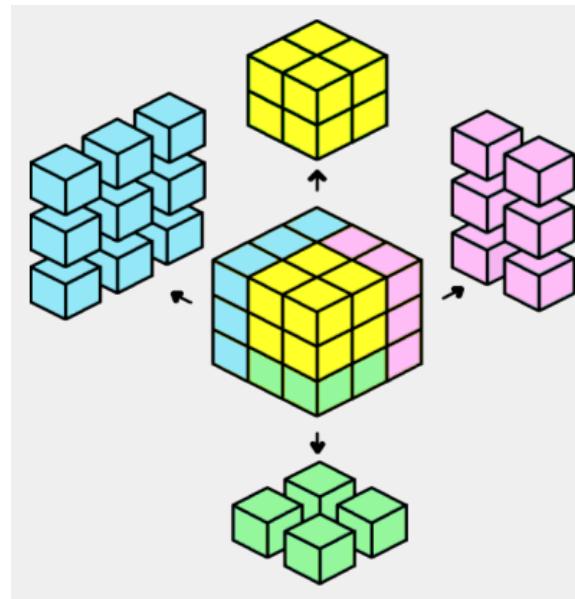
- Now we are ready to apply the algorithm to decompose a schema into 3NF
  - First step: for every FD  $\alpha \rightarrow \beta$  in  $\mathcal{F}_c$  create a subschema  $\mathcal{R}_\alpha = \alpha \cup \beta$ 
    - with  $\mathcal{F}_\alpha := \{\alpha' \rightarrow \beta' \in \mathcal{F}_c \mid \alpha' \cup \beta' \subseteq \mathcal{R}_\alpha\}$
  - Second step: add a schema  $\mathcal{R}_\kappa$  that contains a candidate key
  - Third step: eliminate redundant schemas, i.e., if you find  $\mathcal{R}_i \subseteq \mathcal{R}_j$ , drop  $\mathcal{R}_i$

# Example

- Let's apply this algorithm to finance with  
 $\mathcal{F}_{\mathcal{R}_c} = \{B \rightarrow O, S \rightarrow D, I \rightarrow B, IS \rightarrow N\}$ 
  - $\mathcal{R}_B(B, O)$
  - $\mathcal{R}_S(S, D)$
  - $\mathcal{R}_I(I, B)$
  - $\mathcal{R}_{IS}(I, S, N)$  (contains key)
- What happened to the anomalies we discussed at the beginning of the chapter?



# Decomposition into BCNF and 4NF



- There are algorithms for decomposing schemas into BCNF and 4NF, too
- However, these algorithms cannot guarantee the preservation of dependencies
- Usually, designers stop at 3NF
  - Has another reason: the higher the normal form, the more biased a schema is towards update operations

# Decomposition into BCNF

- Start with  $Z = \{\mathcal{R}\}$
- If there is still an  $\mathcal{R}_i \in Z$  that is not in BCNF
  - Find an FD  $(\alpha \rightarrow \beta) \in \mathcal{F}^+$  with
    - $\alpha \cup \beta \subseteq \mathcal{R}_i$
    - $\alpha \cap \beta = \emptyset$
    - $\alpha \rightarrow \mathcal{R}_i \notin \mathcal{F}^+$
  - Decompose  $\mathcal{R}_i$  into  $\mathcal{R}_{i_1} := \alpha \cup \beta$  and  $\mathcal{R}_{i_2} := \mathcal{R}_i - \beta$
  - Remove  $\mathcal{R}_i$  from  $Z$  and add  $\mathcal{R}_{i_1}$  and  $\mathcal{R}_{i_2}$ :  
$$Z := (Z - \{\mathcal{R}_i\}) \cup \{\mathcal{R}_{i_1}\} \cup \{\mathcal{R}_{i_2}\}$$

# Decomposition into 4NF

- Very similar to BCNF, just replace FD with MVD
- Start with  $Z = \{\mathcal{R}\}$
- If there is still an  $\mathcal{R}_i \in Z$  that is not in 4NF
  - Find an MVD  $(\alpha \twoheadrightarrow \beta) \in \mathcal{F}^+$  with
    - $\alpha \cup \beta \subseteq \mathcal{R}_i$
    - $\alpha \cap \beta = \emptyset$
    - $\alpha \rightarrow \mathcal{R}_i \notin \mathcal{F}^+$
  - Decompose  $\mathcal{R}_i$  into  $\mathcal{R}_{i_1} := \alpha \cup \beta$  and  $\mathcal{R}_{i_2} := \mathcal{R}_i - \beta$
  - Remove  $\mathcal{R}_i$  from  $Z$  and add  $\mathcal{R}_{i_1}$  and  $\mathcal{R}_{i_2}$ :  
$$Z := (Z - \{\mathcal{R}_i\}) \cup \{\mathcal{R}_{i_1}\} \cup \{\mathcal{R}_{i_2}\}$$

# Summary

- Normal forms determine the quality of a relational schema
- If not good enough, a decomposition algorithm can be used to improve quality
  - All algorithms guarantee recoverability
  - Preservation of dependencies can be guaranteed up to 3NF
- High quality for relational schemas means

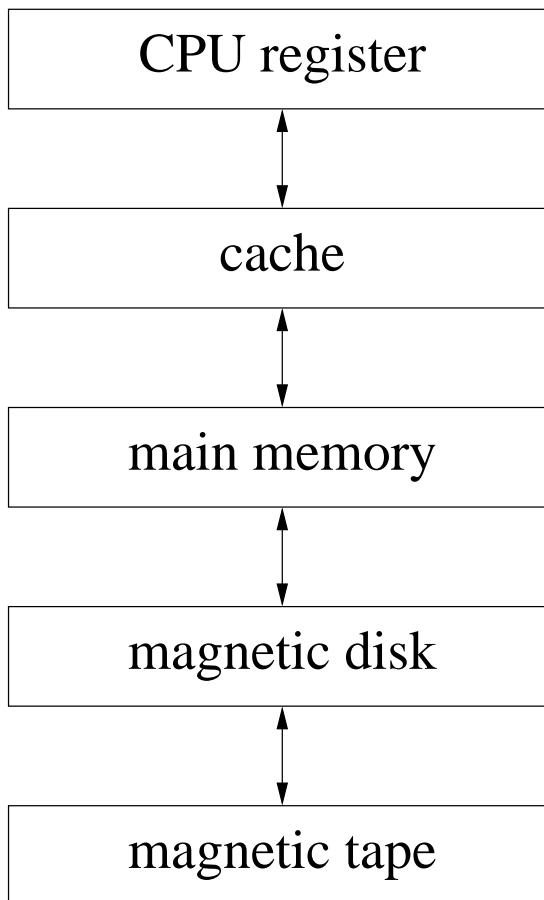


# Chapter 7

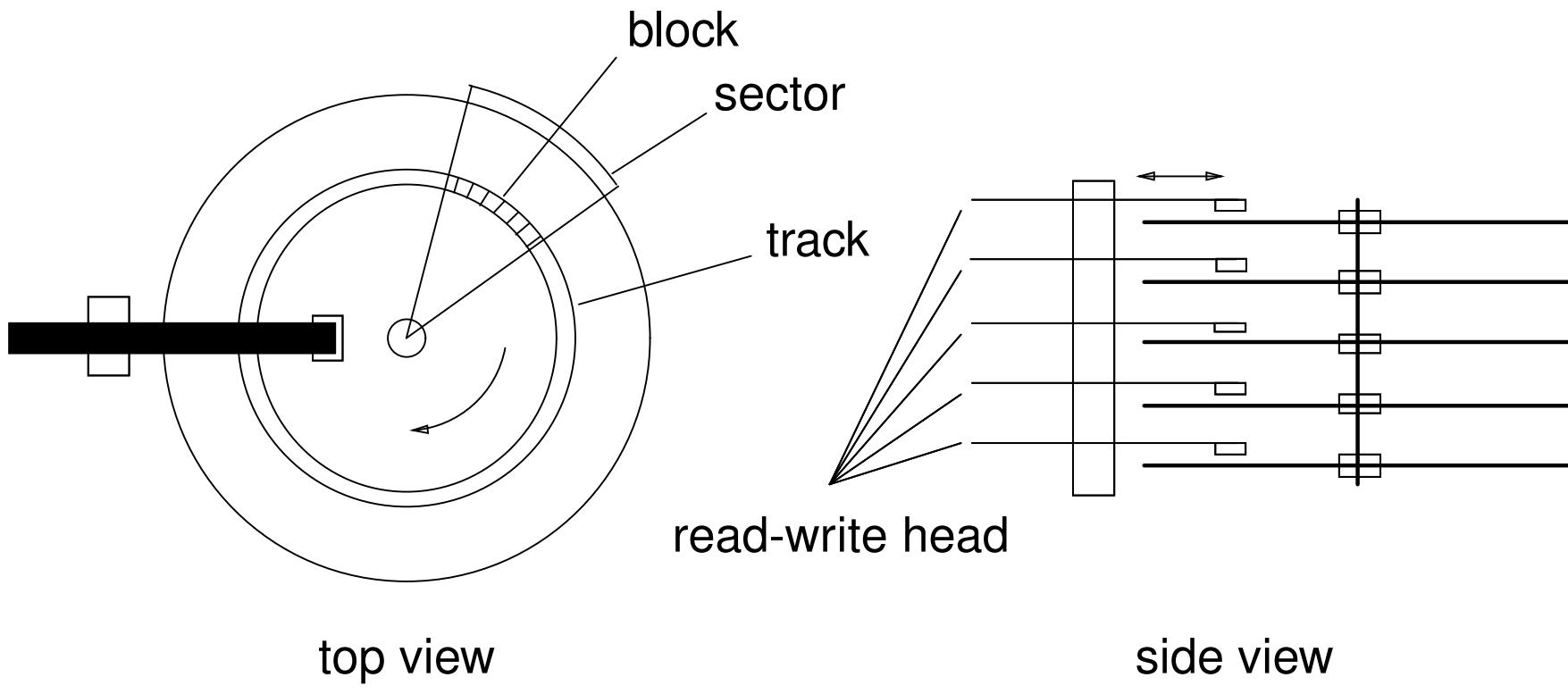
## Data Storage

# Memory Hierarchy

- Memory is organized hierarchically
- The higher in the hierarchy, the faster, smaller, and more expensive
- Differences in orders of magnitude
- We focus on main memory and magnetic disks



# Magnetic Disk



- This is a simplified, schematic view

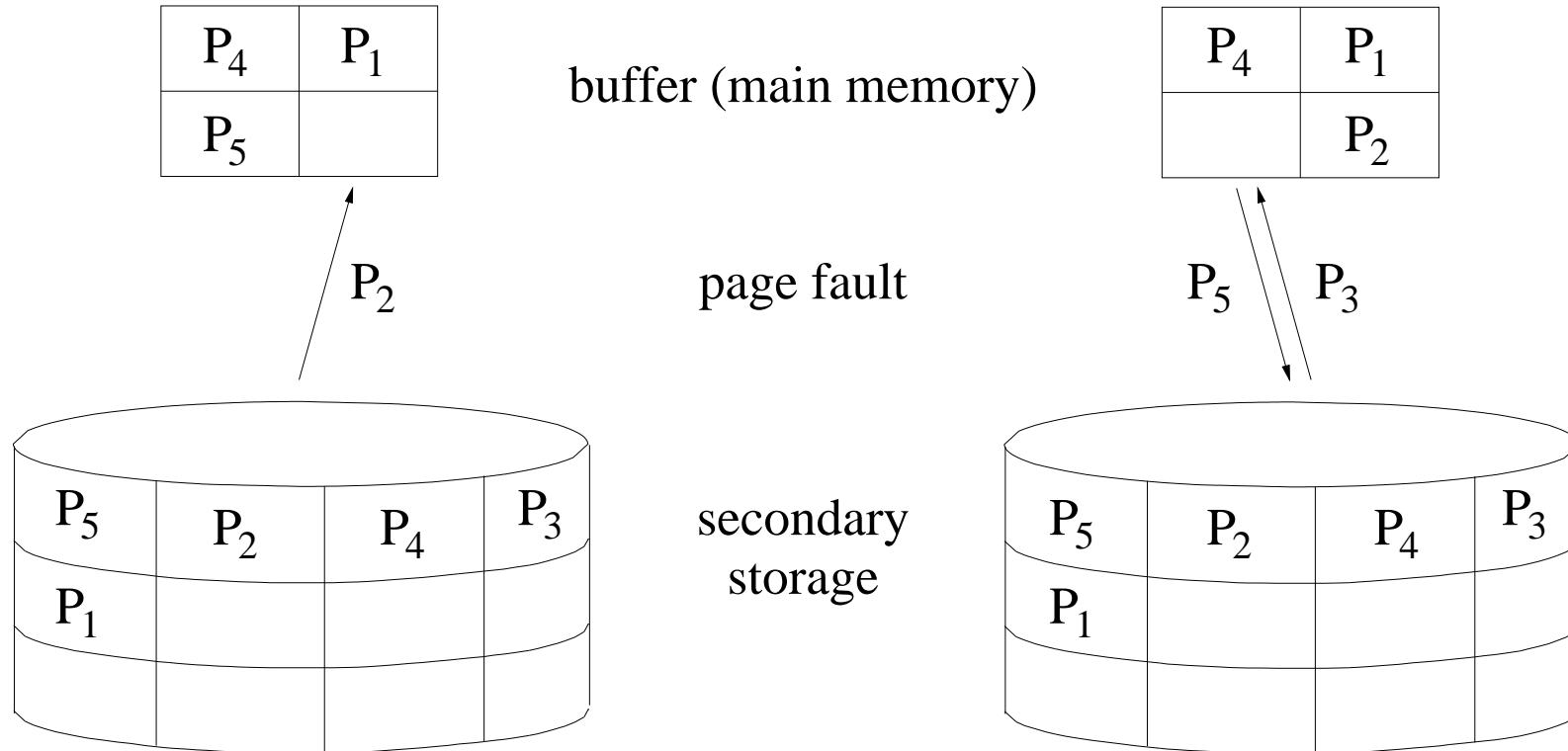
# Magnetic Disk (2)

- Reading a block from disk:
  - Position the read-write head over the correct track (seek time)
  - Wait until sector/block appears under the head (rotational latency)
  - Read the content of the block (data transfer rate)



# Buffer Management

- All data processing takes place in main memory
- Data not in main memory has to be fetched from disk



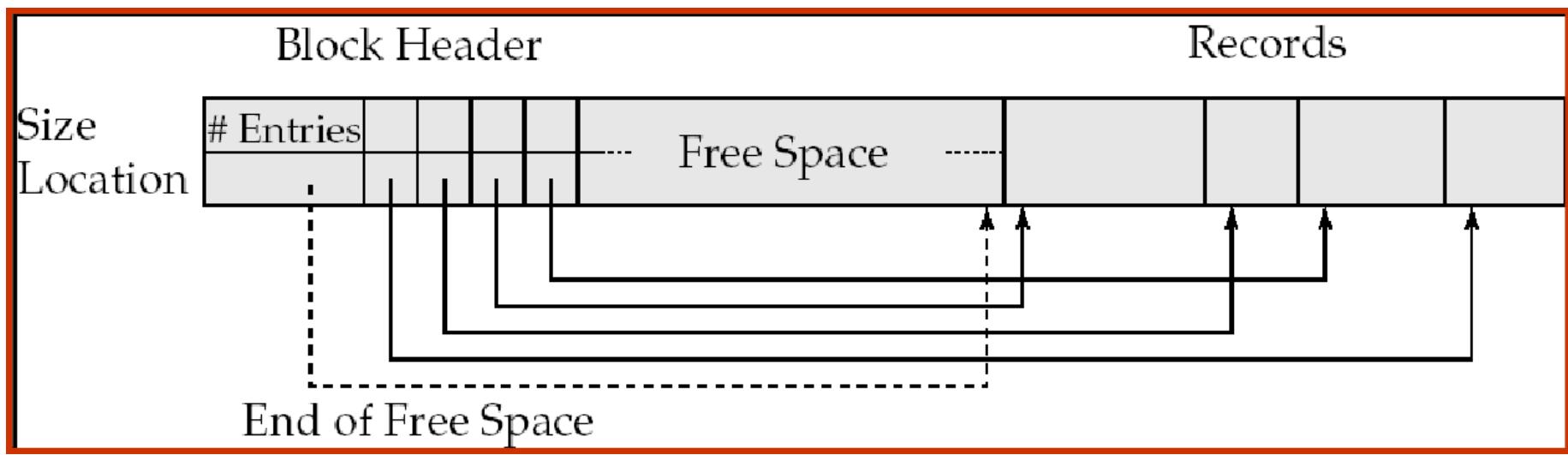
# Storing Relations

- Databases usually don't address the physical blocks of a disk directly
- They use *pages* or *logical blocks* that are mapped to physical blocks
- Tuples or records are then stored on these pages
- As records may contain variable-length fields, we need *slotted pages*

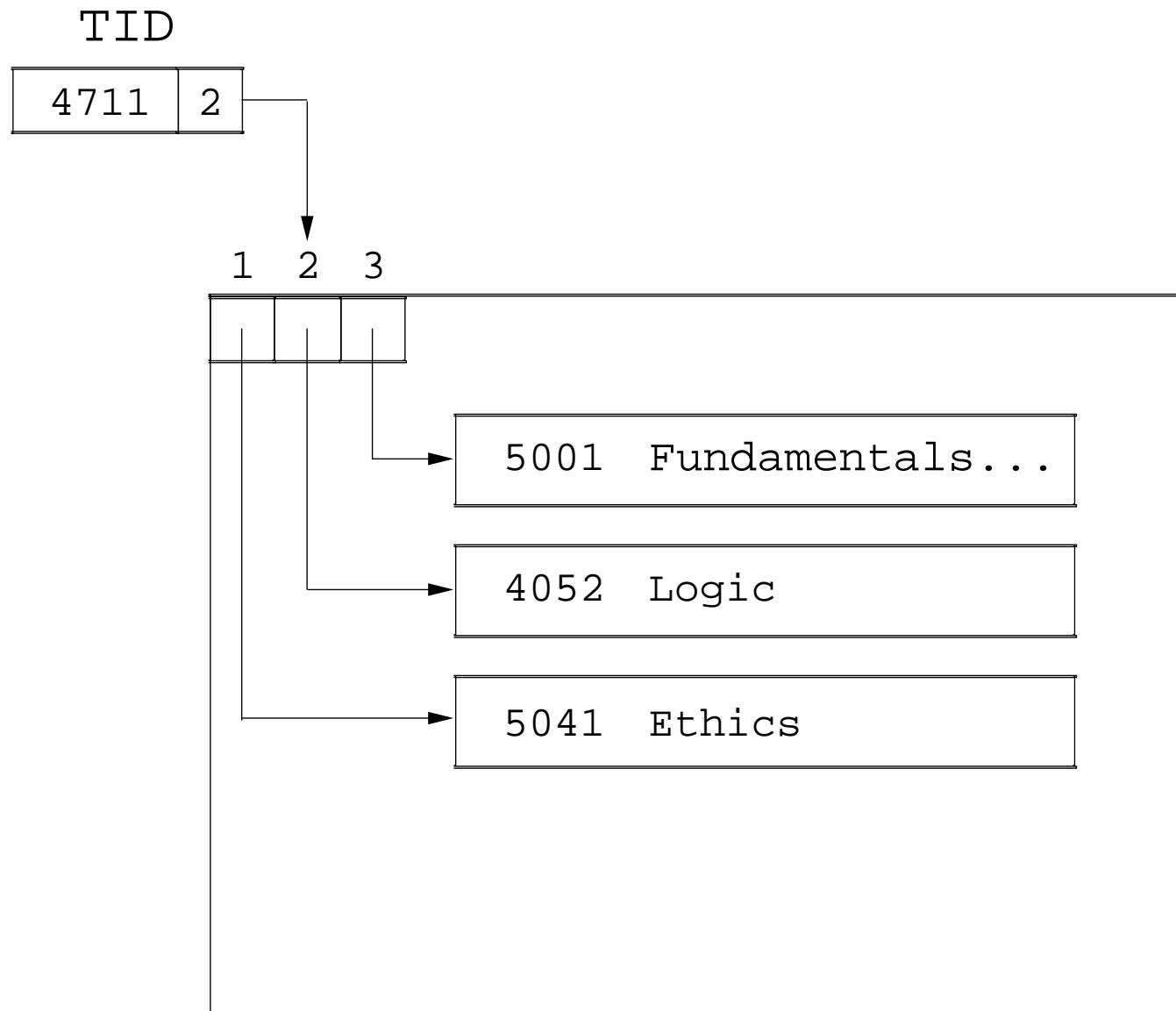


# Slotted Pages

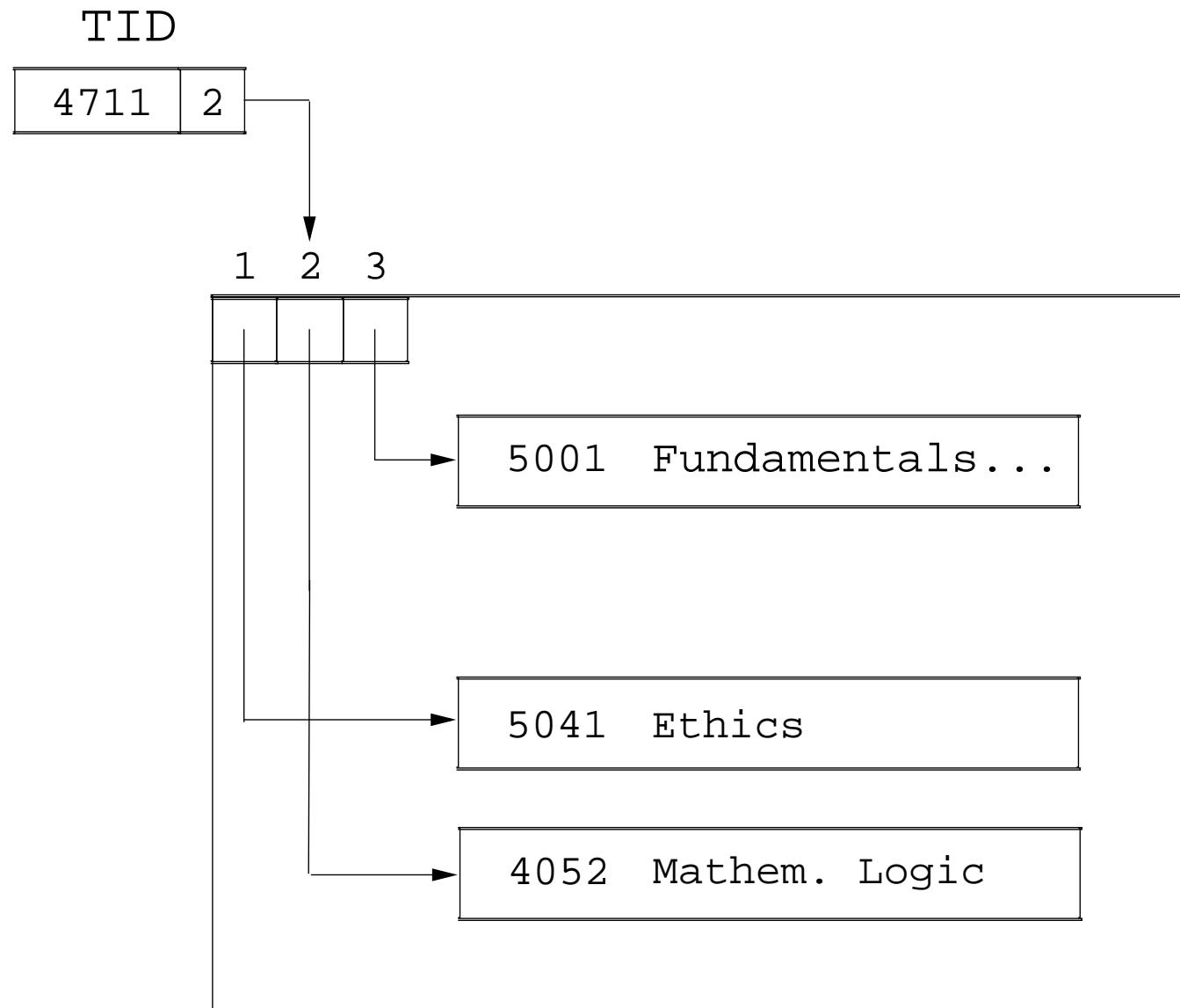
- Every page includes a *page header*, containing information about
  - number of record entries
  - location and size of each record
  - end of free space



# Tuple Identifiers (TIDs)



# Moving a Tuple Within a Page

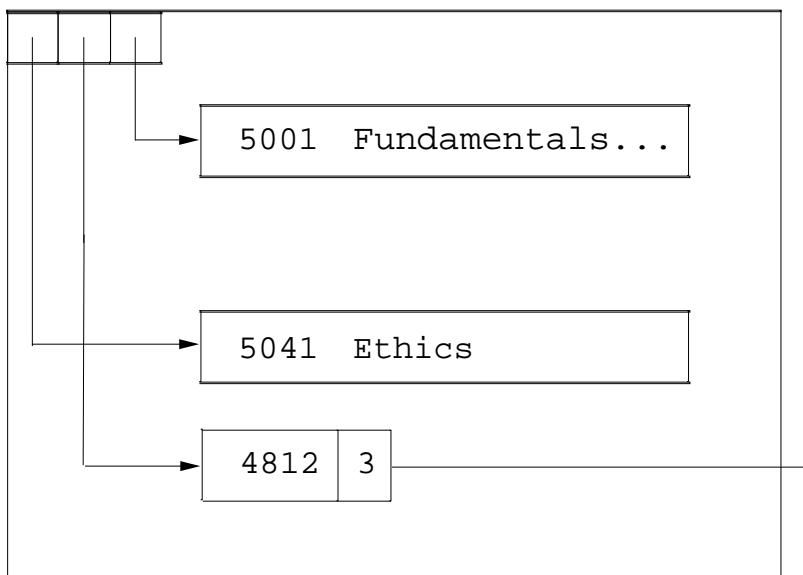


# Moving a Tuple to Another Page

TID

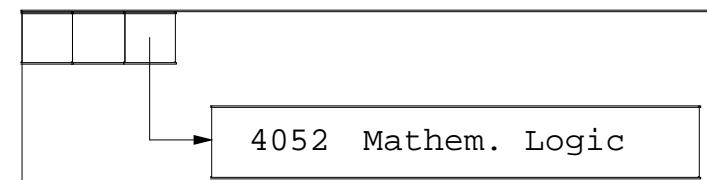
4711	2
------	---

1 2 3



Page 4711

1 2 3

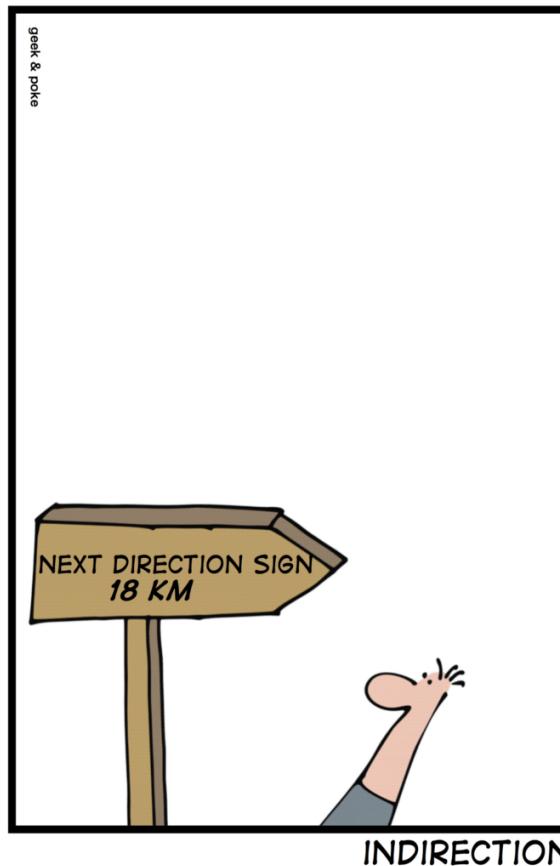


Page 4812

# Moving a Tuple to Another Page (2)

- Computer science proverb:  
“There is nothing which cannot be solved by another level of indirection.”

SIMPLY EXPLAINED



# Data Transfer

- Transferring all the tuples into main memory is the straightforward way to process queries
- Unfortunately, it is also the most expensive approach



- Looking at this more closely, we find out that
  - often only a tiny fraction of all tuples is needed to answer a query
  - queries often access similar parts of the data
  - hard disks allow random access

# Index Structures

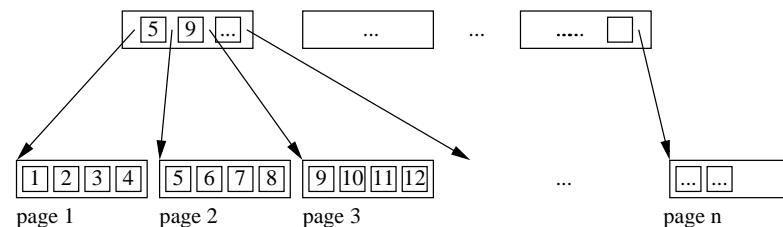
- Index structures exploit this to reduce the amount of transferred data
- Only the part that is needed to answer the query is fetched



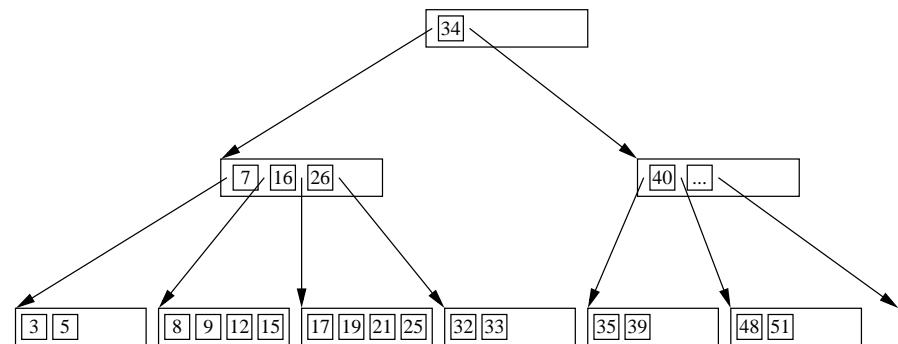
- The two most important approaches for indexing:
  - hierarchical (trees)
  - partitioning (hashing)

# Hierarchical Indexes

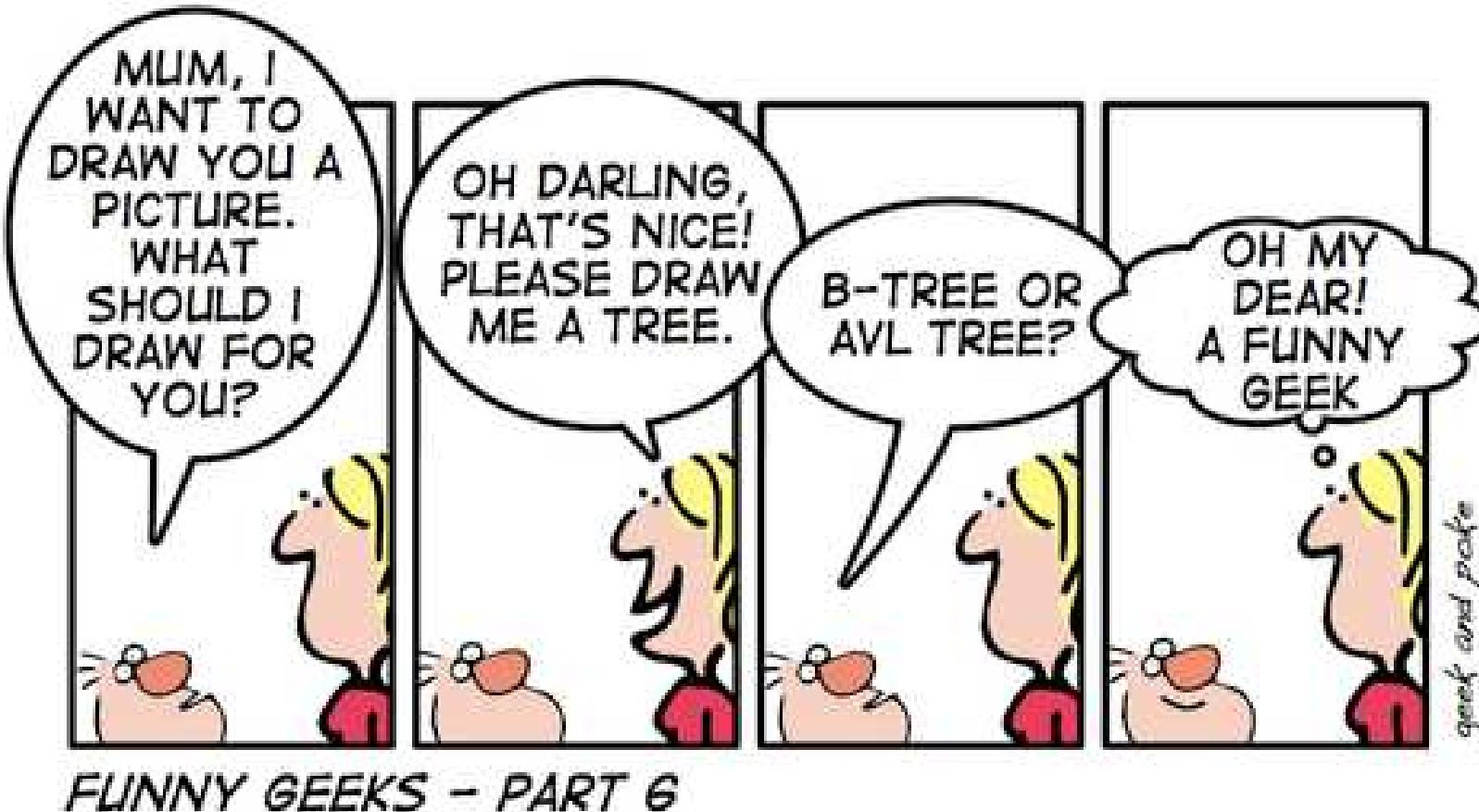
- We look at two hierarchical index structures:
  - ISAM (Index-Sequential Access Method)



- B-trees

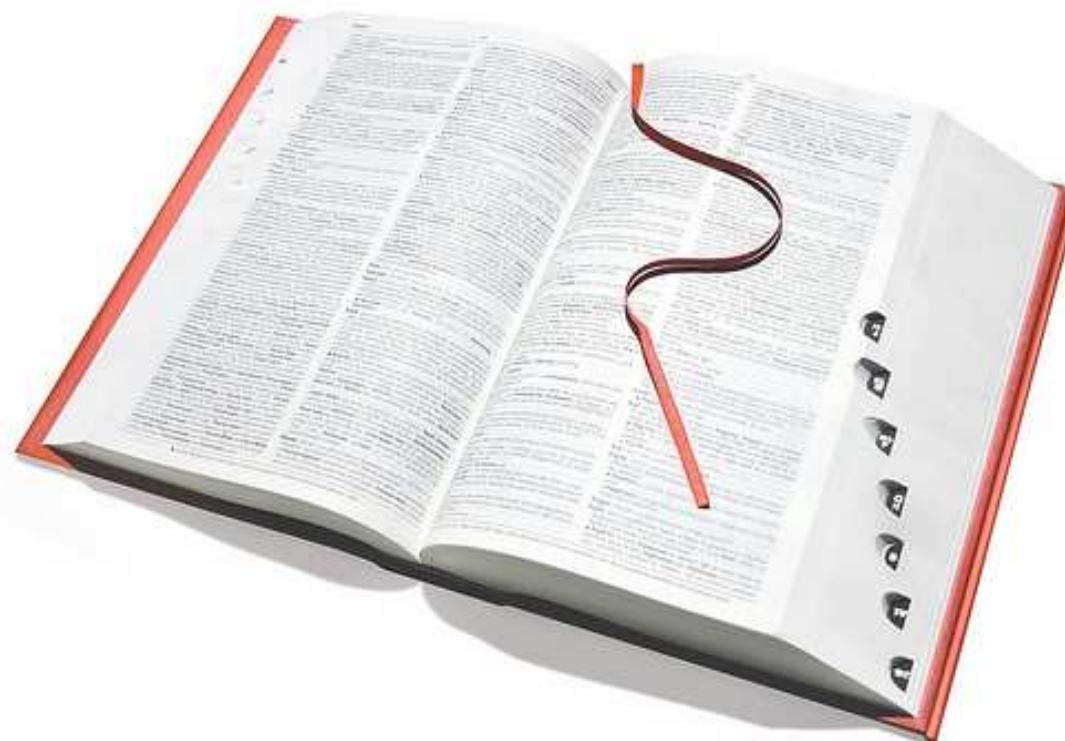


# Hierarchical Indexes (2)



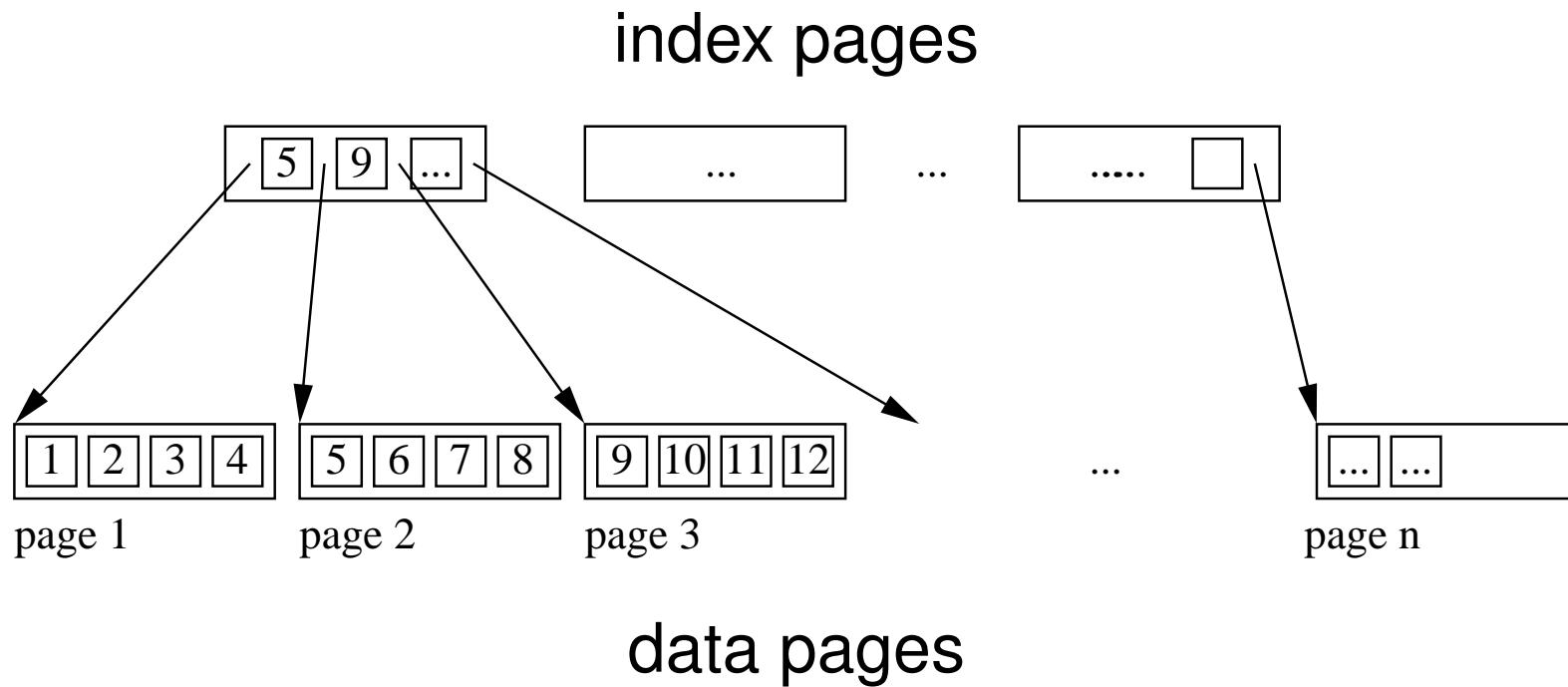
# ISAM

- ISAM is actually an early ancestor of a B-tree
- Main idea is:
  - Sort tuples according to indexed attribute
  - Put an index on top of this
- Similar to a thumb index in a book:



# Example

- We are looking for the student with student number 13542
- Assume that the tuples in the relation `student` are ordered by student number

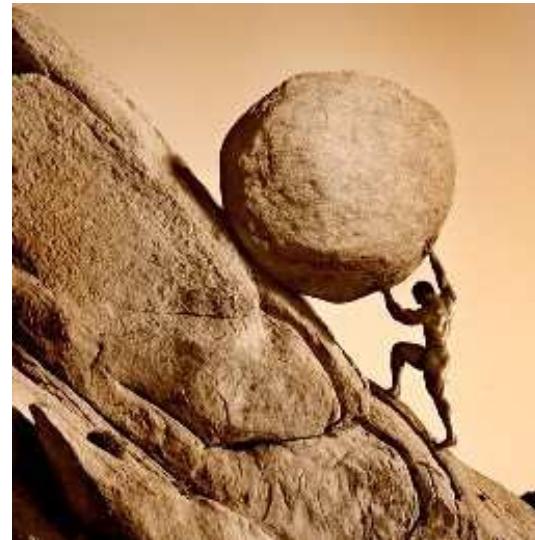


# Example (2)

- When evaluating a query, we scan through the index pages until we find the place where 13542 fits
- From there we go to the data page pointed to
- We save I/O: the number of index pages is significantly lower than the number of data pages
- We can also answer range queries more quickly
  - Find the starting point of the range in the index, then go to the data pages and scan sequentially in the data pages

# Issues with ISAM

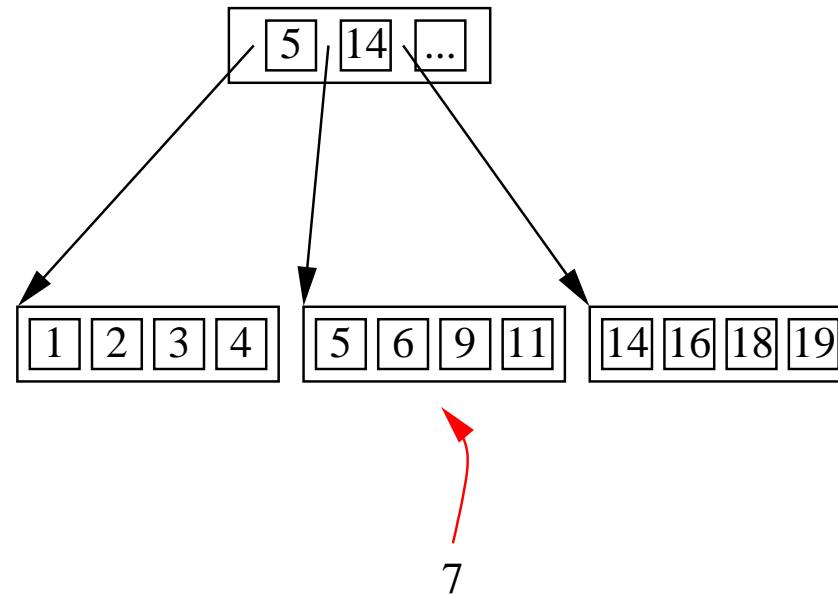
- Although searching an ISAM is straightforward and quick, maintaining the index can be expensive



- The trouble starts when pages become full and we have to split them

# Inserting a New Item

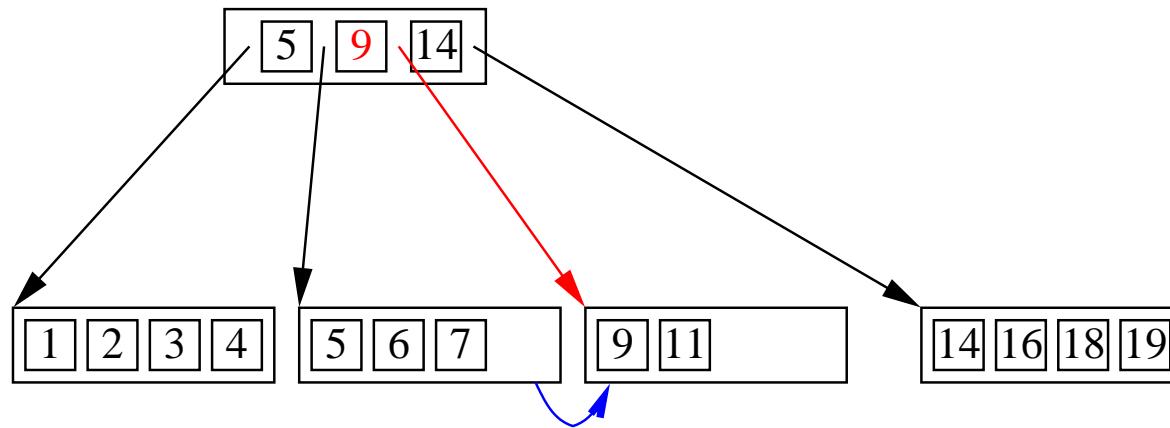
- We want to insert an item into a data page that is full:



- We have different options of handling this

# Inserting a New Item (2)

- We could insert a new entry into the index, adding the value 9 and a new pointer (red option)
- However, that would mean moving index entries (possibly destroying contiguous storage)



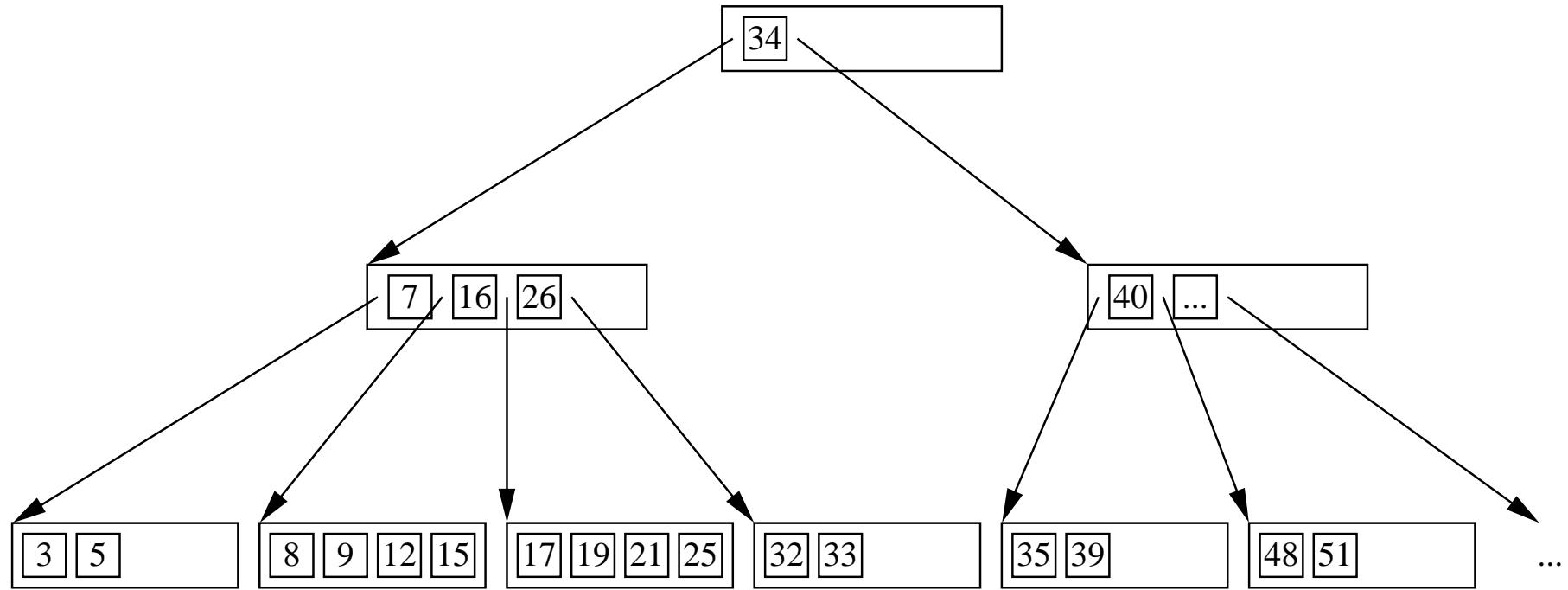
- Or we could treat the new page as an overflow page (blue option)
- We don't have to change the index, but search takes longer with overflow pages

# Further Issues

- Although the number of index pages is significantly smaller than the number of data pages...
  - ...the traversal can still take some time
- Idea: why not have index pages for index pages?
- That is basically the principle of a B-tree!
  - Additionally, we can also store data in index pages
  - A B-tree can also be seen as a generalization of a binary tree

# B-Tree Example

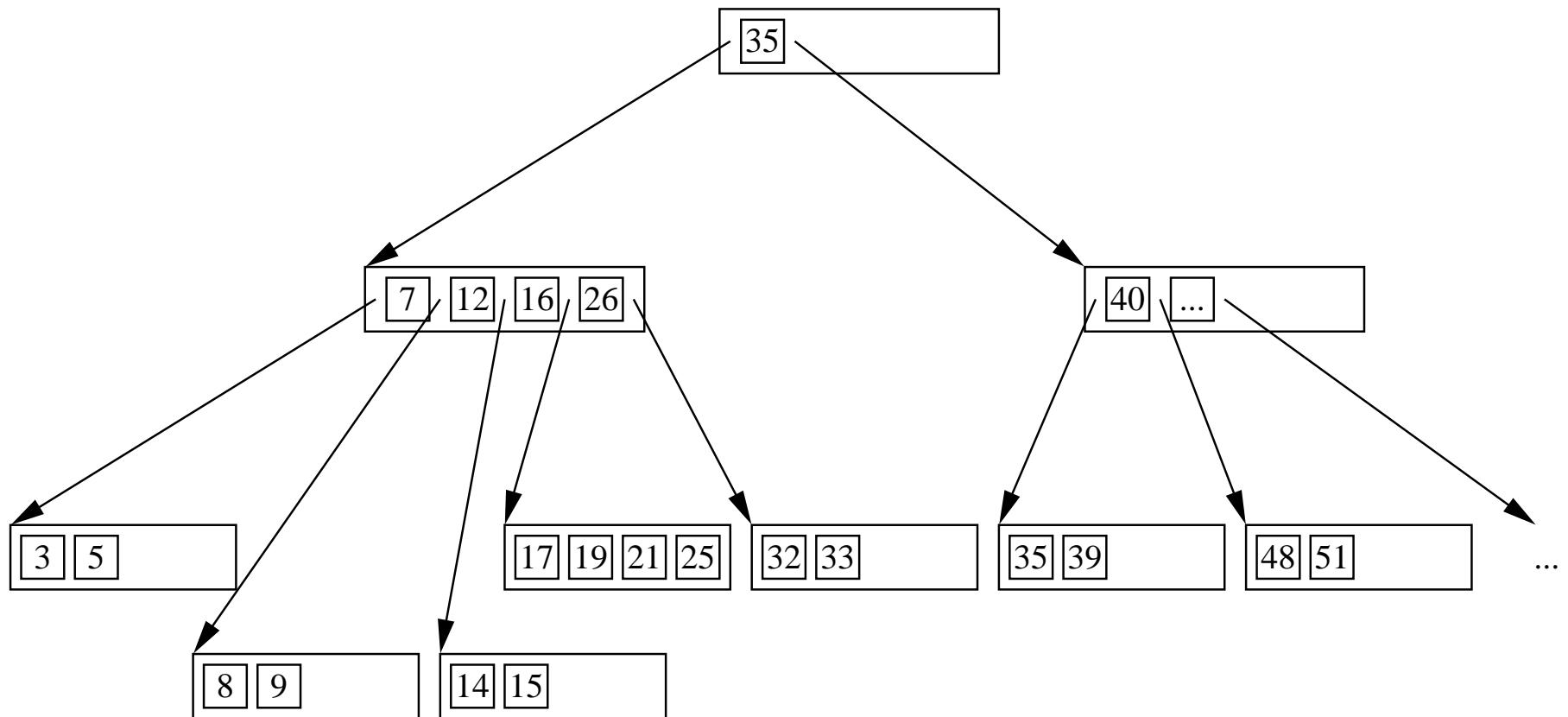
- Let's have a look at an example:



- Let's insert the data item 14 into the tree

# Inserting a New Data Item

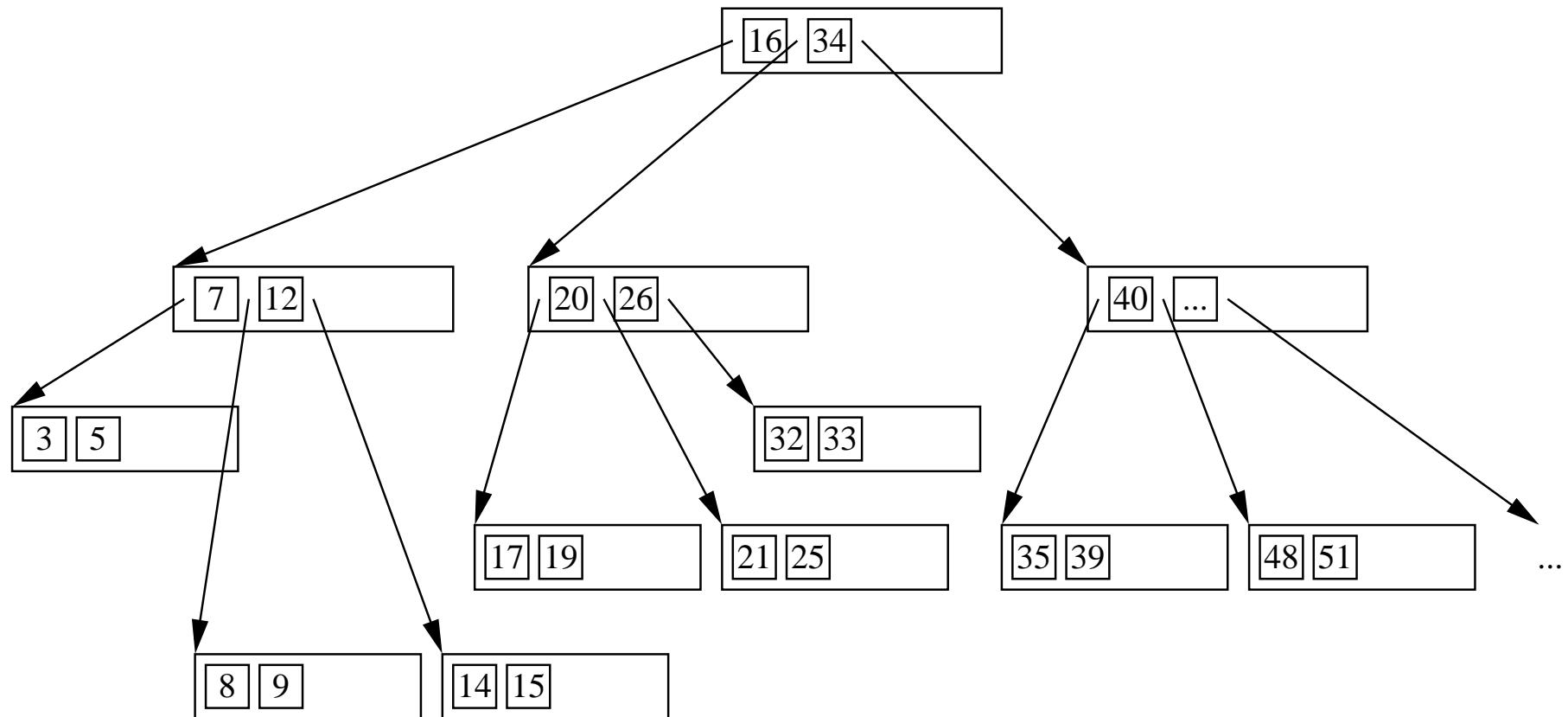
- This leads to a page overflow: median value on the page is pushed upwards and page is split:



- Let's insert another data item: 20

# Inserting a New Data Item (2)

- This time, we have a page overflow on an index page
  - Treated in the same way: push median up, split page



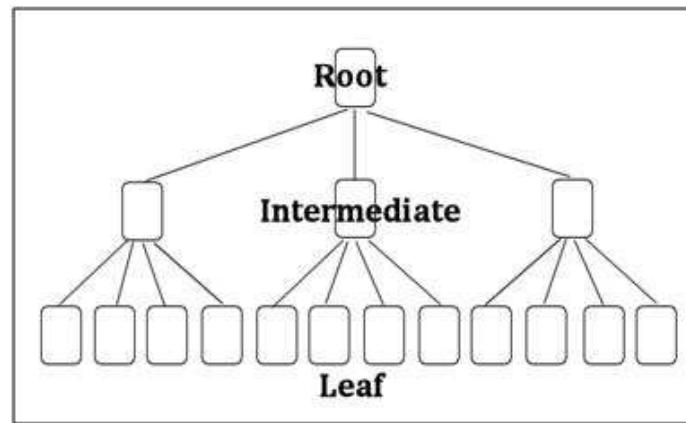
# Properties of a B-tree

- What does the “B” mean?
- No-one is quite sure...
- It was developed by Bayer and McCreight in 1971 while working for Boeing Research



# Properties of a B-tree (2)

- Every path from the root to a leaf has the same length
  - So it's a balanced tree



- Every node (except the root) has at least  $i$  and at most  $2i$  entries
  - In the previous examples,  $i = 2$
- The entries in every node are sorted
- Every node with  $n$  entries (except leaves) has  $n + 1$  children

# Properties of a B-tree (3)

- Assume we have an inner node looking like this:

$p_0$	$k_1$	$p_1$	$k_2$	$\dots$	$k_n$	$p_n$
-------	-------	-------	-------	---------	-------	-------

where  $p_j$  is a pointer,  $k_j$  a (key) value

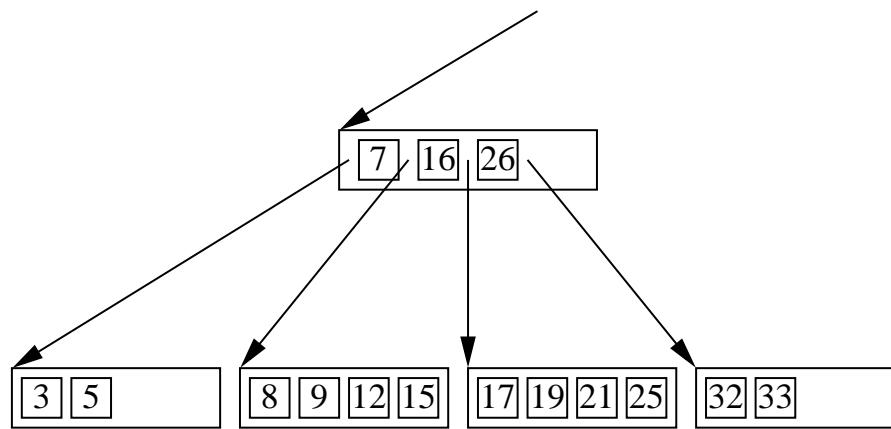
- Then the following holds:
  - The subtree pointed to by  $p_0$  contains values less than  $k_1$
  - $p_j$  points to a subtree with values between  $k_j$  and  $k_{j+1}$
  - The subtree referenced by  $p_n$  contains values greater than  $k_n$

# Insertion Algorithm

1. Find the correct leaf for inserting a value
2. If there is enough space, insert value
3. Otherwise
  - Split node along the median into left and right node
  - Insert all values less than median into left node, all values greater than median into right node
  - Insert median into parent node and adjust pointers there
4. If there is no space in parent node
  - If parent is the root node, create new root, insert median, adjust pointers
  - Otherwise, repeat 3. with parent node

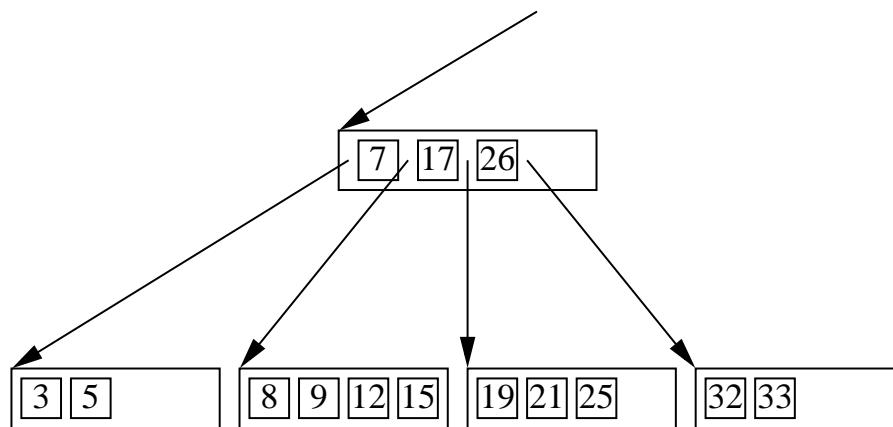
# Deletion Algorithm

- Deletion of a value in a leaf is simple: just delete the value
- An inner node needs to stay properly connected to its children
- For example, we want to delete the value 16 from an inner node (distinguishes values in leaf nodes):



# Deletion Algorithm (2)

- Solution: we delete it and move up the next bigger value from the right child node



- Alternatively, we can move up the previous smaller value (15) from the left child node

# Deletion Algorithm (3)

- After the deletion of a value, a node can have too few entries (fewer than  $i$ )
- In that case we can merge a node with one of its neighbors (pulling down a value from the parent node)
  - This can cause a parent node to have too few entries, i.e., we have to merge the parent node with a neighbor
  - We're not going into details here
    - As databases tend to grow (and not shrink), many databases don't even implement this merging

# B<sup>+</sup>-Trees



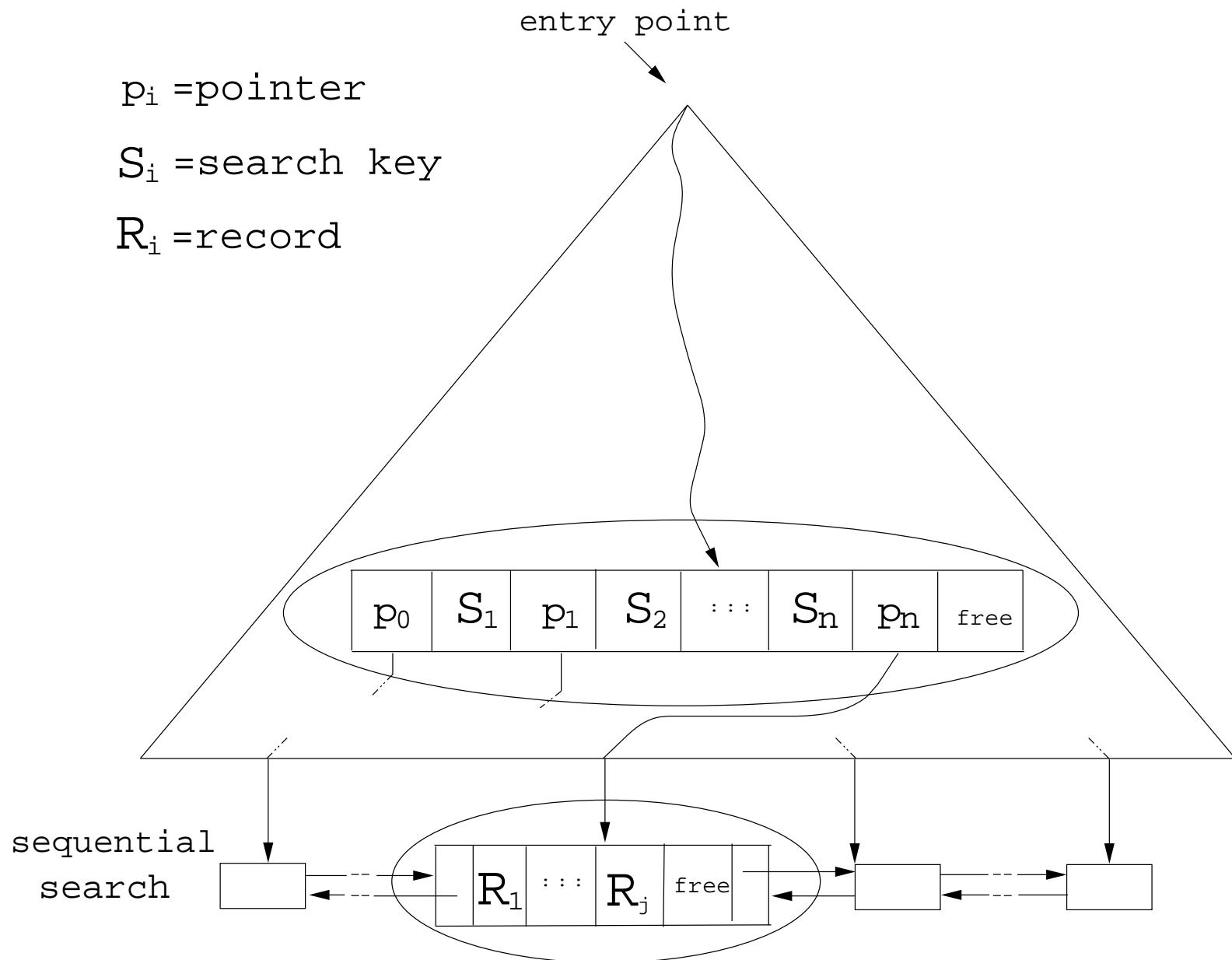
- B<sup>+</sup>-trees are an important variant of B-trees
- The performance of a B-tree depends heavily on the height of the tree
  - The deeper a tree, the more page lookups (on secondary storage) we need to reach a leaf
- So what can we do to “flatten” B-trees?

# B<sup>+</sup>-Trees (2)



- If we can increase the branching (number of pointers) in inner nodes, then the tree will become “flatter”
- Instead of storing data in inner nodes, we only store search keys (take up less space  $\Rightarrow$  more room for pointers)
- We also link all the leaf nodes, allowing a fast sequential search

# Schema of a B<sup>+</sup>-Tree



# Prefix-B<sup>+</sup>-Trees



- We can improve B<sup>+</sup>-trees even further
- What if we have long strings as search keys?
- We only need to find a search key that distinguishes left subtree from right subtree:
  - counterrevolutionaries  $\leq$  d < disestablishmentarianism
  - operating system  $\leq$  ? < operations research

# Partitioning

- On average looking up a data item in a B-tree costs  $\log_k(n)$  page accesses
  - $k$  = branching factor
  - $n$  = number of indexed data items
- Hashing, which partitions the data, needs on average two page accesses



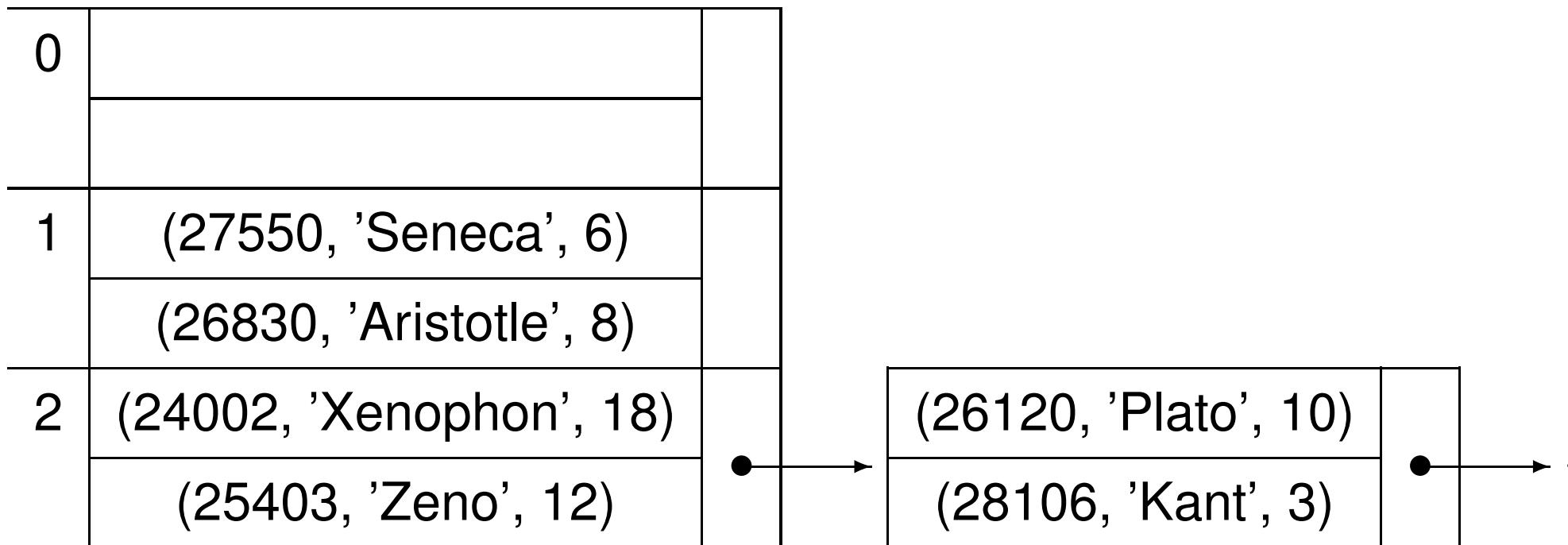
# Hashing

- The value to be indexed is run through a hash function
- For instance, take  $h(x) = x \bmod 3$
- Then insert value into a table according to hash value

0	
1	(27550, 'Seneca', 6)
2	(24002, 'Xenophon', 18)
	(25403, 'Zeno', 12)

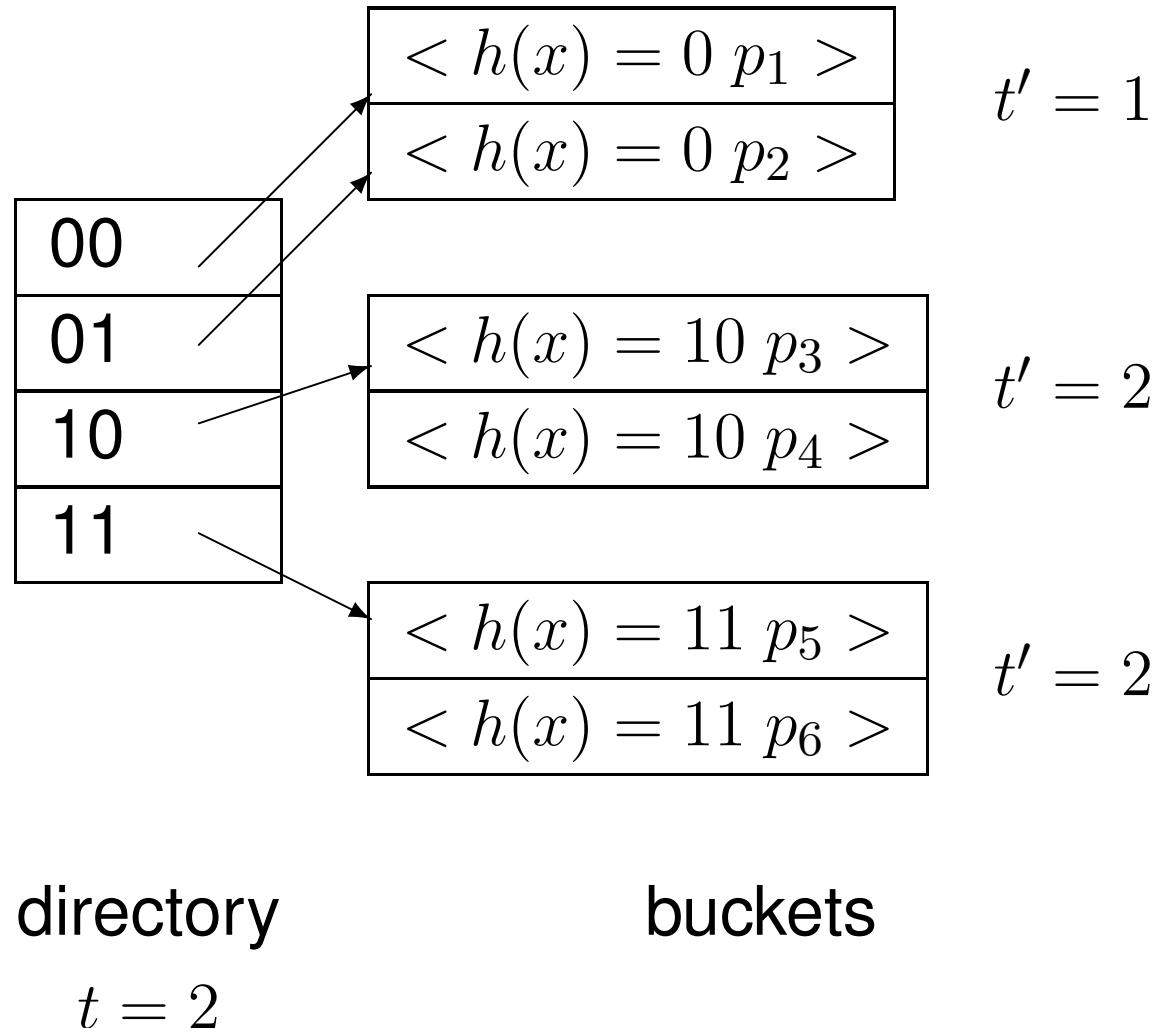
# Hashing(2)

- What happens if we run out of space?
- We have to introduce overflow pages



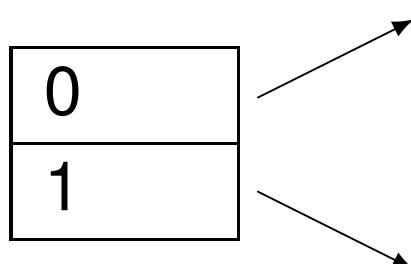
- Becomes inefficient if table becomes too small
- If we could make this adaptable...

# Extendable Hashing



# Concrete Example

$x$	$h(x)$	
	$d$	$p$
2125	1	01100100001
2126	0	11100100001
2127	1	11100100001



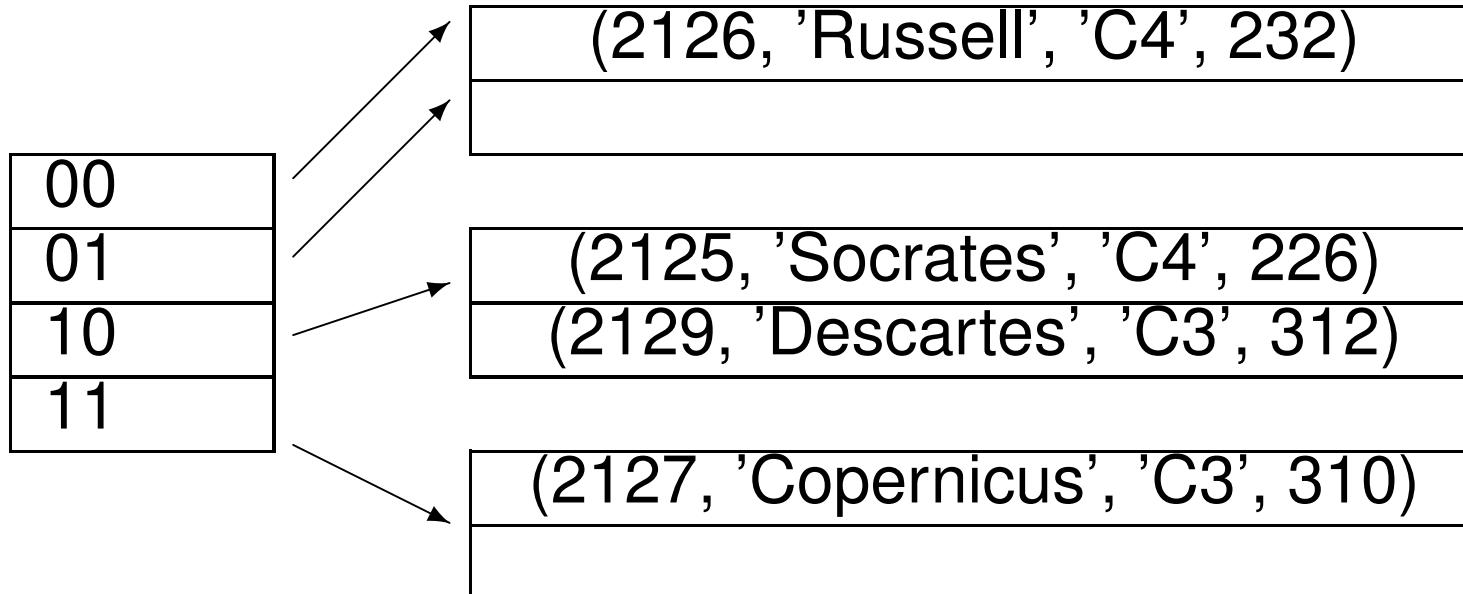
(2126, 'Russell', 'C4', 232)
(2125, 'Socrates', 'C4', 226)
(2127, 'Copernicus', 'C3', 310)

$t' = 1$

$t' = 1$

# Inserting (2129, Descartes, ...)

x	h(x)	
	d	p
2125	10	1100100001
2126	01	1100100001
2127	11	1100100001
2129	10	0010100001



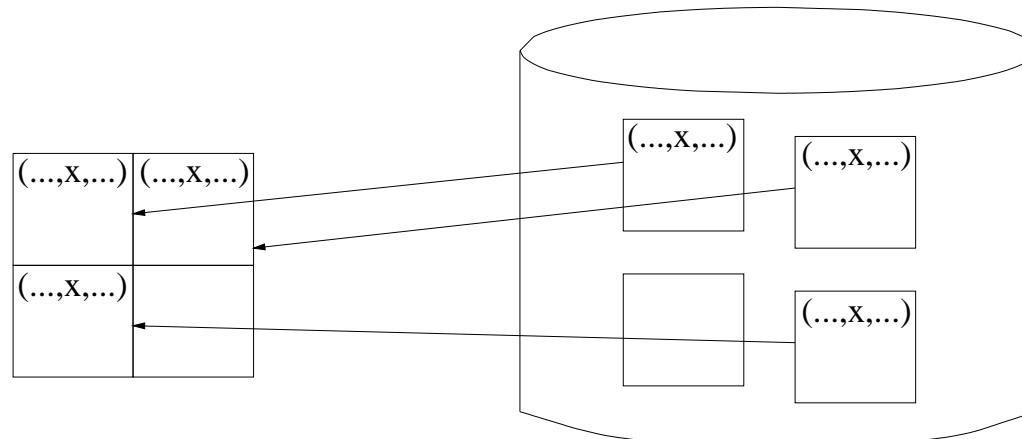
$$t' = 1$$

$$t' = 2$$

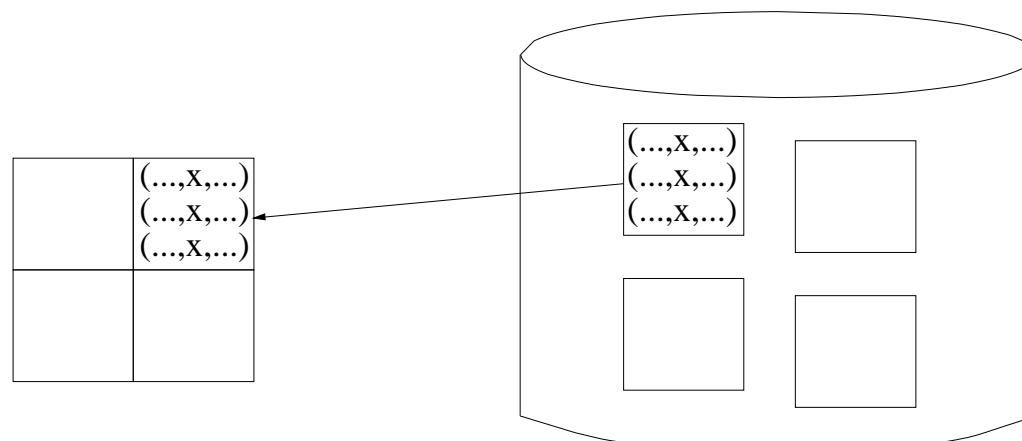
$$t' = 2$$

# Clustering

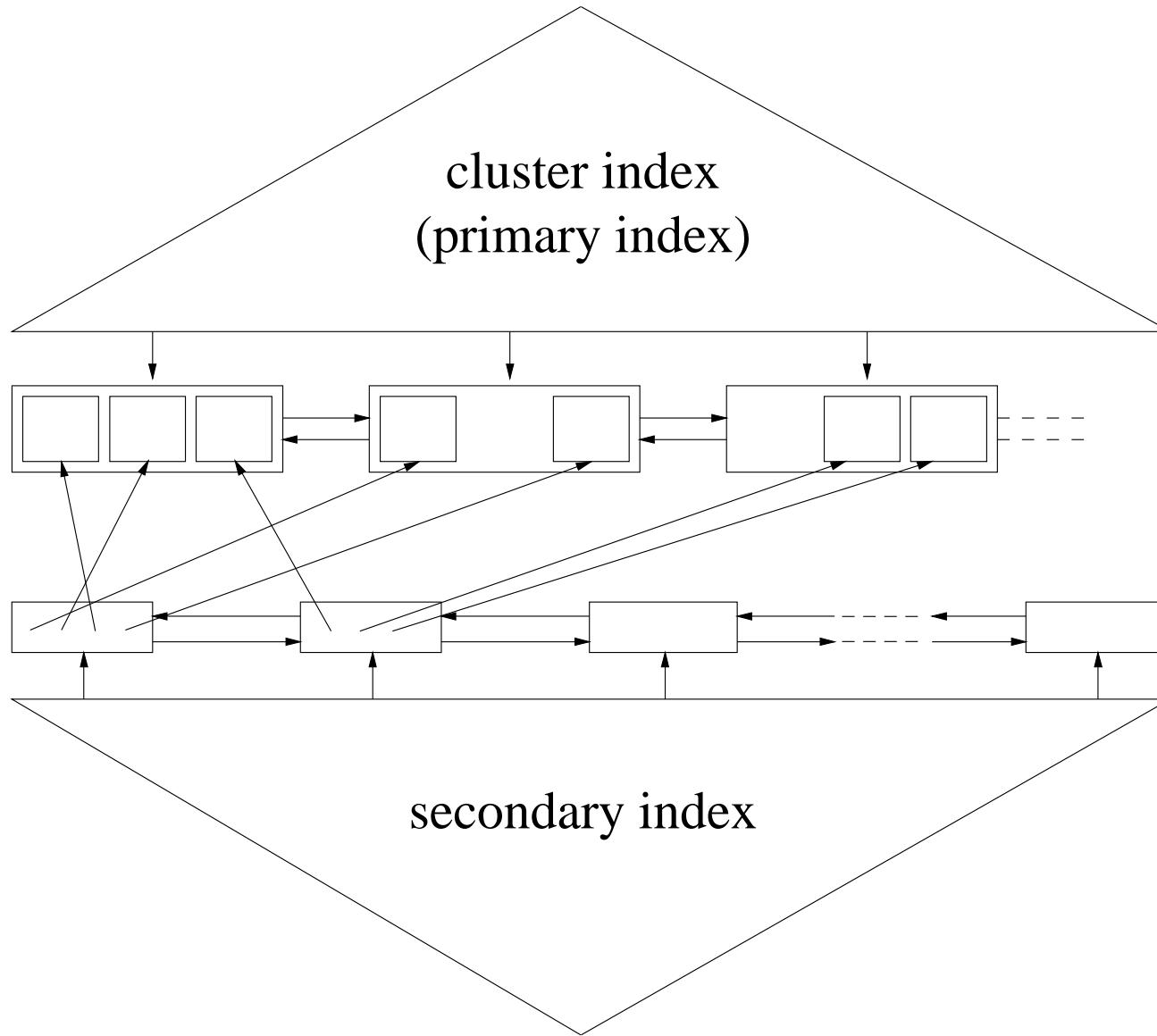
```
select *\nfrom R\nwhere A=x;
```



main memory ← page fault → secondary storage



# Indexes and Clustering



# Interleaved Clustering

Page  $P_i$

2125 ○ Socrates	○ C4 ○ 226 ●
5041 ○ Ethics	○ 4 ○ 2125 ●
5049 ○ Socratic method	○ 2 ○ 2125 ●
4052 ○ Logic	○ 4 ○ 2125 ●
2126 ○ Russell	○ C4 ○ 232 ●
5043 ○ Epistemology	○ 3 ○ 2126 ●
5052 ○ Philosophy of Science	○ 3 ○ 2126 ●
5216 ○ Bioethics	○ 2 ○ 2126 ●

Page  $P_{i+1}$

2133 ○ Pythagoras	○ C3 ○ 52 ●
5259 ○ Geometry	○ 2 ○ 2133 ●
2134 ○ Cicero	○ C3 ○ 309 ●
5022 ○ Rhetoric	○ 2 ○ 2134 ●

⋮

# Summary

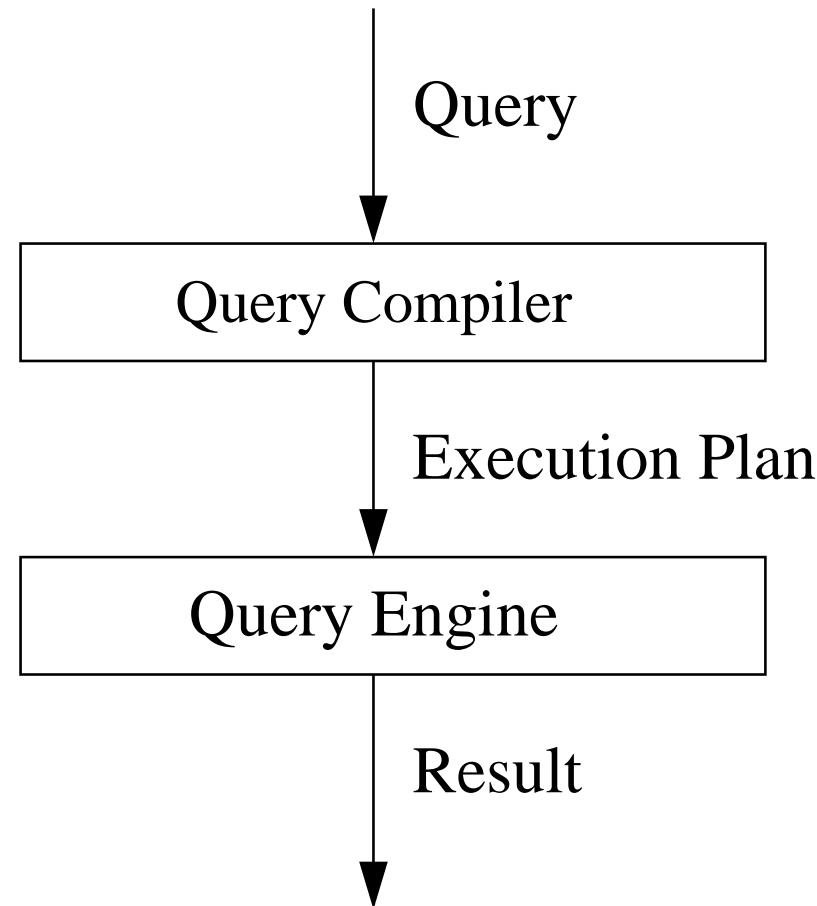
- Using tuple identifiers (TIDs) and slotted pages is a clever way of storing tuples on disk
- Indexes accelerate access to data items
  - But make updates more expensive
- B<sup>+</sup>-trees are the standard index structures used in relational systems
  - Suited for point and range queries



# Chapter 8

## Query Processing

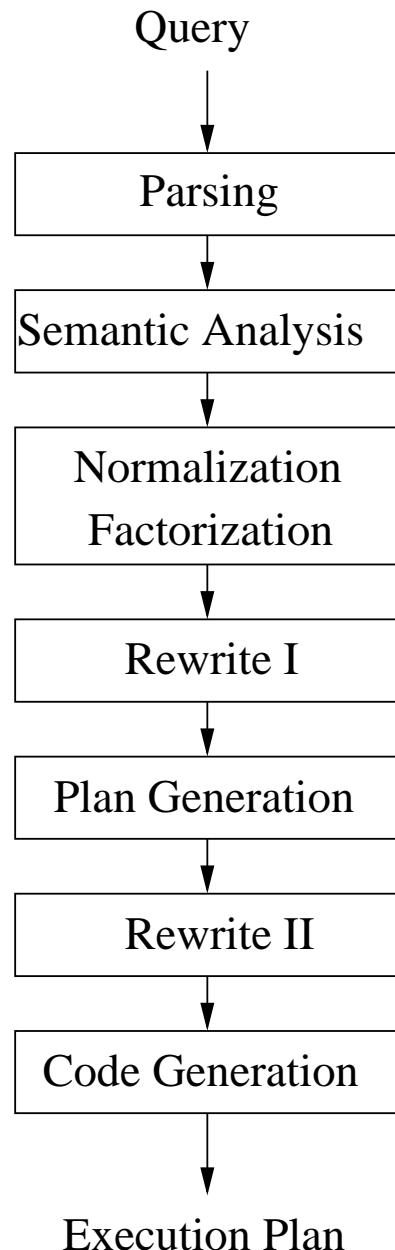
# Overview



# Compilation

- SQL is declarative
  - Query has to be translated into a procedural program that can be run on the query engine
- DBMSs translate SQL into another format
  - A widely-used approach is the translation into relational algebra

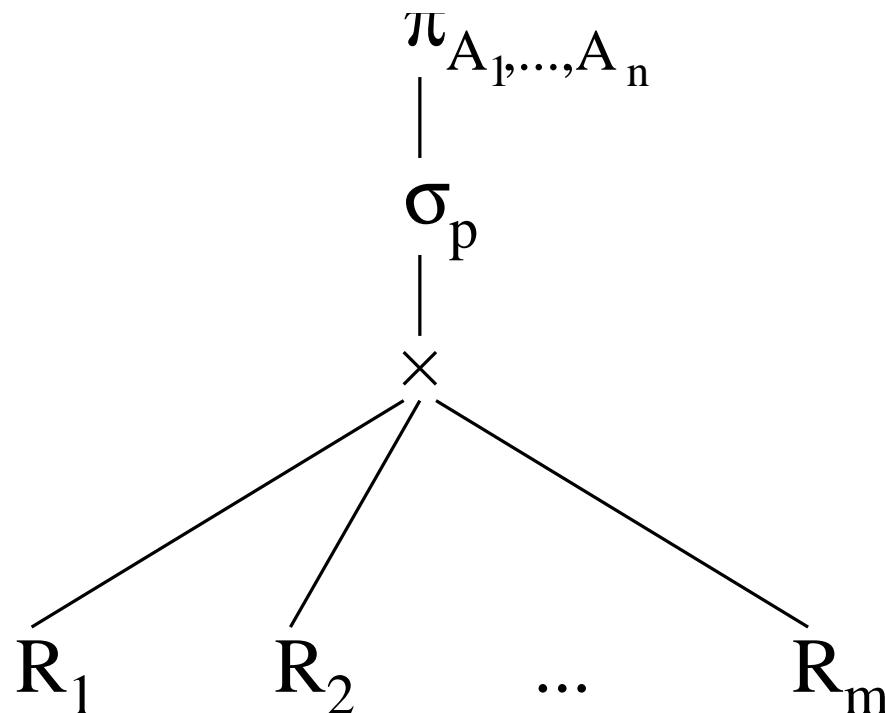
# Compilation (2)



# Canonical Translation

- There is a standard way of translating SQL into relational algebra
- Expressions in relational algebra are often represented graphically

**select**     $A_1, \dots, A_n$   
**from**      $R_1, \dots, R_m$   
**where**     $p$



# Optimization

- Canonical plan is not very efficient (e.g. it contains Cartesian products)
- Most DBMSs have query optimizers rewriting the plan to make it more efficient
- Query rewriting and optimization is a complex problem

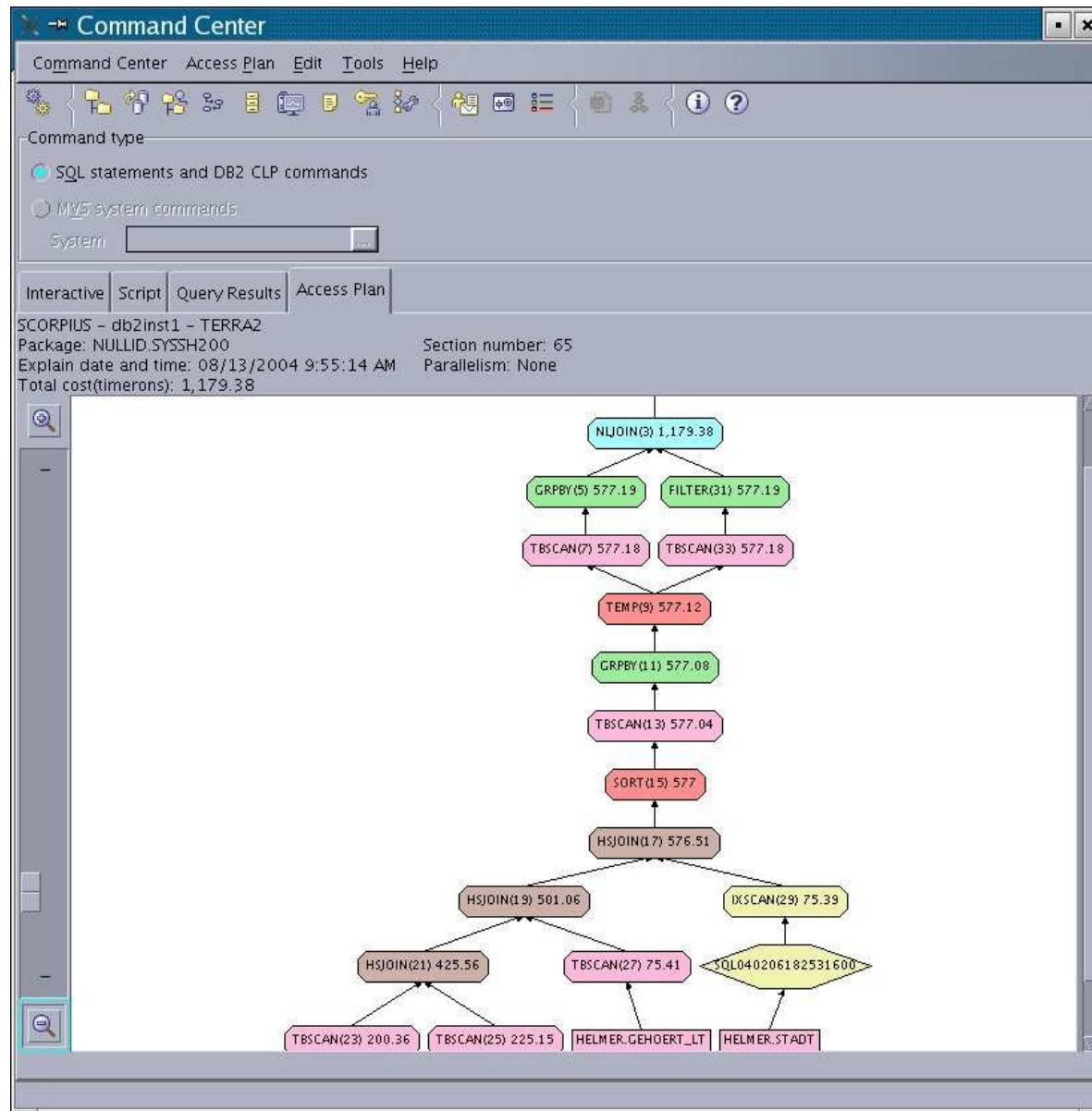


# Optimization (2)

- Why bother about query optimization as an ordinary user?
- Sometimes the query optimizer produces bad plans
- Nevertheless, most DBMSs allow a user to take a look at the generated execution plans
- Plans can be analyzed and query can be changed to make it more efficient

**SUBOPTIMAL**

# Visualizing Plans



# Query Optimization 101

- A DBMS can estimate the execution costs of a plan with the help of
  - cost models
  - statistics
- Investigating all possible plans is usually too expensive
- An optimizer uses heuristics to optimize queries
- Optimization can take place on different levels:
  - Logical level
  - Physical level

# Logical Level

- Starting point is the canonical relational algebra expression
- What can we do?
  - Transformation of algebraic expression into equivalent ones (ideally into faster ones)
  - A rule of thumb is to minimize the input/output of the individual operators



# Logical Level (2)

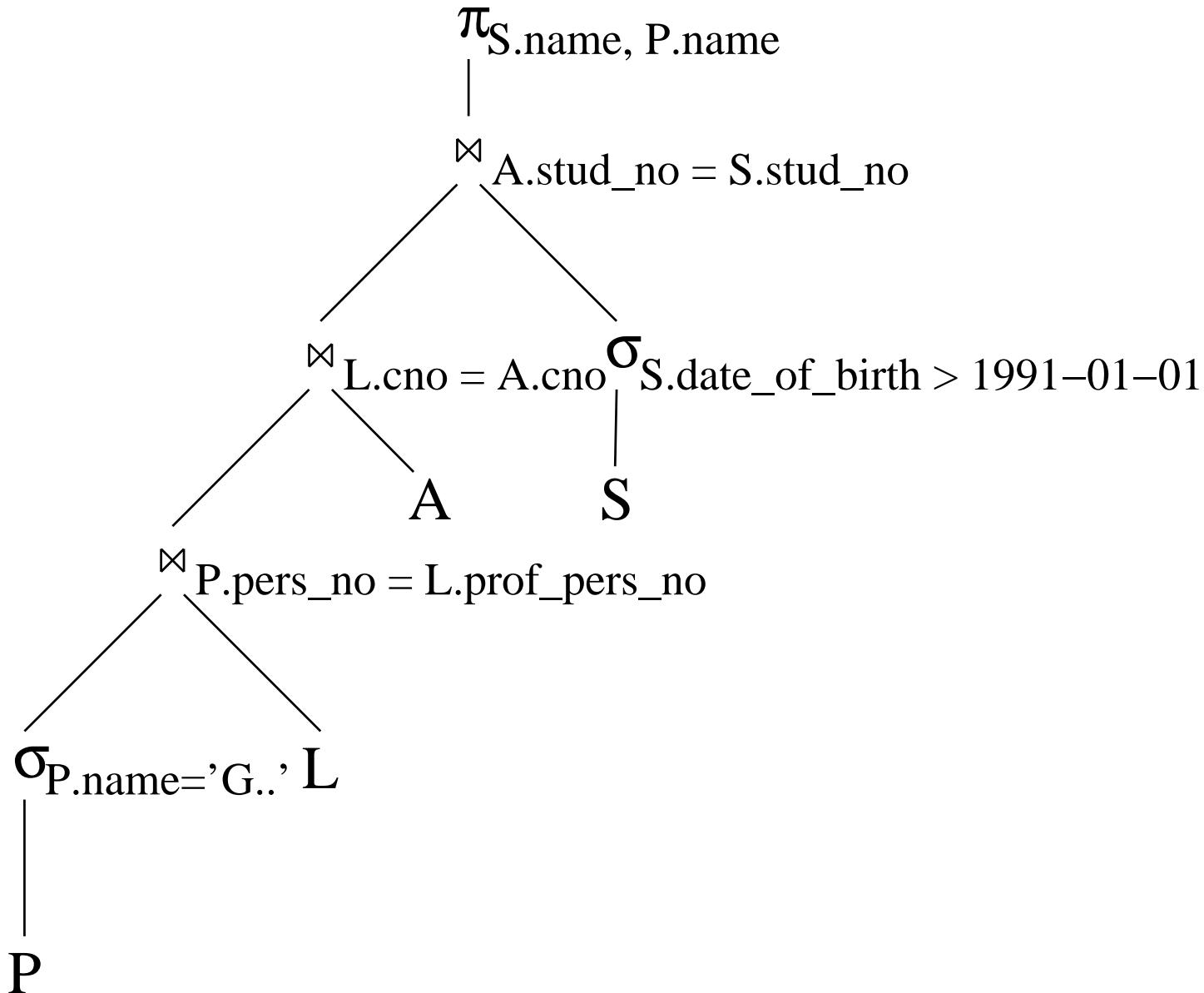
- Basic techniques
  - Breaking up selections
  - Pushing “down” selections
  - Converting selection and Cartesian products into joins
  - Determining join order
  - Inserting projections
  - Pushing “down” projections

# Example Query

- Let's play through some options with a concrete query:

```
select S.name, P.name  
from student S, attends A, lecture L, professor P  
where S.stud_no = A.stud_no  
and A.course_no = L.course_no  
and L.prof_pers_no = P.pers_no  
and S.date_of_birth > 1991-01-01  
and P.name = 'Gamper';
```

# (Optimized) Query Plan



# (A Selection of) Rewrite Rules

$$R_1 \bowtie R_2 = R_2 \bowtie R_1$$

$$R_1 \cup R_2 = R_2 \cup R_1$$

$$R_1 \cap R_2 = R_2 \cap R_1$$

$$R_1 \times R_2 = R_2 \times R_1$$

$$R_1 \bowtie (R_2 \bowtie R_3) = (R_1 \bowtie R_2) \bowtie R_3$$

$$R_1 \cup (R_2 \cup R_3) = (R_1 \cup R_2) \cup R_3$$

$$R_1 \cap (R_2 \cap R_3) = (R_1 \cap R_2) \cap R_3$$

$$R_1 \times (R_2 \times R_3) = (R_1 \times R_2) \times R_3$$

$$\sigma_{p_1 \wedge p_2 \wedge \dots \wedge p_n}(R) = \sigma_{p_1}(\sigma_{p_2}(\dots(\sigma_{p_n}(R))\dots))$$

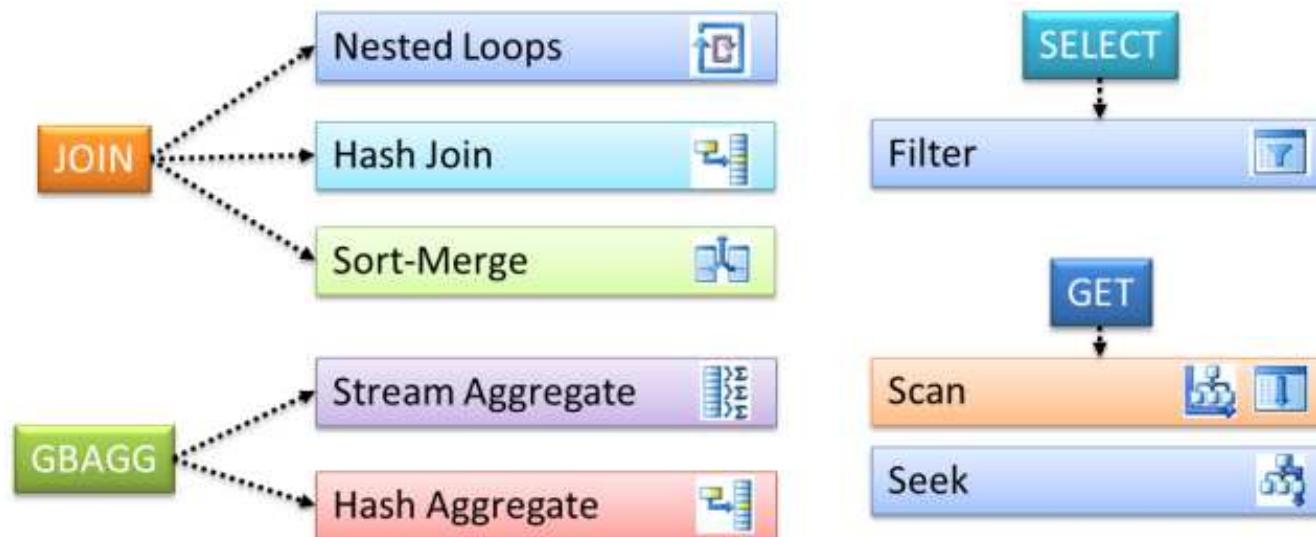
$$\Pi_{l_1}(\Pi_{l_2}(\dots(\Pi_{l_n}(R))\dots)) = \Pi_{l_1}(R)$$

with  $l_1 \subseteq l_2 \subseteq \dots \subseteq l_n \subseteq \mathcal{R} = \text{schema}(R)$

$$\Pi_l(\sigma_p(R)) = \sigma_p(\Pi_l(R)), \text{ if } \text{attr}(p) \subseteq l$$

# Physical Optimization

- Up to now, we have only looked at relational algebra operators on a logical level
- However, an operator can be implemented in different ways
- Optimization is not only about rewriting the plan, but about choosing an implementation for each one



# Physical Optimization (2)

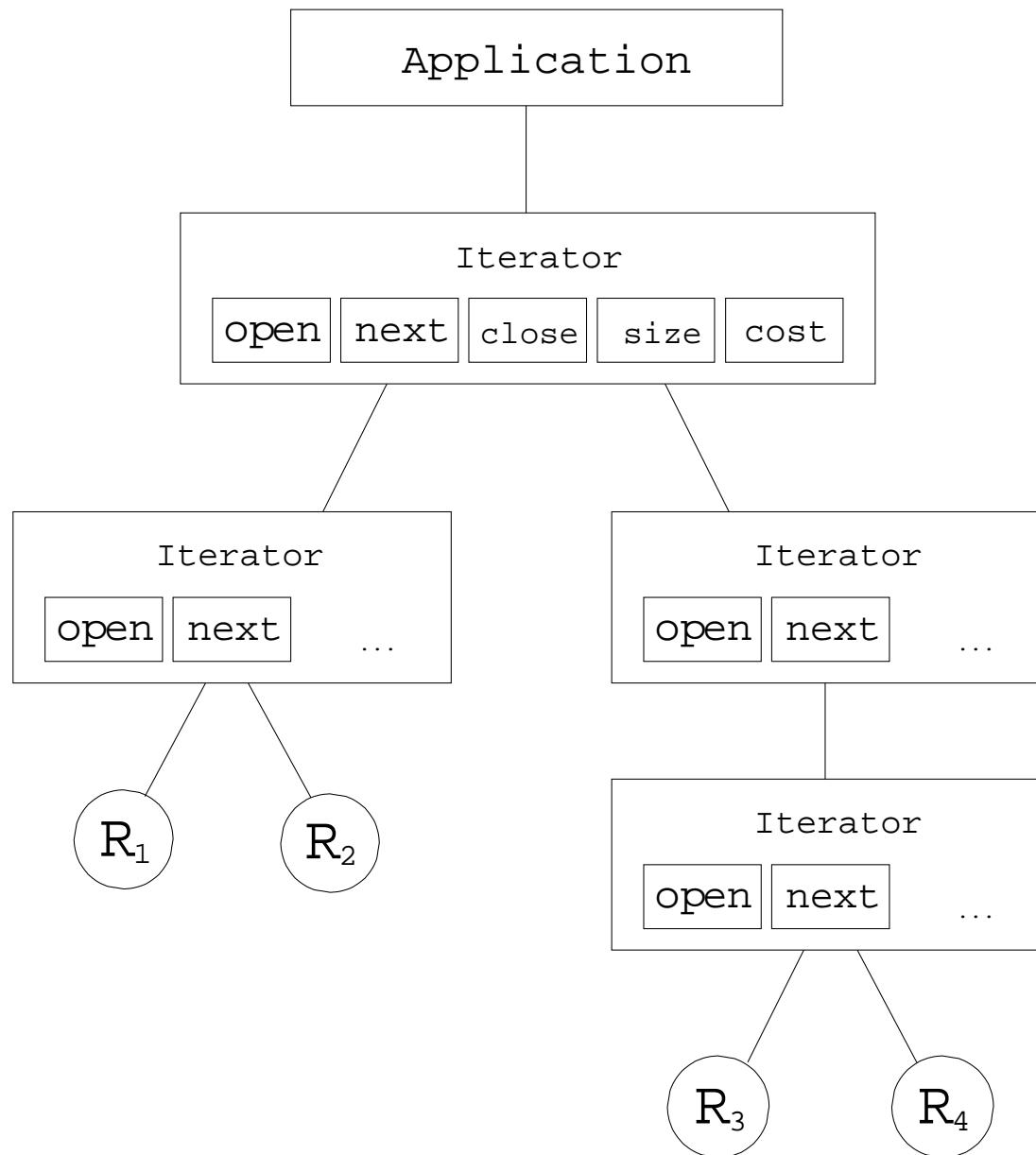
- Other aspects of physical optimization include:
  - Decide whether to use indexes or not (and which ones)
  - Whether to materialize intermediate results or not
  - When to sort and eliminate duplicates
  - Which cost models and statistics to use

# Iterators



- Operators have to be implemented in a way so that they can be plugged together
- One approach is to implement them as *iterators*
  - Each operator produces a tuple on request from the operator above it

# Iterators (2)



# Implementing Selection

## iterator Scan<sub>p</sub>

- **open**
  - open input (relation)
- **next**
  - fetch tuples until one of them satisfies predicate  $p$
  - return this tuple
- **close**
  - close input

# Implementing Selection (2)

**iterator**  $\text{IndexScan}_p$

- **open**
  - open input (index)
  - look up first tuple in index that satisfies predicate  $p$
- **next**
  - return current tuple if it satisfies  $p$
  - go to next tuple in index
- **close**
  - close input

# Implementing Join

- Set difference and intersection can be implemented similarly to join
- Here we only look at equi-joins
- An implementation that always works is a nested-loop join:

```
for each  $r \in R$ 
  for each  $s \in S$ 
    if  $r.A = s.B$  then
       $res := res \cup (r \times s)$ 
```

# Implementing Join(2)

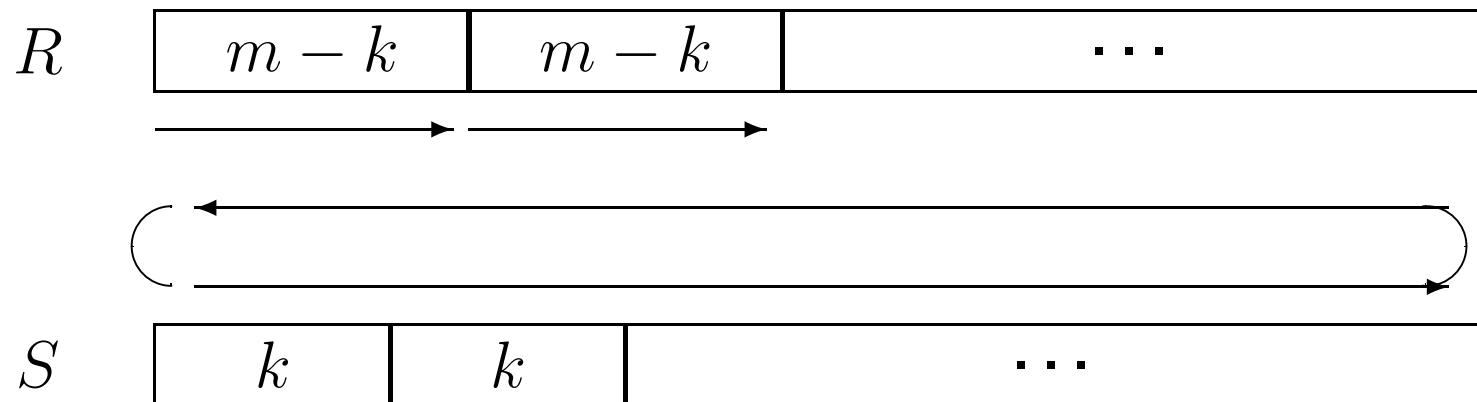
**iterator** NestedLoop<sub>*p*</sub>

- **open**
  - open the left input
  - get first tuple of left input
- **next**
  - right input closed?
    - open right input
  - get tuples from right input until one satisfies join predicate *p*
  - If right input is exhausted
    - close right input
    - get next tuple from left input
    - go to **next**
  - return join of current tuples
- **close**
  - close left and right input

# Refined Join Algorithm

- Relations are stored on pages
- You have  $m$  pages reserved in main memory:
  - $k$  for the inner loop of a nested-loop join
  - $m - k$  for the outer loop

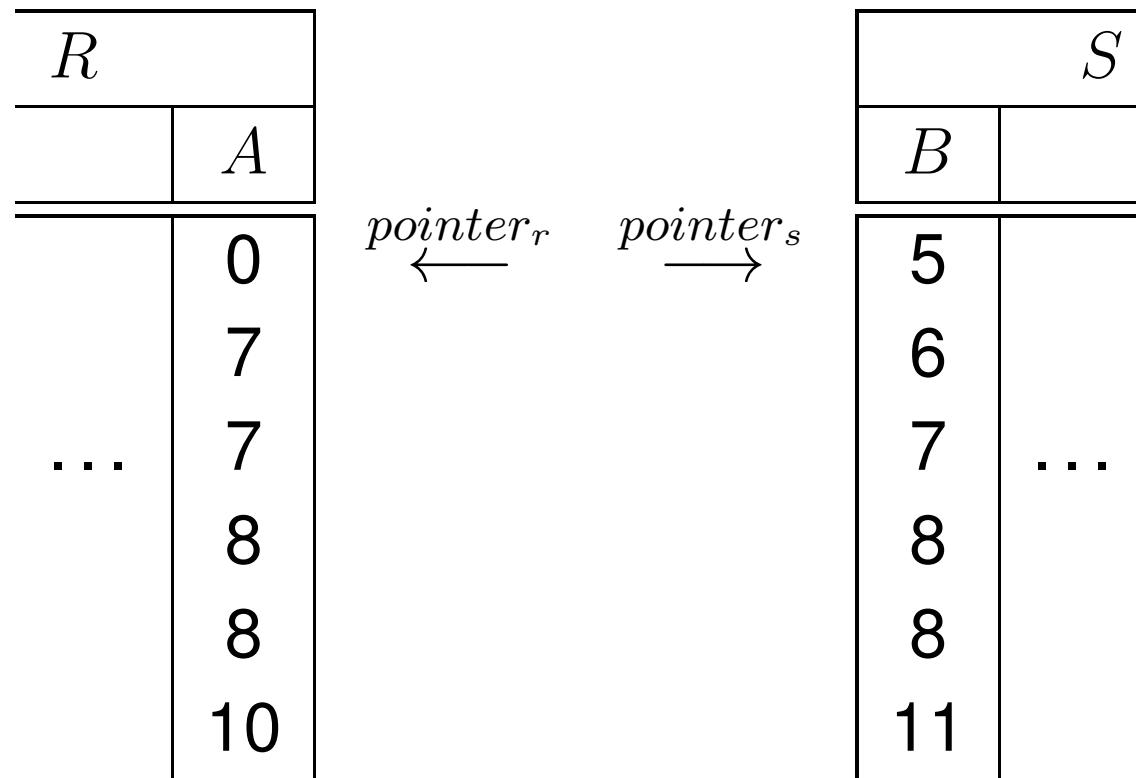
Joining  $R$  and  $S$ :



# (Sort-)Merge-Join

- Prerequisite:  $R$  and  $S$  are sorted
  - You may need to do a sort operation before joining

Example:



# (Sort-)Merge-Join (2)

**iterator**  $\text{MergeJoin}_p$

- **open**
  - open both inputs
  - set  $\text{pointer}_r$  and  $\text{pointer}_s$  to smallest common value
  - set  $\text{marker}$  to  $\text{pointer}_s$
- **close**
  - close both inputs

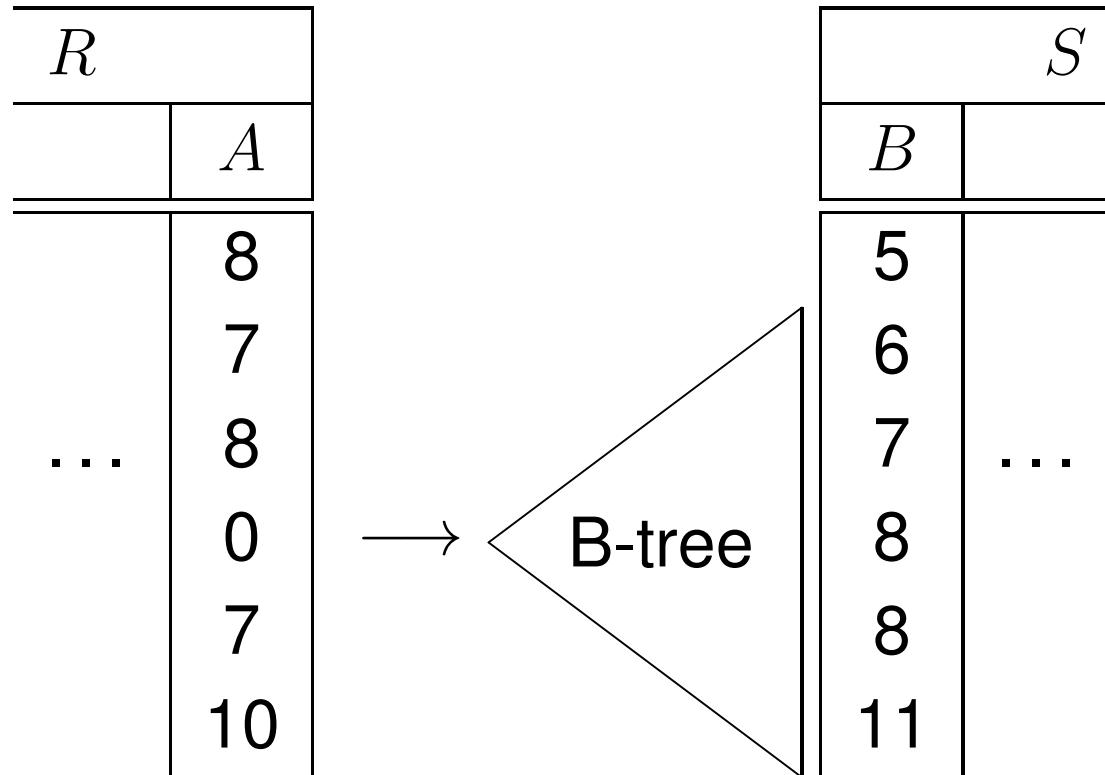
# (Sort-)Merge-Join (3)

**iterator**  $\text{MergeJoin}_p$

- **next**

- join  $\text{pointer}_r$  and  $\text{pointer}_s$
- advance  $\text{pointer}_s$
- if  $\text{pointer}_s$ 's value has changed
  - advance  $\text{pointer}_r$
  - if  $\text{pointer}_r$ 's value has changed
    - set  $\text{pointer}_r$  and  $\text{pointer}_s$  to next common value
    - set **marker** to  $\text{pointer}_s$
  - else
    - set  $\text{pointer}_s$  to **marker**
- return joined tuple

# Index-Join



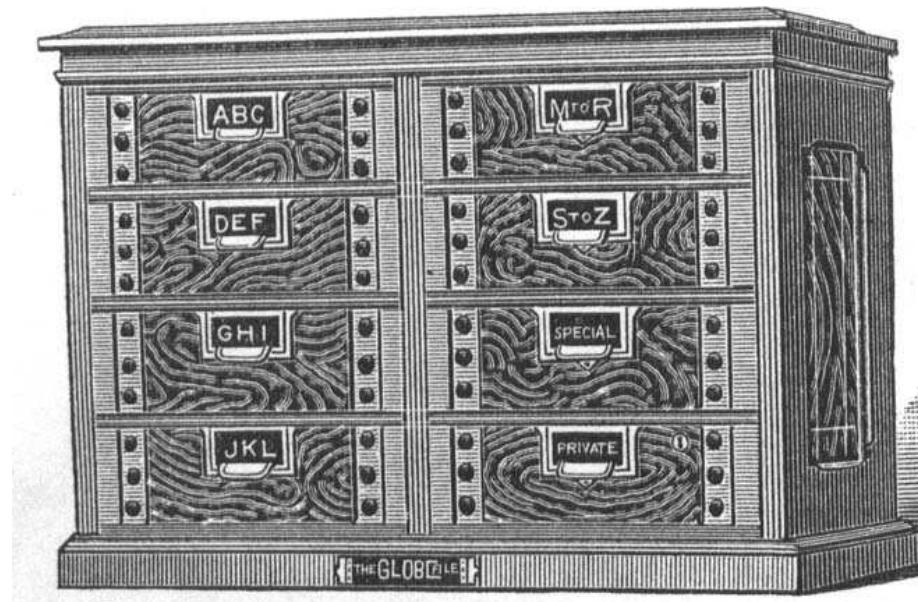
# Index-Join (2)

**iterator** IndexJoin<sub>p</sub>

- **open**
  - open left input
  - fetch first tuple from left input
  - look up value in index
- **next**
  - join tuple if index provides (another) tuple for this value and return tuple
  - otherwise
    - advance left input
    - look up value in index
    - goto **next**
- **close**
  - close input

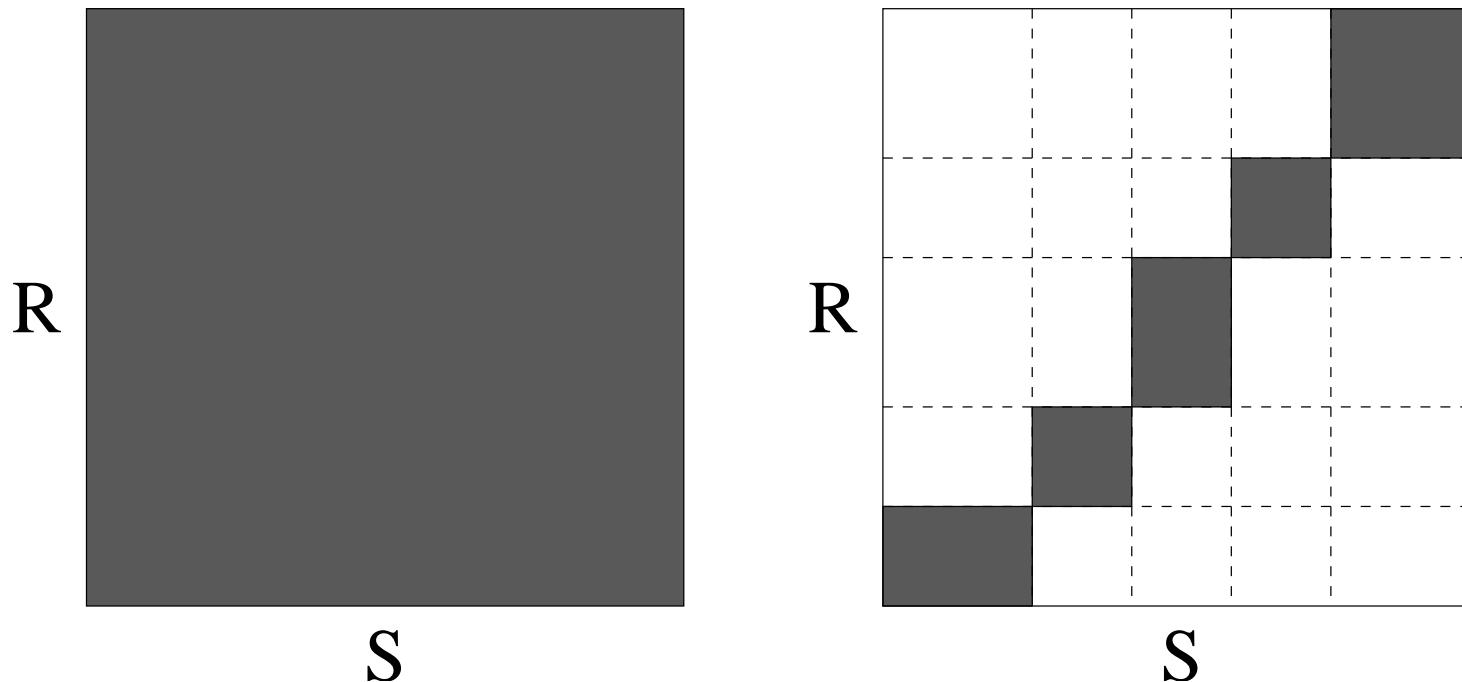
# Index-Join(3)

- Disadvantage of the index join:
  - Indexes only exist on base tables
  - Creating an index for intermediate results usually too costly
  - Looking up values in an index can be quite expensive (lots of random I/O)

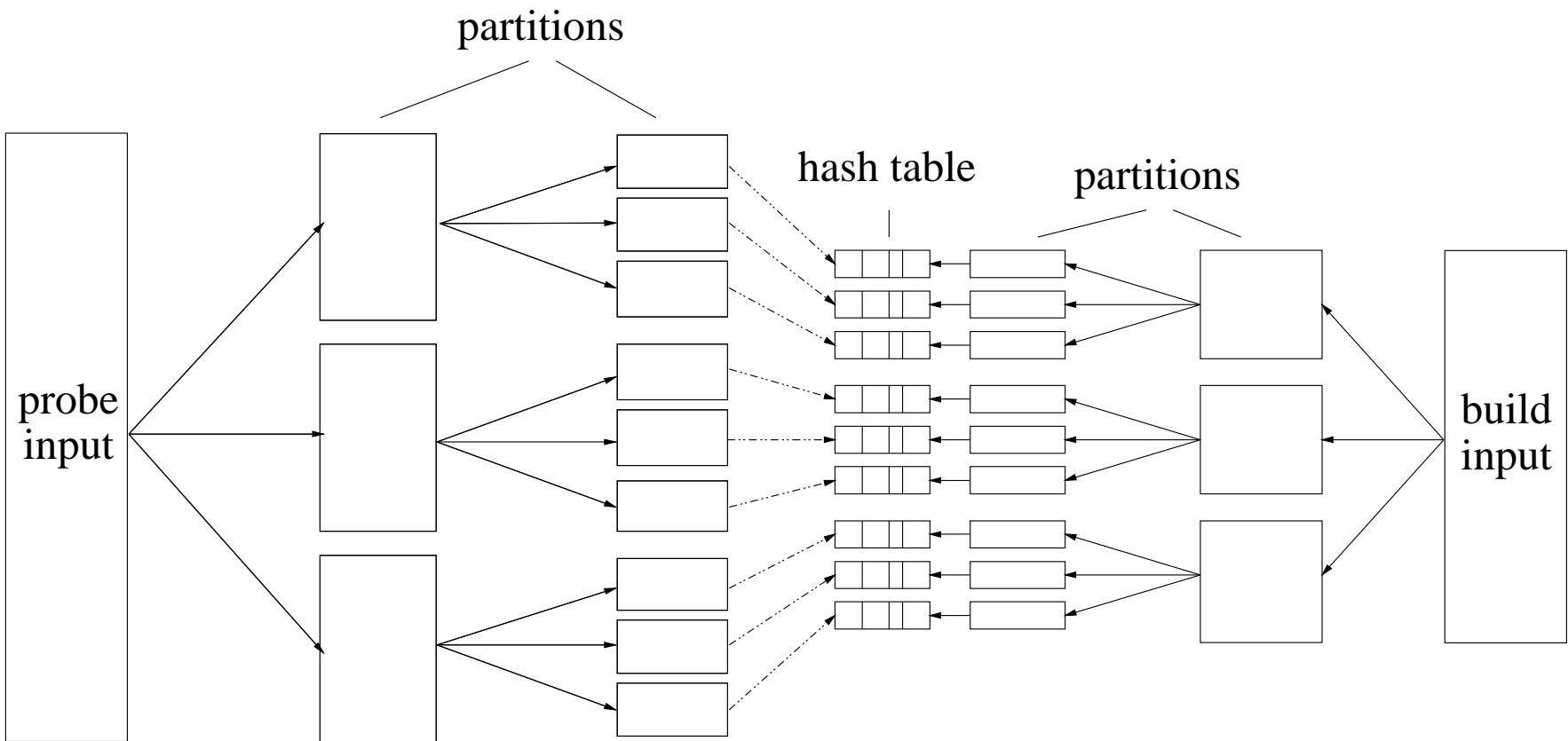


# Hash-Join

- Main idea: partition the relations
- Create a main-memory index for each partition

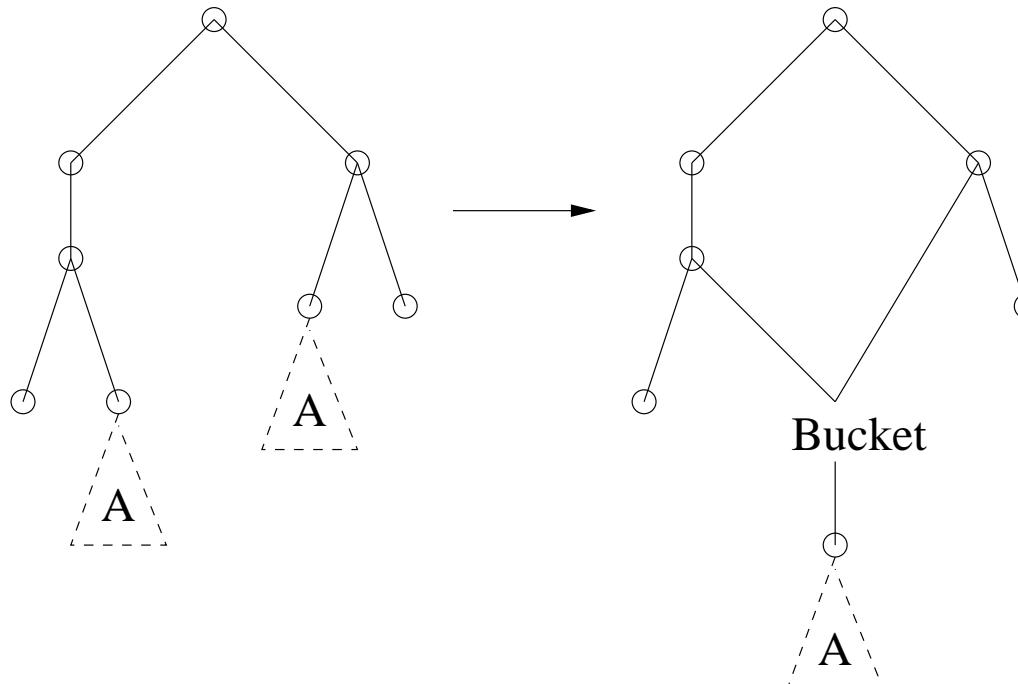


# Partitioning of Relations



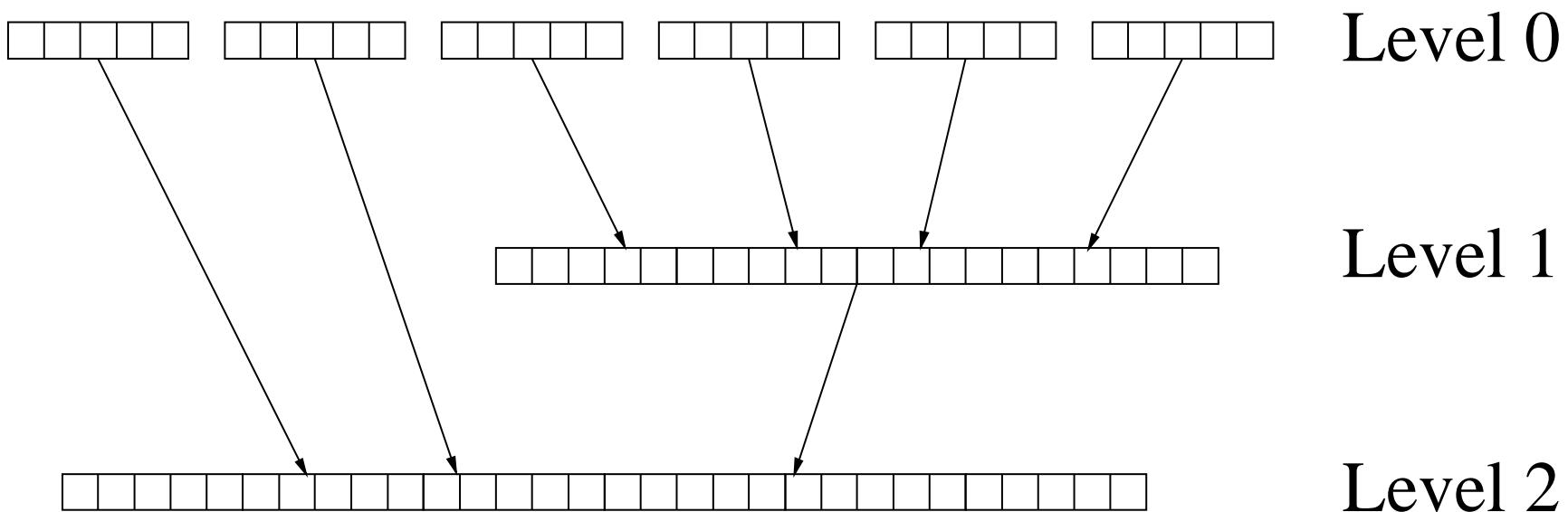
# Materialization/Factorization

- If main memory is not sufficient for processing an operator
  - Intermediate results have to be materialized (stored on disk)
- Also makes sense if there are common subexpressions:

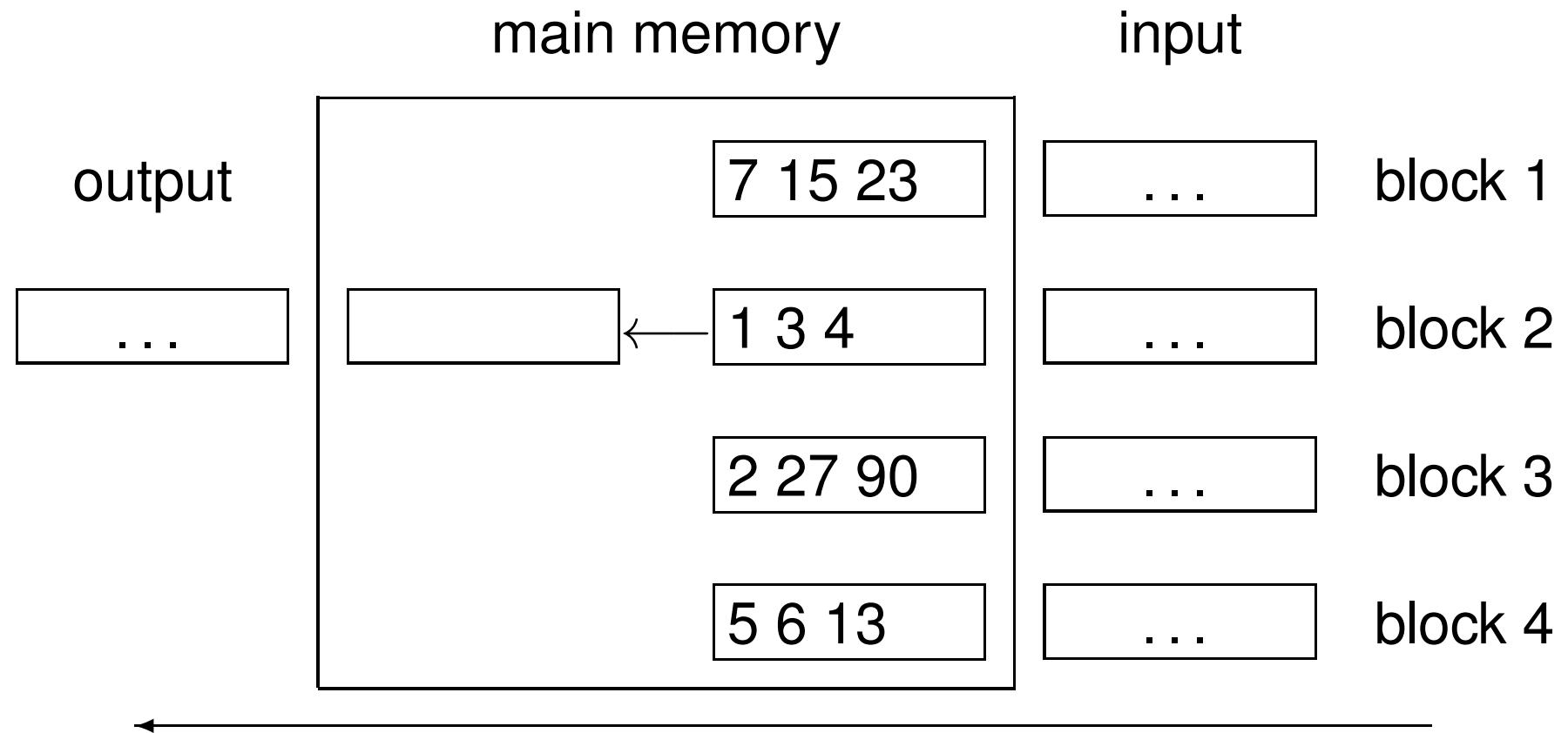


# External Sorting

- Divide up the data into blocks that fit main memory
- Sort each block in main memory
- Sort the data by merging two (or more) blocks in separate steps



# Merging Blocks



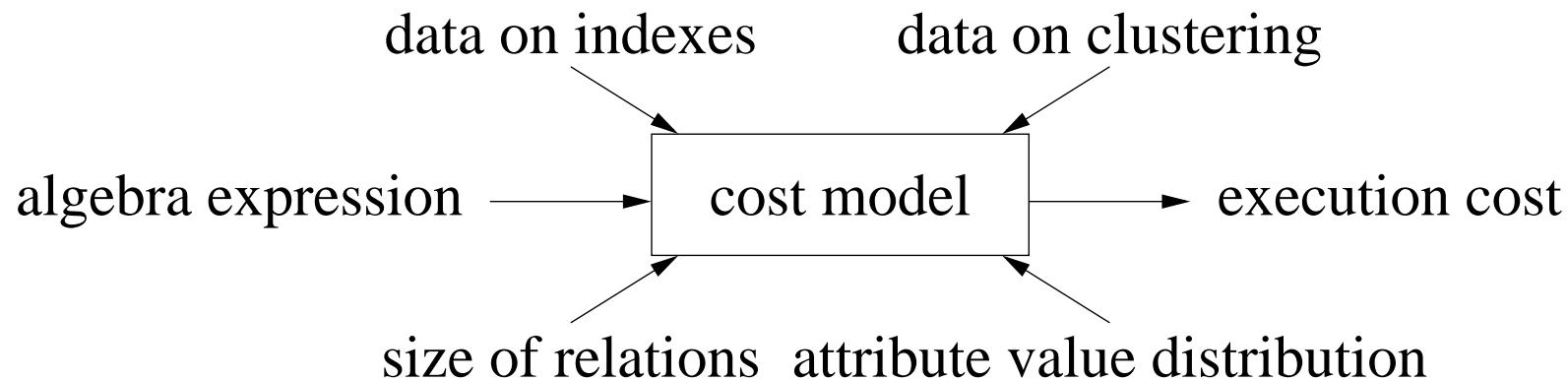
# Replacement Selection

- It's even possible to sort blocks that are larger than main memory:

output					main memory					input						
					10	20	30	40		25	73	16	26	33	50	31
					10	20	25	30	40	73	16	26	33	50	31	
					10	20	25	30	40	73	16	26	33	50	31	
					10	20	25	(16)	30	40	73	26	33	50	31	
					10	20	25	30	(16)	(26)	40	73	33	50	31	
					10	20	25	30	40	(16)	(26)	(33)	73	50	31	
10	20	25	30	40	73	(16)	(26)	(33)	(50)		31					
					16	26	31	33	50							

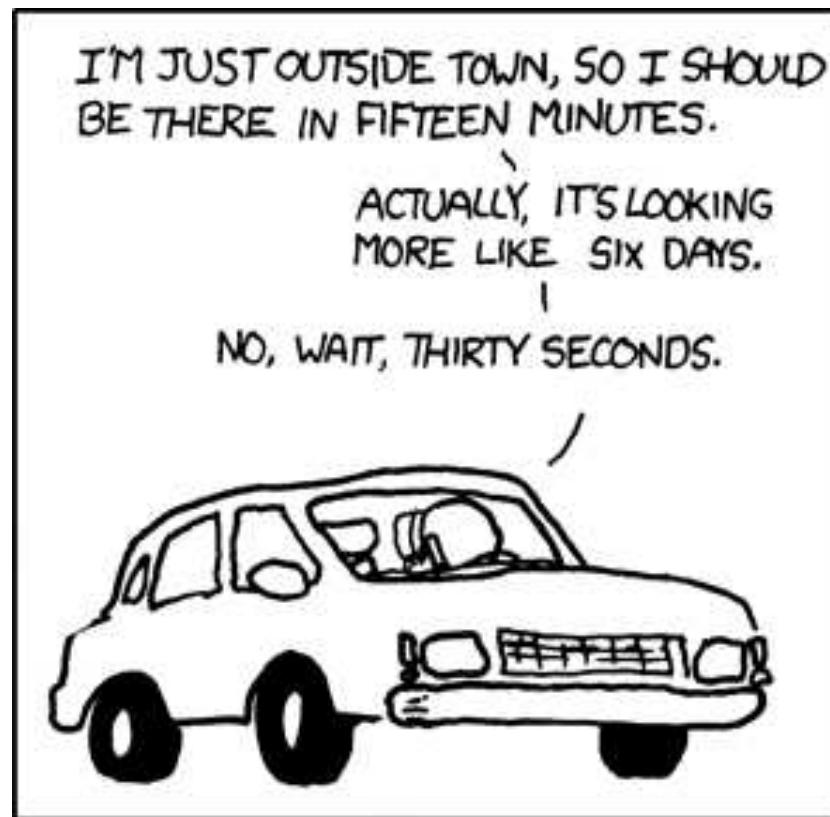
# Estimating Costs

- In order to make informed decisions, an optimizer has to employ cost models
- A cost model estimates the costs of an execution plan based on a number of input parameters:



# Estimating Costs (2)

- Estimating the execution time is far from trivial



THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

# Selectivities

- *Selectivity* is the ratio of tuples passing a filter operation
- Selection with predicate  $p$ :

$$sel_p := \frac{|\sigma_p(R)|}{|R|}$$

- Join of  $R$  with  $S$ :

$$sel_{RS} := \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie S|}{|R| \cdot |S|}$$

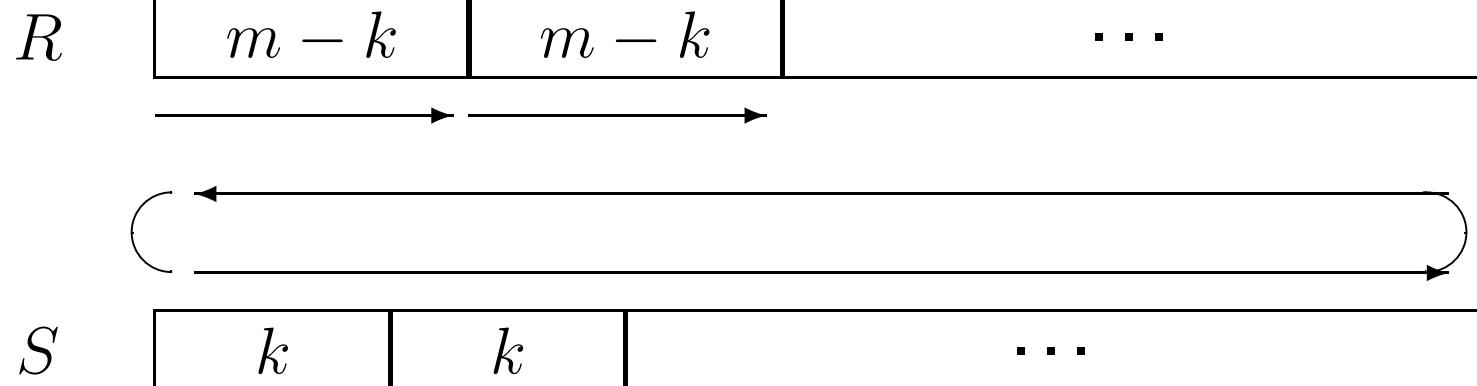
# Selectivities (2)

- Estimating selectivities:
  - $sel_{R.A=C} = \frac{1}{|R|}$   
if  $A$  is a key of  $R$
  - $sel_{R.A=C} = \frac{1}{i}$   
if  $i$  is the number of different values of  $R.A$   
(assuming uniform distribution)
  - $sel_{R.A=S.B} = \frac{1}{|R|}$   
if equi-joining via foreign key  $S.B$
- Otherwise, optimizer has to get information via
  - sampling
  - histograms

# Estimating Costs of a Selection

- Brute force: scanning all pages of  $R$
- $B^+$ -tree index:  $t + \lceil sel_{A\theta c} \cdot b_R \rceil$ 
  - descend to the leaf level of the tree
  - read the tuples satisfying the predicate
- Hash index: one look-up for every attribute value satisfying the predicate

# Estimating Costs of a Join



- Scanning all pages of  $R$ :  $b_R$
- Number of iterations of inner loop:  $\lceil b_R / (m - k) \rceil$
- Total:  $b_R + k + \lceil b_R / (m - k) \rceil \cdot (b_S - k)$
- This is minimal, if  $k = 1$  and  $R$  is the smaller relation

# Summary

- Answering user queries *efficiently* is important for a DBMS
- Most DBMSs have a query optimizer that tries to find an efficient execution plan for every query
- Usually hidden from the eyes of a user
- However, knowing about this helps a user in analyzing bottlenecks

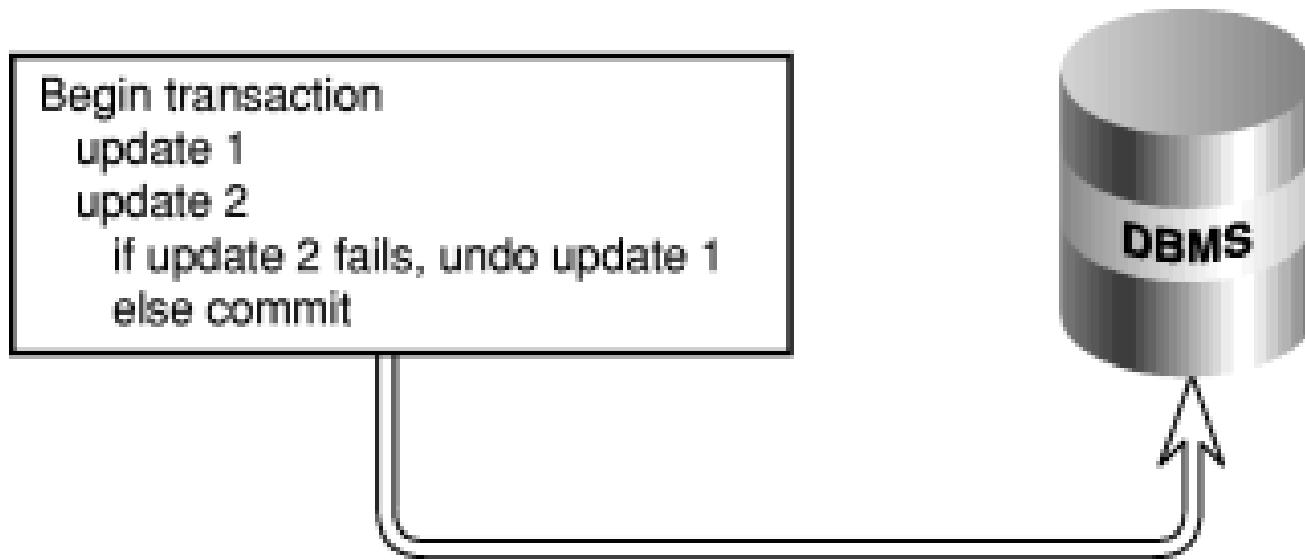


# Chapter 9

## Transaction Management

# Transaction Management

- Transaction management consists of the two subtopics
  - *Recovery*, i.e., recovering from some kind of failure of the DBMS
  - *Synchronization* of concurrently running transactions



# Example for a Transaction (TA)

- Transfer money from account A to account B:
  - Read balance of account A, store it in  $a$ : `read(A,a);`
  - Reduce balance by €50:  $a := a - 50;$
  - Write back the new balance: `write(A,a);`
  - Read balance of account B, store it in  $b$ : `read(B,b);`
  - Increase balance by €50:  $b := b + 50;$
  - Write back the new balance: `write(B,b);`
- Has to be done in one go!
- If run in a *transaction* (TA), DBMS guarantees that either
  - everything is done
  - or nothing at all

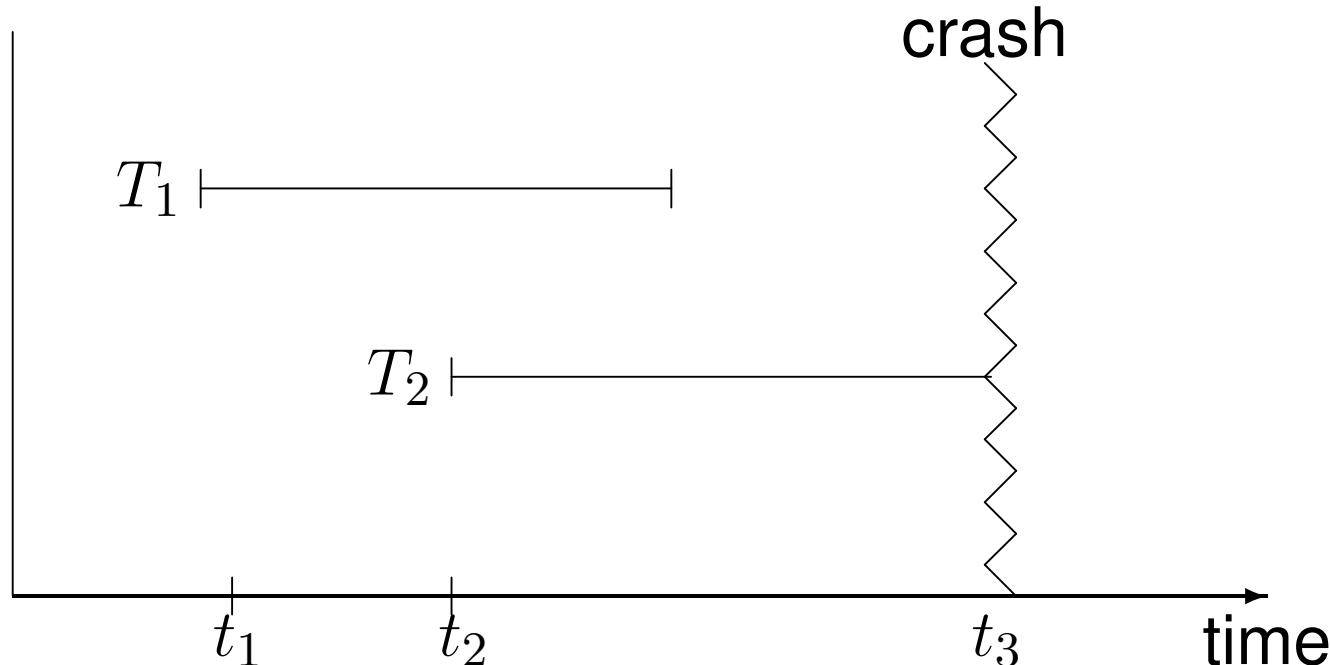
# Operations for Transactions

- **begin of transaction (BOT):**
  - signals the start of a transaction
- **commit:**
  - successful completion of a transaction
  - changes are made persistent in the DBMS
- **abort:**
  - unsuccessful end of a transaction (triggered by the transaction itself)
  - rollback to the state at the start of the transaction

# Operations for Transactions (2)

- **define savepoint:**
  - define a point to which a (still active) transaction can roll back to
- **backup transaction:**
  - active transaction is rolled back to the last savepoint
  - may also be possible to return to an earlier savepoint

# System Crash



- Changes made by  $T_1$  have to be in the database:
  - Because  $T_1$  has successfully committed before  $t_3$
- Changes made by  $T_2$  have to be rolled back
  - Because  $T_2$  has *not* successfully committed before  $t_3$
  - $T_2$  has to be restarted after database is up and running again

# Transactions and SQL

- SQL uses the following keywords (part of DCL):
  - **commit (work):**
    - End transaction successfully and make changes persistent
    - Will only work if no errors occurred
  - **rollback (work):**
    - End transaction (unsuccessfully) and roll back any changes
    - A DBMS always has to guarantee that a rollback can be done “successfully”

# Transactions and SQL (2)

- Example in SQL

**insert into** lecture

```
values (5275, 'nuclear physics', 3, 2141);
```

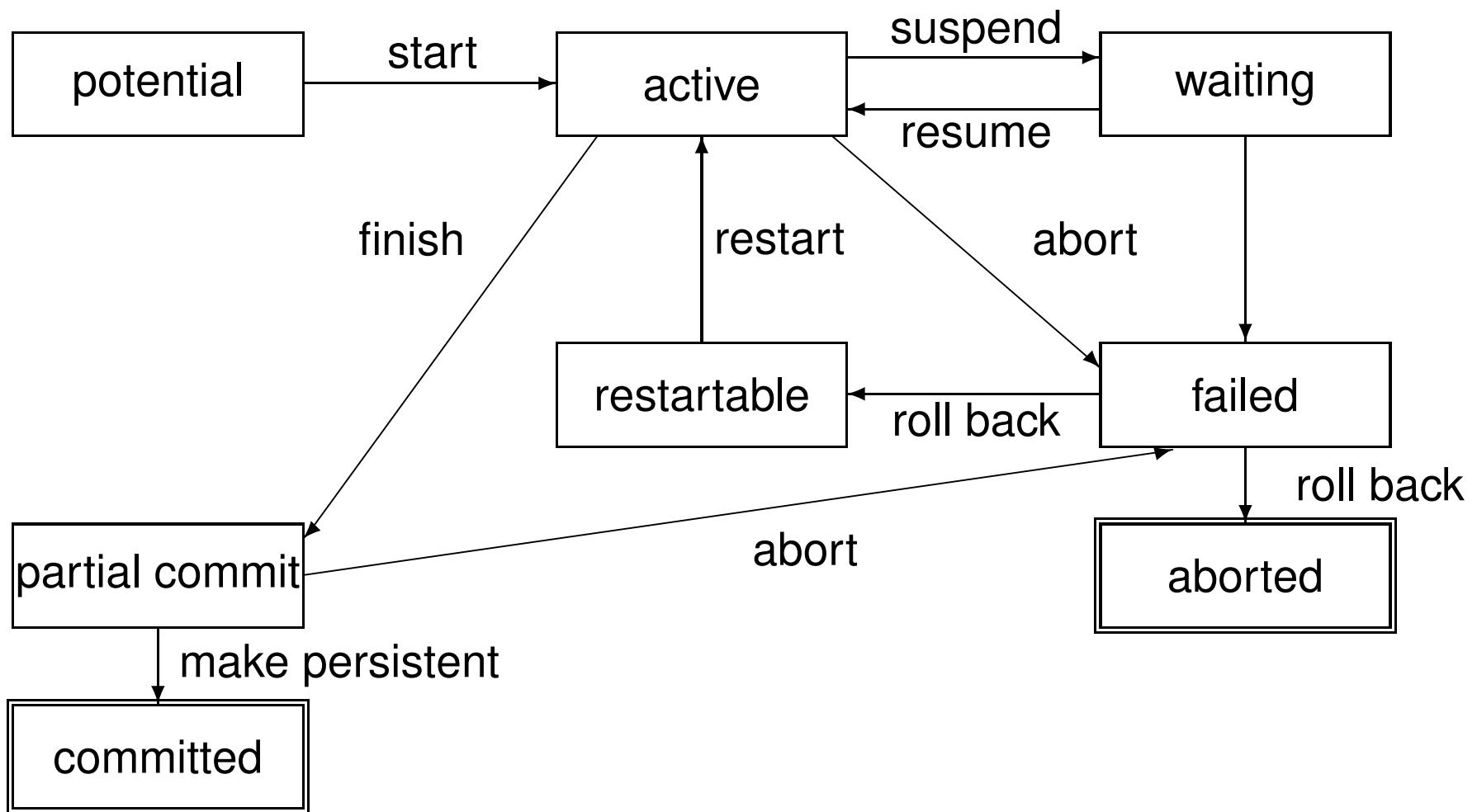
**insert into** professor

```
values (2141, 'Meitner', 'C4', 205);
```

**commit work**

- The begin of a transaction is usually done implicitly
- You couldn't do a successful commit after the first insert statement (referential integrity is violated)
- However, after the second insert statement, everything is fine and the transaction can commit

# State Transitions



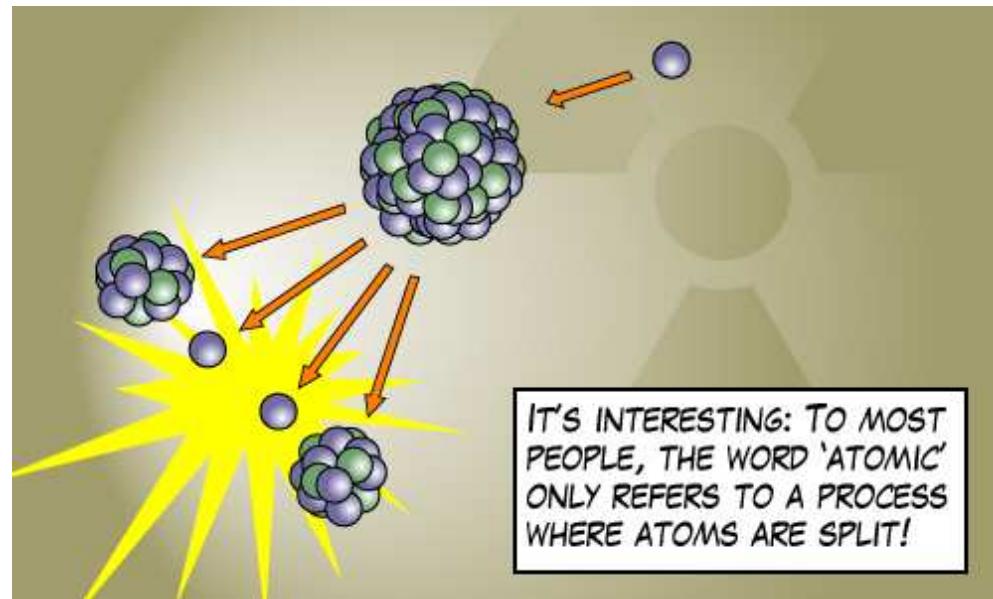
# Summary

- Transactions should follow the ACID paradigm
- ACID:
  - Atomicity
  - Consistency
  - Isolation
  - Durability



# Atomicity

- Atomicity means:
  - “All or nothing”
  - Either all operations of a transaction are executed or none



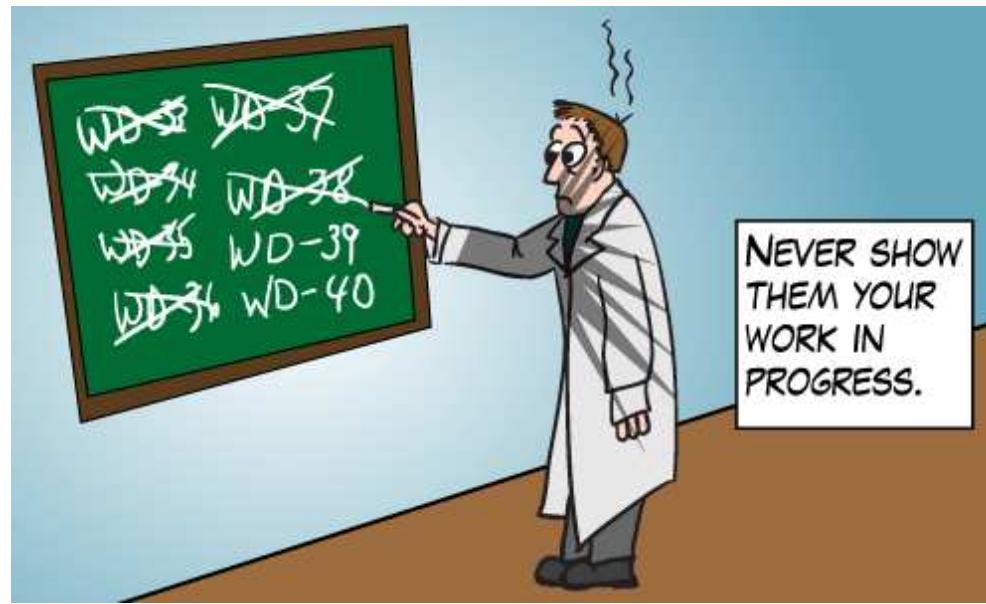
# Consistency

- Consistency means:
  - If a transaction starts with a consistent database state, after its execution the state will still be consistent
  - What happens in between is a different story...



# Isolation

- Isolation means:
  - Concurrently running transactions have no side effects on each other
  - That means, every transaction thinks it is the only one running in the database



# Durability

- Durability means:
  - Once a transaction has committed, a database has to guarantee that its results survive any malfunction



# Chapter 10

## Recovery

# Recovery

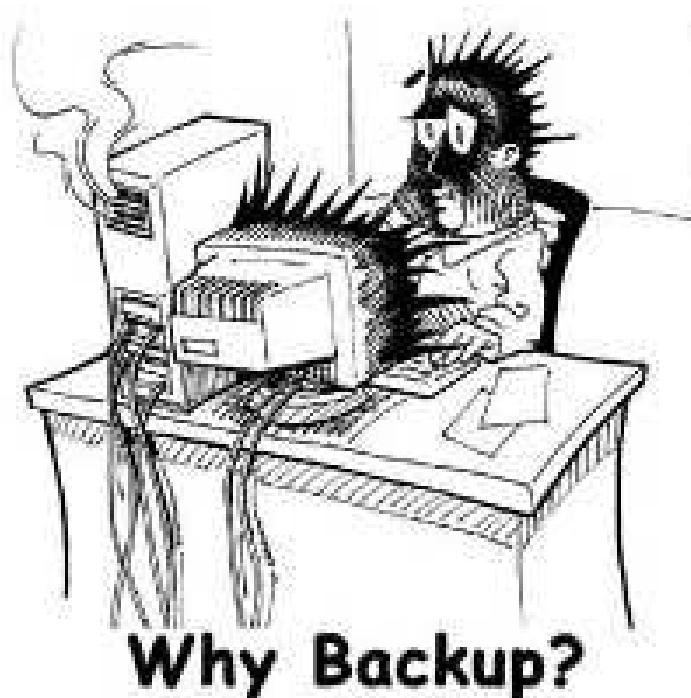
- An important task of a DBMS is to avoid losing data caused by a system crash
- The recovery system of a DBMS utilizes two important mechanisms:
  - Backups
  - Log files

**Our Disaster Recovery Plan  
Goes Something Like This...**

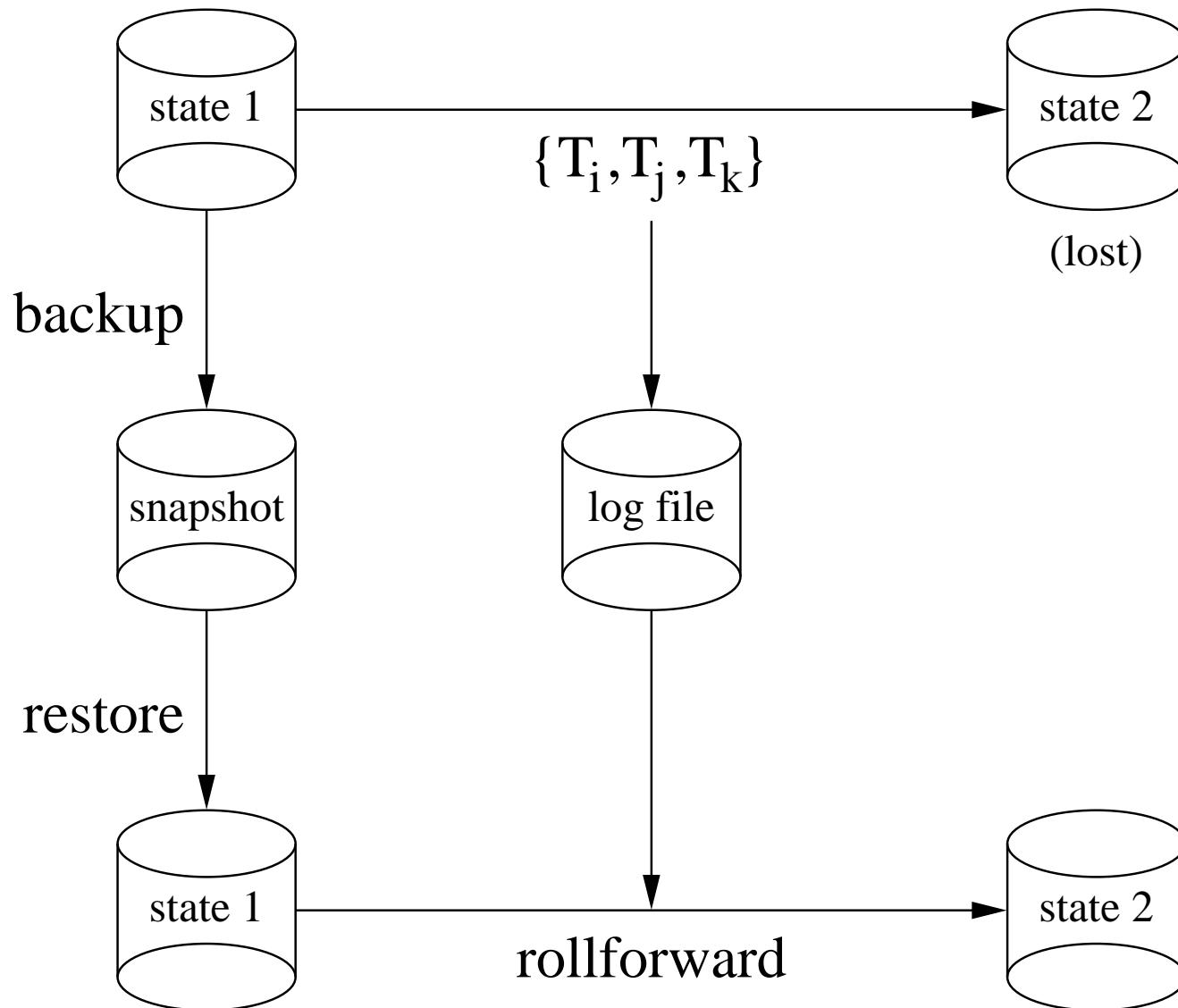


# Recovery(2)

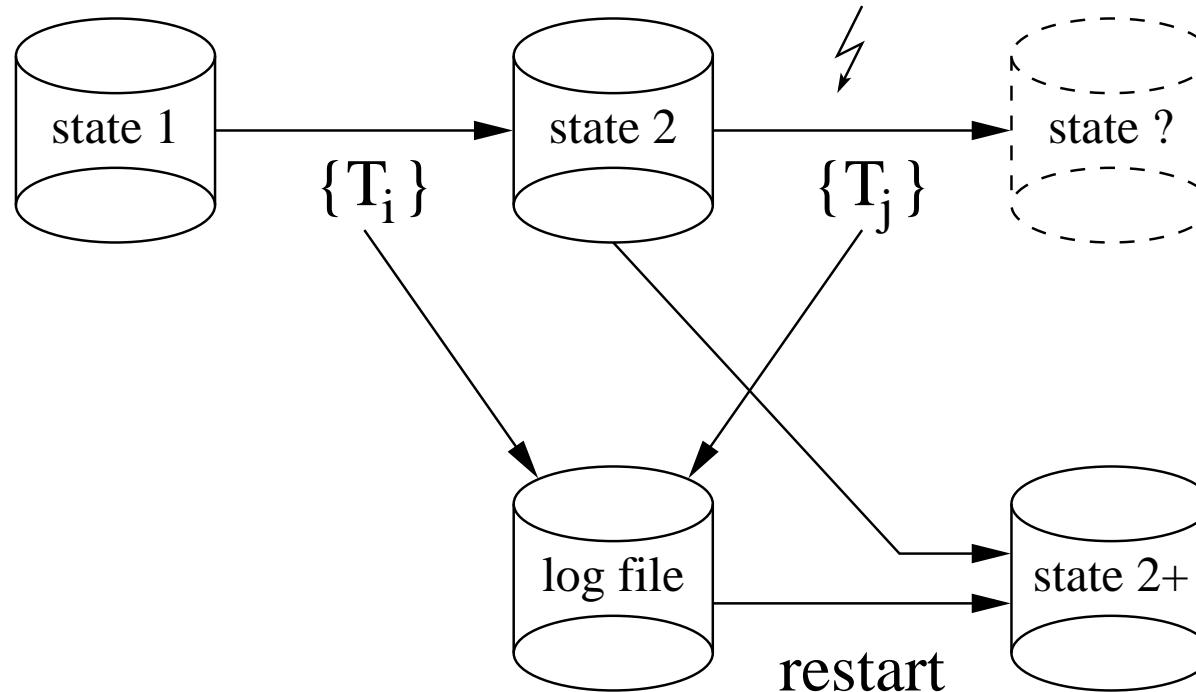
- A *backup* is a snapshot of the database content taken at a specific time (also called *checkpoint*)
- In a *log file* all the changes in the database are recorded
- Clearly, a backup and log files should not reside on the same disk or server...



# Loss of Non-Volatile Storage



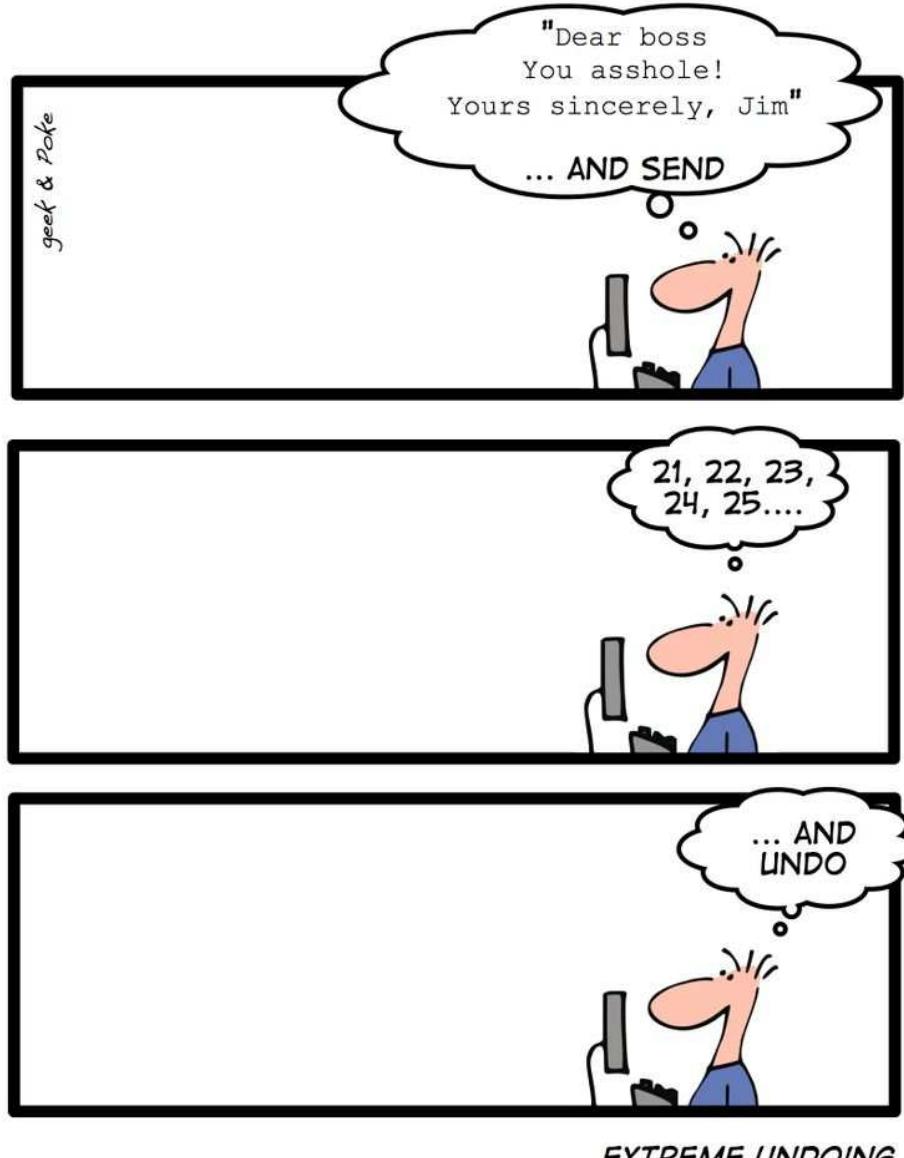
# Loss of Volatile Storage



- Problem: some TAs in  $\{T_j\}$  were still active, others have committed
  - Restart will recover state 2 + the changes made by the committed TAs in  $\{T_j\}$

# Aborting a Transaction

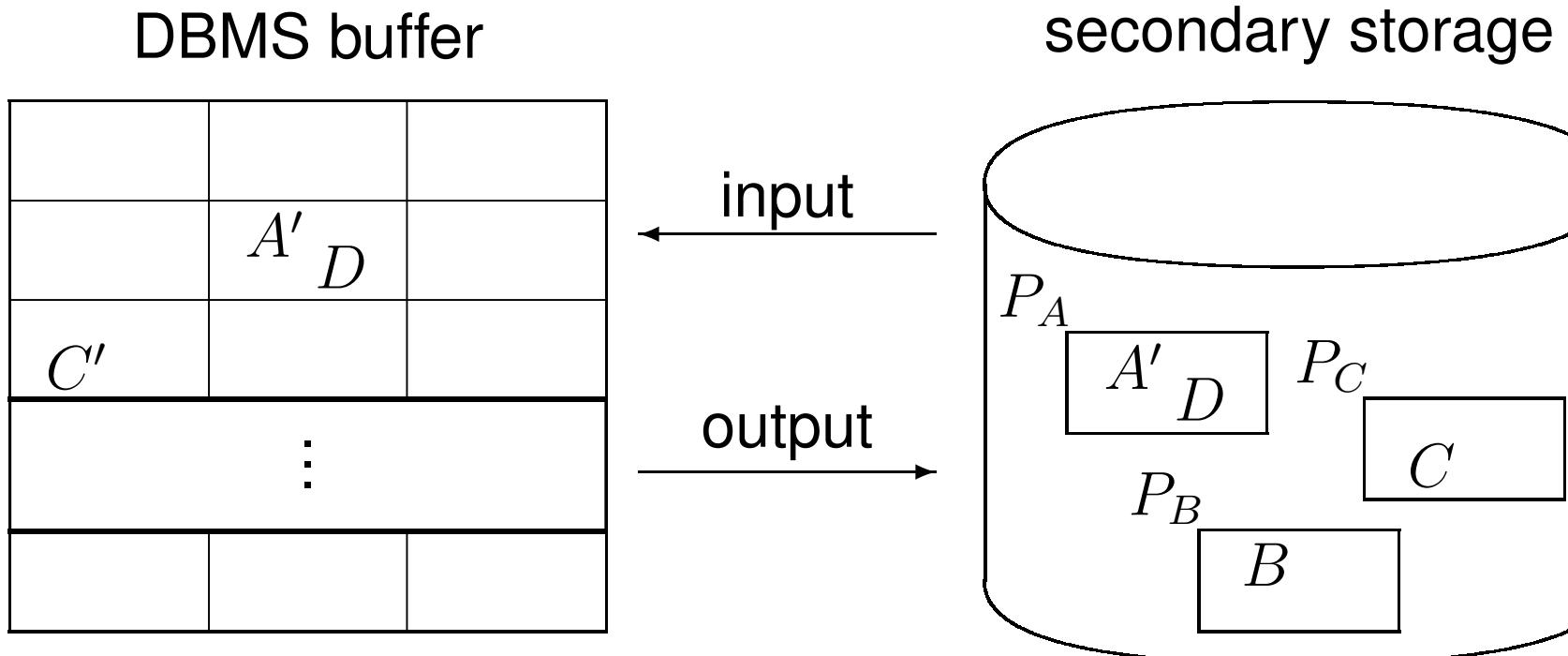
- Log files can also be used to undo changes made by an aborted TA
- Let's have a look at the nitty-gritty details now...



# Classification of Failures

- Local failure of a non-committed TA:
  - Changes made by the TA have to be undone
  - This is called *R1* recovery
- Failure with loss of volatile storage (main memory)
  - Changes made by committed TAs have to be redone (R2 recovery)
  - Changes made by non-committed TAs have to be undone (R3 recovery)
- Failure with loss of non-volatile storage (disk)
  - R4-Recovery: restoring backup + R2 and R3 recovery

# TAs and Memory Hierarchy



# TAs and Memory Hierarchy (2)

- Replacement strategies for buffer pages
  - ↗ steal: pages which have been modified by a TA still running may not be replaced
  - steal: every page that is not pinned is potentially a candidate for replacement

# TAs and Memory Hierarchy (3)

- Strategies for making changes of committed TAs persistent:
  - force: changes are made persistent as soon as a TA commits
  - $\neg$  force: pages which have been changed can remain in main memory and are written back later

# Effect of Strategies on Recovery

	force	$\neg$ force
$\neg$ steal	<ul style="list-style-type: none"><li>• no Redo needed</li><li>• no Undo needed</li></ul>	<ul style="list-style-type: none"><li>• Redo needed</li><li>• no Undo needed</li></ul>
steal	<ul style="list-style-type: none"><li>• no Redo needed</li><li>• Undo needed</li></ul>	<ul style="list-style-type: none"><li>• Redo needed</li><li>• Undo needed</li></ul>

# Write Strategies

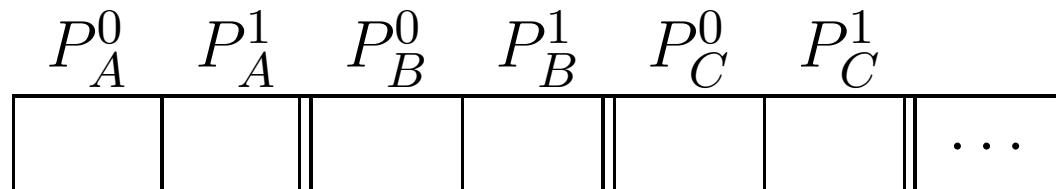
- Update-in-Place
  - Every page has exactly one location on disk
  - The previous state of the page is overwritten
  - DBMS has to make sure that previous state is preserved somewhere (for recovery purposes)



# Write Strategies (2)

- Twin-Block

- Every page has exactly two locations on disk
- Previous state is preserved, pages are used alternately
- Very costly in terms of storage space



# Write Strategies (3)

- Shadow Paging
  - Only modified pages are duplicated
  - Needs less space than twin block
  - However, has some disadvantages:
    - Storage space starts to fragment, pages and shadow pages are usually not clustered
    - Also needs some overhead for concurrently running TAs

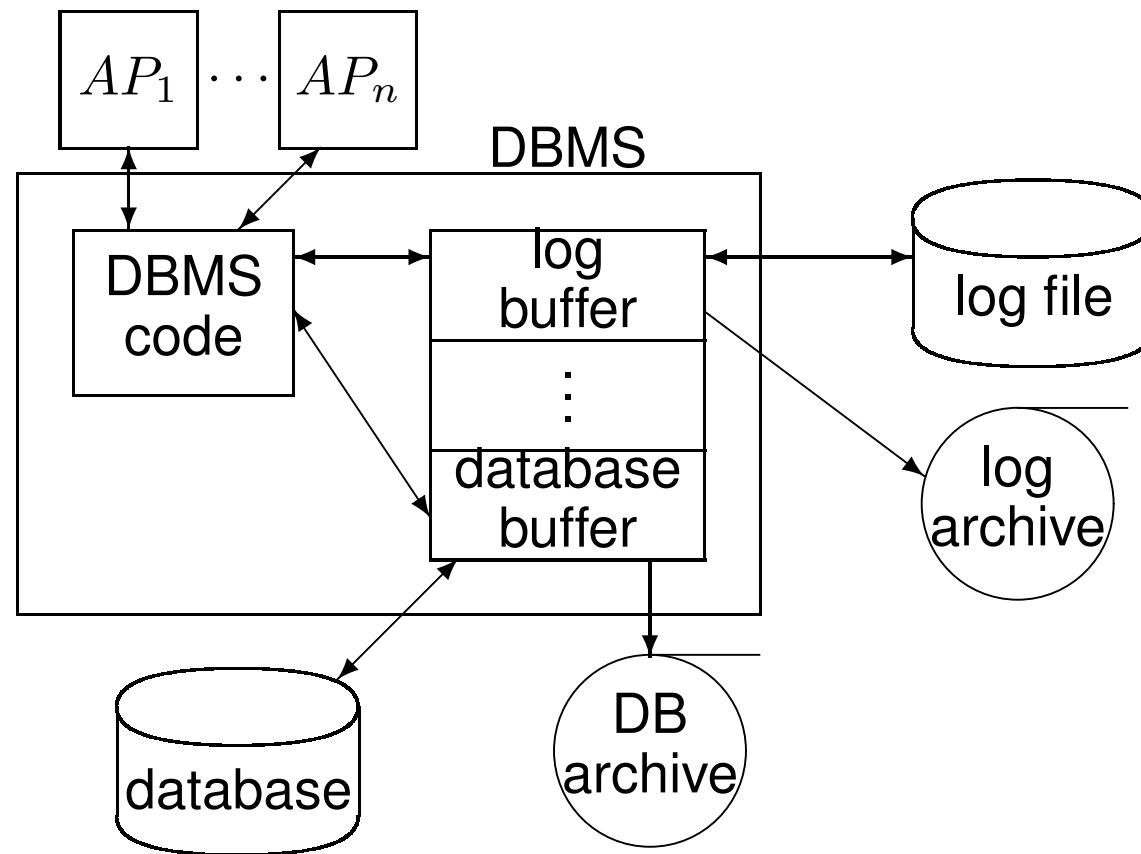
# System Configuration

- We will investigate how to make the most complicated case work (update-in-place):
  - steal
  - $\neg$ force
  - update-in-place
  - Fine granularity of locks

# ARIES Protocol

- The ARIES protocol is a well-known recovery protocol for DBMSs
- Uses log files to reconstruct database state after a crash
- A log file contains
  - Information on how to redo changes
  - Information on how to undo changes

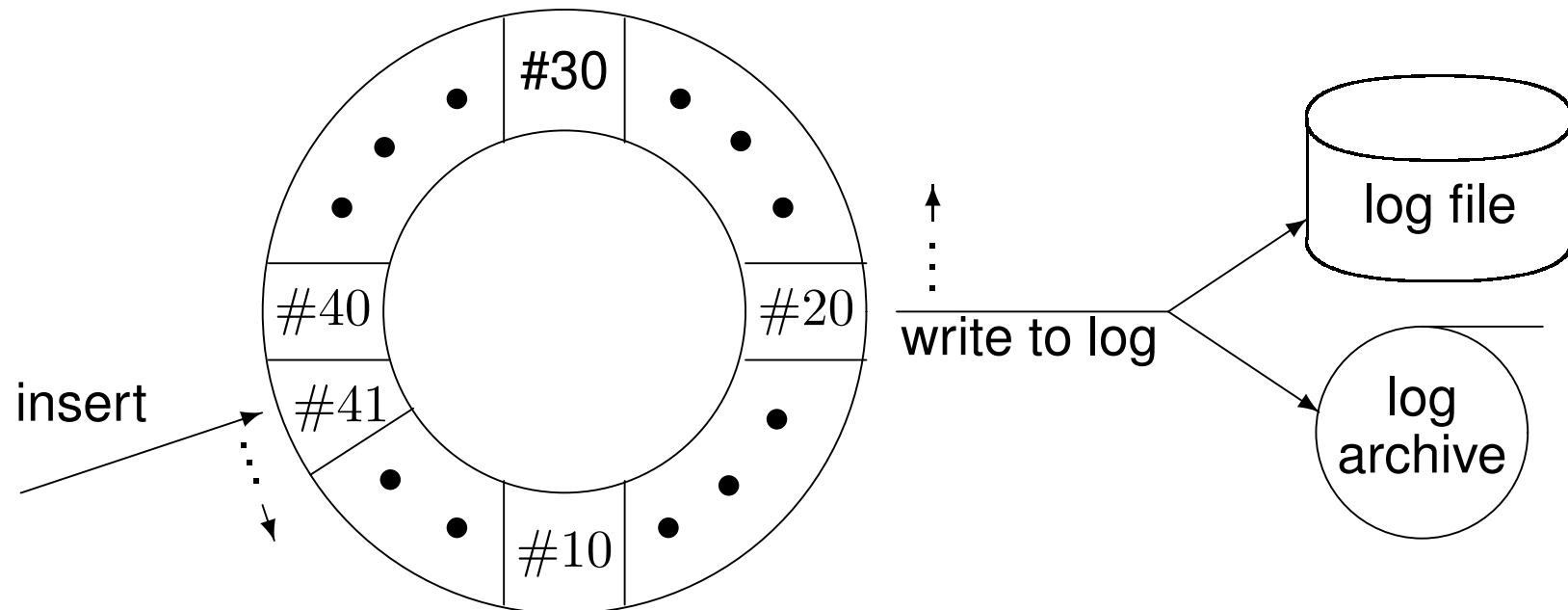
# Writing the Log Files



- The log data is written twice
  - Log file for fast access: R1, R2 and R3 recovery
  - Log archive: R4 recovery

# Writing the Log Files (2)

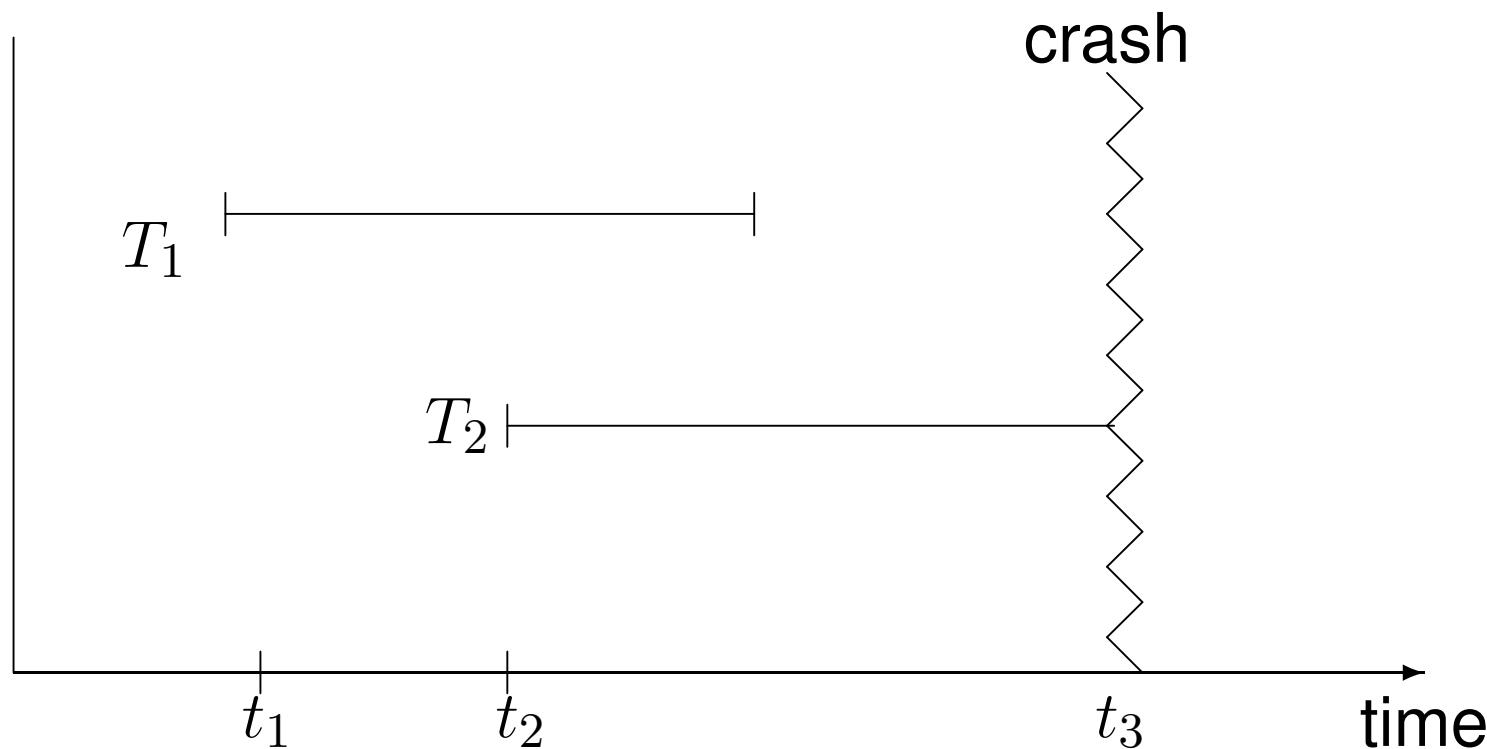
- Log buffer uses a ring buffer:



# Writing the Log Files (3)

- DBMSs use the principle of write-ahead-logging (WAL):
  - Before a TA is allowed to commit, its log records have to be made persistent
  - Before a modified page in main memory is written back to secondary storage
    - every log record associated with this page has to be written to the log file and archive

# Recovery after Failure

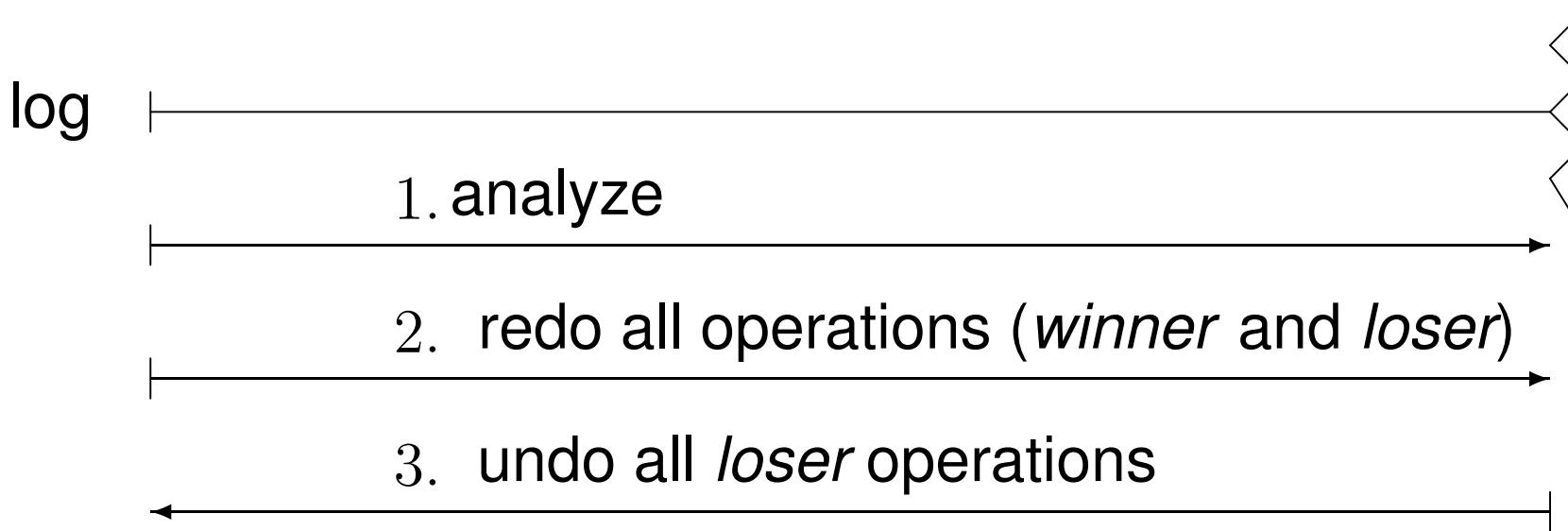


- TAs of type  $T_1$  are *winners*: have to be completely redone
- TAs of type  $T_2$  are *losers*: have to be completely undone

# Phases of Recovery

- *Analysis:*
  - Determine the set of *winners* (TAs of type  $T_1$ )
  - Determine the set of *losers* (TAs of type  $T_2$ )
- *Redo Phase:*
  - All logged operations are processed in the order they were originally executed
- *Undo Phase:*
  - The operations of loser TAs are undone in reverse order

# Phases of Recovery (2)



# Log Entries

[LSN,TA,PageID,Redo,Undo,PrevLSN]

- LSN (Log Sequence Number):
  - Every log entry has a unique identifier
  - LSNs have to be monotonically increasing
  - Determines the chronological order of the entries
- TA
  - ID of the transaction that has executed this operation

# Log Entries (2)

- Redo:
  - Physical logging: after image
  - Logical logging: code that produces after image given before image
- Undo:
  - Physical logging: before image
  - Logical logging: code that produces before image given after image

# Log Entries (3)

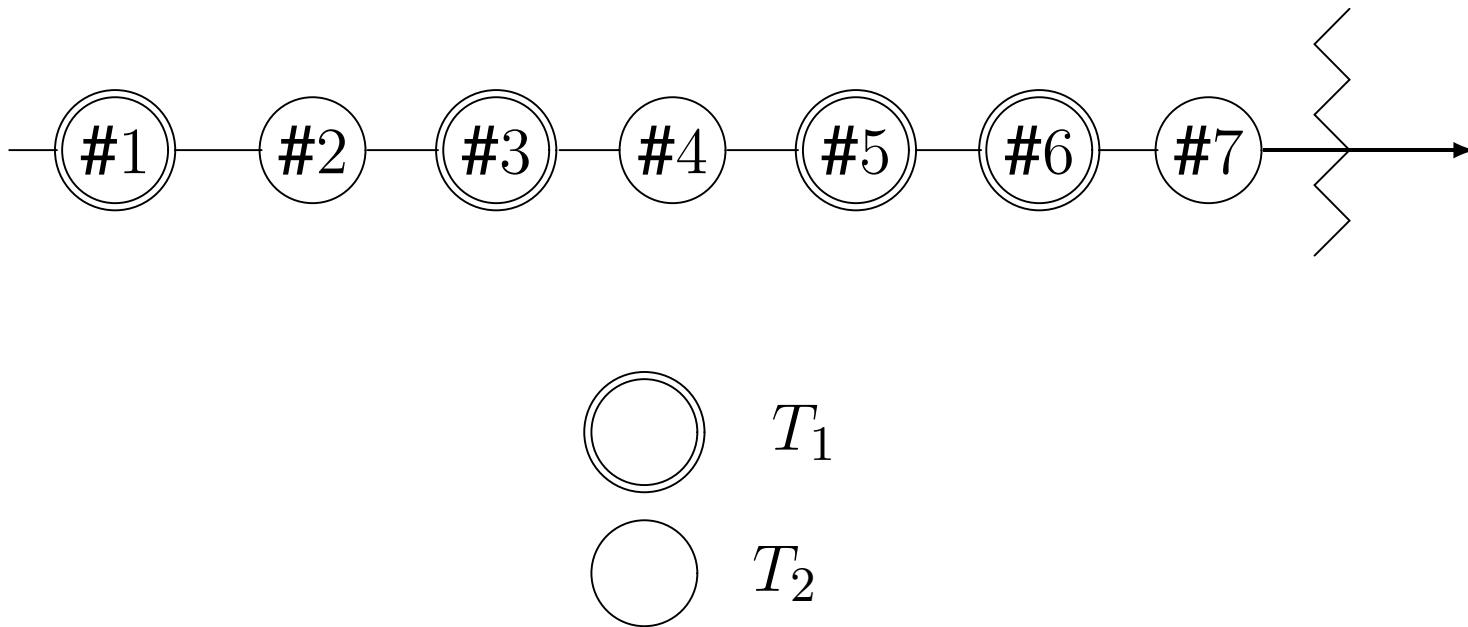
- PageID
  - ID of the page on which change was made
  - If a change affects more than one page, one log entry per page is needed
- PrevLSN
  - Pointer to previous log entry (of this TA)
  - Makes undo more efficient

# Example of a Log File

	$T_1$	$T_2$	Log [LSN,TA,PageID,Redo,Undo,PrevLSN]
1.	<b>BOT</b>		[#1, $T_1$ , <b>BOT</b> , 0]
2.	$r(A, a_1)$		[#2, $T_2$ , <b>BOT</b> , 0]
3.		<b>BOT</b>	
4.		$r(C, c_2)$	
5.	$a_1 := a_1 - 50$		[#3, $T_1$ , $P_A$ , $A-=50$ , $A+=50$ , #1]
6.	$w(A, a_1)$		
7.		$c_2 := c_2 + 100$	
8.		$w(C, c_2)$	[#4, $T_2$ , $P_C$ , $C+=100$ , $C-=100$ , #2]
9.	$r(B, b_1)$		
10.	$b_1 := b_1 + 50$		[#5, $T_1$ , $P_B$ , $B+=50$ , $B-=50$ , #3]
11.	$w(B, b_1)$		
12.	<b>commit</b>		[#6, $T_1$ , <b>commit</b> , #5]
13.		$r(A, a_2)$	
14.		$a_2 := a_2 - 100$	
15.		$w(A, a_2)$	[#7, $T_2$ , $P_A$ , $A-=100$ , $A+=100$ , #4]
16.	-----	<b>commit</b>	[#8, $T_2$ , <b>commit</b> , #7]

# Example of a Log File (2)

- Assume that the DBMS crashes after operation 15
  - That means,  $T_2$  does not successfully commit
- Then the log file shows the following picture:



# Robustness of Recovery

- What happens if DBMS crashes during recovery?
- We have to make sure that we are able to recover from an unsuccessful recovery
- We have to be able to do this an arbitrary number of times

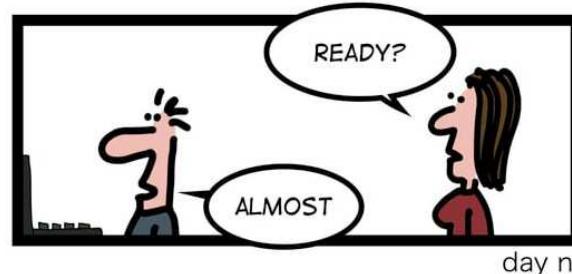
# Idempotence

- An arbitrary number of redos (and undos) has to have the same effect as a single one
- This is called *idempotence*:

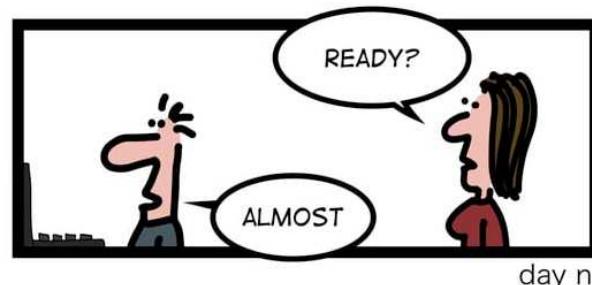
$$\text{undo}(\text{undo}(\dots (\text{undo}(a)) \dots)) = \text{undo}(a)$$

$$\text{redo}(\text{redo}(\dots (\text{redo}(a)) \dots)) = \text{redo}(a)$$

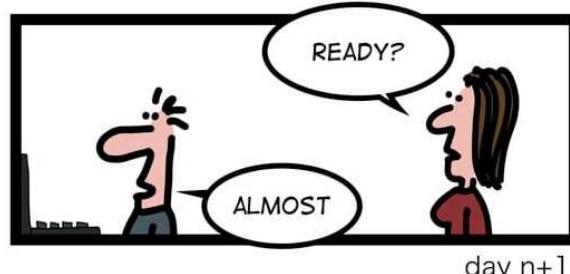
IDEMPOTENT



NON-IDEMPOTENT



Geek & Poke



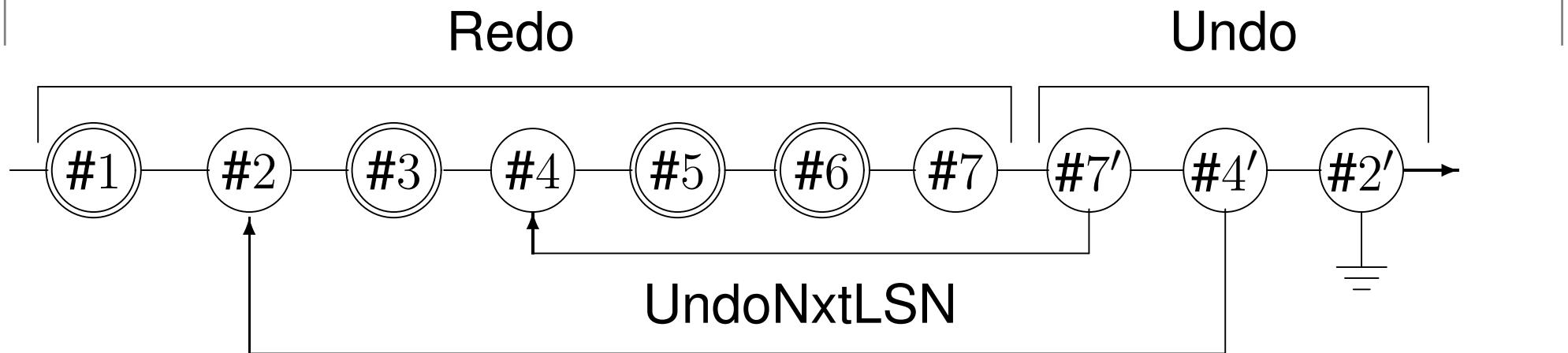
# Idempotence of Redo

- Doing this for Redo is not that hard:
  - Every time we make a change persistent, we write the corresponding LSN on the affected page
  - Before executing a Redo log entry, we check the LSN on the page
    - If the LSN on the page is greater than or equal to the LSN of the current log entry, we skip the redo
    - Otherwise we execute the redo part of the log entry...
    - ... writing its LSN on the page

# Idempotence of Undo

- Making undos idempotent is a bit harder:
  - However, we have a log, so why not use it during recovery?
  - We log every undo operation that we execute, creating a *compensation log record*
  - This little trick shifts the successfully executed undos into the redo phase
    - And we know how to make those idempotent...

# Idempotence of Undo (2)



- Compensation Log Records (CLRs) have an LSN marked with an '
- #7' is the CLR for #7
- #4' is the CLR for #4
- LSNs are still monotonically increasing (notation is used to make it easier to match CLRs)

# Log Entries after Recovery

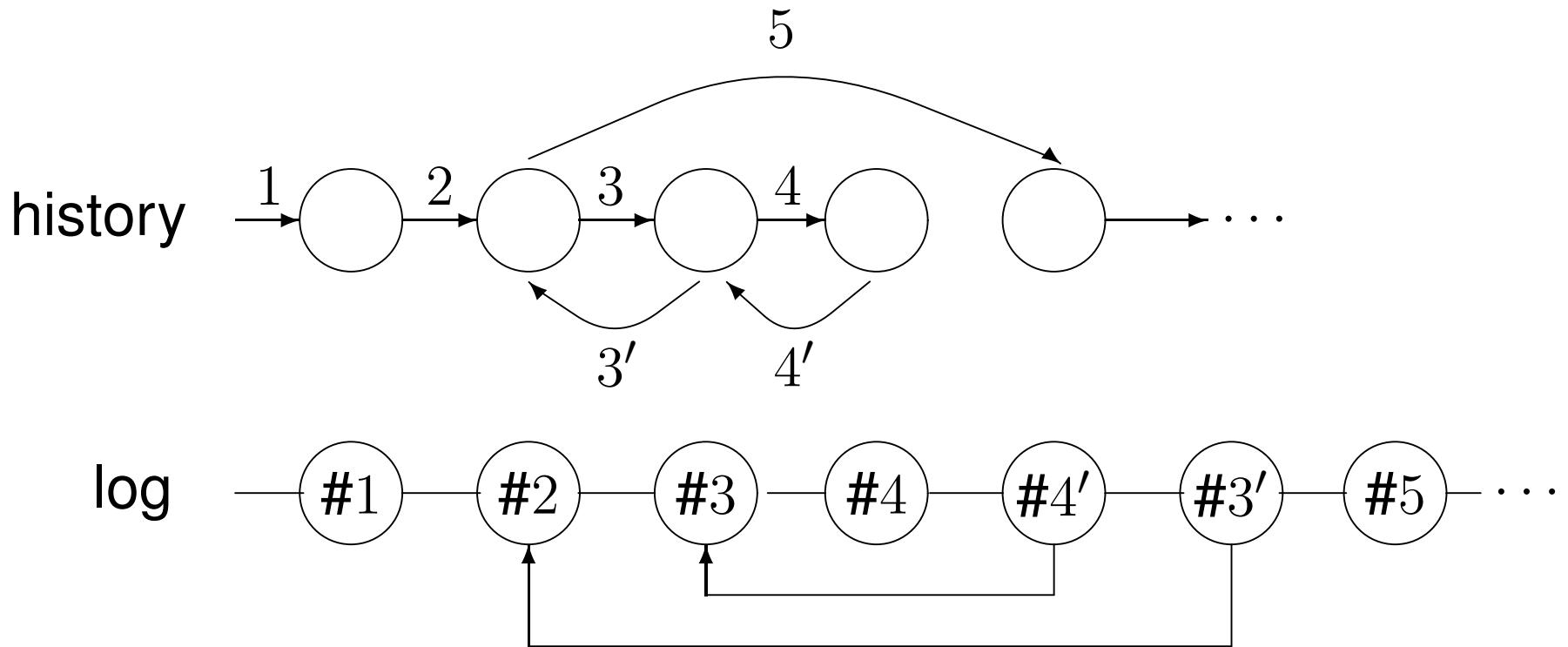
[#1,  $T_1$ , **BOT**, 0]  
[#2,  $T_2$ , **BOT**, 0]  
[#3,  $T_1$ ,  $P_A$ ,  $A-=50$ ,  $A+=50$ , #1]  
[#4,  $T_2$ ,  $P_C$ ,  $C+=100$ ,  $C-=100$ , #2]  
[#5,  $T_1$ ,  $P_B$ ,  $B+=50$ ,  $B-=50$ , #3]  
[#6,  $T_1$ , **commit**, #5]  
[#7,  $T_2$ ,  $P_A$ ,  $A-=100$ ,  $A+=100$ , #4]  
 $\langle \#7', T_2, P_A, A+=100, \#7, \#4 \rangle$   
 $\langle \#4', T_2, P_C, C-=100, \#7', \#2 \rangle$   
 $\langle \#2', T_2, -, -, \#4', 0 \rangle$

- We use angled brackets  $\langle \dots \rangle$  for CLRs

# CLRs

- CLRs look as follows:
  - LSN
  - TA identifier
  - affected page
  - Redo information
  - PrevLSN
  - UndoNxtLSN: pointer to the next operation that has to be undone

# Partial Undo

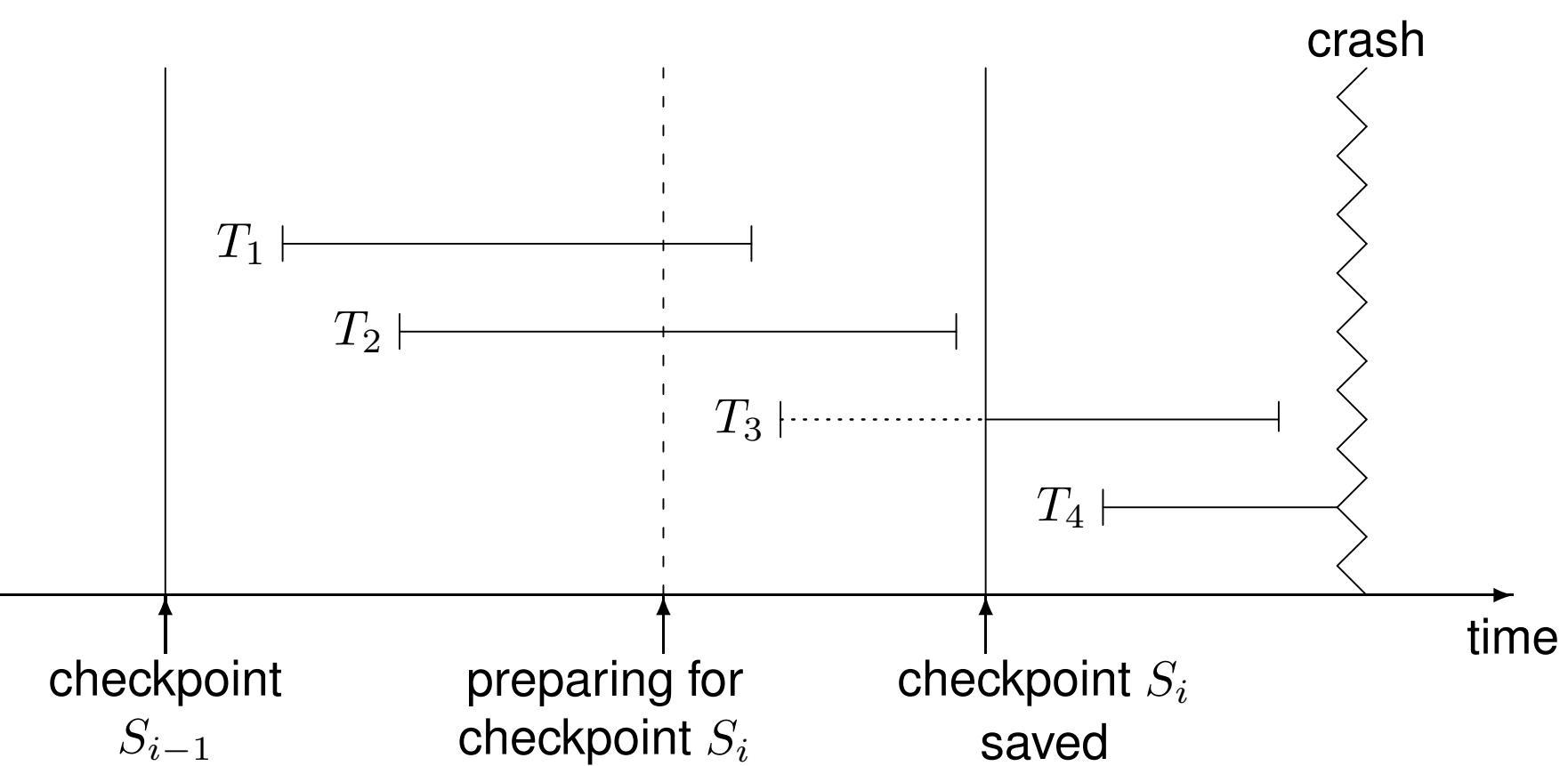


- Necessary for the implementation of savepoints within TAs
- Here operations 3 and 4 are undone

# Checkpoints

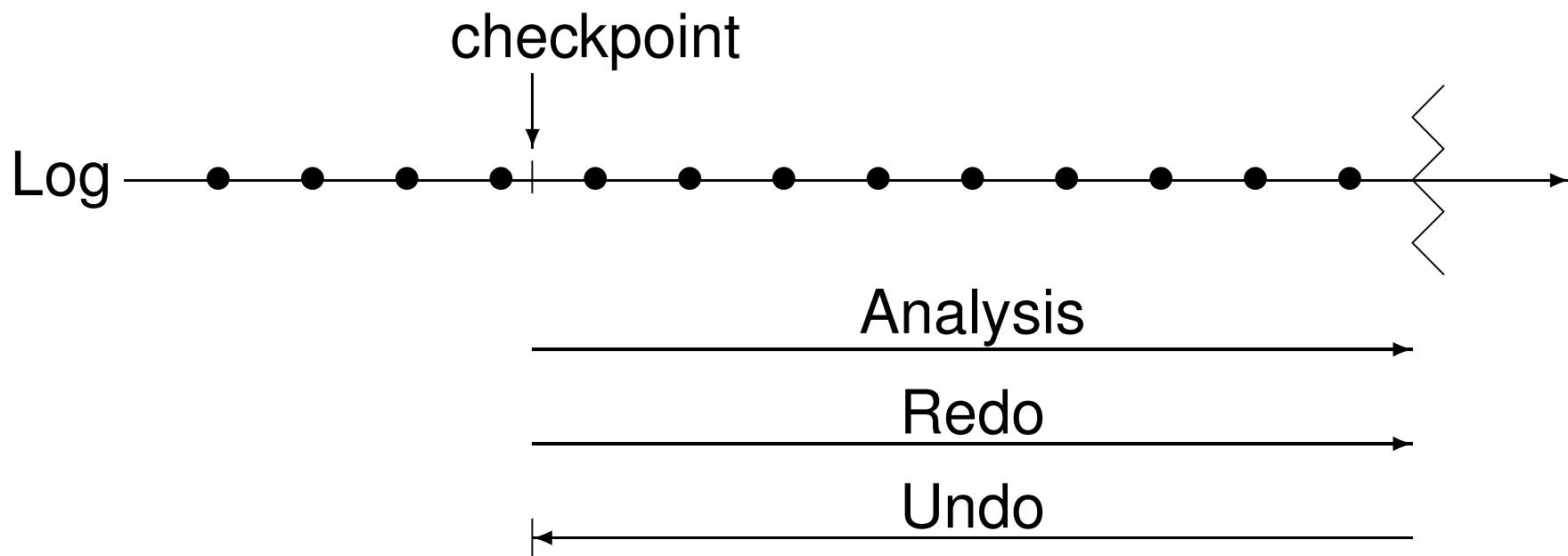
- The longer the DBMS runs, the longer the log files will become
- From time to time you should take a snapshot of the current state of the database and save it
- Otherwise you have to go through weeks worth of log files during recovery...

# Checkpoints (2)



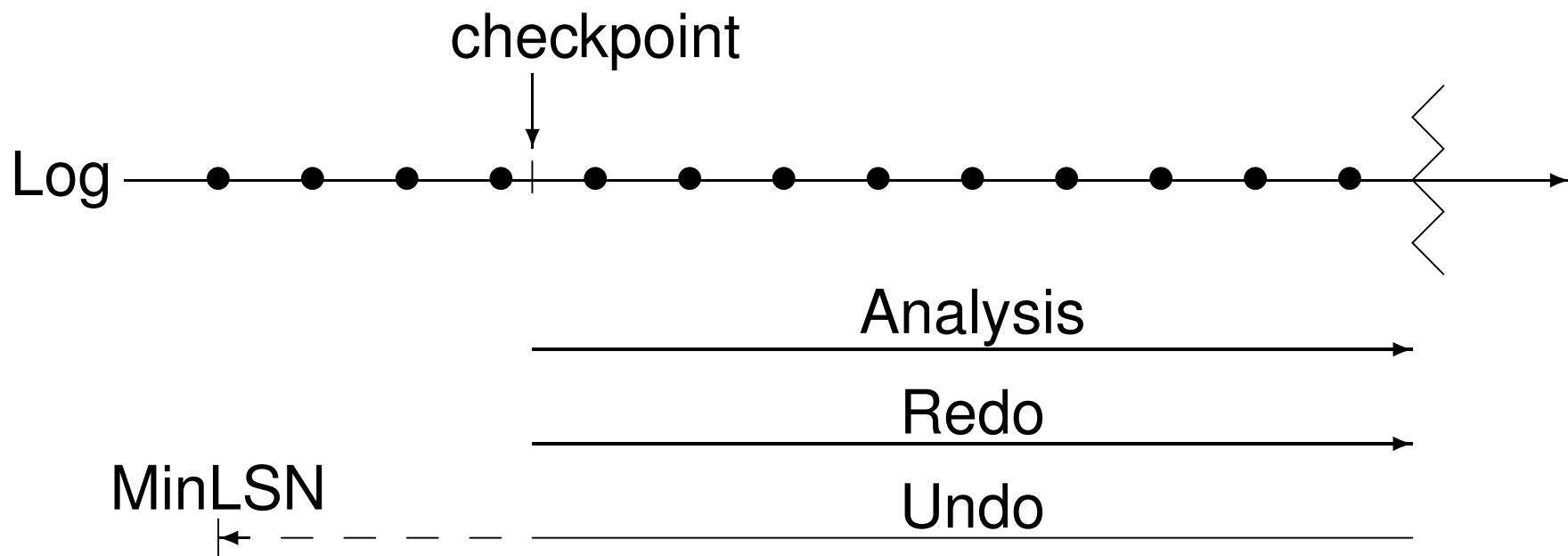
# Types of Checkpoints

- Transaction consistent:

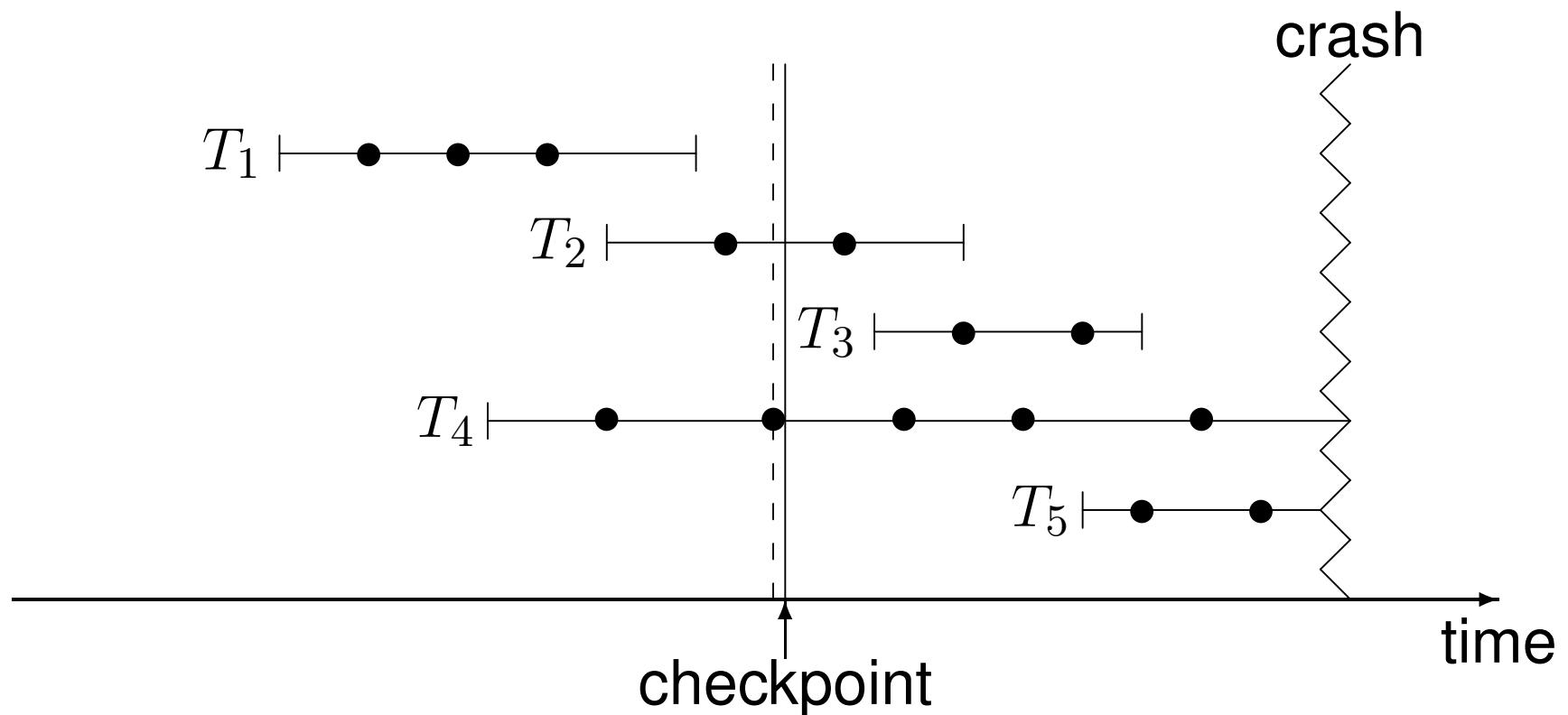


# Types of Checkpoints (2)

- Action consistent:

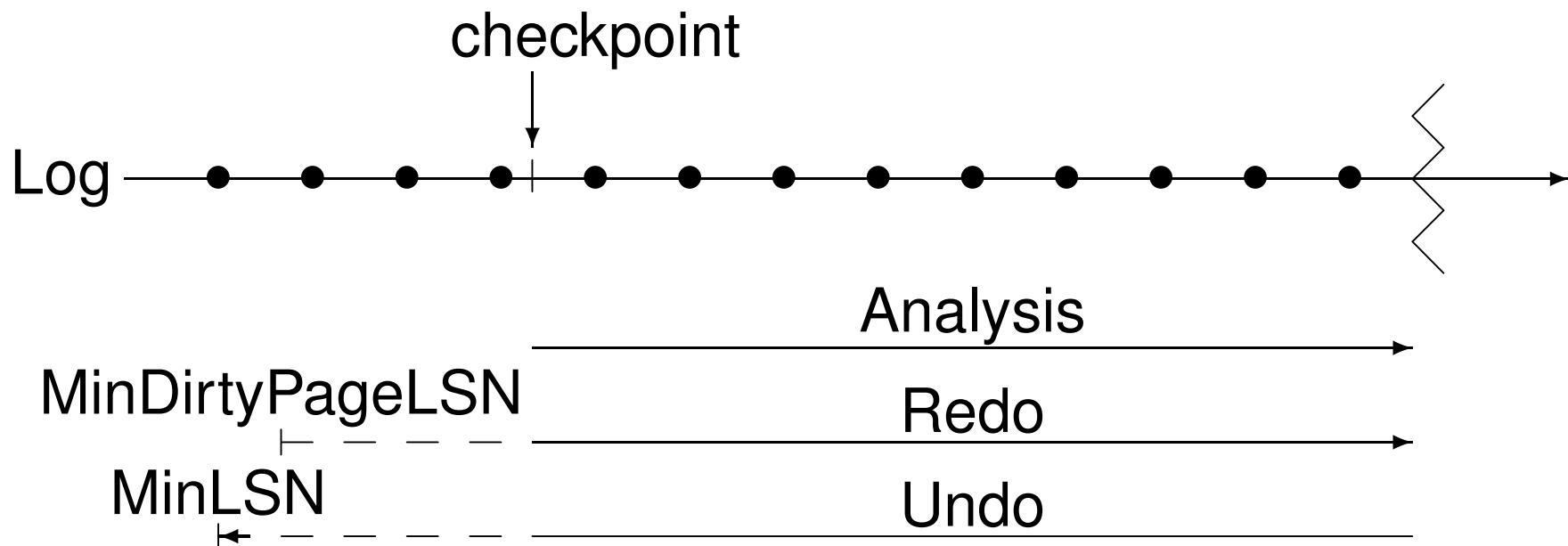


# Action Consistent Checkpoint



# Types of Checkpoints (3)

- Fuzzy:



# Fuzzy Checkpoint

- Modified pages in main memory are not written back to disk before taking the snapshot
- Only their page IDs are made persistent:
  - Dirty Pages = set of IDs of modified pages
- MinDirtyPageLSN: the smallest LSN whose operation has not been made persistent yet

# Summary

- The recovery component of a DBMS is responsible for restoring the state of the database just before a crash
- Two main techniques are
  - log files
  - checkpoints

# Chapter 11

## Multi-User-Synchronization

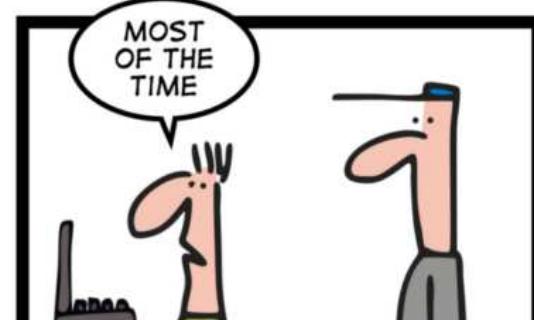
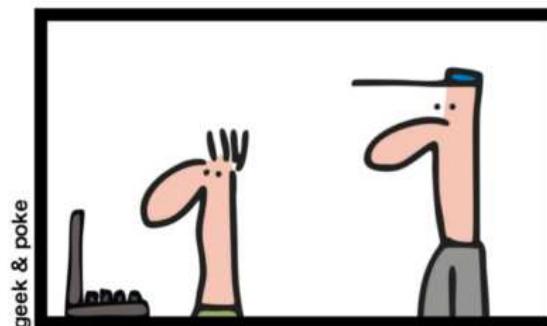
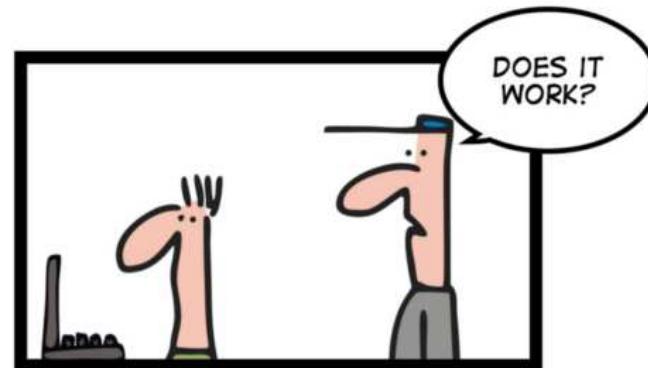
# Why Run TAs Concurrently?

- We could run all TAs serially (one after the other)
  - This would prevent all unwanted side effects
  - However, this would also be slow:
    - For example, a TA is waiting for data from a disk access or a user input
    - This TA would then block all the following ones
- Using concurrency, we could use a system's resources more efficiently

# Problems With Concurrency

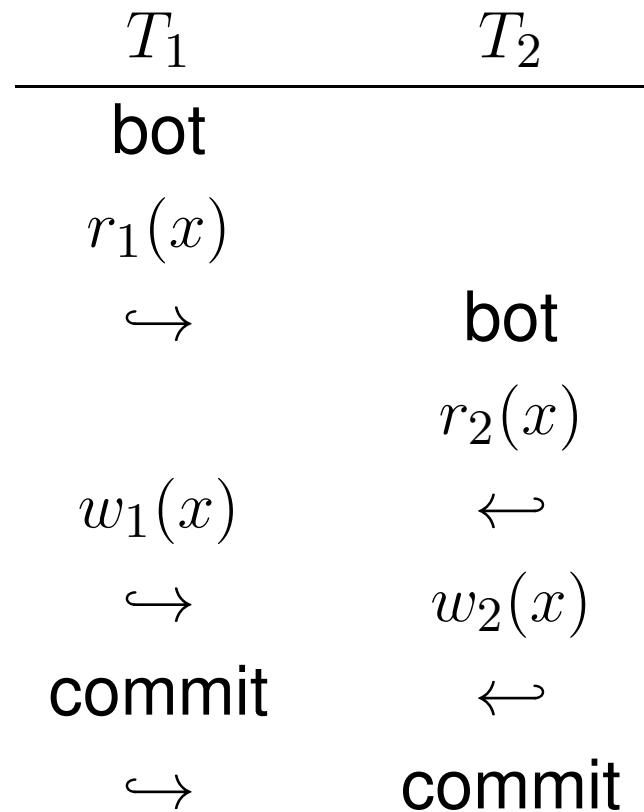
- Not controlling concurrency can lead to the following problems:
  - lost update
  - dirty read
  - non-repeatable read
  - phantom problem

SIMPLY EXPLAINED



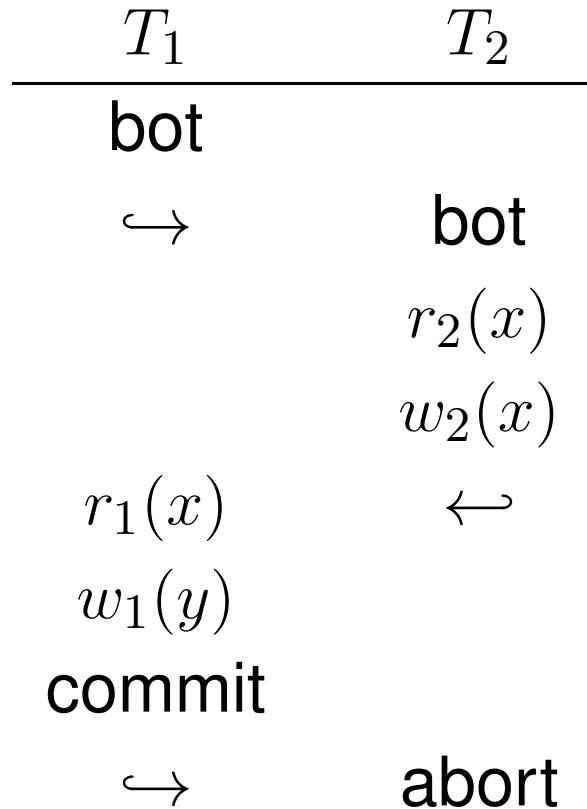
CONCURRENCY

# Lost Update



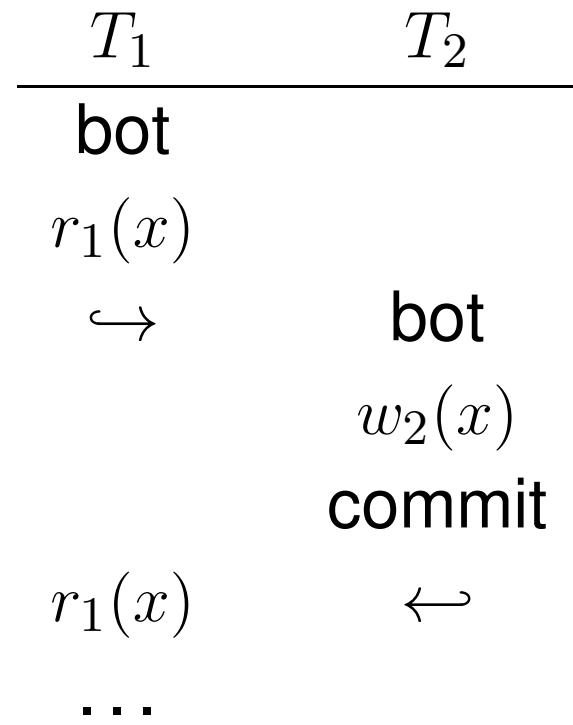
The update of  $T_1$  has been lost!

# Dirty Read



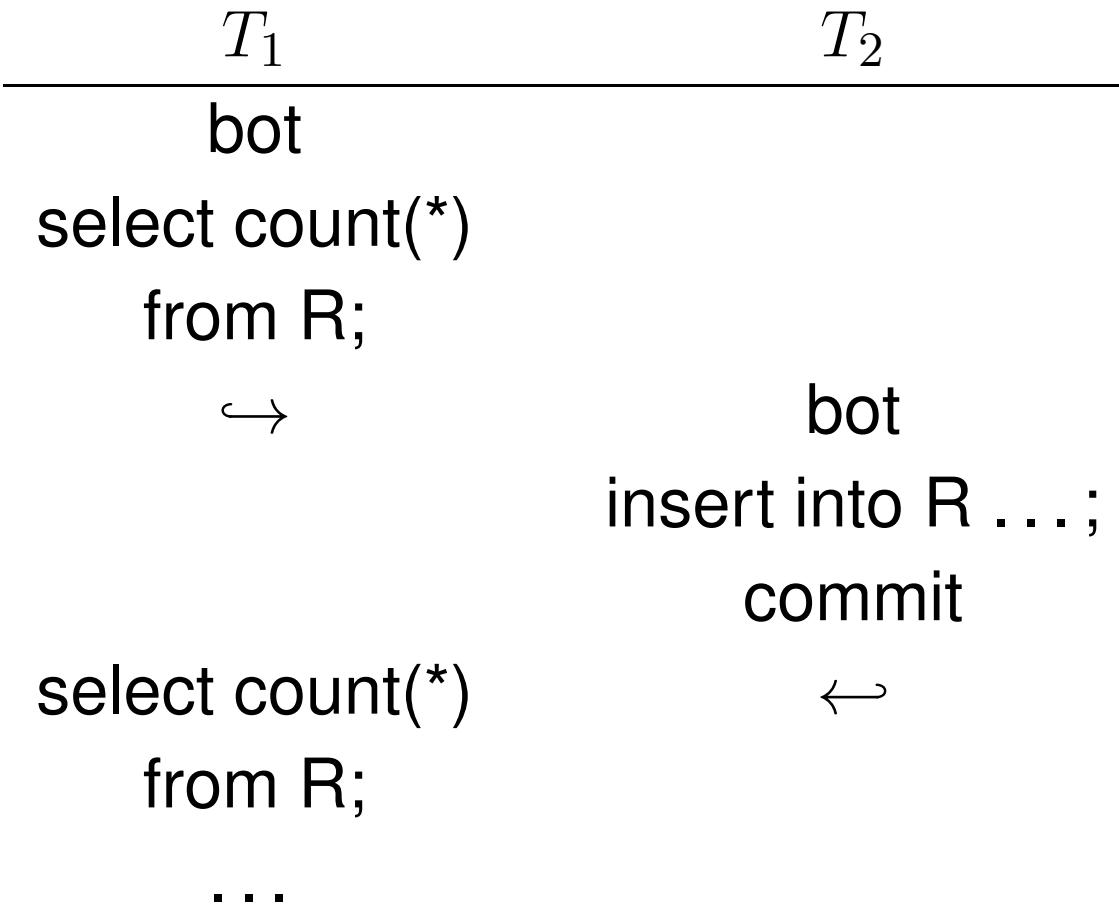
$T_1$  reads a value for  $x$  that is not valid!

# Non-Repeatable Read



$T_1$  reads  $x$  twice with different results!

# Phantom Problem



When running the second query,  $T_1$  finds a tuple that was not there the first time!

# Avoiding These Problems

- In the ideal case, all of these problems are avoided
- However, usually this involves a trade-off between performance and accuracy:
  - the safer, the slower
- Using *isolation levels* you can tell a DBMS which degree of safety you want



# Transactions and SQL

- Setting the parameters for a TA in SQL:

***set transaction level, access mode***

- Usually there are four isolation levels:
  - read uncommitted
  - read committed
  - repeatable read
  - serializable
- Possible access modes:
  - read only
  - read write

# Transactions and SQL (2)

- What does each level do?

	lost update	dirty read	nonrep. read	phant. probl.
read uncommitt.	✓			
read committed	✓	✓		
repeat. read	✓	✓	✓	
serializable	✓	✓	✓	✓

# Transactions and SQL (3)

- “read only” means that a TA will only read data items
- This has a significant impact on performance:
- Concurrent reads do not cause any problems
  - You can have an arbitrary number of concurrent reads without any interference
- As soon as a TA is writing data items, you have to take precautions

# Transactions and SQL (4)

- Marking the beginning of a TA:

**start transaction;**

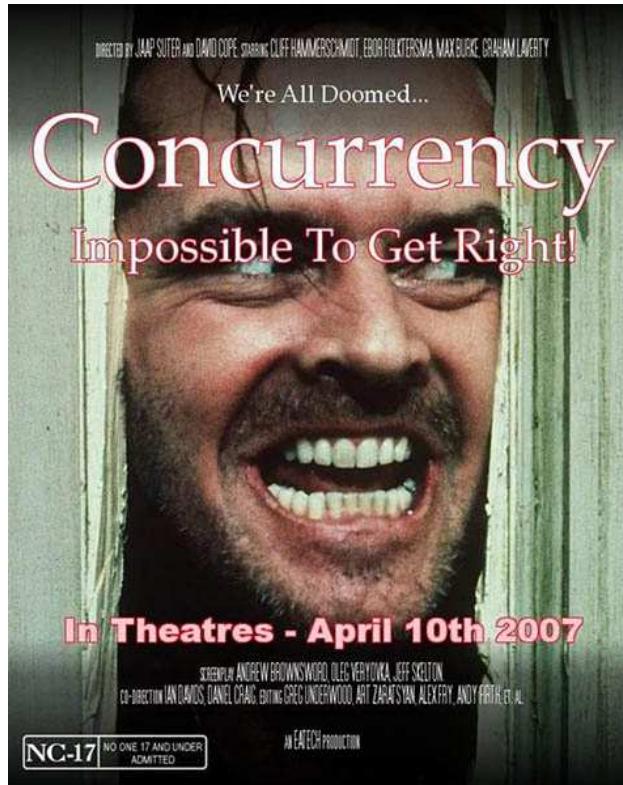
- (Successfully) ending a TA:

**commit [work];**

- Aborting a TA:

**rollback [work];**

# DBMSs and Concurrency



- How do we solve the problems caused by concurrency?
- First of all, we are going to define the problems more accurately...
- ...and then show mechanisms used in a DBMS

# Formal Definition of a TA

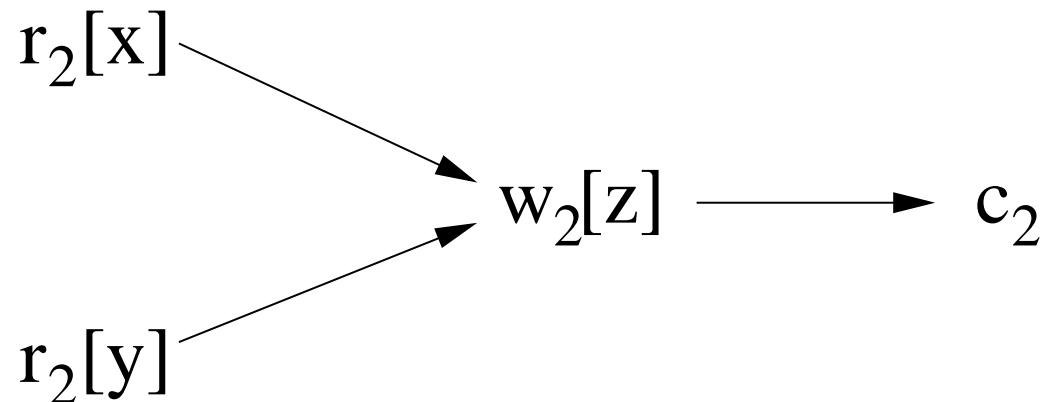
- We allow a TA  $T_i$  to execute the following operations:
  - $r_i(A)$ : reading data item  $A$
  - $w_i(A)$ : writing data item  $A$
  - $a_i$ : abort
  - $c_i$ : commit
- $bot$ : begin of transaction (implicit)

# Formal Definition of a TA (2)

- A  $T_i$  is a partial order of operations with the binary relation  $<_i$  such that
  - $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data item}\} \cup \{a_i, c_i\}$
  - $a_i \in T_i$ , iff  $c_i \notin T_i$
  - Let  $t_i$  be equal to  $a_i$  or  $c_i$ , then for every other operation  $p_i$ :  $p_i <_i t_i$
  - If  $r_i[x]$  and  $w_i[x] \in T_i$ , then either  $r_i[x] <_i w_i[x]$  or  $w_i[x] <_i r_i[x]$

# Visualization

- Transactions are often depicted as directed acyclic graphs (DAGs):



- $r_2[x] <_2 w_2[z]$ ,  $w_2[z] <_2 c_2$ ,  $r_2[x] <_2 c_2$ ,  $r_2[y] <_2 w_2[z]$ ,  
 $r_2[y] <_2 c_2$
- In the graph transitive relations are implicit

# Schedules

- We can execute a number of TAs concurrently
- This is described with the help of a *schedule*
- A schedule describes the relative order of operations from different TAs
- Because operations can be executed in parallel, a schedule is also a partial order



# Conflicts

- If operations conflict, they may not be executed in parallel
- Two operations conflict, if both of them access the same data item and at least one of them is a write

		$T_i$
$T_j$	$r_i[x]$	$w_i[x]$
$r_j[x]$		$\not\in$
$w_j[x]$	$\not\in$	$\not\in$

# Formal Definition of Schedules

- $T = \{T_1, T_2, \dots, T_n\}$  is a set of TAs
- A schedule  $H$  over  $T$  is a partial order with the binary relation  $<_H$  such that
  - $H = \bigcup_{i=1}^n T_i$
  - $<_H \supseteq \bigcup_{i=1}^n <_i$
  - For two conflicting operations  $p, q \in H$  the following has to hold:  
either  $p <_H q$  or  $q <_H p$

# Example of a Schedule

- The following schedule combines the TAs  $T_1$ ,  $T_2$ , and  $T_3$

$$H = \begin{array}{ccccccc} r_2[x] & \rightarrow & w_2[y] & \rightarrow & w_2[z] & \rightarrow & c_2 \\ & \uparrow & & \uparrow & & \uparrow & \\ r_3[y] & \rightarrow & w_3[x] & \rightarrow & w_3[y] & \rightarrow & w_3[z] \rightarrow c_3 \\ & & \uparrow & & & & \\ r_1[x] & \rightarrow & w_1[x] & \rightarrow & & & c_1 \end{array}$$

# (Conflict-) Equivalence

- Two schedules  $H$  and  $H'$  are (*conflict-*)*equivalent* ( $H \equiv H'$ ), if
  - they contain the same set of TAs (with the same operations)
  - They order the conflicting operations of the unaborted TAs in the same way
- Main idea: ordering conflicting operations in the same way will give us the same result

# Example

- The following example shows some schedules that are equivalent and one that is not:

$$\begin{array}{ll} r_1[x] \rightarrow w_1[y] \rightarrow r_2[z] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \\ \equiv \quad r_1[x] \rightarrow r_2[z] \rightarrow w_1[y] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \\ \equiv \quad r_2[z] \rightarrow r_1[x] \rightarrow w_1[y] \rightarrow w_2[y] \rightarrow c_2 \rightarrow c_1 \\ \not\equiv \quad r_2[z] \rightarrow r_1[x] \rightarrow w_2[y] \rightarrow w_1[y] \rightarrow c_2 \rightarrow c_1 \end{array}$$

# Serializability

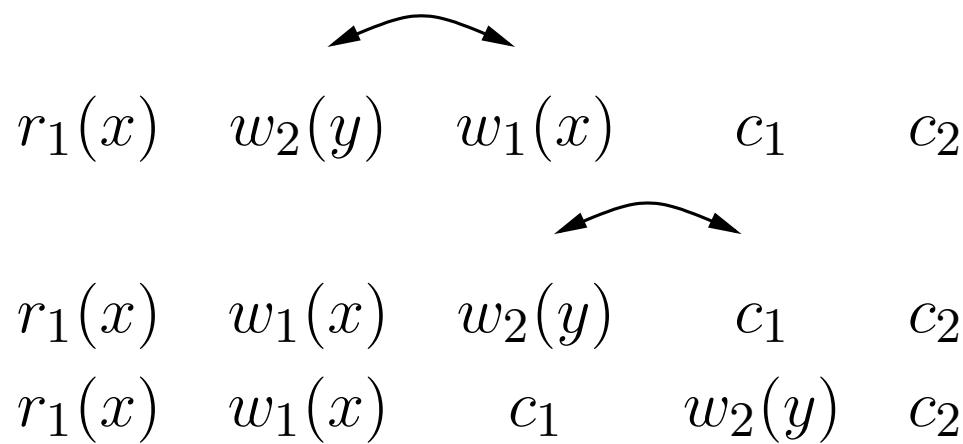
- We know that serial schedules cause no problems
- It would be great to have schedules with similar properties
- We would like to have a schedule that is equivalent to a serial one:
  - Such a schedule is called *serializable*

# Serializability (2)

- More precise definition:
  - The *closed projection*  $C(H)$  of a schedule  $H$  only contains the successfully committed TAs
  - A schedule is serializable, if  $C(H)$  is equivalent to a serial schedule  $H_s$
  - To be even more precise: because we look at conflict-equivalence, this is also called conflict-serializable
  - There are other notions of serializability

# Checking Serializability

- How do you check serializability?
- One way would be to swap operations (without changing the conflict order) in a schedule until we arrive at a serial schedule:



- This is a bit tedious and how do we prove that we are not able to do this?

# Serializability Graph

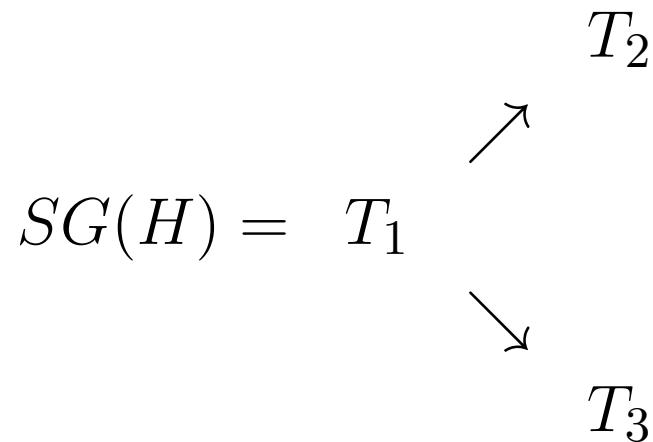
- Theorem: a schedule is serializable, iff its serializability graph is acyclic
- The *serializability graph*  $SG(H)$  of a schedule  $H = \{T_1, \dots, T_n\}$  is a directed graph with the following properties:
  - Successfully committed TAs from  $H$  are the nodes
  - We add an edge from  $T_i$  to  $T_j$ , if there are conflict operations  $p_i$  and  $q_j$  with  $p_i <_H q_j$

# Example

- Schedule  $H$ :

$w_1[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow r_2[x] \rightarrow r_3[y] \rightarrow w_2[x] \rightarrow c_2 \rightarrow w_3[y] \rightarrow c_3$

- Serializability graph  $SG(H)$ :



# Example (2)

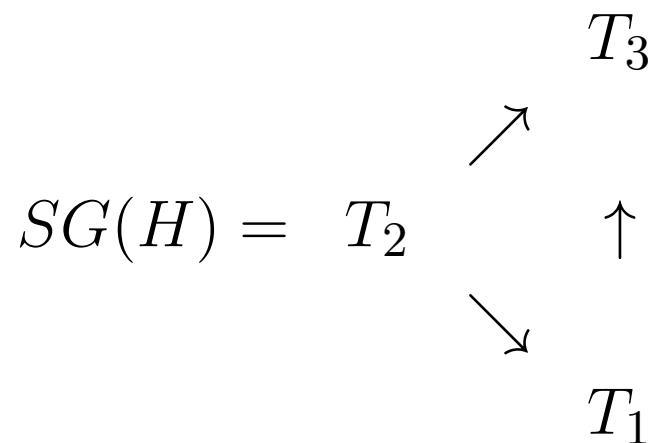
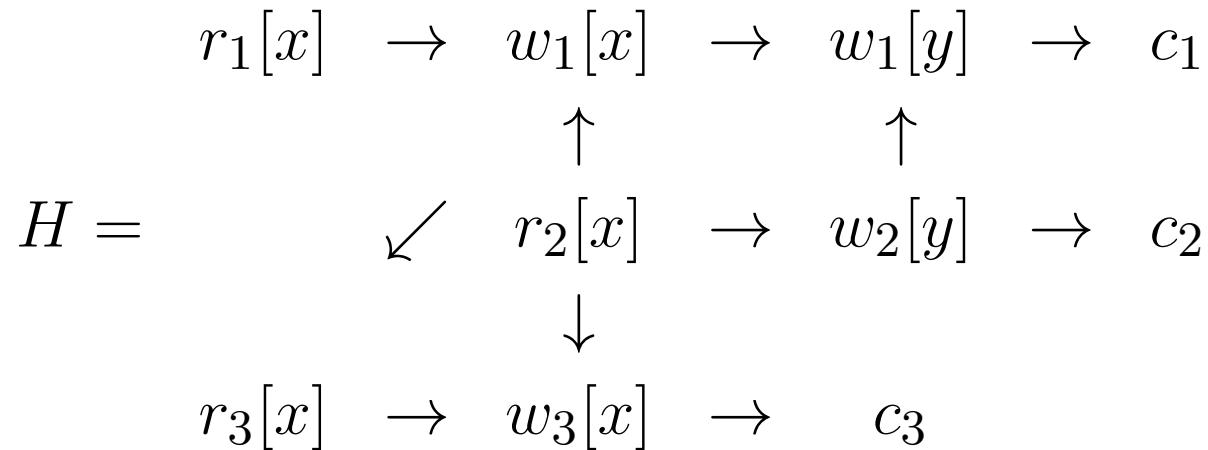
- $H$  is serializable
- Possible serial orders:

$$H_s^1 = T_1 \mid T_2 \mid T_3$$

$$H_s^2 = T_1 \mid T_3 \mid T_2$$

$$H \equiv H_s^1 \equiv H_s^2$$

# Example (3)



# Example (4)

- $H$  is serializable
- Possible serial orders:

$$H_s^1 = T_2 \mid T_1 \mid T_3$$

$$H \equiv H_s^1$$

# Example (5)

$$H = \begin{array}{ccccccc} w_1[x] & \rightarrow & w_1[y] & \rightarrow & c_1 \\ \uparrow & & \downarrow & & \\ r_2[x] & \rightarrow & w_2[y] & \rightarrow & c_2 \end{array}$$

$$SG(H) = T_1 \iff T_2$$

- $H$  is not serializable

# Other Properties

- Serializability is a very important property
- Nevertheless, there are some other useful properties:
  - Recoverability
  - Avoiding cascading aborts: ACA
  - Strictness

# Other Properties (2)

- Before defining these properties, we have to define the concept of a *reads-from relationship*
- A TA  $T_i$  reads (a data item  $x$ ) from TA  $T_j$ , if
  - $w_j[x] < r_i[x]$
  - $a_j \not\in r_i[x]$
  - If there is a  $w_k[x]$  with  $w_j[x] < w_k[x] < r_i[x]$ , then  $a_k < r_i[x]$
- A TA can read from itself

# Recoverability

- A schedule is *recoverable*,
  - if a TA  $T_i$  reads from another TA  $T_j$  ( $i \neq j$ ) and  $c_i \in H$ , then  $c_j < c_i$
- TAs have to follow a certain commit order
- If schedules are not recoverable, we get problems with C and D of the ACID paradigm

# Recoverability (2)

$$H = w_1[x] \ r_2[x] \ w_2[y] \ c_2 \ a_1$$

- $H$  is not recoverable
- This has severe consequences:
  - If we keep all the results, then the database is in an inconsistent state
    - $T_2$  has read data from an aborted TA
  - If we undo the changes made by  $T_2$ , we basically abort an already successfully committed TA

# Cascading Aborts

step	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
0.	...				
1.	$w_1[x]$				
2.		$r_2[x]$			
3.			$w_2[y]$		
4.				$r_3[y]$	
5.				$w_3[z]$	
6.					$r_4[z]$
7.					$w_4[v]$
8.					$r_5[v]$
9.	$a_1$ ( <b>abort</b> )				

# Cascading Aborts (2)

- A schedule *avoids cascading aborts*, if the following holds:
  - If a TA  $T_i$  reads from another TA  $T_j$  ( $i \neq j$ ), then  $c_j < r_i[x]$
- TAs are only allowed to read from already successfully committed TAs

# Strictness

- A schedule is *strict*, if it satisfies the following property:
  - Given two operations  $w_j[x] < o_i[x]$  (with  $o_i[x] = r_i[x]$  or  $w_i[x]$ ) we either have  $a_j < o_i[x]$  or  $c_j < o_i[x]$
- A TA may only read from or overwrite data items of successfully committed TAs

# Strictness (2)

- You may only use physically logging (for recovery) if you have strict schedules

$x = 0$

$w_1[x, 1]$  before image of  $T_1$ : 0

$x = 1$

$w_2[x, 2]$  before image of  $T_2$ : 1

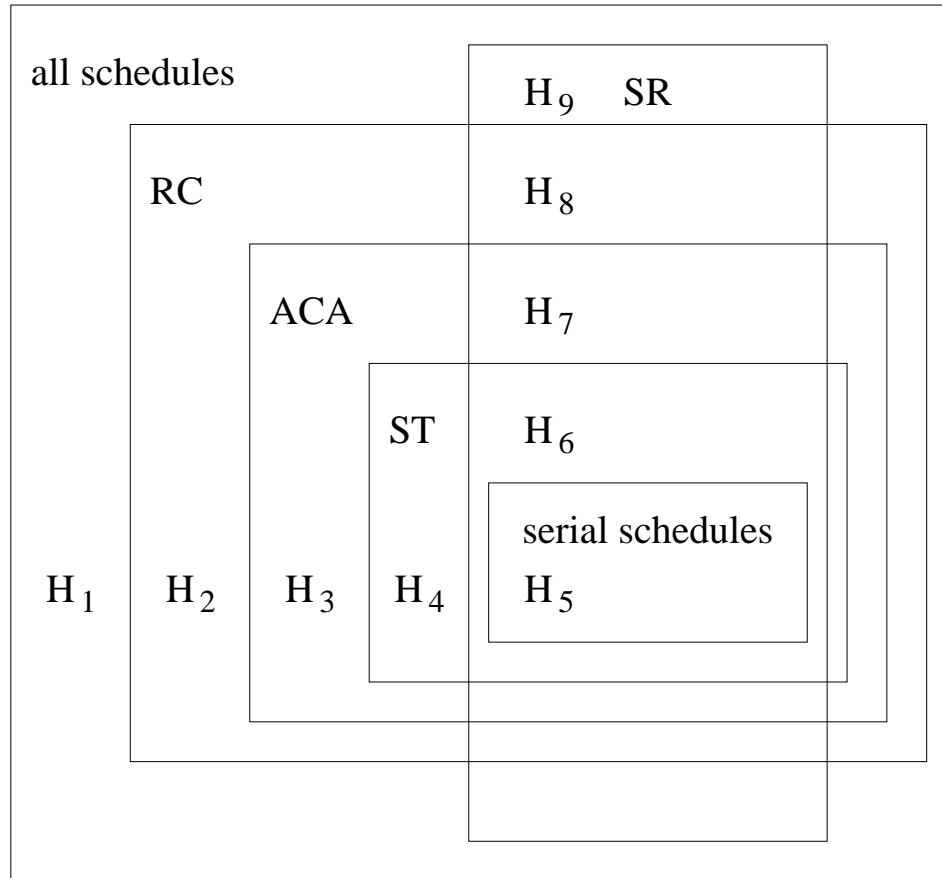
$x = 2$

$a_1$

$c_2$

- When  $T_1$  aborts,  $x$  will be set to 0

# Overview



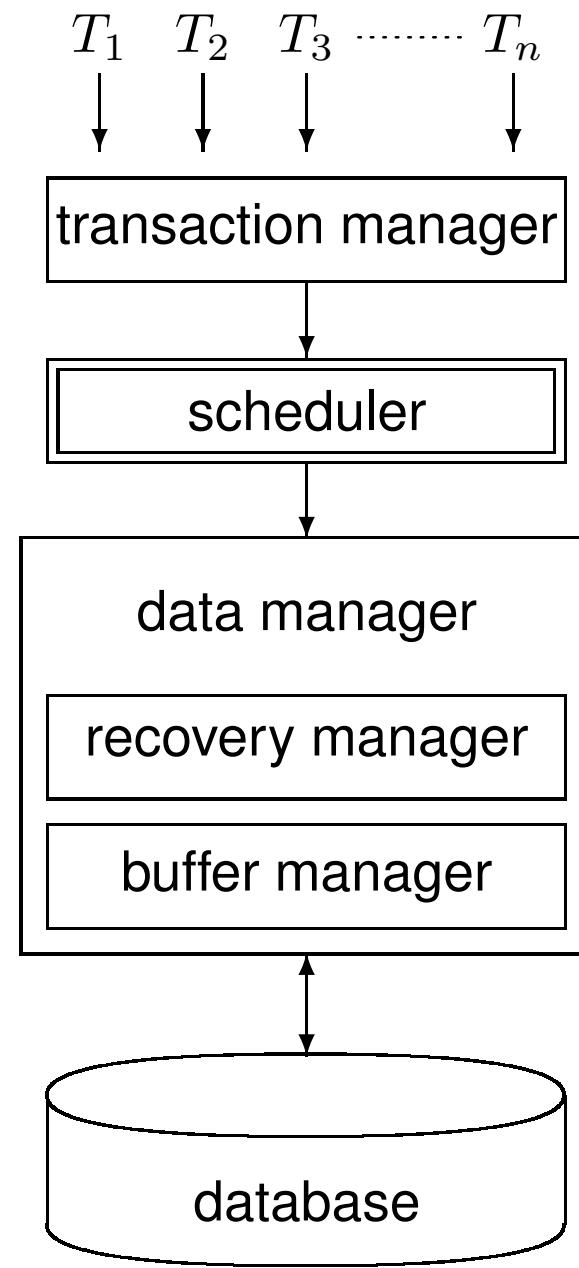
SR: serializable, RC: recoverable, ACA: avoids cascading aborts, ST: strict

# From Theory to Practice

- Now that we have covered the theoretical foundations...
- ...we are going to have a look at the implementation side



# Database Scheduler



# Database Scheduler (2)

- A *scheduler* is a program that orders the arriving operations to make sure that the resulting schedules are at least
  - serializable
  - recoverable
- A scheduler has different options after receiving an operation:
  - execute (immediately)
  - reject
  - delay

# Database Scheduler (3)

- There are two main strategies:
  - Pessimistic
  - Optimistic



# Pessimistic Scheduler

- A pessimistic scheduler usually delays operations
- If it has received a sufficient number of operations, it tries to order/execute them
- Example: lock-based scheduler



# Optimistic Scheduler

- An optimistic scheduler tries to execute operations as soon as possible
- May have to repair the damage later on
- Example: timestamp-based scheduler



# Lock-based Synchronization

- Main idea is relatively simple:
  - Every data item has a corresponding lock
  - Before a TA  $T_i$  may access a data item, it has to acquire its lock
  - Only one TA can hold a lock at a given time
  - If another TA  $T_j$  is currently holding the lock
    - then  $T_i$  is not able to acquire the lock
    - $T_i$  has to wait until  $T_j$  gives up the lock
- How can we guarantee serializability using locks?

# Two-Phase Locking

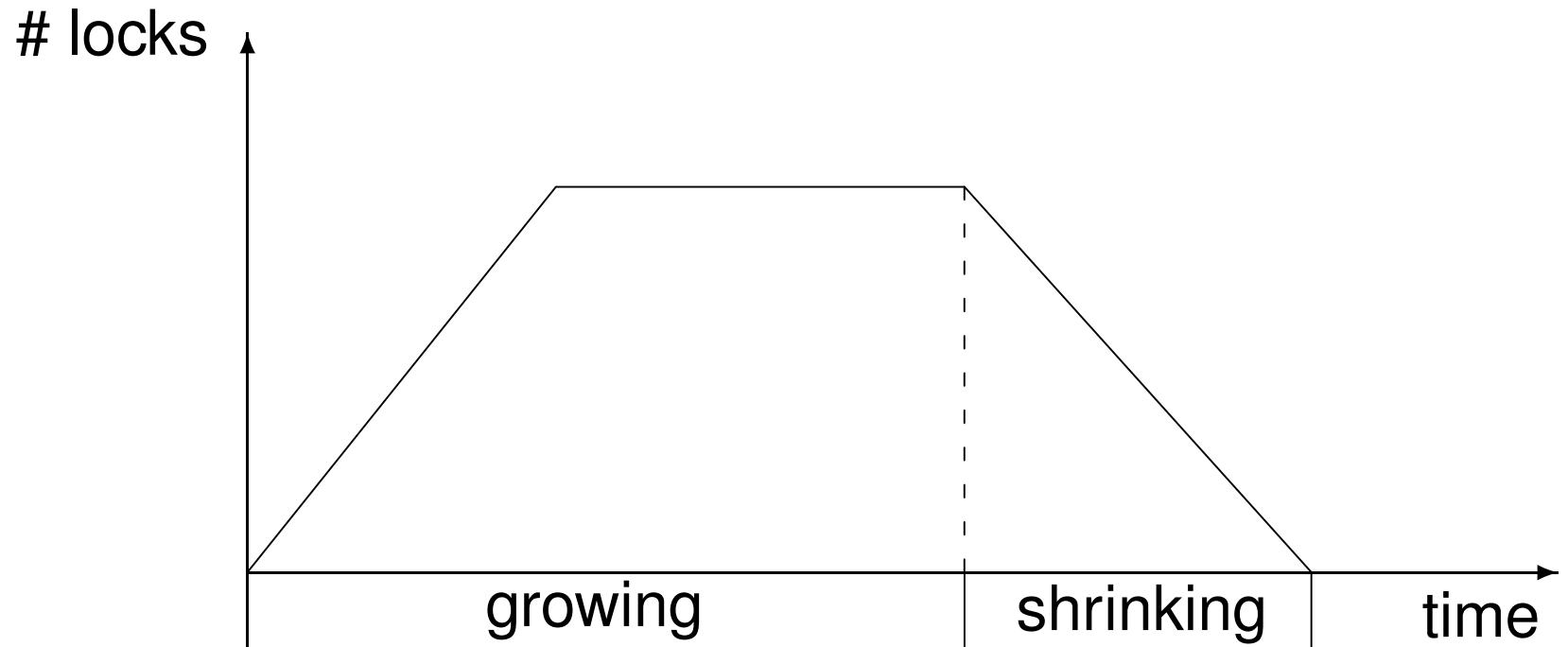
- A well-known lock-based protocol is *two-phase locking* (2PL)
- 2PL uses two different types of locks:  
Two modes of locking:
  - $S$  (shared, read lock)
  - $X$  (exclusive, write lock)
- A lock-compatibility matrix describes their relationships:

		lock held		
		none	$S$	$X$
lock request				
$S$		✓	✓	—
$X$		✓	—	—

# Two-Phase Locking (2)

- If a TA wants to access a data item, it has to lock this data item
- A TA can only request a lock once
- If a lock cannot be granted (due to a conflict), then the requesting TA has to wait
- Once a TA has started releasing locks, it may not obtain further locks
  - There are two phases: a growing phase and a shrinking phase
- When a TA finishes, it has to release all locks

# Two Phases of 2PL



- growing phase: locks are obtained, but none are released
- shrinking phase: locks are released, but none are obtained

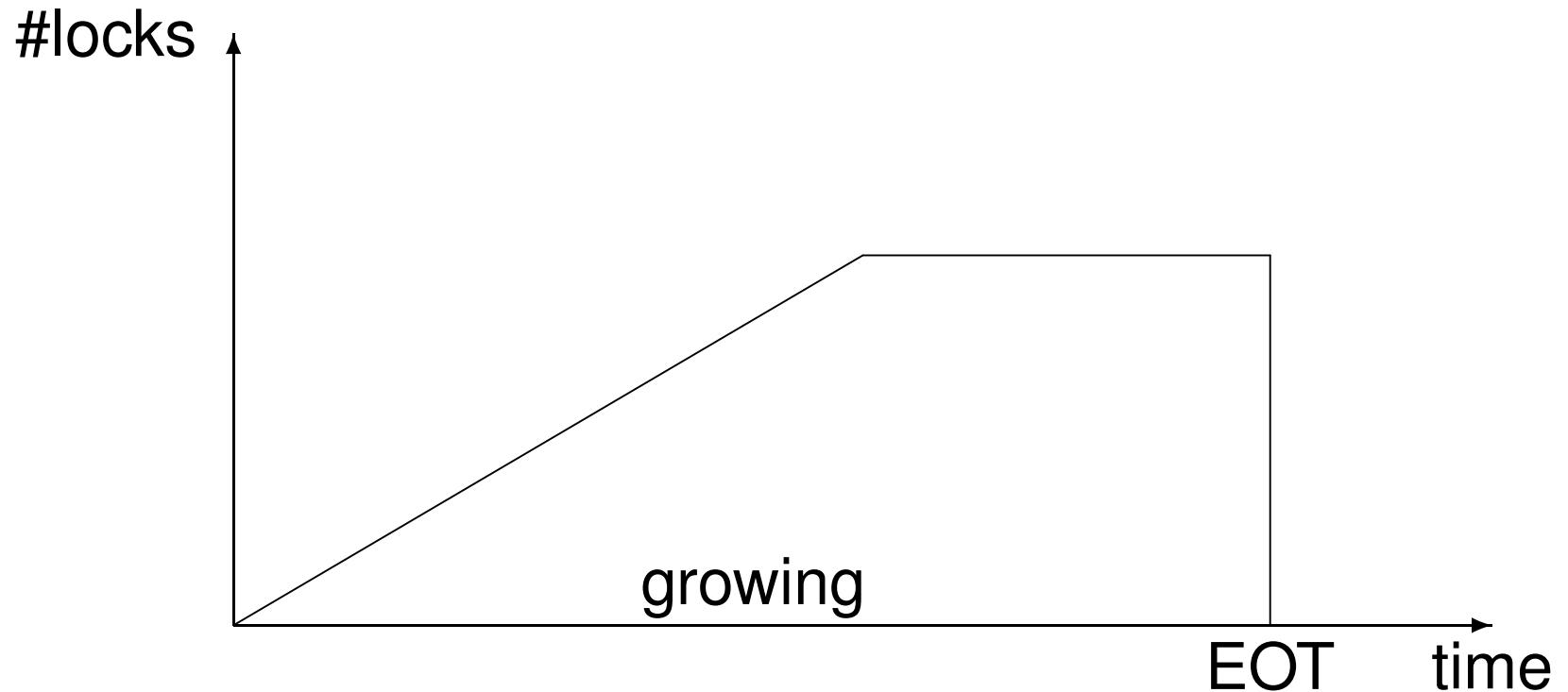
# Interleaving TAs with 2PL

step	$T_1$	$T_2$	comment
1.	<b>BOT</b>		
2.	<b>lockX</b> [ $x$ ]		
3.	$r$ [ $x$ ]		
4.	$w$ [ $x$ ]		
5.		<b>BOT</b>	
6.		<b>lockS</b> [ $x$ ]	$T_2$ has to wait
7.	<b>lockX</b> [ $y$ ]		
8.	$r$ [ $y$ ]		
9.	<b>unlockX</b> [ $x$ ]		$T_2$ resumes
10.		$r$ [ $x$ ]	
11.		<b>lockS</b> [ $y$ ]	$T_2$ has to wait
12.	$w$ [ $y$ ]		
13.	<b>unlockX</b> [ $y$ ]		$T_2$ resumes
14.		$r$ [ $y$ ]	
15.	<b>commit</b>		
16.		<b>unlockS</b> [ $x$ ]	
17.		<b>unlockS</b> [ $y$ ]	
18.		<b>commit</b>	

# Strict 2PL

- Basic 2PL does not guarantee recoverability
- However, it can be modified to a *strict 2PL*:
  - all locks have to be held until the end of a transaction (there is no shrinking phase)
  - this guarantees recoverability (the created schedules are even strict)

# Strict 2PL (2)

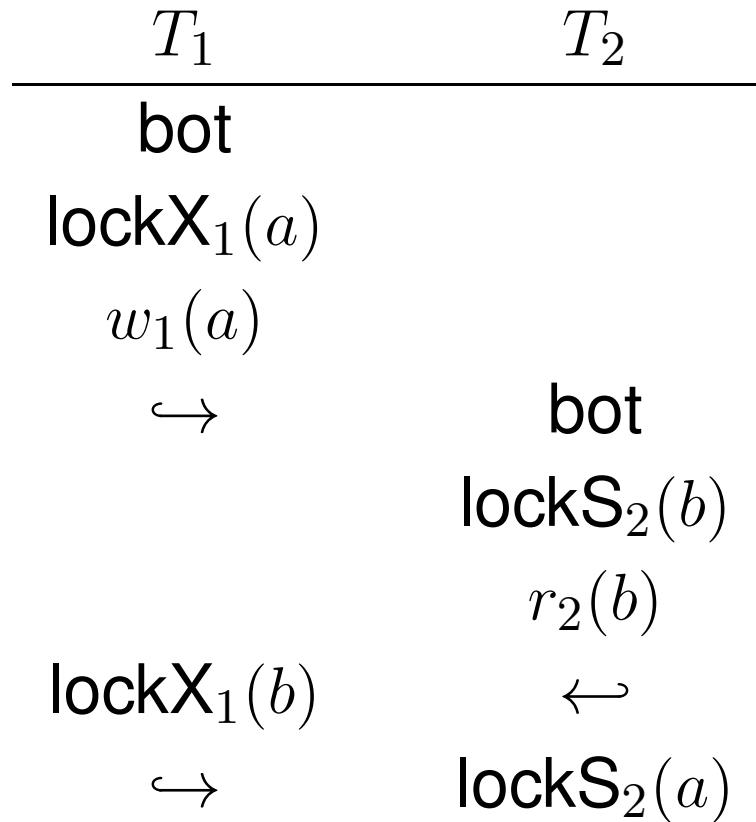


# 2PL and Serializability

- 2PL guarantees serializability
  - The point where a TA has reached the end of its growing phase is called its *lock point*
  - Two TAs whose operations conflict cannot have their lock points at the same time
  - One has to be before the other: this determines their serializability order
- However, 2PL has one disadvantage: it does not avoid deadlocks

# Deadlocks

- Example of a deadlock:

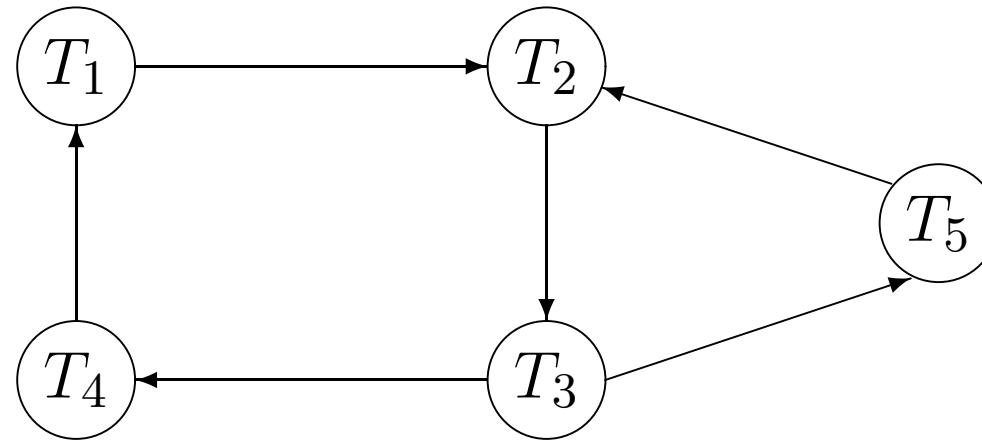


# Recognizing Deadlocks

- If a deadlock occurs, a DBMS should be able to recognize and resolve it
- One strategy is to introduce time-outs
- If a TA waits for a lock for a certain amount of time, then the DBMS assumes that a deadlock has occurred
- This approach has some issues:
  - How do you determine the threshold for the time to wait?
  - How do you resolve the deadlock?
- A better method is to use a waits-for graph

# Waits-For Graph

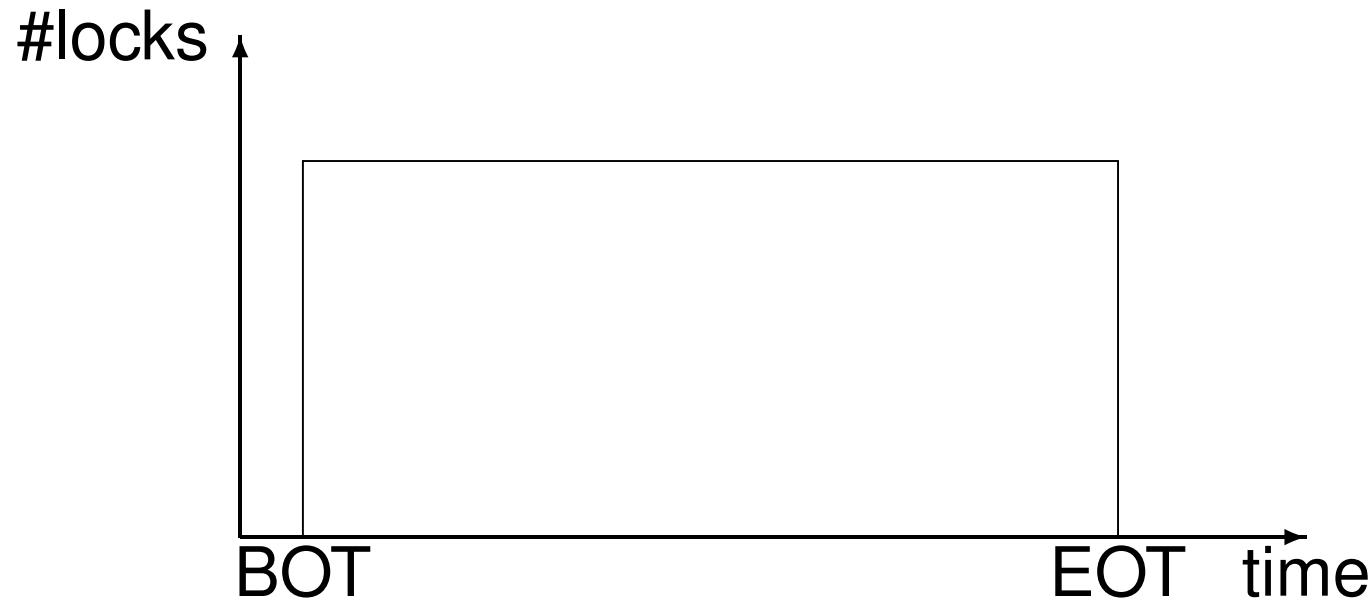
- Waits-for graph: TAs are the nodes, their waits-for relationships are the edges
- If the graph is cyclic, there is a deadlock
- Example:



- Waits-for graph contains a cycle, i.e., we have a deadlock
- We can break the cycle by aborting  $T_2$  or  $T_3$

# Deadlock Prevention

- Theoretically, deadlocks can be prevented by using a technique called preclaiming
- In *preclaiming* all locks are acquired at the beginning of a TA
- However, this is an unrealistic assumption



# Deadlock Prevention (2)

- Another method is based on assigning priorities to TAs
- When  $T_i$  wants to obtain a lock that is currently held by  $T_j$ , then
  - if the priority of  $T_i$  is higher, then  $T_i$  may wait
  - if the priority of  $T_i$  is lower, then  $T_i$  has to abort



# Deadlock Prevention (3)

- With priorities a cyclical wait is prevented
- However, we have to be careful when assigning priorities:
  - When an aborted TA  $T_i$  always gets a low priority when restarting
    - then TAs with higher priorities can jump the queue and  $T_i$  will never finish
    - This is called a *livelock*

# Dead/Livelock Prevention

- We can prevent dead- and livelocks by using timestamps as priorities
- Timestamps are unique and grow monotonically
- When starting, a TA is assigned a timestamp  $ts$ , when restarting it keeps this timestamp
- The older the timestamp, the higher the priority
- At some point a repeatedly aborted TA will have the oldest timestamp  $\Rightarrow$  no more livelocks

# Dead/Livelock Prevention (2)

- For the following, assume that  $T_j$  holds lock, while  $T_i$  requests it
- Actually a system can implement two different locking strategies with timestamp-based priorities:
  - Wait-Die:

if $ts(T_i) < ts(T_j)$	if $ts(T_i) > ts(T_j)$
then $T_i$ waits	then $T_i$ aborts
  - Wound-Wait:

if $ts(T_i) < ts(T_j)$	if $ts(T_i) > ts(T_j)$
then $T_j$ aborts	then $T_i$ waits

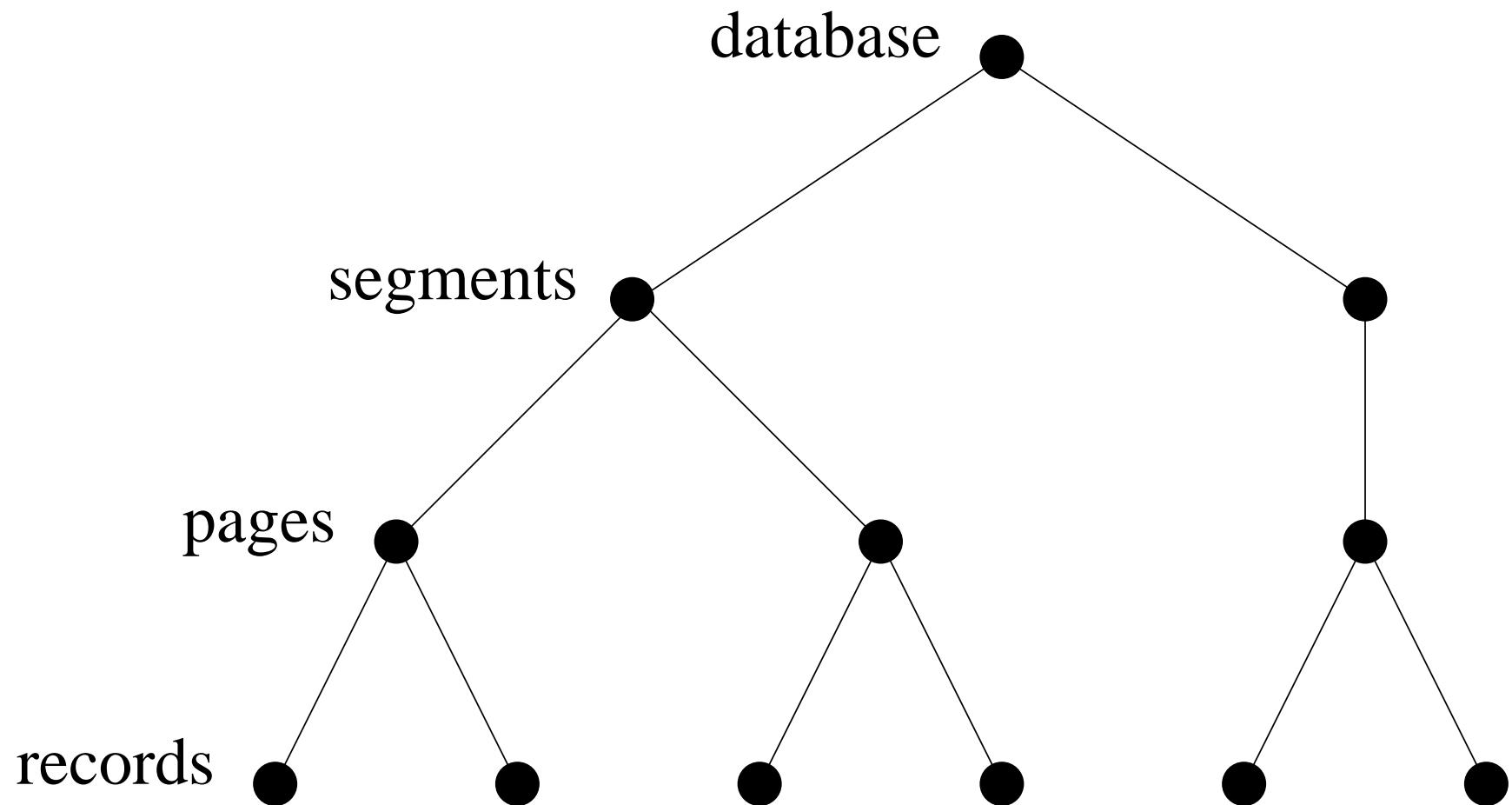
# Phantom-Problem

- With (strict) 2PL we got rid of the problems discussed at the beginning of the chapter...
- ... except for the phantom problem



- The phantom problem cannot be solved by locking data items: TAs cannot obtain locks on non-existing data items
- Has to be solved using other mechanisms: multi-granularity locking (MGL) or range locks

# MGL



# Locking in MGL

- MGL has some additional types of locks (intention locks)
  - $S$  (shared): for reading TAs (same as before)
  - $X$  (exclusive): for writing TAs (same as before)
  - $IS$  (intention share): TA intents to read further down in the hierarchy
  - $IX$  (intention exclusive): TA intents to write further down in the hierarchy

# Compatibility Matrix

- We also need a new compatibility matrix:

lock request	lock held				
	none	S	X	IS	IX
S	✓	✓	—	✓	—
X	✓	—	—	—	—
IS	✓	✓	—	✓	✓
IX	✓	—	—	✓	✓

# Locking Protocol

- Locks have to be requested top-down in the hierarchy
  - For an *S* or *IS* lock:
    - all ancestors in the hierarchy have to be locked in *IS* or *IX* mode
  - For an *X* or *IX* lock:
    - all ancestors in the hierarchy have to be locked in *IX* mode
- Locks have to be released bottom-up
  - A lock may only be released when none of its descendants are still locked

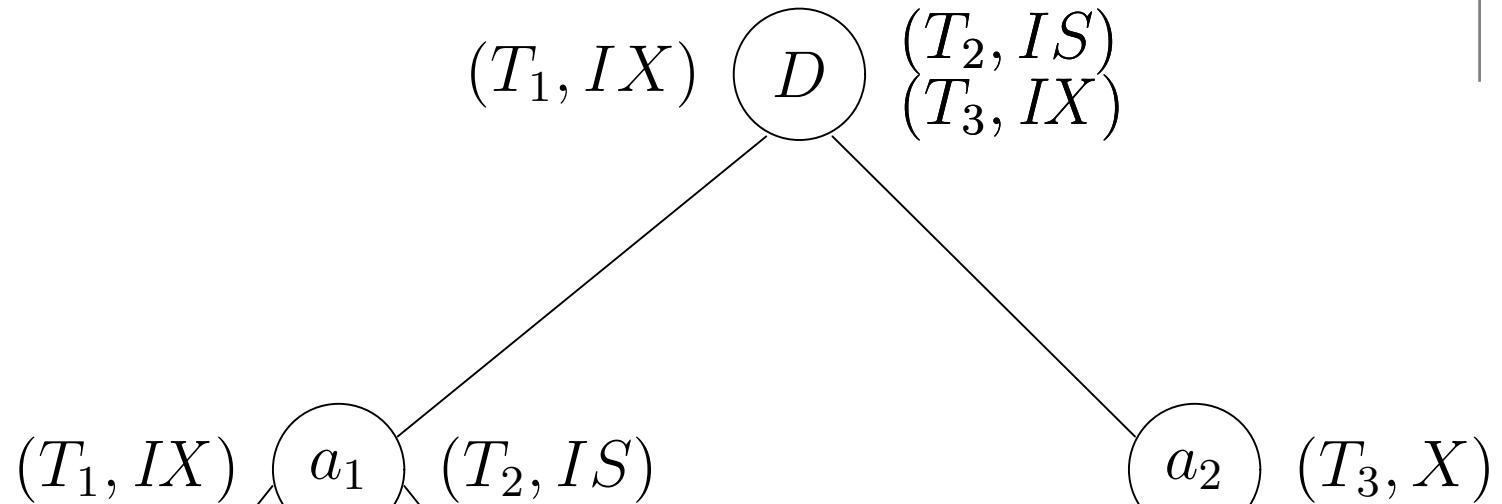
# Example

database

segments  
(areas)

pages  $(T_1, X)$

records



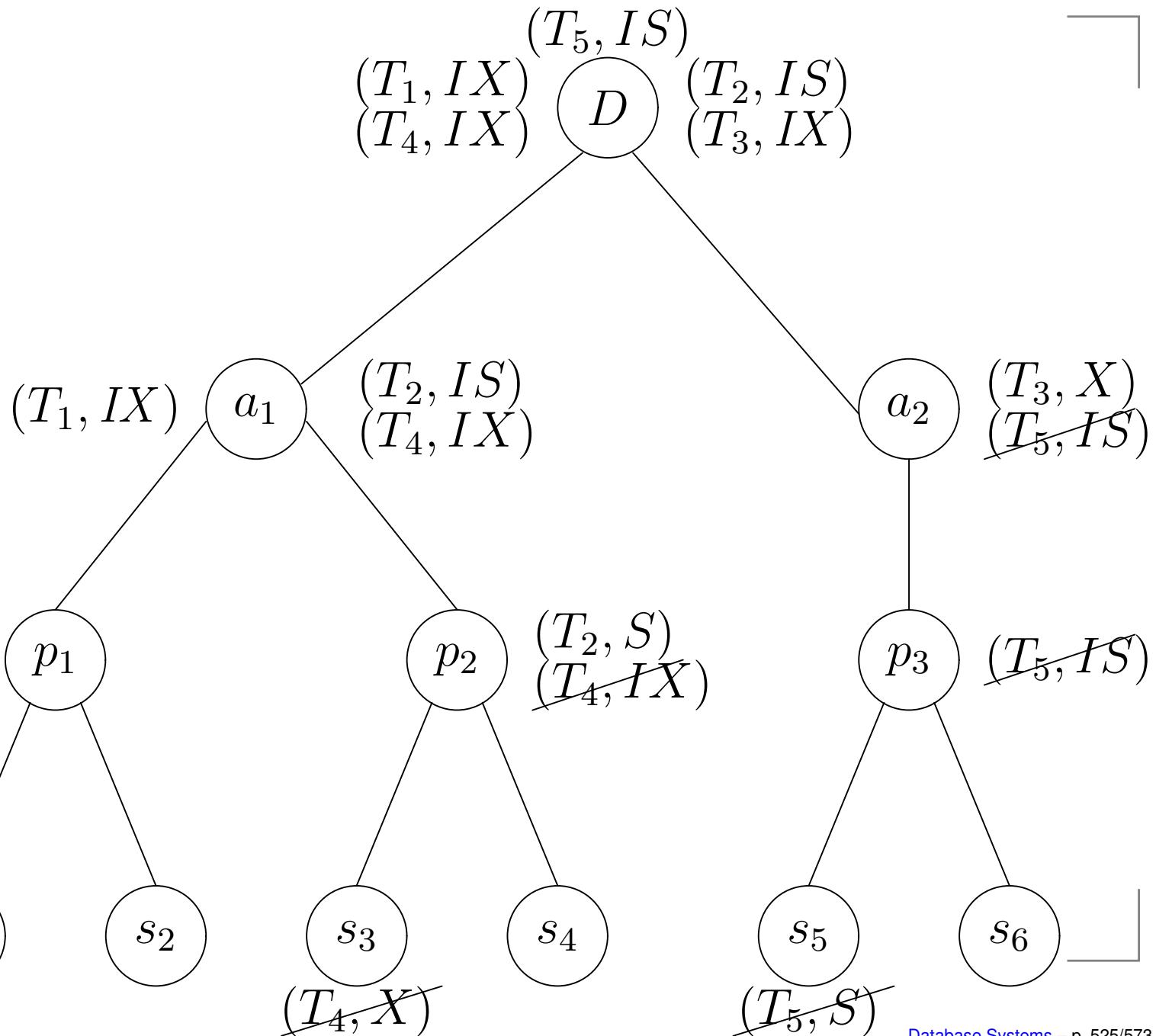
# Example (2)

database

segments  
(areas)

pages  $(T_1, X)$

records

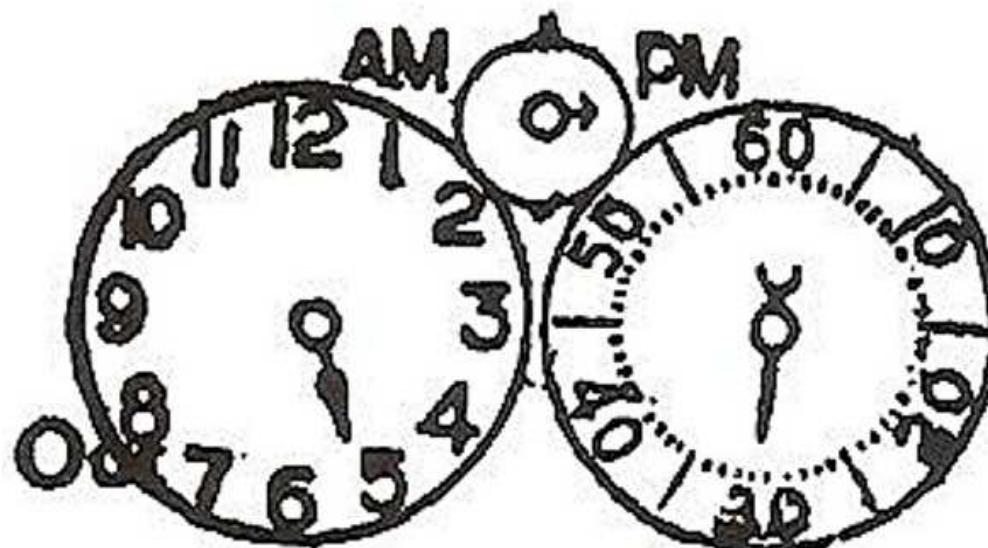


# Example (3)

- TAs  $T_4$  and  $T_5$  have to wait
- In this case we do not have a deadlock
  - However, deadlocks are possible in the (simple) MGL protocol
  - MGL can be combined with the deadlock prevention techniques presented for 2PL
- Using 2PL with MGL gives us serializable schedules without phantoms

# Timestamp-Based Methods

- Let us now have a look at optimistic schedulers
- One well-known type is *timestamp-based synchronization*
- Every TA is assigned a distinct timestamp
- As a consequence, every operation of this TA is also labeled with this timestamp



# Timestamps

- A scheduler uses the timestamps to order conflicting operations
  - Assume that  $p_i[x]$  and  $q_j[x]$  conflict
  - $p_i[x]$  is executed before  $q_j[x]$ , iff the timestamp of  $T_i$  is older than that of  $T_j$



# Timestamps (2)

- With each data item  $x$  we store the largest timestamp of any successful operation on  $x$
- We do this for every operation type  $q$ :  
 $\text{max-}q\text{-scheduled}(x)$ 
  - So there's a *max-read* and a *max-write*
- Scheduling an operation  $p$  means:
  - Check timestamp of  $p$  with all the  $\text{max-}q\text{-scheduled}(x)$  where  $p$  and  $q$  conflict
  - If the timestamp of  $p$  is older than at least one of the conflicting  $\text{max-}q\text{-scheduled}(x)$ , then reject  $p$  (and abort TA)
  - Otherwise execute  $p$  and update  $\text{max-}p\text{-scheduled}(x)$

# Other Properties

- The basic timestamp-based method may create schedules that are not recoverable
- In order to guarantee recoverability, a scheduler has to make sure that TAs commit in order of their timestamps
- If there are still active TAs from which  $T_i$  has read, the commit of  $T_i$  is delayed



# Issues with Timestamps

- Timestamp-based synchronization is not very popular in practice
- Every operation becomes a write operation, as all  $\text{max-q-scheduled}(x)$  records need to be updated
- Similar to lock-based protocols we still have to solve the phantom problem...

# Synchronization Not Fast Enough?

- There is an overhead when synchronizing concurrent accesses by multiple users
- Sometimes you can get away with weakening the isolation guarantees:
  - You only need an approximate answer, e.g. number of accounts with a balance of at least € 1000
  - Transactions with human interaction, e.g. seat reservation systems

# Flight Reservation System

- Reservation consists of three steps:
  - Get list of available seats
  - Let customer pick a seat
  - Book seat
- Running a single transaction:
  - All seats are locked while customer decides
  - Other customer transactions have to wait
- Running two transactions  
(1st TA: get list, 2nd TA: book seat):
  - Seat may have been taken when starting second TA
  - Probably better in this case than delaying all other TAs

# Snapshot Isolation

- Splitting a TA into two parts is not always an option
- Some systems use *snapshot isolation* to make synchronization more efficient
  - Read-only TAs do not lock data items
  - We remember previous values for all data items
    - Can be implemented using log entries
  - Every TA sees values which are consistent with its starting time

# Snapshot Isolation (2)

- Early snapshot isolation was not truly serializable
- Example:  $x = 3, y = 10$ 
  - $T_1$  reads  $x$ , writes  $y$  ( $y \leftarrow x$ )
  - $T_2$  reads  $y$ , writes  $x$  ( $x \leftarrow y$ )
- Possible result with snapshot isolation:
  - $r_1(x) \quad x = 3$
  - $r_2(y) \quad y = 10$
  - $w_1(y) \quad y \leftarrow 3$
  - $w_2(x) \quad x \leftarrow 10$
- Result would be  $x = 10, y = 3$ 
  - Not possible with serializable schedules (either both are 3 or both are 10)

# Snapshot Isolation (3)

- Nevertheless, snapshot isolation still avoids the following problems:
  - Dirty reads
  - Non-repeatable reads
  - Phantom problem
- Newer versions are fully serializable
- Additionally:
  - Readers do not block writers
  - Writers do not block readers

# Summary

- Multi-user synchronization is an important topic in concurrently used DBMSs
- Usually, this is transparent to the user
  - However, by changing the isolation level, accuracy can be traded for performance
- Two of the most well-known synchronization mechanisms are
  - Lock-based synchronization
  - Timestamp-based synchronization

# Chapter 12

## Databases and Security

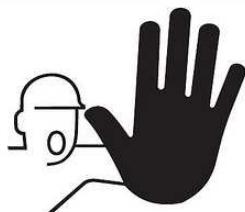
# Authorization

- A user may be assigned authorizations on parts of a database
- Authorizations cover
  - reading data
  - inserting new data
  - updating data
  - deleting data
- Each type is called a *privilege*
- A user may have all, none, or a combination of privileges (for parts of a DB)

# Granting Privileges

- Privileges can be granted to a user . . .
- . . . and later on be revoked again
- One user, the *database administrator*, has all the privileges
- Granting and revoking privileges is done via SQL commands
  - This is part of the Data Definition Language (DDL)

**NOTICE**



**AUTHORIZED  
PERSONNEL  
ONLY**

# SQL Syntax

- The general statement for granting privileges is:

```
grant    privilege list
on      relation or view name
to      user or role list;
```

- A *privilege list* is made up of a combination of **select**, **insert**, **update**, and **delete**
  - ... or **all privileges** for all of them
- This is followed by a relation or view name
- and a user name (we'll come to roles in just a moment)

# Examples

- The users peter, paul, and mary may run select queries on the relation student

```
grant    select  
on      student  
to      peter, paul, mary;
```

- When granting update and insert privileges, attributes can be specified:

```
grant    update(office_no)  
on      professor  
to      peter;
```

- This allows the user peter to update the attribute office\_no in the relation professor

# Revoking Privileges

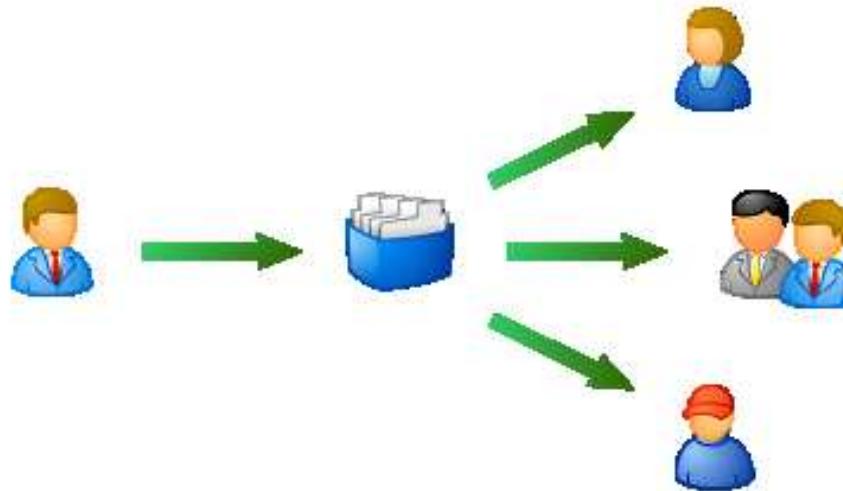


- Privileges can also be withdrawn via a `revoke` statement
- The general syntax is:

**revoke** *privilege list*  
**on** *relation or view name*  
**from** *user or role list;*

- Works like a `grant` statement in reverse

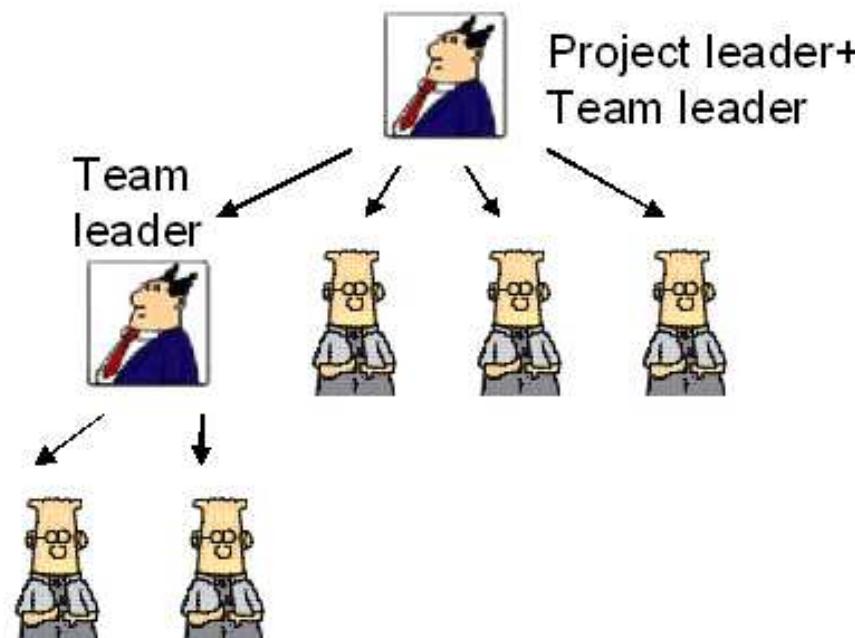
# Multiple Users



- Large database system may have hundreds or even thousands of users
- Granting and revoking privileges individually on all relations may be very tedious
- The user name `public` grants a privilege to every user of the system
- A more fine-grained approach uses *roles*

# Roles

- Often groups of people do similar work and need the same privileges
- In a database it is possible to
  - define a role
  - give privileges to this role
  - and add users to this role



# SQL Syntax

- Here are some examples on how this looks in SQL:

```
create role instructor;
```

```
grant      select  
on        course  
to        instructor;
```

```
grant      instructor to john;  
create role professor;  
grant      instructor to professor;  
grant      professor to sven;
```

# Authorization and Views

- Privileges in combination with views can be used to make parts of a relation visible
- For example, an administrator may only see records of computer science assistants
  - Create the following view:

```
create view csasst as
select *
from   assistant
where  area = 'computer science';
```
  - Then grant select privilege on csasst and revoke all privileges on base table assistant

# Transfer of Privileges

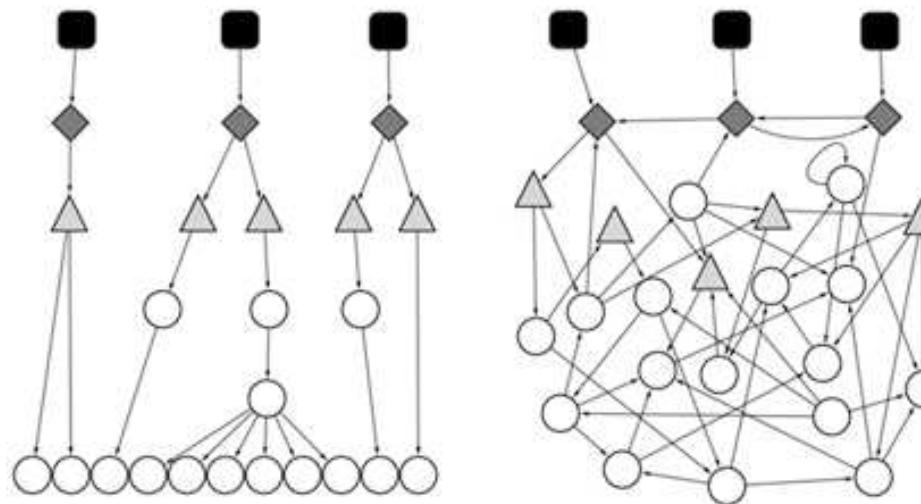
- A user who has been granted a privilege may be allowed to pass it on
  - The default does not allow this
- If we want to allow someone to grant a privilege to others, we use the `with grant option`

```
grant    select  
on        student  
to        peter with grant option;
```



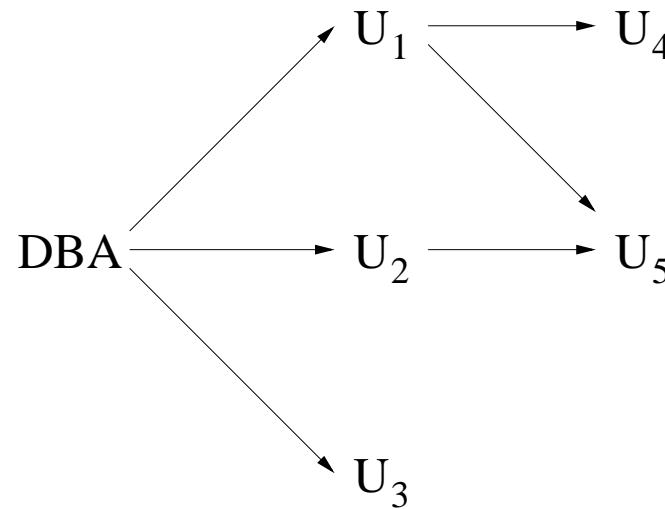
# Transfer of Privileges (2)

- Usually the creator of a database object holds all privileges
  - This includes the privilege to grant privileges
- What happens if there is a whole chain of granted privileges and we start revoking some?



# Authorization Graph

- We can use an *authorization graph* to check:



- A user has a privilege, iff there is a path from the root (DBA) to the user node
- Revoking a privilege from a user
  - removes that user
  - and everyone on outgoing edges of that user not connected to the root otherwise

# Cascading Revokes

- Revoking a privilege from  $U_1$  from the previous graph
  - also removes  $U_4$ 's privilege
  - but not  $U_5$ 's, as he/she is still connected via  $U_2$
- Recursively revoking privileges is called a *cascading revoke*
- Can be prevented by the `restrict` clause
  - Will return an error if there is a cascading revoke

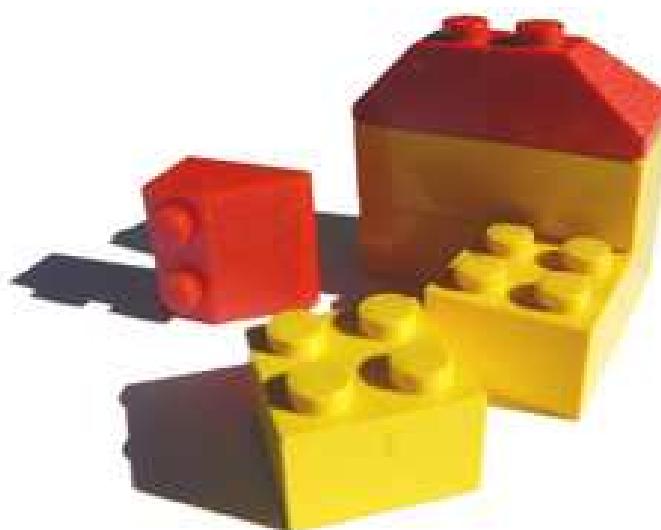
# Cascading Revokes (2)



- Sometimes privileges should be granted by a role, not an individual
- For example, the role of *dean* can grant privileges associated with the role of *professor* or *instructor*
  - If the current dean steps down and the user account is removed, granted privileges should stay
  - Can be done by adding the clause `granted by current_role`

# Limits of Authorization in SQL

- While SQL supports a fairly flexible system, it has limits
- Many applications require a very fine-grained authorization
- For example, we want students to see only their own grades
- That means, we need authorization on the tuple level
  - Databases only support relation, view, or attribute level



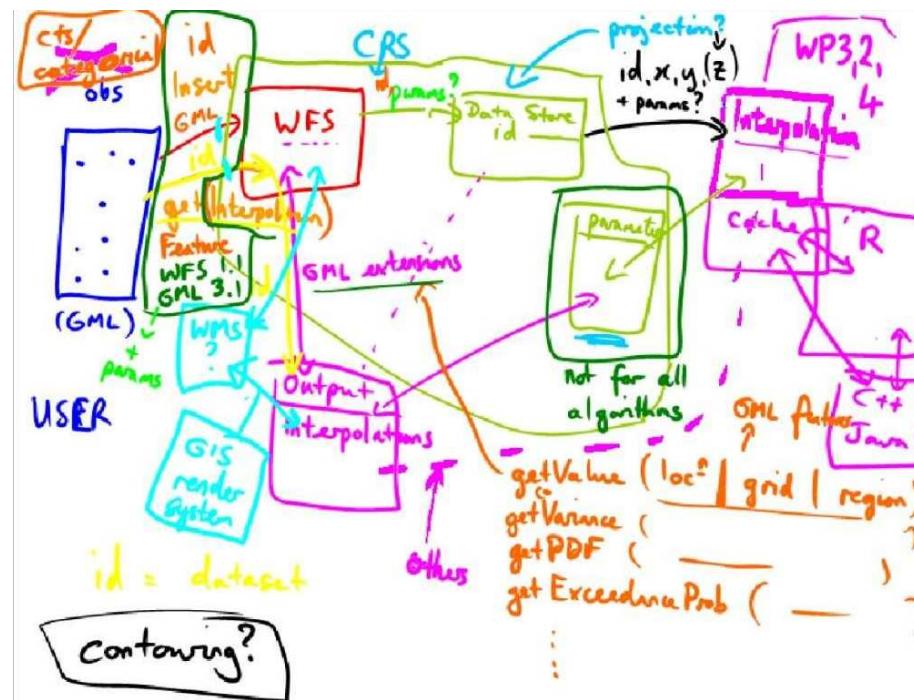
# Limits of Authorization in SQL (2)

- Often, there is a lack of end-user information
- For example, in web applications end users usually do not have individual user IDs in the database
- Makes it difficult to apply the SQL authorization scheme



# Limits of Authorization in SQL (3)

- As a consequence a lot of the authorization moves into the application code
- The point of a DBS was to provide infrastructure and have clear responsibilities



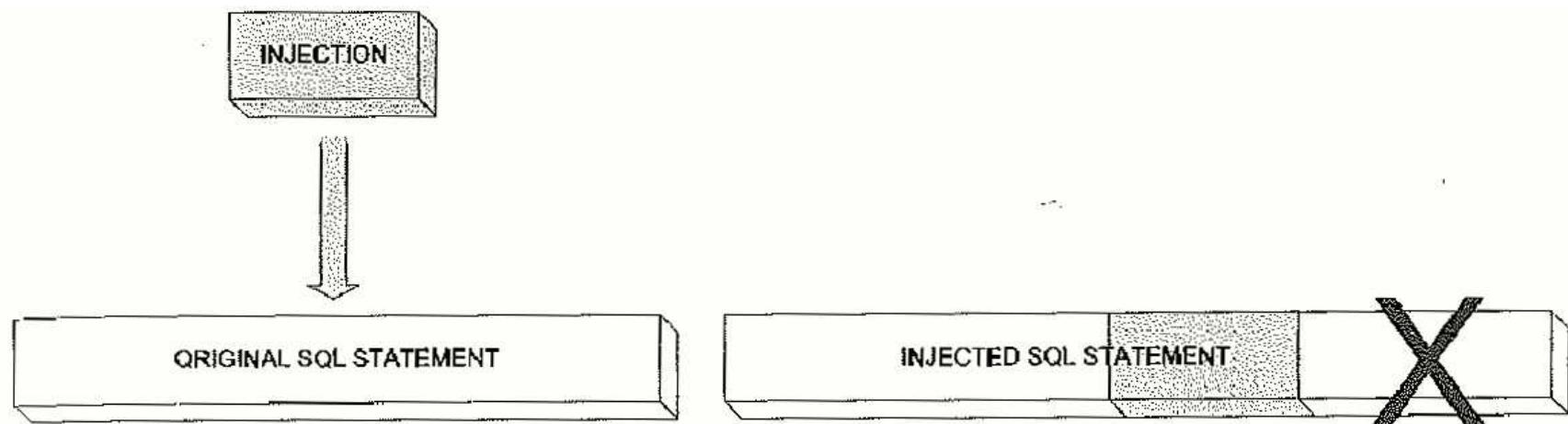
# Application Security

- Even if the database is pretty secure, a badly written application can compromise the whole system
- Many database applications have a web (or mobile) interface that can be exploited
- In particular, we are looking at
  - SQL injection
  - Cross-site scripting and request forgery



# SQL Injection

- In SQL injection attacks, the database runs an SQL query created by an attacker
- This is usually done by manipulating a valid SQL statement:



# SQL Injection (2)

- Applications that build SQL queries on the fly are especially vulnerable to this
- For example, assume a Java application gets a string name and constructs the query

```
"select * from student where name = '" + name + "' ;"
```

- Instead of a name, a user might enter some SQL:

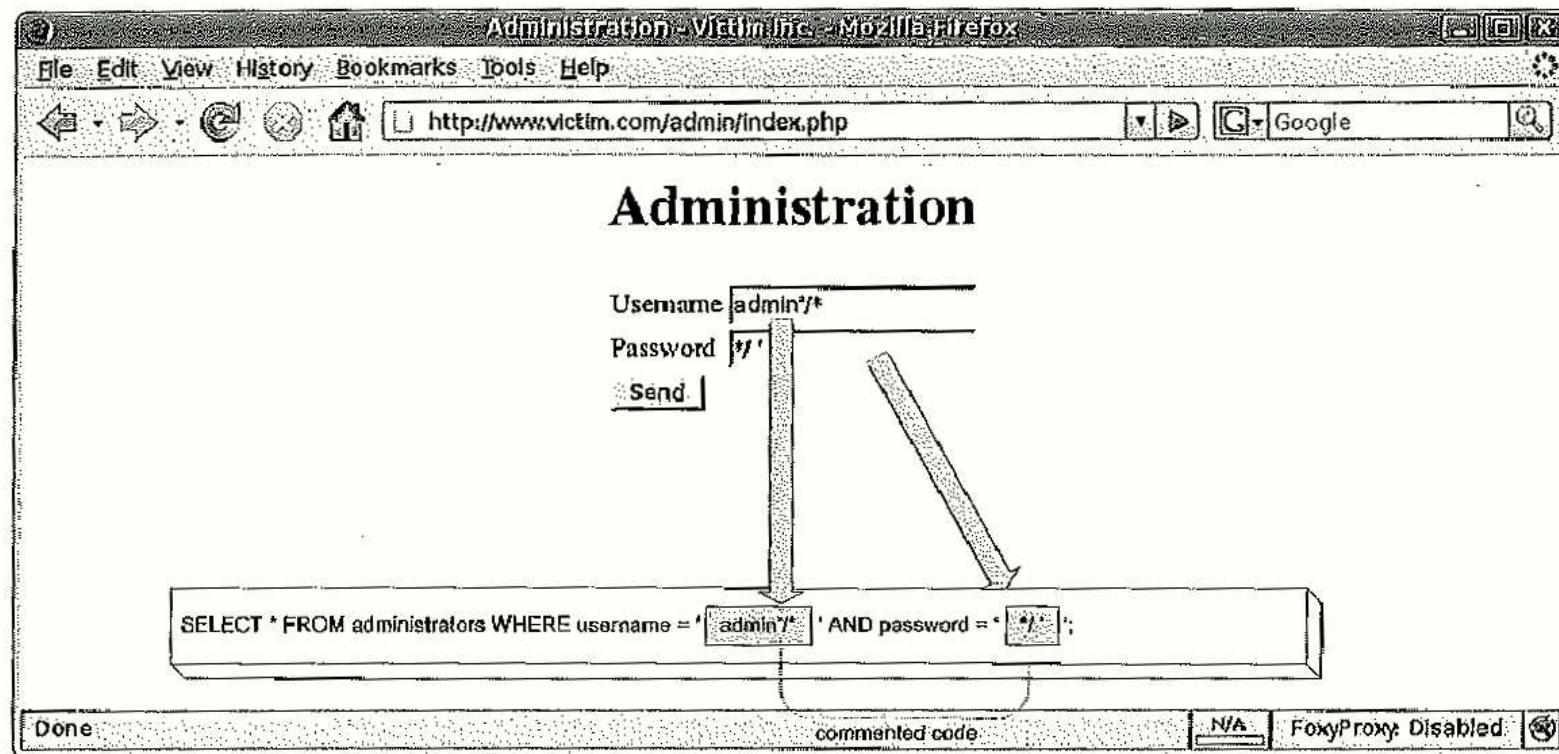
X' or 'Y' = 'Y

turning the SQL statement into

```
select * from student  
where name = 'X' or 'Y' = 'Y' ;
```

# SQL Injection (3)

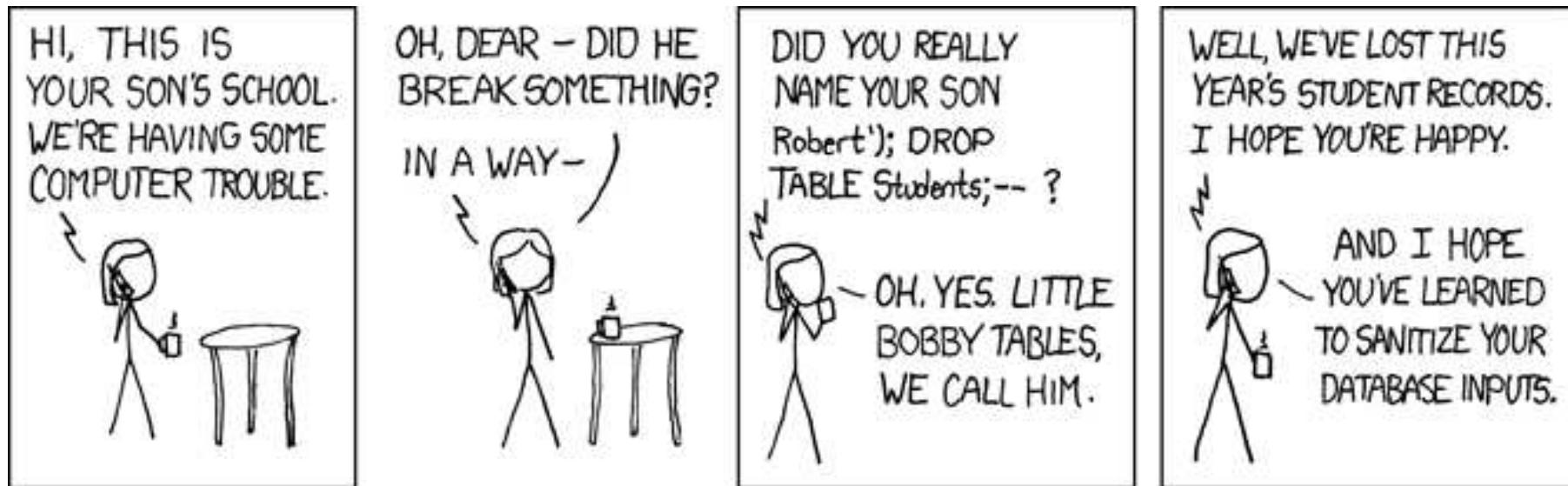
- Depending on the application, this can have serious consequences:



- Here comments are used to cut part of the SQL query

# SQL Injection (4)

- This is not just limited to select statements
- Depending on the configuration of the server, multiple statements may be executed in one go



# Remedies

- So, how should you build your database application?
- Any query that relies on user input should use prepared statements
- In prepared statements, some values are replaced by “?”
- For example, the following will insert a tuple

```
PreparedStatement pSt = con.prepareStatement(  
        "insert into student values (?, ?, ?)");  
pSt.setInt(1, 102093);  
pSt.setString(2, "James Smith");  
pSt.setDate(3, "1991-10-05");  
pSt.executeUpdate();
```

# Prepared Statement

- Not only will this run faster (if SQL statement is used multiple times)
- It will also escape special characters
- For example, the string

X' or 'Y' = 'Y

would become

X\' or \'Y\' = \'Y

rendering the attempted attack harmless

# Other Forms of Attack

- Not every attack can be prevented with prepared statements
- For example, the following lets a user sort a result:

```
"select * from student order by "+orderAtt+";"
```

- Application has to make sure that the variable `orderAtt` can only contain valid attribute names
- In general, any input coming from a user has to be sanitized!

# Cross-Site Scripting (XSS)

- Many web sites rely on the execution of code embedded in HTML on the client side
  - Client-side scripting languages such as JavaScript are a popular option
- If an attacker is able to smuggle code onto a web site, it may be executed on a client
- For a database-related example, assume the following:
  - Users enter data into a database via a web site
  - Later on, other users view this information
  - Malicious users can enter JavaScript instead of data

# XSS (2)

- The effects of executing malicious code include
  - changing or deleting files on the local system
  - monitoring key strokes
  - sending out confidential information (e.g. cookies)
  - interacting with other web sites of a user



# Cross-Side Request Forgery (XSRF)



- XSRF attempts to hijack a session running in another tab or window of the browser
- Can fool a server, as request is coming from a valid client
- Can even be done without scripting, e.g.

```
<img src=
"http://site.com/action?user=alice&action=doThing">
```

# Protection from XSS/XSRF

- We provide some general remarks (there are more complex attacks)
- Preventing your site from becoming an attack launch pad:
  - Sanitize all user input
  - There are functions to strip out HTML, scripts, or other code
- Preventing your site from becoming a target:
  - Check referrer in the HTTP header
  - Tie session not only to cookies, but also to IP address
  - Never use GET to update any data or to send sensitive data

# Password Leakage



- Storing passwords in clear text in application code or a database is not a good idea
- If you have to store a password, it needs to be encrypted
- Many databases can be configured to use authentication scheme of operating system

# Summary

- Information security is an important and wide topic
  - We are only scratching the surface here, covering basic database-related areas
  - It's possible to spend a whole course on this topic

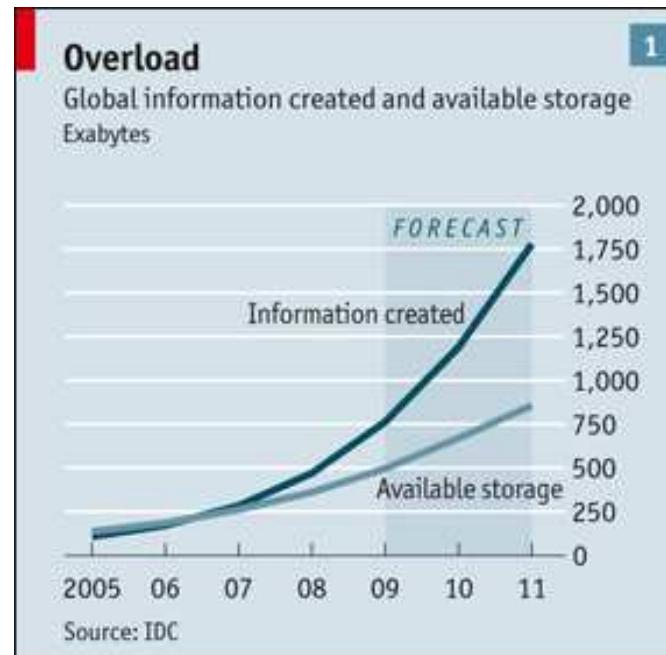


# Chapter 13

Wrapping Up

# Hungry for Data

- The amount of electronic data collected and stored rapidly increases on a daily basis:



- Applications need a way to deal with all this data in an intelligent way
- That's where database systems come in

# Data Processing



## DATABASES

are like scuba diving : They let you get deeper.

- If you want to process, analyze, and mine the data, you need some familiarity with the tools
- In this lecture we had a look at what databases are able to do (and what they cannot do)
- We also had a look at what is going on behind the scenes in a DBMS

# The Future

- While relational databases dominated for decades, it seems the landscape is about to change
- The world seems to be slowly moving away from the one-size-fits-all relational systems to NoSQL
- However, the management of large amounts of data is here to stay
- That is why we also covered general principles applied in data management