

# Databases

Lecture 1

Introduction to Databases. Fundamental Concepts

# Databases

- Lecture2 + Seminar1 + Laboratory2
- grading
  - written exam (W) - 50%
  - practical exam (P) - 25%
  - labs (L) - 25%
  - W, P  $\geq$  5
- To attend the exam, a student must have at least 12 laboratory attendances and at least 5 seminar attendances, according to the Computer Science Department's decision: <https://www.cs.ubbcluj.ro/wp-content/uploads/Hotarare-CDI-29.04.2020.pdf>
- <https://www.cs.ubbcluj.ro/~sabina>
- sabina@cs.ubbcluj.ro

# Context

- databases - virtually everywhere
  - education
  - research
  - financial services
  - media
  - e-commerce
  - social networks
  - tourism
  - telecommunications
  - ...

# Context

- extraordinary data growth, huge & complex **data sets**
  - data owners (organizations, individuals, etc.) need to **efficiently manage** their data, to extract correct information in a timely manner
- => need powerful & flexible **data management systems** that simplify data management

# 1. The Components of an Application

- data (stored in files or databases)
- management algorithm
- user interface

## 2. Data Storage Methods

- files
- databases
- distributed databases

### 3. Files: Characteristics and Limitations

\* ex: bank storing a large collection of data on employees, clients, accounts, transactions, etc.; requirements:

- quick answers to questions about the data
- protecting the data from inconsistent changes made by users accessing it at the same time
- restricting access to some parts of the data, e.g., salaries
- difficulties encountered when storing and managing the data using a collection of files

->

### 3. Files: Characteristics and Limitations

- multiple data storage formats
- data redundancy
  - some parts of the data can be stored in multiple files => potential inconsistencies
- read / write operations are described in the program (using certain record structures) => difficulties in program development (changes in the file structure lead to changes in the program)
- changing data (modifying / removing records), retrieving data based on search criteria - difficult operations
- integrity constraints - checked in the program
- main memory management, e.g., how is a data collection of tens / hundreds of GB loaded for processing?

### 3. Files: Characteristics and Limitations

- no adequate security policies, allowing different users to access different segments of data
- concurrent data access is difficult to manage
- data must be restored to a consistent state in the event of a system failure, e.g., a bank transaction transferring money from account A to account B is interrupted by a blackout after having debited account A, but prior to crediting account B; money must be put back into account A
- files: useful for single-user programs dealing with a small amount of data

## 4. Data Description Models

- data must be described according to a model (to be managed automatically)
- **data description model:** set of **concepts and rules** used to model data; such concepts describe:
  - the structure of the data
  - consistency constraints
  - relationships with other data
- the **schema** of the database (the data structure or template)
  - data structures used to describe a collection of data stored in a database
  - data in the collection: an **instance** of the schema (analogy: classes and objects in object-oriented programming)
- the data description constructs in a model are *high-level*, hiding many low-level details about data storage, e.g., there's a long way from the *Student* entity to the computer stored bits ☺

## 4. Data Description Models

- entity-relationship
- relational
- network
- hierarchical
- object-oriented
- noSQL
- semistructured (XML)

## 4. Data Description Models: The Relational Model

- **relation:** the main concept used to describe data
- the schema of a relation:
  - the relation's name
  - for each field (column): its name and type
- example: Movie(*mid*: string, *title*: string, *director*: string, *year*: integer)

# 4. Data Description Models: The Relational Model

- example:

- instance of the Movie relation
- every row has 4 columns

<i>mid</i>	<i>title</i>	<i>director</i>	<i>year</i>
84386	Hibernatus	Édouard Molinaro	1969
7583	Moscow Does Not Believe in Tears	Vladimir Menshov	1980
47288	Close Encounters of the Third Kind	Steven Spielberg	1977
32	Contact	Robert Zemeckis	1997
46747	E.T. the Extra-Terrestrial	Steven Spielberg	1982

## 4. Data Description Models: The Entity-Relationship Model

- **semantic**, more abstract, high-level model
- eases the task of developing a good initial description of the data
- such a semantic model is useful since, even though the database management system's model hides many details, it's still closer to the manner in which the data is stored than to the user's perspective on the data
- a design in such a model is subsequently expressed in terms of the database management system's model
  - e.g., the ER -> relational mapping

## 4. Data Description Models: The Entity-Relationship Model

- main **concepts**: entities, attributes, relationships
- **entity**
  - a piece of data, an object in the real world
  - described by attributes (properties)
- **entity set**
  - entities with the same structure (e.g., the set of students)
  - name, set of attributes
- **attribute**
  - name, domain of possible values, conditions to check correctness
- **key**
  - a restriction defined on an entity set
  - minimal set of attributes with distinct values in the entity set's instances

## 4. Data Description Models: The Entity-Relationship Model

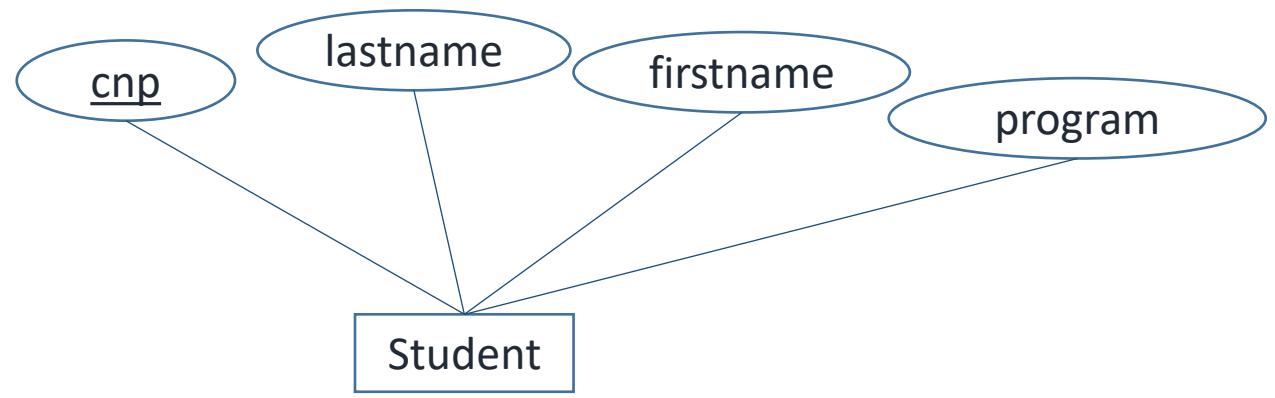
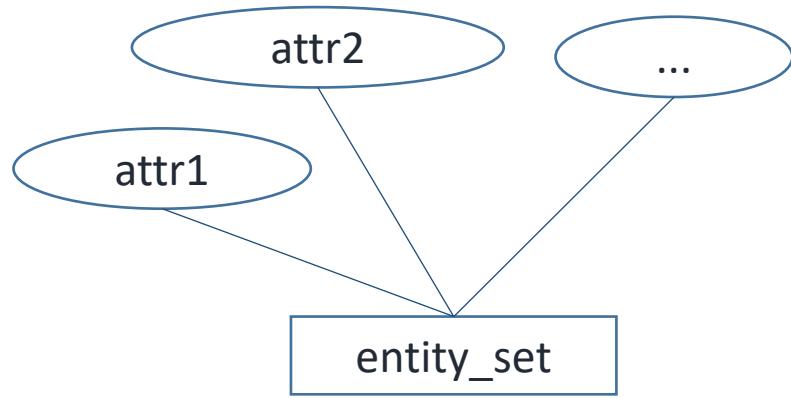
- **relationship**
  - specifies an association among 2 or more entities
  - descriptive attributes can be used
- **relationship set**
  - describes all relationships with the same structure
  - name, entity sets used in the association, descriptive attributes
- **the schema of the model**
  - set of entity sets and relationship sets

## 4. Data Description Models: The Entity-Relationship Model

- **binary relationships** (between entity sets T1 and T2) - **relationship types**:
  - **1:1**: one T1 entity can be associated with at most one T2 entity, and one T2 entity can be associated with at most one T1 entity
    - e.g., the association between group and faculty member (e.g., to specify the groups' tutors)
  - **1:n**: one T1 entity can be associated with any number of T2 entities, and one T2 entity can be associated with at most one T1 entity
    - e.g., the association between group and students
  - **m:n**: one T1 entity can be associated with any number of T2 entities, and one T2 entity can be associated with any number of T1 entities
    - e.g., the association between courses and students
- considered as **restrictions** in the database; when the database is changed, the system checks whether the relationship is of the specified type

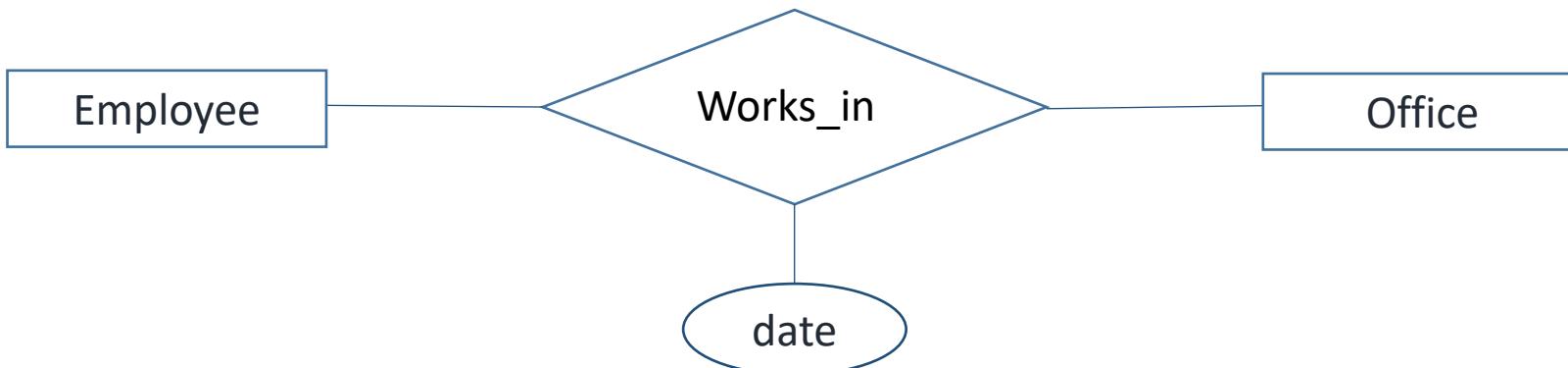
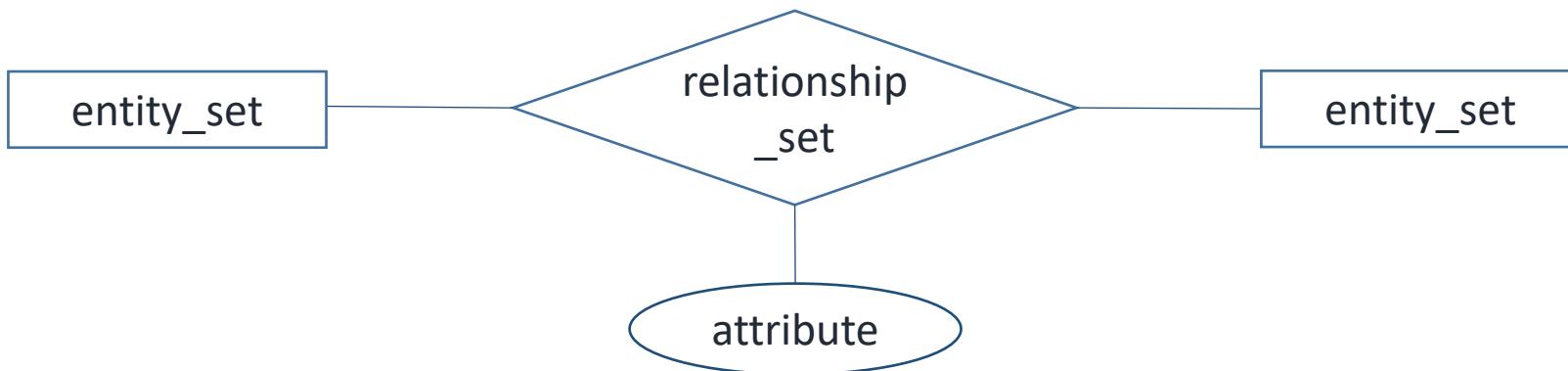
## 4. Data Description Models: The Entity-Relationship Model

- graphical representation of the model
  - entity set and associated attributes



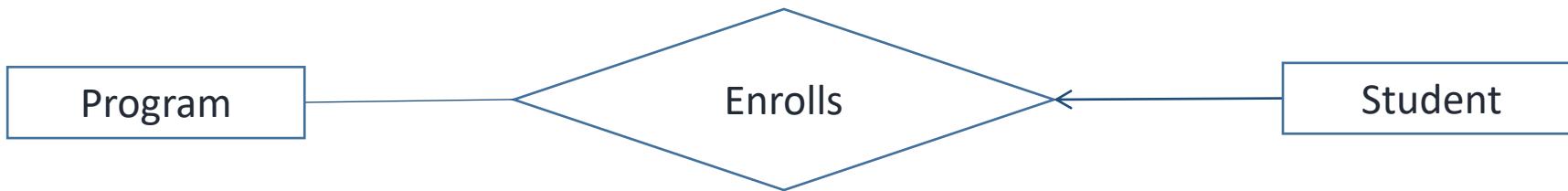
## 4. Data Description Models: The Entity-Relationship Model

- graphical representation of the model
  - relationship set and associated attributes



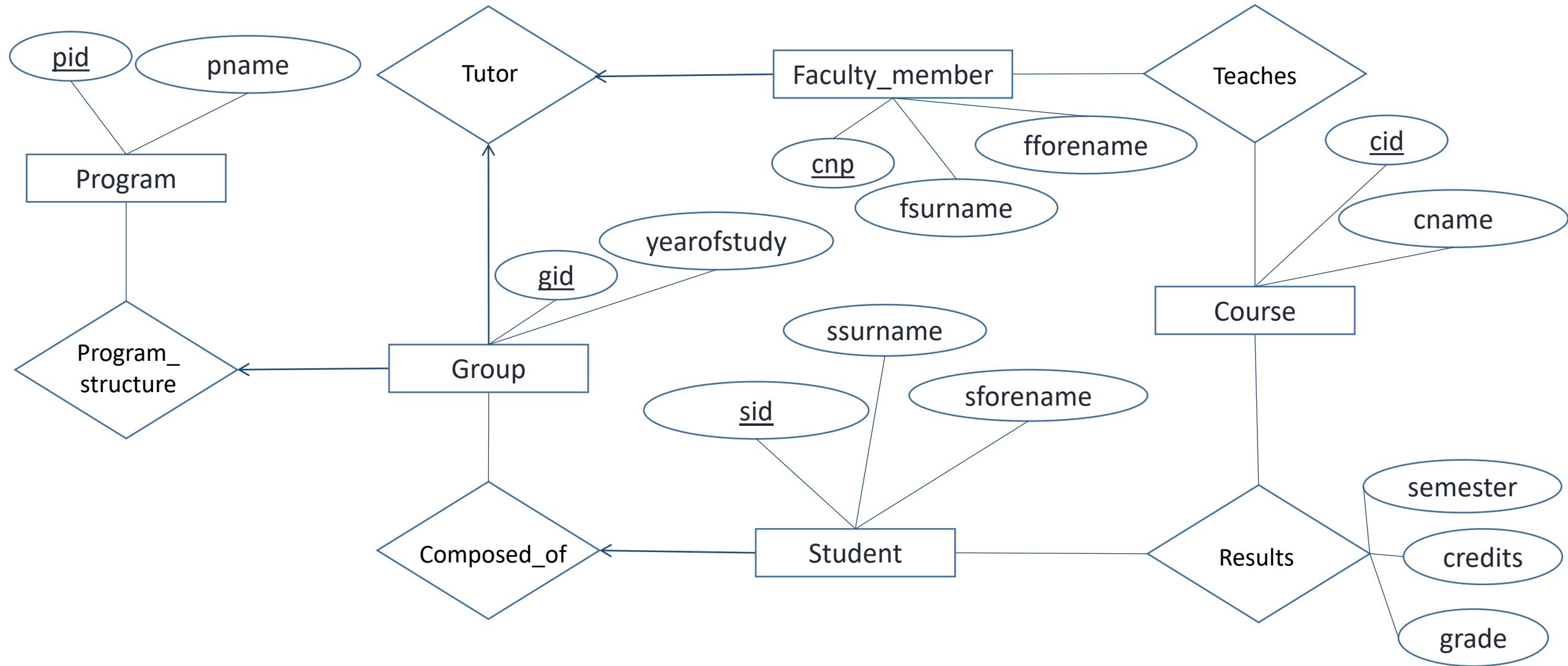
## 4. Data Description Models: The Entity-Relationship Model

- graphical representation of the model
  - 1:n relationship sets - graphical convention:



# 4. Data Description Models: The Entity-Relationship Model

- example:



# 5. Databases and Database Management Systems

- a **database** contains:
  - the database schema
    - **description of data structures** used to model the data
    - kept in a database dictionary
  - a **collection of data** - instances of the schema
  - various **components**: views, procedures, functions, roles, users, etc.
- separation between:
  - **data definition** (kept in the database dictionary)
  - **data management** (insert / delete / update) and querying
- database **design**
  - describe an organization in terms of the data in a database
- data **analysis**
  - answer questions about the organization by formulating queries that involve the data in the database

## 5. Databases and DBMSs

- **database management system (DBMS)**
  - set of programs that are used to manage a database
- DBMS examples
  - Oracle, DB2 (IBM), SQL Server, Informix (IBM), Teradata, MySQL, PostgreSQL, Access, Paradox, Foxpro, ...
- **database system**
  - database + DBMS

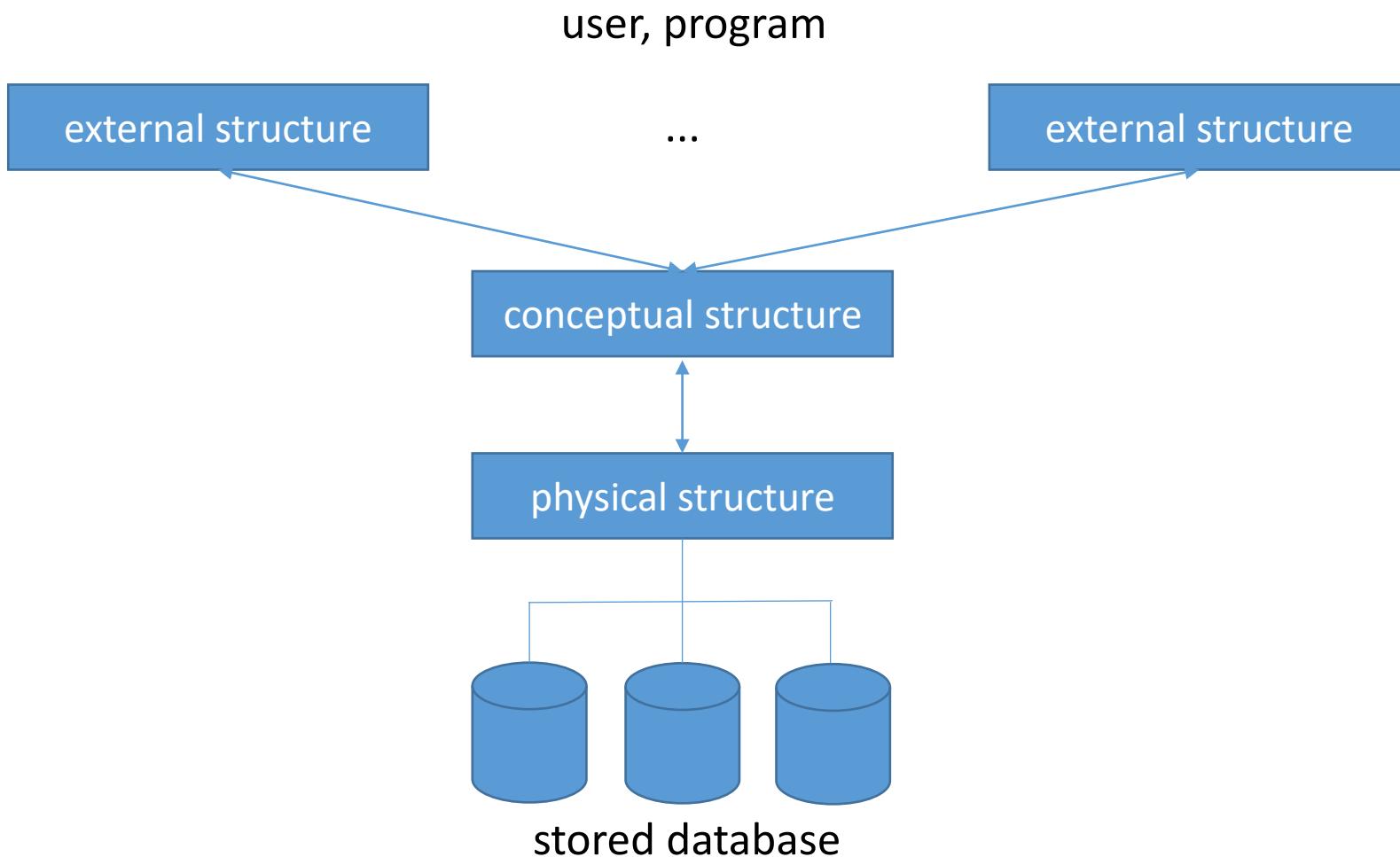
## 6. The Structures of a Database

- when thinking about how to organize and store information about an organization in a database, users operate with high-level concepts corresponding to the entities and relationships in the organization
- on the other hand, the DBMS stores data in the form of a very large number of bits
- the difference between the way users think about their data and the manner in which the data is stored is reconciled through the levels of abstraction in a DBMS

## 6. The Structures of a Database

- the **ANSI-SPARC architecture** - a three-level architecture for a database system, proposed in 1975; in general, this model is used by the main management systems and includes:
  - **the conceptual structure (the database schema)**: describes the data structures and restrictions in the database
  - **external structures**: describe the data structures used by a certain user / program; the description employs a certain model, and the DBMS can find the data in the conceptual structure
  - **the physical structure (internal structure)**: describes the storage structures in the database (data files, indexes, etc.)

# 6. The Structures of a Database



## 6. The Structures of a Database

- e.g., the conceptual structure
  - information about entities, e.g., students, courses, and relationships among entities, e.g., courses taught by teachers:

*Student(sid: string, slastname: string, sfirstname: string, gpa: real)*

*Teacher(tid: string, tlastname: string, tfirstname: string, salary: real)*

*Course(cid: string, cname: string)*

*Grade(sid: string, cid: string, grade: real)*

*Teaches(tid: string, cid: string)*

## 6. The Structures of a Database

- e.g., the physical structure
  - information about how relations are stored on the disk, about the creation of indexes (data structures that speed up queries):
    - relations – stored as unsorted files of records
    - indexes are created on the first column of the Student and Teacher relations

## 6. The Structures of a Database

- e.g., external structure with information about the best result for each student (the student's sid, last name and first name, the name of the corresponding course and the grade)

*BestResults(sid: string, slastname: string, sfirstname: string, cname: string, grade: real)*

- BestResults is a *view* whose definition is based on relations in the conceptual structure; conceptually, it's a relation, but its records are not stored in the database; they are computed on demand using BestResult's definition
- adding BestResults to the conceptual schema => redundancy, the database is prone to errors, e.g., a record for a new grade is introduced in the Grade relation without operating a corresponding (necessary) change in the BestResults relation
- a database can have several external structures, each customized for a group of users

## 7. Logical Independence and Physical Independence

- **data independence:** 3 levels of abstraction => applications are insulated from changes in the data structure / storage
- **logical data independence:** programs using data from the database are not affected by changes in the conceptual structure
  - important: applications can be developed in several stages
    - e.g., the Student relation is replaced by:
      - StudentPublic(*sid*: string, *slastname*: string, *sffirstname*: string)
      - StudentPrivate(*sid*: string, *gpa*: real, *dob*: date)
    - change the definition of BestResults, so it retrieves required data from StudentPublic

## 7. Logical Independence and Physical Independence

- **physical data independence:** applications are insulated from changes in the physical structure of the data
  - important: files (e.g., index) can be added for optimization purposes; users' programs don't check the files (the physical structure) directly

## 8. Functions of Database Management Systems

- database **definition**: definition language (or dedicated applications that generate DDL commands)
- data **management**: insert, update, delete, querying
- database **administration**: database access authorization, database usage monitoring, database performance monitoring and optimization, etc.
- the **protection** of the database: *confidentiality* (protection against unauthorized data access), *integrity* (protection against inconsistent changes)

## 9. Types of Database Users

- database **administrators**
- database **designers**
  - schemas, restrictions, functions and procedures, optimizations
- data management **application users**
- **application programmers**

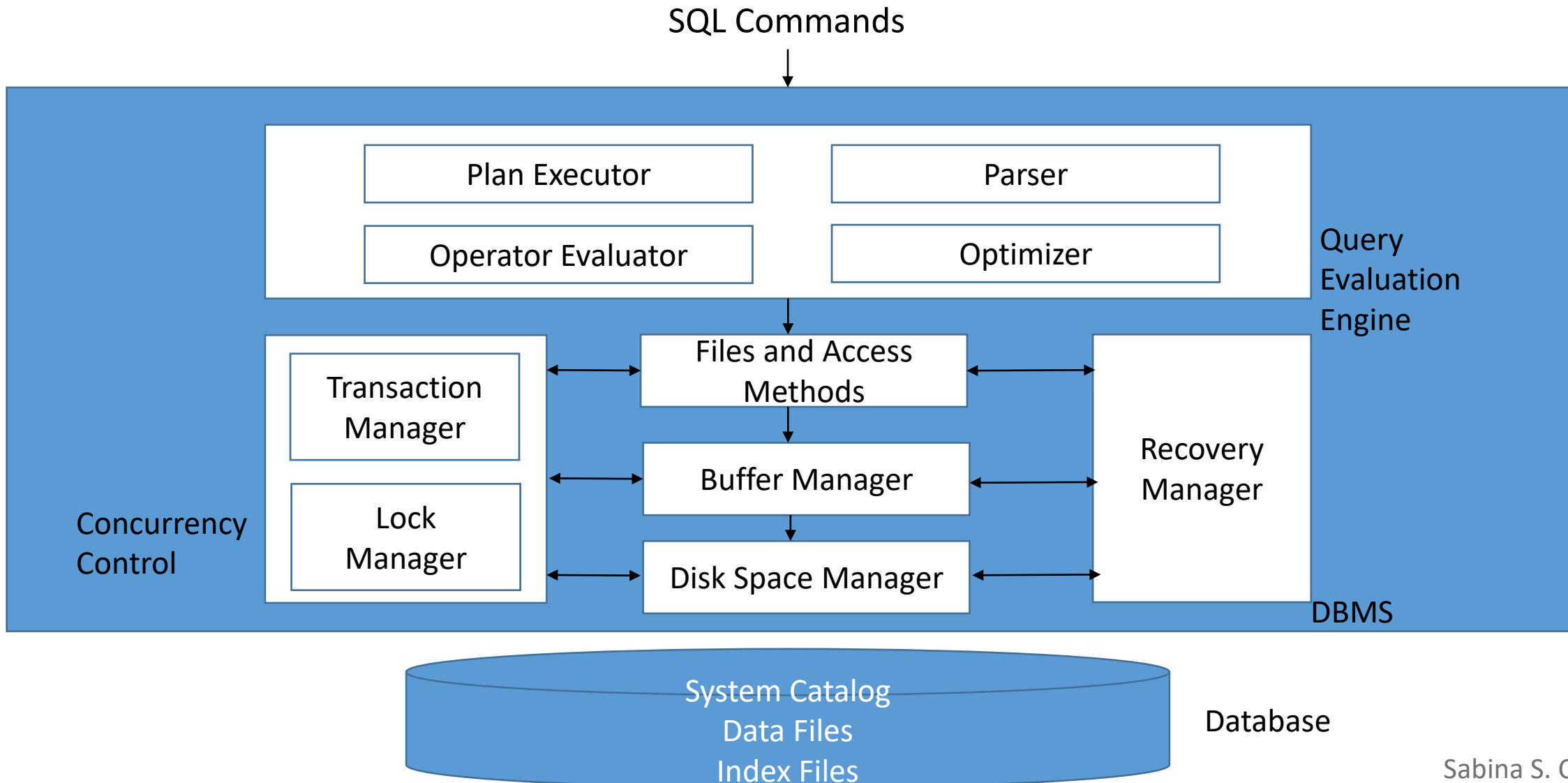
## 9. Types of Database Users

\* applications - developed in different languages / programming environments (Web, Java, .NET apps, etc.)

- executing an operation on a database (client / server technology):
  - app sends a command to the DB system (command written in SQL - Structured Query Language)
  - DB system executes command, sends answer back to the app

# 10. The Architecture of a DBMS

- [Ra02] proposes the following structure for a DBMS:



# 10. The Architecture of a DBMS

- SQL commands can come from different user interfaces (Web Forms, SQL interface, etc.), and can be included in applications written in various programming languages, e.g., Java, C#, etc.
- Optimizer
  - produces an efficient execution plan for query evaluation, taking into account storage information
- File & Access Methods, Buffer Manager, Disk Manager
  - abstraction of files, bringing pages from the disk into memory, managing disk space
- Transaction Manager, Lock Manager
  - concurrency control, monitoring lock requests, granting locks when database objects become available

# 10. The Architecture of a DBMS

- Recovery Manager
  - recovery after a crash

## 11. Using a DBMS - Advantages

- a DBMS can manage large collections of interrelated data
- applications are not managing database implementation details: an app sends a SQL command and receives the result set (the command is evaluated by the DBMS, which uses sophisticated data access programs)
- it allows systems to be developed in several stages (changing the database schema, changing the applications, developing new applications)
- it optimizes data access (very important for large collections of data; it stores and retrieves data efficiently)
- data access applications can be developed in a host of languages / programming environments

## 11. Using a DBMS - Advantages

- if data is always accessed through the system, it is up to date and correct (integrity constraints are automatically checked)
- database access control (for users with different roles)
- concurrent access management
- it enables data recovery (log)
- data import / export - various formats
- it provides data analysis tools (data mining)
- it reduces application development time

# References

- [Ta13] TÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ha96] HANSEN, G., HANSEN, J., Database Management And Design (2<sup>nd</sup> Edition), Prentice Hall, 1996
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3<sup>rd</sup> Edition,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

# Databases

Lecture 2  
The Relational Model

# The Relational Data Model

- proposed by Edgar Codd in 1970
  - 1981 – Turing Award
- the dominant data model today (represented by DBMSs like Oracle, SQL Server, etc.)
- advantages
  - **simple data representation**
  - queries, even complex ones, can be easily expressed, using a **simple, high-level language**
- even non-technical users can understand the contents of a database

# The Relational Data Model

1. relations
2. integrity constraints
3. relational databases
4. managing relational databases

# 1. Relations

- in an application, a **data collection** is used according to a **model**
- in the **relational model**, the data collection is organized as a set of **relations** (tables)
- a relation has a relation **schema** and a relation **instance**
- the relation instance can be thought of as a table
- the relation schema describes the table's column heads

# 1. Relations

- the schema specifies the relation's name, and the name and domain for each field
- the domain of a field is specified through its name and has a set of associated values
- schema example for the Students relation:

Students[*name*, *year\_of\_study*, *email*]

or notation:

Students(*name*: string, *year\_of\_study*: integer, *email*: string)

# 1. Relations

- instance example for the Students relation:

attributes (columns, fields)

name	year_of_study	email
Bratu Lucian	2	<a href="mailto:bl@cs.ubb.ro">bl@cs.ubb.ro</a>
Horvath Krisztina	2	<a href="mailto:hk@cs.ubb.ro">hk@cs.ubb.ro</a>
Nistor Anca	2	<a href="mailto:na@cs.ubb.ro">na@cs.ubb.ro</a>

records (rows, tuples)

# 1. Relations

- $\{A_1, A_2, \dots, A_n\}$  - a set of attributes
- $D_i = \text{Dom}(A_i) \cup \{\text{?}\}$ 
  - **domain of possible values for attribute  $A_i$**
  - the **undefined (null)** value is denoted here by  $\{\text{?}\}$ 
    - unknown / not applicable
    - can be used to check if a value has been assigned to an attribute (the attribute could have the *undefined* value)
    - this value doesn't have a certain data type; attribute values of different types can be compared against this value (numeric, string, etc.)
- relation of **arity** (degree) n:  $R \subseteq D_1 \times D_2 \times \dots \times D_n$
- $R[A_1, A_2, \dots, A_n]$  - the **relation schema**

# 1. Relations

- the relation can be stored in a table of the form:

R	A <sub>1</sub>	...	A <sub>j</sub>	...	A <sub>n</sub>
r <sub>1</sub>	a <sub>11</sub>	...	a <sub>1j</sub>	...	a <sub>1n</sub>
...	...	...	...	...	...
r <sub>i</sub>	a <sub>i1</sub>	...	a <sub>ij</sub>	...	a <sub>in</sub>
...	...	...	...	...	...
r <sub>m</sub>	a <sub>m1</sub>	...	a <sub>mj</sub>	...	a <sub>mn</sub>

where  $a_{ij} \in D_j, j = 1, \dots, n, i = 1, \dots, m.$

# 1. Relations

- the values of an attribute are **atomic** and **scalar**
- the rows in a table are **not ordered**
  - e.g., 2 representations of the same *Students* relation instance

name	year_of_study	email
Bratu Lucian	2	<a href="mailto:bl@cs.ubb.ro">bl@cs.ubb.ro</a>
Horvath Krisztina	2	<a href="mailto:hk@cs.ubb.ro">hk@cs.ubb.ro</a>
Nistor Anca	2	<a href="mailto:na@cs.ubb.ro">na@cs.ubb.ro</a>

name	year_of_study	email
Horvath Krisztina	2	<a href="mailto:hk@cs.ubb.ro">hk@cs.ubb.ro</a>
Bratu Lucian	2	<a href="mailto:bl@cs.ubb.ro">bl@cs.ubb.ro</a>
Nistor Anca	2	<a href="mailto:na@cs.ubb.ro">na@cs.ubb.ro</a>

- the records in a table are **distinct** - a relation is defined as a set of distinct tuples (however, DBMSs allow tables to contain duplicates)

# 1. Relations

- the **cardinality** of an instance is the number of tuples it contains

# 1. Relations

- a relation that is not well-designed:

Students[name, program, year\_of\_study, group]

name	program	year_of_study	group
Alexandra Mihai	CS ro	2	222
Vlad Dan	CS ro	2	222
Andreea Stancu	CS ro	2	226
Flaviu Pop	CS ro	3	226

- the association between group=222 and (program='CS ro', year\_of\_study=2) is stored 2 times, for the first 2 students
- the last 2 students are in the same group, but in different years - error
- to avoid such situations, relations must be **normalized**

## 2. Integrity Constraints

- **integrity constraints (restrictions)** are **conditions specified on the database schema**, restricting the data that can be stored in the database
- these constraints are **checked** when the data is **changed**; operations that violate the constraints are not allowed (e.g., introducing a student with a CNP identical to a different student's CNP, entering data of the wrong type in a column, etc.)
- examples: **domain** constraints, **key** constraints, **foreign key** constraints
- **domain** constraints - conditions that must be satisfied by every relation instance: a column's values belong to its associated domain

## 2. Integrity Constraints

- **key** constraint - a constraint stating that a **subset of the attributes** in a relation is a **unique identifier** for every tuple in the relation; this subset of attributes is minimal
- $K \subseteq \{A_1, A_2, \dots, A_n\}$  is a **key** for relation  $R[A_1, A_2, \dots, A_n]$  if:
  - the attributes in K can be used to identify every record in R
  - no subset of K has this property
- two different records are not allowed to have identical values in all the fields that constitute a key, hence specifying a key is a restriction for the database

## 2. Integrity Constraints

- e.g., in the Doctors[dep, did, lastname, firstname] relation, we can define the constraint: a doctor is uniquely identified by his / her department and did, i.e., 2 different doctors can have the same department or the same did, but not both; the key is the group of attributes **{dep, did}**
- e.g., Books[author, title, publisher, year]

author	title	publisher	year
C. J. Date	An Introduction to Database Systems	Addison-Wesley Publishing Comp.	2004
P. G. Wodehouse	Summer Lightning	Polirom	2003
Simon Singh	Fermat's Last Theorem	Humanitas	2012
Stephen Hawking	A Brief History of Time	Humanitas	2012

- a possible key is the group of attributes **{author, title, publisher, year}**; in such a case, the schema can be augmented to include another attribute (with distinct values; the latter can be automatically generated when records are added)

->

## 2. Integrity Constraints

bid	author	title	publisher	year
457	C. J. Date	An Introduction to Database Systems	Addison-Wesley Publishing Comp.	2004
4	P. G. Wodehouse	Summer Lightning	Polirom	2003
34	Simon Singh	Fermat's Last Theorem	Humanitas	2012
769	Stephen Hawking	A Brief History of Time	Humanitas	2012

- **{bid}** can now be chosen as the key; two different tuples will always have different values in the **bid** column
- the user must be careful not to define a key constraint that prevents the storage of correct tuple sets, e.g., if **{title}** is declared to be a key, the *Books* relation can't contain tuples describing different books with the same title
- **superkey** - a set of fields that contains the key (e.g., the set **{bid, title}** is a superkey)

## 2. Integrity Constraints

bid	author	title	publisher	year
457	C. J. Date	An Introduction to Database Systems	Addison-Wesley Publishing Comp.	2004
4	P. G. Wodehouse	Summer Lightning	Polirom	2003
34	Simon Singh	Fermat's Last Theorem	Humanitas	2012
769	Stephen Hawking	A Brief History of Time	Humanitas	2012

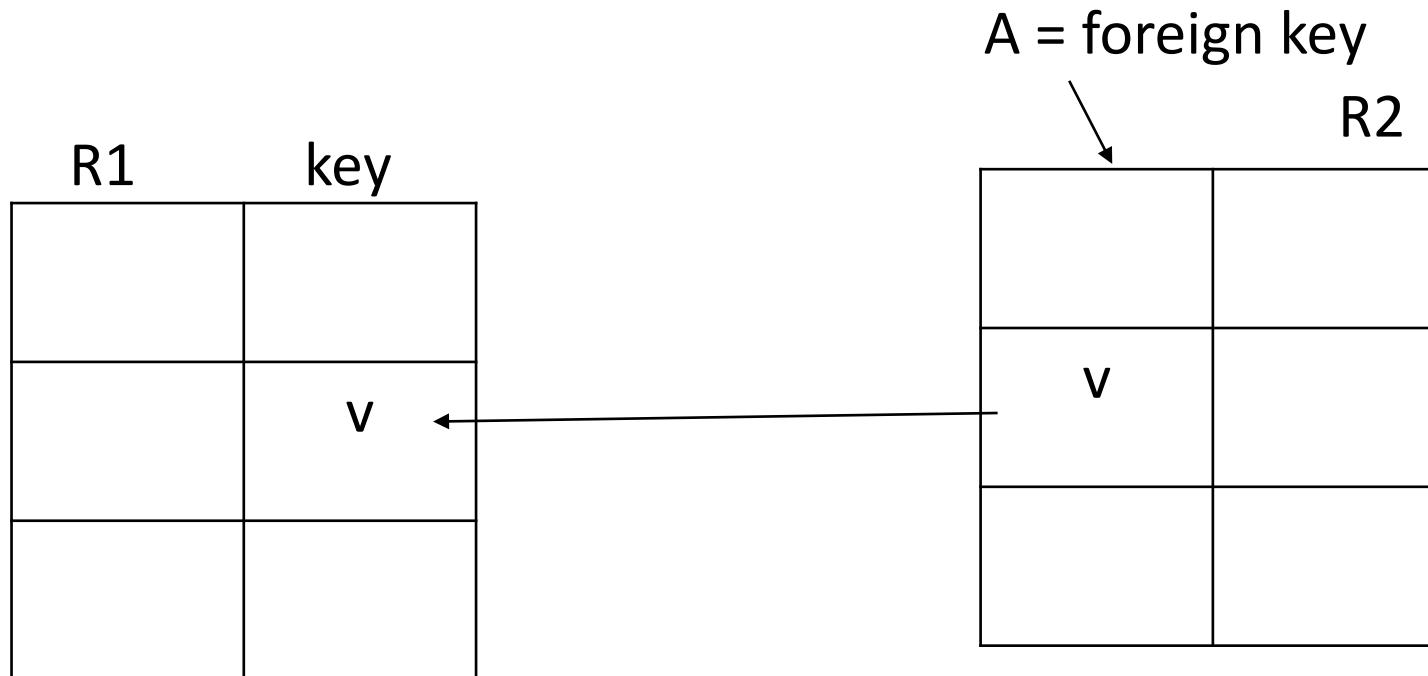
- non-key fields can contain identical values in different tuples: the relation can contain 2 different books from the same year, i.e., 2 tuples with *year* = 2004; or 3 different books written by the same author, i.e., 3 tuples with *author* = *Stephen Hawking*, etc.

## 2. Integrity Constraints

- a relation can have multiple keys: one key (an attribute or a group of attributes) is chosen as the **primary key**, while the others are considered **candidate keys**
- e.g., Schedule[day, hour, room, teacher, group, subject]  
with the schedule for a week
- the following sets of attributes can be chosen as keys:  
**{day, hour, room}; {day, hour, teacher}; {day, hour, group}**

## 2. Integrity Constraints

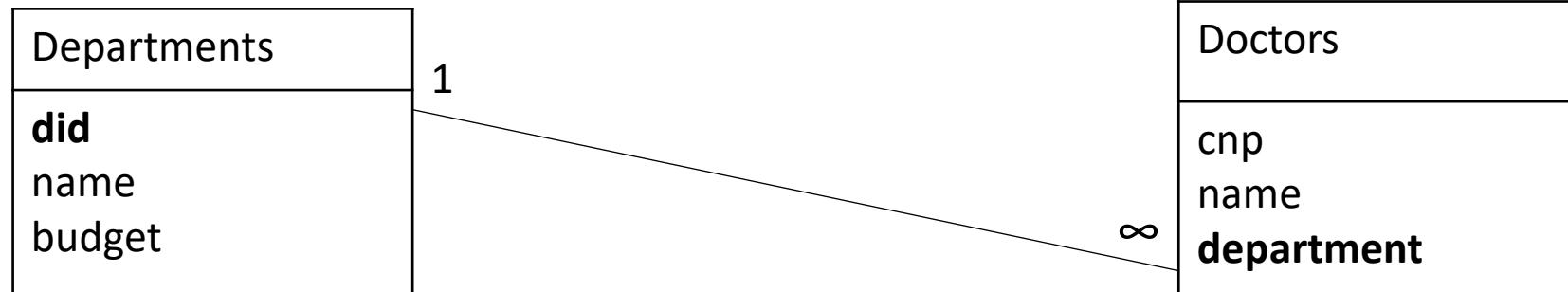
- **foreign key constraints** - the values of some attributes in a relation can also appear in another relation
- in the figure below, A is defined as a foreign key in R2; it refers to R1's key
- R1 and R2 need not be distinct



## 2. Integrity Constraints

- e.g., `Departments[did, name, budget]`

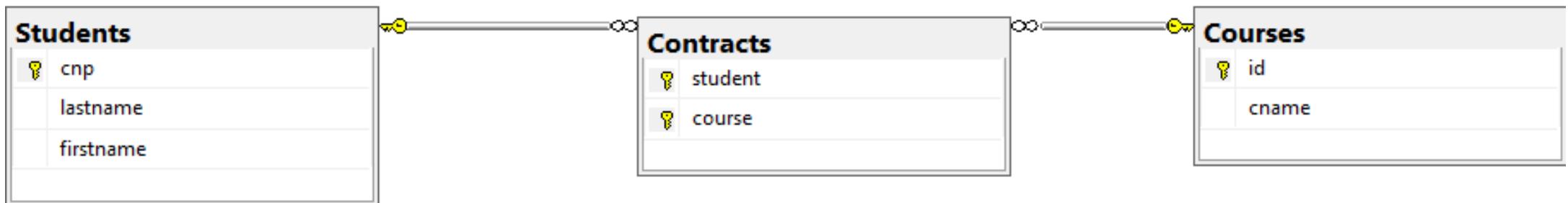
`Doctors[cnp, name, department]`



- establishing a link between *Departments* (as the parent relation) and *Doctors* (child relation): **Departments.did = Doctors.department**
- a department stored in the *Departments* relation and identified through a code corresponds to all doctors with the department's code; the *department* attribute in the *Doctors* relation is a foreign key
- a foreign key can be used to store **1:n** associations among entities: a department can correspond to several doctors, and a doctor can be associated with at most one department

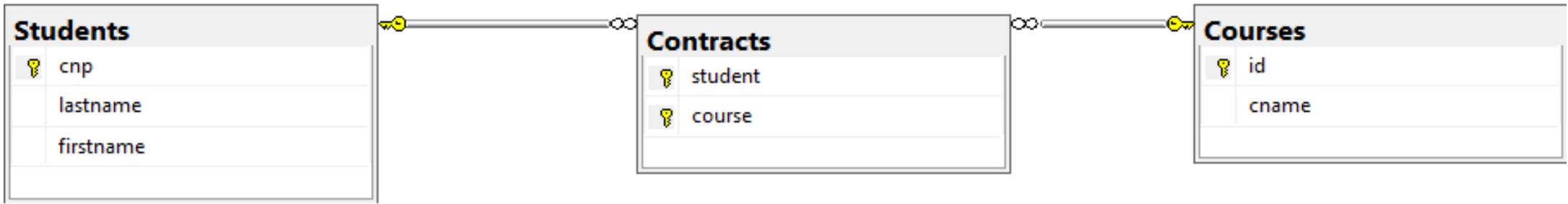
## 2. Integrity Constraints

- foreign keys can be used to store **m:n** associations among entities
- e.g., a student can enroll in several courses; multiple students can take a course; storing such associations requires an additional table



- any value that appears in the *student* field of the *Contracts* relation must appear in the *cnp* field of the *Students* relation, but there may be students who haven't enrolled in any courses yet (i.e., *cnp* values in *Students* that don't appear in the *student* column from *Contracts*)

## 2. Integrity Constraints



- the foreign key in the *Contracts* relation must have the same number of columns as the referenced key from *Students* (preferably, the primary key); the data types of the corresponding columns must be compatible, however, the column names can be different

## 2. Integrity Constraints

### enforcing integrity constraints

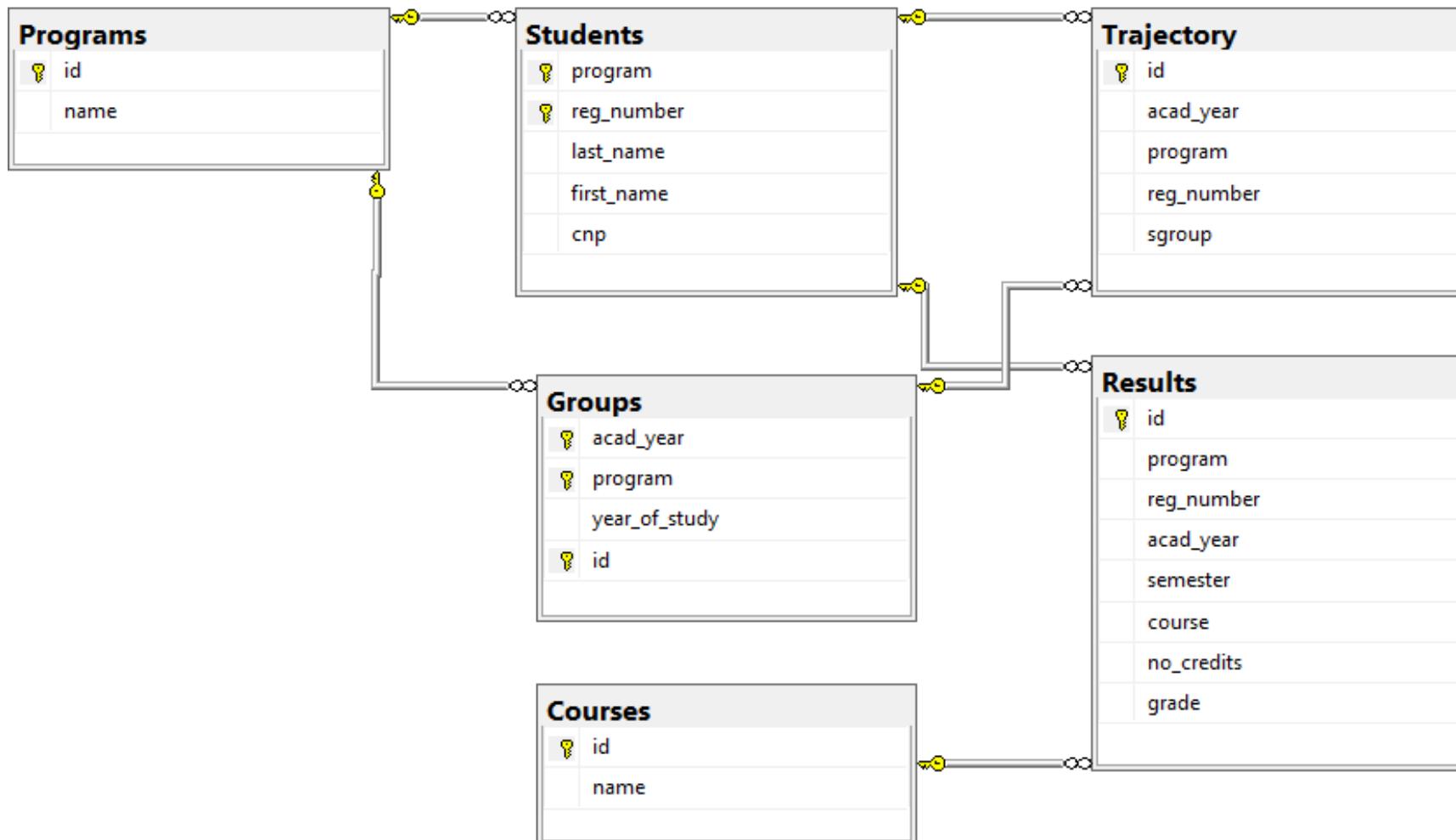
- when the data is changed, the DBMS rejects operations that violate the specified integrity constraints
    - e.g., entering a row in *Contracts* that has no corresponding student row in *Students*
  - in some cases, instead of rejecting the operation, the DBMS will make additional changes to the data, so in the end all integrity constraints are satisfied
    - e.g., when removing a *Students* row that has referencing *Contracts* rows, the operation can be:
      - rejected – the row is not deleted
      - cascaded, i.e., the *Students* row and the referencing rows in *Contracts* are deleted
- \* other options - see the CREATE TABLE statement in this lecture

### 3. Relational Databases

- **relational database**
  - collection of relations with distinct names
- **relational database schema**
  - collection of schemas for the relations in the database
- **database instance**
  - collection of relation instances, one / relational schema in the database schema
  - must be **legal**, i.e., it must satisfy all the specified integrity constraints

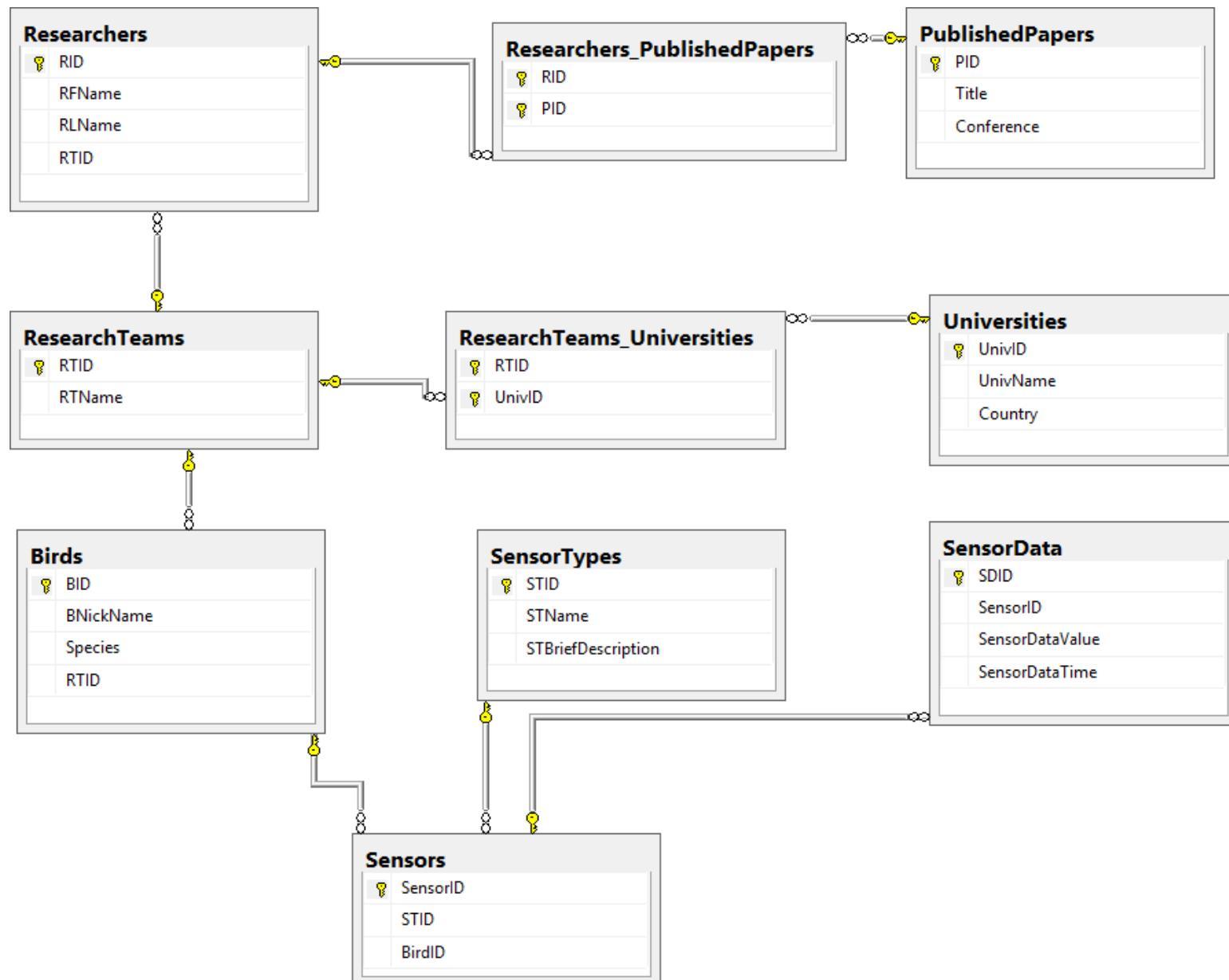
### 3. Relational Databases

- ex1:



# 3. Relational Databases

- ex2:



## 4. Managing Relational Databases with SQL

- **SQL** – Structured Query Language
- 1970 - E.F. Codd formalizes the relational model
- 1974 - the SEQUEL language (Structured English Query Language) is defined at IBM (in San Jose)
- 1976 - SEQUEL/2, a changed version of SEQUEL, is defined at IBM; following a revision, it becomes the SQL language
- 1986 - SQL adopted by ANSI (American National Standards Institute)
- 1987 - SQL adopted by ISO (International Standards Organization)
- versions (extensions): SQL-86, SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011, SQL:2016

## 4. Managing Relational Databases with SQL

- defining components
  - CREATE, ALTER, DROP (SQL DDL)
- managing and retrieving data
  - SELECT, INSERT, UPDATE, DELETE (SQL DML)
- managing transactions
  - START TRANSACTION, COMMIT, ROLLBACK

## 4. Managing Relational Databases with SQL

- CREATE TABLE – defines a new table

**CREATE TABLE table\_name**

**(column\_definition [, column\_definition] ... [, table\_restrictions])**

**column\_definition: column\_name data\_type[(length)] [DEFAULT value]  
[column\_restriction]**

```
CREATE TABLE Students
    (sid INT PRIMARY KEY,
     cnp CHAR(13),
     lastname VARCHAR(50),
     firstname VARCHAR(50) DEFAULT 'TBA',
     age INT CHECK (age >= 18))
```

- data type categories: numeric, string, date-time, etc.

\* we use the *age* attribute for simplicity; it is preferable to store the date of birth, as  
it doesn't change every year

## 4. Managing Relational Databases with SQL

- **restrictions associated with a column:**

- NOT NULL - can't have undefined values
- PRIMARY KEY - the column is a primary key
- UNIQUE - the values in the column are unique
- CHECK(condition) - the condition that must be satisfied by the column's values (simple conditions that evaluate to *true* or *false*)
- FOREIGN KEY REFERENCES parent\_table[(column\_name)] [ON UPDATE action] [ON DELETE action]

## 4. Managing Relational Databases with SQL

- **restrictions associated with a table:**

- PRIMARY KEY(column\_list) - defining the primary key for the table
- UNIQUE(column\_list) - the values in the column list are unique
- CHECK(condition) - the condition that must be satisfied by a row
- FOREIGN KEY(column\_list) REFERENCES parent\_table[(column\_list)][ON UPDATE action][ON DELETE action]

## 4. Managing Relational Databases with SQL

- **possible actions for a foreign key:**

- NO ACTION
  - the operation is not allowed if it violates integrity constraints
- SET NULL
  - the foreign key value is set to *null*
- SET DEFAULT
  - the foreign key value is set to the default value
- CASCADE
  - the delete / update is performed on the parent table, but it generates corresponding deletes / updates in the child table

```
CREATE TABLE Programs
```

```
(id SMALLINT PRIMARY KEY,  
name VARCHAR(70))
```

```
CREATE TABLE Students
```

```
(program SMALLINT REFERENCES Programs(id),  
reg_number CHAR(10),  
last_name CHAR(30),  
first_name CHAR(30),  
cnp CHAR(13) UNIQUE,  
PRIMARY KEY(program, reg_number))
```

```
CREATE TABLE Groups
```

```
(acad_year SMALLINT,  
program SMALLINT REFERENCES Programs(id),  
year_of_study SMALLINT,  
id CHAR(10),  
PRIMARY KEY (acad_year, program, id))
```

```
CREATE TABLE Courses
```

```
(id CHAR(10) PRIMARY KEY,  
name VARCHAR(70))
```

```
CREATE TABLE Trajectory
```

```
(id INT PRIMARY KEY IDENTITY(1,1),  
acad_year SMALLINT,  
program SMALLINT,  
reg_number CHAR(10),  
sgroup CHAR(10),  
FOREIGN KEY(program, reg_number) REFERENCES  
Students(program, reg_number),  
FOREIGN KEY(acad_year, program, sgroup)  
REFERENCES Groups(acad_year, program, id))
```

```
CREATE TABLE Results
```

```
(id INT PRIMARY KEY IDENTITY(1,1),  
program SMALLINT,  
reg_number CHAR(10),  
acad_year SMALLINT,  
semester SMALLINT,  
course CHAR(10) REFERENCES Courses(id),  
no_credits DECIMAL(3,1),  
grade SMALLINT,  
FOREIGN KEY(program, reg_number) REFERENCES  
Students(program, reg_number))
```

```
CREATE TABLE Universities
(UnivID SMALLINT PRIMARY KEY IDENTITY(1,1),
UnivName NVARCHAR(100) NOT NULL,
Country NVARCHAR(40))
```

```
CREATE TABLE ResearchTeams
(RTID SMALLINT IDENTITY(1,1),
RTName NVARCHAR(100) UNIQUE,
CONSTRAINT PK_ResearchTeams PRIMARY KEY(RTID))
```

```
CREATE TABLE ResearchTeams_Universities
(RTID SMALLINT FOREIGN KEY REFERENCES
ResearchTeams(RTID) ,
UnivID SMALLINT REFERENCES Universities(UnivID),
PRIMARY KEY(RTID, UnivID))
```

```
CREATE TABLE SensorTypes
(STID TINYINT PRIMARY KEY IDENTITY(1,1),
STName NVARCHAR(50) UNIQUE,
STBriefDescription NVARCHAR(300) DEFAULT 'TBW')
```

```
CREATE TABLE Birds
(BID INT PRIMARY KEY IDENTITY(1,1),
BNickName NVARCHAR(50),
Species VARCHAR(100),
RTID SMALLINT,
CONSTRAINT FK_Birds_ResearchTeams FOREIGN KEY(RTID)
REFERENCES ResearchTeams(RTID))
```

```
CREATE TABLE Sensors
(SensorID INT PRIMARY KEY IDENTITY(1,1),
STID TINYINT REFERENCES SensorTypes ON DELETE NO ACTION
ON UPDATE CASCADE,
BirdID INT REFERENCES Birds(BID))
```

```
CREATE TABLE SensorData
(SDID INT PRIMARY KEY IDENTITY(1,1),
SensorID INT REFERENCES Sensors(SensorID),
SensorDataValue REAL,
SensorDateTime DATETIME2)
```

```
CREATE TABLE Researchers
(RID SMALLINT PRIMARY KEY IDENTITY(1,1),
RFName NVARCHAR(90) NOT NULL,
RLName NVARCHAR(90) NOT NULL,
RTID SMALLINT REFERENCES ResearchTeams(RTID))
```

```
CREATE TABLE PublishedPapers
(PID INT PRIMARY KEY IDENTITY(1,1),
Title NVARCHAR(200),
Conference NVARCHAR(200))
```

```
CREATE TABLE Researchers_PublishedPapers
(RID SMALLINT REFERENCES Researchers(RID) ,
PID INT REFERENCES PublishedPapers(PID),
PRIMARY KEY(RID, PID))
```

# Managing Relational Databases with SQL

- ALTER TABLE – changes the structure of a defined table

**ALTER TABLE table\_name operation**

```
ALTER TABLE Students  
ADD FavSymphony VARCHAR(50)
```

- possible operations (differences among DBMSs)
  - add / change / remove a column
    - ADD column\_definition
    - {ALTER COLUMN | MODIFY} column\_definition
    - DROP COLUMN column\_name

# Managing Relational Databases with SQL

- add / remove a constraint
  - ADD [CONSTRAINT constraint\_name] PRIMARY KEY(column\_list)
  - ADD [CONSTRAINT constraint\_name] UNIQUE(column\_list)
  - ADD [CONSTRAINT constraint\_name] FOREIGN KEY (column\_list)  
REFERENCES table\_name[(column\_list)] [ON UPDATE action] [ON  
DELETE action]
  - DROP [CONSTRAINT] constraint\_name

# Managing Relational Databases with SQL

- **DROP TABLE** – destroys a table

**DROP TABLE table\_name**

```
DROP TABLE Students
```

- **Data Definition Language (DDL)** - subset of SQL used to create / remove / change components (e.g., tables)

# References

- [Ta13] TÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ha96] HANSEN, G., HANSEN, J., Database Management And Design (2<sup>nd</sup> Edition), Prentice Hall, 1996
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3<sup>rd</sup> Edition,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

# Databases

Lecture 3

Data Manipulation Language

Querying Relational Databases Using SQL

# SQL - INSERT

- the INSERT command – adding records

    INSERT INTO table\_name[(column\_list)] VALUES (value\_list)

    INSERT INTO table\_name[(column\_list)] subquery,

where *subquery* refers to a set of records (generated with the SELECT statement)

```
INSERT INTO Students (sid, cnp, lastname, firstname, age)
VALUES (1, '123456789012', 'Popescu', 'Maria', 20)
```

# SQL - UPDATE

- the UPDATE command – changing records

    UPDATE table\_name

        SET column\_name=expression [, column\_name=expression] ...

        [WHERE condition]

- the command changes the records in the table that satisfy the condition in the WHERE clause; if the WHERE clause is omitted, all the records in the table are changed; the values of the columns specified in SET are changed to the associated expressions' values

```
UPDATE Students
```

```
    SET age = age + 1
```

```
    WHERE cnp = '123456789012'
```

# SQL - DELETE

- the DELETE command – removing records

DELETE FROM table\_name

[WHERE condition]

- the command deletes the records in the table that satisfy the condition in the WHERE clause; if the WHERE clause is omitted, all the table's records are deleted

```
DELETE  
FROM Students  
WHERE lastname = 'Popescu'
```

- **Data Manipulation Language (DML)** - subset of SQL used to pose queries, to add / update / remove data

# Filter Conditions

- expression comparison\_operator expression
- expression [NOT] BETWEEN valmin AND valmax
- expression [NOT] LIKE pattern ("% - any substring, "\_" - one character)
- expression IS [NOT] NULL
- expression [NOT] IN (value [, value] ...)
- expression [NOT] IN (subquery)
- expression comparison\_operator {ALL | ANY} (subquery)
- [NOT] EXISTS (subquery)

# Filter Conditions

- elementary condition (previously described)
- (condition)
- NOT condition
- condition<sub>1</sub> AND condition<sub>2</sub>
- condition<sub>1</sub> OR condition<sub>2</sub>

# 3-Valued Logic

- truth values: *true, false, unknown*

	TRUE	FALSE	NULL
NOT	FALSE	TRUE	NULL
AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

# Querying Relational Databases Using SQL

- basic SQL query:

```
SELECT [DISTINCT] select-list  
FROM from-list  
WHERE qualification
```

- **select-list**

- list of (expressions involving) attributes from relations in the **from-list**

- **from-list**

- list of relation names; each of them can be followed by a range variable

- **qualification**

- selection conditions on the data from the relations in the **from-list**

- conditions ( $expr \ op \ expr$ , where  $op \in \{<, \leq, =, \geq, \neq\}$  and  $expr$  is an expression that can include attributes, constants, etc.) combined with the logical operators AND, OR, NOT

- basic SQL query:

```
SELECT [DISTINCT] select-list  
FROM from-list  
WHERE qualification
```

- the SELECT, FROM clauses - mandatory
- the WHERE clause - optional

- the conceptual evaluation strategy:

```
SELECT [DISTINCT] select-list  
FROM from-list  
WHERE qualification
```

- compute the cross product of tables in the **from-list**
- remove the rows that don't meet **qualification**
- eliminate unwanted columns, i.e., those that don't appear in the **select-list**
- if DISTINCT is specified, remove duplicates
  - by default, duplicates are not eliminated

- examples on the schema

Researchers(RID: integer, *Name*: string, *ImpactFactor*: integer, *Age*\*: integer)

Papers(PID: integer, *Title*: string, *Conference*: string)

AuthorContribution(RID: integer, PID: integer, Year: integer)

\* we use the *Age* attribute for simplicity; it is preferable to store the date of birth, as it doesn't change every year

- Find the names of researchers who have worked on the paper with PID = 307.

```
SELECT R.Name
FROM Researchers R, AuthorContribution A
WHERE R.RID = A.RID AND A.PID = 307
```

Researchers

RID	Name	ImpactFactor	Age
1	Popescu	10	30
2	Ionescu	10	40
4	Andreescu	5	24

AuthorContribution

RID	PID	Year
1	307	2011
1	200	2012
2	307	2011

- compute the cross product of tables *Researchers* and *AuthorContribution*

RID	Name	ImpactFactor	Age	RID	PID	Year
1	Popescu	10	30	1	307	2011
1	Popescu	10	30	1	200	2012
1	Popescu	10	30	2	307	2011
2	Ionescu	10	40	1	307	2011
2	Ionescu	10	40	1	200	2012
2	Ionescu	10	40	2	307	2011
4	Andreeescu	5	24	1	307	2011
4	Andreeescu	5	24	1	200	2012
4	Andreeescu	5	24	2	307	2011

- RID* appears in both *Researchers* and *AuthorContribution* => it must be qualified (e.g., in the WHERE clause)

- remove the rows in the cross product that don't satisfy the condition

R.RID = A.RID AND A.PID = 307

RID	Name	ImpactFactor	Age	RID	PID	Year
1	Popescu	10	30	1	307	2011
1	Popescu	10	30	1	200	2012
1	Popescu	10	30	2	307	2011
2	Ionescu	10	40	1	307	2011
2	Ionescu	10	40	1	200	2012
2	Ionescu	10	40	2	307	2011
4	Andreeescu	5	24	1	307	2011
4	Andreeescu	5	24	1	200	2012
4	Andreeescu	5	24	2	307	2011

- remove the rows in the cross product that don't satisfy the condition

$R.RID = A.RID \text{ AND } A.PID = 307$

RID	Name	ImpactFactor	Age	RID	PID	Year
1	Popescu	10	30	1	307	2011
1	Popescu	10	30	1	200	2012
1	Popescu	10	30	2	307	2011
2	Ionescu	10	40	1	307	2011
2	Ionescu	10	40	1	200	2012
2	Ionescu	10	40	2	307	2011
4	Andreeescu	5	24	1	307	2011
4	Andreeescu	5	24	1	200	2012
4	Andreeescu	5	24	2	307	2011

- remove the rows in the cross product that don't satisfy the condition

R.RID = A.RID AND A.PID = 307

RID	Name	ImpactFactor	Age	RID	PID	Year
1	Popescu	10	30	1	307	2011
1	Popescu	10	30	1	200	2012
1	Popescu	10	30	2	307	2011
2	Ionescu	10	40	1	307	2011
2	Ionescu	10	40	1	200	2012
2	Ionescu	10	40	2	307	2011
4	Andreeescu	5	24	1	307	2011
4	Andreeescu	5	24	1	200	2012
4	Andreeescu	5	24	2	307	2011

- remove the rows in the cross product that don't satisfy the condition

R.RID = A.RID AND A.PID = 307

RID	Name	ImpactFactor	Age	RID	PID	Year
1	Popescu	10	30	1	307	2011
2	Ionescu	10	40	2	307	2011

- remove the columns that don't appear in R.Name

Name
Popescu
Ionescu

- basic queries

Find the names and ages of all researchers. Eliminate duplicates.

```
SELECT DISTINCT R.Name, R.Age  
FROM Researchers R
```

Find the researchers with an impact factor > 3 (all the data about researchers).

```
SELECT R.RID, R.Name, R.ImpactFactor, R.Age  
FROM Researchers AS R  
WHERE R.ImpactFactor > 3  
-- SELECT *
```

Find the names of researchers who have published in the EDBT conference.

```
SELECT R.Name  
FROM Researchers R, AuthorContribution A, Papers P  
WHERE R.RID = A.RID AND A.PID = P.PID AND P.Conference =  
'EDBT'
```

Find the ids of researchers who have published in the EDBT conference.

```
SELECT A.RID  
FROM AuthorContribution A, Papers P  
WHERE A.PID = P.PID AND P.Conference = 'EDBT'
```

Find the names of researchers who have published at least one paper.

```
SELECT R.Name  
FROM Researchers R, AuthorContribution A  
WHERE R.RID = A.RID
```

Find the conferences that published Ionescu's papers.

```
SELECT P.Conference  
FROM Researchers R, AuthorContribution A, Papers P  
WHERE R.RID = A.RID AND A.PID = P.PID AND R.Name = 'Ionescu'  
* obs. There can be more than one researcher named Ionescu.
```

- expressions in SELECT

Compute an incremented impact factor for researchers who worked on two different papers in the same year.

```
SELECT R.Name, R.ImpactFactor + 1 AS NewIF  
FROM Researchers R, AuthorContribution A1, AuthorContribution  
A2  
WHERE R.RID = A1.RID AND R.RID = A2.RID  
AND A1.PID <> A2.PID  
AND A1.Year = A2.Year
```

- nested queries
  - the WHERE clause
- IN

Find the names of researchers who have worked on the paper with PID = 307.

```
SELECT R.Name  
FROM Researchers R  
WHERE R.RID IN  
(SELECT A.RID  
FROM AuthorContribution A  
WHERE A.PID = 307)
```

Find the names of researchers who have published in EDBT.

```
SELECT R.Name  
FROM Researchers R  
WHERE R.RID IN  
(SELECT A.RID  
FROM AuthorContribution A  
WHERE A.PID IN  
(SELECT P.PID  
FROM Papers P  
WHERE P.Conference = 'EDBT'  
)  
)
```

Find the names of researchers who haven't published in EDBT.

```
SELECT R.Name  
FROM Researchers R  
WHERE R.RID NOT IN  
(SELECT A.RID  
FROM AuthorContribution A  
WHERE A.PID IN  
(SELECT P.PID  
FROM Papers P  
WHERE P.Conference = 'EDBT'  
)  
)
```

- EXISTS

Find the names of researchers who have worked on the paper with PID = 307.

```
SELECT R.Name  
FROM Researchers R  
WHERE EXISTS (SELECT *  
                FROM AuthorContribution A  
                WHERE A.PID = 307 AND A.RID = R.RID)
```

- operators ANY and ALL

Find researchers whose IF is greater than the IF of some researcher called *Ionescu*.

```
SELECT R.RID  
FROM Researchers R  
WHERE R.ImpactFactor > ANY  
(SELECT R2.ImpactFactor  
FROM Researchers R2  
WHERE R2.Name = 'Ionescu')
```

**expression = ANY(subquery)  $\longleftrightarrow$  expression IN(subquery)**

```
SELECT R.Name  
FROM Researchers R  
WHERE R.RID = ANY  
(SELECT A.RID  
FROM AuthorContribution A  
WHERE A.PID = 300)
```

```
SELECT R.Name  
FROM Researchers R  
WHERE R.RID IN  
(SELECT A.RID  
FROM AuthorContribution A  
WHERE A.PID = 300)
```

Find researchers whose IF is greater than the IF of every researcher called *Ionescu*.

```
SELECT R.RID  
FROM Researchers R  
WHERE R.ImpactFactor > ALL  
(SELECT R2.ImpactFactor  
FROM Researchers R2  
WHERE R2.Name = 'Ionescu')
```

**expression <> ALL(subquery)  $\longleftrightarrow$  expression NOT IN(subquery)**

```
SELECT R.Name  
FROM Researchers R  
WHERE R.RID <> ALL  
(SELECT A.RID  
FROM AuthorContribution A  
WHERE A.PID = 300)
```

```
SELECT R.Name  
FROM Researchers R  
WHERE R.RID NOT IN  
(SELECT A.RID  
FROM AuthorContribution A  
WHERE A.PID = 300)
```

- union, intersection, set-difference

Find the names of researchers who have published in EDBT or IDEAS.

```
SELECT R.Name  
FROM Researchers R, AuthorContribution A, Papers P  
WHERE R.RID = A.RID AND A.PID = P.PID AND  
(P.Conference = 'IDEAS' OR P.Conference = 'EDBT')
```

Find the names of researchers who have published in EDBT and IDEAS.

\* Don't replace OR by AND!

```
SELECT R.Name  
FROM Researchers R, AuthorContribution A, Papers P  
WHERE R.RID = A.RID AND A.PID = P.PID AND  
(P.Conference = 'IDEAS' AND P.Conference = 'EDBT')
```

Find the names of researchers who have published in EDBT and IDEAS.

```
SELECT R.Name  
FROM Researchers R, AuthorContribution A1, Papers P1,  
AuthorContribution A2, Papers P2  
WHERE R.RID = A1.RID AND A1.PID = P1.PID AND  
P1.Conference = 'IDEAS' AND  
R.RID = A2.RID AND A2.PID = P2.PID AND  
P2.Conference = 'EDBT'
```

Find the names of researchers who have published in EDBT but not in IDEAS.

```
SELECT R.Name  
FROM Researchers R, AuthorContribution A, Papers P  
WHERE R.RID = A.RID AND A.PID = P.PID AND P.Conference = 'EDBT'  
AND A.RID NOT IN (SELECT A2.RID  
                   FROM AuthorContribution A2, Papers P2  
                   WHERE A2.PID = P2.PID AND P2.Conference = 'IDEAS' )
```

- the JOIN operators
- JOIN examples are described on the following relational database:

Students			Courses		Exams			
SID	Name	Group	CID	Name	StdId	CrsId	Grade	Credits
135	Alexandra	922	BD	Baze de date	135	BD	10	6
82	Paul	926	SGBD	Sisteme de Gestiune a Bazelor de Date	82	SGBD	10	6
294	Ştefania	925	DMBD	Data Mining in Big Data	135	SGBD	10	6

Students

SID	Name	Group
135	Alexandra	922
82	Paul	926
294	Ştefania	925

Exams

StdId	CrsId	Grade	Credits
135	BD	10	6
82	SGBD	10	6
135	SGBD	10	6

- find all the students' grades; include the students' names in the answer set

## 1. inner join: source1 [alias] [INNER] JOIN source2 [alias] ON condition

```
SELECT *
```

```
FROM Students S INNER JOIN Exams E ON S.SID = E.StdId
```

SID	Name	Group	StdId	CrsId	Grade	Credits
135	Alexandra	922	135	BD	10	6
135	Alexandra	922	135	SGBD	10	6
82	Paul	926	82	SGBD	10	6

Students

SID	Name	Group
135	Alexandra	922
82	Paul	926
294	Ştefania	925

Exams

StdId	CrsId	Grade	Credits
135	BD	10	6
82	SGBD	10	6
135	SGBD	10	6

- find all the students' grades; include students with no exams; the students' names must appear in the answer set

**2. left outer join:** source1 [alias] **LEFT [OUTER] JOIN** source2 [alias] **ON** condition

```
SELECT *
```

```
FROM Students S LEFT JOIN Exams E ON S.SID = E.StdId
```

SID	Name	Group	StdId	CrsId	Grade	Credits
135	Alexandra	922	135	BD	10	6
135	Alexandra	922	135	SGBD	10	6
82	Paul	926	82	SGBD	10	6
294	Ştefania	925	null	null	null	null

Courses

CID	Name
BD	Baze de date
SGBD	Sisteme de Gestiune a Bazelor de Date
DMBD	Data Mining in Big Data

Exams

StdId	CrsId	Grade	Credits
135	BD	10	6
82	SGBD	10	6
135	SGBD	10	6

find all the exams (including the names of the courses); include courses with no exams

### 3. right outer join: source1 [alias] **RIGHT [OUTER] JOIN** source2 [alias] **ON** condition

```
SELECT *
```

```
FROM Exams E RIGHT JOIN Courses C ON E.CrsId = C.CID
```

StdId	CrsId	Grade	Credits	CID	Name
135	BD	10	6	BD	Baze de date
135	SGBD	10	6	SGBD	Sisteme de Gestiune a Bazelor de Date
82	SGBD	10	6	SGBD	Sisteme de Gestiune a Bazelor de Date
null	null	null	null	DMBD	Data Mining in Big Data

Students

SID	Name	Group
135	Alexandra	922
82	Paul	926
294	Ştefania	925

Exams

StdId	CrsId	Grade	Credits
135	BD	10	6
82	SGBD	10	6
135	SGBD	10	6
737	SGBD	9	6

find all the exams; include students with no exams and grades given by mistake to nonexistent students; the result should also contain students' names

#### 4. full outer join: source1 [alias] **FULL [OUTER] JOIN** source2 [alias] **ON** condition

```
SELECT *
```

```
FROM Students S FULL JOIN Exams E ON S.SID = E.StdId
```

SID	Name	Group	StdId	CrsId	Grade	Credits
135	Alexandra	922	135	BD	10	6
135	Alexandra	922	135	SGBD	10	6
82	Paul	926	82	SGBD	10	6
294	Ştefania	925	null	null	null	null
null	null	null	737	SGBD	9	6

- other JOIN expressions

source1 [alias1] JOIN source2 [alias2] USING (column\_list)

source1 [alias1] NATURAL JOIN source2 [alias2]

source1 [alias1] CROSS JOIN source2 [alias2]

- subquery in the FROM clause

```
SELECT R.*  
FROM Researchers R INNER JOIN  
  (SELECT *  
   FROM AuthorContribution A  
   WHERE A.PID = 400) t  
ON R.RID = t.RID
```

- copy data from one table to another

```
INSERT INTO T2
```

```
SELECT * FROM T1
```

See seminar 2:

- range variables
- the LIKE operator
- the UNION [ALL], INTERSECT, EXCEPT operators
- joins with more than 2 tables
- aggregation operators

# References

- [Ta13] ȚÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ha96] HANSEN, G., HANSEN, J., Database Management And Design (2<sup>nd</sup> Edition), Prentice Hall, 1996
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3<sup>rd</sup> Edition, <http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems, <http://infolab.stanford.edu/~ullman/fcdb.html>
- [Ta03] ȚÂMBULEA, L., Baze de date, Litografiait Cluj-Napoca 2003

# Databases

Lecture 4

Querying Relational Databases Using SQL (II)

Functional Dependencies. Normal Forms

- GROUP BY, HAVING

```
SELECT [DISTINCT] select-list  
FROM from-list  
WHERE qualification  
GROUP BY grouping-list  
HAVING group-qualification
```

- optional GROUP BY clause
  - list of (expressions involving) columns used for grouping
- optional HAVING clause
  - group qualification conditions
- aggregation operators

COUNT, AVG, SUM, MIN, MAX

- GROUP BY, HAVING

```
SELECT [DISTINCT] select-list  
FROM from-list  
WHERE qualification  
GROUP BY grouping-list  
HAVING group-qualification
```

- group

- a collection of rows with identical values for the columns in **grouping-list**
- every row in the result of the query corresponds to a group

- GROUP BY, HAVING

```
SELECT [DISTINCT] select-list  
FROM from-list  
WHERE qualification  
GROUP BY grouping-list  
HAVING group-qualification
```

- **select-list**

- columns (that must appear in **grouping-list**)
- terms of the form *aggop(column) [AS NewName]*
  - e.g., MAX(R.ImpactFactor) AS MaxImpactFactor
  - *NewName* assigns a name to the column in the result table

- GROUP BY, HAVING

```
SELECT [DISTINCT] select-list  
FROM from-list  
WHERE qualification  
GROUP BY grouping-list  
HAVING group-qualification
```

- **group-qualification**

- expressions with a single value / group
- a column in **group-qualification** appears in **grouping-list** or as an argument to an aggregation operator
- records that meet **qualification** are partitioned into groups based on the values of the columns in **grouping-list**
- an answer row is generated for every group that meets **group-qualification**

Find the age of the youngest researcher for each impact factor.

```
SELECT R.ImpactFactor, MIN(R.Age)
```

```
FROM Researchers R
```

```
GROUP BY R.ImpactFactor
```

\* discussion: using the GROUP BY clause vs writing  $n$  queries, one for each of the  $n$  values of the impact factor, where  $n$  depends on the relation instance

Find the age of the youngest researcher who is at least 18 years old for each impact factor with at least 10 such researchers.

```
SELECT R.ImpactFactor, MIN(R.Age) AS MinAge
```

```
FROM Researchers R
```

```
WHERE R.Age >= 18
```

```
GROUP BY R.ImpactFactor
```

```
HAVING COUNT(*) >= 10
```

Find the average age of researchers who are at least 18 years old for each impact factor with at least 10 researchers.

```
SELECT R.ImpactFactor, AVG(R.Age) AS AvgAge  
FROM Researchers R  
WHERE R.Age >= 18  
GROUP BY R.ImpactFactor  
HAVING 9 < (SELECT COUNT(*)  
              FROM Researchers R2  
              WHERE R2.ImpactFactor = R.ImpactFactor)
```

## Problem

See lecture query: *Find the most prolific researchers, i.e., those with the largest number of papers, on schema  $\text{AuthorContribution}(\underline{\text{RID}}, \underline{\text{PID}}, \text{Year})$ .*

Does it compute the same result on  $\text{AuthorContribution}(\underline{\text{RID}}, \underline{\text{PID}}, \text{Year})$ ? If not, change it so the intended result is computed on this schema as well.

Find the name and age of the oldest researcher.

```
SELECT R.Name, MAX(R.Age)
```

```
FROM Researchers R
```

--**error**: if the SELECT clause contains an aggregation operator, then it must contain **only** aggregation operators, unless the query has a GROUP BY clause

Correct query:

```
SELECT R.Name, R.Age
```

```
FROM Researchers R
```

```
WHERE R.Age = (SELECT MAX(R2.Age)
```

```
        FROM Researchers R2)
```

- ORDER BY, TOP

Sort researchers by impact factor (in descending order) and age (in ascending order).

```
SELECT *
FROM Researchers R
ORDER BY R.ImpactFactor DESC, R.Age ASC
```

Retrieve the names and ages of the top 10 researchers ordered by name.

```
SELECT TOP 10 R.Name, R.Age
FROM Researchers R
ORDER BY R.Name
```

Find the top 25% researchers (all the data) ordered by age (descending).

```
SELECT TOP 25 PERCENT *
FROM Researchers R
ORDER BY R.Age DESC
```

Find the number of researchers for each impact factor. Order the result by the number of researchers.

```
SELECT R.ImpactFactor, COUNT(*) AS NoR
FROM Researchers R
GROUP BY R.ImpactFactor
ORDER BY NoR
```

- obs. intersection queries can be expressed with IN.

Find the names of researchers who have published in IDEAS and EDBT.

```
SELECT R.Name
FROM Researchers R INNER JOIN AuthorContribution A
ON R.RID = A.RID
INNER JOIN Papers P ON A.PID = P.PID
WHERE P.Conference = 'IDEAS' AND
R.RID IN (SELECT A2.RID
FROM AuthorContribution A2 INNER JOIN Papers P2
ON A2.PID = P2.PID
WHERE P2.Conference = 'EDBT'))
```

- obs. set-difference queries can be expressed with NOT IN.

Find the researchers who have published in EDBT but not in IDEAS.

```
SELECT A.RID  
FROM AuthorContribution A INNER JOIN Papers P ON A.PID = P.PID  
WHERE P.Conference = 'EDBT' AND  
A.RID NOT IN (SELECT A2.RID  
               FROM AuthorContribution A2 INNER JOIN Papers P2  
               ON A2.PID = P2.PID  
               WHERE P2.Conference = 'IDEAS')
```

Find researchers whose IF is greater than the IF of some researcher called *Ionescu*.

```
SELECT R.RID  
FROM Researchers R  
WHERE R.ImpactFactor >  
(SELECT MIN(R2.ImpactFactor)  
FROM Researchers R2  
WHERE R2.Name = 'Ionescu')
```

Find researchers whose IF is greater than the IF of every researcher called *Ionescu*.

```
SELECT R.RID  
FROM Researchers R  
WHERE R.ImpactFactor >  
(SELECT MAX(R2.ImpactFactor)  
FROM Researchers R2  
WHERE R2.Name = 'Ionescu')
```

- the SELECT statement:

```
SELECT [ { ALL  
          DISTINCT  
          TOP n [PERCENT] } ] {expr1 [AS column1] [, expr2 [AS column2]] ...} *  
FROM source1 [alias1] [, source2 [alias2]] ...  
[WHERE qualification]  
[GROUP BY grouping_list]  
[HAVING group_qualification]  
[ {UNION [ALL]  
    INTERSECT } SELECT_statement  
    EXCEPT ]  
[ ORDER BY { column1 } [ { ASC } ] [ { DESC } ] [, { column2 } [ { ASC } ] [ { DESC } ] ] ... ]
```

- non-procedural query
- SELECT statement evaluation: the result is a relation (table)
- data can be obtained from one or multiple data sources; a source can have an associated *alias*, used only in the SELECT statement
- various expressions are evaluated on the data (from the above-mentioned sources)
- a source column can be qualified with the source's name (or alias)

- a data source can be:
  1. table / view in the database
  2. (SELECT\_statement)
  3. join\_expression:
    - source1 [alias1] join\_operator source2 [alias2] ON join\_condition
    - (join\_expression)
- \* a join condition can be of the form:
  - elementary\_cond
  - (condition)
  - NOT condition
  - condition1 AND condition2
  - condition1 OR condition2

- \* an elementary join condition (elementary\_cond) among two data sources can be of the form:
  - [source1\_alias.]column1 relational\_operator [source2\_alias.]column2
  - expression1 relational\_operator expression2 (expression1 and expression2 use columns from different sources)

- the WHERE clause can contain filter and join conditions
- filter conditions:
  - expression relational\_operator expression
  - expression [NOT] BETWEEN valmin AND valmax
  - expression [NOT] LIKE pattern
  - expression IS [NOT] NULL
  - expression [NOT] IN (value [, value] ...)
  - expression [NOT] IN (subquery)
  - expression relational\_operator {ALL | ANY} (subquery)
  - [NOT] EXISTS (subquery)
- filter conditions can be:
  - elementary (described above)
  - composed with logical operators and parentheses

- obs: not all DBMSs support TOP
  - MySQL: SELECT ... LIMIT n
  - Oracle: SELECT ... WHERE ROWNUM <= n
- rules for building expressions:
  - operands: constants, columns, system functions, user functions
  - operators: corresponding to operands
- ordering records: the ORDER BY clause

- the SELECT statement - logical processing (Transact-SQL)

FROM

WHERE

GROUP BY

HAVING

SELECT

DISTINCT

ORDER BY

TOP

# Functional Dependencies. Normal Forms

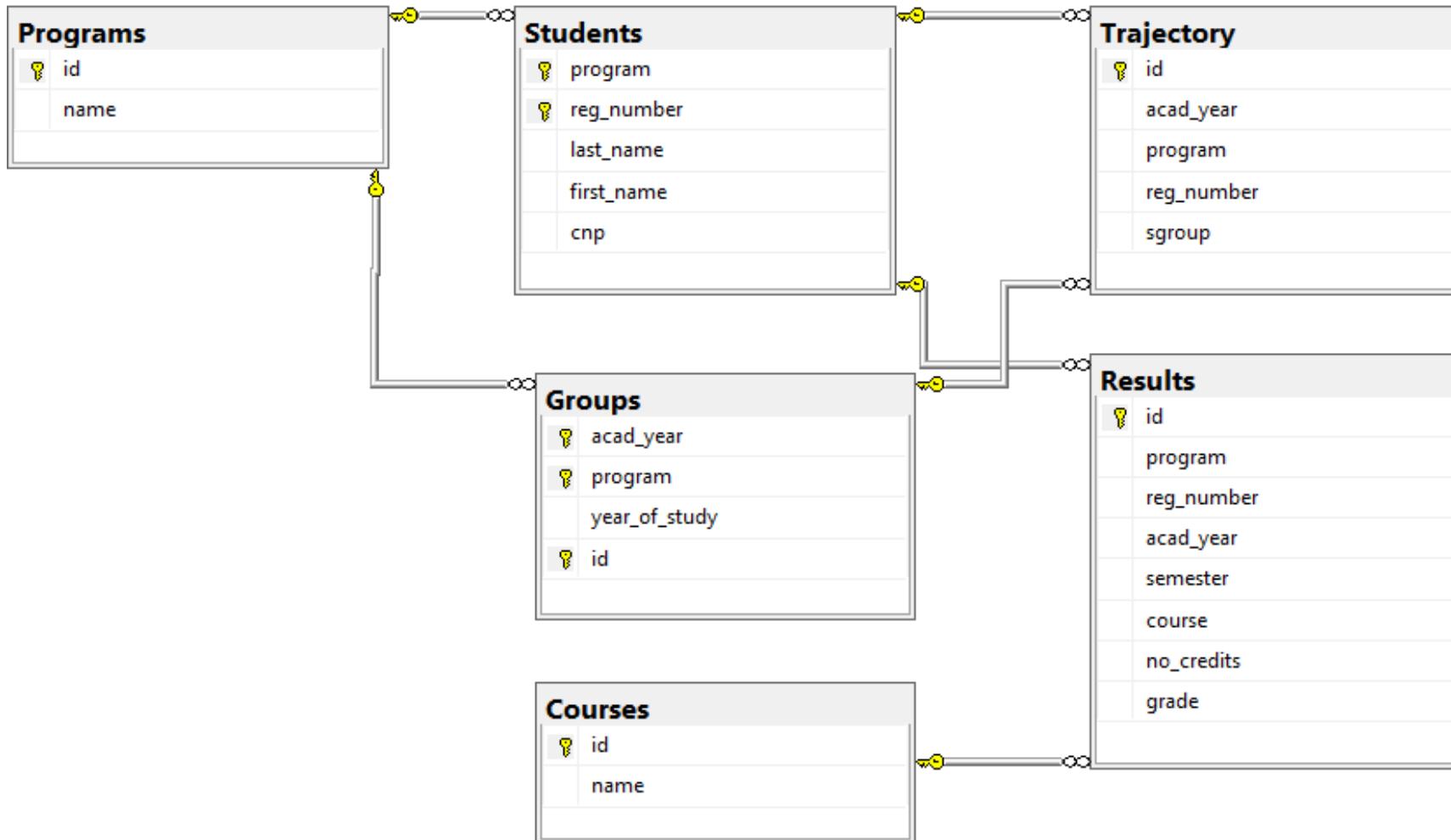
- there are multiple ways to model a data collection in the relational model (using tables)
- database design should be carried out in such a manner that subsequent queries and operations are performed as smoothly as possible, i.e.:
  - no additional tests are required when changing data
  - operations can be performed through SQL statements alone
- the above mentioned operations can be easily carried out when relations satisfy certain conditions (i.e., relations are in a certain normal form)

## data redundancy

- generates a series of problems; some can be tackled by replacing a relation with a collection of smaller relations; each of the latter contains a (strict) subset of attributes from the original relation

## Appendix - More SQL Queries

- more query examples on the database:



- A student's trajectory (academic year and group).

```
SELECT acad_year, sgroup  
FROM Trajectory  
WHERE program = 2 AND reg_number = '7654'
```

- A student's grades.

```
SELECT acad_year, course, no_credits, grade  
FROM Results  
WHERE program = 2 AND reg_number = '7654'
```

- Students who belonged to group 915 in the academic year 2017-2018.

```
SELECT last_name, first_name, s.reg_number  
FROM Students s INNER JOIN Trajectory t  
    ON s.program=t.program AND s.reg_number=t.reg_number  
WHERE acad_year=2017 AND sgroup='915'
```

```
SELECT last_name, first_name, s.reg_number  
FROM Students s INNER JOIN  
    (SELECT *  
     FROM Trajectory  
     WHERE acad_year=2017 AND sgroup='915') t  
    ON s.reg_number=t.reg_number AND s.program=t.program
```

- Students who belong to a group, but have no grades - in the academic year 2017-2018.

```
SELECT last_name, first_name
FROM Students AS s
WHERE EXISTS (SELECT *
               FROM Trajectory t
               WHERE acad_year=2017 AND t.program=s.program
                     AND t.reg_number=s.reg_number)
AND NOT EXISTS (SELECT *
                  FROM Results r
                  WHERE acad_year=2017 AND
                        s.program=r.program AND
                        s.reg_number=r.reg_number)
--ORDER BY last_name, first_name
```

- Students who belong to a group, but have no grades - in the academic year 2017-2018.

```
SELECT last_name, first_name
FROM (Students s INNER JOIN
      (SELECT *
       FROM Trajectory WHERE acad_year=2017) t
      ON s.program=t.program AND s.reg_number=t.reg_number
)
LEFT JOIN
      (SELECT *
       FROM Results
       WHERE acad_year=2017) r
      ON s.program=r.program AND s.reg_number=r.reg_number
WHERE grade IS NULL
```

- The number of students in the database.

```
SELECT COUNT(*) AS NoS  
FROM Students
```

- The number of students born on the same day, regardless of year and month.

```
SELECT SUBSTRING(cnp, 6, 2) AS day, COUNT(*) AS NoS  
FROM Students  
GROUP BY SUBSTRING(cnp, 6, 2)
```

- The grades of a given student (only the maximum grade is required for each course).

```
SELECT course, no_credits, MAX(grade) AS maxgrade  
FROM Results  
WHERE program = 3 AND reg_number='virtualstudent11'  
GROUP BY course, no_credits  
--ORDER BY course
```

- The grades of a given student (only the maximum grade is required for each course). Include the name of the course in the answer set.

```
SELECT id, name, no_credits, maxgrade AS grade
FROM Courses c INNER JOIN
  (SELECT course, no_credits, MAX(grade) AS maxgrade
  FROM Results
  WHERE program = 3 AND reg_number='virtualstudent11'
  GROUP BY course, no_credits) r
ON c.id = r.course
--ORDER BY name
```

- For each student name that appears at least 3 times, retrieve all students with that name.

```
SELECT *
FROM Students
WHERE last_name IN
  (SELECT last_name
   FROM Students
   GROUP BY last_name
   HAVING COUNT(*) >= 3)
--ORDER BY last_name, first_name
* rewrite the query without IN
```

- The number of students in each program and year of study in 2018.

```
SELECT g.program, g.year_of_study, COUNT(*) AS NoS  
FROM Trajectory t INNER JOIN Groups g  
ON t.program=g.program AND t.sgroup=g.id AND  
t.acad_year=g.acad_year  
WHERE t.acad_year=2018  
GROUP BY g.program, g.year_of_study
```

- The last name, first name, program (id), registration number, number of passed exams, number of credits, and gpa for each student with at least 30 credits in passed courses at the end of 2017.

```

SELECT last_name, first_name, s.program, s.reg_number, COUNT(*)
AS noc, SUM(no_credits) AS no_cr,
SUM(grade*no_credits)/SUM(no_credits) AS gpa
FROM Students s INNER JOIN
(SELECT program, reg_number, course, no_credits, MAX(grade)
AS grade
FROM Results
WHERE acad_year=2017 AND grade>=5
GROUP BY program, reg_number, course, no_credits
) r
ON s.program=r.program AND s.reg_number=r.reg_number
GROUP BY s.program, s.reg_number, last_name, first_name
HAVING SUM(no_credits)>=30
ORDER BY 3,1,2

```

- For every course, the number of grades and the number of passing grades in 2017.

\* a. MySQL \*

```
SELECT name, COUNT(*) AS nrg, SUM(IF(r.grade >= 5, 1, 0))  
AS nrpg  
FROM Courses c INNER JOIN  
(SELECT * FROM Results WHERE acad_year = 2017) r  
ON c.id = r.course  
GROUP BY name  
--ORDER BY 1
```

- For every course, the number of grades and the number of passing grades in 2017.

\* b. Oracle \*

```
SELECT name, COUNT(*) AS nrg, SUM(CASE WHEN (r.grade >= 5) THEN  
1 ELSE 0 END) AS nrpg  
FROM Courses c INNER JOIN  
(SELECT * FROM Results WHERE acad_year = 2017) r  
ON c.id = r.course  
GROUP BY name  
--ORDER BY 1
```

- \* The grade average for each group in 2017. \*

### a. Maximum & passing grades in 2017.

```
CREATE VIEW studgrades AS
SELECT program, reg_number, course, no_credits, MAX(grade) AS
mgrade
FROM Results
WHERE acad_year=2017 AND grade>=5
GROUP BY program, reg_number, course, no_credits
```

### b. Students' grade average (in 2017).

```
CREATE VIEW studavg AS
SELECT program, reg_number,
SUM(no_credits*mgrade)/SUM(no_credits) AS gradeavg
FROM studgrades
GROUP BY program, reg_number
HAVING SUM(no_credits) >= 30
```

- \* The grade average for each group in 2017. \*

### c. Final query:

```
SELECT sgroup, AVG(gradeavg) AS gravg  
FROM  
  (SELECT program, reg_number, sgroup  
   FROM Trajectory  
   WHERE acad_year=2017) t  
   INNER JOIN studavg AS s  
     ON t.reg_number=s.reg_number AND t.program=s.program  
GROUP BY sgroup
```

- what happens if two groups in different programs have the same id?

- \* The grade average for each program in 2017. \*

```
CREATE VIEW programsavg AS  
SELECT id, name, AVG(gradeavg) AS pravg  
FROM Programs p INNER JOIN studavg s ON p.id=s.program  
GROUP BY id, name
```

```
SELECT * FROM programsavg
```

# References

- [Ta13] TÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ha96] HANSEN, G., HANSEN, J., Database Management And Design (2<sup>nd</sup> Edition), Prentice Hall, 1996
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3<sup>rd</sup> Edition,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

# Databases

Lecture 5

Functional Dependencies. Normal Forms (II)

- *simple attribute*
- *composite attribute* - a set of attributes in a relation (with at least two attributes)
- in some applications, attributes (simple or composite) can take on multiple values for a record in the relation - *repeating attributes*
- when defining a relation (in the relational model), attribute values must be scalar, atomic (they cannot be further decomposed)

Example 2. Consider the relation:

STUDENT [NAME, BIRTHYEAR, GROUP, COURSE, GRADE]

- key: NAME
- *composite repeating attribute*: the pair {COURSE, GRADE}
- this relation could store the following values:

NAME	BIRTHYEAR	GROUP	COURSE	GRADE
Ioana	2000	921	Databases Distributed Operating Systems Probabilities and Statistics	10 9.99 10
Vasile	2000	925	Databases Distributed Operating Systems Probabilities and Statistics Logical and Functional Programming	10 9.98 10 10

Example 3. Consider the relation:

BOOK [BId, AuthorsNames, Title, Publisher, PublishingYear, Keywords]

- key: BId
- *simple repeating attributes*: AuthorsNames, Keywords
- the BId attribute
  - could have an actual meaning (an id could be associated with each book)
  - could be introduced as a key (with distinct values that are automatically generated)

- repeating attributes cannot be used in the relational model, hence they are to be avoided, without losing data
- let  $R[A]$  be a relation;  $A$  - the set of attributes
- let  $\alpha$  be a repeating attribute in  $R$  (simple or composite)
- $R$  can be decomposed into 2 relations, such that  $\alpha$  is not a repeating attribute anymore
- if  $K$  is a key in  $R$ , the two relations into which  $R$  is decomposed are:

$$\begin{aligned} R'[K \cup \alpha] &= \Pi_{K \cup \alpha}(R) \\ R''[A - \alpha] &= \Pi_{A-\alpha}(R) \end{aligned}$$

Example 4. The STUDENT relation from example 2 is decomposed into the following 2 relations:

GENERAL\_DATA [NAME, BIRTHYEAR, GROUP],

RESULTS [NAME, COURSE, GRADE].

- the values in these relations are:

NAME	BIRTHYEAR	GROUP
Ioana	2000	921
Vasile	2000	925

NAME	COURSE	GRADE
Ioana	Databases	10
Ioana	Distributed Operating Systems	9.99
Ioana	Probabilities and Statistics	10
Vasile	Databases	10
Vasile	Distributed Operating Systems	9.98
Vasile	Probabilities and Statistics	10
Vasile	Logical and Functional Programming	10

Example 5. The BOOK relation in example 3 is decomposed into the following 3 relations (there are two repeating attributes in the BOOK relation):

BOOKS [BId, Title, Publisher, PublishingYear],

AUTHORS [BId, AuthorName],

KEYWORDS [BId, Keyword].

- is a book is not associated with any authors / keywords, it must have one corresponding tuple in AUTHORS / KEYWORDS, with the second attribute set to *null*; in the absence of such tuples, the BOOK relation can't be obtained from the three relations using just the natural join (outer join operators are required)

Definition. A relation is in the first normal form (1NF) if and only if it doesn't have any repeating attributes.

- obs. some DBMSs can manage non-1NF relations (e.g., Oracle)

- the following 3 relational normal forms use a very important notion: the *functional dependency* among subsets of attributes
- the database administrator is responsible with determining the functional dependencies; these depend on the nature, the semantics of the data stored in the relation

->

Definition. Let  $R[A_1, A_2, \dots, A_n]$  be a relation and  $\alpha, \beta$  two subsets of attributes of  $R$ . The (simple or composite) attribute  $\alpha$  functionally determines attribute  $\beta$  (simple or composite), notation:

$$\alpha \rightarrow \beta$$

if and only if every value of  $\alpha$  in  $R$  is associated with a precise, unique value for  $\beta$  (this association holds throughout the entire existence of relation  $R$ ); if an  $\alpha$  value appears in multiple rows, each of these rows will contain the same value for  $\beta$ :

$$\Pi_\alpha(r) = \Pi_\alpha(r') \text{ implies } \Pi_\beta(r) = \Pi_\beta(r')$$

- in the dependency  $\alpha \rightarrow \beta$ ,  $\alpha$  is the *determinant*, and  $\beta$  is the *dependent*
- the functional dependency can be regarded as a property (restriction) that must be satisfied by the database throughout its existence: values can be added / changed in the relation only if the functional dependency is satisfied

- if a relation contains a functional dependency, associations among values can be stored multiple times (data redundancy)
- to describe some of the problems that can arise in this context, we'll consider the EXAM relation, storing students' final exam results

Example 6. EXAM [StudentName, Course, Grade, FacultyMember]

- key: {StudentName, Course}
- a course is associated with one faculty member, a faculty member can be associated with several courses; the following restriction (functional dependency) must be satisfied:  $\{Course\} \rightarrow \{FacultyMember\}$

EXAM	StudentName	Course	Grade	FacultyMember
1	Pop Ioana	Computer Networks	10	Matei Ana
2	Vlad Ana	Operating Systems	10	Simion Bogdan
3	Vlad Ana	Computer Networks	9.98	Matei Ana
4	Dan Andrei	Computer Networks	10	Matei Ana
5	Popescu Alex	Operating Systems	9.99	Simion Bogdan

- if the relation contains such a functional dependency, the following problems can arise (some of them need to be solved through additional programming efforts - executing a SQL command is not enough):
  - wasting space: the same associations are stored multiple times, e.g., the one among *Computer Networks* and *Matei Ana* is stored 3 times;
  - update anomalies: if the *FacultyMember* is changed for course  $c$ , the change must be carried out in all associations involving course  $c$  (without knowing how many such associations exist), otherwise the database will contain errors (it will be inconsistent); if the *FacultyMember* value is changed in the 2<sup>nd</sup> record, but not in the 5<sup>th</sup> record, the operation will introduce an error in the relation (the data will be incorrect);
  - insertion anomalies: one cannot specify the *FacultyMember* for a course  $c$ , unless there's at least one student with a grade in course  $c$ ;

- deletion anomalies: when some records are deleted, data that is not intended to be removed can be deleted as well; e.g., if records 1, 3 and 4 are deleted, the association among *Course* and *FacultyMember* is also removed from the database
- the functional dependency among sets of attributes is what causes the previous anomalies; to eliminate these problems, the associations (dependencies) among values should be kept once in a separate relation, therefore, the initial relation should be decomposed (via a good decomposition - data should not be lost or added); such a decomposition is performed in the database design phase, when functional dependencies can be identified

Obs. The following simple properties for functional dependencies can be easily demonstrated:

1. If  $K$  is a key of  $R[A_1, A_2, \dots, A_n]$ , then  $K \rightarrow \beta$ ,  $\forall \beta$  a subset of  $\{A_1, A_2, \dots, A_n\}$ .
  - such a dependency is always true, hence it will not be eliminated through decompositions

2. If  $\beta \subseteq \alpha$ , then  $\alpha \rightarrow \beta$  - *trivial functional dependency (reflexivity)*.

$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \Rightarrow \begin{matrix} \Pi_\beta(r_1) = \Pi_\beta(r_2) \\ \beta \subseteq \alpha \end{matrix} \Rightarrow \alpha \rightarrow \beta$$

3. If  $\alpha \rightarrow \beta$ , then  $\gamma \rightarrow \beta$ ,  $\forall \gamma$  with  $\alpha \subset \gamma$ .

$$\Pi_\gamma(r_1) = \Pi_\gamma(r_2) \Rightarrow \begin{matrix} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \\ \alpha \subset \gamma, prop. 2 \end{matrix} \Rightarrow \begin{matrix} \Pi_\beta(r_1) = \Pi_\beta(r_2) \\ \alpha \rightarrow \beta \end{matrix} \Rightarrow \gamma \rightarrow \beta$$

Obs. The following simple properties for functional dependencies can be easily demonstrated:

4. If  $\alpha \rightarrow \beta$  and  $\beta \rightarrow \gamma$ , then  $\alpha \rightarrow \gamma$  - *transitivity*.

$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \xrightarrow{\alpha \rightarrow \beta} \Pi_\beta(r_1) = \Pi_\beta(r_2) \xrightarrow{\beta \rightarrow \gamma} \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \Rightarrow \alpha \rightarrow \gamma$$

5. If  $\alpha \rightarrow \beta$  and  $\gamma$  a subset of  $\{A_1, \dots, A_n\}$ , then  $\alpha\gamma \rightarrow \beta\gamma$ , where  $\alpha\gamma = \alpha \cup \gamma$ .

$$\Pi_{\alpha\gamma}(r_1) = \Pi_{\alpha\gamma}(r_2) \Rightarrow \left| \begin{array}{l} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \Rightarrow \Pi_\beta(r_1) = \Pi_\beta(r_2) \\ \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \end{array} \right| \Rightarrow \Pi_{\beta\gamma}(r_1) = \Pi_{\beta\gamma}(r_2)$$

Definition. An attribute A (simple or composite) is said to be *prime* if there is a key K and  $A \subseteq K$  ( $K$  can be a composite key; A can itself be a key). If an attribute isn't included in any key, it is said to be *non-prime*.

Definition. Let  $R[A_1, A_2, \dots, A_n]$  be a relation, and let  $\alpha, \beta$  be two subsets of attributes of  $R$ . Attribute  $\beta$  is *fully functionally dependent on  $\alpha$*  if:

- $\beta$  is functionally dependent on  $\alpha$  (i.e.,  $\alpha \rightarrow \beta$ ) and
- $\beta$  is not functionally dependent on any proper subset of  $\alpha$ , i.e.,  $\forall \gamma \subset \alpha, \gamma \rightarrow \beta$  does not hold.

Definition. A relation is in the second normal form (2NF) if and only if:

1. it is in the first normal form and
2. every (simple or composite) non-prime attribute is fully functionally dependent on every key of the relation.

- obs. Let  $R$  be a 1NF relation that is not 2NF. Then  $R$  has a composite key (and a functional dependency  $\alpha \rightarrow \beta$ , where  $\alpha$  (simple or composite) is a proper subset of a key and  $\beta$  is a non-prime attribute).
- decomposition
  - relation  $R[A]$  ( $A$  - the set of attributes),  $K$  - a key
  - $\beta$  non-prime,  $\beta$  functionally dependent on  $\alpha$ ,  $\alpha \subset K$  ( $\beta$  is functionally dependent on a proper subset of attributes from a key)
  - the  $\alpha \rightarrow \beta$  dependency can be eliminated if  $R$  is decomposed into the following 2 relations:

$$R'[\alpha \cup \beta] = \Pi_{\alpha \cup \beta}(R)$$

$$R''[A - \beta] = \Pi_{A - \beta}(R)$$

- we'll analyze the relation from Example 6

EXAM[StudentName, Course, Grade, FacultyMember]

- key: {StudentName, Course}
- the functional dependency  $\{Course\} \rightarrow \{FacultyMember\}$  holds => attribute *FacultyMember* is not fully functionally dependent on a key, hence the EXAM relation is not in 2NF
- this dependency can be eliminated if EXAM is decomposed into the following 2 relations:

RESULTS[StudentName, Course, Grade]

COURSES[Course, FacultyMember]

Example 7. Consider the following relation, storing students' learning contracts:

CONTRACTS[LastName, FirstName, CNP, CourseId, CourseName]

- key: {CNP, CourseId}
- functional dependencies:  $\{CNP\} \rightarrow \{LastName, FirstName\}$ ,  
 $\{CourseId\} \rightarrow \{CourseName\}$
- to eliminate these dependencies, the relation is decomposed into the following three relations:

STUDENTS[CNP, LastName, FirstName]

COURSES[CourseId, CourseName]

LEARNING\_CONTRACTS[CNP, CourseId]

- the notion of *transitive dependency* is required for the third normal form
- Definition. An attribute  $Z$  is transitively dependent on an attribute  $X$  if  $\exists Y$  such that  $X \rightarrow Y, Y \rightarrow Z, Y \rightarrow X$  does not hold (and  $Z$  is not in  $X$  or  $Y$ ).

Definition. A relation is in the third normal form (3NF) if and only if it is in the second normal form and no non-prime attribute is transitively dependent on any key in the relation.

Another definition: A relation  $R$  is in the third normal form (3NF) iff, for every non-trivial functional dependency  $X \rightarrow A$  that holds over  $R$ :

- $X$  is a superkey, or
- $A$  is a prime attribute.

Example 8. The BSc examination results are stored in the relation:

BSC\_EXAM [StudentName, Grade, Supervisor, Department]

- the relation stores the supervisor and the department in which she works
- since the relation contains data about students (i.e., one row per student), *StudentName* can be chosen as the key
- the following functional dependency holds:  $\{Supervisor\} \rightarrow \{Department\} \Rightarrow$  the relation is not in 3NF
- to eliminate this dependency, the relation is decomposed into the following 2 relations:

RESULTS [StudentName, Grade, Supervisor]

SUPERVISORS [Supervisor, Department]

Example 9. The following relation stores addresses for a group of people:

ADDRESSES [CNP, LastName, FirstName, ZipCode, City, Street, No]

- key: {CNP}
- identified dependency:  $\{ZipCode\} \rightarrow \{City\}$  (can you identify another dependency in this relation?)
- since this dependency holds, relation ADDRESSES is not in 3NF, therefore it must be decomposed:

ADDRESSES'[CNP, LastName, FirstName, ZipCode, Street, No]

ZIPCODES[ZipCode, City]

# References

- [Ta13] ȚÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ha96] HANSEN, G., HANSEN, J., Database Management And Design (2<sup>nd</sup> Edition), Prentice Hall, 1996
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3<sup>rd</sup> Edition, <http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems, <http://infolab.stanford.edu/~ullman/fcdb.html>
- [Ta03] ȚÂMBULEA, L., Baze de date, Litografiat Cluj-Napoca 2003

# Databases

Lecture 6

Functional Dependencies. Normal Forms (III)

\* See recap lecture example with schema decomposition.

Example 10. The following relation stores the exam session schedule:

**EX\_SCHEDULE**[Date, Hour, Faculty\_member, Room, Group]

- the following restrictions are expressed via key definitions and functional dependencies:

1. a group of students has at most one exam per day

=>  $\{Date, Group\}$  is a key

2. on a certain date and time, a faculty member has at most one exam

=>  $\{Faculty\_member, Date, Hour\}$  is a key

3. on a certain date and time, there is at most one exam in a room

=>  $\{Room, Date, Hour\}$  is a key

4. a faculty member doesn't change the room in a day

=> the following dependency holds:  $\{Faculty\_member, Date\} \rightarrow \{Room\}$

- all attributes appear in at least one key, i.e., there are no non-prime attributes
- given the normal forms' definitions specified thus far, the relation is in 3NF
- objective: eliminate the  $\{Faculty\_member, Date\} \rightarrow \{Room\}$  functional dependency

Definition. A relation is in the Boyce-Codd normal form (BCNF) if and only if every determinant (for a functional dependency) is a key (informal definition - simplifying assumption: determinants are not too big; only non-trivial functional dependencies are considered).

- to eliminate the functional dependency, the original relation must be decomposed into:

EX\_SCHEDULE'[Date, Hour, Faculty\_member, Group],

ROOM\_ALLOCATION[Faculty\_member, Date, Room]

- these relations don't contain other functional dependencies, i.e., they are in BCNF
- however, the key associated with the 3<sup>rd</sup> constraint,  $\{Room, Date, Hour\}$ , does not exist anymore
- if this constraint is to be kept, it needs to be checked in a different manner (e.g., through the program)

- $R[A]$  - a relation
  - $F$  - a set of functional dependencies
  - $\alpha$  – a subset of attributes
- 
- problems
    - I. compute the closure of  $F$ :  $F^+$
    - II. compute the closure of a set of attributes under a set of functional dependencies, e.g., the closure of  $\alpha$  under  $F$ :  $\alpha^+$
    - III. compute the minimal cover for a set of dependencies

- $R[A]$  - a relation
- $F$  - a set of functional dependencies
- problems
  - I. compute the closure of  $F$ :  $F^+$ 
    - the set  $F^+$  contains all the functional dependencies implied by  $F$
    - $F$  implies a functional dependency  $f$  if  $f$  holds on every relation that satisfies  $F$
    - the following 3 rules can be repeatedly applied to compute  $F^+$  (Armstrong's Axioms):
      - $\alpha, \beta, \gamma$  - subsets of attributes of  $A$
      - 1. reflexivity: if  $\beta \subseteq \alpha$ , then  $\alpha \rightarrow \beta$
      - 2. augmentation: if  $\alpha \rightarrow \beta$ , then  $\alpha\gamma \rightarrow \beta\gamma$
      - 3. transitivity: if  $\alpha \rightarrow \beta$  and  $\beta \rightarrow \gamma$ , then  $\alpha \rightarrow \gamma$
    - these rules are complete (they generate all dependencies in the closure) and sound (no erroneous functional dependencies can be derived)

- $R[A]$  - a relation
  - $F$  - a set of functional dependencies
  - problems
- I. compute the closure of  $F$ :  $F^+$
- the following rules can be derived from Armstrong's Axioms:

4. union: if  $\alpha \rightarrow \beta$  and  $\alpha \rightarrow \gamma$ , then  $\alpha \rightarrow \beta\gamma$

$$\left. \begin{array}{l} \alpha \rightarrow \beta \Rightarrow \alpha\alpha \rightarrow \alpha\beta \\ \alpha \rightarrow \gamma \Rightarrow \alpha\beta \rightarrow \beta\gamma \end{array} \right\} \begin{array}{l} \text{augmentation} \\ \text{transitivity} \end{array} \quad \alpha \rightarrow \beta\gamma$$

- $R[A]$  - a relation
- $F$  - a set of functional dependencies
- problems
  - I. compute the closure of  $F$ :  $F^+$
- the following rules can be derived from Armstrong's Axioms:

5. decomposition: if  $\alpha \rightarrow \beta\gamma$ , then  $\alpha \rightarrow \beta$  and  $\alpha \rightarrow \gamma$

$$\left. \begin{array}{l} \alpha \rightarrow \beta\gamma \\ \beta\gamma \rightarrow \beta \text{ (reflexivity)} \end{array} \right\} \text{transitivity} \Rightarrow \alpha \rightarrow \beta \text{ } (\alpha \rightarrow \gamma \text{ can similarly be shown to hold})$$

- $R[A]$  - a relation
- $F$  - a set of functional dependencies
- problems
  - I. compute the closure of  $F$ :  $F^+$
- the following rules can be derived from Armstrong's Axioms:

6. pseudotransitivity: if  $\alpha \rightarrow \beta$  and  $\beta\gamma \rightarrow \delta$ , then  $\alpha\gamma \rightarrow \delta$

$$\left. \begin{array}{l} \alpha \rightarrow \beta \Rightarrow \alpha\gamma \rightarrow \beta\gamma \\ \beta\gamma \rightarrow \delta \end{array} \right\} \text{transitivity} \Rightarrow \alpha\gamma \rightarrow \delta$$

- $\alpha, \beta, \gamma, \delta$  - subsets of attributes of  $A$

- $R[A]$  - a relation
- $F$  - a set of functional dependencies
- $\alpha$  – a subset of attributes
- problems

II. compute the closure of a set of attributes under a set of functional dependencies

- determine the closure of  $\alpha$  under  $F$ , denoted as  $\alpha^+$
- $\alpha^+$  - the set of attributes that are functionally dependent on  $\alpha$  under  $F$

- $R[A]$  - a relation
- $F$  - a set of functional dependencies
- $\alpha$  – a subset of attributes
- problems

II. compute the closure of a set of attributes under a set of functional dependencies

- algorithm

**closure** :=  $\alpha$ ;

**repeat until there is no change:**

**for every functional dependency**  $\beta \rightarrow \gamma$  **in**  $F$

**if**  $\beta \subseteq \text{closure}$

**then** **closure** := **closure**  $\cup \gamma$ ;

- $R[A]$  - a relation
- $F$  - a set of functional dependencies
- problems

III. compute the minimal cover for a set of dependencies

Definition:  $F, G$  - two sets of functional dependencies;  $F$  and  $G$  are equivalent (notation  $F \equiv G$ ) if  $F^+ = G^+$ .

- $R[A]$  - a relation
- $F$  - a set of functional dependencies
- problems

### III. compute the minimal cover for a set of dependencies

Definition:  $F$  - set of functional dependencies; a minimal cover for  $F$  is a set of functional dependencies  $F_M$  that satisfies the following conditions:

1.  $F_M \equiv F$
2. the right side of every dependency in  $F_M$  has a single attribute;
3. the left side of every dependency in  $F_M$  is irreducible (i.e., no attribute can be removed from the determinant of a dependency in  $F_M$  without changing  $F_M$ 's closure);
4. no dependency  $f$  in  $F_M$  is redundant (no dependency can be discarded without changing  $F_M$ 's closure).

\* closure of a set of functional dependencies

P1. Let  $R[ABCDEF]$  be a relational schema and  $S$  a set of functional dependencies over  $R$ ,  $S = \{A \rightarrow B, A \rightarrow C, CD \rightarrow E, CD \rightarrow F, D \rightarrow E\}$ .

Show the following FDs are in  $S^+$ :  $A \rightarrow BC, CD \rightarrow EF, AD \rightarrow E, AD \rightarrow F$ .

$$\left. \begin{array}{l} A \rightarrow B \\ A \rightarrow C \end{array} \right\} \text{union} \Rightarrow A \rightarrow BC$$

$$\left. \begin{array}{l} CD \rightarrow E \\ CD \rightarrow F \end{array} \right\} \text{union} \Rightarrow CD \rightarrow EF$$

$$A \rightarrow C \quad \Rightarrow \quad \left. \begin{array}{l} AD \rightarrow CD \\ CD \rightarrow E \end{array} \right\} \text{transitivity} \Rightarrow AD \rightarrow E$$

\* closure of a set of functional dependencies

P1. Let  $R[ABCDEF]$  be a relational schema and  $S$  a set of functional dependencies over  $R$ ,  $S = \{A \rightarrow B, A \rightarrow C, CD \rightarrow E, CD \rightarrow F, D \rightarrow E\}$ .

Show the following FDs are in  $S^+$ :  $A \rightarrow BC, CD \rightarrow EF, AD \rightarrow E, AD \rightarrow F$ .

$$\left. \begin{array}{l} A \rightarrow C \\ CD \rightarrow F \end{array} \right\} \Rightarrow AD \rightarrow F$$

pseudotransitivity

\* closure of a set of attributes under a set of functional dependencies

P2. Let  $R[ABCDEF]$  be a relational schema,  $S$  a set of functional dependencies over  $R$  and  $\alpha$  a subset of attributes of the set of attributes of  $R$ ,  $S = \{A \rightarrow B, A \rightarrow C, CD \rightarrow E, CD \rightarrow F, D \rightarrow E\}$ ,  $\alpha = \{A, D\}$ . Compute  $\alpha^+$ .

$$\alpha^+ = \{A, D\}$$

$$A \rightarrow B \Rightarrow \alpha^+ = \{A, B, D\}$$

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

$$CD \rightarrow E \Rightarrow \alpha^+ = \{A, B, C, D, E\}$$

$$CD \rightarrow F \Rightarrow \alpha^+ = \{A, B, C, D, E, F\}$$

$D \rightarrow E$ ,  $E$  already in  $\alpha^+$

- iterate over all dependencies one more time,  $\alpha^+$  remains unchanged
- $\alpha^+ = \{A, B, C, D, E, F\}$

\* minimal cover for a set of functional dependencies

P3. Let  $R[ABCD]$  be a relational schema and  $S$  a set of functional dependencies over  $R$ ,  $S=\{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C, AC \rightarrow D\}$ . Compute a minimal cover of  $S$ .

- decomposition:  $A \rightarrow BC \Rightarrow A \rightarrow B, A \rightarrow C$

$$\Rightarrow A \rightarrow B$$

$$A \rightarrow C$$

$$B \rightarrow C$$

~~$$A \rightarrow B$$~~

- can be eliminated

$$AB \rightarrow C$$

$$AC \rightarrow D$$

\* minimal cover for a set of functional dependencies

P3. Let  $R[ABCD]$  be a relational schema and  $S$  a set of functional dependencies over  $R$ ,  $S=\{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C, AC \rightarrow D\}$ . Compute a minimal cover of  $S$ .

- augmentation:  $A \rightarrow C \Rightarrow A \rightarrow AC$
- transitivity:  $A \rightarrow AC, AC \rightarrow D \Rightarrow A \rightarrow D$   
 $\Rightarrow C$  in  $AC \rightarrow D$  is redundant

$$\Rightarrow A \rightarrow B$$

$$A \rightarrow C$$

$$B \rightarrow C$$

$$AB \rightarrow C$$

$$A \rightarrow D$$

\* minimal cover for a set of functional dependencies

P3. Let  $R[ABCD]$  be a relational schema and  $S$  a set of functional dependencies over  $R$ ,  $S=\{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C, AC \rightarrow D\}$ . Compute a minimal cover of  $S$ .

- augmentation:  $A \rightarrow C \Rightarrow AB \rightarrow CB$
- decomposition:  $AB \rightarrow CB \Rightarrow AB \rightarrow C$   
 $\Rightarrow$  can eliminate  $AB \rightarrow C$

$\Rightarrow$   $A \rightarrow B$   
 $A \rightarrow C$   
 $B \rightarrow C$   
 $A \rightarrow D$

\* minimal cover for a set of functional dependencies

P3. Let  $R[ABCD]$  be a relational schema and  $S$  a set of functional dependencies over  $R$ ,  $S=\{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C, AC \rightarrow D\}$ . Compute a minimal cover of  $S$ .

- transitivity:  $A \rightarrow B, B \rightarrow C \Rightarrow A \rightarrow C$   
 $\Rightarrow$  can eliminate  $A \rightarrow C$

$$\begin{aligned}\Rightarrow \quad & A \rightarrow B \\ & B \rightarrow C \\ & A \rightarrow D\end{aligned}$$

Example 11. Consider relation DFM[Department, FacultyMembers, MeetingDates], with repeating attributes *FacultyMembers* and *MeetingDates*.

- a possible instance is given below:

Department	FacultyMembers	MeetingDates
Computer Science	FCS1 FCS2 ... FCSm	DCS1 DCS2 ... DCSn
Mathematics	FM1 FM2 ... FMp	DM1 DM2 ... DMq

- eliminate repeating attributes (such that the relation is at least in 1NF) - replace DFM by a relation DFM' in which *FacultyMember* and *MeetingDate* are scalar attributes:

Department	FacultyMember	MeetingDate
Computer Science	FCS1	DCS1
Computer Science	FCS1	DCS2
...	...	...
Computer Science	FCS1	DCSn
Computer Science	FCS2	DCS1
Computer Science	FCS2	DCS2
...	...	...
Mathematics	FM1	DM1
...	...	...
Mathematics	FMp	DMq

Department	FacultyMember	MeetingDate
Computer Science	FCS1	DCS1
Computer Science	FCS1	DCS2
...	...	...
Computer Science	FCS1	DCSn
Computer Science	FCS2	DCS1
Computer Science	FCS2	DCS2
...	...	...
Mathematics	FM1	DM1
...	...	...
Mathematics	FMp	DMq

- in this table, each faculty member has the same meeting dates
- therefore, when adding / changing / removing rows, additional checks must be carried out

# References

- [Ta13] TÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ha96] HANSEN, G., HANSEN, J., Database Management And Design (2<sup>nd</sup> Edition), Prentice Hall, 1996
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3<sup>rd</sup> Edition,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

# Databases

Lecture 7  
Relational Algebra

- query languages in the relational model
  - relational algebra and calculus - formal query languages with a significant influence on SQL
    - relational algebra
      - queries are specified in an operational manner
    - relational calculus
      - queries describe the desired answer, without specifying how it will be computed (declarative)
  - not expected to be Turing complete
  - not intended for complex calculations
  - provide efficient access to large datasets
  - allow optimizations

- relational algebra
  - used by DBMSs to represent query execution plans
  - a relational algebra query:
    - is built using a collection of operators
    - describes a step-by-step procedure for computing the result set
    - is evaluated on the input relations' instances
    - produces an instance of the output relation
  - every operation returns a relation, so operators can be composed; the algebra is closed
  - the result of an algebra expression is a relation, and a relation is a set of tuples
- relational algebra on bags (multisets) - duplicates are not eliminated

## Conditions

- conditions that can be used in several algebraic operators
  - similar to the SELECT filter conditions
1. *attribute\_name relational\_operator value*
    - *value* - attribute name, expression
  2. *attribute\_name IS [NOT] IN single\_column\_relation*
    - a relation with one column can be considered a set
    - the condition tests whether a value belongs to a set
  3. *relation {IS [NOT] IN | = | <>} relation*
    - the relations in the condition must be union-compatible

## Conditions

4. (*condition*)

*NOT condition*

*condition<sub>1</sub>* *AND condition<sub>2</sub>*

*condition<sub>1</sub>* *OR condition<sub>2</sub>*,

where *condition*, *condition<sub>1</sub>*, *condition<sub>2</sub>* are conditions of type 1-4.

## Operators in the Algebra

- equivalent SELECT statements can be specified for the relational algebra expressions
- *selection*
  - notation:  $\sigma_C(R)$
  - resulting relation:
    - schema:  $R$ 's schema
    - tuples: records in  $R$  that satisfy condition C
  - equivalent SELECT statement

```
SELECT *
FROM R
WHERE C
```

- *projection*
  - notation:  $\pi_\alpha(R)$
  - resulting relation:
    - schema: attributes in  $\alpha$
    - tuples: every record in  $R$  is projected on  $\alpha$
  - $\alpha$  can be extended to a set of expressions, specifying the columns of the relation being computed
  - equivalent SELECT statement

SELECT DISTINCT  $\alpha$

FROM R

SELECT  $\alpha$

FROM R -- algebra on bags

- *cross-product*
  - notation:  $R_1 \times R_2$
  - resulting relation:
    - schema: the attributes of  $R_1$  followed by the attributes of  $R_2$
    - tuples: every tuple  $r_1$  in  $R_1$  is concatenated with every tuple  $r_2$  in  $R_2$
  - equivalent SELECT statement

```
SELECT *
FROM R1 CROSS JOIN R2
```

- *union, set-difference, intersection*

- notation:  $R_1 \cup R_2$ ,  $R_1 - R_2$ ,  $R_1 \cap R_2$
- $R_1$  and  $R_2$  must be union-compatible:
  - same number of columns
  - corresponding columns, taken in order from left to right, have the same domains
- equivalent SELECT statements

SELECT \*

FROM R1

UNION

SELECT \*

FROM R2

SELECT \*

FROM R1

EXCEPT

SELECT \*

FROM R2

SELECT \*

FROM R1

INTERSECT

SELECT \*

FROM R2

-- algebra on bags: SELECT statements that don't eliminate duplicates (e.g., UNION ALL)

- join operators
  - *condition join* (or *theta join*)
    - notation:  $R_1 \otimes_{\Theta} R_2$
    - result: the records in the cross-product of  $R_1$  and  $R_2$  that satisfy a certain condition
  - definition  $\Rightarrow R_1 \otimes_{\Theta} R_2 = \sigma_{\Theta}(R_1 \times R_2)$
  - equivalent SELECT statement

```
SELECT *
FROM R1 INNER JOIN R2 ON Θ
```

- join operators

- *natural join*

- notation:  $R_1 * R_2$

- resulting relation:

- schema: the union of the attributes of the two relations (attributes with the same name in  $R_1$  and  $R_2$  appear once in the result)

- tuples: obtained from tuples  $\langle r_1, r_2 \rangle$ , where  $r_1$  in  $R_1$ ,  $r_2$  in  $R_2$ , and  $r_1$  and  $r_2$  agree on the common attributes of  $R_1$  and  $R_2$

- let  $R_1[\alpha]$ ,  $R_2[\beta]$ ,  $\alpha \cap \beta = \{A_1, A_2, \dots, A_m\}$ ; then:

$$R_1 * R_2 = \pi_{\alpha \cup \beta} (R_1 \otimes_{R_1.A_1=R_2.A_1 \text{ AND } \dots \text{ AND } R_1.A_m=R_2.A_m} R_2)$$

- equivalent SELECT statement

```
SELECT *
FROM R1 NATURAL JOIN R2
```

- join operators
  - *left outer join*
    - notation (in these notes):  $R_1 \times_C R_2$
    - resulting relation:
      - schema: the attributes of  $R_1$  followed by the attributes of  $R_2$
      - tuples: tuples from the condition join  $R_1 \otimes_c R_2$  + the tuples in  $R_1$  that were not used in  $R_1 \otimes_c R_2$  combined with the *null* value for the attributes of  $R_2$
    - equivalent SELECT statement

```
SELECT *
FROM R1 LEFT OUTER JOIN R2 ON C
```

- join operators
  - *right outer join*
    - notation:  $R_1 \rtimes_C R_2$
    - resulting relation:
      - schema: the attributes of  $R_1$  followed by the attributes of  $R_2$
      - tuples: tuples from the condition join  $R_1 \otimes_c R_2$  + the tuples in  $R_2$  that were not used in  $R_1 \otimes_c R_2$  combined with the *null* value for the attributes of  $R_1$
    - equivalent SELECT statement

```
SELECT *
FROM R1 RIGHT OUTER JOIN R2 ON C
```

- join operators
  - *full outer join*
    - notation:  $R_1 \bowtie_C R_2$
    - resulting relation:
      - schema: the attributes of  $R_1$  followed by the attributes of  $R_2$
      - tuples:
        - tuples from the condition join  $R_1 \otimes_c R_2$  +
        - the tuples in  $R_1$  that were not used in  $R_1 \otimes_c R_2$  combined with the *null* value for the attributes of  $R_2$  +
        - the tuples in  $R_2$  that were not used in  $R_1 \otimes_c R_2$  combined with the *null* value for the attributes of  $R_1$
      - equivalent SELECT statement

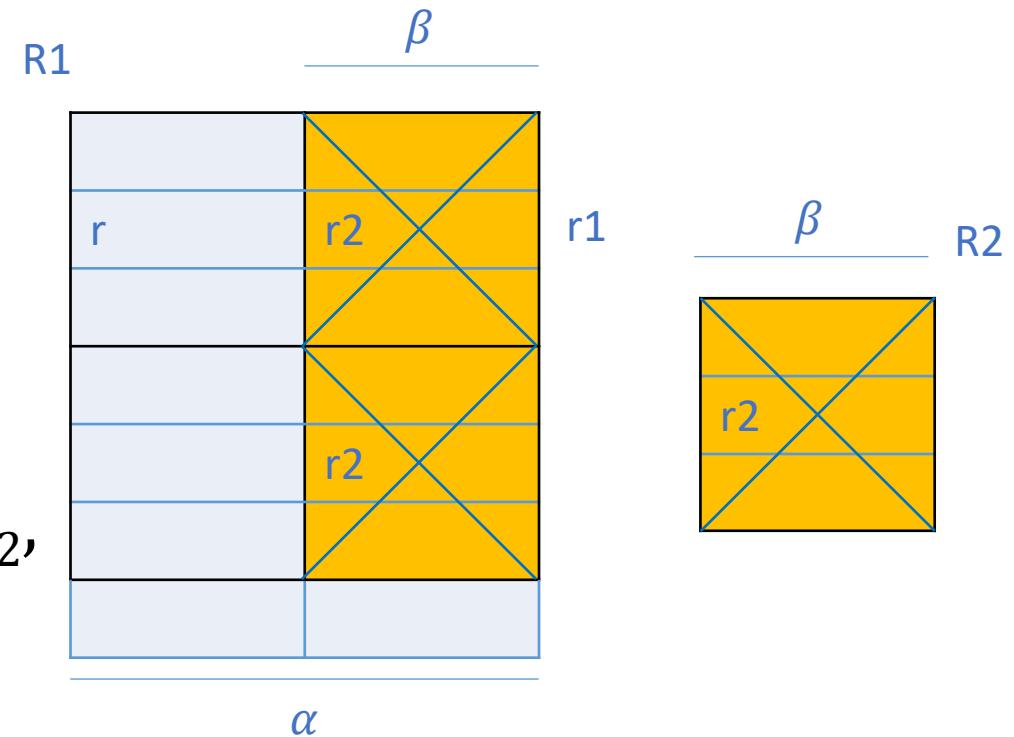
```
SELECT *
FROM R1 FULL OUTER JOIN R2 ON C
```

- join operators
  - *left semi join*
    - notation:  $R_1 \triangleright R_2$
    - resulting relation:
      - schema:  $R_1$ 's schema
      - tuples: the tuples in  $R_1$  that are used in the natural join  $R_1 * R_2$

- join operators
  - *right semi join*
    - notation:  $R_1 \lhd R_2$
    - resulting relation:
      - schema:  $R_2$ 's schema
      - tuples: the tuples in  $R_2$  that are used in the natural join  $R_1 * R_2$

- *division*

- notation:  $R_1 \div R_2$
- $R_1[\alpha], R_2[\beta], \beta \subset \alpha$
- resulting relation:
  - schema:  $\alpha - \beta$
  - tuples: a record  $r \in R_1 \div R_2$  iff  $\forall r_2 \in R_2, \exists r_1 \in R_1$  such that:
    - $\pi_{\alpha-\beta}(r_1) = r$
    - $\pi_\beta(r_1) = r_2$
    - i.e., a record  $r$  belongs to the result if in  $R_1$   $r$  is concatenated with every record in  $R_2$



- see lecture examples (at the board) with algebra queries:
  - selection
  - projection
  - division
  - selection, projection
  - natural join, selection, projection
  - different algebra expressions producing the same result (optimization - reducing the size of intermediate relations)

## An Independent Subset of Operators

- independent set of operators M:
  - eliminating any operator  $op$  from M: there will be a relation that can be obtained using M's operators, but cannot be obtained with the operators in  $M-\{op\}$
- for the previously described query language, with operators:  
 $\{\sigma, \pi, \times, U, -, \cap, \otimes, *, \bowtie, \bowtie, \triangleright, \triangleleft, \div\}$   
an independent set of operators is  $\{\sigma, \pi, \times, U, -\}$
- the other operators are obtained as follows (some expressions have already been introduced):
  - $R_1 \cap R_2 = R_1 - (R_1 - R_2)$
  - $R_1 \otimes_C R_2 = \sigma_C(R_1 \times R_2)$

- the other operators are obtained as follows (some expressions have already been introduced):

- $R_1[\alpha], R_2[\beta], \alpha \cap \beta = \{A_1, A_2, \dots, A_m\}$ , then:

$$R_1 * R_2 = \pi_{\alpha \cup \beta} (R_1 \otimes_{R_1.A_1=R_2.A_1} AND \dots AND R_1.A_m=R_2.A_m R_2)$$

- $R_1[\alpha], R_2[\beta], R_3[\beta] = \{(null, \dots, null)\}, R_4[\alpha] = \{(null, \dots, null)\}$

$$R_1 \bowtie_C R_2 = (R_1 \otimes_C R_2) \cup (R_1 - \pi_\alpha(R_1 \otimes_C R_2)) \times R_3$$

$$R_1 \bowtie_C R_2 = (R_1 \otimes_C R_2) \cup R_4 \times (R_2 - \pi_\beta(R_1 \otimes_C R_2))$$

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

- $R_1[\alpha], R_2[\beta]$

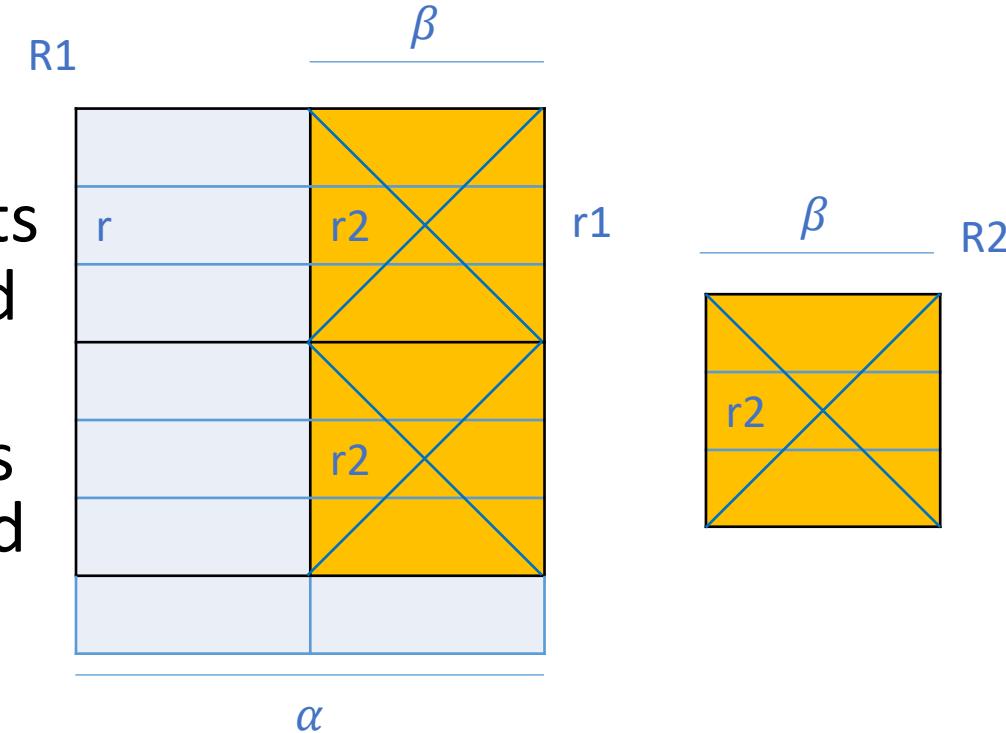
$$R_1 \triangleright R_2 = \pi_\alpha(R_1 * R_2)$$

$$R_1 \triangleleft R_2 = \pi_\beta(R_1 * R_2)$$

- the other operators are obtained as follows (some expressions have already been introduced):

- if  $R_1[\alpha], R_2[\beta], \beta \subset \alpha$ , then  $r \in R_1 \div R_2$  iff  $\forall r_2 \in R_2, \exists r_1 \in R_1$  such that:  $\pi_{\alpha-\beta}(r_1) = r$  and  $\pi_\beta(r_1) = r_2$   
 $\Rightarrow r$  is in  $\pi_{\alpha-\beta}(R_1)$ , but not all the elements in  $\pi_{\alpha-\beta}(R_1)$  are in the result
- $(\pi_{\alpha-\beta}(R_1)) \times R_2$  contains all the elements with one part in  $\pi_{\alpha-\beta}(R_1)$  and the second part in  $R_2$
- to obtain values that are disqualified,  $R_1$  is subtracted from the obtained relation, and the result is projected on  $\alpha - \beta$
- the final expression:

$$R_1 \div R_2 = \pi_{\alpha-\beta}(R_1) - \pi_{\alpha-\beta}((\pi_{\alpha-\beta}(R_1)) \times R_2 - R_1)$$



- the *renaming* operator

$$\rho(R'(A_1 \rightarrow A'_1, A_2 \rightarrow A'_2, A_3 \rightarrow A'_3), E)$$

- E - relational algebra expression
- the result, relation  $R'$ , has the same tuples as the result of E
- attributes  $A_1, A_2$ , and  $A_3$  are renamed to  $A'_1, A'_2$ , and  $A'_3$ , respectively

\* the next examples use the statements below:

- assignment

$R[\text{list}] := \text{expression}$

- the expression's result (a relation) is assigned to a variable ( $R[\text{list}]$ ), specifying the name of the relation [and the names of its columns]

- eliminating duplicates from a relation

$\delta(R)$

- sorting records in a relation

$S_{\{\text{list}\}}(R)$

- grouping

$\gamma_{\{\text{list1}\} \text{ group by } \{\text{list2}\}}(R)$

- R's records are grouped by the columns in *list2*
- *list1* (that can contain aggregate functions) is evaluated for each group of records

students [id, name, sgroup, gpa, dob]  
groups [id, year, program]  
schedule [day, starthour, endhour, activtype, room, sgroup,  
faculty\_m\_id]  
faculty\_members [id, name]

### 1. The names of students in a given group:

$$R := \pi_{\{name\}} \left( \sigma_{sgroup='222'}(students) \right)$$

```
SELECT name  
FROM students  
WHERE sgroup='222'
```

## 2. The students in a given program (alphabetical list, by groups):

$$G := \pi_{\{id\}} \left( \sigma_{program='IG'}(groups) \right)$$
$$R := S_{\{sgroup, name\}} \left( \sigma_{sgroup \text{ is in } G}(students) \right)$$

```
SELECT *                                students [id, name, sgroup, gpa, dob]
FROM students                            groups [id, year, program]
WHERE sgroup IN                          schedule [day, starthour, endhour, activtype, room,
    (SELECT id                           sgroup, facultym_id]
    FROM groups                         faculty_members [id, name]
    WHERE program='IG')
ORDER BY sgroup, name
```

### 3. The number of students in every group of a given program:

$$ST := \sigma_{sgroup \text{ is in } \left( \pi_{\{id\}} \left( \sigma_{program='IG'}(groups) \right) \right)}(\text{students})$$
$$NR := \gamma_{\{sgroup, count(*)\}} \text{ group by } \{sgroup\}(ST)$$

```
SELECT sgroup, COUNT(*)
```

```
FROM (SELECT *
```

```
      FROM students
```

```
      WHERE sgroup IN
```

```
          (SELECT id
```

```
              FROM groups
```

```
              WHERE program='IG')
```

```
      ) t
```

```
GROUP BY sgroup
```

students [id, name, sgroup, gpa, dob]  
groups [id, year, program]  
schedule [day, starthour, endhour, activtype, room,  
 sgroup, facultym\_id]  
faculty\_members [id, name]

4. A student's schedule (the student is given by name):

$$T := \sigma_{sgroup \text{ is in } (\pi_{\{sgroup\}}(\sigma_{name='Ionescu M. Razvan'}(students)))} (schedule)$$

5. The number of hours per week for every group:

$$F(no, sgroup) := \pi_{\{endhour - starthour, sgroup\}}(schedule)$$

$$NoHours(sgroup, nohours) := \gamma_{\{sgroup, sum(no)\}} \text{ group by } \{sgroup\}(F)$$

students [id, name, sgroup, gpa, dob]

groups [id, year, program]

schedule [day, starthour, endhour, activtype, room, sgroup, facultym\_id]

faculty\_members [id, name]

6. The faculty members (their names) who teach a given student:

$$A := (\sigma_{name='Ionescu\ M.\ Razvan'}(students)) \otimes_{students.sgroup=schedule.sgroup} schedule$$
$$B := \pi_{\{faculty\_id\}}(A)$$
$$C := faculty\_members \otimes_{faculty\_members.id=B.faculty\_id} B$$
$$D := \pi_{\{name\}}(C)$$

students [id, name, sgroup, gpa, dob]

groups [id, year, program]

schedule [day, starthour, endhour, activtype, room, sgroup, facultym\_id]

faculty\_members [id, name]

7. The faculty members with no teaching assignments (i.e., not on the schedule):

$$C := \pi_{\{name\}}(faculty\_members) -$$

$$\pi_{\{name\}}(schedule \otimes_{schedule.facultym\_id=faculty\_members.id} faculty\_members)$$

\* Is there a problem if two different faculty members have the same name?

students [id, name, sgroup, gpa, dob]

groups [id, year, program]

schedule [day, starthour, endhour, activtype, room, sgroup, facultym\_id]

faculty\_members [id, name]

8. Students with school activities on every day of the week (all days with school activities considered):

$$A := \delta(\pi_{\{day\}}(schedule))$$

$$B := students \otimes_{students.sgroup=schedule.sgroup} schedule$$

$$C := \delta(\pi_{\{name, day\}}(B))$$

$$D := C \div A$$

\* Is there a problem if two different students have the same name?

students [id, name, sgroup, gpa, dob]

groups [id, year, program]

schedule [day, starthour, endhour, activtype, room, sgroup, facultym\_id]

faculty\_members [id, name]

# References

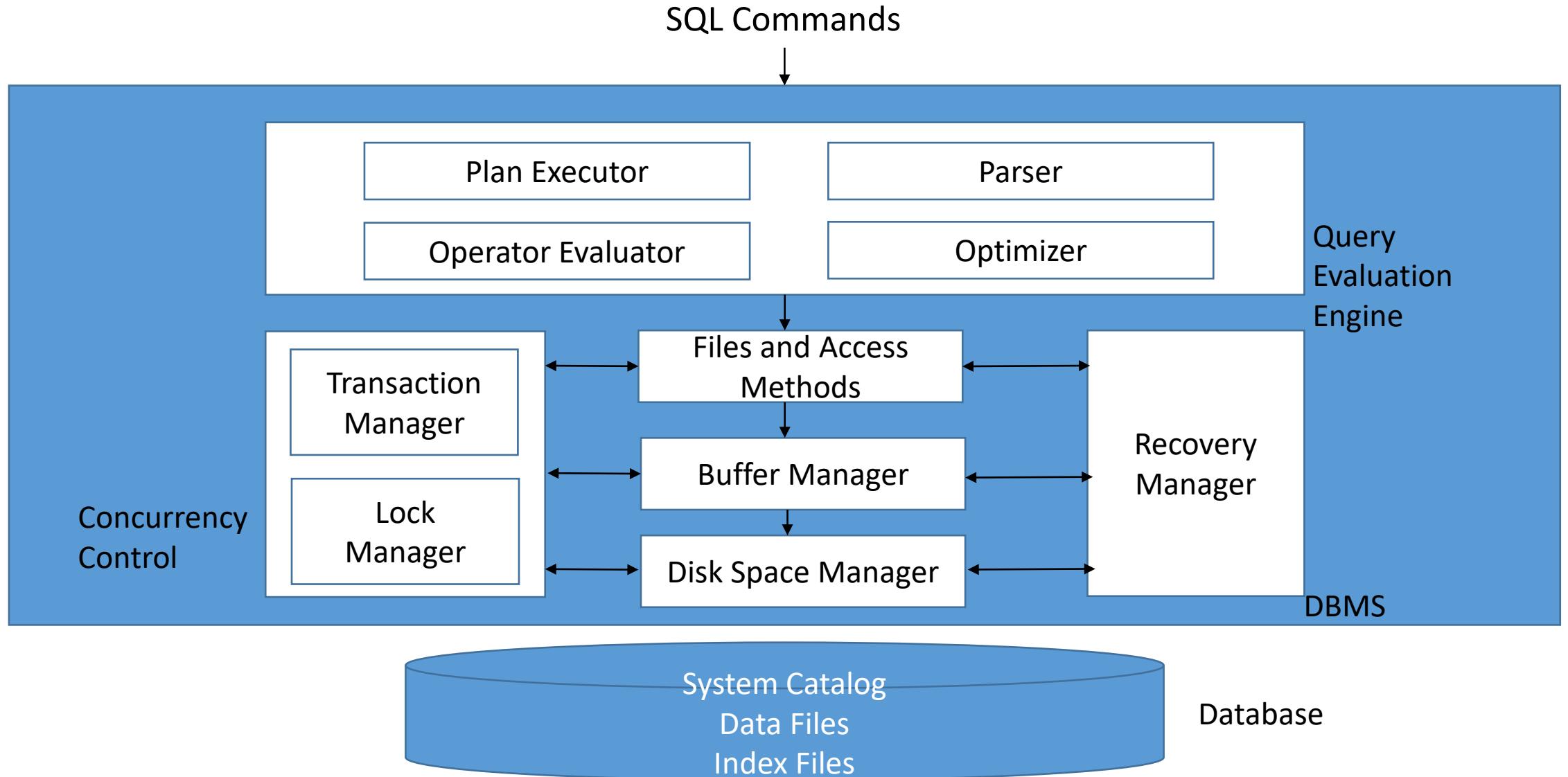
- [Ta13] TÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ha96] HANSEN, G., HANSEN, J., Database Management And Design (2<sup>nd</sup> Edition), Prentice Hall, 1996
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3<sup>rd</sup> Edition,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

# Databases

Lecture 8

The Physical Structure of Databases

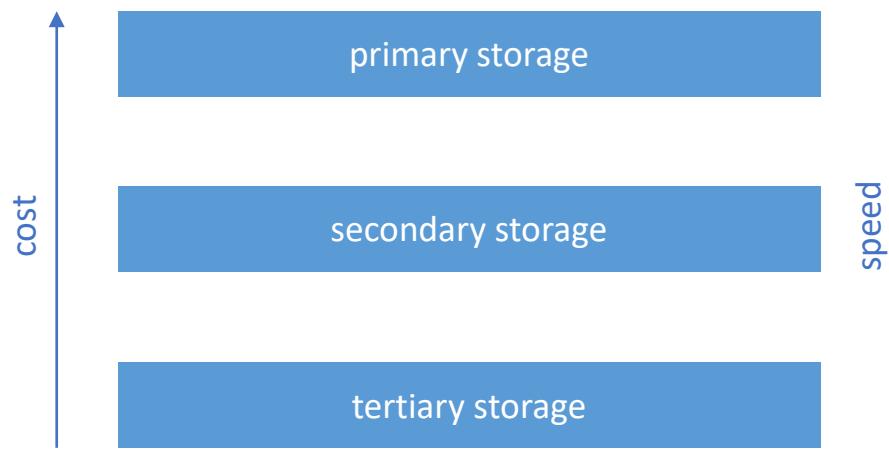
# DBMS Architecture



# The Memory Hierarchy

- primary storage
  - cache, main memory
  - very fast access to data
  - volatile
  - currently used data
- secondary storage
  - e.g., magnetic disks
  - slower storage devices
  - nonvolatile
  - disks - sequential, direct access
  - main database
- tertiary storage
  - e.g., optical disks, tapes
  - slowest storage devices
  - nonvolatile
  - tapes
    - only sequential access
    - good for archives, backups
    - unsuitable for data that is frequently accessed

# The Memory Hierarchy



- disks and tapes - significantly cheaper than main memory
  - large amounts of data that shouldn't be discarded when the system is restarted
- => the need for DBMSs that bring data from disks into main memory for processing

## Secondary Storage – Magnetic Disks

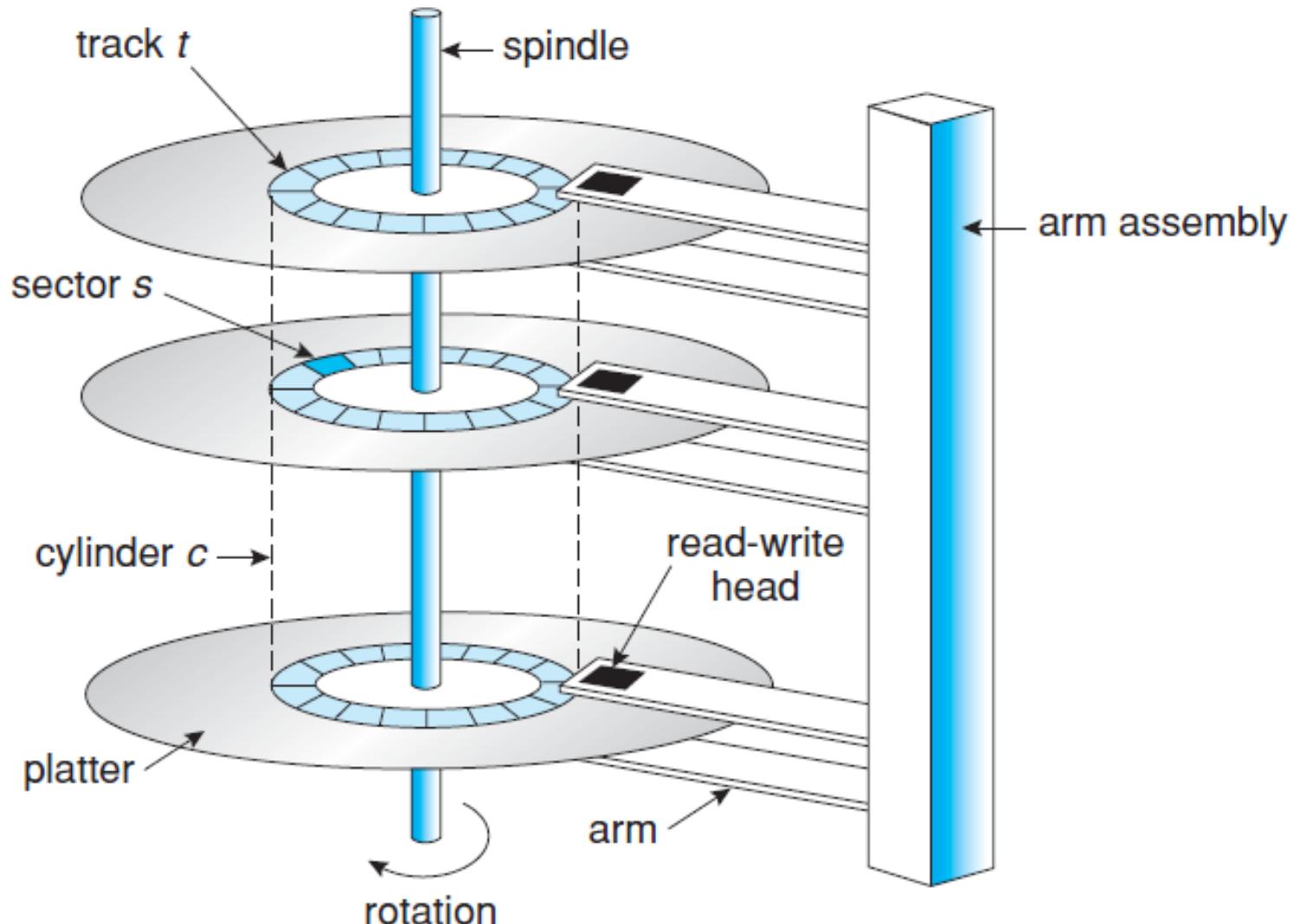
- direct access
- extremely used in database applications
- DBMSs - applications don't need to know whether the data is on disk or in main memory
- *disk block*
  - sequence of contiguous bytes
  - unit for data storage
  - unit for data transfer (reading data from disk / writing data to disk)
  - reading / writing a block - an input / output (I/O) operation
- *tracks*
  - concentric rings containing blocks, recorded on one or more platters

## Secondary Storage – Magnetic Disks

- *sectors*
  - arcs on tracks
- *platters*
  - single-sided, double-sided (data recorded on one / both surfaces)
- *cylinder*
  - set of all tracks with the same diameter
- *disk heads*
  - one per recorded surface
  - to read / write a block, a head must be on top of the block
  - all disk heads are moved as a unit
  - systems with one active head

# Secondary Storage – Magnetic Disks

- sector size
  - characteristic of the disk, cannot be modified
- block size
  - multiple of the sector size



[Si08]

Sabina S. CS

## Secondary Storage – Magnetic Disks

- DBMSs operate on data when it is in memory
- block - unit for data transfer between disk and main memory
- time to access a desired location:
  - main memory - approximately the same for any location
  - disk - depends on where the data is stored
- disk access time:
  - seek time + rotational delay + transfer time
  - seek time
    - time to move the disk head to the desired track (smaller platter size => decreased seek time)
  - rotational delay
    - time for the block to get under the head
  - transfer time
    - time to read / write the block, once the disk head is positioned over it

## Secondary Storage – Magnetic Disks

- time required for DB operations - dominated by the time taken to transfer blocks between disk and main memory
- goal
  - minimize access time
  - for this purpose, data should be carefully placed on disk
- records that are often used together should be close to each other:
  - same block
  - same track
  - same cylinder
  - adjacent cylinder
- accessing data in a sequential fashion reduces seek time and rotational delay

## Secondary Storage – Magnetic Disks

\* characteristics, e.g.:

- storage capacity (e.g., GB)
- platters
  - number, *single-sided* or *double-sided*
- average / max seek time (ms)
- average rotational delay (ms)
- number of rotations / min
- data transfer rate (MB/s)
- ...

## Moore's Law

- Gordon Moore: "the improvement of integrated circuits is following an exponential curve that doubles every 18 months"
  - parameters that follow Moore's law
    - speed of processors (number of instructions executed / sec)
    - no. of bits / chip
    - capacity of largest disks
  - parameters that do not follow Moore's law
    - speed of accessing data in main memory
    - disk rotation speed
- => "latency" keeps increasing
- time to move data between memory hierarchy levels appears to take longer compared with computation time

## Solid-State Disks

- NAND flash components
- faster random access
- higher data transfer rates
- no moving parts
- higher cost per GB
- limited write cycles

## Managing Disk Space

- the *disk space manager* (DSM) manages space on disk
- page
  - unit of data
  - size of a page = size of a disk block
  - R/W a page - one I/O operation
- upper layers in the DBMS can treat the data as a collection of pages
- DSM
  - commands to allocate / deallocate / read / write a page
  - knows which pages are on which disk blocks
  - monitors disk usage, keeping track of available disk blocks

## Managing Disk Space

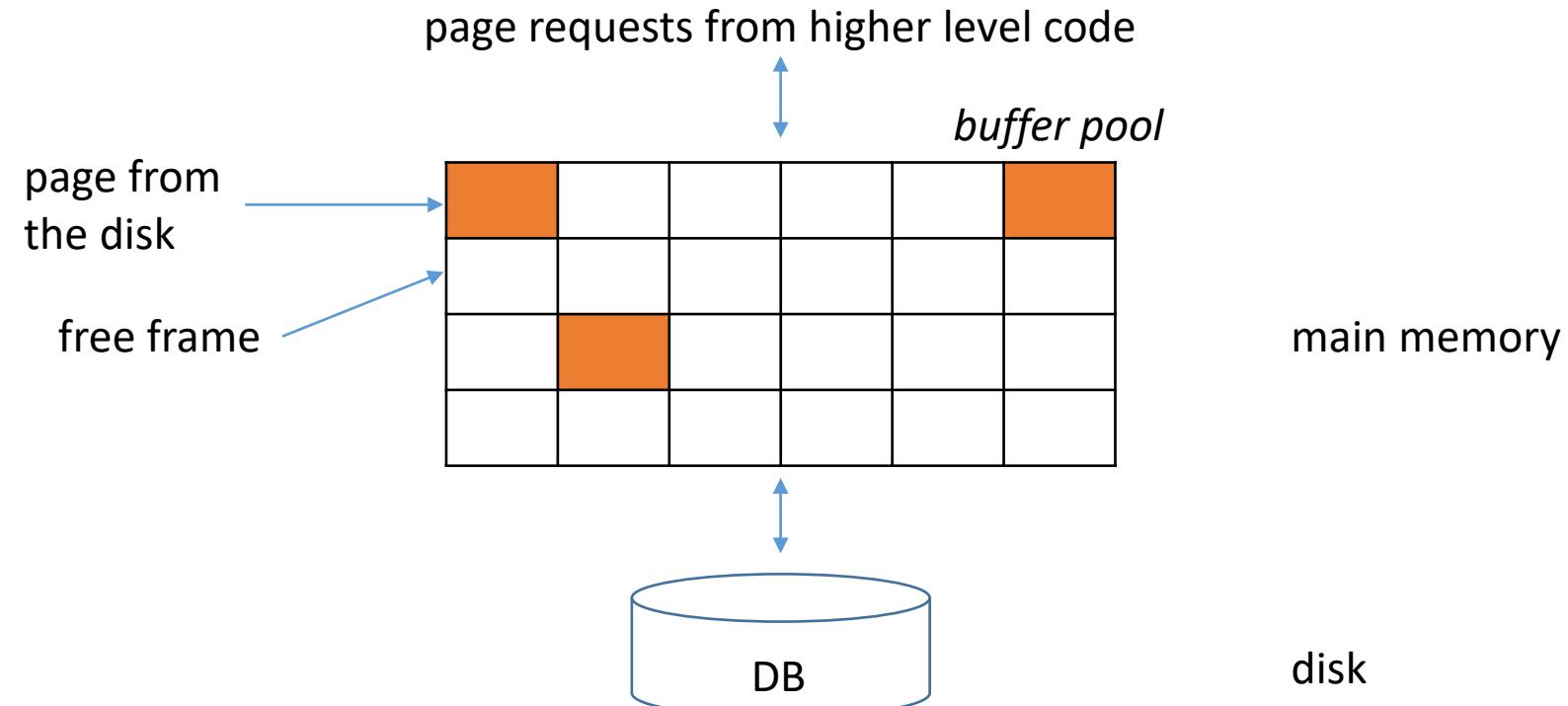
- free blocks can be identified:
  - by maintaining a linked list of free blocks (on deallocation, a block is added to the list)
  - by maintaining a bitmap with one bit / block, indicating whether the corresponding block is used or not
    - allows for fast identification of contiguous available areas on disk

## Buffer Manager

- e.g., DB = 500.000 pages, main memory - 1000 available pages, query that scans the entire file
- *buffer manager* (BM)
  - brings new data pages from disk to main memory as they are required
  - decides what main memory pages can be replaced
  - manages the available main memory
    - collection of pages called the *buffer pool* (BP)
    - *frame*
      - page in the BP
      - slot that can hold a page
- *replacement policy*
  - policy that dictates the choice of replacement frames in the BP

# Buffer Manager

- higher level layer L in the DBMS asks the BM for page P
- if P is not in the BP, the BM brings it into a frame F in the BP
- when P is no longer needed, L notifies the BM (it releases P), so F can be reused
- if P has been modified, L notifies the BM, which propagates the changes in F to the disk



## Buffer Manager

- BM maintains 2 variables for each frame F
  - *pin\_count*
    - number of current users (requested the page in F but haven't released it yet)
    - only frames with *pin\_count* = 0 can be chosen as replacement frames
  - *dirty*
    - boolean value indicating whether the page in F has been changed since being brought into F
- incrementing *pin\_count*
  - pinning a page P in a frame F
- decrementing *pin\_count*
  - unpinning a page

## Buffer Manager

- initially,  $\text{pin\_count} = 0$ ,  $\text{dirty} = \text{off}$ ,  $\forall F \in \text{BP}$
- L asks for a page P; the BM:
  1. checks whether page P is in the BP; if so,  $\text{pin\_count}(F)++$ , where F is the frame containing P  
otherwise:
    - a. BM chooses a frame FR for replacement
      - if the BP contains multiple frames with  $\text{pin\_count} = 0$ , one frame is chosen according to the BM's replacement policy
      - $\text{pin\_count}(FR)++$ ;
    - b. if  $\text{dirty}(FR) = \text{on}$ , BM writes the page in FR to disk
    - c. BM reads page P in frame FR
  2. the BM returns the address of the BP frame that contains P to L

## Buffer Manager

- obs. if no BP frame has `pin_count = 0` and page P is not in BP, BM has to wait / the transaction may be aborted
- page requested by several transactions; no conflicting updates
- crash recovery, Write-Ahead Log (WAL) protocol - additional restrictions when a frame is chosen for replacement
- replacement policies
  - *Least Recently Used (LRU)*
    - queue of pointers to frames with `pin_count = 0`
    - a frame is added to the end of the queue when its `pin_count` becomes 0
    - the frame at the head of the queue is chosen for replacement
  - *Most Recently Used (MRU)*
  - *random*
  - ...

## Buffer Manager

- replacement policies
  - *clock replacement*
    - LRU variant
    - $n$  – number of frames in BP
    - frame - *referenced* bit; set to *on* when *pin\_count* becomes 0
    - *crt* variable - frames 1 through  $n$ , circular order
    - if the current frame is not chosen, then *crt++*, examine next frame
    - if *pin\_count* > 0
      - current frame not a candidate, *crt++*
    - if *referenced* = *on*
      - *referenced* := *off*, *crt++*
    - if *pin\_count* = 0 AND *referenced* = *off*
      - choose current frame for replacement

## Buffer Manager

- replacement policies
  - can have a significant impact on performance
- example:
  - BM uses LRU
  - repeated scans of file  $f$
  - BP: 5 frames,  $f: \leq 5$  pages
    - first scan of  $f$  brings all the pages in the BP
    - subsequent scans find all the pages in the BP
  - BP: 5 frames,  $f: 6$  pages
    - *sequential flooding*: every scan of  $f$  reads all the pages
    - MRU – better in this case

# Disk Space Manager & Buffer Manager

- DSM
  - portability - different OSs
- BM
  - DBMS can anticipate the next several page requests (operations with a known page access pattern, like sequential scans)
  - *prefetching* - BM brings pages in the BP before they are requested
  - prefetched pages
    - contiguous: faster reading (than reading the same pages at different times)
    - not contiguous: determine an access order that minimizes seek times / rotational delays

# Disk Space Manager & Buffer Manager

- BM
  - DBMS needs
    - ability to explicitly force a page to disk
    - ability to write some pages to disk before other pages are written
      - WAL protocol - first write log records describing page changes, then write modified page

# References

- [Ta13] TÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ha96] HANSEN, G., HANSEN, J., Database Management And Design (2<sup>nd</sup> Edition), Prentice Hall, 1996
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3<sup>rd</sup> Edition,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>

# Databases

Lectures 9-10\*

The Physical Structure of Databases. Indexes

\*01.12.2022 - National Day

## Files of Records

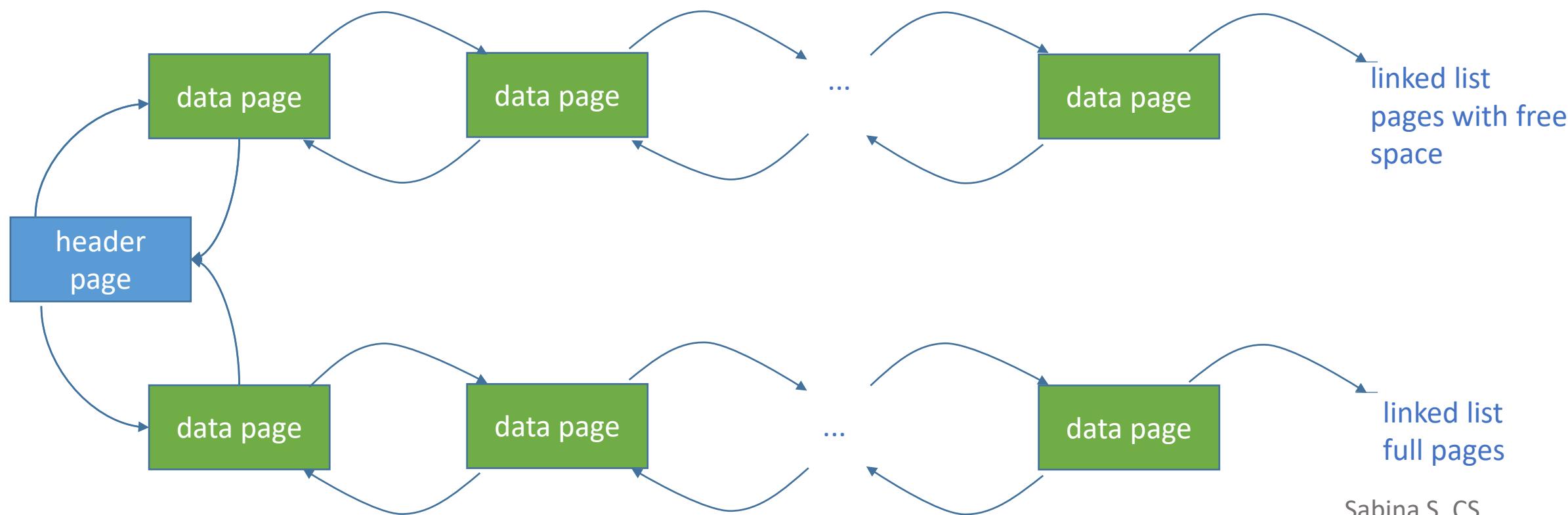
- higher level layers in the DBMS treat pages as collections of records
- file of records
  - collection of records; one or more pages
- different ways to organize a file's collection of pages
- every record has an identifier: the rid
- given the rid of a record, one can identify the page that contains the record

## Heap Files

- the simplest file structure
- records are not ordered
- supported operations
  - create file
  - destroy file
  - insert a record
    - need to monitor pages with free space
  - retrieve a record given its rid
  - delete a record given its rid
  - scan all records
    - need to keep track of all the pages in the file
- appropriate when the expected pattern of use includes scans to obtain all the records

# Heap Files - Linked List

- doubly linked list of pages
  - DBMS stores the address of the first page (*header page*) of each file (a table holding pairs of the form  $\langle \text{heap\_file\_name}, \text{page1\_address} \rangle$ )
  - 2 lists – pages with free space and full pages

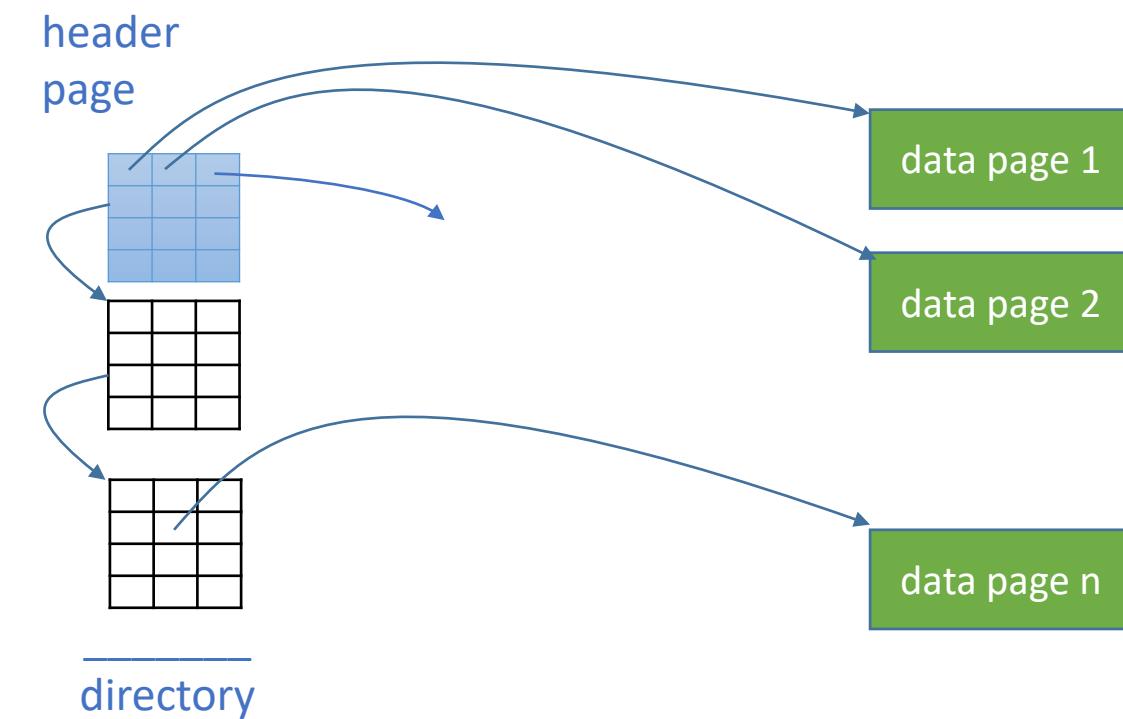


## Heap Files - Linked List

- drawback
  - variable-length records => most of the pages will be in the list of pages with free space
  - when adding a record, multiple pages have to be checked until one is found that has enough free space

## Heap Files - Directory of Pages

- DBMS stores the location of the header page for each heap file
- directory - collection of pages (e.g., linked list)
- directory entry - identifies a page in the file
- directory entry size - much smaller than the size of a page
- directory size - much smaller than the size of the file
- free space management
  - 1 bit / directory entry - corresponding page has / doesn't have free space
  - count / entry - available space on the corresponding page => efficient search of pages with enough free space when adding a variable-length record

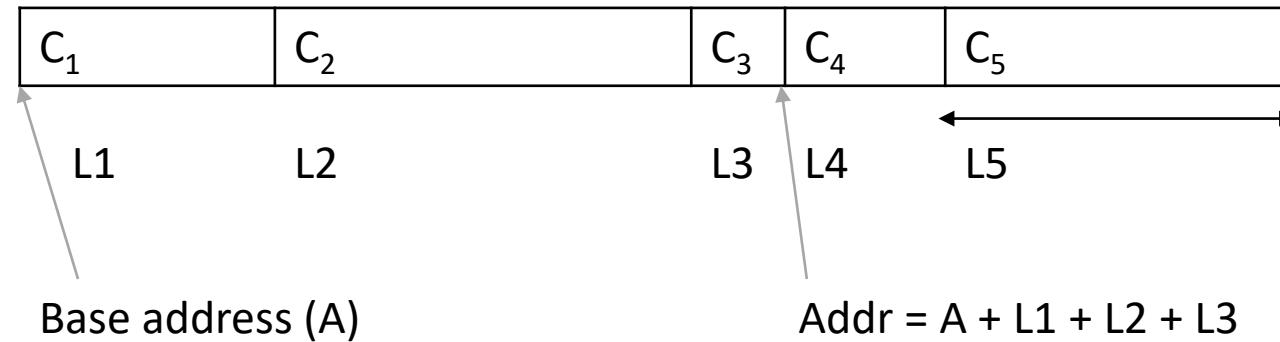


## Other File Organizations

- sorted files
  - suitable when data must be sorted, when doing range selections
- hashed files
  - files that are hashed on some fields (records are stored according to a hash function); good for equality selections

## Record Formats

- fixed-length records



- each field has a fixed length
- fixed number of fields
- fields - stored consecutively
- computing a field's address
  - record address, length of preceding fields (from the system catalog)

## Record Formats

- variable-length records
  - variable-length fields

v1

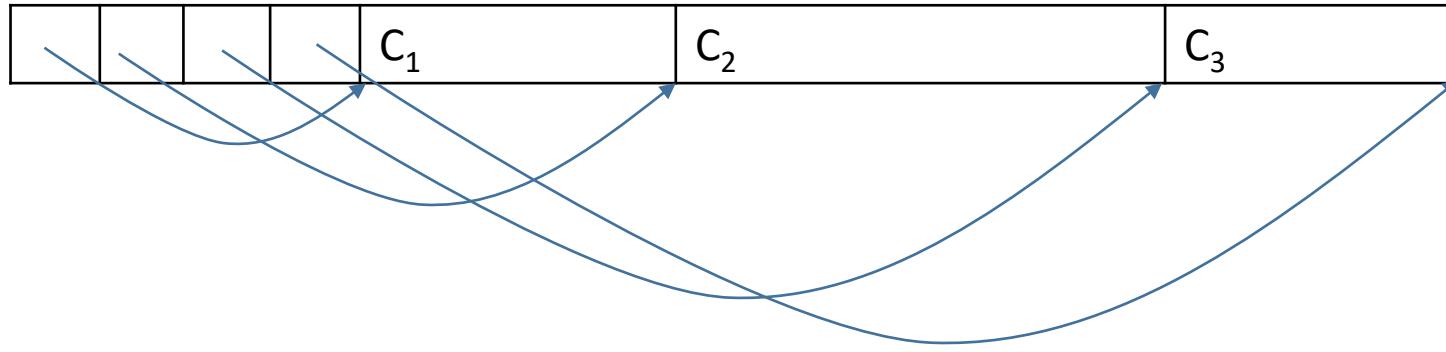
$c_1$	\$	$c_2$	\$	$c_3$	\$
-------	----	-------	----	-------	----

- fields
  - stored consecutively, separated by delimiters
- finding a field
  - a record scan

## Record Formats

- variable-length records

v2



- reserve space at the beginning of the record
  - array of fields offsets, offset to the end of the record
  - array overhead, but direct access to every field

# Page Formats

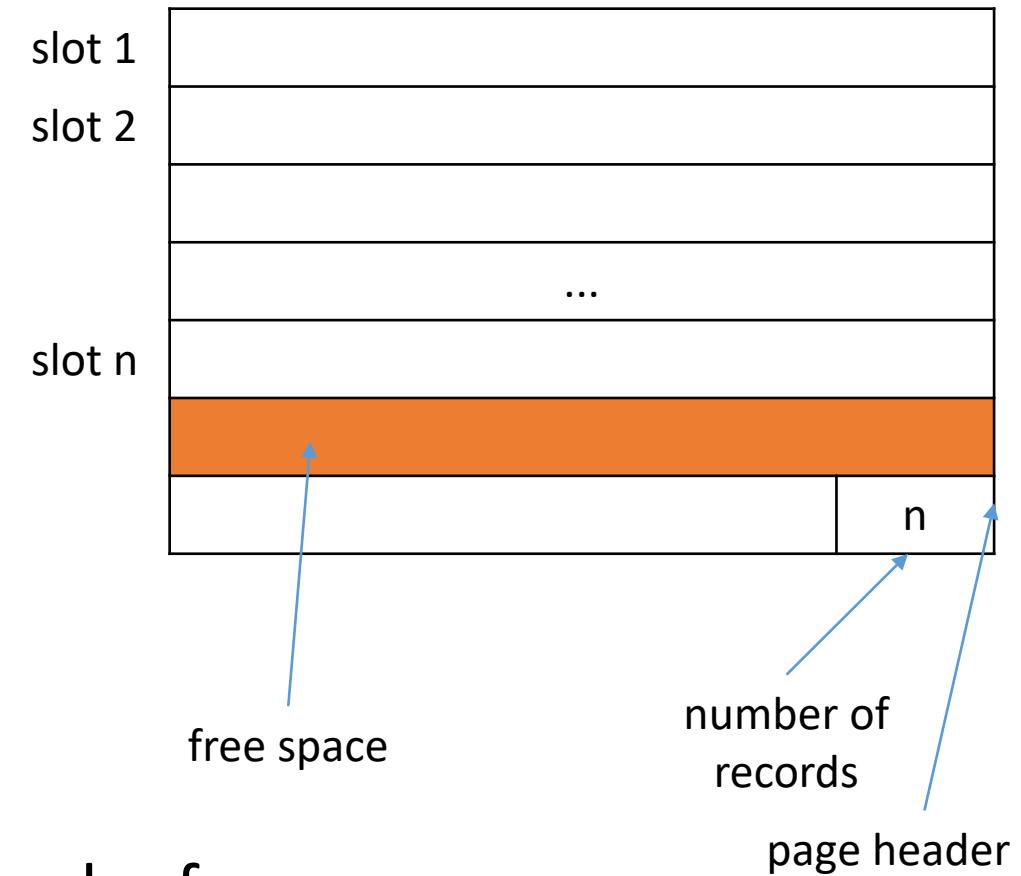
- page
  - collection of slots
  - 1 record / slot
- identifying a record
  - record id (rid): <page id, slot number>
- how to arrange records on pages
- how to manage slots

## Page Formats

- fixed-length records
  - records have the same size
  - uniform, consecutive slots
  - adding a record
    - finding an available slot
- problems
  - keeping track of available slots
  - locating records

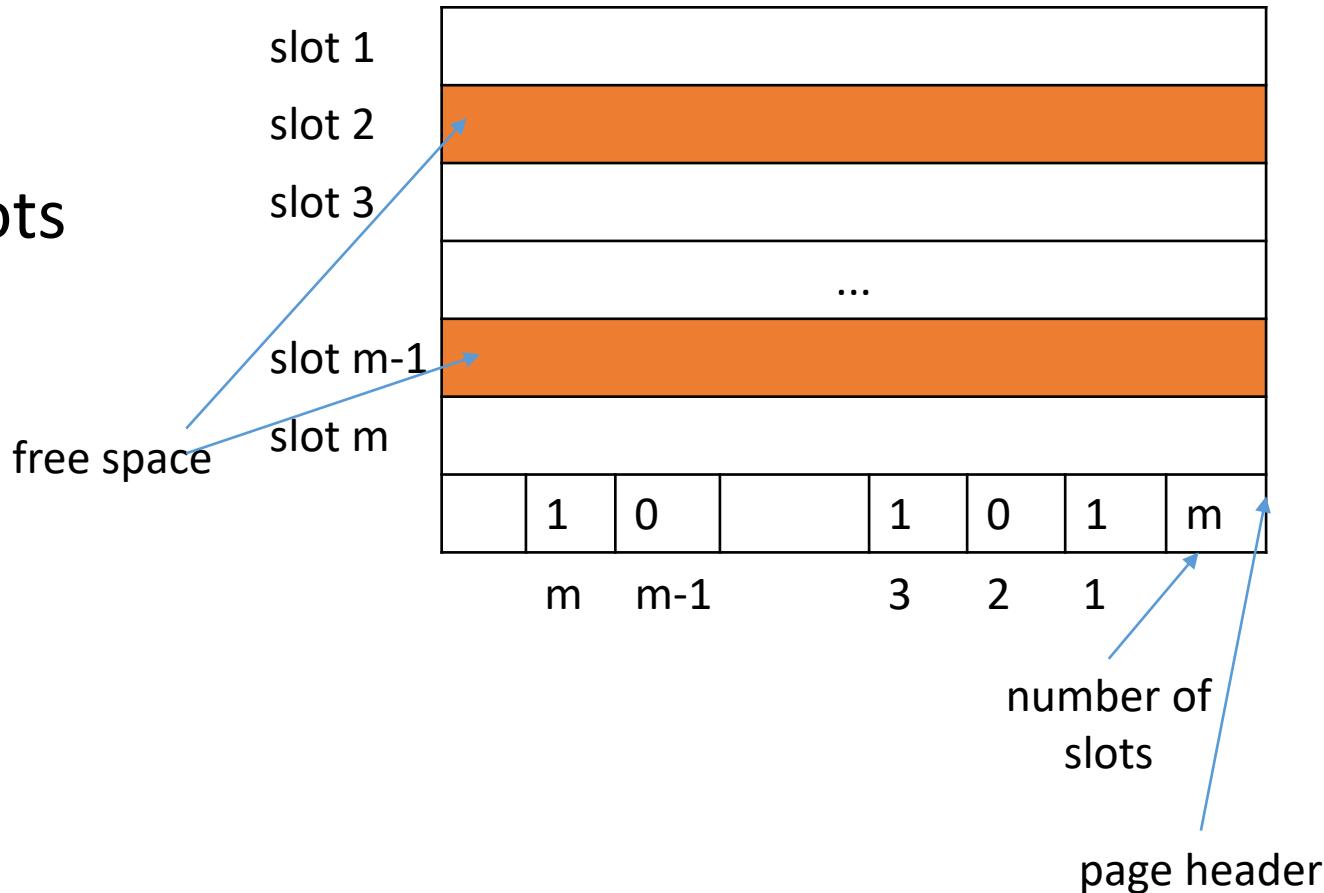
## Page Formats

- fixed-length records - v1
  - $n$  – number of records on the page
  - records are stored in the first  $n$  slots
  - locating record  $i$  - compute corresponding offset
  - deleting a record - the last record on the page is moved into the empty slot
  - empty slots - at the end of the page
- problems when a moved record has external references
  - the record's slot number would change, but the rid contains the slot number!



## Page Formats

- fixed-length records - v2
- array of bits to monitor available slots
- 1 bit / slot
- deleting a record - turning off the corresponding bit

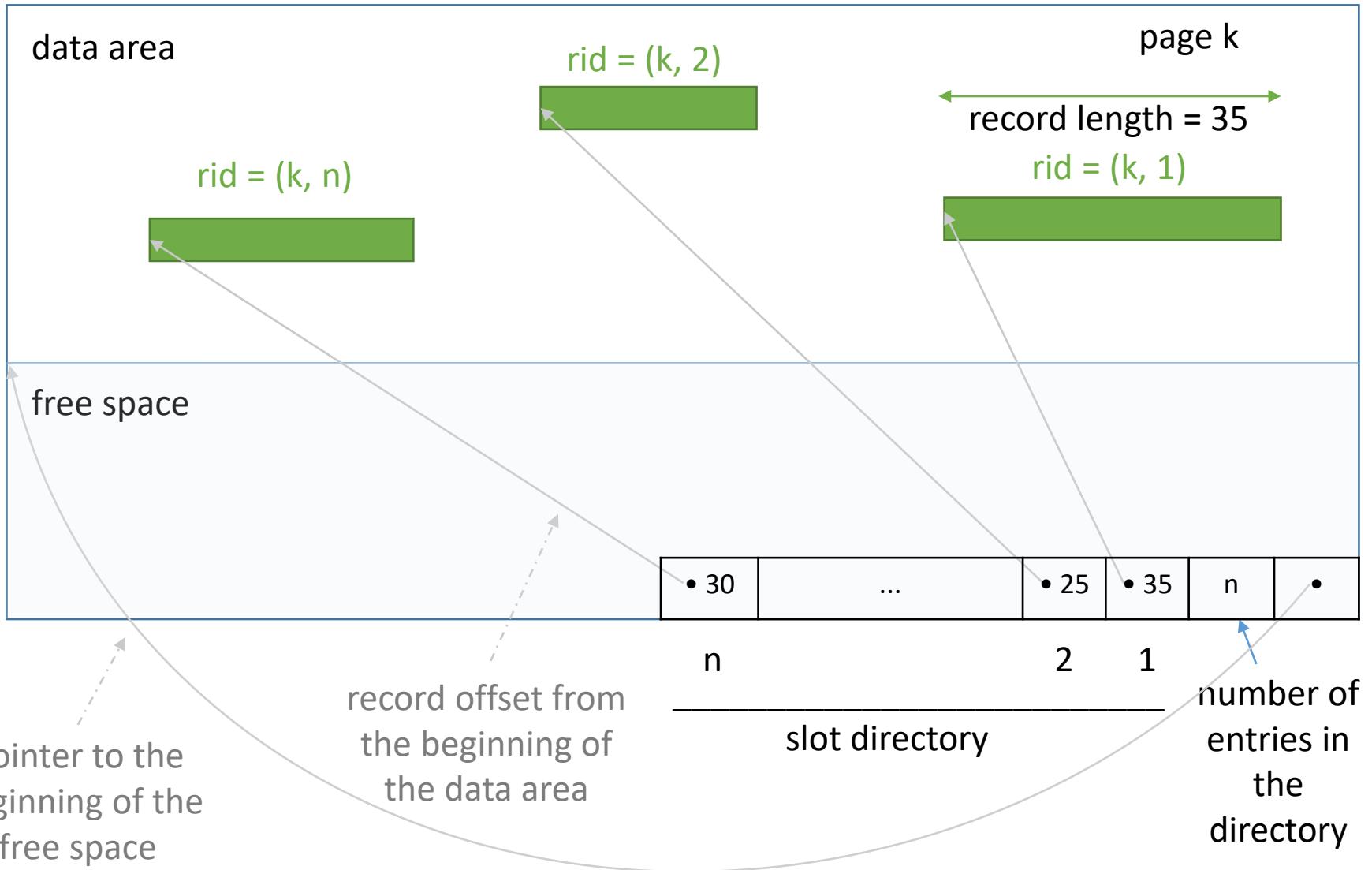


## Page Formats

- variable-length records
  - adding a record
    - finding an empty slot of the right size
  - deleting a record
    - contiguous free space
  - a directory of slots / page
  - a pair <record offset , record length> / slot
  - a pointer to the beginning of the free space area on the page
  - moving a record on the page
    - only the record's offset changes
    - its slot number remains unmodified
  - can also be used for fixed-length records (when records need to be kept sorted)

# Page Formats

- variable-length records



## Indexes

- motivating example
  - file of students records sorted by name
    - good file organization
      - retrieve students in alphabetical order
    - not a good file organization
      - retrieve students whose age is in a given range
      - retrieve students who live in Timișoara
- index
  - auxiliary data structure that speeds up operations which can't be efficiently carried out given the file's organization
  - enables the retrieval of the rids of records that meet a selection condition (e.g., the rids of records describing students who live in Timișoara)

## Indexes

- *search key*
  - set of one or more attributes of the indexed file (different from the *key* that identifies records)
- an index speeds up queries with equality / range selection conditions on the search key
- *entries*
  - records in the index (e.g., <search key, rid>)
  - enable the retrieval of records with a given search key value

## Indexes

- example
  - files with students records
  - index built on attribute *city*
  - entries: <city, rid>, where rid identifies a student record
  - such an index would speed up queries about students living in a given city:
    - find entries in the index with city = '*Timișoara*'
    - follow rids from obtained entries to retrieve records describing students who live in Timișoara

## Indexes

- an index can improve the efficiency of certain types of queries, not of all queries (analogy - when searching for a book at the library, index cards sorted on author name cannot be used to efficiently locate a book given its title)
- organization techniques (access methods) - examples
  - B+ trees
  - hash-based structures
- changing the data in the file => update the indexes associated with the file (e.g., inserting records, updating search key columns, updating columns that are not part of the key, but are included in the index)
- index size
  - as small as possible, as indexes are brought into main memory for searches

## Indexes - Data Entries

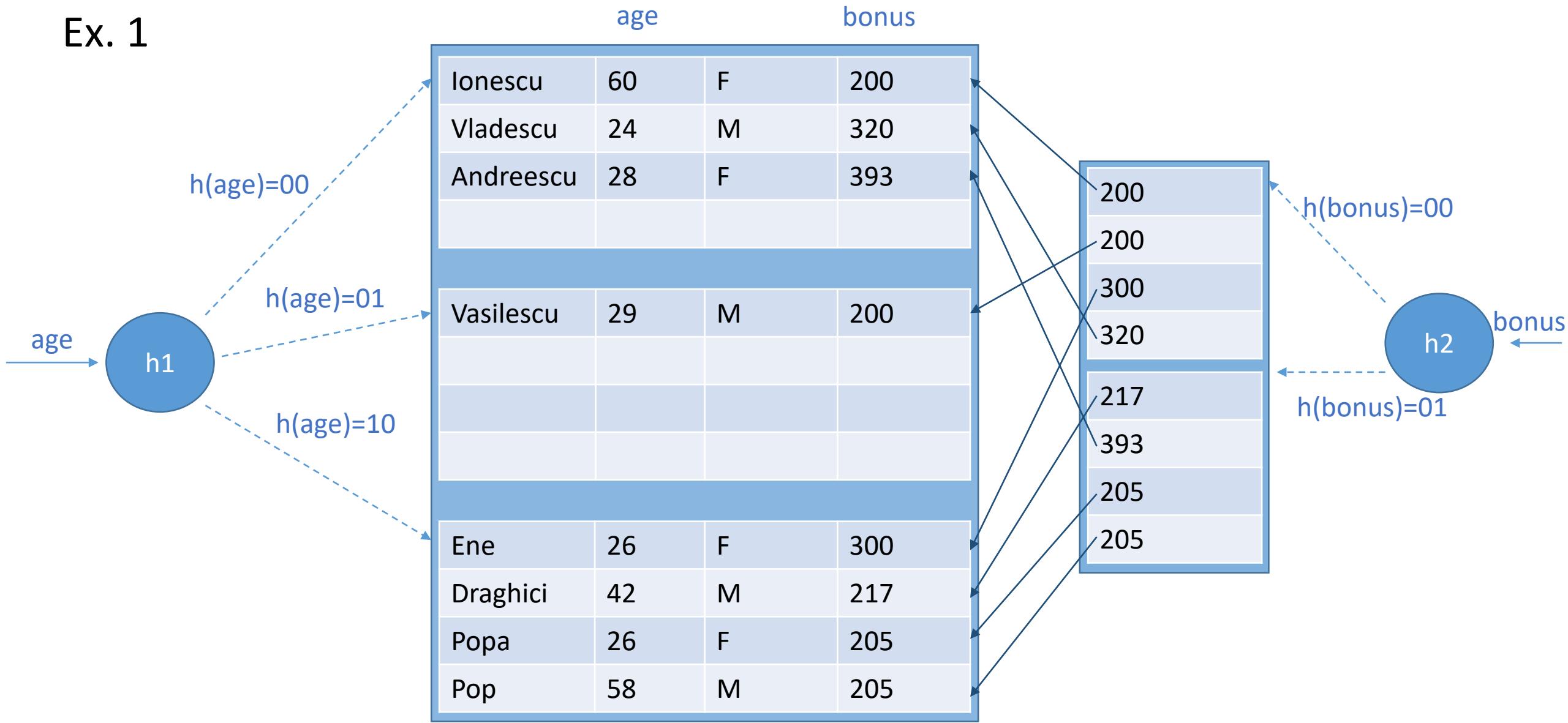
- problems
  - what does a data entry contain?
  - how are the entries of an index organized?
- let  $k^*$  be a data entry in an index; the data entry:
  - alternative 1
    - is an actual data record with search key value =  $k$
  - alternative 2
    - is a pair  $\langle k, \text{rid} \rangle$  ( $\text{rid}$  – id of a data record with search key value =  $k$ )
  - alternative 3
    - is a pair  $\langle k, \text{rid\_list} \rangle$  ( $\text{rid\_list}$  – list of ids of data records with search key value =  $k$ )

## Indexes - Data Entries

- a1
  - the file of data records needn't be stored in addition to the index
  - the index is seen as a special file organization
  - at most 1 index / collection of records should use alternative a1 (to avoid redundancy)
- a2, a3
  - data entries point to corresponding data records
  - in general, the size of an entry is much smaller than the size of a data record
  - a3 is more compact than a2, but can contain variable-length records
  - can be used by several indexes on a collection of records
  - independent of the file organization

# Indexes - Data Entries

Ex. 1



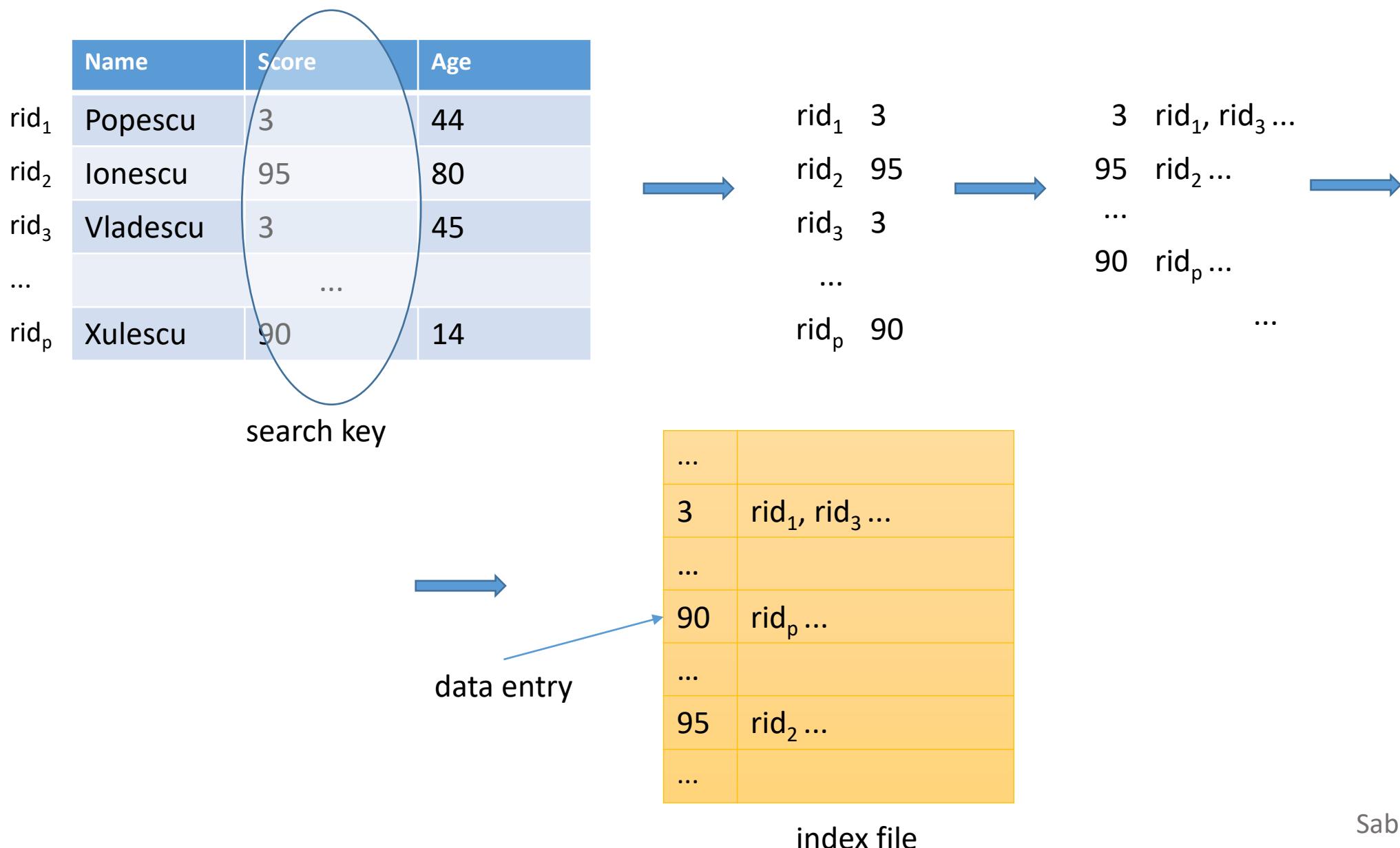
## Indexes - Data Entries

### Ex. 1

- file with Employee records hashed on *age*
  - record <Ionescu, 60, F, 200>:
    - apply hash function to *age*: convert 60 to its binary representation, take the 2 least significant bits as the bucket identifier for the record
- index file that uses alternative 1 (data entries are the actual data records), search key *age*
- index that uses alternative 2 (data entries have the form <search key, rid>), search key *bonus*
- both indexes use hashing to locate data entries

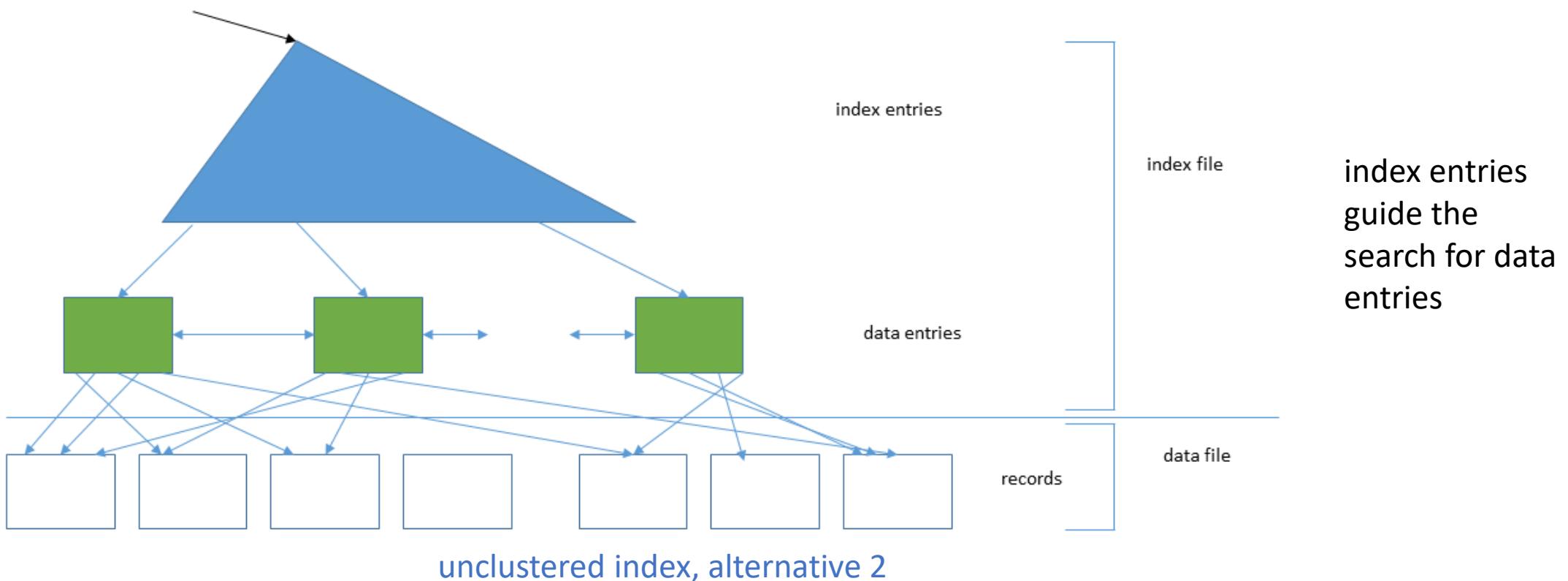
# Indexes - Data Entries

Ex. 2



# Clustered / Unclustered Indexes

- clustered index: the order of the data records is close to / the same as the order of the data entries
- unclustered index: index that is not clustered



## Clustered / Unclustered Indexes

- index that uses alternative 1 - clustered (by definition, since the data entries are the actual data records)
- indexes using alternatives 2 / 3 are clustered only if the data records are ordered on the search key
- in practice:
  - expensive to maintain the sort order for files, so they are rarely kept sorted
  - a clustered index is an index that uses alternative 1 for data entries
  - an index that uses alternative 2 or 3 for data entries is unclustered
- on a collection of records:
  - there can be at most 1 clustered index
  - and several unclustered indexes

## Clustered / Unclustered Indexes

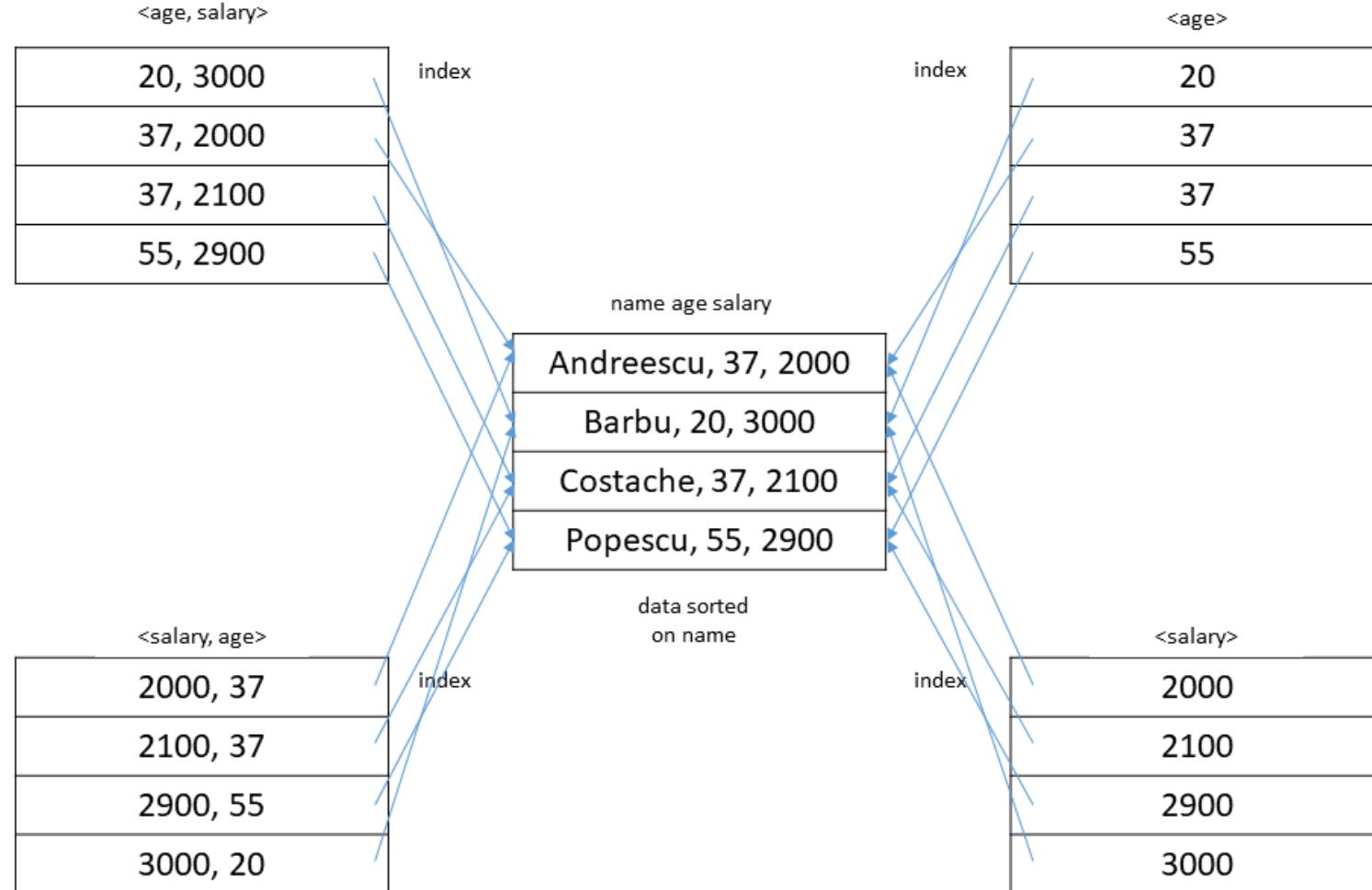
- range search query (e.g., *where age between 20 and 30*)
  - cost of using an unclustered index
    - each data entry that meets the condition in the query could contain a rid pointing to a distinct page
    - the number of I/O operations could be equal to the number of data entries that satisfy the query's condition

## Primary / Secondary Indexes

- primary index
  - the search key includes the primary key
- secondary index
  - index that is not primary
- unique index
  - the search key contains a candidate key
- duplicates
  - data entries with the same search key value
- primary indexes, unique indexes cannot contain duplicates

# Composite Search Keys

- composite (concatenated) search key - search key that contains several fields
  - examples



## References

- [Ta13] ȚÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3<sup>rd</sup> Edition,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, <http://codex.cs.yale.edu/avi/db-book/>

# Databases

Lecture 11

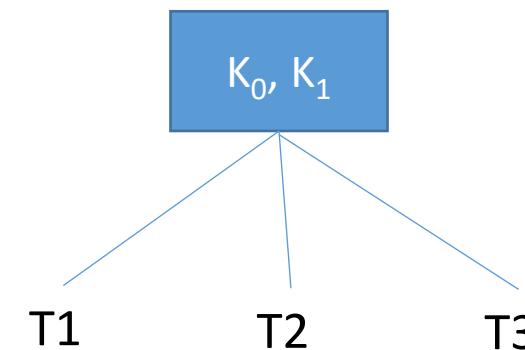
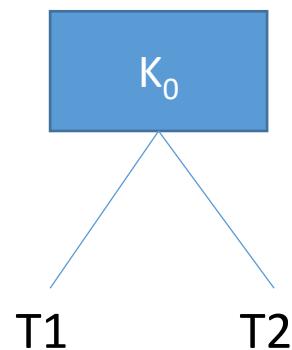
Tree-Structured Indexing. Hash-Based Indexing

# Tree-Structured Indexing

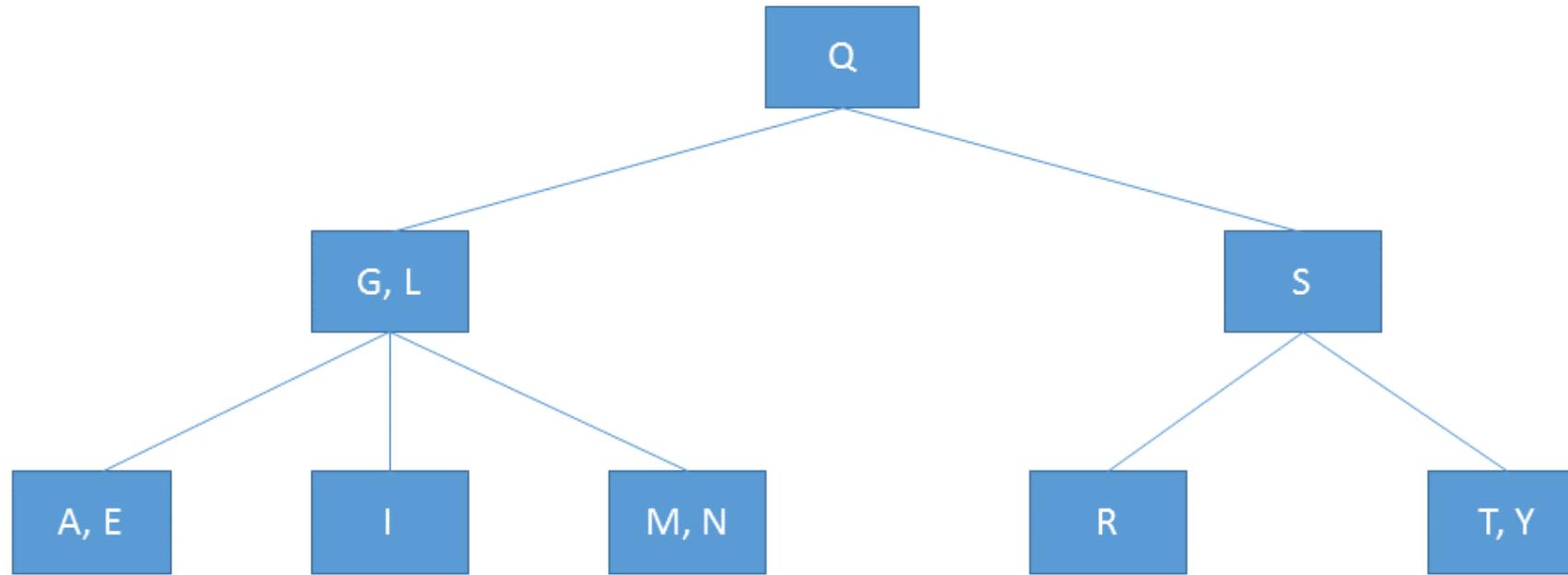
## 2-3 trees

2-3 tree storing key values (collection of distinct values)

- all the terminal nodes are on the same level
- every node has 1 or 2 key values
  - a non-terminal node with one value  $K_0$  has 2 subtrees: one with values less than  $K_0$ , and one with values greater than  $K_0$
  - a non-terminal node with 2 values  $K_0$  and  $K_1$ ,  $K_0 < K_1$ , has 3 subtrees: one with values less than  $K_0$ , a subtree with values between  $K_0$  and  $K_1$ , and a subtree with values greater than  $K_1$



\* Example (key values are letters)



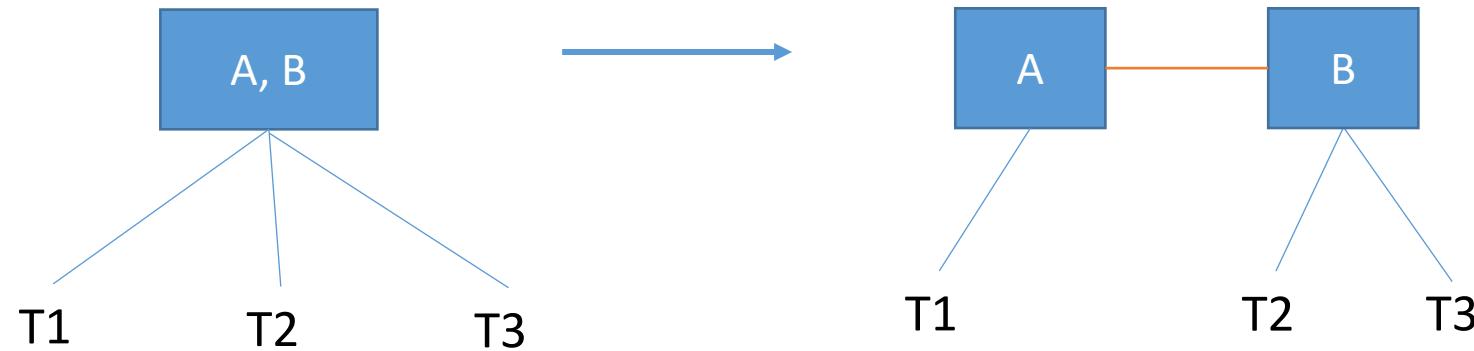
- storing a 2-3 tree

- 2-3 tree index storing the values of a key
- tree - key value + address of record (file / DB address of record with corresponding key value)

- 2 options

1. transform 2-3 tree into a binary tree

- nodes with 2 values are transformed (see figure below)
- nodes with 1 value - unchanged



- the structure of a node



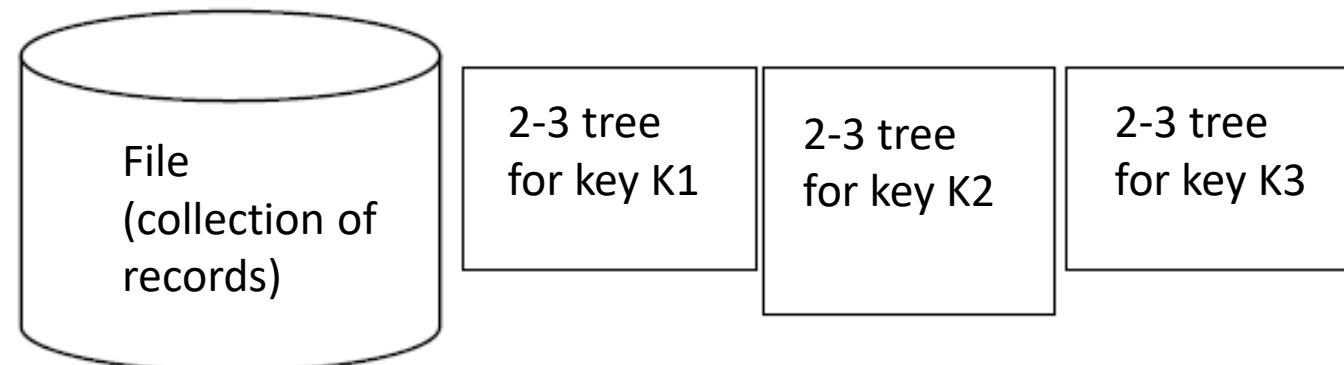
- K - key value
- ADDR - address of the record with the current key value (address in the file)
- PointerL, PointerR - the 2 subtrees' addresses (address in the tree)

- IND - indicator that specifies the type of the link to the right (the 2 possible values can be seen in the previous figure)

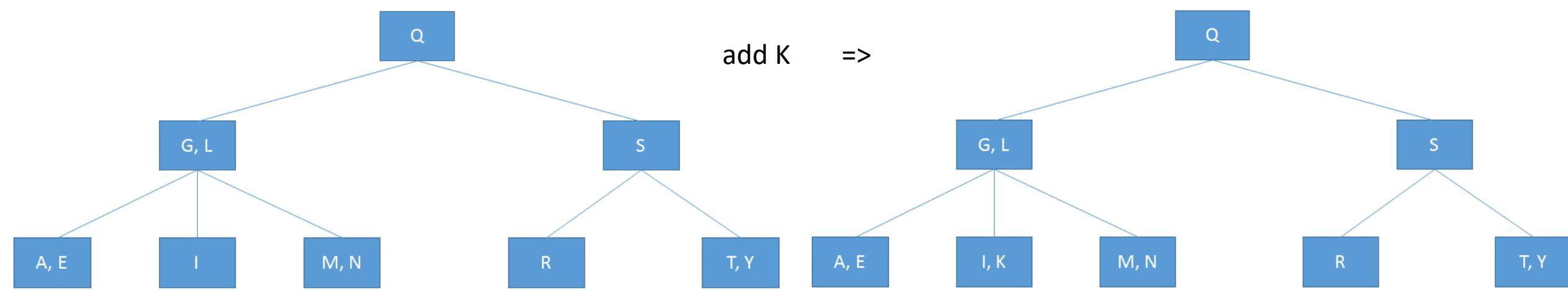
2. the memory area allocated for a node can store 2 values and 3 subtree addresses

NV	$K_1$	$ADDR_1$	$K_2$	$ADDR_2$	$Pointer_1$	$Pointer_2$	$Pointer_3$
----	-------	----------	-------	----------	-------------	-------------	-------------

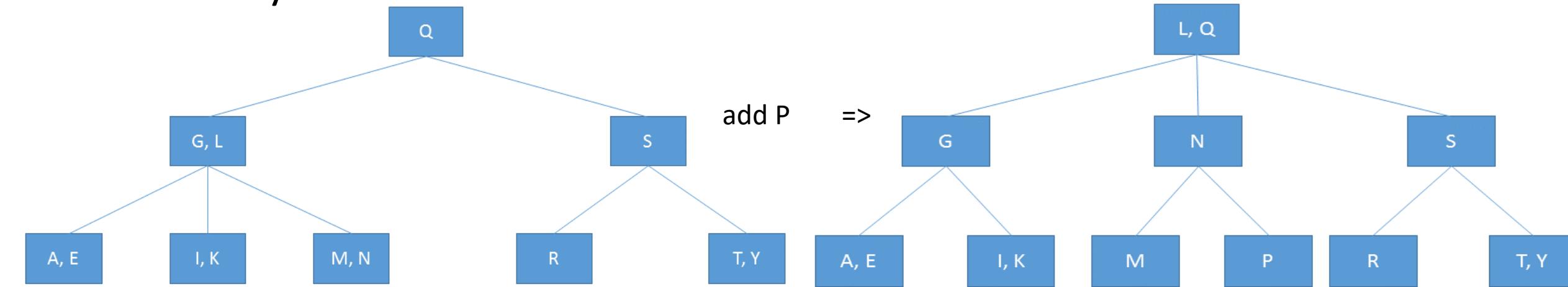
- NV – number of values in the node (1 or 2)
- $K_1, K_2$  – key values
- $ADDR_1, ADDR_2$  – the records' addresses (corresponding to  $K_1$  and  $K_2$ )
- $Pointer_1, Pointer_2, Pointer_3$  – the 3 subtrees' addresses
- obs. a file (a relation in a relational DB) can have several associated 2-3 trees (one tree / key)



- operations in a 2-3 tree
  - searching for a record with key value  $K_0$
  - inserting a record - description
  - removing a record - description
  - tree traversal (partial, total)
- add a new value
  - values in the tree must be distinct (the new value should not exist in the tree)
  - perform a test: search for the value in the tree; if the new value can be added, the search ends in a terminal node
  - if the reached terminal node has 1 value, the new value can be stored in the node



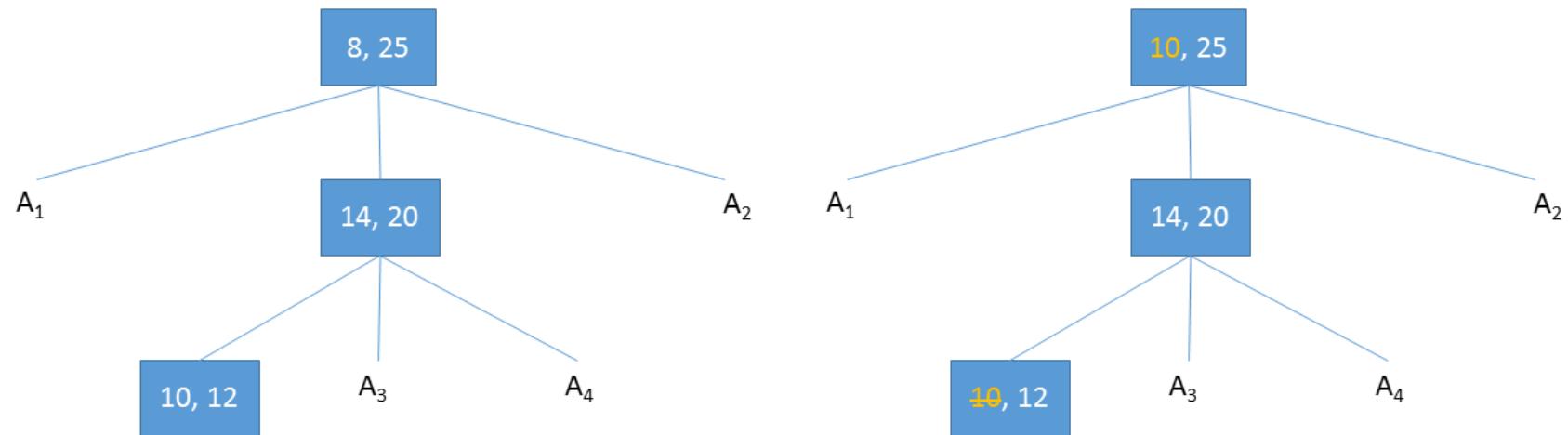
- if the reached terminal node has 2 values, the new value is added to the node, the 3 values are sorted, the node is split into 2 nodes: one node will contain the smallest value, the 2<sup>nd</sup> node - the largest value, and the middle value is attached to the parent node; the parent is then analyzed in a similar manner



- delete a value  $K_0$

1. search for  $K_0$ ; if  $K_0$  appears in an inner node, change it with a neighbor value  $K_1$  from a terminal node (there is no other value between  $K_0$  and  $K_1$ )
  - $K_1$ 's previous position (in the terminal node) is eliminated

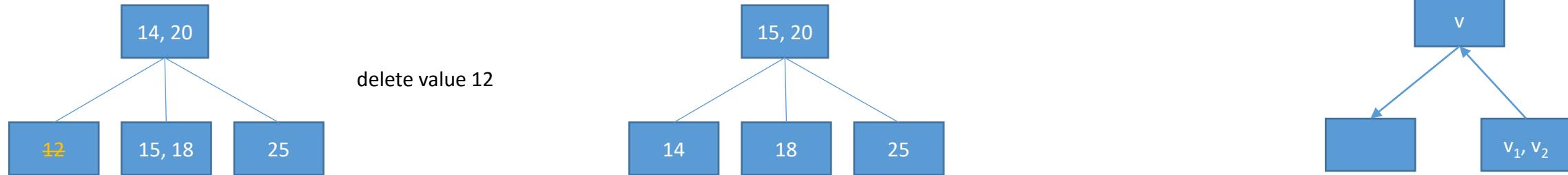
- e.g., remove 8:



2. perform this step until case a / b occurs

- a. if the current node (from which a value is removed) is the root or a node with 1 remaining value, the value is eliminated; the algorithm ends

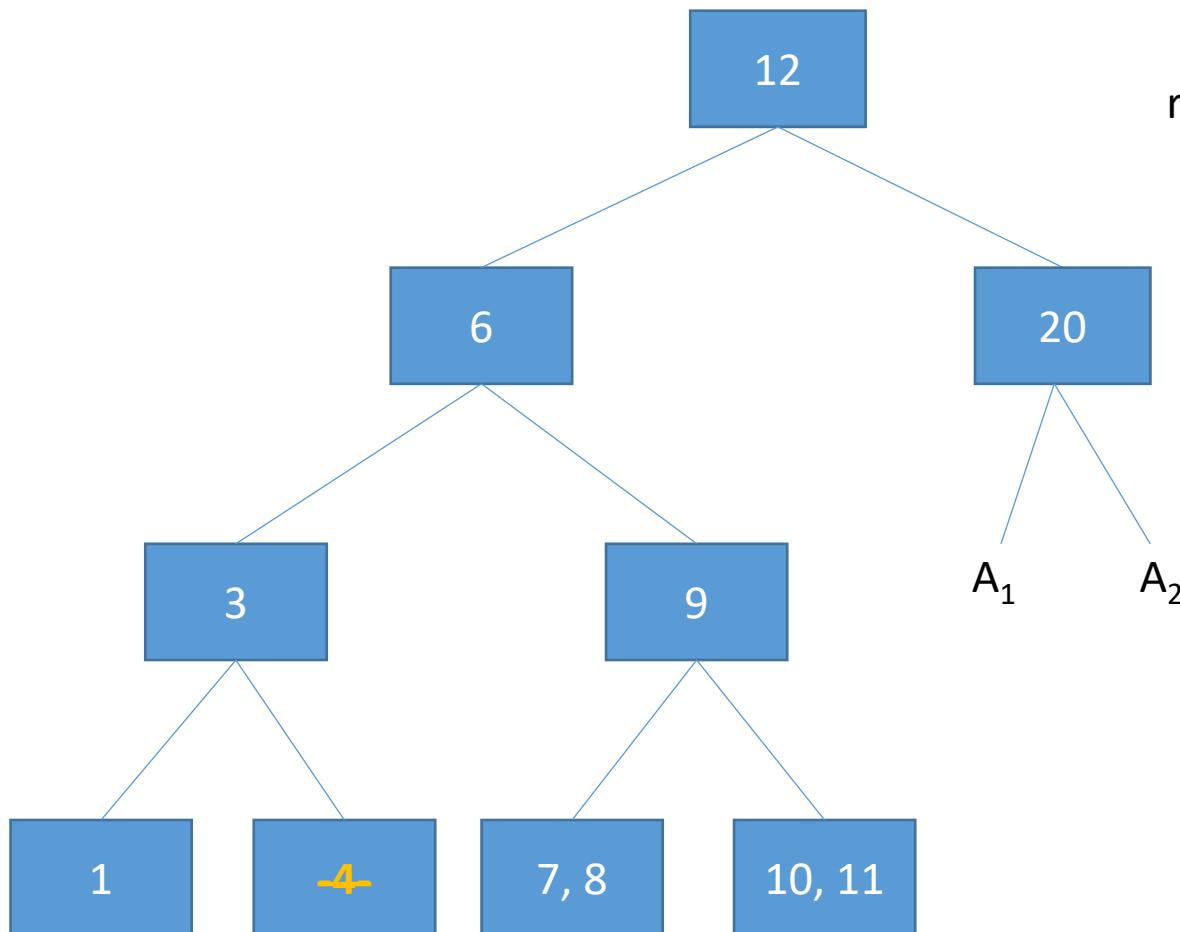
b. if the delete operation empties the current node, but 2 values exist in one of the sibling nodes (left / right), 1 of the sibling's values is transferred to the parent, 1 of the parent's values is transferred to the current node; the algorithm ends



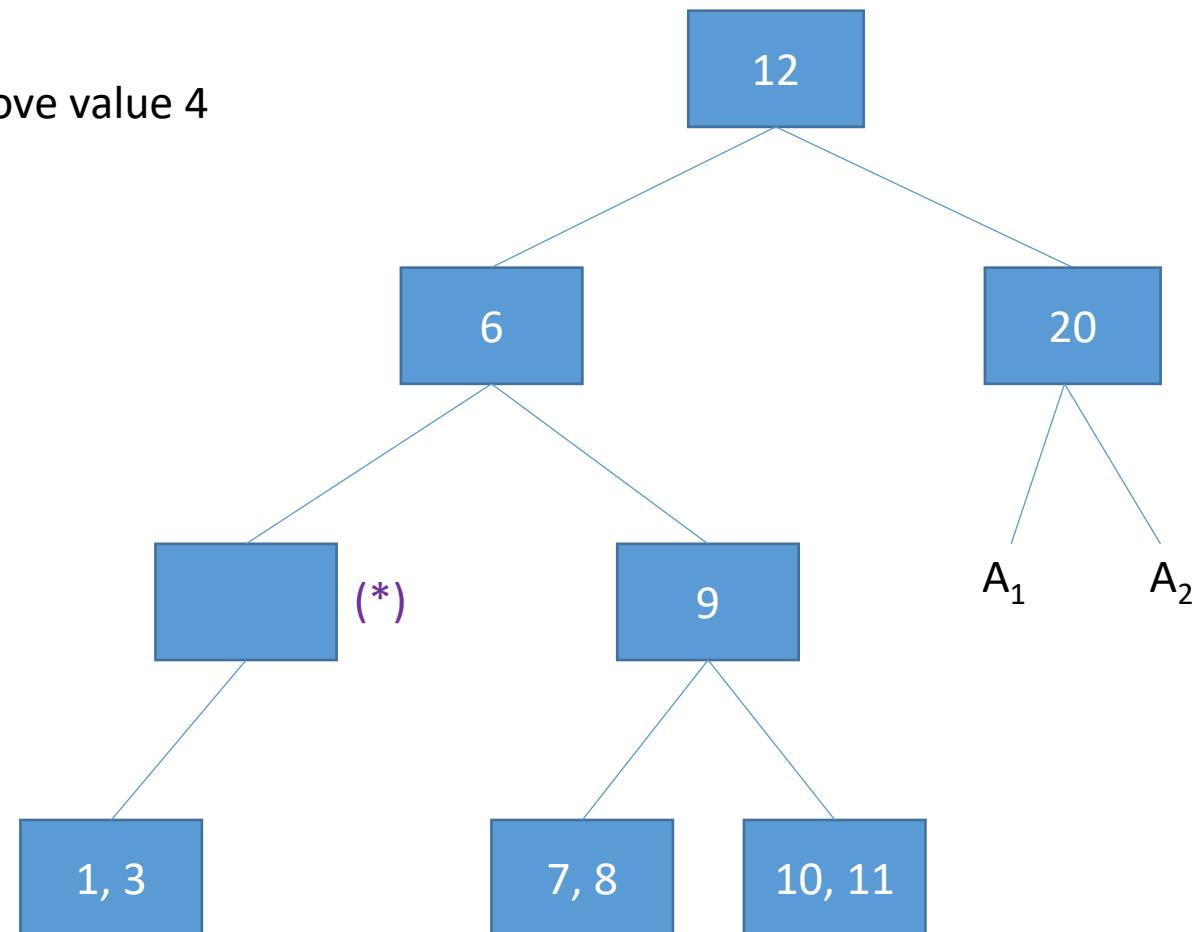
c. if the previous cases do not occur (current node has no values, sibling nodes have 1 value each), then the current node is merged with a sibling and a value from the parent node; case 2 is then analyzed for the parent

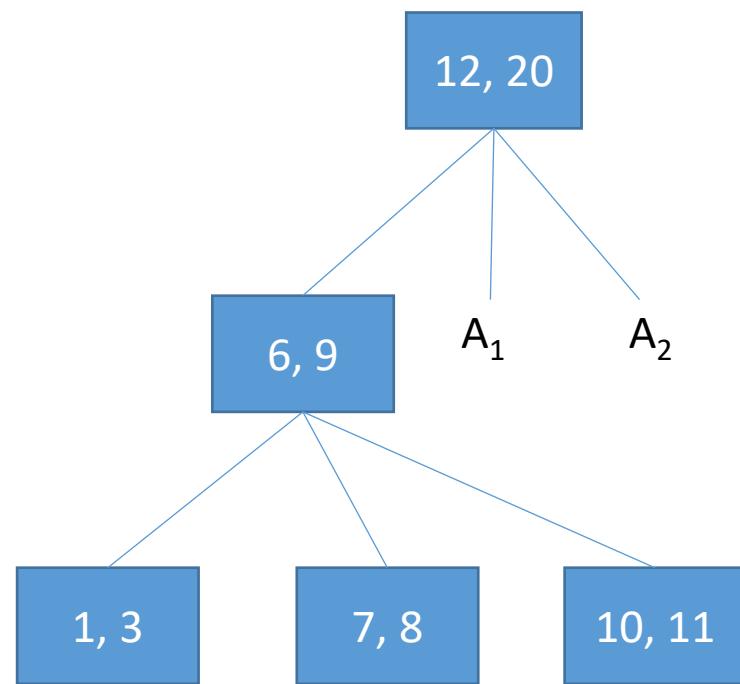
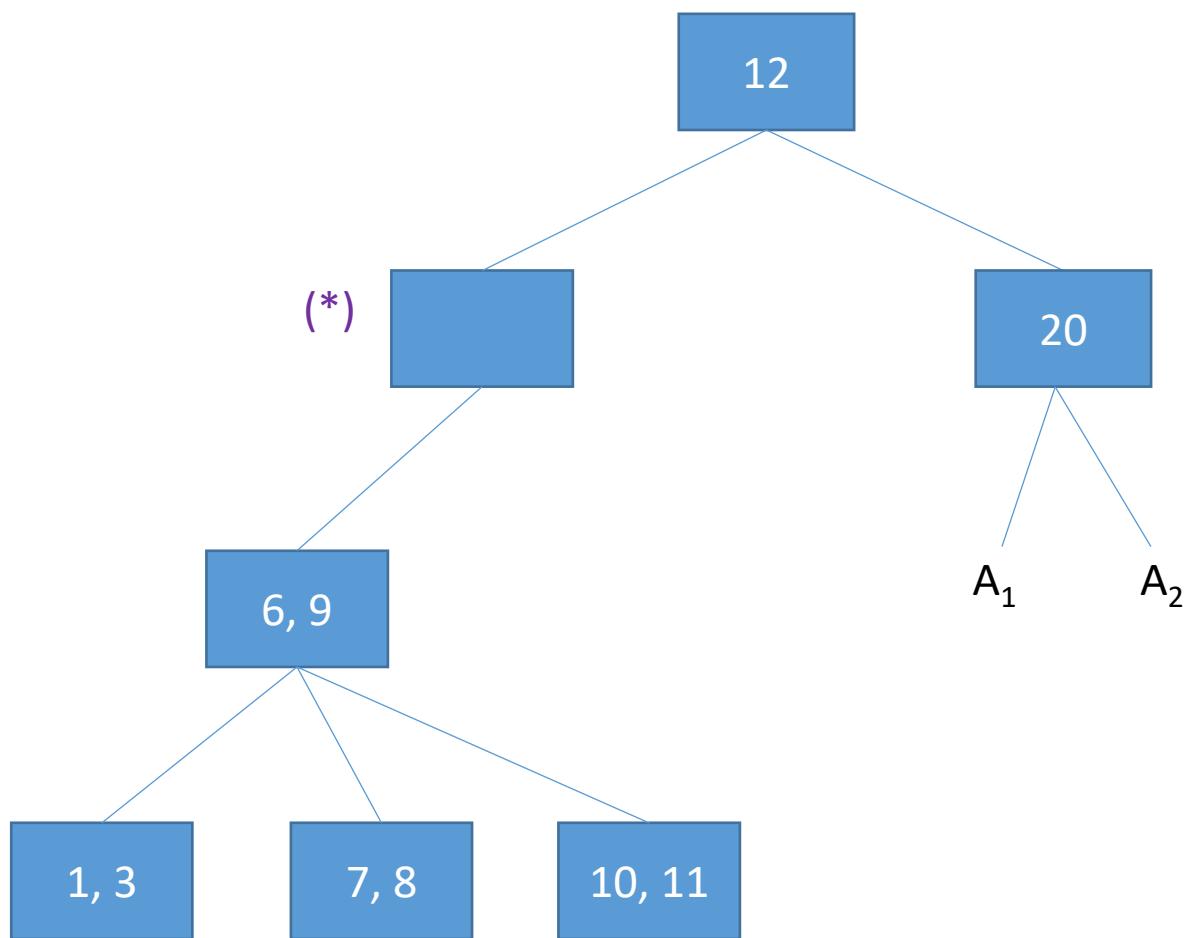
- if the root is reached and it has no values, it is eliminated and the current node becomes the root

- example: case c for the node marked with (\*)



remove value 4





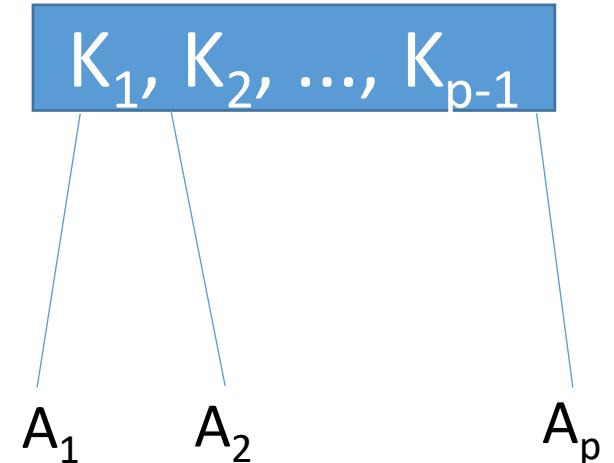
## B-trees - generalization of 2-3 trees

- B-tree of order  $m$ 
  1. if the root is not a terminal, it has at least 2 subtrees
  2. all terminal nodes – same level
  3. every non-terminal node – at most  $m$  subtrees
  4. a node with  $p$  subtrees has  $p-1$  ordered values (ascending order):  $K_1 < K_2 < \dots < K_{p-1}$

- $A_1$ : values less than  $K_1$
- $A_i$ : values between  $K_{i-1}$  and  $K_i$ ,  $i=2,\dots,p-1$
- $A_p$ : values greater than  $K_{p-1}$

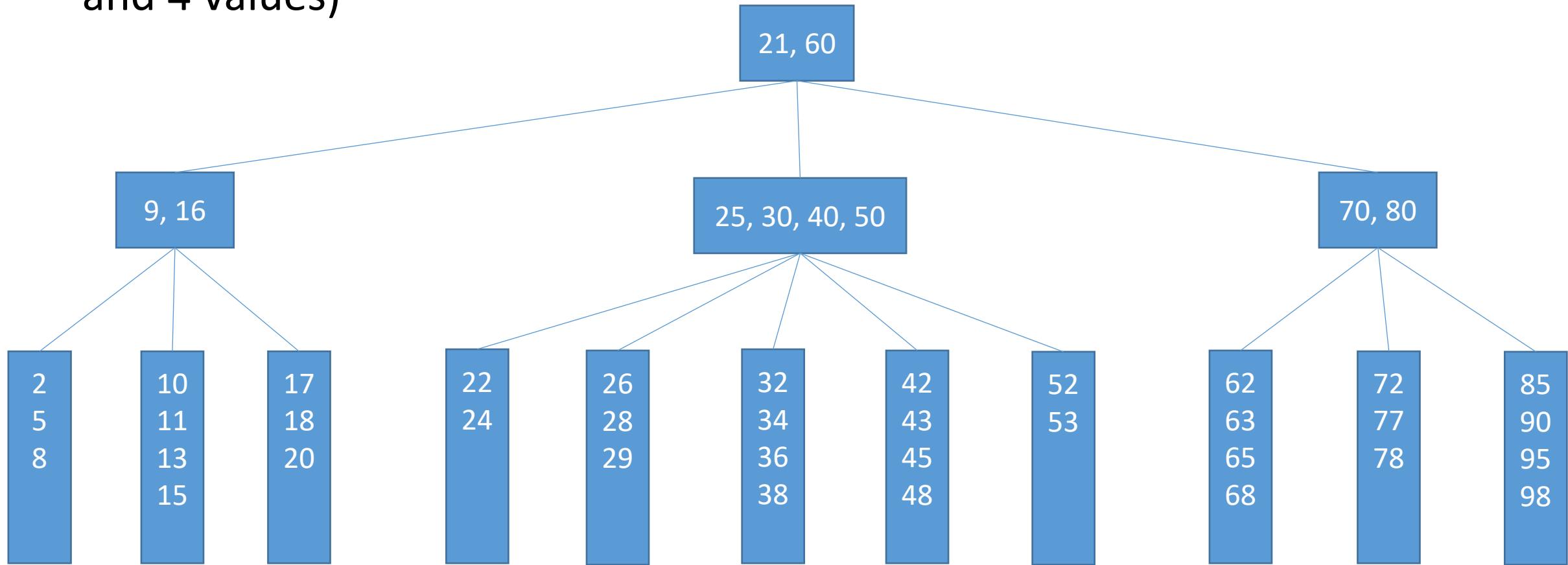
- 5. every non-terminal node – at least  $\left\lceil \frac{m}{2} \right\rceil$  subtrees

- obs. limits on number of subtrees (and values) / node result from the manner in which inserts / deletes are performed such that the second requirement in the definition is met



## \* Example - B-tree of order 5

- non-terminal, non-root node – at most 5, at least 3 subtrees (between 2 and 4 values)



- B-tree of order m
    - storing the values of a key (a database index)
    - tree
      - key value + address of record
1. transformed into a binary tree
    - 2-3 tree method
  2. the memory area allocated for a node can store the maximum number of values and subtree addresses

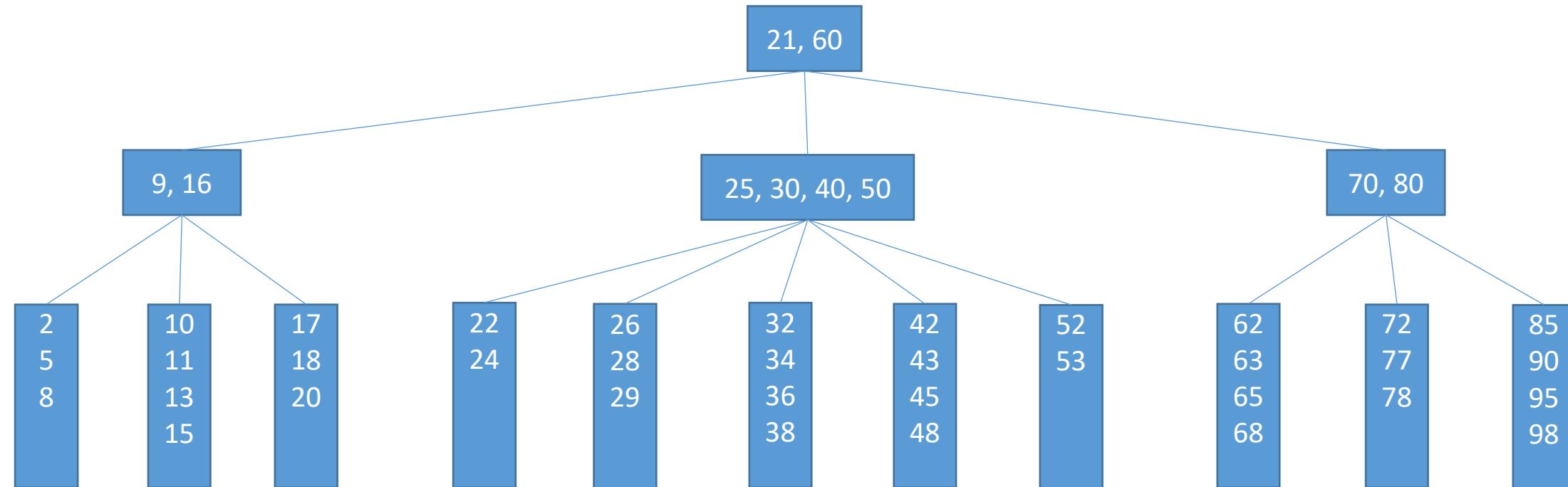
NV	$K_1$	$ADDR_1$	...	$K_{m-1}$	$ADDR_{m-1}$	$Pointer_1$	...	$Pointer_m$
----	-------	----------	-----	-----------	--------------	-------------	-----	-------------

- NV - number of values in the node
- $K_1, \dots, K_{m-1}$  - key values
- $ADDR_1, \dots, ADDR_{m-1}$  - the records' addresses (corresponding to the key's values)
- $Pointer_1, \dots, Pointer_m$  – subtree addresses

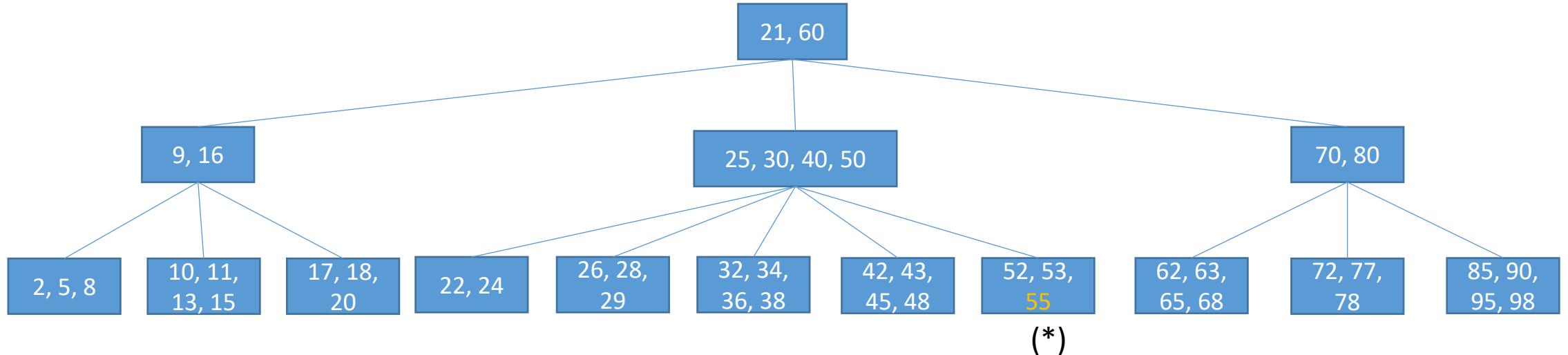
- B-tree of order m
  - useful operations in a B-tree
    - searching for a value
    - adding a value - description
    - removing a value- description
    - tree traversal (partial, total)

- B-tree of order m
  - adding a new value

1. values in the tree must be distinct (the new value should not exist in the tree); perform a test (search for the value in the tree)
  - if the new value can be added, the search ends in a terminal node
2. if the reached terminal node has less than  $m-1$  values, the new value can be stored in the node, e.g., 55 is added to the tree below:

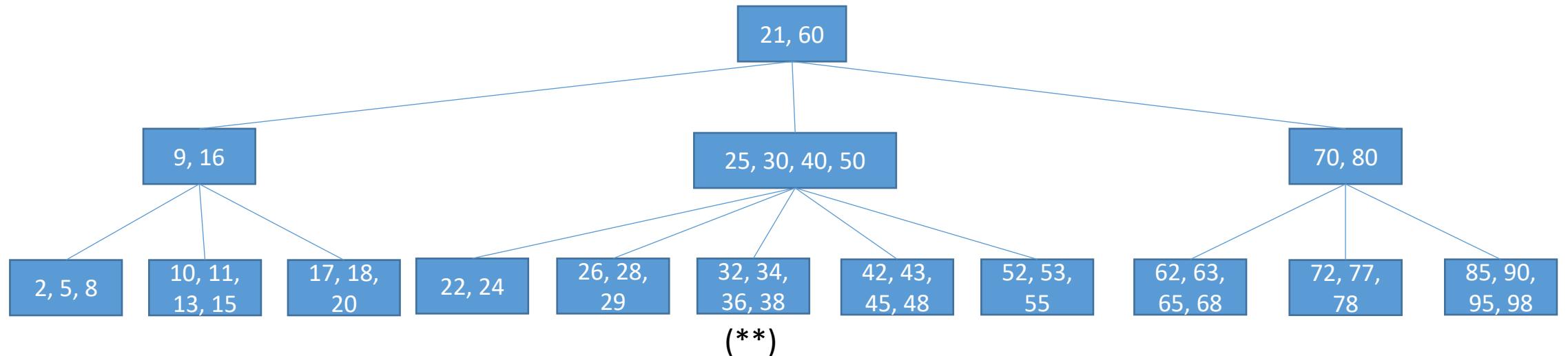


- B-tree of order m
  - adding a new value
  - the resulting tree is shown below:



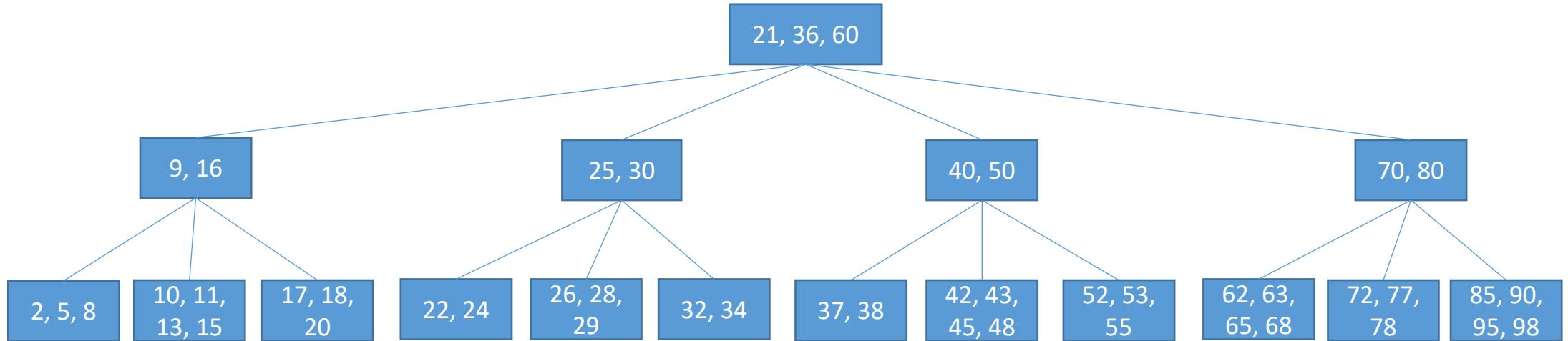
- 55 belongs to the node marked with (\*), which can store at most 4 values

- B-tree of order m
  - adding a new value
    3. if the terminal node already has  $m-1$  values, the new value is attached to the node, the  $m$  values are sorted, the node is split into 2 nodes, and the middle value (median) is attached to the parent node; the parent is then analyzed in a similar manner
      - e.g., add 37 to the tree below



- the node marked with (\*\*) should contain values 32, 34, 36, 37, 38

- B-tree of order m
  - adding a new value
    - since the node's capacity is exceeded, it is split into nodes 32, 34, and 37, 38, and 36 is attached to the parent node (with values 25, 30, 40, 50)
    - in turn, the parent must be split into 2 nodes (values 25, 30, and 40, 50), and 36 is attached to its parent



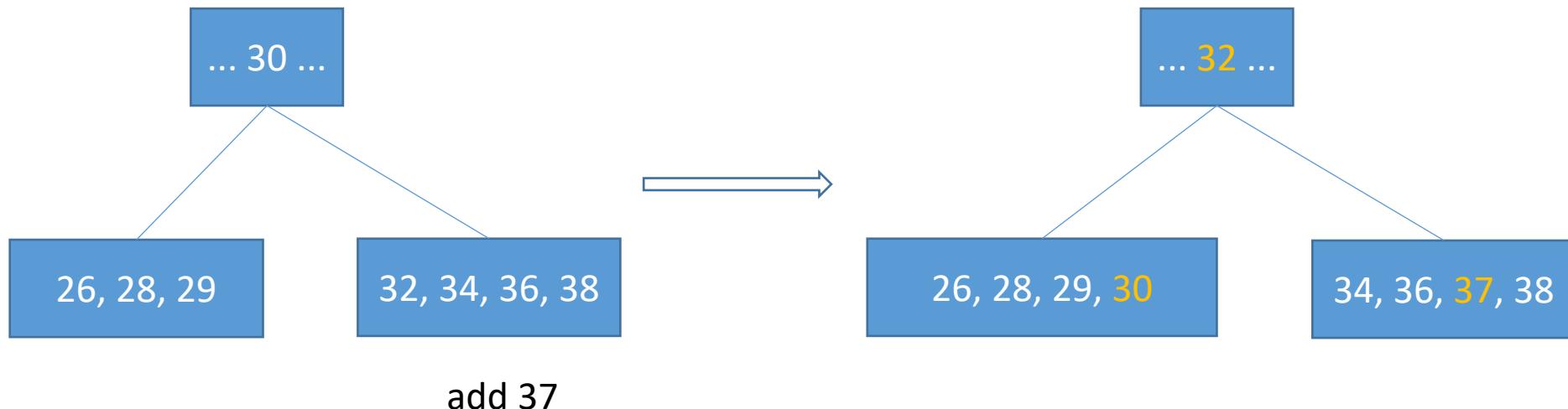
- B-tree of order m

- adding a new value

- optimizations

- before performing a split - analyze whether one or more values can be transferred from the current node (with  $m-1$  values) to a sibling node

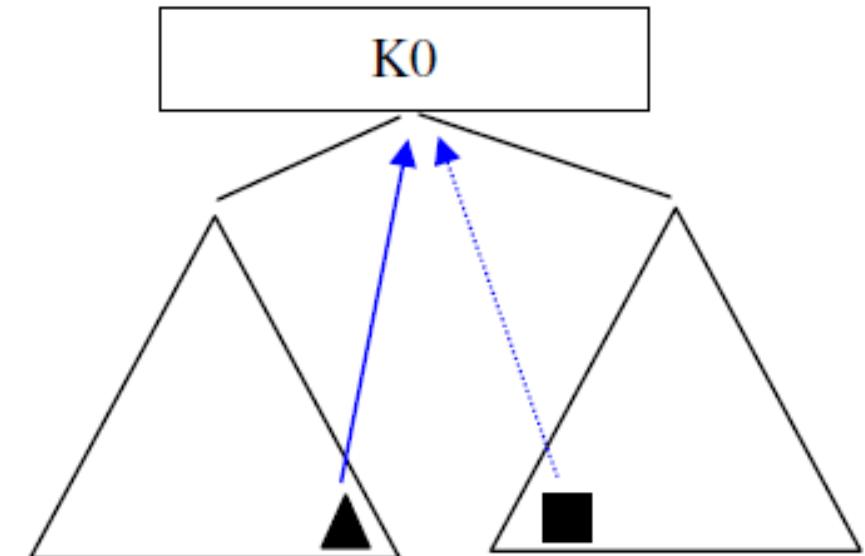
- e.g., B-tree of order 5 (non-terminal node - between 2 and 4 values, i.e., between 3 and 5 subtrees):



- B-tree of order m
  - adding a new value
  - optimizations
  - e.g., B-tree of order 8 (non-terminal node - between 3 and 7 values, i.e., between 4 and 8 subtrees):



- B-tree of order  $m$ 
  - removing a value
    - a node can have at most  $m$  subtrees, i.e., a maximum of  $m-1$  values, and at least  $\lceil \frac{m}{2} \rceil$  subtrees, i.e., at least  $\lceil \frac{m}{2} \rceil - 1 = \lceil \frac{m-1}{2} \rceil$  values
    - when eliminating a value from a node, an underflow can occur (the node can end up with less values than the required minimum)
  - eliminate value  $K_0$ 
    1. search for  $K_0$ ; if it doesn't exist, the algorithm ends
    2. if  $K_0$  is found in a non-terminal node (like in the figure on the right),  $K_0$  is replaced with a *neighbor value* from a terminal node (this value can be chosen between 2 values from the trees separated by  $K_0$ )



- B-tree of order m

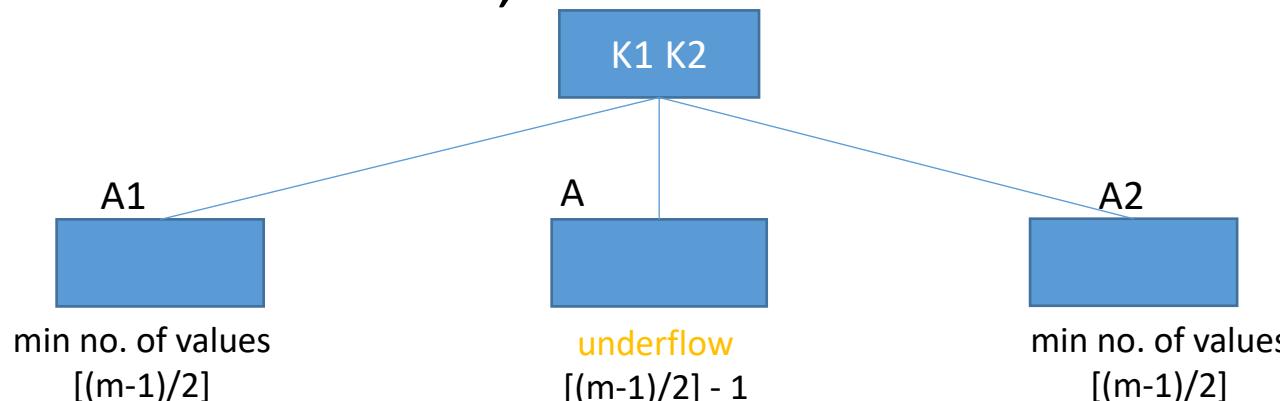
- removing a value

- 3. perform this step until case a / b occurs

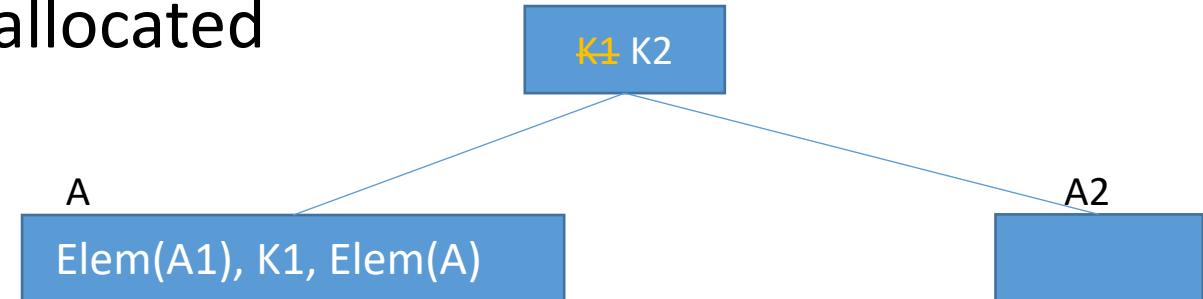
- a. if the current node (from which a value is removed) is the root or underflow doesn't occur, the value is eliminated; the algorithm ends

- b. if the delete operation causes an underflow in the current node (A), but one of the sibling nodes (left / right - B) has at least 1 extra value, values are transferred between A and B via the parent node; the algorithm ends

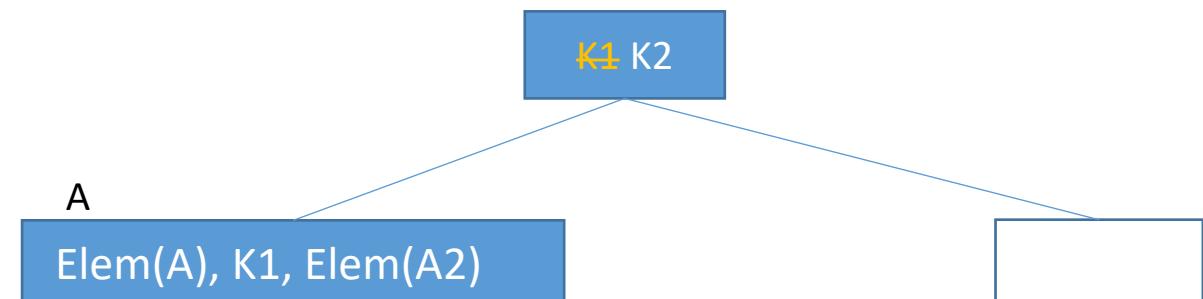
- c. if there is an underflow in A, and sibling nodes A1 and A2 have the minimum number of values, nodes must be concatenated:



- B-tree of order m
  - removing a value
    - if A1 exists, A1 is merged with A and value K1 (separating A1 from A); the node at address A1 is deallocated



- if there is no A1 (A is the first subtree for its parent), A is merged with A2 and K1 (separating A from A2); the node at address A2 is deallocated



- case 3 is then analyzed for the parent node
- if the root is reached and has no values, it is removed and the current node becomes the root

- B-tree of order m
  - obs. a block stores a node from a B-tree
- e.g.:
  - key size: 10b
  - record address / node address: 10b
  - NV value (number of values in the node): 2b
  - block size: 1024b (10b for the header)
  - then:  $2 + (m-1) * (10 + 10) + m * 10 = 1024 - 10 \Rightarrow m = 34$
  - if the size of a block is 2048b and the other values are unchanged, then the order of the tree is  $m = 68$ , i.e., a node can have between 33 and 67 values

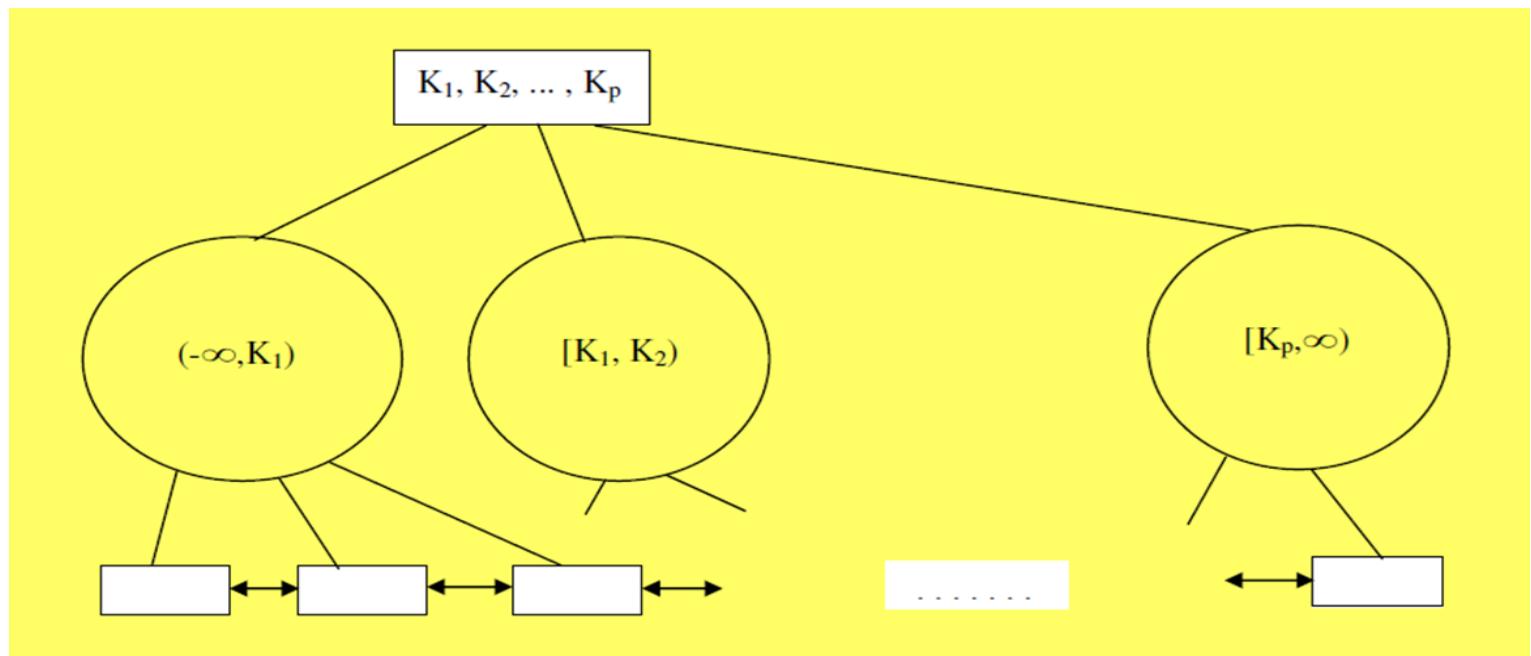
- B-tree of order m
- the maximum number of required blocks (from the file that stores the B-tree) when searching for a value - the maximum number of levels in the tree; for  $m=68$ , if the number of values is 1.000.000, then:
  - the root node (on level 0) contains at least 1 value (2 subtrees)
  - on the next level (level 1) - at least 2 nodes \* 33 values/node = 66 values
  - level 2 – at least  $2 \cdot 34$  nodes \* 33 values/node = 2.244 values
  - level 3 – at least  $2 \cdot 34 \cdot 34$  nodes \* 33 values/node = 76.296 values
  - level 4 – at least  $2 \cdot 34 \cdot 34 \cdot 34$  nodes \* 33 values/node = 2.594.064 values, which is greater than the number of existing values => this level does not appear in the tree

=> at most 4 levels in the tree

- after at most 4 block reads and a number of comparisons in main memory, it can be determined whether the value exists (the corresponding record's address can then be retrieved) or the search was unsuccessful

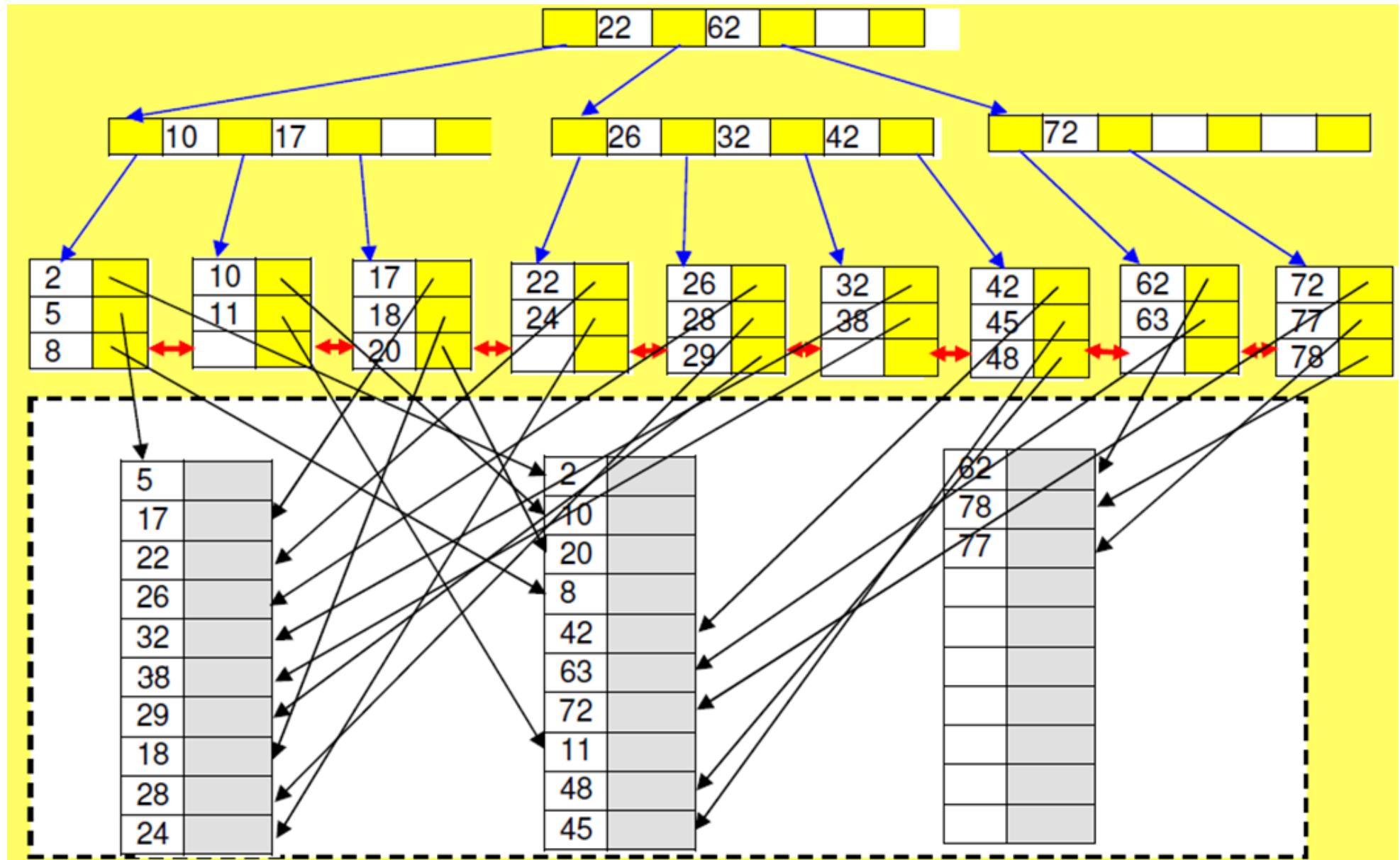
## B+ trees

- B-tree variant
- last level contains all values (key values and the records' addresses)
- some key values can also appear in non-terminal nodes, without the records' addresses; their purpose is to separate values from terminal nodes (guide the search)
- terminal nodes are maintained in a doubly linked list (data can be easily scanned)
- storing a B+ tree
  - B-tree methods
- operations (algorithms)
  - B-tree



# B+ tree

- example



## B+ tree - in practice

- concept of *order* - relaxed, replaced by a physical space criterion (for instance, nodes should be at least half-full)
- terminal / non-terminal nodes - different numbers of entries; usually, inner nodes can store more entries than terminal ones
- variable-length search key => variable-length entries => variable number of entries / page
- if alternative 3 is used ( $\langle k, \text{rid\_list} \rangle$ ) => variable-length entries (in the presence of duplicates), even if attributes are of fixed length

## B+ tree - in practice

- \* prefix key compression
- larger key size => less index entries fit on a page, i.e., less children / index page => larger B+ tree height
- keys in index entries - just direct the search => often, they can be compressed
- adjacent index entries with search key values: *Meteiut*, *Mircqkjt*, *Morqwkj*
- compress key values: *Me*, *Mi*, etc
- what if the subtree also contains *Micfgjh*? => need to store *Mir* (instead of *Mi*)
- it's not enough to analyze neighbor index entries *Meteiut* and *Morqwkj*; the largest key value in *Mircqkjt*'s left subtree and the smallest key value in its right subtree must also be examined
- inserts / deletes - modified correspondingly

## B+ tree - in practice

- values found in practice
  - order – 200
  - fill factor (node) – 67%
  - fan-out – 133
  - capacity
    - height 4:  $133^4 = 312,900,721$
    - height 3:  $133^3 = 2,352,637$
- top levels can often be kept in the BP
  - 1<sup>st</sup> level – 1 page (8KB)
  - 2<sup>nd</sup> level – 133 pages (approx. 1MB)
  - 3<sup>rd</sup> level –  $133^2 = 17689$  pages (approx. 133 MB)

## B+ tree - benefits

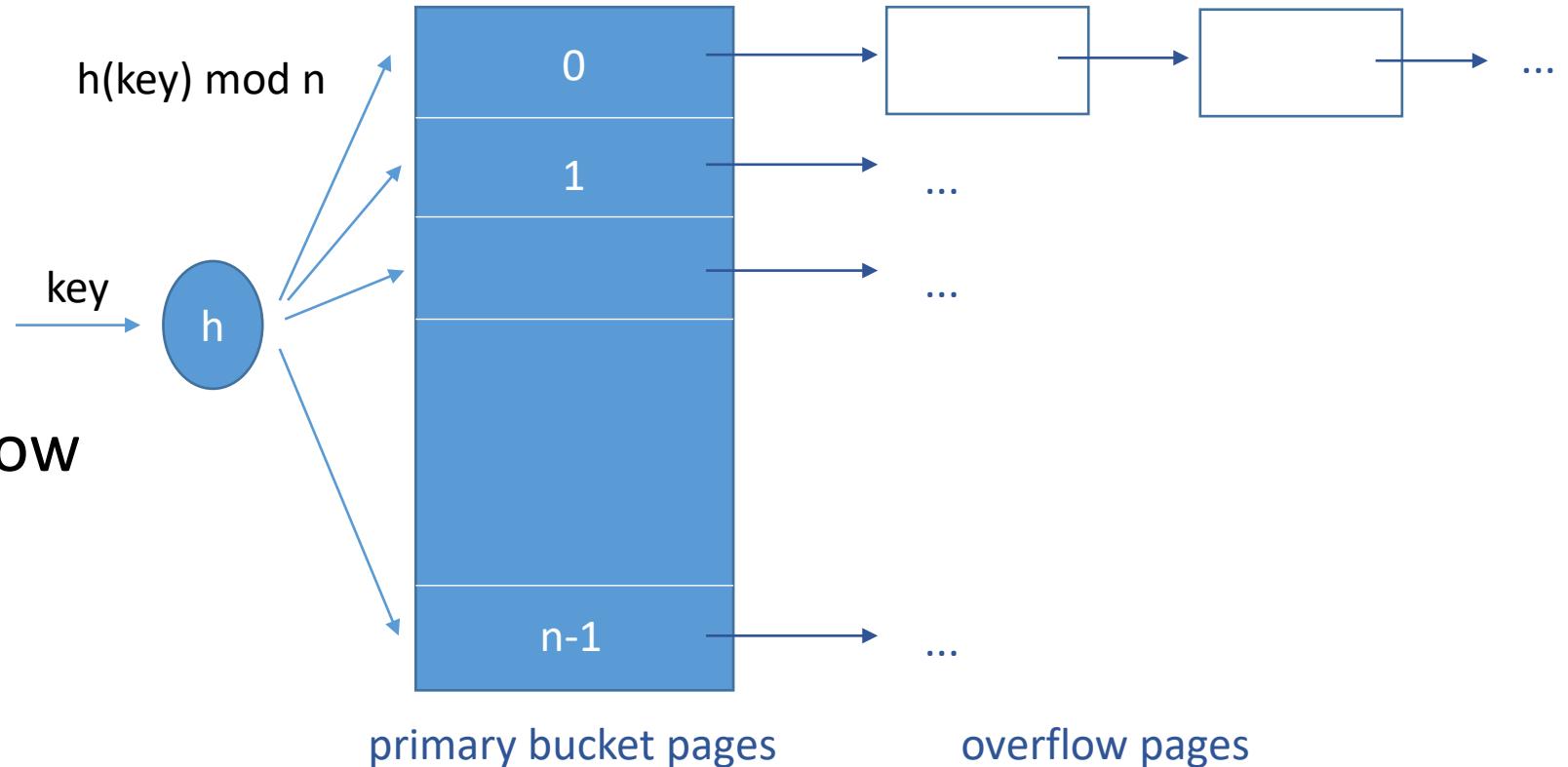
- balanced index => uniform search time
- rarely more than 3-5 levels, the top levels can be kept in main memory => only a few I/O operations are needed to search for a record
- widely used in DBMSs
- ideal for range selections, good for equality selections as well

# Hash-Based Indexing

- hashing function
  - maps search key values into a range of bucket numbers
- hashed file
  - search key (field(s) of the file)
  - records grouped into *buckets*
  - determine record r's bucket
    - apply hash function to search key
  - quick location of records with given search key value
    - example: file hashed on *EmployeeName*
      - Find employee *Popescu*.
- ideal for equality selections

## static hashing

- buckets 0 to  $n-1$
- bucket
  - one primary page
  - possibly extra overflow pages
- data entries in buckets
  - $a_1/a_2/a_3$
- search for a data entry
  - apply hashing function to identify the bucket
  - search the bucket
  - possible optimization
    - entries sorted by search key



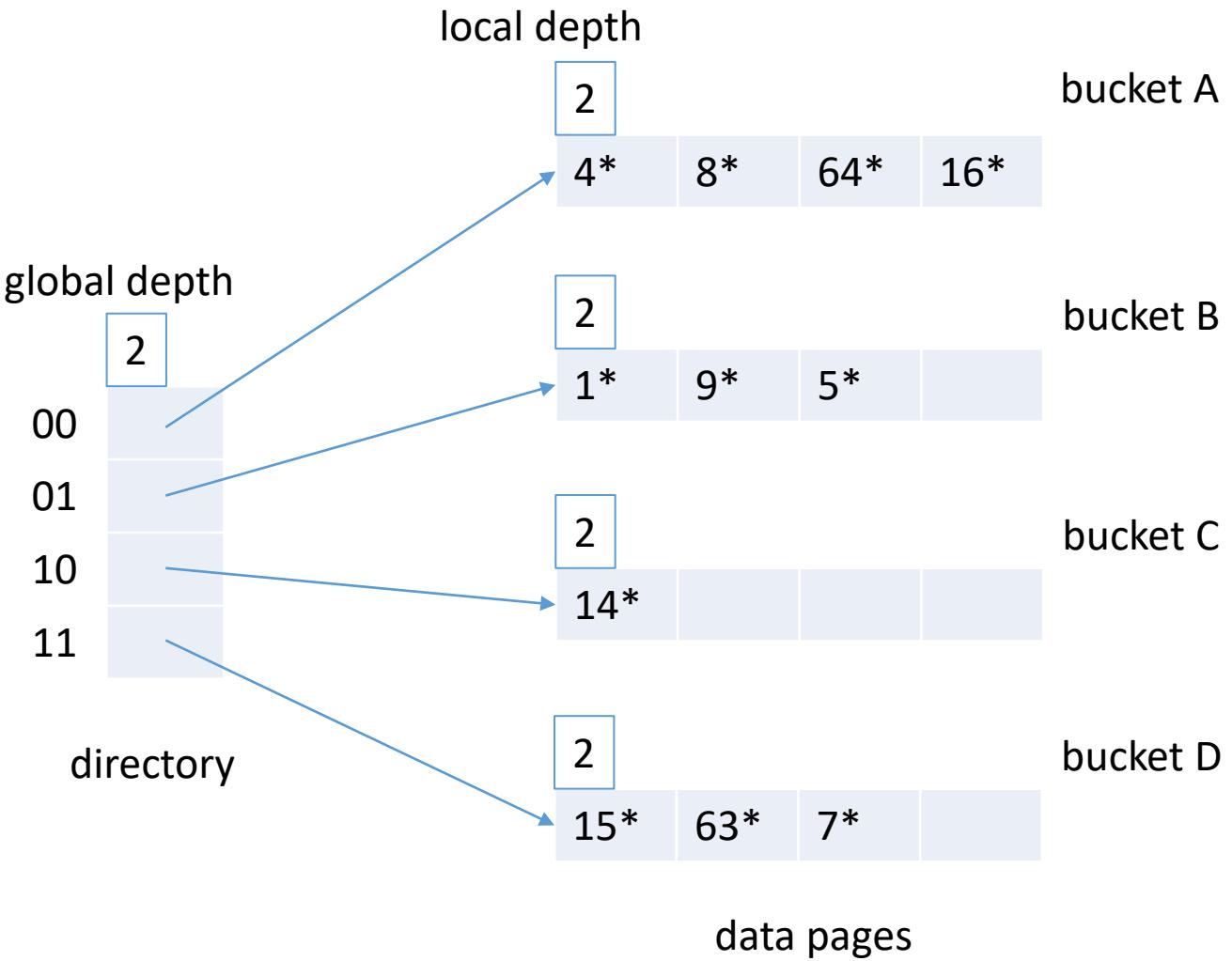
- \* static hashing
- add a data entry
  - apply hashing function to identify the bucket
  - add the entry to the bucket
  - if there is no space in the bucket:
    - allocate an overflow page
    - add the data entry to the page
    - add the overflow page to the bucket's overflow chain
- delete a data entry
  - apply hashing function to identify the bucket
  - search the bucket to locate the data entry
  - remove the entry from the bucket
  - if the data entry is the last one on its overflow page:
    - remove the overflow page from its overflow chain
    - add the page to a free pages list

- \* static hashing
- good hashing function
  - few empty buckets
  - few records in the same bucket
  - i.e., key values are uniformly distributed over the set of buckets
  - good function in practice
    - $h(val) = a*val + b$
    - $h(val) \bmod n$  to identify bucket, for buckets numbered 0..n-1

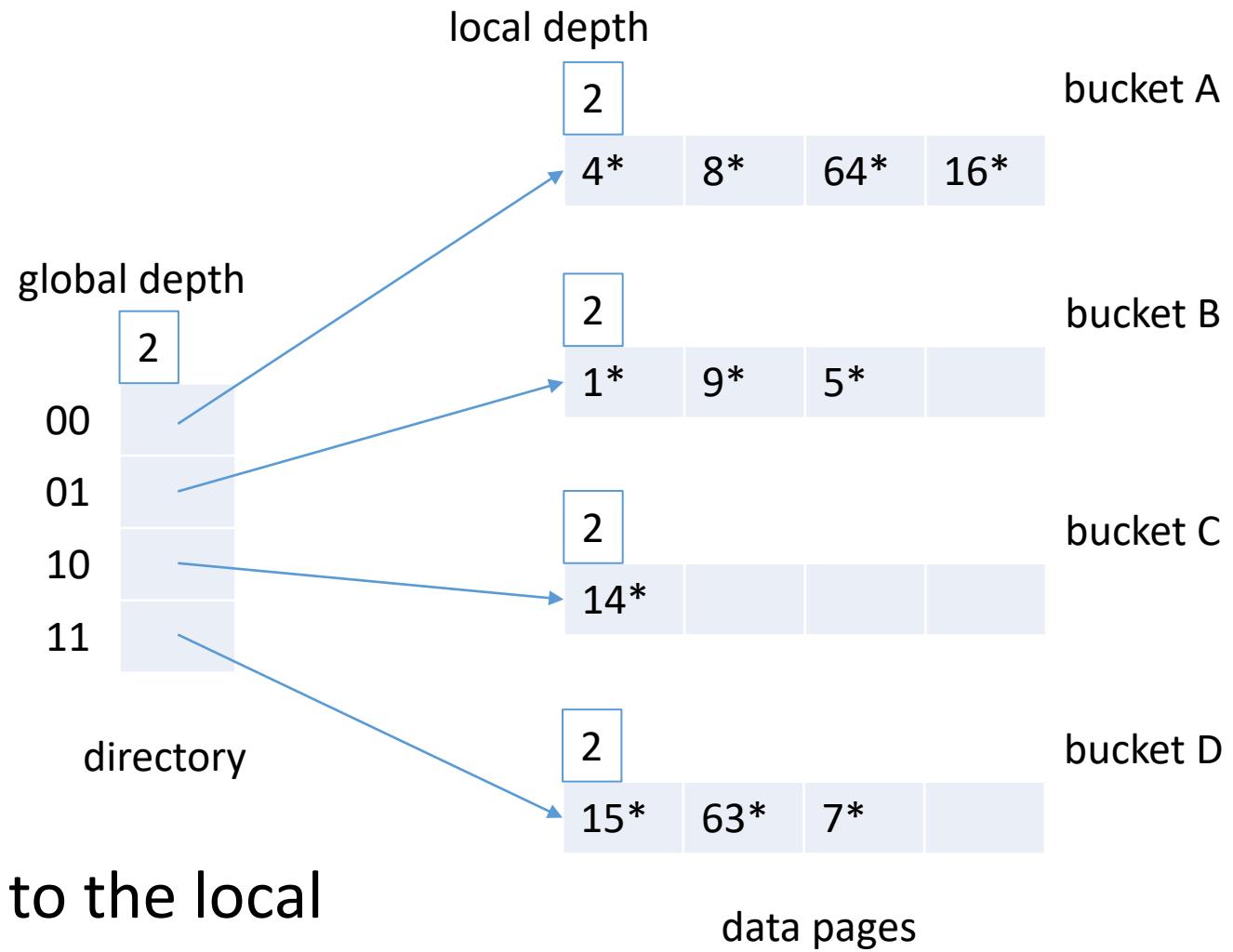
- \* static hashing
- number of buckets known when the file is created
- ideally
  - search: 1 I/O
  - insert / delete: 2 I/Os
- file grows a lot => overflow chains; long chains can significantly affect performance
  - tackle overflow chains
    - initially, pages - 80% full
    - create a new file with more buckets
- file shrinks => wasted space
- main problem: fixed number of buckets
- solutions: periodic rehash, dynamic hashing

## extendible hashing

- dynamic hashing technique
- directory of pointers to buckets
- double the size of the number of buckets
  - double the directory
  - split overflowing bucket
- directory: array of 4 elements
- directory element: pointer to bucket
- entry  $r$  with key value  $K$
- $h(K) = (\dots a_2 a_1 a_0)_2$
- $nr = a_1 a_0$ , i.e., last 2 bits in  $(\dots a_2 a_1 a_0)_2$ ,  $nr$  between 0 and 3
- $\text{directory}[nr]$ : pointer to desired bucket



- \* extendible hashing
- global depth  $gd$  of hashed file
  - number of bits at the end of hashed value interpreted as an offset into the directory
  - kept in the header
  - depends on the size of the directory
    - 4 buckets =>  $gd = 2$
    - 8 buckets =>  $gd = 3$
- initially, the global depth is equal to the local depth of every bucket



- \* extendible hashing

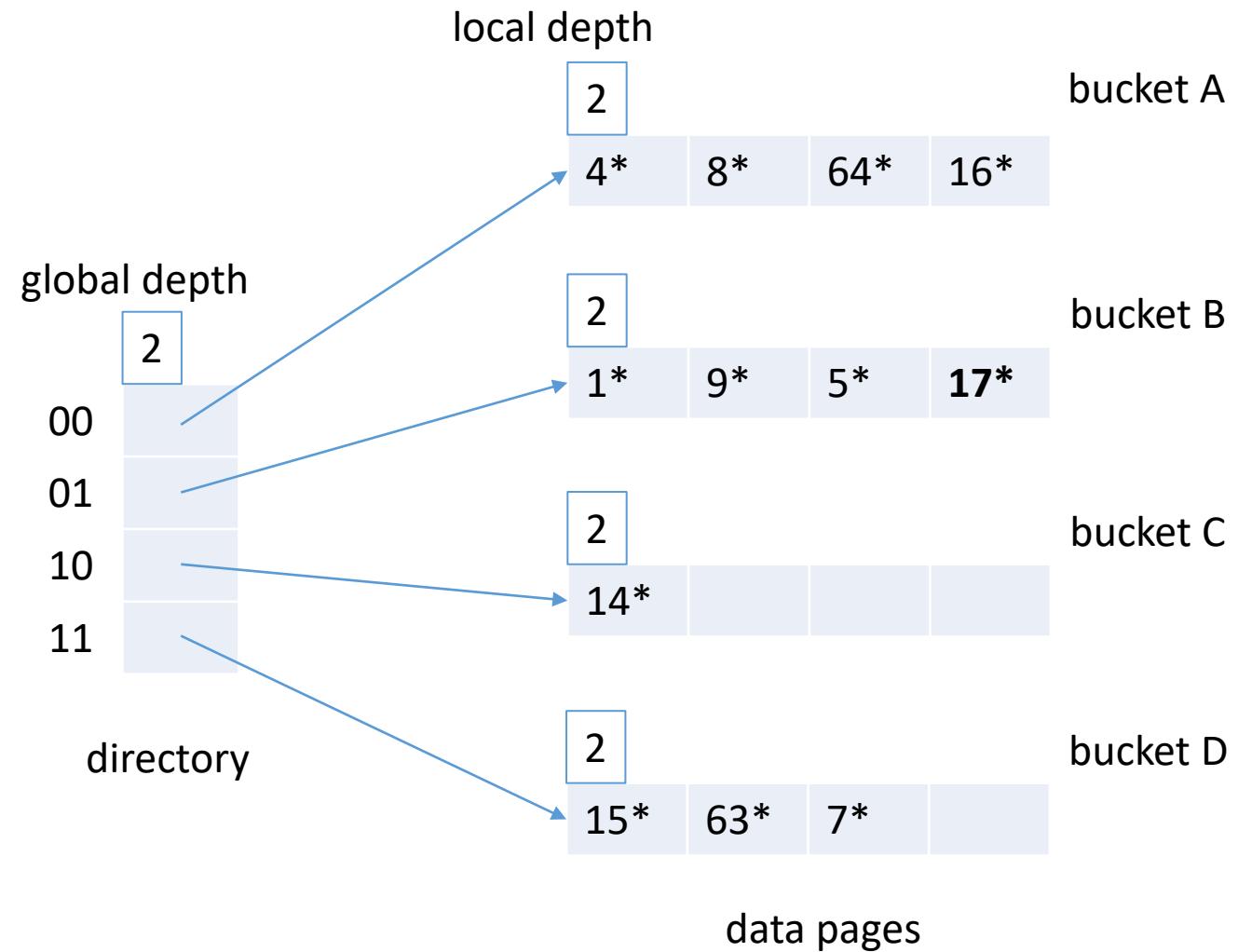
- insert entry

- find bucket

- a. bucket has free space => the new value can be added

- example: add data entry with hash value 17 to bucket B

obs. data entry with hash value 17 is denoted as 17\*



- \* extendible hashing

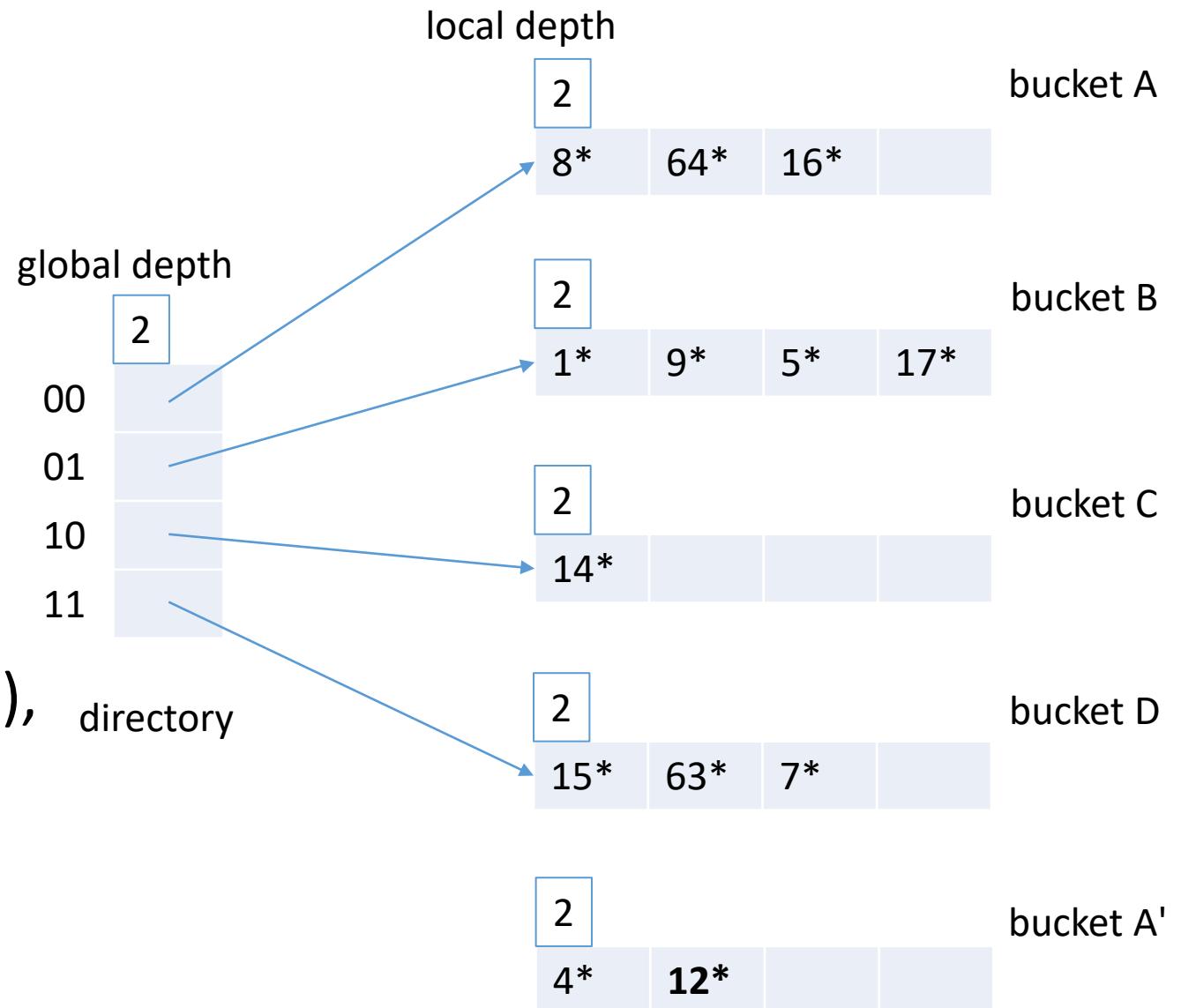
- insert entry

- b. bucket is full

- example: add entry 12\*,  
bucket A full

- split bucket A

- allocate new bucket A'
    - redistribute entries across  
A & A' (the split image of A),  
by taking into account the  
last 3 bits of  $h(K)$

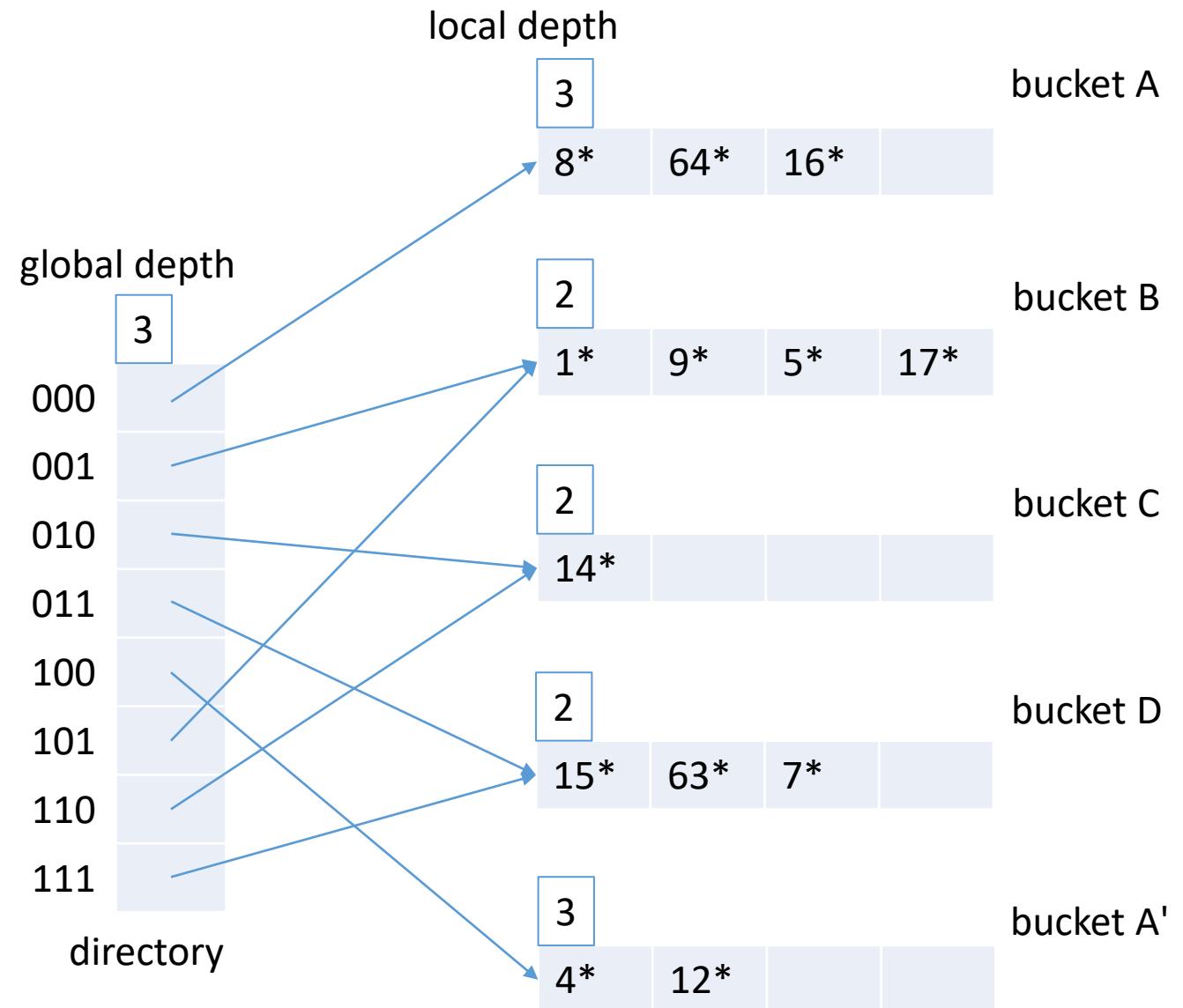


- \* extendible hashing

- insert entry

- b. bucket is full

- if  $gd = \text{local depth of bucket being split} \Rightarrow \text{double the directory, } gd++$
    - 3 bits are needed to discriminate between  $A & A'$ , but the directory has only enough space to store numbers that can be represented on 2 bits, so it is doubled
    - increment local depth of bucket:  $\text{LD}(A) = 3$
    - assign new local depth to bucket's split image:  $\text{LD}(A') = 3$



- \* extendible hashing

- insert entry

- b. bucket is full

- *corresponding elements*

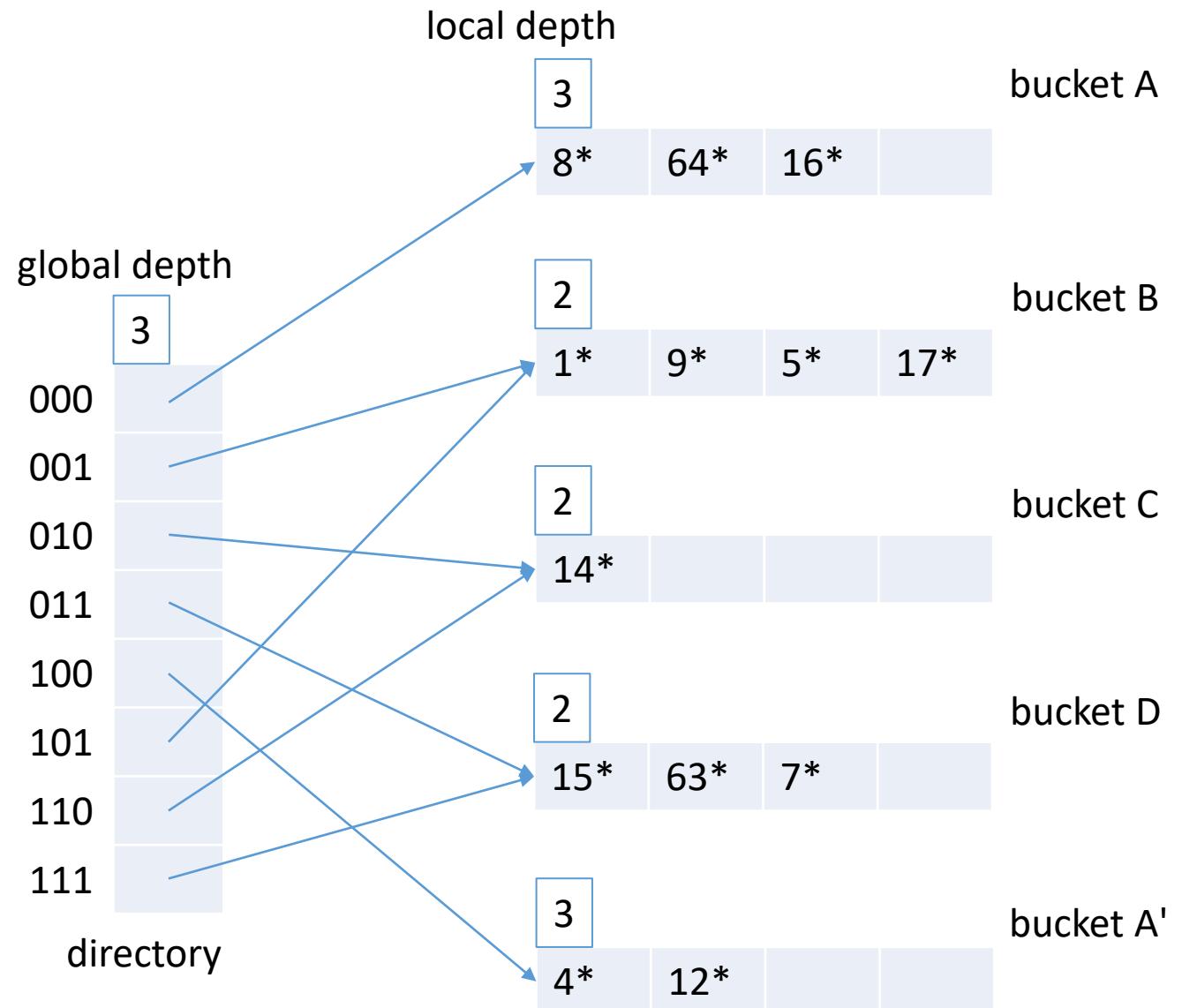
- 000, 100

- 001, 101

- 010, 110

- 011, 111

- point to the same bucket,  
except for 000 and 100,  
which point to A and  
split image A', respectively



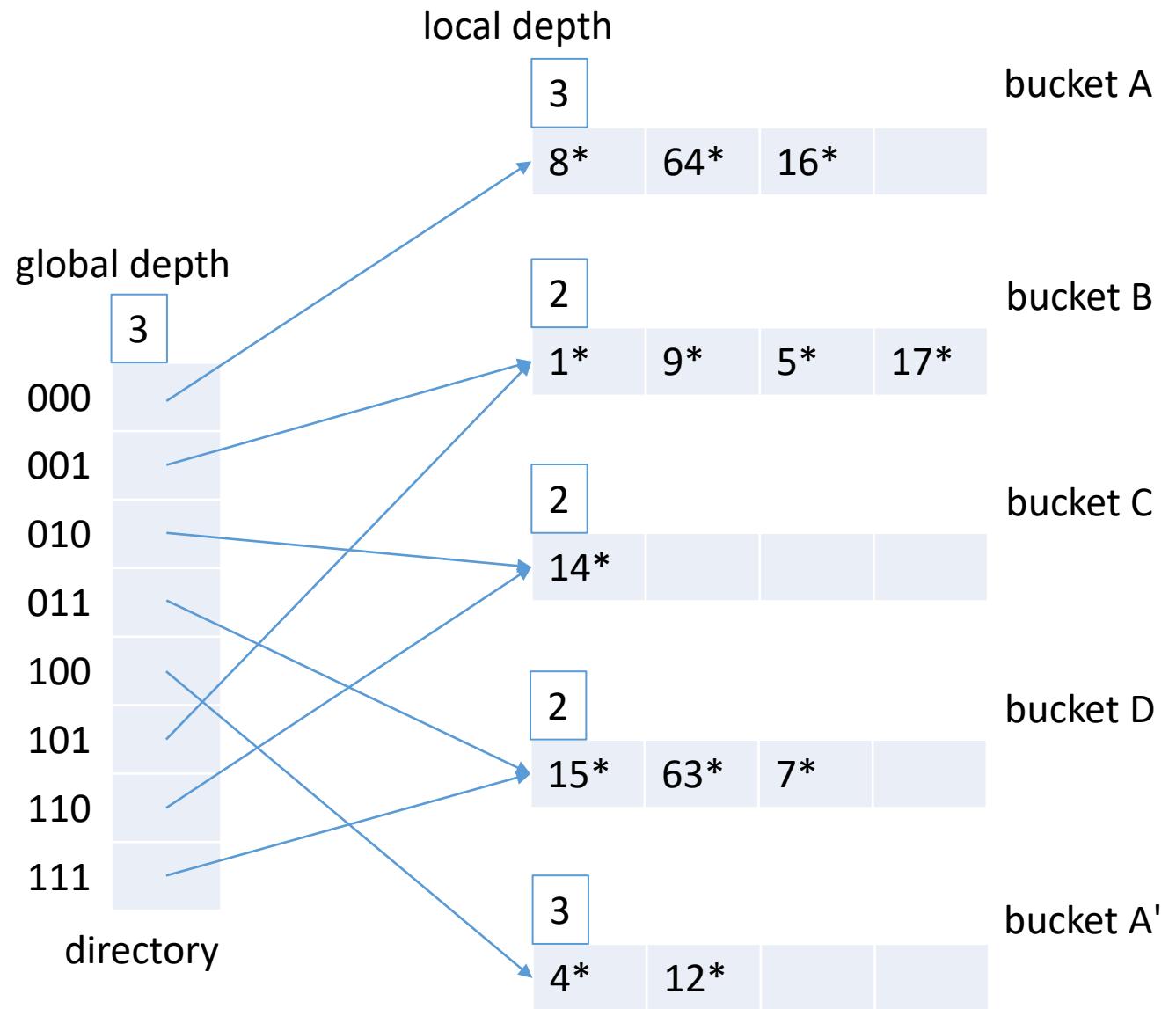
- \* extendible hashing

- insert entry

- b. bucket is full

- example: add 21\*

- it belongs to bucket B, which is already full, but its local depth is 2 and gd = 3



->

- \* extendible hashing

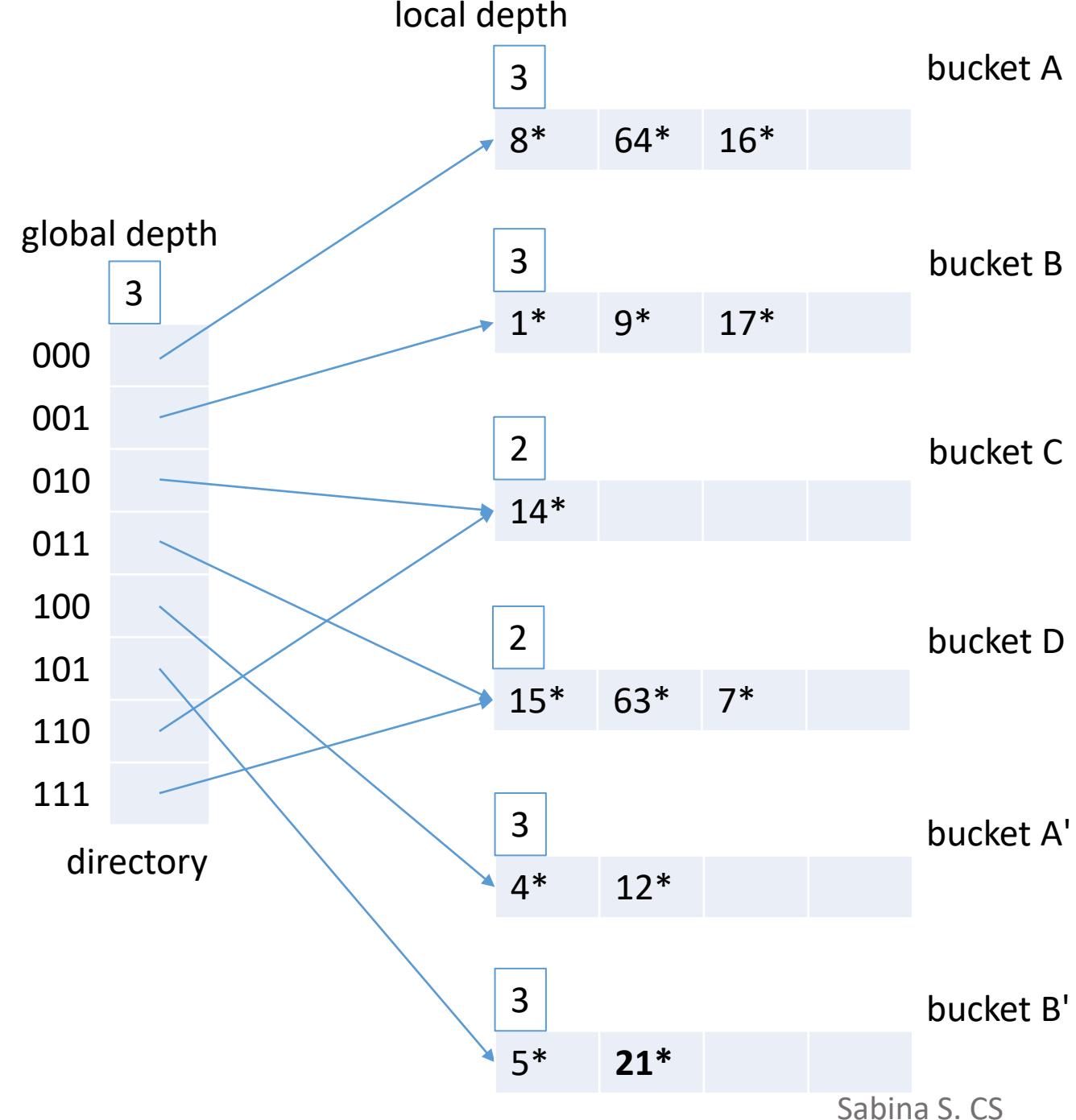
- insert entry

- b. bucket is full

- example: add 21\*

- it belongs to bucket B, which is already full, but its local depth is 2 and gd = 3

=> split B, redistribute entries, increase local depth for B and its split image; directory isn't doubled, gd doesn't change



- \* extendible hashing
- search for entry with key value  $K_0$ 
  - compute  $h(K_0)$
  - take last  $gd$  bits to identify directory element
  - search corresponding bucket
- delete entry
  - locate & remove entry
  - if bucket is empty:
    - merge bucket with its split image, decrement local depth
    - if every directory element points to the same bucket as its split image:
      - halve the directory
      - decrement global depth

## \* extendible hashing

- obs 1.  $2^{gd-ld}$  elements point to a bucket  $B_k$  with local depth  $l_d$ 
  - if  $gd=ld$  and bucket  $B_k$  is split => double directory
- obs 2. manage collisions - overflow pages
- bucket split accompanied by directory doubling
  - allocate new bucket page  $n_{B_k}$
  - write  $n_{B_k}$  and bucket being split
  - double directory array (which should be much smaller than file, since it has 1 page-id / element)
    - if using *least significant bits* (last  $gd$  bits) => efficient operation:
      - copy directory over
      - adjust split buckets' elements

- \* extendible hashing
  - equality selection
  - if directory fits in memory:
    - => 1 I/O (as for Static Hashing with no overflow chains)
  - otherwise
    - 2 I/Os
  - example: 100 MB file, entry = 50 bytes => 2.000.000 entries
  - page size = 8 KB => approx. 160 entries / bucket
- => need  $2.000.000 / 160 = 12.500$  directory elements

## References

- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3<sup>rd</sup> Edition,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Ta13] ȚÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, <http://codex.cs.yale.edu/avi/db-book/>
- [Kn76] KNUTH, D.E., Tratat de programare a calculatoarelor. Sortare și căutare. Ed. Tehnică, București, 1976
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009

# Databases

Lecture 12  
Exam Problems

I Choose the correct answer(s) for the following multiple choice questions. Each question has at least one correct answer.

1. In a SELECT query:

- a. the SELECT clause can contain arithmetic expressions
- b. according to the conceptual evaluation strategy, ORDER BY is evaluated before GROUP BY
- c. HAVING can contain row-level qualification conditions
- d. DISTINCT eliminates duplicates from the answer set
- e. none of the above answers is correct.

2. The natural join operator  $R_1 * R_2$  in the relational algebra:
- a. returns a relation whose schema contains only the attributes in  $R_1$  that don't appear in  $R_2$
  - b. returns a relation whose schema contains all the attributes in  $R_1$  and  $R_2$ , with common attributes appearing only once
  - c. returns 2 relation instances
  - d. is not associative
  - e. none of the above answers is correct.

3. In the ANSI-SPARC architecture (for a database system), a database can have:
- a. exactly one symbolic structure
  - b. several conceptual structures
  - c. several external structures
  - d. several physical structures
  - e. none of the above answers is correct.

4. Consider relation  $S[\underline{A}, \underline{B}, C, D, E, F, G, H]$  with:

- primary key  $\{A, B, C\}$ , no other candidate keys;
  - functional dependencies that are known to hold over  $S$ :  $\{F\} \rightarrow \{H\}$ ,  $\{C\} \rightarrow \{E, G\}$ ;
  - no repeating attributes.
- a.  $S$  is not 1NF
  - b.  $S$  is 2NF
  - c.  $S$  is not BCNF
  - d.  $S$  is 3NF
  - e. none of the above answers is correct.

5. In a B-tree of order 8:

- a. a non-terminal node has at most 8 subtrees
- b. a non-terminal node with 7 values has 8 subtrees
- c. terminal nodes can be on different levels
- d. a non-terminal node with 7 values has 6 subtrees
- e. none of the above answers is correct.

6. Let  $\alpha$ ,  $\beta$  and  $\gamma$  be subsets of attributes in a relational schema. If  $\alpha \rightarrow \beta$  and  $\beta \rightarrow \gamma$ , then by transitivity:

- a.  $\alpha \rightarrow \gamma$
- b.  $\gamma \rightarrow \alpha$
- c.  $\beta \rightarrow \alpha$
- d.  $\gamma \rightarrow \beta$
- e. none of the above answers is correct.

7. Let RepairLog[RID, MechanicID, RollerCoasterID, RepairTime] be a table in a SQL Server database. RepairLog has 100.000 records and 2 indexes: a unique clustered index on RID and a non-clustered index on MechanicID without nonkey columns.

Consider the following query:

```
SELECT RID, MechanicID, RepairTime  
FROM RepairLog  
WHERE MechanicID = 7
```

If the execution plan contains an *Index Seek (NonClustered)*, it also contains a:

- a. *Clustered Index Scan*
- b. *Index Scan (NonClustered)*
- c. *Key Lookup (Clustered)*
- d. *Index Trick (NonClustered)*
- e. none of the above answers is correct.

8-10. Consider the relational schema  $S[\underline{FK1}, \underline{FK2}, A, B, C, D, E]$ , with primary key  $\{\underline{FK1}, \underline{FK2}\}$ . Answer questions 8-10 using the legal instance below:

FK1	FK2	A	B	C	D	E
1	1	a1	b1	c1	7	2
1	2	a_	b3	c1	5	2
1	3	a2	b1	c2	Null	2
2	1	a3	b3	c2	Null	100
2	2	a3	b3	c3	Null	100

8. Consider queries  $Q_1$  and  $Q_2$ :

$Q_1$ :

SELECT \*

FROM S s1 LEFT JOIN S s2 ON s1.FK1 = s2.E

$Q_2$ :

SELECT DISTINCT \*

FROM S s1 INNER JOIN S s2 ON s1.FK1 = s2.E

The cardinality of the answer set of  $Q_i$  is denoted by  $|Q_i|$ .

$|Q_1| - |Q_2|$  is:

a. 0

b. 3

c. 10

d. 2

e. none of the above answers is correct.

FK1	FK2	A	B	C	D	E
1	1	a1	b1	c1	7	2
1	2	a_	b3	c1	5	2
1	3	a2	b1	c2	Null	2
2	1	a3	b3	c2	Null	100
2	2	a3	b3	c3	Null	100

$S[\underline{FK1}, \underline{FK2}, A, B, C, D, E]$

9. Regarding the functional dependencies of S:
- at least one of the following dependencies is not satisfied by the instance:  $\{A\} \rightarrow \{B\}$ ,  $\{FK1, FK2\} \rightarrow \{A, B\}$ ,  $\{FK1\} \rightarrow \{A\}$
  - by examining the instance, we can conclude that at least one of the following dependencies is specified on the schema S:  $\{A\} \rightarrow \{B\}$ ,  $\{FK1\} \rightarrow \{A, B\}$ ,  $\{FK1\} \rightarrow \{A\}$
  - at least two of the following dependencies are not satisfied by the instance:  $\{FK2\} \rightarrow \{A, B\}$ ,  $\{A\} \rightarrow \{E\}$ ,  $\{A, B\} \rightarrow \{E\}$ ,  $\{FK1, FK2\} \rightarrow \{E\}$
  - by examining the instance, we can conclude that at least two of the following dependencies are specified on the schema S:  $\{FK2\} \rightarrow \{A, B\}$ ,  $\{A\} \rightarrow \{E\}$ ,  $\{A, B\} \rightarrow \{E\}$ ,  $\{B\} \rightarrow \{C, E\}$
  - none of the above answers is correct.

FK1	FK2	A	B	C	D	E
1	1	a1	b1	c1	7	2
1	2	a_	b3	c1	5	2
1	3	a2	b1	c2	Null	2
2	1	a3	b3	c2	Null	100
2	2	a3	b3	c3	Null	100

S[FK1, FK2, A, B, C, D, E]

10. Consider queries  $Q_1$  and  $Q_2$ :

$Q_1$ :

```
SELECT FK2, FK1, COUNT(DISTINCT B)
```

```
FROM S
```

```
GROUP BY FK2, FK1
```

```
HAVING FK1 = 0
```

$Q_2$ :

```
SELECT FK2, FK1, COUNT(C)
```

```
FROM S
```

```
GROUP BY FK2, FK1
```

```
HAVING MAX(E) < 0
```

The cardinality of the answer set of  $Q_i$  is denoted by  $|Q_i|$ .

FK1	FK2	A	B	C	D	E
1	1	a1	b1	c1	7	2
1	2	a_	b3	c1	5	2
1	3	a2	b1	c2	Null	2
2	1	a3	b3	c2	Null	100
2	2	a3	b3	c3	Null	100

$S[\underline{FK1}, \underline{FK2}, A, B, C, D, E]$

->

10.

$|Q_1| - |Q_2|$  is:

- a. 0
- b. 2
- c. 1
- d. -1
- e. none of the above answers is correct.

FK1	FK2	A	B	C	D	E
1	1	a1	b1	c1	7	2
1	2	a_	b3	c1	5	2
1	3	a2	b1	c2	Null	2
2	1	a3	b3	c2	Null	100
2	2	a3	b3	c3	Null	100

S[FK1, FK2, A, B, C, D, E]

11. A unique index:

- a. can contain duplicates
- b. cannot contain duplicates
- c. can be non-clustered
- d. cannot be non-clustered
- e. none of the above answers is correct.

12. For the relation  $R[A, B, C]$  below, consider the 3 possible projections on 2 attributes:  $AB[A, B]$ ,  $BC[B, C]$ ,  $AC[A, C]$ . How many extra records does  $AB * BC * AC$  contain (i.e., records that don't appear in  $R$ )?

A	B	C
a1	b2	c1
a1	b1	c2
a2	b1	c1

- a. 0
- b. 1
- c. 2
- d. 3
- e. none of the above answers is correct.

13. The cross-product operator  $R_1 \times R_2$  in the relational algebra:
- a. returns a relation whose schema contains all the attributes in  $R_1$  followed by all the attributes in  $R_2$
  - b. returns a relation whose schema contains only the attributes in  $R_1$
  - c. returns 3 relation instances
  - d. is associative
  - e. none of the above answers is correct.

## II Answer the following questions / solve the following problems.

1. Rewrite the CREATE TABLE statements below such that the following restriction is enforced: one T1 entity can be associated with any number of T2 entities, and one T2 entity can be associated with at most one T1 entity. Don't add other SQL statements.

```
CREATE TABLE T1  
(IDT1 INT PRIMARY KEY,  
C1 VARCHAR(100))
```

```
CREATE TABLE T2  
(IDT2 INT PRIMARY KEY,  
C2 DATE)
```

2. Write the relational algebra expression below as a SQL query.

$$\pi_{\{A,B,C\}}((\pi_{\{A,B, ID\}}(\sigma_{M=70}(R))) \otimes_{ID=IDT1} (\pi_{\{C, IDT1\}}(\sigma_{N>5}(S))))$$

3-6. Consider the relational schema  $R[\underline{\text{RID}}, \text{A}, \text{B}, \text{C}, \text{D}, \text{E}, \text{F}]$ , with primary key  $\{\text{RID}\}$ . Answer questions 3-6 using the legal instance below:

<b>RID</b>	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>
1	100	200	5	200	20	11
2	101	50	11	200	5	12
3	100	100	7	200	5	13
4	200	200	6	200	20	14
5	200	100	2	200	5	9
6	300	50	11	200	5	10

3. What's the result set returned by the following query? Write the tuples' values and the names of the columns.

```
SELECT r1.RID, r1.A + r2.A C2, r1.A * r2.A C3  
FROM R r1 LEFT JOIN R r2 ON r1.RID = r2.RID  
WHERE r1.A > ANY (SELECT B  
                  FROM R  
                  WHERE C < 10)
```

RID	A	B	C	D	E	F
1	100	200	5	200	20	11
2	101	50	11	200	5	12
3	100	100	7	200	5	13
4	200	200	6	200	20	14
5	200	100	2	200	5	9
6	300	50	11	200	5	10

R[RID, A, B, C, D, E, F]

4. Evaluate the expressions below.  $\pi$  doesn't eliminate duplicates. What's the cardinality of  $T$ ?

$$S := \sigma_{F < 13}(R)$$

$$T := \pi_{\{S.RID, S.A\}}(S \otimes_{S.D=R.D} R)$$

RID	A	B	C	D	E	F
1	100	200	5	200	20	11
2	101	50	11	200	5	12
3	100	100	7	200	5	13
4	200	200	6	200	20	14
5	200	100	2	200	5	9
6	300	50	11	200	5	10

R[RID, A, B, C, D, E, F]

5. What's the result set returned by the following query? Write the tuples' values and the names of the columns.

```
SELECT R.*  
FROM  
(SELECT r1.RID, r2.A, r3.B  
FROM R r1 INNER JOIN R r2 ON r1.A = r2.B  
INNER JOIN R r3 ON r2.B > r3.D  
WHERE r1.F > 10) t  
RIGHT JOIN R ON t.RID = R.RID
```

RID	A	B	C	D	E	F
1	100	200	5	200	20	11
2	101	50	11	200	5	12
3	100	100	7	200	5	13
4	200	200	6	200	20	14
5	200	100	2	200	5	9
6	300	50	11	200	5	10

R[RID, A, B, C, D, E, F]

6. Write all functional dependencies  $F$  such that:

- $F$  is satisfied by the current instance of R;
- the dependent of  $F$  is  $\{D\}$ ;
- the determinant of  $F$  has a single column.

RID	A	B	C	D	E	F
1	100	200	5	200	20	11
2	101	50	11	200	5	12
3	100	100	7	200	5	13
4	200	200	6	200	20	14
5	200	100	2	200	5	9
6	300	50	11	200	5	10

R[RID, A, B, C, D, E, F]

7. Rewrite the expression below using only operators in the set  $\{\sigma, \pi, \times, \cup, -\}$ .

$$(R \otimes_{R.ID=S.RID} S) \cap (T \otimes_{T.ID=U.TID} U)$$

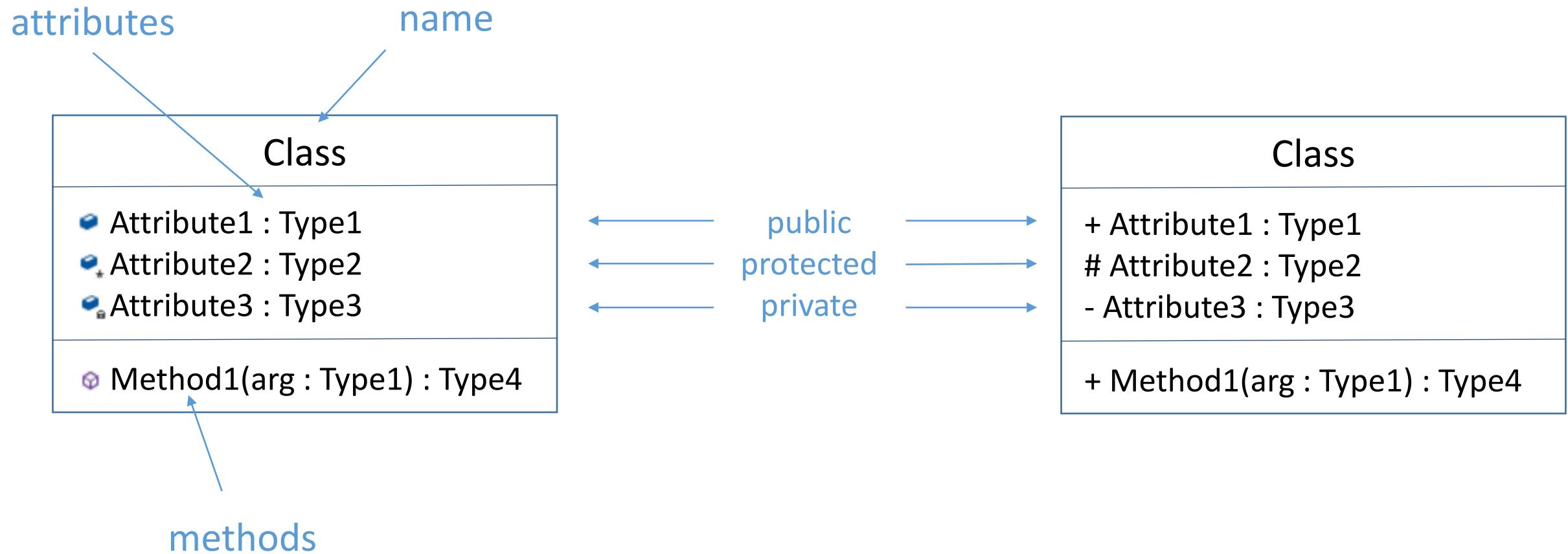
# Databases

Lecture 13  
Conceptual Modeling

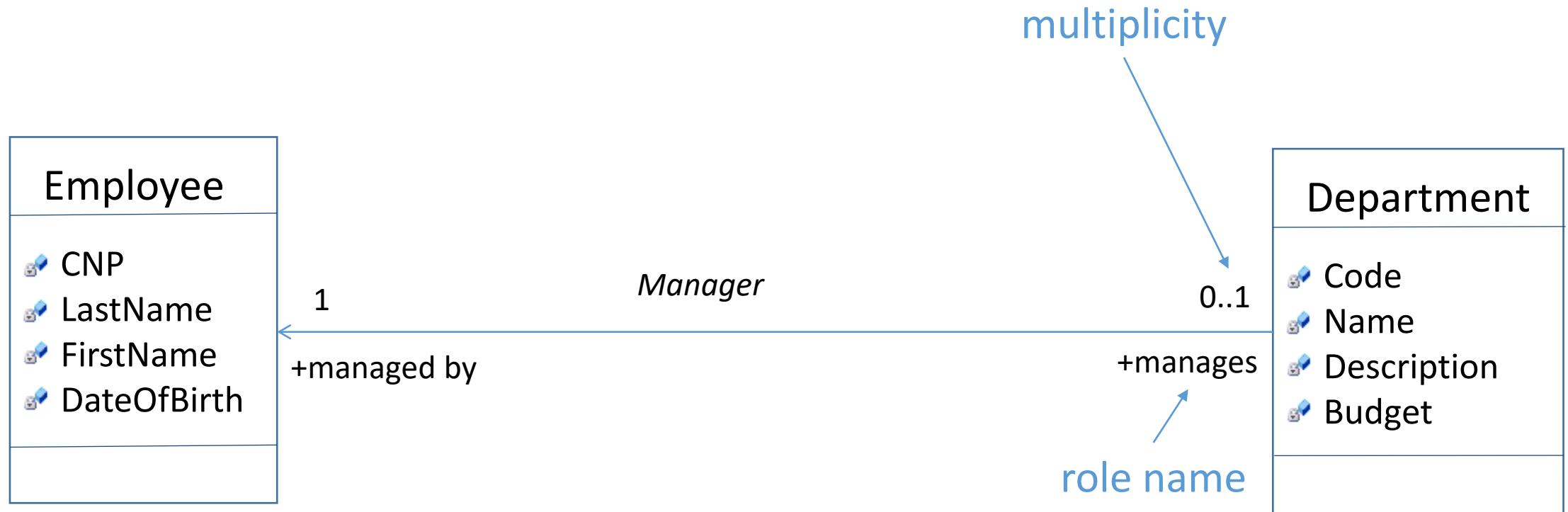
- database design - stages
  - requirements analysis
    - what data will the DB store?
    - what are the main operations to be supported?
    - what apps will be powered by the DB?
  - conceptual DB design
    - high level description of data and integrity constraints
  - logical DB design
    - translate the conceptual DB design to a DB schema in terms of the model supported by the DBMS (e.g., relational)
  - schema refinement
    - normalization
    - eliminate redundancy and associated problems

- database design - stages
  - physical DB design
    - create indexes
    - redesign parts of the schema

- UML class diagram
  - classes

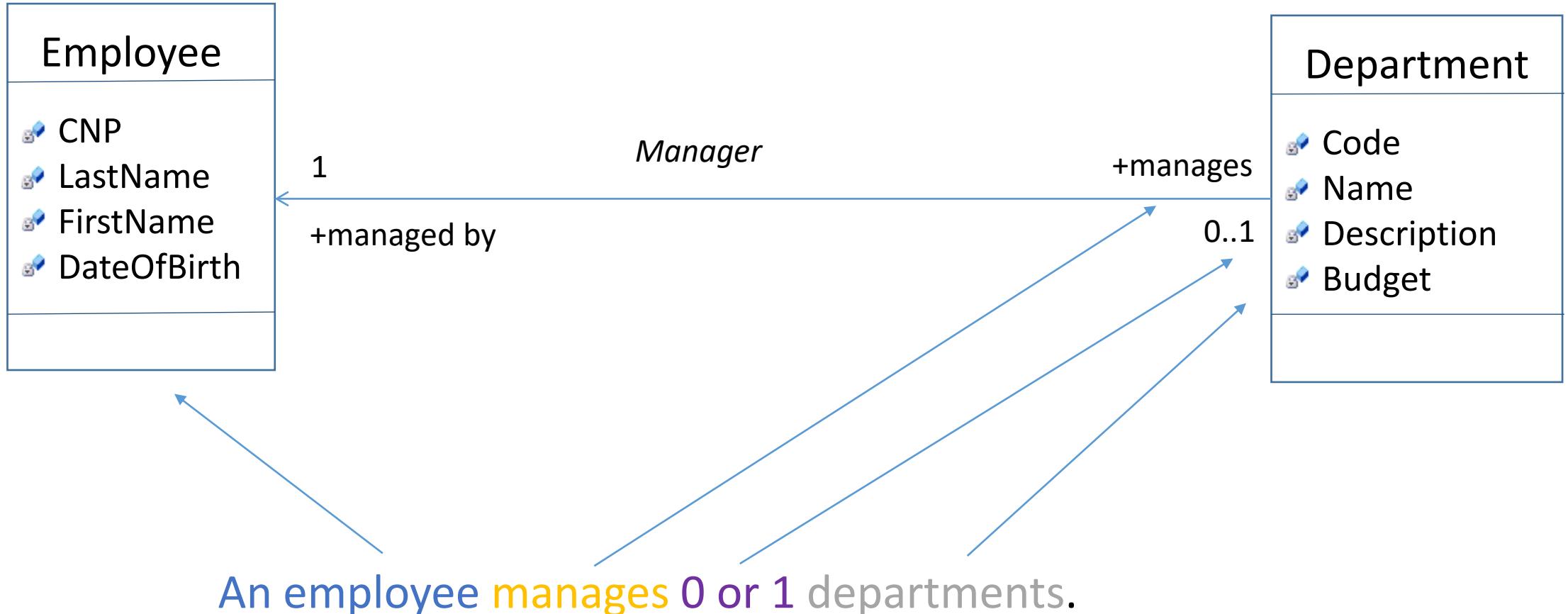


- UML class diagram
  - associations

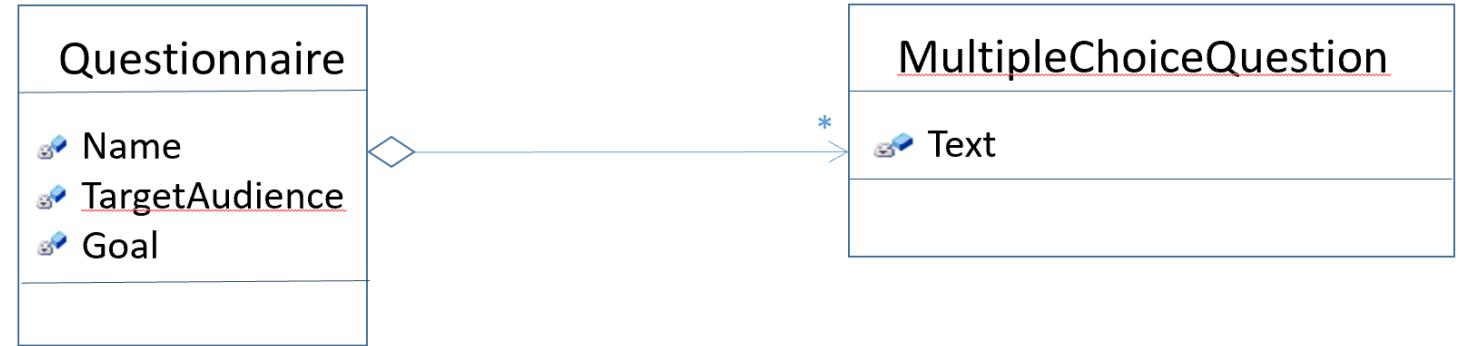


- navigability – unidirectional, bidirectional
- multiplicity – examples
  - **0..1**
  - **7..10**
  - **5**
  - **0..\***

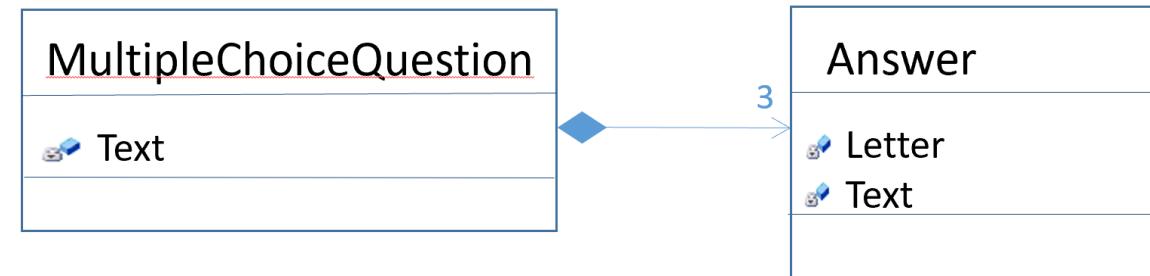
- UML class diagram
  - associations



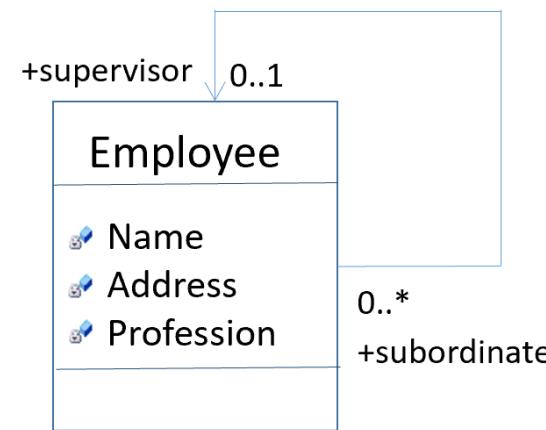
- UML class diagram
  - aggregation



- composition

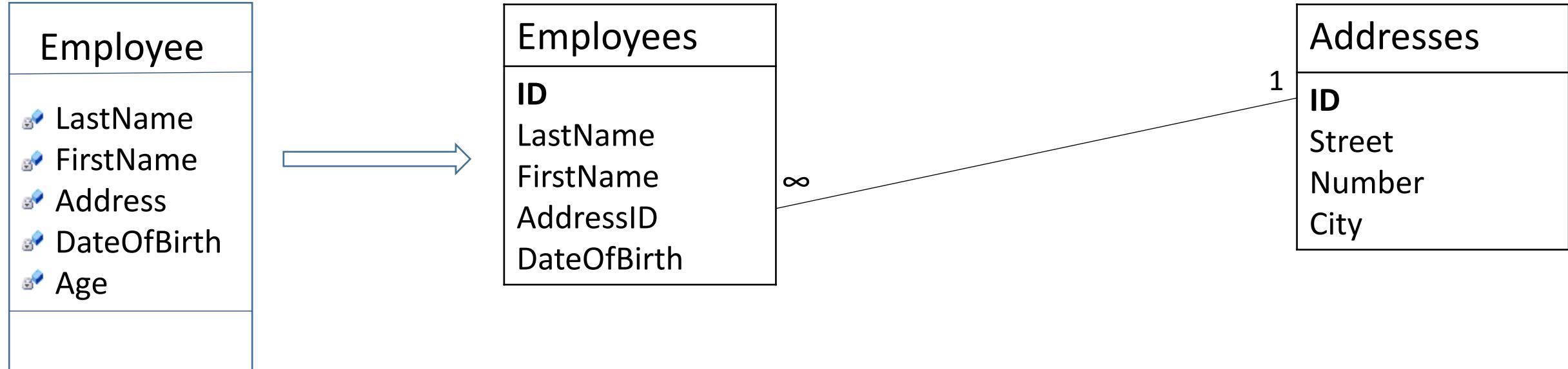


- reflexive association



- conceptual model => relational database
- 1:1 mapping, i.e., classes become tables
- drawbacks
  - one could create too many tables
    - too many tables => too many join operations
  - necessary tables could be omitted; m:n associations require a third table (join table)
  - inheritance is not properly handled

- class -> table



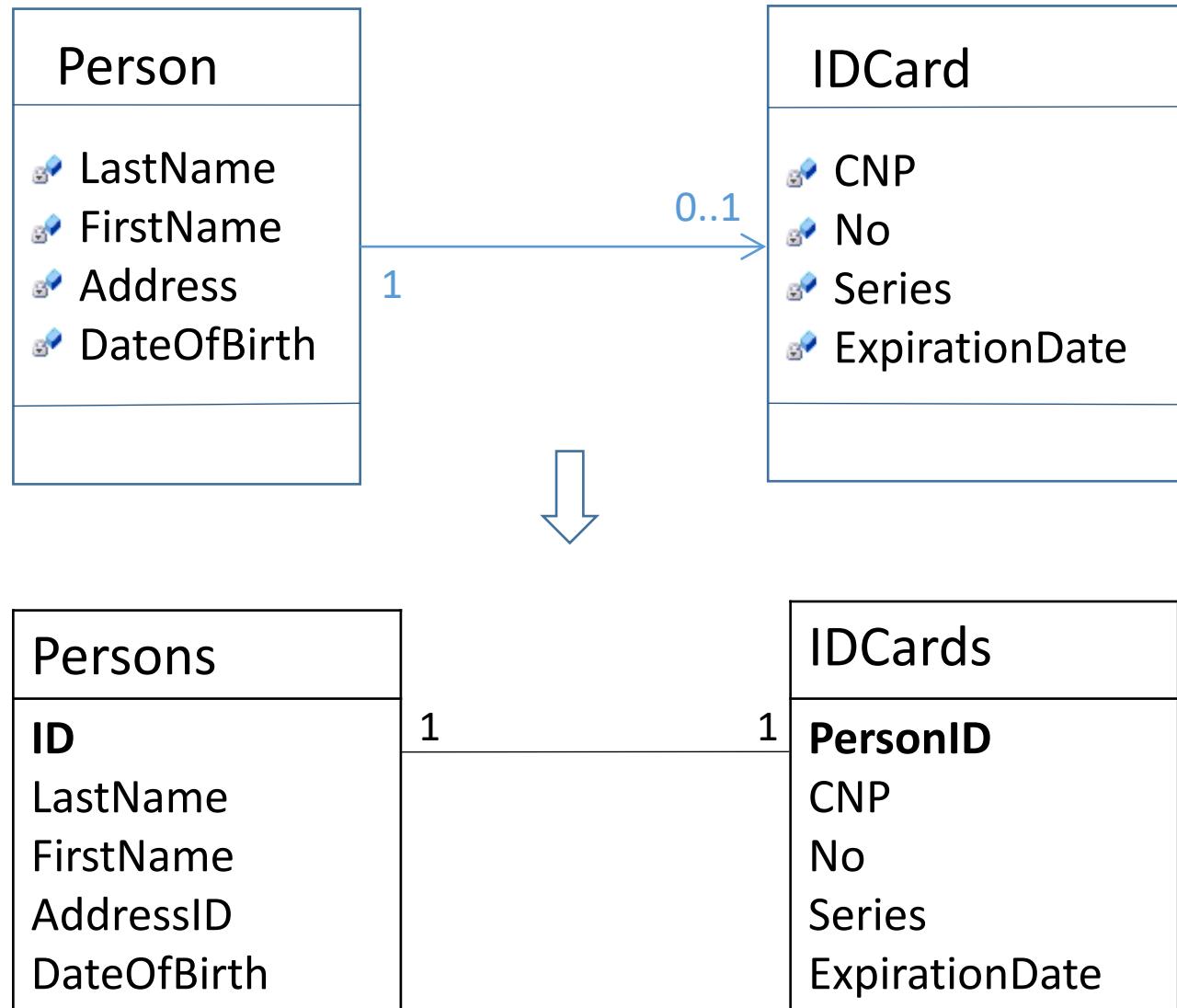
- the plural of the class name becomes the name of the table
- simple class attributes become table fields
- composite attributes become tables
- derived attributes are not mapped to table fields
- surrogate keys are added

- class -> table
  - surrogate key
    - key that isn't obtained from the domain of the modeled problem
  - when possible, use integer keys that are automatically generated by the DBMS
    - easy to maintain - the responsibility of the system
    - efficient approach (fast queries)
    - simplified definition of foreign keys
  - possible approach
    - surrogate key name: *ID*
    - foreign key name: <*SingularTableName*>*ID*

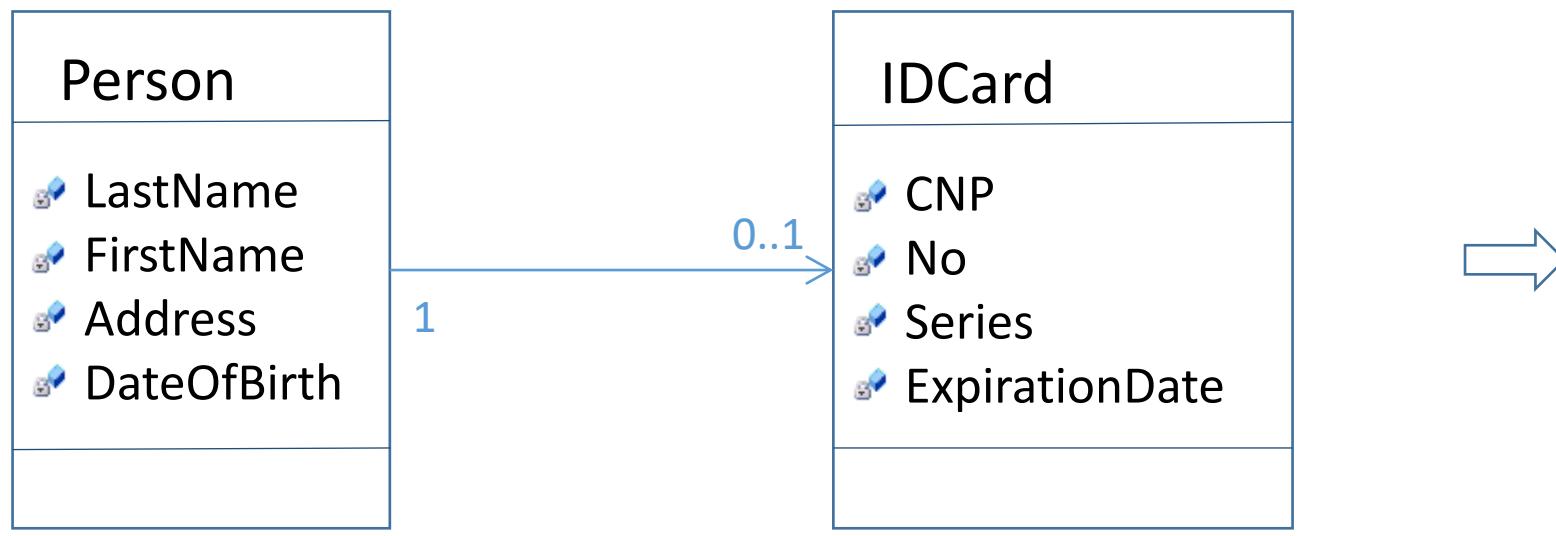
- mapping simple associations
- 1 : 0..1
  - create 1 table per class
  - the key of the 1 table (i.e., table at the 1 end of the association) becomes a foreign key in the 2<sup>nd</sup> table
  - usually, only one key is automatically generated (the one corresponding to the 1 table)

->

- mapping simple associations
- 1 : 0..1

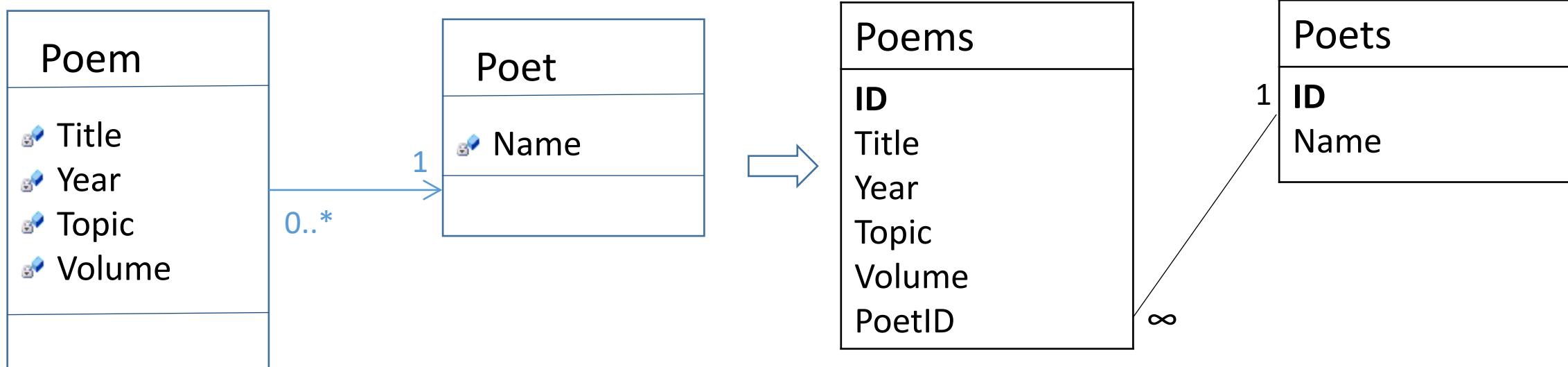


- mapping simple associations
- 1 : 1
  - create 1 table containing the attributes of both classes
  - this approach can also be used for 1 : 0..1 associations (when only a few objects in the 1<sup>st</sup> class are not associated with objects in the 2<sup>nd</sup> class)



Persons	
ID	
LastName	
FirstName	
AddressID	
DateOfBirth	
IDCardCNP	
IDCardNo	
IDCardSeries	
IDCardExpirationDate	

- mapping simple associations
- one-to-many
  - create 1 table / class
  - the key of the 1 table becomes a foreign key in the 2<sup>nd</sup> table

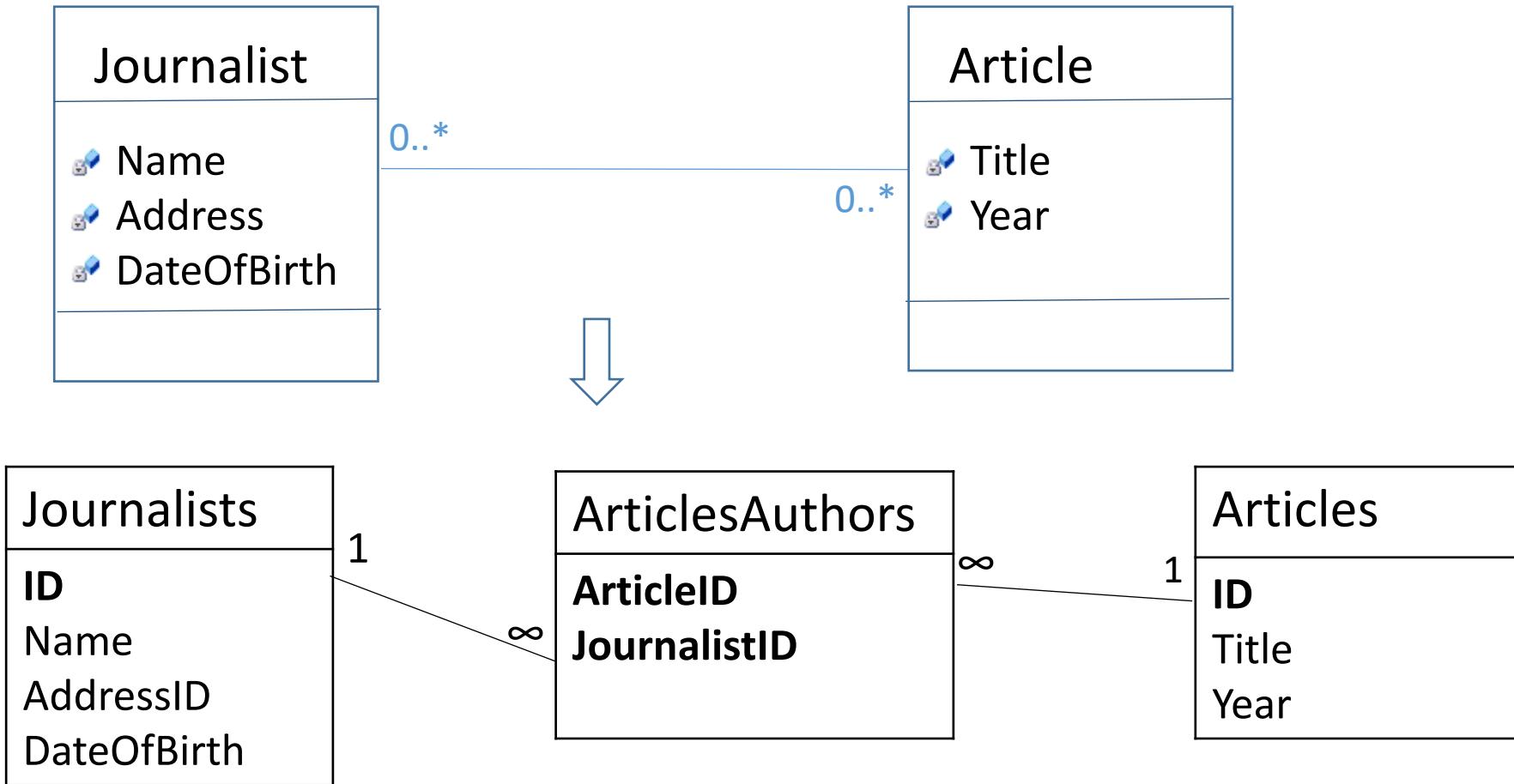


- mapping simple associations
- many-to-many
  - create one table / class
  - create an additional table, i.e., the *join table*
  - the primary keys of the 2 initial tables become foreign keys in the join table
  - the primary key of the join table:
    - composite, containing the 2 foreign keys
    - surrogate key
  - the name of the join table is usually a combination of the names of the 2 initial tables (not mandatory)
  - if an association class exists, its attributes become fields in the join table

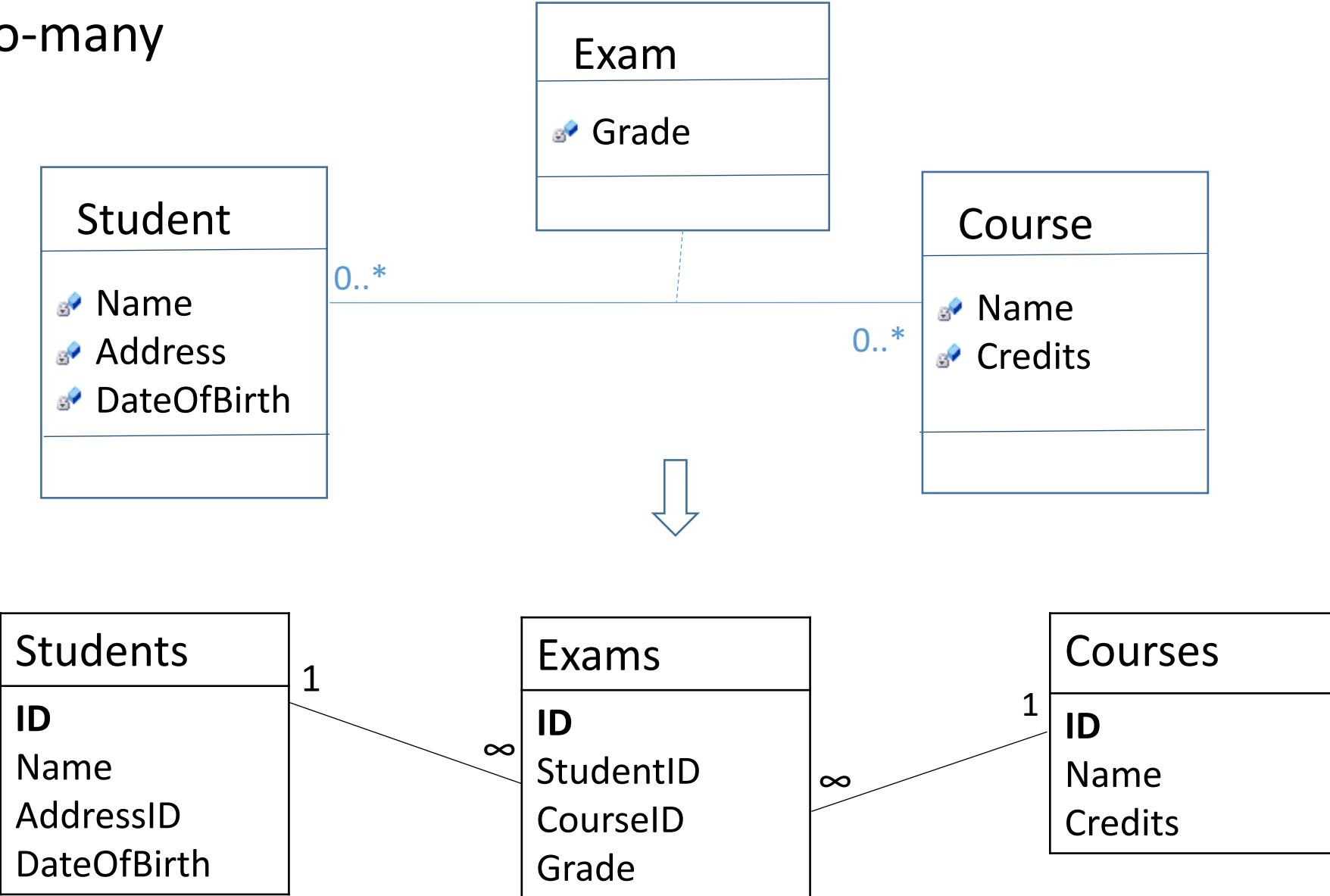
->

Sabina S. CS

- mapping simple associations
- many-to-many



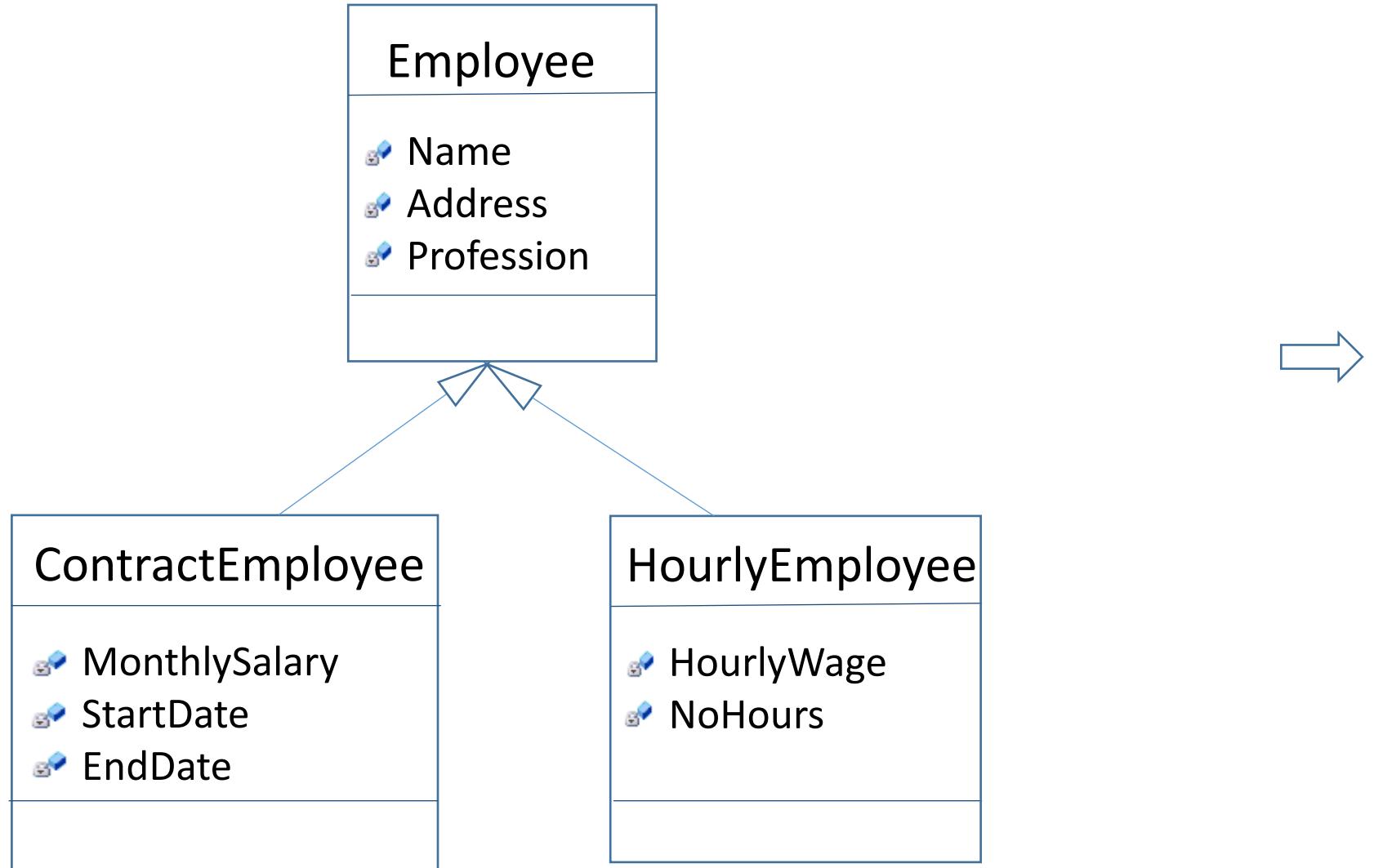
- mapping simple associations
- many-to-many



- mapping inheritance
- a1
  - create one table / class
  - create one view / superclass-subclass pair
  - it generates the largest number of objects (tables, views)
  - flexibility - no impact on existing tables / views when adding other subclasses
  - possible performance problems – every access requires a join through the view
  - can be used when the number of records is relatively small (so performance is not a concern)

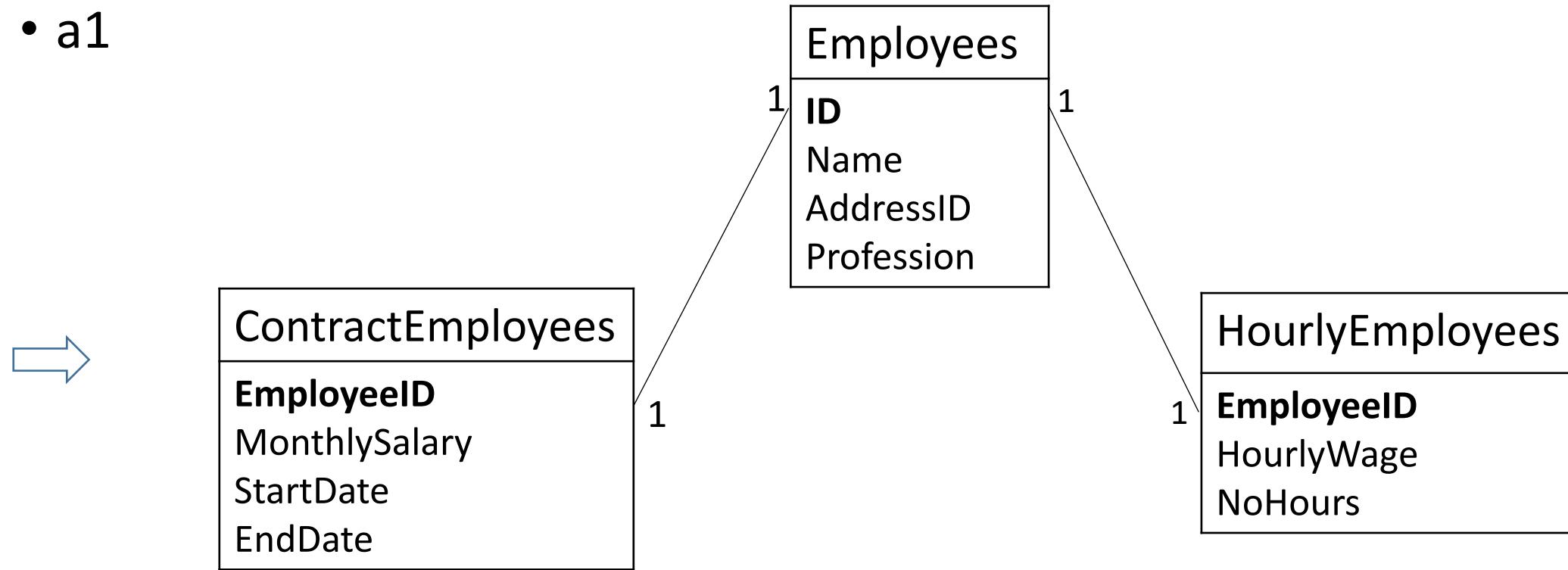
->

- mapping inheritance
- a1



- mapping inheritance

- a1



```
CREATE VIEW ContractEmployeesComplete(....)
```

AS

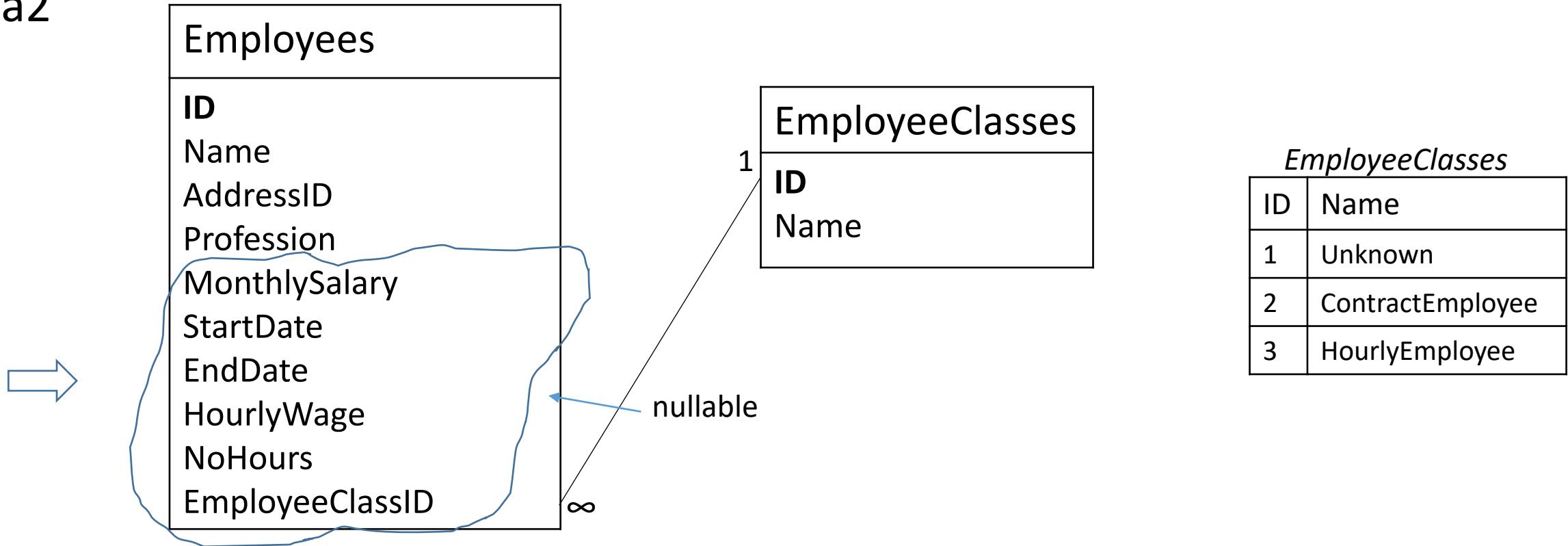
```
SELECT Employees.* , MonthlySalary , StartDate , EndDate
FROM Employees INNER JOIN ContractEmployees
ON Employees.ID = EmployeeID
```

- mapping inheritance
- a2
  - create one table for the superclass
  - the attributes of the subclasses become fields in the table
  - it generates the smallest number of objects
  - optionally, a subclasses table and a view / subclass can be added
  - usually – best performance
  - when adding a subclass, the existing structure has to be changed
  - "artificial" increase of used space

->

- mapping inheritance

- a2



```
CREATE VIEW ContractEmployees(....)
```

AS

```
SELECT ID, Name, AddressID, Profession, MonthlySalary, StartDate,
       EndDate
FROM Employees
WHERE EmployeeClassID = 2
```

- mapping inheritance
- a3
  - create one table / subclass
  - the attributes of the superclass become fields in each of the created tables
  - satisfactory performance
  - subclasses can be subsequently added without affecting existing tables
  - changing the structure of the superclass impacts all existing tables

->

- mapping inheritance
- a3



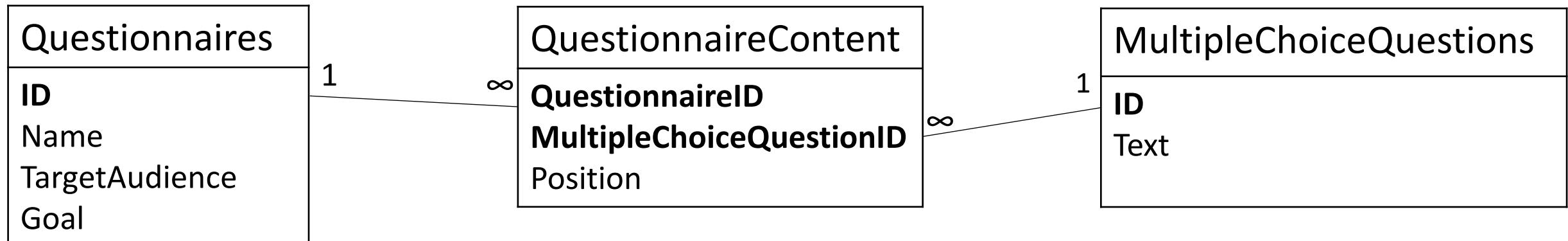
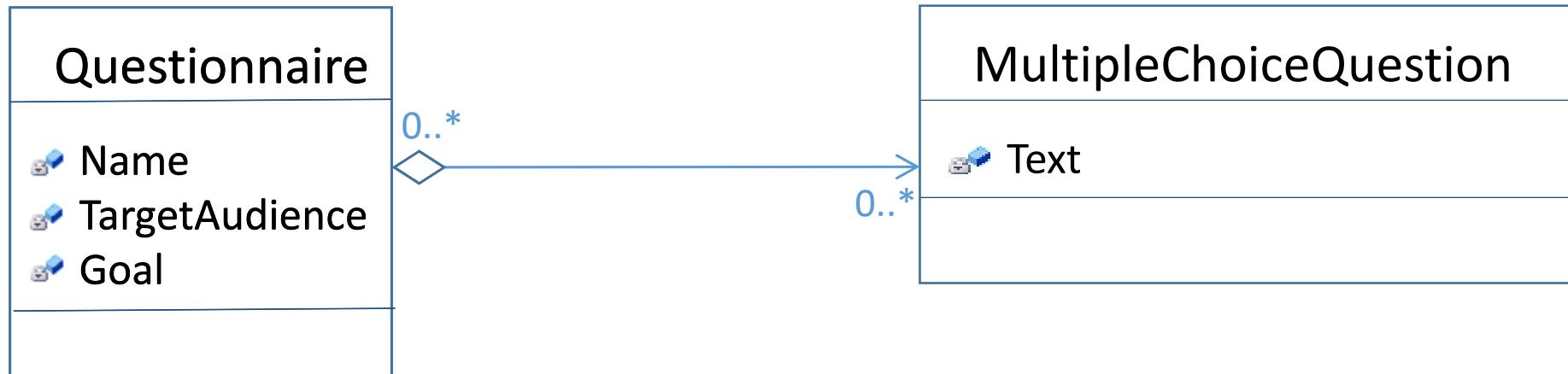
ContractEmployees	
<b>ID</b>	
Name	
AddressID	
Profession	
MonthlySalary	
StartDate	
EndDate	

HourlyEmployees	
<b>ID</b>	
Name	
AddressID	
Profession	
HourlyWage	
NoHours	

- mapping aggregation / composition
  - similar to mapping simple associations
  - fixed number of *parts* in a *whole* => can declare the same number of foreign keys in the *whole* table
  - composition - ON DELETE CASCADE option (not required for aggregation)

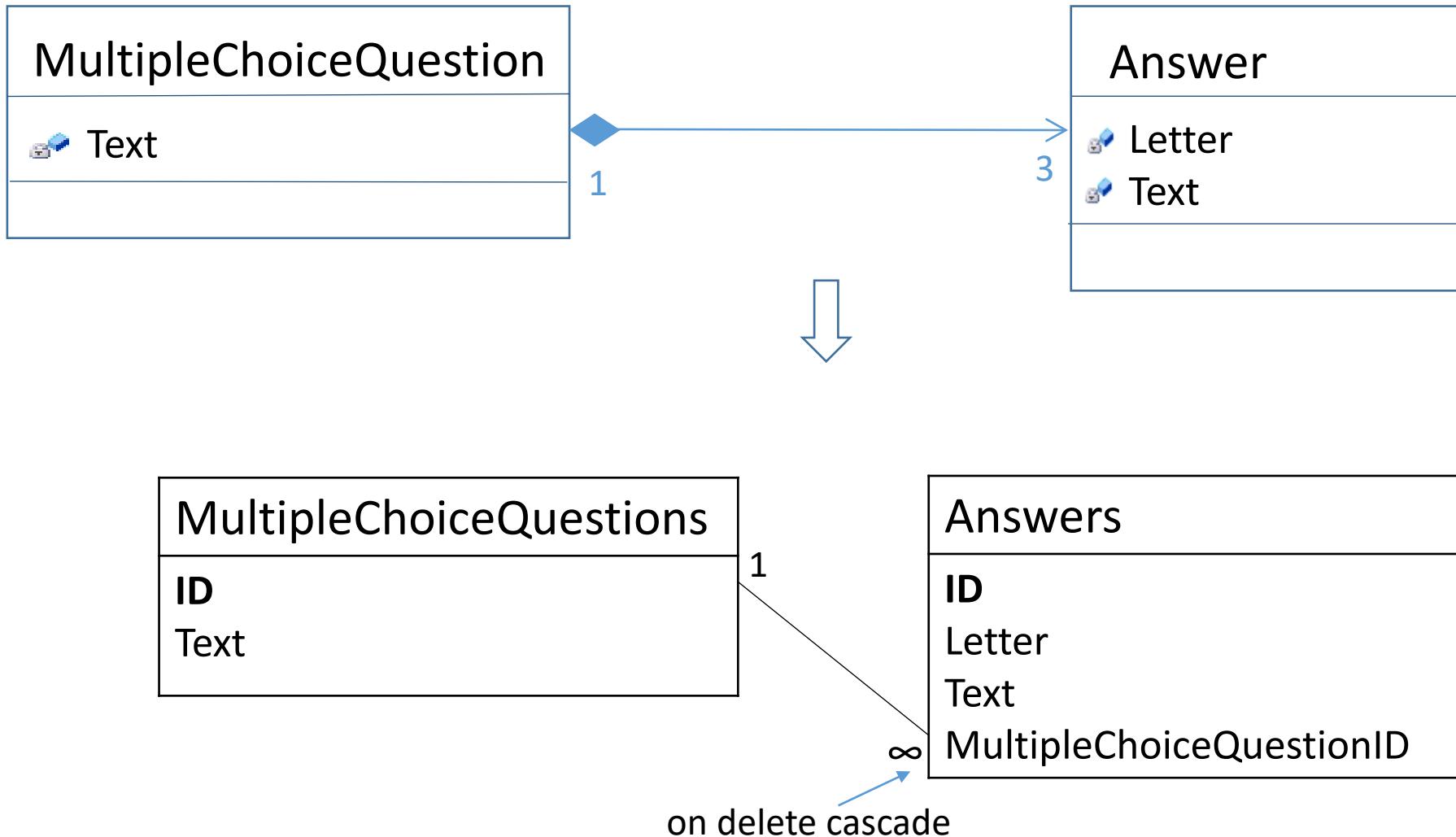
->

- mapping aggregation / composition

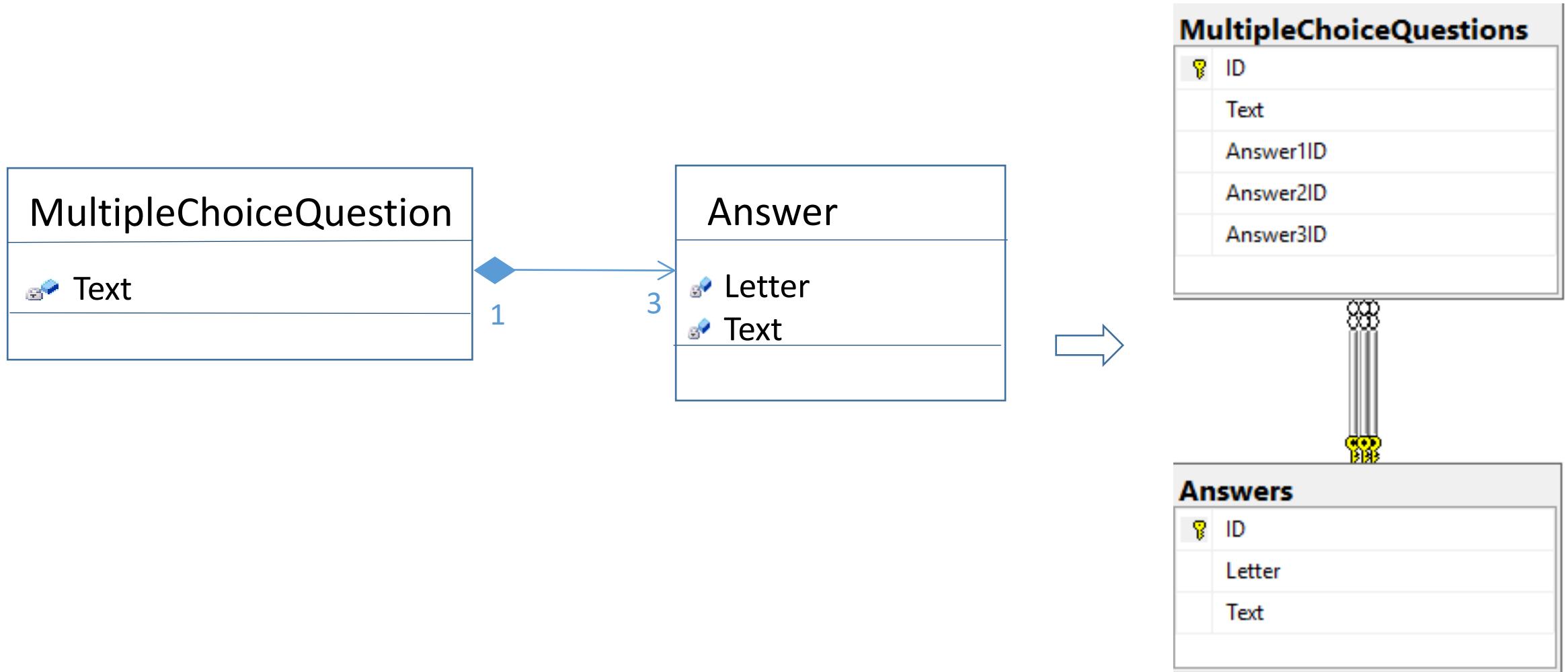


- obs. a questionnaire can also have open answer questions, etc.

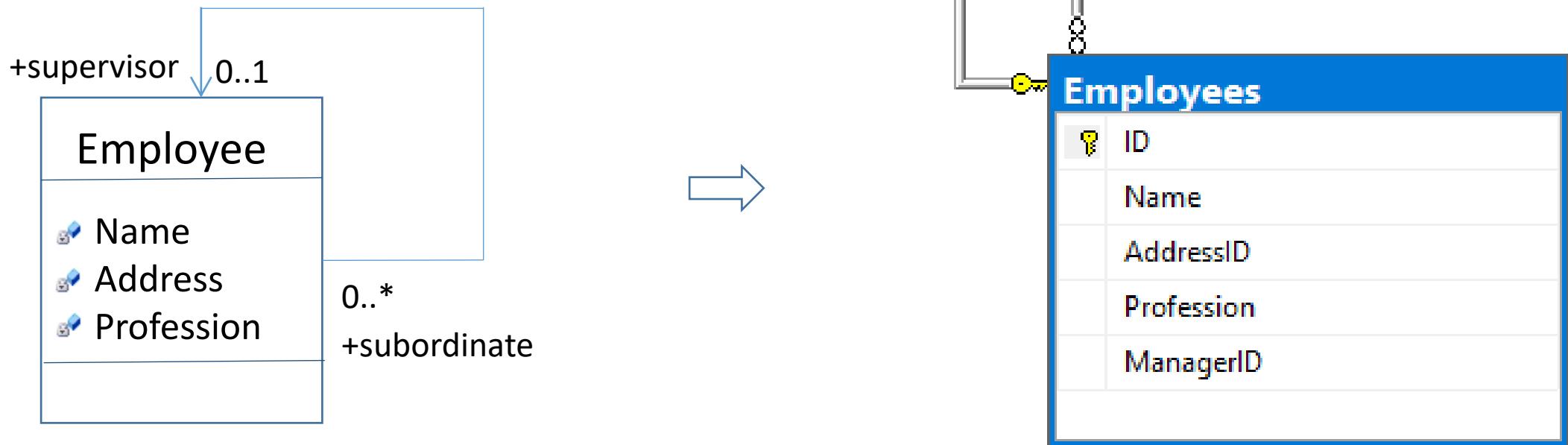
- mapping aggregation / composition



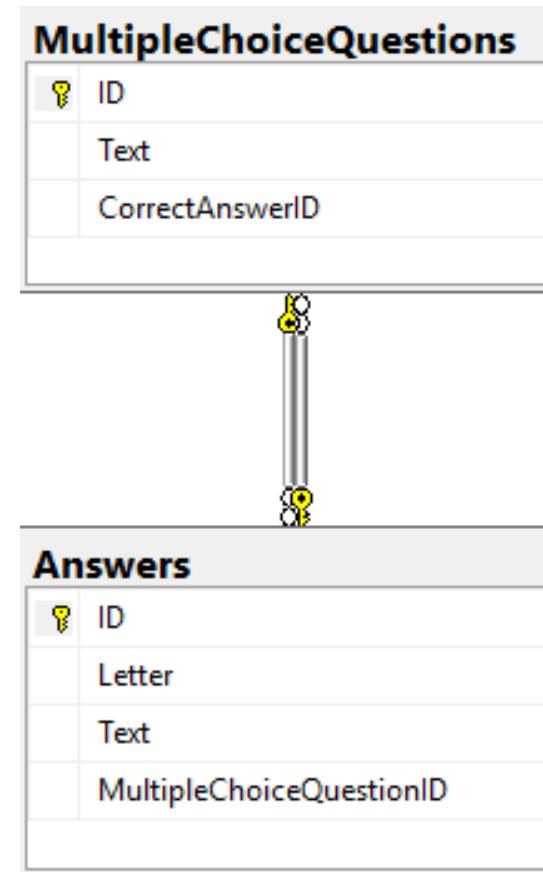
- mapping aggregation / composition



- mapping reflexive associations
- add a new field, referencing the same table (recursive relationship)
- ON DELETE CASCADE - error



Obs. 2 different tables, each with a foreign key referencing the other one, ON DELETE CASCADE - error



## References

- [Ta13] ȚÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009

# Databases

Lecture 14

Query Optimization in Relational Databases. Evaluating Relational Algebra Operators

Data Streams

## SQL Statements Execution

- client application - SQL statement execution request
  - for any query - minimum response time
- statement execution - stages:
  - client: generate SQL statement (non-procedural language), send it to server
  - server:
    - analyze SQL statement (syntactically)
    - translate statement into an internal form (relational algebra expression)
    - transform internal form into an optimal form
    - generate a procedural execution plan
    - evaluate procedural plan, send result to client

- the following operators are necessary in the querying process:
  - selection:  $\sigma_C(R)$
  - projection:  $\pi_\alpha(R)$
  - cross-product:  $R_1 \times R_2$
  - union:  $R_1 \cup R_2$
  - set-difference:  $R_1 - R_2$
  - intersection:  $R_1 \cap R_2$
  - theta join:  $R_1 \otimes_\Theta R_2$
  - natural join:  $R_1 * R_2$
  - left outer join:  $R_1 \ltimes_C R_2$
  - right outer join:  $R_1 \rtimes_C R_2$
  - full outer join:  $R_1 \bowtie_C R_2$
  - left semi join:  $R_1 \triangleright R_2$
  - right semi join:  $R_1 \triangleleft R_2$
  - division:  $R_1 \div R_2$
  - duplicate elimination:  $\delta(R)$
  - sorting:  $S_{\{list\}}(R)$
  - grouping:  $\gamma_{\{list1\} \text{ group by } \{list2\}}(R)$

- an SQL query can be written in multiple ways
- example for a relational database
- primary keys are underlined, foreign keys are written in blue
  - programs[id, pname, pdescription]
  - groups[id, **program**, yearofstudy, gdescription]
  - students[cnp, lastname, firstname, **sgroup**, gpa, addr, email]
- query: find students (lastname, firstname, year of study, program name, gpa) in a given program (e.g., with id = 2, can be a parameter), with a gpa >= 9 (can be a parameter):
  - a)

```
SELECT lastname, firstname, yearofstudy, pname, gpa
FROM students st, groups gr, programs pr
WHERE st.sgroup = gr.id AND gr.program = pr.id
      AND program = 2 and gpa >= 9
```

b)

```
SELECT lastname, firstname, yearofstudy, pname, gpa
FROM (students st INNER JOIN groups gr ON
      st.sgroup = gr.id)
      INNER JOIN programs pr ON gr.program = pr.id
WHERE program = 2 AND gpa >= 9
```

c)

```
SELECT lastname, firstname, yearofstudy, pname, gpa
FROM
(
  (SELECT lastname, firstname, sgroup, gpa
   FROM students
   WHERE gpa >= 9) st
  INNER JOIN
    (SELECT * FROM groups WHERE program = 2) gr
    ON st.sgroup = gr.id
)
  INNER JOIN
    (SELECT id, pname FROM programs WHERE id = 2) pr
    ON gr.program = pr.id
```

- the previous query versions are equivalent (they provide the same answer)
- equivalent relational algebra expressions:

a.

$$\pi_{\beta}(\sigma_C(students \times groups \times programs))$$

b.

$$\pi_{\beta}(\sigma_{C1}((students \otimes_{C2} groups) \otimes_{C3} programs))$$

c.

$$\pi_{\beta}(((\pi_{\beta 1}(\sigma_{C2}(students))) \otimes_{C3} (\sigma_{C4}(groups))) \otimes_{C5} (\pi_{\beta 2}(\sigma_{C6}(programs))))$$

- an evaluation tree can be constructed for a relational algebra expression
- problems:
  - which version is better?
  - when generating the execution plan:
    - which parameters are optimized?
    - what information is required?
  - what can the optimizer (DBMS component) do?

## Relational Algebra Operators - Evaluation

- operands for relational operators:
  - database tables (can have attached indexes)
  - temporary tables (obtained by evaluating some relational operators)
- several evaluation algorithms can be used for a relational algebra operator
- when generating the execution plan:
  - choose the algorithm with the lowest complexity (for the current database context); take into account data from the system catalog, statistical information

- a join can be defined as a cross-product followed by a selection
- joins arise more often in practice than cross-products
- in general, the result of a cross-product is much larger than the result of a join
- it's important to implement the join without materializing the underlying cross-product, by applying selections and projections as soon as possible, and materializing only the subset of the cross-product that will appear in the result of the join

## Cross Join

- this algorithm is used to evaluate a cross-product:
  - R CROSS JOIN S
  - R INNER JOIN S ON C (C evaluates to TRUE)
  - SELECT ... FROM R, S ..., no join condition between R and S
- $b_R, b_S$ 
  - the number of blocks storing R and S, respectively
- m, n
  - the number of blocks from R and S that can simultaneously appear in the main memory (there are  $m+n$  buffers for the 2 tables)

## Cross Join

- the following algorithm can be used to generate the cross-product  $\{(r, s) \mid r \in R, s \in S\}$ :
- for every group of max.  $m$  blocks in  $R$ :
  - read the group of blocks from  $R$  into main memory; let  $M_1$  be the set of records in these blocks
  - for every group of max.  $n$  blocks in  $S$ :
    - read the group of blocks from  $S$  into main memory; let  $M_2$  be the set of records in these blocks
    - for every  $r \in M_1$ :
      - for every  $s \in M_2$ : add  $(r, s)$  to the result

## Cross Join

- algorithm complexity: total number of read blocks (from the 2 tables):

$$b_R + \left\lceil \frac{b_R}{m} \right\rceil * b_S \quad (1)$$

(number of blocks in R; for every group of max. m blocks in R, read S)

- to minimize this value, m should be maximized (the other operands are constants); one buffer can be used for S (so n = 1), while the remaining space can be used for R (m max.)
- switch the 2 relations (in the algorithm and when computing the complexity)  
=> complexity:

$$b_S + \left\lceil \frac{b_S}{n} \right\rceil * b_R \quad (2)$$

- choose better version
- obs.: if  $b_R \leq m$  or  $b_S \leq n \Rightarrow$  complexity  $b_R + b_S$

## Nested Loops Join

- the Cross Join algorithm can be used to evaluate a join between 2 tables
- for every element  $(r, s)$  in the cross-product, evaluate the condition in the join operator
- elements  $(r, s)$  that don't meet the join condition are eliminated

## Indexed Nested Loops Join

- this algorithm is used to evaluate  $R \otimes_C S$ , where  $C \equiv (R.A=S.B)$ , and there is an index on A (in R) or on B (in S)
- in the algorithm description below, we assume there is an index on column B in table S
- for every block in R:
  - read the block into main memory; let M be the set of records in the block
  - for every  $r \in M$ :
    - determine  $v = \pi_A(r)$
    - use the index on B in S to determine records  $s \in S$  with value v for B; for every such record s, the pair  $(r,s)$  is added to the result
- obs.: depending on the type of index - at most 1 / multiple matching records in S

## Merge Join

- this algorithm is used to evaluate  $R \otimes_C S$ , where  $C \equiv (R.A=S.B)$ , and there are no indexes on A (in R) and B (in S)
- sort R and S on the columns used in the join: R on A, S on B
- scan obtained tables; let r in R and s in S be 2 current records
  - if  $r.A = s.B$ : add  $(r', s')$  to the result;  $r'$  is in the set of all consecutive records in R with  $A = r.A$ , similarly for  $s'$  in S;  $\text{next}(r)$ ;  $\text{next}(s)$  (get a record with the next value for A and B)
  - if  $r.A < s.B$ :  $\text{next}(r)$  (determine record in sorted R with the next value for A)
  - if  $r.A > s.B$ :  $\text{next}(s)$  (determine record in sorted S with the next value for B)

## Relational Algebra Equivalences

- SQL statement - transformed into a relational algebra expression (based on a set of transformation rules for the clauses that appear in the statement)
- transform relational expression (such that the evaluation algorithm has a lower complexity)
- certain transformation rules are used (mathematical properties of the relational operators)

- \*  $\sigma_C(\pi_\alpha(R)) = \pi_\alpha(\sigma_C(R))$
- selection reduces the number of records for projection; in the second expression, the projection operator analyzes fewer records
- optimization - algorithm that evaluates both operators in a single pass of R

- \* perform one pass instead of 2:

$$\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_1 \text{ AND } C_2}(R)$$

- \* replace cross-product and selection by condition join (a number of condition join algorithms don't evaluate the cross-product):

$$\sigma_C(R \times S) = R \otimes_C S$$

, where C - join condition between R and S

\* R and S - compatible schemas:

$$\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$$

$$\sigma_C(R \cap S) = \sigma_C(R) \cap \sigma_C(S)$$

$$\sigma_C(R - S) = \sigma_C(R) - \sigma_C(S)$$

\*  $\sigma_C(R \times S)$

particular cases:

- C contains only attributes from R:

$$\sigma_C(R \times S) = \sigma_C(R) \times S$$

- C = C1 AND C2, C1 contains only attributes from R, C2 - only attributes from S:

$$\sigma_{C1 \text{ AND } C2}(R \times S) = \sigma_{C1}(R) \times \sigma_{C2}(S)$$

- C = C1 AND C2, C2 - join condition between R and S:

$$\sigma_{C1 \text{ AND } C2}(R \times S) = \sigma_{C1}(R \otimes_{C2} S)$$

$$* \pi_\alpha(R \cup S) = \pi_\alpha(R) \cup \pi_\alpha(S)$$

$$* \pi_\alpha(R \otimes_C S) = \pi_\alpha(\pi_{\alpha 1}(R) \otimes_C \pi_{\alpha 2}(S))$$

- $\alpha 1$ : attributes in R that appear in  $\alpha$  or C
- $\alpha 2$ : attributes in S that appear in  $\alpha$  or C

\* associativity and commutativity for some relational operators

- associativity and commutativity for U and  $\cap$
- associativity for the cross-product and the natural join
- "equivalent" results (same records, but different column order) when commuting operands in  $\times$  and certain join operators
  - $R \times S = S \times R$  – when using the Cross Join algorithm, the order of the data sources is important

\* transitivity of some relational operators for the join operators - additional filters could be applied before the join:

- $(A > B \text{ AND } B > 3) \equiv (A > B \text{ AND } B > 3 \text{ AND } A > 3)$

- example: A is in R, B is in S:

$$R \otimes_{A > B \text{ AND } B > 3} S = (\sigma_{A > 3}(R)) \otimes_{A > B} (\sigma_{B > 3}(S))$$

- $(A = B \text{ AND } B = 3) \equiv (A = B \text{ AND } B = 3 \text{ AND } A = 3)$

- example: A is in R, B is in S:

$$R \otimes_{A = B \text{ AND } B = 3} S = (\sigma_{A = 3}(R)) \otimes_{A = B} (\sigma_{B = 3}(S))$$

\* evaluating  $\sigma_C(R)$ , where  $C \equiv (R.A \in \delta(\pi_{\{B\}}(S)))$ ; avoid evaluating C for every record of R; the initial evaluation is equivalent to:

$$R \otimes_{R.A = S.B} (\delta(\pi_{\{B\}}(S)))$$

- consider again the query described on the database:  
programs[id, pname, pdescription]  
groups[id, program, yearofstudy, gdescription]  
students[cnp, lastname, firstname, sgroup, gpa, addr, email]
- query: find students (lastname, firstname, year of study, program name, gpa) in a given program (e.g., with id = 2), with a gpa  $\geq 9$ :

```
SELECT lastname, firstname, yearofstudy, pname, gpa  
FROM students, groups, programs  
WHERE students.sgroup = groups.id AND  
groups.program = programs.id AND  
program = 2 and gpa >= 9
```

- denote by:

$C \equiv (\text{students.sgroup} = \text{groups.id} \text{ AND } \text{groups.program} = \text{programs.id} \text{ AND } \text{program} = 2 \text{ and } \text{gpa} \geq 9)$

$\beta = \{\text{lastname}, \text{firstname}, \text{yearofstudy}, \text{pname}, \text{gpa}\}$  – attributes in the SELECT clause

- the corresponding relational expression:

$$\pi_{\beta}(\sigma_C(\text{students} \times \text{groups} \times \text{programs}))$$

\* carry out the following transformations, using previously discussed rules:

- associativity for  $\times$ :

$$students \times groups \times programs = (students \times groups) \times programs \quad \text{or}$$

$$students \times groups \times programs = students \times (groups \times programs)$$

- commute  $\sigma$  with  $\times$  (a particular case); use the transitivity of the equality operator:

$$(groups.program = programs.id \text{ AND } program = 2)$$

$$\equiv (groups.program = programs.id \text{ AND } program = 2 \text{ AND } programs.id = 2)$$

students.sgroup = groups.id AND groups.program = programs.id AND program = 2 AND gpa >= 9 AND programs.id = 2

C1

C2

C3

C4

C5

$$\sigma_C(students \times groups \times programs) =$$

$$\sigma_{C1 \text{ AND } C2}((\sigma_{C4}(students) \times \sigma_{C3}(groups)) \times \sigma_{C5}(programs)) \text{ or}$$

$$\sigma_{C1 \text{ AND } C2}(\sigma_{C4}(students) \times (\sigma_{C3}(groups) \times \sigma_{C5}(programs)))$$

- replace selection and cross-product with condition join:  

$$= ((\sigma_{C4}(students)) \otimes_{C1} (\sigma_{C3}(groups))) \otimes_{C2} (\sigma_{C5}(programs))$$
- or
- $= (\sigma_{C4}(students)) \otimes_{C1} ((\sigma_{C3}(groups)) \otimes_{C2} (\sigma_{C5}(programs)))$

- choose a version based on statistical information from the database; we consider the first version:

$$\Rightarrow e = \pi_\beta((\sigma_{C4}(students)) \otimes_{C1} (\sigma_{C3}(groups))) \otimes_{C2} (\sigma_{C5}(programs))$$

- commute  $\pi$  with join:

$\beta_1 = \{\text{lastname, firstname, gpa, sgroup}\}$  - useful for  $\beta$  and join

$\beta_2 = \{\text{id, program, yearofstudy}\}$  - useful for  $\beta$  and join

$\beta_3 = \{\text{id, pname}\}$  - useful for  $\beta$  and join

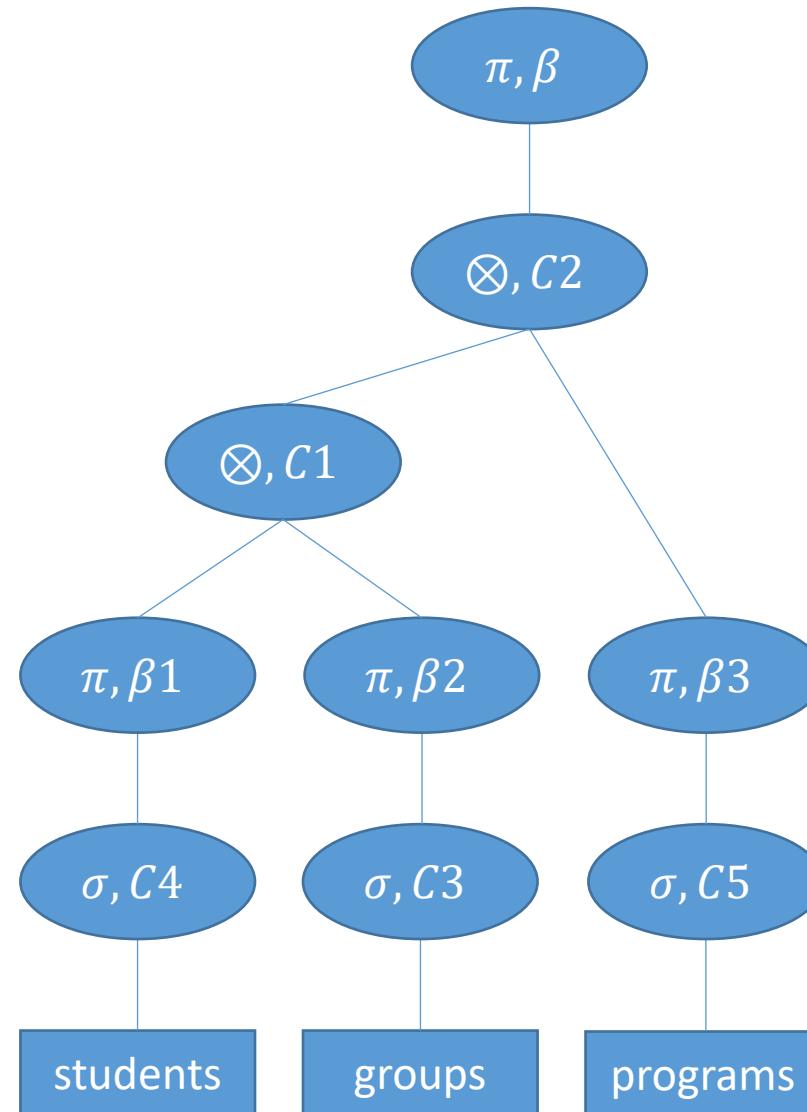
$$e = \pi_\beta((\pi_{\beta_1}(\sigma_{C4}(students))) \otimes_{C1} (\pi_{\beta_2}(\sigma_{C3}(groups)))) \otimes_{C2} (\pi_{\beta_3}(\sigma_{C5}(programs)))$$

- the last expression corresponds to the statement:

```
SELECT lastname, firstname, yearofstudy, pname, gpa
FROM
(
  (SELECT lastname, firstname, gpa, sgroup FROM students WHERE gpa >= 9) st
  INNER JOIN
  (SELECT id, program, yearofstudy FROM groups WHERE program = 2) gr
    ON st.sgroup = gr.id
)
INNER JOIN
  (SELECT id, pname FROM programs WHERE programs.id = 2) pr
  ON gr.program = pr.id
```

- an evaluation tree can be constructed for the last version of the relational algebra expression

- using information from the system catalog and possibly statistical information, an execution plan can be generated from the last version of the expression; every relational operator is replaced by an evaluation algorithm



\* Let P, Q, R be 3 relations with schemas P[PID, P1, P2, P3], Q[QID, Q1, Q2, Q3, Q4, Q5], R[RID, R1, R2, R3], and E an expression in the relational algebra:

$$E = \pi_{\{P2, Q2, Q4, R3\}} (\sigma_{PID = Q1 \text{ AND } QID = R2 \text{ AND } P3 = 'Bilbo' \text{ AND } Q5 = 100 \text{ AND } R1 = 7} (P \times Q \times R))$$

Optimize E and draw the evaluation tree for the optimized version of the expression.

# Data Streams

## Data Processing in Traditional DBMSs

- classical DBMSs answer the needs of traditional business applications
- finite data sets
- users execute queries on the database when necessary
- *one-shot (one-time) query*
  - executed on the current instance of the data (entirely stored)
  - finite time interval
  - specific to traditional DBMSs
- *human-active, DBMS-passive (HADP) model*
  - database - passive repository
  - users execute queries on the database when necessary

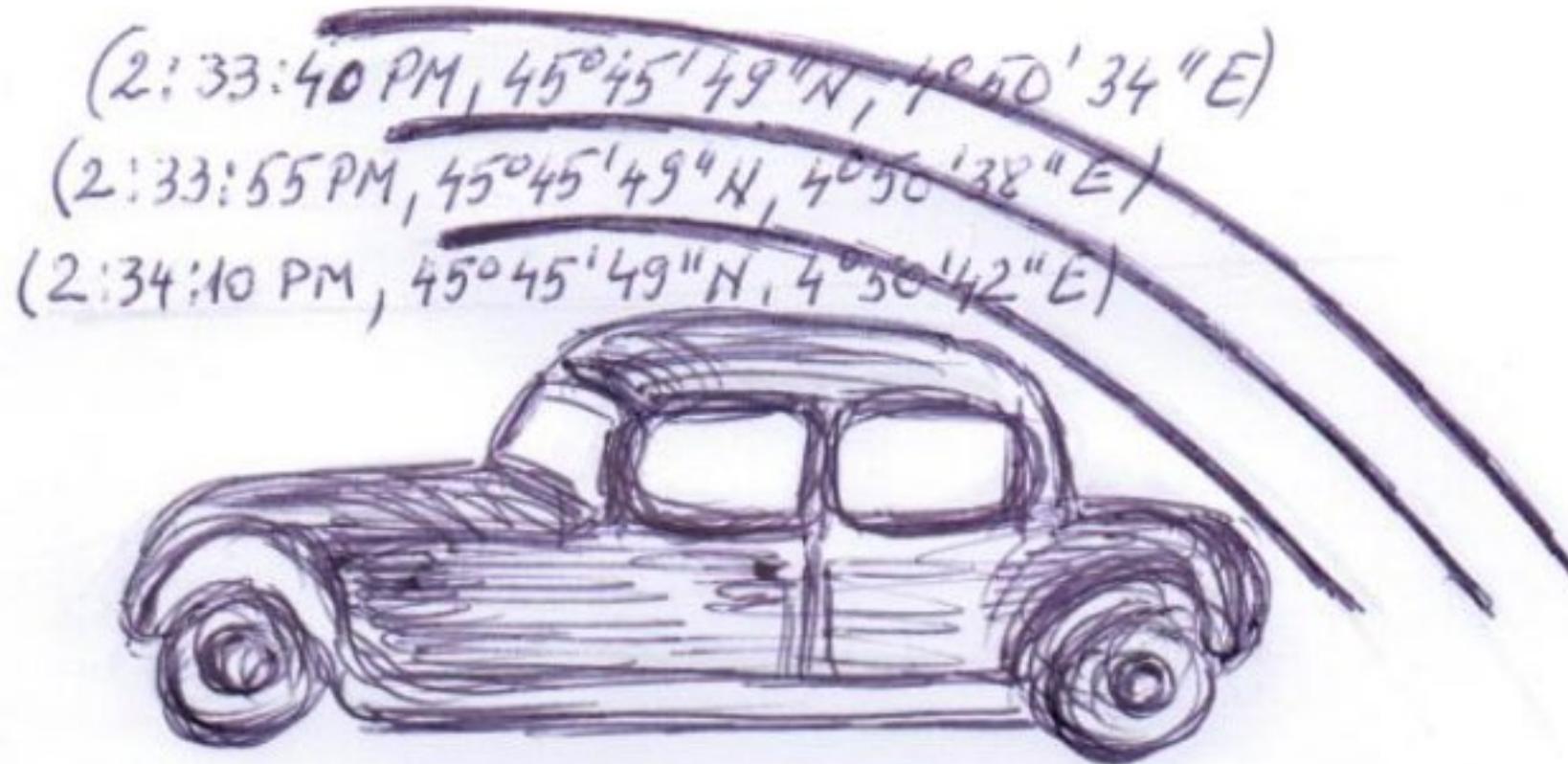
## Data Streams

- in a range of applications, data cannot be efficiently managed with a classical DBMS, as information takes the form of the so-called *data streams*
- e.g., astronomy, meteorology, seismology, financial services, e-commerce, etc.
- *data stream* - temporal sequence of values produced by a data source
  - potentially infinite
  - data arriving on the stream is associated with temporal values, i.e., *timestamps*
- examples
  - a sequence of values provided by a temperature sensor
  - a sequence of GPS coordinates emitted by a car as it runs on a highway
  - a sequence of values representing a patient's heart rate and blood pressure

## Data Streams

- time - common element in the examples above
- *event*
  - elementary unit of information that arrives on a data stream (similar to a record in relational databases); synonyms in this lecture, unless otherwise noted - *tuple, element*
- systems discussed in this lecture – structured data streams
- *data source*
  - a device that provides a stream of values over time, in a digital format (a temperature sensor, a GPS device, a device that monitors a patient's heart, etc.)

## Data Streams



- 3 tuples on a stream of coordinates produced by the GPS device of a car
- the GPS emits the current location of the car (latitude and longitude) every 15 seconds

# Data Stream Monitoring Applications

- *monitoring applications*
  - applications that scan data streams, process incoming values, and compute the desired result
- e.g., military applications, financial analysis applications, variable tolling applications, etc.

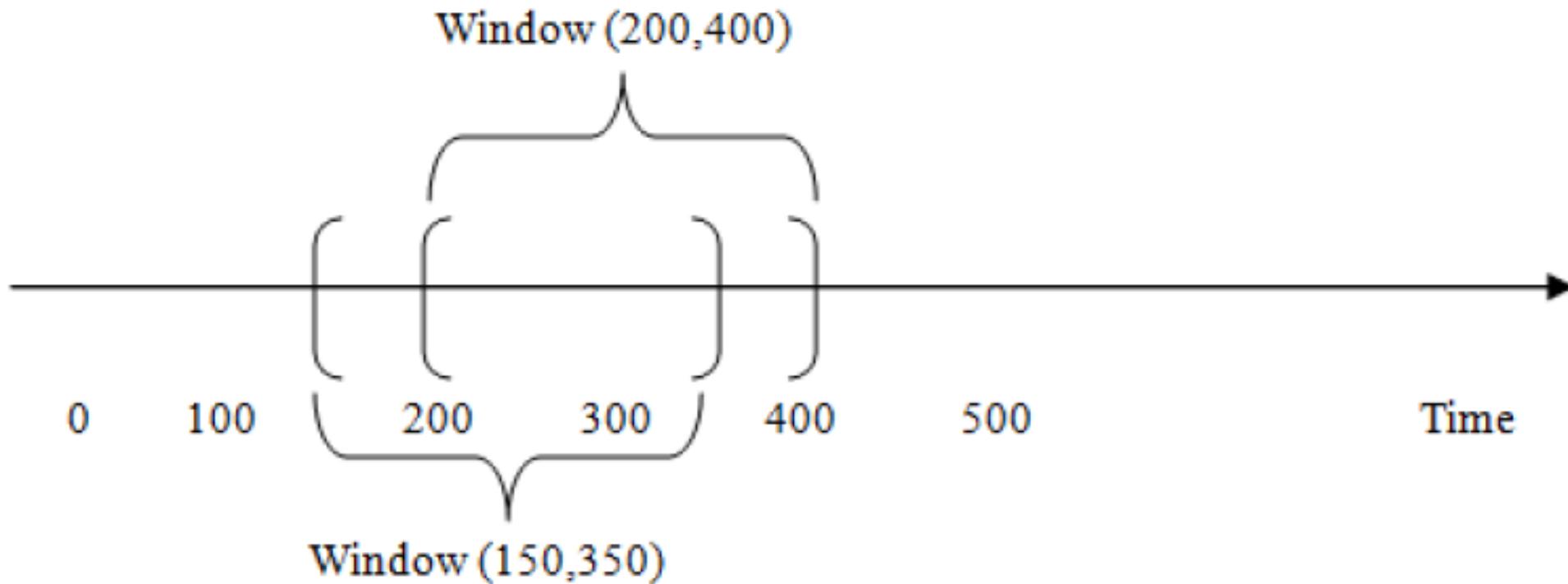
# Window-Based Processing Model

- data streams
  - potentially infinite
  - high data rates
- traditional DBMSs
  - vast storage space, secondary memory
- systems that process streams
  - usually rely on the main memory
- storing all the data - impossible
- data arriving on a stream
  - instantaneously processed, then eliminated
- evaluating queries on data streams
  - window-based model

## Window-Based Processing Model

- consider a temperature sensor in a refrigeration container; the user wants to be alerted whenever the temperature in the container exceeds a threshold 3 times in the last 10 minutes; it's enough to analyze the window of data that arrived on the stream in the previous 10 minutes; as time goes by and new tuples arrive on the stream, the window slides over the data in the stream
- *sliding window*
  - a contiguous portion of data from a stream
  - parameters
    - size - number of events / temporal instants
    - step size - number of events / temporal instants

# Window-Based Processing Model



- sliding window
  - size = 200 timestamps
  - step size = 50 timestamps

## Continuous Queries

- perpetually running queries, continuously producing results, while being fed with data from one or several streams
- provide real-time results, as required by many monitoring applications
  - e.g., variable tolling app that computes highway tolls based on dynamic factors such as accident proximity or traffic congestion
    - a driver must be alerted in real time whenever a new toll is issued for his or her car
    - providing this answer later in the future would be of no use
  - e.g., nuclear plant management
- continuous processing paradigm
  - *DBMS-active, human-passive (DAHP)*
  - database – active role
  - user – passive role

## Data Stream Management Systems

- the number of data sources providing monitored streams can grow significantly
- stream rates can be uniform, but data can also arrive in bursts (e.g., a stream of clicks from the website of a company when a new product is launched)
- the number of continuous queries / monitored data streams can also fluctuate considerably
- the complexity of the running queries can vary over time
- as system resources are limited, the system can become overloaded and unable to provide real-time results
- traditional DBMSs cannot tackle these challenges, being unable to efficiently manage data streams; dedicated systems, that use various strategies to handle such problems, are being used instead

# Data Stream Management Systems

- dedicated systems can execute continuous queries, while meeting the requirements of monitoring applications
- *Data Stream Management System*
  - system that processes streams of data in a perpetual manner, by running continuous queries
  - built around a query processing engine, which performs data manipulation operations
- academic prototypes
  - STREAM, Aurora, Borealis, etc.
- commercial systems
  - Azure Stream Analytics

# Classical Databases Versus Data Streams

- classical DBMSs
  - permanent elements
    - data
  - temporary elements
    - queries
- DSMSs
  - permanent elements
    - continuous queries
  - transient elements
    - data arriving on streams

## STREAM - STandard stREam datA Manager

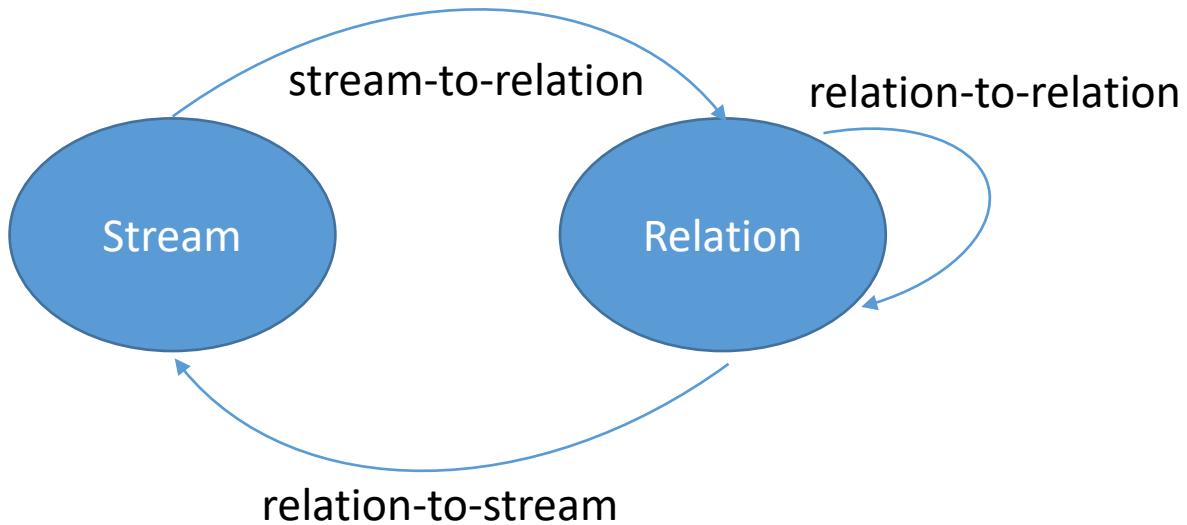
- DSMS prototype developed at Stanford
- objective
  - study data management and query processing in monitoring apps
- continuous queries on streams / stored data sets
- formal abstract semantics for continuous queries
- concrete declarative language, i.e., the Continuous Query Language (similar to SQL)

## STREAM - abstract semantics

- 2 data types
  - streams and relations
- discrete, ordered time domain  $T$ 
  - a timestamp  $t$  - a temporal moment from  $T$
  - $\{0, 1, \dots\}$
- data stream  $S$ 
  - unbounded multiset of tuple-timestamp pairs  $\langle s, t \rangle$
  - fixed schema, named attributes
- relation  $R$ 
  - time-varying multiset of tuples
  - $R(t)$  - instantaneous relation (i.e., the multiset of tuples at time  $t$ )
  - fixed schema, named attributes

# STREAM - abstract semantics

- 3 classes of operators
  - relation-to-relation
  - stream-to-relation
  - relation-to-stream



## STREAM - abstract semantics

- *relation-to-relation* operator
  - takes one or several input relations and produces an output relation
- *stream-to-relation* operator
  - takes an input stream and produces an output relation
- *relation-to-stream* operator
  - takes an input relation and produces an output stream
- stream-to-stream operators can be defined using the 3 classes of operators from the semantics
- operator classes
  - black box components
  - the semantics depends on the generic properties of each class, not on the operators' implementations

# STREAM - Continuous Query Language (CQL)

- minor extension of SQL
- defined by instantiating operators in the abstract semantics
- relation-to-relation operators
  - SQL constructs that transform several relations into a single relation
  - select, project, union, except, intersect, aggregate, etc.
- stream-to-relation operators
  - extract a sliding window from a stream
  - window-specification language derived from SQL-99
  - sliding window - 3 types
    - tuple-based sliding window
    - time-based sliding window
    - partitioned sliding window

# STREAM - Continuous Query Language

- tuple-based sliding window
  - contains the last N tuples from the stream
  - S - stream, N - positive integer
  - $S[\text{Rows } N]$  produces a relation R
  - at time t,  $R(t)$  contains the N tuples that arrived on S and have the largest timestamps  $\leq t$
  - special case
    - $N = \infty$ 
      - $S[\text{Rows Unbounded}]$  - append-only window

## STREAM - Continuous Query Language

- time-based sliding window
  - $S$  - stream,  $t_i$  - temporal interval
  - $S[\text{Range } t_i]$  produces a relation  $R$
  - at time  $t$ ,  $R(t)$  contains the tuples that arrived on  $S$  and have the timestamps between  $t-t_i$  and  $t$
  - special cases
    - $t_i = 0$ 
      - i.e., the tuples on  $S$  with timestamp =  $t$
      - $S[\text{Now}]$
    - $t_i = \infty$ 
      - $S[\text{Range Unbounded}]$

# STREAM - Continuous Query Language

- time-based sliding window
  - e.g., CarStream(CarID, Speed, Position, Direction, Road)
    - CarStream[Range 60 seconds]
    - CarStream[Now]
    - CarStream[Range Unbounded]

# STREAM - Continuous Query Language

- relation-to-stream operators
- Istream (insert stream)
  - applied to a relation  $R$ , it contains  $\langle s, t \rangle$  whenever  $s$  is in  $R(t) - R(t-1)$  ( $s$  is added to  $R$  at time  $t$ )
- Dstream (delete stream)
  - applied to a relation  $R$ , it contains  $\langle s, t \rangle$  whenever  $s$  is in  $R(t-1) - R(t)$  ( $s$  is removed from  $R$  at time  $t$ )
- Rstream (relation stream)
  - applied to a relation  $R$ , it contains  $\langle s, t \rangle$  whenever  $s$  is in  $R(t)$  (every current tuple in  $R$  is streamed at every time instant)

## STREAM - Continuous Query Language

- example CQL queries
- CarStream(CarID, Speed, Position, Direction, Road)
  - at any given time, display the set of active cars (i.e., having transmitted a position report in the past 60 seconds)

```
SELECT DISTINCT CarID  
FROM CarStream [Range 60 Seconds]
```

- the result is a relation

## STREAM - Continuous Query Language

- example CQL queries
- windowed join of 2 streams

```
SELECT *
```

```
FROM S1 [ROWS 200], S2 [RANGE 5 Minutes]
```

```
WHERE S1.Attr = S2.Attr AND S1.Attr < 500
```

- result = relation
- at every temporal instant  $t$ , the result contains the join (on  $Attr$ ) of the last 200 tuples of  $S1$  with the tuples that arrived on  $S2$  in the past 5 minutes; only tuples with  $Attr < 500$  are part of the result

STREAM - maybe in 2 years from now (Master's Programmes) :)

- sharing data & computation within and across execution plans
- exploiting stream constraints - ordering, clustering, etc.
- load-shedding
- etc.

## References

- [Ta13] ȚÂMBULEA, L., Curs Baze de date, Facultatea de Matematică și Informatică, UBB, 2013-2014
- [Da03] DATE, C.J., An Introduction to Database Systems (8<sup>th</sup> Edition), Addison-Wesley, 2003
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Kn76] KNUTH, D.E., Tratat de programare a calculatoarelor. Sortare și căutare. Ed. Tehnică, București, 1976
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009

## References

- Daniel J. Abadi, Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003
- A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava and J. Widom. STREAM: The Stanford Data Stream Management System. Technical Report, Stanford InfoLab, 2004
- Arvind Arasu, Shivnath Babu and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal-Raport tehnic*, 15(2):121–142, 2006
- A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebreaker and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. In VLDB'04, Proceedings of The Thirtieth International Conference on Very Large Data Bases, pages 480–491, 2004

## References

- Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani and Jennifer Widom. Models and Issues in Data Stream Systems. In *PODS*, pages 1–16, 2002
- Y. Gripay, F. Laforest, F. Lesueur, N. Lumineau, J.-M. Petit, V.-M. Scuturici, S. Sebahi, S. Surdu, Colistrack: Testbed For A Pervasive Environment Management System, Proceedings of The 15th International Conference on Extending Database Technology (EDBT 2012), 574-577, 2012
- \*\*\* Azure Stream Analytics - technical documentation, <https://azure.microsoft.com/en-us/services/stream-analytics/>