



# Database Systems

## Session 2 – Sub-Topic 3 Legacy & Non 1NF (NFNF) Databases Dr. Jean-Claude Franchitti

*New York University  
Computer Science Department  
Courant Institute of Mathematical Sciences*

# Agenda

1 Session Overview

2 Legacy Databases

3 OODBs and Object Persistence

4 Extended Relational Databases

5 XML Databases

6 NoSQL Databases

7 Summary and Conclusion

# Session Agenda (1 of 2)

- Legacy Databases
  - Hierarchical Model
  - CODASYL Model
- OODBs and Object Persistence
  - Recap: Basic Concepts of OO
  - Advanced Concepts of OO
  - Basic Principles of Object Persistence
  - OODBMS
  - Evaluating OODBMSs
- Extended Relational Databases
  - Success of the relational model
  - Limitations of the Relational Model
  - Active RDBMS Extensions
  - Object-Relational RDBMS extensions
  - Recursive SQL queries

## Session Agenda (2 of 2)

- XML Databases
  - Extensible Markup Language
  - Processing XML Documents
  - Storage of XML Documents
  - Differences between XML and Relational Data
  - Mappings Between XML Documents and (Object-) Relational Data
  - Searching XML Data
  - XML for Information Exchange
  - Other Data Representation Formats
- NoSQL Databases
  - The NoSQL movement
  - Key-Value stores
  - Tuple and Document stores
  - Column-oriented databases
  - Graph based databases
  - Other NoSQL categories

# Icons / Metaphors



Information



Common Realization



Knowledge/Competency Pattern



Governance



Alignment



Solution Approach

# Agenda

- 
- 1 Session Overview
  - 2 Legacy Databases
  - 3 OODBs and Object Persistence
  - 4 Extended Relational Databases
  - 5 XML Databases
  - 6 NoSQL Databases
  - 7 Summary and Conclusion

# Agenda

- Legacy Databases
  - Hierarchical Model
  - CODASYL Model

## Hierarchical Model

- The hierarchical model originated during the Apollo program conducted by NASA
  - » IBM developed the Information Management System or IMS DBMS (1966-1968)
- No formal description available and lots of structural limitations (legacy)
- Two key building blocks: record types and relationship types



## Hierarchical Model

- A record type is a set of records describing similar entities and has 0, 1 or more records
  - » Examples: product record type, supplier record type
- A record type consists of fields or data items
  - » Examples: product number, product name, product color

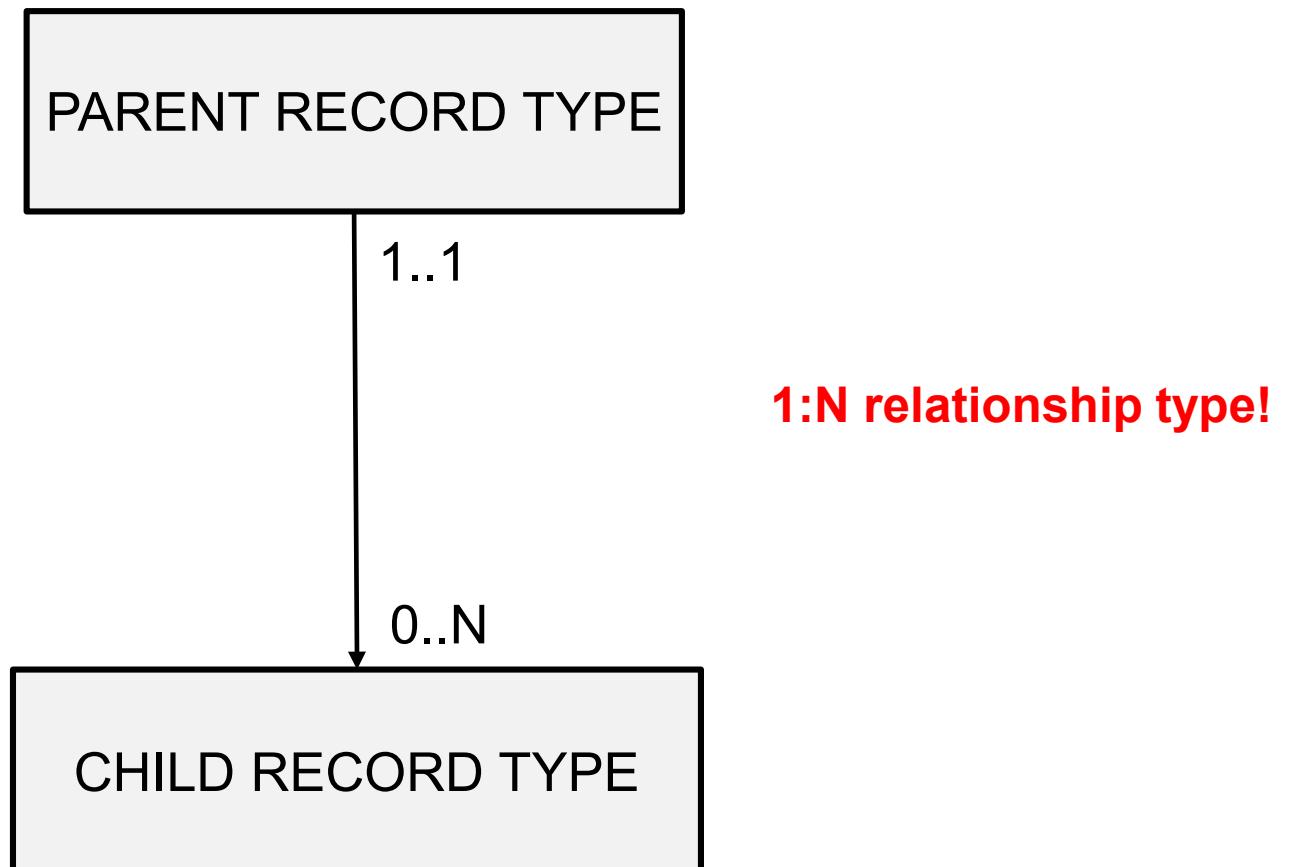


# Hierarchical Model

- A relationship type connects two record types
- Only hierarchical structures are allowed (1:N relationship types)
- A record type can be a parent in multiple parent/child relationship types, but it can participate in at most one relationship type as a child
- Relationship types can be nested
- Root record type sits as the top of the hierarchy, whereas leaf record type sits at the bottom

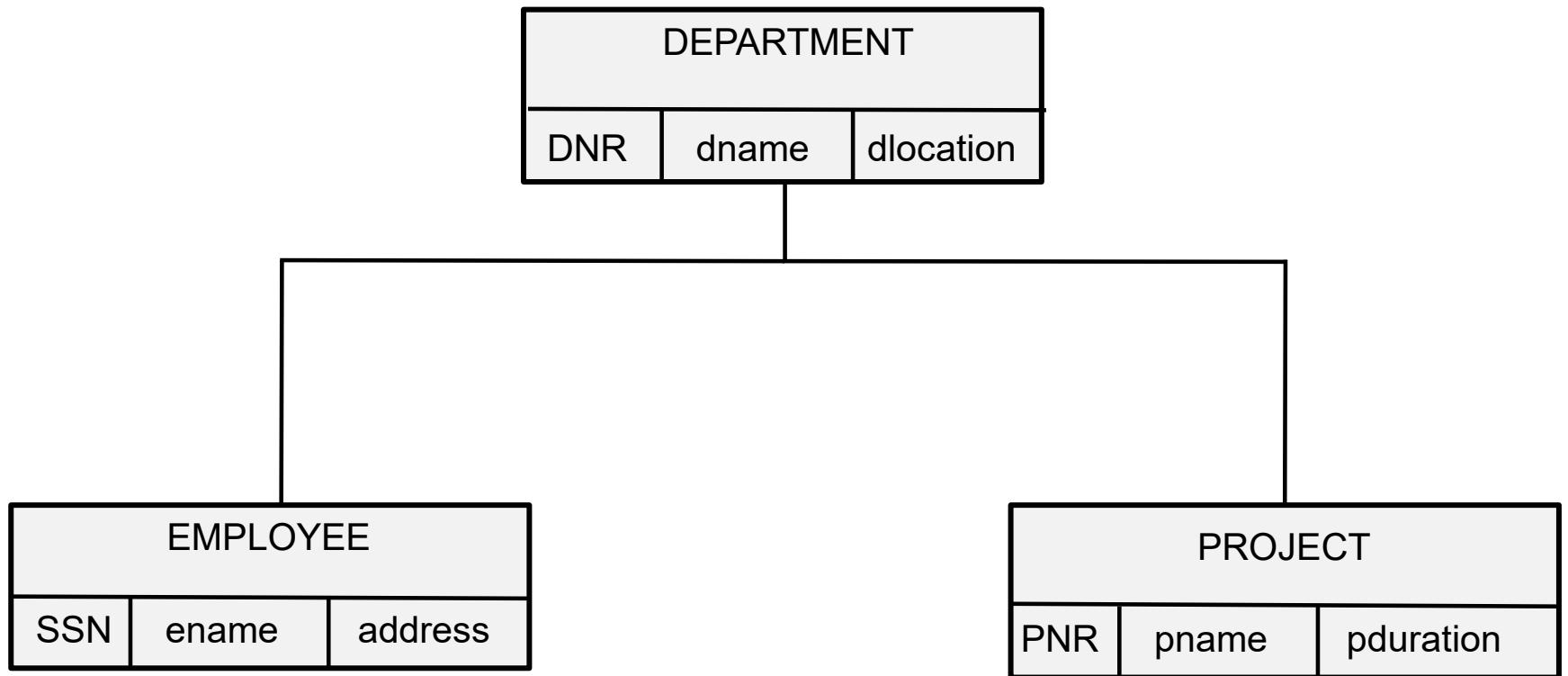


# Hierarchical Model





# Hierarchical Model



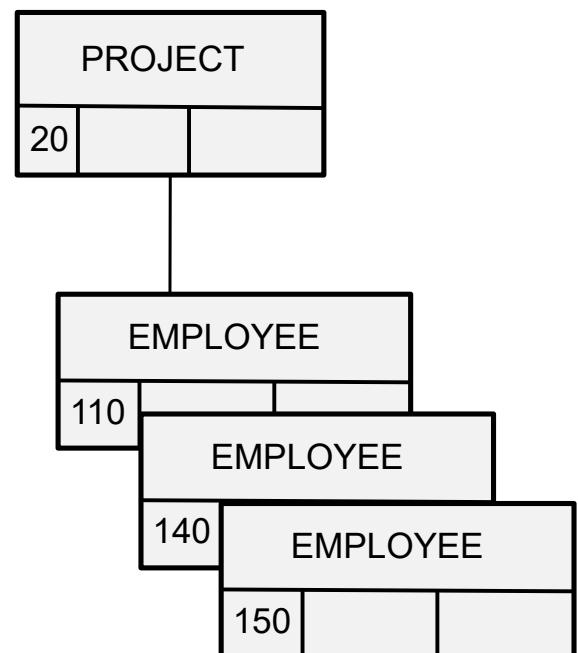
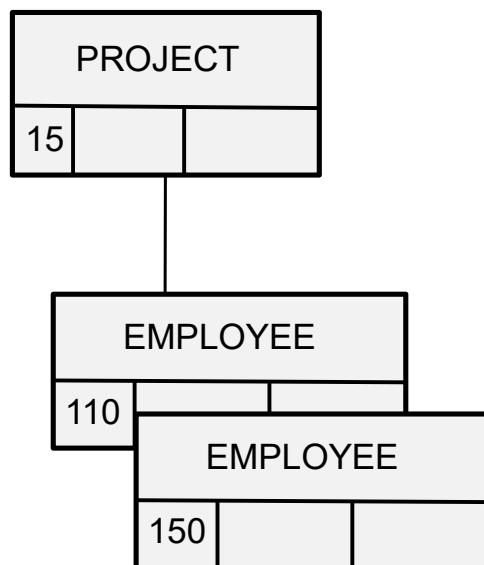
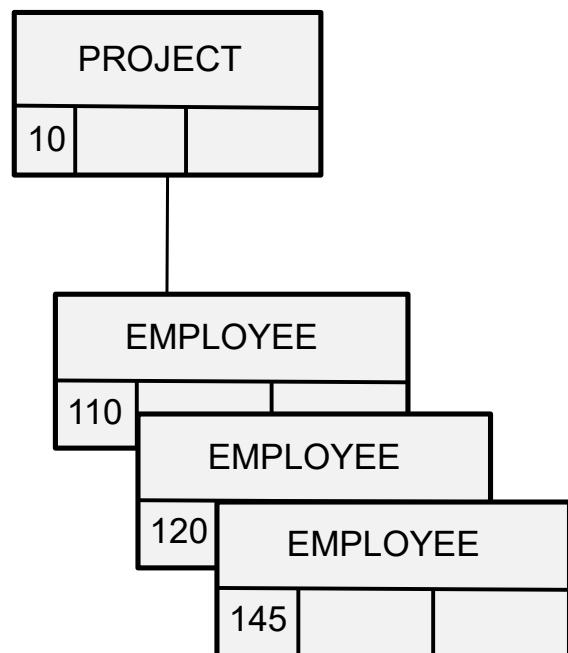


# Hierarchical Model

- All data needs to be retrieved by navigating down from the root node (procedural DML)
- The hierarchical model is also very rigid and thus limited in terms of expressive power
- No support for N:M or 1:1 relationship types
- N:M relationship type
  - » assign one record type as the parent and the other as the child record type
  - » put relationship type attributes in child record type
  - » however: redundancy is introduced!



# Hierarchical Model



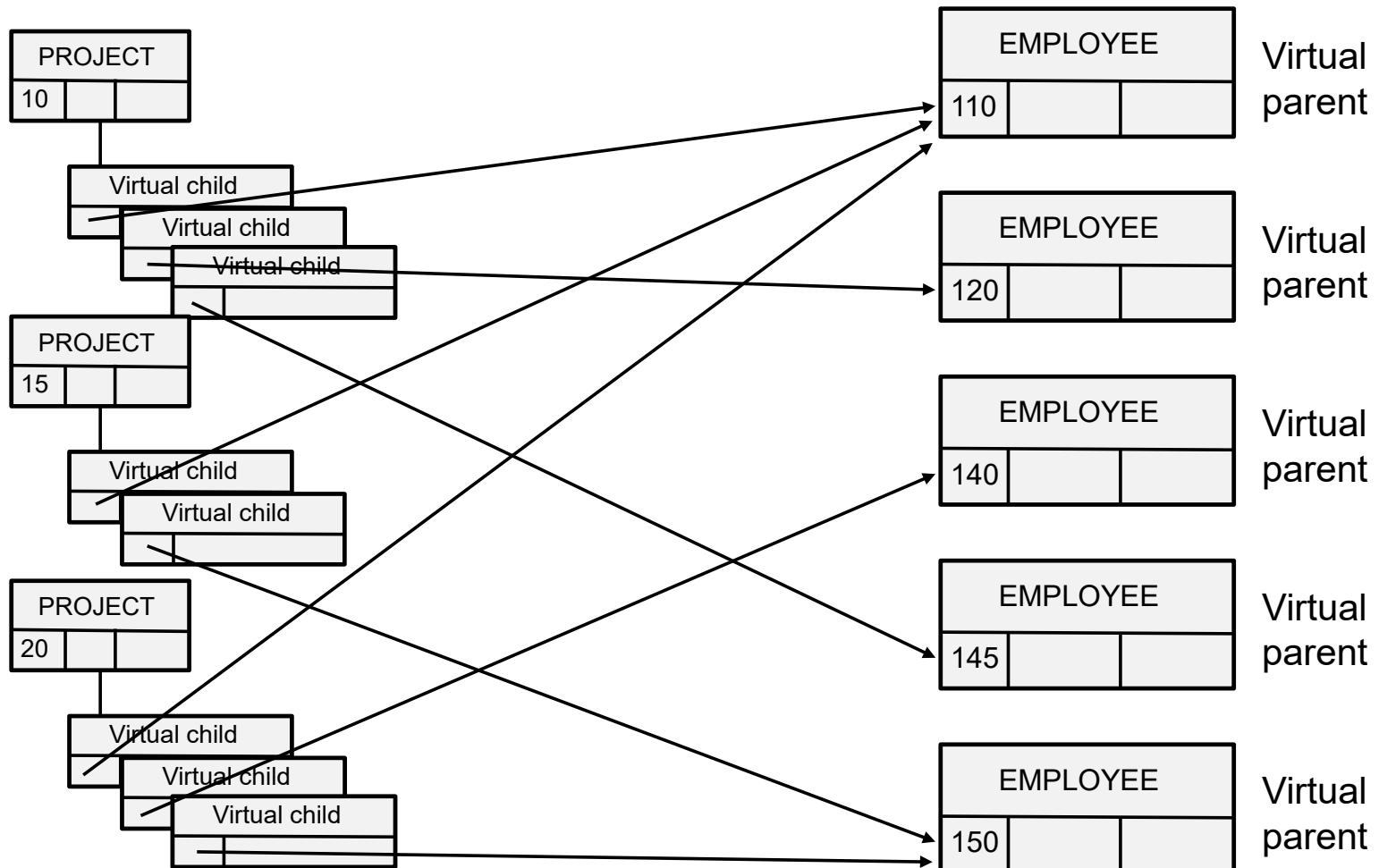


## Hierarchical Model

- Another option for an N:M relationship type is to create two hierarchical structures and connect them using a virtual child record type and a virtual parent/child relationship type
  - » pointers can then be used to navigate between both structures
  - » relationship type attributes can be put in the virtual child record type
  - » no more redundancy



# Hierarchical Model





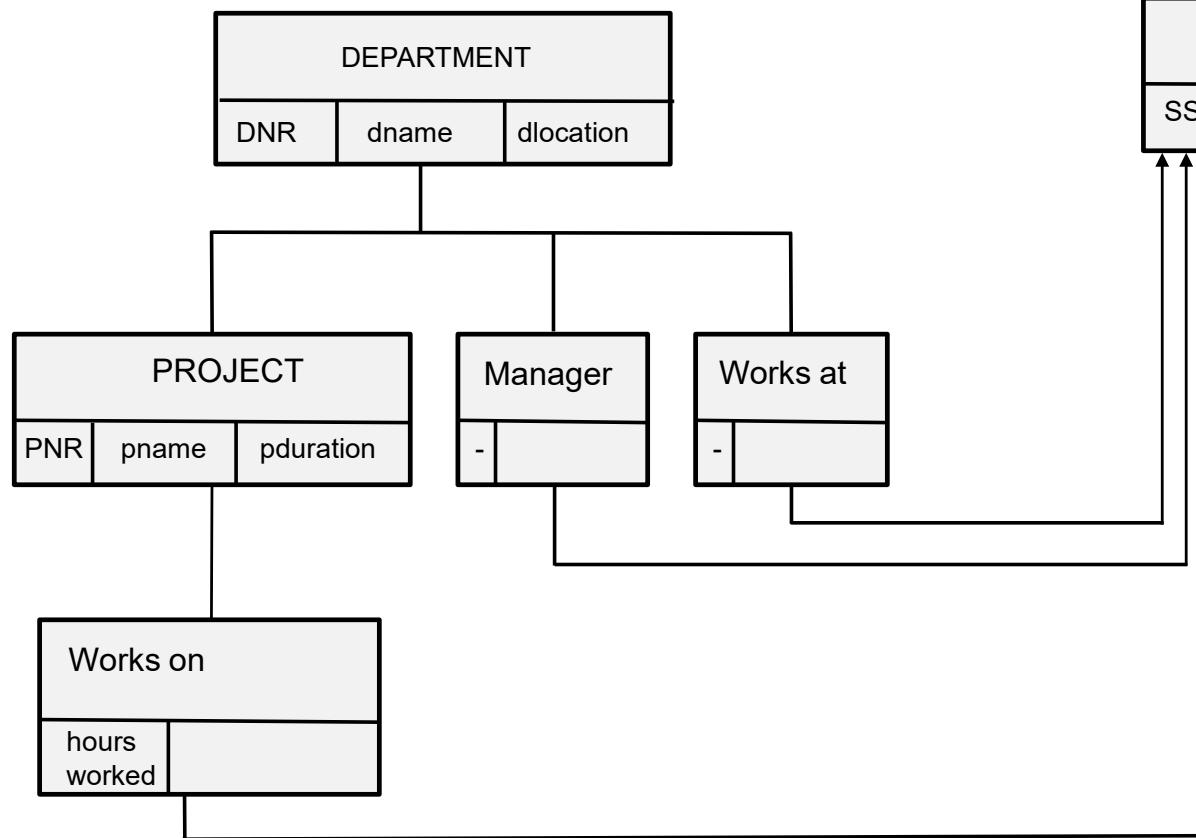
## Hierarchical Model

- 1:1 relationship types should be implemented in application programs
- The hierarchical model only allows relationship types of degree 2
  - » Recursive relationship types or relationship types with more than 2 record types need to be implemented using virtual child record types
- A child can not be disconnected from its parent (on delete cascade)

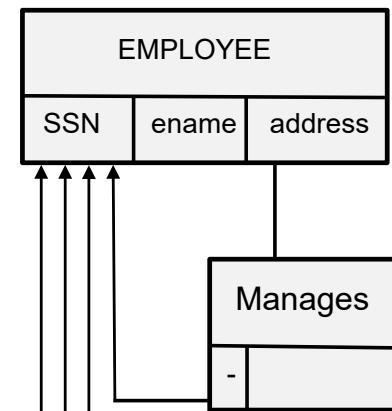


# Hierarchical Model

Hierarchical structure 1



Hierarchical structure 2





## ■ Model limitations

- » no guarantee that each department has exactly 1 manager
- » no guarantee that a department has at least 1 employee



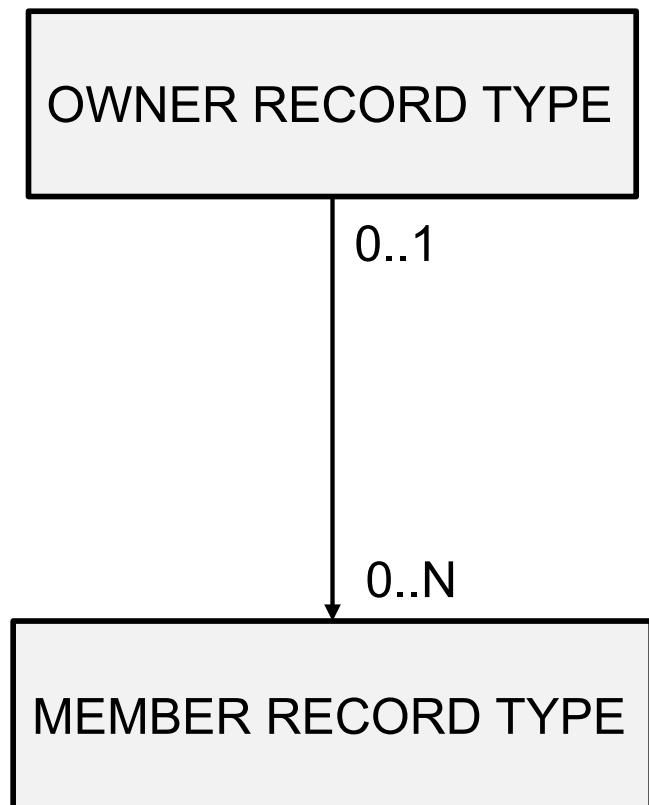
- The CODASYL model was developed by the Data Base Task Group of the COnference on DAta SYstem Languages in 1969
- CA-IDMS (Computer Associates)
- Building blocks
  - » record types
  - » set types
- Lots of structural limitations (legacy)



- A record type is a set of records describing similar entities and has 0, 1 or more records or record occurrences
- A record type consists of various data items
- A vector is a multivalued attribute type
  - » Example: e-mail address
- A repeated group is a composite data item for which a record can have multiple values or a composite multivalued attribute type
  - » Example: address



- A set type models a 1:N relationship type between an owner record type and a member record type
- A set occurrence has 1 owner record and 0, 1 or more member records
- A CODASYL set has both owner and member records and it is also possible to order the member records ( $\leftrightarrow$  mathematical set)

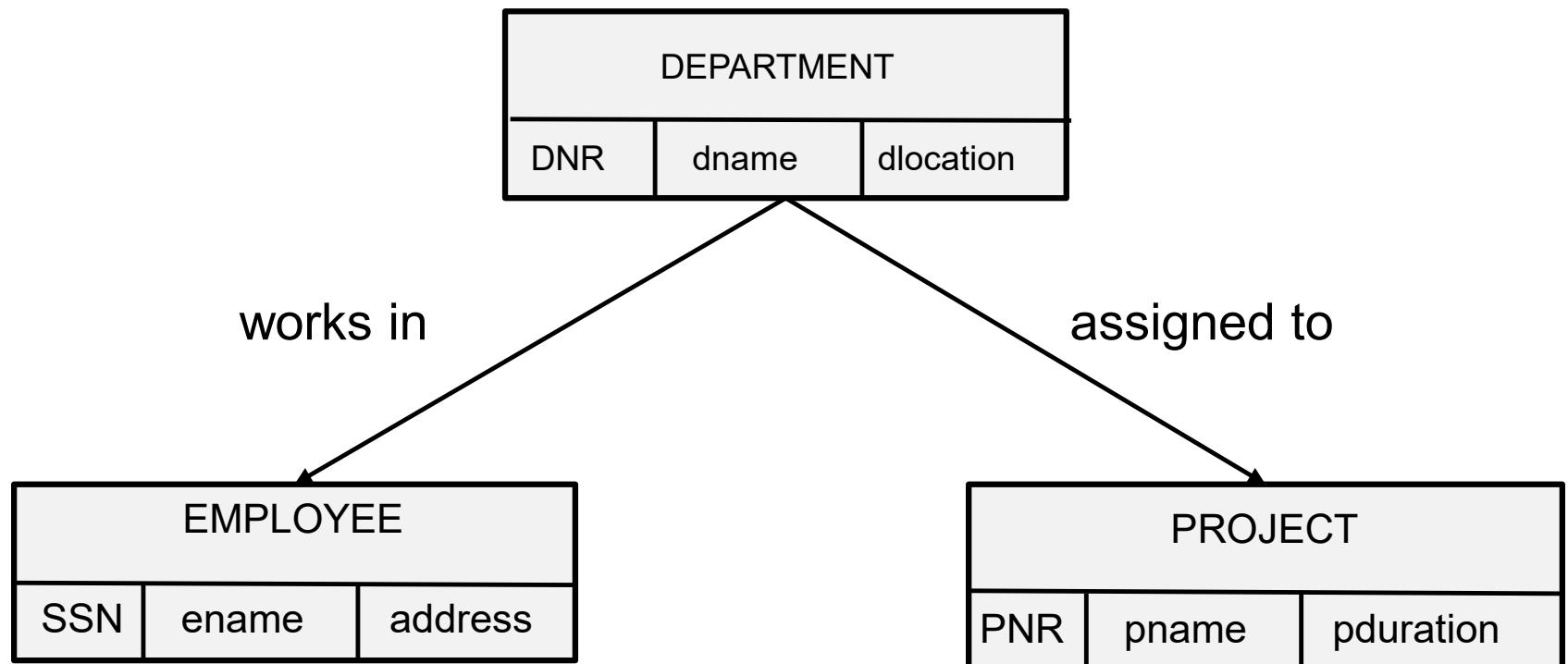




- A member record can exist without being connected to an owner record
- A record type can be a member record type in multiple set types (network structures)
- Multiple set types may be defined between the same record types



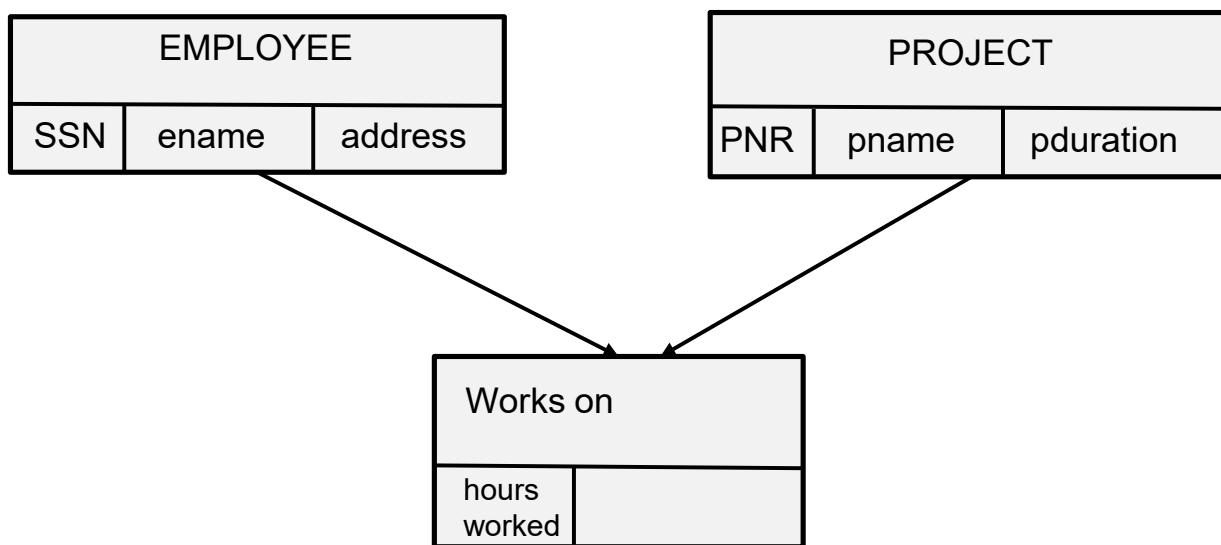
# Bachmann diagram

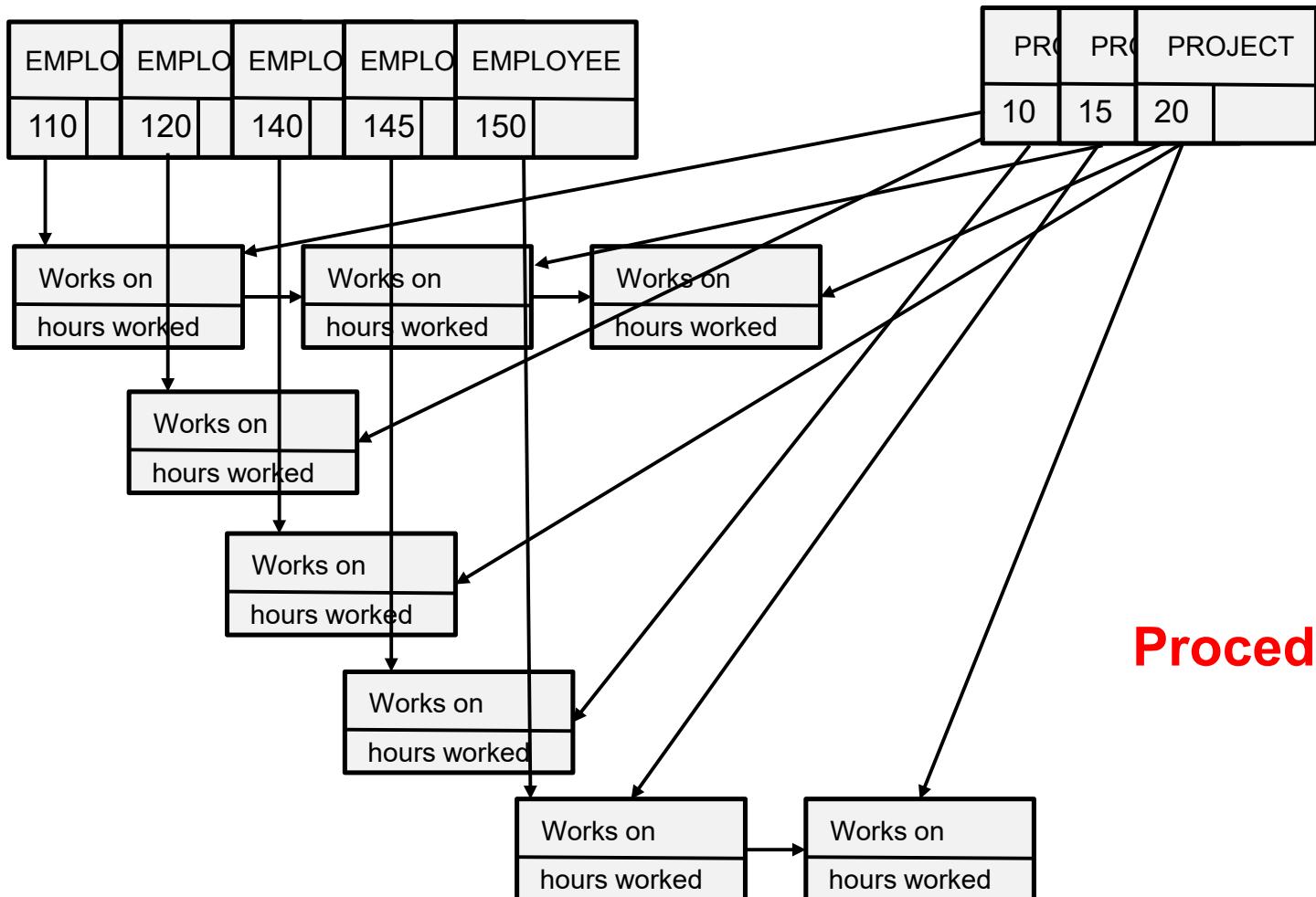




# CODASYL Model

- 1:1 relationship types must be enforced in the application program
- N:M relationship types
  - » introduce a dummy record type as a member record type in 2 set types having as owners the record types of the original N:M relationship type



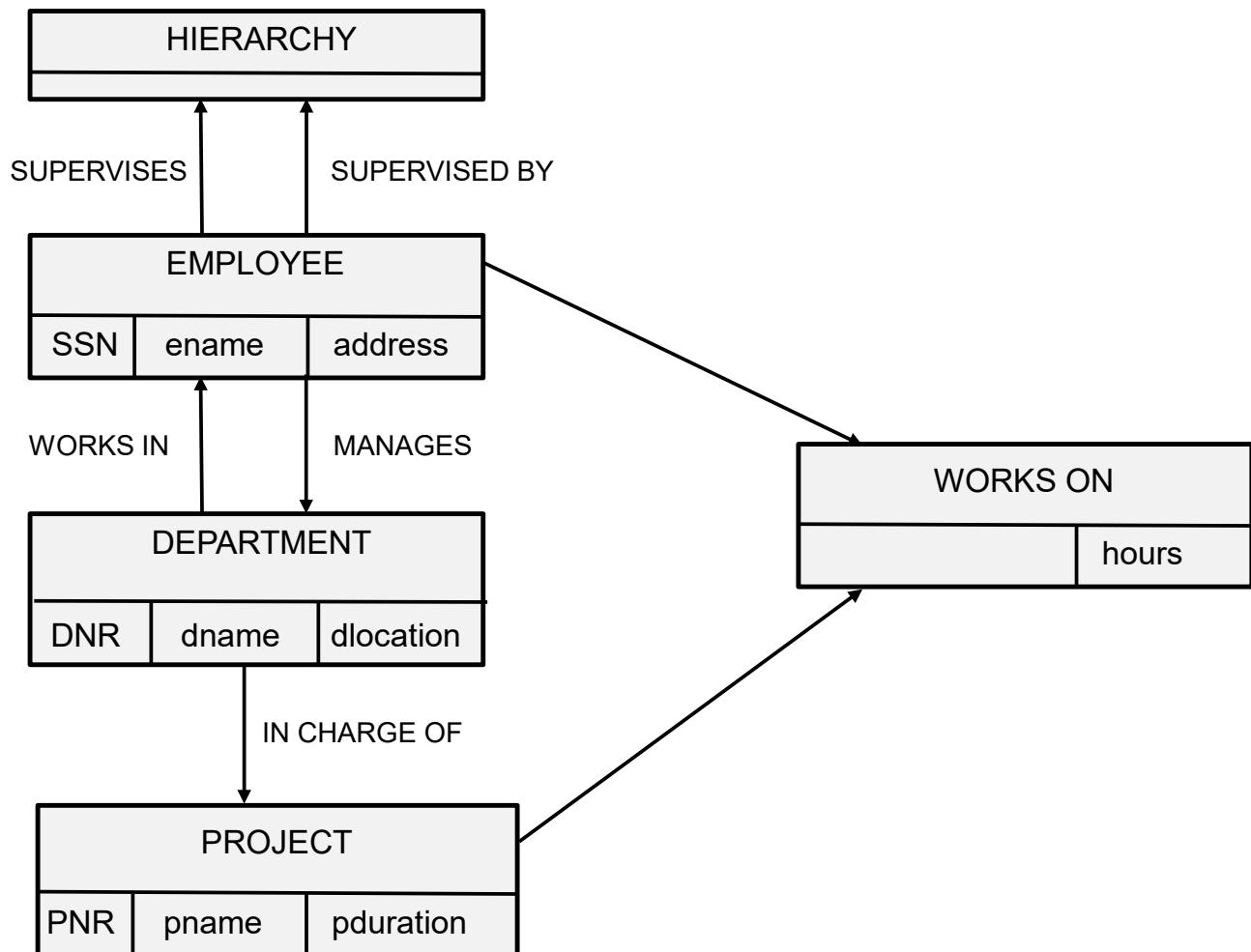


**Procedural DML!**

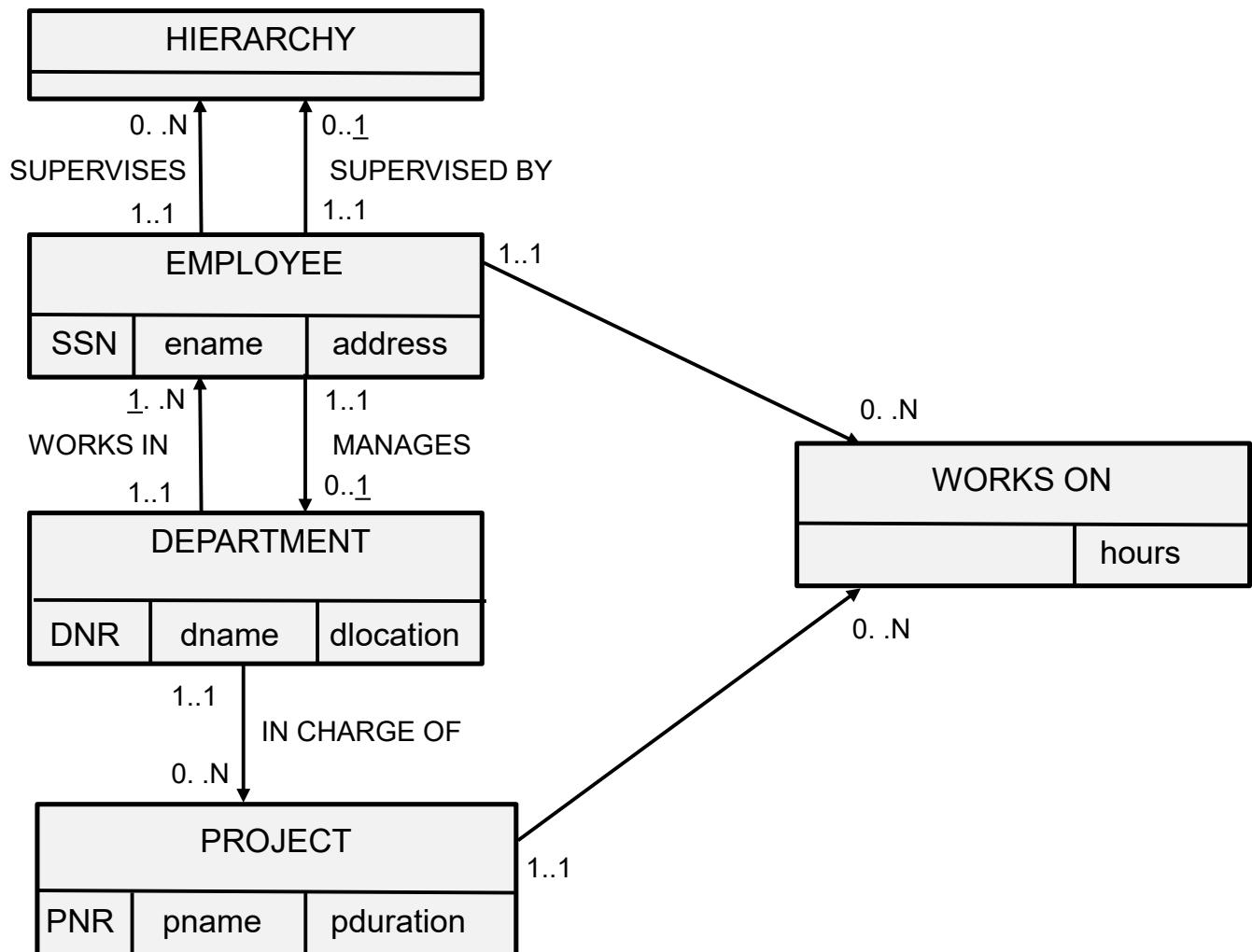


- CODASYL allows to logically order the member records of a set (e.g., alphabetically)
- System can act as the owner for the root record type (singular or system owned set type)
- No support for recursive set types
  - » dummy record type needs to be introduced
- No set types with more than 2 participating record types

# CODASYL Model



# CODASYL Model





## ■ Model limitations

- » an employee can be managed by multiple employees
- » no guarantee that a department must have exactly 1 manager
- » no guarantee that a department has minimal one employee

# Agenda

- 
- 1 Session Overview
  - 2 Legacy Databases
  - 3 OODBs and Object Persistence
  - 4 Extended Relational Databases
  - 5 XML Databases
  - 6 NoSQL Databases
  - 7 Summary and Conclusion

# Agenda

- OODBs and Object Persistence
  - Recap: Basic Concepts of OO
  - Advanced Concepts of OO
  - Basic Principles of Object Persistence
  - OODBMS
  - Evaluating OODBMSs



## Recap: Basic Concepts of OO

- Object is an instance of a class
- Class contains a blueprint description of all the object's characteristics
- Object bundles both variables (which determine its state) and methods (which determine its behavior) in a coherent way



# Recap: Basic Concepts of OO

```
public class Employee {  
  
    private int EmployeeID;  
    private String Name;  
    private String Gender;  
    private Department Dep;  
  
    public int getEmployeeID()  
    {  
        return EmployeeID;  
    }  
    public void setEmployeeID(  
        int id ) {  
        this.EmployeeID = id;  
    }  
    public String getName() {  
        return Name; } }
```

```
public void setName( String  
    name ) {  
    this.Name = name;  
}  
public String getGender() {  
    return Gender;  
}  
public void setGender( String  
    gender ) {  
    this.Gender = gender;  
}  
public Department getDep() {  
    return Dep;  
}  
public void setDep(Department  
    dep) {this.Dep = dep;}}
```



## Recap: Basic Concepts of OO

- Getter and setter methods implement the concept of information hiding (aka encapsulation)
- Encapsulation enforces a strict separation between interface and implementation.
  - » interface consists of the signatures of the methods.
  - » implementation is based upon the object's variables and method definitions



# Recap: Basic Concepts of OO

```
public class EmployeeProgram {  
    public static void main(String[] args) {  
        Employee Bart = new Employee();  
        Employee Seppe = new Employee();  
        Employee Wilfried = new Employee();  
        Bart.setName("Bart Baesens");  
        Seppe.setName("Seppe vanden Broucke");  
        Wilfried.setName("Wilfried Lemahieu");  
    }  
}
```



- Method overloading
- Inheritance
- Method overriding
- Polymorphism
- Dynamic binding



- Method overloading refers to using the same name for more than one method in the same class.
- OO language environment can then determine which method you are calling, provided the number or type of parameters is different in each method



# Advanced Concepts of OO

```
public class Book {  
    String title;  
    String author;  
    boolean isRead;  
    int numberOfReadings;  
  
    public void read(){  
        isRead = true;  
        numberOfReadings++;  
    }  
}
```

```
        public void read(int i){  
            isRead = true;  
            numberOfReadings +=  
                i;  
        }  
    }
```

read(1) same effect as read()

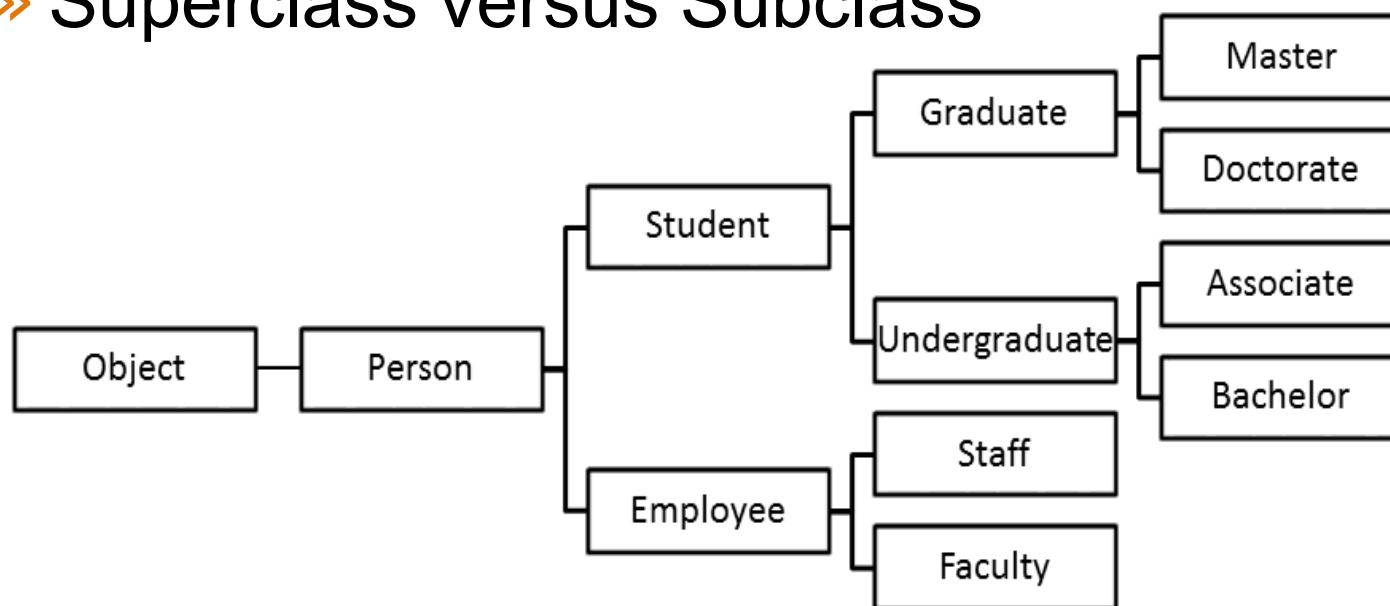


- Method overloading is a handy feature when defining constructors for a class
- A **constructor** is a method which returns an object of a class
- Examples:
  - » `Student(String name, int year, int month, int day)`
  - » `Student(String name)`



# Advanced Concepts of OO

- Inheritance represents an “is a” relationship
  - » E.g. Student and Employee inherit from Person
  - » Superclass versus Subclass





# Advanced Concepts of OO

```
public class Person {  
    private String name;  
  
    public Person(String name){  
        this.setName(name);  
    }  
    public String getName(){  
        return this.name;  
    }  
    public void setName(String name){  
        this.name = name;  
    }  
}  
  
public class Employee extends Person {  
    private Employee manager;  
    private int id;  
  
    public Employee(String name, Employee manager,  
        int empID) {  
        super(name);  
        this.setManager(manager);  
        this.setEmployeeID(empID);  
    }  
    public Employee getManager() {  
        return manager;  
    }  
    public void setManager(Employee manager) {  
        this.manager = manager;  
    }  
    public int getEmployeeID() {  
        return id;  
    }  
    private void setEmployeeID(int employeeID) {  
        this.id = employeeID;}}}
```



- Method overriding: subclasses can override an inherited method with a new, specialized implementation



# Advanced Concepts of OO

## Student Class

```
public double calculateGPA() {  
    double sum = 0;  
    int count = 0;  
    for (double grade : this.getGrades()) {  
        sum += grade;  
        count++;  
    }  
    return sum/count;  
}
```

## Graduate Class

```
public double calculateGPA(){  
    double sum = 0;  
    int count = 0;  
    for (double grade : this.getGrades()){  
        if (grade > 80){  
            sum += grade;  
            count++;  
        }  
    }  
    return sum/count;  
}
```



# Advanced Concepts of OO

- Polymorphism refers to the ability of objects to respond differently to the same method
  - » closely related to inheritance
  - » depending on the functionality desired, the OO environment might consider a particular Master object as a Master, a Graduate, a Student, or a Person
- Static binding binds a method to its implementation at compile time
- Dynamic binding binds a method to its appropriate implementation at runtime, based on the object and its class.



# Advanced Concepts of OO

```
public class PersonProgram {  
    public static void main(String[] args){  
        Student john = new Master("John Adams");  
        john.setGrades(0.75,0.82,0.91,0.69,0.79);  
        Student anne = new Associate("Anne Philips");  
        anne.setGrades(0.75,0.82,0.91,0.69,0.79);  
        System.out.println(john.getName() + ": " +  
            john.calculateGPA());  
        System.out.println(anne.getName() + ": " +  
            anne.calculateGPA());  
    }  
}
```

## OUTPUT:

John Adams: 0.865  
Anne Philips: 0.792



# Basic Principles of Object Persistence

- Transient object is only needed during program execution and can be discarded when the program terminates
- Persistent object is an object that should survive program execution
- Persistence strategies:
  - » Persistence by class
  - » Persistence by creation
  - » Persistence by marking
  - » Persistence by inheritance
  - » Persistence by reachability



# Basic Principles of Object Persistence

- **Persistence by class** implies that all objects of a particular class will be made persistent
- **Persistence by creation** is achieved by extending the syntax for creating objects to indicate at compile-time that an object should be made persistent
- **Persistence by marking** implies that all objects will be created as transient. An object can then be marked as persistent during program execution



# Basic Principles of Object Persistence

- **Persistence by inheritance** indicates that the persistence capabilities are inherited from a pre-defined persistent class
- **Persistence by reachability** starts by declaring the root persistent object(s). All objects that are referred to (either directly or indirectly) by the root object(s) will then be made persistent as well.



# Basic Principles of Object Persistence

- Persistence orthogonality
  - » persistence independence: persistence of an object is independent of how a program manipulates it
  - » type orthogonality: all objects can be made persistent, irrespective of their type or size
  - » transitive persistence: refers to persistence by reachability



# Basic Principles of Object Persistence

- Persistent programming languages extend an OO language with a set of class libraries for object persistence
- Serialization translates an object's state into a format that can be stored (for example, in a file) and reconstructed later



# Basic Principles of Object Persistence

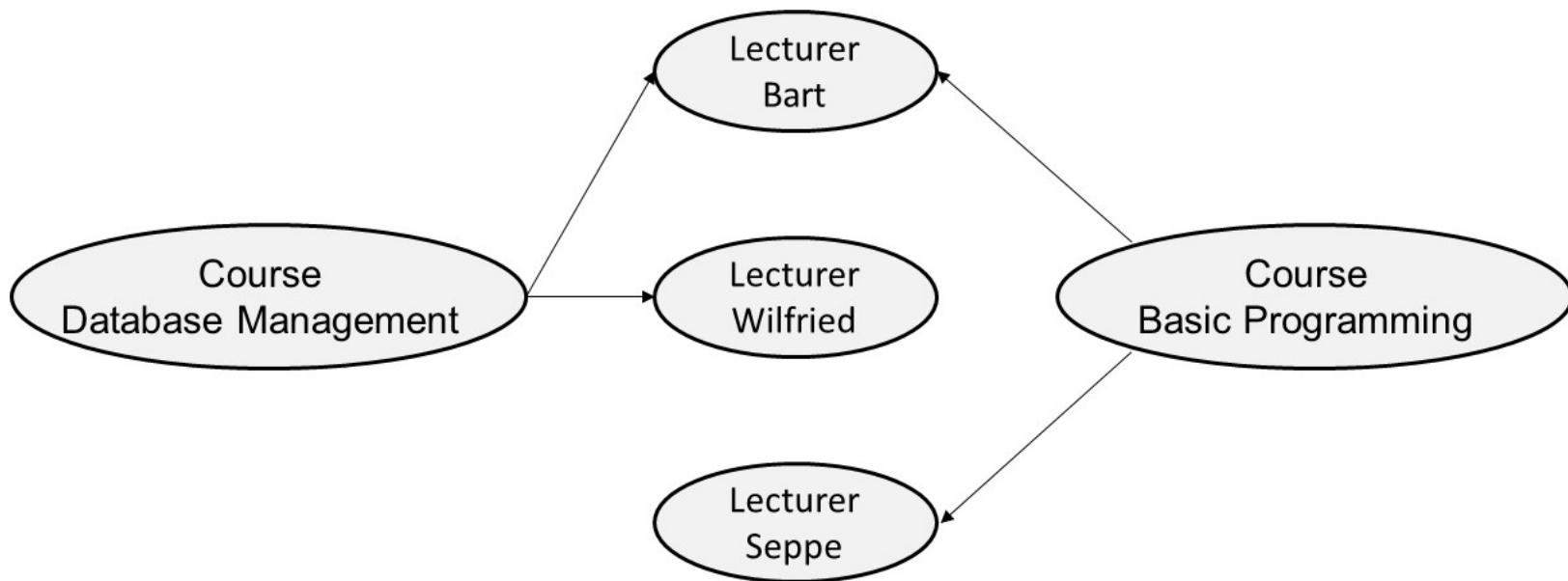
```
public class EmployeeProgram {  
    public static void main(String[] args) {  
        Employee Bart = new Employee();  
        Employee Seppe = new Employee();  
        Employee Wilfried = new Employee();  
        Bart.setName("Bart Baesens");  
        Seppe.setName("Seppe vanden Broucke");  
        Wilfried.setName("Wilfried Lemahieu");  
        try{  
            FileOutputStream fos = new FileOutputStream("myfile.ser");  
            ObjectOutputStream out = new ObjectOutputStream(fos);  
            out.writeObject(Bart);  
            out.writeObject(Seppe);  
            out.writeObject(Wilfried);  
            out.close;  
        }  
        catch (IOException e){e.printStackTrace();}  
    }  
}
```

persistence by reachability!



# Basic Principles of Object Persistence

- Serialization suffers from the same disadvantages of the file based approach
- Lost object identity





- Object-oriented DBMSs (OODBMSs) store persistent objects in a transparent way
- OODBMSs originated as extensions of OO programming languages
- OODBMSs support persistence orthogonality
- OODBMSs guarantee the ACID properties



- Every object has a unique and immutable object identifier (OID)
  - » Not dependent upon state of object ( $\leftrightarrow$  primary key)
  - » Unique within entire OO environment ( $\leftrightarrow$  primary key)
  - » Invisible to the user ( $\leftrightarrow$  primary key)
- OIDs are used to identify objects and to create and manage references between objects
- OO model is often referred to as an identity-based model
  - » Relational model: value based model



- Two objects are said to be **equal** when the values of their variables are the same (object equality)
  - » Shallow versus deep equality
- Two objects are said to be **identical** or equivalent when their OIDs are the same (object identity)

- The Object Database Management Group (ODMG) was formed in 1991 by a group of OO database vendors
  - » Changed to Object Management Group (OMG) in 1998
- Promote portability and interoperability for object persistence by introducing a DDL and DML similar to SQL
- only one language for dealing with both transient and persistent objects

- OMG introduced 5 standards( most recent ODMG 3.0 in 2000 ) with following components:
  - » **Object Model**: provides a standard object model for OODBMS
  - » **Object Definition Language (ODL)**: specifies object definitions (classes and interfaces)
  - » **Object Query Language (OQL)**: allows to define SELECT queries
  - » **Language Bindings** (e.g., for C++, Smalltalk and Java): retrieve and manipulate object data.



- **Object Model** provides a common model to define classes, variables or attributes, behavior and object persistence.
- Two basic building blocks are objects and literals
- A **literal** does not have an OID and cannot exist on its own ( $\leftrightarrow$  an object)
- Types of literals: atomic, collection, structured



- **Atomic literals:** short (short integer), long (long integer), double (real number), float (real number), boolean (true or false), char, and string
- **Collection literals:**
  - » Set: unordered collection of elements without duplicates
  - » Bag: unordered collection of elements which may contain duplicates
  - » List: ordered collection of elements
  - » Array: ordered collection of elements which is indexed
  - » Dictionary: unordered sequence of key-value pairs without duplicates



- A **structured literal** consists of a fixed number of named elements
- E.g., Date, Interval, Time andTimeStamp

```
struct Address{  
    string street;  
    integer number;  
    integer zipcode;  
    string city;  
    string state;  
    string country;  
};
```



- **Object Definition Language (ODL)** is a DDL to define the object types that conform to the ODMG Object Model



```
class EMPLOYEE
(extent employees
key SSN)
{
attribute string SSN;
attribute string ENAME;
attribute struct ADDRESS;
attribute enum GENDER {male, female};
attribute date DATE_OF_BIRTH;
relationship set<EMPLOYEE> supervises
inverse EMPLOYEE:: supervised_by;
relationship EMPLOYEE supervised_by
inverse EMPLOYEE:: supervises;
relationship DEPARTMENT works_in
inverse DEPARTMENT:: workers;
relationship set<PROJECT> has_projects
inverse PROJECT:: has_employees;
string GET_SSN();
void SET_SSN(in string new_ssn);}
...
```



```
class MANAGER extends EMPLOYEE
(extent managers)
{
attribute date mgrdate;
relationship DEPARTMENT manages
inverse DEPARTMENT:: managed_by
}
```

```
class DEPARTMENT
(extent departments
key DNR)
{
attribute string DNR;
attribute string DNAME;
attribute set<string> DLOCATION;
relationship set<EMPLOYEE> workers
inverse EMPLOYEE:: works_in;
relationship set<PROJECT>
assigned_to_projects
inverse PROJECT:: assigned_to_department
relationship MANAGER managed_by
inverse MANAGER:: manages;
string GET_DNR();
void SET_DNR(in string new_dnr);
...}
```



```
class PROJECT
(extent projects
key PNR)
{
attribute string PNR;
attribute string PNAME;
attribute string PDURATION;
relationship DEPARTMENT assigned_to_department
inverse DEPARTMENT:: assigned_to_projects;
relationship SET<EMPLOYEE> has_employees
inverse EMPLOYEE:: has_projects;
string GET_PNR();
void SET_PNR(in string new_pnr);
```



- A class is defined using the keyword **class**
- The **extent** of a class is the set of all current objects of the class
- A variable is declared using the keyword **attribute**
- Operations or methods can be defined by their name followed by parentheses
  - » keywords **in**, **out**, and **inout** are used to define the input, output and input/output parameters
- **extends** keyword indicates the inheritance relationship



- Relationships can be defined using the keyword **relationship**.
- Only unary and binary relationships with cardinalities of 1:1, 1:N, or N:M are supported in ODMG.
- Ternary (or higher) relationships and relationship attributes need to be decomposed by introducing extra classes and relationships.



- Every relationship is defined in a bidirectional way, using the keyword **inverse**

```
relationship DEPARTMENT works_in  
inverse DEPARTMENT:: workers;
```

```
relationship set<EMPLOYEE>  
workers
```

```
inverse EMPLOYEE:: works_in;
```



- N:M relationship can be implemented by defining collection types (e.g. set, bag)

```
relationship set<PROJECT> has_projects  
inverse PROJECT:: has_employees;
```

```
relationship SET<EMPLOYEE> has_employees  
inverse EMPLOYEE:: has_projects;
```



- **Object Query Language (OQL)** is a declarative, non-procedural query language
- OQL can be used for both navigational (procedural) as well as associative (declarative) access



- A **navigational query** explicitly navigates from one object to another

Bart.DATE\_OF\_BIRTH

Bart.ADDRESS

Bart.ADDRESS.CITY



- An **associative query** returns a collection (e.g., a set or bag) of objects which are located by the OODBMS.

## Employees



- **SELECT... FROM ... WHERE OQL** queries
- OQL query returns a bag

```
SELECT e.SSN, e.ENAME, e.ADDRESS, e.GENDER
FROM employees e
WHERE e.name=“Bart Baesens”
```



```
SELECT e.SSN, e.ENAME, e.ADDRESS,  
e.GENDER, e.age  
FROM employees e  
WHERE e.name=“Bart Baesens”
```

```
SELECT e  
FROM employees e  
WHERE e.age > 40
```



## ■ OQL join queries

```
SELECT e.SSN, e.ENAME, e.ADDRESS, e.GENDER, e.age  
FROM employees e, e.works_in d  
WHERE d.DNAME="ICT"
```

```
SELECT e1.ENAME, e1.age, d.DNAME, e2.ENAME, e2.age  
FROM employees e1, e1.works_in d, d.managed_by e2  
WHERE e1.age > e2.age
```



count(employees)

```
SELECT e.SSN, e.ENAME  
FROM employees e  
WHERE EXISTS e IN (SELECT x FROM  
projects p WHERE p.has_employees x)
```

```
SELECT e.SSN, e.ENAME, e.salary  
FROM employees e
```



- ODMG language bindings provide implementations for the ODL and OQL specifications in popular OO programming languages (e.g. C++, Smalltalk or Java)
- Object Manipulation Language (OML) is kept language-specific
- E.g., for the Java language binding, this entails that Java's type system will also be used by the OODBMS, that the Java language syntax is respected and that the OODBMS should handle management aspects based on Java's object semantics



# Evaluating OODBMSs

- Complex objects and relationships are stored in a transparent way (no impedance mismatch!)
- Success of OODBMSs has been limited to niche applications
  - » E.g., processing of scientific data sets by CERN
- Disadvantages
  - » the (ad-hoc) query formulation and optimization procedures
  - » robustness, security, scalability and fault-tolerance
  - » no transparent implementation of the 3 layer database architecture (e.g. views)



# Evaluating OODBMSs

- Most mainstream database applications will, however, typically be built using an OO programming language in combination with an RDBMS
- **Object Relational Mapping (ORM)** framework is used as middleware to facilitate the communication between both environments: OO host language and RDBMS

# Agenda

1 Session Overview

2 Legacy Databases

3 OODBs and Object Persistence

4 Extended Relational Databases

5 XML Databases

6 NoSQL Databases

7 Summary and Conclusion



# Agenda

- Extended Relational Databases
  - Success of the relational model
  - Limitations of the Relational Model
  - Active RDBMS Extensions
  - Object-Relational RDBMS extensions
  - Recursive SQL queries



## Success of the Relational Model

- Relational model: tuples and relations
- The relational model requires all relations to be normalized
- The relational model is also referred to as a value based model ( $\leftrightarrow$  identity based OO model)
- SQL is an easy to learn, descriptive and non-navigational data manipulation language (DML)



# Limitations of the Relational Model

- Complex objects are difficult to handle
- Due to the normalization, the relational model has a flat structure
  - » expensive joins needed to de-fragment the data before it can be successfully used
- Specialization, categorization and aggregation cannot be directly supported



# Limitations of the Relational Model

- Only two type constructors: tuple constructor and set constructor
- Tuple constructor can only be used on atomic values
- Set constructor can only be used on tuples
- Both constructors are not orthogonal
- Not possible to model behavior or store functions
- Poor support for audio, video, text



# Active RDBMS Extensions

- Traditional RDBMSs are **passive**
- Modern day RDBMSs are **active**
- Triggers and stored procedures



- A **trigger** is a piece of SQL code consisting of declarative and/or procedural instructions and stored in the catalog of the RDBMS
- Triggers can also reference attribute types in other tables



# Triggers

EMPLOYEE(SSN, ENAME, SALARY, BONUS,  
JOBCODE, *DNR*)

DEPARTMENT(DNR, DNAME, TOTAL-SALARY,  
*MGNR*)

```
CREATE TRIGGER SALARYTOTAL  
AFTER INSERT ON EMPLOYEE  
FOR EACH ROW  
WHEN (NEW.DNR IS NOT NULL)  
UPDATE DEPARTMENT  
SET TOTAL-SALARY = TOTAL-SALARY + NEW.SALARY  
WHERE DNR = NEW.DNR
```

After Trigger!



# Triggers

**WAGE(JOBCODE, BASE\_SALARY, BASE\_BONUS)**

```
CREATE TRIGGER WAGEDEFAULT
BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AS NEWROW
FOR EACH ROW
SET (SALARY, BONUS) =
(SELECT BASE_SALARY, BASE_BONUS
FROM WAGE
WHERE JOBCODE = NEWROW.JOBCODE)
```

Before Trigger!



- Advantages
  - » Automatic monitoring and verification in case of specific events or situations
  - » Modelling extra semantics and/or integrity rules without changing the user front-end or application code
  - » Assign default values to attribute types for new tuples
  - » Synchronic updates in case of data replication;
  - » Automatic auditing and logging which may be hard to accomplish in any other application layer
  - » Automatic exporting of data



## ■ Disadvantages

- » Hidden functionality, which may be hard to follow-up and manage
- » Cascade effects leading up to an infinite loop of a trigger triggering another trigger etc.
- » Uncertain outcomes if multiple triggers for the same database object and event are defined
- » Deadlock situations
- » Debugging complexities since they don't reside in an application environment
- » Maintainability and performance problems



# Stored Procedures

- A **stored procedure** is a piece of SQL code consisting of declarative and/or procedural instructions and stored in the catalog of the RDBMS
- Needs to be invoked explicitly



# Stored Procedures

```
CREATE PROCEDURE REMOVE-EMPLOYEES
(DNR-VAR IN CHAR(4), JOBCODE-VAR IN CHAR(6)) AS
BEGIN
DELETE FROM EMPLOYEE
WHERE DNR = DNR-VAR AND JOBCODE = JOBCODE-VAR;
END
```

```
import java.sql.CallableStatement;
...
CallableStatement cStmt = conn.prepareCall("{call REMOVE-
EMPLOYEES(?, ?)}");
cStmt.setString(1, "D112");
cStmt.setString(2, "JOB124");
```



## ■ Advantages

- » Similar to OODBMSs, they store behavior in the database
- » They can reduce network traffic
- » They can be implemented in an application-independent way
- » They improve data and functional independence
- » They can be used as a container for several SQL instructions that logically belong together
- » They are easier to debug in comparison to triggers



# Object-Relational RDBMS extensions

- OODBMSs are perceived as very complex to work with
  - » No good standard DML (e.g. SQL)
  - » Lack of a transparent 3-layer database architecture
- Object-Relational DBMSs (ORDBMSs) keep the relation as the fundamental building block and SQL as the core DDL/DML, but with the following OO extensions:
  - » User-Defined Types (UDTs)
  - » User-Defined Functions (UDFs)
  - » Inheritance
  - » Behavior
  - » Polymorphism
  - » Collection types
  - » Large objects (LOBs)



# User-Defined Types (UDTs)

- Standard SQL: CHAR, VARCHAR, INT, FLOAT, DOUBLE, DATE, TIME, BOOLEAN, etc.
- **User-Defined Types (UDT)** define customized data types with specific properties
- Five types:
  - » Distinct data types: extend existing SQL data types
  - » Opaque data types: define entirely new data types
  - » Unnamed row types: use unnamed tuples as attribute values
  - » Named row types: use named tuples as attribute values
  - » Table data types: define tables as instances of table types



# Distinct data types

- A **distinct data type** is a user-defined data type which specializes a standard, built-in SQL data type.

```
CREATE DISTINCT TYPE US-DOLLAR AS DECIMAL(8,2)
```

```
CREATE DISTINCT TYPE EURO AS DECIMAL(8,2)
```

```
CREATE TABLE ACCOUNT  
(ACCOUNTNO SMALLINT PRIMARY KEY NOT NULL,
```

...

```
AMOUNT-IN-DOLLAR US-DOLLAR,  
AMOUNT-IN-EURO EURO)
```



# Distinct data types

- Once a distinct data type has been defined, the ORDBMS will automatically create two casting functions

```
SELECT *
FROM ACCOUNT
WHERE AMOUNT-IN-EURO > 1000
```

**ERROR!**

```
SELECT *
FROM ACCOUNT
WHERE AMOUNT-IN-EURO > EURO(1000)
```



## Opaque data types

- An **opaque data type** is an entirely new, user-defined data type, not based upon any existing SQL data type.
- Examples: data types for image, audio, video, fingerprints, text, spatial data, RFID tags, QR codes, etc.



# Opaque data types

```
CREATE OPAQUE TYPE IMAGE AS <...>
```

```
CREATE OPAQUE TYPE FINGERPRINT AS <...>
```

```
CREATE TABLE EMPLOYEE  
(SSN SMALLINT NOT NULL,  
FNAME CHAR(25) NOT NULL,  
LNAME CHAR(25) NOT NULL,  
...  
EMPFINGERPRINT FINGERPRINT,  
PHOTOGRAPH IMAGE)
```



# Unnamed Row Types

- An **unnamed row type** includes a composite data type in a table by using the keyword ROW.
- It consists of a combination of data types such as built-in types, distinct types, opaque types, etc.

```
CREATE TABLE EMPLOYEE
  (SSN SMALLINT NOT NULL,
   NAME ROW(FNAME CHAR(25), LNAME CHAR(25)),
   ADDRESS ROW(
     STREET ADDRESS CHAR(20) NOT NULL,
     ZIP CODE CHAR(8),
     CITY CHAR(15) NOT NULL),
   ...
   EMPFINGERPRINT FINGERPRINT,
   PHOTOGRAPH IMAGE)
```



## Named row types

- A **named row type** is a user-defined data type which groups a coherent set of data types into a new composite data type and assigns a meaningful name to it
- can be used in table definitions, queries, or anywhere else a standard SQL data type can be used
- Note: the usage of (un)named row types implies the end of the first normal form!



# Named row types

```
CREATE ROW TYPE ADDRESS AS  
(STREET ADDRESS CHAR(20) NOT NULL,  
ZIP CODE CHAR(8),  
CITY CHAR(15) NOT NULL)
```

```
CREATE TABLE EMPLOYEE  
(SSN SMALLINT NOT NULL,  
FNAME CHAR(25) NOT NULL,  
LNAME CHAR(25) NOT NULL,  
EMPADDRESS ADDRESS,  
...  
EMPFINGERPRINT FINGERPRINT,  
PHOTOGRAPH IMAGE)
```



## Named row types

```
SELECT LNAME, EMPADDRESS  
FROM EMPLOYEE  
WHERE EMPADDRESS.CITY = 'LEUVEN'
```

```
SELECT E1.LNAME, E1.EMPADDRESS  
FROM EMPLOYEE E1, EMPLOYEE E2  
WHERE E1.EMPADDRESS.CITY =  
E2.EMPADDRESS.CITY  
AND E2.SSN = '123456789'
```



# Table data types

- A **table data type** (or typed table) defines the type of a table.
  - » Similar to a class in OO

```
CREATE TYPE EMPLOYEE_TYPE  
  (SSN SMALLINT NOT NULL,  
   FNAME CHAR(25) NOT NULL,  
   LNAME CHAR(25) NOT NULL,  
   EMPADDRESS ADDRESS  
  
   ...  
   EMPFINGERPRINT FINGERPRINT,  
   PHOTOGRAPH IMAGE)
```



## Table data types

```
CREATE TABLE EMPLOYEE OF TYPE  
EMPLOYEEETYPE PRIMARY KEY (SSN)
```

```
CREATE TABLE EX-EMPLOYEE OF TYPE  
EMPLOYEEETYPE PRIMARY KEY (SSN)
```



## Table data types

```
CREATE TYPE DEPARTMENTTYPE  
(DNR SMALLINT NOT NULL,  
DNAME CHAR(25) NOT NULL,  
DLOCATION ADDRESS  
MANAGER REF(EMPLOYEEETYPE))
```

Note: reference can be replaced by the actual data it refers to by means of the DEREF (from dereferencing) function.



# User-Defined Functions (UDFs)

- Every RDBMS comes with a set of built-in functions, e.g., MIN(), MAX(), AVG(), etc.
- **User-Defined Functions (UDFs)** allow users to extend these by explicitly defining their own functions
- Every UDF consists of
  - » name
  - » input and output arguments
  - » implementation



# User-Defined Functions (UDFs)

- UDFs are stored in the ORDBMS and hidden from the applications
- UDFs can be overloaded
- Types
  - » sourced functions
  - » external functions



## Sourced function

- UDF which is based on an existing, built-in function

```
CREATE DISTINCT TYPE MONETARY AS DECIMAL(8,2)
```

```
CREATE TABLE EMPLOYEE  
  (SSN SMALLINT NOT NULL,  
   FNAME CHAR(25) NOT NULL,  
   LNAME CHAR(25) NOT NULL,  
   EMPADDRESS ADDRESS,  
   SALARY MONETARY,  
   ...  
   EMPFINGERPRINT FINGERPRINT,  
   PHOTOGRAPH IMAGE)
```



## Sourced function

```
CREATE FUNCTION AVG(MONETARY)
RETURNS MONETARY
SOURCE AVG(DECIMAL(8,2))
```

```
SELECT DNR, AVG(SALARY)
FROM EMPLOYEE
GROUP BY DNR
```



# External functions

- **External functions** are written in an external host language
  - » Python, C, Java, etc.
- Can return a single value (scalar) or table of values



- ORDBMS extends an RDBMS by providing explicit support for inheritance, both at
  - » the level of a data type
  - » the level of a typed table



## Inheritance at data type level

- Child data type inherits all the properties of a parent data type and can then further specialize it by adding specific characteristics

```
CREATE ROW TYPE ADDRESS AS
  (STREET ADDRESS CHAR(20) NOT NULL,
   ZIP CODE CHAR(8),
   CITY CHAR(15) NOT NULL)
```

```
CREATE ROW TYPE INTERNATIONAL_ADDRESS AS
  (COUNTRY CHAR(25) NOT NULL) UNDER ADDRESS
```



# Inheritance at data type level

```
CREATE TABLE EMPLOYEE
  (SSN SMALLINT NOT NULL,
  FNAME CHAR(25) NOT NULL,
  LNAME CHAR(25) NOT NULL,
  EMPADDRESS INTERNATIONAL_ADDRESS,
  SALARY MONETARY,
  ...
  EMPFINGERPRINT FINGERPRINT,
  PHOTOGRAPH IMAGE)
```

```
SELECT FNAME, LNAME, EMPADDRESS
FROM EMPLOYEE
WHERE EMPADDRESS.COUNTRY = 'Belgium'
AND EMPADDRESS.CITY LIKE 'Leu%'
```



# Inheritance at Table Type Level

```
CREATE TYPE EMPLOYEETYPE  
  (SSN SMALLINT NOT NULL,  
  FNAME CHAR(25) NOT NULL,  
  LNAME CHAR(25) NOT NULL,  
  EMPADDRESS INTERNATIONAL_ADDRESS  
  ...  
  ...  
  EMPPFINGERPRINT FINGERPRINT,  
  PHOTOGRAPH IMAGE)
```

```
CREATE TYPE ENGINEERTYPE AS  
  (DEGREE CHAR(10) NOT NULL,  
  LICENSE CHAR(20) NOT NULL) UNDER EMPLOYEETYPE
```

```
CREATE TYPE MANAGERTYPE AS  
  (STARTDATE DATE,  
  TITLE CHAR(20)) UNDER EMPLOYEETYPE
```



# Inheritance at Table Type Level

```
CREATE TABLE EMPLOYEE OF TYPE EMPLOYEEETYPE PRIMARY KEY (SSN)
CREATE TABLE ENGINEER OF TYPE ENGINEERTYPE UNDER EMPLOYEE
CREATE TABLE MANAGER OF TYPE MANAGERTYPE UNDER EMPLOYEE
```

```
SELECT SSN, FNAME, LNAME, STARTDATE, TITLE
FROM MANAGER
```

```
SELECT SSN, FNAME, LNAME
FROM EMPLOYEE
```

```
SELECT SSN, FNAME, LNAME
FROM ONLY EMPLOYEE
```



- E.g, triggers, stored procedures or UDFs
- ORDBMS can include the signature or interface of a method in the definitions of data types and tables
  - » Information hiding



```
CREATE TYPE EMPLOYEE_TYPE  
  (SSN SMALLINT NOT NULL,  
   FNAME CHAR(25) NOT NULL,  
   LNAME CHAR(25) NOT NULL,  
   EMPADDRESS INTERNATIONAL_ADDRESS,  
   ...  
   ...  
   EMPFINGERPRINT FINGERPRINT,  
   PHOTOGRAPH IMAGE,  
   FUNCTION AGE(EMPLOYEE_TYPE) RETURNS INTEGER)
```

**CREATE TABLE** EMPLOYEE **OF TYPE** EMPLOYEE\_TYPE **PRIMARY KEY**  
**(SSN)**

**SELECT** SSN, FNAME, LNAME, PHOTOGRAPH  
**FROM** EMPLOYEE  
**WHERE** AGE = 60



# Polymorphism

- Subtype inherits both the attribute types and functions of its supertype
- Subtype can also override functions to provide more specialized implementations
- Polymorphism: same function call can invoke different implementations



# Polymorphism

```
CREATE FUNCTION TOTAL_SALARY(EMPLOYEE E)
RETURNING INT
AS SELECT E.SALARY
```

```
CREATE FUNCTION TOTAL_SALARY(MANAGER M)
RETURNING INT
AS SELECT M.SALARY + <monthly_bonus>
```

```
SELECT TOTAL_SALARY FROM EMPLOYEE
```



# Collection Types

- Can be instantiated as a collection of instances of standard data types or UDTs
- Set: unordered collection, no duplicates
- Multiset or bag: unordered collection, duplicates allowed
- List: ordered collection, duplicates allowed
- Array: ordered and indexed collection, duplicates allowed
- Note: end of the first normal form!



# Collection Types

```
CREATE TYPE EMPLOYEE_TYPE  
  (SSN SMALLINT NOT NULL,  
   FNAME CHAR(25) NOT NULL,  
   LNAME CHAR(25) NOT NULL,  
   EMPADDRESS INTERNATIONAL_ADDRESS,  
   ...  
   EMPFINGERPRINT FINGERPRINT,  
   PHOTOGRAPH IMAGE,  
   TELEPHONE SET (CHAR(12)),  
   FUNCTION AGE(EMPLOYEE_TYPE) RETURNS INTEGER)
```

```
CREATE TABLE EMPLOYEE OF TYPE EMPLOYEE_TYPE (PRIMARY KEY  
  SSN)
```

```
SELECT SSN, FNAME, LNAME  
FROM EMPLOYEE  
WHERE '2123375000' IN (TELEPHONE)
```



# Collection Types

```
SELECT T.TELEPHONE  
FROM THE (SELECT TELEPHONE FROM EMPLOYEE) AS T  
ORDER BY T.TELEPHONE
```



# Collection Types

```
CREATE TYPE DEPARTMENTTYPE AS
  (DNR CHAR(3) NOT NULL,
  DNAME CHAR(25) NOT NULL,
  MANAGER REF(EMPLOYEEETYPE),
  PERSONNEL SET (REF(EMPLOYEEETYPE)))
```

```
CREATE TABLE DEPARTMENT OF TYPE DEPARTMENTTYPE
(PRIMARY KEY DNR)
```

```
SELECT PERSONNEL
FROM DEPARTMENT
WHERE DNR = '123'
```

```
SELECT DEREF(PERSONNEL).FNAME, DEREF(PERSONNEL).LNAME
FROM DEPARTMENT
WHERE DNR = '123'
```

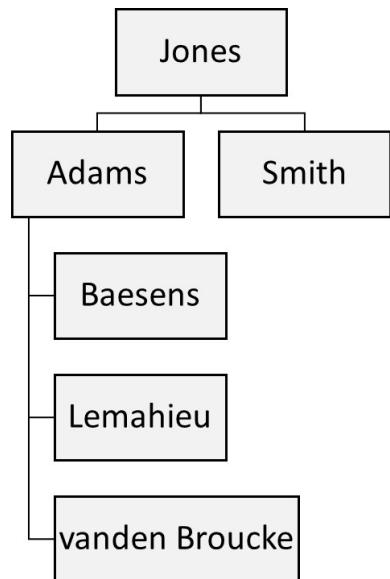


# Large objects

- Multimedia database applications
- LOB data will be stored in a separate table and tablespace
- Types of LOB data:
  - » **BLOB (Binary Large Object)**: a variable-length binary string whose interpretation is left to an external application
  - » **CLOB (Character Large Object)**: variable-length character strings made up of single-byte characters
  - » **DBCLOB (Double Byte Character Large Object)**: variable-length character strings made up of double-byte characters.



# Recursive SQL Queries



- Employee(SSN, Name, Salary, *MNGR*)



# Recursive SQL Queries

<b><u>SSN</u></b>	<b>Name</b>	<b>Salary</b>	<b>MNGR</b>
1	Jones	10.000	NULL
2	Baesens	2.000	3
3	Adams	5.000	1
4	Smith	6.000	1
5	vanden Broucke	3.000	3
6	Lemahieu	2.500	3



# Recursive SQL Queries

```
WITH SUBORDINATES(SSN, NAME, SALARY, MNGR, LEVEL) AS
((SELECT SSN, NAME, SALARY, MNGR, 1
FROM EMPLOYEE
WHERE MNGR=NULL)
UNION ALL
(SELECT E.SSN, E.NAME, E.SALARY, E.MNGR, S.LEVEL+1
FROM SUBORDINATES AS S, EMPLOYEE AS E
WHERE S.SSN=E.MNGR))
```

```
SELECT * FROM SUBORDINATES
ORDER BY LEVEL
```



# Recursive SQL Queries

<u>SS</u>	NAME	SALA	MNG	LEVE
<u>N</u>		RY	R	L
1	Jones	10.000	NULL	1

<u>SS</u>	NAME	SALA	MNG	LEVE
<u>N</u>		RY	R	L
3	Adams	5.000	1	2
4	Smith	6.000	1	2

<u>SS</u>	NAME	SALA	MNG	LEVE
<u>N</u>		RY	R	L
2	Baesens	2.000	3	3
5	vanden Broucke	3.000	3	3
6	Lemahieu	2.500	3	3

<u>SS</u>	NAME	SALA	MNG	LEVE
<u>N</u>		RY	R	L
1	Jones	10.000	NULL	1
3	Adams	5.000	1	2
4	Smith	6.000	1	2
2	Baesens	2.000	3	3
5	vanden Broucke	3.000	3	3
6	Lemahieu	2.500	3	3

# Agenda

1 Session Overview

2 Legacy Databases

3 OODBs and Object Persistence

4 Extended Relational Databases

5 XML Databases

6 NoSQL Databases

7 Summary and Conclusion



# Agenda

- XML Databases
  - Extensible Markup Language
  - Processing XML Documents
  - Storage of XML Documents
  - Differences between XML and Relational Data
  - Mappings Between XML Documents and (Object-) Relational Data
  - Searching XML Data
  - XML for Information Exchange
  - Other Data Representation Formats



# Extensible Markup Language

- Basic Concepts
- Document Type Definitions and XML Schema Definitions
- Extensible Stylesheet Language
- Namespaces
- XPath

# Basic Concepts of XML

- Introduced by the World Wide Web Consortium (W3C) in 1997
- Simplified subset of the Standard Generalized Markup Language (SGML),
- Aimed at storing and exchanging complex, structured documents
- Users can define new tags in XML (↔ HTML)



- Combination of a start tag, content and end tag is called an XML element
- XML is case-sensitive
- Example

```
<author>  
  <name>  
    <first name>Bart</first name>  
    <last name>Baesens</last name>  
  </name>  
</author>
```



# Basic Concepts of XML

- Start tags can contain attribute values

```
<author email="Bart.Baesens@kuleuven.be">Bart Baesens</author>
```

```
<author>
<name>Bart Baesens</name>
<email use="work">Bart.Baesens@kuleuven.be</email>
<email use="private">Bart.Baesens@gmail.com</email>
</author>
```

- Comments are defined as follows

```
<!--This is a comment line -->
```

- Processing instructions are defined as follows

```
<?xml version="1.0" encoding="UTF-8"?>
```



# Basic Concepts of XML

- Self-defined XML tags can be used to describe document structure (↔ HTML)
  - » can be processed in much more detail
- XML formatting rules
  - » only one-root element
  - » start tag should be closed with a matching end tag
  - » no overlapping tag sequence or incorrect nesting



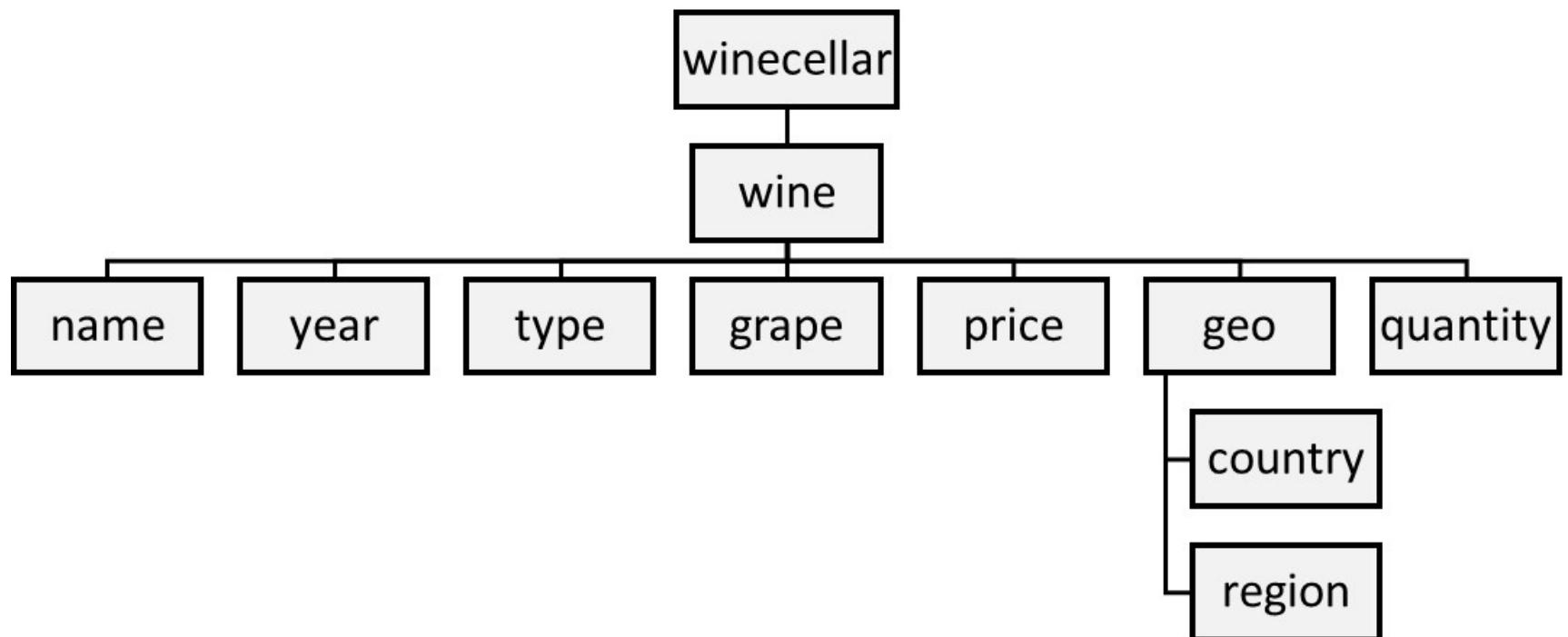
# Basic Concepts of XML

```
<?xml version="1.0" encoding="UTF-8"?>
<winecellar>
  <wine>
    <name>Jacques Selosse Brut Initial</name>
    <year>2012</year>
    <type>Champagne</type>
    <grape percentage="100">Chardonnay</grape>
    <price currency="EURO">150</price>
    <geo>
      <country>France</country>
      <region>Champagne</region>
    </geo>
    <quantity>12</quantity>
  </wine>
</winecellar>
```

```
<wine>
  <name>Meneghetti White</name>
  <year>2010</year>
  <type>white wine</type>
  <grape percentage="80">Chardonnay</grape>
  <grape percentage="20">Pinot Blanc</grape>
  <price currency="EURO">18</price>
  <geo>
    <country>Croatia</country>
    <region>Istria</region>
  </geo>
  <quantity>20</quantity>
</wine>
</winecellar>
```



# Basic Concepts of XML





# Document Type Definitions and XML Schema Definitions

- Document Type Definitions (DTD) and XML Schema Definitions (XSD) specify structure of XML document
- Both define tag set, location of each tag, and nesting
- XML document which complies with DTD or XSD is referred to as valid
- XML document which complies with syntax is referred to as well-formed



- DTD definition for winecellar

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE winecellar [
3.   <!ELEMENT winecellar (wine+)>
4.   <!ELEMENT wine (name, year, type, grape*, price, geo, quantity)>
5.   <!ELEMENT name (#PCDATA)>
6.   <!ELEMENT year (#PCDATA)>
7.   <!ELEMENT type (#PCDATA)>
8.   <!ELEMENT grape (#PCDATA)>
9.   <!ATTLIST grape percentage CDATA #IMPLIED>
10.  <!ELEMENT price (#PCDATA)>
11.  <!ATTLIST price currency CDATA #REQUIRED>
12.  <!ELEMENT geo (country, region)>
13.  <!ELEMENT country (#PCDATA)>
14.  <!ELEMENT region (#PCDATA)>
15.  <!ELEMENT quantity (#PCDATA)>
16. ]>
```



# Document Type Definitions and XML Schema Definitions

- Disadvantages of DTD
  - » only supports character data (no support for integers, dates, complex types)
  - » not defined using XML syntax
- XML Schema supports various data types and user-defined types



# Document Type Definitions and XML Schema Definitions

- XML Schema definition for winecellar

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2. <xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3. <xselement name="winecellar">
4. <xsccomplexType>
5. <xsssequence>
6. <xselement name="wine" maxOccurs="unbounded" minOccurs="0">
7. <xsccomplexType>
8. <xsssequence>
9. <xselement type="xs:string" name="name"/>
10. <xselement type="xs:short" name="year"/>
11. <xselement type="xs:string" name="type"/>
12. <xselement name="grape" maxOccurs="unbounded" minOccurs="1">
13. <xsccomplexType>
14. <xssimpleContent>
15. <xsextension base="xs:string">
16. <xssattribute type="xs:byte" name="percentage" use="optional"/>
17. </xsextension>
18. </xssimpleContent>
```



# Document Type Definitions and XML Schema Definitions

- XML Schema definition for winecellar (contd.)

```
19. </xs:complexType>
20. </xs:element>
21. <xs:element name="price">
22. <xs:complexType>
23. <xs:simpleContent>
24. <xs:extension base="xs:short">
25. <xs:attribute type="xs:string" name="currency" use="optional"/>
26. </xs:extension>
27. </xs:simpleContent>
28. </xs:complexType>
29. </xs:element>
30. <xs:element name="geo">
31. <xs:complexType>
32. <xs:sequence>
33. <xs:element type="xs:string" name="country"/>
34. <xs:element type="xs:string" name="region"/>
35. </xs:sequence>
```



# Document Type Definitions and XML Schema Definitions

- XML Schema definition for winecellar (contd.)

```
36. </xs:complexType>
37. </xs:element>
38. <xs:element type="xs:byte" name="quantity"/>
39. </xs:sequence>
40. </xs:complexType>
41. </xs:element>
42. </xs:sequence>
43. </xs:complexType>
44. </xs:element>
45. </xs:schema>
```



# Extensible Stylesheet Language

- Extensible Stylesheet Language (XSL) can be used to define stylesheet specifying how XML documents can be visualized in a web browser
- XSL encompasses 2 specifications
  - » XSL Transformations (XSLT): transforms XML documents to other XML documents, HTML web pages, or plain text
  - » XSL Formatting Objects (XSL-FO): specify formatting semantics (e.g., transform XML documents to PDFs) but discontinued in 2012
- Decoupling of information content from information visualization



# Extensible Stylesheet Language

- XSLT stylesheet for summary document with only name and quantity of each wine

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3. <xsl:template match='/'>
4. <winecellarsummary>
5. <xsl:for-each select='winecellar/wine'>
6. <wine>
7. <name><xsl:value-of select='name' /></name>
8. <quantity><xsl:value-of select='quantity' /></quantity>
9. </wine>
10. </xsl:for-each>
11. </winecellarsummary>
12. </xsl:template>
13. </xsl:stylesheet>
```



# Extensible Stylesheet Language

```
<?xml version="1.0" encoding="UTF-8"?>
<winecellarsummary>
    <wine>
        <name>Jacques Selosse Brut Initial</name>
        <quantity>12</quantity>
    </wine>
    <wine>
        <name>Meneghetti White</name>
        <quantity>20</quantity>
    </wine>
</winecellarsummary>
```



# Extensible Stylesheet Language

- XSLT stylesheet for transforming XML document to HTML

```
<?xml version="1.0" encoding="UTF-8"?>
<html xsl:version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <body style="font-family:Arial;font-size:12pt;background-
color:#fffff">
<h1>My Wine Cellar</h1>
<table border="1">
    <tr bgcolor="#f2f2f2">
        <th>Wine</th>
        <th>Year</th>
```



# Extensible Stylesheet Language

- XSLT stylesheet for transforming XML document to HTML (contd.)

```
<th>Quantity</th>
  </tr>
  <xsl:for-each select="winecellar/wine">
    <tr>
      <td><xsl:value-of select="name"/></td>
      <td><xsl:value-of select="year"/></td>
      <td><xsl:value-of select="quantity"/></td>
    </tr>
  </xsl:for-each>
</table>
</body>
</html>
```



# Extensible Stylesheet Language

```
<html>
  <body style="font-family:Arial;font-size:12pt;background-color:#fffff">
    <h1>My Wine Cellar</h1>
    <table border="1">
      <tr bgcolor="#f2f2f2">
        <th>Wine</th>
        <th>Year</th>
        <th>Quantity</th>
      </tr>
      <tr>
        <td>Jacques Selosse Brut Initial</td>
        <td>2012</td>
        <td>12</td>
      </tr>
      <tr>
        <td>Meneghetti White</td>
        <td>2010</td>
        <td>20</td>
      </tr>
    </table> </body></html>
```



# Extensible Stylesheet Language

A screenshot of a Windows desktop environment. On the left is a dark taskbar with various icons. In the center is a web browser window titled 'Seppe' showing a page at 'https://www.google.be/'. The page content is a table titled 'My Wine Cellar' with three rows of data:

Wine	Year	Quantity
Jacques Selosse Brut Initial	2012	12
Meneghetti White	2010	20



- To avoid name conflicts, XML introduced concept of a namespace
- Introduce prefixes to XML elements to unambiguously identify their meaning
- Prefixes typically refer to a URI (uniform resource identifier) which uniquely identifies a web resource such as a URL (uniform resource locator)
  - » does not need to refer to physically existing webpage



# Namespaces

```
<winecellar xmlns:Bartns="www.dataminingapps.com/home.html">
```

```
  <bartns:wine>
    <bartns:name>Jacques Selosse Brut Initial</bartns:name>
    <bartns:year>2012</bartns:year>
  </bartns:wine>
```

```
<winecellar xmlns="www.dataminingapps.com/defaultns.html">
```



- XPath is a simple, declarative language that uses path expressions to refer to parts of an XML document
  - » considers an XML document as an ordered tree
- Example XPath expressions

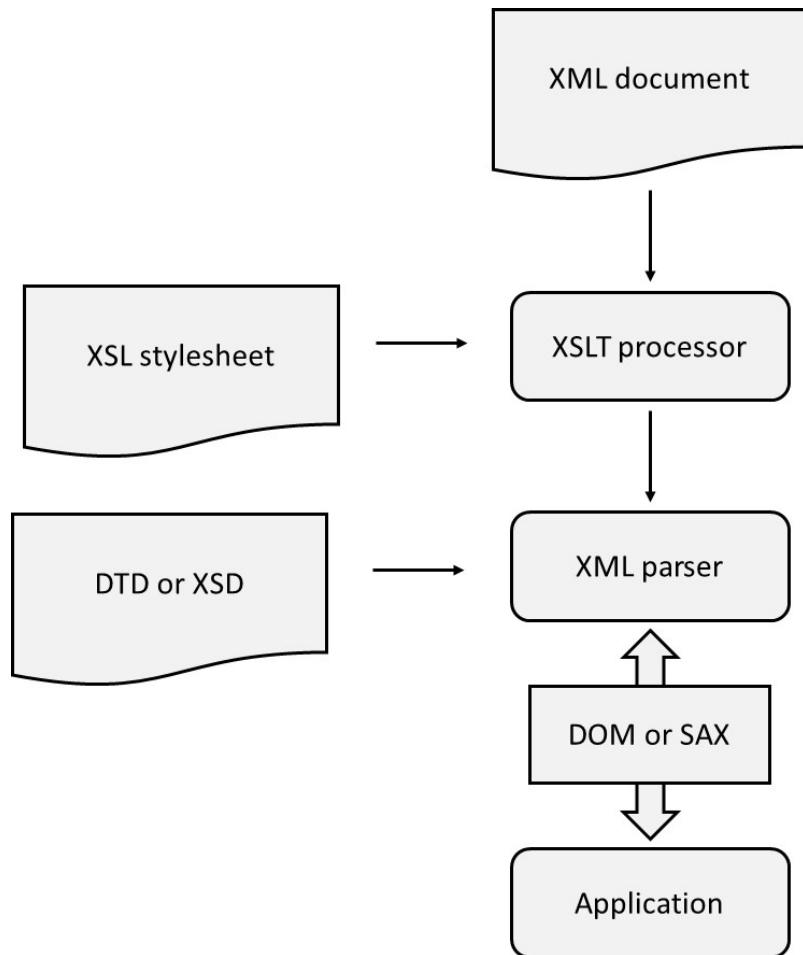
```
doc("winecellar.xml")/winecellar/wine
```

```
doc("winecellar.xml")/winecellar/wine[2]
```

```
doc("winecellar.xml")/winecellar/wine[price > 20]/name
```



# Processing XML Documents





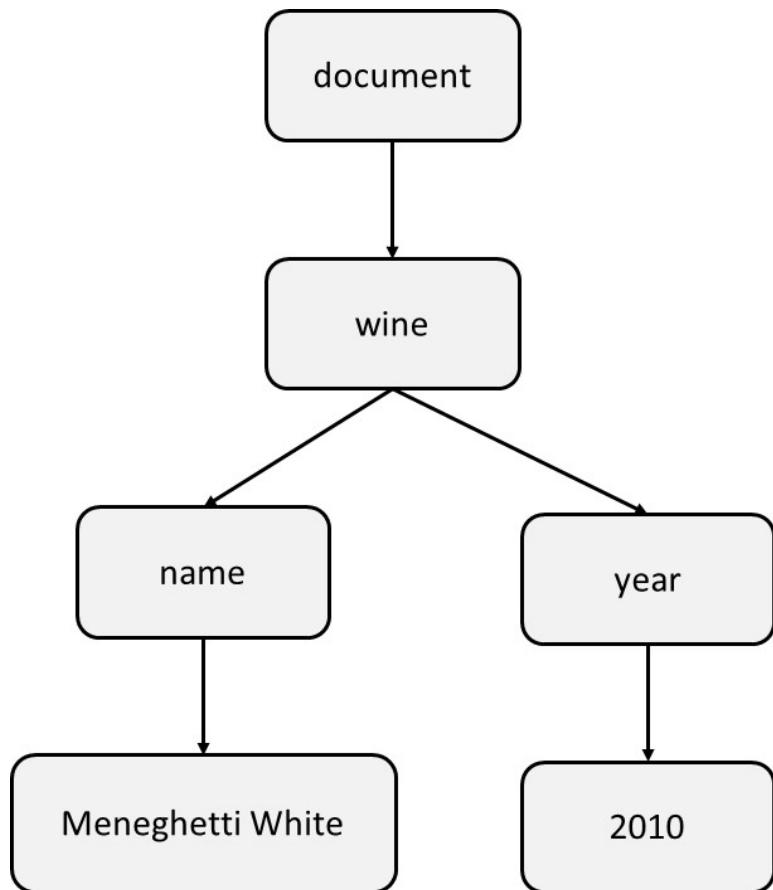
# Processing XML Documents

- DOM API is a tree-based API and represents XML document as a tree in internal memory
  - » developed by W3C
- DOM provides classes with methods to navigate through the tree and do various operations
- DOM is useful to facilitate direct access to specific XML document parts and when a high number of data manipulations are needed, but can get memory intensive



# Processing XML Documents

```
<wine>  
<name>Meneghetti White<name>  
<year>2010<year>  
<wine>
```





# Processing XML Documents

- SAX API (Simple API for XML) is an event-based API

```
start document
start element: wine
start element: name
text: Meneghetti
end element: name
start element: year
text: 2010
end element: year
end element: wine
end document
```



# Processing XML Documents

- Event stream can be passed on to application which will use an event handler
- SAX has smaller memory footprint and is more scalable than DOM
- SAX is excellent for sequential access, but less suited to support direct random access
- SAX is less performing for heavy data manipulation than DOM
- StAX (Streaming API for XML) is a compromise
  - » StAX allows the application to pull XML data using a cursor mechanism



# Storage of XML Documents

- XML documents stored as semi-structured data
- Approaches
  - » document-oriented approach
  - » data-oriented approach
  - » combined approach



# Document-Oriented Approach for Storing XML Documents

- XML document will be stored as a BLOB or CLOB in a table cell
  - » RDBMS considers these as ‘black box’ data
  - » querying based upon full-text search
  - » (O)RDBMSs have introduced XML data type (SQL/XML extension)
- Simple approach
  - » no need for DTD or XSD for the XML document
  - » especially well-suited for storing static content
  - » but: poor integration with relational SQL query processing



# Data-Oriented Approach for Storing XML Documents

- XML document decomposed into data parts spread across a set of connected (object-) relational tables (shredding)
- For highly structured documents and fine-granular queries
- DBMS or middleware can do translation
- Schema-oblivious shredding (starts from XML document) versus schema-aware shredding (starts from DTD/ XSD)
- Advantages
  - » SQL queries can now directly access individual XML elements
  - » reconstruct XML document using SQL joins
- Object-relational DBMS as an alternative



# The Combined Approach for Storing XML Documents

- Combined approach (partial shredding) combines document- and data-oriented approach
- Some parts stored as BLOBs, CLOBs, or XML objects, whereas other parts shredded
- SQL views are defined to reconstruct XML document
- Most DBMSs provide facilities to determine optimal level of decomposition
- Mapping approaches can be implemented using middleware or by DBMS (XML-enabled DBMS)



# Differences Between XML Data and Relational Data

- Building block of relational model is mathematical relation which consists of 0, 1 or more unordered tuples
- Each tuple consists of 1 or more attributes
- The relational model does not implement any type of ordering ( $\leftrightarrow$  XML model)
  - » add extra attribute type in RDBMS
  - » use list collection type in object-relational DBMS



# Differences Between XML Data and Relational Data

- Relational model does not support nested relations (first normal form)
  - » ↔ XML data is hierarchically structured
  - » object-relational DBMS supports nested relations
- Relational model does not support multivalued attribute types (first normal form)
  - » ↔ XML allows same child element to appear multiple times
  - » additional table needed in relational model
  - » object-relational model supports collection types



# Differences Between XML Data and Relational Data

- RDBMS only supports atomic data types, such as integer, string, date, etc.
  - » XML DTDs don't support atomic data types (only (P)CDATA)
  - » XML Schema supports both atomic and aggregated types
  - » aggregated types modeled in object-relational databases using user defined types
- XML data is semi-structured
  - » can include certain anomalies
  - » change to DTD or XSD necessitates re-generation of tables



- Table-Based Mapping
- Schema-Oblivious Mapping
- Schema-Aware Mapping
- SQL/XML



# Table-Based Mapping

- Specifies strict requirements to the structure of the XML document

```
<database>
  <table>
    <row>
      <column1> data </column1>
      ...
    </row>
    <row>
      <column1> data </column1>
      ...
    </row>
    ...
  </table>
  <table>
  ...
  </table>
  ...
</database>
```



## Table-Based Mapping

- Actual data is stored as content of column elements
- Advantage is simplicity given the perfect one-to-one mapping
- Document structure can be implemented using an updatable SQL view
- Disadvantage is rigid structure of XML document
  - » can be mitigated by XSLT



# Schema-Oblivious Mapping

- Schema-oblivious mapping (shredding) transforms XML document without availability of DTD or XSD
- First option is to transform the document to a tree structure, whereby the nodes represent the data in the document
  - » tree can then be mapped to a relational model
- Example table

```
CREATE TABLE NODE(
ID CHAR(6) NOT NULL PRIMARY KEY,
PARENT_ID CHAR(6),
TYPE VARCHAR(9),
LABEL VARCHAR(20),
VALUE CLOB,
FOREIGN KEY (PARENT_ID) REFERENCES NODE (ID)
CONSTRAINT CC1 CHECK(TYPE IN ("element", "attribute")));
```



# Schema-Oblivious Mapping

```
<?xml version="1.0" encoding="UTF-8"?>
<winecellar>
    <wine winekey="1">
        <name>Jacques Selosse Brut Initial</name>
        <year>2012</year>
        <type>Champagne</type>
        <price>150</price>
    </wine>
    <wine winekey="2">
        <name>Meneghetti White</name>
        <year>2010</year>
        <type>white wine</type>
        <price>18</price>
    </wine>
</winecellar>
```

ID	PARENT_ID	TYPE	LABEL	VALUE
1	NULL	element	winecellar	NULL
2	1	element	wine	NULL
3	2	attribute	winekey	1
4	2	element	name	Jacques Selosse Brut Initial
5	2	element	year	2012
6	2	element	type	Champagne
7	2	element	price	150
8	1	element	wine	NULL
9	8	attribute	winekey	2
10	8	element	name	Meneghetti White
11	8	element	year	2010
12	8	element	type	white wine
13	8	element	price	18



# Schema-Oblivious Mapping

- XPath or XQuery (see later) queries can be translated into SQL of which the result can be translated back to XML
- Example

```
doc("winecellar.xml")/winecellar/wine[price > 20]/name
```

```
SELECT N2.VALUE  
FROM NODE N1, NODE N2  
WHERE  
N2.LABEL="name" AND  
N1.LABEL="price" AND  
CAST(N1.VALUE AS INT)> 20 AND  
N1.PARENT_ID=N2.PARENT_ID
```



# Schema-Oblivious Mapping

- Single table requires extensive querying (e.g., self-joins)
- More tables can be created
- Mapping can be facilitated by making use of object-relational extensions
- Due to extensive shredding, reconstruction of XML document can get quite resource intensive
  - » middleware solutions offer DOM API or SAX API on top of DBMS
  - » materialized views



# Schema-Aware Mapping

- Steps to generate database schema from DTD or XSD
  - » simplify DTD or XSD
  - » map complex element type to relational table, or user-defined type, with corresponding primary key
  - » map element type with mixed content to separate table where the (P)CDATA is stored; connect using primary-foreign key relationship
  - » map single-valued attribute types, or child elements that occur only once, with (P)CDATA content to a column in the corresponding relational table; when starting from XSD, choose the SQL data type which most closely resembles
  - » map multi-valued attribute types, or child elements that can occur multiple times, with (P)CDATA content to a separate table; use primary-foreign key relationship; use collection type in case of object-relational DBMS
  - » for each complex child element type, connect the tables using a primary-foreign key relationship



- Generate a DTD or XSD from a database model
  - » map every table to an element type
  - » map every table column to an attribute type or child element type with (P)CDATA in case of DTD, or most closely resembling data type in case of XML Schema
  - » map primary-foreign key relationships by introducing additional child element types
  - » object-relational collections can be mapped to multivalued attribute types or element types which can occur multiple times



- Extension of SQL which introduces
  - » new XML data type with corresponding constructor that treats XML documents as cell values in a column of a relational table, and can be used to define attribute types in user-defined types, variables, and parameters of user-defined functions
  - » set of operators for the XML data type
  - » set of functions to map relational data to XML
- No rules for shredding



```
CREATE TABLE PRODUCT(  
PRODNR CHAR(6) NOT NULL PRIMARY KEY,  
PRODNAME VARCHAR(60) NOT NULL,  
PRODTYPE VARCHAR(15),  
AVAILABLE_QUANTITY INTEGER,  
REVIEW XML);
```

```
INSERT INTO PRODUCT VALUES("120", "Conundrum", "white", 12,  
XML(<review><author>Bart  
Baesens</author><date>27/02/2017</date> <description>This is  
an excellent white wine with intriguing aromas of green apple,  
tangerine and honeysuckle blossoms.<description><rating max-  
value="100">94</rating></review>);
```



- SQL/XML can be used to represent relational data in XML
  - » default mapping whereby names of tables and columns are translated to XML elements and row elements are included for each table row
  - » also adds corresponding DTD or XSD
- SQL/XML also includes facilities to represent the output of SQL queries in a tailored XML format
  - » XMLElement defines XML element using 2 arguments: name of XML element and column name



```
SELECT XMLElement("sparkling wine", PRODNAME)
FROM PRODUCT
WHERE PRODTYPE="sparkling";
```

```
<sparkling wine>Meerdael, Methode Traditionnelle  
Chardonnay, 2014 </sparkling wine>  
<sparkling wine>Jacques Selosse, Brut Initial,  
2012</sparkling wine>  
<sparkling wine>Billecart-Salmon, Brut Réserve,  
2014</sparkling wine>  
...
```



# SQL/XML

```
SELECT XMLElement("sparkling wine", XMLAttributes(PRODNR AS "prodid"),
XMLElement("name", PRODNAME), XMLElement("quantity", AVAILABLE_QUANTITY))
FROM PRODUCT
WHERE PRODTYPE="sparkling";
```

```
<sparkling wine prodid="0178">
<name>Meerdael, Methode Traditionnelle Chardonnay, 2014</name>
<quantity>136</quantity>
</sparkling wine>
<sparkling wine prodid="0199">
<name>Jacques Selosse, Brut Initial, 2012</name>
<quantity>96</quantity>
</sparkling wine>
...
...
```

```
SELECT XMLElement("sparkling wine", XMLAttributes(PRODNR AS "prodid"),
XMLForest(PRODNAME AS "name", AVAILABLE_QUANTITY AS "quantity"))
FROM PRODUCT
WHERE PRODTYPE="sparkling";
```



```
SELECT XMLElement("product", XMLElement(prodid, P.PRODNR), XMLElement("name",
P.PRODNAME, XMLAgg("supplier", S.SUPNR))
FROM PRODUCT P, SUPPLIES S
WHERE P.PRODNR=S.PRODNR
GROUP BY P.PRODNR

<product>
<prodid>178</prodid>
<name>Meerdael, Methode Traditionnelle Chardonnay</name>
<supplier>21</supplier>
<supplier>37</supplier>
<supplier>68</supplier>
<supplier>69</supplier>
<supplier>94</supplier>
</product>
<product>
<prodid>199</prodid>
<name>Jacques Selosse, Brut Initial, 2012</name>
<supplier>69</supplier>
<supplier>94</supplier>
</product>
...
...
```



```
SELECT PRODNR, XMLElement("sparkling wine", PRODNAME),  
AVAILABLE_QUANTITY  
FROM PRODUCT  
WHERE PRODTYPE="sparkling";
```

0178, <sparkling wine>Meerdael, Methode Traditionnelle Chardonnay, 2014</sparkling wine>, 136

0199, <sparkling wine>Jacques Selosse, Brut Initial, 2012</sparkling wine>, 96

0212, <sparkling wine>Billecart-Salmon, Brut Réserve, 2014</sparkling wine>, 141

...



- Template-based mapping
  - » embed SQL statements in XML documents using tool-specific delimiter (e.g., <selectStmt>)

```
<?xml version="1.0" encoding="UTF-8"?>
<sparklingwines>
  <heading>List of Sparkling Wines</heading>
  <selectStmt>
    SELECT PRODNAME, AVAILABLE_QUANTITY FROM PRODUCT WHERE
    PRODTYPE="sparkling";
  </selectStmt>
  <wine>
    <name> $PRODNAME </name>
    <quantity> $AVAILABLE_QUANTITY </quantity>
  </wine>
</sparklingwines>
```



```
<?xml version="1.0" encoding="UTF-8"?>
<sparklingwines>
  <heading>List of Sparkling Wines</heading>
  <wine>
    <name>Meerdael, Methode Traditionnelle Chardonnay, 2014</name>
    <quantity>136</quantity>
  </wine>
  <wine>
    <name>Jacques Selosse, Brut Initial, 2012</name>
    <quantity>96</quantity>
  </wine>
  ..
</sparklingwines>
```



# Searching XML Data

- Full-text search
- Keyword-Based Search
- Structured Search with Xquery
- Semantic Search with RDF and SPARQL



## Full-text search

- Treat XML documents as textual data and conduct brute force full-text search
- Does not take into account any tag structure
- Can be applied to XML documents that have been stored as files or as BLOB/CLOB objects
- Usually by means of object-relational extension
- No semantically-rich queries targeting individual XML elements



## Keyword-Based Search

- Assumes XML document is complemented with a set of keywords describing document metadata
- Keywords can be indexed by text search engines
- Document still stored in a file or as BLOB/CLOB
- Still not full expressive power of XML for querying



# Structured Search with XQuery

- Structured search uses structural metadata which relates to actual document content
- E.g., XML book reviews
  - » document metadata: properties of the document such as, author of the review document (e.g., Wilfried Lemahieu) and creation date (e.g., June 6<sup>th</sup>, 2017)
  - » structural metadata: role of individual content fragments within the overall document structure, e.g., title of book ('Analytics in a Big Data World'), author of book ('Bart Baesens'), ...



# Structured Search with XQuery

- Structured search queries query document content by means of structural metadata
  - » E.g., search for reviews of books authored by Bart Baesens
- XQuery formulates structured queries for XML documents
  - » can consider both document structure and elements' content
  - » XPath path expressions are used for navigation
  - » includes constructs to refer to and compare content of elements
  - » syntax similar to SQL



- XQuery statement is formulated as a FLOWR instruction

```
FOR $variable IN expression  
LET $variable:=expression  
WHERE filtercriterion  
ORDER BY sortcriterion  
RETURN expression
```



# Structured Search with XQuery

```
LET $maxyear:=2012  
RETURN doc("winecellar.xml")/winecellar/wine[year <$maxyear]
```

```
FOR $wine IN doc("winecellar.xml")/winecellar/wine  
ORDER BY $wine/year ASCENDING  
RETURN $wine
```

```
FOR $wine IN doc("winecellar.xml")/winecellar/wine  
WHERE $wine/price < 20 AND $wine/price/@currency="EURO"  
RETURN <cheap wine> {$wine/name, $wine/price}</cheap wine>
```

```
FOR $wine IN doc("winecellar.xml")/wine  
    $winereview IN doc("winereview.xml")/winereview  
WHERE $winereview/@winekey=$wine/@winekey  
RETURN <wineinfo> {$wine, $winereview/rating} </wineinfo>
```



- Example of semantically-complicated query

*“Retrieve all spicy, ruby colored red wines with round texture raised in clay soil and Mediterranean climate which pair well with cheese”*
- Semantic web technology stack
  - » RDF
  - » RDF Schema
  - » OWL
  - » SPARQL



# Semantic Search with RDF and SPARQL

- Resource Description Framework (RDF) provides data model for semantic web
  - » encodes graph-structured data by attaching semantic meaning to relationships
  - » data model consists of statements in subject-predicate-object format (triples)

Subject	Predicate	Object
Bart	name	Bart Baesens
Bart	likes	Meneghetti White
Meneghetti White	tastes	Citrusy
Meneghetti White	pairs	Fish



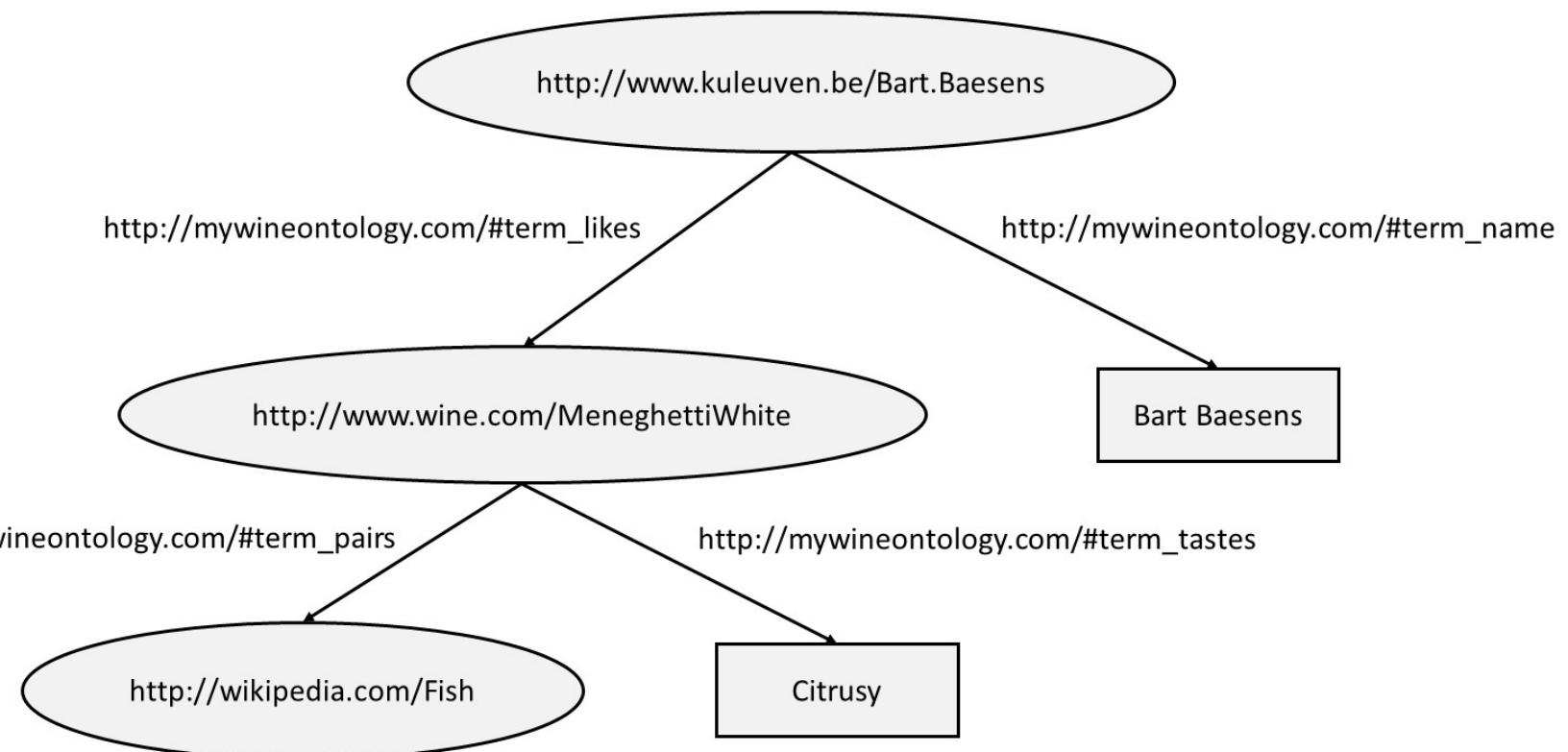
# Semantic Search with RDF and SPARQL

- Represent subjects and predicates using URIs, and objects using URIs
  - » universal unique identification becomes possible
- Note: predicate refers to vocabulary or ontology

Subject	Predicate	Object
<a href="http://www.kuleuven.be/Bart.Baesens">http://www.kuleuven.be/Bart.Baesens</a>	<a href="http://mywineontology.com/#term_name">http://mywineontology.com/#term_name</a>	"Bart Baesens"
<a href="http://www.kuleuven.be/Bart.Baesens">http://www.kuleuven.be/Bart.Baesens</a>	<a href="http://mywineontology.com/#term_likes">http://mywineontology.com/#term_likes</a>	<a href="http://www.wine.com/MeneghettiWhite">http://www.wine.com/MeneghettiWhite</a>
<a href="http://www.wine.com/MeneghettiWhite">http://www.wine.com/MeneghettiWhite</a>	<a href="http://mywineontology.com/#term_taste">http://mywineontology.com/#term_taste</a>	"Citrusy"
<a href="http://www.wine.com/MeneghettiWhite">http://www.wine.com/MeneghettiWhite</a>	<a href="http://mywineontology.com/#term_pairs">http://mywineontology.com/#term_pairs</a>	<a href="http://wikipedia.com/Fish">http://wikipedia.com/Fish</a>



# Semantic Search with RDF and SPARQL





- RDF data can be serialized by means of RDF/XML

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/TR/PR-rdf-syntax/"
  xmlns:myxmlns="http://mywineontology.com/" />
<rdf:Description
  rdf:about="http://www.kuleuven.be/Bart.Baesens">
  <myxmlns:name>Bart Baesens</myxmlns:name>
  <myxmlns:likes
    rdf:resource="http://www.wine.com/MeneghettiWhite"/>
</rdf:Description>
</rdf:RDF>
```



- RDF is one of the key technologies to realize Linked Data
- RDF Schema enriches RDF by extending its vocabulary with classes and subclasses, properties and subproperties, and typing of properties
- Web Ontology Language (OWL) is an even more expressive ontology language which implements various sophisticated semantic modeling concepts



# Semantic Search with RDF and SPARQL

- RDF data can be queried using SPARQL (“SPARQL Protocol and RDF Query Language”)
- SPARQL is based upon matching graph patterns against RDF graphs
- Examples

```
PREFIX: mywineont: <http://mywineontology.com/>
```

```
SELECT ?wine
```

```
WHERE {?wine, mywineont:tastes, "Citrusy"}
```

```
PREFIX: mywineont: <http://mywineontology.com/>
```

```
SELECT ?wine, ?flavor
```

```
WHERE {?wine, mywineont:tastes, ?flavor}
```



- Message Oriented Middleware (MOM)
- SOAP-Based Web Services
- REST-Based Web Services
- Web Services and Databases



# Message Oriented Middleware (MOM)

- Enterprise Application Integration (EAI): set of activities aimed at integrating applications within an enterprise
- EAI can be facilitated by 2 types of middleware
  - » Remote Procedure Call (RPC): communication is established through procedure calls (e.g., RMI, DCOM); usually synchronous; strong coupling
  - » Message Oriented Middleware (MOM) integration is established by exchanging XML messages; usually asynchronous; loose coupling



# SOAP-Based Web Services

- Web services: self-describing software components, which can be published, discovered and invoked through the web
- Simple Object Access Protocol (SOAP)
  - » Extensible, neutral, and independent XML-based messaging framework

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetQuote xmlns="http://www.webserviceX.NET/">
      <symbol>string</symbol>
    </GetQuote>
  </soap:Body>
</soap:Envelope>
```



# SOAP-Based Web Services

- Before a SOAP message can be sent to a web service, it must be clear which type(s) of incoming messages the service understands and what messages it can send in return
- Web Services Description Language (WSDL) is an XML-based language used to describe the interface or functionalities offered by a web service



# SOAP-Based Web Services

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://www.webserviceX.NET/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.webserviceX.NET/" xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/">
  <wsdl:types><s:schema targetNamespace="http://www.webserviceX.NET/" elementFormDefault="qualified">
    <s:element name="GetQuote">
      <s:complexType>
        <s:sequence>
          <s:element type="s:string" name="symbol" maxOccurs="1" minOccurs="0"/></s:sequence>
        </s:complexType></s:element>
    <s:element name="GetQuoteResponse">
      <s:complexType>
        <s:sequence>
          <s:element type="s:string" name="GetQuoteResult" maxOccurs="1" minOccurs="0"/>
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element type="s:string" name="string" nillable="true"/>
  </s:schema>
  </wsdl:types>
  ...
</wsdl:definitions>
```



# SOAP-Based Web Services

- Web service represented as set of port types that define set of abstract operations
  - » operation has input message and optional output message (SOAP based)
  - » message specifies attributes and their types using XML Schema
  - » port types can be mapped to an implementation (port) by specifying URL
  - » same WSDL document can refer to multiple implementations
- E-business transactions take place according to predefined process model based on web services and XML



- REST (Representational State Transfer) is built on top of HTTP and is completely stateless and light
  - » less verbose than SOAP
  - » based on request-reply functionality, for which HTTP is already perfectly suited
  - » has become the architecture of choice by “modern” web companies to provide APIs
  - » REST is tightly integrated with HTTP whereas SOAP is communication agnostic



# REST-Based Web Services

```
GET /stockquote/IBM HTTP/1.1
Host: www.example.com
Connection: keep-alive
Accept: application/xml
```

```
HTTP/1.0 200 OK
Content-Type: application/xml
<StockQuotes>
<Stock>
<Symbol>IBM</Symbol>
<Last>140,33</Last>
<Date>22/8/2017</Date>
<Time>11:56am</Time>
<Change>-0.16</Change>
<Open>139,59</Open>
<High>140,42</High>
<Low>139,13</Low>
<MktCap>135,28B</MktCap>
<P-E>11,65</P-E>
<Name>International Business Machines</Name>
</Stock>
</StockQuotes>
```



# Web Services and Databases

- Web service can make use of underlying database
- Database can act as web service provider or web service consumer
- Stored procedures can be extended with WSDL interface and published as web services
  - » results can be returned as XML (e.g., SQL/XML)
- Stored procedures or triggers can include calls to external web services
  - » E.g., trigger which monitors (local) stock data and if safety stock level is reached automatically generates a (e.g. SOAP) message with a purchase order to the web service hosted by the supplier
- Implications on transaction management (e.g. WS-BPEL)!



# Other Data Representation Formats

- JSON and YAML are optimized for data interchange and serialization
- JavaScript Object Notation (JSON) provides a simple, lightweight representation based on name-value pairs
  - » JSON provides 2 structured types: objects and arrays
  - » primitive types supported: string, number, Boolean, and null
  - » JSON is human and machine readable and models data in hierarchical way
  - » structure of JSON specification can be defined using JSON Schema
  - » JSON is not a markup language and not extensible
  - » JSON documents can be parsed using the eval() function
  - » native and fast JSON parsers in modern day web browsers



# Other Data Representation Formats

```
{  
  "winecellar": {  
    "wine": [  
      {  
        "name": "Jacques Selosse Brut Initial",  
        "year": "2012",  
        "type": "Champagne",  
        "grape": {  
          "__percentage": "100",  
          "__text": "Chardonnay"  
        },  
        "price": {  
          "__currency": "EURO",  
          "__text": "150"  
        },  
      ]  
    }  
  }  
}
```



# Other Data Representation Formats

```
"geo": {  
    "country": "France",  
    "region": "Champagne"  
},  
"quantity": "12"  
,  
{  
    "name": "Meneghetti White",  
    "year": "2010",  
    "type": "white wine",  
    "grape": [  
        {  
            "_percentage": "80",  
            "__text": "Chardonnay"  
        },  
        {  
            "_percentage": "20",  
            "__text": "Pinot Blanc"  
        }  
    ],  
    "price": {  
        "_currency": "EURO",  
        "__text": "18"  
    },  
    "geo": {  
        "country": "Croatia",  
        "region": "Istria"  
    },  
    "quantity": "20"  
}  
}
```



## Other Data Representation Formats

- **YAML Ain't a Markup Language (YAML)** is a superset of JSON with support for relational trees, user-defined types, explicit data typing, lists and casting
  - » better alternative for object serialization
  - » uses inline and white space delimiters
  - » works with mappings, which are sets of unordered key/value pairs and sequences which correspond to arrays
  - » supports numbers, strings, Boolean, dates, timestamps, and null



# Other Data Representation Formats

```
winecellar:  
  wine:  
    -  
      name: "Jacques Selosse Brut Initial"  
      year: 2012  
      type: Champagne  
      grape:  
        _percentage: 100  
        __text: Chardonnay  
      price:  
        _currency: EURO  
        __text: 150  
      geo:  
        country: France  
        region: Champagne  
      quantity: 12  
    -  
      name: "Meneghetti White"  
      year: 2010  
      type: "white wine"  
      grape:  
        -  
          _percentage: 80  
          __text: Chardonnay  
        -  
          _percentage: 20  
          __text: "Pinot Blanc"  
      price:  
        _currency: EURO  
        __text: 18  
      geo:  
        country: Croatia  
        region: Istria  
      quantity: 20
```

# Agenda

1 Session Overview

2 Legacy Databases

3 OODBs and Object Persistence

4 Extended Relational Databases

5 XML Databases

6 NoSQL Databases

7 Summary and Conclusion



# Agenda

- NoSQL Databases
  - The NoSQL movement
  - Key-Value stores
  - Tuple and Document stores
  - Column-oriented databases
  - Graph based databases
  - Other NoSQL categories



# The NoSQL movement

- RDBMSs put a lot of emphasis on keeping data consistent.
  - » Entire database is consistent at all times (ACID)
- Focus on consistency may hamper flexibility and scalability
- As the data volumes or number of parallel transactions increase, capacity can be increased by
  - » Vertical scaling: extending storage capacity and/or CPU power of the database server
  - » Horizontal scaling: multiple DBMS servers being arranged in a cluster



- RDBMSs are not good at extensive horizontal scaling
  - » Coordination overhead because of focus on consistency
  - » Rigid database schemas
- Other types of DBMSs needed for situations with massive volumes, flexible data structures and where scalability and availability are more important → NoSQL databases



- NoSQL databases
  - » Describes databases that store and manipulate data in other formats than tabular relations, i.e. non-relational databases (NoREL)
- NoSQL databases aim at near linear horizontal scalability, by distributing data over a cluster of database nodes for the sake of performance as well as availability
- Eventual consistency: the data (and its replicas) will become consistent at some point in time after each transaction



# The NoSQL movement

	<b>Relational Databases</b>	<b>NoSQL Databases</b>
<b>Data paradigm</b>	Relational tables	Key-value (tuple) based Document based Column based Graph based XML, object based Others: time series, probabilistic, etc.
<b>Distribution</b>	Single-node and distributed	Mainly distributed
<b>Scalability</b>	Vertical scaling, harder to scale horizontally	Easy to scale horizontally, easy data replication
<b>Openness</b>	Closed and open source	Mainly open source
<b>Schema role</b>	Schema-driven	Mainly schema-free or flexible schema
<b>Query language</b>	SQL as query language	No or simple querying facilities, or special-purpose languages
<b>Transaction mechanism</b>	ACID: Atomicity, Consistency, Isolation, Durability	BASE: Basically available, Soft state, Eventual consistency
<b>Feature set</b>	Many features (triggers, views, stored procedures, etc.)	Simple API
<b>Data volume</b>	Capable of handling normal-sized data sets	Capable of handling huge amounts of data and/or very high frequencies of read/write requests



# Key-value Stores

- Key-value based database stores data as (key, value) pairs
  - » Keys are unique
  - » Hash map, or hash table or dictionary



# Key-value Stores

```
import java.util.HashMap;
import java.util.Map;
public class KeyValueStoreExample {
    public static void main(String... args) {
        // Keep track of age based on name
        Map<String, Integer> age_by_name = new HashMap<>();

        // Store some entries
        age_by_name.put("wilfried", 34);
        age_by_name.put("seppe", 30);
        age_by_name.put("bart", 46);
        age_by_name.put("jeanne", 19);

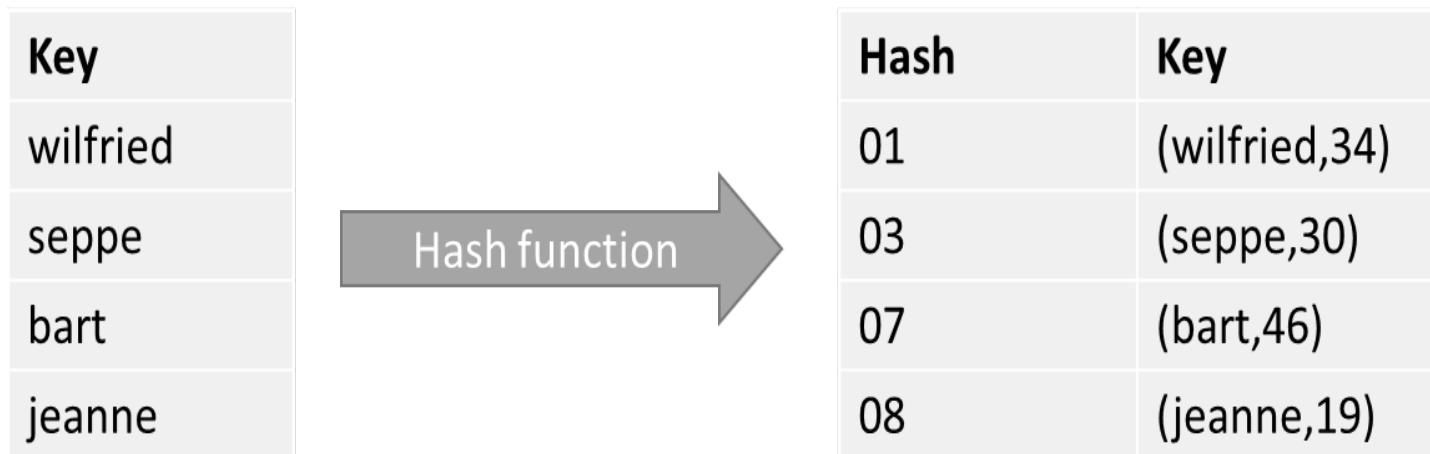
        // Get an entry
        int age_of_wilfried = age_by_name.get("wilfried");
        System.out.println("Wilfried's age: " + age_of_wilfried);

        // Keys are unique
        age_by_name.put("seppe", 50); // Overrides previous entry
    }
}
```



# Key-value Stores

- Keys (e.g., “bart”, “seppe”) are hashed by means of a so-called **hash function**
  - » A hash function takes an arbitrary value of arbitrary size and maps it to a key with a fixed size, which is called the hash value.
  - » Each hash can be mapped to a space in computer memory

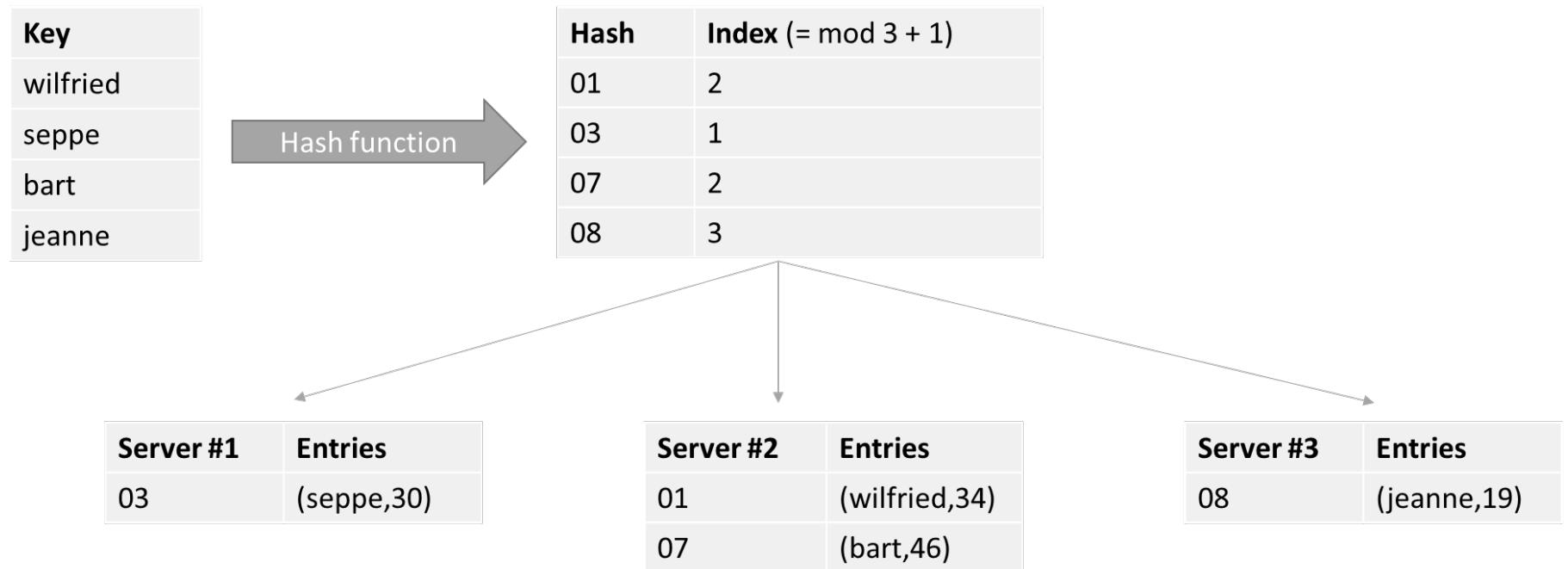




- NoSQL databases are built with horizontal scalability support in mind
- Distribute hash table over different locations
- Assume we need to spread our hashes over three servers
  - » Hash every key ("wilfried", "seppe") to a server identifier
  - »  $\text{index}(\text{hash}) = \text{mod}(\text{hash}, \text{nrServers}) + 1$



# Key-value Stores



Sharding!



- Example: Memcached
  - » Implements a distributed memory-driven hash table (i.e. a key-value store), which is put in front of a traditional database to speed up queries by caching recently accessed objects in RAM
  - » Caching solution



# Key-value Stores

```
import java.util.ArrayList;
import java.util.List;
import net.spy.memcached.AddrUtil;
import net.spy.memcached.MemcachedClient;

public class MemCachedExample {
    public static void main(String[] args) throws Exception {
        List<String> serverList = new ArrayList<String>() {
            {
                this.add("memcachedserver1.servers:11211");
                this.add("memcachedserver2.servers:11211");
                this.add("memcachedserver3.servers:11211");
            }
        };
    }
}
```



# Key-value Stores

```
MemcachedClient memcachedClient = new MemcachedClient(  
    AddrUtil.getAddresses(serverList));  
  
// ADD adds an entry and does nothing if the key already exists  
// Think of it as an INSERT  
// The second parameter (0) indicates the expiration - 0 means no expiry  
memcachedClient.add("marc", 0, 34);  
memcachedClient.add("seppe", 0, 32);  
memcachedClient.add("bart", 0, 66);  
memcachedClient.add("jeanne", 0, 19);  
  
// SET sets an entry regardless of whether it exists  
// Think of it as an UPDATE-OR-INSERT  
memcachedClient.add("marc", 0, 1111); // <- ADD will have no effect  
memcachedClient.set("jeanne", 0, 12); // <- But SET will
```



# Key-value Stores

```
// REPLACE replaces an entry and does nothing if the key does not exist
// Think of it as an UPDATE
memcachedClient.replace("not_existing_name", 0, 12); // <- Will have no effect
memcachedClient.replace("jeanne", 0, 10);

// DELETE deletes an entry, similar to an SQL DELETE statement
memcachedClient.delete("seppe");

// GET retrieves an entry
Integer age_of_marc = (Integer) memcachedClient.get("marc");
Integer age_of_short_lived = (Integer) memcachedClient.get("short_lived_name");
Integer age_of_not_existing = (Integer) memcachedClient.get("not_existing_name");
Integer age_of_seppe = (Integer) memcachedClient.get("seppe");
System.out.println("Age of Marc: " + age_of_marc);
System.out.println("Age of Seppe (deleted): " + age_of_seppe);
System.out.println("Age of not existing name: " + age_of_not_existing);
System.out.println("Age of short lived name (expired): " + age_of_short_lived);

memcachedClient.shutdown();

}
```



- Request Coordination
- Consistent Hashing
- Replication and Redundancy
- Eventual Consistency
- Stabilization
- Integrity Constraints and Querying



- In many NoSQL implementations (e.g. Cassandra, Google's BigTable, Amazon's DynamoDB) all nodes implement the same functionality and are all able to perform the role of request coordinator
- Need for membership protocol
  - » Dissemination
    - Based on periodic, pairwise communication
  - » Failure detection



- **Consistent hashing** schemes are often used, which avoid having to remap each key to a new node when nodes are added or removed
- Suppose we have a situation where 10 keys are distributed over 3 servers ( $n = 3$ ) with the following hash function
  - »  $h(\text{key}) = \text{key} \bmod n$



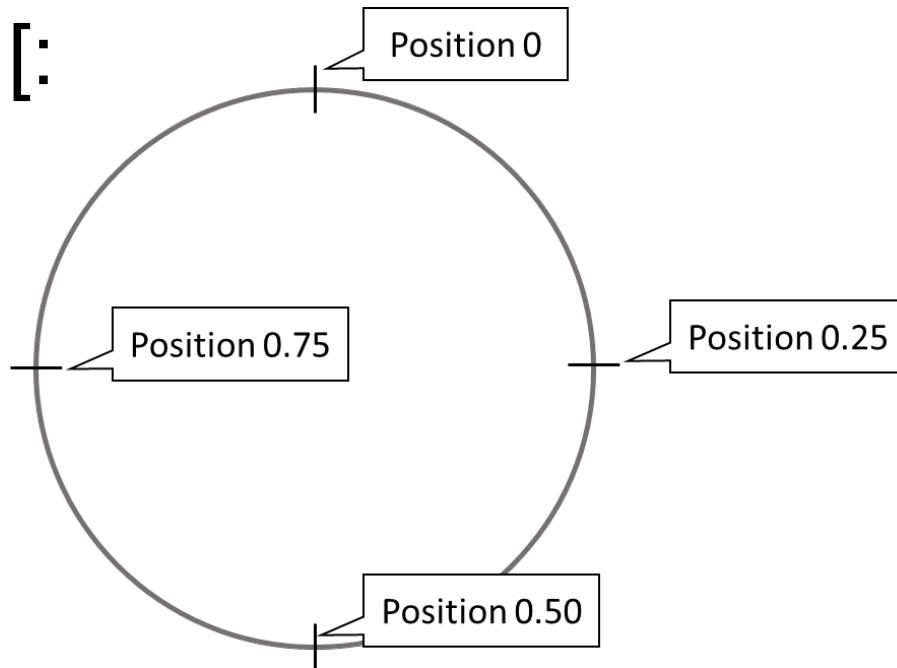
# Consistent Hashing

	n		
key	3	2	4
0	0	0	0
1	1	1	1
2	2	0	2
3	0	1	3
4	1	0	0
5	2	1	1
6	0	0	2
7	1	1	3
8	2	0	0
9	0	1	1



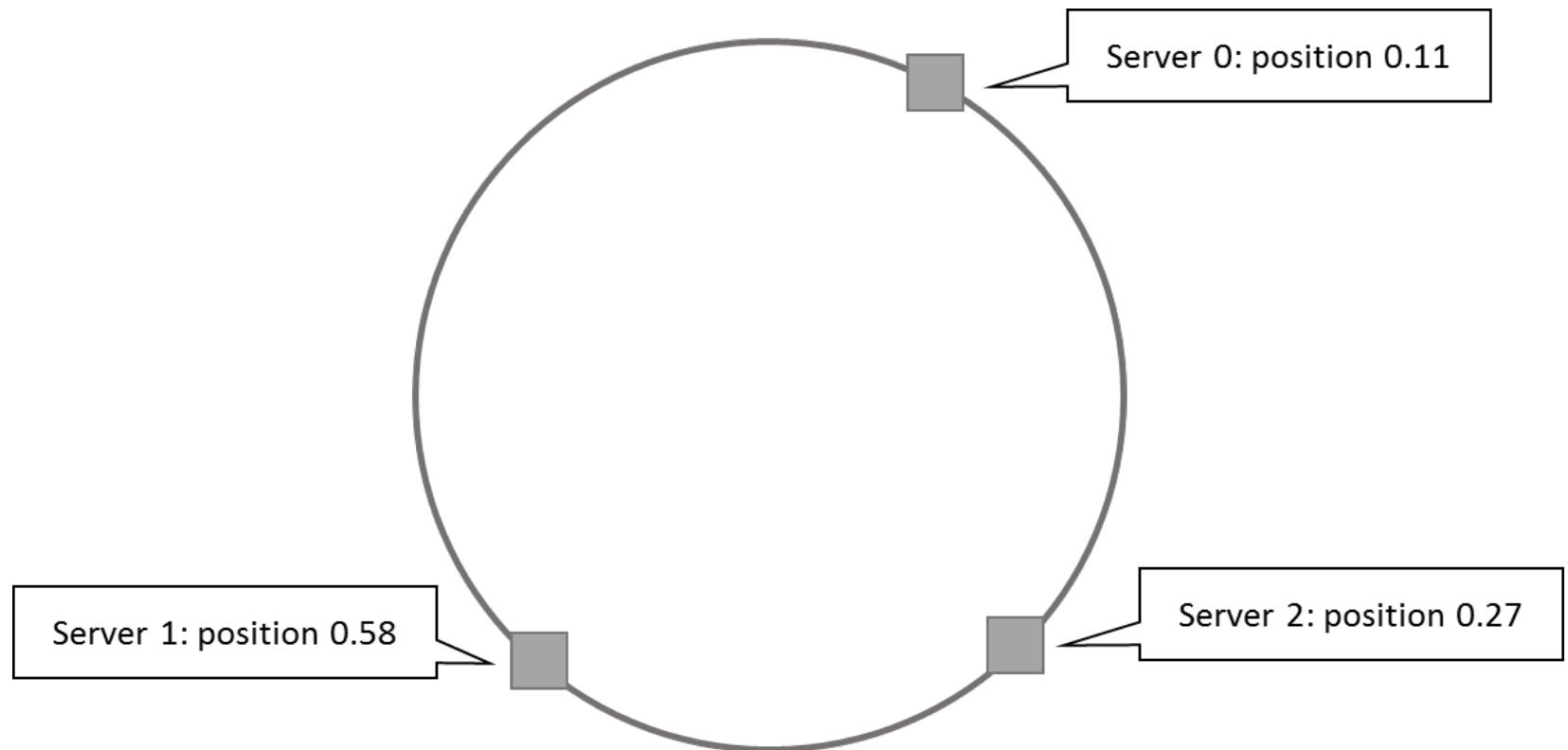
# Consistent Hashing

- At the core of a consistent hashing setup is a so called “**ring**”-topology, which is basically a representation of the number range  $[0,1[$ :





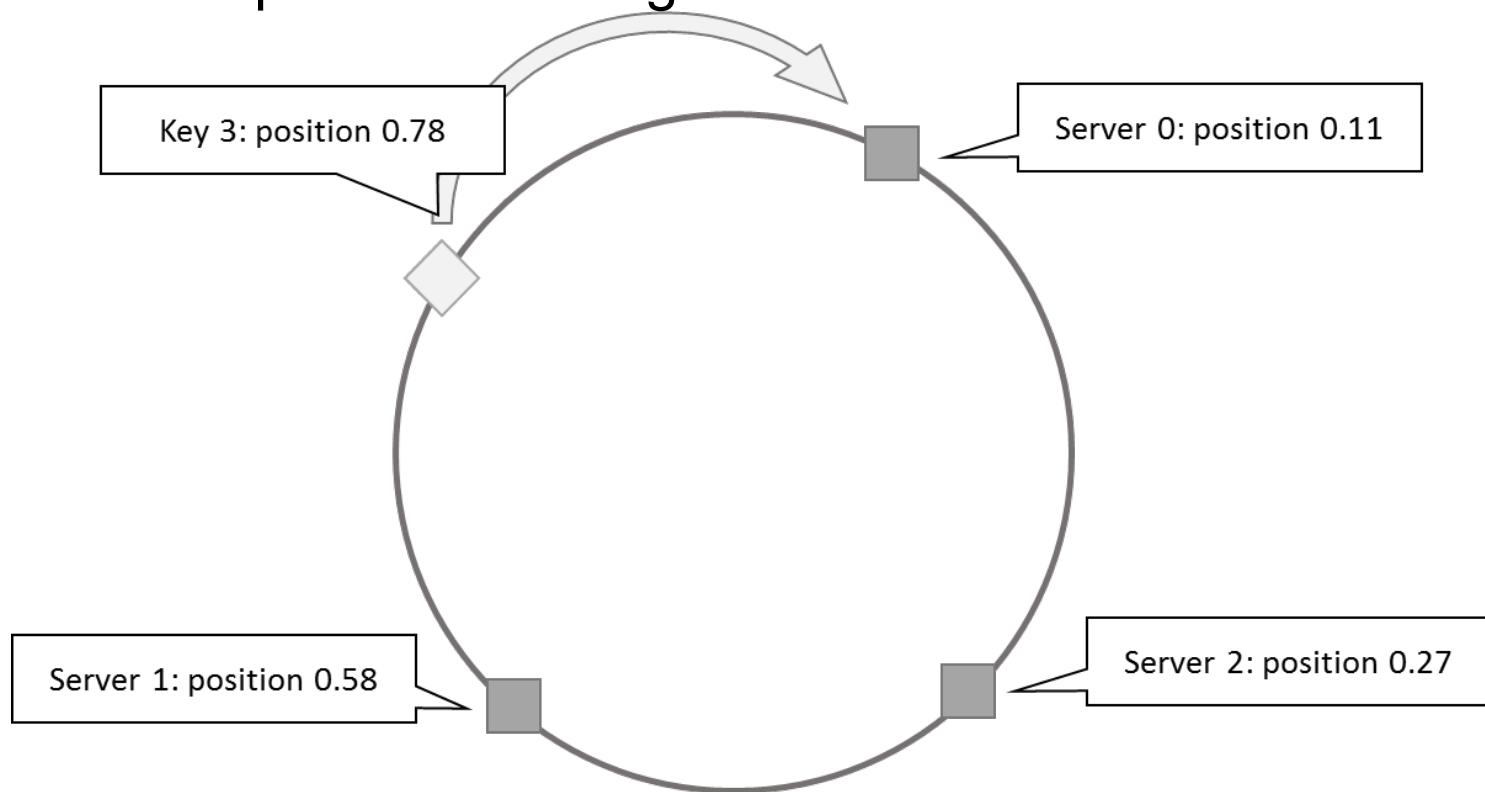
# Consistent Hashing





# Consistent Hashing

- Hash each key to a position on the ring, and store the actual key-value pair on the first server that appears clockwise of the hashed point on the ring



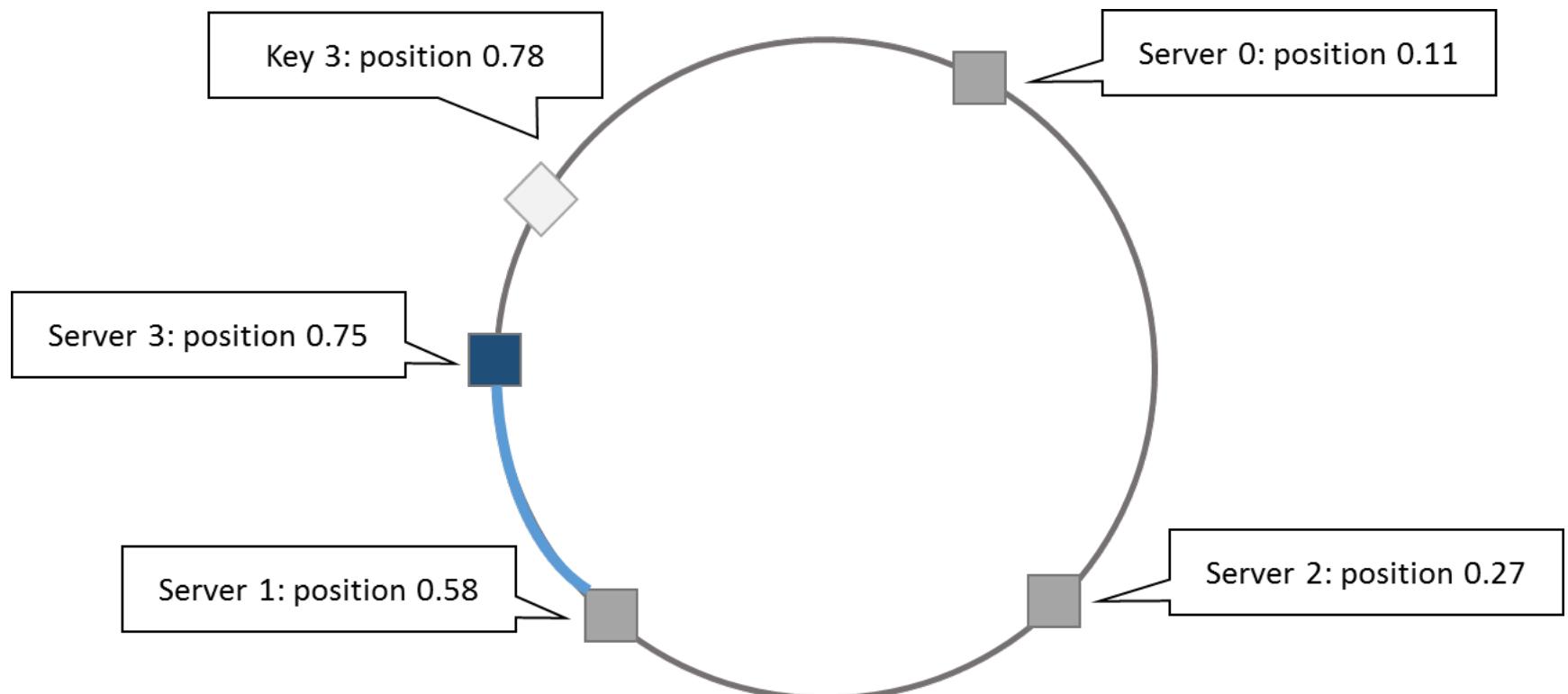


# Consistent Hashing

- Because of the uniformity property of a “good” hash function, roughly  $1/n$  of key-value pairs will end up being stored on each server
- Most of the key-value pairs will remain unaffected in case a machine is added or removed



# Consistent Hashing





# Replication and Redundancy

- Problems with consistent hashing:
  - » If 2 servers end up being mapped close to one another, one of these nodes will end up with few keys to store
  - » In case a server is added, all of the keys moved to this new node originate from just one other server
- Instead of mapping a server  $s$  to a single point on our ring, we map it multiple positions, called **replicas**
- For each physical server  $s$ , we hence end up with  $r$  (the number of replicas) points on the ring
- Note: each of the replicas still represents the same physical instance ( $\leftrightarrow$  redundancy)
  - » Virtual nodes



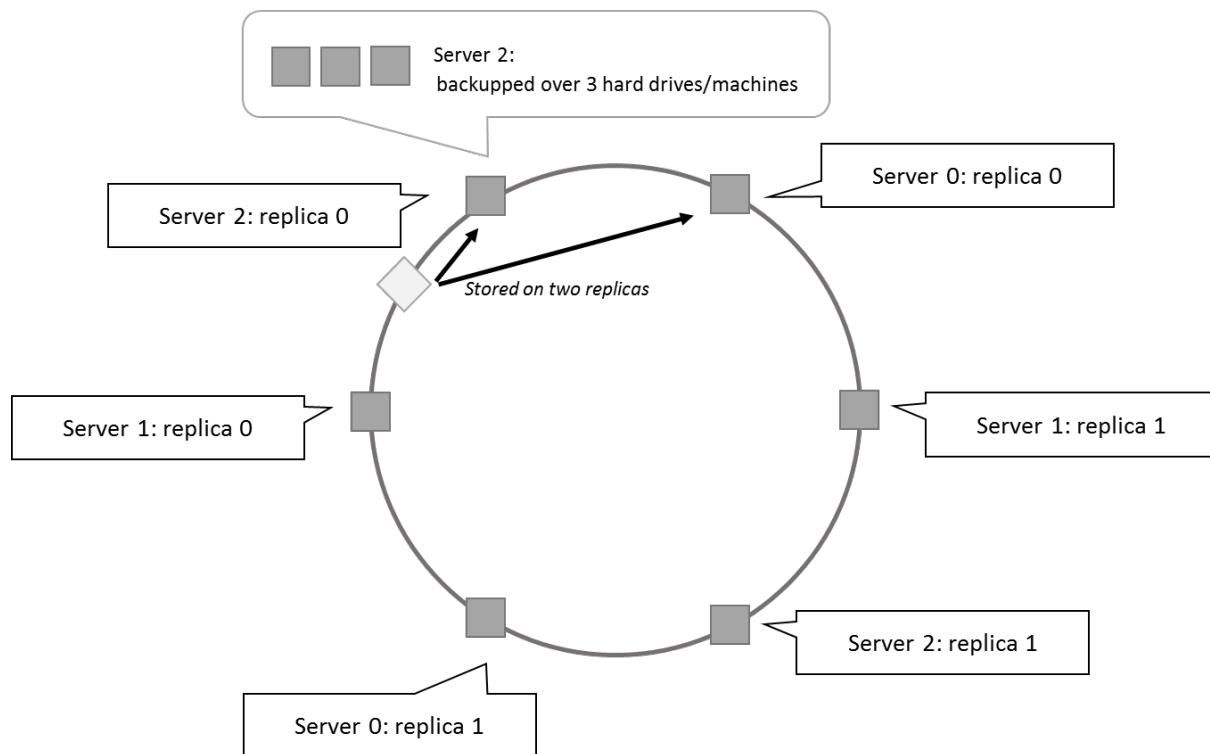
# Replication and Redundancy

- To handle data replication or redundancy, many vendors extend the consistent hashing mechanism so that key-value pairs are duplicated across multiple nodes
  - » E.g., by storing the key-value pair on two or more nodes clockwise from the key's position on the ring



# Replication and Redundancy

- It is also possible to set up a full redundancy scheme where each node itself corresponds to multiple physical machines each storing a fully redundant copy of the data





# Eventual Consistency

- Membership protocol does not guarantee that every node is aware of every other node *at all times*
  - » It will reach a consistent state over time
- State of the network might not be perfectly consistent at any moment in time, though will become eventually consistent at a future point in time
- Many NoSQL databases guarantee so called **eventual consistency**



# Eventual Consistency

- Most NoSQL databases follow the **BASE** principle
  - » Basically available, Soft state, Eventual consistency
- **CAP theorem** states that a distributed computer system cannot guarantee the following three properties at the same time:
  - » Consistency (all nodes see the same data at the same time)
  - » Availability (guarantees that every request receives a response indicating a success or failure result)
  - » Partition tolerance (the system continues to work even if nodes go down or are added).



# Eventual Consistency

- Most NoSQL databases sacrifice the consistency part of CAP in their setup, instead striving for eventual consistency
- The full BASE acronym stands for:
  - » Basically available: NoSQL databases adhere to the availability guarantee of the CAP theorem
  - » Soft state: the system can change over time, even without receiving input
  - » Eventual consistency: the system will become consistent over time



- The operation which repartitions hashes over nodes in case nodes or added or removed is called **stabilization**
- If a consistent hashing scheme being applied, the number of fluctuations in the hash-node mappings will be minimized.



- Key value stores represent a very diverse gamut of systems
- Full blown DBMSs versus caches
- Only limited query facilities are offered
  - » E.g. put and set
- Limited to no means to enforce structural constraints
  - » DBMS remains agnostic to the internal structure
- No relationships, referential integrity constraints or database schema, can be defined



- A **tuple store** is similar to a key-value store, with the difference that it does not store pairwise combinations of a key and a value, but instead stores a unique key together with a vector of data
- Example:
  - » marc -> ("Marc", "McLast Name", 25, "Germany")
- No requirement to have the same length or semantic ordering (schema-less!)



- Various NoSQL implementations do, however, permit organizing entries in semantical groups, (aka collections or tables)
- Examples:
  - » Person:marc -> ("Marc", "McLast Name", 25, "Germany")
  - » Person:harry -> ("Harry", "Smith", 29, "Belgium")



- **Document stores** store a collection of attributes that are labeled and unordered, representing items that are semi-structured
- Example:

```
{  
    Title      = "Harry Potter"  
    ISBN       = "111-1111111111"  
    Authors    = [ "J.K. Rowling" ]  
    Price      = 32  
    Dimensions = "8.5 x 11.0 x 0.5"  
    PageCount  = 234  
    Genre      = "Fantasy"  
}
```



- Most modern NoSQL databases choose to represent documents using JSON

```
{  
    "title": "Harry Potter",  
    "authors": ["J.K. Rowling", "R.J. Kowling"],  
    "price": 32.00,  
    "genres": ["fantasy"],  
    "dimensions": {  
        "width": 8.5,  
        "height": 11.0,  
        "depth": 0.5  
    },  
    "pages": 234,  
    "in_publication": true,  
    "subtitle": null  
}
```



- Items with Keys
- Filters and Queries
- Complex Queries and Aggregation with MapReduce
- SQL After all ...



## Items with Keys

- Most NoSQL document stores will allow you to store items in tables (collections) in a schema-less manner, but will enforce that a primary key be specified
  - » E.g. Amazon's DynamoDB, MongoDB ( `_id` )
- Primary key will be used as a partitioning key to create a hash and determine where the data will be stored



# Filters and Queries

```
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoDatabase;
import java.util.ArrayList;
import static com.mongodb.client.model.Filters.*;
import static java.util.Arrays.asList;

public class MongoDBExample {
    public static void main(String... args) {
        MongoClient mongoClient = new MongoClient();
        MongoDatabase db = mongoClient.getDatabase("test");

        // Delete all books first
        db.getCollection("books").deleteMany(new Document());

// Add some books
        db.getCollection("books").insertMany(new ArrayList<Document>() {{
            add(getBookDocument("My First Book", "Wilfried", "Lemahieu", 12, new String[]{"drama"}));
            add(getBookDocument("My Second Book", "Seppe", "vanden Broucke", 437, new String[]{"fantasy", "thriller"}));
            add(getBookDocument("My Third Book", "Seppe", "vanden Broucke", 200, new String[]{"educational"}));
            add(getBookDocument("Java Programming", "Bart", "Baesens", 100, new String[]{"educational"}));
        }});
    }
}
```



# Filters and Queries

```
// Perform query
FindIterable<Document> result = db.getCollection("books").find(
    and(      eq("author.last_name", "vanden Broucke"),
              eq("genres", "thriller"),
              gt("nrPages", 100)));

for (Document r : result) {
    System.out.println(r.toString());
    // Increase the number of pages:
    db.getCollection("books").updateOne(
        new Document("_id", r.getId()),
        new Document("$set",
                     new Document("nrPages", r.getInteger("nrPages") + 100)));
}
mongoClient.close();

public static Document getBookDocument(String title,
    String authorFirst, String authorLast,
    int nrPages, String[] genres) {
    return new Document("author", new Document()
        .append("first_name", authorFirst)
        .append("last_name", authorLast))
        .append("title", title)
        .append("nrPages", nrPages)
        .append("genres", asList(genres));}}
```



# Filters and Queries

```
Document{{_id=567ef62bc0c3081f4c04b16c,  
author=Document{{first_name=Seppe, last_name=vanden Broucke}},  
title=My Second Book, nrPages=437, genres=[fantasy, thriller]}}
```



# Filters and Queries

```
// Perform aggregation query
AggregateIterable<Document> result = db.getCollection("books")
    .aggregate(asList(
        new Document("$group",
            new Document("_id", "$author.last_name")
                .append("page_sum", new Document("$sum",
"$nrPages")))));
for (Document r : result) {
    System.out.println(r.toString());
}
```

```
Document{{_id=Lemahieu, page_sum=12}}
Document{{_id=Vanden Broucke, page_sum=637}}
Document{{_id=Baesens, page_sum=100}}
```



- Queries can still be slow because every filter (such as “author.last\\_name = Baesens”) entails a complete collection or table scan
- Most document stores can define a variety of indexes
  - » unique and non-unique indexes
  - » compound indexes
  - » geospatial indexes
  - » text-based indexes



# Complex Queries and Aggregation with MapReduce

- Document stores do not support relations
- **First approach:** embedded documents

```
{  
    "title": "Databases for Beginners",  
    "authors": ["J.K. Sequel", "John Smith"],  
    "pages": 234  
}
```

```
{  
    "title": "Databases for Beginners",  
    "authors": [  
        {"first_name": "Jay Kay", "last_name": "Sequel", "age": 54},  
        {"first_name": "John", "last_name": "Smith", "age": 32}  
    ],  
    "pages": 234  
}
```

BUT: Data duplication!



# Complex Queries and Aggregation with MapReduce

- **Second approach:** create two collections

## book collection:

```
{  
    "title": "Databases for Beginners",  
    "authors": ["Jay Kay Rowling", "John Smith"],  
    "pages": 234  
}
```

## authors collection:

```
{  
    "_id": "Jay Kay Rowling",  
    "age": 54  
}
```

BUT: Need to resolve complex relational queries in application code!



## ▪ Third Approach: MapReduce

- » a map-reduce pipeline starts from a series of key-value pairs ( $k_1, v_1$ ) and maps each pair to one or more output pairs
- » the output entries are shuffled and distributed so that all output entries belonging to the same key are assigned to the same worker (e.g. physical machines)
- » workers then apply a reduce function to each group of key-value pairs having the same key, producing a new list of values per output key
- » the resulting, final outputs are then (optionally) sorted per key  $k_2$  to produce the final outcome



# Complex Queries and Aggregation with MapReduce

- Example: get a summed count of pages for books per genre
- Create a list of input keys-value pairs

k1	v1
1	{genre: education, nrPages: 120}
2	{genre: thriller, nrPages: 100}
3	{genre: fantasy, nrPages: 20}
...	...

- Map function is a simple conversion to a genre-nrPages key-value pair

```
function map(k1, v1)
    emit output record (v1.genre, v1.nrPages)
end function
```



# Complex Queries and Aggregation with MapReduce

- Workers have produced the following three output lists, with the keys corresponding to genres

Worker 1	
k2	v2
education	120
thriller	100
fantasy	20

Worker 2	
k2	v2
drama	500
education	200

Worker 3	
k2	v2
education	20
fantasy	10

- A working operation will be started per unique key k2, for which its associated list of values will be reduced
  - » E.g., (education,[120,200,20]) will be reduced to its sum, 340

```
function reduce(k2, v2_list)
    emit output record (k2, sum(v2_list))
end function
```



# Complex Queries and Aggregation with MapReduce

- Final output looks as

k2	v3
education	340
thriller	100
drama	500
fantasy	30

- Can be sorted based on k2 or v3



# Complex Queries and Aggregation with MapReduce

- Suppose we would now like to retrieve an average page count per book for each genre

- Reduce function becomes

```
function reduce(k2, v2_list)
    emit output record (k2, sum(v2_list) / length(v2_list))
end function
```

- After mapping the input list, workers produce the following three output lists

Worker 1	
k2	v2
education	120
thriller	100
fantasy	20

Worker 2	
k2	v2
drama	500
education	200

Worker 3	
k2	v2
education	20
fantasy	10



# Complex Queries and Aggregation with MapReduce

- Average as follows

k2	v3
education	$(120 + 200 + 20) / 3 = 113.33$
thriller	$100 / 1 = 100.00$
drama	$500 / 1 = 500.00$
fantasy	$(20 + 10) / 2 = 15.00$

- Note: reduce-operation can happen more than once, and can already start before all mapping operations have finished!
  - » Need to ensure that results are correct by rewriting map and reduce functions!



# Complex Queries and Aggregation with MapReduce

```
function map(k1, v1)
    emit output record (v1.genre, (v1.nrPages, 1))
end function

function reduce(k2, v2_list)
    for each (nrPages, count) in v2_list do
        s = s + nrPages * count
        newc = newc + count
    repeat
        emit output record (k2, (s/newc, newc))
    end function
```



# Complex Queries and Aggregation with MapReduce

- Example: count the number of occurrences per word in a document

```
function map(document_name, document_text)
    for each word in document_text do
        emit output record (word, 1)
    repeat
end function
```

```
function reduce(word, partial_counts)
    emit output record (word, sum(partial_counts))
end function
```



# Complex Queries and Aggregation with MapReduce

- Example: return the average number of pages per genre, but now taking into account that books can have more than one genre associated to them (in MongoDB)



# Complex Queries and Aggregation with MapReduce

```
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.client.MongoDatabase;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import static java.util.Arrays.asList;

public class MongoDBAggregationExample {
    public static Random r = new Random();

    public static void main(String... args) {
        MongoClient mongoClient = new MongoClient();
        MongoDatabase db = mongoClient.getDatabase("test");

        setupDatabase(db);
        for (Document r : db.getCollection("books").find())
            System.out.println(r);

        mongoClient.close();}
}
```



# Complex Queries and Aggregation with MapReduce

```
public static void setupDatabase(MongoDatabase db) {
    db.getCollection("books").deleteMany(new Document());

    String[] possibleGenres = new String[] {
        "drama", "thriller", "romance", "detective",
        "action", "educational", "humor", "fantasy" };

    for (int i = 0; i < 100; i++) {
        db.getCollection("books").insertOne(
            new Document("_id", i)
            .append("nrPages", r.nextInt(900) + 100)
            .append("genres",
                getRandom(asList(possibleGenres), r.nextInt(3) + 1)));
    }
}
```



# Complex Queries and Aggregation with MapReduce

```
public static List<String> getRandom(List<String> els, int number) {  
    List<String> selected = new ArrayList<>();  
    List<String> remaining = new ArrayList<>(els);  
    for (int i = 0; i < number; i++) {  
        int s = r.nextInt(remaining.size());  
        selected.add(remaining.get(s));  
        remaining.remove(s);  
    }  
    return selected;  
}  
}
```

Document{{\_id=0, nrPages=188, genres=[action, detective, romance]}}

Document{{\_id=1, nrPages=976, genres=[romance, detective, humor]}}

Document{{\_id=2, nrPages=652, genres=[thriller, fantasy, action]}}

Document{{\_id=3, nrPages=590, genres=[fantasy]}}

Document{{\_id=4, nrPages=703, genres=[educational, drama, thriller]}}

Document{{\_id=5, nrPages=913, genres=[detective]}}

...



# Complex Queries and Aggregation with MapReduce

- Manual construction of the aggregation query looks as follows

```
public static void reportAggregate(MongoDatabase db) {  
    Map<String, List<Integer>> counts = new HashMap<>();  
    for (Document r : db.getCollection("books").find()) {  
        for (Object genre : r.get("genres", List.class)) {  
            if (!counts.containsKey(genre.toString()))  
                counts.put(genre.toString(), new ArrayList<Integer>());  
            counts.get(genre.toString()).add(r.getInteger("nrPages"));  
        }  
    }  
    for (Entry<String, List<Integer>> entry : counts.entrySet()) {  
        System.out.println(entry.getKey() + " --> AVG = " +  
                           sum(entry.getValue()) / (double)  
                           entry.getValue().size());  
    }  
  
    private static int sum(List<Integer> value) {  
        int sum = 0;  
        for (int i : value) sum += i;  
        return sum;  
    }  
}  
  
romance --> AVG = 497.39285714285717  
drama --> AVG = 536.88  
detective --> AVG = 597.1724137931035  
humor --> AVG = 603.5357142857143  
fantasy --> AVG = 540.0434782608696  
educational --> AVG = 536.1739130434783  
action --> AVG = 398.9032258064516  
thriller --> AVG = 513.5862068965517
```



# Complex Queries and Aggregation with MapReduce

- In case the list of genres is known beforehand, we can optimize by performing the aggregation per genre directly in MongoDB itself

```
public static void reportAggregate(MongoDatabase db) {  
    String[] possibleGenres = new String[] {  
        "drama", "thriller", "romance", "detective",  
        "action", "educational", "humor", "fantasy" };  
  
    for (String genre : possibleGenres) {  
        AggregateIterable<Document> iterable =  
            db.getCollection("books").aggregate(asList(  
                new Document("$match", new Document("genres", genre)),  
                new Document("$group", new Document("_id", genre)  
                    .append("average", new Document("$avg", "$nrPages"))));  
  
        for (Document r : iterable) {  
            System.out.println(r);  
        }  
    }  
}
```

Document{{\_id=drama, average=536.88}}  
Document{{\_id=thriller, average=513.5862068965517}}  
Document{{\_id=romance, average=497.39285714285717}}  
Document{{\_id=detective, average=597.1724137931035}}  
Document{{\_id=action, average=398.9032258064516}}  
...



# Complex Queries and Aggregation with MapReduce

- Assume now that we have millions of books in our database and we do not know the number of genres beforehand → use Map Reduce
- Map in MongoDB:

```
function() {  
    // No arguments, use “this” to refer to the  
    // local document item being processed  
    emit(key, value);  
}
```

- Reduce in MongoDB:

```
function(key, values) {  
    return result;  
}
```



# Complex Queries and Aggregation with MapReduce

- **Map function**

```
function() {  
    var nrPages = this.nrPages;  
    this.genres.forEach(function(genre) {  
        emit(genre, {average: nrPages, count: 1});  
    });  
}
```

- **Reduce function**

```
function(genre, values) {  
    var s = 0;  
    var newc = 0;  
    values.forEach(function(curAvg) {  
        s += curAvg.average * curAvg.count;  
        newc += curAvg.count;  
    });  
    return {average: (s / newc), count: newc};  
}
```



# Complex Queries and Aggregation with MapReduce

```
public static void reportAggregate(MongoDatabase db) {  
    String map = "function() { " +  
        "    var nrPages = this.nrPages; " +  
        "    this.genres.forEach(function(genre) { " +  
        "        emit(genre, {average: nrPages, count: 1}); " +  
        "    }); " +  
        "}; "  
  
    String reduce = "function(genre, values) { " +  
        "    var s = 0; var newc = 0; " +  
        "    values.forEach(function(curAvg) { " +  
        "        s += curAvg.average * curAvg.count; " +  
        "        newc += curAvg.count; " +  
        "    }); " +  
        "    return {average: (s / newc), count: newc}; " +  
        "}; "  
  
    MapReduceIterable<Document> result = db.getCollection("books")  
        .mapReduce(map, reduce);  
    for (Document r : result)  
        System.out.println(r);}
```



# Complex Queries and Aggregation with MapReduce

```
Document{{_id=action, value=Document{{average=398.9032258064516,  
count=31.0}}}}  
Document{{_id=detective, value=Document{{average=597.1724137931035,  
count=29.0}}}}  
Document{{_id=drama, value=Document{{average=536.88, count=25.0}}}}  
Document{{_id=educational, value=Document{{average=536.1739130434783,  
count=23.0}}}}  
Document{{_id=fantasy, value=Document{{average=540.0434782608696,  
count=23.0}}}}  
Document{{_id=humor, value=Document{{average=603.5357142857143,  
count=28.0}}}}  
Document{{_id=romance, value=Document{{average=497.39285714285717,  
count=28.0}}}}  
Document{{_id=thriller, value=Document{{average=513.5862068965517,  
count=29.0}}}}
```



- GROUP BY style SQL queries are convertible to an equivalent map-reduce pipeline
- Many document store implementations express queries using an SQL interface
- Couchbase, also allows to define foreign keys and perform join operations

```
SELECT books.title, books.genres,  
authors.name  
FROM books  
JOIN authors ON KEYS books.authorId
```



- Many RDBMS vendors start implementing NoSQL by
  - » Focusing on horizontal scalability and distributed querying
  - » Dropping schema requirements
  - » Support for nested data types or allowing to store JSON directly in tables
  - » Support for Map-Reduce operations
  - » Support for special data types, such as geospatial data



- A **column-oriented DBMS** is a database management system that stores data tables as sections of columns of data
- Useful if
  - » aggregates are regularly computed over large numbers of similar data items
  - » data is sparse, i.e. columns with many null values
- Can also be an RDBMS, key-value or document store



# Column-oriented Databases

## ■ Example

		Title	Price	Audiobook
1	fantasy	My first book	20	30
2	education	Beginners guide	10	null
3	education	SQL strikes back	40	null
4	fantasy	The rise of SQL	10	null

- Row based databases are not efficient at performing operations that apply to the entire data set
  - » Need indexes which add overhead



# Column-oriented Databases

- In a column-oriented database, all values of a column are placed together on disk

Genre: fantasy:1,4 education:2,3

Title: My first book:1      Beginners guide:2    SQL strikes back:3    The rise of SQL:4

Price: 20:1      10:2,4      40:3

Audiobook price: 30:1

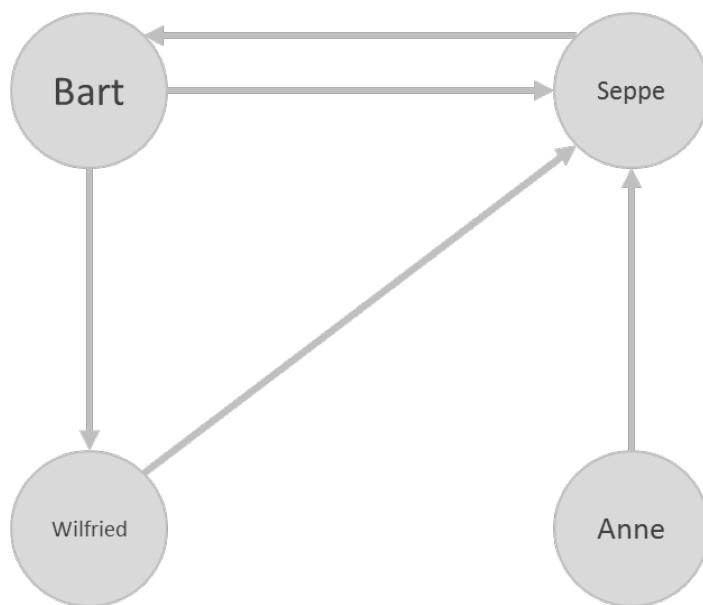
- A column matches the structure of a normal index in a row-based system
- Operations such as: find all records with price equal to 10 can now be executed directly
- Null values do not take up storage space anymore



- Disadvantages
  - » Retrieving all attributes pertaining to a single entity becomes less efficient
  - » Join operations will be slowed down
- Examples
  - » Google BigTable, Cassandra, HBase, and Parquet



- **Graph databases** apply graph theory to the storage of information of records
- Graphs consist of **nodes** and **edges**





# Graph based databases

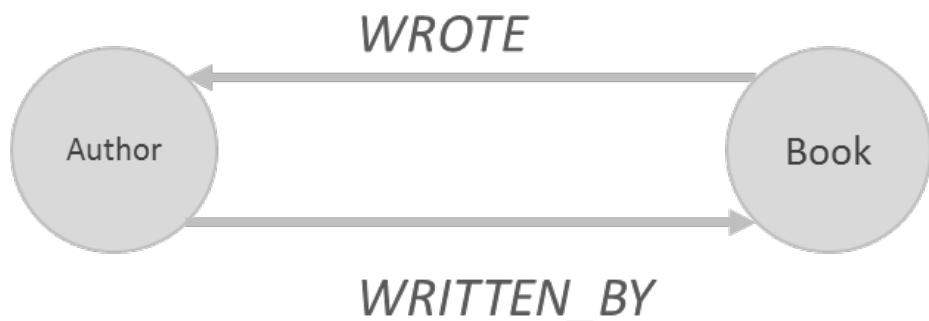
- One-to-one, one-to-many, and many-to-many structures can easily be modeled in a graph
- Consider N-M relationship between books and authors
- RDBMS needs 3 tables: Book, Author and Books\_Authors
- SQL query to return all book titles for books written by a particular author would look like follows

```
SELECT title
FROM books, authors, books_authors
WHERE author.id = books_authors.author_id
      AND books.id = books_authors.book_id
      AND author.name = "Bart Baesens"
```



# Graph based databases

- In a graph database (using **Cypher query language** from Neo4j)



```
MATCH (b:Book)-[:WRITTEN_BY]-(a:Author)  
WHERE a.name = "Bart Baesens"  
RETURN b.title
```



- A graph database is a hyper-relational database, where JOIN tables are replaced by more interesting and semantically meaningful relationships that can be navigated and/or queried using graph traversal based on graph pattern matching.



# Graph based databases

- Cypher Overview (Neo4j)
- Exploring a Social Graph



# Cypher Overview

- Cypher is a declarative, text-based query language, containing many similar operations as SQL
- Contains a special **MATCH** clause to match those patterns using symbols that look like graph symbols as drawn on a whiteboard
- Nodes are represented by parentheses, representing a circle: ()
- Nodes can be labeled in case they need to be referred to elsewhere, and be further filtered by their type, using a colon: (b:Book)
- Edges are drawn using either -- or -->, representing a unidirectional line or an arrow representing a directional relationship respectively



- Relationships can be filtered by putting square brackets in the middle:  
`(b:Book)<- [:WRITTEN_BY]- (a:Author)`



# Cypher Overview

```
MATCH (b:Book)
```

```
RETURN b;
```

```
MATCH (b:Book)
```

```
RETURN b
```

```
ORDER BY b.price DESC
```

```
LIMIT 20;
```

```
MATCH (b:Book)
```

```
WHERE b.title = "Beginning Neo4j"
```

```
RETURN b;
```

```
MATCH (b:Book {title:"Beginning Neo4j"})
```

```
RETURN b;
```



# Cypher Overview

- JOIN clauses are expressed using direct relational matching

```
MATCH (c:Customer)-[p:PURCHASED]->(b:Book)<-
  [:WRITTEN_BY]-(a:Author)
WHERE a.name = "Wilfried Lemahieu"
      AND c.age > 30
      AND p.type = "cash"
RETURN DISTINCT c.name;
```



# Cypher Overview

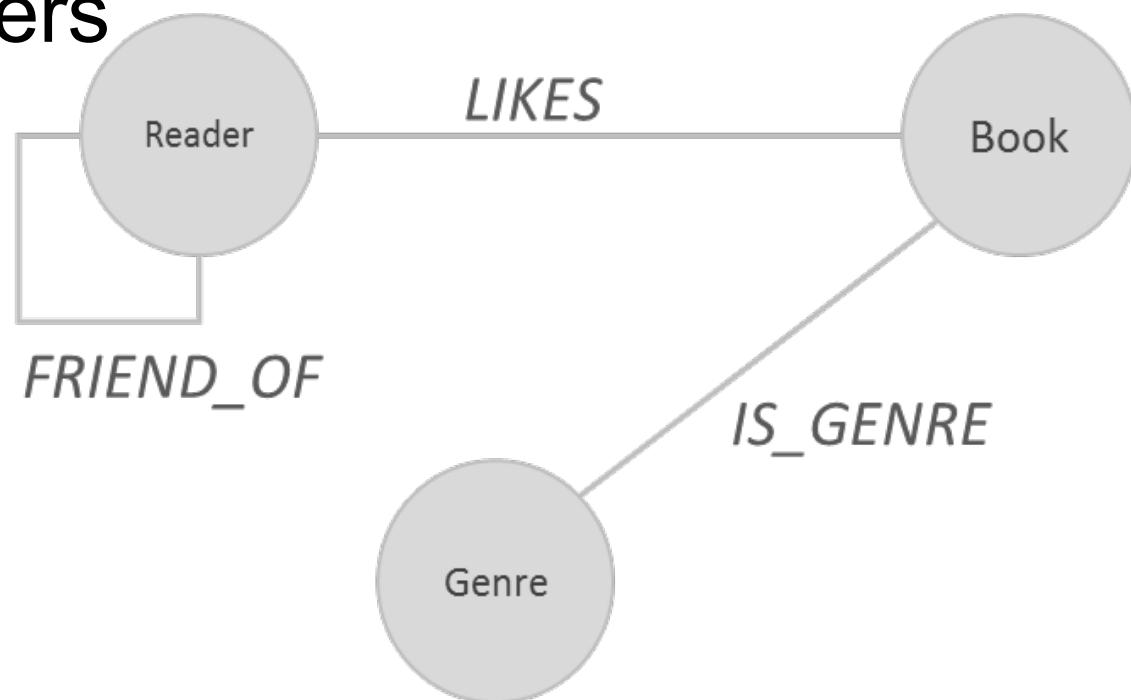
- Graph databases are great at managing tree structures
- Example:
  - » tree of book genres, and books can be placed under any category level
  - » a query to fetch a list of all books in the category “Programming” and all its subcategories
- Cypher can express queries over hierarchies and transitive relationships of any depth simply by appending a star \* after the relationship type and providing optional min..max limits

```
MATCH (b:Book)-[ :IN_GENRE ]->(:Genre)
          -[ :PARENT*0.. ]-(:Genre
{name:"Programming"})
RETURN b.title;
```



# Exploring a Social Graph

- Example: a social graph for a book reading club, modeling genres, books and readers



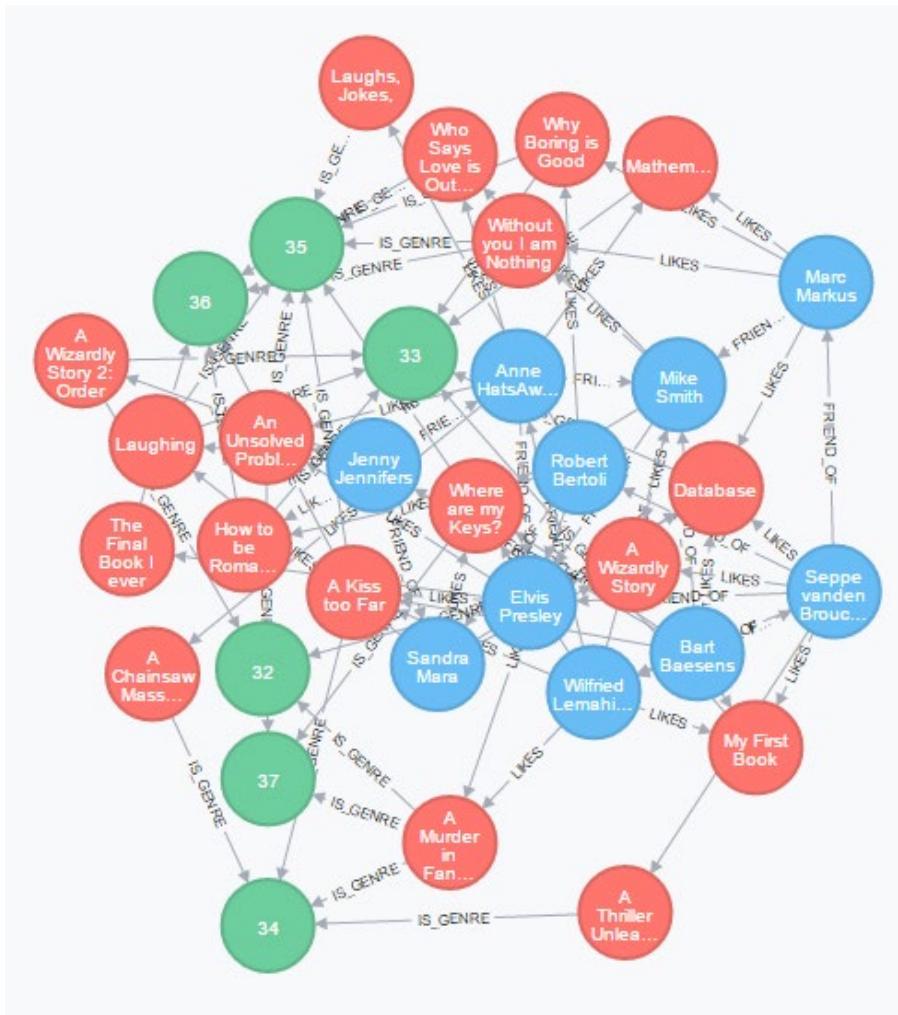


# Exploring a Social Graph

```
CREATE (Bart:Reader {name:'Bart Baesens', age:32})
CREATE (Seppe:Reader {name:'Seppe vanden Broucke', age:30})
...
CREATE (Fantasy:Genre {name:'fantasy'})
CREATE (Education:Genre {name:'education'})
...
CREATE (b01:Book {title:'My First Book'})
CREATE (b02:Book {title:'A Thriller Unleashed'})
...
CREATE
  (b01)-[:IS_GENRE]->(Education),
  (b02)-[:IS_GENRE]->(Thriller),
...
CREATE
  (Bart)-[:FRIEND_OF]->(Seppe),
  (Bart)-[:FRIEND_OF]->(Wilfried),
...
CREATE
  (Bart)-[:LIKES]->(b01), (Bart)-[:LIKES]->(b03),
  (Bart)-[:LIKES]->(b05), (Bart)-[:LIKES]->(b06),
...
...
```



# Exploring a Social Graph





# Exploring a Social Graph

- Who likes romance books?

```
MATCH (r:Reader)--(:Book)--(:Genre  
{name:'romance'})  
RETURN r.name
```

Returns:

Elvis Presley

Mike Smith

Anne HatsAway

Robert Bertoli

...



# Exploring a Social Graph

- Who are Bart's friends that liked Humor books?

```
MATCH (me:Reader)--(friend:Reader)--(b:Book)--(g:Genre)
WHERE g.name = 'humor' AND me.name = 'Bart Baesens'
RETURN DISTINCT friend.name
```

- Can you recommend some humor books that Seppe's friends liked and Seppe has not liked yet?

```
MATCH (me:Reader)--(friend:Reader),
      (friend)--(b:Book),
      (b)--(genre:Genre)
WHERE NOT (me)--(b)
      AND me.name = 'Seppe vanden Broucke' AND genre.name = 'humor'
RETURN DISTINCT b.title
```



# Exploring a Social Graph

- Get a list of people who have liked books Bart liked, sorted by most liked books in common

```
MATCH (me:Reader)--(b:Book),  
      (me)--(friend:Reader)--(b)  
WHERE me.name = 'Bart Baesens'  
RETURN friend.name, count(*) AS common_likes  
ORDER BY common_likes DESC
```

friend.name	common_likes
Wilfried Lemahieu	3
Seppe vanden Broucke	2
Mike Smith	1



- Location-based services
- Recommender systems
- Social media (e.g. Twitter and FlockDB)
- Knowledge based systems



## Other NoSQL Categories

- XML databases
- OO databases
- Database systems to deal with time series and streaming events
- Database systems to store and query geospatial data
- Database systems such as BayesDB which let users query the probable implication of their data



- Most NoSQL implementations have yet to prove their true worth in the field
- Some queries or aggregations particularly difficult with Map-Reduce interfaces harder to learn and use
- Some early-adaptors of NoSQL were confronted with some sour lessons
  - » E.g. Twitter and HealthCare.gov



- NoSQL vendors start focusing again on robustness and durability whereas RDBMS vendors start implementing features to build schema-free, scalable data stores
- NewSQL: blend the scalable performance and flexibility of NoSQL systems with the robustness guarantees of a traditional RDBMS



# Evaluating NoSQL DBMSs

	RDBMSs	NoSQL databases	NewSQL
<b>Relational</b>	Yes	No	Yes
<b>SQL</b>	Yes	No	Yes
<b>Column stores</b>	No	Yes	Yes
<b>Scalability</b>	Limited	Yes	Yes
<b>Eventually consistent</b>	Yes	Yes	Yes
<b>BASE</b>	No	Yes	No
<b>Big volumes of data</b>	No	Yes	Yes
<b>Schema-less</b>	No	Yes	No

# Agenda

1 Session Overview

2 Legacy Databases

3 OODBs and Object Persistence

4 Extended Relational Databases

5 XML Databases

6 NoSQL Databases

7 Summary and Conclusion



# Summary (1 of 2)

- Legacy Databases
  - Hierarchical Model
  - CODASYL Model
- OODBs and Object Persistence
  - Recap: Basic Concepts of OO
  - Advanced Concepts of OO
  - Basic Principles of Object Persistence
  - OODBMS
  - Evaluating OODBMSs
- Extended Relational Databases
  - Success of the relational model
  - Limitations of the Relational Model
  - Active RDBMS Extensions
  - Object-Relational RDBMS extensions
  - Recursive SQL queries

## Summary (2 of 2)

- XML Databases
  - Extensible Markup Language
  - Processing XML Documents
  - Storage of XML Documents
  - Differences between XML and Relational Data
  - Mappings Between XML Documents and (Object-) Relational Data
  - Searching XML Data
  - XML for Information Exchange
  - Other Data Representation Formats
- NoSQL Databases
  - The NoSQL movement
  - Key-Value stores
  - Tuple and Document stores
  - Column-oriented databases
  - Graph based databases
  - Other NoSQL categories

# Any Questions?

