COS 326
Database Systems

Lecture 2

Object-oriented Databases (1)

Chapter 12 (7th edition)

Thursday 25 July 2024

In this lecture

- Introduction to OODBMSs
- Limitations (‘weaknesses’) of Relational DB systems
- Storing Objects in Relational Databases
- OODB and OODBMS
- Commercial & Open-Source OODBMSs
- Alternative strategies for OODBMSs

Limitations ('weaknesses') of RDBMSs (1)

1. Poor representation of "real world" entities

- **Normalization** leads to relations that do not correspond to entities in the "real world"
- Fragmentation of the "real world"

- **Information** is most commonly **obtained** from the DB through performing **a join (intensive computation) of 2 or more tables**

Staff

staffNo	sName	position	salary	branchNo
SL21	John White	Manager	30000	B005
SG37	Ann Beech	Assistant	12000	B003
SG14	David Ford	Supervisor	18000	B003
SA9	Mary Howe	Assistant	9000	B007
SG5	Susan Brand	Manager	24000	B003
SL41	Julie Lee	Assistant	9000	B005

Branch

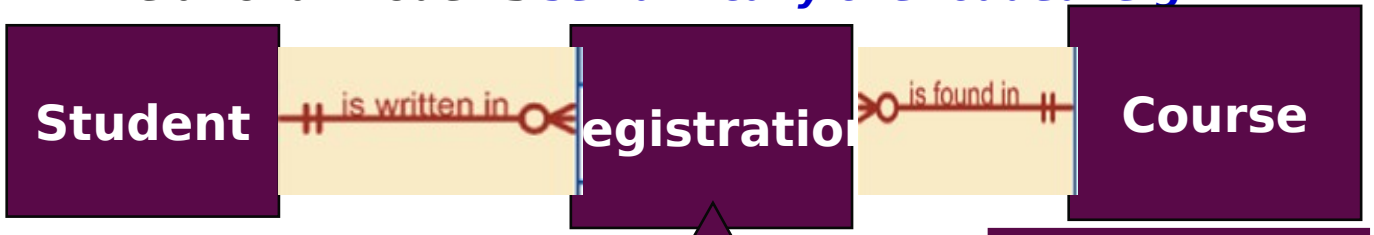
branchNo	bAddress
B005	22 Deer Rd, London
B007	16 Argyll St, Aberdeen
B003	163 Main St, Glasgow

e.g. JOIN required to see branch address together with staff member's name

Limitations ('weaknesses') of RDBMSs (2)

2. Semantic Overloading

- **Single construct for representing data and data relationships: *the relation***
- **No mechanism to distinguish between relations & relationships or between different types of relationships**
- **Relational model is *semantically overloaded e.g. ERD***



relationship 'is written in' is not recorded in Relational DB

This is a relation and a

relationship 'is found in' is not recorded in Relational DB

Limitations ('weaknesses') of RDBMSs (3)

3. Limited Operations

- RDBMSs only have a fixed set of operations (using SQL) which cannot be extended
  - i.e. **operations on sets of tuples as defined in the Relational calculus & algebra**
- Not possible for software developers to specify new operations
  - **this is in contrast to OOP methods**

BUT

  - **Database programmers can create stored procedures on the server side.**

Limitations ('weaknesses') of RDBMSs (4)

4. Homogeneous data structure

- **Horizontal and vertical homogeneity is restrictive for 'real world' objects**

- **Horizontal homogeneity**
  - **each tuple (row) composed of the same attributes**
- **Vertical homogeneity**
  - **values of an attribute (column) come from same domain**
- **Intersection of row and column must be atomic**
  - **cannot store an object that can be decomposed into its components**
  - **Note: can store a BLOB or CLOB (pointer to a file containing object) for viewing purposes**

position	salary
Manager	30000
Assistant	12000
Supervisor	18000
Assistant	9000
Manager	24000
Assistant	9000

Limitations (‘weaknesses’) of RDBMSs (5)

5. Impedance

- *Physics*
  - a measure of the opposition (resistance) to the flow of a current in an electrical circuit *(I = V / R)*
- *Software engineering*
  - a measure of the opposition (difficulty) caused by differences between two programming paradigms

- **Embedded SQL**
  - for applications that access database data
  - SQL standard provides embedded SQL
  - creates problem of impedence mismatch for software developers: Mixing different programming paradigms

(6) e.g.Database application - JDBCSample.java

```
import java.sql.*
public class JDBCSample {
public static void main( String args[ ]) {
String connectionURL =
     "jdbc:postgresql://localhost:5432/movies;user=java;password=samples";
try {
   Class.forName("org.postgresql.Driver");
   Connection con = DriverManager.getConnection (connectionURL);
   Statement stmt = con.createStatement();
   ResultSet rs =
     stmt.executeQuery("SELECT moviename, releasedate FROM movies");
   while (rs.next())
      System.out.println("Name= " + rs.getString("moviename")
                                    + " Date= " + rs.getString("releasedate");
}
 finally    {  con.close();    }
```

Limitations ('weaknesses') of RDBMSs (7)

6. RDBMS supports short-lived transactions

- Transaction is: an indivisible sequence of DB operations
- e.g BEGIN TRANSACTION

  SQL statements for transaction go here

  END TRANSACTION

Short-lived (business) vs long-duration (design)

7. Schema changes create problems

- Database change requires program changes
- e.g. of change: add new attributes (columns) to a table

Storing Objects in Relational Databases

- **Storing objects in relational DB:**
  - **An approach to achieving persistence with an OOP language:**
  - **use RDBMS as underlying storage engine**

- **Mapping class instances (i.e. objects) to one or more rows distributed over one or more relations**
  - **write code to convert between objects and rows**

- **UN-NECESSARY WORK THAT CAN BE AVOIDED**

e.g. a STUDENT registers for COURSES for a DEGREE PROGRAM

Object-oriented Databases (1)

- **Originates from Engineering & design domains**

  e.g. CAD (Computer Aided Design) database stores data relating to engineering designs

- **e.g. of designs**

  (1) Cars

  (2) Buildings

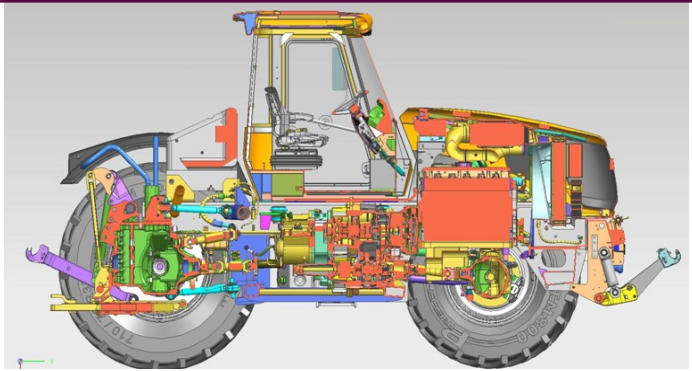
Using AutoCAD, Revit, etc.

- **OODMS market is small compared to RDBMS market**

- **Some common characteristics for design data:**

(1) design has many parts & interdependent subsystems

(2) large number of types & small number of instances cf. relational: few relations, many tuples



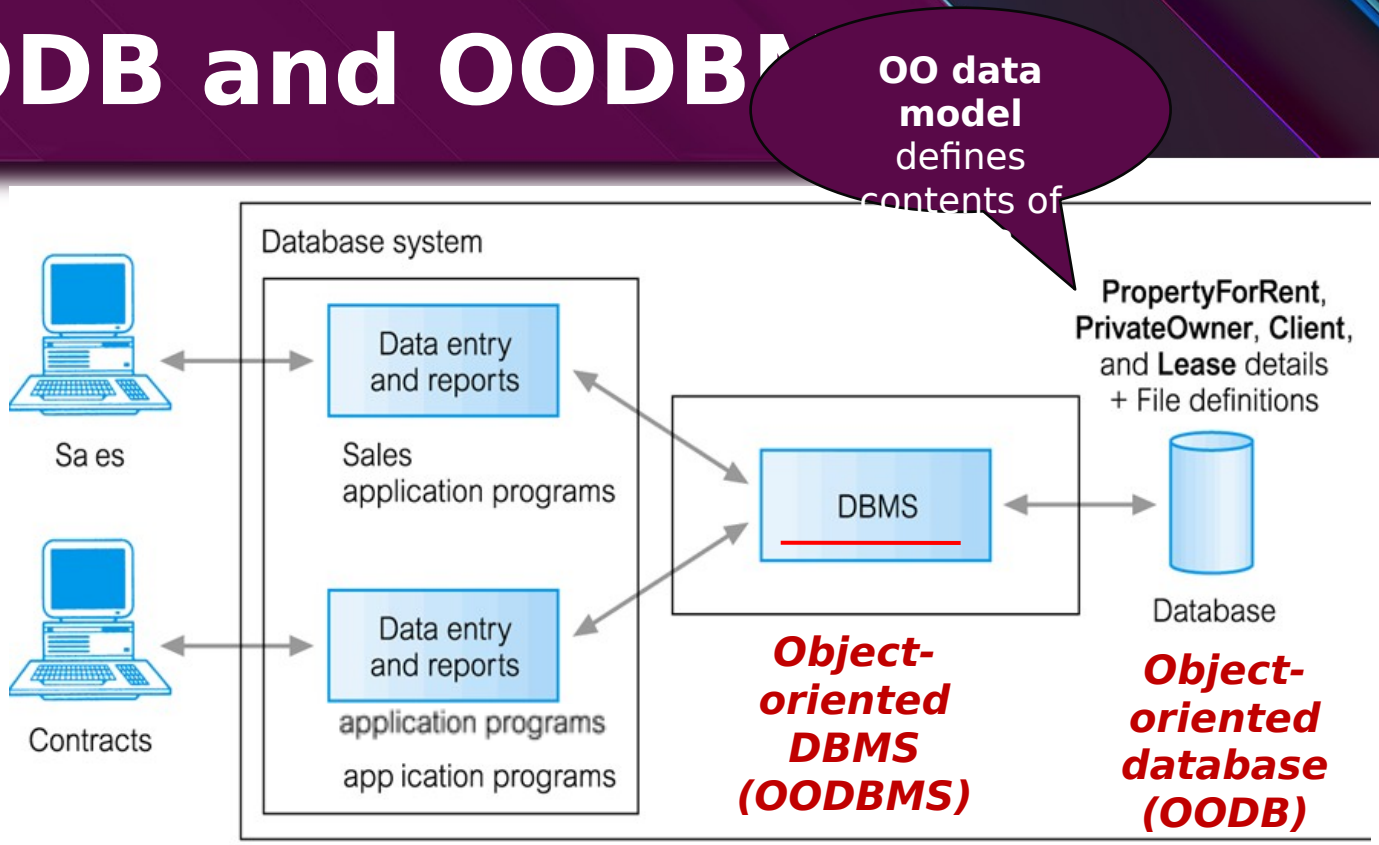
Object-oriented Databases (2)

- **Object databases (ODB)**
  - **Object data management systems (ODMS)**
  - Meet some of the needs of more complex applications
  - Specify:
    - Structure of complex objects
    - Operations that can be applied to these objects

Object-oriented DB Terminology

- **OO data model**
  - (logical) data model that captures the semantics of objects supported in object-oriented programming
- **OO database**
  - persistent sharable collection of objects defined by an OO data model
- **OODBMS**
  - manager (software) of an OODB

OODB and OODBM



- **OO data model**
  - **(logical) data model that supported in object-oriented programming**

OODBMSs

- **Commercial**
  - ***GemStone*** **from Gemstone Systems Inc.**
  - ***Objectivity/DB*** **from Objectivity Inc.**
  - ***ObjectStore*** **from Progress Software Corp.**

- **Open source**
  - ***ZooDB*** **(by Tilmann Zäschke)**
  - **ObjectDB (by ObjectDB Corp)**
  - **ConceptBase (by University of Skövde)**

Strategies for developing an OODBMS

1. **Extend existing object-oriented programming language**
   - **add traditional database capabilities to existing OOP language,**
     - **e.g. Smalltalk, C++, Java**
     - **e.g. *GemStone* extended Smalltalk, C++, Java**

2. **Provide extensible OODBMS library**
   - **No extensions to the OOP language**
   - **provide off-the-shelf library**
   - **Library provides persistence and database capabilities**
   - **Work in known programming language with same compiler**
   - **examples are: ObjectStore (for C++) and ObjectDB(for Java)**

Strategies for developing an OODBMS

Reading for student

3. **Imbed OODB language constructs** in conventional host language: e.g. *O_2* : extensions for C

4. **Extend existing database language (SQL)** with object-oriented capabilities
   - SQL (1999) by ISO supports OO features
   - Object SQL standard by ODMG
   - *Ontos* and *Versant* provide a version of Object SQL

5. **Develop novel database data model/language**
   - No history / baggage, start from scratch
   - No installed base, no skills
   - *Semantic Information Manager (SIM)*

COS 326
Database Systems

Lecture 3

Object-oriented Databases (2)

Chapter 12 (7th edition)

Tuesday 30 July 2024

In this lecture

- Case studies of OODBMSs
- ObjectDB
- Objectstore
- OO Database design
- Reading for the student
- How do OODBMSs overcome the limitations (weaknesses) of RDBMSs as discussed previously?

Recap: Strategies for developing an OODBMS

One of the strategies is:

To provide an extensible OODBMS library

- ‘Off-the-shelf’ library
- Library provides persistence and database capabilities

- **No need for separate compiler**
- Work with known programming languages e.g Java or C++
- **Examples of OODBMSs are**:
  - **ObjectDB : (Java) open source**
  - **ObjectStore (C++) commercial**

OODBMS Case study: ObjectDB for Java (1)

- **ObjectDB**
  - ObjectDB is a powerful Object-Oriented Database Management System (ODBMS).
  - ObjectDB provides all the standard database management services (storage and retrieval, transactions, lock management, query processing, etc.) in a way that makes development easier and applications faster.

– **Features**
  - 100% pure Java Object-Oriented Database Management System (ODBMS).
  - No proprietary API - managed only by standard Java APIs (JPA 2 / JDO 2).
  - Suitable for database files ranging from kilobytes to terabytes.
  - Supports both Client-Server mode and Embedded mode.
  - Single JAR with no external dependencies.
  - Database is stored as a single file.
  - Advanced querying and indexing capabilities.
  - Effective in heavy loaded multi-user environments.
  - Can easily be embedded in applications of any type and size

OODBMS Case study: ObjectDB for Java (2)

- **Object Identifier (OID)**
- **One goal of an OODB is to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be identified and operated upon.**
- **Hence, a unique identity is assigned to each independent object stored in the database.**
- **This unique identity is typically implemented via a unique, system-generated object identifier (OID).**

OODBMS Case study: ObjectDB for Java (3)

- **Defining a JPA Entity Class**
- **Jakarta Persistence API (formerly Java Persistence API) defines a standard for management of persistence and object/relational mapping in Java(R) environments.**
- **A JPA entity class is a POJO (Plain Old Java Object) class, i.e. an ordinary Java class that is marked (annotated) as having the ability to represent objects in the database.**

OODBMS Case study: ObjectDB for Java (4)

- **The Point Entity Class**
- **The following Point class, which represents points in the plane, is marked as an entity class,**
- **Provides the ability to store Point objects in the database and retrieve Point objects from the database**

ObjectDB– example from the ObjectDB Manual

```
package com.objectdb.tutorial;
import javax.persistence.Entity;
@Entity
public class Point {
    // Persistent Fields:
    private int x;
    private int y;
    // Constructor:
    Point (int x, int y) {
        this.x = x;
        this.y = y;
    }
    // Accessor Methods:
    public int getX() { return this.x; }
    public int getY() { return this.y; }
    // String Representation:
    @Override
    public String toString() {
        return String.format("(%d, %d)", this.x, this.y);
    }
}
```

OODBMS Case study: ObjectDB for Java (5)

- **Database Connections in Object DB**
- **In JPA a database connection is represented by the EntityManager interface. Therefore, in order to manipulate an ObjectDB database we need an EntityManager instance.**
- **Operations that modify database content also require an EntityTransaction instance.**

OODBMS Case study: ObjectDB for Java (6)

Obtaining an EntityManagerFactory

- **Obtaining an EntityManager instance consists of two steps.**
- **1. Obtain an instance of EntityManagerFactory that represents the relevant database**
- **2. Use that factory instance to get an**

```
EntityManagerFactory emf =
     Persistence.createEntityManagerFactory(
          "objectdb:$objectdb/db/points.odb");
```

OODBMS Case study: ObjectDB for Java (7)

- **Obtaining an EntityManager**
- **Once we have an EntityManagerFactory we can easily obtain an EntityManager instance:**

```
EntityManager em = emf.createEntityManager();
```

- **The EntityManager instance represents a connection to the database.**
- **When using JPA, every operation on a database is associated with an EntityManager.**

OODBMS Case study: ObjectDB for Java (8)

- **Using an EntityTransaction**
- **Operations that modify database content, such as store, update, and delete should only be performed within an active transaction.**
- **Given an EntityManager, em, you begin a transaction in the following way:**

```
em.getTransaction().begin();
```

- **There is a one to one relationship between an EntityManager instance and its associated EntityTransaction instances that the getTransaction method returns.**

OODBMS Case study: ObjectDB for Java (9)

- **CRUD Operations using ObjectDB**
- **Given an EntityManager, em, that represents a JPA connection to the object database, we can use it to store, retrieve, update and delete database objects as follows:**

OODBMS Case study: ObjectDB for Java

Operations	Examples

1. Storing New Entity Objects	`em.getTransaction().begin();`
	`for (int i = 0; i < 1000; i++) {`
	`  Point p = new Point(i, i);`
	`  em.persist(p);`
	`}`
	`em.getTransaction().commit();`

2. Reading Entity objects	`Query q1 = em.createQuery("SELECT COUNT(p) FROM Point p");` `System.out.println("Total Points: " + q1.getSingleResult());`

OODBMS Case study: ObjectDB for Java

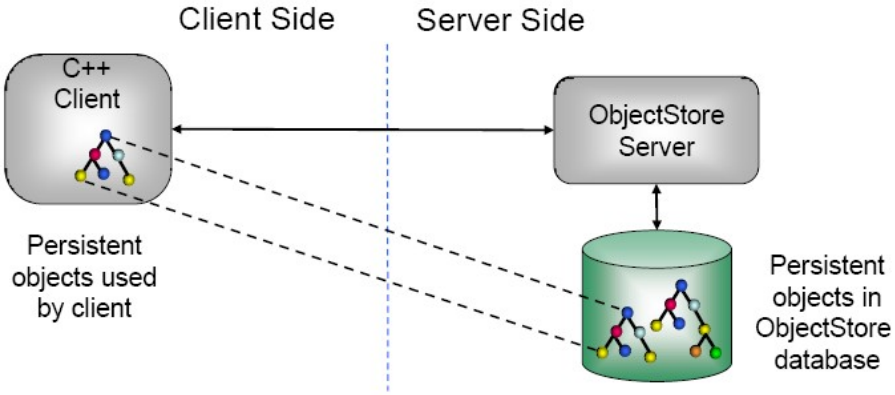
Operation	Examples

3. Delete Entities	`em.getTransaction().begin();`
	`em.remove(p); // delete entity`
	`em.getTransaction().commit();`

4. Update Entities	`em.getTransaction().begin();`
	`p.setX(p.getX() + 100); // update entity`
	`em.getTransaction().commit();`

OODBMS Case study: ObjectStore and C++ (1)

What is ObjectStore?



- ObjectStore is a 'pure' object database
- Stores C++ objects in the same format as they are used in memory
- Accessed and updated using the same C++ syntax as heap-allocated objects

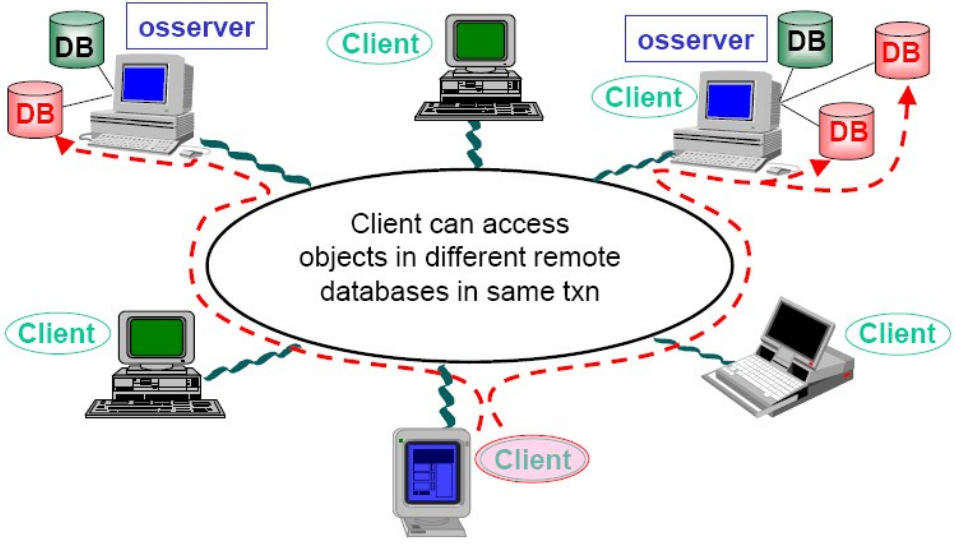
Heap-allocated objects

Commercial OODBMS

© 2009 Progress Software Corporation

OODBMS Case study: ObjectStore and C++ (2)

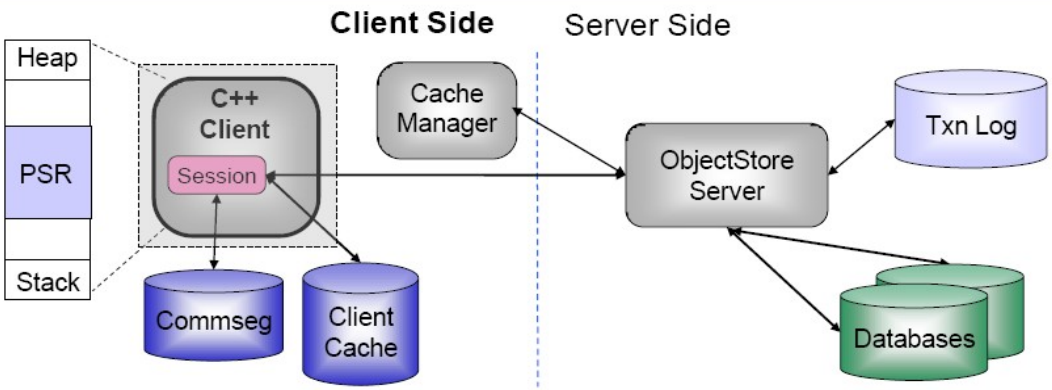
ObjectStore is Distributed



© 2009 Progress Software Corporation

OODBMS Case study: ObjectStore and C++ (3)

ObjectStore C++ Client



- The C++ client is the program you write
- It is linked with the ObjectStore libraries
- It opens databases and runs transactions against them
- It allocates objects, calls methods on them, and deletes them
- Pages are fetched automatically from the DB as needed
- ObjectStore automatically maintains a cache of recently accessed pages

Client is a C++ application program

© 2009 Progress Software Corporation

Recap: Limitations ('weaknesses') of RDBMs (1)

1. Poor representation of "real world" entities
   - Normalization leads to relations that do not correspond to entities in the "real world"
   - joins needed to obtain information from relations (tables)

2. Semantic Overloading
   - Single construct for representing data and data relationships: the relation
   - No mechanism to distinguish between relations & relationships or between different types of relationships

How does OODBMSs e.g. ObjectDB overcome the above limitations?

Solutions provided by OODBMs e.g ObjectDB

1. **Representation of "real world" entities**
   - **OOP constructs are used:** classes, inheritance, composition, etc.
   - **Joins are not used (there are no tables in the db)**
2. **No semantic overloading.**

- **Object identifier (OID) generated by DBMS**
  - **Logical OID:** **Independent of physical location on disk**
  - **Physical OID:** **Points to an actual location on disk**

- **Representation of relationships (business rules):**
  - **one-to-one , one-to-many, many-to-many**

Object access: e.g. Resident object table (ROT)

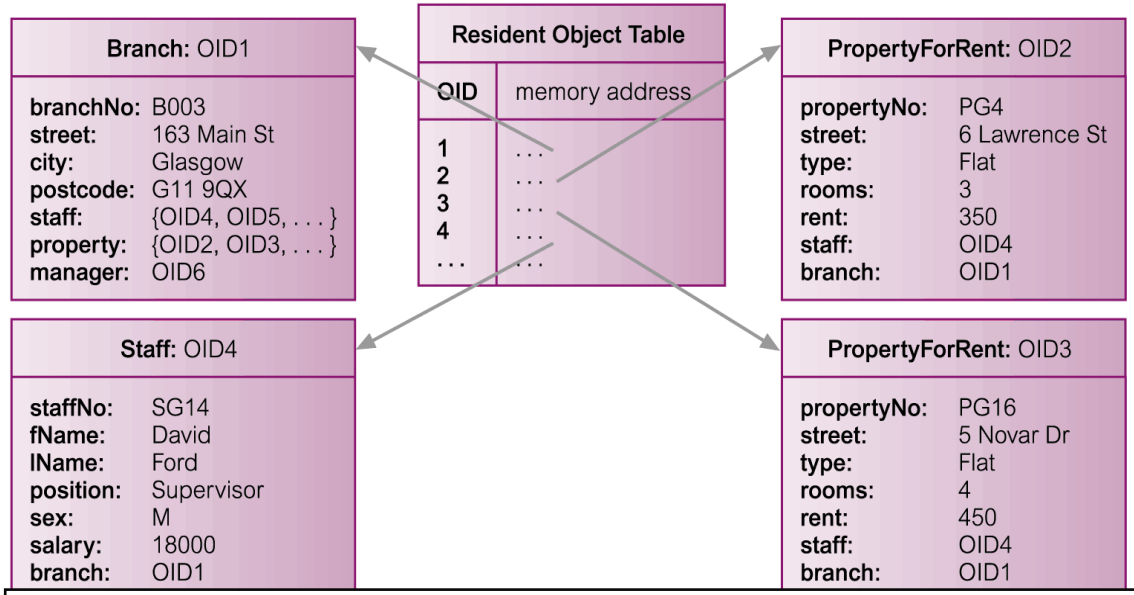


Figure 26.5 Resident Object Table referencing four objects in main memory.

e.g. of locating objects:

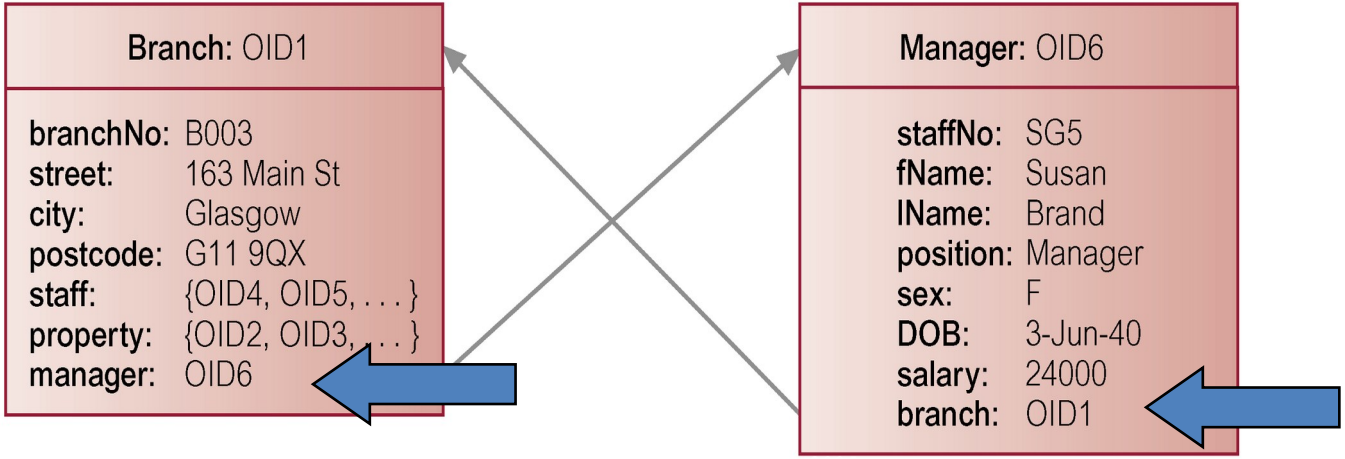
Lookup virtual memory pointer (in ROT) with each object access

OIDs obtained from DBMS

1:1 Relationship between objects A and B

-to-one relationship

- Add reference attributes to both objects
- Reference is an OID (Recall: **OIDs obtained from DBMS)**



1:* Relationship between objects A and B,C,D,...

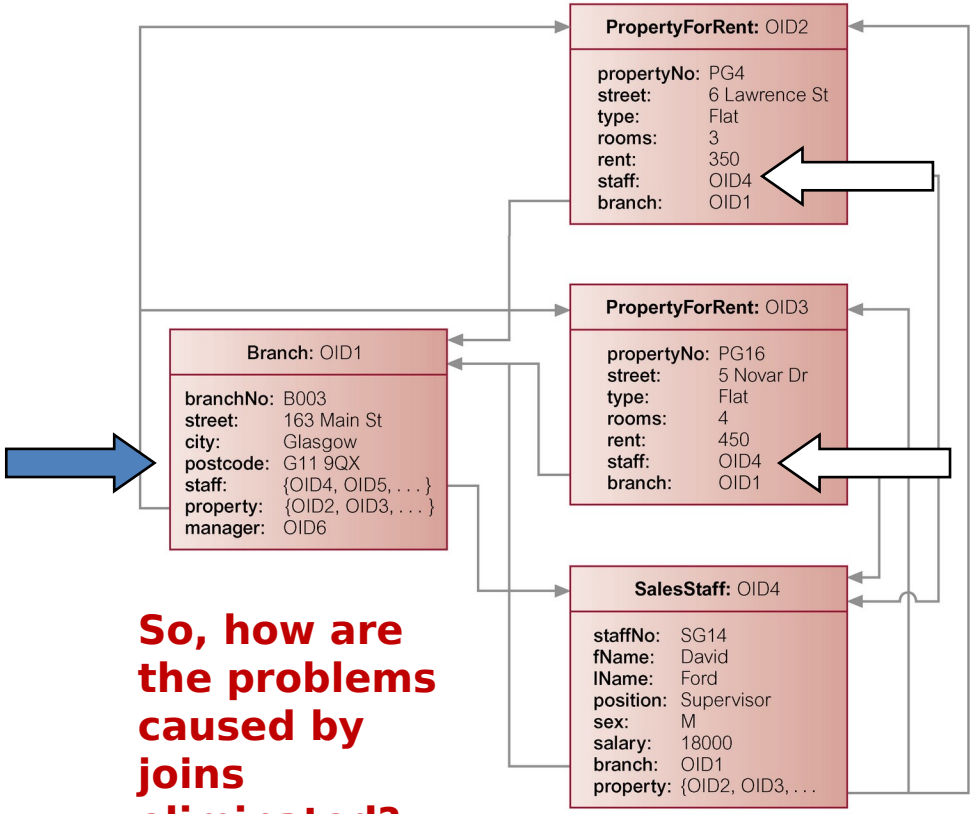
One-to-many Relationship

e.g. one Branch has many Properties for rent

Add reference attribute to one object (**branch** in PropertyForRent)

AND

attribute containing set of references to the other objects



So, how are the problems caused by joins eliminated?

Many-to-many relationships (*:*)

Client can view many properties.

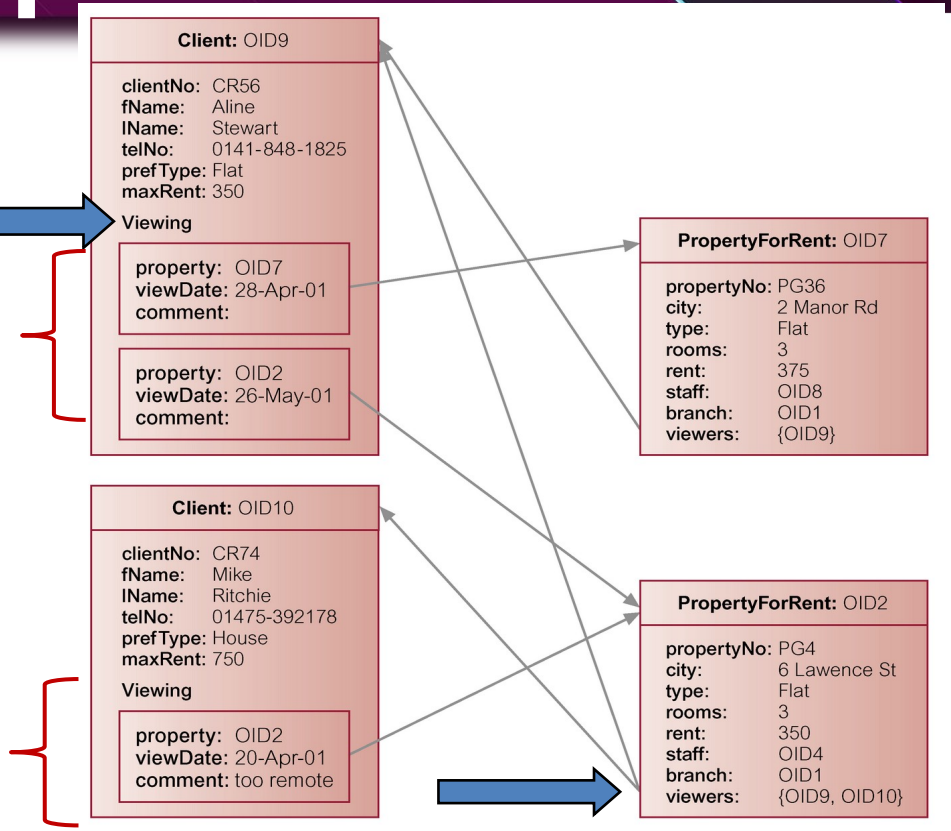
A property can-be viewed by many clients

Add

attribute containing set of references

to

each object



Recap: Limitations (weaknesses) of RDBMs (2)

3. **Limited Operations**

   - **RDBMSs only have a fixed set of operations which cannot be extended i.e. operations on relations (sets of tuples) & tuples**
   - **Not possible to specify new operations (BUT: can use stored procedures)**

- **How do OODBMSs e.g. ObjectDB overcome the above limitations?**

- **Solution: OOP methods are defined for classes**

  e.g. Pilot class

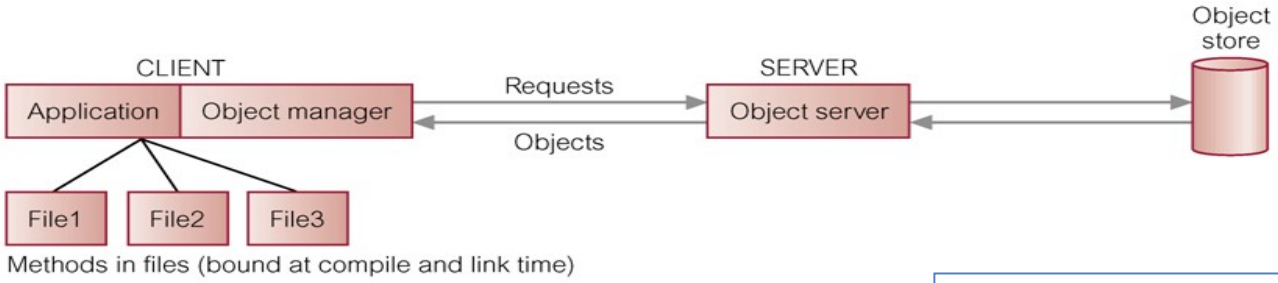
Pilot class for OODB

```
//package com.db4o.f1.chapter1;

public class Pilot
{
    private String name;
    private int points;

    public Pilot(String name,int points)
    {
        this.name=name;
        this.points=points;
    }
    public int getPoints()
    {
        return points;
    }
    public void addPoints(int points)
    {
        this.points+=points;
    }
    public String getName()
    {
        return name;
    }
    public String toString()
    {
        return name+"/"+points;
    }
```

Storing and executing methods

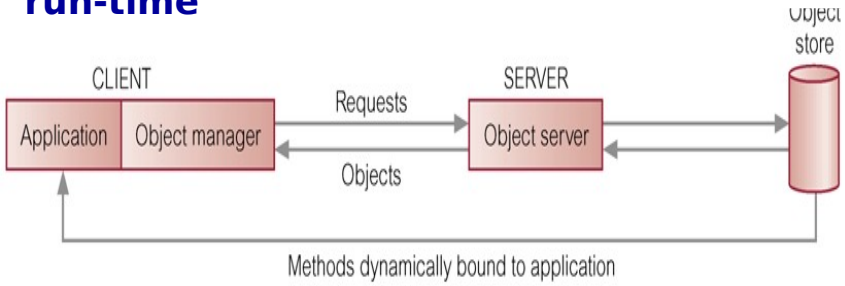
1. **Store methods in external files**

   Link libraries into the application at compile / link time



2. **Store methods in database**

   Dynamically bound to application at run-time



- » **Eliminates redundant code,**
- » **Simplifies modifications,**
- » **Methods are more secure,**
- » **Methods can be shared concurrently,**

  Improved integrity

COS 326
Database Systems

Lecture 4

Object-Oriented Databases (3)

Chapter 12 (7th edition)

Thursday 01 August 2024

In this lecture

- ObjectDB Demo
- JPA Query API
- JPA Query Structure (JPQL/Criteria)
- JPA Query Expressions (JPQL / Criteria)
- Overview of Object database concepts

ObjectDB for Java: Demo Time

- **ObjectDB Demo**
- **For the next 20-30 minutes, follow on the demo**

ObjectDB for Java: JPA Queries (1)

- **JPA Queries**
- **The JPA Query Language (JPQL) is defined as an object-oriented version of SQL.**
- **Users familiar with SQL find JPQL very easy to learn and use.**
- **We will discuss the following:**
- **JPA Query API**
- **JPA Query Structure (JPQL/Criteria)**
- **JPA Query Expressions (JPQL/ Criteria)**

ObjectDB for Java

- **JPA Query API**
- Queries are represented in JPA 2 by two interfaces:
- **the old Query interface:** only interface available for representing queries in JPA 1
- **new TypedQuery interface:** introduced in JPA 2. The TypedQuery interface extends the Query interface.

ObjectDB for Java: JPA Queries (2)

- **JPA Query API**

```
Query q1 = em.createQuery("SELECT c FROM Country c");

  TypedQuery<Country> q2 =
      em.createQuery("SELECT c FROM Country c",
Country.class);
```

ObjectDB for Java: JPA Queries (3)

- **JPA Query Structure (JPQL / Criteria)**
- The syntax of the Java Persistence Query Language (JPQL) is very similar to the syntax of SQL.
- The main difference between SQL and JPQL is that SQL works with relational database tables, records and fields, whereas JPQL works with Java classes and objects.

ObjectDB for Java: JPA Queries (4)

- **JPA Query Structure**
- As with SQL, a JPQL SELECT query also consists of up to 6 clauses in the following format:
  - SELECT … FROM …
  - [WHERE …]
  - [GROUP BY … [HAVING …]]
  - [ORDER BY …]
  - DELETE FROM … [WHERE …]
  - UPDATE … SET … [WHERE …]

ObjectDB for Java: JPA Queries (5)

- **JPA Query Expressions**
- Query expressions are the foundations on which JPQL and criteria queries are built.
- Every query consists of clauses - SELECT, FROM, WHERE, GROUP BY, HAVING and ORDER BY, and each clause consists of JPQL query expressions.

ObjectDB for Java: JPA Atomic Query Expressions (1)

- **JPA Atomic Query Expressions**
- The atomic query expressions are:
  - JPQL / Criteria Variables e,g Range
  - JPQL / Criteria Parameters e.g named, ordinal
  - JPQL / Criteria Literals e.g NULL, Boolean etc
- Every query expression consists of at least one atomic component. More complex query expressions are built by combining atomic expressions with operators and functions.

Atomic Query Expressions (2)

- **JPA Operators and Functions**
- JPQL / Criteria queries support the following operators (in order of decreasing precedence):
- **(1) Navigation operator** (.)
- **(2) Arithmetic operators**: * (multiplication), / (division), + (addition) and - (subtraction).
- **(3) Comparison operators**:

=, <>, <, <=,>, >=, IS [NOT] NULL, [NOT] BETWEEN,

including Collection operators: [NOT] IN, IS [NOT] EMPTY, [NOT] MEMBER [OF] and the [NOT] LIKE operator.

- **(4) Logical operators:** AND, OR, NOT.

Overview of Object Database Concepts

- Introduction to object-oriented concepts and features
  - Origins in OO programming languages
  - Object has two components:
    - State (value) and behavior (operations)
  - Instance variables (attributes)
    - Hold values that define internal state of object
  - Operation is defined in two parts:
    - Signature (interface) and implementation (method)

Overview of Object Database Concepts (cont'd.)

- Inheritance
  - Permits specification of new types or classes that inherit much of their structure and/or operations from previously defined types or classes
- Operator overloading
  - Operation's ability to be applied to different types of objects
  - Operation name may refer to several distinct implementations

Object Identity, and Objects versus Literals

- Object has Unique identity
  - Implemented via a unique, system-generated object identifier (OID)
  - **Immutable**
- Most OO database systems allow for the representation of both objects and literals (simple or complex values)

Other Object-Oriented Concepts

- **Polymorphism** of operations
  - Also known as **operator overloading**
  - Allows same operator name or symbol to be bound to two or more different implementations
  - Type of objects determines which operator is applied
- **Multiple inheritance**
  - Subtype inherits functions (attributes and operations) of more than one supertype

COS 326
Database Systems

Lecture 5

Object-Relational Databases (1)

Chapter 12 (7th edition)

Tuesday 06 August 2024

In this lecture

- Introduction to ORDBs
- Why use ORDBs? advantages & disadvantages of ORDBMSs: Reading for the student
- Database manifestos: Reading for the student
- History of Postgres
- SQL:2011

Arguments against Relational data model

- **Relational DBMS are satisfactory for normal / standard business applications**
  - **storing & querying business transactions data**

BUT

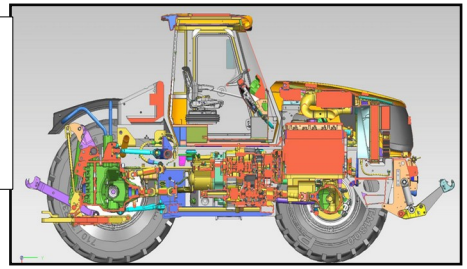
- ***Cannot support applications that need to store & query complex data***
  - **e.g. CAD, GIS**

Order: G7NNJHJ8P5H **Order Date:** 08-Oct-2015 3:50 PM ET

Item	Price	Quantity	Amount
Author - Conference Registration	$840.00	1	$840.00
Wire Transfer Fee	$20.00	1	$20.00
		Order Total	$860.00

Business invoic

CAD:
Computer Aided Design



GIS:
Geographical Information Systems

Arguments for Relational data model

- **Relational model is part of any real-world DB environment**
- **Businesses have made major investments in Relational DBs**
- **Complex applications can be *handled by extensions* to the relational model (support for: ORDBMS)**
- **Smooth transition through upgrades to newer DBMS versions which are ORDBMS**
- **Database Market Share**
  - **RDBMS: very large (billions of dollars)**
  - **ORDBMS: very small but growing**

1. ORDBMS

- **Extends the relational model with OO features**
  - **classes & objects,**
  - **aggregation & inheritance**
  - **polymorphism & dynamic binding etc**

- **Bridges the gap between**
  - **OO programming languages and RDBMS data**
  - **OO data modeling and RDBMS**

2. Reading for the student

- **Advantages & disadvantages of the ORDBMS approach**

- **Database manifestos**
  - **Third-generation database manifesto**
  - **Third manifesto**

3. Postgres – early (Historical) ORDBMS

- **Postgres ('Post Ingres')**
  - **research DBMS designed to be potential successor to INGRES.**
  - **query language for Postgres was called postquel**

- **Postgres extended the Relational model to include:**
  - **Abstract Data Types (ADTs) (DEFINE TYPE)**
  - **Data of type 'procedure' (DEFINE PROCEDURE)**
  - **Operators and precedence + associativity of operators (DEFINE OPERATOR)**

- **Supported OO constructs such as:**
  - **inheritance**
  - **aggregation**
    - **attributes that reference tuples (objects) in other relations**

4. SQL:2011 - New OO Features

- **Row types**
- **User-defined types (UDT)**
  - **distinct types, structured types**
  - **Supertype/subtype relationship**
- **User-defined routines (UDR)**
  - **procedures, functions, methods, operators**
- **Collection types**
  - **arrays, sets, lists, and multisets**
- **Note: The SQL:2016 has additional OOP extensions**

4. SQL:2011 Row type

- **Row type**
  - **sequence of *(field_name, data_type)* pairs**
- **Column of a table can contain row values**
- **A complete row can be:**
  - **stored in a variable**
  - **passed as an argument to a function**
  - **returned as return value from a function call**

Branch

branchNo	Address (is a row type)		
	street	city	postcode
B005	22 Deer Rd	London	SW14Eh

4. SQL:2011 Example - Row type

```
CREATE TABLE Branch (
        branchNo CHAR(4),
    address ROW (   street VARCHAR(25),
                city  VARCHAR(15),
                postcode VARCHAR(8) ) );

INSERT INTO Branch
    VALUES ( 'B005',
                ROW ('22 Roper street', 'Pretoria', '0002'));
```

brachNo	Address (is a row type)		
	street	city	postcode
B005	22 Roper street	Pretoria	0002

4. SQL:2011 User-defined types (UDTs == ADTs of postquel) (1)

- A user-defined data type (UDT) is a data type that derived from an existing data type.
- a UDT may be a distinct (simple) type or structured type

- Distinct (simple) types (= domain), e.g.

```
CREATE TYPE OwnerNoType AS VARCHAR(5) FINAL;
CREATE TYPE StaffNoType AS VARCHAR(5) FINAL;
CREATE TYPE NameType AS VARCHAR(15) FINAL;
```

- makes a distinction between same underlying data
- enforce type checking
- what does FINAL mean ?

4. SQL:2011 User-defined types (UDTs == ADTs of postquel) (2)

- structured UDT definition consists of
  - Attribute definitions
  - Routine declarations (methods)

- e.g. of structured types:

```
CREATE TYPE PersonType AS (
        dateOfBirth DATE,
    gender CHAR,
        fName NameType NOT NULL,
        lName NameType NOT NULL)
```

attribute definitions

4. SQL:2011 User-defined types (UDTs == ADTs of postquel) (3)

- **structured UDT definition consists of**
  - **Attribute definitions**
  - **Routine declarations (methods)**

- **e.g. of structured types:**

```
CREATE TYPE  PersonType AS (
    dateOfBirth DATE  ……. )

        INSTANTIABLE
        NOT FINAL
        REF IS SYSTEM GENERATED
        INSTANCE METHOD age() RETURNS
INTEGER;
        INSTANCE METHOD ……
```

Stored attribute

Instances can be created

Subtypes can be defined

methods

© Dr. Mohamed Hamdi

Creating Tables Based on the UDTs

- **INSTANTIABLE**
  - Specify that UDT is instantiable
  - The user can then create one or more tables based on the UDT
  - If keyword INSTANTIABLE is left out, can use UDT only as attribute data type – not as a basis for a table of objects
- **NOT FINAL:**

  The keyword NOT FINAL indicates that subtypes can be created for that type

4. SQL:2011 User-defined types (UDTs == ADTs of postquel) (4)

- structured UDT definition consists of
  - Attribute definitions
  - Routine declarations (methods) etc.

```
CREATE TYPE  PersonType AS (
   ................)
     INSTANCE METHOD age() RETURNS INTEGER;...
```

methods

```
CREATE INSTANCE METHOD age ( )
  RETURNS INTEGER FOR PersonType
   BEGIN
     RETURN  /*val calculated from SELF.dateOfBirth*/
  END;
```

method implementation

4. SQL:2011 UDTs
Supertypes & subtypes: Inheritance Example

- The subtype is created a new type that is similar but not identical to an already defined type.
- The subtype then inherits all the functions of the predefined type, which is referred to as the supertype.

```
CREATE TYPE StaffType UNDER PersonType AS (
    staffNo      StaffNoType,
    position     VARCHAR(10) DEFAULT 'Assistant',
    salary       DECIMAL(7, 2),
    branchNo     CHAR(4)     )

    INSTANTIABLE
    NOT FINAL
    INSTANCE METHOD isManager()
                                    RETURNS BOOLEAN;
```

4. SQL:2011 Ref types & object identity

- e.g. of structured types:

```
CREATE TYPE  PersonType AS (
    dateOfBirth DATE  ……. )
INSTANTIABLE
NOT FINAL
REF IS SYSTEM GENERATED
INSTANCE METHOD age() RETURNS INTEGER; …
```

System generates references

same meaning as:
OODBMS OID

- Reference type
  - uniquely identifies a row within a table
  - Similar to OID of OODBs

4. SQL:2011 Object Identifiers Using Reference Types

- **Reference type**
  - Create unique object identifiers (OIDs)
  - Can specify system-generated object identifiers
  - Alternatively can use primary key as OID as in traditional relational model
  - Examples:
    - `REF IS SYSTEM GENERATED`
    - `REF IS <OID_ATTRIBUTE> <VALUE_GENERATION_METHOD> ;`

4. SQL:2011 Creation of typed (UDT) tables

```
CREATE TABLE Person OF PersonType
        REF IS personID SYSTEM GENERATED;
```

- Rows are objects
- One column for every attribute in the UDT
- Cannot add any columns to this table

personID	dateOfBirth	fName	lName
1	1/1/1965	John	Smith

5. SQL:2011 Reference types & object identity

- e.g. CREATE TABLE PropertyForRent (

  propertyNo PropertyNumber NOT NULL,

  street StreetType, ...

  personID REF (PersonType) SCOPE PERSON)

  REFERENCES ARE CHECKED ON DELETE CASCADE,

  PRIMARY KEY (propertyNo));

- Replace complex join definitions in queries with much simpler path expressions e.g.
- INSTEAD OF:

  SELECT P.street, S.fname, S.lname

  FROM propertyForRent P, Person S WHERE P.personID = S.personID

This is a JOIN

- USE THE DEREFERENCING OPERATOR INSTEAD:

  SELECT P.street, P.personID->fname, P.personID->lname,

  FROM PropertyForRent P;

COS 326
Database Systems

Lecture 6

Object-Relational Databases (2)

Chapter 12 (7th edition)

Thursday 08 August 2024

In this lecture

- **PostgreSQL**
  - **Relationship between Postgres, SQL:2011 & PostgreSQL**
  - **PostgreSQL basics**
  - **ODMG Object Model and the Object Definition Language ODL**
  - **Object Database Conceptual Design**
  - **The Object Query Language OQL**

1. Postgres, SQL:2011 & PostgreSQL

- **Originally postgres, renamed PostgreSQL in 1996**
  - **query language postquel was replaced**

- **PostgreSQL uses a client/server model**

  (1) A server process (called postgres)
  - **manages the database files**
  - **accepts connections to the database from client applications,**
  - **performs database operations on behalf of the clients.**

  (2) Users' client (front end) applications
  - **make requests for database operations.**

- **Uses SQL + extensions for OOP as defined in SQL:2011 + other postgres specific features**

PostgreSQL: Demo Time

- **PostgreSQL Demo**
- **For the next 10-15 minutes, follow on the demo**

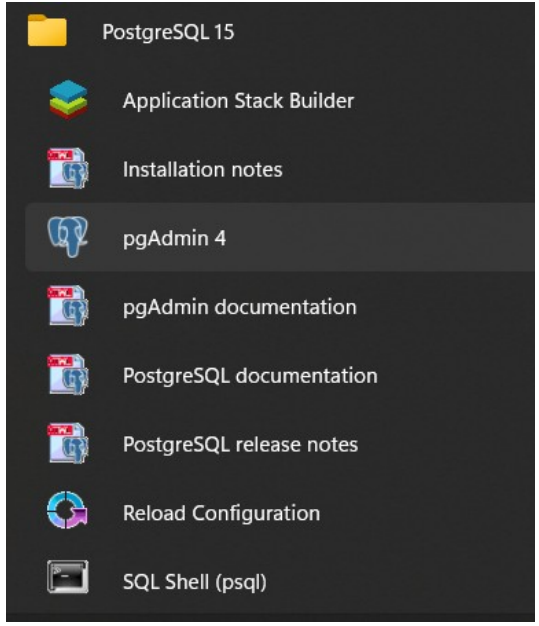
2. Using PostgreSQL in Windows (1)

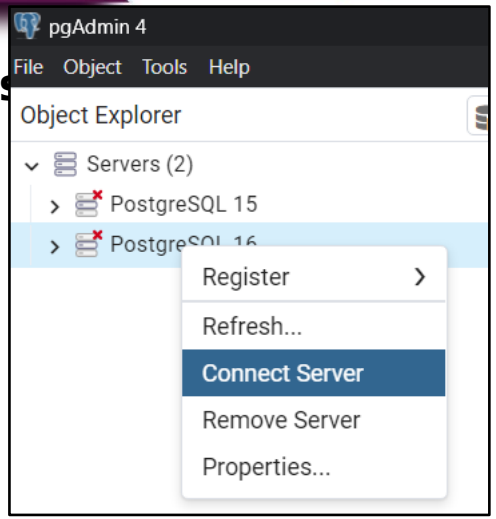
Installation: Run postgresql-16.3-2-windows-x64.exe

pgAdmin 4

database admin tool for PostgreSQL

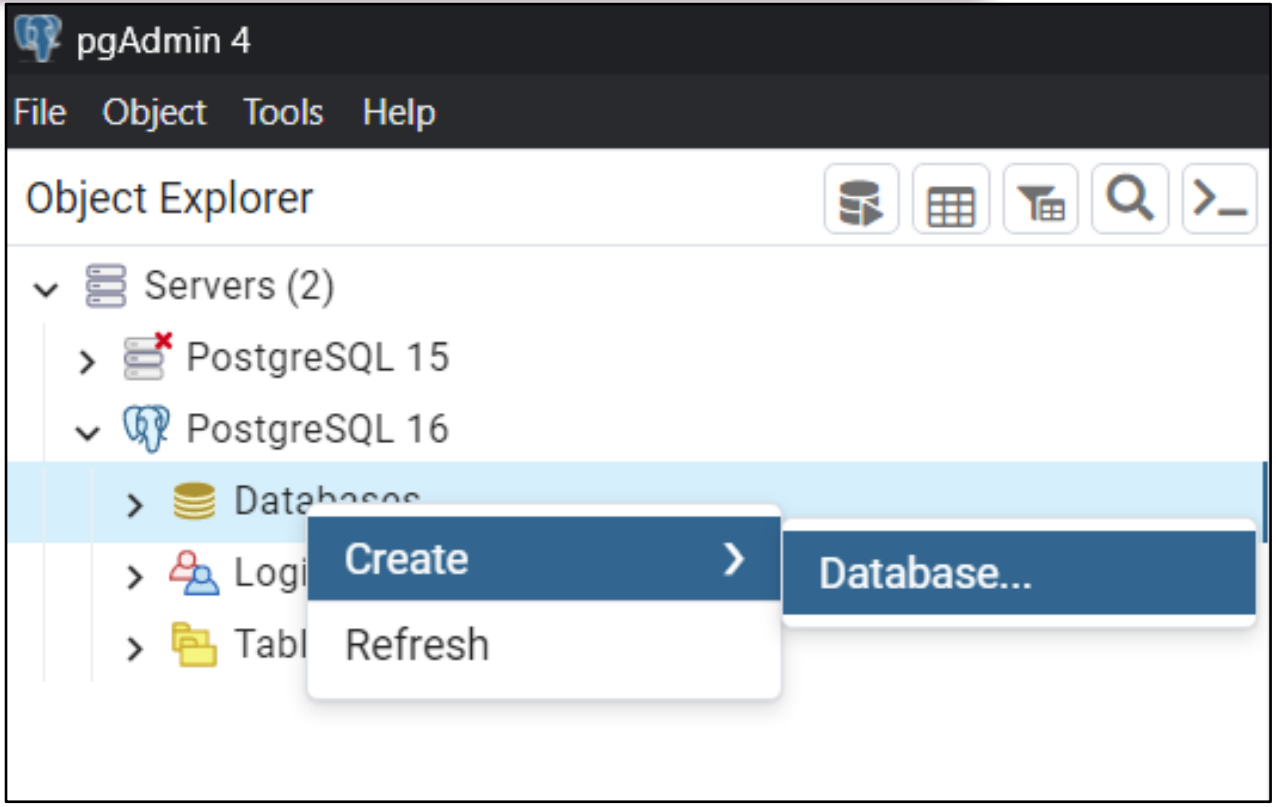
MS Windows 11 Menu





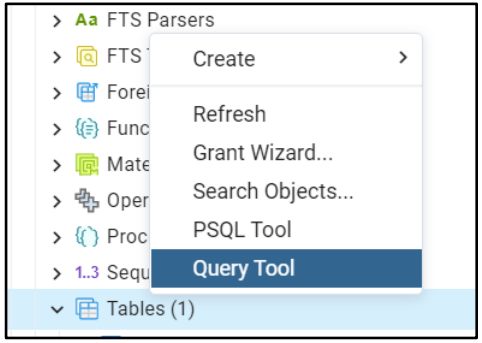
2. Using PostgreSQL in Windows (2a)

- pgAdmin 4 (database admin tool for PostgreSQL)



2. Using PostgreSQL in Windows (2b)

- pgAdmin 4 (database admin tool for PostgreSQL)
- Using the query tool



3. PostgreSQL – Data types

1. **Numeric Types**
2. **Monetary Types**
3. **Character Types**
4. **Binary Data Types**
5. **Date/Time Types**
6. **Boolean Type**
7. **Enumerated Types**
8. **Network Address Types**
9. **Bit String Types**
10. **Geometric Types**
    - Points
    - Line Segments,
    - Boxes, Paths
    - Polygons
    - Circles
11. Text Search Types
12. XML Type
13. **Composite Types**
14. **Arrays**
15. **Object Identifier Types**
16. Pseudo-Types

4. PostgreSQL – user defined types (1)

- **Enumerated types: data types that comprise of a static, ordered set of values.**
- **Similar to HLL enum types e.g.:**

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

- **e.g. of Enum Usage**

```
CREATE TABLE person (
                    name text, current_mood   mood  );

INSERT INTO person
                    VALUES ('Cyril', 'happy');

SELECT * FROM person WHERE current_mood = 'happy';
```

5. PostgreSQL – user defined types (2a)

- **Composite types** similar to HLL record / struct types e.g.:

```
CREATE TYPE inventoryItem AS (
        name          text,
        supplierID    integer,
        price         numeric);
```

- e.g. of usage of composite type

```
CREATE TABLE StockOnHand (
                item  inventoryItem,  quantity  integer);

INSERT INTO StockOnHand VALUES
                ( ROW  ('gismo', 42, 1.99), 1000);
```

6. PostgreSQL – user defined types (2b)

- Composite types contnd.

```
CREATE TYPE inventoryItem AS (
        name   text,   supplierID   integer,
price   numeric);
```

- e.g. of usage of composite type

```
CREATE TABLE   StockOnHand (
    item  inventoryItem,  quantity  integer);

SELECT   (StockOnHand.item).name,
      (StockOnHand.item).price,
      StockOnHand.quantity
FROM      StockOnHand
     WHERE   (StockOnHand.item).price   >   0.99;
```

ODMG Object Model and Object Definition Language ODL

- ODMG object model
  - **ODMG**- Object Data Management Group
  - Data model for **object definition language (ODL)** and **object query language (OQL)**
- Objects and Literals
  - Basic building blocks of the object model
- Object has five aspects:
  - **Identifier**, **name**, **lifetime, structure**, and **creation**
- **Literal**
  - Value that does not have an object identifier

The ODMG Object Model and the ODL (cont'd.)

- **Behavior** refers to operations
- **State** refers to properties (attributes)
- **Interface**
  - Specifies only behavior of an object type
  - Typically **noninstantiable**
- **Class**
  - Specifies both state (attributes) and behavior (operations) of an object type
  - **Instantiable**

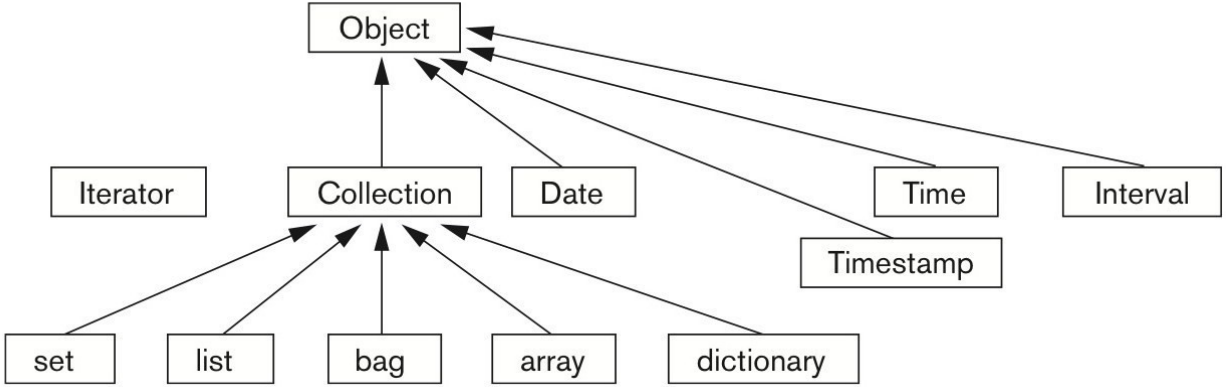
Inheritance in the Object Model of ODMG

- **Behavior inheritance**
  - Also known as IS-A or interface inheritance
  - Specified by the colon (:) notation
- **EXTENDS inheritance**
  - Specified by keyword **`extends`**
  - Inherit both state and behavior strictly among classes
  - Multiple inheritance via extends not permitted

Built-in Interfaces and Classes in the Object Model

- **Collection objects**
  - Inherit the basic Collection interface
- `i = o.create_iterator()`
  - Creates an iterator object for the collection
  - To loop over each object in a collection
- Collection objects further specialized into:
  - `set`, `list`, `bag`, `array`, and `dictionary`

Figure 12.6 Inheritance hierarchy for the built-in interfaces of the object model



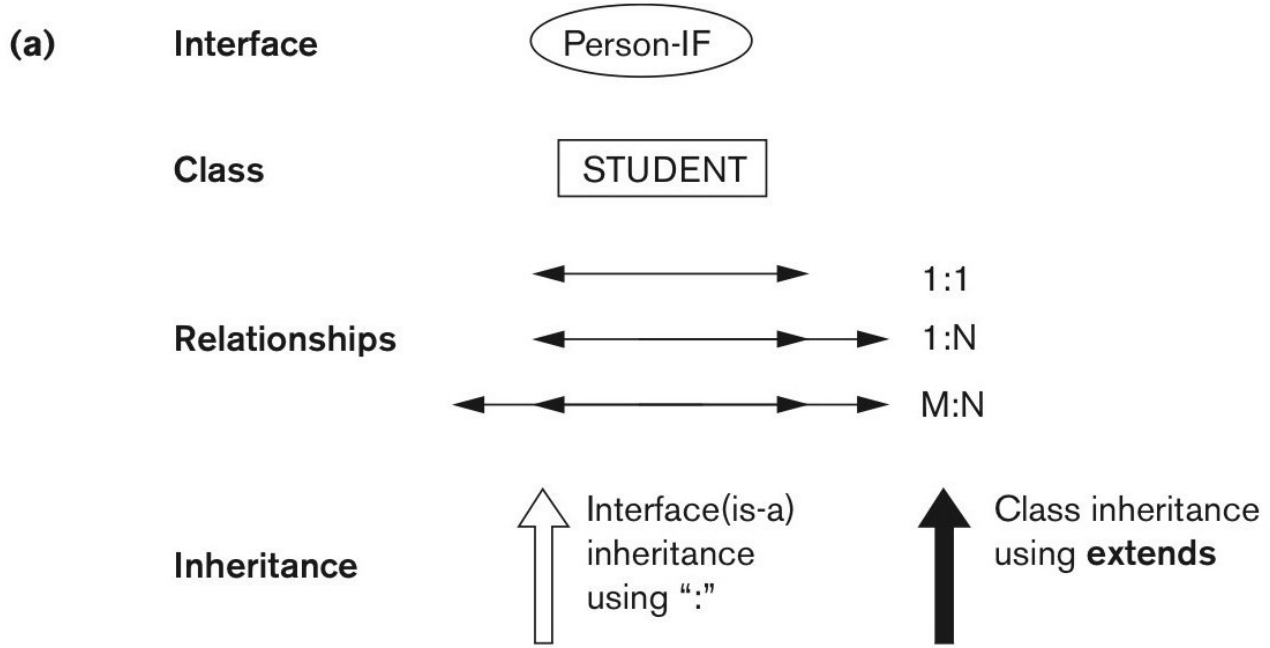
Extents, Keys, and Factory Objects

- **Extent**
  - A persistent named collection object that contains all persistent objects of class
- **Key**
  - One or more properties whose values are unique for each object in extent of a class
- **Factory object**
  - Used to generate or create individual objects via its operations

Object Definition Language (ODL)

- Support semantic constructs of ODMG object model
- Independent of any particular programming language
- Example on next slides of a UNIVERSITY database
- Graphical diagrammatic notation is a variation of EER diagrams

Figure 12.9a An example of a database schema. Graphical notation for representing ODL schemas



continued on next slide

Figure 12.9b An example of a database schema. A graphical object database schema for part of the UNIVERSITY database (GRADE and DEGREE classes are not shown).

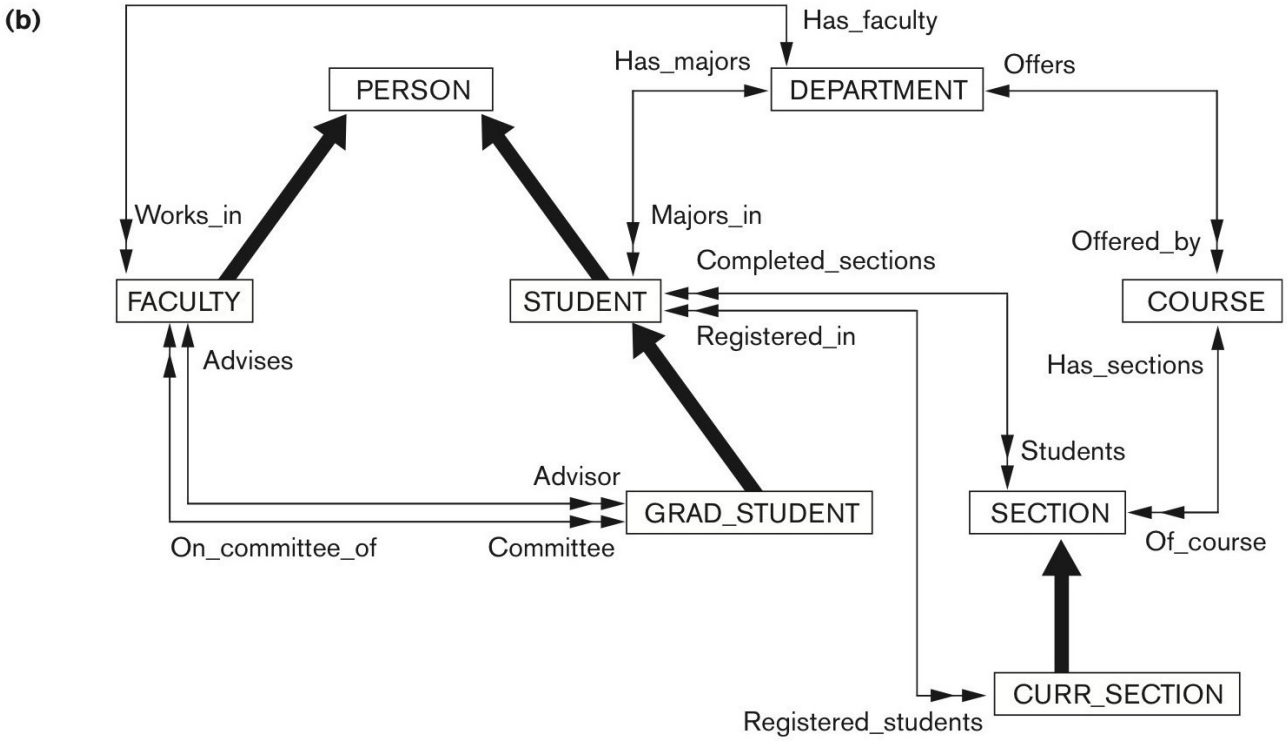


Figure 12.10 Possible ODL schema for the UNIVERSITY database in Figure 12.9(b).

```
class PERSON
(   extent      PERSONS
    key         Ssn )
{   attribute   struct Pname {   string   Fname,
                                 string   Mname,
                                 string   Lname }      Name;
    attribute   string                                 Ssn;
    attribute   date                                   Birth_date;
    attribute   enum Gender{M, F}                      Sex;
    attribute   struct Address {  short   No,
                                 string   Street,
                                  short   Apt_no,
                                 string   City,
                                 string   State,
                                  short   Zip }        Address;
    short       Age();   };
class FACULTY extends PERSON
(   extent      FACULTY )
{   attribute   string          Rank;
    attribute   float           Salary;
    attribute   string          Office;
    attribute   string          Phone;
    relationship  DEPARTMENT    Works_in inverse DEPARTMENT::Has faculty;
    relationship  set<GRAD_STUDENT> Advises inverse GRAD_STUDENT::Advisor;
    relationship  set<GRAD_STUDENT> On_committee_of inverse GRAD_STUDENT::Committee;
    void        give_raise(in float raise);
    void        promote(in string new rank); };
class GRADE
(   extent GRADES )
{
    attribute   enum GradeValues{A,B,C,D,F,I, P} Grade;
    relationship  SECTION Section inverse SECTION::Students;
    relationship STUDENT Student inverse STUDENT::Completed_sections; };
class STUDENT extends PERSON
(   extent      STUDENTS )
{   attribute   string          Class;
    attribute   Department      Minors_in;
    relationship  Department Majors_in inverse DEPARTMENT::Has_majors;
    relationship  set<GRADE> Completed_sections inverse GRADE::Student;
    relationship  set<CURR_SECTION> Registered_in INVERSE CURR_SECTION::Registered_students;
    void        change_major(in string dname) raises(dname_not_valid);
    float       gpa();
    void        register(in short secno) raises(section_not_valid);
    void        assign_grade(in short secno; IN GradeValue grade)
                    raises(section_not_valid,grade_not_valid); };
```

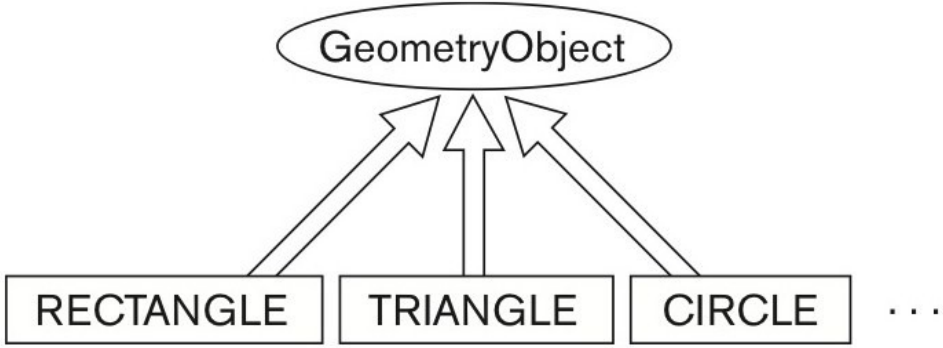
continued on next slide

Figure 12.10 (continued) Possible ODL schema for the UNIVERSITY database in Figure 12.9(b).

```
class DEGREE
{   attribute      string              College;
    attribute      string              Degree;
    attribute      string              Year; };
class GRAD_STUDENT extends STUDENT
(   extent         GRAD_STUDENTS )
{   attribute      set<Degree>         Degrees;
    relationship   Faculty advisor inverse FACULTY::Advises;
    relationship   set<FACULTY>        Committee inverse FACULTY::On_committee_of;
    void           assign_advisor(in string Lname; in string Fname)
                       raises(faculty_not_valid);
    void           assign_committee_member(in string Lname; in string Fname)
                       raises(faculty_not_valid); };
class DEPARTMENT
(   extent         DEPARTMENTS
    key            Dname )
{   attribute      string              Dname;
    attribute      string              Dphone;
    attribute      string              Doffice;
    attribute      string              College;
    attribute      FACULTY             Chair;
    relationship   set<FACULTY> Has_faculty inverse FACULTY::Works_in;
    relationship   set<STUDENT> Has_majors inverse STUDENT::Majors_in;
    relationship   set<COURSE> Offers inverse COURSE::Offered_by; };
class COURSE
(   extent         COURSES
    key            Cno )
{   attribute      string              Cname;
    attribute      string              Cno;
    attribute      string              Description;
    relationship   set<SECTION> Has_sections inverse SECTION::Of_course;
    relationship   <DEPARTMENT> Offered_by inverse DEPARTMENT::Offers; };
class SECTION
(   extent         SECTIONS )
{   attribute      short               Sec_no;
    attribute      string              Year;
    attribute      enum Quarter{Fall, Winter, Spring, Summer}
                       Qtr;
    relationship   set<Grade> Students inverse Grade::Section;
    relationship   COURSE Of_course inverse COURSE::Has_sections; };
class CURR_SECTION extends SECTION
(   extent         CURRENT_SECTIONS )
{   relationship   set<STUDENT> Registered_students
                       inverse STUDENT::Registered_in
    void           register_student(in string Ssn)
                       raises(student_not_valid, section_full); };
```

Figure 12.11a An illustration of interface inheritance via ":". Graphical schema representatio

(a)



continued on next slide

Figure 12.11b An illustration of interface inheritance via “:”. Corresponding interface and class definitions in ODL

```
(b) interface GeometryObject
    {   attribute   enum        Shape{RECTANGLE, TRIANGLE, CIRCLE, … }
                                    Shape;
        attribute   struct      Point {short x, short y} Reference_point;
        float       perimeter();
        float       area();
        void        translate(in short x_translation; in short y_translation);
        void        rotate(in float angle_of_rotation); };
    class RECTANGLE : GeometryObject
    (   extent      RECTANGLES )
    {   attribute   struct      Point {short x, short y} Reference_point;
        attribute   short       Length;
        attribute   short       Height;
        attribute   float       Orientation_angle; };
    class TRIANGLE : GeometryObject
    (   extent      TRIANGLES )
    {   attribute   struct      Point {short x, short y} Reference_point;
        attribute   short       Side_1;
        attribute   short       Side_2;
        attribute   float       Side1_side2_angle;
        attribute   float       Side1_orientation_angle; };
    class CIRCLE : GeometryObject
    (   extent      CIRCLES )
    {   attribute   struct      Point {short x, short y} Reference_point;
        attribute   short       Radius; };
    …
```

Object Database Conceptual Design

- Differences between conceptual design of ODB and RDB, handling of:
  - Relationships
  - Inheritance
- Philosophical difference between relational model and object model of data
  - In terms of behavioral specification

The Object Query Language OQL

- Query language proposed for ODMG object model
- Simple OQL queries, database entry points, and iterator variables
  - Syntax: select ... from ... where ... structure
  - Entry point: named persistent object
  - Iterator variable: define whenever a collection is referenced in an OQL query

Query Results and Path Expressions

- **Result of a query**
  - Any type that can be expressed in ODMG object model
- OQL orthogonal with respect to specifying path expressions
  - Attributes, relationships, and operation names (methods) can be used interchangeably within the path expressions

Other Features of OQL

- **Named query**
  - Specify identifier of named query
- OQL query will return collection as its result
  - If user requires that a query only return a single element use **`element`** operator
- Aggregate operators
- Membership and quantification over a

Other Features of OQL (cont'd.)

- Special operations for ordered collections
- **Group by** clause in OQL
  - Similar to the corresponding clause in SQL
  - Provides explicit reference to the collection of objects within each group or **partition**
- **Having clause**

COS 326
Database Systems

Lecture 7

Object-Relational Databases (3)

Chapter 12 (7th edition)

Tuesday 13 August 2024

In this lecture

PostgreSQL

1. **Data types: arrays**
2. **User defined types & arrays**
3. **Object-oriented (OO) features**
4. **User defined functions**
5. **How do ORDBMS overcome the limitations (weaknesses) of the RDBMS? (HOMEWORK)**

Recap

- **PostgreSQL**
  - **Relationship between Postgres, SQL:2011 & PostgreSQL**
  - **PostgreSQL basics**
  - **ODMG Object Model and the Object Definition Language ODL**
  - **Object Database Conceptual Design**
  - **The Object Query Language OQL**

PostgreSQL – user defined types (1)

- Read more on data types supported in PostgreSQL in Chapter 8 of the user manual
- Array types similar to HLL array types e.g.:
  - column of a table can be defined as variable-length array.
- Array elements may be of:
  - any built-in type (e.g. integer, text, etc)
  - user-defined base type,
  - enum type ,
  - composite type
- e.g. of declaration

```
CREATE TABLE sal_Emp (
name          text,
payByQuarter  integer[ ],    /* 1D array  */
schedule      text[ ][ ]   );  /* 2D array  */
```

PostgreSQL – user defined types (2)

- **Array types** similar to HLL array types e.g.:

```
 CREATE TABLE sal_Emp (
name        text,
payByQuarter  integer[ ],
schedule          text[ ][ ]);
INSERT INTO sal_Emp    VALUES (
            'Thando',
      '{20000, 25000, 25000, 25000}'  ,
            '{ { "breakfast", "consulting" } , { "lunch" ,
"meeting"  } }’);
OR:

INSERT INTO sal_Emp    VALUES
    (‘Thando',
    ARRAY[20000, 25000, 25000, 25000 ] ,
    ARRAY[ [ ‘breakfast', 'consulting‘  ] , [  'lunch','meeting‘  ] ]   );
```

PostgreSQL – user defined types (3)

- **Array types** **si**milar to HLL array types e.g.:

  CREATE TABLE sal_Emp (

  name text, payByQuarter integer[ ], schedule text[ ][ ]);

- The **current dimensions of any array value** can be retrieved with the **array_dims function**

- e.g. to determine the **dimensions of the 'schedule' column:**

  SELECT array_dims(schedule)

  FROM sal_Emp WHERE name = 'Thando';

	array_dims text
1	[1:2][1:2]

PostgreSQL – user defined types (4)

- **Array types** similar to HLL array types e.g.:

```
CREATE TABLE sal_Emp (
    name  text, payByQuarter integer[ ], schedule
text[ ][ ]);
```

- **other select examples:**

(1)

```
SELECT name FROM sal_Emp
WHERE payByQuarter[1]   <>   payByQuarter[2];
```

	name text
1	Thando

(2)

```
SELECT name, schedule[1:1][1:2] FROM sa
```

	name text	schedule text[]
1	Thando	{{breakfast,consulting}}

PostgreSQL OOP features: Inheritance (1)

```
CREATE TABLE CITIES (
        name          text,
        population    real,          /* in millions */
        area          int  );        /* in sq km    */

CREATE TABLE CAPITALS (
        state     char(2)  )
 INHERITS (CITIES);
```

=> a row of CAPITALS *inherits* all columns (name, population, and area) from its *parent*, CITIES.

State CAPITALS have an extra column, state.

In PostgreSQL, a table can inherit from one or more other tables.

PostgreSQL OOP features : Inheritance (2)

```
CREATE TABLE CITIES (  name text,  population real,  area
      int  );
CREATE TABLE CAPITALS (  state  char(2) )    INHERITS
     (CITIES);

INSERT INTO CITIES VALUES ('Johannesburg', 2.5, 2500);
INSERT INTO CAPITALS V
```

	name text	population real	area integer
1	Johannesburg	2.5	2500
2	Pretoria	1.5	900

e.g. **list the names of all**
including state capital

```
SELECT *  FROM CITIES
```

e.g. **list all the capitals:**

```
SELECT *  FROM CAPITA
```

	name text	population real	area integer	state character
1	Pretoria	1.5	900	GP

PostgreSQL –Procedural Language Functions

- **PostgreSQL allows user-defined functions to be written in other languages besides SQL and C.**
- **These other languages are generically called procedural languages (PLs).**

PostgreSQL –User defined functions (5)

Four categories user defined functions:

(1) Query language functions (SQL functions: written in SQL)

(2) Procedural language functions

(functions written in, e.g. PL/pgSQL, PL/Tcl, PL/Perl and PL/Python) – see Chapters 44-46 Postgres User Manual

(3) Internal functions

(4) C-language functions

Two categories of SQL functions:

(1) Row functions: operate on values of a table row

(2) Table functions: operate on whole table

- are functions that produce a set of rows, made up of:
  - either base data types (scalar types) or
  - composite data types (table rows)

The $$ is a delimiter you use to indicate where the function definition starts and ends.

PostgreSQL – SQL functions (1)

SQL functions use the following syntax:

```
CREATE FUNCTION functionName( pmt1Type, pmt2Type,   ....  ) RETURNS
    returntype AS
    $$
            SQL statement
     $$ LANGUAGE SQL;
```

e.g.

```
-- DROP FUNCTION populationDensity(real, integer);   if you need to
CREATE FUNCTION populationDensity(real, int) RETURNS int  AS
$$
            SELECT CAST (  (($1 * 1000000) / $2 ) AS int )
            AS populationDensity;
$$ LANGUAGE SQL;
```

The $$ is a delimiter you use to indicate where the function definition starts and ends.

PostgreSQL – SQL functions (2)

```
CREATE FUNCTION populationDensity(real, int ) RETURNS
    int  AS
$$
                SELECT CAST (  (($1 * 1000000) / $2 ) AS int )
                AS populationDensity;
$$ LANGUAGE SQL;

SELECT   name, population, area,
                populationDensity( population, area )
FROM CITIES ;
```

	name text	population real	area integer	populationdensity integer
1	Johannesburg	2.5	2500	1000
2	Pretoria	1.5	900	1667

PostgreSQL- PL/pgsql functions (1)

The syntax of a PL/pgSQL function is as follows:

```
CREATE FUNCTION functionname(   pmttype1, pmttype2, …   )
                                RETURNS returntype AS
$$
    DECLARE             --comment 1: declare vars only if necessary
                         var1   vartype1,      var2    vartype2,….  ;
    BEGIN
                        -- comment 2: code to compute return value
                        RETURN returnvalue ;

                        --comment 3:  return value must be of
    returntype
    END;
 $$ LANGUAGE plpgsql;
```

PostgreSQL –PL/pgsql functions (2)

e.g.

```
CREATE FUNCTION popDensity(real, int ) RETURNS int  AS
$$
    DECLARE   density int;
    BEGIN
            density = CAST (  (($1 * 1000000) / $2 ) AS int );
            RETURN density;
    END;
$$ LANGUAGE plpgsql;

SELECT   name, population, area,  popDensity( population,
    area )
FROM CITIES :
```

	name text	population real	area integer	popdensity integer
1	Johannesburg	2.5	2500	1000
2	Pretoria	1.5	900	1667

Homework

- How does an ORDBMS e.g. PostgreSQL overcome the limitations (weaknesses) of the RDBMS?
- **Practice creating the various functions in PostgreSQL**

COS 326
Database Systems

Lecture 8

Object-Relational Databases (4)

Chapters 12 & 26 (7th edition)

Thursday 15 August 2024

In this lecture

PostgreSQL advanced features

1. More on functions

2. Triggers

Recap

PostgreSQL

1. **Data types: arrays**
2. **User defined types & arrays**
3. **Object-oriented (OO) features**
4. **User defined functions**
5. **How do ORDBMS overcome the limitations (weaknesses) of the RDBMS? (HOMEWORK)**

Summary of SQL:2011

Activity	Statement example
Special operations for changes to database data: **Trigger:** **SQL (compound) statement executed *automatically* by DBMS as *side effect* of a modification to named table**	**e.g. 1** **CREATE TRIGGER** InsertPropTable **BEFORE INSERT ON** PropertyForRent ..... **e.g. 2** **CREATE TRIGGER** DeletePropTable **AFTER DELETE ON** PropertyForRent .....

SQL:2011 SQL/PSM

- **SQL/PSM**
  - **SQL Persistent Stored Modules**
  - Extensions to SQL that provide a (server-side) procedural language
  - used to write functions, stored procedures, Triggers that are stored on the DB server.
- Databases (Relational / Object-Relational) from different suppliers provide their own version of SQL/PSM e.g.

  - **ORACLE: PL/SQL language**
  - **PostgreSQL: PL/pgSQL**

PostgreSQL –User defined functions (1a)

Four categories of user defined functions:

(1) query language functions (**SQL functions:** written in SQL)

(2) **procedural language functions**

(functions written in, e.g. **PL/pgSQL** or PL/Tcl)

(3) Internal functions

(4) C-language functions

Two categories of SQL functions:

(1) **Row functions**: operate on values of a table row

(2) **Table functions:** operate on whole table

- are functions that **produce a set of rows, made up of:**
  - **either base data types (scalar types) or**
  - **composite data types (table rows).**

PostgreSQL – user defined functions (1b)

```
----SQL function
CREATE FUNCTION functionName(pmttype1, pmttype2, .. )
 RETURNS
                                returntype AS
    $$
          SQL statement
     $$ LANGUAGE SQL;

---PLpgSQL function
CREATE FUNCTION functionName(pmttype1, pmttype2, .. )  RETURNS
                                returntype AS
$$
    DECLARE          --comment 1:      declare vars  if necessary
       var1 vartype1,  var2 vartype2,….  ;
 BEGIN                -- comment 2:     code to compute return value
       RETURN returnvalue ;     -- must be of returntype
 END;
```

PostgreSQL – user defined functions (1c-i)

```
CREATE FUNCTION functionname(... ) RETURNS returntype AS
$$
   DECLARE var1 vartype1,  var2 vartype2,….  ;
 BEGIN

 ---1.  e.g. var1 = 10;     or      var1 := 10;


 --2.      IF THEN      END IF
  --       IF THEN      ELSE      END IF
  --       IF THEN      ELSE IF     THEN      END IF

      --3.    CASE  varx     WHEN   END CASE
 RETURN returnvalue ;
 END;
$$ LANGUAGE plpgsql;
```

PostgreSQL – user defined functions (1c-ii)

EXAMPLE: Given the table created as:

```
CREATE TABLE cities  ( name text,    population real,    area int );
```

and function:

```
CREATE FUNCTION is_big_city( integer ) RETURNS text AS
$$
 DECLARE   big text;
 BEGIN
 IF   $1   >=   1000   THEN
      big   :=   'YES';
 ELSE
      big   :=   'NO';
 END IF;
       RETURN   big;
 END;
$$ LANGUAGE plpgsql
```

and query:

```
SELECT name, area, is_big_city(area) FROM CITIES;
```

	name text	area integer	is_big_city text
1	Johannesburg	2500	YES
2	Pretoria	900	NO

PostgreSQL – user defined functions (1d)

```
CREATE FUNCTION functionname(... ) RETURNS returntype AS
$$
   DECLARE  var1 vartype1,  var2 vartype2,….  ;
 BEGIN
 --programming constructs for looping
 --4.   LOOP      EXIT WHEN        END LOOP
 --     LOOP      EXIT WHEN        CONTINUE WHEN       END LOOP

 --5.   WHILE expr  LOOP    END LOOP

        RETURN returnvalue ;
 END;
 $$ LANGUAGE plpgsql;
```

PostgreSQL – user defined functions (1e-i)

```
CREATE FUNCTION functionname(... ) RETURNS returntype AS
$$
   DECLARE var1 vartype1,  var2 vartype2,....  ;
 BEGIN
 --programming constructs for looping contnd

 --6.   FOR    var   IN    listexpr   LOOP    END LOOP
 --      FOR    var   IN    REVERSE listexpr LOOP    END LOOP

          --7.   FOREACH   target  IN ARRAY arrayexpr LOOP   END LOOP

 --     FOREACH   target  SLICE num  IN ARRAY expression
                                        LOOP  END LOOP

 RETURN returnvalue ;
 END;
$$ LANGUAGE plpgsql;
```

PostgreSQL – user defined functions (1e-ii)

```
INSERT INTO sal_Emp    VALUES (
            'Adriaan',
              '{22000, 27000, 27000, 27000}'  ,
            '{ { "breakfast", "consulting" } ,
   {"lunch" , "meeting"  } }');

INSERT INTO sal_Emp    VALUES (
            'Katlego',
              '{30000, 35000, 35000, 35000}'  ,
            '{ { "breakfast", "consulting" } ,
   { "lunch" , "meeting"  } }');
```

PostgreSQL – user defined functions (1e-iii)

Given:

```
CREATE TABLE salEmp ( name   text,    payByQuarter  int[ ],  ..);
```

and function:

```
CREATE FUNCTION total_salary( int[ ] )   RETURNS   int  AS
$$
 DECLARE   val int;   total int;
 BEGIN
 total = 0;
 FOREACH val IN ARRAY $1
 LOOP
    total = total + val;
 END LOOP;
        RETURN total;
 END;
$$ LANGUAGE plpgsql
```

and query:

```
SELECT name, paybyquarter, total_salary(paybyquarter) I
 sal_Emp;
```

	name text	paybyquarter integer[]	total_salary integer
1	Thando	{20000,25000,25000,25000}	95000
2	Adriaan	{22000,27000,27000,27000}	103000
3	Katlego	{30000,35000,35000,35000}	135000

PostgreSQL – user defined functions (1f)

```
CREATE FUNCTION functionname(... ) RETURNS
returntype AS
$$
    DECLARE var1 vartype1,  var2 vartype2,....  ;
BEGIN
--other constructs
--8.   SQL statement  (YES!!!)

RETURN returnvalue ;
END;
$$ LANGUAGE plpgsql;
```

Triggers (3a)

- **A trigger defines an action** that the DBMS should take **when some event occurs.**

- Most commonly the event arises due to an **operation (INSERT, UPDATE, or DELETE)** on a database table.

- A trigger may be defined as a **BEFORE trigger or an AFTER trigger.**

- A **BEFORE trigger** executes before the specified operation while
- an **AFTER trigger** executes after the specified operation.

- **A trigger may be used to:**
  - **enforce some referential constraints**
  - **enforce complex constraints (business rules)**
  - **audit changes to data**

Triggers in PL/SQL (3b)

- **(refer to Chapter 39 of PostgreSQL 16,4 documentation).**
- Syntax of PL/SQL create trigger statement (similar to SQL/PSM standard):

```
CREATE TRIGGER   TriggerName
            BEFORE | AFTER |  INSTEAD OF
           INSERT | DELETE | UPDATE
                 [OF optional TriggerColumnList] ON TableName
 [REFERENCING {OLD | NEW}
                        AS { OldName | newName }  ]
[FOR EACH { ROW | STATEMENT } ]
[WHEN condition]
< trigger action>
```

Triggers in PL/SQL (3c)

- **Example of PL/SQL create trigger statement:**

```
CREATE TRIGGER   StaffAfterInsert
        AFTER INSERT ON  Staff
            REFERENCING NEW AS new

            FOR EACH ROW
            BEGIN
                    INSERT INTO StaffAudit
                    VALUES
  ( :new.staffNo, :new.fName, :new.lName,user, now());
            END;
```

PostgreSQL – Trigger syntax (4a)

```
CREATE FUNCTION triggerfunction( ) RETURNS TRIGGER
    AS
$$
   DECLARE -- declare vars  if necessary
   BEGIN
                        -- code to compute any required values goes
    here.
            RETURN NEW;
    END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER triggername            -- e.g. of a before
    trigger
    BEFORE tableoperation ON tablename
        --- table operation is INSERT or DELETE or UPDATE
     FOR EACH ROW
          -- enables access to the NEW and OLD table rows
    EXECUTE PROCEDURE triggerfunction( );
```

PostgreSQL – Trigger example (4b)

Given the table created as:

```
CREATE TABLE cities  ( name text, population real, area
    int  );
CREATE FUNCTION check_valid_area() RETURNS TRIGGER AS
$$
    BEGIN
            IF (NEW.area <= 0) THEN
                    RAISE EXCEPTION 'Invalid area value!';
            END IF;
            RETURN NEW;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER valid_area
   BEFORE INSERT  OR UPDATE ON Cities
    FOR EACH ROW
            EXECUTE PROCEDURE check_valid_area();
```

PostgreSQL – Trigger example cntnd(4c)

RECALL:

CREATE TABLE cities (name text, population real, area int);

CREATE TRIGGER valid_area BEFORE INSERT OR UPDATE ON Cities……

SELECT * FROM Cities;

	name text	population real	area integer
1	Johannesburg	2.5	2500
2	Pretoria	1.5	900

INSERT INTO Cities VALUES ('Durban', 4.7, -3000);

Data Output | Messages | Notifications

```
ERROR:  Invalid area value!
CONTEXT:  PL/pgSQL function check_valid_area() line 4 at RAISE

SQL state: P0001Query returned successfully in 227 msec.
```

INSERT INTO Cities VALUES ('Durban', 4.7, 3000);

	name text	population real	area integer
1	Johannesburg	2.5	2500
2	Durban	4.7	3000
3	Pretoria	1.5	900

SELECT * FROM Cities;

PostgreSQL – Trigger example cntnd(4d)

CREATE TABLE Cities (name text, population real, area int);

CREATE TRIGGER valid_area BEFORE INSERT ... ON Cities ....

CREATE TABLE Capitals(state character(2)) INH

INSERT ....

SELECT * FROM Cities;

	name text	population real	area integer
1	Johannesburg	2.5	2500
2	Durban	4.7	3000
3	Pretoria	1.5	900

SELECT * FROM Capitals;

	name text	population real	area integer	state character
1	Pretoria	1.5	900	GP

INSERT INTO Capitals VALUES ('Upington', 0.5, -

SELECT * FROM Cities;

	name text	population real	area integer
1	Johannesburg	2.5	2500
2	Durban	4.7	3000
3	Pretoria	1.5	900
4	Upington	0.5	-1000

*****SOLUTION??? ******

PostgreSQL – Trigger example cntnd(4e)

CREATE TABLE cities (name text, population real, area int);

CREATE TABLE Capitals(state character(2) UNIQUE NOT NULL) INHERITS (cities);

So,

CREATE TRIGGER valid_area
BEFORE INSERT OR UPDATE ON Capitals
FOR EACH ROW
EXECUTE PROCEDURE check_valid_area();

Data Output | Messages | Notifications

```
ERROR:  Invalid area value!
CONTEXT:  PL/pgSQL function check_valid_area() line 4 at RAISE

SQL state: P0001Query returned successfully in 180 msec.
```

DELETE FROM CAPITALS WHERE name = 'Upington';

INSERT INTO Capitals VALUES ('Upington', C
'NC');

	name text	population real	area integer	state character
1	Pretoria	1.5	900	GP

	name text	population real	area integer
1	Johannesburg	2.5	2500
2	Durban	4.7	3000
3	Pretoria	1.5	900

PostgreSQL – Trigger example (4f)

```
CREATE TABLE cities  ( name text, population real, area int  );
CREATE TABLE Capitals( state character(2)) INHERITS (cities);
      CREATE FUNCTION check_valid_area() RETURNS TRIGGER AS
      $$
          BEGIN    ....RETURN NEW;  END;
      $$ LANGUAGE plpgsql;

      CREATE TRIGGER valid_area
         BEFORE INSERT  OR UPDATE ON Cities
                        FOR EACH ROW
                                    EXECUTE PROCEDURE check_valid_area();
      CREATE TRIGGER valid_area
         BEFORE INSERT  OR UPDATE ON Capitals
                        FOR EACH ROW
                                    EXECUTE PROCEDURE check_valid_area();
```

NOTE: (1) TRIGGERS cannot be inherited

(2) TRIGGER FUNCTIONS can be RE-USED

(3) can use the same name for triggers on different tables

PostgreSQL – Trigger example (4g)

```
CREATE TABLE deletedCities    (
    name   text,              population real,      area integer,
    deleteDateTime   timestamp,   deleteUserID   text    );

CREATE  FUNCTION record_deleted_city() RETURNS TRIGGER AS
$$
BEGIN
    INSERT INTO DeletedCities
    SELECT OLD.*,   now(),    user;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER audit_deleted_cities
    AFTER DELETE ON Cities
    FOR EACH ROW
    EXECUTE PROCEDURE record_deleted_city();
```

PostgreSQL – Trigger example (4h)

```
CREATE TRIGGER audit_deleted_cities
    AFTER DELETE ON Cities
     FOR EACH ROW
                EXECUTE PROCEDURE record_deleted_city();
```

```
INSERT INTO Cities VALUES ('Cape Town', 2
SELECT * FROM Cities;

DELETE FROM  Cities
WHERE name = 'Cape Town';

SELECT * FROM DeletedCities;
```

	name text	population real	area integer
1	Johannesburg	2.5	2500
2	Durban	4.7	3000
3	Cape Town	2	3500
4	Pretoria	1.5	900

	name text	population real	area integer	deletedatetime timestamp without time zone	deleteuserid text
1	Cape Town	2	3500	2023-08-14 15:31:44.168757	postgres

PostgreSQL –Trigger syntax ()

A more detailed syntax of a trigger function (procedure) is as follows:

```
CREATE FUNCTION triggerfunction() RETURNS TRIGGER AS
$$
   DECLARE
      --comment 1: declare vars  if necessary. var1 vartype1, var2 vartype2,..
   BEGIN
      -- comment 2: code to compute any required values goes here. In the code,
     -- use NEW.attribute1,.. to access values of new input row        e.g.  for INSERT.
      -- use OLD.attribute1,...to access values of existing  row in table,  e.g   for
     UPDATE,
      -- DELETE

      IF (condition to test for unwanted situation) THEN
     RAISE EXCEPTION 'error message for exception';
     END IF;
     RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

COS 326
Database Systems

Lecture 9

Semi-Structured Data and XML (1)

Chapter 13 (7th edition)

Tuesday 20 August 2024

In this lecture

- Introduction to Structured, Semi-structured and Unstructured Data
- XML
- XML Hierarchical (Tree) Data Model
- XML Schema

Introduction

- Databases function as data sources for Web applications
  - HTML
    - Used in static Web pages
  - XML, JSON
    - Self-describing documents
    - Dynamic Web pages
- Chapter focus: XML data model and languages

Structured, Semistructured, and Unstructured Data

- Structured data
  - Stored in relational database
- Semistructured data
  - Not all data has identical structure
  - Schema information mixed in
    - Self-describing
  - Directed graph model
- Unstructured data
  - Limited data type indication
  - Example: Web pages in HTML

Semi-structured data

Data that may be irregular or incomplete and have a structure that may change rapidly or unpredictably.

e.g. XML documents

- **may have a structure, but may change**
- **aka schema-less or self-describing**
  - **no separate schema**
  - **schema is part of the data**
- **relational, object-oriented, object-relational DBMSs**
  - **Require pre-defined schema**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<STAFFLIST>
    <STAFF branchNo="B002">
        <STAFFNO>SL20</STAFFNO>
        <NAME>
            <FNAME>Thando</FNAME>
            <LNAME>Mandela</LNAME>
        </NAME>
        <POSITION>Manager</POSITION>
        <DOB>1-Oct-98</DOB>
        <SALARY>30000</SALARY>
    </STAFF>
    <STAFF branchNo="B003">
        <STAFFNO>SL21</STAFFNO>
        <NAME>
            <FNAME>Marlize</FNAME>
            <LNAME>Kruger</LNAME>
        </NAME>
        <POSITION>Accountant</POSITION>
        <SALARY>12000</SALARY>
        <DOB>31-Nov-02</DOB>
        <SALARY>27000</SALARY>
```

Directed Graph Model for Semistructured Data

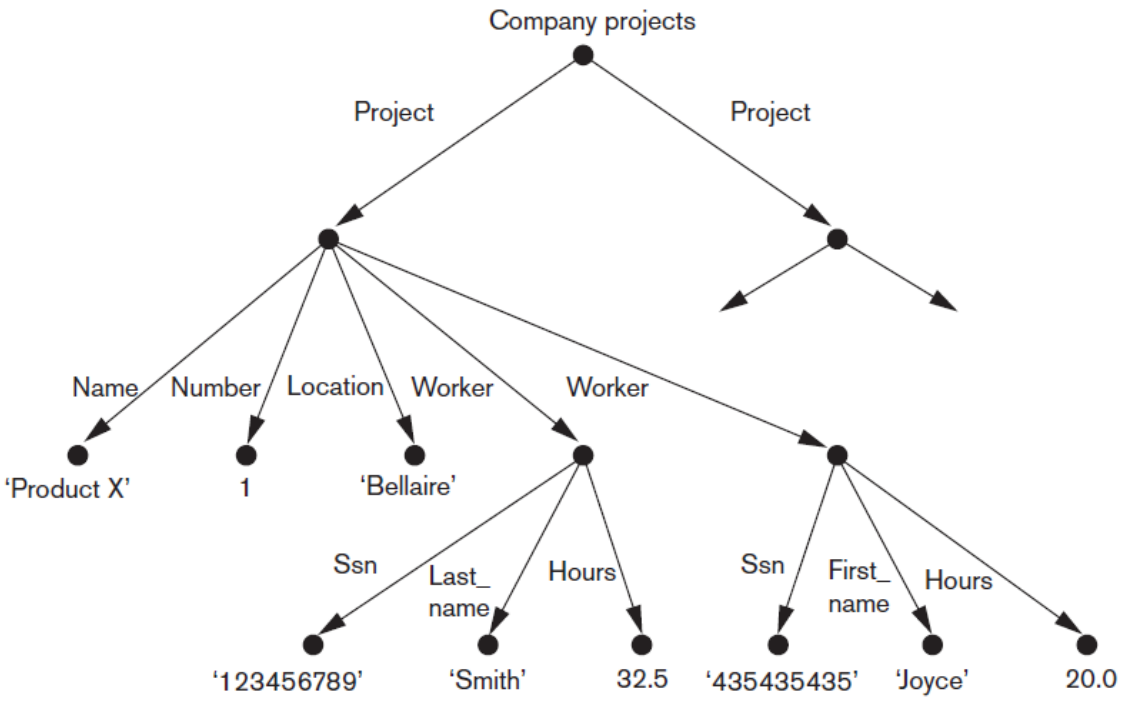


Figure 13.1 Representing semistructured data as a graph

XML (eXtensible Markup Language)

- **Meta-language (language for describing other languages)**
  - **enables designers to create their own customized tags**
  - **provide functionality not available with HTML**
- **1998: XML 1.0 by World Wide Web Consortium (W3C)**

  Has impacted many types of systems e.g.
  - **distributed systems**
  - **database management etc.**
- ***de facto* standard for data communication**
  - **Replacing EDI (Electronic Data Interchange)**

XML: what you already know

COS216 & INF272 Revision

1. XML file: e.g. *myfile.xml*
2. Format of an XML document
   - header, tags, attributes, namespaces, etc
3. Document Object Model (DOM)
   - tree structure for an xml document
4. XML document validation
   - Document Type Definitions (DTDs)
   - Schemas (Microsoft XML schema, W3C schema)
5. Conversion of XML data to formatted text documents
   - Extensible Stylesheet Language (XSL, XSLT)

Reading for the student

- **Introduction to XML**
  - **Advantages of XML**
  - **Overview of XML**
  - **Document Type Definitions (DTDs)**

- **XML-related technologies**
  - **DCOM and SAX**
  - **Namespaces**
  - **XLS and XLST**

XML Hierarchical (Tree) Data Model

- Basic object: XML document
  - Element
    - Simple
    - Complex
    - Schema document defines element names
  - Attribute
- Document types
  - Data-centric
  - Document-centric
  - Hybrid

XML Schema (1)

- XML schema language
  - Commonly known as XML Schema Definition (XSD)
  - Specifies document structure
  - Same syntax rules as XML documents
  - Elements, attributes, keys, references, and identifiers
  - Example: Figure 13.5 in the prescribed textbook

XML Schema (2)

- XML namespace
  - Defines set of commands that can be used
- Annotations, documentation, and language used
- Elements and types
  - Root element
  - First-level elements
  - Specifying element type and min and max occurrences

XML Schema Example

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
  <xs:element name = "contact">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "name" type = "xs:string" />
        <xs:element name = "company" type = "xs:string" />
        <xs:element name = "phone" type = "xs:int" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML Schema (3)

- Keys
  - Constraints that correspond to relational database
  - Primary key
  - Foreign key
- Complex elements
  - xsd:complexType
- Composite (compound) attributes

XML Schema Definition Types

- You can define XML schema elements in the following ways:
- Simple Types
- Complex Types
- Global Types

XML Schema Definition Types: Simple Type

- Simple type element is used only in the context of the text.
- Some of the predefined simple types are:
- xs:integer, xs:boolean, xs:string, xs:date.

- An example of a simple type element

- <xs:element name = "phone_number" type = "xs:int" />

XML Schema Definition Types: Complex Type

- A complex type is a container for other element definitions.
- This allows you to specify which child elements an element can contain and to provide some structure within your XML documents.
- An example of a complex type element:

```
<xs:element name = "Address">
  <xs:complexType>
    <xs:sequence>
      <xs:element name = "name" type = "xs:string" />
      <xs:element name = "company" type = "xs:string" />
      <xs:element name = "phone" type = "xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

XML Schema Definition Types: Global Type

- A global type allows you to define a single type in your document, which can be used by all other references.
- For example, suppose you want to generalise the person and company for different addresses of the company. In such case, you can define a general type as follows
- Example of a global type element:

```
<xs:element name = "AddressType">
  <xs:complexType>
    <xs:sequence>
      <xs:element name = "name" type = "xs:string" />
      <xs:element name = "company" type = "xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Storing and Extracting XML Documents from Databases

- Use file system or DBMS to store documents as text
- Use DBMS to store document contents as data elements
- Design specialized system to store XML data
- Create or publish custom XML documents from preexisting relational databases
  - This approach explored further in section 13.6 (prescribed textbook)

COS 326
Database Systems

Lecture 10

Semi-Structured Data and XML (2)

Chapter 13 (7th edition)

Thursday 22 August 2024

In this lecture

- XML Languages
- XPath: path expressions
- XQuery
- Introduction to BaseX
- Writing XPath expressions in BaseX

- Files available on ClickUP
  - XML file for this lecture: staff_list.xml

XML Languages

- Query language standards
  - XPath
  - XQuery
- Specifying XPath expressions in XML
  - Returns sequence of items satisfying certain pattern
    - Values, elements, or attributes
  - Qualifier conditions
  - Separators
    - Single slash / or double slash //

XML Path Language (XPath)

- Declarative query language
- Specifying a directory-like path
  - Possibly with conditions placed in the path
  - Retrieves collections of elements
- Treats XML document as logical tree of nodes

Query component	Description
context node	starting point
location path	path from one point in document to another, composed of steps
step	consists of basis and predicates
basis	axis and node test
axis	direction e.g. grandparent, parent, child, etc.
node test	identifies node type, e.g. element name or

XPath query example

basis

axis

node test

Details of 1st staff member?

child::STAFF[1] *or*
/child::STAFF[position()=1]

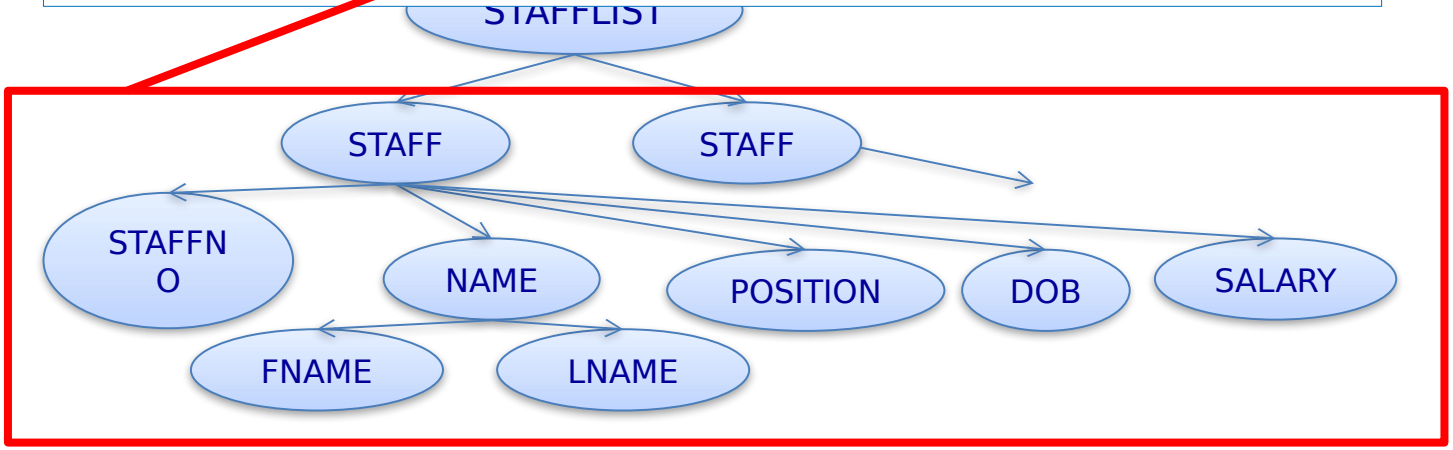
predicate

Component	Description
context node	starting point in a document
location path	path from one point in document to another, composed of steps /
step	consists of basis and predicates
basis	axis and node test
axis	direction e.g. grandparent, parent, child etc.
node test	identifies node type, e.g. element name or function text()
predicate	in square brackets after the basis

Location path	Selects…
.	…context node
..	…parent of context node
/	…root node, or separator between steps in path
//	…descendants of current node
/child::STAFF *or* /STAFF	…STAFF elements that are children of the root
child::STAFF *or* STAFF	…STAFF element children of context node
attribute::branchNo *or* @branchNo	…branchNo attribute of context node
attribute::* *or* @*	…all attributes of context node
child::STAFF[3]	…third STAFF element that is a child of the context node
/child::STAFF[@branchNo="B005"]	…STAFF elements that have attribute branchNo with value B005
/child::STAFF[@branchNo="B005"]	…first STAFF elements that have

XPath: path expressions

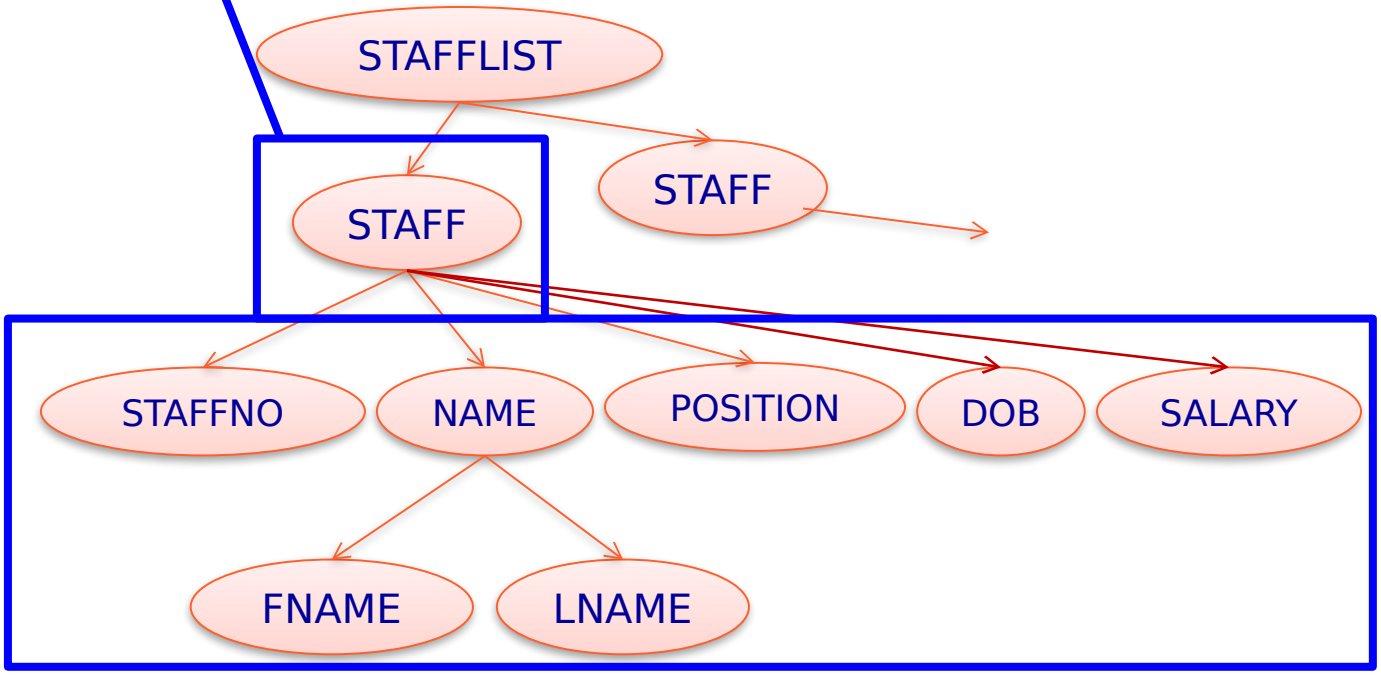
Details of all staff? **//STAFFLIST/child::STAFF** ***or*** **//STAFFLIST/STAFF**



XPath: path expressions

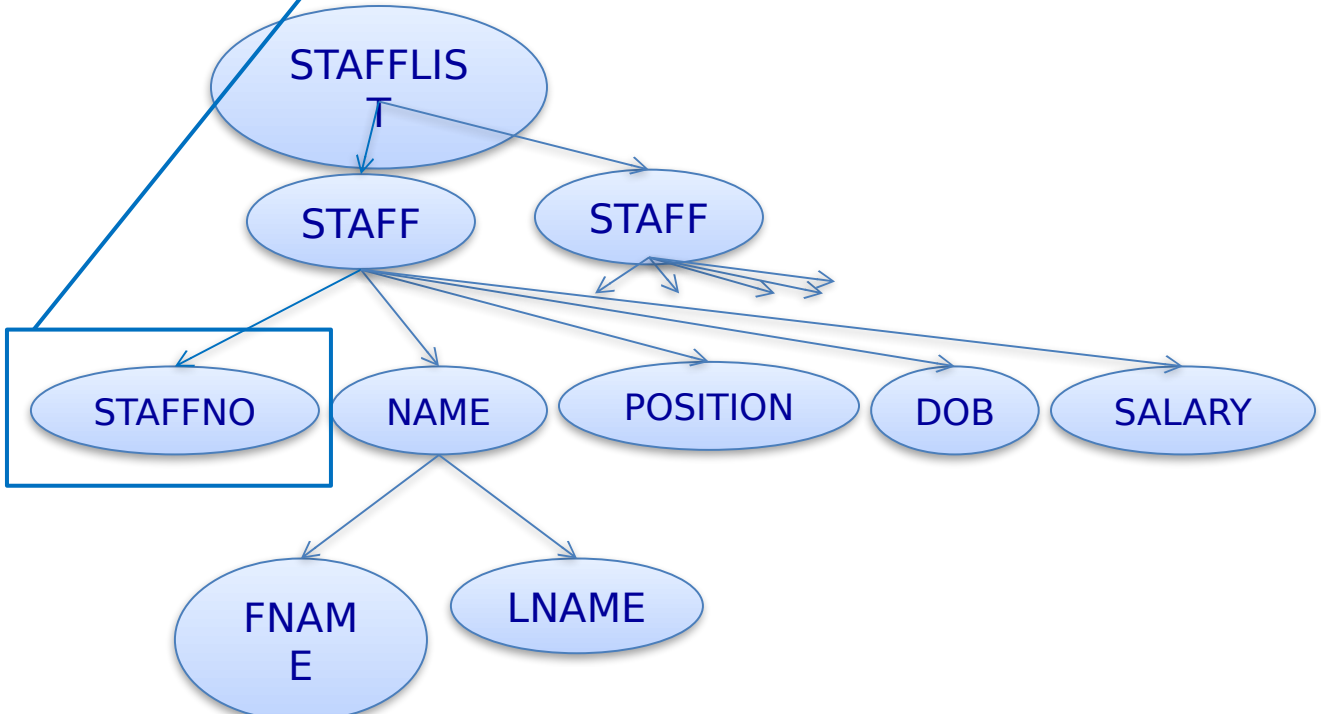
Details of 1st staff?

//STAFFLIST/child::STAFF[1] *or* *//STAFFLIST/***STAFF[1]** *or*
//STAFFLIST/child::STAFF[position()=1]



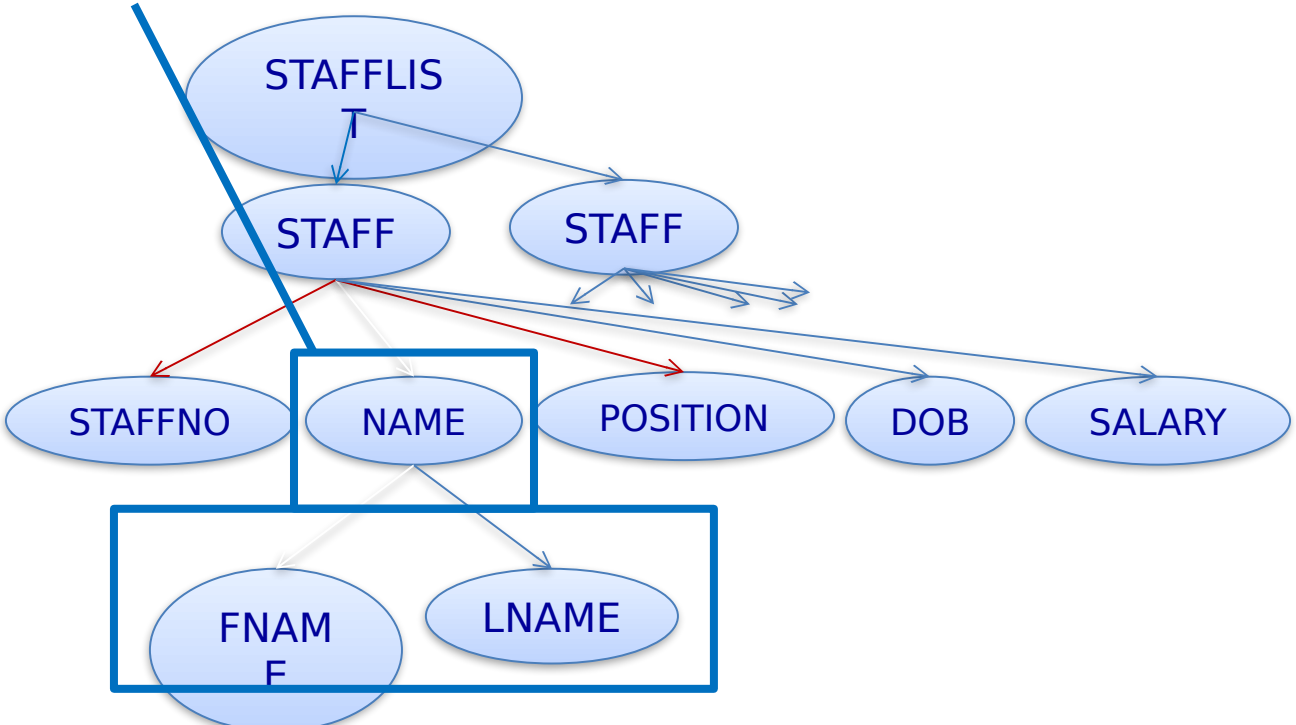
XPath: path expressions Staff number of 1st staff?

//STAFFLIST/STAFF[1]/STAFFNO



XPath: path expressions

name of 1st staff?
//STAFFLIST/STAFF[1]/NAME



XML Query Languages

- **XML Query language**
  - **XML-QL, UnQL, XQL, XQuery, …**

- **XQuery by W3C XML Query Working**
  - **Contributions from**
    - **database community**
    - **document community**
    - **programming language community**

Note: SQL not good for XML (irregularity of XML data)

XQuery

- **XQuery**
  - **a functional language**
    - **Meaning: a query is represented as an expression**
    - **nested expressions possible**

- **XQuery uses**
  1. **Path expressions (use the syntax of XPath)**
  2. **FLWOR expressions**

- **The queries**
  - operate on ***single documents***
  - operate also on ***fixed collections of documents***
  - **select *sub-trees* of documents**

XQuery: Specifying Queries in XML

- FLWOR expression

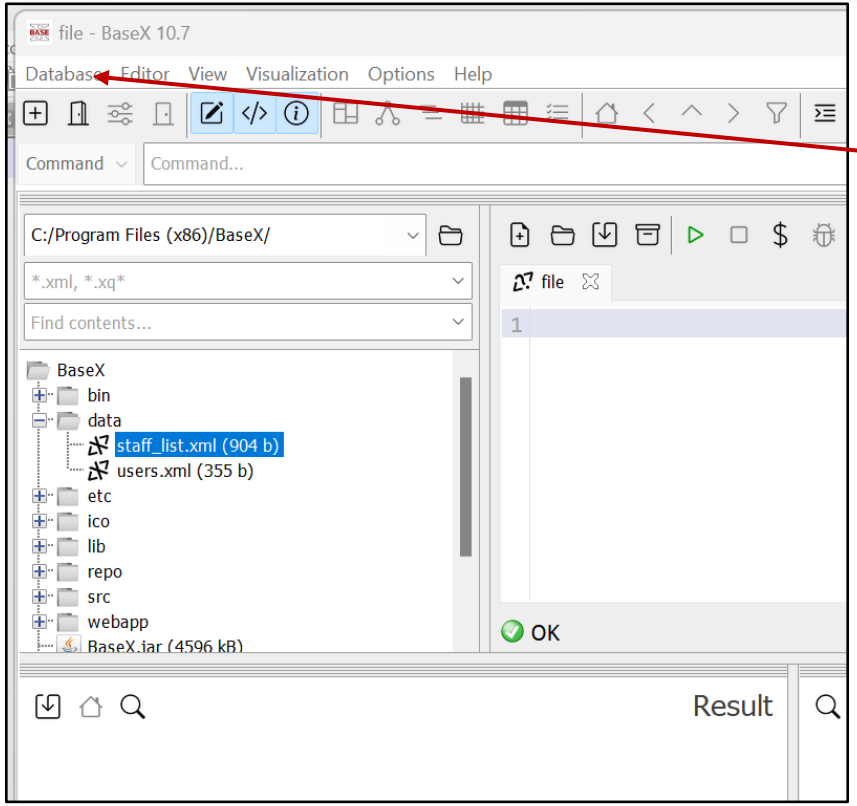
```
FOR <variable bindings to individual nodes (elements)>
LET <variable bindings to collections of nodes (elements)>
WHERE <qualifier conditions>
ORDER BY <ordering specifications>
RETURN <query result specification>
```

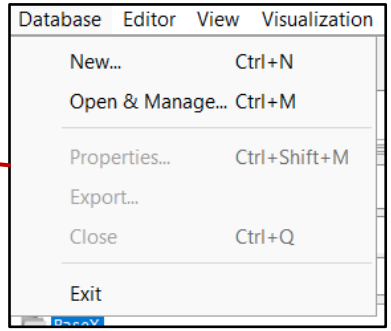
- Variables preceded with $
- For assigns variable to a range
- Where specifies additional conditions
- Order by specifies order of result elements
- Return specifies elements for retrieval

Introduction to BaseX

- BaseX is an open source robust, high-performance XML database engine.
- It was created by Christian Grün in 2005 at the University of Konstanz in Germany.
- It is specialised in storing, querying, and visualizing large XML documents and collections
- It serves as excellent framework for building complex data-intensive web applications.
- It comes with interactive user interfaces (desktop, web-based) giving great insight into semi-structured data.
- Supports both the XPath and XQuery languages.

BaseX 11.2 – DBMS for XML data (1)





Database menu

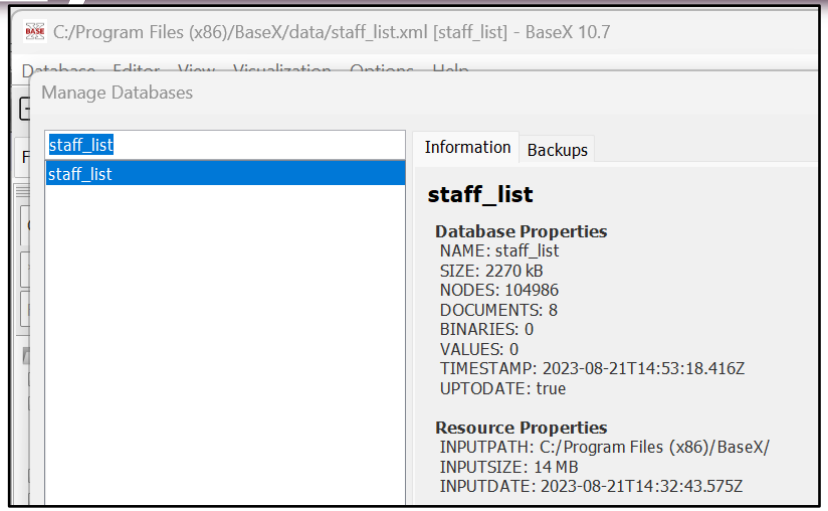
(1) Navigate to the Data folder under C:\ Program Files (x86) \BaseX & copy your XML files to this folder. **Then open BaseX**

(2) To create a new DB

Click on *Database | New*.

Then type in the

BaseX 11.2 – DBMS for XML data (2)

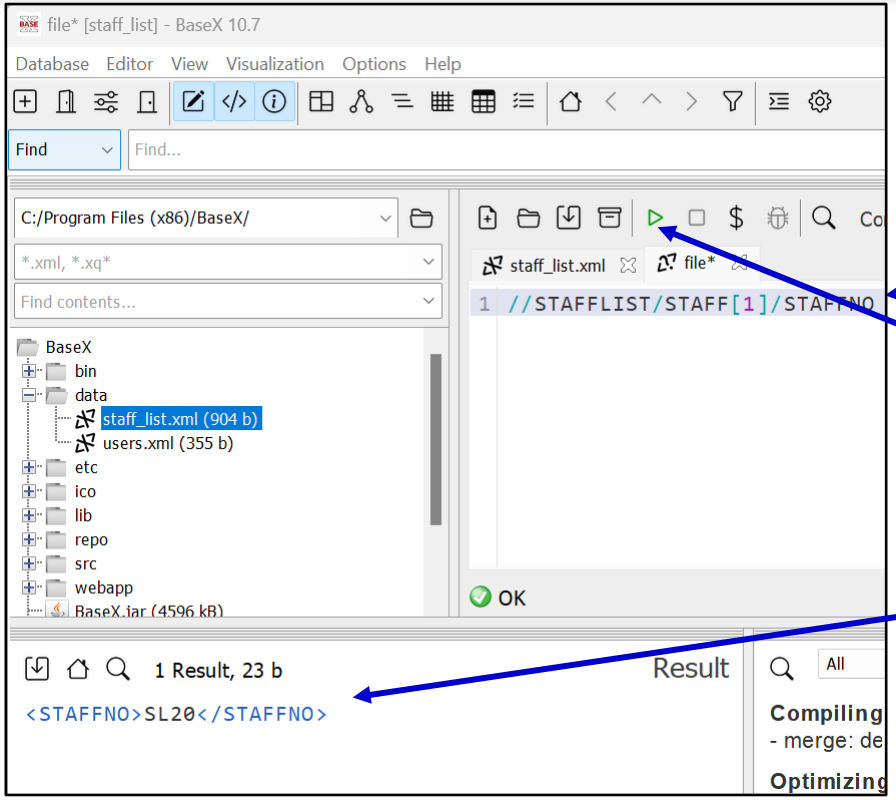


To open an existing DB in the folder data (or other name)

Click on *Database | Open & Manage*...

Then click on the database name, e.g. Data and then click on Open.

BaseX 11.2 – DBMS for XML data (3)



e.g. Query:
//STAFFLIST/STAFF[1]/STAFFNO

1. **Type a query in Query Editor**
2. **Click on Execute query button**
3. **Query results pane**

Find staff number of first member of staff in staff_list.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<STAFFLIST>
    <STAFF branchNo="B002">
        <STAFFNO>SL20</STAFFNO>
        <NAME>
            <FNAME>Thando</FNAME>
            <LNAME>Mandela</LNAME>
        </NAME>
        <POSITION>Manager</POSITION>
        <DOB>1-Oct-98</DOB>
        <SALARY>30000</SALARY>
    </STAFF>
    <STAFF branchNo="B003">
        <STAFFNO>SL21</STAFFNO>
        <NAME>
            <FNAME>Marlize</FNAME>
            <LNAME>Kruger</LNAME>
        </NAME>
        <POSITION>Accountant</POSITION>
        <SALARY>12000</SALARY>
        <DOB>31-Nov-02</DOB>
        <SALARY>27000</SALARY>
```

Query: //STAFFLIST/STAFF[1]/STAFFNO

1 Result, 23 b Result

```xml
<STAFFNO>SL21</STAFFNO>
```

Find the last names for staff at branch B003

Query: //STAFFLIST/STAFF[@branchNo="B003"]/NAME/LNAME/t

```
13    <STAFF branchNo="B003">
14        <STAFFNO>SL21</STAFFNO>
15        <NAME>
16            <FNAME>Marlize</FNAME>
17            <LNAME>Kruger</LNAME>
18        </NAME>
19        <POSITION>Accountant</POSITION>
20        <SALARY>12000</SALARY>
21        <DOB>31-Nov-02</DOB>
22        <SALARY>27000</SALARY>
23    </STAFF>
24    <STAFF branchNo="B003">
25        <STAFFNO>SL22</STAFFNO>
26        <NAME>
27            <FNAME>Lerato</FNAME>
28            <LNAME>Mpho</LNAME>
29        </NAME>
30        <POSITION>Administrator</POSITION>
31        <SALARY>25000</SALARY>
32        <DOB>12-Jan-01</DOB>
33        <SALARY>30000</SALARY>
```

2 Results, 12 b

Result

```
Kruger
Mpho
```

Find the first names for all staff

```xml
<?xml version="1.0" encoding="UTF-8"?>
<STAFFLIST>
    <STAFF branchNo="B002">
        <STAFFNO>SL20</STAFFNO>
        <NAME>
            <FNAME>Thando</FNAME>
            <LNAME>Mandela</LNAME>
        </NAME>
        <POSITION>Manager</POSITION>
        <DOB>1-Oct-98</DOB>
        <SALARY>30000</SALARY>
    </STAFF>
    <STAFF branchNo="B003">
        <STAFFNO>SL21</STAFFNO>
        <NAME>
            <FNAME>Marlize</FNAME>
            <LNAME>Kruger</LNAME>
        </NAME>
        <POSITION>Accountant</POSITION>
        <SALARY>12000</SALARY>
        <DOB>31-Nov-02</DOB>
        <SALARY>27000</SALARY>
    </STAFF>
    <STAFF branchNo="B003">
        <STAFFNO>SL22</STAFFNO>
        <NAME>
            <FNAME>Lerato</FNAME>
            <LNAME>Mpho</LNAME>
        </NAME>
        <POSITION>Administrator</POSITION>
        <SALARY>25000</SALARY>
```

Query:
//STAFFLIST/STAFF/NAME/FNAME/text()

3 Results, 23 b Result

```
Thando
Marlize
Lerato
```

COS 326
Database Systems

Lecture 11

Semi-Structured Data and XML (3)

Chapter 13 (7th edition)

Tuesday 27 August 2024

In this lecture

- Semi-structured data (continued)
- XQuery
  - XPath expressions (last lecture)
  - FLWOR queries
- Files available on ClickUP
  - staff_list.xml and nok.xml

Recap

- XML Languages
- XPath: path expressions
- XQuery
- Introduction to BaseX
- Writing XPath expressions in BaseX

- Files available on ClickUP
  - XML file for this lecture: staff_list.xml

XQuery – FLWOR Expressions

FLWOR ("flower") expression construction:

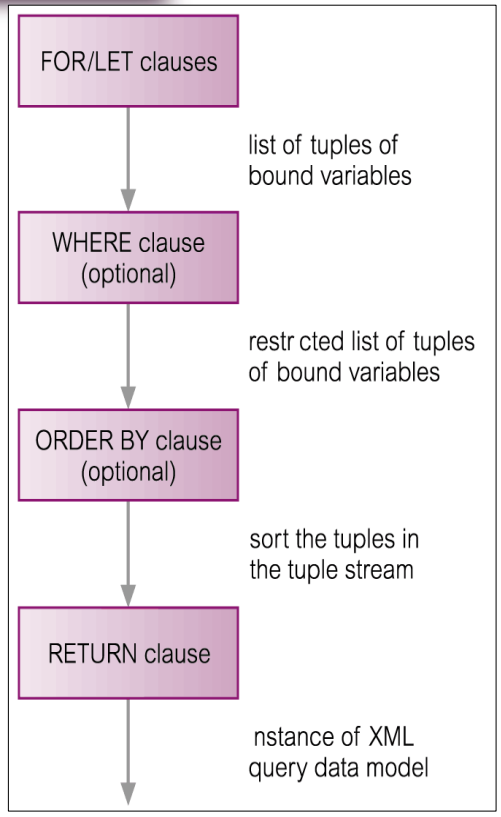
(1) starts with one or more **FOR or LET** clauses (any order)

(2) followed by optional **WHERE** clause

(3) optional **ORDER BY** clause

(4) required **RETURN** clause

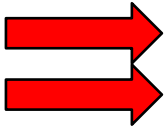
```
FOR      forVar    IN    inExpression
LET      letVar := letExpression
[WHERE             filterExpression ]
[ORDER BY  orderSpec ]
RETURN             expression
```



FLWOR Expressions

- **FOR e.g. for $S in //STAFF**
  - **iteration:** associates variable $S with each expression value
  - result is tuple stream

- **LET e.g let $sal := 30000**
  - **binds one or more variables to one or more expressions**
  - without iteration (single binding for each variable)



```
FOR       forVar    IN   inExpression
LET       letVar   :=    letExpression
[WHERE    filterExpression]
[ORDER BY  orderSpec]
RETURN    expression
```

FLWOR Expressions

- **WHERE** **e.g.** **where** **$S/SALARY** **>** **10000**
  - one or more conditions to restrict tuples generated by FOR and LET (*optional*)

```
FOR        forVar IN inExpression
LET        letVar := letExpression
[WHERE     filterExpression]
[ORDER BY  orderSpec]
RETURN     expression
```

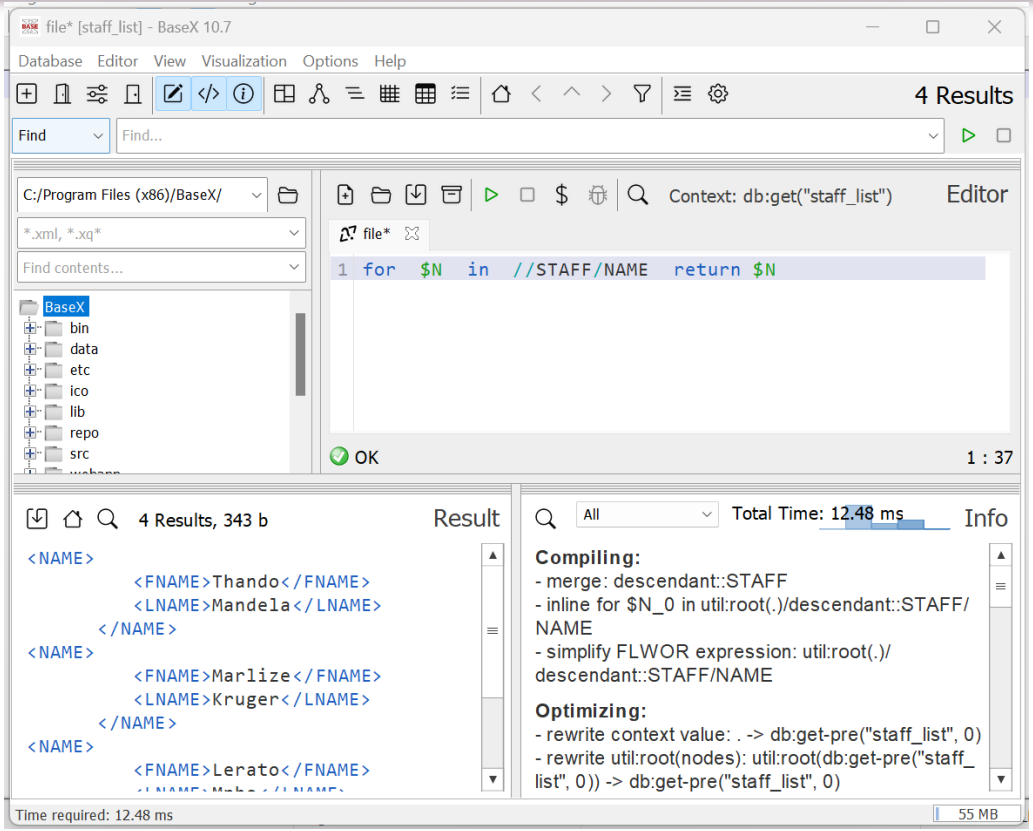
FLWOR Expressions

- **ORDER BY e.g. order by $S/STAFFNO descending**
  - order of the tuple stream
  - determines order in which RETURN clause is evaluated

- **RETURN e.g return $S**
  - evaluated once for each tuple in tuple stream
  - *results concatenated to form final result*

```
FOR       forVar IN inExpression
LET       letVar := letExpression
[WHERE            filterExpression]
[ORDER BY  orderSpec]
RETURN            expression
```

Example 1 – XQuery FLWOR Expressions in BaseX 11.2



List the names of all staff

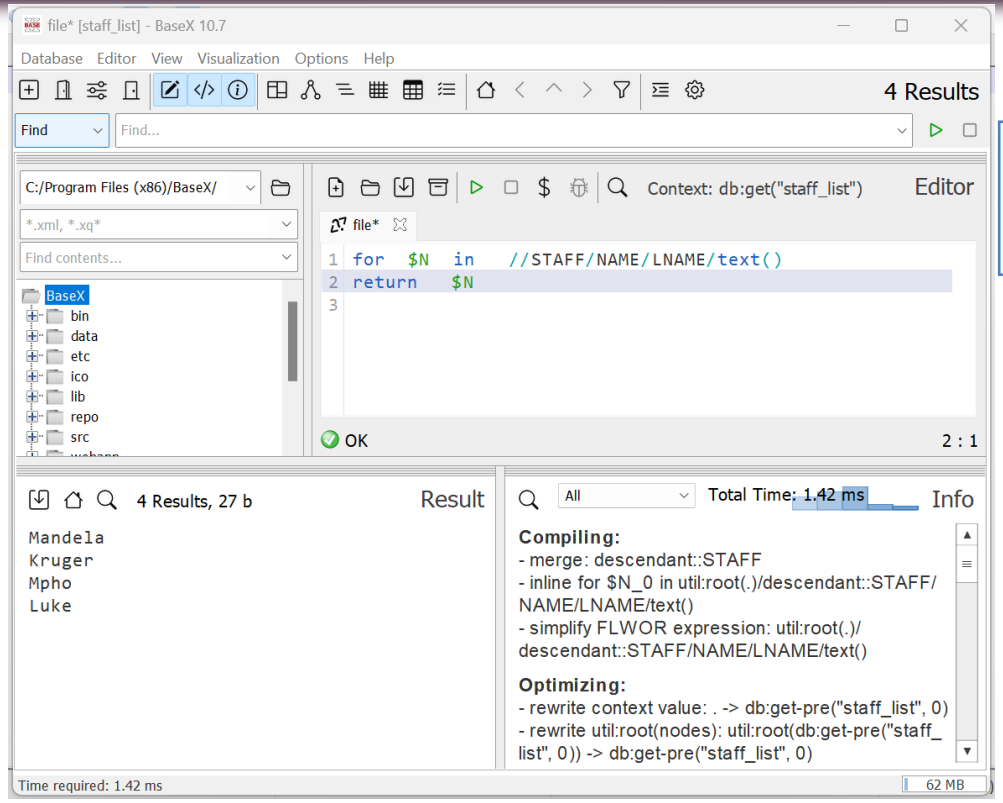
```
for $N in //STAFF/NAME
return $N
```

Note:

Must use lowercase for:

FOR, LET, WHERE, RETURN, etc.

Example 2 – XQuery FLWOR Expressions in BaseX 11.2



List the last names of

```
for $N in
//STAFF/NAME/LNAME/text()
return $N
```

Note:

You can use the text() function to extract values

Example 3 – XQuery FLWOR Expressions

2 Results, 511 b

```
<STAFF branchNo="B002">
      <STAFFNO>SL20</STAFFNO>
      <NAME>
         <FNAME>Thando</FNAME>
         <LNAME>Mandela</LNAME>
      </NAME>
      <POSITION>Manager</POSITION>
      <DOB>1-Oct-98</DOB>
      <SALARY>30000</SALARY>
   </STAFF>
<STAFF branchNo="B004">
      <STAFFNO>SL22</STAFFNO>
      <NAME>
         <FNAME>Lerato</FNAME>
         <LNAME>Mpho</LNAME>
      </NAME>
      <POSITION>Administrator</POSITION>
→   <DOB>12-Jan-01</DOB>
      <SALARY>30000</SALARY>
   </STAFF>
```

List staff with salary =

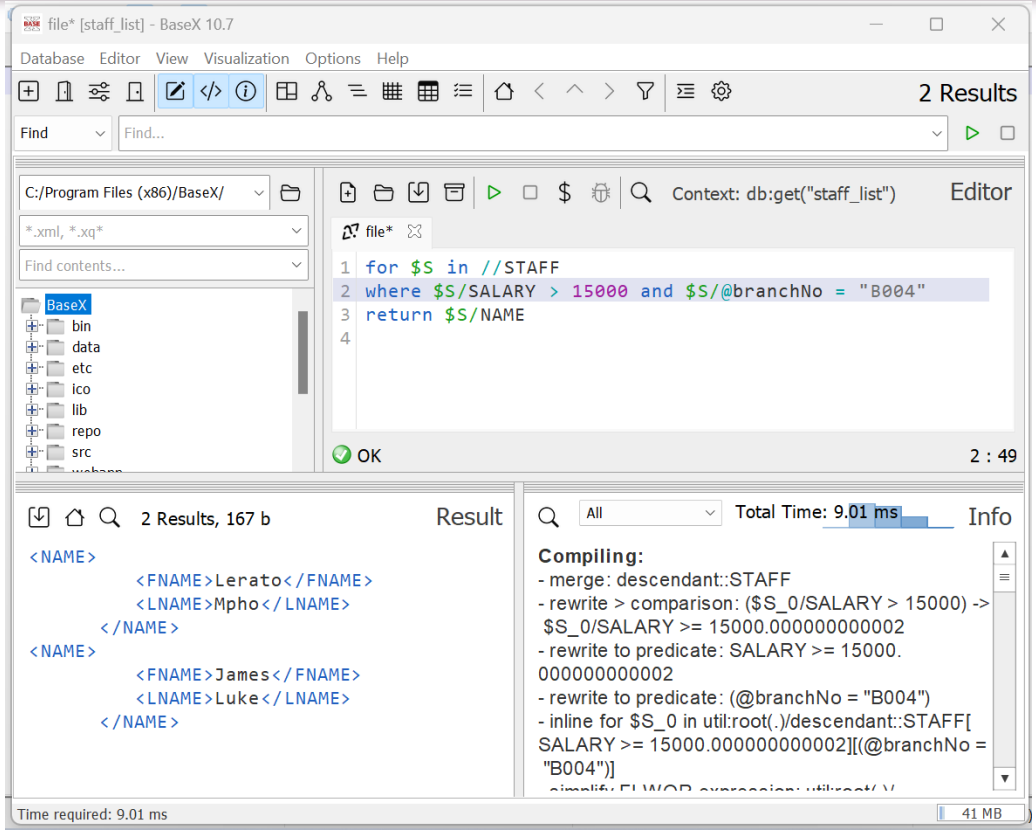
```
let $SAL :=
30000
return
//STAFF[SALARY
= $SAL]
```

**Note:
var names are case sensitive

- other operators:
  - 'eq', 'ne', 'lt', 'le', 'gt', 'ge'

value comparison operators

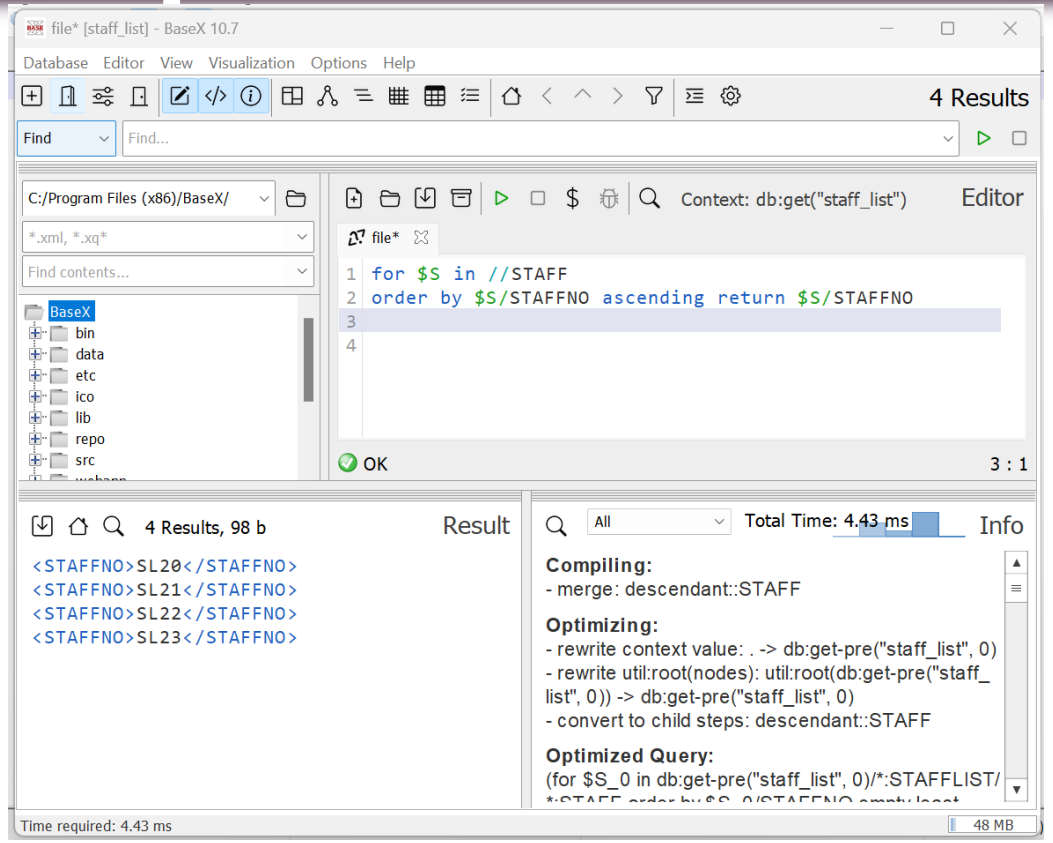
Example 4 – XQuery FLWOR Expressions



Find staff name at branch B004 with salary >

```
for $S in //STAFF
where $S/SALARY > 15000 and $S/@branchNo = "B004"
return $S/NAME
```

Example 5 – XQuery FLWOR Expressions



List staff numbers of all staff in ascending order of staff

```
for $S in //STAFF
order by $S/STAFFNO
ascending return
$S/STAFFNO
```

Example 6 – XQuery FLWOR Expressions

List each branch office and average salary at

```
for $B in  distinct-values(//@branchNo)
let $avgSalary :=
          avg(//STAFF[@branchNo =
$B]/SALARY)
return
          <BRANCH>
                    <BRANCHNO>{$B
}</BRANCHNO>
```

Functions for aggregates are: avg, etc.

Homework

3 Results, 214 b Result

```
<BRANCH><BRANCHNO>B002</BRANCHNO><AVGSALARY>30000</AVGSALARY></
BRANCH>
<BRANCH><BRANCHNO>B003</BRANCHNO><AVGSALARY>27000</AVGSALARY></
BRANCH>
<BRANCH><BRANCHNO>B004</BRANCHNO><AVGSALARY>31500</AVGSALARY></
BRANCH>
```

Example 7 – XQuery FLWOR Expressions

List branches with at least one member of staff with

```
for $B in distinct-values(//@branchNo)
let $S := //STAFF[@branchNo = $B]
   where some $sal in $S/SALARY
         satisfies ($sal > 15000)

   return <BRANCHNO>{
$B}</BRANCHNO>
```

sub-query

some: existential qualifier

every: universal qualifier

3 Results, 79 b

Result

```
<BRANCHNO>B002</BRANCHNO>
<BRANCHNO>B003</BRANCHNO>
<BRANCHNO>B004</BRANCHNO>
```

Example 1 – Joining Two Documents

List names of staff and names of their next of kin

```
for $S in //STAFF,
    $NOK in //NOK
where $S/STAFFNO =
$NOK/STAFFNO
return
    <STAFFNOK>
     { $S/NAME,
$NOK/NAME }
    </STAFFNOK>
```

4 Results, 527 b Result

```
<STAFFNOK><NAME>
          <FNAME>Thando</FNAME>
          <LNAME>Mandela</LNAME>
       </NAME><NAME>Xolani Mandela</NAME></STAFFNOK>
<STAFFNOK><NAME>
          <FNAME>Marlize</FNAME>
          <LNAME>Kruger</LNAME>
       </NAME><NAME>Johan Kruger</NAME></STAFFNOK>
<STAFFNOK><NAME>
          <FNAME>Lerato</FNAME>
          <LNAME>Mpho</LNAME>
       </NAME><NAME>Tumelo Mpho</NAME></STAFFNOK>
<STAFFNOK><NAME>
          <FNAME>James</FNAME>
          <LNAME>Luke</LNAME>
       </NAME><NAME>Martha Luke</NAME></STAFFNOK>
```

Example 2 – Joining Two Documents

List names of staff and names of their next of

```
for $S in //STAFF
return <STAFFNOK>
{ $S/NAME }
   { for $NOK in //NOK
   where $S/STAFFNO
=$NOK/STAFFNO
return $NOK/NAME }
</STAFFNOK>
```

Correlated sub-query

4 Results, 527 b

Result

```
<STAFFNOK><NAME>
          <FNAME>Thando</FNAME>
          <LNAME>Mandela</LNAME>
      </NAME><NAME>Xolani Mandela</NAME></STAFFNOK>
<STAFFNOK><NAME>
          <FNAME>Marlize</FNAME>
          <LNAME>Kruger</LNAME>
      </NAME><NAME>Johan Kruger</NAME></STAFFNOK>
<STAFFNOK><NAME>
          <FNAME>Lerato</FNAME>
          <LNAME>Mpho</LNAME>
      </NAME><NAME>Tumelo Mpho</NAME></STAFFNOK>
<STAFFNOK><NAME>
          <FNAME>James</FNAME>
          <LNAME>Luke</LNAME>
      </NAME><NAME>Martha Luke</NAME></STAFFNOK>
```

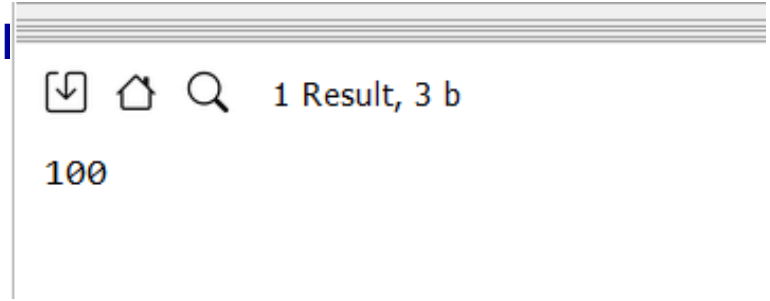
Example 3 – User-Defined Function (1)

Simple arithmetic function in BaseX

FUNCTION DEFINITION:

```
let $mult := function($x, $y) {$x * $y}
return $mult(10,10)
```

FUNCTION CALL



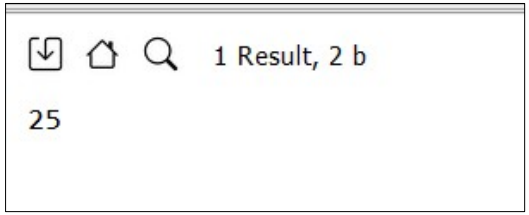
Example 4– User-Defined Function (2)

Simple arithmetic function in

FUNCTION DEFINITION:

```
let $f := math:pow#2
return $f(5, 2)
```

FUNCTION CALL:



Built-in functions:

Reading for the student

COS 326
Database Systems

Lecture 12

Semi-Structured Data and XML (4)

Chapter 13 (7th edition)

Thursday 29 August 2024

In this lecture

XML databases and SQL

1. Storing XML data in a RDB or ORDB
2. XML and SQL standards
3. Creating a DB table using the XML data type
4. SQL / XML functions
5. SQL / XML operators
6. SQL / XML mapping functions
7. PostgreSQL and XML data
8. Importing XML data into a PostgreSQL DB

1. Storing XML data in a RDB or ORDB

Storing XML in a Relational or Object-Relational database

- **Four general approaches:**
  1. **Store XML document as *value of a column* in one row or several rows**
  2. **Store XML document in a *shredded* form across a number of columns and tables**
  3. **Store XML document in a *schema independent* form**

Approach 1: Storing XML in a table column

docNo	docDate	XMLdoc (whole doc or fragment)
D001	2020-05-20	<STAFFLIST> <STAFF branchNo = "B002"> ... </STAFF> </STAFFLIST>

- **Data types (for XML data)**
  - **CLOB (character large object) – used in the past**
  - **Native XML data type: xml (PostgreSQL) or XMLType (Oracle)**

- **Raw XML stored as an attribute (in a column) in a table row**
  - **efficient to *insert documents* into database**
  - **efficient to *retrieve documents*** easy to apply full-text *indexing to documents*
  - **updates: *entire XML document* is replaced (more recently an XML doc can be updated using SQL)**
  - **general query performance is poor due to parsing on the fly**

Approach 2: Storing XML in Shredded Form

XML decomposed (shredded) into constituent elements

- **data distributed over number of attributes (columns) in one or more relations**
- **easier to *index values of individual elements***
- **need additional *data relating to hierarchical nature of the XML***
  - **to recompose original document**
  - **XML updates**
- **Have to create appropriate database structure from schema: relational or object-relational**

(Reading for the student: ch. 13.6)

BRANCHNO	STAFFNO	NAME	FNAME	LNAME
BOO2	SL20		Thando	Mandela
BOO3	SL21		Marlize	Kruger

Approach 3: Schema-Independent Representation

e.g.
create a relation (table) from the Document Object Model (DOM) for the xml document

```
<STAFFLIST>
   <STAFF branchNo="B002">
      <STAFFNO>SL20</STAFFNO>
      <NAME>
         <FNAME>Thando</FNAME>
         <LNAME>Mandela</LNAME>
      </NAME>
      <POSITION>Manager</POSITION>
      <DOB>1-Oct-98</DOB>
      <SALARY>30000</SALARY>
   </STAFF>
```

nodeID	nodeType	nodeName	nodeData	parentID	rootID
0	Document	STAFFLIST			0
1	Element	STAFFLIST		0	0
2	Element	STAFF		1	0
3	Element	STAFFNO		2	0
4	Text		SL20	3	0
5	Element	NAME		2	0
6	Element	FNAME		5	0
7	Text		Thando	6	0
8	Element	LNAME		5	0
9	Text		Mandela	8	0

2. XML and SQL standards

- In SQL:2003, SQL:2008 and SQL:2011

1. Native XML data type: XML
2. Set of XML/SQL operators for the type
3. Set of XML/SQL functions
4. Set of mappings from relational data to XML

Creating Table using XML data type

```
CREATE TABLE XMLStaff (
                docNo CHAR(4) PRIMARY KEY,
                 docDate DATE, staffData XML  );

INSERT INTO XMLStaff VALUES ('D001',
DATE '2023-08-31',
XML      ('<STAFF branchNo  =    "B002">
<STAFFNO>   SL20       </STAFFNO>
<POSITION>  Manager   </POSITION>
<DOB>       1998-10-01   </DOB>
<SALARY>    30000       </SALARY>  </STAFF>') );
SELECT * FROM XMLStaff;
```

	docno [PK] character	docdate date	staffdata xml
1	D001	2023-08-31	<STAFF branchNo = "B002">

Note: staffdata result is partially displayed in this figure. Double click to see all data

SQL/XML Functions (reading for the student)

- **XMLFOREST**
  - generates XML value with a list of elements as children of a root item.
- **XMLCONCAT**
  - concatenates a list of XML values
- **XMLPARSE**
  - performs a non-validating parse of a character string to produce an XML value
- **XMLROOT**
  - creates an XML value by modifying the properties of the root item of another XML value
- **XMLCOMMENT**
  - generates an XML comment
- **XMLPI**
  - generates an XML processing instruction
- **XMLSERIALIZE**
  - generates a character or binary string from an XML value
- **XMLAGG**
  - aggregate function, to generate a forest of elements from a collection of elements

SQL/XML Operators (1)

```
CREATE TABLE staff (
      staffNo CHAR(4), fName CHAR(10),
       lName CHAR(10), branchNo CHAR(4) );

INSERT INTO staff VALUES ('SL20', 'Thando', 'Mandela', 'B002');
INSERT INTO staff VALUES ('SL21', 'Marlize', 'Kruger', 'B003’);

SELECT * FROM STAFF;
```

	staffno character	fname character	lname character	branchno character
1	SL20	Thando	Mandela	B002
2	SL21	Marlize	Kruger	B003

SQL/XML Operators (2)

- **XMLELEMENT**
  - generates XML value with a single element as a child of its root item.

e.g.

```
SELECT
    xmlelement ( NAME staffname,
  concat(fName,lName)  )
FROM Staff;
```

	xmlelement xml
1	<staffname>Thando Mandela </staffname>
2	<staffname>Marlize Kruger </staffname>

SQL/XML Operators (3)

- **Element can have zero or more attributes**
  - specified in the **XMLATTRIBUTES** sub-clause.
- **e.g.**

```
SELECT xmlelement   ( NAME staff,
                    xmlattributes (branchNo AS
"branchNo" ) )
FROM Sta
```

	xmlelement xml
1	<staff branchNo="B002"/>
2	<staff branchNo="B003"/>

SQL/XML Operators (4)

List all staff as XML elements with name and branch number as an attribute

```
SELECT staffNo,  xmlelement
                        (   NAME staff,
                            xmlattributes ( branchNo AS "branchNo" ),
                             concat(fName,lName)
                        )   AS   staffXMLCol   FROM Staff;
```

	staffno character (4)	staffxmlcol xml
1	SL21	<staff branchNo="B005">John White </staff>
2	SG37	<staff branchNo="B003">Ann Beech </staff>

SQL/XML Mapping Functions

SQL/XML

- **defines functions for mapping from tables to XML documents**

- mappings produces two types of XML documents:

  1. **mapped table data**
     - e.g. **PostgreSQL:** **table_to_xml**

  2. **XML Schema describing the mapped table data**
     - e.g. **PostgreSQL:** **table_to_xmlschema**

PostgreSQL and XML data (1)

Functions for converting tables to XML

(PostgreSQL 16 documentation)

Queries:

```
CREATE TABLE numbers( num1 int, nu
    int);
INSERT INTO NUMBERS VALUES(1,2);
INSERT INTO NUMBERS VALUES(3,4);
INSERT INTO NUMBERS VALUES(5,6);

SELECT * FROM NUMBERS;
```

	num1 integer	num2 integer
1	1	2
2	3	4
3	5	6

PostgreSQL and XML data (2)

Queries to generate XML from table

```
SELECT table_to_xml('Numbers', false, false,'');
--OR:
SELECT query_to_xml('SELECT num1, num2 FROM
    Numbers', false,false, '');
```

Output (when table_to_xml is used):

	table_to_xml xml
1	<numbers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"> <row> <num1>1</num1> <num2>2</num2> </row> <row> <num1>3</num1> <num2>4</num2> ...

PostgreSQL and XML data (3)

Generate XML schema from table

Query:

```
SELECT
    table_to_xmlschema('Numbers',
    false,false,'');
--OR:
SELECT
    query_to_xmlschema('SELECT
    num1, num2
FROM Numbers', false,false,'');
```

Output when table_to_xmlschema is used (partial listing):

```
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:simpleType name="INTEGER">
  <xsd:restriction base="xsd:int">
    <xsd:maxInclusive value="2147483647"/>
    <xsd:minInclusive value="-2147483648"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="RowType.StaffList.public.numbers">
  <xsd:sequence>
    <xsd:element name="num1" type="INTEGER" minOccurs="0"></xsd:element>
    <xsd:element name="num2" type="INTEGER" minOccurs="0"></xsd:element>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="TableType.StaffList.public.numbers">
  <xsd:sequence>
    <xsd:element name="row" type="RowType.StaffList.public.numbers" minOccurs="0" maxOc
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="numbers" type="TableType.StaffList.public.numbers"/>

</xsd:schema>
```

Creating Table using XML Type

```
CREATE TABLE XMLStaff (
            docNo CHAR(4) PRIMARY KEY,
             docDate DATE, staffData XML  );

INSERT INTO XMLStaff VALUES ('D002', DATE '2023-08-31',
XML('<STAFF branchNo = "B003">
<STAFFNO>SL21</STAFFNO>
<POSITION>Accountant</POSITION>
<DOB>2002-11-02</DOB>
<SALARY>27000</SALARY> </STAFF>') );
SELECT * FROM XMLStaff;
```

	docno [PK] character	docdate date	staffdata xml
1	D001	2023-08-31	<STAFF branchNo = "B002">
2	D002	2023-08-31	<STAFF branchNo = "B003">

PostgreSQL and XML data (4a)

SQL and XPath queries:

syntax is: *xpath('xpath_expression', colname)*

Returns an XML element in curly brackets

```
SELECT    xpath ('/STAFF/STAFFNO',staffData) AS StaffNumberXML
FROM XMLStaff;
```

Results:

	staffnumberxml xml[]
1	<STAFFNO> SL20 </STAFFNO>
2	<STAFFNO>SL21</STAFFNO>

PostgreSQL and XML data (4b)

SQL and XPath queries:

syntax is: ***xpath('xpath-expression', colname)***

***Use text() function to remove tags

```
SELECT  docNo, docDate,
    xpath ('/STAFF/@branchNo',staffData)        AS
    BranchNumber,
    xpath ('/STAFF/STAFFNO/text()',staffData) AS StaffNumber,
    xpath ('/STAFF/POSITION/text()',staffData) AS Position,
    xpath ('/STAFF/DOB/text()',staffData)          AS DOB,
    xpath ('/STAFF/SALARY/text()',staffData)   AS Salary
FROM XMLStaff;
```

Results:

	docno [PK] character	docdate date	branchnumber xml[]	staffnumber xml[]	position xml[]	dob xml[]	salary xml[]
1	D001	2023-08-31	B002	SL20	Manager	1998-10-01	30000
2	D002	2023-08-31	B003	SL21	Accountant	2002-11-02	27000

PostgreSQL and XML data (5)

SQL, XPath queries & unnest function:

. The unnest function is converting an array into a table-like structure; we can also generate a table structure of an array using unnest function in PostgreSQL

```
SELECT  docNo, docDate,
    unnest(xpath ('/STAFF/@branchNo',staffData) )    AS
    BranchNumber,
    unnest(xpath ('/STAFF/STAFFNO/text()',staffData)) AS
    StaffNumber,
    unnest(xpath ('/STAFF/POSITION/text()',staffData) ) AS Position,
    unnest(xpath ('/STAFF/DOB/text()',staffData) ) AS DOB,
    unr
FROM
```

Result

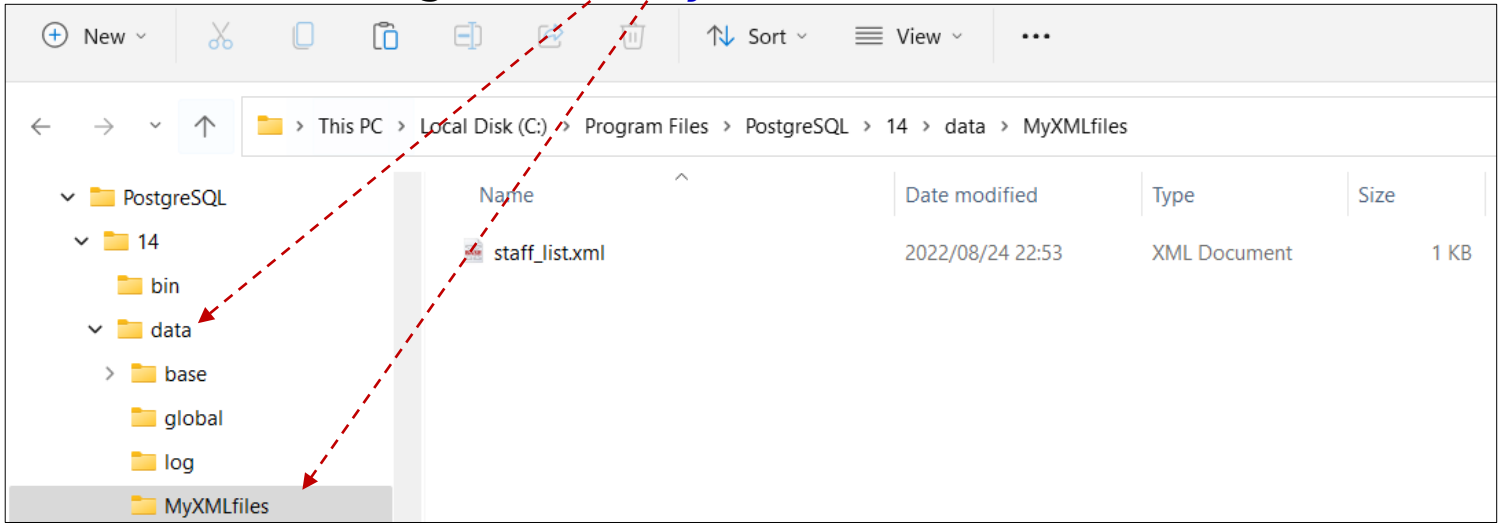
	docno [PK] character	docdate date	branchnumber xml	staffnumber xml	position xml	dob xml	salary xml
1	D001	2023-08-31	B002	SL20	Manager	1998-10-01	30000
2	D002	2023-08-31	B003	SL21	Accountant	2002-11-02	27000

Importing XML data into PostgreSQL DB (1)

- the current working directory (cwd) for pgAdmin4 is

  C:\Program Files\PostgreSQL\16\data

- So, create a folder under ‘data’ and store your XML files here, e.g. folder *‘MyXMLfiles’*



Importing XML data into PostgreSQL DB (2)

e.g. query to list the files/folders under the cwd (data)

SELECT pg_ls_dir('.');

query to list files/folders in the cwd sub-folder MyXML files:

SELECT pg_ls_dir('./MyXMLfiles');

Data output | Messages | Graph Visualiser

	pg_ls_dir text
1	base
2	current_logfiles
3	global
4	log
5	MyXMLfiles
6	pg_commit_ts
7	pg_dynshmem
8	pg_hba.conf
9	pg_ident.conf
10	pg_logical
11	pg_multixact
12	pg_notify

Data output | Messages | Graph Visualiser

	pg_ls_dir text
1	staff_list.xml

Importing XML data into PostgreSQL DB (3)

Query to read contents of a file & return contents **as text type**

```
SELECT
pg_read_file('./MyXMLfiles/staff_list.xml');
```

Data output | Messages | Graph Visualiser | Notifications

	pg_read_file text
1	<?xml version="1.0" encoding="UTF-8"?> <STAFFLIST> <STAFF branchNo="B005"> <STAFFNO>SL21</STAFFNO> <NAME> <FNAME>John</FNAME> <LNAME>...

Importing XML data into PostgreSQL DB (4)

Query to read & present file contents as xml type

```
SELECT
CAST(pg_read_file('./MyXMLfiles/staff_list.xml')
```

	pg_read_file text
1	<?xml version="1.0" encoding="UTF-8"?>

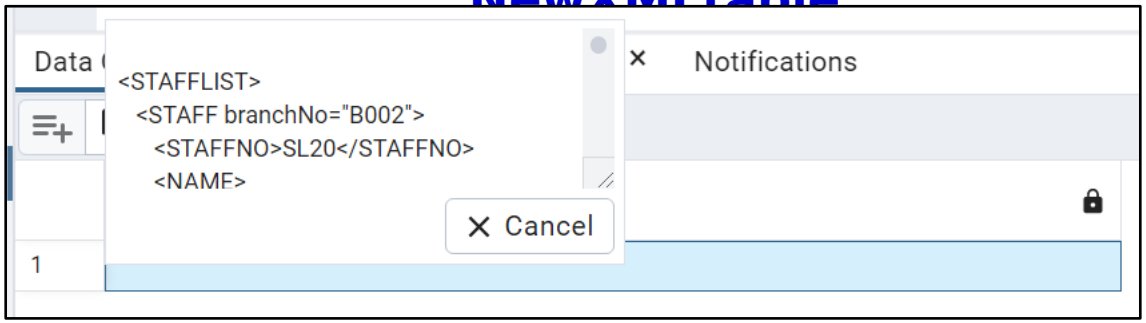
Importing XML data into PostgreSQL DB (5)

Create a temp xml table & load xml file contents into this table

```
SELECT (CAST( pg_read_file
( './MyXMLfiles/staff_list.xml')
AS xml) )
AS      xmldata
INTO  NewXMLtable;
```

then:

```
SELECT * FROM
NewXMLtable;
```



Importing XML data into PostgreSQL DB (6)

Use xpath to create an array of **<STAFF> …</STAFF>** elements.

```
SELECT
    xpath
('//STAFF',xmldata)
    AS XMLcol
    FROM NewXMLtable;
```

HOMEWORK:

Write a plpgsql function to store array elements into a table, with one element per row

Query result:

```
{ " <STAFF
branchNo=\"B002\">
  <STAFFNO>SL20</STAFFNO>
  <NAME>
<FNAME>Thando</FNAME>

<LNAME>Mndela</LNAME>
   </NAME>

<POSITION>Manager</POSITIO
N>
   <DOB>1998-10-01</DOB>
  <SALARY>30000</SALARY>
   </STAFF> ",

  " <STAFF
```

COS 326
Database Systems

Lecture 13

Big Data & NoSQL Databases – MongoDB (1)

Chapters 24 & 25(7th edition)

Tuesday 03 September 2024

In this lecture

1. **What is Big Data?**
2. **Sources of Big Data**
3. **NoSQL databases & data models for big data**
4. **MongoDB NoSQL document database**

References (available on ClickUP):

1. **Moniruzzaman, A.B.M. & Hossain, S.A. (2013)**
   NoSQL database: new era of databases for big data analytics – classification, characteristics and comparison. *International Journal of Database Theory and Application*, vol. 6, no. 4, 2013.

2. **MongoDB installation and documentation links.**

1. What is big data?

Big data is a collection of data that is huge in volume, yet growing exponentially with time. The data is so large in size and complexity that none of the traditional data management tools can store it or process it efficiently.

The 3 Vs of Big Data: Volume, Velocity, Variety

a. Big volume:

- Data at the scale of terabytes to petabytes (now even yottabytes)

b. Big velocity:

- Data in motion

  => need for analysis of streaming data to enable decisions within fractions of a second

c. Big variety:

Data in many forms:

- structured,
- semi-structured,
- unstructured, text, multimedia

2. Sources of Big Data (1)

A. Web-generated structured & unstructured data e.g.

(1) Click stream data

- Website browsing activities

(2) E-commerce purchasing histories

- Recommender systems

(3) Social networks services

- Facebook (4PB per day), Twitter (X) (12TB per day), LinkedIn, Instagram, etc.

(4) Other social media

- YouTube (440TB per day), blogs, Google street view etc.

2. Sources of Big Data (2)

B. Sensor data

(1) e.g Car manufacturers

- **Ford Focus electric car produces & transmits data as it is driven**

(2) Other sources e.g wireless sensor networks

C. Mobile devices e.g.

- **smart phones (GPS, etc.)**

D. Business data is big data e.g.

(1) Stock exchange

- **Stock trading continuously 24/7**

(2) SANRAL: e-toll sensors

3. Databases for Big Data (1a)

Storage of Big data

Problem: **RDBMS not suitable for storing & processing Big Data**

Solution:

NoSQL database systems:

(NoSQL means Not Only SQL)

- are **distributed non-relational** DBs
- designed for **large-scale data storage**
- and for massively parallel data processing **using a large number of low cost servers (scalability, availability & fault tolerance)**
- **can support thousands or millions of interactive users**

3. Databases for Big Data (1b)

Who uses NoSQL database systems?

- **Arose alongside major Internet businesses which had challenges in storing & processing huge quantities of data**
  - **Google, Amazon, Facebook, Twitter (Now X), Yahoo!, etc.**

- **Today**

  they are used by organisations that
  - **collect large amounts of**
    - **semi-structured &**
    - **unstructured data (for analysis)**

3. Databases for Big Data (2)

Querying the NoSQL database

- **no standard query language**
- **NoSQL DBMSs provide native API with simple data access commands or native Query language e.g**
  - **MongoDB client uses JavaScript**
  - **Neo4j uses Cypher Query language and REST API (get, put, post)**

Classification of NoSQL databases

(1) **Key-value stores**

(2) **Document databases**

(3) **Wide-column stores**

(4) **Graph databases**

24.2 The CAP Theorem

- Various levels of consistency among replicated data items
  - Enforcing serializabilty the strongest form of consistency
    - High overhead – can reduce read/write operation performance
- CAP theorem
  - Consistency, availability, and partition tolerance
  - Not possible to guarantee all three simultaneously
    - In distributed system with data replication

The CAP Theorem (cont'd.)

- Designer can choose two of three to guarantee
  - Weaker consistency level is often acceptable in NOSQL distributed data store
  - Guaranteeing availability and partition tolerance more important
  - Eventual consistency often adopted

3. Data models for Big Data DBs (3a)

(1) Key-value stores:

- **Key:**
  uniquely identifies data item (value)
- **Value:**
  word, number or complex structure with unique semantics (e.g. lists, sets)

- **Data searches:**
  performed on keys, not values
  & limited to exact matches

Example:

Car	
Key	Attributes
1	Make: Nissan Model: Pathfinder Color: Green Year: 2003
2	Make: Nissan Model: Pathfinder Color: Blue Color: Green Year:2005 Transmission: Auto

'igure3: Key/Value Store NoSQL Database (

3. Data models for Big Data DBs (3b)

(1) Key-value stores continued:

- **Primary use:**
  very fast & highly scalable retrieval of values needed for application tasks
  e.g.
  retrieving product names (Amazon's DynamoDB)

- **Example DBs**:
  DynamoDB (Amazon),
  Voldemort (LinkedIn)

Example:

Car	
Key	Attributes
1	Make: Nissan Model: Pathfinder Color: Green Year: 2003
2	Make: Nissan Model: Pathfinder Color: Blue Color: Green Year:2005 Transmission: Auto

'igure3: Key/Value Store NoSQL Database (

3. Data model for Big Data DBs (4a)

(2) Document databases

- **Designed to store documents**
- **Documents are encoded in a standard data exchange format**

  e.g.

  - **XML,**
  - **JSON (JavaScript Object Notation),**
  - **BSON (Binary JSON)**

- **Data stored in key-value pair style but value column is un-structured data**

Example:

Document data model

Collection of complex documents with arbitrary, nested data formats and varying “record” format.

3. Data model for Big Data DBs (4b)

(2) Document databases cntnd

- Number & type of attributes can vary from doc to doc.
- both keys and values are searchable.

Primary use:

- storing text documents,
- e-mail messages,
- XML documents

Example DBs:

- MongoDB,
- Apache's CouchDB

Example:

Document data model

Collection of complex documents with arbitrary, nested data formats and varying "record" format.

3. Data models for Big Data DBs (5a)

(3) Wide-column (or column-family) stores

- use a distributed, column-oriented data structure
- data structure accommodates multiple values per key
- use Google's Bigtable structure and file systems (GFS)
- use MapReduce parallel processing

Example:

A row has many super columns

A super column has many

3. Data models for Big Data DBs (5a)

Example:

A row has many Super columns

A super column has many columns

3. Data models for Big Data DBs (5b)

(3) Wide-column (or column-family) stores continued:

Primary uses:

- **distributed data storage**
- **large-scale batch processing (MapReduce parallel processing),**
- **exploratory + predictive analytics.**

Example:

A row has many super columns.

A y

c

3. Data models for Big Data DBs (5c)

Example:
A row has many supercolumns.
A supercol has many columns.

Example DBs:

- **Google's Bigtable**,
- **Apache's Hadoop**,
- **Cassandra : (used by Meta Platforms: Facebook, WhatsApp, and Instagram)**
- **Amazon's SimpleDB**,
- **Microsoft Azure's Cosmos DB**

http://bi-bigdata.com/2013/01/13/what-is-wide-column-stores

3. Data models for Big Data DBs (6a)

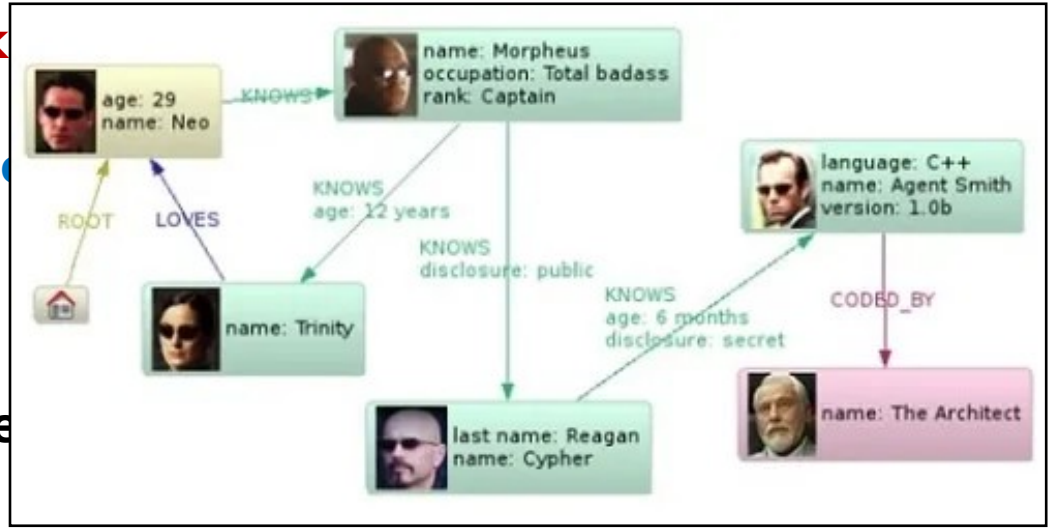
4) Graph databases

Exam ple:

- use structured relationship graphs of interconnected k value pairings

Graph represented as an Objec oriented network of:

- nodes (data objects),
- edges (node relationships),
- properties (object attribute

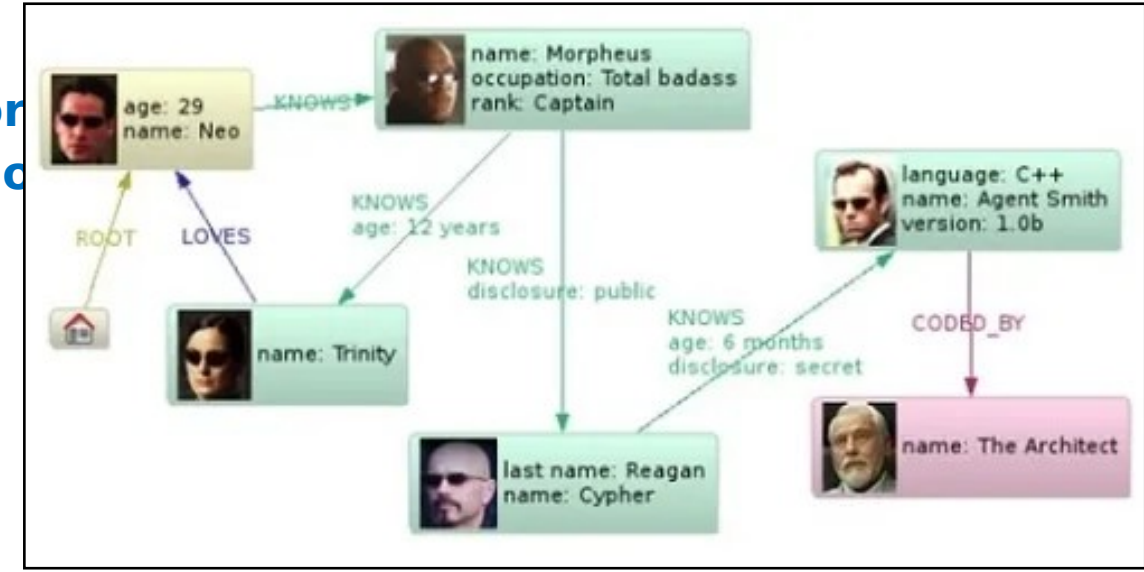


expressed as key-value pairings)

3. Data models for Big Data DBs (6b)

(4) Graph databases cntnd:

Example:



Primary uses:

- representing social networ
- generating recommendatio
- conducting forensic investigations

Example DBs:

- Neo4j,
- InfoGrid,
- AllegroGraph

MongoDB document database (1)

A. Definition of MongoDB

1. MongoDB is a:

a. document-oriented database which provides

i. high performance

ii. high availability

iii. high scalability

b. **database which works on the concepts of collections and documents**

MongoDB stores data records as documents (specifically BSON documents) which are gathered together in collections.

2. A MongoDB database:

physical container for collections

3. **A MongoDB DBMS**

single server with multiple databases

MongoDB document database (2)

4. A MongoDB document:

a. is a set of key-value pairs (a JSON object)

b. is located in a *Collection* with other documents

c. has a dynamic schema, i.e.

docs in the same collection do not need to have the

```
{
  name: "sue",                          ← field: value
  age: 26,                              ← field: value
  status: "A",                          ← field: value
  groups: [ "news", "sports" ]          ← field: value
}
```

Figure 3.1: A MongoDB document.

MongoDB document database (3)

C. Sample MongoDB document
(e.g. of blog data)

```
{
_id:        ObjectId(7df78ad8902c)

title:      'MongoDB Overview',

description:
'MongoDB is no sql database',

by:         'tutorials point',

url:        'http://www.up.ac.za',

tags: ['mongodb', 'database', 'NoSQL'],

likes: 100,
}
```

```
comments:
{[
  {          user:                'user1',
   message:          'My first comment',
   dateCreated:      new
   Date(2011,1,20,2,15),
   likes: 0
   },
  {  user:           'user2',
   message:          'My second
   comments',
   dateCreated:      new
   Date(2011,1,25,7,45),
   likes:  5
   }
 ] }
```

** HOMEWORK: identify data types in

MongoDB document database (4)

3. A MongoDB collection is a:

a. group of MongoDB documents

b. approx. same purpose as RDBMS table

c. does not enforce a schema

d. documents within a collection can have different fields

e. typically all documents in a collection are of similar or related

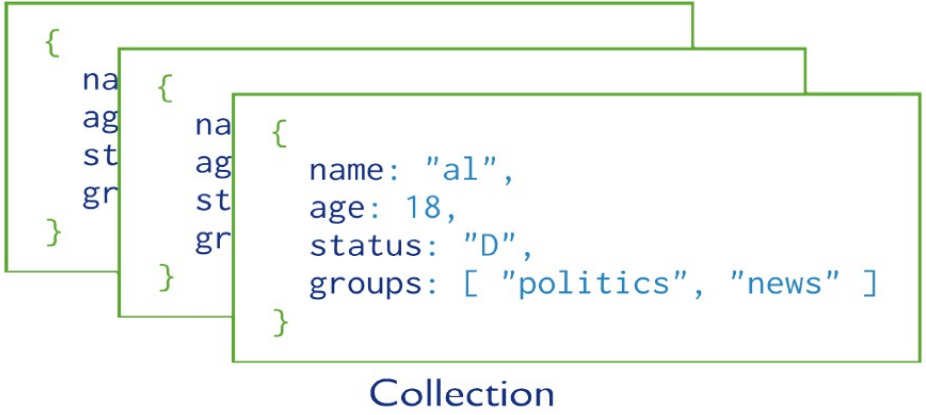
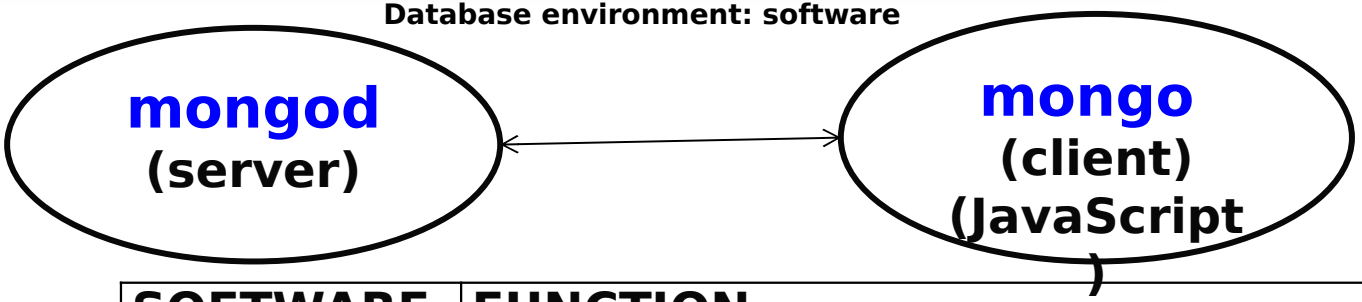


Figure 3.2: A collection of MongoDB documents.

MongoDB document database (5)

Database environment: software



SOFTWARE	FUNCTION
mongod	DBMS server
mongo	client
mongoimport	tool for importing JSON docs into DB
mongoexport	tool for exporting JSON docs from DB
mongos	tool for sharding a database (Sharding = storing data across multiple

MongoDB document database (6)

Operation	e.g. MongoDB statement
CREATE a database **OR** **Open an existing database**	**use mydb**
CREATE **a document** **and a collection**	**doc = { name : "value", x : 3 }** **mydb.createCollection(“myCollection”)**
Store data in database	**mydb.mycollection.insert(doc)** **mydb.myCollection.find ({ x:**

Examples of statements for database operations.

The statements are typed in

COS 326
Database Systems

Lecture 14

NoSQL Databases – MongoDB (2)

Chapter 24 (7th edition)

Thursday 05 September 2024

In this lecture

1. MongoDB document database
2. CRUD operations
3. MongoDB data types
4. Indexing

The following are available on ClickUP:

- Link to MongoDB manual (documentation)
- Link to MongoDB installer file

Recap

1. **What is Big Data?**
2. **Sources of Big Data**
3. **NoSQL databases & data models for big data**
4. **MongoDB NoSQL document database**

References (available on ClickUP):

1. **Moniruzzaman, A.B.M. & Hossain, S.A. (2013)**
NoSQL database: new era of databases for big data analytics – classification, characteristics and comparison. *International Journal of Database Theory and Application*, vol. 6, no. 4, 2013.

MongoDB document database (1)

A. Definition of MongoDB

1. **MongoDB is a:**
   a. **document-oriented database** which provides
      i. high performance
      ii. high availability
      iii. high scalability
   b. database which **works on the concepts of collections and documents**

2. **A MongoDB database:**
   is a physical container for collections

3. **MongoDB DBMS**
   a. **uses a** single server with multiple databases

MongoDB document database (2)

4. A MongoDB collection is a:
   a. group of MongoDB documents
   b. exists within a single database
   c. does not enforce a schema
   d. documents within a collection can have different fields

e. typically all documents in a collection are of similar or related purpose

Figure 3.2: A collection of MongoDB documents.

MongoDB document database (5)

Database environment: MongoDB Compass

- **MongoDB Compass is a powerful GUI for querying, aggregating, and analyzing your MongoDB data in a visual environment.**

- **Compass is free to use and source available, and can be run on macOS, Windows, and Linux.**

MongoDB document database (6)

Database environment: Demo session Time

Demonstration on how to launch MongoDB and perform some operations

MongoDB document database (7)

Database environment: Mongo Shell (mongosh)

Mongo shell

COMMAND LINE STATEMENT (JS helpers)	EXAMPLE
show	show dbs show collections
use (create a new database or specify existing database to use)	use mydb
db (show name of currently selected db)	db
help	

The screenshot shows the Mongosh screen when the example commands are typed

```
>_MONGOSH

> show dbs
< admin    40.00 KiB
  config  108.00 KiB
  local    40.00 KiB
  test1   112.00 KiB
> use test1
< 'switched to db test1'
> show collections
< mine
  newmine
test1 >
```

mongo shell helpers vs JavaScript

mongo shell helpers	JavaScript Equivalents
show dbs show databases	db.adminCommand('listDatabases')
use <db_name> e.g. use mydb	db = db.getSiblingDB('<db_name> ') e.g. db = db.getSiblingDB('mydb')
show collections	db.getCollectionNames( )

Data types (1)

- **String :** This is most commonly used data type to store the data. String in mongodb must be UTF-8 valid.
- **Integer :** used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean :** used to store a boolean (true/ false) value.
- **Double :** used to store floating point values.
- **Array :** used to store arrays or lists or multiple values with one key.
- **Timestamp :** ctimestamp. Commonly used to record

Data types (2)

- **Object :** used for embedded documents.
- **ObjectID :** used to store the document’s ID.
- **Null** : used to store a Null value.
- **Symbol :** used identically to a string however, it's generally reserved for languages that use a specific symbol type.
- **Date** : used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Binary data :** used to store binary data.
- **Code :** used to store **javascript code** into document.
- **Regular expression :** used to store regular expression

MongoDB CRUD operations

– CRUD = CREATE, READ, UPDATE, DELETE

Operation	Mongo statement in JavaScript	Equivalence for SQL DBs
CREATE a database	use mydb	CREATE DATABASE …
CREATE a data container	doc = {… doc content ….} mydb.mycollection.insert (doc)	CREATE TABLE … INSERT INTO …
READ	mydb.myCollection.find(..)	SELECT …. WHERE
UPDATE	mydb.myCollection.update(..)	UPDATE ….WHERE

CRUD operations: CREATE

STATEMENT	EXAMPLE
create a document	doc = { name : “value", x : 3 }
create a collection and insert documents into the collection	db.createCollection(“myCollection”) THEN db.myCollection.insertOne({ name : "value", x : 3 }) Can implictly create a collection without first using ‘createCollection’ db.myCollection.insertOne(doc) OR: db.myCollection.insertOne({name : “value", x : 3 })
drop() method (to drop a collection)	db.myCollection.drop( )

insert( ) method **note help with brackets***

METHOD	STATEMENT EXAMPLE
insert( ) into a specific collection. if collection doesn't exist it is created. can specify unique ID or MongoDB will assign a unique ID. can insert multiple docs using array notation)	db.myCollection.insertOne ({ name : "mongo", x: 3 }) db.myCollection.insertOne ({ _id: ObjectID(7d77dg), name : "mongo", x: 3 }) db.myCollection.insertMany ([{name : "mongo", x: 3}, {name : "Neo4j", x: 4} {next doc }])
save( ) if no _id specified same as insert. if _id specified then overwrite.	

QUERYING: Relational operators (1)

e.g. db.myCollection.find(<selection criterion>)

Operation	S (selection criterion)	
equality	{ <key>:<value> }	db.myCollection.find ({ x: 3 })
less than	{<key>: {$lt: <value>} }	db.myCollection.find ({ x: { $lt: 3 } })
less than or equal	{<key>: {$lte: <value>} }	db.myCollection.find ({ x: { $lte: 3 } })

QUERYING: Relational operators (2)

e.g. db.myCollection.find(<selection criterion>)

Opera-tion	Syntax (selection criterion)	Example
greater than	{<key>: {$gt: <value>} }	db.myCollection.find ({ x: { $gt: 3 } })
greater than or equal	{<key>: {$gte: <value>} }	db.myCollection.find ({ x: { $gte: 3 } })
not equal	{<key>: {$ne: <value>} }	db.myCollection.find ({ x: { $ne: 3 } })

QUERYING: Logical operators AND, OR

Operation	Syntax	Example
AND	$and: { <key1>:<value1>, <key2>:<value2>, ... <keyn>:<valuen> }	db.myCollection.find({ $and: { name: "mongo", x: 3 }] })
OR	$or: [{ <key1>:<value1> }, { <key2>:<value2> }, ... { <keyn>:<valuen> }]	db.myCollection.find({ $or: [{ name:"mongo" }, { name: "Neo4j" }] })
Alter-native OR	{ <key>: { $in : [val1, val2, ..] } }	db.myCollection.find({ name: { $in : ["mongo", "Neo4j"] } })

QUERYING (2a): find( )

List database contents: db.myCollection.find( )

```
mongosh mongodb://127.0.0.
db> db.myCollection.insertOne({name: "mongo", x:3})
{
  acknowledged: true,
  insertedId: ObjectId("64f77f99cf3b7de1d4d48756")
}
db> db.myCollection.insertOne({name: "Neo4j", x:4})
{
  acknowledged: true,
  insertedId: ObjectId("64f77fa7cf3b7de1d4d48757")
}
db> db.myCollection.find()
[
  { _id: ObjectId("64f77f32cf3b7de1d4d48755"), name: 'value', x: 3 },
  { _id: ObjectId("64f77f99cf3b7de1d4d48756"), name: 'mongo', x: 3 },
  { _id: ObjectId("64f77fa7cf3b7de1d4d48757"), name: 'Neo4j', x: 4 }
]
db>
```

1. **_id values are automatically generated by mongod**
2. **use it helper if more results need to be shown for find()**
3. **findOne(...) returns only one document**

QUERYING find: projection

syntax	Notes
find ({ <criterion> }, { <field1>: <selector>, <field2>: <selector>, ... <fieldn>: <selector> })	**selector = 1** means include field **selector = 0** means don't include field

Example:

```
db.myCollection.find
    (   { x:  { $gt: 2 } },    {  name: 1,
id:  0} )
```

QUERYING find: sort

syntax	Notes
<find query>.sort ({ <field1>:<order>, <field2>:<order>, ... <fieldn>:<order> })	**order = 1** means ascending order **order = -1** means descending order

Example:

```
db.myCollection.find( { x:  { $gt: 2 }  } ).sort( {  name: 1} )
```

Updating

syntax	Notes
update ({ <field>: { <criterion> } }, { $set: { <action> } } { multi: <selector>})	selector = true means more than one document can be updated

Example:

```
db.myCollection.update(   { x: 3 } },
                          { $set:
                                  {name: "mongodb" },
                                  {multi: true})
```

Deleting

syntax	Notes
deleteMany({ <field>: { <criterion> })	e.g. statement 1 (delete all) To delete one document use deleteOne statement 2 (delete based on criterion)

statement 1:
db.myCollection.deleteMany({ })

statement 2:
db.myCollection.findOneAndDelete({ x: 3 })

QUERYING find: Indexing

- **Indexes created for collections, especially large collections (Big Data)**
  - **Speeds up find, update and remove queries**
  - **Speeds up sort queries**

e.g. index on a field (1 = ascending, -1 = descending):

```
db.myCollection.createIndex( { name: 1 } )   OR
db.myCollection.ensureIndex( { "name": 1 } )

db.myCollection.createIndex( { name: -1 } )   OR
db.myCollection.ensureIndex( { "name": -1 } )
```

e.g. composite index:

```
db.myCollection.createIndex( { name: 1, x: -1 } )
```

COS 326
Database Systems

Lecture 15

NoSQL Databases – MongoDB (3)

Chapter 24 (7th edition)

Tuesday 10 September 2024

In this lecture

MongoDB

1. Semester Test details
2. Cursors and iteration
3. Embedded documents
4. Operations on arrays
5. Aggregates
6. Database design

Reference: MongoDB manual (documentation)

Recap

1. MongoDB document database
2. CRUD operations
3. MongoDB data types
4. Indexing

Semester Test Information

- ClickUP Closed book test
- **Date:** Monday 16 September 2024
- **Time:** 17:30 Hrs to 19:00 Hrs
- **Venue:** Informatorium (Blue Labs 1 to 3) & SIT 1 Lab.
- Bring your pen and student card only.
- No extra time will be provided unless there is loadshedding interruption 🏝
- You will have no access to lecture slides, practicals, database software etc

Semester Test Structure

- ClickUP Closed Book Test
- 1 Hour 30 minutes long
- Total of **100 Marks**
- Consist of 3 sections, namely:
  - ❖ Multiple Choice/Fill in the blank questions – **20 Marks**
  - ❖ Theory and Application Based Questions – **40 Marks**
  - ❖ Scenario Based Questions – **40 Marks**

➢ **PLEASE NOTE:** There will be questions where you

Cursors & iteration

Cursors: A cursor is a pointer to current db 'record'

e.g: iterate over the cursor using a Loop

Action	JavaScript statement
1. query testCollection & assign resulting cursor object to var c:	var c = db.testCollection.find()
2. Print the full result set using a while loop	while (c.hasNext()) printjson(c.next()) OR: print(tojson(c.next()))
3. Manipulate a cursor object as an array, e.g. print doc at array index 4	printjson(c [4])

hasNext() method returns true if the cursor has documents.

next( ) method returns the next document.

printjson( ) method renders the document in a JSON like

Example: Embedded documents

An embedded (or nested) document is a document inside another document. Consider the following examples:

```
doc =  {  username :  "gooduser",
          contact :       {    phone : "012 420 1234",
                               email : "champion@mybusiness.co.za"
                            },
          access :        {level : 5,  group : "dev"}
       }
db.users.insertOne(doc)

doc =  {  username :    "serioususer",
          contact :       {    phone : "012 420 5678",
                               email : "worker@mybusiness.co.za"
                            },
          access :        { level : 3,  group : "dev" }
       }
```

Querying the collection

e.g. **use db**
db.users.find()

mongosh mongodb://127.0.0.

```
db> db.users.find()
[
  {
    _id: ObjectId("6511b985c3fd370917c4e1fa"),
    username: 'gooduser',
    contact: { phone: '012 420 1234', email: 'champion@mybusiness.co.za' },
    access: { level: 5, group: 'dev' }
  },
  {
    _id: ObjectId("6511ba0ac3fd370917c4e1fb"),
    username: 'serioususer',
    contact: { phone: '012 420 5678', email: 'worker@mybusiness.co.za' },
    access: { level: 3, group: 'dev' }
  }
]
db>
```

Querying using iteration & printjson

e.g. var c =
db.users.find()
while (c.hasNext())
printjson(c.next())

Use this querying style when documents are complex.

```
mongosh mongodb://127.0.0.

db> var c =  db.users.find()

db> while (c.hasNext()) printjson(c.next())
{
  _id: ObjectId("6511b985c3fd370917c4e1fa"),
  username: 'gooduser',
  contact: {
    phone: '012 420 1234',
    email: 'champion@mybusiness.co.za'
  },
  access: {
    level: 5,
    group: 'dev'
  }
}
{
  _id: ObjectId("6511ba0ac3fd370917c4e1fb"),
  username: 'serioususer',
  contact: {
    phone: '012 420 5678',
    email: 'worker@mybusiness.co.za'
  },
  access: {
    level: 3,
    group: 'dev'
  }
}

db>
```

Querying Embedded documents (1)

Match on the field values of an embedded document

```
var c =  db.users.find( { access: { level:5, group: "dev" } } )
while ( c.hasNext() )   printjson( c.next()  )
```

mongosh mongodb://127.0.0.

```
db> var c =  db.users.find( { access: { level:5, group: "dev" } } )

db> while ( c.hasNext() )   printjson( c.next()  )
{
  _id: ObjectId("6511b985c3fd370917c4e1fa"),
  username: 'gooduser',
  contact: {
    phone: '012 420 1234',
    email: 'champion@mybusiness.co.za'
  },
  access: {
    level: 5,
    group: 'dev'
  }
}

db>
```

Querying Embedded documents (2)

match on value of a specific field of an embedded document.

use dot notation

var c = db.users.find ({ "access.level": 5 })
while (c.hasNext( )) printjson(c.next ( ))

```
mongosh mongodb://127.0.0.

db> var c =  db.users.find ( { "access.level":   5  }  )

db> while (c.hasNext( )  )   printjson(c.next ( ) )
{
  _id: ObjectId("6511b985c3fd370917c4e1fa"),
  username: 'gooduser',
  contact: {
    phone: '012 420 1234',
    email: 'champion@mybusiness.co.za'
  },
  access: {
    level: 5,
    group: 'dev'
  }
}

db>
```

Array operations (1)

- **If a field holds an array, you can query for:**
  (1) an exact array match
  OR
  (2) specific values in the array.

- **If the array holds embedded documents, you can query for:**
  - **specific fields in the embedded documents using dot notation.**

- **Consider an inventory collection that contains documents of the form:**

  {"_id": 5, "type": "food", "item": "cake", "ratings": [5, 8, 9]}

Array operations (2)

Exact Match on an Array

the array field should match exactly the specified values & element order.

e.g. List all documents where the field 'ratings' holds exactly three elements, 5, 8, 9, in this order:

```
db.inventory.find({ ratings: [ 5, 8, 9 ] })
```

Match an Array Element

- **specify a single element in the array to match.**

e.g. List all documents where 'ratings' contains 5 as one of its elements:

Array operations (3a)

Specify Multiple Criteria for Array Elements (1)

e.g. (1)

List documents where the ratings array contains elements that satisfy the following query conditions:

1) one element can satisfy the greater than 5 condition

2) another element can satisfy the less than 9 condition,

3) or a single element can satisfy both:

```
db.inventory.find( { ratings:    { $gt: 5,   $lt: 9 } }  )
```

Array operations (3b)

Specify Multiple Criteria for Array Elements (2)

e.g. (2)

List documents where the ratings array contains at least one element that is greater than 5 and less than 9:

```
db.inventory.find ({ ratings: { $elemMatch: { $gt: 5, $lt: 9 } } } )
```

Array operations (4)

Array of Embedded Documents (1)

- **Suppose the inventory collection contains documents in the following format:**

```
{ _id: 100,    type: "food",    item: "xyz",   qty: 25, price: 2.5,
               ratings: [ 5, 8, 9 ],
               memos: [{ memo: "on time",   by: "shipping" },
                         { memo: "approved",    by: "billing" }
                       ] }
```

e.g. (1) select all documents where the memos field contains an array that contains an embedded document that contains the field '*by*' with the value 'shipping':

```
db.inventory.find( { 'memos.by': 'shipping' } )
```

Array operations (5)

Array of Embedded Documents (2)

- Suppose the inventory collection contains documents in the following format:

```
{  _id: 100,     type: "food",     item: "xyz",   qty: 25, price: 2.5,
                 ratings: [ 5, 8, 9 ],
                 memos: [     { memo: "on time",   by: "shipping" },
                              { memo: "approved",    by: "billing" }
                         ]    }
```

e.g. (2)

selects all documents where the memos field contains an array whose first element (at index 0) is a document that contains the field '*by*' whose value is 'shipping':

```
db.inventory.find( { 'memos.0.by': 'shipping' } )
```

Example collection for aggregates

e.g. data for blogs

Listing of the blogs collection without the _id

```
> db.blogs.find ( { },  {_id: 0, title: 1, likes: 1} )
[
{ "title" : "NoSQL",      "likes" : 400 }
{ "title" : "MongoDB",   "likes" : 100 }
{ "title" : "MongoDB",   "likes" : 150 }
{ "title" : "MongoDB",   "likes" : 50 }
{ "title" : "Neo4j",      "likes" : 50 }
{ "title" : "Neo4j",      "likes" : 150 }
{ "title" : "Neo4j",      "likes" : 250 }
]
```

Aggregation (1)

aggregate( ) method

syntax:

db.COLLECTION_NAME.aggregate(AGGR_OPERATION)

FUNCTION	PURPOSE
$sum	compute the sum
$avg	compute the average
$min	find the minimum value
$max	find the maximum value

Aggregation (3): pipeline (a)

- An aggregation pipeline consists of one or more stages that process documents:
- Each stage performs an operation on the input documents. For example, a stage can filter documents, group documents, and calculate values.
- The documents that are output from a stage are passed to the next stage.
- An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum, and minimum values.

Aggregation (3): pipeline (b)

db.COLLECTION_NAME.aggregate(AGGR_OPERATION)

- Aggregation pipeline syntax:

```
  aggregate(  [
 { $match: <document criterion> },
 { $group:  <group spec> },
 { $group:  <group spec> },
 ..................................................
 { $match: <group criterion> }
 { $sort: <sort spec> }
   ]
            )
```

Aggregation (2): example - sum

Query: compute the total number of 'likes' for each title

FUNCTION	EXAMPLE
$sum	db.blogs.aggregate([{ $group: { _id: "$title", num_likes: { $sum: "$likes" } } }])

```
db> db.blogs.aggregate( [   {   $group:{_id: "$title",num_likes:    { $sum: "$likes"  }}}] )
[
  { _id: 'NoSQL', num_likes: 400 },
  { _id: 'MongoDB', num_likes: 300 },
  { _id: 'Neo4j', num_likes: 450 }
]
```

Aggregation (4): pipeline example

Query:

compute the total number of ‘likes’ for the title ‘Neo4j’

```
db.blogs.aggregate(
          {  $match: {  title:  "Neo4j"}  },
   {  $group:
    {  _id:   "$title",
             num_likes: { $sum: "$likes"} }
     }
        )
```

```
db> db.blogs.aggregate([{   $match: {  title:  "Neo4j"}  },{   $group:{  _id:   "$title",num_likes: { $sum : "$likes"} }}])
[ { _id: 'Neo4j', num_likes: 450 } ]
db>
```

Aggregation (5): pipeline example

Query: **compute the total number of 'likes' for the documents & show results in ascending order of number of likes.**

```
db.blogs.aggregate(
      { $group:
     {   _id: "$title",
    num_likes:   {  $sum: "$likes"} } },
      { $sort: {"num_likes": 1} }
  ]  )
```

```
db>  db.blogs.aggregate([{ $group:{_id:"$title",num_likes:{$sum: "$likes"} } },{ $sort: {"num_likes": 1} }])
[
  { _id: 'MongoDB', num_likes: 300 },
  { _id: 'NoSQL', num_likes: 400 },
  { _id: 'Neo4j', num_likes: 450 }
]
```

Aggregation (6): pipeline example

Query: **compute total number of 'likes' for the documents & show results in ascending order of number of likes, and only those with a total > 300**

```
db.blogs.aggregate(
   [        { $match:        { } },
   { $group:        { _id: "$title",
                          num_likes:  { $sum: "$likes" } } },
   { $match: {num_likes: { $gt: 300} } },
        { $sort: {"num_likes": 1} }
   ] )
```

***** the second $match is equivalent to the SQL "HAVING"**

```
db> db.blogs.aggregate([{ $match:{ } },{ $group:{ _id: "$title",num_likes:  { $sum: "$likes" } } },{ $match: {num_likes: { $gt: 300}
} },{ $sort: {"num_likes": 1} } ] )
[ { _id: 'NoSQL', num_likes: 400 }, { _id: 'Neo4j', num_likes: 450 } ]
db>
```

Data Model Design (Refer to MongoDB manual)

- **The key consideration for the structure of a MongoDB database is the decision :**
  - **to use embedded documents OR**
  - **to use references.**

(1) Embedded Data Models (NO JOINS NEEDED)

- **With MongoDB, you may embed related data in a single structure or document.**
- **These schema are generally known as “denormalized” models, and take advantage of MongoDB’s rich documents.**

(2) Normalised Data Models (JOINS NEEDED)

- **Normalised data models implement relationships using references between documents**

Embedded Data Models

Computer system user

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

Embedded sub-document

Figure 4.3: Data model with embedded fields that contain all related information.

Use embedded data models

- **you have "contains" or "has" relationships between entities.**
- **you have one-to-many relationships between entities, i.e. "many" or child documents always appear with or are viewed in the context of the "one" or parent document.**

Normalized Data Models

Computer system user

user document

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

contact document

```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

access document

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

use normalized data models:

- **when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.**
- **to represent more complex many-to-many relationships.**
- **to model large hierarchical data sets.**

COS 326
Database Systems

Lecture 16

NoSQL Databases – MongoDB (4)

Chapter 24 (7th edition)

Thursday 12 September 2023

In this lecture

MongoDB

1. **Searching for Text**
2. **Mongo shell functions**
3. **JDBC & Java - MongoDB applications**

Reference: MongoDB manual (documentation)

Searching for Text (1)

Consider the following 'blogs' collection:

```
{   title:          "NoSQL",
    comments:  "NoSQL databases were invented for Big Data",
    likes: 400  }

{  title:          "MongoDB",
   comments: "MongoDB is a NoSQL database",
    likes: 100   }

{   title:          "MongoDB",
    comments:  "MongoDB is a very easy database to use",
    likes: 150   }
```

Searching for Text (2)

```
{   title:  "MongoDB",
    comments:    "MongoDB uses the JavaScript language",
    likes:    50  }

{   title:    "Neo4j",
   comments:     "Neo4j is a NoSQL database",
   likes: 50  }

{  title:    "Neo4j",
   comments:      "Neo4j can store social networks data",
   likes:    150  }

{   title:     "Neo4j",
   comments:      "Neo4j is a client-server database",
   likes:    250   }
```

Searching for Text (3)

Text Indexes: Needed for text search of string content in documents of a collection.

***text* indexes can include any field whose value is a string (e.g. the comments field) or an array of string elements.**

- **To create a text index (e.g. for the blogs collection),**
  - **use the *createIndex()* method, e.g.**

    ***db.blogs.createIndex*({comments: "text"})**
- **A collection can have at most one text index.**
- **To remove an index from a collection use the *dropIndex()* method**

Searching for Text (4)

Search for documents which have the word "big" in the comments field:

var c = *db.blogs.find({ $text: { $search: "big"} })*
while (c.hasNext()) printjson(c.next())

Note: **text search is not case-sensitive**

```
db> db.blogs.createIndex({comments: "text"} )
comments_text
db> var c = db.blogs.find( { $text:  { $search: "big"} } )

db> while (c.hasNext()  )  printjson(c.next() )
{
  _id: ObjectId("6511c4e158d33a1abbda5ebb"),
  title: 'NoSQL',
  likes: 400,
  comments: 'NoSQL databases were invented for Big Data'
}
```

Searching for Text (5)

Search for documents which have the word “big” or “data” in the comments field:

```
var c = db.blogs.find( { $text:  { $search: "big data"} } )

while (c.
```

```
mongosh mongodb://127.0.0.

db> var c = db.blogs.find({ $text: { $search: "big data" } })

db> while (c.hasNext()  )  printjson(c.next() )
{
  _id: ObjectId("6511c4e158d33a1abbda5ebb"),
  title: 'NoSQL',
  likes: 400,
  comments: 'NoSQL databases were invented for Big Data'
}
{
  _id: ObjectId("6511c4e158d33a1abbda5ec0"),
  title: 'Neo4j',
  likes: 150,
  comments: 'Neo4j can store social networks data'
}
```

Searching for Text (6)

Wildcard Text Indexes

- **To allow for text search on all fields with string content,**
  - use the **wildcard specifier ($**) to index all fields in the collection that contain string content.**
  - **useful for highly unstructured data if it is unclear which fields to include in the text index** or for ad-hoc querying.
  - **MongoDB will index every field that contains string data** for each document in the collection.
- **Example:** creates a text index using the wildcard specifier:

db.collection.createIndex({ "$**": "text" })

Searching for Text (7)

Text Search in the Aggregation Pipeline

- text search is performed by **using:**
  - the ***$text* query operator in the *$match* stage.**

Restrictions

- **The *$match* stage that includes a *$text* must be the first stage in the pipeline.**
- **A text operator can only occur once in the stage.**
- The text operator expression cannot appear in ***$or*** or ***$not*** expressions.

Searching for Text (8)

Text Search in the Aggregation Pipeline

- **e.g**. Aggregation query **for the blogs collection**:
  - **searches for the term *"NoSQL"*** in the ***$match* stage** &
  - **calculates the total *"likes"*** for the matching documents in the ***$group* stage.**

```
db.blogs.aggregate(
   [    { $match:  { $text:  { $search: "NoSQL" }}},
        { $group:  {     _id:    null,
                         totalLikes: { $sum: "$likes"}}}     ] )
```

- **Query result:**

  { "_id" : null, "totalLikes" : 550 }

Mongo shell (client) functions (1)

1. **e.g. A JavaScript function which creates new data**

```
function insertData(dbName, colName, num)
{    var col = db.getSiblingDB(dbName).getCollection(colName);
     for (i = 0; i < num; i++)
     {   col.insert({ x: i} );   }
     print(col.count());
}
```

2. Can store the function in file **.mongoshrc.js** for future use. Store this file in : **C:\Users\UserName**

(contents are loaded when the shell starts)

3. **e.g. of function call:** **insertData("mydb", "testCollection", 20)**

Mongo shell (client) functions (2a)

Recall from Lecture 15:

Query: compute the total number of **'likes'** for the title 'Neo4j'

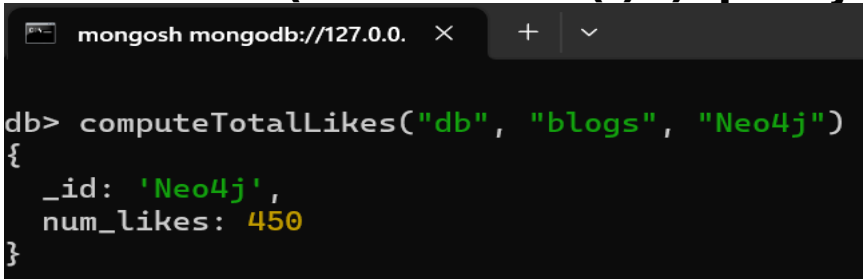
```
db.blogs.aggregate(
    [
        {   $match: {title:"Neo4j"}  },
        {   $group:
       {   _id: "$title",
          num_likes:
                    { $sum : "$likes"}
              }
        }
    ]   )
```

*****Not nice to have to type this query whenever**
*****the aggregate is needed**

Mongo shell (client) functions (2b)

*********Function definition in the file *.mongoshrc.js*********

```
function computeTotalLikes(dbName, colName, blogTitle)
{   var col = db.getSiblingDB(dbName).getCollection(colName);

    var res = col.aggregate( [ { $match: {title: blogTitle } },
        { $group: { _id: blogTitle,
         num_likes: { $sum : "$likes"} } } ] );

         while (res.hasNext( ) ) printjson(res.next() );
}
```



Function call:
computeTotalLikes("db", "blogs", "Neo4j")

Creating & testing a Java-MongoDB application (1)

- **JDBC = Java Database Connectivity**
- **provides API for Java applications to access a database**

- **JDBC library for MongoDB**
- **mongodb-jdbc-2.2.0-all.jar**

- **Refer to the demo video for instructions on how to add the jar file**

Creating & testing a Java-MongoDB application (2)

Demo Time

Follow on the following demo inorder to understand how to create and test a Java application that uses a MongoDB database

Creating & testing a Java-MongoDB application (3)

To learn more about the MongoDB Driver classes and methods, refer to this link:
https://mongodb.github.io/mongo-java-driver/2.13/getting-started/quick-tour/

COS 326
Database Systems

Lecture 17

NoSQL Databases – Neo4j (1)

Chapter 24 (7th edition)

Tuesday 01 October 2024

Class Attendance Register

Scan me with your

In this lecture

1. Graph theory

2. Graph data structure

3. Graph database concepts

4. Neo4j Desktop v1.6.0

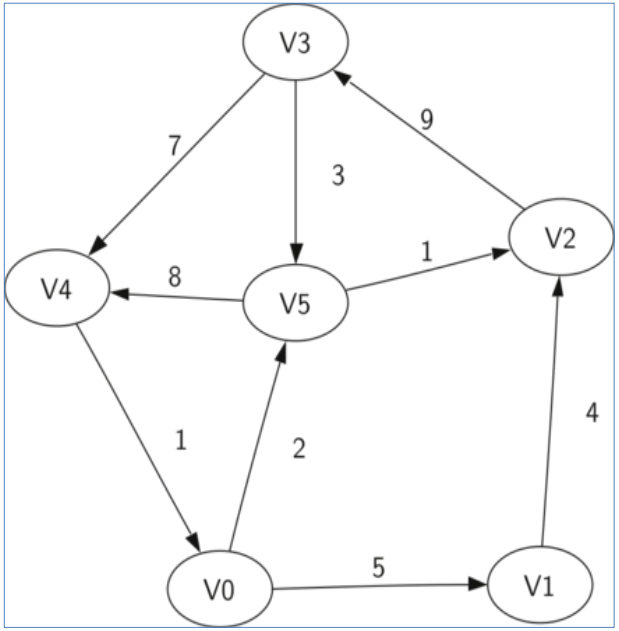
Reference: Neo4j manual. Can be accessed from here: https://neo4j.com/docs/getting-started/current/

Graph theory: Mathematical perspective (1a)

Graph theory:

- **A branch of discrete mathematics.**
- **Distinguished by its geometric approach to the study of various objects.**
- **The principal object of the theory is**
  - **a graph and its generalisations.**

A Simple example of a Directed Graph



Graph theory: Mathematical perspective (1b)

A graph:

- **may be denoted as *G* where**

  $$G = (V, E).$$

- ***V* is a set of vertices, *E* is a set of edges.**

- **Each edge is a tuple (v_i, v_j)**

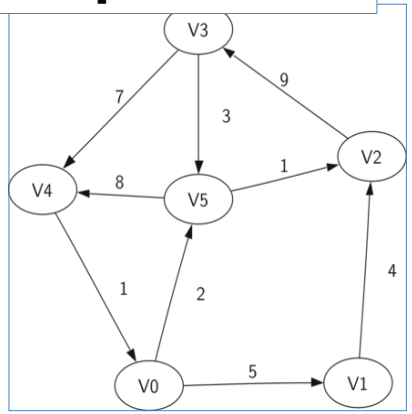
  where $v_i, v_j \in V$.

- **a third component w can be added to the edge tuple to represent a weight,**

  i.e. (v_i, v_j, w)

- **A subgraph *s* is a set of edges *e* and vertices *v* such that $e \subset E$ and $v \subset V$.**

A Simple example of a Directed Graph



Graph theory: Mathematical perspective (2a)

A vertex (also called a "node")

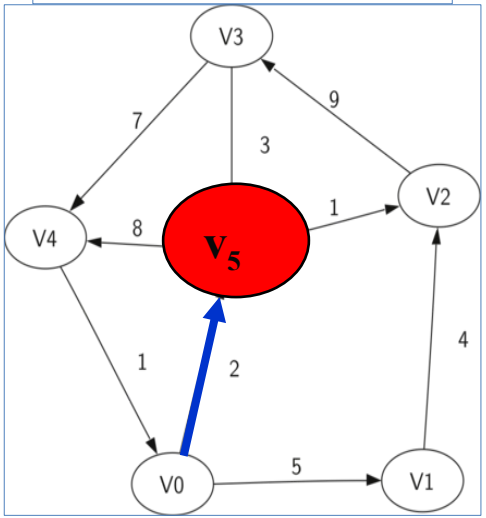
- Can have a name, aka the "key."
- May also have additional information.

An edge (also called an "arc")

- Connects two vertices to show that there is a relationship between them.

- Edges may be one-way or two-way.
  - If the edges in a graph are all one-way, the graph is called a directed graph, or a digraph.

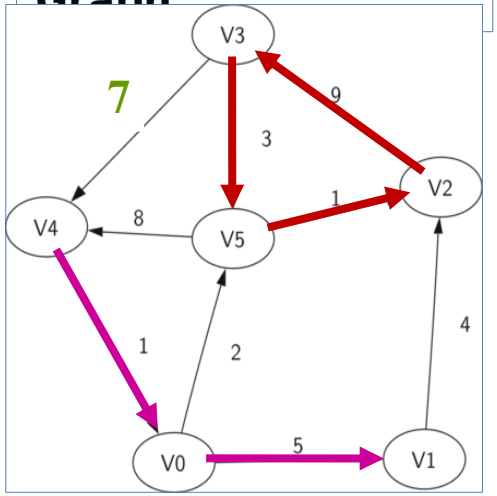
A Simple Example of a Directed Graph



Graph theory: Mathematical perspective (2b)

- **Weight:**
  - **Edges may be weighted to show that there is a cost to go from one vertex to another.**
  - **e.g in a graph of roads that connect one city to another, the weight on the edge might represent the distance between two cities.**
- **A path in a graph**
  - **is a sequence of vertices that are connected by edges.**
- **A cycle in a directed graph**
  - **is a path that starts and ends at the same vertex.**

A Simple Example of a Directed Graph



Graph data structure: computational perspective(1)

The Graph Abstract Data Type - (part of the CS curriculum)

The graph abstract data type (ADT) supports the following operations:

- **Graph()** **creates a new, empty graph.**
- **addVertex(vert)** **adds an instance of Vertex to the graph.**
- **addEdge(fromVert, toVert)** **Adds a new, directed edge to the graph that connects two vertices.**

Graph data structure: computational perspective(2)

- **addEdge(fromVert, toVert, weight)** Adds a new, weighted, directed edge to the graph that connects two vertices.
- **getVertex(vertKey)** finds the vertex in the graph named vertKey.
- **getVertices()** returns the list of all vertices in the graph.
- **in** returns True for a statement of the form **vertex in graph**, if the given vertex is in the graph, False otherwise.

Graph database concepts

Recall: Classification of NoSQL databases

(1) Key-value stores (2) document databases
(3) wide-column stores (4) Graph databases

A graph database can store data using a few simple concepts:

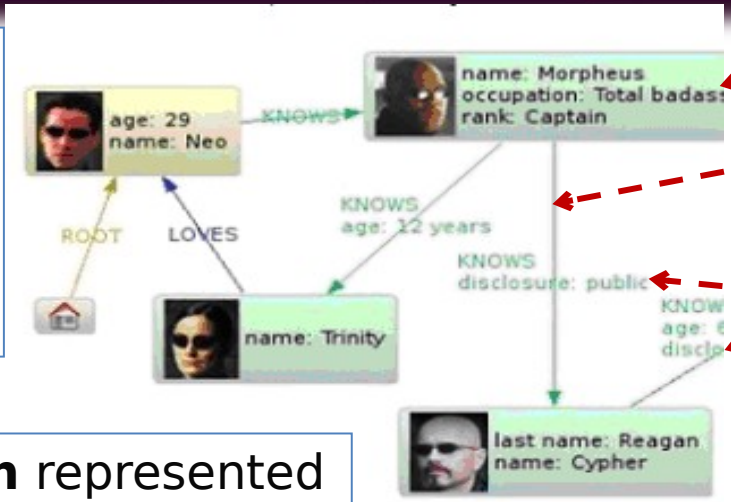
- **Nodes** - graph data objects (records)
- **Relationships** – edges which connect nodes (objects)
- **Properties** - named data values (attributes expressed as *key-value* pairs)

Graph databases

- use structured **relationship graphs of** interconnected **key-value pairings**

Graph databases

Example:



node (object)

edge
(node relationship)

properties
(object attributes expressed as *key-value* pairs)

Graph represented as an **Object-oriented network** of:

- **nodes**,
- **edges**
- **properties**

- **Primary uses:**
  - representing **social networks**,
  - **generating recommendations**,
  - conducting forensic investigations
- **Example DBs: Neo4j, InfoGrid, AllegroGraph**

Neo4j graph database

- Neo4j is a graph database management system developed by Neo4j, Inc.
- Version 1.0 was released in February 2010
- Neo4j is available in a non-open-source "community edition" licensed with a modification of the GNU General Public License, with online backup and high availability extensions licensed under a closed-source commercial license.
- Neo4j is implemented in Java and accessible from software written in other languages using the Cypher query language through a transactional HTTP endpoint, or through the binary "Bolt" protocol.
- The "4j" in Neo4j is a reference to its being built in

Neo4j graph database: Nodes (1)

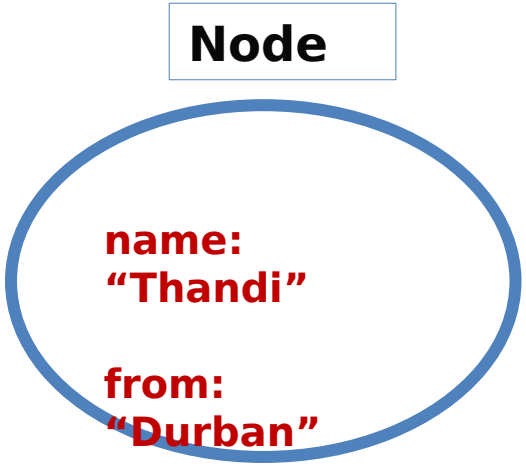
- **The simplest graph**
  - **has just a single node.**
  - **with some named values called Properties.**

- **In general:**
  - **Node is the name for a data record in a graph, i.e. data records are called nodes.**

  - **Data is stored as Properties**
  - **Properties are simple**
    - **e.g.**

    name: "Thandi"

    from: "Durban"



Neo4j graph database: Nodes (2)

- Labels are used to associate (group) a set of nodes.
- Nodes can be grouped together by applying a label to each member
  - E.g. nodes for people can all be given the label 'Person'
- A node can have zero or more labels
  - Labels do not have any properties
- Neo4j is schema-optional
  - nodes can have a mix of common and

This node can be assigned the label Person

name: "Thandi"

from: "Durban"

Neo4j : Relationships

Relationships: Connect nodes in the graph

- **The real power of a graph database**
  - » **e.g. Neo4j is in connected data.**
- **Relationships describe how the records are related.**
  - » **e.g. the FOLLOWS relationship**
  - » **Thandi FOLLOWS Neo and Johan**
  - » **Johan FOLLOWS Thandi**
- **Relationships always have direction & a type & form patterns of data**

Neo4j : Relationship properties

- **Relationship properties**
  - **Store information shared by two nodes.**
  - **In a property graph, relationships are data records (objects) that can also contain properties. e.g**
    - **Thandi FOLLOWS Neo since 2009**
    - **Thandi FOLLOWS Johan since 2012**
    - **Johan FOLLOWS Thandi since 2010**

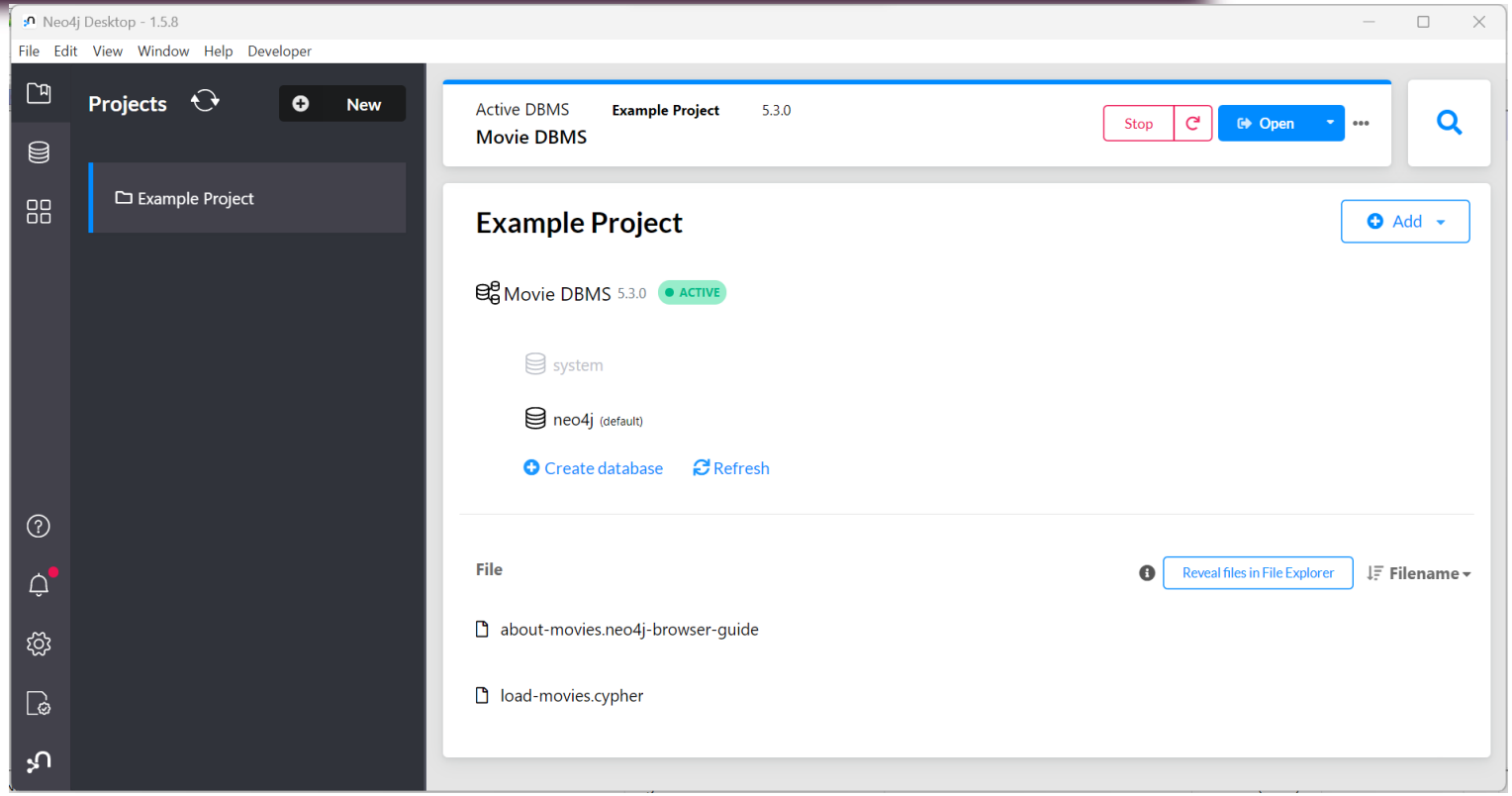
Neo4j installation

- **The version to be used in practicals:**
  - **is Neo4j Desktop version 1.6.0**
  - **The installers for Neo4j are available from** [Neo4j Desktop Download | Free Graph Database Download]

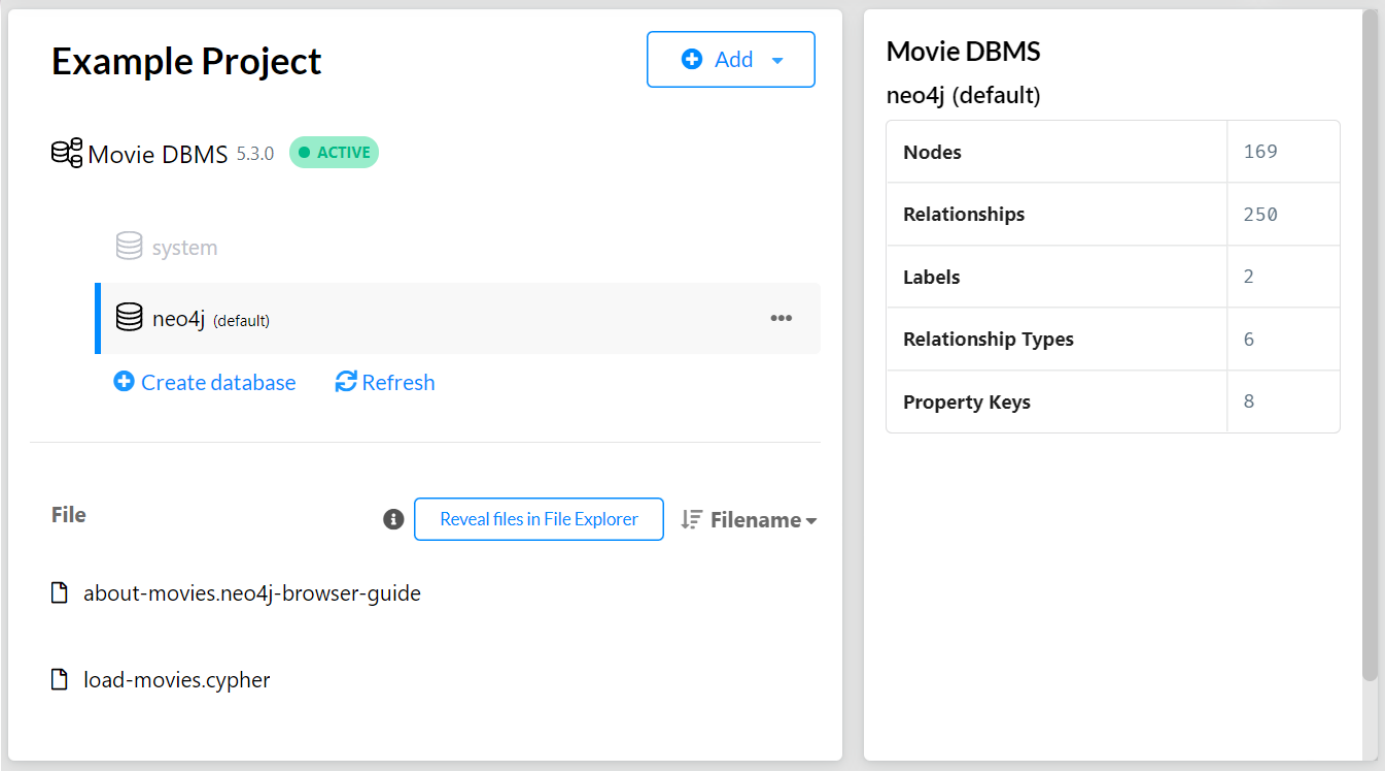
MS Windows:

installer file: ***Neo4j Desktop Setup 1.6.0.exe***

Neo4j Desktop interface (Windows 11)



Neo4j Desktop interface (Windows 11)



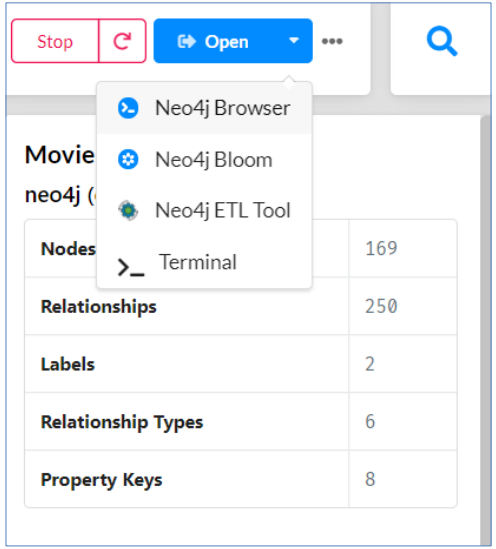
Neo4j Desktop interface (Windows 11)

- Neo4j Browser is an interactive cypher command shell that allows you to interact with your graph and visualize the information in it.

- Neo4j Bloom is a data exploration tool that visualizes data in the graph and allows users to navigate and query the data without any query language or programming.

- Neo4J ETL Tool is used to import relational databases into Neo4j

- The Neo4j CLI tool is a tool for managing a



Neo4j Browser

- Neo4j Browser is a command-driven client as a web-based shell environment. It is perfect for running ad-hoc graph queries, with just enough ability to prototype a Neo4j-based application.

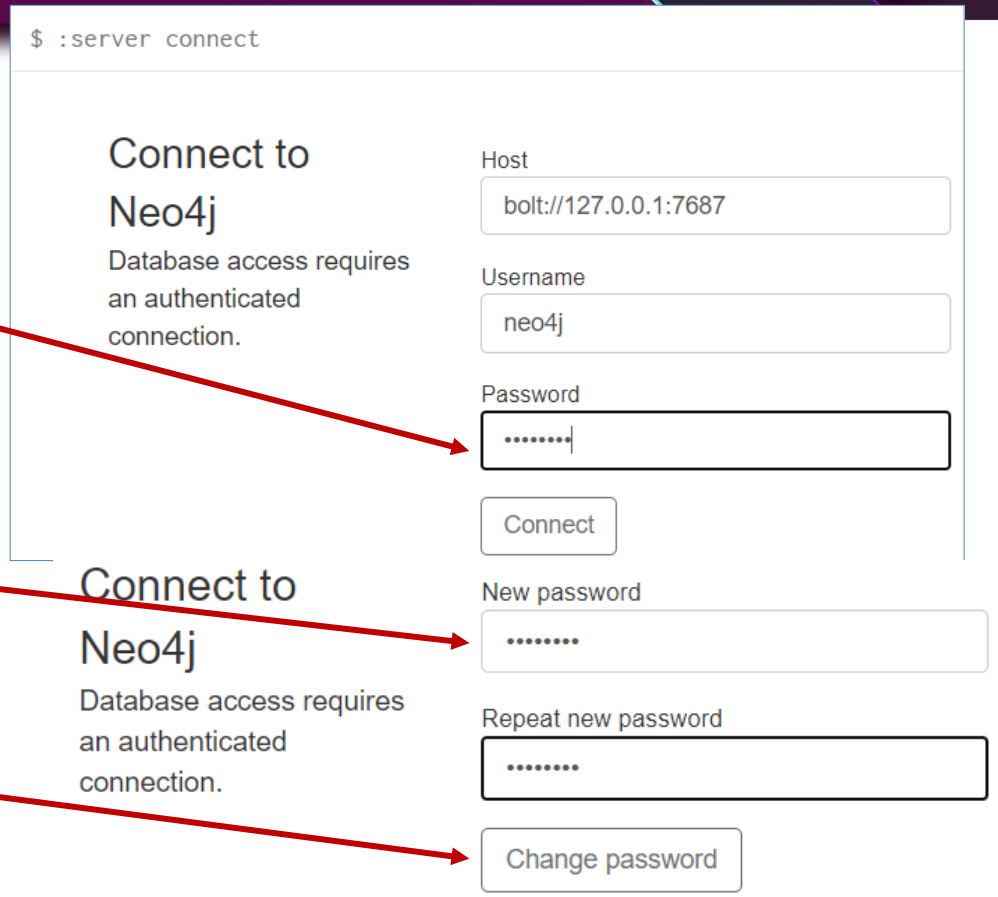
- Developer focused.
- Used for writing and running Cypher graph queries.
- Exportable tabular results of any query result.
- Graph visualization of query results containing nodes and relationships.
- Convenient exploration of the Neo4j's HTTP API (REST).

Neo4j Browser Login : http://localhost:7474

(1)
Type neo4j for the password

(2)
Then in the next dialog type a new password of your choice

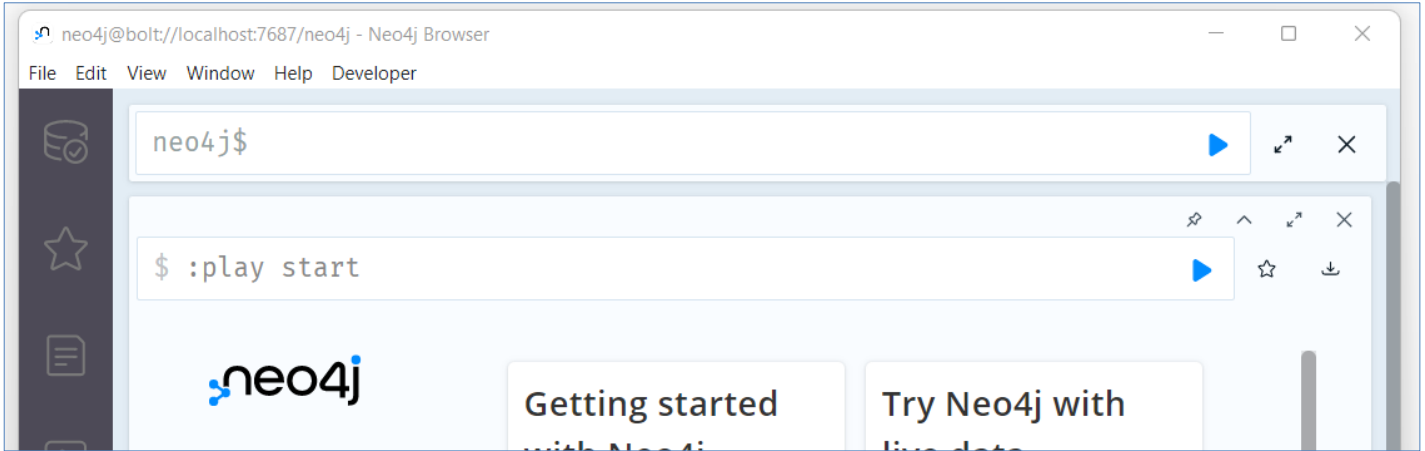
(3)
Click on this button



Neo4j Browser Editor

for query editing and execution

- **The editor is the primary interface for entering and running commands.**
- **Enter Cypher queries to work with graph data.**
- **Click on the execute button to run a query.**



Neo4j Cypher query language

Cypher: Neo4j's graph query language

- **purpose built for working with graph data.**
- **uses patterns to describe graph data**
- **uses familiar SQL-like clauses**
- **is declarative (like SQL) i.e. describes what to do, not how to do it**

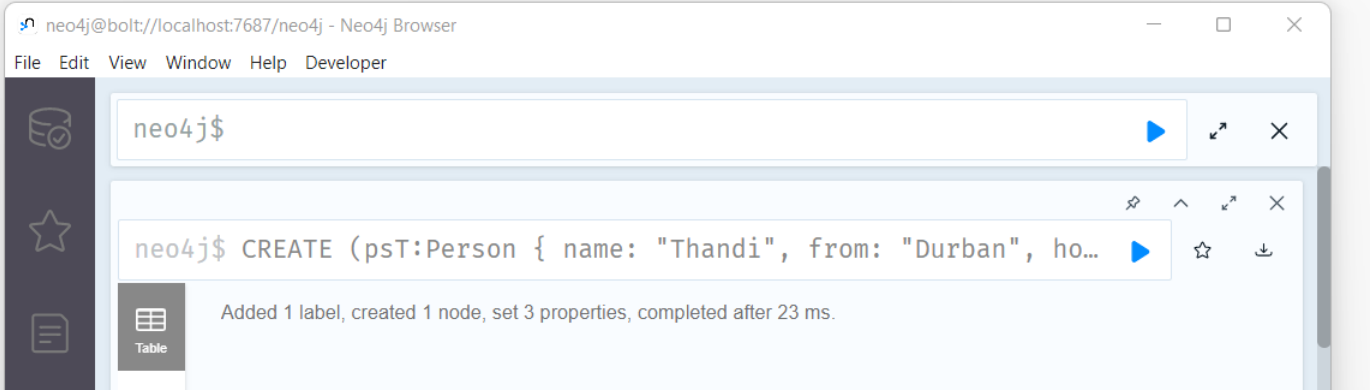
Neo4j: CREATE statement

Create a node:

CREATE (psT:Person { name: "Thandi", from: "Durban", hobby: "singing"})

CREATE clause is used to create data

- **( ) parenthesis to indicate a node**
- **psT:Person a variable 'psT' and label 'Person' for the new node**
- **{ }**



Neo4j: MATCH statement (1)

MATCH: Finding nodes

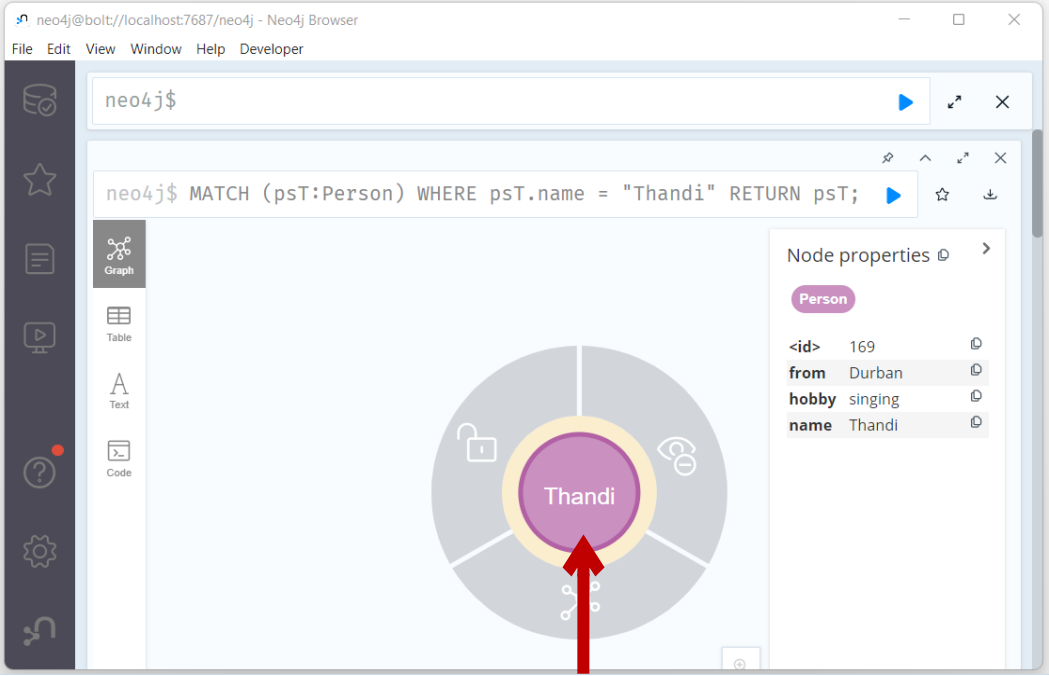
```
MATCH (psT:Person)
WHERE psT.name = "Thandi"
RETURN psT;
```

- **MATCH** clause to specify a pattern of nodes and relationships
- **(psT:Person)** a single node pattern with label 'Person' which will assign matches to the variable ‘psT‘
- **WHERE** clause to constrain the results
- **pst.name = "Thandi"** compares name property to the value " Thandi "
- **RETURN** clause used to request particular results

Neo4j: MATCH statement (2)

MATCH: Finding nodes

MATCH (psT:Person) WHERE psT.name = "Thandi" RETURN psT;

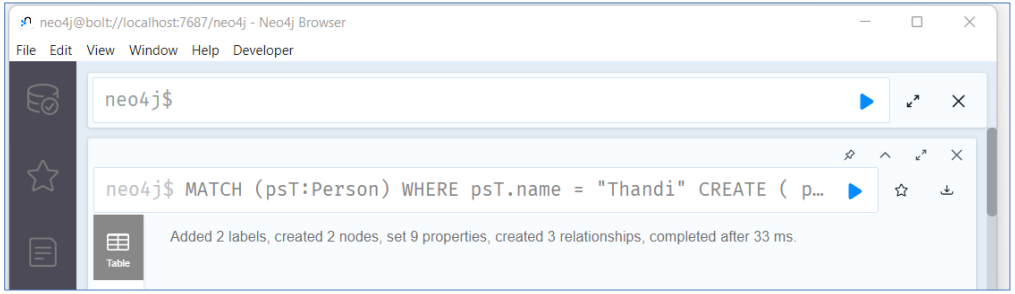


displayed when you click on the Thandi node

Neo4j: CREATE statement - continued

CREATE clauses can create many nodes and relationships at once.

```
MATCH (psT:Person) WHERE psT.name = "Thandi"
CREATE
        ( psJ:Person { name: "Johan", from: "Pretoria",  hobby: "surfing" }),
         (psN:Person { name: "Neo", from: "Cape Town", hobby: "soccer" }),

         (psT)-[:FOLLOWS {since: 2012}]->(psJ),
         (psT)-[:FOLLOWS {since: 2009}]->(psN),
         (psJ)-[:FOLLOWS {since: 2010}]->(psT)
```



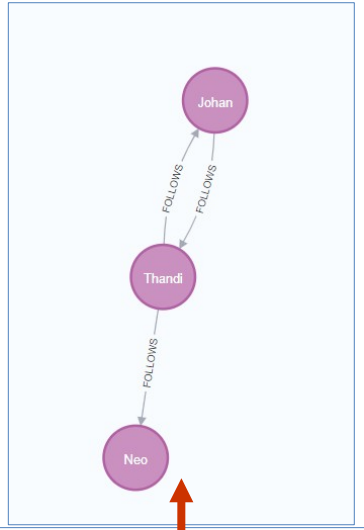
Neo4j: MATCH statement – again (1)

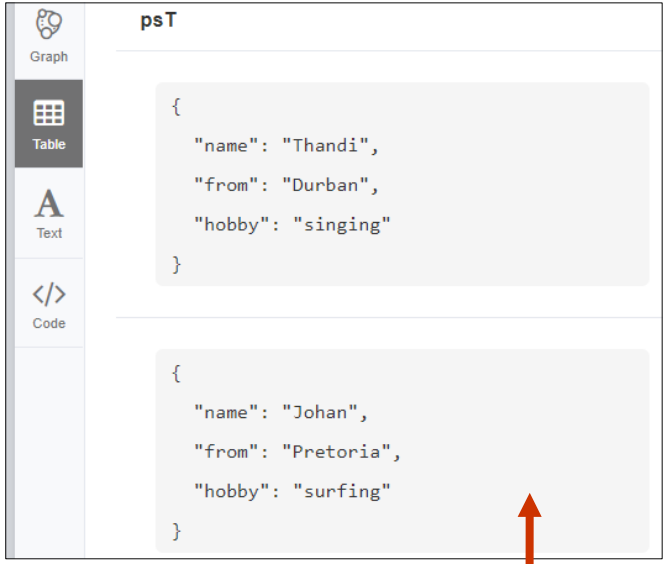
CREATE clauses can create many nodes and relationships at once.

MATCH (psT:Person) WHERE psT.name = "Thandi"

CREATE ..... **and then***

MATCH (psT:Person) RETURN psT;





Displayed in 'Graph' mode mode

Displayed in 'Table'

Neo4j: MATCH statement – again (2)

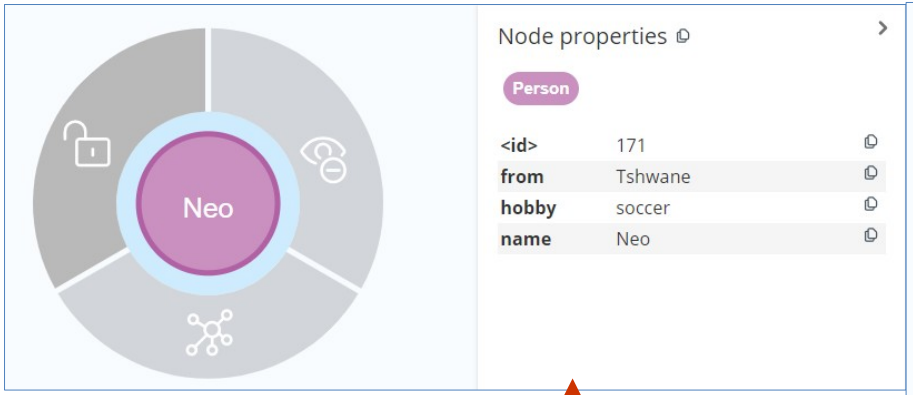
- Pattern matching
  - Describe what to find in the graph
  - a pattern can be used to find persons that Thandi follows

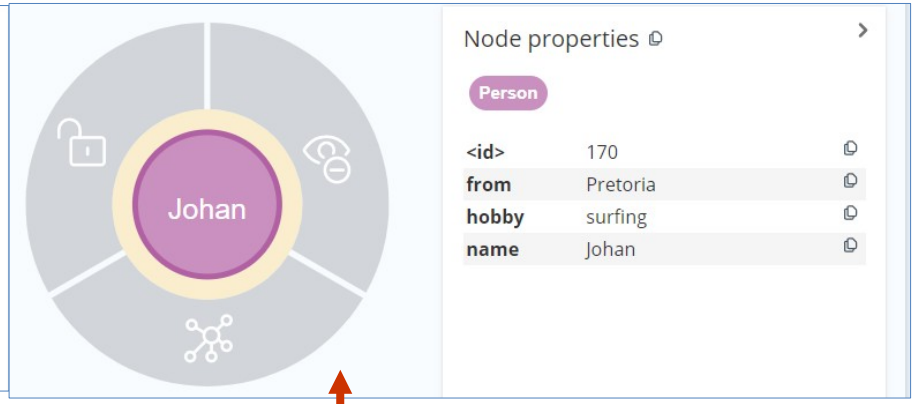
```
MATCH (psn:Person) -[:FOLLOWS]->(followed)
WHERE psn.name = "Thandi"
RETURN psn, followed
```

- MATCH clause to describe the pattern from known Nodes to found Nodes
- (psn) starts the pattern with a Person (qualified by WHERE)
- -[:FOLLOWS]-> matches “FOLLOWS”
- (followed) will be bound to the people Thandi follows

Neo4j: MATCH statement – again (3)

MATCH (psn:Person)-[:FOLLOWS]->(followed)
WHERE psn.name = "Thandi" RETURN followed





In 'Graph' mode when you click on Neo

In 'Graph' mode when you click on Johan

COS 326
Database Systems

Lecture 18

NoSQL Databases – Neo4j (2)

Chapter 24 (7th edition)

Thursday 03 October 2024

In this lecture

1. RECAP: Neo4j DB Environment

2. Neo4j CRUD operations
    a. WRITE clause
    b. READ clauses
    c. AGGREGATION clauses

3. Neo4j Desktop v1.6.0

Reference: Neo4j manual. Can be accessed from here:
https://neo4j.com/docs/getting-started/current/

Recap: Neo4j Cypher query language

Cypher: Neo4j's graph query language

- purpose built for working with graph data.
- uses patterns to describe graph data
- uses familiar SQL-like clauses
- is declarative i.e. describes what to do, not ho do it

[nodeID: 85]
name: "Thandi"
from: "Durban"

- A Neo4j graph consists of:
  - Nodes which store data as Properties
  - Properties are simple key/value pairs e.g. name: "Thandi" from: "Durban"
  - Plus arcs (relationships)
  - Plus optional components: Constraints
  - System assigns unique node IDs

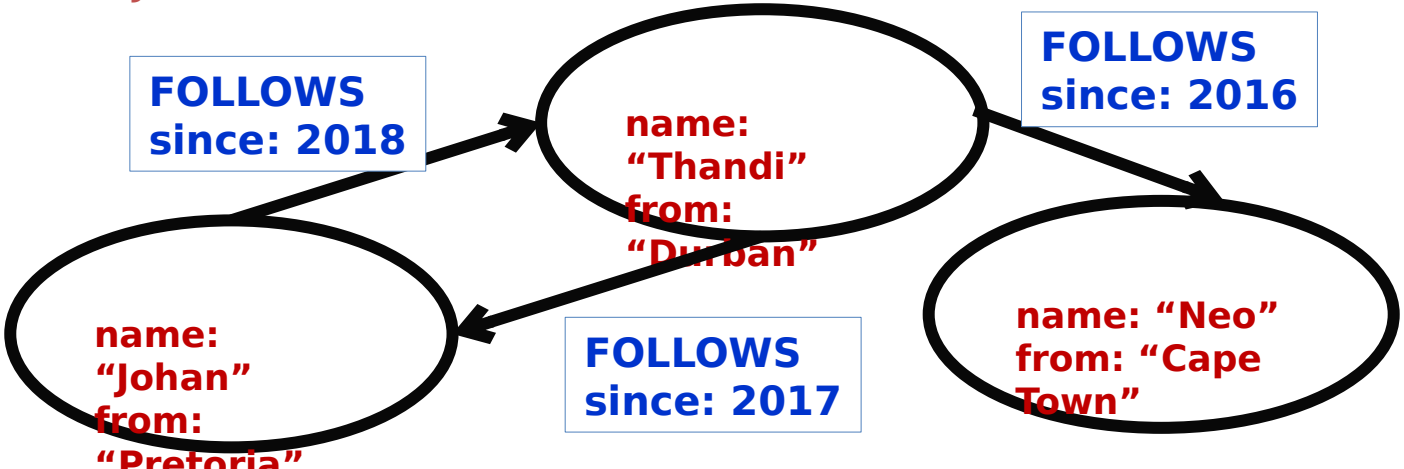
Nodes and Relationships (1)

Relationships: Connect nodes in the graph

- **The real power of a graph database e.g. Neo4j is in connected data.**
- **Relationships describe how the records are related. e.g. the FOLLOWS relationship**
  - **Thandi FOLLOWS Neo and Johan**
  - **Johan FOLLOWS Thandi**
- **Relationships always have direction & a type & form patterns of data**

Nodes and Relationships (2)

- **Relationship properties**
  - Store information shared by two nodes.
  - In a property graph, relationships are data records that can also contain properties. e.g
    - Thandi FOLLOWS Neo since 2016
    - Thandi FOLLOWS Johan since 2017
    - Johan FOLLOWS Thandi since 2018



CRUD operations: WRITING clauses (1)

- Writing clauses https://neo4j.com/docs/cypher-manual/current/clauses

Writing clause	Definition
CREATE	**Creating** graph elements**: nodes, relationships, constraints**
MERGE	a **combination of MATCH & CREATE** that additionally allows you to specify what happens if the data was matched or created.
SET	for **updating labels** on nodes **and properties** on nodes and relationships
DELETE	**Deleting** graph elements: **nodes and relationships**

CRUD operations: WRITING clauses (2)

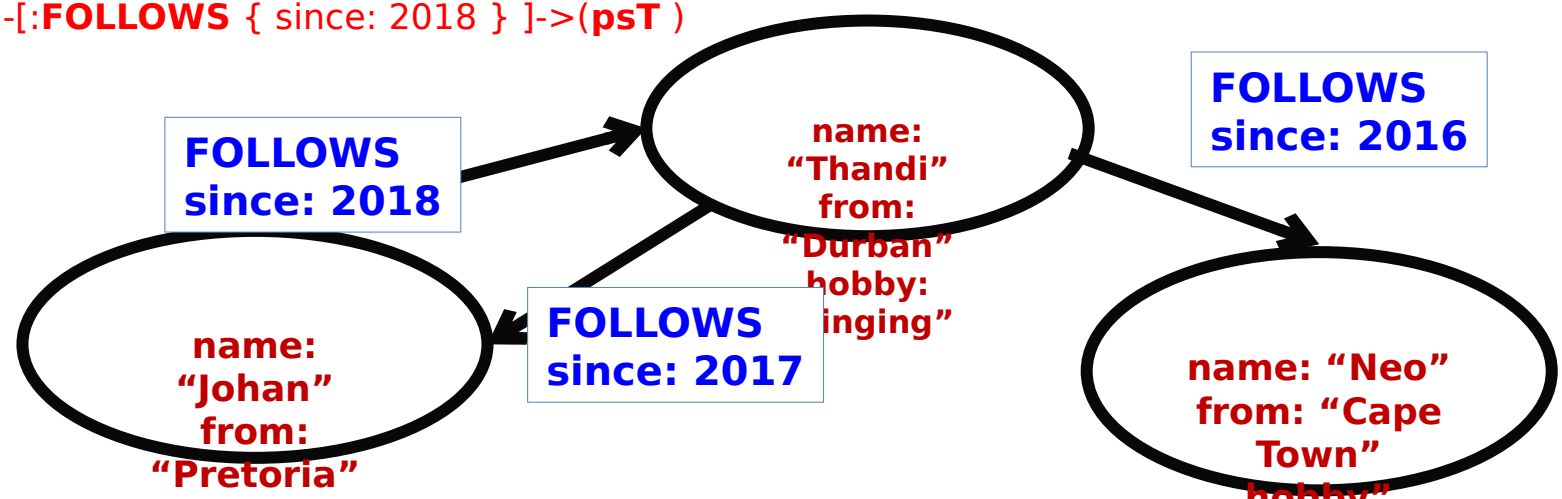
- Writing clauses https://neo4j.com/docs/cypher-manual/current/clauses

Writing clause	Definition
REMOVE	for **removing properties and labels** from graph elements
FOREACH	use **for updating commands** on elements in a collection - a path, or a collection created by aggregation.
CREATE UNIQUE	**Create unique nodes. Create node if missing.** If the pattern described needs a node, and it can't be matched, a new node will be created.

CRUD operations: CREATE (1)

A CREATE clause can create many nodes and relationships at once.

```
CREATE
    ( psT:Person { name: “Thandi", from: “Durban",  hobby: “singing" }  ),
    ( psJ:Person { name: "Johan", from: "Pretoria",  hobby: "surfing" }   ),
    (psN:Person { name: "Neo", from: "Cape Town", hobby: "soccer“   }  ),
    (psT)-[:FOLLOWS { since: 2017 } ]->(psJ ),
    (psT)-[:FOLLOWS { since: 2016 } ]->(psN ),
    (psJ)-[:FOLLOWS { since: 2018 } ]->(psT )
```



CRUD operations: CREATE (2)

A CREATE clause can create a constraint and an index

CREATE CONSTRAINT ON (psn: Person)
ASSERT psn.name IS UNIQUE

Note: adding the unique constraint will add an index on that property

Alternatively: CREATE INDEX ON : Person(name)

```
CYPHER CREATE CONSTRAINT ON ( psn: Person) ASSERT psn.name IS UNIQUE;
```

✔ Added 1 constraint, returned 0 rows in 1140 ms

CRUD operations: SET

For updating labels on nodes and properties on nodes & relationships

```
MATCH (psT)
WHERE  psT.name =  "Thandi"   SET psT.age   =   20   RETURN psT
```

BUT for relationships:

```
MATCH (psN),  (psT)
WHERE psN.name =  "Neo" AND  psT.name =   "Thandi"
CREATE  (psN)-[:FOLLOWS { since: 2018 } ]->(psT )
```

CRUD operations: MERGE

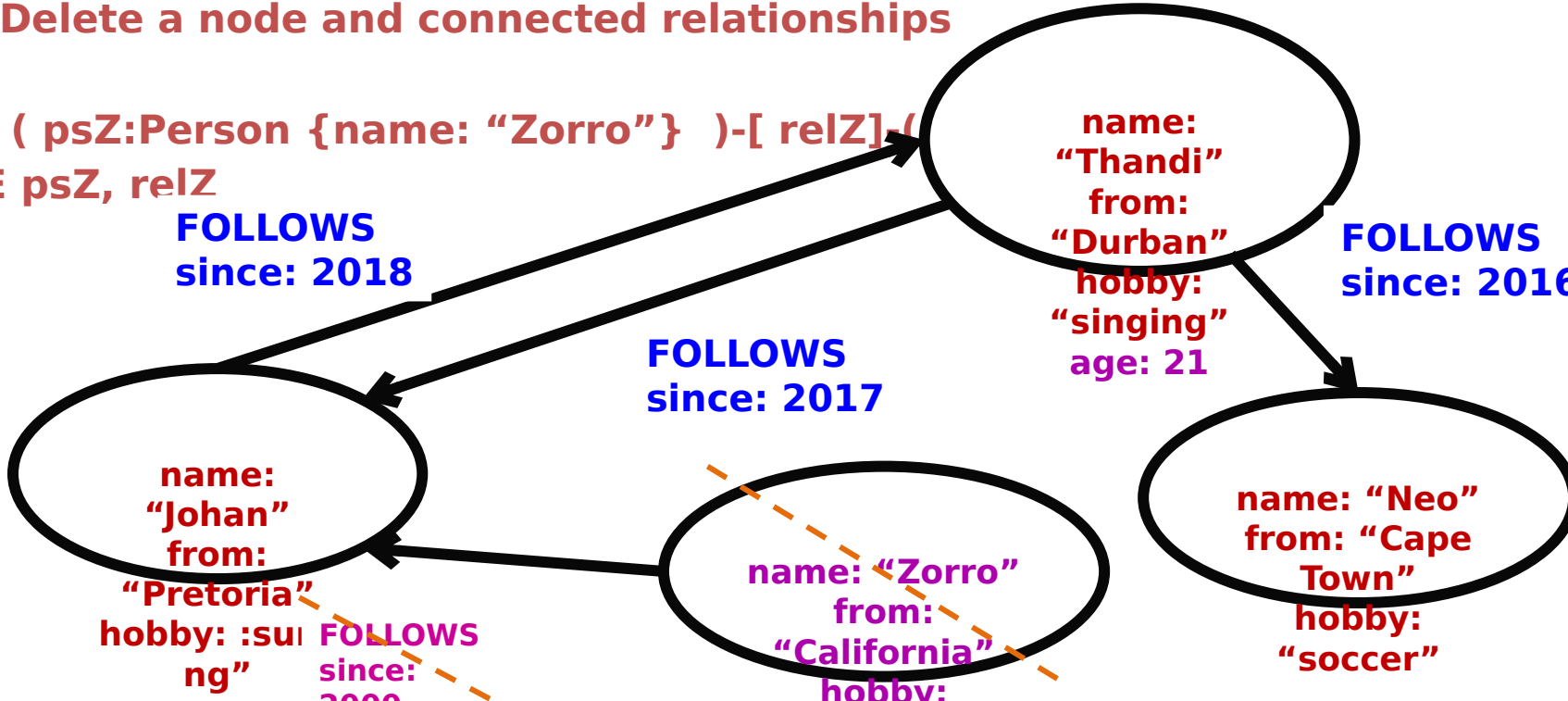
either matches existing node and binds it, or creates new node and binds it. (Can use only ON CREATE or only ON MATCH or both)

```
MERGE  ( psT:Person { name: "Thandi"} )
ON MATCH SET    psT.age = 21
ON CREATE SET psT.name = "Thandi", psT.from = "Durban",
        psT.hobby = "singing", psT.age = 21
RETURN   psT
```

CRUD operations: DELETE (1)

- Deleting graph elements: nodes and relationships
- e.g. Delete a node and connected relationships

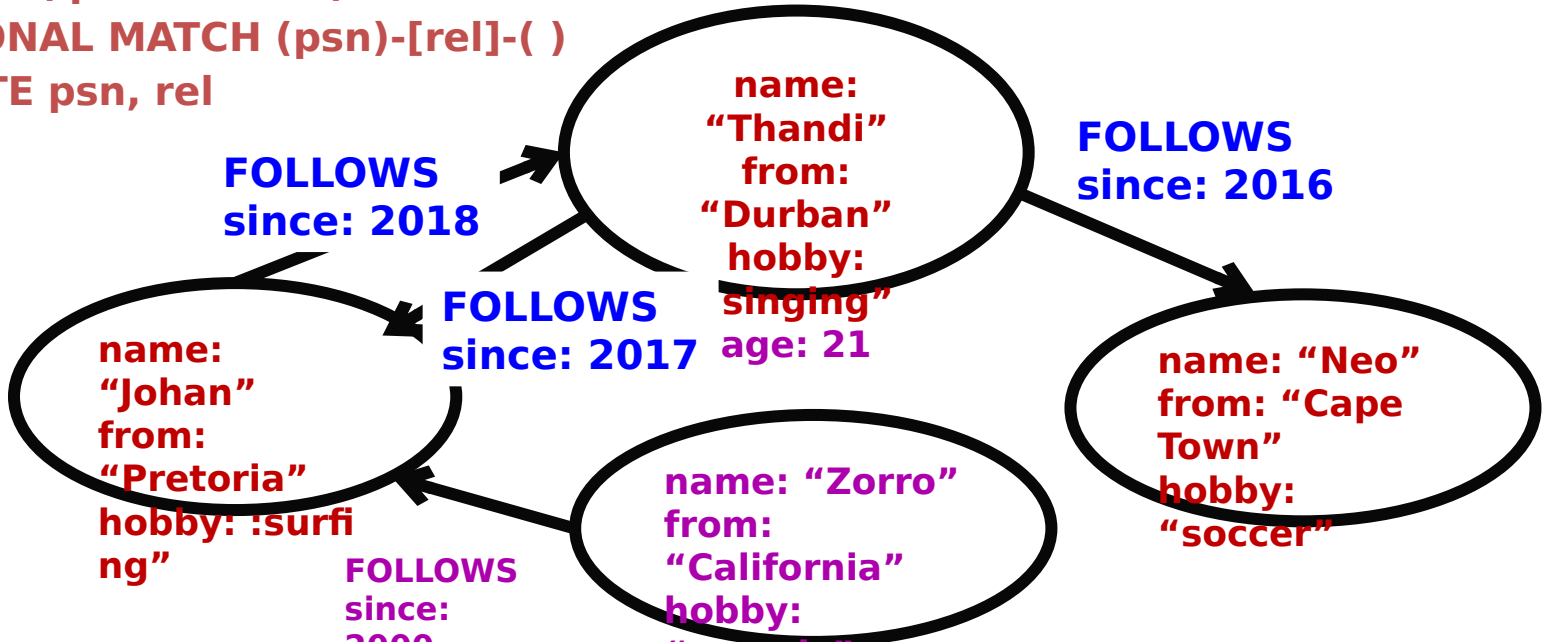
```
MATCH ( psZ:Person {name: “Zorro”}  )-[ relZ]-(
DELETE psZ, relZ
```



CRUD operations: DELETE (2)

- **Deleting graph elements: nodes and relationships**
- e.g. Delete all person nodes and connected relationships

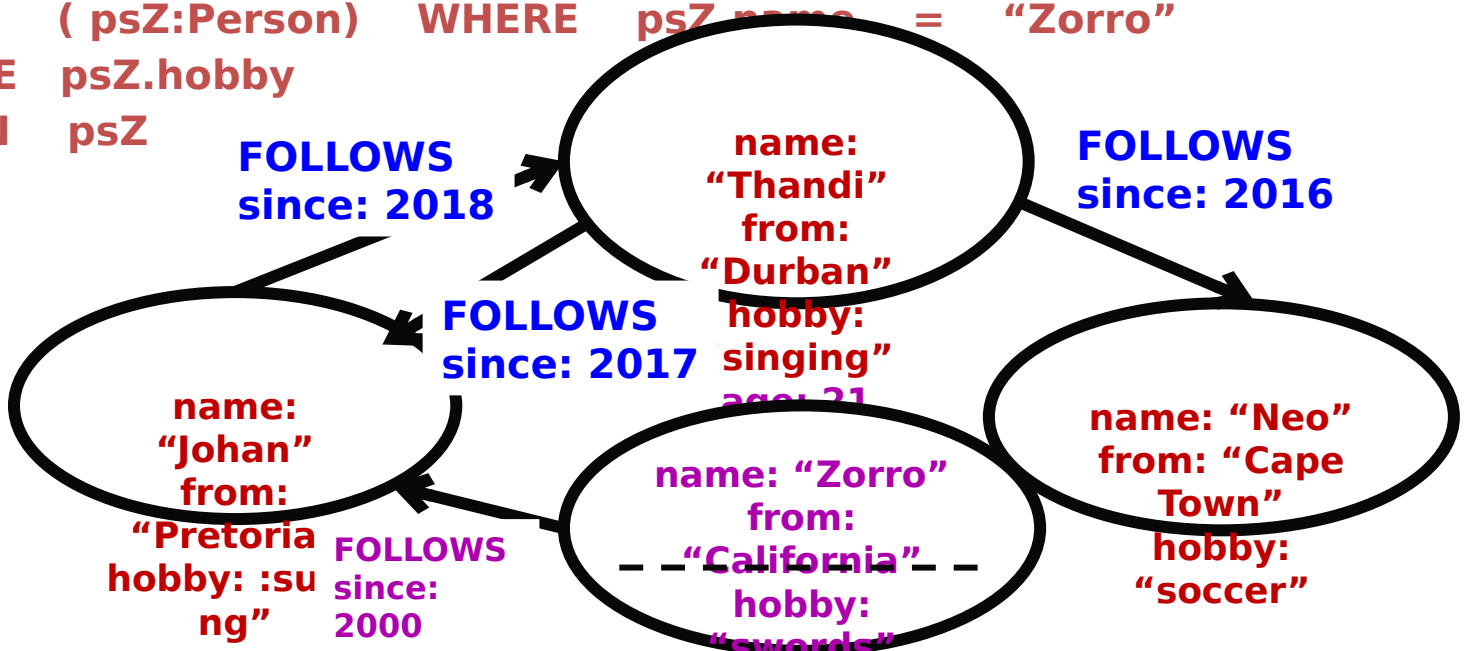
```
MATCH ( psn:Person )
OPTIONAL MATCH (psn)-[rel]-( )
DELETE psn, rel
```



CRUD operations: REMOVE

- **removing properties and labels** from graph elements (nodes & relationships) e.g.

```
MATCH    ( psZ:Person)   WHERE   psZ.name   =   “Zorro”
REMOVE   psZ.hobby
RETURN   psZ
```



CRUD operations: READING clauses (1)

- Reading clauses https://neo4j.com/docs/cypher-manual/current/clauses

Reading clause	Definition
MATCH	The MATCH clause allows: you to specify the patterns Cypher will search for in the database.
OPTIONAL MATCH	if no matches are found, OPTIONAL MATCH will use NULLs for missing parts of the pattern.

CRUD operations: READING clauses (2)

- Reading clauses https://neo4j.com/docs/cypher-manual/current/clauses

Reading clause	Definition
WHERE	**WHERE is not a clause in it's own right**, rather, it's **part of MATCH, OPTIONAL MATCH, START and WITH.** In the case of WITH and START, WHERE simply filters the results.
AGGREGATION clauses	To calculate aggregated data, Cypher offers aggregation, **much like SQL's aggregates and GROUP BY.**

CRUD operations: READING clauses (3)

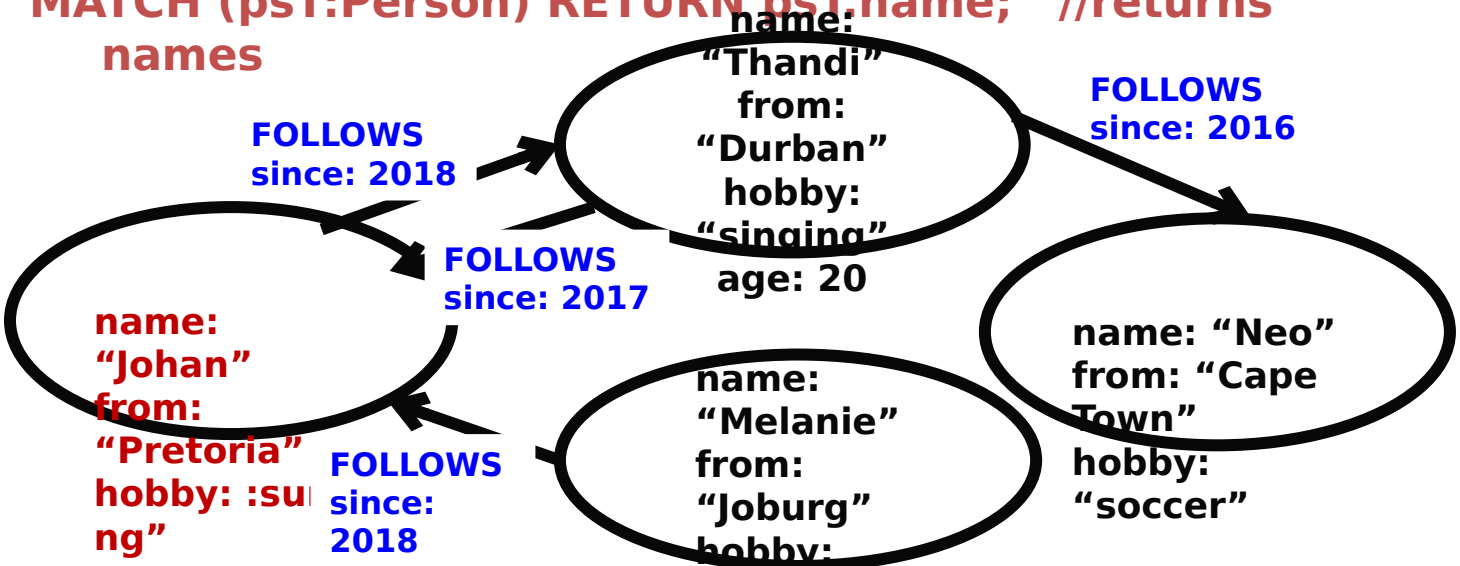
- **Patterns for MATCH**

Pattern	meaning
MATCH (<var>)	any node
MATCH (<var>: <label>)	single node pattern with label <label> which will assign matches to the variable <var>
MATCH (<var1>:<label>) -[:<relationship]- (<var2>)	
MATCH (<var1> { <key>: <value> })-->(<var2>)	--> specifies a path

CRUD operations: MATCH

MATCH (psT:Person) RETURN psT; //returns nodes

MATCH (psT:Person) RETURN psT.name; //returns names



CRUD operations: MATCH & WHERE

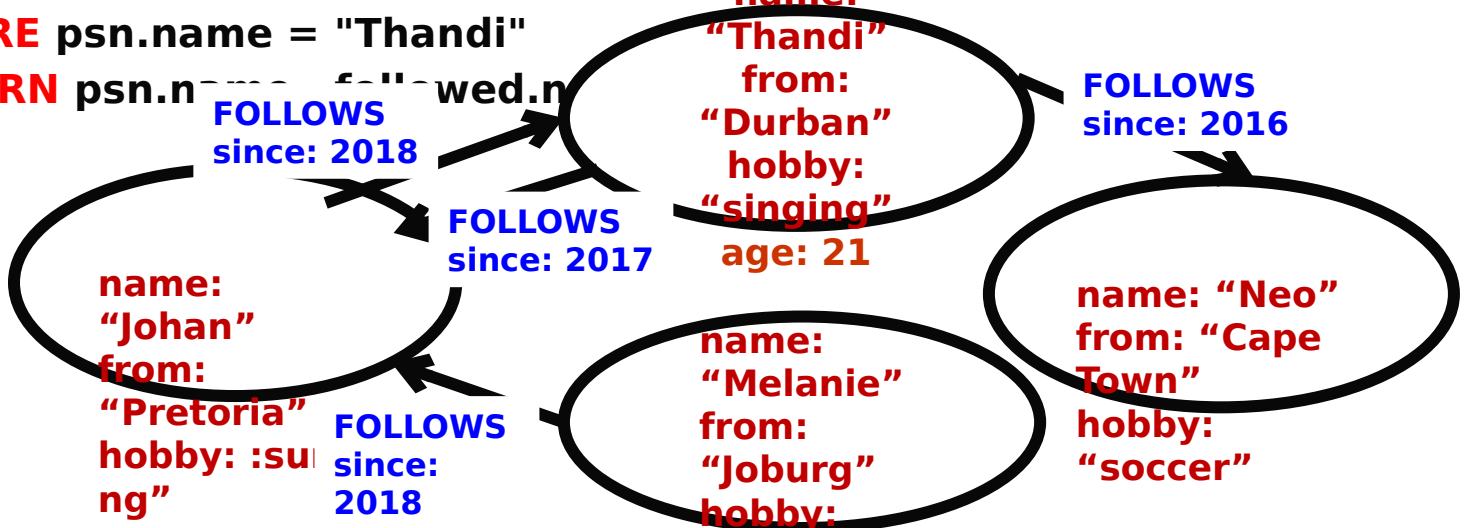
specify selection criteria for patterns

e.g. show Melanie's details

```
MATCH (psM)   WHERE psM.name =  "Melanie"
 RETURN psM.name,    psM.from,   psM.hobby
```

e.g. **find the people that Thandi follows**

```
MATCH (psn:Person)-[:FOLLOWS]->(followed)
WHERE psn.name = "Thandi"
RETURN psn.name, followed.name
```

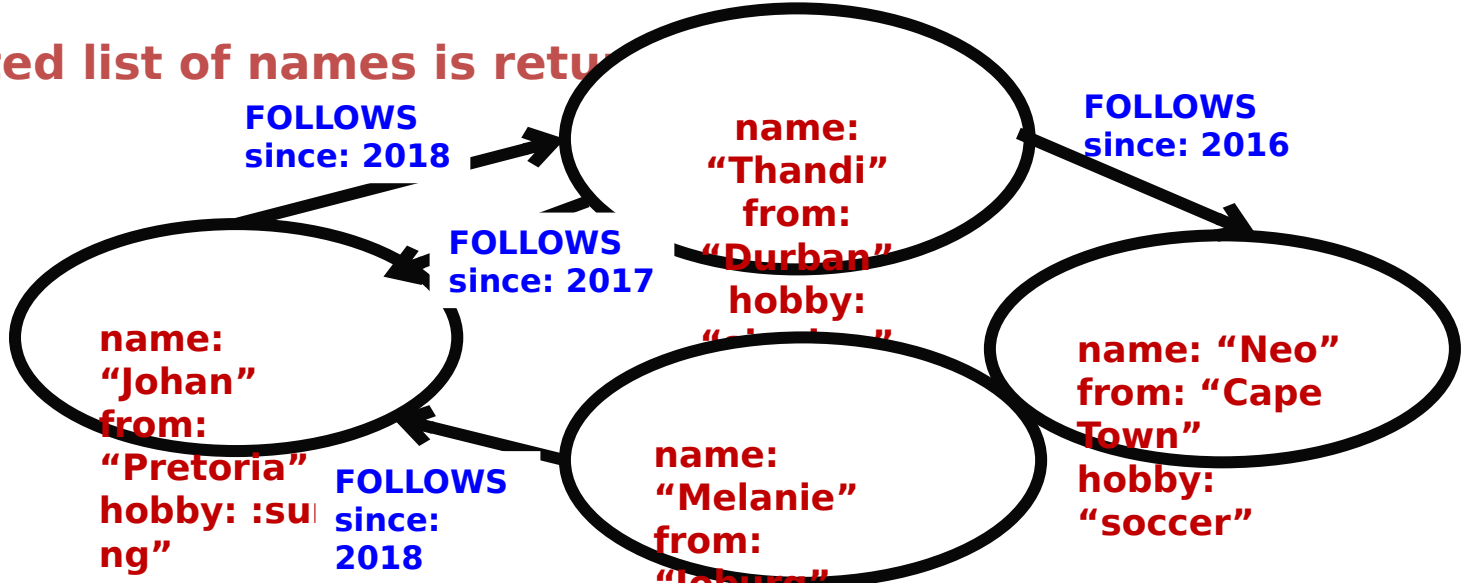


CRUD operations: MATCH & ORDER BY

- General clauses https://neo4j.com/docs/cypher-manual/current/clauses
- **ORDER BY is used to sort the output.**

MATCH (psn) RETURN psn.name
ORDER BY psn.name

- **Sorted list of names is retu**



CRUD operations: AGGREGATION (1a)

Aggregation function	Example
‘GROUP BY’	MATCH (n { name: 'A' })-->(x) RETURN n.name, count(x) n.name is the grouping key, count(x) is the aggregate
count	RETURN count(*)
sum	RETURN sum(n.property) property is numeric

To compute aggregate values from data, Cypher offers aggregation, much like SQL’s aggregates and GROUP BY.

CRUD operations: AGGREGATION (1b)

To calculate aggregated data, Cypher offers aggregation, much like SQL's aggregates and GROUP BY.

Aggregation function	example
max	RETURN max(n.property)
min	RETURN min(n.property)
collect	RETURN collect(n.property) displays a collection. a collection = list of values
distinct	RETURN count(DISTINCT b.name)

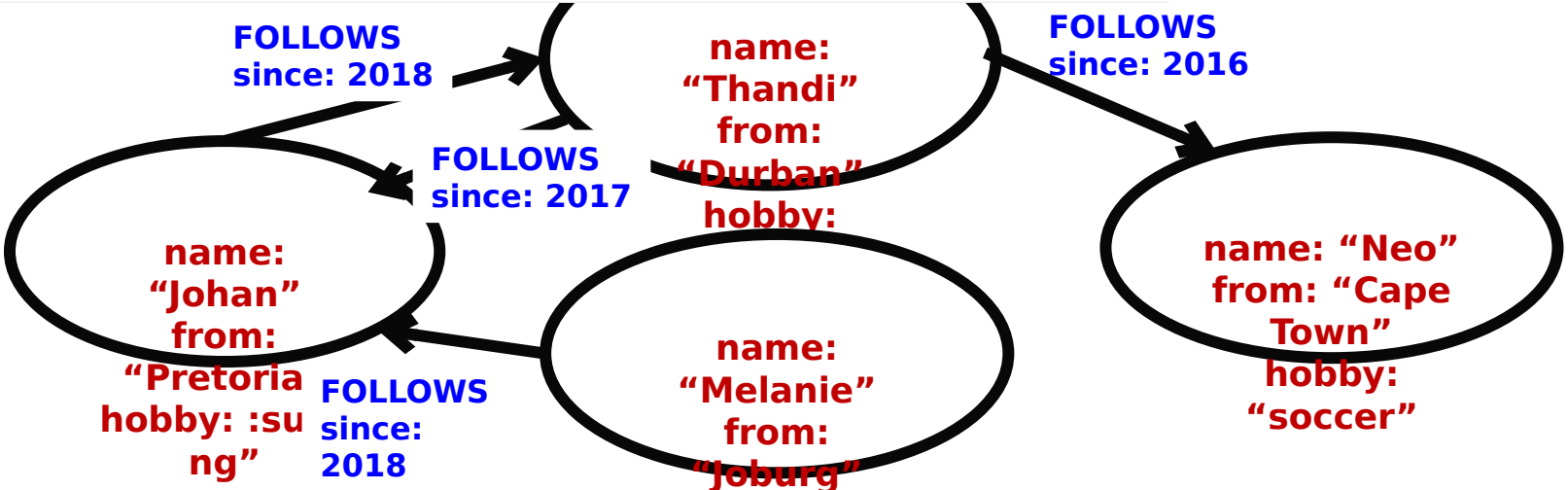
CRUD operations: AGGREGATION (2)

e.g.1 count the people that Thandi follows:

```
MATCH (psT {name: "Thandi"} )-[:FOLLOWS]-> (followed)
```

```
CYPHER MATCH (psT {name: "Thandi"} )-[:FOLLOWS]-> followed RETURN psT.name, count(*) AS count_follows;
```

psT.name	count_follows
Thandi	2



CRUD operations: AGGREGATION (3)

e.g.2 Count the number of relationships in the graph

```
MATCH    (psn )-[relation]->( )
RETURN  type(relation), count(*)
```

CRUD operations: AGGREGATION (4)

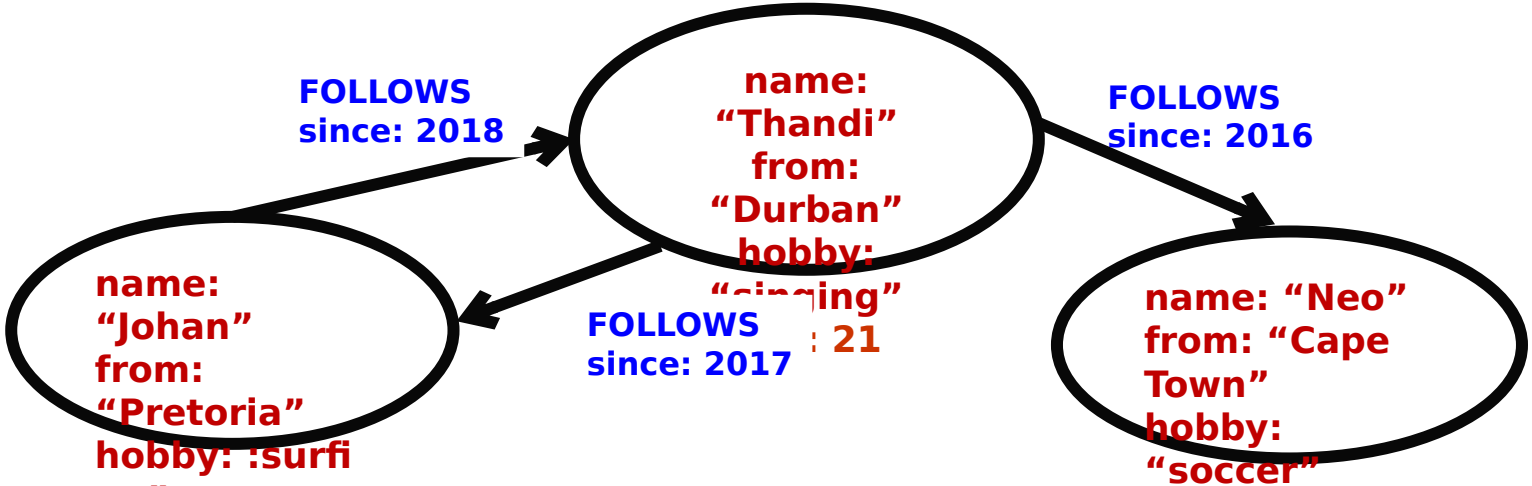
Collect,
e.g

```
MATCH    (psn:Person )
RETURN  collect(psn.name);
```

CYPHER MATCH (psn:Person)

collect(psn.name)
Thandi, Johan, Neo

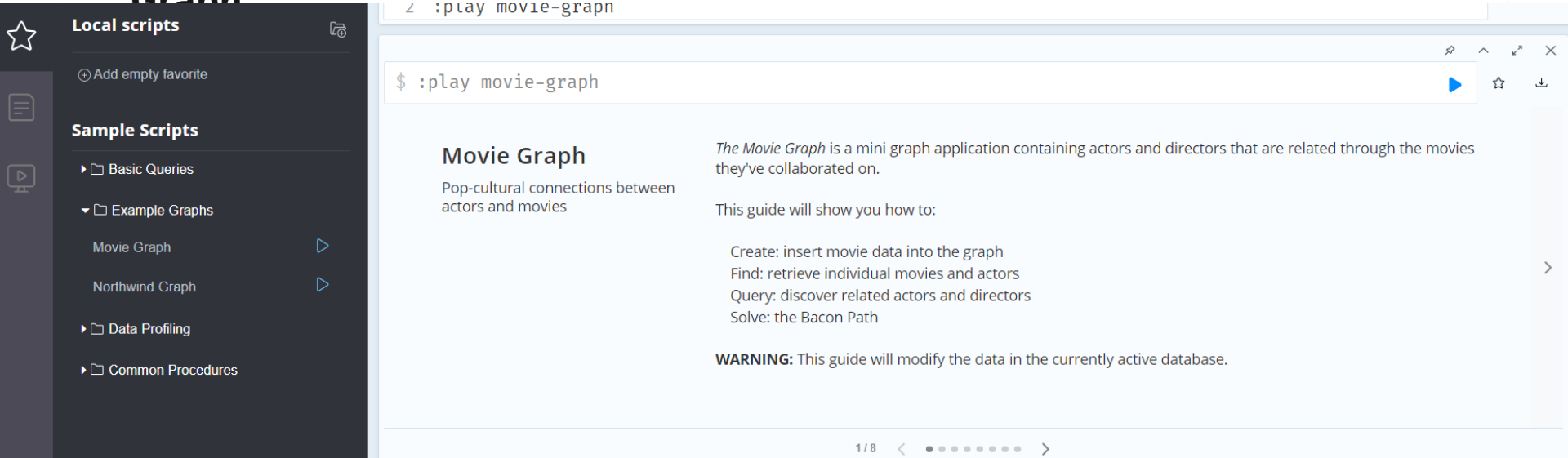
✔ Returned 1 row in 328 ms



HOME WORK (1)

- Work through the Neo4j client tutorial on ‘The Movie Graph’

- To access the Movie Graph, click **Favourites** (star icon on the left of the Neo4j Browser window)
- Then under **Sample Scripts**, expand **Example Graphs**, then you will see **Movie Graph**



HOME WORK (2)

- **Work through the Neo4j tutorial on ‘The Movie Graph’.**

e.g. query:

```
MATCH (people:Person)-[relatedTo]-(:Movie {title: "Cloud Atlas"})
RETURN people.name, Type(relatedTo), relatedTo
```

```
neo4j$ MATCH (people:Person)-[relatedTo]-(:Movie {title: "Cloud Atlas"}) RETURN people.name, Ty…
```

Table | Text | Code

"people.name"	"Type(relatedTo)"	"relatedTo"
"Tom Hanks"	"ACTED_IN"	{"roles":["Zachry","Dr. Henry Goose","Isaac Sachs","Dermot Hoggins"]}
"Jim Broadbent"	"ACTED_IN"	{"roles":["Vyvyan Ayrs","Captain Molyneux","Timothy Cavendish"]}
"Tom Tykwer"	"DIRECTED"	{}
"Lana Wachowski"	"DIRECTED"	{}
"Jessica Thompson"	"REVIEWED"	{"summary":"An amazing journey","rating":95}
"Halle Berry"	"ACTED_IN"	{"roles":["Luisa Rey","Jocasta Ayrs","Ovid","Meronym"]}
"Hugo Weaving"	"ACTED_IN"	{"roles":["Bill Smoke","Haskell Moore","Tadeusz Kesselring","Nurse Noakes","Boardman Mephi","Old Georgie"]}
"Andy Wachowski"	"DIRECTED"	{}

COS 326
Database Systems

Lecture 19

NoSQL Databases – Neo4j (3)

Chapter 24 (7th edition)

Tuesday 08 October 2024

In this lecture

1. Neo4j functions
    a. READ functions
    b. Other functions: EXISTS, LENGTH, NODES
    c. Mathematical & Statistical functions
2. Database operations
    a. patterns & paths
    b. graph algorithms
3. Importing data into a Neo4j database
4. Aspects of graph databases
    a. Data models for graph databases
    b. Database design
5. Problems solved by the Neo4j DB

CRUD operations: READ-FUNCTIONS (1a)

- Functions: https://neo4j.com/docs/cypher-manual/current/functions/

Function category	Description & example
Predicates **ALL, ANY, NONE, SINGLE, EXISTS**	**return true or false for a given set of input.** used to filter out subgraphs in the WHERE part of a query
scalar functions **LENGTH, TYPE,ID, etc**	Scalar functions return a single value
collection functions **NODES, RELATIONSHIPS, LABELS, etc.**	Collection functions **return collections of things: nodes in a path,** etc.

CRUD operations: READ-FUNCTIONS (1b)

- Functions: https://neo4j.com/docs/cypher-manual/current/functions/

Function category	Description & example
mathematical & statistical functions: **trigonometric, statistical, other**	**operate on numerical expressions**
string functions **SUBSTRING, TRIM, LOWER, UPPER etc.**	**operate on string expressions**

CRUD operations: functions (1a)

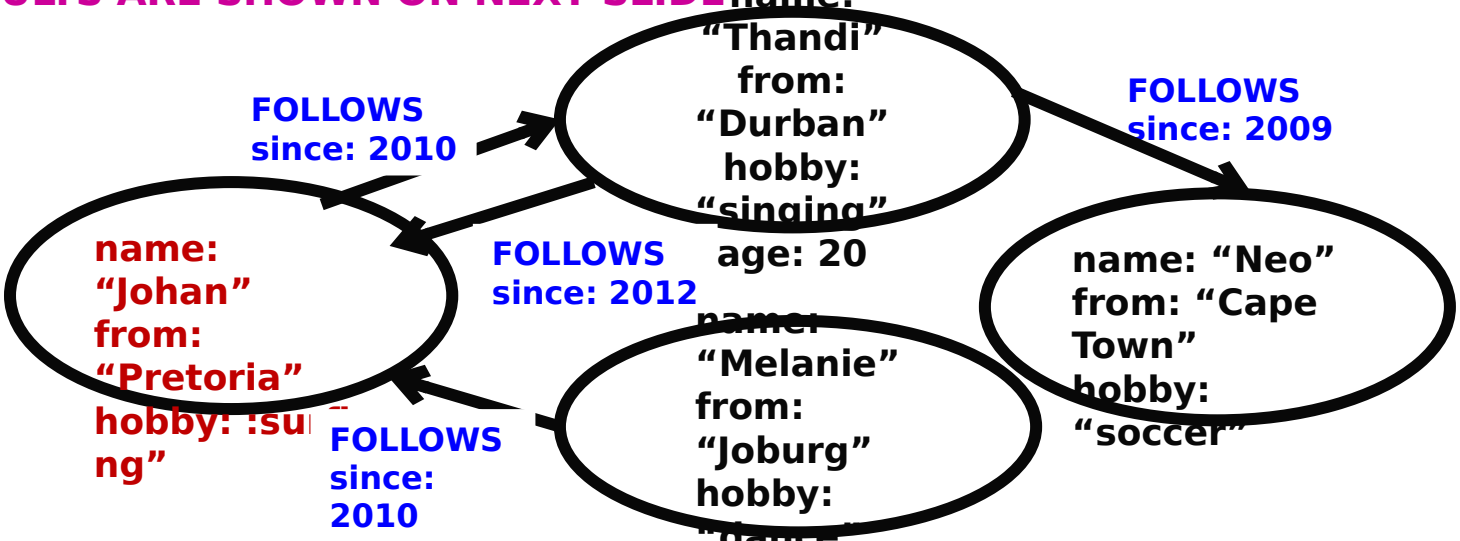
Predicates: ALL, ANY, NONE, SINGLE, EXISTS

List all the nodes with a name property and true/false indicating if they follow someone

MATCH (n) WHERE (n.name) IS NOT NULL

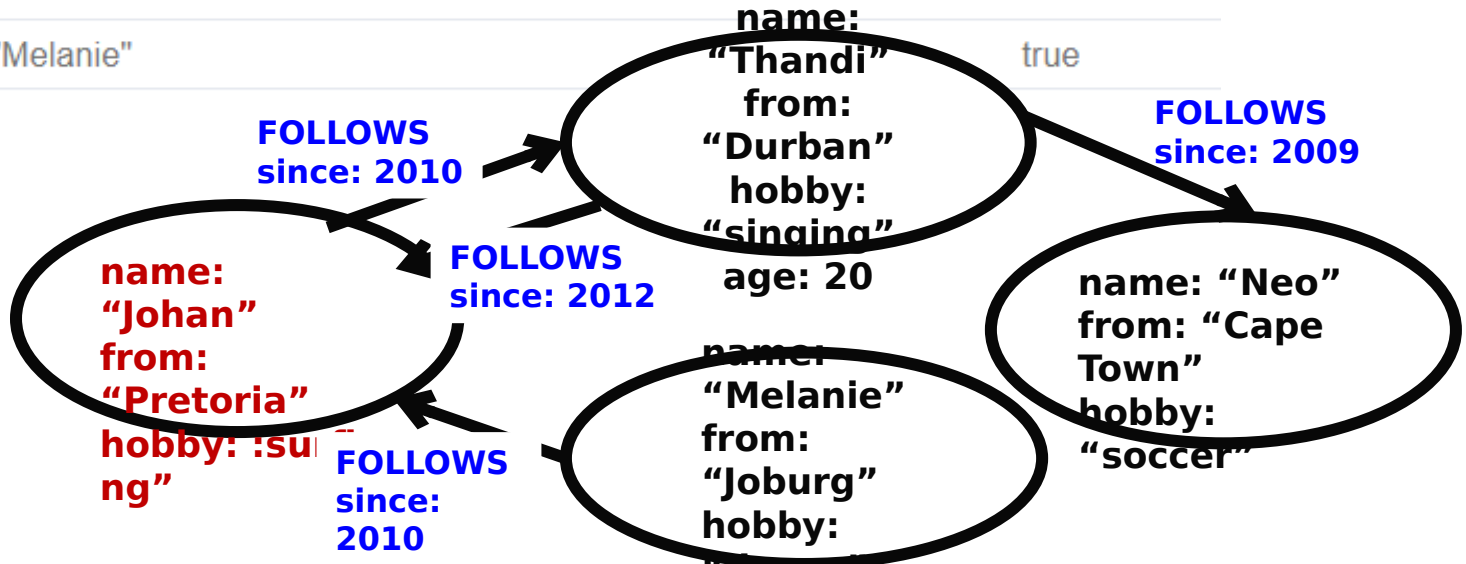
RETURN n.name AS name, EXISTS((n)-[:FOLLOWS]->( )) AS isAfollower

******RESULTS ARE SHOWN ON NEXT SLIDE********



CRUD operations: functions (1b)

name	isAfollower
"Thandi"	true
"Johan"	true
"Neo"	false
"Melanie"	true



CRUD operations: functions (2a)

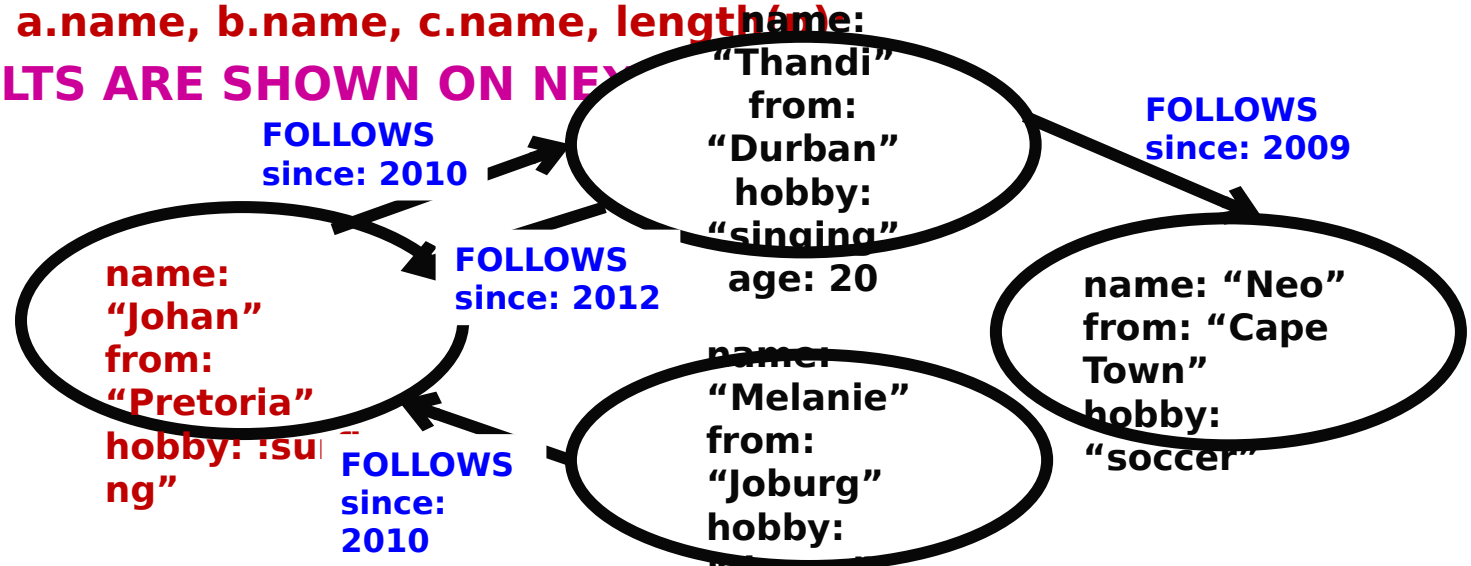
Scalar functions: LENGTH, TYPE, ID, etc

For all paths of length 2, list the node names and path length:

MATCH p = (a)-->(b)-->(c)

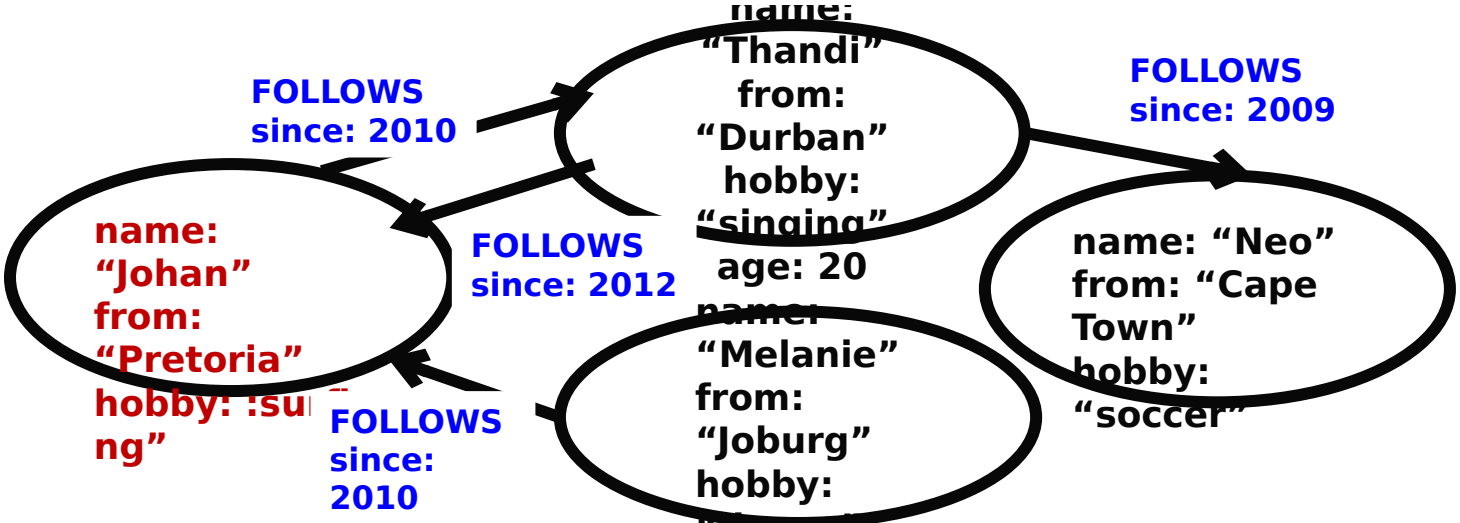
RETURN a.name, b.name, c.name, length(p)

****RESULTS ARE SHOWN ON NEX**



CRUD operations: functions (2b)

a.name	b.name	c.name	length(p)
"Johan"	"Thandi"	"Neo"	2
"Johan"	"Thandi"	"Johan"	2
"Melanie"	"Johan"	"Thandi"	2
"Thandi"	"Johan"	"Thandi"	2



CRUD operations: functions (3a)

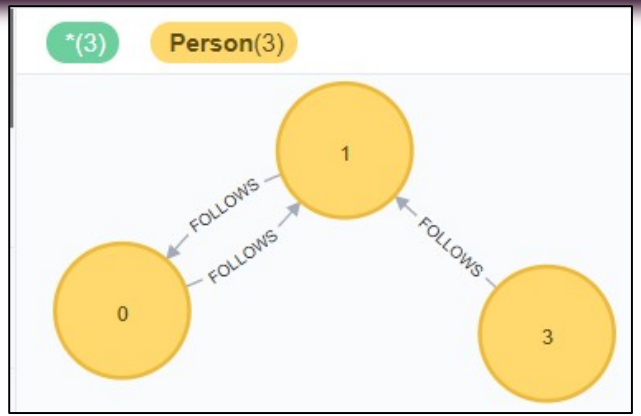
Collection functions: NODES, RELATIONSHIPS, LABELS, etc.

For all paths of length 2, between Melanie & Thandi, show the nodes in the path p:

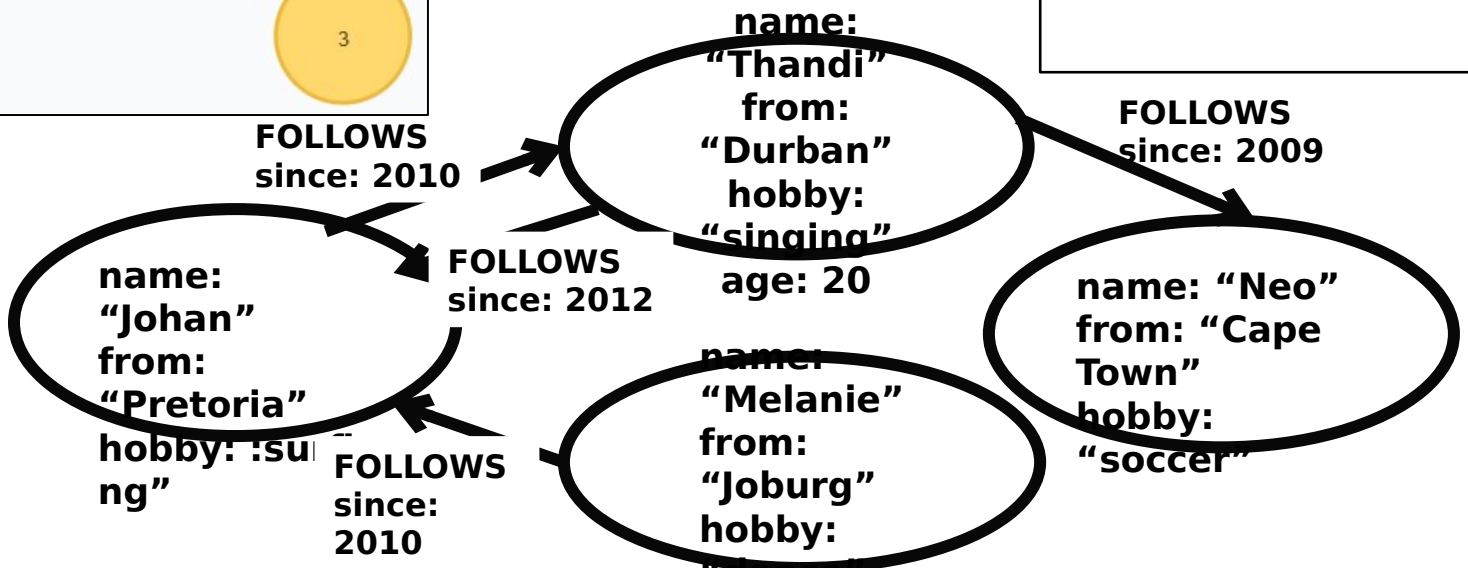
```
MATCH   p  =   (a)-->(b)-->(c)
WHERE a.name = "Melanie" AND c.name = "Thandi"
RETURN nodes(p)
```

******RESULTS ARE SHOWN ON NEXT SLIDE********

CRUD operations: functions (3b)



Node id	Person
0	Thandi
1	Johan
3	Melanie

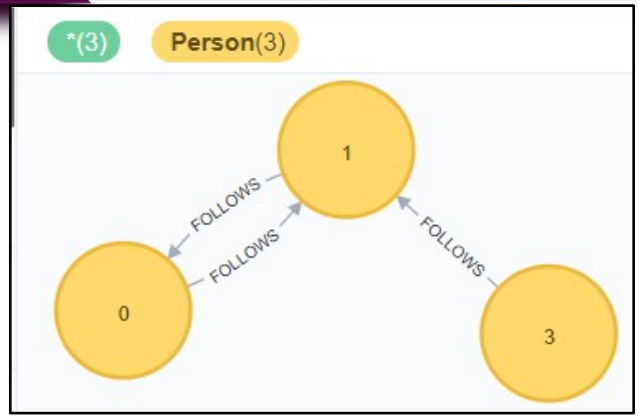


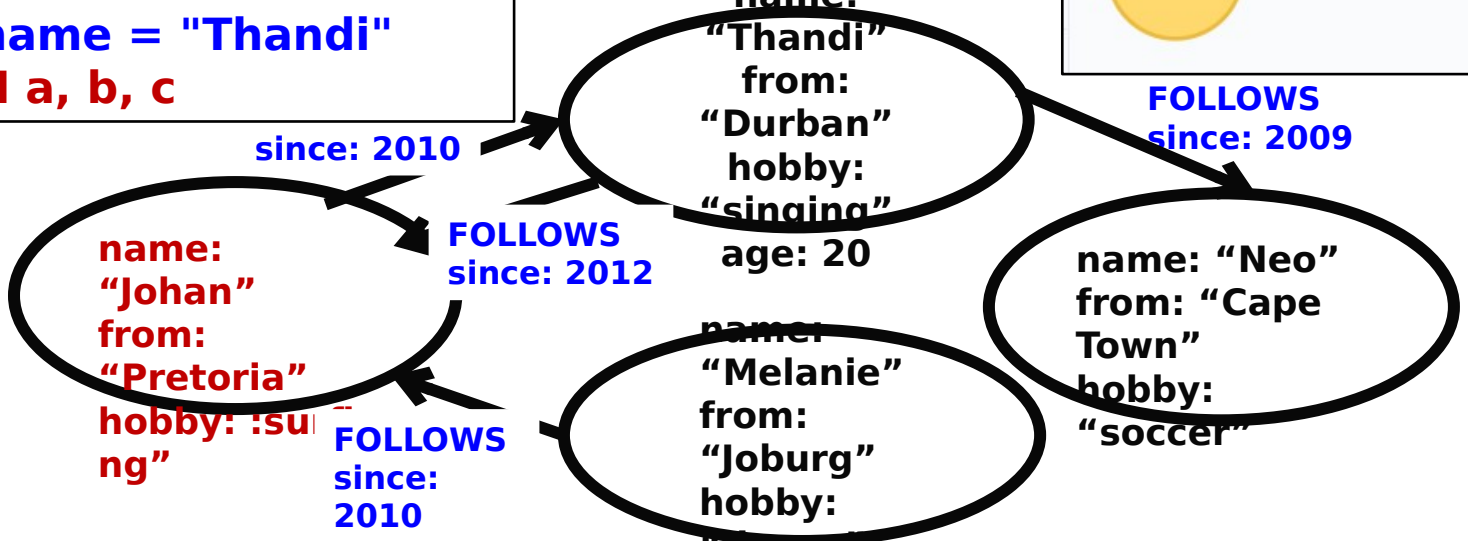
CRUD operations: functions (3c)

Note that this query will return the same results:

```
MATCH p = (a)-->(b)-->(c)
WHERE a.name = "Melanie"
AND c.name = "Thandi"
RETURN a, b, c
```

Node id	Person
0	Thandi
1	Johan
3	Melanie





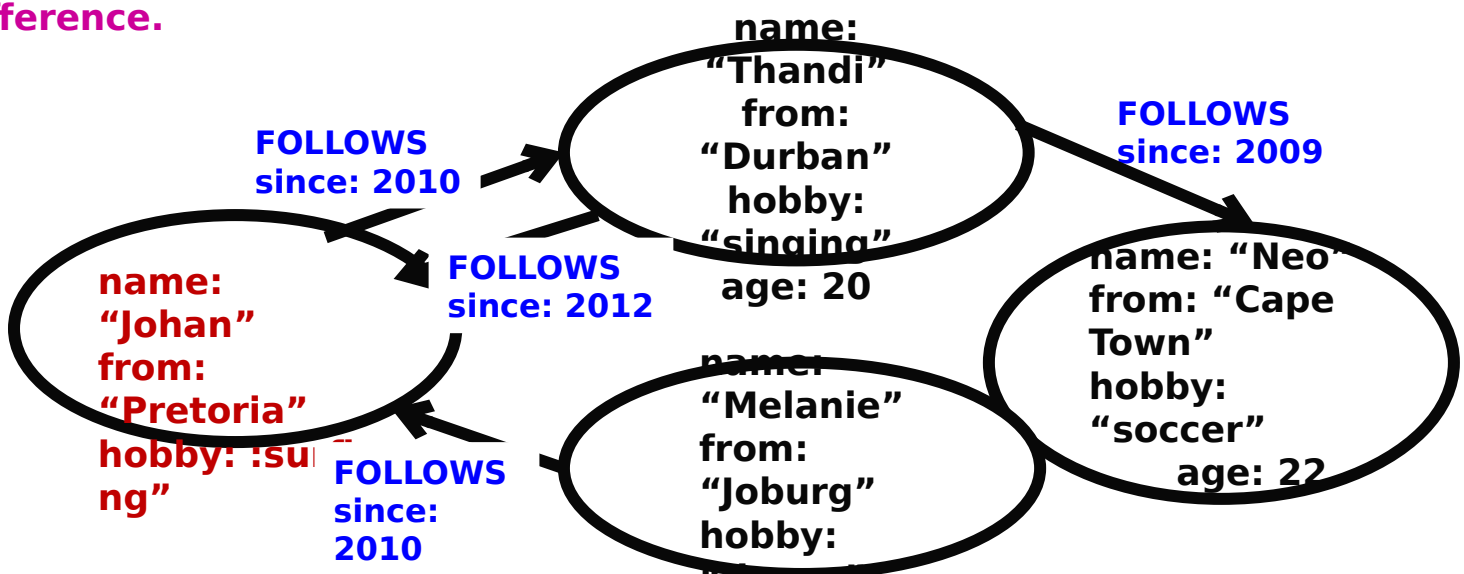
CRUD operations: functions (4)

Mathematical & Statistical functions:

```
MATCH (a), (b)
WHERE a.name = "Thandi" AND b.nar
RETURN a.age, b.age, abs(a.age - b.age)
```

a.age	b.age	ageDifference
20	22	2

NOTE: abs(..) is the absolute value of the age difference.



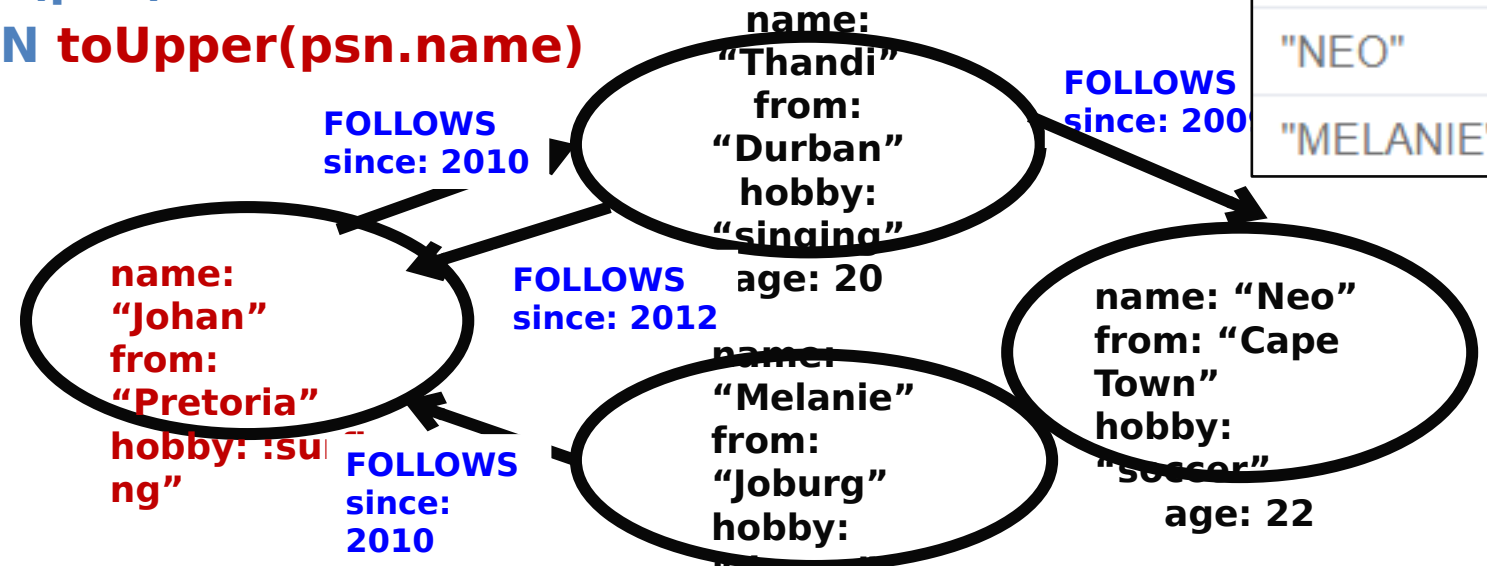
CRUD operations: functions (5)

String functions: **SUBSTRING, TRIM, toLOWER, toUPPER etc.**

returns the original strings in uppercase.

MATCH (psn)
RETURN toUpper(psn.name)

upper(psn.name)
"THANDI"
"JOHAN"
"NEO"
"MELANIE"



CRUD operations: patterns & paths (1)

You can specify variable length patterns for paths:

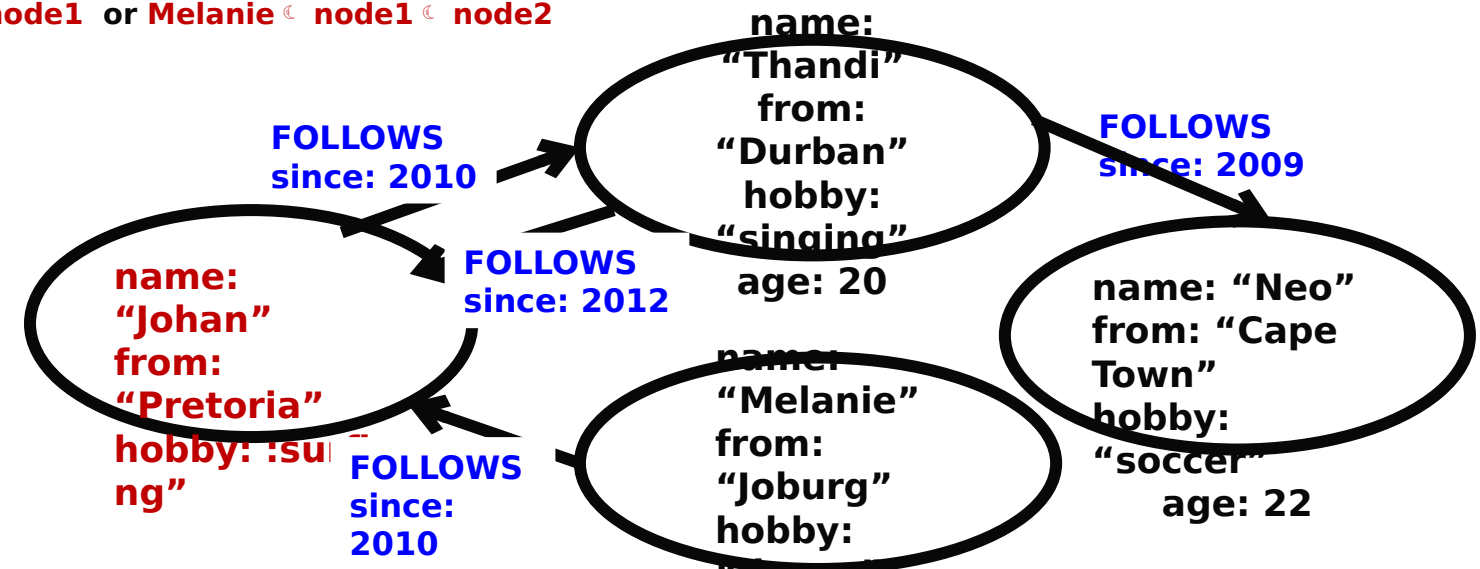
e.g. all Persons that are 1 or 2 links away from Melanie

```
MATCH (psM:Person {name:"Melanie"})-[*1..2]-(psn)
RETURN DISTINCT psn.name;
```

returns names for nodes node1 & node 2 where

Melanie ☾ node1 or Melanie ☾ node1 ☾ node2

psn.name
Johan
Thandi



CRUD operations: patterns & paths (2a)

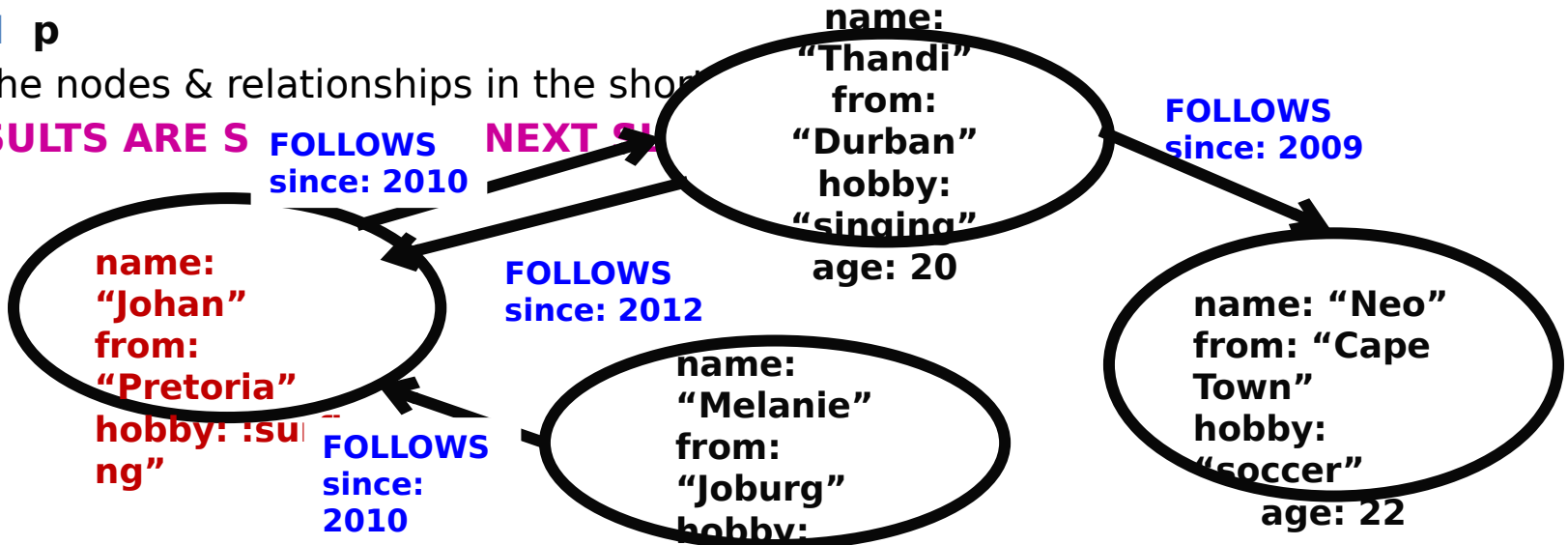
Variable length patterns for paths:

e.g. shortest path from Johan to Neo:

```
MATCH p = shortestPath (
    (psJ:Person {name:"Johan"})-[*]-(psN:Person
    {name:"Neo"})  )
RETURN  p
```

returns the nodes & relationships in the shor

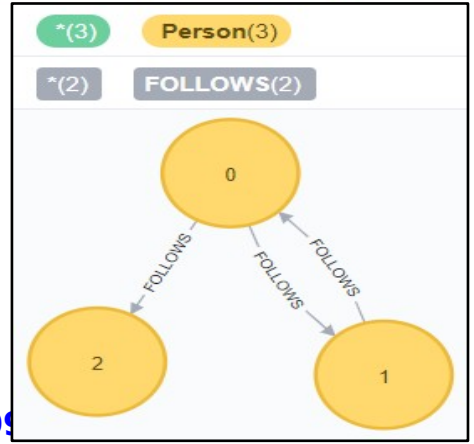
****RESULTS ARE S NEXT S

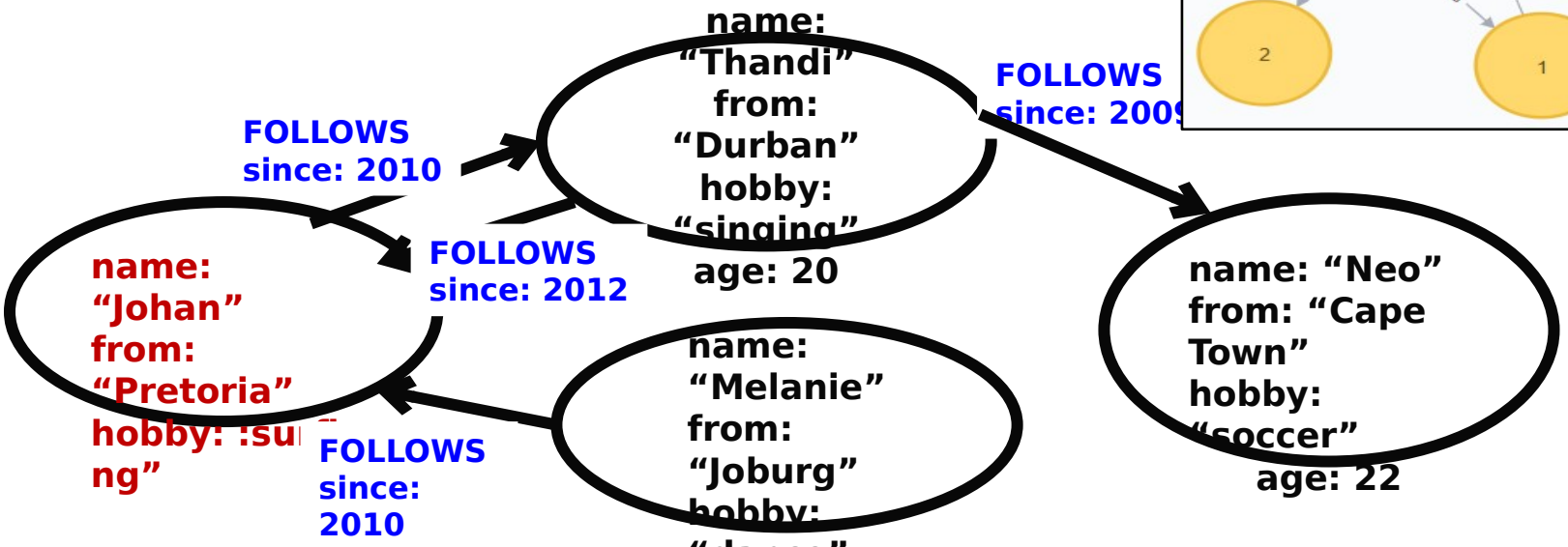


CRUD operations: patterns & paths (2b)

```
MATCH p = shortestPath (
    (psJ:Person {name:"Johan"})-[*]-
     (psN:Person {name:"Neo"})  )
RETURN  p
```

Node id	Person
0	Thandi
1	Johan
2	Neo





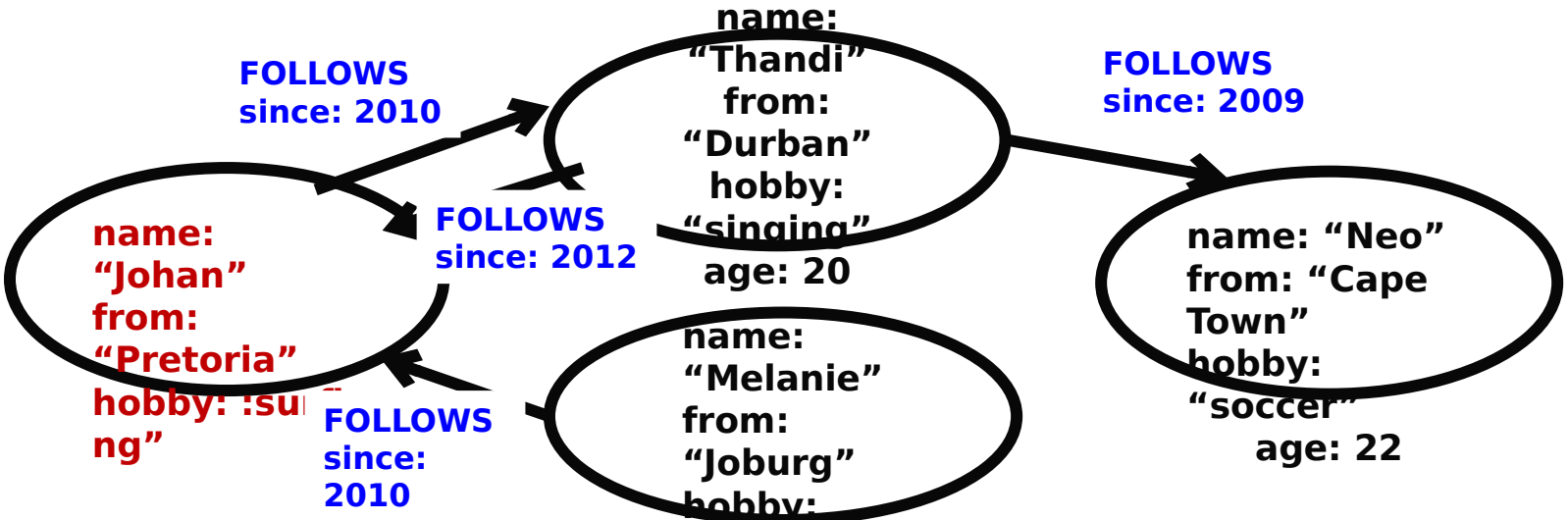
CRUD operations: patterns & paths (3)

variable length patterns for paths:

e.g. shortest path from Johan to Neo (limit search to a max of 5):

```
MATCH p = shortestPath (
    (psJ:Person {name:"Johan"})-[*
RETURN psJ.name, psN.name, length(p)
```

psJ.name	psN.name	length(p)
"Johan"	"Neo"	2



Graph algorithms

- **Neo4j graph algorithms**
- **Reference:** https://neo4j.com/docs/graph-data-science/current/algorithms/ **is a component that contains Neo4j implementations of some common algorithms for graphs. It includes algorithms like:**
    - **Shortest path,**
    - **all paths,**
    - **all simple paths,**
    - **Dijkstra**
    - **A***

Reading for the very keen student

Importing data into a Neo4j database (1)

Refer to Neo4j documentation:
https://neo4j.com/docs/getting-started/cypher-intro/load-csv/

Importing CSV files with Cypher

- Tutorial shows how to import data from CSV files **using LOAD CSV.** **(& gives URLs for the csv files)**

- **given three CSV files:**
  - **persons.csv** **a list of persons**
  - **movies.csv** **a list of movies:**
  - **roles.csv** **a list of the roles played by some of the persons in each movie.**

- **CSV files:**
  - Stored on the database server **are accessible using a file:/// URL** **or HTTPS, HTTP, and FTP.**

- **For Neo4j v1.60:** **csv files must be in folder 'import' under the active DB's folder**

Importing data into a Neo4j database (2)

importing persons data: *persons.csv*

Sample data: **id, name**

1, Charlie Sheen
2, Oliver Stone
3, Michael Douglas
4, Martin Sheen
5, Morgan Freeman

```
LOAD CSV WITH HEADERS FROM
        "file:///persons.csv"   AS csvLine
CREATE (p:Person    { id: toInt(csvLine.id),
            name:     csvLine.name })
```

Importing data into a Neo4j database (3)

importing movies data:*movies.csv*

Sample data:

id, title, country, year

1, Wall Street, USA, 1987
2, The American President, USA, 1995
3, The Shawshank Redemption, USA, 1994

```
CREATE INDEX ON :Country(name)    // to speed up execution
LOAD CSV WITH HEADERS FROM
            "file:///movies.csv" AS csvLine
MERGE (country:Country { name: csvLine.country })
    CREATE  (movie:  Movie   { id: toInt(csvLine.id),
                title:   csvLine.title,  year:toInt(csvLine.year)})
    CREATE (movie)-[:MADE_IN]->(country)
```

Importing data into a Neo4j database (4)

importing roles data:*roles.csv*

data:	**personId,**	**movieId,**	**role**
	1,	1,	Bud Fox
	4,	1,	Carl Fox
	2,	1,	Gordon Gekko
	4,	2,	A.J. MacInerney

```
CREATE CONSTRAINT ON (person:Person) ASSERT person.id IS UNIQUE
CREATE CONSTRAINT ON (movie:Movie) ASSERT movie.id IS UNIQUE

USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM
        "file:///roles.csv" AS csvLine
MATCH   (person:Person { id: toInt(csvLine.personId)}),
        (movie:Movie { id: toInt(csvLine.movieId)})
CREATE  (person)-[:PLAYED { role: csvLine.role }]->(movie)
```

HOMEWORK (1)

What is the difference in meaning between the **patterns** in the following **MATCH clauses**?

- **MATCH (psn:Person)-->(psm)**
- **MATCH (psn:Person)-[:FOLLOWS]-(followed)**
- **MATCH (psn:Person)-[:FOLLOWS]->(followed)**
- **MATCH (psn:Person)<-[:FOLLOWS]-**

Data models for graph databases

In the property graph model:

- information is modeled using three basic building blocks:
  1. **node (a.k.a. vertex)**
  2. **relationship (edge, with direction & type. labeled & directed)**
  3. **property (a.k.a attribute) on nodes and relationships**

More specifically:

- the graph model is a labeled and directed multigraph.
  - a labeled graph has a label for each edge (type for edge)
  - a directed graph uses edges with fixed direction
  - a multigraph allows multiple edges between two nodes

Many graph algorithms (from graph theory) are available

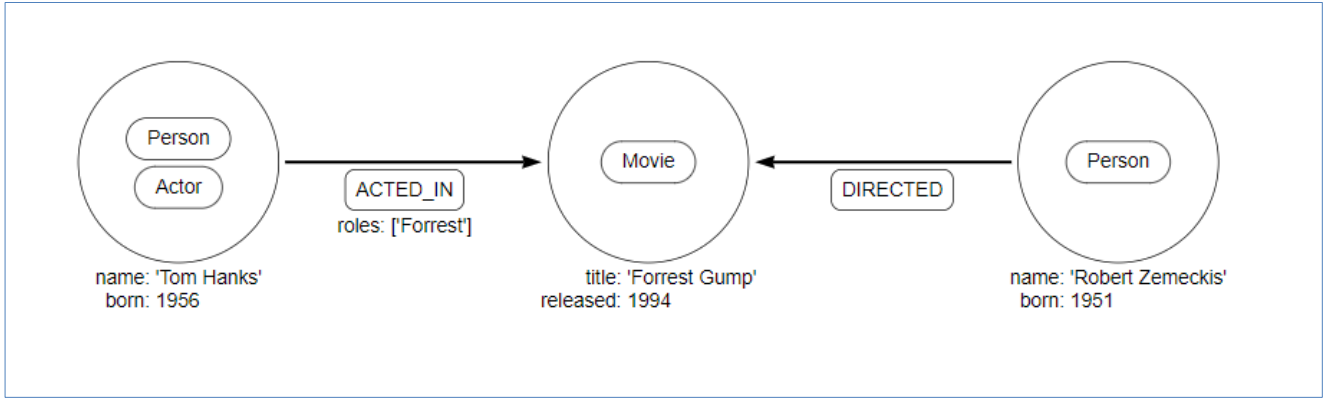
- e.g shortest path, closeness, betweenness, etc.

Neo4j uses the property graph model

Neo4j graph data modelling (1)

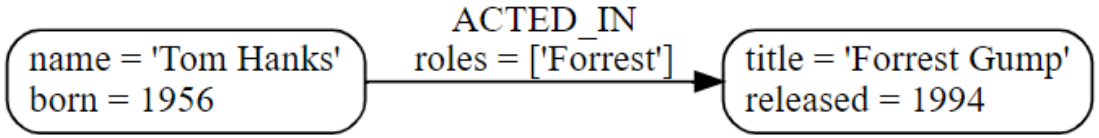
Reference:
https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/

The example graph shown below, introduces the basic concepts of the property graph:



Neo4j graph data modeling (2)

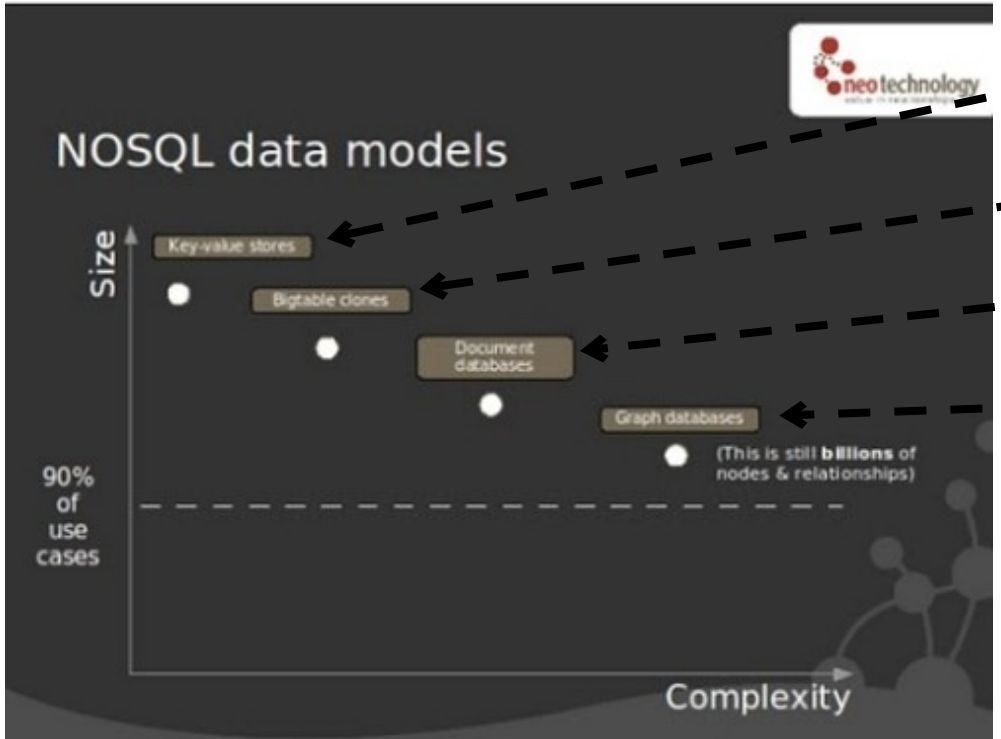
- **Example of entities & relationships in the graph database**
- **Reference: https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/**



- **Observe that the Tom Hanks node has an outgoing relationship, while the Forrest Gump node has an incoming relationship.**

NoSQL data models

Diagram for the positioning of the main NoSQL categories in terms of complexity & scalability (Neo technology):



Key-value stores

Bigtable stores

Document DBs

Graph DBs (billions of nodes & relationships)

HOMEWORK (2)

“Weaknesses” (limitations) of Relational DB systems

(1) Normalisation leads to relations that do not correspond to entities in the “real world”. Information is obtained from the DB through performing a

(2) Semantic overloading: **No mechanism to distinguish** between **relations & relationships** or between different **types of relationships**

(3) Schema changes are difficult: Database change requires program changes

(4) Designed for content-based associative access

- Poor for navigational access: Movement between records is not supported (except through stored procedures)

How does Neo4j avoid these weaknesses?

COS 326
Database Systems

Lecture 20

New SQL Databases

Ref: Notes

Thursday 10 October 2024

In this lecture

1. Introduction
2. NewSQL Database Features
3. Differences between NewSQL, NoSQL and SQL
4. Examples of NewSQL Databases

Introduction

- **NewSQL is a type of database language that incorporates and builds on the concepts and principles of Structured Query Language (SQL) and NoSQL languages. By combining the reliability of SQL with the speed and performance of NoSQL, NewSQL provides improved functionality and services.**
- **NewSQL databases attempt to provide enterprises with the top benefits of both relational and NoSQL databases on one platform.**
- **NewSQL is a unique database system that combines ACID compliance with horizontal scaling. The database system strives to keep the best of both worlds. OLTP-based transactions and the high performance of NoSQL combine in a single solution.**

NewSQL Database Features (1)

- In-memory storage and data processing supply fast query results.
- Partitioning scales the database into units. Queries execute on many shards and combine into a single result.
- ACID properties preserve the features of RDBMS.
- Secondary indexing results in faster query processing and information retrieval.
- High availability due to the database replication mechanism.
- A built-in crash recovery mechanism delivers fault tolerance and minimizes downtime

NewSQL Database Features (2)

- NewSQL databases can handle complex data
- NewSQL databases are highly scalable
- Enhanced concurrency control system benefits traditional ones.
- Presence of a secondary index allowing it to support faster query processing times.
- Configure its systems to provide synchronous updates of data over the WAN.
- Minimizes Downtime, provides fault tolerance with its crash recovery mechanism.

Differences in relation to SQL and NoSQL

Feature	SQL	NoSQL	NewSQL
Schema	Relational (table)	Schema-free	Both
SQL	Yes	Depends on the system	Yes, with enhanced features
ACID	Yes	No (BASE)	Yes
OLTP	Partial support	Not supported	Full support
Scaling	Vertical	Horizontal	Horizontal
Distributed	No	Yes	Yes
High availability	Custom	Auto	Built-in
	Low	High complexity	

Examples of NewSQL Databases

1. VoltDB

- Is a database used high-speed transactional applications. The database performs in-memory processing on a distributed architecture. The software is available as both open source and proprietary.

Key features of VoltDB:

- Real-time decision-making.
- Support for Kafka import and export.
- Disaster recovery through database replication.
- Hadoop and OLAP export integration

Examples of NewSQL Databases

2. CockroachDB

- CockroachDB is a scalable and robust database. CockroachDB offers strong data consistency and works well with low-latency resources.

Key features:

- Robust disaster recovery system.
- Historical data view, record, and storage options.
- Built-in cleaning processes for disks and storage devices.
- CockroachDB works in unfavorable conditions.

Examples of NewSQL Databases

3. NuoDB

- **NuoDB is a geo-distributed database with flexible scaling for various geographic locations. The database maps data across various points while staying ACID compliant.**

Key features:

- **High-quality data transformations.**
- **Always available with online schema evolutions and rolling upgrades.**
- **Tailored features for data storage and control.**
- **Full ACID transaction support.**

Examples of NewSQL Databases

4. ClustrixDB

- ClustrixDB is a self-managing NewSQL database. The software automates scaling operations and supports high availability.

Key features:

- Efficient data categorisation.
- SQL code migration options.
- Built-in health metrics in a browser interface.
- DevOps assistance and query caching.

Examples of NewSQL Databases

5. Altibase

- **Altibase is an in-memory database with a hybrid architecture. The database reduces hardware and software costs by combining in-memory data processing with an on-disk DBMS with a single license. Altibase comes in both community and proprietary versions.**

Key Features

- **Memory-optimized engine for increased speeds.**
- **Custom persistence and performance balance levels.**
- **Flexible deployment options.**
- **Real-time access to vital data.**

Homework

- **Outline the advantages and disadvantages of NewSQL databases**
- **How do NewSQL databases overcome the limitations of SQL and NoSQL Databases?**

COS 326
Database Systems

Lecture 21

Big Data Technologies (1)

Chapter 25

Tuesday 15 October 2024

In this lecture

- Introduction to Big Data
- Sources of Big Data
- MapReduce
- Hadoop Distributed File System
- Hadoop v2 or YARN
  - Generic data services platform

1. **Reading for student: MapReduce: Simplified Data Processing on Large Clusters**

Introduction

- Phenomenal growth in data generation
  - Social media
  - Sensors
  - Communications networks and satellite imagery
  - User-specific business data
- “Big data” refers to massive amounts of data
  - Exceeds the typical reach of a DBMS
- Big data analytics

25.1 What is Big Data?

- Big data ranges from terabytes (10^{12} bytes) or petabytes (10^{15} bytes) to exobytes (10^{18} bytes)
- **Volume**
  - Refers to size of data managed by the system
- **Velocity**
  - Speed of data creation, ingestion, and processing
- **Variety**
  - Refers to type of data source
  - Structured, unstructured

What is Big Data? (cont'd.)

- **Veracity**
  - Credibility of the source
  - Suitability of data for the target audience
  - Evaluated through quality testing or credibility analysis

Three Characteristics of Big Data V3s

Volume

- Data quantity

Velocity

- Data Speed

Variety

- Data Types

1st Characteristic of Big Data

Volume

- **A typical PC might have had 10 gigabytes of storage in 2000.**

- **Today, Facebook ingests 500 terabytes of new data every day.**

- **Boeing 737 will generate 240 terabytes of flight data during a single flight across the US.**

- **The smart phones, the data they create and consume; sensors embedded into everyday objects will soon result in billions of new, constantly-updated data feeds**

2nd Characteristic of Big Data

Velocity

- **Clickstreams and ad impressions capture user behavior at millions of events per second**
- **high-frequency stock trading algorithms reflect market changes within microseconds**
- **machine to machine processes exchange data between billions of devices**
- **infrastructure and sensors generate massive log data in real- time**
- **on-line gaming systems support millions of concurrent users, each producing multiple**

3rd Characteristic of Big Data Variety

- **Big Data isn't just numbers, dates, and strings. Big Data is also geospatial data, 3D data, audio and video, and unstructured text, including log files and social media.**

- **Traditional database systems were designed to address smaller volumes of structured data, fewer updates or a predictable, consistent data structure.**

- **Big Data analysis includes different types of**

Leading Technology Vendors

- Example Vendors

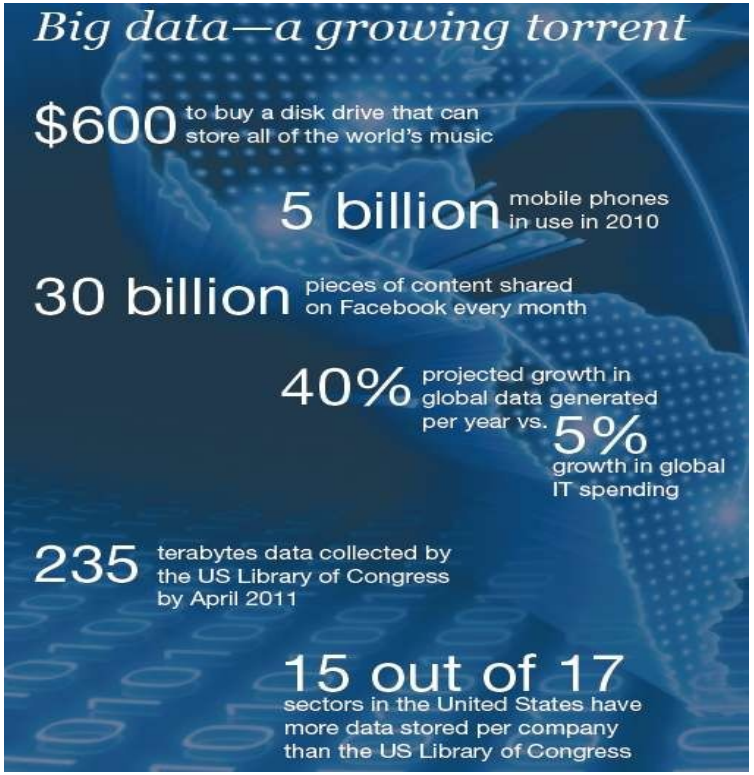
- Amazon – DynamoDB
- Microsoft - Azure Cosmos DB
- IBM – Netezza
- EMC – Greenplum
- Oracle – Exadata

Commonality

- MPP architectures
- Commodity Hardware
- RDBMS based
- Full SQL compliance

Data generation points examples

- Mobile Devices
- Microphones
- Readers/Scanner
- s Science
- facilities
- Programs/
- Software Social



Application Of Big Data analytics

Smarter Healthcare

Multi-channel sales

Homeland Security

Telecom

Traffic Control

Trading Analytics

Manufacturing

Search Quality

25.2 Introduction to MapReduce and Hadoop

- Core components of Hadoop
  - MapReduce programming paradigm
  - Hadoop Distributed File System (HDFS)
- Hadoop originated from quest for open source search engine
  - Developed by Cutting and Carafella in 2004
  - Cutting joined Yahoo in 2006
  - Yahoo spun off Hadoop-centered company in 2011
  - Tremendous growth

Introduction to MapReduce and Hadoop (cont'd.)

- **MapReduce**
  - Fault-tolerant implementation and runtime environment
  - Developed by Dean and Ghemawat at Google in 2004
  - Programming style: map and reduce tasks
    - Automatically parallelized and executed on large clusters of commodity hardware
  - Allows programmers to analyze very large datasets
  - Underlying data model assumed: key-value pair

The MapReduce Programming Model

- **Map**
  - Generic function that takes a key of type K1 and value of type V1
  - Returns a list of key-value pairs of type K2 and V2
- **Reduce**
  - Generic function that takes a key of type K2 and a list of values V2 and returns pairs of type (K3, V3)
- Outputs from the map function must match the input type of the reduce function

The MapReduce Programming Model (cont'd.)

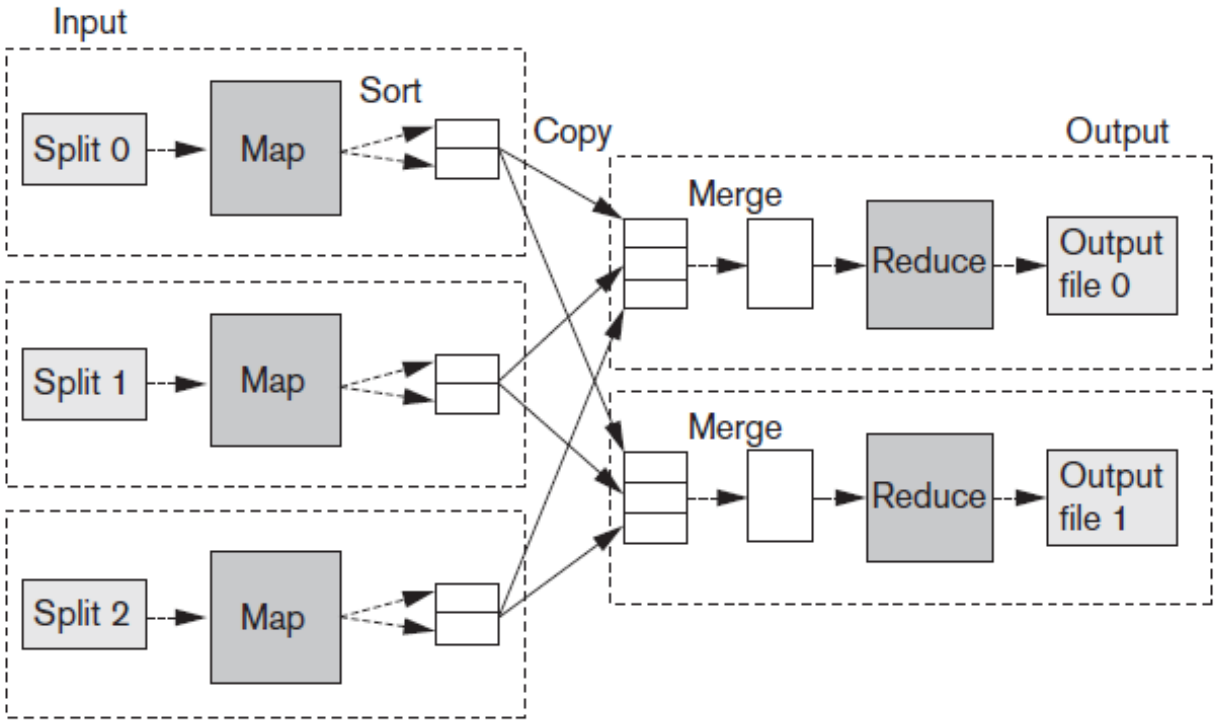


Figure 25.1 Overview of MapReduce execution (Adapted from T. White, 2012)

The MapReduce Programming Model (cont'd.)

- MapReduce example
  - Make a list of frequencies of words in a document
  - Pseudocode

```
Map (String key, String value):
    for each word w in value Emitintermediate (w, "1");

Reduce (String key, Iterator values) : // here the key is a word and values are
lists of its counts //
    Int result =0;
    For each v in values :
        result += Parseint (v);
    Emit (key, Asstring (result));
```

The MapReduce Programming Model (cont'd.)

- MapReduce example (cont'd.)
  - Actual MapReduce code

```
map[LongWritable,Text](key, value) : List[Text, LongWritable] = {
    String[] words = split(value)
    for(word : words) {
        context.out(Text(word), LongWritable(1))
    }
}
reduce[Text, Iterable[LongWritable]](key, values) : List[Text, LongWritable] = {
    LongWritable c = 0
    for( v : values) {
        c += v
    }
    context.out(key,c)
    }
```

25.3 Hadoop Distributed File System (HDFS)

- **HDFS**
  - File system component of Hadoop
  - Designed to run on a cluster of commodity hardware
  - Patterned after UNIX file system
  - Provides high-throughput access to large datasets
  - Stores metadata on NameNode server
  - Stores application data on DataNode servers
    - File content replicated on multiple DataNodes

Hadoop Distributed File System (cont'd.)

- HDFS design assumptions and goals
  - Hardware failure is the norm
  - Batch processing
  - Large datasets
  - Simple coherency model
- HDFS architecture
  - Master-slave
  - Decouples metadata from data operations
  - Replication provides reliability and high availability
  - Network traffic minimized

Hadoop Distributed File System (cont'd.)

- **NameNode**
  - Maintains image of the file system
    - i-nodes and corresponding block locations
  - Changes maintained in write-ahead commit log called Journal
- **Secondary NameNodes**
  - Checkpointing role or backup role
- **DataNodes**
  - Stores blocks in node's native file system
  - Periodically reports state to the NameNode

Hadoop Distributed File System (cont'd.)

- **File I/O operations**
  - Single-writer, multiple-reader model
  - Files cannot be updated, only appended
  - Write pipeline set up to minimize network utilization
- **Block placement**
  - Nodes of Hadoop cluster typically spread across many racks
    - Nodes on a rack share a switch

Hadoop Distributed File System (cont'd.)

- **Replica management**
  - NameNode tracks number of replicas and block location
    - Based on block reports
  - Replication priority queue contains blocks that need to be replicated
- **HDFS scalability**
  - Yahoo cluster achieved 14 petabytes, 4000 nodes, 15k clients, and 600 million files

The Hadoop Ecosystem

- Related projects with additional functionality
  - **Pig and hive**
    - Provides higher-level interface for working with Hadoop framework
  - **Oozie**
    - Service for scheduling and running workflows of jobs
  - **Sqoop**
    - Library and runtime environment for efficiently moving data between relational databases and HDFS

The Hadoop Ecosystem (cont'd.)

- Related projects with additional functionality (cont'd.)
  - HBase
    - Column-oriented key-value store that uses HDFS

25.4 MapReduce: Additional Details

- MapReduce runtime environment
  - **JobTracker**
    - Master process
    - Responsible for managing the life cycle of Jobs and scheduling Tasks on the cluster
  - **TaskTracker**
    - Slave process
    - Runs on all Worker nodes of the cluster

MapReduce: Additional Details (cont'd.)

- Overall flow of a MapReduce job
  - Job submission
  - Job initialization
  - Task assignment
  - Task execution
  - Job completion

MapReduce: Additional Details (cont'd.)

- Fault tolerance in MapReduce
  - Task failure
    - Runtime exception
    - Java virtual machine crash
    - No timely updates from the task process
  - TaskTracker failure
    - Crash or disconnection from JobTracker
    - Failed Tasks are rescheduled
  - JobTracker failure
    - Not a recoverable failure in Hadoop v1

MapReduce: Additional Details (cont'd.)

- The shuffle procedure
  - Reducers get all the rows for a given key together
  - Map phase
    - Background thread partitions buffered rows based on the number of Reducers in the job and the Partitioner
    - Rows sorted on key values
    - Comparator or Combiner may be used
  - Copy phase
  - Reduce phase

MapReduce: Additional Details (cont'd.)

- Job scheduling
  - JobTracker schedules work on cluster nodes
  - Fair Scheduler
    - Provides fast response time to small jobs in a Hadoop shared cluster
  - Capacity Scheduler
    - Geared to meet needs of large enterprise customers

MapReduce: Additional Details (cont'd.)

- Strategies for equi-joins in MapReduce environment
  - Sort-merge join
  - Map-side hash join
  - Partition join
  - Bucket joins
  - N-way map-side joins
  - Simple N-way joins

MapReduce: Additional Details (cont'd.)

- **Apache Pig**
  - Bridges the gap between declarative-style interfaces such as SQL, and rigid style required by MapReduce
  - Designed to solve problems such as ad hoc analyses of Web logs and clickstreams
  - Accommodates user-defined functions

MapReduce: Additional Details (cont'd.)

- **Apache Hive**
  - Provides a higher-level interface to Hadoop using SQL-like queries
  - Supports processing of aggregate analytical queries typical of data warehouses
  - Developed at Facebook

Hive System Architecture and Components

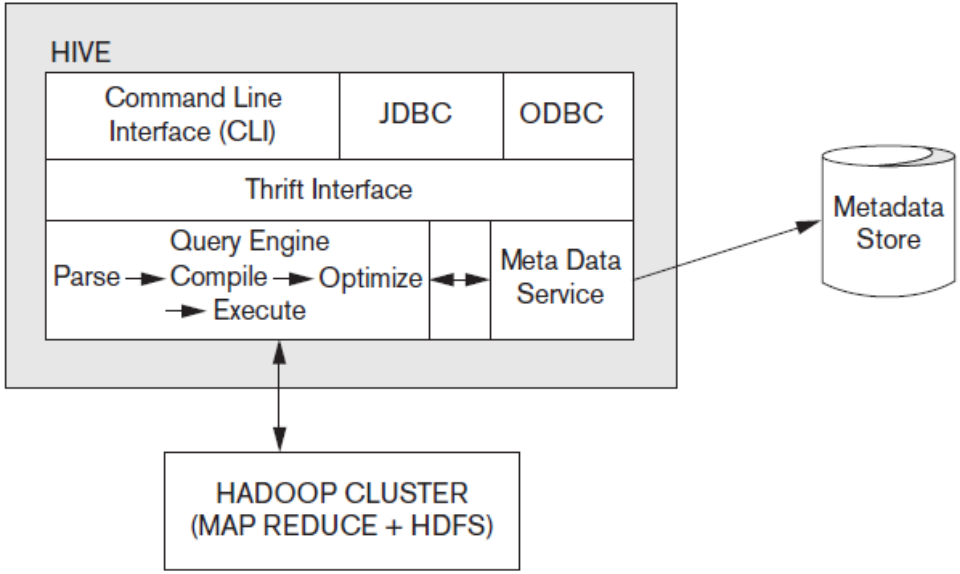


Figure 25.2 Hive system architecture and components

Advantages of the Hadoop/MapReduce Technology

- Disk seek rate a limiting factor when dealing with very large data sets
  - Limited by disk mechanical structure
- Transfer speed is an electronic feature and increasing steadily
- MapReduce processes large datasets in parallel
- MapReduce handles semistructured data and key-value datasets more easily
- Linear scalability

25.5 Hadoop v2 (Alias YARN)

- Reasons for developing Hadoop v2
  - JobTracker became a bottleneck
  - Cluster utilization less than desirable
  - Different types of applications did not fit into the MR model
  - Difficult to keep up with new open source versions of Hadoop

YARN Architecture

- YARN stands for “Yet Another Resource Negotiator“
- Separates cluster resource management from Jobs management
- ResourceManager and NodeManager together form a platform for hosting any application on YARN
- ApplicationMasters send ResourceRequests to the ResourceManager which then responds with cluster Container leases
- NodeManager responsible for managing Containers on their nodes

Hadoop Version Schematics

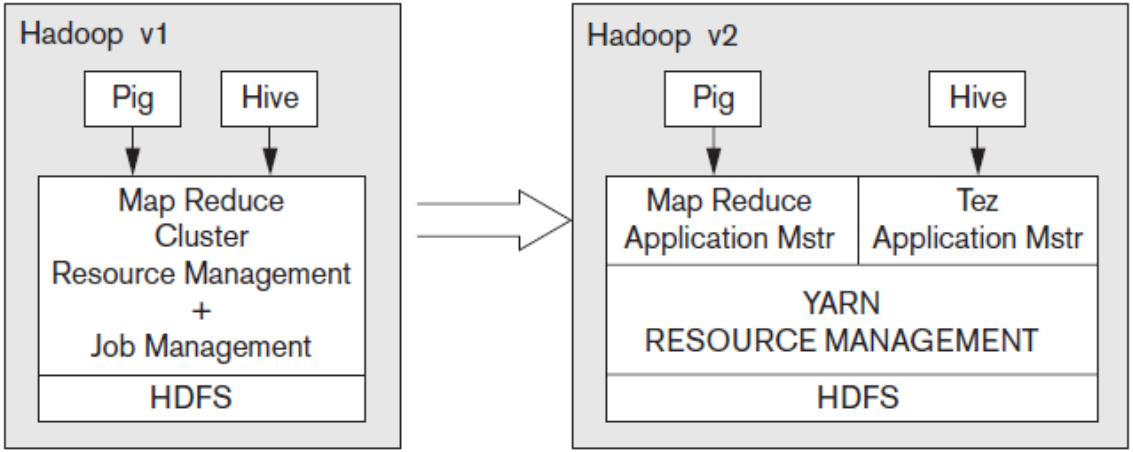


Figure 25.3 The Hadoop v1 vs. Hadoop v2 schematic

Other Frameworks on YARN

- Apache Tez
  - Extensible framework being developed at Hortonworks for building high-performance applications in YARN
- Apache Giraph
  - Open-source implementation of Google’s Pregel system, a large-scale graph processing system used to calculate Page-Rank
- Hoya: HBase on YARN
  - More flexibility and improved cluster utilization

25.7 Summary

- Big data technologies at the center of data analytics and machine learning applications
- MapReduce
- Hadoop Distributed File System
- Hadoop v2 or YARN
  - Generic data services platform

COS 326
Database Systems

Lecture 23

Data warehousing and OLAP

Chapter 29

Tuesday 22 October 2024

In this lecture

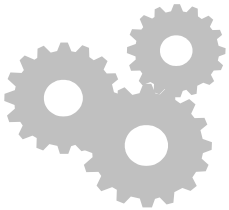
Business Intelligence & data warehousing

1. The business environment
2. How operational systems differ from informational systems
3. Business intelligence
4. The architecture & main components of a data warehouse.
5. The data mart and the main reasons for implementing a data mart.

The business environment

- What is the purpose of a business? To make money
- What do businesses sell? Products or services
- To who do businesses sell? Customers (individuals or organisation)
- From who do businesses get the things that they sell? Suppliers or internal
- How are business operations organised? Business units, business processes
- Who makes decisions in a business? Managers
- What major threats do businesses face? Losing customers, closing down

Operational systems in a business



Operational systems:

- **Purpose:**
  - **Make the wheels of the business turn**
  - **Record & process business transactions as they happen**

- **Examples of systems:**
  - **Point of sale system (POS)**
  - **inventory management system**
  - **Financial systems (e.g. General ledger)**
  - **Insurance claims processing**
- **Used by:**
  - **Shop floor workers (who directly interact with customers)**
- **Classified as:**
  - **Online Transaction Processing (OLTP) systems**

Informational systems in a business

Informational systems

- **Purpose:**
  - Watching the wheels of the business turn
  - support informed decision making
- **Examples of usage**:
  - Show me the top selling products
  - Show me the problem branches
  - Alert me when a branch sells below target
- **Used by:**
  - Managers (and business analysts)
- **Classified as:**
  - Online Analytical Processing (OLAP) systems

Tools: Dashboards: day-to-day management

- **Managing a business is like: Driving a car, Flying a large aircr**

What is Business Intelligence (BI)?

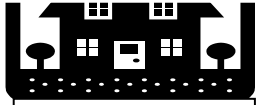
- ***BI* refers to a class of applications and technologies used to:**
  - **gather**
  - **provide access to**
  - **and analyse data about business operations.**

- **Business intelligence systems:**
  enable businesses to have an in-depth knowledge of factors that affect a business eg:
  - **customer, supplier behaviour**
  - **sales volumes, production volumes**
  - **& other internal operations.**

- **This knowledge helps managers to make better business decisions.**

Why is data warehousing a viable solution for informational systems ?

Operational systems



POS system

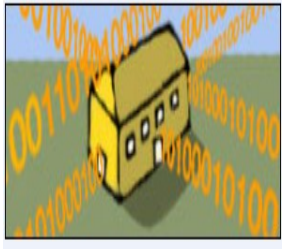
Inventory management

Data warehouse

Key business measures
Business dimensions

- Data from multiple applications
- Content includes current and historical data
- Data designed for analytical tasks

Data warehouse definition



Key business measures,

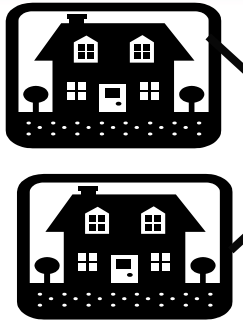
Business dimensions

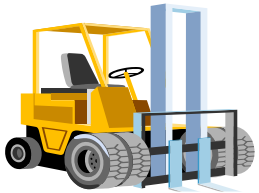
A data warehouse is a

- **subject-oriented**
  - **Data is organised along business subjects**
  - **eg. sales, policies, claims**
- **Integrated**
  - **Data extracted from different operational systems**
  - **eg. Claims processing, policy sales**
- **Non-volatile**
  - **Data loaded at specific time periods**
  - **eg. daily, weekly, monthly**
- **Time-variant**
  - **Time is explicitly stored with data**
  - **Supports analysis along**
    - **e.g. sales for day, week, month, quarter, year**

Collection of data to support management decisions

The data warehousing environment





2. Data staging
Extraction, Transformation & Loading (ETL)

3.Enterprise Data warehouse

4. Metadata

Data mart

Data mart

1. Source data

Internal - on user PCs

External - from agoperational systems

encies

6. Information delivery

7. Interactive users

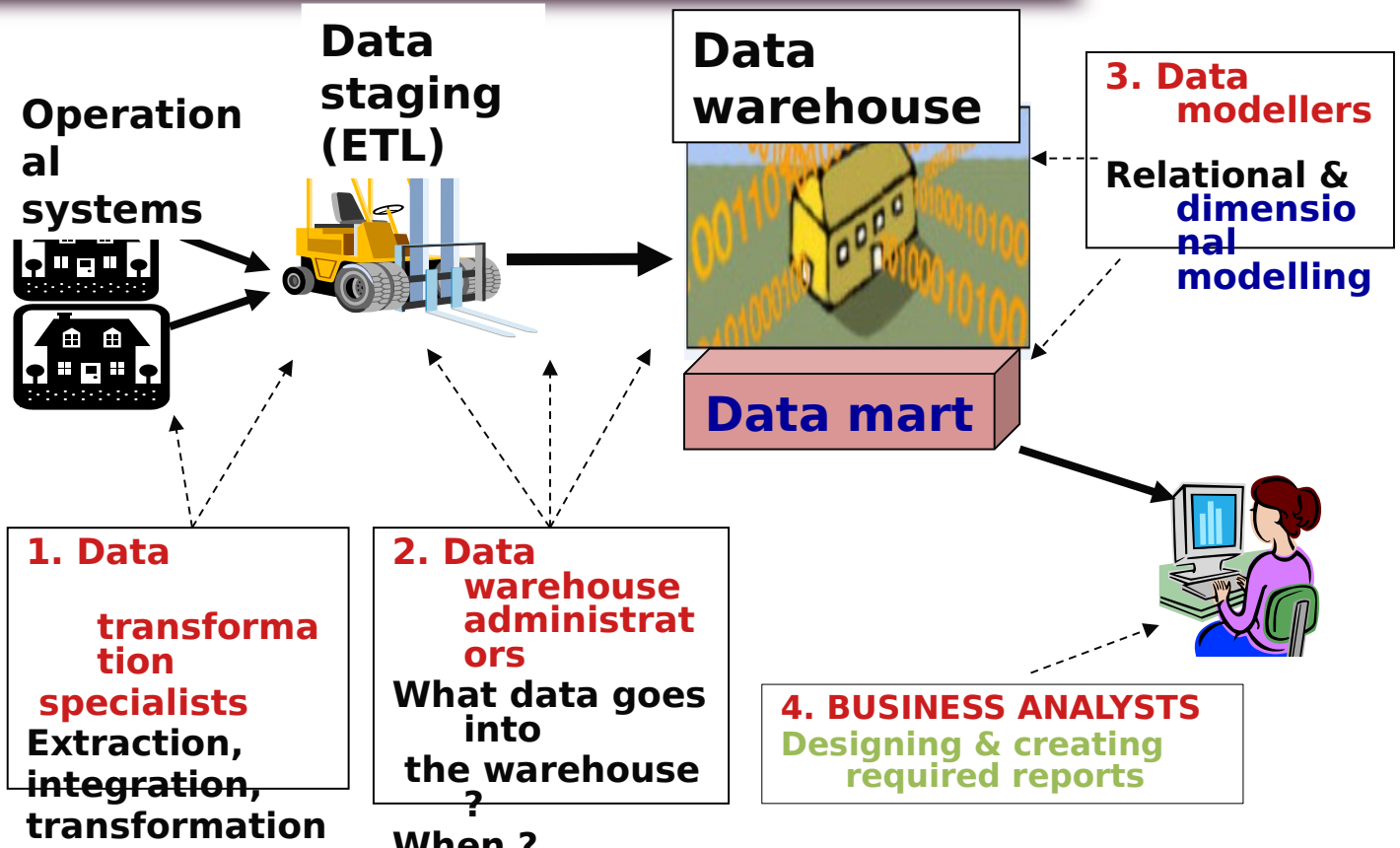
Data minin OLAP analysis

8. Management & control

9. (Hardware, OS, DBMS)

10. DB personnel

Data warehousing roles & responsibilities



Dimensional modeling

Examples of business dimensions:

Business: **Supermarket**
Subject: **Sales**
Dimensions: **Time, branch, product, promotion**

Business: **Airline company**
Subject: **Frequent flyers**
Dimensions: **Time, customer, flight, fare class, airport**

Business: **Hotel chain**
Subject: **Hotel occupancy**
Dimensions: **Time, hotel branch, room type**

=>Dimensions & measures will be different for different businesses, but TIME dimension is always used

Information from a data warehouse

Sales Director

Example question that managers/ users ask when analysing business performance:

eg. Sales Director for a hotel chain

How much revenue did luxury rooms generate month by month in the Gauteng province by hotel branch ?

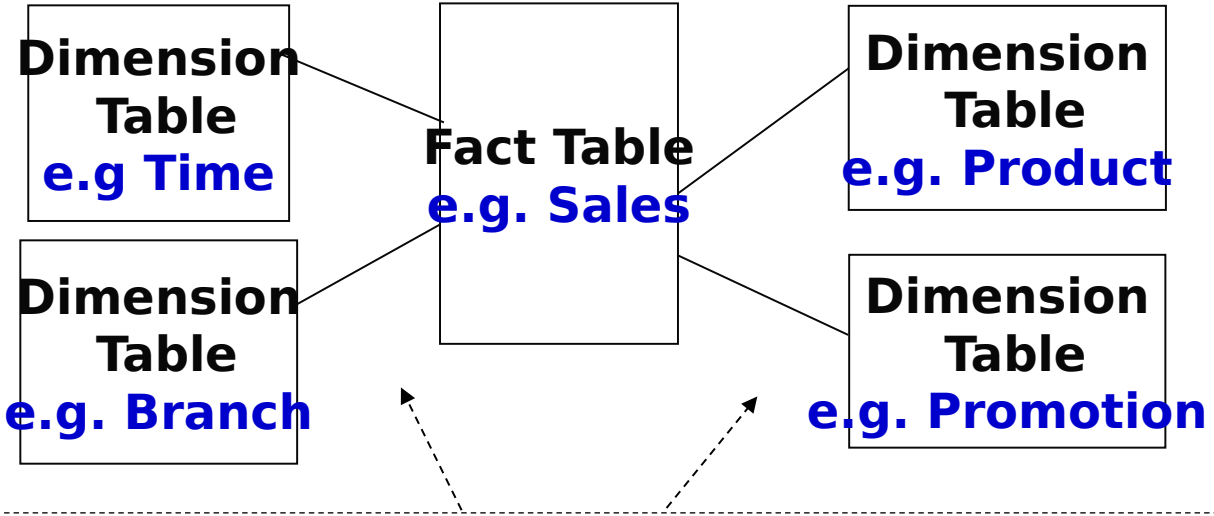
⇒ Users analyse business performance based on business **dimensions** & performance **measures (e.g. revenue)**

⇒ **Need for a database structure that supports dimensional analysis**

The Star Schema (for data mart)

- Provides Relational OLAP (ROLAP)
- Dimensional view of data using Relational structures & queries

Links are formed by foreign keys in the fact table
One-to-many relationship



Each row of the fact table has foreign key values from the dimension tables & one value for a

Information packages & their use in defining business requirements

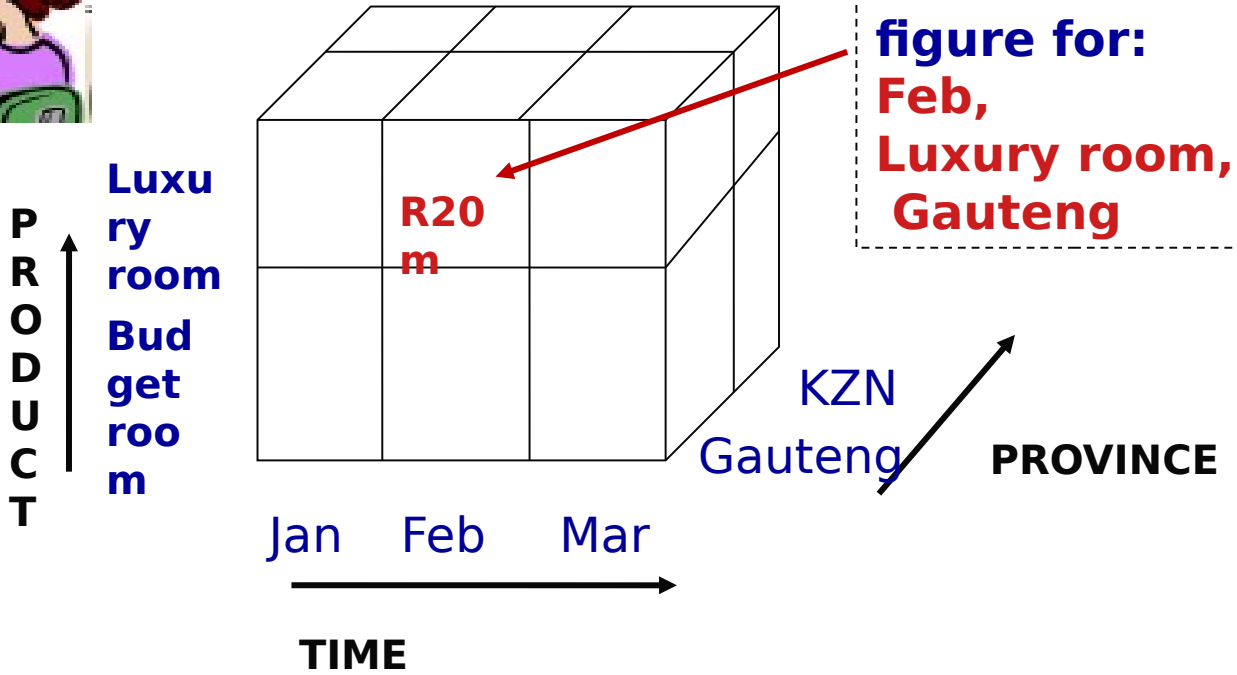
- Create information packages as requirements are gathered
- One information package per specific subject
- Eg. Subject: Hotel occupancy

Dimensions

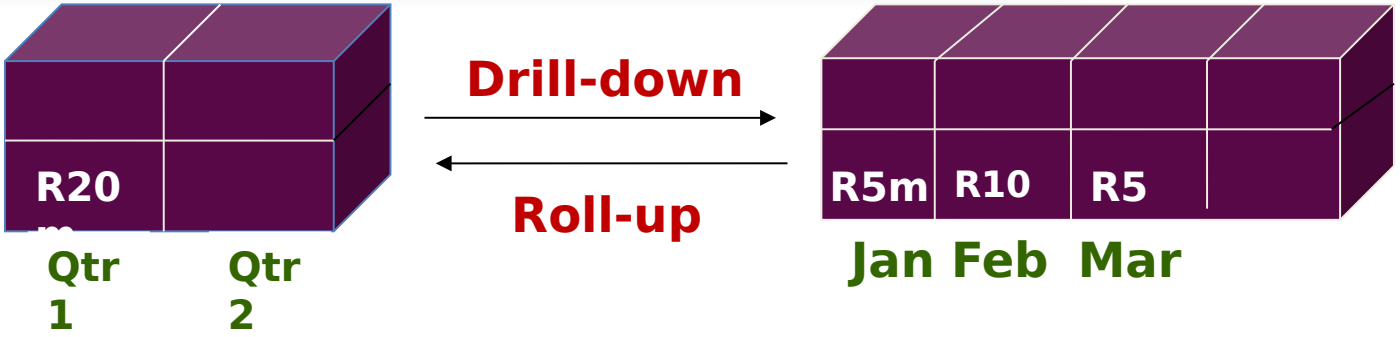
TIME	HOTEL	ROOM TYPE
Year	Hotel line	Room type
Quarter	province	Size
Month	Branch	etc
Facts (key measures): occupied rooms, vacant rooms, Revenue		

Hierarchies or Categories

Dimensional view of data (hyper-cube)



Dimension Hierarchies & Categories



TIME	HOTEL	ROOM TYPE
Year	Hotel line	Room type
Quarter	Province	Size
Month	Branch	etc
Facts (key measures): occupied rooms, vacant rooms, Revenue		

Hierarchies:

- used to perform
  - Drill-down operations
  - Roll-up operations

Creating the Dimension Tables & Fact table

Dimension table

HOTEL

Hotel_Line
Province
Branch

TIME	HOTEL	ROOM TYPE
Year	Hotel line	Room type
Quarter	Province	Size
Month	Branch	etc
Facts (key measures): occupied rooms, vacant rooms, Revenue		

Fact Table

OCCUPANCY_FACTS

Occupied_rooms
Vacant_rooms
Revenue

STARjoin, STARindex & materialised views

SQL statements are executed to provide answers to users' questions

Eg. How much revenue was realised from luxury rooms by province by month ?

STARjoin

High-speed, single-pass, parallisable, multitable join

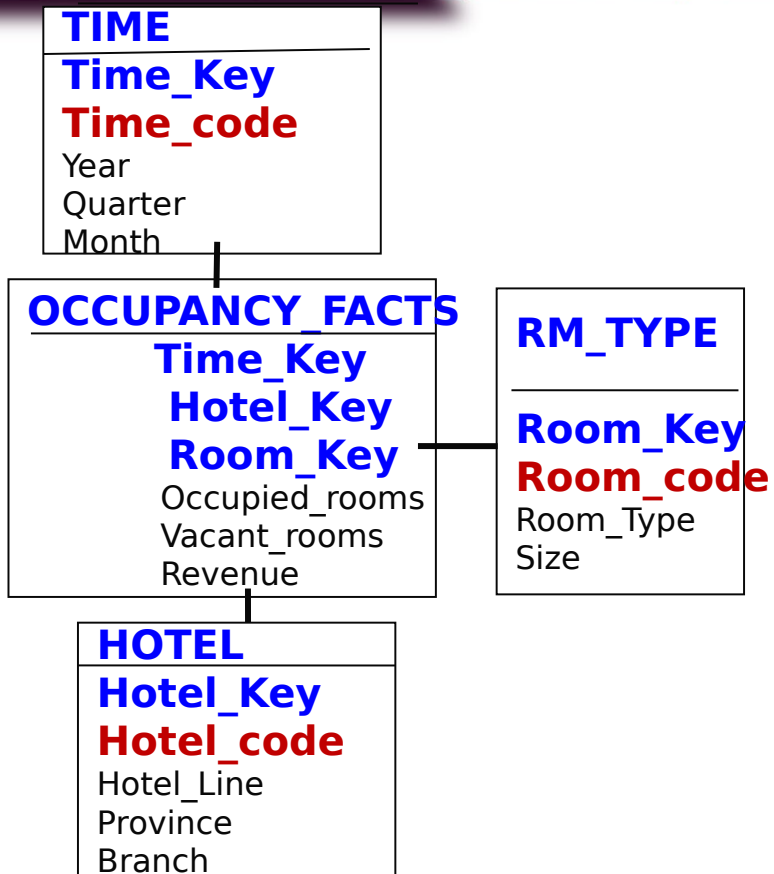
STARindexes

specialised indexes to speed up joins between the dimension and fact tables

(eg Amazon.com fact table: 10 billion rows)

Materialised views:

store results of commonly used queries



SQL statements for data warehousing & BI

- **Creating the ROLAP structure**
  - CREATE SEQUENCE
  - CREATE TABLE
  - **CREATE DIMENSION**

- **Loading data into the ROLAP structure**
  - SQL MERGE statement
  - UPDATE / INSERT statement

- **Generating reports from the ROLAP structure**
  - SELECT … GROUP BY CUBE
  - SELECT … GROUP BY ROLLUP
  - SELECT .. GROUP BY GROUPING SETS

CREATE DIMENSION STATEMENT

e.g.
hotel dimension

HOTEL
Hotelline_code Hotelline_desc
Province_code Province_name
Hotel_key Hotel_code Hotel_name Hotel_addres

1. CREATE TABLE HOTEL .....;

then

```
2  CREATE DIMENSION HOTELS_DIM
     LEVEL Hotel  IS HOTEL.Hotel_code
     LEVEL Province IS HOTEL.Province_code
     LEVEL Hotelline IS HOTEL.Hotelline_code

   HIERARCHY Hotel_Rollup
     (Hotel      CHILD OF
      Province CHILD OF
      Hotelline)

 ATTRIBUTE Hotel DETERMINES
         (Hotel_name, Hotel_address, Hotel_city......)
ATTRIBUTE  Province DETERMINES
     Province_name
ATTRIBUTE Hotelline DETERMINES Hotelline_desc
```

COS 326
Database Systems

Lecture 24

Data warehousing and OLAP (2)

Chapter 29

Thursday 24 October 2024

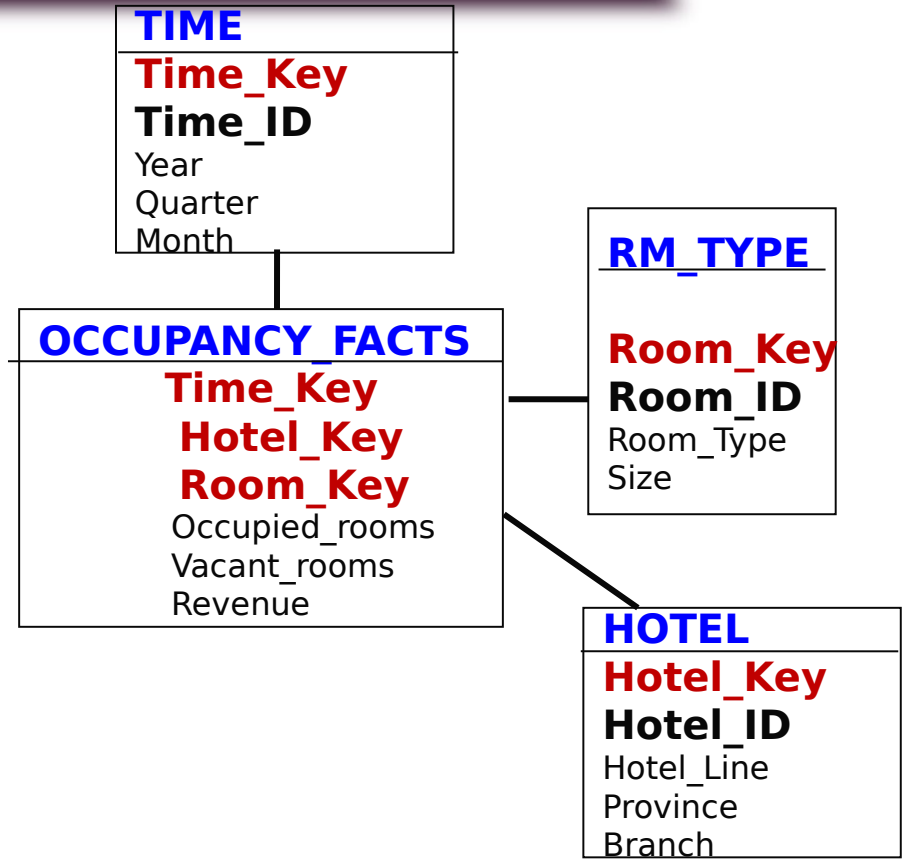
In this lecture

Data warehousing & OLAP

- Implementation of a ROLAP schema
- OLAP extensions to the SQL standard.
- How Oracle supports OLAP.

Recap on ROLAP: Primary, Foreign & Surrogate keys

1. **Dimension tables & fact table for a data mart**
2. **Hierarchies:** a dimension may have > 1 hierarchy. **e.g. TIME** may have ***calendar_rollup*** and ***fiscal_rollup***
3. **Use of relational structure to implement dimensional schema for OLAP (ROLAP)**
4. Use of **Surrogate keys** generated by the system



SQL statements for data warehousing & OLAP

- **Creating the ROLAP structure**
  - **CREATE SEQUENCE** (for surrogate keys)
  - **CREATE TABLE** (for dims & fact tables)
  - **CREATE DIMENSION** (for dimensions)
  - **CREATE MATERIALIZED VIEW** (for speed)

- **Loading data into the ROLAP structure**

  - **SQL MERGE statement**
  - **UPDATE / INSERT statement**

- **Generating reports from the ROLAP structure**
  - **SELECT … GROUP BY CUBE**
  - **SELECT … GROUP BY ROLLUP**
  - **SELECT … GROUP BY GROUPING SETS**

CREATE DIMENSION STATEMENT

e.g.
hotel dimension

1. CREATE TABLE HOTEL .....;
then

2. CREATE DIMENSION HOTELS_DIM
    LEVEL Hotel IS HOTEL.Hotel_ID
    LEVEL Province IS HOTEL.Province_ID
    LEVEL Hotelline IS HOTEL.Hotelline_ID

  HIERARCHY Hotel_Rollup
    (Hotel CHILD OF
     Province CHILD OF
     Hotelline)

ATTRIBUTE Hotel DETERMINES
    (Hotel_name, Hotel_address, Hotel_city.....)
ATTRIBUTE Province DETERMINES Province_name
ATTRIBUTE Hotelline DETERMINES Hotelline_desc

Slowly changing dimensions

A data warehouse stores historical data for a long period.

So, data stored in the dimension tables may change over time, e.g. marital status of a customer.

=> Need a strategy to handle the changes

CUSTOMER
Cust_Key
Cust_ID
Cust_sname
Marital_stat
etc

- **Type 1 change:**
  the changed attribute value is over-written
- **Type 2 change:**
  changed attribute value causes a new record to be created.
- **Type 3 change:**
  changed attribute value causes an alternate attribute to be created, so that old & new values are in same record

Loading data into the ROLAP structure

- Make a distinction between
  - **Initial data load & subsequent incremental loads**
- **Slowly changing dimensions:**
- **Type 1 change: overwrite prev. value**
- **Type 2 change: create a new entry**
- **Type 3 change: major change to table**
- **Dimension table design should support type 2 changes**
- **e.g. in CUSTOMER table:**

CUSTOMER

Cust_Key
Cust_ID
Eff_start_date
Eff_end_date
Cust_sname
Marital_stat
etc.

Cust_Key	Cust_ID	Start_date	End_date	Marital_stat	Cust_sname
12345	850002	01/03/2015	09/03/2018	single	Jones
12347	850002	10/03/2018		married	Charming

The SQL MERGE statement

```
//used for initial load & subsequent incremental loads

MERGE INTO CUSTOMER C   --dimension table
      USING  SELECT Cust_ID,.....
                FROM CUST_STG) S  --staging table
      ON ( (C.Cust_ID = S.Cust_ID)  AND
            (C.Marital_Status = S.Marital_status))

      WHEN MATCHED THEN
         // a type 1 change
                     UPDATE  SET (what should change in C)

      WHEN NOT MATCHED THEN
        //a type 2 change on existing customer or a new customer
         INSERT VALUES (S.Cust_ID,  S.Marital_stat...);
```

OLAP Analysis for BI (1)

- A limitation of SQL is inability to answer routinely asked business queries, e.g. compute:
  - **percentage change in values between this month and a year ago**
  - **moving averages (time series)**
  - **cumulative sums, other statistical functions.**

- **ANSI adopted a set of OLAP functions** as an extension to SQL to enable these calculations as well as many others that used to be impossible or even impractical within SQL.

- **IBM and Oracle jointly proposed these extensions** early in 1999 and they **now form part of the current SQL standard since SQL: 2008, SQL:2011, and later**

OLAP Analysis for BI (2)

- The extensions are collectively referred to as the **'OLAP package'** and are described as follows:
  - **Extended Grouping capabilities**
    - **extensions to the GROUP BY clause such as the ROLLUP and CUBE functions.**
  - **Extended OLAP operators**
    - Supports a **variety of operations such as ranking and window calculations.**
    - **Windowing allows the calculation of cumulative and moving aggregations** using functions such as **SUM, AVG, MIN, and COUNT over a range of rows.**
    - **Ranking functions** include **cumulative distributions, percent rank, and N-tiles (quartiles, deciles, percentiles)**

Traditional SQL

- Eg. The manager asks: How much revenue did product X generate by province and by month this year?

- **The SQL:**

```
SELECT B.province, T.month, SUM(S.revenue) AS ..
FROM BRANCH B, TIME T, PRODUCT P, SALES S

WHERE P.ProductName = 'Product X'  AND
      T.year = 2024                AND
      B.BranchKey = S.BranchKey    AND
      T.TimeKey   = S.TimeKey      AND
      P.ProductKey = S.ProductKey
GROUP BY B.province, T.month;
```

Groups are formed as:

{province, month, SUM}

SQL Extensions For Aggregation

- Since SQL 2008 extensions: SELECT .. GROUP BY ..
  - SELECT … GROUP BY CUBE
    - Compute aggregations across all possible combinations of grouping attributes
  - SELECT … GROUP BY ROLLUP
    - Compute subtotals along a dimension hierarchy
  - SELECT .. GROUP BY GROUPING SETS
    - Compute subtotals only for specified groupings

SELECT … GROUP BY CUBE

How much revenue did product X generate by province and by month this year?

```
SELECT B.province, T.month, SUM(S.revenue) AS ..
FROM BRANCH B, TIME T, PRODUCT P, SALES S

WHERE P.ProductName = 'Product X' AND
      T.year = 2024                    AND
      B.BranchKey = S.BranchKey     AND
      T.TimeKey    = S.TimeKey      AND
      P.ProductKey = S.ProductKey

GROUP BY CUBE (B.province, T.month);
```

Groups are formed as:

- {province, month, SUM}
- {province, SUM}
- {month, SUM}
- {all, SUM}

SELECT … GROUP BY ROLLUP

How much revenue did product X generate by province and by month this year?

```
SELECT B.province, T.month, SUM(S.revenue) AS ..

FROM BRANCH B, TIME T, PRODUCT P, SALES S
WHERE P.ProductName = 'Product X'    AND
      T.year = 2024                  AND
      B.BranchKey = S.BranchKey      AND
      T.TimeKey    = S.TimeKey       AND
      P.ProductKey = S.ProductKey

GROUP BY ROLLUP (B.province, T.month);
```

Groups are formed as:

{all, SUM}
{province, SUM}
{province, month, SUM}

Rollup

SELECT … GROUP BY GROUPING SETS

How much revenue did product X generate by province and by month this year?

```
SELECT B.province, T.month,  SUM(S.revenue) AS ..

FROM BRANCH B, TIME T, PRODUCT P, SALES S
WHERE P.ProductName = 'Product X' AND
          T.year = 2024                          AND
          B.BranchKey = S.BranchKey    AND
          T.TimeKey    = S.TimeKey         AND
          P.ProductKey = S.ProductKey

GROUP BY GROUPING SETS
          (  (B.province, T.month)   (province)  ( )    );
```

Groups are formed as:

- {province, month, SUM}
- {province, SUM}
- {all, SUM}

OLAP and Statistical analysis (1)

- **Elementary OLAP Operators**
  - Support a variety of operations such as **ranking** and **window** calculations.

- **1. Ranking functions include:**
  - **cumulative distributions,**
  - **percent rank,**
  - **N-tiles (quartiles, deciles, percentiles)**

OLAP and Statistical analysis (2)

- **2. Windowing** allows the calculation of:
  - **cumulative and**
  - **moving aggregations**
    - using functions such as
    - **SUM,**
    - **AVG: moving average**
    - **MIN, COUNT**
    - **over a range of rows**

Month	Jan	Feb	March	April	May	June
Revenue	1m	4m	1m	2m	3m	4m
3 month moving avg			(6 / 3)m	(7 /3)m	(6 / 3)m	(9/3)m

OLAP: Windowing calculations (1)

e.g., Show the monthly figures and three-month moving averages and sums for property sales at branch office B003 for the first six months of 2020.

```
SELECT yearMonth,  SUM(saleAmount) AS monthlySales,

    AVG( SUM( saleAmount )  )
    OVER (ORDER BY yearMonth, ROWS 2 PRECEDING)
     AS 3month_moving_avg,
    SUM(SUM(salesAmount))
    OVER (ORDER BY yearMonth ROWS 2 PRECEDING)
    AS 3month_moving_sum
FROM   PropertySale
WHERE  branchNo = ‘B003’ AND yearMonth BETWEEN ('2020-01' AND '2020-
    06’)
GROUP BY yearMonth   ORDER BY yearMonth;
```

OLAP: Windowing calculations (2)

Show the monthly figures and three-month moving averages and sums for property sales at branch office B003 for the first six months of 2020.

e.g. of results:

yearMonth	monthlySales	3-moths Moving Avg	3-moths Moving Sum
2020-01	21,000	21,000	21,000
2020-02	35,000	28,000	56,000
2020-03	40,000	32,000	96,000
2020-04	42,000	39,000	117,000
2020-05	44,000	42,000	126,000
2020-06	43,000	43,000	129,000

OLAP: Ranking Functions (1)

- **Computes the rank of a record compared to other records in the dataset based** on the values of a set of measures.

- e.g. **RANK** and **DENSE_RANK**.
- syntax for each ranking function is:

```
RANK( ) OVER
      (ORDER BY columnList)

DENSE_RANK( )
      OVER (ORDER BY columnList)
```

- **Note:**
- **DENSE_RANK leaves no gaps in the sequential ranking sequence when there are ties for a ranking.**

OLAP: Ranking Functions (2)

e.g. Rank the total sales of properties for branch offices in Pretoria.

```
SELECT branchNo,  SUM(saleAmount) AS sales,
RANK()
OVER (ORDER BY SUM(saleAmount)) DESC AS Ranking,

DENSE_RANK()
OVER (ORDER BY SUM(saleAmount))
DESC AS DenseRanking

FROM Branch, PropertySale
WHERE        Branch.branchNo   =   PropertySale.branchNo
  AND Branch.city     =   'Pretoria'
                (branchNo);
```

OLAP: Ranking Functions (3)

Rank the total sales of properties for branch offices in Pretoria.

e.g. of results:

branchNo	sales	ranking	dense_ranking
B009	120,000,000	1	1
B018	92,000,000	2	2
B022	92,000,000	2	2
B028	92,000,000	2	2
B033	45,000,000	5	3
B046	42,000,000	6	4

Automated generation of SQL statements

- Why is this necessary?
  - dimensional tables are very wide
    => long attribute lists
  - SQL queries are big!

- What tools?
  - e.g. Oracle:
  - Oracle Warehouse Builder
    - Statements for creating the ROLAP structure
    - Statements for loading data into ROLAP structure

  - Oracle Discoverer
    - Statements for analysis (SELECT)

OLAP tools

- OLAP tools are categorised according to the architecture used to store and process multi-dimensional data.

- **There are four main categories:**
  - Multi-dimensional OLAP (MOLAP)
  - Relational OLAP (ROLAP)
  - Hybrid OLAP (HOLAP)
  - Desktop OLAP (DOLAP)

Open source software & BI

- **HOMEWORK (1):**

  Look for information on:

  - **PostgreSQL and OLAP**

    or

  - **MySQL and OLAP**

Homework (2)

1. Write the SQL statement to create the TIME dimension. This dimension should have a hierarchy that rolls up as follows: DAY -> MONTH -> QUARTER ☾ YEAR

2. Write the SQL statement that will compute the total revenue for each hotel, for each month. Use a GROUP BY CUBE. Explain all the different values you expect to see in the output.

3. Write the SQL statement that will compute successively rolled up revenues for each hotel, for each quarter. Use a GROUP BY ROLLUP. Explain all the different values you expect to see in the output.

4. Write the SQL statement that will compute the total revenue for the following groupings:

(Hotel, month), (Room_type, month), (Hotel, Room_type, month). Use a GROUP BY GROUPING SETS. Explain all the different values you expect to see in the output.

COS 326
Database Systems

Lecture 25

Revision and Exam Preparation

All lecture slides and textbook chapters

Tuesday 29 October 2024

Highest Semester Test Mark Award

Keanu Canto

In this lecture

1. Examination Information
2. Examination Structure
3. Scope of Coverage
4. Examination Preparation Tips

Examination Information

- COS326 – Database Systems
- ClickUP Closed book Examination
- **Date:** Thursday 14 November 2024
- **Time:** 07:30 Hrs to 10:30 Hrs
- **Venue:** Informatorium (Blue Labs 1 -3 and Green Lab).
- Bring your pen and student card only.
- No flash drives, notes, books, smartwatches, smartphones, tablets etc allowed.
- No extra time will be provided unless there is loadshedding interruption 🏝
- You will have **no access** to lecture slides, practicals, database software etc

Examination Structure

- ClickUP Examination
- 3 Hours long
- Total of **100 Marks** as the semester test
- Consist of 3 sections, namely:
  - ❖ Multiple Choice/Fill in the blanks questions – **20 Marks**
  - ❖ Theory and Application Based Questions – **40 Marks**
  - ❖ Scenario Based Questions – **40 Marks**

➢ **PLEASE NOTE:** There will be questions where you

Scope of Coverage

- All content in all lecture slides is examinable (Lectures 2 – 25) (this includes homeworks + references to reading material provided on ClickUP)
- All content in all 10 practicals is examinable
- Please note: There will be more questions for content that was not covered in the semester test i.e from lectures 12 – 25.
- **Prescribed Book sections:**
  - Chapter 12 Object and Object-Relational Databases (All sections)
  - Chapter 13 XML: Extensible Markup Language (All sections)
  - Chapter 24 NOSQL Databases and Big Data Storage Systems (All sections)
  - Chapter 25 Big Data Technologies Based on MapReduce and Hadoop (All sections)
  - Chapter 26 Enhanced Data Models: Introduction to Active, Temporal, Spatial, Multimedia, and Deductive Databases (Section 26.1 only)
  - Chapter 28 Data Mining Concepts (All sections, refer to lecture video for focus areas)
  - Chapter 29 Overview of Data Warehousing and OLAP (All sections)

Examination Preparation Tips

- Use the lecture slides as guidance when doing revision.
- If some of the sections in the prescribed book were not covered in the slides, then do not waste your time reading that section.
- Start with the easier questions before going for the tough ones.
- Do the homeworks and study the additional reading materials.