Introduction to Modern Databases

Redwan Ahmed Rizvee

(https://rizveeredwan.github.io)

Modern Database Systems: Concepts & Extensions

- An overview of advanced database concepts including
  - object-oriented,
  - object-relational, and
  - specialized databases such as : *temporal, spatial, multimedia,* and *mobile databases*.
  - Complex data types: structured, array, multiset
  - Inheritance, object identity, and reference types
  - Object-relational queries and implementation
  - Persistent programming languages

Introduction to Modern Databases

- Traditional relational databases **are no longer sufficient** for many modern applications.
- The rise of OOP, multimedia, mobile devices, and time/location-sensitive data demands more advanced models.
- Modern DBs extend relational models with richer types, object orientation, persistence, and support for temporal, spatial, multimedia data.

Object-Oriented Databases (OODB)

- Built on object-oriented programming principles.
- **Key features:**
  - Encapsulation: Data + methods encapsulated within objects.
  - Inheritance: Subtypes inherit attributes and methods from supertypes.
  - Object identity: Every object has a unique, system-generated identifier independent of its values.
- **Advantages:**
  - Closer alignment with application programming models.
  - Improved support for complex and user-defined data types.
  - Reusability and extensibility through class hierarchies.

OODB Example Scenario

- **Domain:** University Course Management System
- **Classes:**
  - Person { ID, name, address }
  - Student extends Person { major, GPA }
  - Instructor extends Person { department, salary }
  - Course { courseID, title, instructor (ref), enrolledStudents (list of refs) }
- **Relationships:**
  - Inheritance: Student and Instructor inherit from Person.
  - Associations via references (e.g., instructor assigned to a course).

Query Example in OODB

- **Goal:** List the names of students enrolled in "Database Systems".
- **OQL (Object Query Language):**

```
SELECT s.name
FROM Course c, c.enrolledStudents s
WHERE c.title = "Database Systems";
```

- **Interpretation:** Navigates from course to associated students using p**ath expressions** and **object references**.

```
for course in Course:
        for student in
course.enrolledStudents:
        if course.title == "Database
Systems":
        print(student.name)
```

This is not cross!
`Course c`: Iterates over all Course objects.
`c.enrolledStudents s`: For each course `c`, it iterates over the list/collection `enrolledStudents` and binds each student object to `s`.

Implementation Aspects of OODB

Storage: Objects are stored with OIDs and can refer to each other directly.

Indexing: Path indexes support efficient access to nested or referenced attributes.

Persistence: Objects remain in database across program executions without needing serialization.

Indexing in OODB

Indexing: Path Indexes Support Efficient Access to Nested or Referenced Attributes

- In object-oriented databases (OODB), **data is stored as objects that can reference other objects**, often in nested structures or via pointers.

- When querying, you may want to quickly ***find objects based on attributes that are not directly in the root object, but inside a referenced object or nested deep within the structure***.

- **Path indexing** *creates indexes not just on flat attributes, but on* ***paths*** *through these object references*.

How Indexing Helps?

a) Suppose you have an object Order that references a Customer object, which has an attribute city.

b) A path index **could index the path** Order.customer.city.

c) When you query orders where the customer lives in "New York," the database *uses the path index to quickly find all matching orders without scanning all orders and then fetching the customer info.*

Persistence in OODB

Persistence: Objects Remain in Database Across Program Executions Without Needing Serialization

- Persistence means that data (objects) outlive the program that created or modified them — they are stored permanently until explicitly deleted.

- In traditional programming, to save objects to disk or send them over a network, you typically serialize them (convert them into a flat format like JSON or binary), then later deserialize back to objects.

- **In OODBs, persistence is transparent.**

Tabular Data vs OODB

Category	Tabular Data	OODB
Data Representation	Data is stored in flat tables with rows and columns.	Data is stored as objects with attributes and methods.
Relationships	Represented using foreign keys.	Represented using direct object references.
Complexity Handling	Requires joins for complex relationships.	Uses navigational access and encapsulation.
Schema Evolution	Altering schemas may require data migration.	Easier to evolve due to inheritance and class structure.
Use Case Fit	Well-suited for structured, transactional data.	Ideal for complex data and close integration with object-oriented applications.

Object-Relational Databases (ORDB)

- Merge of **relational (tabular)** and object-oriented (OODB) models.
- Supported by SQL:1999, SQL:2003.
- Features:
  - User-defined types (UDTs)
  - Table inheritance (subtypes)
  - References (Object Pointers)
  - Structured and collection types
  - Methods on types
- Examples: PostgreSQL, Oracle, IBM DB2

Object-Relational Queries allow you to manipulate and retrieve data **using these extended features while maintaining compatibility with SQL**.

Example Schema of ORDB

```
-- Define a user-defined type for Address
CREATE TYPE address_type AS (
  street VARCHAR,
  city VARCHAR,
  zip VARCHAR
);
```

```
-- Define a base table for Person
CREATE TABLE Person (
  id SERIAL PRIMARY KEY,
  name VARCHAR,
  addr address_type
);
```

```
-- Define a table Instructor that inherits from Person
CREATE TABLE Instructor (
  department VARCHAR,
  salary NUMERIC
) INHERITS (Person);
```

```
-- Define a table Student that inherits from Person
CREATE TABLE Student (
  major VARCHAR,
  gpa FLOAT
) INHERITS (Person);
```

```
-- Table Course with a reference to Instructor
CREATE TABLE Course (
  course_id SERIAL PRIMARY KEY,
  title VARCHAR,
  instructor_id INT REFERENCES Instructor(id)
);
```

Mainly PostGres Supports this, not all.

Example Query of ORDB

Relational approach

```
SELECT s.name, s.major, c.title
FROM Student s
JOIN Enrollment e ON s.id = e.student_id
JOIN Course c ON e.course_id = c.course_id
JOIN Instructor i ON c.instructor_id = i.id
WHERE s.major = 'Computer Science' AND i.department = 'CS';
```

- we combine inheritance (Person), UDTs (Address) and References (instructor)

```
SELECT DEREF(s).name, c.title
FROM Course c, UNNEST(c.enrolledStudents) AS s
WHERE c.title = 'Database Systems';
```

Object-Relational approach

Key Features of Object Relational Queries

- **Path Expressions:**
  Navigate through nested attributes or referenced objects using dot notation.
  E.g., student.address.city

- **Dereferencing References:**
  Objects can have attributes that are references (REF) to other objects. You can dereference these in queries.
  E.g., student.advisor.name — advisor is a reference to an Instructor object.
  For OODBMS, student.advisor.name However, for ORDBMS, need to DREF
  DEREF(student.advisor).name

- **Table Inheritance Queries:**
  Queries can exploit inheritance hierarchies in tables.
  E.g., Querying a base table returns all rows including those in child tables.
  SQL: SELECT * FROM Person returns rows from Person, Student, and Instructor tables if they inherit from Person.

- **User-Defined Types and Methods:**
  You can call methods (functions) on objects in queries.
  E.g., student.getAge() to compute the student's age dynamically.

- **Polymorphic Queries:**
  Queries can return results of various subtypes, with the ability to discriminate based on actual type.

Common Syntax Extensions in ORDB Queries

Feature	Example	Description
Path Expression	student.address.city	Access nested fields
Dereference Reference	student.advisor.name DREF(student.advisor).name	Follow reference links
Table Inheritance	SELECT * FROM Person	Retrieve rows from all subtypes
Polymorphic Filter	WHERE typeOf(person) = 'Student'	Filter by subtype
Method Call	student.getGPA()	Invoke methods on objects
Collection Queries	SELECT s FROM course.enrolledStudents s	Query elements of collection attributes

New Primitives for Object-Relational Queries

1	User-Defined Types (UDTs)

	CREATE TYPE address_type AS (
	  street VARCHAR,
	  city VARCHAR,
	  zip VARCHAR
	);

2	References (REF) and Object Identity (OID) (advisor is a REF to an Instructor object)
	SELECT s.name, DREF(s.advisor).name FROM Student s;
3	Path Expressions Navigate nested attributes or referenced objects
	SELECT DEREF(c.instructor).department FROM Course c;
4	Inheritance and Type Hierarchies Tables can inherit from base tables Polymorphic queries on super- and subtypes
	SELECT * FROM Person; -- Returns Persons, Students, Instructors

New Primitives for Object-Relational Queries

5	Collection Types Attributes can be arrays, lists, multisets Query elements in collections
	SELECT DEREF(s).name FROM Course c, UNNEST(c.enrolledStudents) AS s; // UNNEST() is used to flatten collection-type attributes (e.g., arrays, multisets, or lists) into individual rows, allowing relational-style querying over collections stored within tuples.
6	Methods / Member Functions Functions bound to types, invoked in queries
	SELECT s.name, s.getGPA() FROM Student s WHERE s.getGPA() > 3.5;
7	Polymorphic Queries and Type Checking
	SELECT p.name FROM Person p WHERE TYPEOF(p) = 'Student';
8	Nested and Correlated Queries
	SELECT c.title FROM Course c WHERE EXISTS (SELECT 1 FROM UNNEST(c.enrolledStudents) s WHERE s.gpa > 3.8);

New Primitives for Object-Relational Queries

9	Extended Join Operations
	SELECT s.name, c.title FROM Student s JOIN Enrollment e ON s.id = e.student_id JOIN Course c ON e.course_id = c.course_id;
10	Object Creation and Manipulation
	INSERT INTO Student VALUES (NEW Student(...));

Complex Data Types

```
CREATE TYPE Address AS (
  street VARCHAR(50),
  city VARCHAR(30),
  zip VARCHAR(10)
);

CREATE TABLE Employee (
  empID INT,
  name VARCHAR(50),
  address Address
);
```

Structured Types: Nested records/tuples.
Used when you want to encapsulate multiple related fields as a single unit.

```
CREATE TABLE Product (
  productID INT,
  name VARCHAR(50),
  tags TEXT[] -- array of tags
);
```

Arrays: Indexed collections of fixed or variable length.

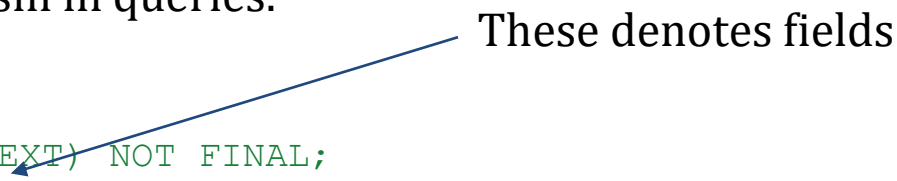
Useful in scientific data where you store series of measurements.

```
CREATE TYPE IntBag AS MULTISET(INTEGER);
```

Multisets: Unordered collections allowing duplicates.

Inheritance and Object Identity

Inheritance: Enables polymorphism in queries.




```
CREATE TYPE Person AS (name TEXT) NOT FINAL;
CREATE TYPE Student UNDER Person AS (major TEXT);
CREATE TYPE Instructor UNDER Person AS (department TEXT);
```

```
CREATE TABLE People OF Person; //
```
This creates a table named People where **each row is an object** of type Person or any of its subtypes (Student or Instructor).

Insertion Query

```
INSERT INTO People VALUES (Student('Alice', 'Computer Science'));
```

Read Query

```
SELECT p.name FROM People p WHERE p IS OF (ONLY Student);
```

Inheritance and Object Identity

- Object Identity (OID): Unique ID for objects.
- Each object has **a system-defined identifier** that remains constant, regardless of the object's values.
- Example:
  - An object of Student type inserted into a table will be assigned an internal OID.
  - You can use this OID **to track** or r**eference the object** even if its attributes change.

Inheritance and Object Identity

- Reference Types: Objects pointing to others using REF types in SQL.

```
CREATE TYPE Department AS (name TEXT);
CREATE TABLE Departments OF Department;

CREATE TYPE Instructor UNDER Person AS (dept REF(Department));
CREATE TABLE Instructors OF Instructor;
```

Querying with Dereferencing

```
SELECT i.name, DEREF(i.dept).name FROM Instructors i;
```

Query Differences — OODB vs ORDB

1) Navigating Collections

OODB: Implicit navigation

```
SELECT s.name
FROM Course c, c.enrolledStudents s
WHERE c.title = 'Database Systems';
```

- enrolledStudents is a list of object references, and the query language (e.g., OQL) handles iteration automatically.

ORDB: Explicit unnesting required

```
SELECT s.name
FROM Course c, UNNEST(c.enrolledStudents) AS s
WHERE c.title = 'Database Systems';
```

- *You must manually flatten collection attributes like arrays or multisets.*

Query Differences — OODB vs ORDB

Dereferencing Object References

OODB: Implicit dereferencing

```
SELECT s.name, s.advisor.name

FROM Student s;
```

- The system understands advisor is a reference and automatically fetches attributes.

ORDB: Explicit dereferencing using DEREF()

```
SELECT s.name, DEREF(s.advisor).name

FROM Student s;
```

- Required to access referenced object's attributes.

Query Differences — OODB vs ORDB

Inheritance in Tables

OODB: Classes with inheritance — queries behave like object method calls

ORDB: Tables support type hierarchies with queries like:

SELECT * FROM Person; -- includes Student and Instructor
SELECT * FROM ONLY Person; -- excludes subtypes
SELECT p.name FROM People p WHERE p IS OF (Student); – polymorphism

Query Differences — OODB vs ORDB

Method Invocation

- **OODB:** Object methods are naturally available (e.g., s.getAge())

ORDB: Only possible if the DBMS supports methods on UDTs
-- Hypothetical example

- SELECT s.name, s.getAge() FROM Student s;

Pros and Cons of OODB and ORDB

OODB

Pros:

- **Seamless integration** with object-oriented programming.
- Natural modeling of complex and hierarchical data.
- Supports encapsulation and behavior through methods.
- Ideal for applications like CAD, simulations, and multimedia.

Cons:

- **Limited standardization** and vendor support.
- **Less mature query optimization and indexing capabilities.**
- Steeper learning curve for developers not familiar with OOP.
- **Poor interoperability with existing relational tools.**

ORDB

Pros:

- **Combines strengths of relational and object-oriented models.**
- **SQL-based with extensions for structured and user-defined types**.
- Better support and adoption in commercial RDBMS (e.g., PostgreSQL, Oracle).
- Easier transition from legacy relational systems.

Cons:

- More **complex SQL syntax and schema design**.
- **Method support varies across platforms and may be limited.**
- **Object behavior (methods) not as naturally integrated** as in OODBs.
- **Inheritance and polymorphism can be**

Persistent Programming Languages

- A persistent programming language integrates data persistence directly into the language semantics.
- Objects or data structures created in the language can persist beyond program termination without requiring explicit serialization or database commands.

key Features:

Orthogonal Persistence
Any object, regardless of type, can be made persistent without changing how it is written.

Transparency
No special code or syntax is required to store or retrieve persistent data—access is the same as for transient (in-memory) data.

Type Safety
Type checking is maintained even for persistent data.

Garbage Collection
Unused persistent objects may be collected over time.

Advantages: 1) Simplifies code: less boilerplate for storing and retrieving data.
2) Enhances productivity for complex and long-running applications (e.g,, notebook)
3) Reduces impedance mismatch between programming language and database, ala, direct object based communication rather than translation to SQL

Temporal Databases

- Time-varying data support.
- Time dimensions:
  - Valid time: Real-world validity
  - Transaction time: Stored in DB
- SQL extensions: AS OF, VALID BETWEEN, TEMPORAL JOIN
- Applications: Auditing, compliance, time-series analysis.

```
SELECT salary
FROM Employee
WHERE empID = 101 AND PERIOD [2018-
01-01 TO 2019-01-01] OVERLAPS
employment_period;
```

Real-World Examples:

- **Healthcare Records:** Track patient information over time, including diagnoses and treatments.
- **Financial Systems:** Monitor stock prices, account balances, and transactions over time.
- **Human Resources:** Maintain employee roles, salaries, and department changes historically.
- **Insurance Claims:** Track policy changes, claim timelines, and customer status.

Spatial Databases

- Handle **geometric, topological, and geographic data**.
- **Use Cases:** GIS, mapping, location-based services, urban planning.
- **Features:**
  - Spatial data types (points, lines, polygons)
  - Spatial indexing (e.g., R-trees)
  - Spatial predicates (e.g., ST_Within, ST_Distance)
- **Example Query:**

```
SELECT name FROM Locations WHERE ST_Distance(geom,
ST_Point(23.8, 90.4)) < 1000;
```

Multimedia Databases

- Manage **image, audio, video,** and **associated metadata**.
- **Use Cases:** Digital libraries, surveillance, media asset management.
- **Features:**
  - Support for media types and attributes (resolution, codec, duration)
  - **Content-based retrieval** (e.g., image similarity)
  - **Storage and indexing of large binary objects** (BLOBs)
- **Example Query:**

```
SELECT title FROM Videos WHERE duration < 300 AND resolution = '1080p';
```

Mobile Databases

- Designed for use on mobile or distributed devices.
- **Use Cases:** Offline applications, field data collection, mobile banking.
- **Features:**
  - Disconnected operation and synchronization
  - Location/context-aware data services
  - **Lightweight storage engines** (e.g., SQLite, Realm)
- **Example Scenario:**
  - A delivery app syncing route logs collected offline once network is available.

Summary

- Modern DBs extend beyond traditional models.
- ORDB/OODB: Add structure and identity.
- Support for temporal, spatial, multimedia data.
- Persistence and mobility are shaping DBMS evolution.

References

- • Elmasri & Navathe – Fundamentals of Database Systems
- • Silberschatz et al. – Database System Concepts
- • Garcia-Molina et al. – Database Systems: The Complete Book
- • C.J. Date et al. – Temporal Data & the Relational Model