Object – Oriented Design with UML and Java

PART XVIII: Database Technology

What is a Database?

- Computerized record-keeping system.
- Collection of stored (persistent) operational data.
- The Data Model (schema)
- Data (integrated and shared)
- Hardware (physical considerations)
- Software (the DBMS, client applications)
- Users (programmer, end-user, DBA)
- Numerous other features...

Motivations

- Businesses usually want their data to persist independently of software.
- When a program is re-started, it should appear as if the objects were always there.
- Multiple applications may need to work with the same objects.
- When one object wishes to talk to another, it should not need to know if that object is in memory or must be fetched from the persistent storage.
- A business has a large number of customers, suppliers, etc… Detailed information about everyone and everything needs to be collected and stored somewhere.

What is Persistence?

- Most useful software stores data in some sort of persistent storage.
  - Files / Databases / Tapes / CDs / ...
- Conventional programs read and write persistent data in a notably deliberate manner.
  - Concerns about reading and writing to/from the persistent storage are separate from the concerns of the problem domain.
- Object-oriented programs seek to make the storing and fetching of persistent *objects* as transparent as possible.
  - Both data and behavior can be persistent.
- Relational database technology is fast and mature with numerous available tools for manipulating and reporting on the data.

Centralized Control Achieves:

- Less redundancy
- Less inconsistency
- Data sharing
- Enforcing standards and naming conventions
- Enforcing security
- Integrity
- Concurrency
- Balancing conflicting requirements
- Data Independence from applications
- Having a DBA (Database Administrator) to be responsible for all of this.

Different Types of DBMSs

- Flat-File (2200 b.c. - Present)
  - One record per data entity.
  - Works for simple data with no need for transactions.
- Pre-Relational (Hierarchic (1965), Network (1970))
- Relational (Ted Codd, 1980)
- Object (1990 - present)
- NoSQL Databases have become popular for many scenarios, each with pros and cons compared to the powerful Relational standard.
  - Documents, often JSON or XML
  - Graphs, such as social networks
  - Key-Value pairs
  - Consider performance, consistency, support for transactions and complex queries

Transaction ACID Properties

- Atomicity (indivisible unit of work)
  - **All or nothing.**
- Consistency (leaves system in a stable state)
  - **If this cannot be done, the transaction will abort.**
- Isolation (not affected by other transactions that execute concurrently)
  - **Requires a locking strategy.**
- Durability (effects are permanent after commit)
  - **Implies persistent storage, such as a database.**

```
Database.beginTransaction( )
CheckingAccount.debit( $1000 )
SavingsAccount.credit( $1000 )
Database.commitTransaction( )
```

Flat-file Solution: Serialization

- Supported directly by language in Java and Smalltalk.
- Supported through vendor-specific toolkits in C++.

Pros:

- – With language or toolkit support, it is simple and straightforward.
- – Can be used for quick-and-dirty persistence on a project before the real database is up and running.
- – Useful for things other than persistence, such as for distributed and multi-tiered applications.

Cons:

- – No transaction support.
- – Startup or searching can be expensive if a large number of objects are stored.
- – Not a substitute for a real database.

Persistence in the Real World

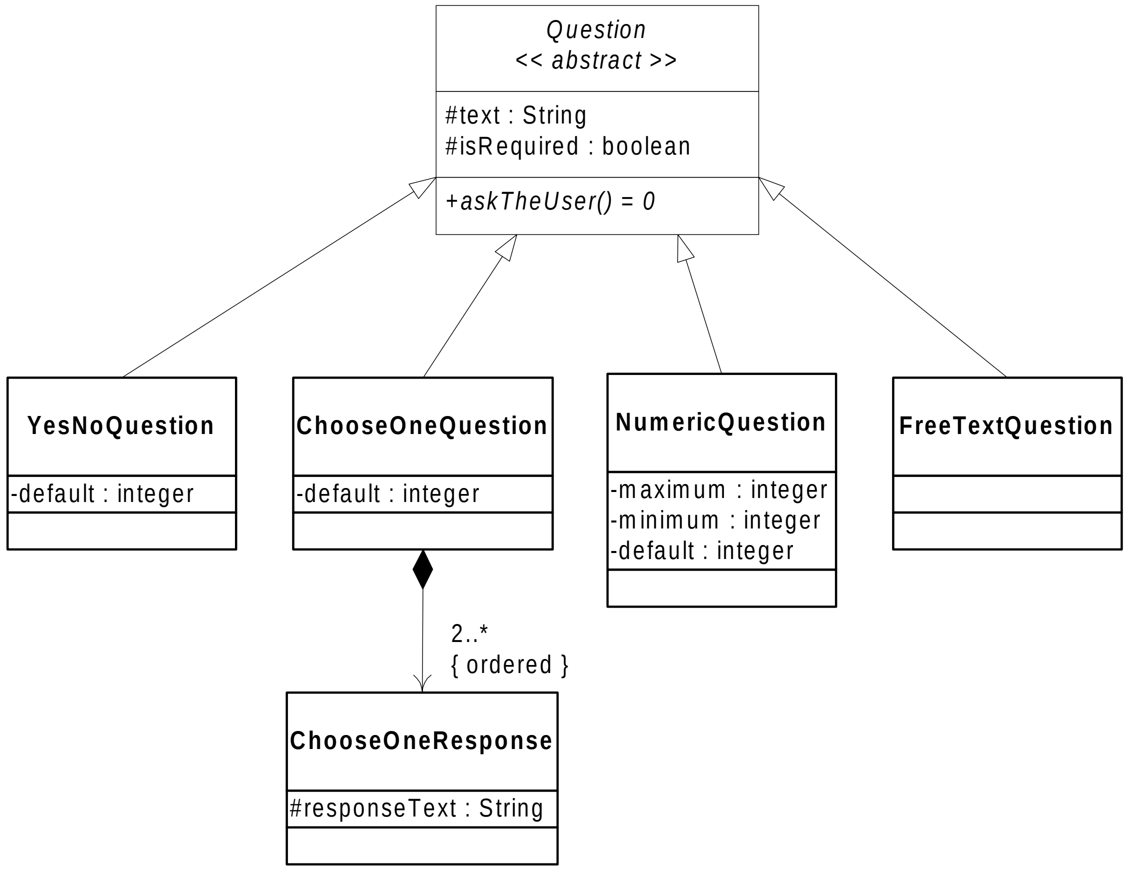
- Most of the real world uses ***relational databases***.
- Based on one simple concept: the table.
- Four major parts:
  - Data that is presented in tables (entities).
  - Operators for manipulating tables.
  - Integrity rules on tables.
  - Associations (relationships) between tables.
- The design for a particular database is called a *schema*.
- Very mature technology.
- Available on all platforms.
- Wide variety of robust tools available.

RDBMS Integrity

- Primary key.
  - A combination of one or more attributes whose value(s) unambiguously locates each row in a table.
  - Each table must have exactly one primary key.
- Foreign key.
  - A primary key of one table that is used as an attribute in another (or the same) table.
- Referential integrity.
  - The RDBMS must keep each foreign key consistent with its corresponding primary key. This can be a challenge to maintain. It is easy to delete a row from a table, for example, causing a row in some other table to have its foreign key refer to the no-longer-existent row.

Example Data Model

- Design a Relational "ER" Data Model for the following domain:



Example Data Model

Normalized Data Model:

(one table per class hierarchy)

<< FK >> = Foreign Key

<< PK >> = Primary Key

- Note: the *Primary Key* for the **Choose_one_response** Table is a *Composite Key*: **question_id + response_num**

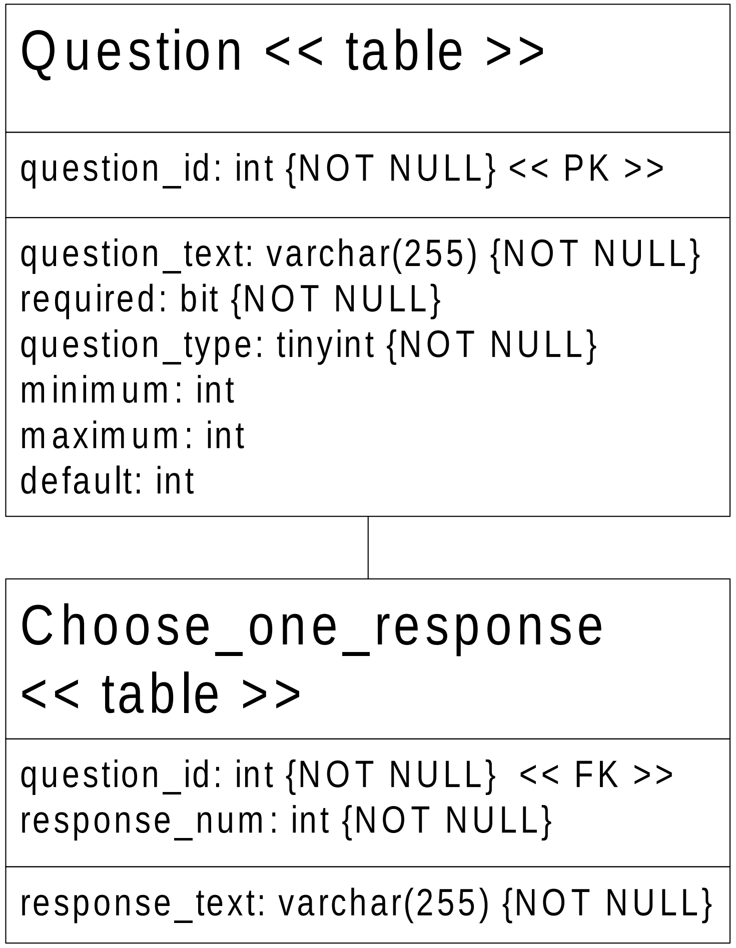
<u>question_type</u>

0 - YesNoQuestion

1 - ChooseOneQuestion

2 - TextQuestion

3 - NumericQuestion



Question << table >>

question_id: int {NOT NULL} << PK >>

question_text: varchar(255) {NOT NULL}
required: bit {NOT NULL}
question_type: tinyint {NOT NULL}
minimum: int
maximum: int
default: int

Choose_one_response << table >>

question_id: int {NOT NULL} << FK >>
response_num: int {NOT NULL}

response_text: varchar(255) {NOT NULL}

Example Data Model

Question

question_id	question_text	required_bit	question_type	minimum	maximum	default
0	Could it be?	1	0	NULL	NULL	0
1	Which is it?	1	1	NULL	NULL	2
2	Why?	0	2	NULL	NULL	NULL
3	How many?	1	3	0	999	42

Choose_one_response

response_num	question_id	response_text
0	1	Not me.
1	1	Me neither.
2	1	Pick me!
3	1	Pick #2.

Example data for the tables on the previous slide:

Note: Tables are analogous to classes, while rows are like objects.

SQL (Structured Query Language)

- SQL is a standardized declarative (not procedural) language. This implies that it does not support looping or if - then logic.
- Note however that all RDBMS vendors provide *stored procedure* languages that do provide looping and other procedural constructs. The problem is that these languages are not standardized.
- The RDBMS will *compile* the SQL into procedural form "under the covers" using **search** which will be slow if the correct indexes are not maintained.
- *Searches* are linear, O(n), by default, but can be made to be binary, O(log(n)), by maintaining an *index* on every set of search keys.

```
select c.response_text from Question q, Choose_one_response c
where q.question_type = 1 and
q.question_id = c.question_id and  // "join" the two tables
c.response_num = q.default
```

- *Outputs:* **Pick me!**

JDBC (Java DataBase Connectivity)

- JDBC is a Java API which makes it possible to write 100% Java code that can connect with any database in a portable way.
- Similar to Microsoft’s ODBC, which has a C interface.
- Refer to: **`http://java.sun.com/products/jdbc/`**

```
import java.sql.*;
Connection c = DriverManager.getConnection( “jdbc:/foo”, “a”, “1” );
Statement s = c.createStatement();
ResultSet rs = s.executeQuery( “Select foo, bar from TableFoo” );
while( rs.next() )
{
  // Uses the Iterator Design Pattern
  int    _foo = rs.getInt( “foo” );
  String _bar = rs.getString( “bar” );
}
```

Normalized vs. Denormalized

- *Normalization* is the removal of structural redundancies in a data model. This implies that queries generally require more table joins, which hurts performance.
- Experienced data modelers generally design their schemas to be fully normalized to eliminate duplication, and then denormalize in a few select places where appropriate, for performance gains (or reporting).
- The force against denormalization is that whenever there are redundancies, there is risk of having mis-matches, with a corresponding increase in effort required to ensure integrity.
- Queries run against denormalized schemas can require fewer joins and therefore run faster.

Mapping Classes to Tables

- **One table per class hierarchy.**
  - No subclass tables; bring subclass attributes up to the superclass level; add type field. If any Question subclass changes a persistent attribute, the table will change, affecting the mapping for *all* Question subclasses.
- **One table per class.**
  - Each instance has one row in each table in its inheritance chain.
- **One table per concrete class.**
  - Superclass attributes are replicated for each subclass.
- **One class, many tables.**
  - More tables can be used to improve performance in some cases, but will degrade performance whenever two tables must be "joined."
  - Horizontal partitioning: Put infrequently used instances in another table.
  - Vertical partitioning: Put infrequently used attributes in another table.

Relational Data Modeling

Question

question_id	response_num	question_text	required	question_type	minimum	maximum	default	response_text
0	0	Could it be?	1	0	NULL	NULL	0	NULL
1	0	Which is it?	1	1	NULL	NULL	2	Not me.
1	1	Which is it?	1	1	NULL	NULL	2	Me neither.
1	2	Which is it?	1	1	NULL	NULL	2	Pick me!
1	3	Which is it?	0	1	NULL	NULL	2	Pick #2.
2	0	Why?	0	2	NULL	NULL	NULL	NULL
3	0	How many?	1	3	0	999	42	NULL

- *Denormalized* Data Model.
- One table for class hierarchy, and the Choose_one_response data too!
- The Primary Key is: question_id + response_num!
- Note the increase in `NULL` data elements.
- Can increase performance for some queries at the expense of others.
- Note the duplicate data for questions of question_type 1 (and the corresponding *effort* required to ensure consistency; find the bug).

Relational Data Modeling

Choose_one_question

question_id	response_num	question_text	required	default	response_text
1	0	Which is it?	1	0	Not me.
1	1	Which is it?	1	2	Me neither.
1	2	Which is it?	1	2	Pick me!
1	3	Which is it?	1	2	Pich #2

Numeric_question

question_id	question_text	required	default	minimum	maximum
3	How many?	1	42	0	999

Text_question

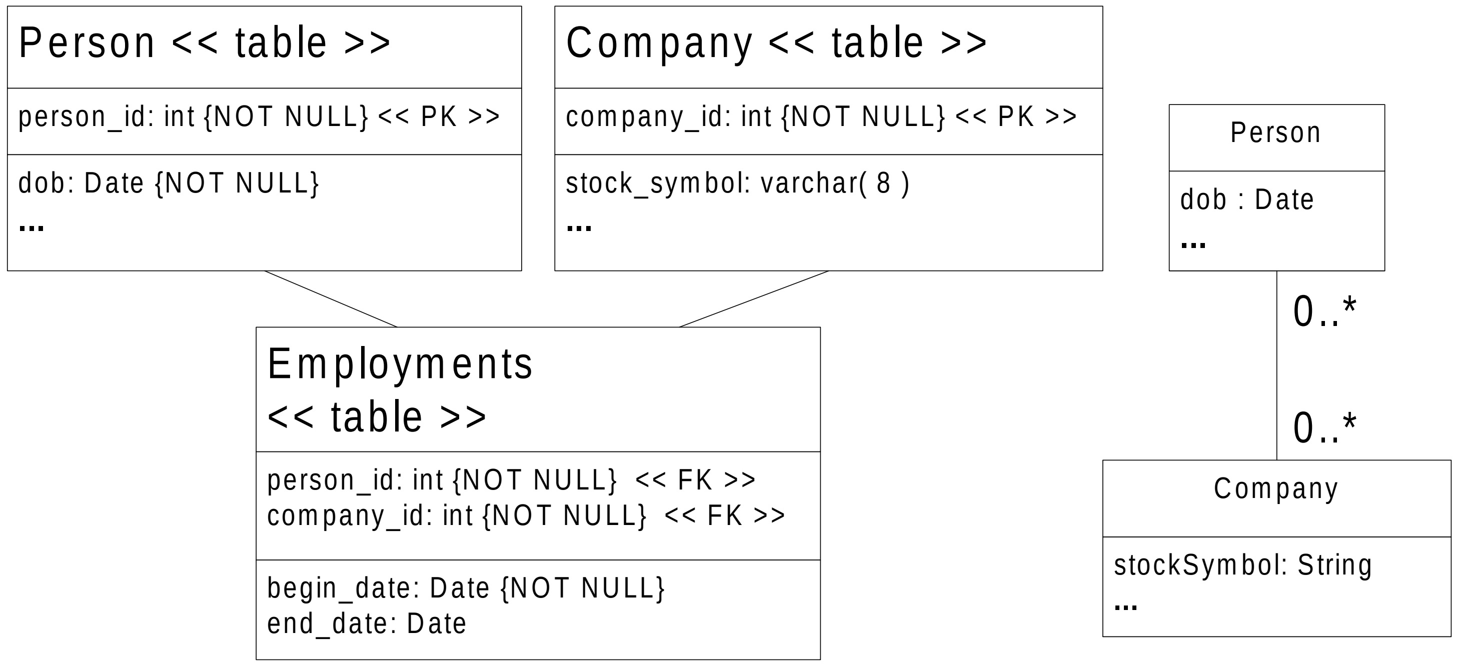
question_id	question_text	required
2	Why?	0

Yes_no_question

question_id	question_text	required	default
0	Could it be?	1	0

- One table for each *concrete* class (more *normalized*).
- Choose_one data is still denormalized (find the corrupt data bug).
- Given list of question_ids, need 4 queries to create objects.
- Note: no `NULL` data elements.

Mapping Many-to-Many Associations


Person << table >>
person_id: int {NOT NULL} << PK >>
dob: Date {NOT NULL}
...
Company << table >>
company_id: int {NOT NULL} << PK >>
stock_symbol: varchar(8)
...
Employments
<< table >>
person_id: int {NOT NULL} << FK >>
company_id: int {NOT NULL} << FK >>
begin_date: Date {NOT NULL}
end_date: Date
Person
dob : Date
...
0..*
0..*
Company
stockSymbol: String
...

- Whenever two classes have a *many-many* relationship, the database requires a third *association table* (or *link table*) to explicitly represent the mapping.
- The primary key on the link table will be the pair of foreign keys from the other tables. The Primary Key for Employments is: person_id + company_id.

OO to RDBMS Mapping Issues

This infamous “Impedance Mismatch” …

- Object Ids (OIDs).
- Mapping Classes to Tables is not one-to-one.
- XML, too!
- It is an *effort* to properly deal with all of the different kinds of database errors that can possibly occur.
  - Deadlock, out of memory for data, out of memory for log file, bug in a stored procedure or sql query, process timeout, server crash, inadequate permissions, optimistic concurrency retries, **dirty data**, etc, …
- When ***deadlock*** happens in a database, some DBMSs will chooses one of the two processes at random to kill, allowing the other one to proceed normally.

Object IDs

- Every object has an ID that is unique either within the object's class or across all objects. If you use integers for IDs and generate them yourself, use 64-bit integers. This relieves you of reclaiming and reusing Ids.

Pros:

- Objects have identity apart from their properties.
- Is uniform mechanism for identification across all classes.
- DBMS perform well with numeric keys.
- Some RDBMS provide direct support for ID generation.

Cons:

- Many classes do not have good natural primary keys (e.g. people).
- Users want to access data based on domain attributes.
- Tools such as report writers don't have an object view of the world.

Impedance Mismatch Example

The program will maintain a list of Questions, and invoke askTheUser() for each...

- Given a list of question_ids (retrieved from a query), the code must instantiate a heterogeneous list of concrete subclasses of Question...
- Logically, the code must query the Question table given for each question_id. Then, depending on the value of question_type, it can choose the correct concrete subclass to instantiate. If the question_type is 1 (ChooseOneQuestion) then the code must go back to the database to query Choose_one_responses.
- It is common for the mapping code to have to loop over the *result set* returned by a query, typically using a database *cursor*.
- Note that the running example models Questions, but not Answers. Data will have to be updated / inserted into the *Answer* table(s) within one *transaction*, respecting *referential integrity constraints*, using either *optimistic* or *pessimistic* concurrency control.

This inherent difficulty in mapping objects to relational database tables is referred to as *impedance mismatch*. It is a very common problem in the real world.

Optimistic vs. Pessimistic Concurrency

- Pessimistic locking prevents multiple concurrent users from accessing the same object at the same time. When an object is read into memory, the corresponding record(s) in the database are locked, preventing all other processes from access to the object. This is a simple and secure approach, but can lead to serious performance problems if a lock is held for a long time on an object that others wish to manipulate.
- With optimistic concurrency control, objects are locked in the database only for the time it physically takes to read or write their records. Whenever an updated object gets written to the database, there must be code to prevent overwriting someone else’s previous update. This is done using *timestamps*. When you read an object you read its timestamp; when you write it, you must compare the object’s timestamp with the database’s (an additional query). If the two timestamps are the same, it is OK to do the update; else, there must be code to gracefully retry the failed operation.

Persistence Designs

All of the mapping code can be implemented as the responsibility of:

- Each and every persistent class.
  - One straight-forward approach uses an abstract class that defines public interface methods **save**(), **delete**(), and **retrieve**().
  - Note that save() has two modes, **insert** and **update**.
  - The class has an Object ID which maps to the primary key from some table.
  - The class keeps a timestamp for *optimistic* concurrency control.
  - Hard-coded embedded SQL (maybe using JBDC) is used to map every object’s persistent attributes to the tables’ columns.
- Persistence frameworks
  - Automate as much of the object to relational mapping burden as possible.
  - Refer to EJB 3.0+ ***Container Managed Persistence*** and ***Hibernate***.
  - Data Access Object (DAO) design pattern.

Persistence Designs

The mapping must be represented somewhere, either as code or as *metadata*.

- Framework code must be written to interpret and maintain metadata. Tools exist to support mapping metadata (often in XML).
- Java's reflection capability is convenient for such a design; the metadata might say that class Foo's attribute bar maps to table Foo's column bar. The class that interprets the metadata will dynamically create a call to class Foo's methods getBar() and setBar().
- With JDK 5+ the mapping may be defined with *annotations*.

Using annotations to codify mapping

```
@Entity // EJB entity bean
@Table(name=“ANNOTATION_STORAGE”)
public class MyAnnotationStorageExample
{
  private Long id;
  private String foo;

  @Id // this column is the primary key
  @Column(name=“ANNOTATION_ID")
  public Long getId() { return id; }
  public void setId(Long id) { this.id = id; }

  @Column(name=“FOO”)
  public String getFoo() { return foo; }
  public void setFoo( String foo) { this.foo = foo; }
}
```

Persistence Designs

- A performance optimization allows the object to be instantiated either:
  - with just the Object ID that uniquely identifies it.
  - with all of the object’s other state data as well.
- If all that is required is the Object ID, then it is a waste of resources to fetch the rest of the data. In this case, use the ***Proxy*** design pattern:
  - The proxy is a “smart” handle for the persistent object.
  - The proxy knows whether the real persistent object is in memory or must be fetched, and whether it must be saved to persistent storage when it is no longer needed to be kept in memory.
  - The proxy has the same interface as the real persistent object.
  - Calls made to the proxy are forwarded to the real persistent object.

Persistence Designs

- How does the proxy know if an object is in memory or must be fetched?
  - Each class keeps a cache of its in-memory instances, indexed by primary key.

Another performance optimization:

- The persistent object keeps a *dirty* bit, set to *false* when the object is fetched.
- Every *set<xxx>* method sets *dirty = true*. When the object is saved, the proxy tells the actual object to store itself only if *dirty = true*.

Relational Databases can scale ***vertically*** (use a bigger server) but not necessarily ***horizontally*** (multiple RDBMS instances in parallel). Therefore scalability can be an important consideration in the architectural decision process. One common pattern used to help with this is called ***Sharding***.

Relational DBMS Concepts

- Entities (tables, primary keys) & Relationships (foreign keys).
- Normalization (elimination of structural redundancies).
- Structured Query Language (SQL).
- Stored procedures (procedural SQL).
- Indexes (ad-hoc performance tuning).
- Referential Integrity (can be declarative - implemented w/ triggers).
- Triggers (stored procedures that execute upon inserts, updates or deletes).
- Multi-user concurrency (locks & transactions).
- Security (access control).
- Data Independence (from client applications).
- “Impedance mismatch” with OO languages.
- Mathematical foundation in set theory.
- Fault tolerance (log file for recovery).
- Mature tools for administration and notably, report writing.

Object DBMS Concepts

Not all of these statements hold true for all OODBMS vendors:

- Trivial mapping to OO programming languages.
- Lack of standards... each vendor has significant variation.
- Many vendors do not supply an ad-hoc query language.
- Support for all OO relationships (not just sets).
- Common use of proprietary collection classes and iterators.
- Often better performance (at the expense of data independence).
- Data is tightly coupled with programs and programming language.
- Abstract user-defined data types (not just a few primitive types).
- Procedures and Data together.
- Preserve data encapsulation.
- Single model for persistent and transient data.
- Record-at-a-time (does not utilize set theory).
- “Active” processing of OO language code (compare to SQL triggers).

Considerations in choosing a DBMS

- Estimate the volume of reads and writes separately.
- How many concurrent users will there be?
- Separate OLTP (on-line transaction processing) and reporting databases?
- Data Warehouse / Data Lake / Data Mesh
- Disaster Recovery and Replication
- How fast does it have to be?
- How mature are the tools?
- Is a No-SQL database a better fit?
- Many large applications have many different databases.
- Consider support for transactions and complex queries.
- Is strong consistency required, or eventual consistency?
- Further discussion of databases is beyond the scope of this course.