Project # 1: Database Programming

CSE462 Database Concepts

Demian Lessa

Department of Computer Science and Engineering
State University of New York, Buffalo

February 21, 2011

Outline

Database Applications

Typical business scenario.

- Business data resides on a relational DBMS.
- Business applications are coded using OOPL.
- Applications are data driven (i.e. data intensive).
- Overview of data retrieval interaction.
  - Application requests data from the DBMS.
  - DBMS uses request info to retrieve data and send to application.
  - Application processes retrieved data and renders a view.
  - View is displayed by application.
- Overview of data update interaction.
  - Application requests an update to the DBMS.
  - DMBS uses request info to perform update and send status to application.
  - Application checks update status and renders a view.
  - View is displayed by application.

Database Applications

Detailed data retrieval interaction (analogous for update).

- Application requests data from the DBMS.
  - Uses a data API to send SQL strings to DBMS.
  - API is DBMS specific (e.g., libpq.so) or agnostic (e.g., JDBC).
- DBMS executes SQL, retrieves data, and send it to application.
  - The API is the actual receiver of all data.
- Application processes retrieved data and renders a view.
  - Uses the data API to process the returned data.
  - E.g., iterate over every tuple in a result set.
- View is displayed by application.

Data API

Database Abstraction APIs (Applications Programming)

- Abstracts away particularities of the underlying DBMS.
- Applications become (in theory) portable across DBMSs.
- Advanced features of particular DBMSs may not be available.
- Requires a driver/provider specification (connection string).
- In our project, we will use Java's JDBC.

Data API

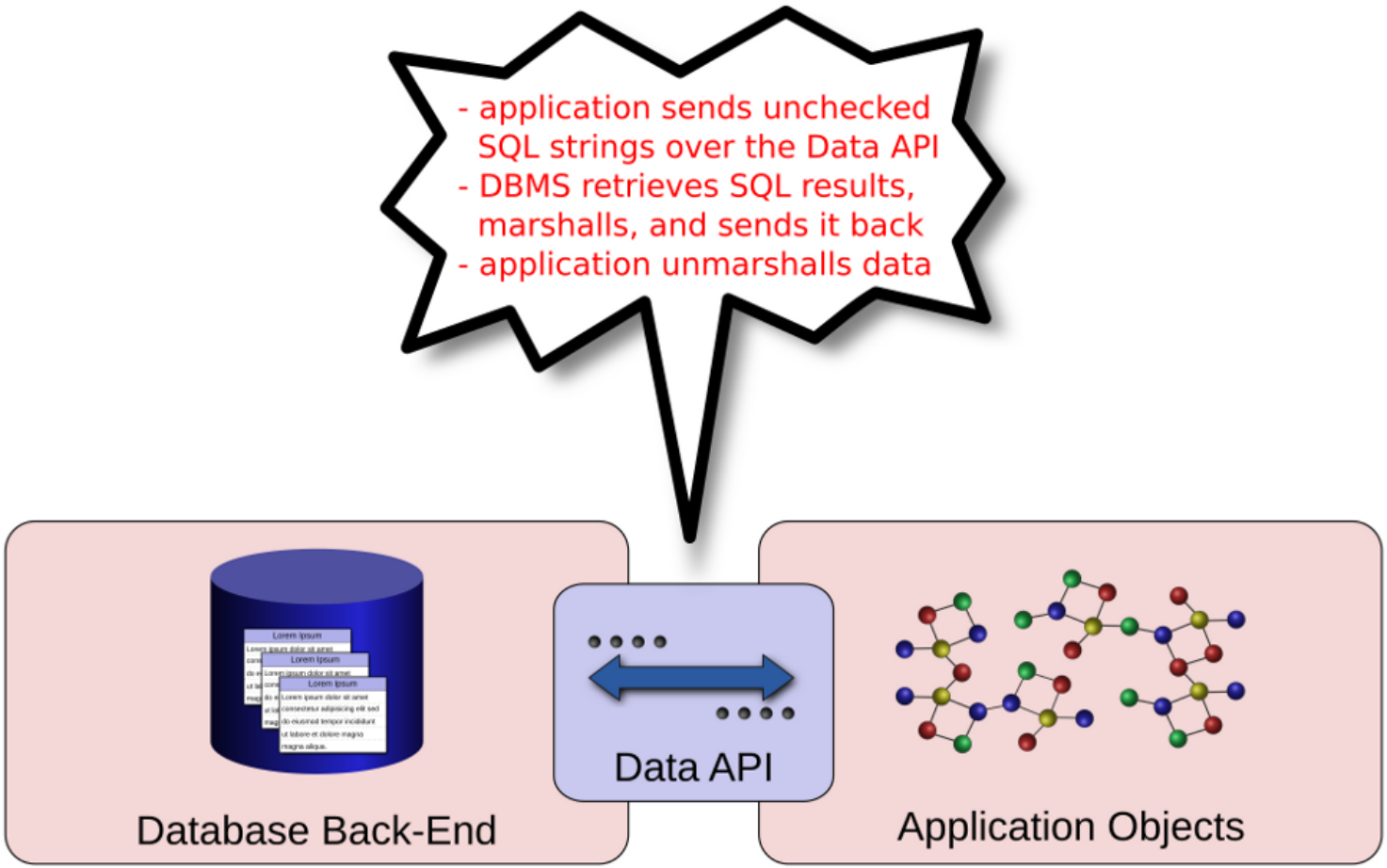

Figure: The data API used directly in application code.

Data API

What are some of the problem with the scenario we just saw?

- SQL spaghetti.
  - SQL strings are scattered throughout the code.
  - Some are even built dynamically, based on user input.
  - What to do if you get, e.g., an *invalid statement* exception?
  - Are there similar statements with possibly the same error?
  - How do you find them *systematically*?
- Lack of consistency.
  - Methods may use different SQL statements for the same purpose.
  - Invariant: different programmers, different SQL statements!
  - Are statements different due to error or sylistic differences?
- Unclear design strategies.
  - How are connections shared and/or pooled?
  - How are transactions controlled?
  - Are statements properly and consistently sanitized?
  - Each portion of code may do things its own way!
  - Hard to reuse code across modules and/or applications.

Data API

What are some of the problem with the scenario we just saw?

- Object-relational impedance mismatch.
  - Design goals: data vs behavior.
  - Building blocks: tables/rows/fields vs classes/instances.
  - Type systems: e.g. BLOB vs PDFDocument.
  - Data retrieval: DML queries vs navigational access using getters.
  - Data modification: DML modification statements vs setters.
  - Error handling: (almost) no recovery vs structured error handling.
  - DBMS only: referential integrity, transactions, concurrency control, etc.
  - OOPL only: inheritance, interfaces, relationships, reflection, etc.

Object-Relational Mapping (ORM)

ORM is one solution, not “the” solution.

- Natural programming model.
- You program OOP, the mapping layer does the data plumbing.
- ORM classes used and tested independently of application.
- Minimizes DBMS trips with optimized fetching strategies.
- A good ORM library should do better than an average programmer.
- Reduces coding time and total code size.
- Code is easier to read and maintain.
- Error frequency is significantly decreased.

Object-Relational Mapping (ORM)

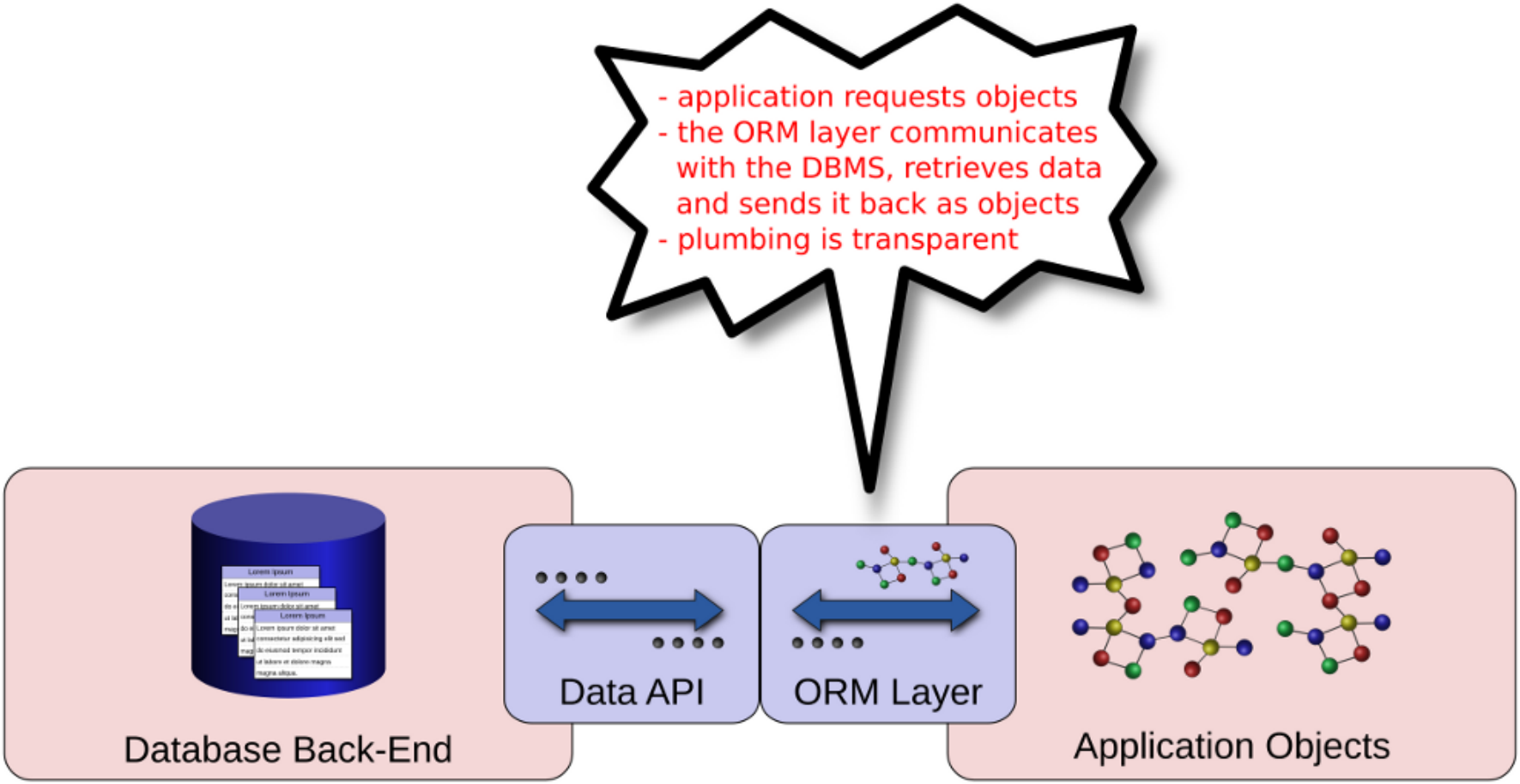

Figure: An ORM layer effectively isolates application code from the data API.

Object-Relational Mapping (ORM)

ORM Desirable Features (not exhaustive).

- Transparency (POJOs/Beans).
- Transitivity (relationships).
- Persistent/transient instances (attached/detached).
- Automatic dirty instance detection.
- Inheritance strategies (single table, class per table, etc).
- Fetching strategies (lazy/eager/hybrid).
- Transaction control.
- Performance.
- Flexible, “sensible defaults” based configuration.
- Availability of development tools and learning resources.

An Optimal Solution?

What does an optimal solution look like?

- A single data model across PL and DBMS.
- No approach has succeeded (so far) for programming-in-the-large.
- What does a sub-optimal solution look like?
- It brings PL and DBMS data models as close as possible.
- Bridges model differences as automatically as possible.
- Effectively isolates all necessary plumbing from business layer.
- Allows freedom of choice (e.g., PL, data API, DBMS).

Outline

Database Application Design

- Typical enterprise database applications are very complex.
- Business logic is anything but trivial.
- These applications are normally designed in multiple layers.
  - A layer receives requests only from the layer immediately above.
  - To satisfy requests, it forwards its requests to the layer immediately below.
- A common break-down of these layers is as follows:
  - Layer 1: Data Store(s).
  - Layer 2: Data Access.
  - Layer 3: Business.
  - Layer 4: Controller.
  - Layer 5: View.

Database Application Design

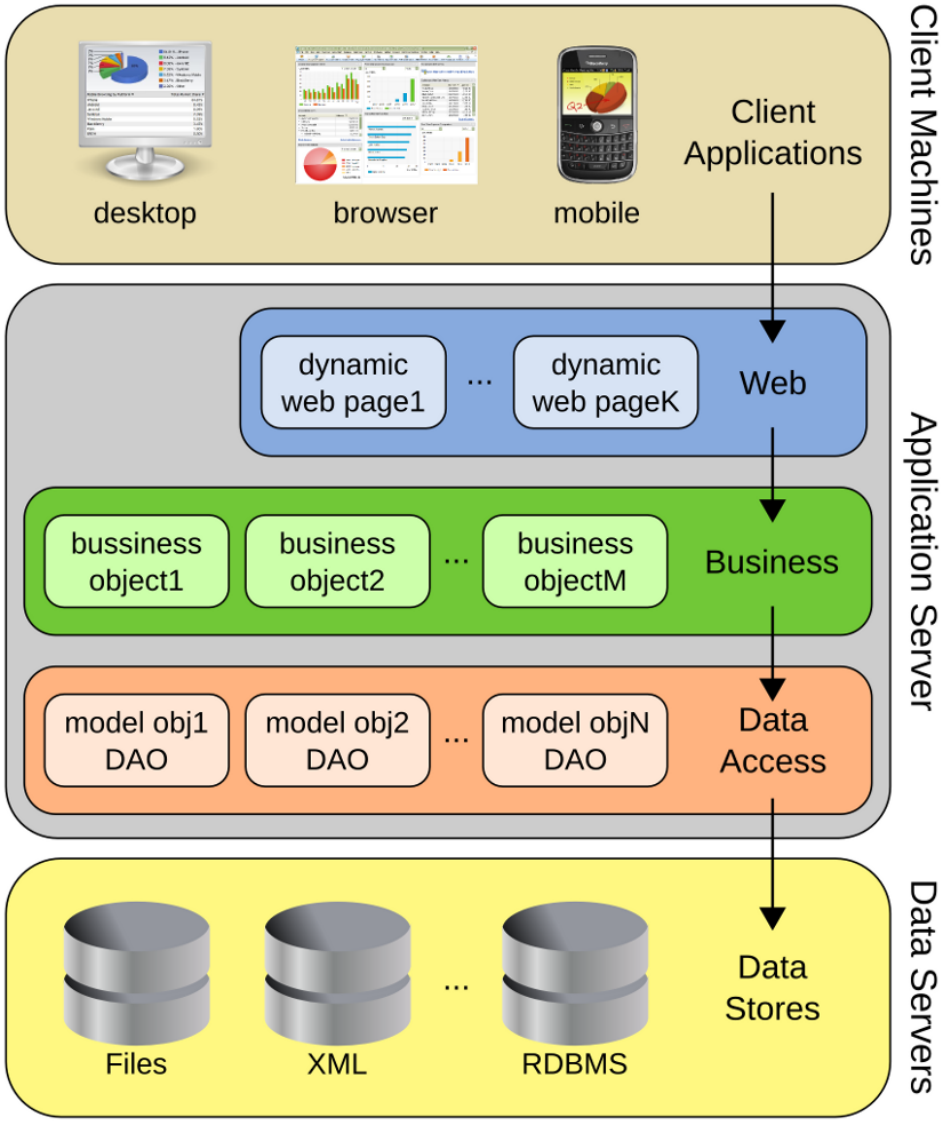

Figure: Complex, multi-tier design for enterprise applications.

Database Application Design

Layer 1: Data Store

Essentially, this layer provides persistent storage for application data. We will use a relational store in our project, wherein data resides in a relational database management system (RDBMS) and thus, may be shared across multiple applications. The RDBMS provides a high-level language (SQL) through which data modifications and queries are executed. The services provided by the store are typically accessed by applications through a low-level data access API.

Layer 2: Data Access

A typical approach is to have all data access go through an ORM layer. The layer receives query and update requests from the layer above in terms of *model objects* (a.k.a. *data objects*). Layers above are oblivious to the low-level details of how data is stored, updated, or queried. To satisfy a request, the ORM layer generates requests to the RDBMS using the low-level data API (e.g., JDBC). Responses from low-level requests are processed to generate a response in terms of model objects.

Layer 3: Business

This layer is concerned with the application's (business) logic. The *business objects* in this layer use model objects and the services of the data access layer to implement the application logic. In simpler applications, there is no distinction between data and business objects, in which case business logic is implemented in the data objects themselves and Layers 2 and 3 are effectively merged. As business logic evolves and application complexity increases, the need for separating data and application logic becomes obvious.

Layer 4: Controller

Complex applications do not make direct requests to business objects. Instead, they rely on an intermediate controller layer capable of dispatching requests to appropriate business objects, based on the nature of the request (very common in web applications). Requests may originate from a user interface (view) or some other application. The controller analyzes and validates the request, then decides which services to invoke on the business objects. Based on the responses from the business objects, the controller decides which user interface (view) to render and pass control next. This layer effectively determines the application workflow.

Layer 5: View

Users interact with the components of this layer. These components are user interface elements that collect data and request actions. For instance, collecting customer information (using a form) and requesting that the customer information be inserted into the store (through a click on the appropriate button).

Outline

The Reference Project

- A skeleton implementation of an ORM layer.
  - All the “plumbing” between objects and relational data is in place.
  - Simple variant of the Data Access Object (DAO) design pattern.
  - `http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html`.
- Status of the reference project.
  - Parts of the implementation is provided as a reference/guide.
  - But most of the DAO coding is left for you to complete.
  - The reference project is commented as clearly as possible.
  - Read the source and the comments before you start.
  - Try to adhere to the code standards used in the reference project.
- The application involves a number of familiar model objects:
  - Customer, Product, Order, OrderEntry.
  - DDL statements are provided in the respective model Java files.

The Reference Project

- The entry point to the DAO functionality is the `DAOFactory` class.
  - Use this class to obtain a `DAOSession` object.
- `DAOSession` is your DAO provider.
  - Encapsulates a connection to the data store.
  - The connection is shared among all DAO objects.
  - This strategy may be easily and transparently modified.
  - Supports transaction control for grouping data operations into atomic units.
  - Provides factory methods for obtaining DAO objects.
- Every DAO object.
  - Interfaces with the data store through the shared connection.
  - Implements all basic services defined by `GenericDAO`.
  - Implements additional services to query/modify one type of model object.
  - E.g., `CustomerDAO` provides services to query/modify `Customer` objects.

Fetching Strategies

- Say a model object P has n related (children) objects $C_1, \ldots, C_n$.
- Eager fetching.
  - When fetching P from the data store, also fetch $C_1, \ldots, C_n$.
  - Can be done with one natural join query or two queries.
  - E.g., fetching an order and all associated order entries.
- Lazy fetching.
  - When fetching P from the data store, do not fetch $C_1, \ldots, C_n$.
  - E.g., fetching a customer but none of the associated orders.
  - To fetch any $C_1, \ldots, C_n$, you must do it yourself, manually.
  - E.g., use an `OrderDAO` object to find orders by customer.
- Semi-lazy fetching.
  - When fetching P from the data store, do not fetch $C_1, \ldots, C_n$.
  - Instead, eagerly fetch *the keys* of every $C_1, \ldots, C_n$.
  - E.g., fetching a customer with the `oid`'s of its associated orders.
  - To fetch any $C_1, \ldots, C_n$, you must still do it yourself, manually.
  - But now you can fetch individual child objects.
  - E.g., use an `OrderDAO` object to find a particular order by key.

Referential Actions

- Consider the following scenario.
  - You fetch a customer object.
  - You update the `cid` value of the customer.
  - You save the customer.
- What happens to the orders of this customer?
  - All orders reference the “old” `cid` value.
  - If the Customer update completes, all orders become orphaned.
  - In other words, the Order table would be left in an inconsistent state.
  - How so? There would be a foreign key constraint violation!

Referential Actions

- General solution.
  - Specify *referential actions* on your tables.
  - The actions determine what happens on updates and deletes to the table.
  - Choices: restrict, set null, set default, restrict, no action, cascade.
- Semantics.
  - You update or delete a tuple.
  - The modification may affect tuples in referencing tables.
  - restrict: if there exist referencing tuples, fail.
  - set null: referencing foreign key values are set to null.
  - set default: referencing foreign key values are set to their default values.
  - no action: update/delete; if any referencing tuple is orphaned, rollback and fail.
  - cascade: update/delete all referencing tuples.

Referential Actions

- Deferring foreign key constraint checking.
  - Specify the constraint as DEFERRABLE INITIALLY DEFERRED.
  - This means relations may be inconsistent during a transaction.
  - But must be put back to a consistent state before the transaction commits.
  - At commit time, the constraints are checked.
  - If any tuple violates some deferred constraint, the transaction fails.
  - Partially solves the problem of updating primary keys.
  - Solves the chicken-and-the-egg problem.
- Orders and OrderEntry foreign keys.
  - By default, use the restrict referential action.
  - Are defined as DEFERRABLE INITIALLY DEFERRED.
  - Cascading behavior, if necessary, is implemented by the respective DAOs.
- Further reading (optional):

  `http://infolab.stanford.edu/~ullman/fcdb/oracle/or-triggers.html`.

Outline

Reading

- Required:
  - Textbook, 9.1 The Three-Tier Architecture.
  - Textbook, 9.6 JDBC.
  - Textbook, 7.1 Keys and Foreign Keys.
- Recommended:

  `http://infolab.stanford.edu/~ullman/fcdb/oracle/or-triggers.html`.