CSE 530A

Review – Part 2

Washington University
Fall 2013
JDBC

• JDBC is a Java API for accessing databases
  – Early versions were JDBC-to-ODBC bridges
    • ODBC is a C API
  – Newer versions are "pure Java"
JDBC

• Generally, any SQL statement can be executed using JDBC
  – Including DDL (data definition language) statements such as CREATE TABLE

• In most situations, though, the database schema is created manually and the applications use only DML (data manipulation language)
  – SELECT, INSERT, UPDATE, DELETE
Overview

1. Initialize driver
   - Only done once
   - Not necessary in JDBC 4.0 and newer
2. Open connection
3. Execute one or more statements
4. Use result set returned from executing statement
5. Commit or rollback transaction if autocommit is off
6. Close all resources (result set, statement, and connection)
Setup

• Old versions of JDBC required explicitly loading the driver class by name

    Class.forName("org.postgresql.Driver");

• JDBC 4.0 and later no longer require this
  – But the appropriate jar must still be on the classpath
    • Each DBMS has its own implementation of JDBC
Connections

• Connection factory needs URL with
  – DBMS type
  – server name
  – port number
  – database name
• As well as username and password

Connection conn = DriverManager.getConnection(
    "jdbc:postgresql://serverName:portNumber/databaseName",
    "username",
    "password");
Connections

• Very important to close connections!

```java
Connection conn = DriverManager.getConnection(
    "jdbc:postgresql://serverName:portNumber/databaseName",
    "username",
    "password");
try {
    // use connection
} catch (SQLException e) {
    // handle error
} finally {
    // very important to close the connection
    try {
        conn.close();
    } catch (Throwable th) {
        // it's annoying to always have to catch this
    }
}
```
Connection Pooling

• Opening a connection is an expensive operation
  – Most applications will use a connection pool
    • Can be tricky to implement: must be careful not to share connection between active threads
      – A thread should:
        » Get an unused connection (should keep the connection from being used by other threads)
        » Execute statements
        » Commit or rollback transaction
        » Release connection back to pool when done
Connection Pooling

• A thread should generally use only one connection at a time
  – A thread can self-deadlock if it starts using a second connection while holding open a transaction on another connection

• Third-party connection pooling libraries exist
  – C3P0
Basic Use

// -- create connection --

Statement stmt = conn.createStatement();
try {
    ResultSet rs = stmt.executeQuery("SELECT * FROM table WHERE ...");
    try {
        while (rs.next()) {
            // use result set
        }
    } finally {
        try {
            rs.close();
        } catch (Throwable th) {
        }
    }
} finally {
    try {
        stmt.close();
    } catch (Throwable th) {
    }
}

// -- close connection --
Closing Resources

• All resources, such as Statement and ResultSet, should be closed as soon as possible after use
  – Free up resources on both client and server
  – An active transaction could be holding locks on the database

• If autocommit is off then the transaction remains open until a commit or rollback or the connection is closed

• If autocommit is on then the transaction completes when the statement completes
  – For SELECT statements, this does not happen until the ResultSet is closed!
Transactions

// -- create connection --

conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
try {
    int count = stmt.executeUpdate("INSERT INTO table ...");
    conn.commit();
} catch (SQLException e) {
    conn.rollback();
} finally {
    try {
        stmt.close();
    } catch (Throwable th) {
    }
}

// -- close connection --
executeUpdate()

• INSERT, UPDATE, DELETE and DDL statements (e.g., CREATE TABLE) use `executeUpdate()` method
  – Returns the number of records affected

```java
int count = stmt.executeUpdate("INSERT INTO table ...");
```
executeQuery()

- SELECT statements use the executeQuery() method
  - Returns a ResultSet which can be iterated over to get the matching records

```java
ResultSet rs = stmt.executeQuery("SELECT * FROM table WHERE ...");
try {
    while (rs.next()) {
        // use result set
    }
} finally {
    try {
        rs.close();
    } catch (Throwable th) {
    }
}
```
ResultSet

• A ResultSet object can be viewed as allowing access to a set of rows
  – Can picture the ResultSet as having an internal cursor which points to the "current" row
  – Starts out initialized to before the first row
    • Must call next() to move to the first row before accessing it
    • next() will return false if there are no more rows
ResultSet

ResultSet rs = stmt.executeQuery("SELECT * FROM table ...");
try {
    while (rs.next()) {

        // use result set

    }
} finally {
    try {
        rs.close();
    } catch (Throwable th) {
    }
}
ResultSet

• Can access columns of the current row by index or by name
  – Column index starts at 1, not 0

  Object obj = rs.getObject(1);

  Object obj2 = rs.getObject("student_id");
ResultSet

• Convenience methods for converting values to compatible types

```java
int student_id = rs.getInt(1);       // returns 0 if field is NULL
String name = rs.getString("name");  // returns null if field is NULL
```

Note that generally index-based access and name-based access are not mixed. Just for illustrative purposes.
ResultSet

• Generally, the developer will know the types of the columns so will know when to use the type-specific getters such as `getInt()`
  – Getters will throw an exception if the type is incorrect

• `getString()` can be used on columns of any type
  – Converts all values to Strings
Metadata

• Information about a ResultSet can be obtained by getting its metadata

```java
ResultSet rs = stmt.executeQuery("SELECT * FROM table ...");

ResultSetMetaData metadata = rs.getMetaData();
int count = metadata.getColumnCount();
for (int i = 1; i <= count; ++i) {
    String label = metadata.getColumnLabel(i);
...
}
```
PreparedStatement

• If a query is parameterized, it is best to use a PreparedStatement

```java
PreparedStatement stmt = conn.prepareStatement(
    "SELECT * FROM users WHERE username = ? AND password = ?"
);

stmt.setString(1, name);
stmt.setString(2, pass);

ResultSet rs = stmt.executeQuery();
while (rs.next()) {
    // use result set
}
```
• What's wrong with doing this?

```java
Statement stmt = conn.createStatement();

String sql = 
    "SELECT * FROM users WHERE username = '\'' + name + 
    '\'' AND password = '\'' + pass + '\'';

ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) {
    // use result set
}
```
SQL Injection

- What happens if the user enters
  - "admin" for the username
  - "password' OR '1' = '1" for the password

```sql
SELECT * FROM users WHERE username = 'admin'
AND password = 'password' OR '1' = '1';
```
SQL Injection

• What happens if the user enters
  – "admin" for the username
  – "password' OR '1' = '1" for the password

```
SELECT * FROM users WHERE username = 'admin'
AND password = 'password' OR '1' = '1';
```

The WHERE clause is always true!
SQL Injection

• Don't fall victim to "Little Bobby Tables"
  – http://xkcd.com/327/
SQL Injection

• A SQL injection vulnerability caused when unsanitized user inputs are used to create SQL statements

```java
String sql = "SELECT * FROM users WHERE username = '" + name + "' AND password = '" + pass + "'";

ResultSet rs = stmt.executeQuery(sql);
```

• Could allow an attacker to run arbitrary SQL code
SQL Injection

- Entering something like "Robert'; DROP TABLE users; --" for the username (and "x" for the password) would give us:

```
SELECT * FROM users WHERE username = 'Robert'; DROP TABLE users; --' AND password = 'x'
```

Note that "--" (two dashes) typically starts a comment in SQL.
Sanitizing Inputs

- Using a PreparedStatement instead of concatenating strings and using Statement can protect against SQL injection, but only if used properly
Sanitizing Inputs

• This

    String sql = "SELECT * FROM users WHERE username = " + name + "' AND password = " + pass + "'";

    PreparedStatement stmt = new PreparedStatement(sql);

    ResultSet rs = stmt.executeQuery();

• Is no safer than this

    String sql = "SELECT * FROM users WHERE username = " + name + "' AND password = " + pass + "'";

    Statement stmt = new Statement();

    ResultSet rs = stmt.executeQuery(sql);
Instead do this

```java
PreparedStatement stmt = conn.prepareStatement(
    "SELECT * FROM users WHERE username = ? AND password = ?"
);

stmt.setString(1, name);
stmt.setString(2, pass);

ResultSet rs = stmt.executeQuery();
```
ResultSets

• By default, PreparedStatements (and Statements) create ResultSets that are FORWARD ONLY
  – This means that they can only be iterated over once
    • Calls to ResultSet.previous() or ResultSet.first() will throw an exception
A common pattern if more flexible access is needed is to read the results into an ArrayList of arrays or ArrayList of HashMaps.

```java
ResultSet rs = stmt.executeQuery();
ResultSetMetaData md = rs.getMetaData();
int columnCount = md.getColumnCount();
ArrayList<Object[]> results = new ArrayList<Object[]>();
while (rs.next()) {
    Object[] row = new Object[columnCount];
    for (int i = 0; i < columnCount; ++i) {
        row[i] = rs.getObject(i + 1);
    }
    results.add(row);
}
// close rs and use results
```
**ResultSets**

```java
ResultSet rs = stmt.executeQuery();
ResultSetMetaData md = rs.getMetaData();
int columnCount = md.getColumnCount();
ArrayList<HashMap<String, Object>> results =
    new ArrayList<HashMap<String, Object>>() {
        while (rs.next()) {
            HashMap<String, Object> row = new HashMap<String, Object>() {
                for (int i = 1; i <= columnCount; ++i) {
                    row.put(md.getColumnLabel(i), rs.getObject(i));
                }
            results.add(row);
        }
    } // close rs and use results
```

- But this can be inefficient for large data sets
Scrollability

• ResultSets can be made *scrollable* for random access
  – Must be supported by the DBMS-specific driver
    • Supported by most drivers

• Three types:
  – ResultSet.TYPE_FORWARD_ONLY
    • Movement can only be forward (default)
  – ResultSet.TYPE_SCROLL_INSENSITIVE
    • Movement can be random, not sensitive to changes to underlying dataset
  – ResultSet.TYPE_SCROLL_SENSITIVE
    • Movement can be random, may reflect changes to underlying dataset
Scrollability

• If a ResultSet's type is *scroll sensitive* then it *may* reflect changes to the underlying dataset
  – Depends on many factors
    • Support by DBMS, transaction isolation level, whether entire dataset is fetched in one call or multiple calls, etc.
  – Generally, should not depend on this behavior working
Scrollability

• The type of the ResultSet must be specified when creating the Statement (*not* when creating the ResultSet)

```java
PreparedStatement stmt = conn.prepareStatement(sql,
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY,
    ResultSet.CLOSE_CURSORS_AT_COMMIT);

...

ResultSet rs = stmt.executeQuery();
```
Concurrency

• Theoretically, ResultSets can be updated
  – But support for this varies by DMBS
• Two options:
  – ResultSet.CONCUR_READ_ONLY
    • The values of a ResultSet are read-only (default)
  – ResultSet.CONCUR_UPDATABLE
    • The values of a ResultSet can be changed via
      ResultSet.updateObject() (or updateString(), updateFloat(), etc.) and flushed back to the database
      with updateRow()
• Generally, I suggest not relying on this functionality
Holdability

• It is sometime possible keep using a ResultSet after a commit
  – Support varies by DBMS and other factors

• Two options:
  – ResultSet.CLOSE_CURSORS_AT_COMMIT
    • The ResultSet is closed on a commit and cannot be used after (default)
  – ResultSet.HOLD_CURSORS_OVER_COMMIT
    • The values of the ResultSet can be accessed after a commit

• Generally, I suggest not relying on this functionality
Limited Use

• In general, I would not use
  • TYPE_SCROLL_SENSITIVE
  • CONCUR_UPDATABLE
  • HOLD_CURSORS_OVER_COMMIT
    – Too many situations where it doesn't work
    – Can interfere with transaction isolation

• However, TYPE_SCROLL_INSENSITIVE can be useful
Data Access Objects

• Data Access Objects (DAOs) are an object-oriented design pattern that provides an abstract interface to persistent storage
  – A DAO provides an interface for accessing persistent data while hiding details of the underlying persistence mechanism
• DAOs typically provide the ability to
  – CREATE
    • dao.createUser(username, password, name, …);
  – RETRIEVE
    • dao.getUser(username, password);
  – UPDATE
    • dao.updateUser(id, username, password, name, …);
  – DELETE
    • dao.deleteUser(id);
CRUD

- Maps fairly straightforwardly to DBMS functionality
  - CREATE
    - INSERT INTO...
  - RETRIEVE
    - SELECT...
  - UPDATE
    - UPDATE...
  - DELETE
    - DELETE...

- But what should operations like CREATE or RETRIEVE return?
ResultSets

• Having the DAOs return ResultSets breaks the independence of the persistence layer from the rest of the code
  – Ties the code to a particular type of storage

• ResultSets are cumbersome
  – Access to data values error prone and dependent on underlying design

```java
ResultSet rs = dao.getUser(username, password);
if (rs.next()) {
    String name = rs.getString(3);
}
```
HashMaps

• HashMaps are more generic than ResultSets
  – Persistent storage layer can be replaced without affecting the rest of the code

• Still error prone
  – Compare
    • HashMap<Object> userMap = dao.getUser(username, password);
    • String name = (String) userMap.get("name");

  – to
    • User user = dao.getUser(username, password);
    • String name = user.getName();
Model Objects

• Model Objects are data-centric objects/classes that forms the data model of an application
  – In Java, usually POJOs (Plain Old Java Objects) with accessor (getter/setter) methods

• Safer than HashMaps as data objects

• Still (mostly) independent of underlying storage mechanism

• Can be expensive to instantiate
Servlets

- A *servlet* is a Java class that responds to HTTP requests
- Used for generating dynamic web content
- Runs in a *web container*
  - Manages lifecycle of servlets
  - Handles mapping of URLs to servlets
- A web server forwards requests to the servlet container
  - Some web containers include web server functionality
HttpServlet

- Servlet classes extend HttpServlet and implement one or more of the doGet, doPost, doPut, doHead, or doDelete methods

```java
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        // do something...
    }
}
```
Servlet URL Mapping

• Used to be done by XML configuration files
  – Still often done that way
• New way uses Java annotations

```java
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
  // ...
}
```

A call to "http://host/webapp/login" will call the appropriate doGet or doPost, etc., method of this class.

For our lab, the webapp name is "lab-2" so the URL path would be "/lab-2/login".
The web server uses the URL path to decide to forward to the servlet container.
The servlet container uses the URL path to decide which web app to forward to.
The web app is configured to forward each URL to a servlet class.
The servlet saves and/or retrieves data and forwards the request to the templating engine.
Servlets

• Request object
  – Only exists for the lifetime of this HTTP request
  – Put data here that is only needed immediately in the JSP

• Session object
  – Exists for the lifetime of the user's session
    • Usually multiple HTTP requests
  – Put data here that is needed for the entire session
  – Data can be retrieved in subsequent servlet calls in the same session
Sessions

• HTTP is a *stateless* protocol
  – Each request is independent
  – How can multiple requests be grouped into a session?
    • Send session ID as a cookie
    • Rewrite URLS to include session ID as a request parameter
Servlets

• The web container and HttpServlet superclass handle the parsing of form data
• Data from form submissions are available as parameters in the request object

String username = request.getParameter("username");
String password = request.getParameter("password");
Servlets

• Servlet classes are instantiated only once
  – The same instance of a servlet class is used for every request to that URL
  – Since requests can happen simultaneously, this means that multiple threads can be using a servlet object at the same time
  – Servlets must be *reentrant*
    • No instance variables!
B+ Trees

- PostgreSQL (and other databases) actually use B+ Trees
  - B+ Trees are a variant of B Trees
    - All of the data are stored at the leaves
    - All of the keys appear at the leaves (in sorted order)
    - Some keys are duplicated in internal nodes
    - Leaf nodes are linked together (in order) to create a linked list
B+ Trees

- $x < 3$
- $3 \leq x < 5$
- $x \geq 5$
Searching

• To find key x in the tree
  – Start at the root node
  – Find the keys in the node where m <= x < n
  – Follow pointer to child node
  – Repeat until leaf is reached
  – Find key x in leaf

• Note that search always goes all the way to the leaves
Inserting

• Find leaf where new key \( x \) belongs
• If leaf is not full then add \( x \) to leaf
• If leaf is full then split leaf and push splitting key up to parent
  – If parent is full then split parent and push splitting key up
  – If we need to add a key to the root and the root is full then split the root and create a new parent as root
• Insert 28
  – Leaf is not full so we can just add it
• Insert 28
  – Leaf is not full so we can just add it
• Insert 70
  – Should go in leaf with (50, 55, 60, 65), but leaf is full
    • Need to split the leaf and then insert
• Insert 70
  – Should go in leaf with (50, 55, 60, 65), but leaf is full
    • Need to split the leaf and then insert
    • 60 is pushed up to the parent
• Insert 95
  – Should go in leaf with (75, 80, 85, 90), but leaf is full
    • Need to split the leaf and then insert
    • 85 is pushed up to the parent, but parent is full
    • Need to split parent node then insert 85
Inserting

- Insert 95
  - Should go in leaf with (75, 80, 85, 90), but leaf is full
    - Need to split the leaf and then insert
    - 85 is pushed up to the parent, but parent is full
    - Need to split parent node then insert 85
Inserting

- Insert 95
  - Should go in leaf with (75, 80, 85, 90), but leaf is full
    - Need to split the leaf and then insert
    - 85 is pushed up to the parent, but parent is full
    - Need to split parent node then insert 85
    - 60 is then pushed up to the new root
Inserting Alternative

- If inserting into a full node with a non-full sibling, we could shift keys from the full node to the non-full sibling
  - Requires modifying parent keys
• Insert 95
  – Should go in leaf with (75, 80, 85, 90), but leaf is full
    • Shift 75 to left sibling
      – Modify key in parent node
    • Insert 95 in newly opened node
Deleting

- Find leaf containing key x
- Delete x from leaf
  - If x was leftmost key then replace x in inner nodes with new leftmost key
- If leaf is not below lower limit
  - If x was a key in the parent then fix parent
  - Propagate change up to root if necessary
- If leaf is below lower limit then ...
  - If sibling is above lower limit then shift key from sibling
    - Adjust keys in ancestors
  - If sibling is at lower limit then merge nodes
    - This removes a key from parent
    - Merge parent with sibling if below limit and propagate up
      - Potentially will remove the current root
Deleting

- Delete 70
  - 70 is removed from leaf (60, 65, 60, 65)
    - No other change needed
• **Delete 70**
  - 70 is removed from leaf (60, 65, 60, 65)
  • No other change needed
• Delete 25
  – 25 is removed from leaf (25, 28, 30)
    • 25 is used in inner nodes so the inner nodes must be fixed
• Delete 25
  – 25 is removed from leaf (25, 28, 30)
    • 25 is used in inner nodes so the inner nodes must be fixed
Deleting

• Delete 60
  – Remove from leaf (60, 65)
    • And replace 60 in inner nodes with new leftmost key
Deleting

- Delete 60
  - Leaf is now below lower limit
    - Can't shift key from sibling as sibling is at lower limit
    - Combine with leaf (75, 80)
Deleting

- Delete 60
  - Extra key now needs to be removed from parent
• **Delete 60**
  – Inner node is now below lower limit
  – Combine with sibling
    • This will eliminate root
Deleting

- Delete 60
  - Inner node is now below lower limit
  - Combine with sibling
    - This will eliminate root
B+ Trees

• For a $b$ order B+ tree (max of $b$ children per node)
  – Find, insert, and delete are all $O(\log_b n)$
  – Space is $O(n)$
  – Range queries can be done in $O(\log_b n + k)$ for a range of $k$
    • Range queries are queries that ask for all elements between two values
  – Elements in a range are already in order
Database Index

- An inner node with \( n \) keys needs \( n + 1 \) pointers
- A leaf nodes with \( n \) keys also needs \( n + 1 \) pointers (\( n \) pointers to the actual data and 1 pointer to its sibling)
- For a key size \( k \) and a pointer size \( p \)
  - A node holding \( n \) keys needs \( kn + p(n + 1) = (k + p)n + p \) bytes
Database Index

- Key size varies depending on type
  - Assume 8 bytes per key (long int)
- Pointer size varies depending on architecture
  - Assume 8 bytes (64 bits)
- A node holding $n$ keys needs a minimum of $16n + 8$ bytes
Database Index

• In practice, disk fetches are much, much more expensive than RAM reads
  – Want to minimize disk fetches
  – No point in reading less than a page at a time from disk

• If we make our B+ tree nodes the size of a page then
  – Page size is typically 4 or 8 KB
  – For 4 KB and 8-byte keys and pointers, a node can hold a maximum of 255 keys
Database Index

- A tree just 3 levels deep can hold more than 16 million keys at the leaves
- A tree just 4 levels deep can hold more than 4 billion keys at the leaves
- Searching for a key in a 4 billion record tables takes just 4 page fetches using an index
  - A sequential scan of the table would take at least 16 million page fetches
Object-Relational Mapping

• An object-relational mapping tool attempts to bridge the divide between OO languages and relational databases
  – We did this by hand in lab 2. There are tools and libraries that can do it for us

• ORM tools exist for many different languages
  – ActiveRecord (Ruby on Rails)
  – ADO.NET Entity Framework (Microsoft)
  – Django ORM (Python)
  – JPA, Hibernate (Java)
Entity Classes

• Annotations or XML configuration files are used to map classes and fields to tables and columns
  – @Entity marks the class as a model object
  – @Table specified the corresponding database table
  – @SequenceGenerator defines a database sequence to use for auto-generated ID fields

```java
@Entity
@Table(name = "employees")
@SequenceGenerator(name = "EMPLOYEE_SEQ", sequenceName = "employees_employee_id_seq")
public class Employee implements Serializable {
    private static final long serialVersionUID = 6803370824626024108L;
    ...
}
```
Entity Classes

- Must have a default (no argument) constructor
- Must be a top-level class (not an inner class or an interface)
- Must not be final
- Should implement Serializable

```java
@Entity
@Table(name = "employees")
@SequenceGenerator(name = "EMPLOYEE_SEQ", sequenceName = "employees_employee_id_seq")
public class Employee implements Serializable {
    private static final long serialVersionUID = 6803370824626024108L;

    ...}
```
Entity Classes

• Annotations can be placed on either the field declaration or the getter method (preferred) to map fields to columns

```java
private Long id;

@Id
@GeneratedValue(strategy = GenerationType.AUTO, generator = "EMPLOYEE_SEQ")
@Column(name = "employee_id"
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

...```
Entity Classes

• @Id indicates primary key field
• @GeneratedValue indicates how to create database generated values
  – GenerationType.AUTO is the most DBMS agnostic
• @Column maps the field to a database column

```java
private Long id;

@Id
@GeneratedValue(strategy = GenerationType.AUTO, generator = "EMPLOYEE_SEQ")
@Column(name = "employee_id")
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}
...
```
Entity Classes

• Basic Java types can be mapped with a simple @Column
  – including: primitive types, primitive wrapper types, String, Date, byte[], other entity types
• Mapped fields must have standard getter and setter methods

```java
private String name;

@Column(name = "name")
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

...```
Entity Classes

• Relationships between entities can be
  – one-to-one
  – one-to-many
  – many-to-one
  – many-to-many

• Relationships can be
  – unidirectional
    • One entity has a reference to another, but not vice-versa
  – bidirectional
    • Each entity has a reference to the other
Entity Classes

- Collection-valued fields must use generic collection interfaces
  - Collection, Set, List, Map

```java
public class Department {
    ...
    public List<Employee> getEmployees() { ... }
    public void setEmployees(List<Employee> employees) { ... }
    ...
}
```
Entity Classes

• `@OneToOne`
  – Defines a one-to-one mapping between entities
  – Uses `@JoinColumn` to specify foreign key column

```java
public class Employee {

  private Address address;

  @OneToOne
  @JoinColumn(name = "address_id")
  public Address getAddress() {
    return address;
  }

  public void setAddress(Address address) {
    this.address = address;
  }

  ...
}
```
Entity Classes

- **@ManyToOne**
  - Defines a many-to-one mapping between entities
  - Uses [@JoinColumn](#) to specify foreign key column

```java
public class Employee {
    ...

    private Department department;

    @ManyToOne
    @JoinColumn(name = "department_id")
    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department department) {
        this.department = department;
    }

    ...
}
```
Entity Classes

• `@OneToMany`
  – Defines a one-to-many mapping between entities
  – `mappedBy` indicates field (not column) in target class

```java
public class Department {
    ...

    private List<Employee> employees;

    @OneToMany(mappedBy = "department", targetEntity = Employee.class)
    public List<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(List<Employee> employees) {
        this.employees = employees;
    }

    ...
}
```
Entity Classes

- **@ManyToMany**
  - `@JoinTable` declares the mapping table for the many-to-many relationship
  - `@JoinTable` can also be used for one-to-many/many-to-one relationships with a separate mapping table

```java
public class Employee {
    ...

    private List<Project> projects;

    @ManyToMany
    @JoinTable(name = "employee_project_map",
                joinColumns = { @JoinColumn(name = "employee_id") },
                inverseJoinColumns = { @JoinColumn(name = "project_id") })
    public List<Project> getProjects() {
        return projects;
    }

    public void setProjects(List<Project> projects) {
        this.projects = projects;
    }

    ...
}
```
Entity Classes

• A bidirectional relationship is simply one where both entity types have a reference to the other
  – Employee and Department in the previous slides defined a bidirectional relationship

• A unidirectional relationship is simply where one entity type has a reference to another but the reverse mapping is not defined
Hibernate

• Hibernate is library which implements the JPA API
  – Just one of many JPA implementations

• Hibernate pre-dates the existence of JPA
  – Personally prefer it over the JPA
Hibernate

• If we're going to use hibernate then we might as well use it everywhere
  – Create a SessionFactory instead of an EntityManagerFactory
  – Get Session objects instead of EntityManager objects
  – Use hibernate's query syntax
SessionFactory

public class ContextListener implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent event) {
        ServletContext sc = event.getServletContext();
        Configuration configuration = new Configuration();
        configuration.configure();
        ServiceRegistry serviceRegistry = new ServiceRegistryBuilder()
            .applySettings(configuration.getProperties()).buildServiceRegistry();
        SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);
        sc.setAttribute("hibernateSessionFactory", sessionFactory);
        LOGGER.info("Created hibernate session factory");
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) {
        ServletContext sc = event.getServletContext();
        SessionFactory sessionFactory = (SessionFactory) sc.getAttribute("hibernateSessionFactory");
        if (sessionFactory != null) {
            sessionFactory.close();
        }
    }
}
private void process(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
  ...

  SessionFactory sessionFactory =
    (SessionFactory) request.getServletContext().getAttribute("hibernateSessionFactory");

  Session session = sessionFactory.openSession();
  Transaction tx = null;

  try {
    tx = session.beginTransaction();

    User user = UserDao.retrieveUser(session, username);

    ...

    request.getRequestDispatcher("WEB-INF/jsp/Welcome.jsp").forward(request, response);
    tx.commit();
  } catch (Exception e) {
    LOGGER.log(Level.SEVERE, "error retrieving user", e);
    request.getRequestDispatcher("WEB-INF/jsp/Error.jsp").forward(request, response);
    tx.rollback();
  } finally {
    session.close();
  }
}
Accessing Entities

• Hibernate's pseudo-SQL syntax is a little different
  – Notice the similar use of placeholders, though
• The **Query** is now an **org.hibernate.Query** instead of a **javax.persistence.Query**

```java
Query query = session.createQuery("from User user where user.username = :name");
query.setString("name", name);
User user = (User) query.uniqueResult();
```
Accessing Entities

• Hibernate also has the equivalent of `find` to retrieve by ID

```java
public static User retrieveUser(Session session, Long id) {
    return (User) session.get(User.class, id);
}
```
Creating Entities

• Create the entity object but do not set the ID (assuming an auto-generated ID)

• Call `session.save`
  – The object will be assigned a new ID and saved to the database

```java
public static User createUser(Session session, String username, String password) {
    User user = new User();
    user.setUsername(username);
    user.setPassword(password);
    session.save(user);
    return user;
}
```
Updating Entities

- Update the fields on the entity object but do not change the
- Call `session.save`
  - The row in the database will be updated

```java
public static void updateUser(Session session, User user) {
    session.save(user);
}
```
Deleting Entities

• **Call** `session.delete`

```java
public static void deleteUser(Session session, User user) {
    session.delete(user);
}
```
Lazy vs Eager Loading

- When using a relationship between Entity objects Hibernate will
  - Lazy load collections (@OneToMany and @ManyToMany)
  - Eager load single-valued references (@OneToOne and @ManyToOne)
Lazy vs Eager Loading

• Lazy loading means that the SQL to load the objects is not executed until they are actually accessed
  – Hibernate accomplishes this through the use of proxy objects
• An attempt to access a lazily-loaded field after transaction end will result in an exception
  – To be safe, all Entity objects should be considered invalid after transaction end
Lazy vs Eager Loading

• The fetch type can be changed via the annotations
  – Be careful! In this example, all of the Employee objects in the Department will be fetched from the database

```java
public class Department {

  ... 

  private List<Employee> employees;

  @OneToMany(mappedBy = "department", targetEntity = Employee.class,
           fetch = FetchType.EAGER)
  public List<Employee> getEmployees() {
    return employees;
  }

  ...

```
N+1 Selects Problem

• Assume we have a one-to-many relationship from Department to Employee and a one-to-one relationship from Employee to Address

• Assume we set the FetchType of employees in Department to EAGER

• What happens when get a Department?
N+1 Selects Problem

• What happens when get a Department?
  – One SELECT to get the department record
  – One SELECT to get all of the employee records for that department
  – One SELECT for each of the employee records to get its address record

• This is the N+1 selects problem
N+1 Selects Problem

- Changing the FetchType of employees in Department partially solves the problem
  - As long as we don't access employees we're OK, but once we do we get the extra SELECTs for the addresses

- We could change the FetchType of address in Employee to LAZY
  - When the employees are loaded the addresses are not
  - But what if we actually need the addresses?
    - Every address we access causes a SELECT
Join Fetch

• We can use join fetch in HQL to force the use of joins instead of multiple selects

"from Department dep left join fetch d.employees emp left join fetch emp.address where ..."
Entity Lifetime

• As discussed before, entity objects are only valid while in a Hibernate session
  – When the session is closed, the entity objects are invalid
  – This means we cannot store the User object in the session context like we did in lab 2

• Note that we are talking about two different "sessions" here
  – Hibernate Session
  – HttpSession
Entity Lifetime

• HttpSession
  – Created when a browser connects to our web app
  – Exists until explicitly invalidated (see LogoutServlet) or a timeout occurs
    • A timeout occurs if the browser does not send another request for some time
      – Configurable via tomcat setup
  – Can store things in the HttpSession that we want to exist across multiple requests
Entity Lifetime

• Hibernate Session
  – Generally use a session-per-request pattern
    • Wraps a database connection
  – Created at the beginning of request processing, closed at the end
  – Should **not** be shared by multiple requests
  – Entity objects are only valid while a Hibernate Session is open
Entity Lifetime

• Entity objects cannot be stored in the HttpSession
  – Objects would be invalid when accessed in subsequent requests
  • There actually is a way to detach an Entity object from a persistence session and reattach to another persistence session
    – Need to be careful about changes and conflicts
    – Should generally be avoided
Query Implementation

• Assume we have the table

    CREATE TABLE students (
        sid BIGSERIAL PRIMARY KEY,
        name TEXT NOT NULL,
        year TEXT NOT NULL,
        major TEXT
    );

• What must the DBMS do to answer a query like

    SELECT * FROM students WHERE year = '2010';
Query Implementation

• If we assume
  – the table is not ordered by the \texttt{year} field and
  – there are not any indexes on that field
• then we must do a table scan
  – A table scan scans the entire table looking for records that match the WHERE predicate
Query Implementation

• Since the time required for disk IO usually dominates all the other time, we typically measure cost as the number of page fetches required to execute a query
  – For our example, if we assume 10,000 records and 100 records a page then the query would require 100 page fetches
Query Implementation

• What if we could somehow keep the table in sorted order by the year field?
  – We could then do a binary search to find the first record with the value '2010' and then read the matching record sequentially
    • If we do the binary search by pages then the cost would be $\log_2 100 \approx 6.6$ plus the number of pages with matching records
Query Implementation

- But it is generally too cost prohibitive to keep a table in sorted order
  - For performance reasons, inserted and updated rows are added to the end of the table rather than in sorted order
- And what about searching on a different field?
Query Implementation

• If we know that most searches on a table will be by a particular field then we could try to keep the table ordered by that field
  – Many DBMSs have the ability to *cluster* a table
    • Reorders the table on disk by the specified field
    • An expensive operation with usually requires a lock on the table
    • A frequently modified table (inserts and updates) can quickly get out of order
    • Using a *fill factor* of less than 100% when clustering will cause the DBMS to leave space in the pages to use later to try to keep in order as long as possible
Query Implementation

• What if the query was

```sql
SELECT * FROM students WHERE sid = 12345;
```

• `sid` is the primary key, which means there is an index on that field by default
  – If we assume a B+ tree index of order 101 (that is, between 50 and 100 keys per node) then
    • For 10,000 records, or 100 pages, the depth of the tree will be at most 3
    • So, 3 pages for the index lookups and 1 page for the record
      – Only one matching record since `sid` is a key
Query Implementation

• What if we created a B+ tree index on the year field?
  – 3 pages to find the first matching key in the index
  – Some number, say k, pages for matching index leaves
  – Some number of pages for the matching records
    • Best case: matching records are packed tightly in pages
    • Worst case: each matching record is on a different page
Multi-Key Indexes

• What if our queries often contain a conjunction?

SELECT * FROM students WHERE year = '2010' and major = 'CSE';

• We could create a multi-key index
  – Orders first by the first field and then by the second
  – Conceptually, can think of the index key as the concatenation of the fields
Multi-Key Indexes

• If we create an index on multiple fields, which field should we put first?
  – Index can only be used if the conjunction of terms contains the first index field
  – Generalizing, if an index is created on fields \((x, y, z)\) then it can be used for queries on
    • \(x\)
    • \(x\) and \(y\)
    • \(x\) and \(y\) and \(z\)
  – But not on
    • \(y\)
    • \(z\)
    • \(y\) and \(z\)
Disjunctions

• What if our query contains a disjunction?

```
SELECT * FROM students WHERE year = '2010' or major = 'CSE';
```

• Could use a table scan to find matching records
• If an index exists on year and another index exists on major then the results of scanning both indexes could be combined
  – But duplicates would somehow have to be eliminated
• If an index exists on one field but not the other then a table scan would need to be done anyway
  – So might as well not do the index scan
Sorting

• Sorting is often necessary in a DBMS
  – Explicitly requested order in query
  – Intermediate step in implementing joins (more on this later)
Sorting

• If the data set that needs sorting is small enough then it can be sorted in memory
  – What is "small enough"? Depends on amount of memory available
• In memory sorting is usually done using either quicksort or merge sort
Sorting

<table>
<thead>
<tr>
<th></th>
<th>Average case time</th>
<th>Worst case time</th>
<th>Space (extra)</th>
</tr>
</thead>
<tbody>
<tr>
<td>quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

- Despite worst-case time of $O(n^2)$ quicksort is often preferred for in-memory sorting
  - Worst-case is rare
  - No extra space other than call stack
Merge Sort

- Merge sort is often learned as a recursive algorithm, but can also be done iteratively
  - Requires additional buffer of size $n$
  - Pass 1: start by merging pairs of values into ordered sets of size 2
  - Pass 2: merge pairs of size-2 sets into ordered sets of size 4
  - Repeat until single ordered set of size $n$ is reached
  - Each pass copies values from one buffer to the other
Merge Sort

Start: 4 2 8 1 5 6 3 7

Pass 1: 2 4 1 8 5 6 3 7

Pass 2: 1 2 4 8 3 5 6 7

Pass 3: 1 2 3 4 5 6 7 8
External Sorting

• What if the data set to sort is too large to fit in available memory?
  – Can use an external merge sort
External Sorting

- External merge sort
  - Assume we have $n$ values to sort and three buffers of size $b$ (where $b < n$).
  - Pass 1: read $b$ values at a time from the set of $n$, sort using an in-memory sort, write sorted set of $b$ values to a temporary file.
    - Results in $\left\lceil \frac{n}{b} \right\rceil$ files of size $b$.
    - Can potentially do this in parallel using the three buffers.
  - Pass 2: read two of the sorted temp files of size $b$ and merge into a sorted temp file of size $2b$. Repeat for each pair of size $b$ temp files.
    - Use two of the size $b$ buffers as input buffers and the other buffer as output buffer.
    - Note that the output buffer will need to be flushed halfway through.
    - Results in $\left\lceil \frac{n}{2b} \right\rceil$ files of size $2b$.
  - Repeat until single sorted file of size $n$ is reached.
Double Buffering

- On modern machines, disk IO can occur in parallel with CPU computation
- Can take advantage of this to avoid blocking and waiting for IO to complete by using double buffering
  - Consider the output buffer in pass 2
    - When the buffer is full it needs to be written to disk before more computation can be done. Need to block waiting for IO to complete before continuing to write to the output buffer. (Problem is worse is subsequent passes when more data needs to be output.)
    - If we used two output buffers then we could write to the second one while the first is being flushed to disk.
    - Could use double buffering on input as well.
- Requires twice the amount of buffer space (or reading/writing half the amount of data at a time)
Increasing Fan-In

• With the external merge sort we'd like to reduce the number of passes as much as possible
  – Each pass requires reading in and writing out the entire set of $n$ values
• We can reduce the number of passes by merging together more than two temp files at a time
  – Merging four files at a time instead of two requires half the number of passes
• Trade-off: increasing the number of files merged at a time requires increasing the number of input buffers, so less of each file can be read at a time
  – But since we're reading each file sequentially we can effectively stream the files, especially if using double buffering
Implementing Queries

• A query on a single table can be implemented using either
  – table scan: scan the entire table for matching records
  – index scan: if the predicate includes a term for which an index exists

• What about joins?
Implementing Joins

• Suppose we change our students table and add a majors table as shown

CREATE TABLE students (  
    sid BIGSERIAL PRIMARY KEY,  
    name TEXT NOT NULL,  
    year TEXT NOT NULL,  
    mid INTEGER  
    REFERENCES majors(mid)  
);  

CREATE TABLE majors (  
    mid SERIAL PRIMARY KEY  
    title TEXT NOT NULL  
);  

• And we execute a join query such as

SELECT * FROM students INNER JOIN majors USING (mid);
Nested Loop Join

- In a nested loop join
  - A scan is made of one table
  - For each record in the one table, a scan is made of the other table to find records that match on the join clause
- Runs in time $O(mn)$ where $m$ is the number of records in one table and $n$ is the number of records in the other
Block Nested Loop Join

• Since we are likely reading the records of the first table a block at a time, we can scan the records of the second table once per block instead of once per record
Nested Loop Join

• When implementing an inner join or natural join we could further optimize by doing a single scan of the larger table and making the smaller table the one that is scanned repeatedly
  – Big gain if the smaller table can fit in memory

• When implementing an outer join then the "outer" table needs to be the outside one scanned
  – So we can include the records in the output that do not match any records in the "inner" table
Nested Loop Join

• We can potentially optimize further by filtering by the WHERE predicate first

```sql
SELECT * FROM students LEFT OUTER JOIN majors USING (mid)
WHERE students.year = '2010';
```

– If there is an index on `students(year)` then we can do an index scan to get the matching records from `students` and then only scan the `majors` table for those records

• And if there is another index on `majors(mid)` then we can use an index scan instead of a table scan on the `majors` table
Pipelining

• If we are first getting the students records using an index scan and then for each of those (or block of those) scanning the majors table (or index) then we have two choices
  – Choice 1: materialize the matching students records, either storing in memory (if small enough) or writing to a temporary file
  – Choice 2: as we get matching students records from the index, immediately scan the majors table for each students record (or block)
    • We never actually create the intermediate set of students records that match the WHERE predicate
    • This is called pipelining
Merge Join

• A *merge join* (also called a *sort-merge join*) is accomplished by
  - sorting the two relations by the join attributes
  - merging the two sorted relations

• For example, if we
  - sort the *students* records by *mid* and
  - sort the *majors* records by *mid* then
  - we can merge the two sorted relations in a single pass over both
Merge Join

• A merge join often requires *materializing* the sorted sets
  – Though if the join attributes match an index then that index can potentially be used to get the records in sorted order
  • For example, since the *majors* table has an index on mid we can use that to get the *majors* records in the order we need
    – However, we would still need to materialize the *students* records sorted by mid
Merge Join

• We can potentially reduce the size of the sets we need to materialize by filtering by the WHERE predicate first
  – For example, we only need to materialize and sort the students records where year = '2010'

• Pipelining is generally not an option going into a merge join as we need to materialize the sets before sorting
  – But we could potentially pipeline the output of the merge into the next stage
Hash Join

- To implement a hash join
  - Create a temporary hash table out of one relation (usually the smaller) using the join attributes as the key to the hash function
  - Scan the other table using the hash table to find matching records

- Similar to a nested loop join except we create a hash table of the "inner" table first

- Only works well if the hash table is small enough to keep in memory
Partition Hash Join

• A partition hash join can be used with larger tables
  – For each of the two tables being joined
    • Create a hash table using the join attributes as the key to the hash function
    • Partition the hash table using the hash function, storing the different partitions in different temporary files
  – For each partition, join with the matching partition from the other hash table
Bitmap Scans

- When performing an index scan it is usually necessary to also retrieve the actual matching records from the table
  - Most often, the records in the table file are not in the order accessed
    - This leads to many random accesses in the table file
  - Sequential file access is generally faster than random access
    - If we could access the table file in sequentially then performance could be increased
Bitmap Scans

• In a *bitmap scan*, a bitmap is created while performing an index scan
  – The bitmap indicates the position of the matching records in the table file (also called the heap)
  – The index scan step is called a *bitmap index scan*

• A sequential scan of the heap (table file) is then done, retrieving the records indicated by the bitmap
  – The table scan step is called a *bitmap heap scan*
Lossy Bitmap Scans

• If a table is too large to efficiently create a bitmap of the records then a bitmap of the pages can be used
  – The bitmap indicates which pages contain records that match the predicate
• The bitmap heap scan then fetches the indicated pages in sequential order
  – Must recheck the records in each fetched page to find the ones that match the predicate
Disjunctions

• Bitmap scans can also be helpful in implementing queries with disjunctions
  – If we have
    • a query like "... WHERE year < '2010' OR major = 'CSE'"
    • and separate indexes on year and major
  – then we could
    • do an index scan on year to create the bitmap
    • do an index scan on major and set the matching bits in the bitmap
    • do the bitmap heap scan
Query Planning

• We now have three ways of scanning a relation
  – table scan
  – index scan
  – bitmap index scan

• When should each method be used?
  – Answer is not always obvious
Query Planning

• Need to access every record in the relation
  – Do a table scan. Accessing the index is just unnecessary overhead.

• No appropriate index is available
  – Table scan is only option.

• Index is available
  – Best method depends on size of table and percentage of matching records
    • If the table is small enough then it might be fastest to just do a table scan
    • If the number of matching records is small then an index scan is probably fastest
    • If the number of matching records is above some limit then a bitmap scan or table scan might be fastest
Query Planning

• What is the limit at which we should switch from index scan to bitmap or table scan?
  – No hard and fast rule
  – Some references say just 5% or greater matching records
    • Seems a little low to me
Query Planning

• Three ways of scanning relations
  – table scan
  – index scan
  – bitmap index scan

• Three ways of joining relations
  – nested loop join
  – merge join
  – hash join
Query Planning

- Selecting from a single table gives 3 possible options
- Joining two tables gives $3^3$ options
  - 3 for selecting from each table
  - 3 for the actual join
- Each additional table includes multiplies the possibilities by $3^2$
  - 3 for selecting from the addition table
  - 3 for joining
- In addition, there are often multiple possibilities for when to handle the predicates
- How does the DBMS choose a query plan?
Query Planning

• A query plan can be represented as a tree
  – leaves are scans
  – inner nodes are joins, sorts, materializations

```
nested loop join
  condition: students.mid = majors.mid
```

```
table scan on students
  condition: year='2010'
```

```
index scan on majors
  condition: title='CSE'
```
Query Planning

• Joins can be done in different orders
  – For example, if we have
    • x inner join y inner join z
  – we'll get the same result if
    • x join y is done first and then joined to z or
    • y join z is done first and then joined to x
Query Planning

- It is the job of the query planner to pick the minimum cost plan
  - Two issues
    - The number of possible plans grows exponentially in the number of joined tables
    - How is the cost of a plan calculated?
Query Planning

- We can assign a cost to a plan node if given
  - The cost of the child nodes
  - The size of the sets generated by the child nodes
  - The cost of the actual operations of the node
Query Planning

• What do we mean by "cost"?
  – Time! We're looking for the plan that takes the shortest time
  – The cost of an operation is generally calculated as a function of the number of pages fetches/writes and the CPU time spent
    • IO time usually dominates, but CPU time also plays a part
    • Also often subdivided into sequential page and random page access
    • Number of page accesses and CPU time directly dependent on size of data
Query Planning

• For a given plan node, the query planner tries to estimate both the cost (in some unit of time) and the size of the resulting set based on the type of operation and the size of the input sets

• The cost of a node in the plan tree includes the cost of its children
  – So the cost of the root node is the total estimated cost of executing the plan
Query Planning

• With the number of possible plans increasing exponentially with the complexity of the query, it can quickly become impossible for a query planner to evaluate all possible plans
  – The query planner itself needs to be fast
• Since the planner is choosing among various trees which often have identical subtrees, dynamic programming techniques (e.g., memoization) can be used
  – Still may not have enough time to evaluate all possible plans
Query Planning

• Some query optimizers favor left-deep trees
  – A left-deep tree adds each new table as the right child of a join
Query Planning

• The nodes can be made interchangeable by using an *iterator* interface
  – Each child supplies the set resulting from its operation as an iterator to its parent

• Using an iterator interface also facilitates pipelining
  – A parent node doesn't need to know if the child has a materialized set or is providing elements on demand
Query Planning

• Part of estimating the cost for a node requires estimating the number of records resulting from a scan or join
  – For a select without a predicate we could use the number of rows in the table
  – For an equals predicate on a unique column we know there will be at most one
  – How do we know how many rows a scan with a predicate will return before actually doing the scan?
Query Planning

- The query planner tries to estimate the *selection factor* of predicates by using statistics on the distribution of values for each column
  - Basically keeps a *histogram* for each column
Query Planning

• If the statistics kept by the DBMS for a column are significantly wrong then the performance of queries using that column can be greatly impacted
  – The query planner could have vastly incorrect cost estimates and therefore pick a bad plan

• Tables with lots of modifications (inserts/updates/deletes) should periodically be analyzed
  – The DBMS will scan the table and regenerate its histogram
Sargable Predicates

- A predicate is **sargable** if it does not interfere with the DBMS's ability to use an index
  - Applying a function to a column in the WHERE clause usually makes it non-sargable
    - Non-Sargable: ... WHERE year(date) = 2012
    - Sargable: ... WHERE date BETWEEN '2012-01-01' AND '2013-01-01'
    - Non-Sargable: ... WHERE substring(model, 6) = 'Toyota'
    - Sargable: ... WHERE model LIKE 'Toyota%'

("sargable" comes from Search ARGument ABLE)