3-Tier Architectures
(or 3-Tier Applications)

Adapted from Chuck Cusack’s Notes

Simple Programs
• The programs you have written so far have probably had the following properties:
  – Input/output was text-based, GUI, and/or with files
  – Ran locally—that is, they had to be downloaded to the machine on which you (or your T.A.) ran them
  – The data had to be on the same machine as your program
  – Were not able to be run simultaneously by more than one person.
  – Had no interaction with other programs
• In the real world, most software is much more complicated than this

Databases and the Web
• Many real applications are different than what you have programmed in two important ways
  – They make extensive use of databases to store data
  – They are accessible from anywhere by multiple people simultaneously via the World Wide Web
• Sites like amazon.com and ebay.com are entirely database driven—that is, each time you go to the site, what you see is determined by data in their database
• As we will see, applications that use databases and are web accessible can be implemented using a 3-tier architectural model

3-Tier Architectures
• Definition: A 3-tier architecture is one which has
  – The database tier manages the database
  – The middle tier contains most of the logic and communicates between the other tiers
  – The client tier is the interface between the user and the system
• Definition: An n-tier architecture is one which has n tiers, usually including a database tier, a client tier, and n-2 tiers in between.

Thin-Client 3-Tier Models
• The thin-client 3-tier model has these tiers:
  – The database management system (DBMS)
  – The main application software
  – A web browser
• Examples
  – http://cse.unl.edu/~scott/teach
  – http://contests.unl.edu
  – http://ebay.com/
  – http://amazon.com/
  – Lotus Notes Web Client
(These may actually be n-tier)

Thick-Client 3-Tier Models
• The thick-client 3-tier model has these tiers:
  – The database management system (DBMS)
  – The main application software
  – Some sort of interface software which must be installed on each client machine
• Examples:
  – Lotus Notes
  – Desktop applets that display weather, etc.
  – RealPlayer and other applications that download CD information from the Web
Another 3-Tier Model

- Another common model has these tiers:
  - The database management system (DBMS) and a persistence manager which controls all data flow into and out of the database
  - The main application software
  - A GUI (thin or thick)

- The main difference here is that the main application software is not allowed to interact directly with the database

- You could also think of this as a 4-tier architecture:
  - The database management system (DBMS)
  - A persistence manager
  - The main application software
  - A GUI (thin or thick)

2- and n-Tier Models

- The 2-tier model is more simple, but more limited, than a 3-tier model, and often includes
  - The database management system (DBMS)
  - The main application software, including GUI

- Here, the entire application is generally run on the client machine (certainly a thick-client)

- In some contexts, the 2-tier model is also known as the client-server model, where the server can be something other than a database

- In general an n-tier model will have
  - The database management system (DBMS)
  - (n-2) application layers
  - A GUI (thin or thick)

n-Tier Questions

- The following are important questions one must ask when thinking about n-tier architectures
  - How many tiers should be used?
  - What tasks should be done by each tier? In other words, how exactly should the layers be divided?
  - Should I use thin or thick clients?
  - Should the application be web-accessible?
  - How should connections to the database be managed?
  - What database management system (DBMS) should be used?
  - What languages(s), platform(s), and software should the system use?

n-Tier Answers

- The purpose of these notes is not to
  - Present clear answers to all of the questions on the previous slide
  - Be the authoritative source for information about n-tier architectures
  - Make you an expert in n-tier architectures

- Rather, the purpose is to
  - Introduce you to the concept of n-tier architectures
  - Get you to start thinking about the issues involved
  - Give you partial answers to some of the question

Database Choices

- There are many popular database management systems (DBMSs), including
  - IBM DB2
  - Oracle
  - Microsoft SQL Server
  - Microsoft Access
  - MySQL

- Which one you should use depends on many factors, including number of expected users, size of the application and/or the database, budget, etc.

- Fortunately, the interfaces to these DBMSs have a lot in common, so if you learn to use one, most of what you learn is transferable to the others

Middle Tier Choices

- Almost anything is possible, with some common choices being
  - Java
  - JSP
  - PHP
  - C++
  - Perl
  - Visual Basic
  - C#
  - .NET
  - ASP

- However, whether or not the client will be thin or thick will influence this choice

- For a thin client, the obvious middle tier choices are
  - Java applets, JSP, PHP, ASP, and Perl

- Of course with all of these, HTML is involved as well
Client Choices

- Thin clients are generally web browsers, so the important choice was made in the middle tier.
- For thick clients, we might use
  - Java applications
  - C++ applications with GUI provided by
    - MFC (Microsoft Foundation Classes)
    - Tcl/Tk (Tool command language)
    - GTK (GIMP Toolkit)
    - Qt

Examples

- You could build a 3-tier application which has
  - Oracle DBMS, C++ middle, and C++/MFC client
  - MySQL DBMS, PHP middle, and web browser client
  - IBM DB2 DBMS, JSP middle, and Java applet client
  - MySQL DBMS, Java applet middle, and web browser client
  - MySQL DBMS, ASP middle, and web browser client

Our Choices

- In this class, we will use the following:
  - Thin-client 3-tier architecture model
  - Database tier: MySQL
  - Middle tier: PHP/HTML and Java applets
  - Client tier: Your favorite web browser
- It is important that you realize that
  - Thin clients are not always the best choice
  - Three is not always the best number of tiers
  - Our database and language choices are not the best or only choices, but are reasonable for this class
  - The approach we take to design and implement 3-tier applications is not the only, and not necessarily the best way

References

- Hugh E. Williams & David Lane, *Web Database Applications with PHP and MySQL*, 2nd Ed., O’Reilly, 2004
A Brief Introduction to Relational Databases

Adapted from Chuck Cusack’s Notes

Course Roster File #1
- Consider an instructor who wishes to store a list of his courses and students enrolled in those courses.
- Below is how one might store the information in a file
- There are several problems with this:
  - Some information is repeated several times
  - Some entries are incomplete
  - It is hard to see who is in what course
  - It is hard to see how many students or courses there are
  - Updating student information can be problematic because of repeats

<table>
<thead>
<tr>
<th>Course</th>
<th>Professor</th>
<th>Section</th>
<th>CourseID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theatre</td>
<td>John Cusack</td>
<td>543210987</td>
<td>Spring 2004</td>
</tr>
<tr>
<td>Computer Science</td>
<td>Computer Engineering</td>
<td>Data Structures and Algorithms</td>
<td>CSCE310</td>
</tr>
<tr>
<td>Computer Engineering</td>
<td>Phil Philson</td>
<td>505555555</td>
<td>Fall 2003</td>
</tr>
<tr>
<td>Art</td>
<td>Computer Engineering</td>
<td>Data Structures and Algorithms</td>
<td>CSCE155</td>
</tr>
<tr>
<td>Computer Science</td>
<td>Data Structures and Algorithms</td>
<td>Data Structures and Algorithms</td>
<td>CSCE156</td>
</tr>
</tbody>
</table>

Course Roster File #2
- Could store the information so it is a little easier to get rosters for each course as shown below
- There are still several problems with this format
  - Student records are still repeated, making updating tough
  - It is hard to see how many courses a student is enrolled for
  - It is also hard to see how many students there are total

<table>
<thead>
<tr>
<th>Course</th>
<th>Professor</th>
<th>Section</th>
<th>CourseID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCE310 Data Structures and Algorithms</td>
<td>Spring 2004</td>
<td>111111111</td>
<td>Fall 2003</td>
</tr>
<tr>
<td>CSCE155 Intro to Computer Science I</td>
<td>Fall 2003</td>
<td>123456789</td>
<td>Spring 2004</td>
</tr>
<tr>
<td>CSCE156 Intro to Computer Science II</td>
<td>Spring 2004</td>
<td>505555555</td>
<td>John Cusack</td>
</tr>
</tbody>
</table>

Course Roster: A Better Solution
- It turns out that no matter how you store the information in a simple text file, there will be problems.
- The bottom line is that sometimes storing things in a simple text file is not the best way
- A better solution would use a database to store the information.
- As we will see, the benefits will include
  - Duplication is minimized
  - Updating information is easier
  - Getting information like lists for each course or number of courses or roster for students
  - Getting more complex information can also be easily accomplished

Course Roster Database
- One way to store the same information in a database is to use the following tables
- We will discuss the design aspect of the database later

<table>
<thead>
<tr>
<th>Students</th>
<th>Enrollments</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnrollID</td>
<td>FirstName</td>
</tr>
<tr>
<td>1</td>
<td>John</td>
</tr>
<tr>
<td>2</td>
<td>Mary</td>
</tr>
<tr>
<td>3</td>
<td>Phil</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Courses</th>
<th>Enrollments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CourseID</td>
<td>Department</td>
</tr>
<tr>
<td>1</td>
<td>CSCE</td>
</tr>
<tr>
<td>2</td>
<td>CSCE</td>
</tr>
<tr>
<td>3</td>
<td>CSCE</td>
</tr>
</tbody>
</table>

Database Terminology
- Relational databases store information in a set of tables
- Each table has a unique name which describes what type of information is stored in the table
- Each column (field, attribute) of the table stores a single piece of information
- Each row (record) of the table represents one object
- One or more columns in each table are the primary key, which uniquely identifies each record, and is indicated by underlining the attribute name

<table>
<thead>
<tr>
<th>Courses</th>
<th>Enrollments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CourseID</td>
<td>Department</td>
</tr>
<tr>
<td>1</td>
<td>CSCE</td>
</tr>
<tr>
<td>2</td>
<td>CSCE</td>
</tr>
<tr>
<td>3</td>
<td>CSCE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Students</th>
<th>Enrollments</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnrollID</td>
<td>Department</td>
</tr>
<tr>
<td>1</td>
<td>CSCE</td>
</tr>
<tr>
<td>2</td>
<td>CSCE</td>
</tr>
<tr>
<td>3</td>
<td>CSCE</td>
</tr>
</tbody>
</table>
Relationships
- If two entries in different tables are related in some way, foreign keys are used.
- A foreign key is a reference to the primary key in another table.
- We will discuss relationships in more detail later.

Database Properties
- The order of the rows and columns is not meaningful.
- Rows can easily be added and deleted.
- Each attribute has a type, like integer or character or string, and can only store entries of that type.
- Every row in a table must have a unique primary key.
- The value of a foreign key must correspond to the value of some primary key.
- There are often other constraints on the values of attributes, such as:
  - The value cannot be NULL. In other words, there has to be a value.

Database Design
- Each table should store a specific type of information.
- Each attribute should be a single piece of information.
  - For instance, first names and last names, street number and street name, department and course number should be stored as two attributes, not one.
  - If a column exists for course description, it might consist of several sentences describing the course. This is still a single “thing,” so it is O.K.
- Information should (almost) never be duplicated.
- Results from computations should not be stored in a table.

Accessing Databases
- To access a database, you need a database management system (DBMS), like MySQL or Microsoft Access.
- Most DBMSs use SQL (Structured Query Language) as the interface language.
- SQL is a (mostly) standard language that allows the user to:
  - Create, update, and delete tables.
  - Add, update, and delete single entries.
  - Query the database to extract information.
- The details of SQL are the subject of another lecture.

References
Designing Relational Databases

Stephen Scott

CSE 156

Introduction

• Now that we know basic database terminology, it’s time to delve into designing one
• The problem: given a text-based description (specification) of a design problem, come up with a concrete specification of the database and the SQL statements to implement it
• Where to begin????

The General Process

1. Identify the entities (tables), and the attributes (fields) that are associated with each one
   • Kind of like naming the objects in a software design problem
   • E.g. a winery is an object, and each has a name
2. Identify the relationships between the entities, and their cardinality
   • E.g. a wine comes from a single winery, but one winery makes multiple wines
3. Use this information to build an initial entity-relationship (ER) model
   • A graphical representation of our database
   • Represents each entity, their attributes, and the relationships (including cardinality) between the entities

The General Process (cont’d)

4. Identify or add a primary key for each table
   • Must be unique
   • Many possibilities
   • Keep it simple, if possible!
5. Label the weak entities of the ER model
   • Entities that depend on other entities
6. Convert the ER model to SQL
   • Start with regular entities, then the weak ones, then do the relationships: one-to-one, then one-to-many, then many-to-many

An Example Specification

The owner of a local scuba store wants to track his customers and their purchases. Each customer has a first and a last name and has a highest certification level (e.g. Open Water, Advanced OW) from some agency (e.g. PADI, NAUI, SSI). Also, each customer has purchased a (possibly empty) set of items from {BCD, regulator, wet suit, dry suit, mask, tank, fins, snorkel}, but at most one item of each type. Each such item has a manufacturer, and multiple manufacturers make similar items. Each manufacturer has a name and address. For simplicity, assume that each manufacturer makes only one item of each type.

• (Also assume that store owner is insane and destroys all serial numbers and other identifying information on each product.)

Step 1

• What are the entities and their attributes?
Step 1
• What are the entities and their attributes?
  1. Customer
     – First name (string)
     – Last name (string)
  2. Certification
     – Level (one of {Open Water, Advanced OW, …})
     – Agency (one of {PADI, NAUI, SSI, …})
  3. Item
     – Type (one of {BCD, regulator, …})
  4. Manufacturer
     – Name (string)
     – Address (string)

Step 2
• What are the relationships between the entities?
  1. A customer has a certification level
     – Each customer has one highest level, and each
       level can be held by multiple customers
       ➢ One-to-many
  2. A customer purchases items
     – Each customer can purchase multiple items, and
       each item can be purchased by multiple
       customers
     – E.g. each customer can own 0 or more from
       {BCD, regulator, …}, and a regulator can be
       owned by multiple customers
       ➢ Many-to-many

Step 2 (cont’d)
• What are the relationships between the entities?
  3. An item is made by a manufacturer
     – Each item has only one manufacturer, but a manufacturer
       can make multiple items
     ➢ One-to-many

Step 3: Initial ER Model
• Entities represented by rectangles
• Attributes represented by ellipses, connected to entities by lines
• Relationships represented by diamonds
• Lines connect entities to relationships, cardinality labeled by letters
Step 4: Primary Keys

- Combination of level and agency uniquely defines a certification
- Customer may not be uniquely identified by first and/or last name, so we’ll add a customer ID (int)
  ➢ Even if combination is unique for each customer, a numeric ID is less prone to spelling errors when inserting, querying, etc.
- We’ll do the same with manufacturer
- With respect to a particular manufacturer, type uniquely identifies item
  – More on this later

Step 5: Label Weak Entities

- Are there any entities that cannot exist without another?

Step 6: Convert to SQL

1. Regular (non-weak) entities:
   ```
   CREATE TABLE customer (customerID int NOT NULL AUTO_INCREMENT, firstname varchar(30) NOT NULL, lastname varchar(30) NOT NULL, PRIMARY KEY (customerID)) type=MyISAM;
   CREATE TABLE manufacturer (manufacID int NOT NULL AUTO_INCREMENT, name varchar(50) NOT NULL, address varchar(100) NOT NULL, PRIMARY KEY (manufacID)) type=MyISAM;
   CREATE TABLE certification (level enum('Open Water','Advanced OW','Rescue Diver',...) NOT NULL, agency enum('PADI', 'NAUI', 'SSI',...) NOT NULL, PRIMARY KEY (level,agency)) type=MyISAM;
   ```
Step 6: Convert to SQL (cont’d)

2. Weak entity:
   - Item is uniquely identified if manufacturer is also specified, so add manufacID as a primary key for item

   CREATE TABLE item (  
   itemtype enum(‘BCD’,’regulator’,’wet suit’,... ) NOT NULL,  
   manufacID int NOT NULL,  
   PRIMARY KEY (manufacID,itemtype) ) type=MyISAM;

Step 6: Convert to SQL (cont’d)

3. No one-to-one relationships, so do one-to-many:
   - The idea is as follows: if a certification can be held by multiple customers, then we need to relate each customer to the held certification
   - Easiest way is to add certification’s primary key(s) to item, to serve as foreign keys:

   CREATE TABLE customer (  
   customerID int NOT NULL AUTO_INCREMENT,  
   firstname varchar(30) NOT NULL,  
   lastname varchar(30) NOT NULL,  
   level enum(‘Open Water’,’Advanced OW’,’Rescue Dive’,... ) NOT NULL,  
   agency enum(‘PADI’, ’NAUI’, ’SSI’,... ) NOT NULL,  
   PRIMARY KEY (customerID) ) type=MyISAM;

   - We have a similar case for the makes relation, but in this case we don’t need to update the item table, since we already added manufacID to it (since item was weak)
   - Note that if item were not weak, then we would add manufacID as a foreign (but not primary) key to capture the relationship

Step 6: Convert to SQL (cont’d)

4. Many-to-many relationships:
   - Each item can be purchased by multiple customers, and each customer can purchase multiple items
   - Thus we need to be able to relate each customer to multiple purchases, and vice-versa
   - Create a new table that takes the primary keys of the related entities as its primary keys
   - Note that since item is weak, one of its primary keys is manufacID, so we need to add that as well

   CREATE TABLE purchase (  
   itemtype enum(‘BCD’,’regulator’,’wet suit’,... ) NOT NULL,  
   manufacID int NOT NULL,  
   customerID int NOT NULL,  
   PRIMARY KEY (manufacID,itemtype,customerID) ) type=MyISAM;

   (Where’d the AUTO_INCREMENT go?)

Final Notes

1. Relationships can have attributes as well!
   - E.g. if a purchase has a date

   CREATE TABLE purchase (  
   itemtype enum(‘BCD’,’regulator’,’wet suit’,... ) NOT NULL,  
   manufacID int NOT NULL,  
   customerID int NOT NULL,  
   dateofpurchase timestamp(8),  
   PRIMARY KEY (manufacID,itemtype,customerID) ) type=MyISAM;

   (could also add date and certification number to “has” relationship)

2. Also, be careful about one-to-one relationships!
   - Ask if they’re really necessary
A Brief MySQL Primer

Stephen Scott

Introduction

• Once you’ve designed and implemented your database, you obviously want to add data to it, then query it to process its information
• This process is pretty simple, once you have the syntax down, but some more complex queries require a bit of thought
• We’ll focus on MySQL, which mostly conforms to SQL standards
• This will be brief; you’ll have to refer to the manuals and cheat sheets for more details, e.g. for the homework

Outline

• Getting started
• Creating a database and inserting data
• Updating and deleting data
• Basic queries via SELECT statement:
  – WHERE, ORDER BY, GROUP BY, HAVING
• Joins
• Temporary tables and nested queries

Getting Started

• From the cse (UNIX) command line:
  mysql -u user -p passwd
  – No space after –p, space optional after –u
  – The user option is optional if your account name = cse login name (typical)
  – If you don’t want to put your password on command line, use ‘-p’ option without password, then you’ll be prompted

Getting Started (cont’d)

• Can also load a database at startup:
  mysql -p < file
  – file contains commands to create tables and insert rows
  – Can also use SOURCE command within MySQL

Getting Started (cont’d)

• Important note: You only have one database, named the same as your MySQL account name!
• Thus your first SQL command will be
  USE username;
• This also means that you’ll have to modify some db source files, e.g. winestore.data
  – Change ‘DROP DATABASE winestore’ to a series of ‘DROP TABLE IF EXISTS table’ statements
Creating a Database

• To create a new table, use the (duh!) CREATE TABLE command:
  
  CREATE TABLE tablename ( 
  field1 field1type [opts], ... 
  PRIMARY KEY (keys) ); 
  
• opts are options like NOT NULL, AUTO_INCREMENT, and DEFAULT

Creating a Database (cont’d)

• “SHOW TABLES;” will list all tables in the current database
• “DESCRIBE table;” will list table’s columns

Inserting Into a Database

INSERT INTO tablename VALUES ( 
value1, value2, ..., valuen ); 
• Must have one-to-one correspondence with fields in the order the fields were specified in the CREATE TABLE statement
  
  Use the describe command to get the order
  
  INSERT INTO table SET 
  field1=value1, field2=value2, ...;

• Can use any subset of fields, in any order

Updating and Deleting

UPDATE table SET field1=value1, 
field2=value2, ... 
[WHERE condition(s)]; 
• WHERE option specifies a subset of records (rows) to update:
  
  WHERE FirstName = ‘John’ 
  WHERE LastName like '%Smith%'

DELETE FROM table 
[WHERE condition(s)];

SELECT

• SELECT is the command you’ll use most
  
  SELECT * FROM table; 
  
  Shows every column and every row in table

  SELECT col1, col2, ... FROM table 
  [WHERE condition(s)]; 

  Shows the specified columns from rows that match WHERE conditions

ORDER BY

• Adding an ORDER BY clause to SELECT specifies the column(s) to sort the rows by
  
  Can also use “DESC” to sort in descending order
GROUP BY and HAVING

- **GROUP BY** gathers rows into sets
  - Use this when using MIN, MAX, AVG, etc.
  
  ```
  SELECT city, MIN(birth_date) FROM customer GROUP BY city;
  ```
  - Prints each city, and lists the birth date of oldest person in that city
  - Can also use HAVING clause to further condition inclusion in groups
    - Use HAVING only with GROUP BY, never as a substitute for WHERE
  
  [WDBA, p. 156-7]

- **Joins**
  - SELECT is fine, but if we have a relational database, shouldn’t we relate things?
  - E.g. say we want to list all the items manufactured by AquaLung
    ```
    SELECT itemtype FROM item WHERE manufacID = (SELECT manufacID FROM manufacturer WHERE name = 'AquaLung');
    ```
  - Can use a SELECT to get AquaLung’s manufacID, then use it in a second SELECT in item table to get the list:
  
  ```
  SELECT itemtype FROM item WHERE manufacID = [let this number be x];
  ```

- **Joins (cont’d)**
  - Is there a better way??? YES!!
  - List both tables in the SELECT, and join them in the WHERE clause:
    ```
    SELECT itemtype FROM manufacturer, item
    WHERE manufacturer.name = 'AquaLung' AND item.manufacID = manufacturer.manufacID;
    ```
  - Notes:
    - Can drop the table name in front of '.' if field name unique
    - Can use > 2 tables
    - The last part of the WHERE is very important!!

- **Beware the Cartesian Product!**
  - Recall that the **Cartesian product** of sets A={a,b,c} and B={d,e} is the set of all pairs of something from A and something from B:
    ```
    A x B = \{(a,d),(b,d),(c,d),(a,e),(b,e),(c,e)\}
    ```
  - A join of two tables A and B generates a new table whose rows are from the Cartesian product of A and B
    ```
    SELECT name FROM A, B WHERE A.name = B.name;
    ```
  - So if table A has 100 rows and table B has 50 rows, the join has 50 x 100 = 5000 rows, most of which are worthless (e.g. the manufactID’s don’t match, so rows are unrelated)
  - Thus we add a WHERE to this to filter out rows that are not related

- **Temporary Tables**
  - Some tasks cannot be done in a single query
    - E.g. you use a join to collect data across multiple orders, then sum it up, and want to find the max of all the sums
    ```
    CREATE TEMPORARY TABLE tmpname
    SELECT f1, f2, SUM(quantity) AS sum
    FROM ...;
    ```
    - Notes:
      - The last part of the WHERE is very important!!

- **Nested Queries**
  - More recent feature of MySQL
  - Can use a query result as part of a WHERE condition
    ```
    SELECT max(ordercount) AS max FROM table;
    ```
    - Or:
    ```
    SELECT lastname, firstname, ordercount AS max
    FROM table WHERE ordercount = (SELECT MAX(ordercount) FROM table);
    ```

  [WDBA, p. 156-7]