Database Basics for Java Programmers

> Most applications just manage some data and just implement CRUD (create, read, update and delete), the four basic functions of persistent storage.

A *database* is just a collection of data. (The SQL standard calls databases *catalogs*, but the two terms are the same in practice.) A database *schema* is the design or plan of the database.

> **[on-line] Transaction-processing (OLTP) systems** are optimized for the CRUD operations used to capture information and to be updated quickly. They are constantly changing and are often online 24 hours a day. Examples of transaction-processing systems include order entry systems, scanner-based point-of-sale registers, automatic teller machines, and airline reservation applications. These systems provide operational support to a business and are used to run a business.
>
> **Decision-support systems** (OLAP) are optimized for query operations used to allow analysts to extract information quickly and easily. The data being analyzed is often historical: daily, weekly, and yearly results. Examples of decision-support systems include applications for analysis of sales revenue, marketing information, insurance claims, and catalog sales. A decision-support database within a single business can include data from beginning to end: from receipt of raw material at the manufacturing site, entering orders, tracking invoices, and monitoring database inventory to final consumer purchase. These systems are used to manage a business. They provide the information needed for business analysis and planning.
>
> The data in decision-support systems are often captured by OLTP systems, and then loaded into a decision-support system (i.e., a separate DB).

Object Model – Uses **JNDI** (*Java Naming and Directory Interface*) to retrieve objects (or other things) by name. A *directory* is a type of hierarchical object database and an organization may have several. JNDI permits you to search multiple directories. This sort of database is useful with Java, since it would permit you to store objects and later retrieve them, across a cluster of application servers.

Relational Model – Uses *tables*, where each row is one record and each column is one attribute/property/field. Multiple tables can be linked on a common column (a *join*), to produces a composite table. Today this is by far the most commonly used model. Many companies may have invested millions of dollars in a proprietary relational database, and you may need to use one from your Java program.

A set of tables and the constraints between them form a *database*. The description of the tables (names, column names, types, indexes, and constraints) is called the database *schema*. The term schema can also mean a group of table definitions that form a namespace. This provides a way to name a subset of tables and other dictionary objects within a database. To refer to a table *t* in a (different) schema *s,* use the notation *s.t*. (This is rarely needed for simple databases, which will use a default schema (namespace).

> Defining a proper schema for a given application takes some training. Small changes to the schema can result in such large performance changes as to render an entire application unusable.

A database can be something as simple as a single file with a standard format. However all databases have many issues that must be addressed in an enterprise application. A **DBMS** or **RDBMS** (*Relational Database Management System*) provides support for creating and allowing access to

databases, automatically handling such issues as concurrent access, caching, storage onto disk, transaction support, network access, security, etc. Examples include SQL Server, Informix, Oracle, and Sybase.

> SQL supports many useful features besides tables (but not every RDBMS supports all of these). *Views* are virtual tables formed by a query. *Stored procedures* are sets of SQL statements that can be stored in the server. Once this has been done clients can refer to the stored procedure, passing in parameters. (This is very efficient.) A *trigger* defines a set of actions that are executed when a database event occurs (delete, insert, or update) on a specified table. A *sequence* is a counter used to generate unique ID; each time you use it you get the next number. An *identity* column is a primary key column that uses a (private to the table) sequence to generate the values automatically. A *tablespace* consists of one or more datafiles, used to specify a physical location of a database object.

> **Files or Database for persistent storage?**
>
> For read-only access to the data, a file (text, XML, or other format) can work well. But the reason most people use a database even for simple data is that any (web) application may be used by several users at the same time. If one user is reading a file while another writes to it, you will have problems of one sort or another.
>
> In a cluster or grid of web servers, a *load balancer* (middleware) will send requests to different web servers. So either you would need to keep a copy of your data file on each server and synchronize any updates, or your application will need network access to the file.
>
> *Failover* can be an issue for direct file access; what happens when an application crashes while updating the file? Also efficiency can be a factor—some filesystems are slow or can even lock a file while being accessed.
>
> All in all, you are better off using a database. A database management system is designed to efficiently handle all these issues so you don't have to.
>
> Some web / presentation layer frameworks will handle the JDBC for you. Also, Java 6 now comes bundled with *Derby*, an Apache RDBMS. So using a DB isn't much harder than using a file, and can have many advantages. Of course, you must learn how to use that part of Java.
>
> For simpler data persistence, use the new Preferences API, or use `Property` files.

Embedded Database or Client / Server

A database may be *embedded* in an application. Such a DB is easy to set up; it is just a set of classes (that access a native library). The resulting DB is available to the one application only. A typical use would be a PDA application.

The data in an embedded DB can't be shared easily with other applications. For this (more common) situation you need a single central DB that can be accessed by many applications/users. This is where you use a DBMS, or *DB server*. Applications must establish a connection to the server (usually across a network) and authenticate themselves (e.g., username and password). Also a single server may serve several different DBs, so the client application must also identify which DB it wants to connect with. Connecting to a server is thus slower than using an embedded DB.

By far the most common commercial DBMS is Oracle, which includes extra tools for creating complete DB-centric applications. However Java SE now includes Derby, an excellent DBMS that can be run either as a server or embedded.

> Derby was originally the commercial DBMS *Cloudscape*. It was written by Californian startup, bought by Informix, acquired by IBM when it bought Informix, and renamed as Derby when IBM donated it to the Apache organization.

Even if eventually using an embedded DB it may pay to use a client-server DB during development. Doing so will allow you to examine the DB during testing. If using Derby you only need to change the JDBC driver used (and the connection code) to switch from client-server to embedded — the SQL stays the same.

> The bundled Java DB, a.k.a. Derby, is good for small to medium databases (up to a few tens of millions of rows each for dozens of tables). MySQL (now owned by Sun) is great for Gigabyte sized databases that require fast connections, such as for websites. Go with Oracle or DB2 for Terabyte sized DBs. (Google's DB is measured in Petabytes!) Note, even a small DB needs to be well designed (including proper indexes) or it will suffer performance issues.

SQL

The data may be updated, deleted, or added to the tables using **SQL** (*structured query language*). Note that while SQL is standardized, each different DBMS omits some SQL, or provides proprietary SQL extensions that are useful enough to tempt one into using them rather than standard SQL. Even the standard parts don't always work the same way.

The language has three parts, the ***data definition language*** (DDL, the SQL where you define and change schemas) the ***data manipulation language*** (DML, the SQL where you lookup, add, change, or remove data). The third part is called ***Data Control Language*** (DCL, the SQL used to manage the server and the databases). (Often additional, non-standard DCL commands are available.) SQL supports software's need for CRUD.

Typically, the application makes *queries* on the database and returns a **ResultSet**, a collection of the rows that matched the query. An example query might be "list all customer names, addresses, and phone numbers, who live in Tampa and who have made a purchase over $500 within the past year". The other typical operation is to add a new record to the database (which may mean adding rows to several tables in a single transaction). An example is adding a purchase order.

Most SQL statements are called *queries* or *updates*, and can be entered on one line or several lines. They end with a semicolon ("`;`"), although not all DBMSes will require this. The SQL keywords are not case sensitive; only data inside of quotes is case sensitive. SQL uses single quotes around literal text values (some systems also accept double quotes). Some examples of SQL:

```
INSERT INTO table (col1, col2, ...)
  VALUES (val1, val2, ...);

SELECT [DISTINCT] col1, col2, ...
      or use wildcard:  * instead of a column  list
  FROM table [, table2, ...]
  WHERE condition (e.g. WHERE amount < 100)
  ORDER BY col;

UPDATE table
  SET col2 = value2, col3 = value3, ...
  WHERE condition; (e.g. WHERE col1 = value1)

DELETE FROM table
  WHERE condition;     (e.g., WHERE col1=value1)
```

Designing a good DB is a job for an expert. A bad design may be so slow as to be unusable, cause security vulnerabilities, or may even induce errors in the data! This holds true if either using SQL or the Java query API.

> *Normal forms* are a way to prevent normal DML operations from destroying real data or creating false data. That can happen if the schema isn't designed for the types of queries and multi-user activity that is common. Creating normal forms is a straight-forward process of transforming a schema from normal form *n* to normal form *n+1*. Although there are many normal forms (at least 9, or over 300, depending on how you count them), normal forms 4 and higher cover obscure potential problems that very rarely ever manifest. Most DB schema designers are happy with third normal form.

A common use for a DB is to allow customers to sign up for services from your website. Suppose you want to send a "Click here to confirm your registration" URL in an email to a new subscriber. This is commonly called in "invite".

Each user needs a unique ID to refer to their data in the DB, and it is "obvious" to use a sequence number. But then you have problems with exposing those numbers in URLs, or having attackers guess the next transaction number.

The correct solution is to use a sequence number, encrypted and the result encoded to base-64. When the user then clicks the link, your web server decodes the link, then decrypts it to recover the serial number. The user sees a large random-looking link only. Internally you can use simple sequence numbers in your DB.

To prevent attackers using random strings (that will decode into random IDs), encrypt the sequence number plus a checksum of that number. Now most random URLs won't decode into valid sequence number-checksum pairs.

The DB table where you store these should include a timestamp as well (to time out invitations), a "responded" Boolean field (to prevent replay attacks), and the IP where the invitation was sent (to help prevent spoofing). Of course the table includes other details of the invite, such as the username, etc.

When the reply is received, you can validate the checksum, check the time against the timestamp, check if this has been replied to before, and check the source IP.

All this and we haven't designed the table for performance yet! Clearly a job for an expert!

DB Driver types

Sending commands (queries, inserts, ...) to some DBMS requires sending commands in some way (that is, using a protocol). The commands and protocol are different for each different DBMS. (That is, to talk to an Oracle DBMS you would need to use the proprietary Oracle DBMS protocol. These protocols are not part of SQL and are not standardized.) Few applications do that, but it does provide the best performance and allows your application access to any proprietary features of your DBMS.

Instead, the common solution (not just for Java) is to use a *driver* that translates some standard DBMS connection method into the specific protocol needed for a particular DBMS. (Of course this approach has drawbacks: not all standard operations are available on all DBMSes, and some DBMSes have extra features not part of a standard.)

In Java, this standard language is **JDBC (Java Database Connectivity)**. (This is the lowest-level of access to a DB in Java. Later you will see how to use a higher level of abstraction, to make your code easier to write.) Your applications can use JDBC to connect to a DBMS, and if you change the DBMS, the code doesn't care.

To make this work, the JDBC part of the JRE must have a JDBC driver that can translate the standard JDBC access into the proprietary protocol actually used.

> ***ODBC*** is a standard connection protocol for talking to different DBMSes. (From Wikipedia: ODBC uses as its basis specifications from the SQL Access Group, X/Open (now part of The Open Group), and the ISO/IEC. Microsoft, in partnership with Simba Technologies, created ODBC by adapting the SQL Access Group CLI. It released ODBC 1.0 in September 1992. After ODBC 2.0, Microsoft decided to align ODBC 3.0 with the CLI specification making its way through X/Open and ISO. In 1995, SQL/CLI became part of the international SQL standard.

There are four types of JDBC drivers available:

1. **JDBC-ODBC Bridge** –Since all DBMSes provide ODBC drivers, you can use this JDBC-ODBC bridge to access them. This isn't really a type of driver, it is a "ODBC" driver that is standard with Java. However it is commonly referred to as a "*type 1* driver". This solution is limited in features and performance (your queries go though JDBC driver, then the ODBC, driver, then the proprietary driver, and finally to the DBMS), but may be your only option if no JDBC drivers are provided by your DBMS vendor.

   > This type of driver uses the native OS's ODBC driver, and thus can't be used in an Applet.

> Note! There are two ODBC systems on Windows for 64-bit systems, ODBC32 and ODBC64. If you use 32-bit Java on a 64-bit Windows system, you must use ODBC32 drivers or your program won't work. Unfortunately, the default on 64-bit Windows 7 is ODBC64. **To use ODBC32, you must launch the 32-bit utility, not found in the Start menu: `%windir%\SysWOW64\odbcad32.exe`**.

2. **Native API** – This is the *type 2* driver that translates JDBC API into some propriety DBMS API. This solution requires the vendors DBMS client be running locally, so this driver can talk to it. The DBMS client in turn talks to the DBMS server across a network, using a proprietary protocol. This solution is only a little better than a type 1 driver, as it provides slightly better performance. But this solution has deployment, manageability, and scalability issues.

3. **Net Protocol** – This is the *type 3* driver designed for use in Java EE client-server applications. The client code uses this driver type to send requests to an application server (middle-tier server), which in turn talks to the actual DBMS using some other type of driver.

   This solution is server-based, so there is no need for any vendor database library to be present on client machines. It provides opportunities to optimize portability, performance, and scalability. The net protocol can be designed to make the client JDBC driver very small and fast to load. Additionally, a type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced system administration such as logging and auditing.

   On the other hand, Type 3 drivers require database-specific coding to be done in the middle tier, generally not a good idea. Also accessing a `ResultSet` may take longer since the data comes through the backend server.

4. **Native-Protocol** – This all-Java, *type 4* driver converts JDBC calls into the vendor-specific database management system (DBMS) protocol, so that client applications can communicate directly with the database server across a network. Level 4 drivers are completely implemented in Java to achieve platform independence and eliminate deployment administration issues. This provides the best performance, but can lead to deployment issues since if you change your DBMS you must update all clients with a new driver.

Java DB drivers are loaded in a strange way. The driver class isn't directly accessed by your code so it won't normally be loaded by the class loader. To force the driver to load you use some ugly code:

```
try {Class.forName("name.of.jdbc.driver" );}
catch ( classNotFoundException e ) {...}
```

Or the simpler: `Class c = name.of.jdbc.driver.class;`

Or even: `new name.of.jdbc.driver();`

> In Java 6 and newer, this should not be needed (but should still work); JDBC type 4 drivers are loaded automatically when you create a `Connection`. The JRE determines the driver class name from the connection URL. You still use this older method to use non-type 4 drivers however.

Once this is done, you can create an SQL statement to be sent over a connection to the database. In typical Java fashion you create a `Connection` object from the DB driver, then a `Statement` object from the `Connection`. Finally you execute the statement, which (hopefully!) returns a `ResultSet` of the results of your statement. Here's an outline of the code needed:

```
try { Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" ); }
catch ( Exception e ) {
 System.out.println( "**** Cannot load ODBC driver!" );
 return;
}
Connection con = null; Statement stmt = null;
try { con = DriverManager.getConnection( "URL",
        "username", "password" );
 stmt = con.createStatement();
} catch ( Exception e ) {
  System.err.println( "**** Cannot open connection to "
     + URL + "!" );  }
try {
 ResultSet results = stmt.executeQuery( "SQL_Query" );
 ...  }
```

(Show `DBDump.java` and `Coffee/Cups.txt` DB.) The ResultSet contains *metadata* you can access, telling you the number or rows returned, the number, name, and type of each column. You access each row using a loop that causes the ResultSet object to internally "point" to the next row. (The pointer is called a *cursor*.) You can only advance the pointer; you can't jump around in the results. Also, note that some SQL statements don't return ResultSets, just a status.

> With JDBC, your Java program must access each column of each row, convert it to an appropriate Java data type, and build an object representing that row. The objects must be manually put in a collection. Java EE supports a more object-oriented approach, using query objects that return collections of objects directly.

Using Java DB (Derby)

JavaDB is really the Apache Derby DB. (For now; who knows what Oracle with do with this in the future.) It is included with the JDK, but may not install by default, especially on 64 bit computers. In that case, there is a separate installer from Sun (no longer available). Best advice is to install Derby directly from Apache.

To use Derby, make sure the environment variable **JAVA_HOME** is set to where you installed the JDK, and **DERBY_HOME** to where you installed Derby. Make sure that **PATH** includes "`%JRE%\bin;%JAVA_HOME%\bin;%DERBY_HOME%\bin`" (`%JRE%` is

where you installed the JRE, say "`C:\Program Files\Java\jre6`"). In PATH, don't use environment variables but only full absolute pathnames.

Make sure **CLASSPATH** contains "`%DERBY_HOME%\lib\derbytools.jar`" and one or both of "`%DERBY_HOME%\lib\derby.jar`" (if using the embedded DB) and "`%DERBY_HOME%\lib\derbyclient.jar`" (if using the server). Or copy the jars to your *extensions* directory.

Now you create and use a database. The easiest way is to use the embedded database rather than setup the server. You should `cd` to the directory where you want to create the DB, and run the command line tool "`ij`".

> Using the embedded DB is easier, but if you want you can use the server. **To start the Derby server**, run: `start/b startNetworkServer.bat`
>
> **To shutdown the server**, run: `stopNetworkServer.bat`. (You can always hit ^C in the server's DOS window if you didn't use "`start/b`".)

Now create a DB named `AddressBook`, using the embedded driver:

```
C:\Temp> mkdir myfirstdb
C:\Temp> cd myfirstdb
C:\Temp\myfirstdb> ij
ij> CONNECT 'jdbc:derby:AddressBook;create=true';
```

> A Derby DB can be used from a zip or jar file. (In this case, the DB is *read-only*.) To access a DB in a zip/jar, use this connection URL: "`jdbc:derby:jar:(pathToArchive)dbPathWithinArchive`".

The next time you want to use this DB you can (but don't have to) omit the "`;create=true`" part.

> The *system directory* is where Derby keeps its databases (in a subdirectory for each). Derby uses the property **derby.system.home** to determine which directory is its system directory, and thus what databases are in its system, where to create new databases, and what configuration parameters to use. You need to define that on the command line when starting Derby or from within your Java code. If you specify a system directory at startup that does not exist, Derby creates this new directory automatically (of course it won't contain any DBs yet). You can't specify this directory on the `ij.bat` command line, but there are other ways.
>
> **If you do not specify the system directory when starting up Derby, the current directory becomes the system directory.** To set this property from within your Java code (on Windows), use:
> ```
> Properties p = System.getProperties();
> p.setProperty("derby.system.home", "C:\mydb");
> ```

Now you can type in SQL statements to create tables and add rows to them. If the SQL is saved in a text file, you can run that code easily with "ij> `run 'someFile.sql';`". To see a list of tables, use "`show tables;`",

and to see the description of some table, use "`describe` *tablename*`;`". When done, use "`exit;`".

If you examine the directory, you will see Derby created a new sub-directory for the database (and named for it). You will also see a log file "`derby.log`" (in Unix text format) that is useful to troubleshoot connection problems. You can zip this folder to backup your DB. Inside you will see a directory "`log`" that contains the DB transaction log files, a directory "`seg0`" that contains the DB data. You will also see a Java properties file (DOS text) named "`services.properties`". **Never edit any of the files in your DB directly!**

> Derby doesn't support the SQL standard `INFORMATION_SCHEMA.*` views, so you can't use the following to list the tables in your DB:
>
> ```
> SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
> WHERE TABLE_SCHEMA = 'AddressBook';
> ```
>
> However this will work for Derby:
>
> ```
> SELECT t.tablename FROM SYS.SYSTABLES AS t
> WHERE t.schemaid = (SELECT s.schemaid FROM
>    SYS.SYSSCHEMAS AS s WHERE s.schemaname = 'APP');
> ```

You can use the command "`dblook -d 'jdbc:derby:AddressBook'`" to see or save the DDL statements for your DB. (Derby includes `ij`, `dblook`, `sysinfo`, and other command line tools.)

> **You can set system-wide Derby DB properties** in a text file called **`derby.properties`**, which must be placed in the directory specified by the `derby.system.home` Java property. Database-wide properties are stored within the database itself. You set and verify database-wide properties using system procedures within SQL statements. To set a property, use the `CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY` procedure, passing a property name and the value. To view the current value of a property use the `VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY` function, passing in the name of the property.

To use an Derby embedded DB from your Java program, use code such as this:

```java
import java.sql.*;
public class Foo {
  public static void main ( String [] args ) {
    String driver = "org.apache.derby.jdbc.EmbeddedDriver";
    String dbName="jdbcDemoDB";
    String connectionURL = "jdbc:derby:" + dbName;
       // + ";create=true";  // If creating the DB

    try { Class.forName(driver); }
    catch ( ClassNotFoundException e ) { ... }

    try {
      Connection conn =
```

```
|           DriverManager.getConnection(connectionURL);
        Statement s = conn.createStatement();
        s.executeUpdate( SQL-string ); // insert or other DDL
        ResultSet rs = s.executeQuery( SQL-string );
        ResultSetMetaData meta = rs.getMetaData();
        ...
    } catch ( Exception e ) { ... )
```

> Unlike most interactive DB browser tools (such as "ij"), **the SQL statements used with JDBC must <u>not</u> end with a semicolon!**

The `Connection` can be used to obtain a `DatabaseMetaData` object, which has methods you can use to query the DB in general (for example, get a list of the tables in that DB). See the `DerbyDemo.java` resource.

To use a DBMS server (such as Derby, MySQL, Oracle, Postgres, DB2, ...), you only need to make a few changes to the connection (and start the server):

```
String driver = "org.apache.derby.jdbc. ClientDriver";
String connectionURL = "jdbc:derby://localhost:1527/"
    + dbName;
```

Transaction Processing

When updating a DB that is used by multiple clients simultaneously, the changes need to appear *atomic*. This is done by creating a *transaction*, which is a group of statements. A transaction is a unit of work that has "**ACID**" properties: *Atomic* (the group is done or undone completely), preserves *Consistency* of the data, *Isolation* (the transaction appears atomic to other transactions), and *Durable* (persistent).

To make a transaction, start with "`START TRANSACTION;`". Now all the SQL statements you enter next will have no effect as far as other clients are concerned! After the last statement, you use the command "`COMMIT`;" which will make the changes visible to all. If you decide not to commit your changes, use "`ROLLBACK`;" instead of `COMMIT`. The transaction will also rollback automatically if there was an error that prevents the commit from completing.

Many DBMS servers can make each SQL update/insert/delete a transaction automatically. With MySQL, just "`set AutoCommit=1`" and each statement executed is a single transaction. (Setting to zero instead turns off that feature.) With Derby use "`AUTOCOMMIT ON|OFF;`".

ORM and the DAO pattern

Since most programming code is designed to work with objects, or collections of objects, to use a traditional DB requires *object to relational DB mapping*.

Example:

```
class Person { String name; int age; ... }  List<Person> =
new ArrayList<Person>(); ...
```

But to store/retrieve people from a relational DB:

```
Create table person (...)
Select * from person where id=123
...
```

Thus, you must manage people as a table of raw data, or convert each fetched (stored) Person object to/from the DB. One solution is to use an *object based* database, but today most still use relational DBs. Another solution is to have software that automatically converts the objects to/from the database for you. This is ORM, and is common in Java EE and other (e.g., Ruby on RAILS) solutions.

ORM is also called the ***data access object*** (DAO) pattern. A DAO is an object that provides an abstract interface to some type of database or persistence mechanism, providing some specific operations without exposing details of the database. Commonly used with Java EE, the DAO pattern separates object persistence and data access logic from any particular persistence mechanism or API. It allows one to automatically associate objects and database records. Naturally all the work must still be done, to query the back-end DB (usually with SQL, but increasingly, other query languages), convert the results to Java data types, and store changed objects. But your code uses just objects and generic commands to get or update persistent data. Somewhere else is the definition of the actual DB to use, the driver to use, a connection URL, etc. All that can be changed without requiring any changes to the code itself.

> In fact, Java EE includes its own query language, and the results are automatically translated into the required SQL. However best results may be obtained if you are an SQL expert and use SQL instead. Note the actual query strings should be stored in external property or text files, to make it simple to change the queries later.

The DAO design pattern provides a simple, consistent API for data access that does not require knowledge of JDBC, EJB, Spring, or Hibernate interfaces. A typical DAO design pattern interface may look something like this:

```
public interface CustomerDAO
{
 public void insert(Customer customer)
 throws CustomerDAOException;

 public void update(CustomerID id, Customer customer)
 throws CustomerDAOException;

 public void delete(CustomerID id)
 throws CustomerDAOException;

 public Customer[] findAll()
 throws CustomerDAOException;

 public Customer findByEmailAddr(String email)
 throws CustomerDAOException;

 public Customer[] findByCity(CityID cityId)
 throws CustomerDAOException;
```

```
    . . .
}
```

It is important to note that DAO does not just apply to simple mappings of one object to one row in one relational table, but also allows complex queries to be performed and allows for stored procedures and database views to be mapped into Java data structures.