COMP 1140: Database and SQL | Brendan Shea, PhD (Brendan.Shea@rctc.edu)

BEYOND THE RELATIONAL MODEL: OBJECTS, XML, AND NOSQL

In this lesson, we'll be answering the following questions

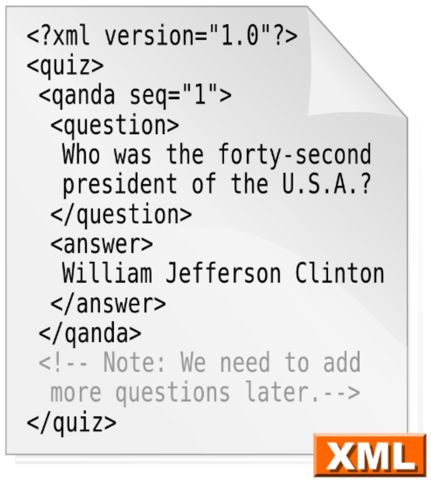
1. Which sorts of data models are most widely used in database development? How has this changed over time?
2. What are the basic concepts of the **Object-Oriented** data model?
3. What is **XML?** What is an **XML database?**
4. What is **"Big Data"?** How do **NoSQL** databases relate to it?

A VERY BRIEF HISTORY OF MODELING FOR DATABASES

Ideas about how best to model data for databases have changed over the years. These changes were driven in part by changes to computer hardware (since faster hardware served to make different sorts of models realistic), in part by change in computer use (modeling web pages is different from modeling banking data), and in part by continued research on the underlying algorithms and efficiencies of different models. The main models were as follows:

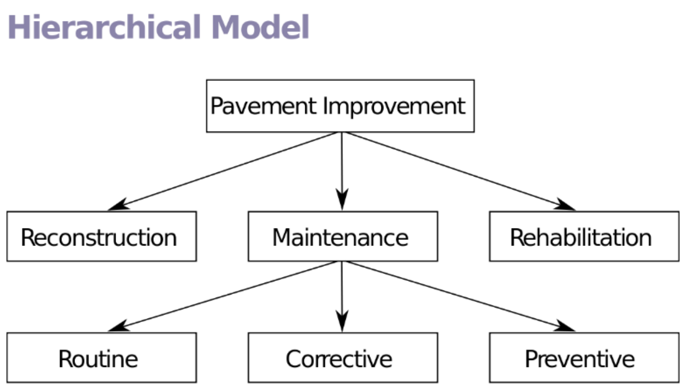
1. **File system** organization of records was common well into the 1960s, even as companies increasingly began to computerize their records. As you'll recall from previous lessons, a "file" is simply a collection of records chosen for some purpose or other. This is highly inefficient.
2. Starting in 1960s, **hierarchal** (tree-like) and **network** (graph-like) data modeling focused on *linking* different types of data to other types. It required that users have fairly extensive knowledge how each piece of data was related to each other piece of data. (So, small changes to underlying data structure could cause large disruptions for users and programmers).
3. The 1970s saw the rise of **relational** database models and of the related **entity-relationship** model of picturing data. The database software (and the SQL language) designed for this model are the basis for most modern databases.
4. The **object-oriented** data model, which allowed user to define complex types "objects" that bundled data with methods that acted on this data. Many of its insights and capacities could be integrated with the relational database model (forming the **Object-Relationship** data model).
5. The explosion of web content starting in the 1990s led to the widespread use of **eXtensible Markup Language (XML)** in storing, accessing, and (especially) exchanging this sort of semi-structured data. **XML** databases and **XML Hybrid Databases** (Postgres, MS SQL Server, Oracle) incorporate XML with the relational model.
6. The increasing importance of "big data" in the 21st century (and its analysis by corporations such as Google, Facebook, and Amazon) has led to **NoSQL** databases that are NOT based on the relational model. Many (though not all) of these databases deal with cloud-scale operations dealing with semi-structured data, such as that stored in JSON format. As was the case with XML, relational databases have adapted to this by providing increased support for JSON.

The relational model became something like the "standard" data model starting in the 1970s for a number of reasons. Among other things, it (1) made user's access to data **structurally independent** from database design, (2) provided easy support for transactions, (3) made resolving "ad hoc" queries about different tables relatively straightforward, and (4) minimized data redundancy and inconstancy. However, other data models work better for certain sorts of specialized uses.'



```
<?xml version="1.0"?>
<quiz>
 <qanda seq="1">
  <question>
   Who was the forty-second
   president of the U.S.A.?
  </question>
  <answer>
   William Jefferson Clinton
  </answer>
 </qanda>
 <!-- Note: We need to add
  more questions later.-->
</quiz>
```

XML can be used to store things like "a question and answer" data structure. (Image source: Wikipedia)



The hierarchal database model requires we navigate to data items by knowing their path. For example, "Pavement Improvements: Maintenance: Preventative." (Image source: Wikipedia)

THE OBJECT-ORIENTED MODEL

An **object-oriented** database is one that is based on the same sort of data model that underly object-oriented programming languages, such as C++or Java. Where the relational and E-R models see the world in terms of internally SIMPLE entities (expressed as "tables" or "relations") that have attributes are related to other entities, the object-oriented sees the world as fundamentally made up of different *objects* that have complex internal structures. In comparison to the relational model, the object-oriented model attempts to capture more of the **semantics** (or "meaning") of what exactly these objects are. The main concepts of the model are as follows:

1. A **class** is an **abstract** type of object (that is, classes don't exist without instances!). So, for example, we might have a class for "Student." The "objects" are the individual students (Hermione, Ginny, Harry, etc.) who are the **instances** of this class, and they **inherit** all of the properties that we have defined that class to have. So, for example, HarryPotter might be an instance of Student.
2. Each object **encapsulates** (or "contains") both the *data* associated with it (student names, home address, email address, etc.) as well as "methods" that operate on this data. For example, HarryPotter.enroll(course_id) would enroll Harry in a particular class.
3. One powerful feature of the OO approach is its support for **Polymorphism,** which allows us to access certain sorts of data (or call certain sorts of methods) *without knowing what sort of object we are dealing with.* So, for example, if both the Student and Professor classes are sub-classes of the more general Person class, we might be able to call methods like getBirthDate() on both Students and Professors.

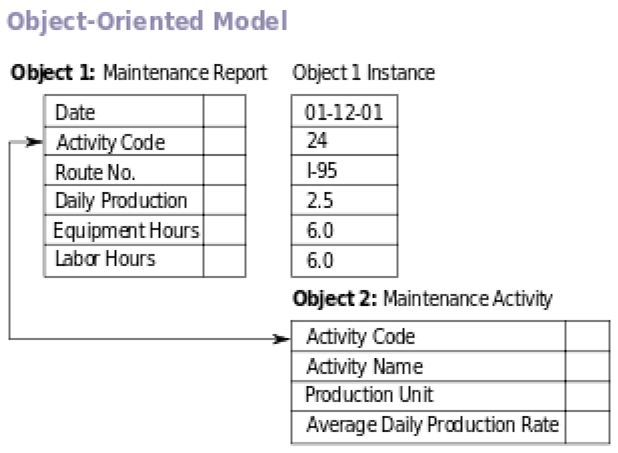


Figure 1 From wikipedia.com. Here "Object 1" and "Object 2" are classes, while "Object 1 instance" is an object!

Don't worry is this seems a bit confusing! Getting a hang of the object-oriented model usually requires spending a fair amount of time programming in a language like C++ or Java. In the context of databases, the most important aspect of this model is its robust support for *defining* new types of data, and supporting the creation of methods that *act* on this sort data. The main application of pure **object databases** thus tends to come in domains where we need to interact repeatedly with "complex" sorts of objects, such as in certain areas of science (physics, astronomy, engineering) or specialized **real-time** systems (such as air traffic control systems).

"BIG" DATA PROBLEMS AND THE RISE OF NOSQL DATABASES

Recent developments in technology—including digital cameras, smart speakers, mobile phones, and countless Internet of Things (iOt) devices—have made it much easier to record lots of data. Simultaneously, the cost of *storing* such data (on hard drives or solid state drives) has continued to go down. In fact, we currently produce and store more data *every day* than all of human history up to the computer age. However, human intellectual capacity (and even our computer algorithms) have NOT advanced at the same rate. This had led to the study of **"Big Data",** as companies, governments, and researchers struggle to make sense of the data.

The Three 'Vs'. While there is no hard-and-fast distinction between "Big" data and ordinary (small?) data, Big Data is often defined in terms of velocity, volume, and variety:

- **Velocity** is speed with which data collects and changes. Social media companies such as Facebook, for example, must deal with billions of simultaneous users interacting with each other and generating new data almost continuously. For better or worse, this is a much *faster* way of generating social interaction data than was possible when people communicated using speech or sent letters through the mail.
- **Volume** is the sheer amount of data that must be analyzed. Companies such as Google, for example, take in enormous amounts of data every day. They must then process this to deliver search results, target advertisements, etc.
- **Variety** refers to fact that there are many different types of data that are produced and must somehow be "reconciled" with one another to produce usable results. For example, Artificial Intelligence agents such as IBM's Watson (which beat humans on "Jeopardy") needed to learn to "understand" how the many, many textual sources it processed "related" to one another, even

when this wasn't explicitly encoded. This becomes even more challenging when we attempt to incorporate things such as images, sensor data, etc.

Big Data presents a number of challenges for traditional relational databases. Among the most important are:

1. **The data is often unstructured and "sparse."** In a relational database, each record on a table contains the *same* attributes. So, for example one record of a bank account table looks much like the other (thought with different values for different customer name, balance, etc.). By contrast, "Big" data gathered from social media sites, sensor, etc. are often wildly diverse. One social media user, for example, might have many photos, but few posts, while another might have the opposite. In technical terms, the data is **sparse,** with most items have a value of "null" for most attributes. Relational databases aren't set up for this.
2. **Expanding relational databases is expensive.** Adding millions of new unstructured data items to a relational database is expensive (in terms of storage, processing power, etc.). Depending on a user's needs, it is also wasteful, if they don't *need* the full suite of capabilities (such as "ad hoc" queries using SQL) that a relational database offers.
3. **Data analysis requires new tools.** The sorts of analytic tools built into SQL were designed for structured data. Large amounts of unstructured data can be better analyzed using other tools.

NoSQL ("Not only SQL") databases represent a variety of *different* ideas about how to respond to these challenges. Many take advantage of the **Hadoop** or similar frameworks for "distributing" the storage and processing of data across a large number of machines. They are, in general, designed to have high scalability (easily add more machines to meet demand), high availability (users don't have to wait) and fault tolerance (the failure of a few machines won't bring the database down).

"ABSTRACTING" AWAY THE DETAILS AND LEVELS OF MODELING

It's important to remember that "data modeling" is a process (as opposed to a "result") and that is perfectly fine to use different sorts of models at different steps in the process. Which model is "best" in a given situation depends not only the data that needs to be stored, but also on things like the purposes for which the data needs to be processed, the function of the model, etc. In general, there are the least three (and maybe four) levels of data modeling:

1. **End-users models** present data as *one particular sort of end-user might see it.* For example, in the Hogwarts case, an instructor like Snape might care about student grades, while a groundskeeper like Hagrid might care about efficient access to inventory lists. These might be expressed by E-R diagrams or Object diagrams in which unnecessary things are "left out."
2. **Conceptual models** present a "big picture" view of the data as the leaders of the organization might see it. For example, the school head Dumbledore would want a model that contains both Snape's and Hagrid's data. Again, this might involve ERDs or Object diagrams, though with more depth than the previous stage.
3. **Internal (or logical) models** must encode the formal logical "rules" that will eventually be enforced by the chosen DBMS. We need to think about things like the data types for different attributes, the structures of objects/entities, and so on. Here, modeling for a (pure) object database will look very different from modeling for a (pure) relational database.
    a. We say that data has **logical independence** if and only if changes to the internal model do NOT affect the conceptual level. This is a big deal for end users and programmers, and was a major reason behind the adoption of the relational model.
4. **Physical models** concern how the data will actually be stored (on local drives? Shared across a network? On the "cloud"?). While these aren't the sort of details database developers and administrators spend most of their time worrying about (thanks to the development of modern data models!), this can in some cases impose constraints on how we go about doing step 3.
    a. We say that data has **physical independence** if and only if changes to the physical substructure (e.g., changing from a single large drive to a collection of networked drives) doesn't affect the logical level.

REVIEW QUESTIONS

1. Suppose that you were designing a database for a library (which had data on books, patrons, and loans). What might the end-user model look like (either for a staff member or patron)? A conceptual model?
2. Compare and contrast the object-oriented and relational model. Give an example of how each might model something like a Library database. (You don't need to include all of the details!).
3. The successful analysis of Big Data offers many potential advantages, such as producing new medical treatments, creating improved and useful AI-systems (such as self-driving cars), and even predicting events such as earthquakes or volcanoes. However, there are also some threats, in particular to personal privacy (it's increasingly easy for companies to gather *a lot* of data about you, even if you would rather they wouldn't!).
    a. Do you think there is a general "right to privacy"? If so, what kind of data should be protected?
    b. How can data professionals (developers, software engineers, etc.) help make sure that their work will *help* rather than *harm* people?