CPS352 Lecture - Other Databsse Models

May 3, 2017

Objectives:

1. To elucidate reasons for a desire to move to models other than the relational model for some applications
2. To introduce OO extensions to the basic relational model
3. To introduce key-value, document, column store, and graph data models.
4. To discuss issues involved in choosing an approach for a particular problem

Materials:

1. Projectable of Sandalage/Fowler Figure 1.1
2. Projectables of Sandalage/Fowler figures 2.1, 2.3, 2.3 with corresponding JSON, 2.4 with corresponding JSON, 2.5
3. Prrojectable of Sandalage/Fowler Figures 11.1, 11.2
4. Map-reduce example to project.

I. Introduction

A. The relational model has become the dominant database model and remains the best choice for many applications. However, there are good reasons why an alternate model might be desirable in certain cases. The reasons why the relational model has prevailed include:

1. Support for integrating data from multiple applications, and allowing multiple applications to share a common database.
2. An interactive query facility which allows accessing data without custom programming through the use of a common language: SQL.
3. Strong support for ACID transactions that are central to maintaining database consistency in the face of concurrent operations and threats of data lost due to things like hardware failure.

   This is actually not unique to the relational model nor do all relational databases support transactions.

a) Databases based on other models can alsp offer support for transactions.

b) Not all relational databases support transactions. For example, if transaction support is wanted in mysql, one must specify this when creating the database - otherwise ACID transactions are not supported.

c) However, support for ACID transactions has become recognized as a key feature of relational DBMS's.

B. However, the relational model itself does create some difficulties because of what has come to be known as the "impedance mismatch" between OO programming and data storage in relational tables.

1. The OO and relational models have very different histories.

a) OO: Comes out of the world of discrete simulation. Much subsequent work was motivated by the development of GUI's in the desktop/laptop/mobile device world. OO is the natural paradigm for designing a GUI and for certain kinds of problems involving objects with complex structure (e.g. computer-aided design).

b) Database systems: comes out of the world of business data processing. Much of the work has been done in the mainframe world. Database systems historically have had a strong batch processing flavor.

2. The OO and relational have very different fundamental concepts.

a) In the OO world, an object has three essential properties: identity, state, and behavior. These are distinct ideas, in the sense that it is conceivable that two objects with different identity might have the same state. Of course, all objects of the same class have the same behaviors.

Typically, the identity of an object is established by the location in memory where it resides - i.e. when all is said and done an object's identity is basically a memory address. This, of course, is not of any semantic significance, and may even change from run to run of the same program.

b) What are the essential properties of a table row? Certainly, a row also has identity and state. But - in the basic relational model, rows do not have behavior. More importantly, though, the notions of identity and state are not distinct: it is inconceivable that a given table might have two rows with same state. [This would violate the set/primary key concept] That is, OO table rows really only have one property: state.

Thus, the identity of an entity is based on the value of certain of its attributes - something which is semantically meaningful, and doesn't change from run to run of the same program.

c) This sometimes leads designers of OO classes to incorporate some sort of "key" attribute in order to provide semantically meaningful identify and to guarantee that no two objects have identical state - e.g.

```
class Product
{
      int id;
      ...
```

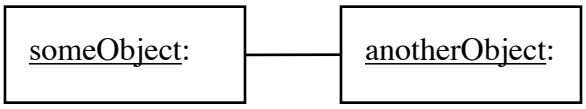
strictly speaking, this is not necessary in an OO system (unless some sort of map is going to be needed to locate the objects, of course) However, something like this is often needed in creating a relational scheme if one cannot guarantee that the remaining attributes of an object will be unique.

d) That is, we have the following:

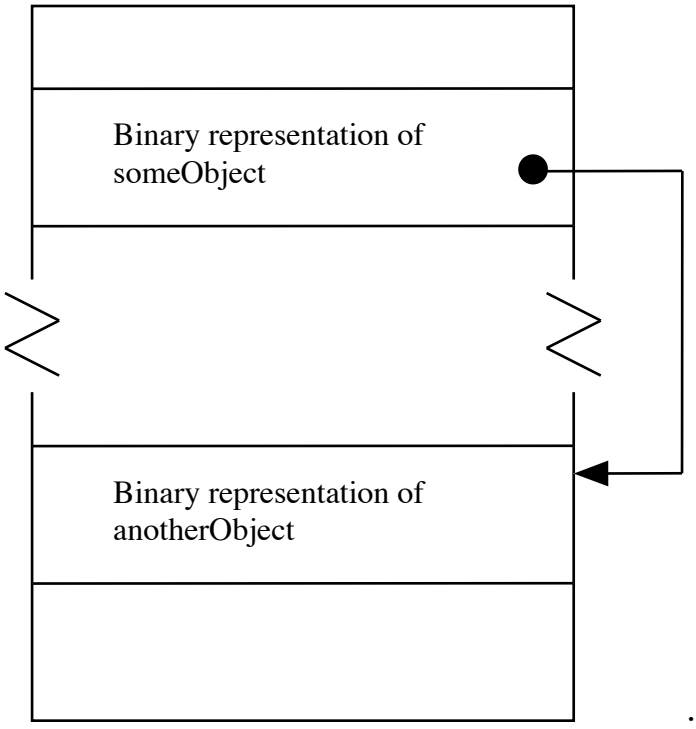
OO:	identity ≠ state
relational:	identity is the same thing as state

3. Related to this is how the two "worlds" handle relationships between objects. Though both can use ER diagrams as design tools, they translate relationships quite differently.

   a) In an OO system, relationships (associations) are handled by pointers (called references in Java) - an object that is associated with another object contains a reference to the location in memory where the other object "lives" - e.g.



   is represented in memory by



   b) In a relational system, relationships are managed by foreign keys. Often, relationships are modeled by tables as well - i.e. a relational database table may correspond to an entity (object) or a relationship (association between objects).

c) There are two consequences of this difference:

(1) Foreign keys are semantically meaningful; pointers to locations in memory or on disk are not (they're just numbers, and not particularly meaningful ones at that.)

Example: Can you easily tell someone your student ID or SSN if asked?

Can you easily tell someone the location on disk where information about you is stored by the Gordon administrative system?

(2) From a performance standpoint, there is a huge difference. One can follow a pointer directly to the object it references; but following a foreign key involves some sort of table lookup. Quantitatively, the time difference - even in the presence of an index - may a less than a microsecond versus a few milliseconds - a ratio of over 1000:1.

(3) As an illustration of the issues of efficiency and semantic meaningfulness, consider the following:

Suppose you wanted to find the office of a given faculty member, given their name. You would have to look up the office number (say on the go site), and then go there - something which would take some amount of time and effort. But if you had the office number to begin with, you could go there directly without having to look anything up.

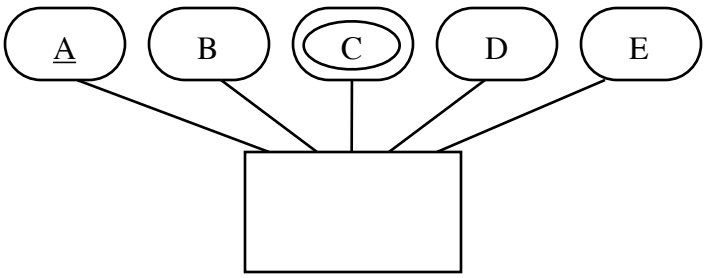
OTOH, as I was revising a similar lecture several years ago, someone asked me "where do I find room 209?" After playing "20 questions" for a while, I determined that what she was looking for was the secretary's office - at which point giving directions became easy. Like most people, I can remember where a person's office is much more easily than I can remember a room number - when I need to drop something off for the secretary, I never think "I go to room 209".

d) That is, we have the following:

OO:	relationship (association) modeled by pointer
relational:	relationship modeled by foreign key

4. Normalization requirements result in often needing to store individual objects in multiple tables.

   a) Relational databases need to be normalized. Even 1NF precludes non-atomic attributes. We have seen how normalization to 4NF requires us to isolate multivalued attributes in their own relation - e.g.something like



   normalizes to two tables: `(ABDE)` and `(AC)` to satisfy the requirements of 4NF

   b) However, OO classes can contain attributes which are collections of various sorts, and can contain non-atomic attributes.

   c) This has some profound implications, which someone has illustrated as follows:

   Is it possible to store a car in standard office filing cabinets?

   ASK

   Yes - if you're willing to take it apart far enough! The problem comes in putting it back together when you need it.

5. The difference between OO and relational representations for information is summed up neatly by a diagram in Sandalage:

   PROJECT Figure 1.1

   where the OO aggregate might be modeled by a class like the following:

```
class someClass
{
    private int id;
    private Customer customer;
    private Set<OrderLine> lineItems;
    private CrediitCard paymentDetails;
    ...
```

6. Some writers have pointed out that there is also a fundamental philosophical difference between the two approaches to representing information.

   a) Database systems seek to foster "data independence" - the idea that data should be stored in such a way as to be independent of the programs that use it - which facilitates sharing of data between application areas and the ability to perform ad-hoc queries.

   b) Object-orientation emphasizes the close connection between data and behavior. An object doesn't just incorporate data - it also incorporates methods that operate on the data. Moreover, one of the components of the OO "PIE" is encapsulation, which precludes operating on data except through the set of methods that a class furnishes.

   c) In a generic database, one often needs to use the data in ways that might not have been anticipated when the database was designed. This can be easily done in SQL, of course, but it means breaking encapsulation if he data being stored actually represents objects that belong to OO classes.

      What does one do then? Wait for new methods to be coded? Break encapsulation? Neither seems like a good solution.

7. Differences like these between the two models create challenges for application programmers who are writing an OO program but need to store data in a relational database. This, in turn, engenders greater development effort for applications, and flies in the face of approaches such as agile development.

C. There are several ways one might approach these problems.

1. One might simply live with the differences. Essentially, this is what many existing systems do. This means, of course, that the programmer must consciously arrange for transmission of entities between main memory and the database, and must explicitly code transformation between information representations when needed.

2. Another possibility is to develop a new object-oriented database model. There was a lot of interest in this at one time, but efforts to produce a standardized OO model have been unsuccessful and there is currently little work in this regard except for certain niche areas.

   a) For example, the 3rd and 4th editions of our text had a separate chapter on "Object-Oriented Databases", while the 5th edition combines this with the Object-relational approach to produce a single chapter on "Object-Based Databases".

   b) However, OODBs are quite important in a number of niche markets (e.g. medical information systems), and new developments in this area are always possible.)

3. A third possibility is to extend the relational model to make it a better match for the requirements of OO - creating what is often known as an object-relational model.

   a) Several of the extensions to SQL in SQL 1999 (and beyond) have been concerned with reducing the impedance mismatch between the

relational model and the needs of OO systems, by moving away from the restrictions the straight relational model imposes on attributes.

(1) Allowing non-atomic data types

(a) Columns whose values may contain internal structure (i.e. have fields of their own.

The SQL 1999 standard specifies a `create type .. as` facility which allows defining a structured type.

(b) Columns which may store a collection (e.g. set, array) of value rather than a single value - a relation nested within a relation. A key motivation here is efficiency: multivalued attributes are associated with multivalued dependencies, which force decomposition during normalization and lead to the need for joins in queries. However, a join is inherently a computationally expensive operation., and transferring an object containing a collection between memory and the database typically requires a loop that translates between members of a collection and rows in some table arising from normalization.

The SQL 1999 standard defined `array` and `multiset` data types. A table column can be defined as some standard type, followed by either `array[some constant]` or `multiset` . What this does, in effect, is to create a attribute for each row in the table which is itself another tabe. The DBMS transparently translates between a collection field in an object and a table columns in a database row.

(c) Support for complex data types (binary large objects (`blob`) and character large object (`clob`)).

The DBMS has no knowledge of the internal structure of a large object; it simply allows reading/writing the entire object.

(d) Reference data types

i) System-generated object identifiers (oids)

ii) The ability for a field to store the oid of a row in another table (in effect a pointer to it) rather than a key that must be looked up. Again, efficiency is a key motivation.

(2) Direct support for inheritance

(a) Types based on other types - so that a type includes fields of its own plus fields inherited from a parent type or types.

(b) Tables based on other tables - so that a given row, when inserted into a subtable, also becomes a row in a base table. (This moves into the realm of class = entity-set, of course)

(c) Note that these are really two different extensions - in the one case the domains of individual columns use inheritance; in the other case, whole table structures use inheritance. A given database implementation may, of course, implement neither, either or both.

(3) Storing procedures in the database along with the data

(a) As methods of specific data types.

(b) As "stand-alone" functions or procedures.

b) Extensions intended to make SQL be a more fully-functional programming language in its own right, including various control structures and an ability to call routines written in other languages from within SQL.

Example: the `CASE` construct in SQL you used in an earlier homework.

c) Various commercial systems developed during the 1990's incorporated some or all of these facilities, which became part of the 1999 and subsequent SQL standards - though standardization of implementations is still a ways off and no vendor comes even close to fully supporting the standard. (And vendors often do things contained in the standard in their own, non-standard way.)

4. A final approach is to move away from the relational model altogether - which leads to various models lumped together under the "NoSQL" heading, which we will look at shortly.

D. Another reason for interest in non-relational models arises from support for ACID transactions in the context of cluster or distributed systems.

1. Traditionally, databases have been stored on single computer systems. If more performance was needed, the solution as to use a faster computer and disk subsystem.

a) But there are fundamental limits on this, of course.

b) Typically, when high performance is needed, the solution of choice has been to use clusters of scores, hundreds, or even thousands of inexpensive processors - perhaps housed in a single location or multiple locations.

c) However, if the database is stored on a single disk subsystem, then it becomes the bottleneck for performance.

(1) This might be addressed by "sharding" the data so that different portions are stored on different systems - which would mean that for many transactions only a single system needs to be accessed.

Note that this is similar to what we called "horizontal partitining" - except that here the motivation is performance rather than distributing control of the data. The sharding scheme may be set up

intentionally, or the DBMS itself may handle the sharding and the mapping of accesses to the correct shard.

(2) Doing this in a location-transparent way is not easy, though. This becomes particularly important in the context of cloud systems, where a database may be housed on a platform leased form a vendor that runs on numerous individual CPU's and disks - perhaps at different physical locations.

(3) But if multiple disk are used, then support for ACID transactions entails communication between the various processors and/or physical locations.

2. If the system is distributed across multiple locations - as would be the case if robustness in the face of physical problems is needed - then, the CAP theorem says that you can have any two of Consistency, Availability, and Partition Tolerance - but you cannot have all three. (Recall that, in the context of the CAP Theorem, availability is defined in this way "Every request received by a non-failing node in the system must result in a response" [Lynch and Gilbert cited by Sandalage and Fowler]).

   a) Since the kinds of physical failures that result in partitioning a network are not avoidable, sacrificing Partition Tolerance is not usually an option, since this would mean shutting down the entire system should a partition occur.

   b) In practice, then, this boils down to saying that there is a tradeoff between Consistency and Availability - where there is no one "right" answer for all systems.

3. The relational model exacerbates the problem because normalization typically requires partitioning entities across multiple tables - increasing the number of tables that must be updated atomically.

4. Again, this becomes a motivation for moving away from the traditional relational model.

E. The impedance mismatch between the relational data model and the requirements of OO, together with the performance implications of supporting ACID transactions in the relational model on high performance systems has led to an interest in exploring other data models that differ significantly from the relational model.

1. These models are known collectively as "NoSQL models". Actually, that's basically a term of convenience and something of a misnomer.

   The term was originally used as the name for a relational database that didn't support SQL.

2. The term has come to be used collectively a variety of different data models - most of which share some common characteristics.

   a) They are not relational, and don't use standard SQL.

   b) They have no database schema - the structure of individual records is dynamic and can vary from record to record.

   c) They are generally - but not always - open-source.

   d) They are generally - but not all - cluster-oriented.
   e) The first two of these (not relational, no schema) are characteristic of all NoSQL models. The last two (open-source, cluster-oriented) are not true of all.

3. Sandalage and Fowler distinguish four broad categories of NoSQL models: key-value, document, column-family, and graph. These are all quite different from each other!

F. Before we look at these models in detail, though,, it is worth noting that there are things that are lost by going this way - hence the need for considering tradeoffs.

1. The Sandalage/Martin book draws a distinction between what it calls "integration databases" and "application databases".

   a) An integration database supports integrating multiple applications using a common database. This is a strength of the relational model.

   b) An application database is "owned" by a single application (like in he old file-processing days) - though this can be ameliorated by making data available through web services.

   c) Relational databases do well as integration databases; while NoSQL databases can serve well as application databases but do not support integration of multiple applications with diverse requirements well.

      One writer (C.J. Date) puts it this way: "Although the programming language and database management disciplines certainly have a lot in common, they do also differ in in certain important aspects (of course). To be specific: * An application program is intended - by definition - to solve some specific problem.* By contrast, a database is intended - again by definition - to solve a variety of different problems, some of which might not even be known at the time the database is established." (*An Introduction to Database Systems* - 7th ed (Addison Wesley. 2000) p. 813)

2. Support for ad-hoc queries. It would be hard to imagine an ordinary user formulating queries interactively in a programming language such as C++, Java, or C# - though some of the NoSQL models do provide some support for interactive queries, even including some SQL-like facilities, as we shall see.

3. Support for “set at a time” processing - to perform some operation on all the members of a top-level collection, one must code a loop using something akin to an iterator.

4. Support for referential integrity through notion of keys, etc.

5. For this reason, Sandalage and Martin argue for the notion of "polyglot persistence" - choosing the appropriate model to fit a particular set of needs. This may be a relational database or one of the NoSQL models.

II. Database Models Based on Aggregates

A. Three of the four kinds of NoSQL model store data in aggregates, rather than in tables. Something of the difference between the two approaches can be seen from examples in the Sandalage and Fowler book:

1. The reality to be modeled.

   PROJECT Sandalage Figure 2.1

2. A corresponding relational database structure

   PROJECT Sandalage Figure 2.2

3. An aggregate structure using two kinds of aggregates - customer and order. (Full information on products is represented elsewhere).

   PROJECT Sandalage Figure 2.3 and JSON for aggregates

4. Orders could also be embedded in the customer aggregate.

   PROJECT Sandalage Figure 2.4 and JSON

B. Two things are gained by using aggregates.

   ASK

1. Fewer disk accesses - in the second case everything needed to handle an order can be transferred by one disk read and one disk write.

2. Simpler code.

C. Some things are also lost by using aggregates.

   ASK

   1. The ability to access parts of an aggregate without writing special code - e.g. the ability to look at all orders for a given product, rather than for a given customer (requires accessing all the customer aggregates.)

   2. Sandalage and Fowler cite this as another example of the difference between integration and application databases.

D. Aggregates facilitate sharding, since if a database using aggregates is sharded, data that is accessed together will often end up in the same aggregate and hence in the same shard.

E. Several of the NoSQL models are built on use of aggregates.

   1. Key-value databases store key-value pairs - the key is a unique identifier and the value is an aggregate, whose internal structure is opaque to the database. (Thus all the database can do is allow the program to access, store or delete aggregates by key, but getting at what is stored within is the responsibility of the application program.)

      a) The value associated with a given key may be stored as JSON - as in the examples earlier of aggregate structures, or XML, or any sort of binary data.

      b) Sandalage and Fowler suggest places where a key-value store might be used, including:

(1) Shopping carts - the key would be the user's identifier (like the user's username or email), and the value would be the entire contents of the shopping cart. Stored this way, the entire cart can be accessed by a single disk access.

(2) User preferences / profile information - again the key would be the user's identifier and the values would be the users' preferences / profile - all accessible via a single disk access.

c) They also cite places where a key-value database might not be desirable.

(1) Applications involving relationships between items that are not part of the same bucket.

(2) Transactions that involve operations on two or more different aggregates - since. while access to a single aggregate is atomic, there is no mechanism for accessing several aggregates as a single atomic operation.

(3) Queries based on data stored in an aggregate.

(4) Queries requiring access to multiple aggregates (set at a time processing.)

d) Amazon's DynamoDB - which it part of the Amazon Web Services (AWS) portfolio is a member of this family.

2. Document databases are similar to key-value databases, but have some knowledge of the internal structure of the aggregate - and so can allow some access to the content of the aggregate through the DBMS.

a) This is done by requiring the document to be stored using a notation such as JSON or XML or similar.

b) However, there is no schema requiring all documents to have the same structure.

c) The DBMS is able to perform queries and based on the content of the document, not just on the key, and can do partial updates of the content of the document.

d) The DBMS also supports indexes based on the content of aggregates.

e) MongoDB - a widely-used open-source DBMS - is an example of a member of this family that stores documents represented using JSON.

3. Column-family databases can be thought of as two-level aggregates - i.e. each aggregate is itself a map of keys and values stored in that aggregate. The result ends up looking something like this:

   PROJECT Sandalage/Fowler Figure 2.5

   Google's BigTable is an example of a member of this family

III.Graph Databases

A. As we pointed out earlier, the term "NoSQL" is a generic term that encompasses a wide variety of databases. We now consider a type of database that is included in the NoSQL category, though it is not an aggregate-oriented model (though it does share the other characteristics of NoSQL databases: the Graph Database.

B. One of the weaknesses of aggregate-oriented models is that relationships are only represented within aggregates (e.g. by physical proximity). There is no support for relationships between entities in different aggregates. A graph database, on the other hand, models relationships explicitly by means of nodes that represent entities and links that represent relationships (that can have names and values).

   PROJECT: Sandalage/Fowler Figure 11.1

C. Though both relational and graph databases provide for modeling relationships between entities, the way that they do it is very different.

1. In a relational database, the relationships are part of the schema - so every row in the same table participates (at least potentially) in a defined set of relationships.

2. In a graph database, there is no schema that defines things like:

   a. What relationships a given node can participate in.

   b. What properties a node may have.

   c. What properties a relationship may have.

   PROJECT Sandalage/Fowler Figure 11.2.

D. Graph databases support a variety of queries based on traversing the graph.

E. In contrast to the other NoSQL models, many graph databases are designed to run on a single system, rather than a cluster. If sharding becomes necessary for performance reasons, the shards need to be based on application-specific criteria, as in the following example: But note that graph databases work typically work better with a single node.

   PROJECT Sandalage/Fowler Figure 11.3

   (Note that this is akin to the notion of horizontal partitioning in a distributed relational database.)

F. Graph databases support ACID transactions involving various nodes and relationships.

IV. Other Issues

A. NoSQL databases are schema-less. (The technical term is they have emergent schemas - i.e. each aggregate has its own schema). This has advantages and disadvantages.

1. Advantages

    a. No need to predefine data structures.

    b. Easy to change as needed.

    c. Good support for non-uniform data.

2. Disadvantages

    a. Potential for inconsistent names for the same value in different aggregates - e.g. quantity, Quantity, QUANTITY, qty etc.
    (A scheme in a relational database would prevent this)

    b. Instead, there is an implicit schema which the application must now enforce. So you need to look at the code to see what the scheme is.

    c. What do you do if multiple applications need to access the same database?

B. In an aggregate-oriented database, relationships can pose problems.

A. There may be no DBMS support for traversing relationships (equivalent to foreign keys in a relational database), though some provide mechanisms to make link-walking easier.

B. But since there is no support for updating more than one aggregate atomically, updating relationships consistently can present a challenge.

C. There is no support for set-at-processing. So something like querying all the orders for a given item (across multiple aggregates) is complex.

Some NoSQL databases provide for precomputing results of such queries and storing them as materialized views. (Some relational DBMSs have something similar).

But how does one keep matrialized views up-to-date. Two basic approaches:

1. Eager approach - update view when the base data is updated. This gives good support for frequent reads of a view that needs to be always current, but adds complexity to update operations.

2. Regular batch updates of materialized views. The view will almost always be a bit stale, but in many cases this can be tolerated.

V. Map-Reduce

A. Map-reduce is a programming paradigm that supports a very high degree of parallelism.

1. It is not directly tied to NoSQL databases, but is often used in conjunction with them.

2. For example: Google makes considerable use of the map-reduce paradigm, often using data stored in the BigTable database model they developed.

B. This paradigm makes use of two functions - a mapper and a reducer.

1. The mapper processes aggregates to produce key-value pairs. Since the processing of each aggregate is independent of the others, this can be done using massive parallelism.

2. All the key-value pairs having the same key - presumably from multiple processors are collected and submitted to a reducer, which reduces them to a

single value. Reducing for different keys can also be done using massive parallelism.

Example: PROJECT