CPS352 Lecture - NoSQL Database Models

Last revised January 10, 2019

Objectives:

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

Materials:

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

I. **Introduction**

A. As we noted, last class, 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.

B. However, significant issues arise with the use of the relational model for some - typically very large - applications.

II. Issues Arising From the Relational Model

A. One of the key reasons for for interest in non-relational models arises from the ramifications of support for ACID transactions in the context of cluster or distributed systems.

1. Traditionally, databases have been stored on single computer systems.

   a. If more performance was needed, the solution was to use a faster computer and disk subsystem.

      (1)But there are fundamental speed and capacity limits on this, of course.

      (2)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.

   b. Often distributed systems are used to prevent temporarily losing the ability to access the database at all in the event of the failure of a single site due to power or network issues.

2. For a variety of reasons, it is generally impractical to store a complete copy of the database on each of the systems managing it - which would entail maintaining hundreds or thousands of complete copies of the database.

   a. Instead, this is typically addressed by "sharding" the data so that different portions are stored on different systems - which could mean that for many transactions only a single system needs to be accessed.

      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.

    b. In addition, each shard is generally stored in more than one place to avoid losing access to its data if the system on which it is stored goes down - which implies than an update transaction will need to update multiple copies.

    c. Support for ACID transactions then entails communication between the various processors and/or physical locations when a transaction involves accessing data stored at more than one location or updating replicated data.

3. A theorem known as the CAP theorem says that, in a multi-system context, you can have any two of Consistency, Availability, and Partition Tolerance - but you cannot have all three. (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 Sadalage and Fowler]).

    a. Since the kinds of physical failures that result in partitioning a network are not avoidable, sacrificing Partition Tolerance is generally not 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.

    c. This is a motivation for moving away from a database model that entails support for ACID transactions.

4. 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.

   This becomes a motivation for moving away from the traditional relational model.

B. The Normalization requirement for Relational Databases leads to needing to use joins - sometimes several of them - for many queries. But joins are computationally expensive, which becomes a significant issue when handling a large volume of queries.

C. Another Issue Relates to Structured and Unstructured Data

1. The relational model expects that every row in a given table has the same set of columns, each of which holds the same type of data (whether atomic or non-atomic in a relational extension). The structure of the individual tables - and their relationships to one another - is represented by an explicit schema.

2. While this works well for many applications, there are some applications that require the ability to handle data that is inherently nonstructured - eg processing involving the <u>content</u> of social media posts.

D. The impedance mismatch between the relational data model and the requirements of OO we talked about last time, together with the performance implications of supporting ACID transactions in the relational model on high performance systems, the dependence on joins for many queries, and the need to deal with unstructured data in some cases 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 <u>relational</u> 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 - though some support a variant of 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. Sadalage and Fowler distinguish four broad categories of NoSQL models: key-value, document, column-family, and graph. These are all quite different from each other!

E. 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 Sadalage/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, Sadalage 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.

III. **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 Sadalage and Fowler book:

1. The reality to be modeled.

PROJECT Sadalage Figure 2.1

2. A corresponding relational database structure

   PROJECT Sadalage Figure 2.2

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

   PROJECT Sadalage Figure 2.3 and JSON for aggregates

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

   PROJECT Sadalage 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. Sadalage 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) Sadalage 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 Sadalage/Fowler Figure 2.5

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

IV. 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 <u>within</u> 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: Sadalage/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 Sadalage/Fowler Figure 11.2.

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

F. 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 Sadalage/Fowler Figure 11.3

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

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

V. 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 materialized views up-to-date. Two basic approaches:

4. 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.

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

VI. 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.

   (Google has recently replaced this with a more flexible system called Cloud Dataflow)

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