Emergence of NoSQL Databases

Reading

- NoSQL Distilled[1] chapter 1

Reading questions

- What are the main benefits of relational databases (and databases in general) as a storage model for application data? (NoSQL Distilled, section 1.1)
- What is an **application database** and what is an **integration database**? What are some concerns that one type has that the other does not? (section 1.3)
- Explain in what sense web services replaced integration databases. (section 1.3)
- What effect did the emergence of web services have on the use of non-relational databases? (section 1.3)
- What are examples of large data sets that came about with the increase in scale of modern large web properties? (section 1.4)
- What are the two main ways of dealing with an increase in traffic? (section 1.4)
- What benefits do **clusters** bring? (section 1.4)
- What problems do clusters bring on the relational database model? (section 1.4)
- What is **sharding**? What problem does it solve? What limitations does it impose on queries and integrity? (section 1.4)
- What are the names of the databases that Google and Amazon created to deal with their scaling issues? What were their reasons for building these databases?
- Do NoSQL databases have no query language?
- NoSQL databases tend to use no schema. What are some advantages of this? (section 1.5)
- What are the main two reasons to consider a NoSQL database? (section 1.5)
- What does the term **polyglot persistence** mean?

Notes

Benefits of Relational Databases

Relational Databases solve a number of problems that modern applications face:

Persistence As all "databases", they offer a **persistent storage** for our data, organized in a flexible way that allows us to reach in and retrieve exactly the parts we need. By contrast, navigating a file system in search of information can be cumbersome.

Concurrency With the use of **transactions**, relational databases offer protection against the problems arising from **concurrency**, when multiple parts of the application try to access and change the same piece of data at the same time.

[1] http://learning.acm.org/books/book_detail.cfm?id=2381014&type=safari

Integration Relational databases with their uniform interface offer a smooth **integration** into other applications and allow for easier **inter-application collaboration**. This is further amplified by the use of a common standard model, the SQL model. With only slight variations, all relational database vendors offer essentially the same functionality.

A main drawback is what is known as Impendance Mismatch:

Impendance Mismatch The way data is stored in a relational database is fundamentally different with the way it is stored in the application memory. Most languages employ objects with nested properties that correspond to a possibly complex set of SQL table entries. A translation needs to be made between the SQL record tables and application objects. This is often facilitated by technologies like ORM.

Different database usage

Databases can be used in two fundamentally different ways:

Integration Databases Integration Databases interact with multiple applications, written by various teams. A relational model offers great advantages there as a common usage model for all these different teams. Changes to the database schema however are difficult to make, as multiple codebases all read from and write to the database.

Application Databases Application Databases communicate with a single application, written by a single team. That application is then responsible for interacting with the other applications that need to use the database. Therefore the interface boundary is now at the application API, and the internals of the database storage are only the concern of the single application responsible for managing it. The only thing that the other applications need to know is how to interact with the application responsible for the database, and for that they just need its API.

A primal example of this is the web services we are discussing. These can act as the single application that is responsible for the database, and everyone else has to go through the service's API. That API can therefore stay relatively stable while the database internal change. This offers considerable flexibility on the kind of database to use.

Clusters and Relational Databases

As big companies like Amazon and Google came to the forefront, demands on database storage grew tremendously, from access logs to sensor data to massive amounts of user data. In order for a project to scale, you can either opt for *larger* machines, or for *more* machines. With the prohibitive costs of larger machines as well as their limits, most companies aimed for using more hardware, in the form of **clusters**.

For instance there are numerous clusters of computers around the world that service Google search queries. When you do a Google search, one of these clusters will be contacted, mostly likely based on your geographical proximity to it and its current workload.

The problem that arises however, is that these clusters must still address the same underlying data storage, turning that portion of the query into a bottleneck. One approach to reduce the problems is to **shard** the database, whereupon each shard contains different entries. So different shards can be accessed simultaneously by different clusters, and thus speed up the processing. This does however cause considerable complications when it comes to managing the storage of information into multiple shards, and maintaining transactional integrity and querying capabilities.

The emergence of NoSQL Databases

In the presence of clustering and the problems when clusters work with relational databases, coupled with the impendance mismatch problem discussed earlier, a number of database approaches came forward. These are collectively called "NoSQL", though this is hardly an apt term.

These databases vary a lot in their approach:

column-oriented databases like BigTable, Cassandra and HBase focus on column families as the primary unit of storage, as opposed to table rows. In effect each row contains sets of column values organized in "families". This provides a richer structure to the normal relational database approach.

document-oriented databases like MongoDB and CouchDB focus on complex aggregate units called *documents* as their primary unit of storage. We can loosely think of a document as a JSON structure with all the information contained therein.

graph databases like FlockDB and Neo4j focus on relations between objects, and they are particularly effective at encoding the complex interactions in social networks.

key-value stores like Riak and Redis offer very simple storage of values based on their keys. As a result they can be extremely efficient and fast. They are often used as in-memory cache stores.

What they all have in common is that they do not follow the relational model, that they are schema-less, and that they are designed to perform well in clustered settings.