Relational Model vs Document Model

The best known data model is probably that of SQL, based on the relational model proposed by Edgar Codd in 1970. Data is organized into relations (called tables in SQL), where each relation is an unordered collection of tuples (called rows in SQL)

By the 1980s, RDBMSs and SQL became the tool of choice, and has largely remained so for 30+ years -- an eternity in computing history!

Since then, the following have tried but withered away:

- network model
- hierarchy model
- object dbs
- XML dbs

NoSQL

In the 2010s, NoSQL came to be -- an unfortunate name since it doesnt actually refer to any particular technology -- now interpreted as Not Only SQL.

Why?

- A need for greater scalability than relational dbs can easily achieve, including very large datasets or very large throughput
- A widespread preference for open source produts
- Specialized query ops that are not supported well by relational model
- Need for more dynamic and expressive data model

The object-relational mismatch

App development is done mostly in OOP, which leads to a common criticism of the SQL data model -- if data is stored in relational tables, an awkward transitional layer is required between objects in application code and the database model of tables, rows, and columns -- this is known as *impedence mismatch.*

Object-relational mapping (ORM) frameworks like ActiveRecrod an dHibernate reduce some of the boilerplate here, but it is still overhead.

1-to-many relationships

Representing 1-to-many relationships (one person → many jobs) in a SQL table can incur

significant overhead

- resume example. Variable number of jobs and education quals. How do you model in a sql store?
  - pre 1999SQL normalized representation: put positions, education, and contact info in separate tables, with a foreign key reference to the users table
  - later versions of the SQL standard added support for strucutred data types and XML data -- allowing multi valued data to be stored within a single row, with support for querying and indexing within the documents
    * eg PostgreSQL and MySQL supports a JSON datatype
  - another option is to encode jobs, education, and contact info as JSON or XML document and store it as a text columns -- let the application interpret its structure and content
    * typically cannot use the database to query for values inside the encoded columns
- for a self contained *document* data structure like a resume, a JSON representation can be quite appropriate
  - supported by document oriented databases like MongoDB, RethinkDB, CouchDB, Espresso
- The JSON model perhaps reduces impendence mismatch, but it has its own problems as a data encoding format -- the lack of schema can sometimes hurt
  - it does however have better *locality* than an equivalent multi table schema -- one query is is sufficient since all the information is one place, vs having to do multiple lookups.joins to get what you need

Many-to-1 relationships

Why refer to an ID that identifies a canonical representation of a piece of text vs encoding the free text directly in a storage system?
- consistency in style and spelling across profiles
- avoiding ambiguity
- ease of update
- localization support, for translation and the like
- better search

Whether you store an ID or the text it represents is a question of duplication. Would you rather duplicate a meaningless ID? Or the human-meaningful text itself?

An ID reference never needs to change even when the information it identifies changes.
Rule of thumb: anything that is meaningful to humans may need to change in the future, and if this information is duplicated, all the redundant copies need to be updated. This incurs write overheards and risks inconsistency.
Removing this duplication is the idea behind *normalization* in databases.

However, normalizing data requires many-to-one relationships (eg: many people → one city), which does not fit nicely into the document model, because joins are often not naturally supported!
- Unlike in relational dbs, where it is normal to refer to other tables by an ID, because joins are easy. In document databases, joins are not needed for many-to-one tree structures, and support for joins is weak.
    - So if you want to maintain canonical lists with a document db, the work of making the join is probably going to be shifted from the database to the application code

Many-to-many relationships

Another general rule of thumb: even if the initial version of an application fits well in a join-free document model, data has a tendency of becoming more interconnected as features are added to applications -- which means a relational model may be more useful in the long run?
- eg: many schools → many people -- representation these relationships requires references, and joins when queried
- modelling many-to-many relationships is usually better handled by a relational model, or least a model that easily allows for joins

In short:
Document databases are excellent for 1-to-many relationships, vis a vis relational dbs.
But can struggle with Many-to-1 and many-to-many relationships, vis a vis relational dbs, since they dont natively support joins.
- Should a developer duplicate data (giving up on normalization)? Or manually resolve references from one record to another?

Are document databases repeating history?

The debate about how to represent many-to-many relationships in a database is far older than the recent emergence of NoSQL -- it goes back to the earliest computerized db systems

IBM used a 'hierarchical' model, which has many similarities to JSON -- it represented all data as a tree of records nested with records
- it ran into the same issues with many-to-1 and many-to-many relationships
- out of this debate came the relational model and the network model, the latter eventually fading away

Network model -- CODASYL -- allowed for multiple parents per record -- ie "Greater Seattle Area" could be linked to every user that lived in it
- the links between records in the model were not foreign keys, but more like pointers in

a programming language (while still being stored on disk)
- the way to access a record was to follow a path from a root record along a chain of links -- called an access path
- Trouble is that in this model, you can have cycles to different records -- so the programmer had to account for all the access paths leading to the same record

The network model made querying and updating a database complicated and inflexible.
- rewriting code to to handle new access paths was difficult -- making it difficult to change an application's data model

By contrast, the relational model just lays on all the data out in the open with no complicated access paths to access a record.
- The Query Optimizer decides which order to execute a query in -- ie the 'access path' -- and which indexes to use -- this is abstracted away from the developer, unlike with the network model!
- It is easy to add new records to the database without any concern for adding new paths or existing foreign key links

Document databases are similar to the hierarchical model in one aspect: they store 1-to-many relationships within their parent record, rather than in a separate table.

However, when it comes to representing many-to-1 and many-to-many relationships, modern document databases are similar to relational dbs. They use a *document reference* in lieu of a *foreign key identifier.* The identifier is resolved are read time using a join or follow up queries. This is unlike CODASYL

Document vs Relational databases today

There are many differences to consider between the two
- fault tolerance properties
- handling of concurrency

However, this discussion will just focus on the data model differences.

Args for document dbs:
- schema flexibility
- better performance due to locality
- for some applications, it is closer to the data structures that are used by the applications

Args for relational dbs:
- better support for joins

- therefore better support for many-to-one and many-to-many relationships

If your application uses data with a document-like structure (ie a tree of one-to-many relationships where typically the entire tree is loaded at once) -- then is probably useful to use a document model. Or if there is no relation between records.

- *Shredding* a document into mutliple tables can be cumbersome

There are limitations to this model -- you cannot refer directly to a nested item within a document, instead having to refer to its index (e.g. 'the second item in the list of positions for user 251") but this may not be a problem if the data structure is not too deeply nested.

Poor join support may or may not be a problem. For example, many-to-many relationships may not be needed in an analytics application that uses a document db to record which events occurred at what time.

However, if an application does use many-to-many relationships, the document model becomes less appealing. There are options:
1. reduce the need for joins by denormalizing: but then more work needs to be done to keep the denormalized data consistent
2. emulate joins in application code by making multiple requests to the db: but this induces significant complexity into the application code, and is likely going to be slower than an optimized join performed by db software.

For some highly interconnected data structures, a graph-like data model may be most appropriate.

Schema flexibility?

Most document dbs, and the JSON support in relational dbs, do not enforce a schema on the data in documents. XML support usually comes with optional schema validation. No schema means that arbitrary keys and values can be added to a document, and when reading, clients have no guarantees as to what fields the documents may contain.

'schemaless' is a bit misleading -- clients usually have an implicit expectation of what data they will receive, it is just not enforced by the database.

A more accurate term is *schema-on-read*, in contrast with *schema-on-write.*
Schema-on-read is similar to dynamic (runtime) type checking in programming languages, while schema-on-write is similar to static (compile-time) type checking.

The difference in approach is noticeable when you need to change the format of the data.

E.g. from 'full_name' to 'first_name', 'last_name'

- in a document store, you can add arbitrary keys, and check in the application code for the existence of a key to determine whether it exists or not. On read, you can create the new key with the data you want.
- in a relational db, you generally need to perform a schema migration: ALTER then UPDATE
  - these may be unfarily maligned -- ALTER is not usually terribly slow, though mySQL does create a new copy of the existing table, which can mean several minutes of downtime when altering a large table
  - UPDATE likely will take a while with a large table. However, as wth document dbs, you can make the update change on read as well (instead of doing a 'backend' migration)

Schema on read is advantageous if the items in the collection are hetergeneous, e.g. there are many different kinds of objects and they are all in the same table, or if the structure of the data changes frequently for reasons beyond the developer's control.

But when the records are expected to have the same structure, a schema is likely a useful way to enforce the structure.

Data locality for queries

A document is usually stored as a single continuous string (JSON, XML) or a binary variant thereof (Mongo's BSON),

If the application needs to often access an entire document, there is an advantage to this storage locality. If the data is instead split across mutiple tables, mutliple index lookups will be required to retrieve them, which usually means multiple disk seeks.

However, this is only an advantage if you need to access the whole document at the same time. For large documents, the db will have to spend time loading the whole thing.

And on updates, the whole document will need to be re-written.

So it is generally useful to keep documents small and avoid writes that increase document size.

Data locality is not restricted to the dcoument model - for e.g. Google's Spanner db offers locality properties in a relational model. The Column family concept in Bigtable does something similar.

All in all, relational dbs and document models are getting more similar to each other -- this is a good thing!

Query Languages

Imperative vs Declarative

- an imperative language tells the computer to perform certain operations in a certain order
- in a declarative, you specify the pattern of the data you want -- what conditions need to be met, how you want the data to be transformed -- but not *how to achieve* the goal!
  - it is left to the database's query optimizer to decide which indexes and join methods to use, and in what order to execute the query
  - hides implementation details of the database engine, which makes it possible ot introduce perf improvements to the database without changing any queries
  - declarative languages often lend themselves to parallel execution
    * CPUs are faster today because of more cores, not because wall times are any better
    * imperative code must be executed in an order -- declarative languages only specify the pattern of ther esults, not the algorithm used to get them

In general, for data querying, declarative languages are much better

MapReduce Querying

MapReduce

- a (fairly low level) programming model for distributed execution on a cluster of machines

MongoDb Map Reduce paradigm

- map and reduce functions must be *pure* functions -- they can only use the data passed to them as input, and cannot have any side effects
- can be a bit tricky to write two carefully coordinated js functions -- and makes performance optimization harder
- Mongo allows for *aggregation pipeline* declarative query language -- exposes SQL like declarative semantics

Graph-like Data models

Earlier, we noted that many-to-many relationships are an important distinguishing feature between different data models.
What if many-to-many relationships are very common in the data? Perhaps it becomes

natural to model the data as a graph..

Graph = vertices/nodes + edges/relationships

Vertices dont have to be homogeneous -- can store different types of objects in a single database

- for example, facebook stores people <> events <> locations

There are several ways of structuring and querying data in graphs.

1. property graph model
    - Neo4j, Titan, InfiniteGraph
2. Triple-store model
    - Datomic, AllegroGraph, others

Declarative query languages for graphs

- Cypher, SPARQL, Datalog

Property graph model

A vertex consists of:

- a unique identifuer
- a set of outgoing edges
- a set of incoming endges
- a collection of properties (k-v pairs)

An edge consists of:

- a unique identifier
- the tail vertex (where edge starts)
- the head vertex (where edge ends)
- a label to describe the relationship between the two vertices
- a collection of properties (k-v pairs)

A graph store can be thought of as consisting of two relational tables -- one for vertices, one for edges. The edges table has indices on tail_vertex and head_vertex so you can traverse the graph from any vertex by finding its incoming and its outgoing edges