**Chapter 2: Data Models and Query Languages**

“Data models are perhaps the most important part of developing software, because they have such a profound effect: not only on how the software is written, but also on how we think about the problem that we are solving

Most applications are built by layering one data model on top of another. For each layer, the key question is: how is it represented in terms of the next-lower layer? For example:

1. As an application developer, you look at the real world (in which there are people, organizations, goods, actions, money flows, sensors, etc.) and model it in terms of objects or data structures, and APIs that manipulate those data structures. Those structures are often specific to your application.
2. When you want to store those data structures, you express them in terms of a general-purpose data model, such as JSON or XML documents, tables in a relational database, or a graph model.
3. The engineers who built your database software decided on a way of representing that JSON/XML/relational/graph data in terms of bytes in memory, on disk, or on a network. The representation may allow the data to be queried, searched, manipulated, and processed in various ways.
4. On yet lower levels, hardware engineers have figured out how to represent bytes in terms of electrical currents, pulses of light, magnetic fields, and more.”

Each layer in an application hides the complexity of the layers below it by providing a clean data model.

These layers allow people working on different parts of the application to work together effectively.

This chapter looks at various data models for storage and retrieval.

**Relational Model Versus Document Model**

**Relational model**: proposed in 1970 by Edgar Codd, data is organized into relations (tables) represented as an unordered collection of tuples (rows)

By mid 1980s RDBMSes and SQL were tools of choice for structured data (primarily business data)

In the 70s and 80s, the *network model* and *hierarchical model* were the main alternatives

**The Birth of NoSQL**

Retroactively reinterpreted as *not only SQL*

“There are several driving forces behind the adoption of NoSQL databases, including:

* A need for greater scalability than relational databases can easily achieve, including very large datasets or very high write throughput
* A widespread preference for free and open source software over commercial database products
* Specialized query operations that are not well supported by the relational model
* Frustration with the restrictiveness of relational schemas, and a desire for a more dynamic and expressive data model”

**The Object-Relational Mismatch**

There is a mismatch between objects in object oriented application code and relational databases (aka *impedance mismatch*)

A common option for handling one to many relationships is to normalize the data into separate, related tables.

Another option would be to store the many side record as json or XML within the record of the one side of the relationship

JSON has the advantage of *locality* – all the information associated with a record is stored in one place, rather than being separated out into different tables. A single query gets you everything, as opposed to multiple queries or joins

One to many relationships imply a tree structure, which json makes explicit

**Many-to-One and Many-to-Many Relationships**

Rather than using actual values (like strings), IDs can be used to identify those values, especially if the value is an enumeration. Then anywhere the value is referenced, the ID is used, resulting in deduplication of data. This is referred to as *normalization*.

Normalization requires the implementation of many-to-one relationships, which are not well-supported by the document model. Instead, the tree-like structure of many-to-one relationships is represented in a nested way in the document. In a relational db it requires joins

**Are Document Databases Repeating History?**

Hierarchical models preceded json and have a lot of similarities. Data is represented as a tree of records nested within records, like json can be.

An example of this is IBM’s Information Management System (IMS), which is still around and runs on OS/390 on IBM mainframes

The document and hierarchical models work well for many-to-one, but not many-to-many.

The relational model and the network model are good alternatives for handling many-to-many

**The network model:** standardized by a committee called the Conference on Data Systems Languages (CODASYL) and implemented by several different database vendors; it is also known as the *CODASYL model*

The network model is an extension of the hierarchical model, where instead of each child having only a single parent, they could have multiple parents.

Links between records are more like pointers in a programming language than foreign keys, and an access path is the way to get from one record to another.

**Relational Versus Document Databases Today**

Document model has schema flexibility and locality advantage (better performance), and greater similarity to data models as represented in code

Relational model has better support for joins and many-to-one and many-to-many relationships

**Which to use?**

If the application data has a document like structure, it can be good to use a document database. The relational technique of *shredding* (separating records into different tables), can lead to overly complex schemas and code.

The app may have to refer to nested items, e.g. by getting the second item in a nested list. This isn’t usually a major problem, but deeply nested documents can be problematic.

If the application requires many-to-many relationships, the relational model is likely better.

**Schema flexibility in the document model**

Document databases are often referred to as *schemaless*, however the code that reads the data typically expects some structure.

A more accurate term is *schema-on-read*, where the schema is only enforced when the data is read, as opposed to *schema-on-write*, where the schema is enforced when data is written (like in SQL)

The process of making changes to a schema is different between document and relational dbs. In the document model, fields can simply be added to the new schema and the new code adjusted to handle both the old and the new schemas, or removed in the new schema and ignored by the new code. In a “statically-typed” database schema, a *migration* is required, where the schema must be updated along with the data when fields are added or changed.

Schema-on-read can be good if the documents in the database don’t all have the same structure for reasons like heterogeneity of the data or external data sources where you have no control over the data.

**Data locality for queries**

A document is usually stored as a continuous string encoded as JSON, XML, or a binary representation of those.

If the application often needs to access the *entire document*, there is the advantage of *storage locality* in document databases.

In document database, the entire document is usually read and on update rewritten.

“It’s worth pointing out that the idea of grouping related data together for locality is not limited to the document model. For example, Google’s Spanner database offers the same locality properties in a relational data model, by allowing the schema to declare that a table’s rows should be interleaved (nested) within a parent table [27]. Oracle allows the same, using a feature called multi-table index cluster tables [28]. The column-family concept in the Bigtable data model (used in Cassandra and HBase) has a similar purpose of managing locality [29]”

**Convergence of document and relational databases**

XML and JSON are supported in many databases

“It seems that relational and document databases are becoming more similar over time, and that is a good thing: the data models complement each other. If a database is able to handle document-like data and also perform relational queries on it, applications can use the combination of features that best fits their needs.

A hybrid of the relational and document models is a good route for databases to take in the future.”

**Query Languages for Data**

*Imperative* query languages tell the computer to perform certain operations in a certain order

*Declarative* query languages specify what the output should look like, but not how to do it

Declarative languages have the benefit of being able to do performance optimizations without changing the query.

They also lend themselves well to parallelization:

“Declarative languages have a better chance of getting faster in parallel execution because they specify only the pattern of the results, not the algorithm that is used to determine the results.”

**Declarative Queries on the Web**

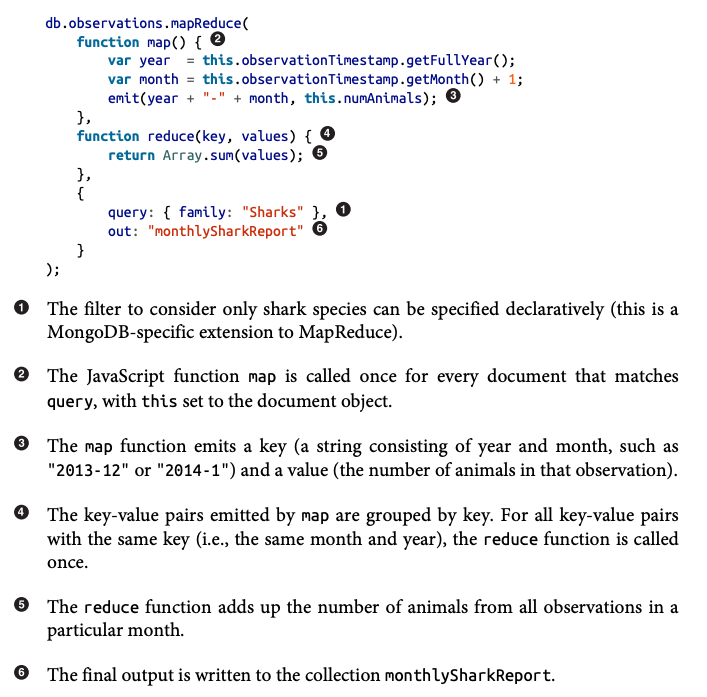
CSS uses a declarative structure for front-end code. A comparison is made to JavaScript’s Document Object Model (DOM), an imperative version of the same thing. The declarative approach (CSS) is easier to read and understand, and is also better for user interactivity.

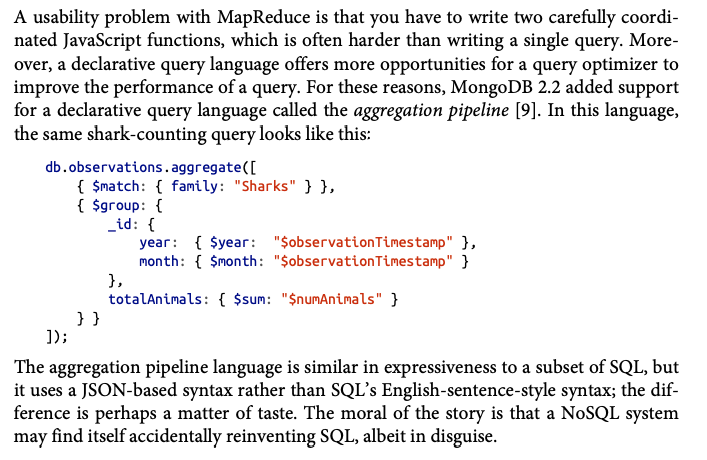
**MapReduce Querying**

“*MapReduce* is a programming model for processing large amounts of data in bulk across many machines, popularized by Google”

Some NoSQL databases (MongoDB & CouchBD) provide limited support for MapReduce for read-only queries across many documents.

Example with MongoDB:





**Graph-Like Data Models**

The document model handles many-to-one relationships well, and the relational model can handle simple many-to-many relationships

But more complex relationships can be better handled by modeling the data as a graph

“A graph consists of two kinds of objects:

* Vertices (also known as nodes or entities)
* Edges (also known as relationships or arcs)

Typical examples include:

*Social graphs*

Vertices are people, and edges indicate which people know each other.

*The web graph*

Vertices are web pages, and edges indicate HTML links to other pages.

*Road or rail networks*

Vertices are junctions, and edges represent the roads or railway lines between them.

Well-known algorithms can operate on these graphs: for example, car navigation systems search for the shortest path between two points in a road network, and PageRank can be used on the web graph to determine the popularity of a web page and thus its ranking in search results.”

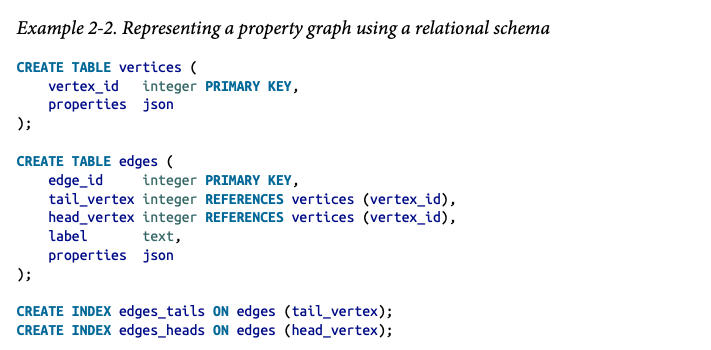
**Property Graphs**

“In the property graph model, each vertex consists of:

* A unique identifier
* A set of outgoing edges
* A set of incoming edges
* A collection of properties (key-value pairs)

Each edge consists of:

* A unique identifier
* The vertex at which the edge starts (the tail vertex)
* The vertex at which the edge ends (the head vertex)
* A label to describe the kind of relationship between the two vertices
* A collection of properties (key-value pairs)”



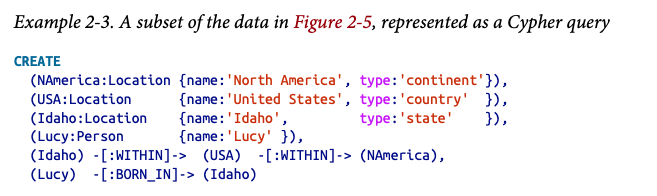
Important aspects of the graph model:

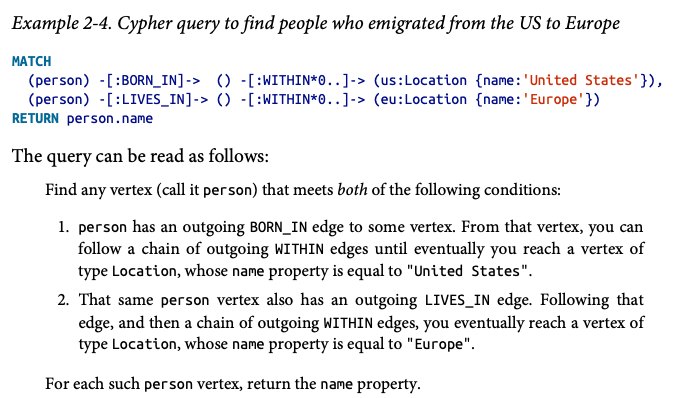
1. Any vertices can be connected. No schema enforces what kinds of things can be associated
2. Given a vertex, you can find all other vertices connected to it by traversing the graph (following a vertex’s incoming and outgoing edges)
3. *Labels* can be used to describe different kinds of relationships, so different kinds of information can be stored in a single graph while still maintaining a simple and clean data model

**The Cypher Query Language**

“*Cypher* is a declarative query language for property graphs

Each vertex is given a symbolic name like USA or Idaho, and other parts of the query can use those names to create edges between the vertices, using an arrow notation: (Idaho) -[:WITHIN]-> (USA) creates an edge labeled WITHIN, with Idaho as the tail node and USA as the head node.”





**Graph Queries in SQL**

Graph queries can be done using recursive CTEs

**Triple-Stores and SPARQL**

pass

**The Foundation: Datalog**

pass

**Summary**

Chapter covered 3 common data models:

* Relational model (SQL)
  + Tables
  + Support one-to-many and simple many-to-many relationships
* Document model (NoSQL)
  + No schema enforcement at the db layer (schema-on-read)
  + Good support for one-to-many relationships
  + Locality advantage
  + Compatible with APIs and application code
* Graph model (NoSQL)
  + No schema enforcement at the db layer (schema-on-read)
  + Composed of vertices and edges
  + Model data with many complex relationships, where required joins are not known in advance

Each data model has it’s own query language or framework (SQL, MapReduce, MongoDB’s aggregation pipeline, Cypher, SPARQL, Datalog)

Some other interesting data models not covered:

“

* Researchers working with genome data often need to perform *sequence-similarity searches*, which means taking one very long string (representing a DNA molecule) and matching it against a large database of strings that are similar, but not identical. None of the databases described here can handle this kind of usage, which is why researchers have written specialized genome database software like GenBank [48].
* Particle physicists have been doing Big Data–style large-scale data analysis for decades, and projects like the Large Hadron Collider (LHC) now work with hundreds of petabytes! At such a scale custom solutions are required to stop the hardware cost from spiraling out of control [49].
* *Full-text search* is arguably a kind of data model that is frequently used alongside databases. Information retrieval is a large specialist subject that we won’t cover in great detail in this book…”