Chapter 2 Data Models and Query Languages

- Data models are the most important part of developing software, because they have a profound effect.
- 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?
- Model the real world in terms of objects or data structures, and APIs that manipulate those data structures.

Relational Model Versus Document Model

- SQL: data is organized into relations (**tables** in SQL), each relation is an unordered collection of tuples (**rows** in SQL).

- Used in Business data processing, transaction processing and batch processing

The Birth of NoSQL

- Scalability, large datasets or high write throughput
- For free and open source software over commercial database products
- Specialized query operations
- Less restrictiveness of relational schemas, and a more dynamic and expressive data model

The Object-Relational Mismatch

- SQL data model: if data is stored in relational tables, a translation layer is required between the objects in the application code and the database model of tables, rows, and columns.
- The disconnect between the models is sometimes called an **impedance mismatch**.
- **Object-relational mapping (ORM) frameworks** like **ActiveRecord** and **Hibernate** can reduce code for transactions but they can't hide the differences between the two models.
- SQL model normalized representation is to put objects in separate tables, with a foreign key reference to the fact table.
- For a data structure like a résumé, which is mostly a self-contained document, a JSON representation is better. JSON is much simpler than XML. Document-oriented databases like MongoDB, RethinkDB, CouchDB, and Espresso support this data model.
- The JSON representation has better locality than the multi-table schema in SQL.

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

- That incurs write overheads, and risks inconsistencies (where some copies of the information are updated but others aren't). Removing such duplication is the key idea behind **normalization in databases**.
- **Normalizing this data requires many-to-one relationships.**

Are Document Databases Repeating History?

The network model

- CODASYL model: The network model was standardized by a committee, the Conference on Data Systems Languages (CODASYL) and implemented by several database vendors.
- The CODASYL model was a generalization of the hierarchical model. In the tree structure of the hierarchical model, every record has exactly one parent; in the network model, a record could have multiple parents.
- E.g, one record for the "Greater Seattle Area" region, every user who lived in that region is linked to it. This allowed many-to-one and many-to-many relationships to be modeled.

- The links between records in the network model were not foreign keys, but more like pointers in a programming language.
- **Access path: The way of accessing a record was to follow a path from a root record along these chains of links.**
- A query in CODASYL was performed by moving a cursor through the database by iterating over lists of records and following access paths. If a record had multiple parents (i.e., multiple incoming pointers from other records), the application code had to keep track of all the various relationships. This is like navigating an *n*-dimensional data space.
- The problem is the code for querying and updating the database is complicated and inflexible. With both the hierarchical and the network model, if you didn't have a path to the data you wanted, you were in a difficult situation. You can change the access paths, but you have to go through handwritten database query code and rewrite it to handle the new access paths. It is difficult to make changes to an application's data model.

The relational model

- In a relational database, the query optimizer automatically decides which parts of the query to execute in which order, and which indexes to use.
- They are made automatically by the query optimizer, not by the application developers.

Comparison to document databases

- Document databases reverted back to the hierarchical model in one aspect: storing nested records (one-to-many relationships, like positions, education, and contact_info) within their parent record rather than in a separate table.
- When representing many-to-one and many-to-many relationships, relational and document databases are not fundamentally different: the related item is referenced by a **unique identifier**, **foreign key** in the relational model and a **document reference** in the document model. That identifier is resolved at read time by using a join or follow-up queries.

Relational Versus Document Databases Today

- Document data model: schema flexibility, better performance due to locality, and for some applications it is closer to application data structures.
- Relational model: better support for joins, and many-to-one and many-to-many relationships.

Which data model leads to simpler application code?

- It depends on the kinds of relationships that exist between data items. For highly interconnected data, the relational model is acceptable, and graph models are the best.
- If the application uses many-to-many relationships, use the relational model.

Schema flexibility in the document model

- It is not schemaless, there is an implicit schema, but not enforced by the database. More accurate terms:
- **Schema-on-read:** the structure of the data is implicit, and only interpreted when the data is read. Similar to dynamic (runtime) type checking in programming languages. Changing schemas is easy.
- **Schema-on-write**: the traditional approach of relational databases, where the schema is explicit and the database ensures all written data conforms to it. Similar to static (compile-time) type checking. Changing schemas is expensive.

Data locality for queries

- A document is stored as a single continuous string, encoded as JSON, XML, or a binary variant(MongoDB's BSON). If your application needs to access the entire document (render it on a web page), there is a performance advantage to this **storage locality**.
- The locality advantage only applies if you need large parts of the document at the same time. The database needs to load the entire document, even if you access only a small portion of it.
- On **updates** to a document, the entire document usually needs to be rewritten, modifications that don't change the encoded size of a document can be performed in place.
- Better to keep documents small and avoid writes that increase the size of a document.
- The idea of grouping related data together for locality is not limited to the document model. Relational databases like Oracle also have multi-table index cluster tables. The column-family concept in the Bigtable data model (Cassandra and HBase) has a similar purpose of managing locality.

Convergence of document and relational databases

- Most relational database systems (other than MySQL) have supported XML.
- This includes functions to make local modifications to XML documents and the ability to index and query inside XML documents, which allows applications to use data models very similar to a document database.
- PostgresSQL also have a similar level of support for JSON documents.
- Some document databases supports relational-like joins in its query language, or effectively client-side join.

Query Languages for Data

Declarative Queries on the Web

In a web browser, using declarative CSS styling is much better than manipulating styles imperatively in JavaScript. Similarly, in databases, declarative query languages like SQL turned out to be much better than imperative query APIs.

MapReduce Querying

- MapReduce is a programming model for processing large amounts of data in bulk across many machines. A limited form of MapReduce is supported by some NoSQL datastores(MongoDB), as a mechanism for performing read-only queries across many documents.
- MapReduce is neither a declarative query language nor an imperative query API, but somewhere in between: the logic of the query is expressed with code snippets, and are called repeatedly by the processing framework. It is based on the map (collect) and reduce (fold or inject) functions in many functional programming languages.
- In PostgreSQL you might express that query like this:

```
SELECT date_trunc('month', observation_timestamp) AS observation_month,

sum(num_animals) AS total_animals

FROM observations

WHERE family = 'Sharks'

GROUP BY observation_month;
```

- MongoDB's MapReduce

```
db.observations.mapReduce(

function map() {

var year = this.observationTimestamp.getFullYear();

var month = this.observationTimestamp.getMonth() + 1;

emit(year + "-" + month, this.numAnimals);

},

function reduce(key, values) {
```

```
    return Array.sum(values);

    },

    {

    query: { family: "Sharks" },

    out: "monthlySharkReport"

    }

    );
```

- MapReduce is a **low-level programming model** for **distributed execution** on a cluster of machines. **High-level query languages like SQL** can be implemented as a **pipeline of MapReduce operations**.

Graph-Like Data Models

A graph consists of two kinds of objects: vertices (nodes or entities) and edges (relationships or arcs). Many kinds of data can be modeled as a graph. 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.