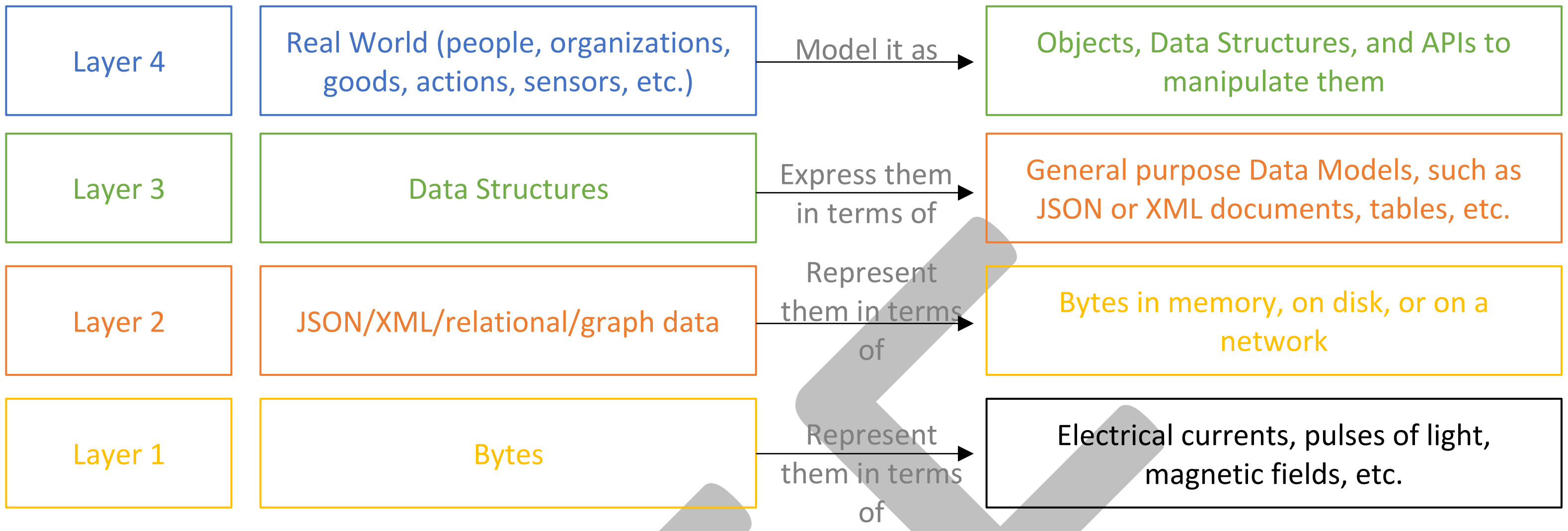
Chapter II – Data Models and Query Languages

Table of Contents

Data models are the **most important part** of **developing software** because **dictate** how the **software is written** and how we **think about the problem** we are solving. It is very important to choose the right **data model** because it has such a profound **effect** on **what the software above it can and can't do.**

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?*


Layer 4
Real World (people, organizations, goods, actions, sensors, etc.)
Model it as
Objects, Data Structures, and APIs to manipulate them
Layer 3
Data Structures
Express them in terms of
General purpose Data Models, such as JSON or XML documents, tables, etc.
Layer 2
JSON/XML/relational/graph data
Represent them in terms of
Bytes in memory, on disk, or on a network
Layer 1
Bytes
Represent them in terms of
Electrical currents, pulses of light, magnetic fields, etc.

The basic idea is: **Each layer hides the complexity of the layers below it by providing a clean data model.** These abstractions allow groups of people to work together effectively.

Relational Model Versus Document Model

The **best-known data model** today 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 (rows in SQL)**.

In the mid-1980s, relational database management systems (RDBMSes) and SQL became the tools of choice for most people who needed to store and query data. Its dominance has lasted around 25-30 years.

The **roots** of relational databases lie **in business data processing**. The **goal** of the relational model is to **hide the implementation details behind a clear interface**.

In the 1970s and early 1980s, the **network model** and **hierarchical model** were the **main alternatives**. In the late 1980s and early 1990s, **object databases** were "in". In the early 2000s, **XML databases appeared**. **Relational Model always dominated them**.

The Birth of NoSQL

In the 2010s, **NoSQL** is the latest attempt to overthrow the relational model's dominance.

Fun Fact	**NoSQL does not refer to any particular technology.**

NoSQL was originally intended simply as a catchy Twitter hashtag for a meetup on open source, distributed, non-relational databases in 2009. Since then, many interesting database systems are now associated with the #NoSQL, and it has been retroactively reinterpreted as Not only SQL.

The driving forces behind the adoption of NoSQL databases are:

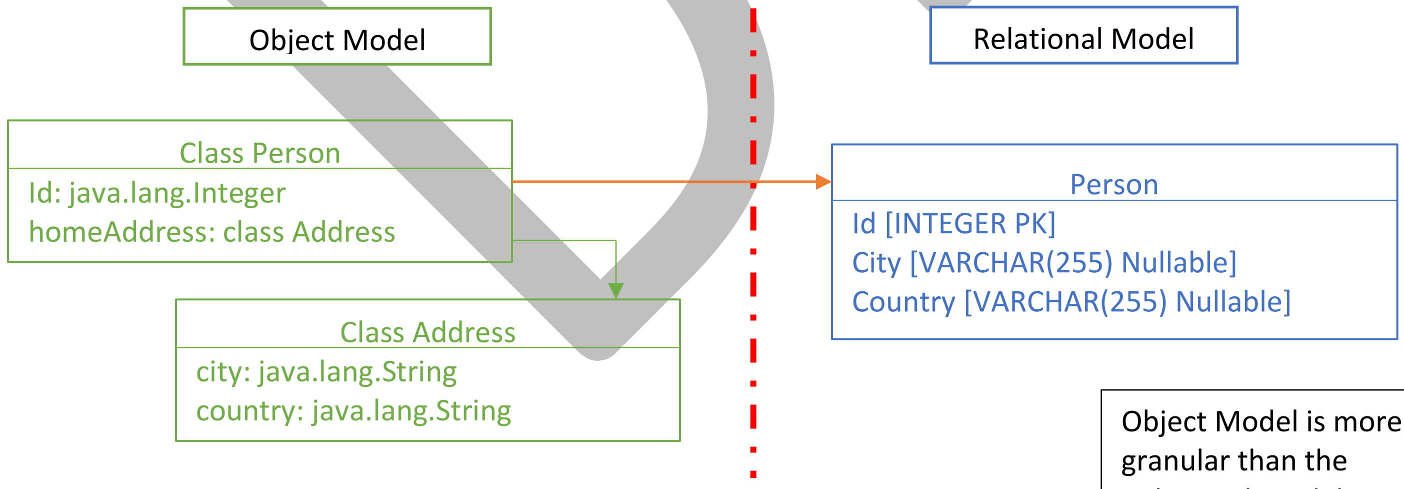
- A need for greater scalability
- Preference for free and open-source software
- Specialized query operations that are not well supported by the relational model
- Frustration with the restrictiveness of relational schemas
- A desire for a more dynamic and expressive data model

It seems like relational databases will continue to be used alongside a broad variety of non-relational datastores – an idea that is sometimes called polyglot persistence.

The Object-Relational Mismatch

Most application development today is done in object-oriented programming language, which leads to a common criticism of the SQL data model: if data is stored in relational tables, an awkward 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.

In other words, the impedance mismatch is a term used when problems occur due to differences between the database model and the programming language model. For example, in our code, we have two classes: Person class and Address class, but we only have one table in the database to store data for it.



Object Model is more granular than the Relational Model

Object-relational mapping (ORM) frameworks like ActiveRecord and Hibernate **reduce** the amount of **code required** for this **translation layer**, but they can't completely hide the differences between two models.

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

The advantages of having standardized lists versus having free-text fields are:

- **Consistent style** and **spelling** across values
- **Ease of updating** – if the value is stored in only one place, it is easy to update across the board if it ever needs to be changed
- **Localization support** – it is easy to translate into other language
- **Better search**

Whether you store an **ID, or a text string** is a question of **duplication**. When you use an **ID**, the information that is meaningful to human is **stored in only one place**, and everything that refers to it uses an ID (which only has meaning within the database). When you store the **text** directly, you are **duplicating** the human-meaningful information in every record that uses it.

The advantage of using an **ID** is that because it has no meaning to humans, it **never needs to change**: the ID can remain the same, even if the information it identifies change. **Removing duplication is the key idea behind normalization in databases**.

Are document databases repeating history?

While **many-to-many** relationships and joins are routinely used in **relational databases**, document databases and NoSQL reponed the debate on how best to represent such relationships in a database. This debate is old and continue being a topic of interest.

The **most popular** database for business data processing in the **1970s** was **IBMs Information Management System (IMS)**. The design of IMS used a fairly simple data model called the hierarchical model, which has some remarkable similarities to the JSON model used by document databases.

Like **document databases**, IMS **worked well for one-to-many relationships**, but it made **many-to-many relationships difficult**, and it did **not support joins**. Various **solutions** were proposed to solve the limitations of the hierarchical model. The two most prominent were the **relational model** (which became SQL and took over the world) and the **network model** (which initially had a large following but eventually faded into obscurity).

The Network Model

The network model was **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 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 models**.

The **link between records** in the network model were not foreign keys, but more like **pointers** in a programming language (while still being stored on disk). The only way of **accessing a record** was to **follow a path** from a root record along these chains of links. This was called an **access path.** In the world of many-to-many relationships, several different paths can lead to the same record.

A query in CODASYL was performed by moving a cursor through the database by iterating over lists of records and following access paths. This was like navigating around an n-dimensional data space. This **"access path"** made the **code** for querying and updating the database **complicated** and **inflexible**. With both, the network and the hierarchical model, if you did not have a path to the data you wanted, you were in a difficult situation.

The relational model

Relational model lay out all the **data in the open**: a relation (table) is simply a collection of tuples (rows), and that's it. No access paths, labyrinths to nested structures, no complicated tracking.

In a relational database, the **query optimizer** automatically **decides** which part of the **query to execute** in which order, and which indexes to use. Those choices are effectively the "access path", but the big difference is that they are made automatically by the query optimizer, not by the application developer.

Query optimizers are beasts, but the good news is that you only need to build a query optimizer once, and then all the applications that use the database can *benefit from it.*

Comparison to document databases

Document databases reverted back to the hierarchical model in one aspect: **storing nested records** (one-to-many relationships) **within their parent record** rather than in a separate table.

To **represent many-to-one or many-to-many relationships**, relational and document databases reference the related item by a **unique identifier** called **foreign key** in the **relational data model** and **document reference** in the **document model**. That identifier is resolved at read time by using a join or follow-up queries. To date, document databases have not followed the path of CODASYL.

Relational Versus Document Databases Today

Document Data Model	Relational Model
1. Schema Flexibility **2. Better performance due to locality** **3. Closer to the data structures used by the application**	1. Better support for joins 2. Better support for many-to-one and many-to-many relationships

Which data model leads to simpler application code?

It is not possible to say in general which data model leads to simpler application code; it depends on the kinds of relationships that exist between the data items. For highly interconnected data, the document model is awkward, the relational model is acceptable, and graph models are the most natural.

Schema flexibility in the document model

Document databases are sometimes called **schemaless**, but **that's misleading**, as the code that reads the data usually assumes some kind of structure – i.e., there is an implicit schema, but it is not enforced by the database. A more accurate term is **schema-on-read** (the **structure** of the data is implicit, and only **interpreted when the data is read**), in contrast with **schema-on-write** (the traditional approach of relational databases, where the **schema is explicit**, and the database ensures all written data conforms to it).

Schema-on-read is similar to **dynamic (runtime)** type checking in programming language, whereas **schema-on-write** is similar to **static (compile-time)** type checking.

The **Schema-on-read** is **advantageous** if the **items** in the collection **don't all have the same structure** for some reason (i.e., the data is heterogenous). For example, there are many different types of objects, or the structure of the data is determined by external systems. In situations like these, a schema may hurt more than it helps. When all the **records** are expected to have the **same structure**, **schemas are a useful** mechanism for documenting and enforcing that structure.

Data Locality for queries

A **document** is usually stored as a single **continuous string**, encoded as JSON, XML, or a binary variant thereof (such as Mongo DB's BSON). If your **application** often **needs to access the entire document** (for example, to render it on a web page), there is a **performance advantage** to this storage locality. If data is split across multiple tables, multiple index lookups are required to retrieve it all, which may require more disks seeks and take more time. The locality advantage only applies if you need large parts of the document at the same time.

Convergence of document and relational databases

Most relational database systems (other than MYSQL) have supported XML since the mid-2000s. 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

SQL is a declarative query language, whereas **IMS and CODASYL** queried the database using **imperative code**. An imperative language tells the computer to perform certain operations in a certain order. In a declarative query language, like SQL, you just specify the pattern of the data you want but not how to achieve that goal. It is up to the database system's query optimizer to decide which indexes, and which join methods to use, and in which order to execute various parts of the query.

Declarative Queries on the Web

- **Declarative** query language:
    - It is **more concise and easier to work** with than an imperative API.
    - It **hides implementation** details of the database engine
    - It has more run for automatic **optimizations**
    - It has a better chance of getting **faster in parallel execution** because they specify only pattern of the results, not the algorithm that is used to determine the results

In a web browser, using declarative CSS styling is so 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 API's (COBOL to iterate over records in the database, one record at a time).

MapReduce Querying

MapReduce is a programming model for processing large amounts of data in bulk across many machines, popularized by Google. MapReduce is neither a declarative query language nor a fully imperative query API, but somewhere in between: **the logic of the query is expressed with snippets of code which are called repeatedly by the processing framework**. It is based on the map (also known as collect) and reduce (also known as fold or inject) functions that exist in many functional programming languages.

Map and reduce must be pure functions, which means they only use the data that is passed to them as input, they cannot perform additional database queries, and they must not have side effects. These restrictions allow the database to run the functions anywhere, in any order, and rerun them on failure.

A usability problem with mapReduce is that you have to write two carefully coordinated JavaScript functions, which is harder than writing a single query.

```
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) {
        reduce Array.sum(values);
    }
    {
        query: {family: “Sharks”},
        out: “monthlySharkReport”
    }
```

Graph-Like Data Models

A graph consists of two kinds of objects: **vertices** (also known as node or entities) and **edges** (also known as relationships or arcs).

Graphs are good for evolvability: as you add features to your application, a graph can easily be extended to accommodate changes in your application’s data structures.

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:

1. Any vertex can have an edge connecting it with any other vertex. There is no schema that restricts which kinds of things can or cannot be associated.
2. Given any vertex, you can efficiently find both its incoming and its outgoing edges, and thus traverse the graph – i.e., follow a path through a chain of vertices – both forward and backward.
3. By using different labels for different kinds of relationships, you can store several different kinds of information in a single graph, while still maintaining a clean data model.

The **cypher query language** is a declarative language for property graphs, created for the Meo4j graph database.

If you put graph data in a relational structure, you can query it using SQL, but it has some difficulties. In a relational database, you usually know in advance which joins you need in your query. In a graph query, you may need to transverse a variable number of edges before you find the vertex you are looking for – that us, the number of joins is not fixed in advance.

Triple-Stores and SPARQL

In a **triple-store**, all information is stored in the form of very simple three-part statements: **(subject, predicate, object).** The **subject** of a triple is equivalent to a **vertex** in a graph. The **object** is one of two things:

1. **A value in a primitive datatype**, such as string or a number. In that case, the predicate and object of the triple are equivalent to the key and value of a property on the subject vertex.
2. **Another vertex in the graph**. In that case, the predicate is an edge in the graph, the subject is the tail vertex, and the object is the head vertex.

```
-- Subset of data represented as Turtle triplets
_:idaho :within _:usa -- the predicate represents an edge, the object
is a vertex
_:usa :name "United States" -- the predicate is a property, the
object is a string literal
```

The sematic web

The triple-stores data model is completely independent of the semantic web. The semantic web idea is: websites already publish information as texts and pictures for humans to read, so why don't they also publish information as machine readable data for computers to read? The Resource Description framework (RDF) is a mechanism for different websites to publish data in a consistent format, allowing data from different websites to automatically combined into a web of data – a kind of internet-wide "database of everything".

The RDF (Resource Description Model) data Model

RDF has few peculiarities due to the fact that it is designed for internet-wide data. **The subject, predicate, and object of a triple are often URIs.** For example, <http://my-company.com/namespace#within>. The reason behind this design is that you should be able to combine your data with someone else's data, and if they attach a different meaning to the word within, you won't get a conflict because their predicates are actually <http://other.org/foo#within>.

The SPARQL query language

SPARQL is a query language for triple-stores using the RDF data model. SPARQL is a nice query language – even if the semantic web never happens, it can be a powerful tool for applications to use internally.

```
SELECT ?personName WHERE {
    ?person :name ?personName.
    ?person :bornIn / :within* / name "United States".
    ?person :livesIn / :within* / name "Europe".
}
```

Graph Databases compared to the network model

CODASYL	Graph Database
- **It has a schema that specifies which record type can be nested within which other record type**	- It does not have a schema: any vertex can have an edge to any other vertex -> This gives much greater flexibility to adapt to changing requirements
- **To reach a particular record, you need to traverse one of the access paths to it**	- To reach a particular record, you can refer directly to any vertex by its unique ID, or you can use an index to find vertices with a particular value
- **Children of a record are an ordered set. Database must maintain that**	- Vertices and edges are not ordered

order and applications must worry about that order while inserting new records	
- **All queries are imperative, difficult to write and easily broken by changes in the schema**	- It supports imperative code but also high-level declarative languages

The Foundation: Datalog

Datalog is a declarative logic programming language that syntactically is a subset of Prolog. It is often used as a query language for deductive databases. Datalog's data model is similar to the triple-store model, generalized a bit. Instead of writing a triple as (subject, predicate, object), we write it as predicate (subject, object).

```
Name(namerica, ‘North America’).
Type(namerica, continent).

Within_recursive(Location, Name) :- name(Location, Name). /* Rule 1 */

Migrated(Name, BornIn, LivingIn) :- name(Person, Name),   /* Rule 2 */
                                    Born_in(Person, BornLoc),
                                    Within_recursive(BornLoc, BornIn).

?- migrated(Name, ‘United States’, ‘Europe’)              /* Rule 3 */
```

The Datalog approach requires a different kind of thinking to the other query languages. It is a very powerful approach, because rules can be and reused in different queries. It's less convenient for simple one-off queries, but it can cope better if your data is complex.