Data Models & Query Languages (chapter 2 from Martin Kleppmann’s Book)

There are many different kinds of data models, and **every data model embodies**

**assumptions about how it is going to be used.** Some kinds of usage are easy and some

are not supported; some operations are fast and some perform badly; some data

transformations feel natural and some are awkward.

since the data model has such a profound effect on what the software above it can

and can’t do, it’s important to choose one that is appropriate to the application.

# Relational Model vs Document Model

## intro

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).**

Some other databases came to overthrow relational databases like, Object and XML databases but they failed to do so.

In the early 2010’s NoSQL (Not only SQL) databases came across and the following are some of the driving forces for people to adopt them:

* 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

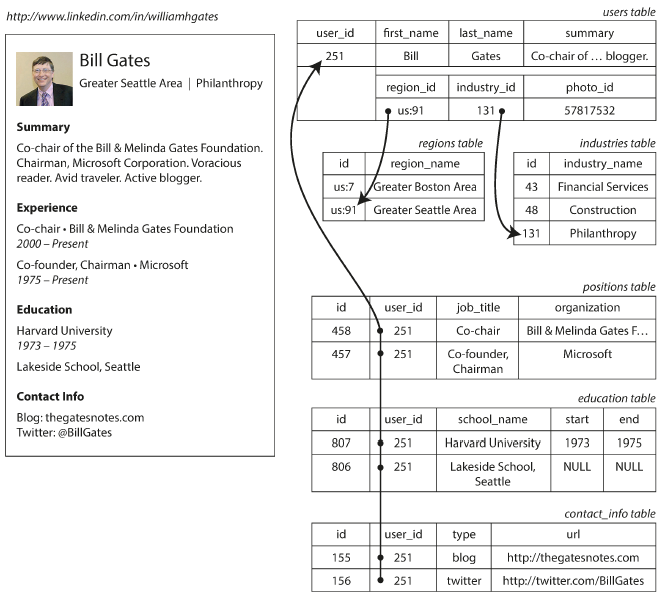
It seems likely that in the foreseeable future, relational databases will continue to be used alongside a broad variety of nonrelational datastores—an idea that is sometimes called ***polyglot persistence****.*

## The Object/Relational Mismatch

Object-relational mapping (ORM) frameworks like ActiveRecord and Hibernate

reduce the amount of boilerplate code required for the translation layer between application code and database rows and tables, but they can’t completely hide the differences between the two models.

For example, let’s consider a profile in Linked-in and how it can be represented in a relational model:



The profile as a whole can be identified by a unique identifier, user\_id. Fields like first\_name and last\_name appear exactly once per user, so they can be modeled as columns on the users table. However, most people have had more than one job in their career (positions), and people may have varying numbers of periods of education and any number of pieces of contact information. **There is a one-to-many relationship from the user to these items, which can be represented in various ways:**

* In the traditional SQL model (prior to SQL:1999), the most common normalized

representation is to put positions, education, and contact information in separate

tables, with a foreign key reference to the users table, as in figure above.

* Later versions of the SQL standard added support for **structured datatypes** and

**XML data**; this allowed **multi-valued** data to be stored within a single row, **with** support for **querying** and **indexing** inside those documents. These features are supported to varying degrees by Oracle, IBM DB2, MS SQL Server, and PostgreSQL. A **JSON** datatype is also supported by several databases, including IBM DB2, MySQL, and PostgreSQL.

* A third option is to **encode** jobs, education, and contact info **as a** **JSON or XML**

**document**, store it **on a text column in the database**, and let the application interpret its structure and content. In this setup, you typically cannot use the database to query for values inside that encoded column.

For a data structure like **a resume**, which is **mostly a self-contained *document***, a JSON

representation can be quite appropriate: see Example 2-1. **JSON has the appeal of being much simpler than XML**. Document-oriented databases like MongoDB, RethinkDB, CouchDB, and Espresso support this data model. Example 2-1:

{

**"user\_id"**: 251,

**"first\_name"**: "Bill",

**"last\_name"**: "Gates",

**"summary"**: "Co-chair of the Bill & Melinda Gates... Active blogger.",

**"region\_id"**: "us:91",

**"industry\_id"**: 131,

**"photo\_url"**: "/p/7/000/253/05b/308dd6e.jpg",

**"positions"**: [

{**"job\_title"**: "Co-chair", **"organization"**: "Bill & Melinda Gates Foundation"},

{**"job\_title"**: "Co-founder, Chairman", **"organization"**: "Microsoft"}

],

**"education"**: [

{**"school\_name"**: "Harvard University", **"start"**: 1973, **"end"**: 1975},

{**"school\_name"**: "Lakeside School, Seattle", **"start"**: **null**, **"end"**: **null**}

],

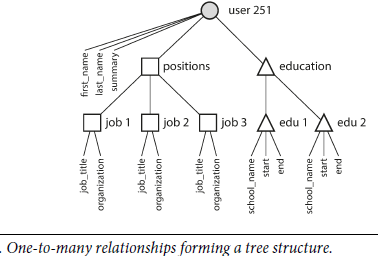
**"contact\_info"**: {

**"blog"**: "http://thegatesnotes.com",

**"twitter"**: "http://twitter.com/BillGates"

}

* Some developers feel that the JSON model reduces the impedance mismatch between the application code and the storage layer. However, as we shall see in Chapter 4, there are also problems with JSON as a data encoding format. The lack of a schema is often cited as an advantage; we will discuss this in “Schema flexibility in the document model”.
* The JSON representation has better ***locality***than the multi-table schema in. If you want to fetch a profile in the relational example, you need to either perform multiple queries (query each table by user\_id) or perform a messy multiway join between the users table and its subordinate tables. In the JSON representation, all the relevant information is in one place, and one query is sufficient.
* The one-to-many relationships from the user profile to the user’s positions, educational history, and contact information imply a tree structure in the data, and the JSON representation makes this tree structure explicit:



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

In the relational model showed in the first picture in the preceding section region\_id and industry\_id are given as IDs, not as plain-text strings "Greater Seattle Area" and "Philanthropy". Why?

If the user interface has free-text fields for entering the region and the industry, it makes sense to store them as plain-text strings. But there are advantages to having standardized lists of geographic regions and industries, and letting users choose from a drop-down list or auto-completer:

• Consistent style and spelling across profiles

• Avoiding ambiguity (e.g., if there are several cities with the same name)

• Ease of updating—the name is stored in only one place, so it is easy to update across the board if it ever needs to be changed (e.g., change of a city name due to political events)

• Localization support—when the site is translated into other languages, the standardized

lists can be localized, so the region and industry can be displayed in the viewer’s language

• Better search—e.g., a search for philanthropists in the state of Washington can match this profile, because the list of regions can encode the fact that Seattle is in Washington (which is not apparent from the string "Greater Seattle Area")

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 humans (such as the word *Philanthropy*)

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

changes. Anything that is meaningful to humans may need to change sometime in

the future—and if that information is duplicated, all the redundant copies need to be

updated. 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.

Unfortunately, normalizing this data requires *many-to-one* relationships (many people

live in one particular region, many people work in one particular industry), which

don’t fit nicely into the document model. In relational databases, it’s normal to refer

to rows in other tables by ID, because joins are easy. In document databases, joins are

not needed for one-to-many tree structures, and support for joins is often weak.

If the database itself does not support joins, you have to emulate a join in application

code by making multiple queries to the database. (In this case, the lists of regions and

industries are probably small and slow-changing enough that the application can

simply keep them in memory. But nevertheless, the work of making the join is shifted

from the database to the application code.)

Moreover, 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. For example, consider some changes we could make to the

resume example:

**Organizations and schools as entities:**

In the previous description, organization (the company where the user worked)

and school\_name (where they studied) are just strings. Perhaps they should be

references to entities instead? Then each organization, school, or university could

have its own web page (with logo, news feed, etc.); each resume could link to the

organizations and schools that it mentions, and include their logos and other

information (see Figure 2-3 for an example from LinkedIn).

**Recommendations:**

Say you want to add a new feature: one user can write a recommendation for

another user. The recommendation is shown on the resume of the user who was

recommended, together with the name and photo of the user making the recommendation.

If the recommender updates their photo, any recommendations they

have written need to reflect the new photo. Therefore, the recommendation

should have a reference to the author’s profile.

**these new features require many-to-many relationships.**

### Are Document Databases Repeating History?

While many-to-many relationships and joins are routinely used in relational databases,

document databases and NoSQL reopened the debate on how best to represent

such relationships in a database. This debate is much older than NoSQL—in fact, it

goes back to the very earliest computerized database systems.

The most popular database for business data processing in the 1970s was IBM’s *Information Management System* (IMS), originally developed for stock-keeping in the

Apollo space program and first commercially released in 1968 It is still in use

and maintained today, running on OS/390 on IBM mainframes.

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 It represented all **data as a tree of records** **nested within records**, much **like the JSON structure.**

Like document databases, IMS **worked well for one-to-many relationships**, **but it made many-to-many relationships difficult**, and it didn’t support joins. Developers had to decide **whether to duplicate (de-normalize)** data **or to manually resolve references** from one record to another. These problems of the 1960s and ’70s were very much like the problems that developers are running into with document databases today.

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 “great debate” between these two camps lasted

for much of the 1970s.

* **The Network Model ( 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 parents**. For example, there could be one

record for the "Greater Seattle Area" region, and every user who lived in that region could be 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 (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 simplest case, an access path could be like the traversal of a linked list: start at

the head of the list, and look at one record at a time until you find the one you want.

But in a world of many-to-many relationships, several different paths can lead to the

same record, and a programmer working with the network model had to keep track of these different access paths in their head.

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. Even CODASYL committee members

admitted that this was like navigating around an *n*-dimensional data space.

Although manual access path selection was able to make the most efficient use of the

very limited hardware capabilities in the 1970s (such as tape drives, whose seeks are

extremely slow), the problem was that they made the code for querying and updating

the database 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 could change the access paths, but then you had to go through a lot of

handwritten database query code and rewrite it to handle the new access paths. It was

difficult to make changes to an application’s data model.

* **The Relational Model**

What the relational model did, by contrast, was to **lay out all the data in the open**: a

relation (table) is simply a collection of tuples (rows), and that’s it. There are no labyrinthine nested structures, no complicated access paths to follow if you want to look

at the data. You can read any or all of the rows in a table, selecting those that match

an arbitrary condition. You can read a particular row by designating some columns

as a key and matching on those. You can insert a new row into any table without

worrying about foreign key relationships to and from other tables.

In a relational database, the query optimizer automatically decides which parts 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,** so we rarely need to think about them. If you want to query your data in new ways, you can just declare a new index, and

queries will automatically use whichever indexes are most appropriate. **You don’t**

**need to change your queries to take advantage of a new index.** The relational model thus made it much easier to add new features to applications.

A key insight of the relational model was this: you only need to build a query optimizer once, and then all applications that use the database can benefit from it. If you don’t have a query optimizer, it’s easier to hand-code the access paths for a particular query than to write a general-purpose optimizer—but the general-purpose solution wins in the long run.

* **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. However, when it comes to representing **many-to-one and many-to-many relationships, relational and document databases are not fundamentally different**: in both cases, the related item is referenced by a unique identifier, which is called a ***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**. To date, document databases have not followed the path of CODASYL.

# Relational vs Document Databases Today

There are many differences to consider when comparing relational databases to

document databases, including their **fault-tolerance properties** (see Chapter 5) and

**handling of concurrency** (see Chapter 7). In this chapter, we will concentrate only on

the differences in **the data model.**

The main arguments in favor of the **document** **data model** are:

* **schema flexibility**
* **better performance due to locality**
* **for some applications it is closer to the data structures used by the application.**

The **relational model** counters by:

* **providing better support for joins, and many-to-one and many-to-many relationships.**

## Which Data Model Leads to Simpler Application Code?

* If the data in your application has a document-like structure (i.e., **a tree of one-to-many relationships, where typically the entire tree is loaded at once**), then it’s probably a good idea to use a document model. The relational technique of *shredding— splitting* a document-like structure into multiple tables (like positions, education, and contact\_info)—can lead to cumbersome schemas and unnecessarily complicated application code.
* The document model has limitations: for example, **you cannot refer directly to a nested item within a document,** but instead you need to say something like “the second item in the list of positions for user 251” (much like an access path in the hierarchical model). **However, as long as documents are not too deeply nested, that is not usually a problem.**
* The poor support for joins in document databases may or may not be a problem, depending on the application. For example, many-to-many relationships may never be needed in an analytics application that uses a document database to record which events occurred at which time.
* However, if your application does use many-to-many relationships, the document model becomes less appealing. It’s possible to reduce the need for joins by denormalizing, but then the application code needs to do additional work to keep the denormalized data consistent. Joins can be emulated in application code by making multiple requests to the database, but that also moves complexity into the application and is usually slower than a join performed by specialized code inside the database. In such cases, using a document model can lead to significantly more complex application code and worse performance.

It’s not possible to say in general 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 **document model is awkward**, the **relational model is acceptable**, and **graph models** (see “Graph-Like Data Models”) **are the most** **natural**.

## Schema Flexibility in the Document Model

* Most document databases, and the JSON support in relational databases, do not enforce any schema on the data in documents. XML support in relational databases 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.
* 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 languages, whereas schema-on-write is similar to static (compile-time) type checking. Just as the advocates of static and dynamic type checking have big debates about their relative merits, enforcement of schemas in database is a contentious topic, and in general there’s no right or wrong answer.

**The difference between the approaches is particularly noticeable in situations where an application wants to change the format of its data**. For example, say you are currently storing each user’s full name in one field, and you instead want to store the first name and last name separately [23]. In a document database, you would just start writing new documents with the new fields and have code in the application that handles the case when old documents are read. For example:

**if** (user && user.name && !user.first\_name) {

*// Documents written before Dec 8, 2013 don't have first\_name*

user.first\_name = user.name.split(" ")[0];

}

On the other hand, in a “statically typed” database schema, you would typically perform

a ***migration***along the lines of:

**ALTER TABLE** users **ADD COLUMN** first\_name text;

**UPDATE** users **SET** first\_name = split\_part(name, ' ', 1); *-- PostgreSQL*

**UPDATE** users **SET** first\_name = substring\_index(name, ' ', 1); *-- MySQL*

Schema changes have a bad reputation of being slow and requiring downtime. This

reputation is not entirely deserved: most relational database systems execute the

ALTER TABLE statement in a few milliseconds. MySQL is a notable exception—it copies the entire table on ALTER TABLE, which can mean minutes or even hours of downtime when altering a large table—although various tools exist to work around this limitation.

Running the UPDATE statement on a large table is likely to be slow on any database,

since every row needs to be rewritten. If that is not acceptable, the application can

leave first\_name set to its default of NULL and fill it in at read time, like it would with

a document database.

**The schema-on-read approach is advantageous if the items in the collection don’t all have the same structure** for some reason (i.e., the data is heterogeneous)—for example,because:

* There are many different types of objects, and it is not practical to put each type of object in its own table.
* The structure of the data is determined by external systems over which you have no control and which may change at any time.

In situations like these, a schema may hurt more than it helps, and schemaless documents

can be a much more natural data model. But in cases where all records are expected to have the same structure, schemas are a useful mechanism for documenting and enforcing that structure. We will discuss schemas and schema evolution in more detail in Chapter 4.

## 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 MongoDB’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 disk seeks and take more time.
* The locality advantage only applies if you need large parts of the document at the same time. The database **typically**(**Not for Mongo DB**) needs to load the entire document, even if you access only a small portion of it, which can be wasteful on large documents. On updates to a document, the entire document usually needs to be **rewritten**(**Check for Mongo but I guess he’s right but there is a small padding inside which changes won’t impose a rewrite**) —only modifications that don’t change the encoded size of a document can easily be performed in place. For these reasons, it is generally recommended that you keep documents fairly small and avoid writes that increase the size of a document. These performance limitations significantly reduce the set of situations in which document databases are useful.
* 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(nosql) (used in Cassandra and HBase) has a similar purpose of managing locality [29].

We will also see more on locality in Chapter 3.

## Convergence of Document and Relational Databases

* Most relational database systems (other than MySQL) have supported XML since the

mid-2000s. 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 what they would do when using a document database.

PostgreSQL since version 9.3, MySQL since version 5.7, and IBM DB2 since version 10.5 also have a similar level of support for JSON documents. Given the popularity of JSON for web APIs, it is likely that other relational databases will follow in their footsteps and add JSON support.

* On the document database side, RethinkDB supports relational-like joins in its query

language, and some MongoDB drivers automatically resolve database references (effectively performing a client-side join(But I think with mongo DB lookup it’s handled server-side), although this is likely to be slower than a join performed in the database since it requires additional network round-trips and is less optimized).

* 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

**…………………………………………………………………………….**

# Graph-Like Data Models

To Be Read About Later…………………………………………….