Harrison, G. (2015). Next Generation Databases: NoSQLand Big Data. Apress.

CHAPTER 4

Document Databases

A relational database is like a garage that forces you to take your car apart and store the pieces in little drawers

—Object-oriented data community, mid-1990s

An Object database is like a closet which requires that you hang up your suit with tie, underwear, belt, socks and shoes all attached.

—David Ensor, same period

A document database is a nonrelational database that stores data as structured documents, usually in XML or JSON formats. The "document database" definition doesn't imply anything specific beyond the document storage model: document databases are free to implement ACID transactions or other characteristics of a traditional RDBMS, though the dominant document databases provide relatively modest transactional support.

JSON-based document databases flourished after the nonrelational breakout of 2008, for three main reasons. First, they address the conflict between object-oriented programming and the relational database model that had frustrated software developers, and that motivated much of the object-oriented database movement of the mid-1990s. Second, because the self-describing document formats could be interrogated independently of the program that had created them, they supported ad hoc query access to the database that was absent in pure key-value stores. Third, they aligned well with the dominant web-based programming paradigms, particularly the AJAX programming model.

A document database, by allowing some form of data description without enforcing a schema, perhaps provides a happy medium between the rigid schema of the relational database and the completely schema-less key-value stores. Programmers remain free to change the data model as requirements shift within an application, but data consumers are still able to interrogate the data to determine its meaning.

The alignment with web-development programming practices has resulted in JSON document databases—and the MongoDB database in particular—becoming the default choice for many web developers.

Note Describing something as a document database only tells us that it stores data in XML or JSON format. The term does not define any specific transaction or clustering model.

XML and XML Databases

The first document databases were built around the XML document standard. XML databases are interesting to us primarily as the architectural precursors to the modern JSON document database; XML databases today represent a significant but small niche in the overall database market.

XML (*eXtensible Markup Language*) arose as a result of the convergence of efforts to develop a generalized markup language as the successor to various specialized formats such as SGML and a realization that HTML—the foundation of Web 1.0—uncomfortably combined layout and data. XML was capable of representing almost any form of information and, together with *Cascading Style Sheets* (*CSS*) that controlled rendering, allowed second-generation websites to separate data and format.

XML was widely used beyond these Web 2.0 use cases and became a standard format for many document types, eventually including word processing documents and spreadsheets. XML is also the basis for many data interchange protocols and, in particular, was a foundation for web service specifications such as *SOAP* (*Simple Object Access Protocol*).

During the early 2000s, it was widely expected that most documents in an organization outside of a relational database would end up represented as XML. And while the momentum behind XML has slowed, today a huge variety of document types use XML under the hood.

XML Tools and Standards

XML is supported by a rich ecosystem that includes a variety of standards and tools to assist with authoring, validation, searching, and transforming XML documents. These include:

- **XPath**: A syntax for retrieving specific elements from an XML document. XPath provides a simple and convenient way to filter documents using wildcards and tag references.
- **XQuery**: A query language for interrogating XML documents. The related **XQuery Update** specification provides mechanisms for modifying a document. XQuery is sometimes referred to as "the SQL of XML."
- **XML schema**: A special type of XML document that describes the elements that may be present in a specified class of XML documents. XML schema can be used to validate that a document is in the correct format, or to assist programs that wish to interrogate XML documents that conform to that format.
- **XSLT** (**Extensible Stylesheet Language Transformations**): A language for transforming XML documents into alternative formats, including non-XML formats such as HTML.
- **DOM** (**Document Object Model**): An object-oriented API that programs can use to interact with XML, XHTML, and similarly structured documents.

As the "SQL for XML," XQuery figures prominently in most XML database architectures. Figure 4-1 shows a simple XQuery statement that searches for documents with an ADDRESS element that includes the string "Berlin" within an embedded CITY element.

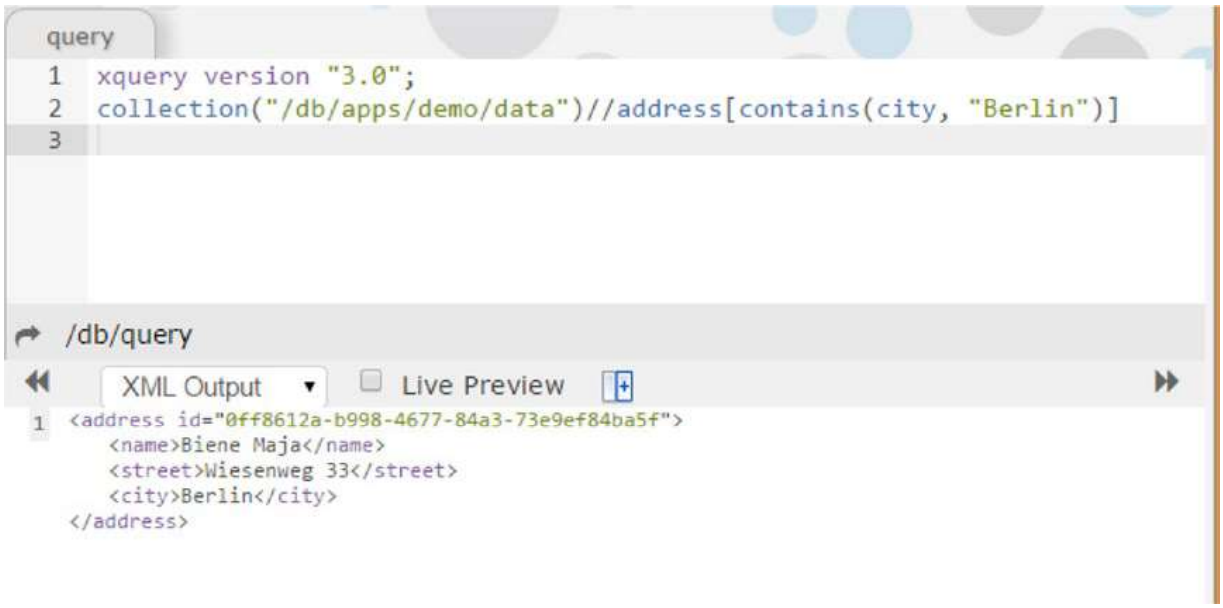

Figure 4-1. *Example of XQuery*

XML Databases

The increasing volume of XML documents within organizations provided a motivation for some form of XML document management system—or native *XML database*.

XML databases generally consist of a platform that implements the various XML standards such as XQuery and XSLT, and that provides services for the storage, indexing, security, and concurrent access of XML files. Figure 4-2 illustrates a simplified generic XML database architecture.

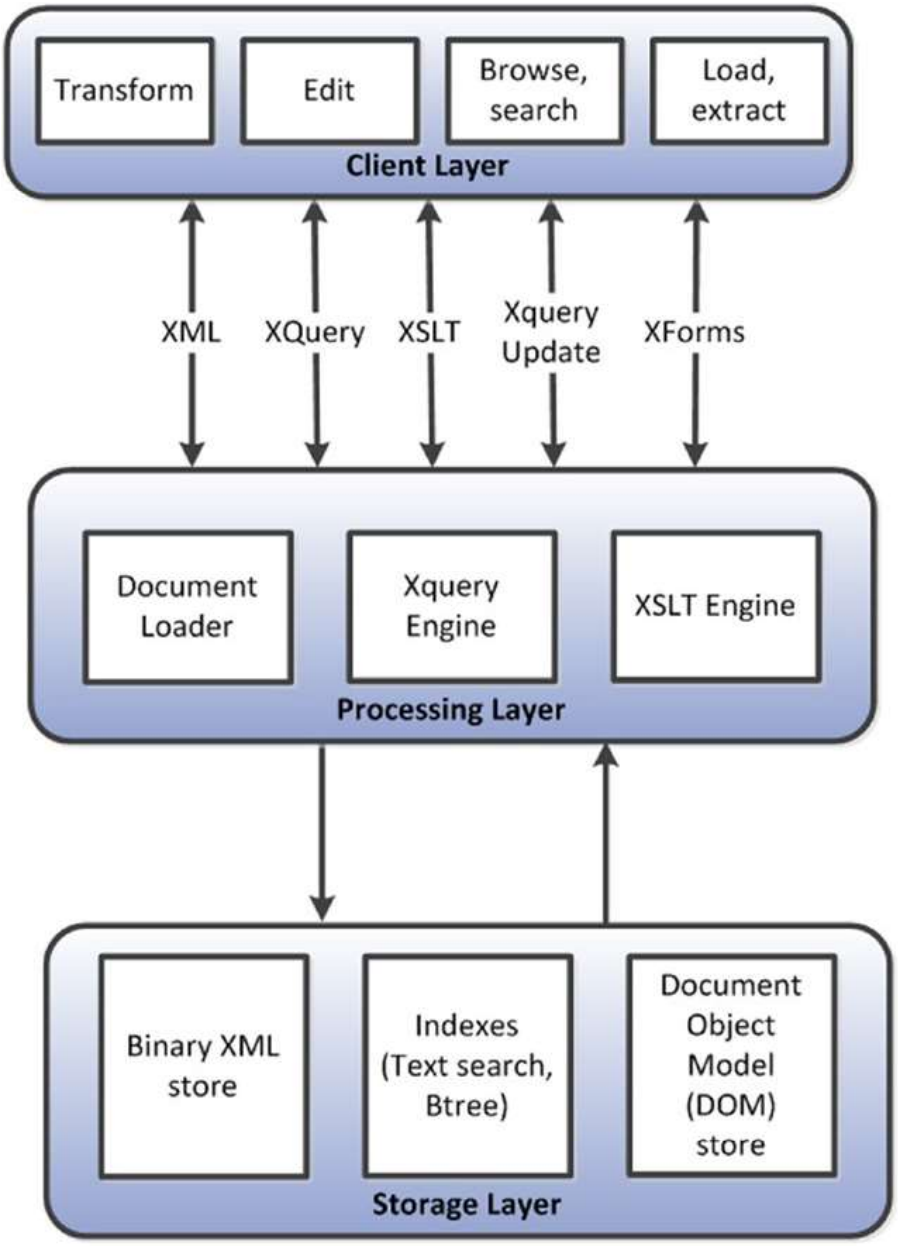

Figure 4-2. *Generic XML database architecture*

A variety of XML databases emerged during the first half of the 2000s, and they experienced a healthy uptake. However, they were not positioned as alternatives to the RDBMS but, rather, as a means of managing the XML document sprawl that had confronted many organizations.

Two of the more significant XML databases include the open-source XML database *eXist* and the commercial XML database *MarkLogic*.

XML Support in Relational Systems

Relational database vendors all introduced XML support within their core offerings, typically allowing XML documents to be stored within long/BLOB columns within database tables and providing support for the various XML standards such as DOM, XSLT, and XQuery. In addition, XML support was added to the ANSI SQL definition (SQL/XML), allowing XML manipulation within standard SQL language statements. Support for these SQL extensions appears in Oracle, Postgres, DB2, MySQL, and SQL Server.

JSON Document Databases

XML has many advantages as a standard format for file-based documents and for data interchange. But it has a number of drawbacks as a storage format for serious database applications. XML is justifiably criticized as being wasteful of space and computationally expensive to process. XML tags are verbose and repetitious, typically increasing the amount of storage required by several factors. Partly as a result, the XML language format is relatively expensive to parse.

XML has long been the predominant format for structured files, but for data interchange and for document databases themselves, a more recent format—*JavaScript Object Notation* (*JSON*)—promised greater benefits and has achieved far greater popularity than their XML predecessors.

JSON document-based databases and XML document-based databases share many similarities, not the least of which is the superficial similarity between XML and JSON formats. However, each is designed to support quite dissimilar use cases and significantly different applications. XML databases typically are used as content-management systems; that is, organizing and maintaining collections of text files in XML format—academic papers, business documents, and so on. JSON document databases, on the other hand, mostly support web-based operational workloads—storing and modifying the dynamic content and transactional data at the core of modern web-based applications.

■ **Note** JSON and XML are similar formats, but XML document databases and JSON document databases generally support dissimilar applications. XML document databases excel for content management systems, JSON document databases generally aim to support operational web applications.

JSON and AJAX

JSON was created by JavaScript pioneer Douglas Crockford as part of an attempt to build a framework for more dynamic and interactive web applications. JSON was deliberately intended as a more lightweight substitute for XML, and it became a significant alternative to XML in the AJAX programming model that drove a revolution in web applications during the mid-2000s.

AJAX (Asynchronous JavaScript And XML) is a flexible programming pattern in which JavaScript on a web browser communicates with a back-end web server through asynchronous interchange of XML or JSON documents, rather than by the exchange of complete HTML pages. It was AJAX that enabled the rich and interactive experience provided by groundbreaking web applications such as Google Maps and Gmail.

Although the "X" in AJAX refers to XML, JSON documents can also be used as the medium of interaction with the web server. Indeed, because of its tight integration with JavaScript, JSON increasingly became more popular than XML for data interchange. Within a few years of the introduction of the AJAX approach, all serious web developers became familiar with JavaScript programming and with the JSON model.

Figure 4-3 compares the appearance of JSON and XML documents.

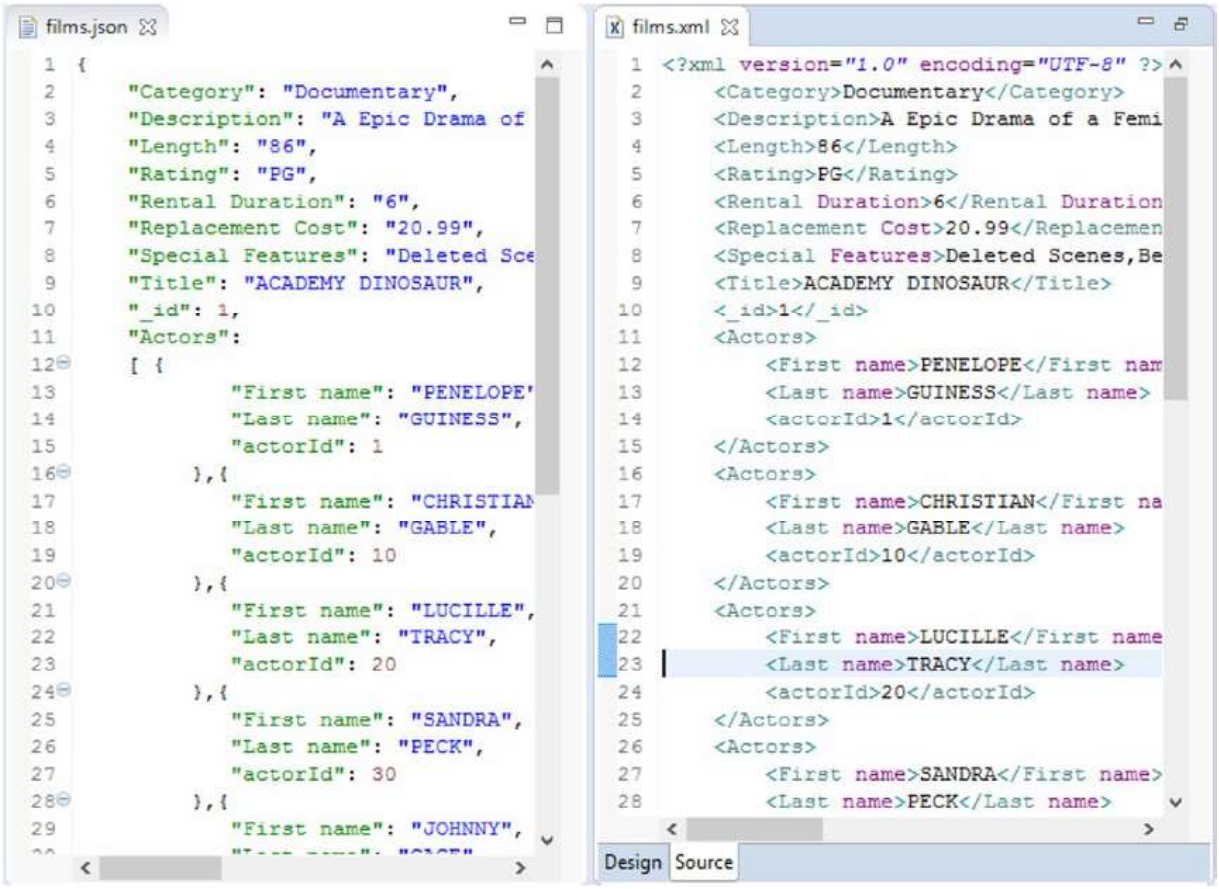

Figure 4-3. *Comparison of XML and JSON file formats*

JSON Databases

The emergence of key-value store databases such as Amazon's Dynamo, together with the growing popularity of JSON as a data interchange format, set the stage for the emergence of the JSON document database.

There is not a single specification or manifesto outlining the properties of a JSON-based document database. To be a JSON document database, all you need to do is store data in the JSON format.

In a JSON document database, the hierarchy of storage is typically as follows:

- A *document* is the basic unit of storage, corresponding approximately to a row in an RDBMS. A document comprises one or more key-value pairs, and may also contain nested documents and arrays. Arrays may also contain documents allowing for a complex hierarchical structure.
- A *collection* or *data bucket* is a set of documents sharing some common purpose; this is roughly equivalent to a relational table. The documents in a collection don't have to be of the same type, though it is typical for documents in a collection to represent a common category of information.

While a document database could theoretically implement a third normal form schema exactly as it would be used within a relational system, document databases more typically model data in a smaller number of collections, with nested documents representing master-detail relationships.

For instance, consider a database of movies and actors. In a relational data model, we would represent actors and films in separate tables, and create a join table that would indicate which actors appeared in which films, as shown in Figure 4-4.

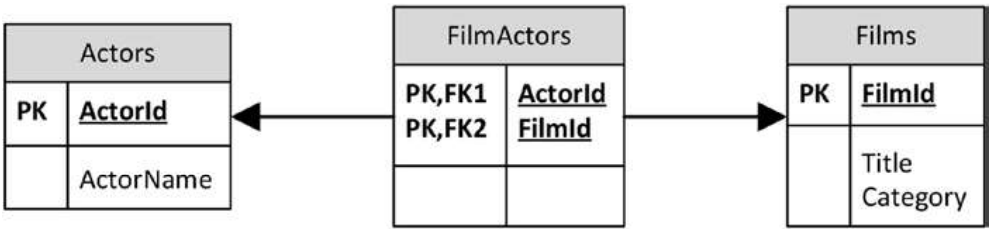

Figure 4-4. *Simple Relational movie database*

In a JSON document database, we could create three collections with key-value pairs corresponding to the columns in a relational schema—but to do so would be unnatural. Document databases do not generally provide join operations, and programmers generally like to have the JSON structure map closely to the object structure of their code. So it would be more natural to represent this data as shown in Figure 4-5.

Figure 4-5. *JSON movie collection*

Data Models in Document Databases

In Figure 4-5, "actors" are nested as an array within the "films" documents. This pattern is often referred to as *document embedding*. The design pattern has the advantage of allowing a film and all its actors to be retrieved in a single operation, and it avoids the need to perform joins within the application code.

On the other hand, the approach results in "actors" being duplicated across multiple documents, and in a complex design this could lead to issues and possibly inconsistencies if any of the "actor" attributes need to be changed. The number of actors in a film is relatively small, but in other application scenarios, problems can also occur if the number of members in an embedded document increases without limit (because the size of a single JSON document is usually capped—64MB in MongoDB, for instance).

For these reasons, a database designer might choose instead to link multiple documents using document identifiers, much in the way a relational database relates rows via foreign keys. For instance, in Figure 4-6, we embed an array of actor IDs into the "films" document, which can be used to locate the actors who appear in a film.

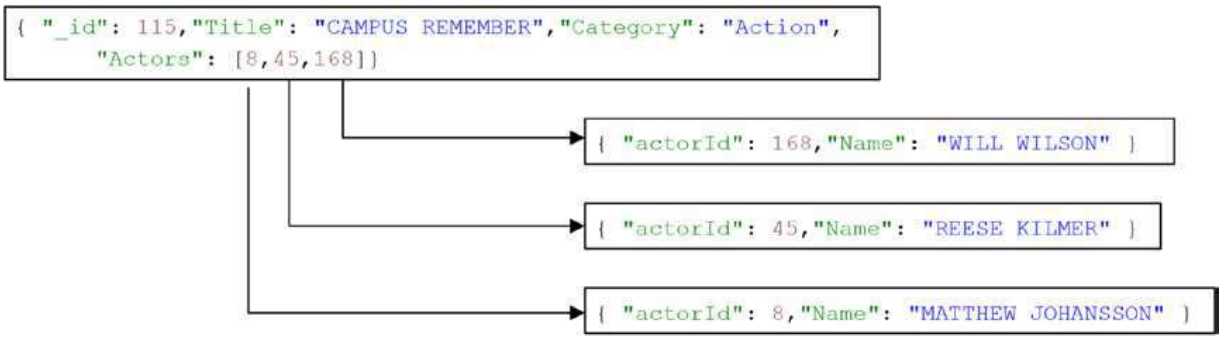

Figure 4-6. *Linking documents in a document database*

In Figure 4-7, we see another example of document linking. In this case, the design is in a relational style: a link document relates films to actors. Although this is somewhat an unnatural style for a document database, for some workloads it may provide the best balance between performance and maintainability.

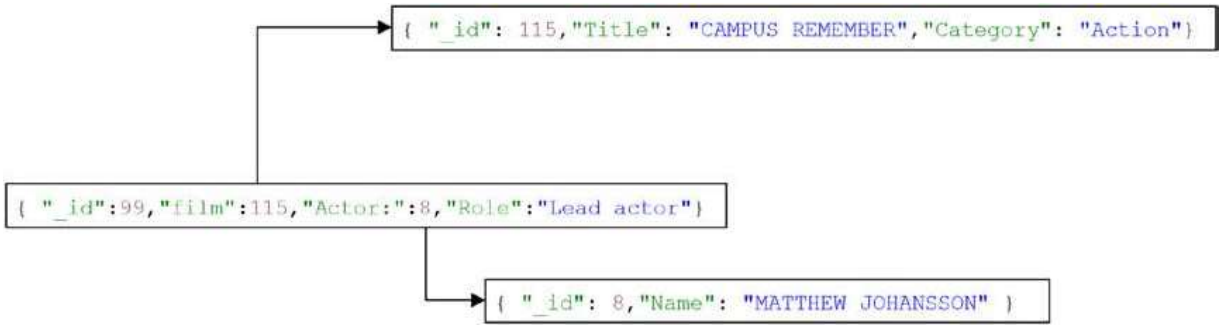

Figure 4-7. *Document linking can resemble relational third normal form*

Data modeling in document databases is less deterministic than for relational databases: there is no equivalent of third normal form that defines a "correct" model. Furthermore, while modeling in a relational database is driven primarily by the nature of the data to be stored, in a document database the nature of the queries to be executed is far more significant.

Early JSON Databases

CouchDB, created by Damien Katz, was the first notable JSON-based database system. Damien Katz had worked on Lotus Notes, a collaboration system with strong document-handling capabilities. In 2005, he decided to create a database system more closely aligned with web development and object-oriented programming models. The result was CouchDB.

Initially, CouchDB was written in C++ and it stored XML documents. Around 2007, a new architecture emerged that incorporated JSON as the primary storage format, a JavaScript command, and query interface and a core engine rewritten in the Erlang language.

As it matured, CouchDB embraced other paradigms that had inspired key-value stores and Hadoop, including a JavaScript implementation of MapReduce, an eventual consistency and multiple versioning model, and a hash-based clustering/sharding model to allow CouchDB to scale across nodes.

CouchDB became an Apache project in 2008 and was supported by IBM. In 2009, a commercial company couch.io (later CouchOne) was formed to maintain and promote the technology.

MemBase and CouchBase

When the interest in nonrelational databases exploded in 2009, CouchDB had already several years of active development and seemed well placed to benefit from the increasing buzz surrounding "NoSQL."

Membase was another nonrelational system experiencing significant uptake during this period. The Membase database provided a persistent variation on the extremely popular *Memcached* framework, which we discussed in Chapter 3. Memcached is a distributed read-only object cache that is commonly deployed in conjunction with MySQL to reduce database load. Objects are distributed across multiple Memcached nodes, and it can be located by a hash key lookup. If the data is in a Memcached server, a database read is avoided.

Membase provided a Memcached-compatible solution in which data could also be modified and persisted to disk. Membase was therefore particularly attractive for those who had an existing investment in Memcached technology, offering the possibility that an application could be converted to Membase from Memcached/MySQL. Membase was initially well known as the database underlying Zynga's incredibly popular Farmville online game.

Meanwhile, despite CouchDB's many technical achievements, the CouchDB database and the CouchOne company appeared to be struggling to establish a commercial niche; it lacked a viable scale-out architecture. In early 2011, a merger between Membase and CouchOne was announced. The resulting company was called CouchBase. CouchBase donated the existing CouchDB code to the Apache community, and it embarked on a new effort to merge the capabilities of CouchDB and MemBase. In the resulting Couchbase server, the JSON engine from CouchDB was merged with the Memcached-compatible key-value layer from MemBase.

Couchbase inherited CouchDB's MapReduce interface for creating queries and views, but Couchbase 4.0 introduced a SQL-like layer for document access called *N1QL* (*Non-first Normal Form Query Language*). See Chapter 11 for more details on this and other SQL-like NoSQL interfaces.

MongoDB

In 2007, founders and senior engineers from the online ad-serving company DoubleClick, which had just been acquired by Google, established a new startup called 10gen. The company aimed to create a PaaS (Platform as a Service) offering similar to Google App Engine. The platform required a scalable and elastic data storage engine; in the absence of a suitable existing candidate, the team created its own database, which they called MongoDB. In 2008, 10gen pivoted to focus exclusively on MongoDB, and in 2009, it released the product under an open-source license together with a commercial enterprise distribution.

MongoDB is a JSON-oriented document database, although internally it uses a binary encoded variant of JSON called BSON. The BSON format supports lower parse overhead than JSON, as well as richer support for data types such as dates and binary data.

MongoDB provides a JavaScript-based query capability that is reasonably easy to learn—at least when compared to MapReduce approaches such as were initially required in CouchDB. Figure 4-8 compares a MongoDB JavaScript query to retrieve films that feature a specific actor with the SQL equivalent in MySQL. We'll look at the MongoDB query language in detail in Chapter 11.

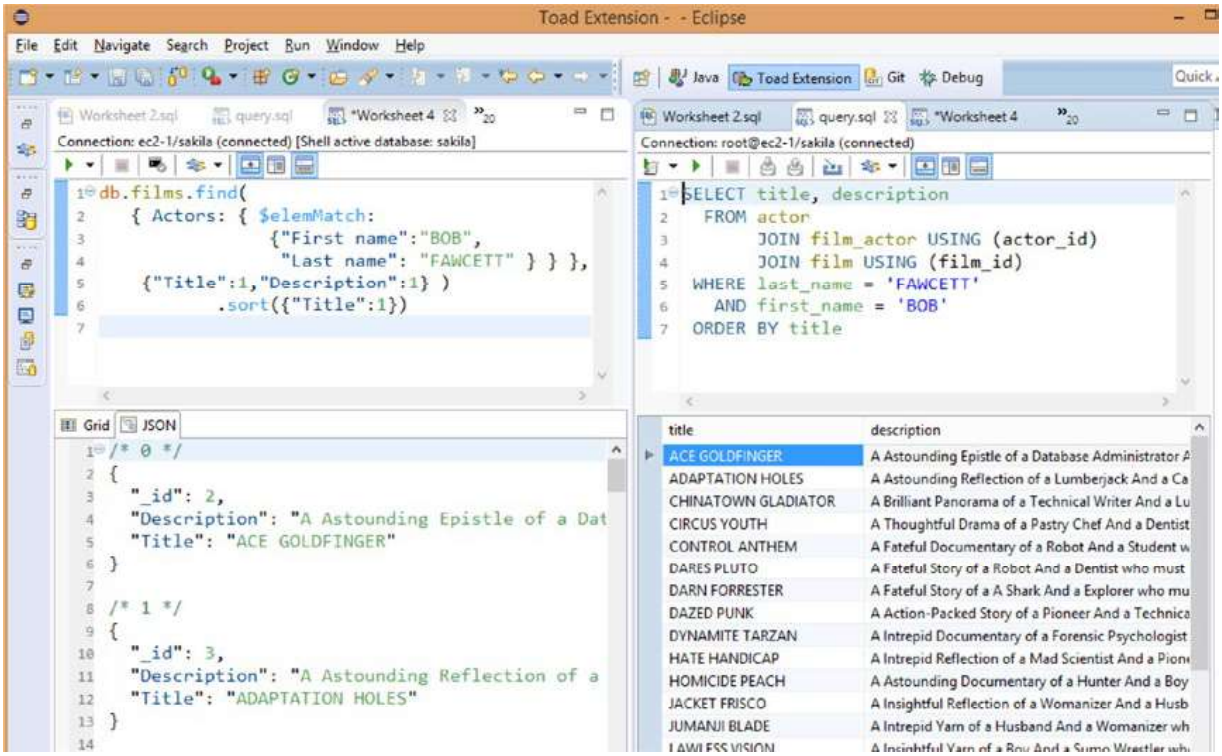

Figure 4-8. *MongoDB JavaScript query and SQL equivalent*

MongoDB established a strong lead in the NoSQL database space by providing a developer-friendly ecosystem and architecture. Developers looking for a nonrelational alternative—typically to MySQL or Oracle—found it relatively easy to get started with MongoDB. Developer-led adoption of MongoDB was robust, and today MongoDB can claim to be the most widely used nonrelational database.

In many respects, the rise of MongoDB today resembles the ascendance of MySQL in the last decade. In both cases these databases entered the organization not as a result of some sort of l strategic technology plan, but as the result of its popularity with developers. Just as MySQL became the default database for LAMP (Linux-Apache-MySQL-PHP) stack applications in the early 2000s, MongoDB seems to have achieved a similar position within the modern web-development community.

MongoDB may lack some of the scalability and throughput capabilities of other NoSQL offerings such as Cassandra or HBase, although the implementation of the WiredTiger storage engine in version 3.0 has provided the base engine with a significant increase in capability. Nevertheless, it has powered many high-end, large-scale websites, and it looks poised to be a leading NoSQL database for the immediate future.

JSON, JSON, Everywhere

The central role of JSON within modern document databases does not preclude the use of JSON in other systems.

Just as the relational world embraced XML and introduced XML features into the SQL standard, many RDBMS vendors are offering native support for JSON. For instance, Oracle and other relational vendors have introduced features in the SQL language to allow JSON documents stored within database tables to be manipulated within SQL statements. Indeed, as we'll see in Chapter 12, Oracle now provides a completely non-SQL access path for JSON.

It has, of course, always been possible and not unusual to store JSON documents within a key-value store object. However, pure key-value databases such as Riak have also introduced JSON-aware indexing schemes. Furthermore, the latest release of the Cassandra database allows JSON structures to be mapped directly to Cassandra table columns.

MarkLogic—the successful commercial XML database vendor mentioned earlier in this chapter—added support for native JSON storage in 2014. Based on its broad penetration as an XML database, MarkLogic can now claim to be a leading contender in the modern document database market, though it remains to be seen if MarkLogic can break out of its XML content management niche to compete directly with databases such as MongoDB and CouchBase for the hearts and minds of web application developers.

In any case, we can expect to see some level of JSON support in almost all database systems.

Conclusion

Document databases differentiate from other relational and nonrelational systems through their adoption of a document format—XML or JSON—as the data model. Document databases that support XML document formats have been primarily important as content management systems, providing a management repository for XML-based text files. JSON document databases, on the other hand, use JSON documents as the data layer for web-based applications—the sort of purpose to which MySQL would previously have been tasked. To some degree, XML-based document databases represent a previous generation of database system, while JSON databases can claim to be the next generation.

The popularity of JSON and the growing support for JSON within relational databases and key-value stores blur the lines that initially existed between databases like MongoDB and Dynamo-inspired databases such as Riak. In the near future, support for JSON may be a feature common to all databases, rather than one that differentiates a specific database technology segment.

However, today JSON document databases represent a distinct and important niche within next generation of database systems. They provide useful features beyond those of pure key-value stores, particularly in terms of programmer productivity and data accessibility.

The simplistic self-describing model of JSON and simple key-value access paths work well for web applications where simple CRUD (Create, Read, Update, Delete) access paths dominate. However, modern applications have increasingly generated data models that not only exceed the sophistication of document oriented key-value stores but also exceed the comfortable modeling capability of even the relational models. We'll consider these graph databases in the next chapter.