CHAPTER 28

Relational Databases and "Big Data": The Alternative of a NoSQL Solution

Abstract

This chapter provides an overview of NoSQL databases, an alternative to relational databases for handling big data (particularly for data in data warehouses). It covers the four major types of NoSQL data models, hardware architecture differences between relational and NoSQL databases, and NoSQL transaction control (BASE transactions).

Keywords

big data
NoSQL
NoSQL databases
NoSQL DBMSs
key-value store databases
document databases
column databases
graph databases
BASE transactions
NoSQL concurrency control
NoSQL transaction control
NoSQL benefits
NoSQL problems

Although relational database design theory has been relatively stable for the past 45 years, as you have read, other data models continue to be developed to handle changes in the data management environment. In particular, relational DBMSs can have trouble retrieving data with acceptable performance from extremely large data stores. These databases—often simply called *big data*—include massive installations, such as Amazon and Google.

The largest group of DBMSs designed to handle such huge amounts of data are known as *NoSQL DBMSs*. There doesn't seem to be any agreement on the origin of the "No" part of the name. There are certainly some products that use SQL in the background, so it doesn't necessarily mean that a NoSQL DBMS avoids SQL altogether. Some people insist that it means "not only SQL," which may indeed be the case. Regardless of the source of the name, however, NoSQL DBMSs have become very useful in data warehousing and in environments with enormous volumes of operational data.

In this chapter, we will look at the types of NoSQL databases and how they differ from relational databases. The goal is for you to understand enough about them so that you can recognize circumstances where a NoSQL solution might be appropriate. Keep in mind, however, that at the time this book was written, NoSQL databases were being used by less than 10 percent of even very large businesses.

Types of NoSQL Databases

Unlike the relational data model, there is no single NoSQL data model. There are, however, four commonly used types of NoSQL design.

Note: Much of what you read in this section is a generalization of each type of NoSQL design. Individual DBMSs often provide additional features to compensate for limitations of a particular type of storage.

Key-Value Store

A *key-value store* database assigns a unique key to a data value. The values are stored as an undifferentiated block and are retrieved by supplying the correct key. Because the data are seen as a single value, the data are not searchable. This pattern has some important implications:

- Retrieval by the key will be extremely fast.
- There is virtually no restriction on the type of data that can be stored. You could store text (for example, the HTML code for a Web page) or any type of multimedia binary (still images, audio, and video).
- To gain what appears to be search access to text data, the database designer must create indexes on the text. Keys are paired with text in the index and the search occurs on the index rather than the stored data.

Key-value store DBMSs use three commands taken from Web languages to manipulate the values in the database:

- PUT: inserts a key-value pair into the database

- PUT: inserts a key-value pair into the database
- GET: retrieves a value using a supplied key
- DELETE: removes a key-value pair from the database

Why would you want to use a database where you can't search the data? Anywhere you need to store and retrieve entire files. You could, for example, store an entire Web site where the key is a page's URL. When a user makes a request from a Web server to display a particular page, the Web server doesn't need access to the page's internal code; it just needs the HTML text as a unit. Therefore, the Web server submits the URL to the key-value store DBMS, which, in turn, uses the key to find the requested page. Any scripts, images, and videos used by the page could be stored and retrieved in the same way.

A key-value store database has no schema. In fact, the values associated with the keys do not need to be the same type of item. For example, one value could be a string of text, while another was a graphic image.

The biggest advantage to key-value store databases is extremely fast retrieval. However, if you need to be able to search within stored values rather than always retrieving by the key, this type of DBMS may not be a good fit for your data. Keep in mind that you can't update parts of a "value" while it's in the database. You must replace the entire value with a new copy if modifications are needed.

Key-value store databases are therefore best suited for applications where access is only through the key. They are being used for Web sites that include thousands of pages, large image databases, and large catalogs. They are also particularly useful for keeping Web app session information.

Document

A *document store* NoSQL database is very similar to a key-value store, but rather than storing "values," it stores "documents," which are made up of individual pieces of named data. Documents can be nested within documents.

Text data can be searched by creating indexes to the elements within the documents. Although not all documents need to have the same structure, the elements within a document can be named, making it possible to index on all items of the same name.

Document store databases do not require a schema, but each document does have a primary key, a value that uniquely identifies the document. The primary key index is a permanent feature of the database, but secondary indexes can be created and deleted as needed.

Document store databases work well for large on-line catalogs, for example. They provide fast access by a catalog number. However, the ability to build secondary indexes supports fast access using keywords such as those a user might use in a search.

Note: Some NoSQL products (for example, Amazon's DynamoDB) support both key-value and document stores. DynamoDB is completely cloud-based. You pay for the number of reads and writes you perform each month, as well as a fee for the amount of storage you use. Amazon handles all maintenance, including scaling the database when extra capacity becomes necessary.

Column

A *column-oriented* NoSQL database at first looks deceptively like a relational database: The concepts of columns and rows are present; data manipulation is provided by commands that are a variation on SQL syntax.[1] However, there are some major differences. In particular, column-oriented databases don't support joins.

In a relational database, each table has a fixed structure: Every row has space for every column. This is not the case in a column-oriented NoSQL database. Because the basic unit of storage is a column rather than a row, rows can be assigned different columns as needed. A column has a name (for example, first_name) that identifies the row within the column and a value (for example, "George") and, in many implementations, a timestamp. The timestamp not only anchors the data in time, but makes it easy to purge data that has been aggregated at any given point, as well as to keep multiple versions of the same data to track changes over time (*versioning*).

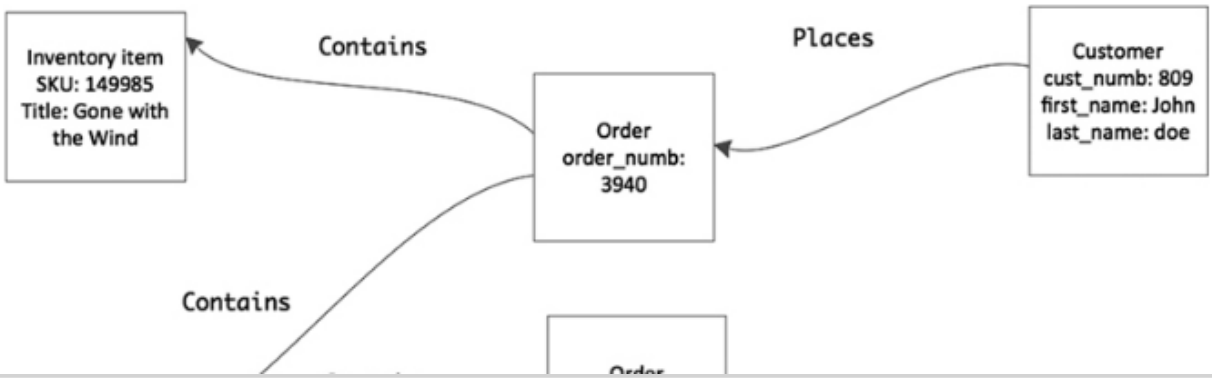
Columns are grouped into *column families*. When you place related columns in the same family, instances of those columns will be stored as physically close to each other as possible. You might, for example, create a column for each of street address, city, state, and zip. Then, you can group them into a column family with its own name (perhaps "address"). In contrast, most relational DBMSs try to store data in a single row together.

The biggest benefit of a column-oriented database is fast data aggregation. In other words, it will be extremely good at extracting data from a single column and providing summaries of that data. Google uses one, for example, to aggregate Web page visitation data. When the software has compiled the Web sites visited and the number of times each was visited, the data are archived.

Graph

A *graph* NoSQL database is made from *nodes* that are similar to rows in relational database tables. However, rather than using composite entities to show relationships and to store relationship data, graph databases use a representation of the relationships; the relationships can have properties, just like a composite entity stores relationship data. The result is something that looks like a mathematical directed graph.

The biggest advantage of the graph store is that joins aren't necessary. The pathways between related data exist within the database. As an example, consider the graph in Figure 28.1. It shows six instances of nodes from an *Antique Opticals* database.



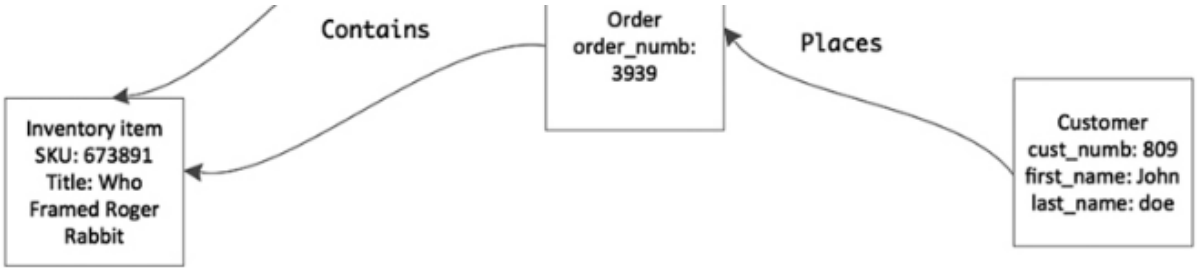


FIGURE 28.1 **Instances of nodes in a NoSQL graph database.**

The price charged the customer and the quantity ordered are properties of the Contains relationship. There is no composite entity between the order and the inventory item. To retrieve data, the DBMS follows the relationships stored in the database.

Note: Graph databases are navigational, like the data models discussed in Appendix A, because access to data follows predefined paths. However, the stored relationships in the older data models cannot have properties.

Graph databases are well suited to large databases that must track networks of relationships. They are used by many online dating sites, as well as LinkedIn and Twitter. Avoiding the joins to traverse friend relationships can speed up performance significantly. In addition, they can be used effectively to plot routes (such as for a package delivery service).

Other Differences Between NoSQL Databases and Relational Databases

At this point, it should be clear that the designs of relational and NoSQL databases are very different. There are also major differences in the hardware architectures, the way in which data are accessed and manipulated, and transaction control.

Hardware Architecture Differences

In Chapter 1 we looked at a variety of hardware architectures that are used for relational database implementations. The most difficult to implement successfully is a distributed architecture, particularly because of the concurrency control challenges. However, NoSQL databases are designed to be distributed. They scale horizontally by adding more servers to a data storage cluster. Keep in mind that many NoSQL databases are hosted in the cloud, an environment in which distribution is relatively easy to achieve.

NoSQL distributed solutions can also be very cost effective. Rather than needing large machines with massive amounts of storage, they can be implemented on small, commodity servers, such as those you might buy for a LAN subnet. Run out of space? Just add another machine. Doing so is significantly cheaper than adding similar capacity to a mainframe. Some people argue that the total-cost-of-ownership of a mainframe can be less than a distributed cluster, but this doesn't take into account the idea of incremental upgrades. It costs less to add a small, single server to a cluster, than it does to add RAM or permanent storage to a mainframe.

Note: It certainly would be nice to have some real cost comparisons here, but it's very difficult to find current pricing for mainframe components. Although commodity server prices are widely advertised, mainframe prices are typically negotiated with each individual customer and kept secret.

There are two major types of NoSQL distributed architectures that can be used individually or together.

Sharding

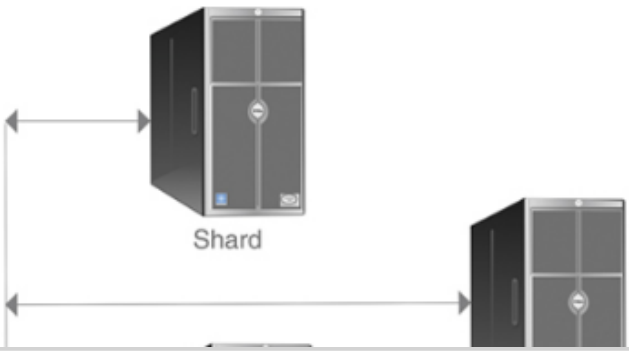
Sharding is another word for what we called partitioning in Chapter 8. It splits the database into unique pieces, each of which is hosted on a different server. For best performance, you want to keep data that are accessed together in the same shard (in other words, on the same physical machine).

The physical arrangement of your servers depends, to some extent, on the specific NoSQL DBMS. As an example, let's look at one option for MongoDB, a document-oriented open-source product. (Please don't forget that this architecture is very specific to MongoDB and that each DBMS will require something a bit different.)

Note: MongoDB also supports a combination of sharding and replication, which we will look at in just a bit.

As you can see in Figure 28.2, there are three types of servers in a MongoDB sharded cluster:

- Query router: A cluster requires at least one query router. It runs a copy of the MongoDB software and acts as a director for data manipulation requests.
- Config router: Although a cluster can run with only one config router, Mongo suggests that a cluster use three. Each contains metadata about where data are stored.
- Shard: The shard machines are the storage locations for the data.



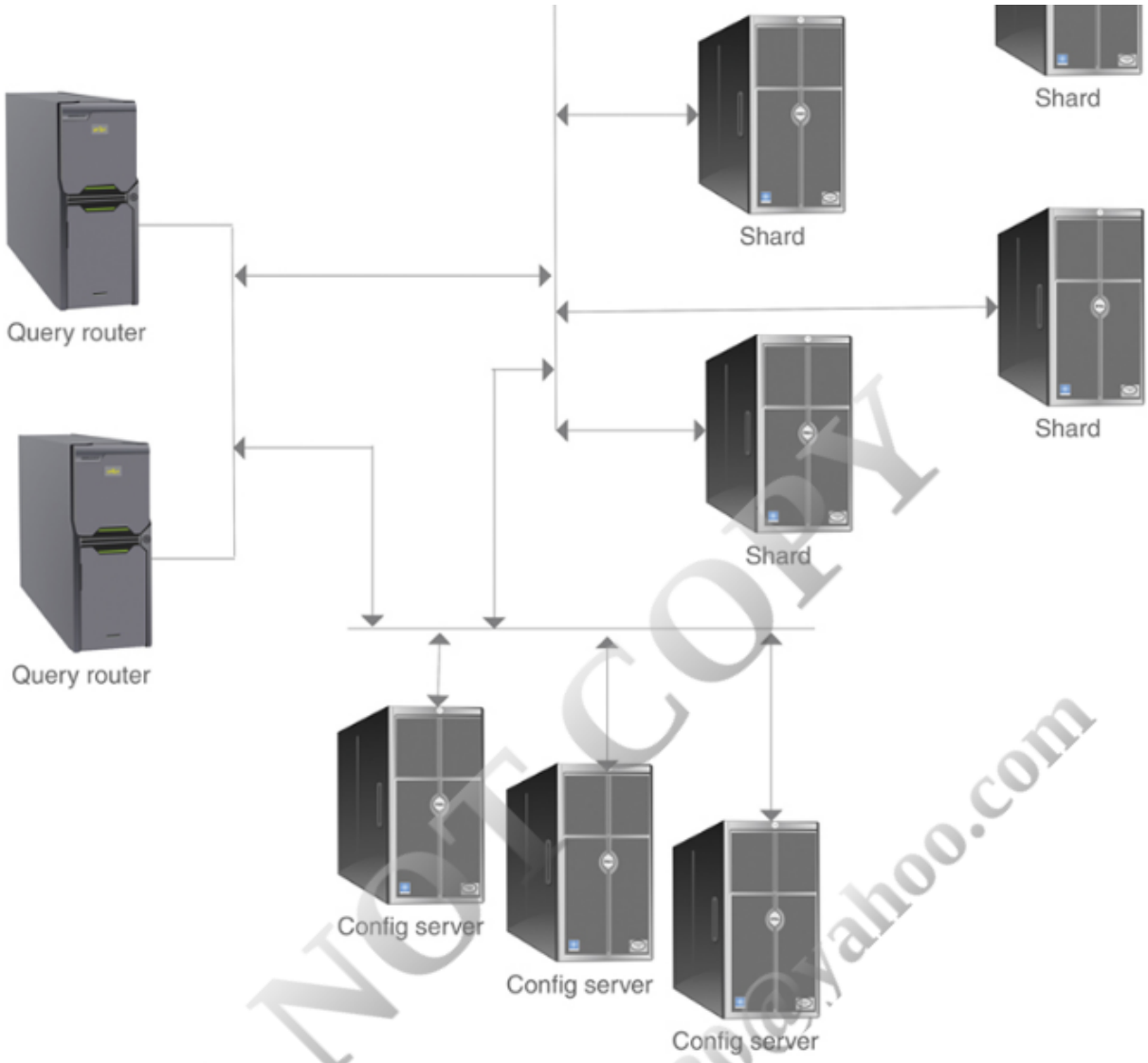


FIGURE 28.2 **MongoDB's sharding architecture.**

Data manipulation requests are sent to the query routers, which, in turn, use the metadata in the config routers to locate or place requested data. As each shard is a minimum of 64 Mb, sharding is truly intended for large databases.

Setting up a sharding environment is much more difficult than creating a NoSQL database on a single server. Doing so requires system programmer expertise in decisions such as the number of shards, the size of each shard, and the location of shards. It may also require selecting a *shard key* from which the location of a database element can be deduced by the DBMS. Shard keys must be unique, but because they contain location information, they aren't precisely the same as primary keys.

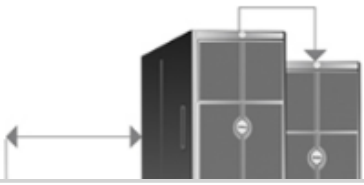
Replication

Replication means exactly what you might think: All or part of the database is replicated on more than one machine. If the copies are stored at remote locations, replication can provide faster retrieval for remote locations using those local data stores, as well as fault tolerance.

A replicated NoSQL database is subject to all of the problems of a distributed database that we discussed in Chapter 1. The biggest issue is consistency: How do you ensure that all the replicated copies are identical? As you will remember, locking isn't a particularly effective means of concurrency control in a distributed environment. NoSQL databases have therefore taken another tack to provide consistency. We'll look at it when we talk about BASE transactions later in this section.

Combining Sharding and Replication

Most NoSQL DBMSs support combining sharding and replication. As you can see in Figure 28.3, each shard is backed up by a second server that contains the copy of the shard.



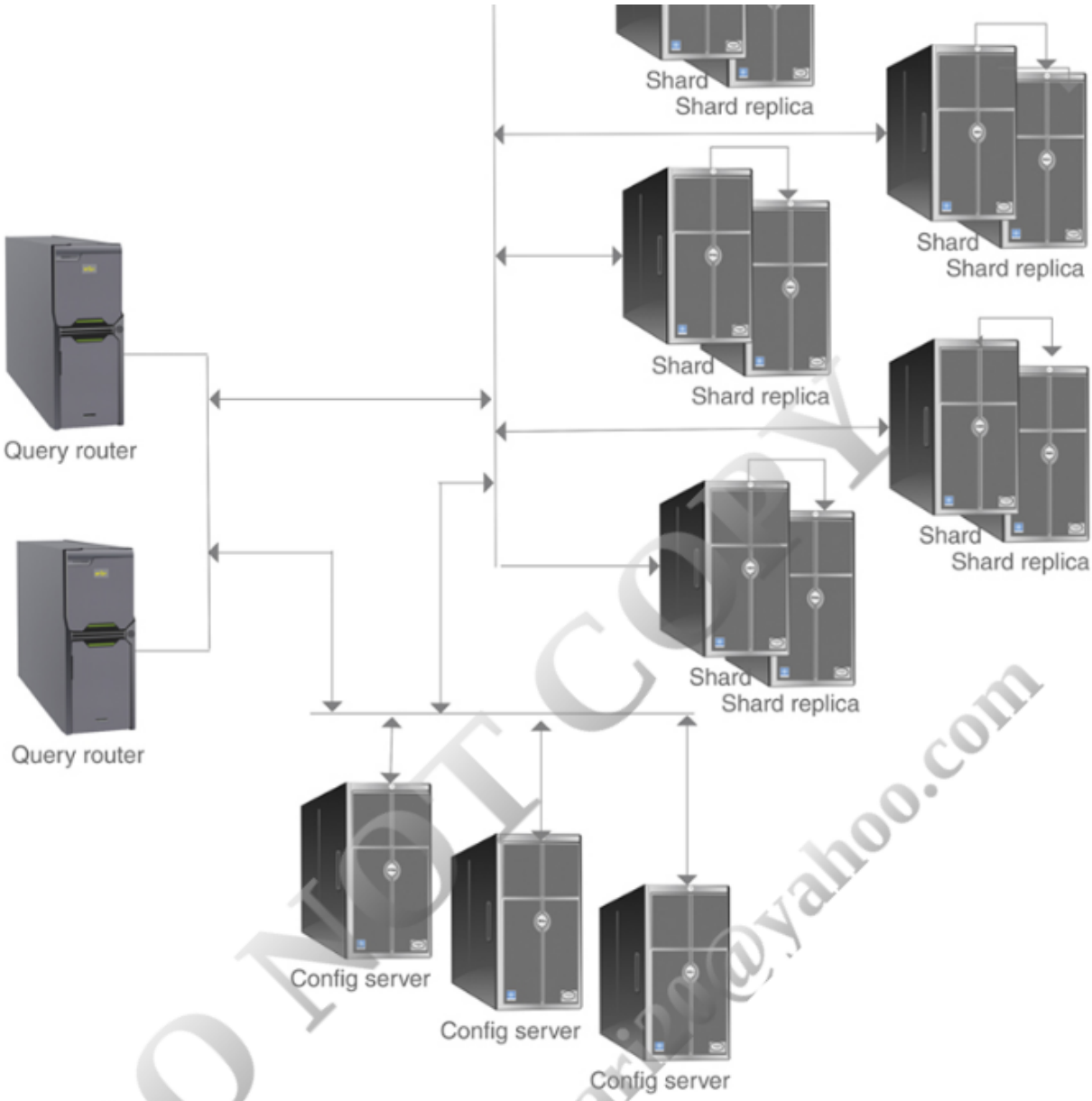


FIGURE 28.3 **Combining sharding and replication in a MongDB cluster.**

A database that is both sharded and replicated will have more fault tolerance than a database that is sharded alone. However, it will need to deal with the concurrency control issues associated with distributed databases.

Data Access and Manipulation Techniques

We are rather spoiled with SQL. You don't need to be a programmer to work at the command line to perform sophisticated database interactions. If you don't want to type commands, there are many query tools that support graphic or form-based query specification; the SQL is generated by the query tool, out of sight of the user. This works well because SQL is relatively standardized across relational DBMSs. The same can't be said for NoSQL products.

Virtually all current NoSQL products require configuration and data manipulation, either from a command line or from within an application program. This means that a company wanting to implement a NoSQL solution must have access to both systems and application programmers.

Transaction Control: BASE Transactions

One of the major goals of relational database transaction control is data consistency. As you read in Chapter 22, relational DBMSs typically enforce ACID transactions, ensuring that the database is internally consistent at all times. The main drawback is the overhead required by locking and the associated performance degradation. This is particularly problematic in distributed relational DBMSs.

With their emphasis on performance, many NoSQL DBMSs tolerate some data inconsistency, with the idea that eventually all the data will become consistent. Therefore, transaction control is BASE (Basic Availability, Soft State, and Eventual Consistency) rather than ACID.

With a relational database, we want serializable transactions (multiple transactions running interleaved produce the same result as they would if the transactions were run in a series rather than interleaved). The DBMS therefore locks portions of the database to prevent problems such as the lost update and inconsistent analysis.

Avoiding lost updates is essential in a transaction processing system. However, systems that are used primarily for querying, rather than a high volume of updates, can be allowed to suffer from lost updates. A report may be inaccurate for a while, until the writes catch up, but will reflect the

volume of updates, can be allowed to suffer from lost updates. A report may be inaccurate for a while, until the writes catch up, but will reflect the modified data when it is rerun.

In a BASE concurrency control environment, no writes are blocked. The goal is to avoid locking to make the database as available as possible. Whereas ACID concurrency control is pessimistic (something might go wrong, so we to make sure that doesn't happen), BASE transaction control is optimistic (everything will become consistent in the end). BASE transaction control is also well suited to distributed databases because it doesn't worry about keeping replicated copies consistent at every moment.

Benefits of NoSQL Databases

Given the difficulty in setting up and manipulating data in a NoSQL database, what can a company gain from one? There are several major advantages when you are maintaining an extremely large database:

- Fast retrieval: Through keys and fast-access paths such as indexes, NoSQL databases can provide access that is much faster than a relational search. A NoSQL solution can allow direct retrieval of a single unit of data using a key, thus avoiding the retrieval of one or more entire tables of data.
- Cloud storage that relieves the database owner of hardware and DBMS maintenance: Using a cloud-based product is typically cheaper than maintaining a NoSQL solution within an organization's premises. When you use a cloud-based solution, you are paying for what you use, rather than paying for excess capacity that might be used some time in the future.
- Lower-cost incremental hardware upgrades: As mentioned earlier in this chapter, it costs much more to add storage to a mainframe (cost per Mb, be it RAM or permanent storage) than it does to add the equivalent storage by adding commodity PCs to a cluster.

Problems with NoSQL Databases

NoSQL databases are well suited to some very specific environments involving large amounts of data, but there a number of problems that those developing the databases will encounter:

- Lack of design standards: Unlike like the relational data model, there are no standards for NoSQL designs. Implementations are not consistent, even within the same type. Any solution that a business implements will be locked into a specific DBMS.
- Lack of access language standards: There are no query languages for NoSQL databases. Data manipulation is provided through application programs using an API (*application programmer interface*) embedded in a high-level programming language, such as Python, JavaScript, Java or C + +. Prior to SQL, this is how we gained access to database data. The lack of standards also makes it very costly for an organization to change NoSQL products, because all previous code will need to be modified in some way to account for the new API.
- Access restrictions: NoSQL databases are not well suited to ad hoc querying. This is a direct result of there being no interactive query languages. They provide support for extremely large databases for which almost all queries can be anticipated and therefore coded into application programs.

Open Source NoSQL Products

If you want to play with NoSQL DBMSs, there are a number of open-source products that you can download. Unless stated otherwise, distributions are available for Windows, Mac OS X, and various flavors of UNIX:

DBMS	Type	Download Page
Apache CouchDB	Document	http://couchdb.apache.org
Apache HBase	Column	http://www.apache.org/dyn/closer.cgi/hbase/ (UNIX only)
LucidDB	Column	http://sourceforge.net/projects/luciddb/
Monetdb	Column	https://www.monetdb.org/Downloads
MongoDB	Document	https://www.mongodb.org/downloads
Neo4J	Graph	http://neo4j.com/download/ (Be certain to download the Community edition; the Enterprise edition is commercial.) (UNIX, including Mac OS X, only)
Redis	Key-value	http://redis.io/download (UNIX only)
Riak	Key-value	http://docs.basho.com/riak/latest/downloads/

For Further Reading

Alvina H. *RDBMS vs ORDBMS vs NoSQL*. Lambert Academic Publishing; 2015.
Dayley B. *NoSQL with MongoDB in 25 Hours*. Sams; 2015.
Fowler A. *NoSQL for Dummies*. For Dummies; 2015.
Kemme B, Jiménez-Peris R. *Database Replication*. Morgan and Claypool; 2010.
McCreary D, Kelly A. *Making Sense of NoSQL: A Guide for Managers and the Rest of Us*. Manning; 2014.
Redmond E, Wilson JR. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. Pragmatic Bookshelf; 2012.
Sadalage PJ, Fowler M. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley; 2012.
Sullivan D. *NoSQL for Mere Mortals*. Addison-Wesley; 2015.
Vaish G. *Getting Started with NoSQL*. Packt Publishing; 2013.

[1] In my opinion, the problem with using something that is "like" SQL, but with a number of syntax differences, will lead to a lot of confusion, especially if users know standard SQL.