chapter 24

NOSQL Databases and Big Data Storage Systems

We now turn our attention to the class of systems developed to manage large amounts of data in organizations such as Google, Amazon, Facebook, and Twitter and in applications such as social media, Web links, user profiles, marketing and sales, posts and tweets, road maps and spatial data, and e-mail. The term **NOSQL** is generally interpreted as Not Only SQL—rather than NO to SQL—and is meant to convey that many applications need systems other than traditional relational SQL systems to augment their data management needs. Most NOSQL systems are distributed databases or distributed storage systems, with a focus on semistructured data storage, high performance, availability, data replication, and scalability as opposed to an emphasis on immediate data consistency, powerful query languages, and structured data storage.

We start in Section 24.1 with an introduction to NOSQL systems, their characteristics, and how they differ from SQL systems. We also describe four general categories of NOSQL systems—document-based, key-value stores, column-based, and graph-based. Section 24.2 discusses how NOSQL systems approach the issue of consistency among multiple replicas (copies) by using the paradigm known as **eventual consistency**. We discuss the **CAP** theorem, which can be used to understand the emphasis of NOSQL systems on availability. In Sections 24.3 through 24.6, we present an overview of each category of NOSQL systems—starting with document-based systems, followed by key-value stores, then column-based, and finally graph-based. Some systems may not fall neatly into a single category, but rather use techniques that span two or more categories of NOSQL systems. Finally, Section 24.7 is the chapter summary.

24.1 Introduction to NOSQL Systems

24.1.1 Emergence of NOSQL Systems

Many companies and organizations are faced with applications that store vast amounts of data. Consider a free e-mail application, such as Google Mail or Yahoo Mail or other similar service—this application can have millions of users, and each user can have thousands of e-mail messages. There is a need for a storage system that can manage all these e-mails; a structured relational SQL system may not be appropriate because (1) SQL systems offer too many services (powerful query language, concurrency control, etc.), which this application may not need; and (2) a structured data model such the traditional relational model may be too restrictive. Although newer relational systems do have more complex object-relational modeling options (see Chapter 12), they still require schemas, which are not required by many of the NOSQL systems.

As another example, consider an application such as Facebook, with millions of users who submit posts, many with images and videos; then these posts must be displayed on pages of other users using the social media relationships among the users. User profiles, user relationships, and posts must all be stored in a huge collection of data stores, and the appropriate posts must be made available to the sets of users that have signed up to see these posts. Some of the data for this type of application is not suitable for a traditional relational system and typically needs multiple types of databases and data storage systems.

Some of the organizations that were faced with these data management and storage applications decided to develop their own systems:

- Google developed a proprietary NOSQL system known as **BigTable**, which is used in many of Google's applications that require vast amounts of data storage, such as Gmail, Google Maps, and Web site indexing. Apache Hbase is an open source NOSQL system based on similar concepts. Google's innovation led to the category of NOSQL systems known as **column-based** or **wide column** stores; they are also sometimes referred to as **column family** stores.
- Amazon developed a NOSQL system called **DynamoDB** that is available through Amazon's cloud services. This innovation led to the category known as **key-value** data stores or sometimes **key-tuple** or **key-object** data stores.
- Facebook developed a NOSQL system called **Cassandra**, which is now open source and known as Apache Cassandra. This NOSQL system uses concepts from both key-value stores and column-based systems.
- Other software companies started developing their own solutions and making them available to users who need these capabilities—for example, **MongoDB** and **CouchDB**, which are classified as **document-based** NOSQL systems or **document stores**.
- Another category of NOSQL systems is the **graph-based** NOSQL systems, or **graph databases**; these include **Neo4J** and **GraphBase**, among others.

- Some NOSQL systems, such as **OrientDB**, combine concepts from many of the categories discussed above.
- In addition to the newer types of NOSQL systems listed above, it is also possible to classify database systems based on the object model (see Chapter 12) or on the native XML model (see Chapter 13) as NOSQL systems, although they may not have the high-performance and replication characteristics of the other types of NOSQL systems.

These are just a few examples of NOSQL systems that have been developed. There are many systems, and listing all of them is beyond the scope of our presentation.

24.1.2 Characteristics of NOSQL Systems

We now discuss the characteristics of many NOSQL systems, and how these systems differ from traditional SQL systems. We divide the characteristics into two categories—those related to distributed databases and distributed systems, and those related to data models and query languages.

NOSQL characteristics related to distributed databases and distributed systems. NOSQL systems emphasize high availability, so replicating the data is inherent in many of these systems. Scalability is another important characteristic, because many of the applications that use NOSQL systems tend to have data that keeps growing in volume. High performance is another required characteristic, whereas serializable consistency may not be as important for some of the NOSQL applications. We discuss some of these characteristics next.

1. **Scalability:** As we discussed in Section 23.1.4, there are two kinds of scalability in distributed systems: horizontal and vertical. In NOSQL systems, **horizontal scalability** is generally used, where the distributed system is expanded by adding more nodes for data storage and processing as the volume of data grows. Vertical scalability, on the other hand, refers to expanding the storage and computing power of existing nodes. In NOSQL systems, horizontal scalability is employed while the system is operational, so techniques for distributing the existing data among new nodes without interrupting system operation are necessary. We will discuss some of these techniques in Sections 24.3 through 24.6 when we discuss specific systems.
2. **Availability, Replication and Eventual Consistency:** Many applications that use NOSQL systems require continuous system availability. To accomplish this, data is replicated over two or more nodes in a transparent manner, so that if one node fails, the data is still available on other nodes. Replication improves data availability and can also improve read performance, because read requests can often be serviced from any of the replicated data nodes. However, write performance becomes more cumbersome because an update must be applied to every copy of the replicated data items; this can slow down write performance if serializable consistency is required (see Section 23.3). Many NOSQL applications do not require serializable

consistency, so more relaxed forms of consistency known as **eventual consistency** are used. We discuss this in more detail in Section 24.2.

3. **Replication Models:** Two major replication models are used in NOSQL systems: master-slave and master-master replication. **Master-slave replication** requires one copy to be the master copy; all write operations must be applied to the master copy and then propagated to the slave copies, usually using eventual consistency (the slave copies will *eventually* be the same as the master copy). For read, the master-slave paradigm can be configured in various ways. One configuration requires all reads to also be at the master copy, so this would be similar to the primary site or primary copy methods of distributed concurrency control (see Section 23.3.1), with similar advantages and disadvantages. Another configuration would allow reads at the slave copies but would not guarantee that the values are the latest writes, since writes to the slave nodes can be done after they are applied to the master copy. The **master-master replication** allows reads and writes at any of the replicas but may not guarantee that reads at nodes that store different copies see the same values. Different users may write the same data item concurrently at different nodes of the system, so the values of the item will be temporarily inconsistent. A reconciliation method to resolve conflicting write operations of the same data item at different nodes must be implemented as part of the master-master replication scheme.

4. **Sharding of Files:** In many NOSQL applications, files (or collections of data objects) can have many millions of records (or documents or objects), and these records can be accessed concurrently by thousands of users. So it is not practical to store the whole file in one node. **Sharding** (also known as **horizontal partitioning** ; see Section 23.2) of the file records is often employed in NOSQL systems. This serves to distribute the load of accessing the file records to multiple nodes. The combination of sharding the file records and replicating the shards works in tandem to improve load balancing as well as data availability. We will discuss some of the sharding techniques in Sections 24.3 through 24.6 when we discuss specific systems.

5. **High-Performance Data Access:** In many NOSQL applications, it is necessary to find individual records or objects (data items) from among the millions of data records or objects in a file. To achieve this, most systems use one of two techniques: hashing or range partitioning on object keys. The majority of accesses to an object will be by providing the key value rather than by using complex query conditions. The object key is similar to the concept of object id (see Section 12.1). In **hashing**, a hash function $h(K)$ is applied to the key K, and the location of the object with key K is determined by the value of $h(K)$. In **range partitioning**, the location is determined via a range of key values; for example, location i would hold the objects whose key values K are in the range $Ki_{\text{min}} \le K \le Ki_{\text{max}}$. In applications that require range queries, where multiple objects within a range of key values are retrieved, range partitioned is preferred. Other indexes can also be used to locate objects based on attribute conditions different from the key K. We

will discuss some of the hashing, partitioning, and indexing techniques in Sections 24.3 through 24.6 when we discuss specific systems.

NOSQL characteristics related to data models and query languages. NOSQL systems emphasize performance and flexibility over modeling power and complex querying. We discuss some of these characteristics next.

1. **Not Requiring a Schema:** The flexibility of not requiring a schema is achieved in many NOSQL systems by allowing semi-structured, self-describing data (see Section 13.1). The users can specify a partial schema in some systems to improve storage efficiency, but it is *not required to have a schema* in most of the NOSQL systems. As there may not be a schema to specify constraints, any constraints on the data would have to be programmed in the application programs that access the data items. There are various languages for describing semistructured data, such as JSON (JavaScript Object Notation) and XML (Extensible Markup Language; see Chapter 13). JSON is used in several NOSQL systems, but other methods for describing semi-structured data can also be used. We will discuss JSON in Section 24.3 when we present document-based NOSQL systems.
2. **Less Powerful Query Languages:** Many applications that use NOSQL systems may not require a powerful query language such as SQL, because search (read) queries in these systems often locate single objects in a single file based on their object keys. NOSQL systems typically provide a set of functions and operations as a programming API (application programming interface), so reading and writing the data objects is accomplished by calling the appropriate operations by the programmer. In many cases, the operations are called **CRUD operations**, for Create, Read, Update, and Delete. In other cases, they are known as **SCRUD** because of an added Search (or Find) operation. Some NOSQL systems also provide a high-level query language, but it may not have the full power of SQL; only a subset of SQL querying capabilities would be provided. In particular, many NOSQL systems do not provide join operations as part of the query language itself; the joins need to be implemented in the application programs.
3. **Versioning:** Some NOSQL systems provide storage of multiple versions of the data items, with the timestamps of when the data version was created. We will discuss this aspect in Section 24.5 when we present column-based NOSQL systems.

In the next section, we give an overview of the various categories of NOSQL systems.

24.1.3 Categories of NOSQL Systems

NOSQL systems have been characterized into four major categories, with some additional categories that encompass other types of systems. The most common categorization lists the following four major categories:

1. **Document-based NOSQL systems:** These systems store data in the form of documents using well-known formats, such as JSON (JavaScript Object Notation). Documents are accessible via their document id, but can also be accessed rapidly using other indexes.
2. **NOSQL key-value stores:** These systems have a simple data model based on fast access by the key to the value associated with the key; the value can be a record or an object or a document or even have a more complex data structure.
3. **Column-based or wide column NOSQL systems:** These systems partition a table by column into column families (a form of vertical partitioning; see Section 23.2), where each column family is stored in its own files. They also allow versioning of data values.
4. **Graph-based NOSQL systems:** Data is represented as graphs, and related nodes can be found by traversing the edges using path expressions.

Additional categories can be added as follows to include some systems that are not easily categorized into the above four categories, as well as some other types of systems that have been available even before the term NOSQL became widely used.

5. **Hybrid NOSQL systems:** These systems have characteristics from two or more of the above four categories.
6. **Object databases:** These systems were discussed in Chapter 12.
7. **XML databases:** We discussed XML in Chapter 13.

Even keyword-based search engines store large amounts of data with fast search access, so the stored data can be considered as large NOSQL big data stores.

The rest of this chapter is organized as follows. In each of Sections 24.3 through 24.6, we will discuss one of the four main categories of NOSQL systems, and elaborate further on which characteristics each category focuses on. Before that, in Section 24.2, we discuss in more detail the concept of eventual consistency, and we discuss the associated CAP theorem.

24.2 The CAP Theorem

When we discussed concurrency control in distributed databases in Section 23.3, we assumed that the distributed database system (DDBS) is required to enforce the ACID properties (atomicity, consistency, isolation, durability) of transactions that are running concurrently (see Section 20.3). In a system with data replication, concurrency control becomes more complex because there can be multiple copies of each data item. So if an update is applied to one copy of an item, it must be applied to all other copies in a consistent manner. The possibility exists that one copy of an item *X* is updated by a transaction T_1 whereas another copy is updated by a transaction T_2, so two inconsistent copies of the same item exist at two different nodes in the distributed system. If two other transactions T_3 and T_4 want to read *X*, each may read a different copy of item *X*.

We saw in Section 23.3 that there are distributed concurrency control methods that do not allow this inconsistency among copies of the same data item, thus enforcing serializability and hence the isolation property in the presence of replication. However, these techniques often come with high overhead, which would defeat the purpose of creating multiple copies to improve performance and availability in distributed database systems such as NOSQL. In the field of distributed systems, there are various levels of consistency among replicated data items, from weak consistency to strong consistency. Enforcing serializability is considered the strongest form of consistency, but it has high overhead so it can reduce performance of read and write operations and hence adversely affect system performance.

The CAP theorem, which was originally introduced as the CAP principle, can be used to explain some of the competing requirements in a distributed system with replication. The three letters in CAP refer to three desirable properties of distributed systems with replicated data: **consistency** (among replicated copies), **availability** (of the system for read and write operations) and **partition tolerance** (in the face of the nodes in the system being partitioned by a network fault). *Availability* means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed. *Partition tolerance* means that the system can continue operating if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other. *Consistency* means that the nodes will have the same copies of a replicated data item visible for various transactions.

It is important to note here that the use of the word *consistency* in CAP and its use in ACID *do not refer to the same identical concept*. In CAP, the term *consistency* refers to the consistency of the values in different copies of the same data item in a replicated distributed system. In ACID, it refers to the fact that a transaction will not violate the integrity constraints specified on the database schema. However, if we consider that the consistency of replicated copies is a *specified constraint*, then the two uses of the term *consistency* would be related.

The **CAP theorem** states that it *is not possible to guarantee all three* of the desirable properties—consistency, availability, and partition tolerance—at the same time in a distributed system with data replication. If this is the case, then the distributed system designer would have to choose two properties out of the three to guarantee. It is generally assumed that in many traditional (SQL) applications, guaranteeing consistency through the ACID properties is important. On the other hand, in a NOSQL distributed data store, a weaker consistency level is often acceptable, and guaranteeing the other two properties (availability, partition tolerance) is important. Hence, weaker consistency levels are often used in NOSQL system instead of guaranteeing serializability. In particular, a form of consistency known as **eventual consistency** is often adopted in NOSQL systems. In Sections 24.3 through 24.6, we will discuss some of the consistency models used in specific NOSQL systems.

The next four sections of this chapter discuss the characteristics of the four main categories of NOSQL systems. We discuss document-based NOSQL systems in Section 24.3, and we use MongoDB as a representative system. In Section 24.4, we discuss