This book will introduce Redis and help you understand its various facets.

Starting with an introduction to NoSQL, you will learn how to install Redis and how to classify and work with data structures. By working with real world scenarios pertaining to using Redis, you will discover sharding and indexing techniques, along with how to improve scalability and performance through persistent strategies and data migration techniques. With the help of multiple examples, you will learn to design web and business applications. You will also learn how to configure Redis for setting up clusters and tuning it for performance.

At the end of this book, you will find essential tips on backup and recovery strategies for the Redis environment.

Who this book is written for
This book is for SQL developers who want to learn about Redis, the key value database for scalability and performance. Prior understanding of a programming language is essential; however no knowledge of NoSQL is required.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introduction to NoSQL'
- A synopsis of the book’s content
- More information on Learning Redis
About the Author

Vinoo Das has 16 years of experience in the software industry and has worked in various domains, such as telecom, banking, payment gateways, information management, and so on. He is highly motivated and loves to work on new and upcoming technologies. He is currently architecting a platform for an information technology giant, which will enable the company to position the platform at an enterprise level as well as a cloud solution.
Preface

Learning Redis is meant to be a guide and handbook for developers, architects, solution providers, consultants, engineers, and anyone planning to learn, design, and architect an Enterprise Solution and looking for an in-memory datastore that is agile and fast and extends its capabilities beyond just storing data.

This book starts with an introduction to the evolving landscape of NoSQL, explores commands in easy-to-understand examples, and then uses Redis in a few sample applications, with Redis as the backbone. The later sections of the book focus on the management of Redis for performance and scalability.

This book covers core concepts to design and create fast, agile, and concurrent applications, but it is not meant to be a replacement for the official documentation guide for Redis, published by Redis.io.

What this book covers

Chapter 1, Introduction to NoSQL, covers the ecosystem of NoSQL. It discusses the evolution of the NoSQL landscape and provides an introduction to the various types of NoSQL and their characteristics.

Chapter 2, Getting Started with Redis, taps into the world of Redis. It also covers areas such as the installation of Redis on various platforms and running a sample program in Java to connect to Redis.

Chapter 3, Data Structures and Communicating Protocol in Redis, covers data structures that are available in Redis and the communicating protocol in Redis. It also covers examples that the user can execute and get a feel of using it. By the end of this chapter, you should have a basic feel of the capabilities of Redis.
Preface

Chapter 4, *Functions in the Redis Server*, takes you from learning commands to the various in-built capabilities of Redis. These capabilities include messaging in Redis, transactions, and the pipeline capabilities in Redis, which have differences between them. The chapter also introduces the users to a scripting language called LUA.

Chapter 5, *Handling Data in Redis*, focuses on the in-depth data handling capability of Redis. This includes the master-slave arrangement, the way data is stored in Redis, and various options it provides to persist data.

Chapter 6, *Redis in Web Applications*, is all about positioning Redis in web applications. To make it interesting, there are some example applications, which you can take ideas from, about the wide range of use cases where Redis can be used.

Chapter 7, *Redis in Business Applications*, is all about positioning Redis in business applications. To expand its applicability further in the Enterprise Solution design landscape, some example applications have been explained from which you can see its versatility.

Chapter 8, *Clustering*, talks about the clustering capability, how the end user can make use of various patterns in clustering for Redis, and use these patterns accordingly in their solutions.

Chapter 9, *Maintaining Redis*, is all about maintaining Redis in a production environment.
Introduction to NoSQL

In this chapter, you will learn about the emerging realm of NoSQL and get introduced to various classifications in the NoSQL domain. We will also understand the position of Redis in the NoSQL domain. We'll cover the following topics:

- Data in Enterprise
- NoSQL
- Use cases for NoSQL

An Internet-enabled world

We live in interesting times; in the last decade, a lot of changes have happened that have changed the way we experience the world of the Internet and the ecosystem around it. In this chapter, we will focus on some of the reasons that led to progress and discuss the developments happening in the world of data storage.
The following figure is a rough sketch of the evolution process that happened in cyberspace, the data for which is collected from the Internet, and gives a rough idea of the growth experienced in Internet-based services:

<table>
<thead>
<tr>
<th></th>
<th>2000 - 2005</th>
<th>2006 - 2009</th>
<th>2010 – present day</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware</strong></td>
<td>Pentium series</td>
<td>Dual Core makes an entry, after that cores have been adding up (computer, tablets, smartphones)</td>
<td></td>
</tr>
<tr>
<td><strong>Internet users</strong></td>
<td>360,985,492 (2006) to 2,405,518,376 (present day and increasing)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Evolution: Social media, processors and cores, databases (NoSQL)

The preceding chart indicates that the hardware industry saw a paradigm shift during the middle half of the first decade. Instead of new processors coming out with increased clock speeds, the newer generation of processors came with multiple cores and their numbers increased in processors with a subsequent release. Gone were the days when a big machine with lots of memory and a powerful processor could solve any problem or, in other words, when an Enterprise depended on vertical scaling to solve their performance issues. What it signaled, in a way, was that parallel computing was the future and it will be deployed on commodity-based machines.

With the hardware industry signaling the arrival of parallel computing, the newer generation of solutions had to be distributed and parallel in nature. This means that they needed to have logic executed in parallel and data stored in distributed datastores; in other words, horizontal scaling was the way to go. Moreover, with Web 2.0, there was an emergence of social media, online gaming, online shopping, collaborative computing, cloud computing, and so on. The Internet was becoming a ubiquitous platform.
The popularity of the Internet and the number of people using the Internet was increasing by the day, and the amount of time spent on the Internet was also increasing. Another important aspect to be looked at was that users across geographies were coming together in this Internet-enabled world. There are many reasons for this; for one, websites were becoming intelligent and in a way, were engaging end users far effectively than their predecessors. Another factor that was making Internet adoption faster and easier were innovative handheld devices, such as smartphones, tablets, and so on. Nowadays, the kind of compute power these handheld devices have can be compared to that of computers. In this dynamically changing world, Internet-based software solutions and services are expanding the horizon of social media, which brings people together on a common platform. This created a new business domain like social-Enterprise media, where social media bridges with Enterprise. This was definitely going to have an impact on traditional Enterprise solutions.

The Internet effect made Enterprise solutions undergo a metamorphic shift. The shift in Enterprise architecture went from a nuanced set of requirements, typically expected from Enterprise solutions, to adopting newer requirements, which were the bastion of social media solutions. Nowadays, Enterprise solutions are integrating with social media sites to know what their customers are talking about; they themselves have started creating platforms and forums where the customer can come and contribute their impressions about products and services. All this data exchange happens in real time and needs a highly concurrent and scalable ecosystem. To sum it up, Enterprise solutions want to adopt the features of social media solutions, and this has a direct and proportional bearing on the nonfunctional requirements of their architectures. Features such as fault management, real-time big data crunching, eventual consistency, high numbers of reads and writes, responsiveness, horizontal scalability, manageability, maintainability, agility, and so on, and their impact on Enterprise architecture, are being looked at with renewed interest. Techniques, paradigms, frameworks, and patterns that were used in social media architecture are being studied and reapplied in Enterprise architecture.

One of the key layers in any solution (social media or Enterprise) is the data layer. Data, the way it is arranged and managed, and the choice of datastore forms the data layer. From a designer's perspective, data handling in any datastore is governed by perspectives such as consistency, availability, and partition tolerance, or better known as Eric Brewer's CAP theorem. While it is desirable to have all the three, in reality, any data layer can have a combination of two of the mentioned perspectives. What this means is that the data in a solution can have many combinations of perspectives, such as availability-partition tolerance (this combination has to forego consistency in data handling), availability-consistency (this combination has to forego partition tolerance which will impact the amount of data that the data layer can handle), and consistency-partition tolerance (this combination has to forego availability).
The CAP theorem has a direct bearing on the behavior of the system, read/write speeds, concurrency, maintainability, clustering patterns, fault tolerance, data loads, and so on.

The most common approach when designing the data model is to arrange it in a relational and normalized way. This works well when the data is in transactional mode, needs consistency, and is structured, that is, it has a fixed schema. This approach of normalizing data appears over-engineered when the data is semistructured, has a tree-like structure, or is schema-less, where consistency can be relaxed. The end result of making semistructured data fit into a structured data model is the explosion of tables and a complicated data model to store simple data.

Due to the lack of alternatives, the solutions have been overtly relying on RDBMS to address concerns regarding data handling. The problem with this approach is RDBMS, which was primarily designed to address consistency and the availability perspective of data handling, also started to store data, which had concerns of partition tolerance. The end result was a bloated RDBMS with a very complex data model. This started impacting the nonfunctional requirements of a solution negatively, in the areas of fault management, performance, scalability, manageability, maintainability, and agility.

Another area of concern was Data Interpretation, which is very important while designing the data layer. In a solution, the same data is viewed and interpreted differently by a different concerned group. To give a better idea, let's say that we have an e-commerce website that sells products. Three basic functional domains come into play in the design of this data layer; they are inventory management, account management, and customer management. From a core business standpoint, all the domains need atomicity, consistency, isolation, durability (ACID) properties in their data management, and from the CAP theorem point of view, they need consistency and availability. However, if the website needs to understand its customer in real time, an analytics team needs to analyze data from the inventory management, account management, and customer management domains. Apart from other data, it might collect separately at real time. The way the analytics team views the same data is totally different from the way other teams view it; for them, consistency is less of a concern, as they are more interested in the overall statistics, and a little inconsistent data will have no impact on the overall report. If all the data required for analytics from these domains is kept in the same data model as that for core business, the analytics will run into difficulty because it has to now work with this highly normalized and optimized structured data for business operations. The analytics team will also like to have their data denormalized for faster analysis.
Now, running real-time analytics on this normalized data on a RDBMS system will require heavy compute resources, which will impact the performance of core business during business hours. So, it is better for overall business if separate data models are created for these domains, one for business and one for analytics, where each is maintained separately as they have separate concerns. We will see in subsequent topics why RDBMS is not the right fit for analytics and some other use cases and how NoSQL solves the problem of explosion of data.

The NoSQL primer

Not only SQL or NoSQL, as it is popularly called, was coined by Carlo Strozzi in 1998 and was reintroduced by Eric Evans in 2009. This is an exciting area in data handling which, in a way, has filled up the many gaps existing in the data handling layer. Before the emergence of NoSQL as an alternate choice to store data, SQL-oriented databases (RDBMS) were the only choice available for the developers to position or retrofit their data. In other words, RDBMS was one hammer to nail all data problems. When NoSQL and its different categories started emerging, data models and data sizes that were not meant for RDBMS started finding NoSQL as a perfect datastore. There was also a shift in attention from a consistency standpoint; there was a shift was from ACID to BASE properties.

ACID properties represent the consistency and availability of the CAP theorem. These properties are exhibited by RDBMS and stand for the following:

- **Atomicity**: In a transaction, all operations will complete or none will be completed (rollback)
- **Consistency**: The database will be in a consistent state during the start and end of a transaction and cannot leave the state in between
- **Isolation**: There will be no interference among the concurrent transactions
- **Durability**: Once a transaction commits, it will remain so even after the server restarts or fails

BASE properties are exhibited by NoSQL; they represent the availability and partition tolerance of the CAP theorem. They basically give up on the strong consistency shown by RDBMS. BASE stands for following features:

- **Basically available**: This guarantees a response to a request even if the data is in the stale state.
Introduction to NoSQL

- **Soft state**: The state of the data is always in a position to accept change even when there is no request to change its state. What this means is that suppose there are two nodes holding the same state of a data (the replication of data), if there is a request to change the state in one of the nodes, the state in the other node will not change during the lifespan of the request. The data in the other node will change its state due to an asynchronous process triggered by the datastore, thus making the state soft.

- **Eventually consistent**: Due to the distributed nature of the nodes, the system will eventually become consistent.

```
  The data write and reads should be faster and easier.
```

Another interesting development took place in the field of software development. Vertical scalability had reached its limit and solutions had to be designed that were horizontally scalable in nature, so the data layer also had to be distributed and partition tolerant. Apart from that social media solution, online gaming and game theory-based websites (where target marketing was done, that is, users are rewarded based on their purchase history with the site. These kind of sites need real-time analytics) started gaining prominence. Social media wanted the synching of huge amount of data from across geographies in the shortest possible time, and the gaming world was interested in high performance. E-commerce sites were interested in knowing about their customers and products in real time, as well as profiling their customers to know their needs before they could realize the need for it. The categories in NoSQL that emerged based on different data models are as follows:

- Graph-oriented NoSQL
- Document-oriented NoSQL
- Key-value oriented NoSQL
- Column-oriented NoSQL

**Graph-oriented NoSQL**

Graph databases are a special kind of NoSQL databases. The data models stored by graph databases are graph structures, which are a bit different from other datastores. A graph structure consists of a node, edges, and properties. The way to understand graph databases is to think of them as mindmaps with bidirectional relationships. What this means is that if A is related to B and B is related to C, then C is related to A. Graph databases tend to solve the problems that arise out of relationships formed among unstructured entities at runtime, which can be bidirectional. As compared to this, RDBMS also has a concept of relationships called **table joins**, but these relationships are on structured data and cannot be bidirectional.
Moreover, these table joins add complexity to the data model with foreign keys and have performance penalties on table join-based queries when the dataset grows over a period time. A few of the most promising graph datastores are Neo4i, FlockDB, OrientDB, and so on.

To understand this better, let's take a sample use case and see how easy it becomes to solve complex graph-based business use cases with graph-oriented NoSQL. The following figure is a sample use case, which an e-commerce website might be interested in solving. The use case is to capture visitors' purchase history and people's relationships in the microblogging component of the website.

Business entities such as the publisher, author, customer, product, and so on are represented as nodes in the graph. Relationships such as authored by, author, publisher, published by, and so on are represented by edges in the graph. Interestingly, a nonbusiness node, such as user-1, which is from the blogging site, can be represented in the graph along with its relationship, follows, with the other node, user-2. By combining the business and nonbusiness entities, the website can find target customers for the products. In the graph, both nodes and edges have properties that are used while running analytics.
The following set of questions can be easily answered by a graph database based on the relationships stored in the systems:

- **Who authored *Learning Redis***?
  Answer: Vinoo Das

- **How are Packt Publishing and *Learning Redis* related?**
  Answer: Publisher

- **Who has their own NoSQL book published by Packt Publishing?**
  Answer: user-2

- **Who is following the customer who has purchased *Learning Redis* and is interested in NoSQL?**
  Answer: user-1

- **List all the NoSQL books that cost less than X USD and that can be bought by the followers of user-2.**
  Answer: *Learning Redis*

## Document-oriented NoSQL

Document-oriented datastores are designed to store data with the philosophy of storing a document. To understand this simplistically, the data here is arranged in the form of a book. A book can be divided into any number of chapters, where each chapter can be divided into any number of topics, and each topic is further divided into subtopics and so on and so forth.

![Composition of a book](image-url)
If the data has a similar structure, that is, it is hierarchical and does not have a fixed depth or schema, then document-oriented datastores are the perfect option to store such data. MongoDB and CouchDB (Couchbase) are two well-known document-oriented datastores that are getting a lot of attention these days. Like a book, which has indexes for faster searches, these datastores also have the indexes of keys stored in memory for faster searches.

Document-oriented datastores have data stored in the XML, JSON, and other formats. They can hold scalar values, maps, lists, and tuples as values. Unlike RDBMS, where the data is viewed as rows of data stored in a tabular form, the data stored here is in a hierarchical tree-like structure where every value stored in these datastores is always associated with a key. Another unique feature is that document-oriented datastores are schema-less. The following screenshot shows an example which shows how the data is stored in document-oriented datastores. The format in which the data is stored is JSON. One of the beauties of document-oriented datastores is that the information can be stored in the way you think of the data. This, in a way, is a paradigm shift from RDBMS, where the data is broken into various smaller parts and then stored in rows and columns in a normalized way.

```

"learning redis": {
    "publisher": "Packt Publisher",
    "author": "Vinod Das",
    "page": 250,
    "tags": ["NoSQL", "Redis", "Packt Publisher", "database"],
    "chapter Info": {
        "title": "Introduction to NoSQL",
        "number of pages": 20
    },
    "title": "Installing Redis",
    "number of pages": 20
}

```

Composition of sample data in JSON format

The two most famous document-oriented stores in use are MongoDB and CouchDB, and it will be interesting to pit them against each other in order to have a better overview.
Salient features of MongoDB and CouchDB

Well, the fact that both MongoDB and CouchDB are document-oriented is established, but both differ in various aspects, which will be of interest to people who want to learn about document-oriented datastores and adopt them in their projects. Following are some features of MongoDB and CouchDB:

- **Insertion of small and large data sets**: Both MongoDB and CouchDB are very good for the insertion of small data sets. MongoDB is a tad better than CouchDB when it comes to the insertion of large data sets. Overall, speed consistencies are very good in both of these document datastores.

- **Random reads**: Both MongoDB and CouchDB are fast when it comes to read speeds. MongoDB is a tad better when it comes to reading large data sets.

- **Fault tolerance**: Both MongoDB and CouchDB have comparable and good fault tolerance capability. CouchDB uses **Erlang/OTP** as the underlying technology platform for its implementation. Erlang is a language and a platform that was developed to make fault-tolerant, scalable, and highly concurrent systems. The fact that Erlang act as a backbone for CouchDB gives it a very good fault-tolerant capability. MongoDB uses C++ as the primary language for its underlying implementation. Industry adoption and its proven track record in the area of fault tolerance give MongoDB a good heads-up in this area.

- **Sharding**: MongoDB has an in-built sharding capability, whereas CouchDB does not. Nevertheless, Couchbase, which is another document datastore built on top of CouchDB, has an automatic sharding capability.

- **Load balancing**: MongoDB and CouchDB have a good load balancing capability. However, since the underlying technology, that is the actor paradigm, in CouchDB has a good provision for load balancing, it can be said that the capability in CouchDB scores over the capability in MongoDB.

- **Multi-data center support**: CouchDB has multi-data center support, whereas MongoDB at the time of researching for this book, didn't have this support. However, I guess that in the future, with the popularity of MongoDB, we can expect it.

- **Scalability**: Both CouchDB and MongoDB are highly scalable.

- **Manageability**: Both CouchDB and MongoDB have good manageability.

- **Client**: CouchDB has JSON for data exchange, whereas MongoDB has BSON, which is proprietary to MongoDB.
**Column-oriented NoSQL**

Column-oriented NoSQL is designed with the philosophy to store data in columns rather than rows. This way to store data is diametrically opposite to the way data is stored in RDBMS, such as in rows. Column-oriented databases are designed from the ground up to be highly scalable and hence, are distributed in nature. They give up on consistency to have this massive scalability.

The following screenshot is a depiction of a small inventory for smart tablets based on our perception; here, the idea is to show how the data is stored in RDBMS as compared to the data stored in a columnar database:

The preceding tabular data is stored in RDBMS in the hard disk, in the format shown here:

<table>
<thead>
<tr>
<th>Product ID</th>
<th>Product Name</th>
<th>Unit Price (INR)</th>
<th>Stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>IPAD AIR</td>
<td>37000</td>
<td>100</td>
</tr>
<tr>
<td>234</td>
<td>SAMSUNG TAB 3</td>
<td>17000</td>
<td>100</td>
</tr>
<tr>
<td>345</td>
<td>KINDLE FIRE</td>
<td>18000</td>
<td>100</td>
</tr>
<tr>
<td>456</td>
<td>NEXUS 7</td>
<td>21000</td>
<td>100</td>
</tr>
<tr>
<td>567</td>
<td>MS SURFACE</td>
<td>25000</td>
<td>100</td>
</tr>
</tbody>
</table>

Presentation of data in columns and rows


The same data in a columnar datastore will be stored as shown in the following figure; here, the data is serialized in columns:

<table>
<thead>
<tr>
<th>Product ID</th>
<th>Product Name</th>
<th>Unit Price (INR)</th>
<th>Stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>IPAD AIR</td>
<td>37000</td>
<td>100</td>
</tr>
<tr>
<td>234</td>
<td>SAMSUNG TAB 3</td>
<td>37000</td>
<td>100</td>
</tr>
<tr>
<td>345</td>
<td>KINDLE FIRE</td>
<td>18000</td>
<td>100</td>
</tr>
<tr>
<td>456</td>
<td>NEXUS 7</td>
<td>21000</td>
<td>100</td>
</tr>
<tr>
<td>567</td>
<td>MS SURFACE</td>
<td>25000</td>
<td>100</td>
</tr>
</tbody>
</table>
A world where vertical scalability is reaching its limit and horizontal scalability is the way organizations want to adopt to store data, columnar datastores are offering solutions that can store petabytes of data in a very cost-effective way. Google, Yahoo!, Facebook, and so on have pioneered the storage of data in a columnar way, and the proof is in the pudding, that is, the amount of data that these companies store is a well-known fact. HBase and Cassandra are a few of the well-known products that are columnar in nature and can store a huge amount of data. Both the datastores are built with eventual consistency in mind. The underlying language in the case of HBase and Cassandra is Java; it will be interesting to put them against each other in order to have a better overview.

**Salient features of HBase and Cassandra**

HBase is a datastore that belongs to the category of columnar-oriented datastores. This datastore came into existence after Hadoop became popular with its HDFS file storage system, inspired from the *Google File System* paper published in 2003. The fact that HBase is based on Hadoop makes it an excellent choice for data warehousing and large-scale data processing and analysis. HBase provides a SQL-type interface over the existing Hadoop ecosystem, which is similar to the way we have been viewing data in a RDBMS, that is row-oriented, but the data is stored in a column-oriented way internally. HBase stores row data against a row key, and it is in a sorted order as per the row key. It has components such as the Region Server, which can be plugged to the DataNode provided with Hadoop. This means that the Region Server is collocated with the DataNode and acts as a gateway for interacting with HBase clients. Behind the scenes, the HBase master handles the DDL operations. Apart from this, it also manages the Region assignments and other bookkeeping activities associated with that. Cluster information and management, which includes state management, is taken care of by Zookeeper nodes. HBase clients interact directly with Region Servers to put and get data. Components such as Zookeeper (used to coordinate between the master and slave nodes), Name Node, and HBase master node do not participate directly in the exchange of data between the HBase client and Region Server nodes.
Cassandra is a datastore which belongs to the category of columnar-oriented datastores and also shows some features of the key-value datastore. Cassandra, which was initially started by Facebook but later forked to the Apache open source community, is best suited for real-time transaction processing and real-time analytics.

One of the key differentiators between Cassandra and HBase is that unlike HBase, which depends on the existing architecture of Hadoop, Cassandra is standalone in nature. Cassandra takes its inspiration from Amazon's Dynamo to store data. In short, the architectural approach of HBase makes the Region Server and DataNodes dependent on other components such as HBase master, Name Node, Zookeeper, whereas the nodes in Cassandra manage these responsibilities within and thus are not dependent on external components.

A Cassandra cluster can be viewed as a ring of nodes, of which there are a few seeds. These seeds are like any node but are responsible for up-to-date cluster state data. In the event of a seed node going down, a new seed can be elected among the available nodes. The data is distributed evenly across the ring, depending on the hash value of the row key. In Cassandra, data can be queried according to its row-key. Clients for Cassandra come in many flavors; that is, Thrift is one of the most native clients that can be used to interact with the Cassandra ring. Apart from this, there are clients that expose the Cassandra Query Language (CQL) interface, which has quite a resemblance to SQL.
• **Insertion of small and large data sets:** Both HBase and Cassandra are very good at the insertion of small data sets. The fact that both these datastores use multiple nodes to distribute writes on top of it. Both of them write the data first to memory-based storage such as RAM, which makes its insertion performance good.

• **Random reads:** Both HBase and Cassandra are fast when it comes to read speeds. In HBase, consistency was one of the key features that was kept in mind when designing the architecture. In Cassandra, data consistency was kept tunable, but one has to sacrifice speed in order to have higher consistency.

• **Eventual consistency:** HBase has strong consistency and Cassandra has eventual consistency, but interestingly, the consistency model in Cassandra is tunable. It can be tuned to have better consistency, but one has to give up performance in the read and write speeds.

• **Load balancing:** HBase and Cassandra have load balancing built into them. The idea is to have many nodes serving read and writes on a commodity grade node. Consistent hashing is used to distribute the load between the nodes.

• **Sharding:** HBase and Cassandra both have sharding capability. This is essential since both claim to give good performance from a commodity grade node, which has limited disk and memory space.

• **Multi-data center support:** Of the two, Cassandra has multi-data center support.
• **Scalability**: HBase and Cassandra have very good scalability, which was one of the design requirements.

• **Manageability**: Of the two, Cassandra has better manageability. This is because in Cassandra, there are nodes to manage but in HBase, there are many components that need to work in tandem, such as Zookeeper, DataNode, Name Node, Region Server, and so on.

• **Client**: Both HBase and Cassandra have clients in Java, Python, Ruby, Node.js, and many more, making it easy to work with heterogeneous environments.

### Key value-oriented NoSQL

Key-value datastores are probably one of the fastest and simplest NoSQL databases. In their most simplistic form, they can be understood as a big hash table. From a usage perspective, every value stored in the database has a key. The key can be used to search for values and the values can be deleted by deleting the key. Some popular choices in key-value databases are Redis, Riak, Amazon’s DynamoDB, project voldemort, and more.

### How does Redis fare in some of the nonfunctional requirements as a key-value datastore?

Redis is one of the fastest key-value stores, which is seeing a very fast adoption throughout the industry, cutting across many domains. Since this book focuses on Redis, let’s find out a bit more about how Redis fares in some of the nonfunctional requirements in brief. We will be talking about them in length as the book progresses:

• **Insertion of data sets**: The insertions of data sets is very fast in key-value datastores and Redis is no exception.

• **Random reads**: Random reads are very fast in key-value datastores. In Redis, all the keys are stored in memory. This ensures faster lookups, so the read speeds are higher. While it will be great if all the keys and values are kept in memory, this has a drawback. The problem with this approach is that memory requirements will be very high. Redis takes care of this by introducing something called *virtual memory*. Virtual memory will keep all the keys in the memory but will write the least recently-used values to disk.

• **Fault tolerance**: Fault handling in Redis depends on the cluster’s topology. Redis uses the master-slave topology for its cluster deployment. All the data in the master is asynchronously copied to the slave; so, in case the master node goes to the failure state, one of the slave nodes can be promoted to master using the Redis sentinel.
• **Eventual consistency**: Key-value datastores have master-slave topology, which means that once the master is updated, all the slave nodes are updated asynchronously. This can be envisaged in Redis since slaves are used by clients for a read-only mode; it is possible that the master might have the latest value written but while reading from the slave, the client might get the stale value because the master has not updated the slaves. Thus, this lag can cause inconsistency for a brief moment.

• **Load balancing**: Redis has a simple way of achieving load balancing. As previously discussed, the master is used to write data, and slaves are used to read the data. So, the clients should have the logic built into them, have the read request evenly spread across the slave nodes, or use third-party proxies, such as Twemproxy to do so.

• **Sharding**: It is possible to have datasets that are bigger than the available memory, which makes presharding the data across various peer nodes a horizontal scalable option.

• **Multi-data center support**: Redis and key-value NoSQL do not provide inherent multi-data center support where the replications are consistent. However, we can have the master node in one data center and slaves in the other data center, but we will have to live with eventual consistency.

• **Scalability**: When it comes to scaling and data partitioning, the Redis server lacks the logic to do so. Primarily, the logic to partition the data across many nodes should reside with the client or should use third-party proxies such as Twemproxy.

• **Manageability**: Redis as a key value NoSQL is simple to manage.

• **Client**: There are clients for Redis in Java, Python, and Node.js that implement the **REdis Serialization Protocol (RESP)**.

### Use cases of NoSQL

Understand your business first; this will help you to understand your data. This will also give you deep insights on the kind of data layer that you need to have. The idea is to have a top-to-bottom design methodology. Deciding on the persistence mechanism first and then fitting the data for the business use case in that persistence mechanism is a bad idea (bottom-to-top design methodology). So, define your business requirements first, decide on the roadmap for the future, and then decide on the data layer. Another important factor to take into consideration when understanding the business requirements specification is to factor the nonfunctional requirements for every business use case, which I believe is paramount.
Failing to add a nonfunctional requirement in the business or, functional requirement causes problems when the system goes to performance test or worse, when it goes live. If you feel that the data model requires NoSQL from a functional requirement standpoint, then ask a few questions as follows:

- What type of NoSQL do you need for the data model?
- How big can the data grow, and how much scalability is required?
- How will you handle node failure? What is its impact on your business use case?
- Which is better data replication or infrastructure investment when data is growing?
- What are the strategies for handling read/write loads and how much concurrency is planned?
- What is the level of data consistency required for the business use case?
- How will the data reside (on a single data center or multiple data centers across geographies)?
- What are the clustering strategies and data synch strategies?
- What are the data backup strategies?
- What kind of network topology do you plan to use? What is the impact of network latency on performance?
- How comfortable is the team in handling, monitoring, administrating, and developing in the polyglot persistence environment?
Here's the summary of some of the NoSQL databases and how they are placed as per the CAP theorem. The following chart does not claim to be exhaustive, but is a snapshot of the most popular ones:

<table>
<thead>
<tr>
<th>NoSQL databases placed as per CAP theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Partition-tolerance</strong></td>
</tr>
<tr>
<td><strong>Consistency</strong></td>
</tr>
<tr>
<td>Document Oriented NoSQL</td>
</tr>
<tr>
<td>Key-Value Oriented NoSQL</td>
</tr>
<tr>
<td>Memcache-DB</td>
</tr>
<tr>
<td>Dynamo</td>
</tr>
<tr>
<td>Graph Oriented NoSQL</td>
</tr>
<tr>
<td>Column Oriented NoSQL</td>
</tr>
<tr>
<td>HBASE</td>
</tr>
<tr>
<td>VERTICA</td>
</tr>
</tbody>
</table>

Let's analyze how companies are using NoSQL, which will give us ideas on how we can use NoSQL in our solutions effectively:

- **Big data**: This very term evokes a picture of hundreds and thousands of servers crunching petabytes of data for analysis. The use case for big data is self-evident and simple to argue for using NoSQL datastores. Columnar databases, one of the patterns of NoSQL, are the obvious choice for this kind of activity. Being distributed in nature, these solutions also have no single point of failure, parallel computing, write availability, and scalability. The following is a sample list of the different types of use cases where companies have successfully used columnar datastores in their business:

  - Spotify uses Hadoop for data aggregation, reporting, and analysis
  - Twitter uses Hadoop to process tweets and log files
  - Netflix uses Cassandra for their backend datastore in order to stream services
Zoho uses Cassandra to generate inbox previews for mail services
Facebook uses Cassandra for its Instagram operations
Facebook uses HBase in its message infrastructure
Su.pr uses HBase for real-time data storage and the analytics platform
HP IceWall SSO uses HBase to store user data in order to authenticate users for their web-based single sign-on solution

Heavy read/write: This nonfunctional requirement instantly gives us the impression of a social or a gaming website. For Enterprises where this is a requirement, they can take inspiration for the choice of NoSQL.

LinkedIn uses Voldemort (the key-value datastore) to cater to millions of read and writes per day under a few milliseconds
Wooga (a social network game and mobile developer) uses Redis for its gaming platform; some of the games have a million plus users in a day
Twitter caters to 200 million tweets a day and uses NoSQL, such as Cassandra, HBase, Memcached, and FlockDB, and also uses RDBMS, such as MySQL
Stack overflow uses Redis to cater to 30 million registered users in a month

Document store: The growth of Web 2.0 adoption and the rise in Internet content is creating data that is schema-less in nature. Having NoSQL (document-oriented) specially designed to store this kind of data makes the job of a developer simpler and the solution more stable in nature. Following are the examples of some companies that use different document stores:

SourceForge uses MongoDB to store front pages, project pages, and download pages; Allura on SourceForge is based on MongoDB
MetLife uses MongoDB for datastore for the wall, a customer service platform
Semantic News Portal uses CouchDB to store news data
Vermont public radio website's homepage uses CouchDB to stores news headlines, commentaries and more
AOL advertising uses Couchbase (a new avatar of CouchDB) to serve billions of impressions a month for 100 million plus users
• **Real-time experience and e-commerce platform**: Shopping carts, user profile management, voting, user session management, real-time page counters, real-time analytics, and more are the services that are being offered by companies to give real-time experience to the end user. Following are the examples of some companies that use real-time experience and e-commerce platform:
  - Flickr push uses Redis to push real-time updates
  - Instagram uses Redis to store hundreds and millions of media content against keys and to serve them in real time
  - Digg uses Redis for its page views and user clicks solution
  - Best Buy uses Riak for its e-commerce platform

**Summary**
In this chapter, you saw how the Internet world is undergoing a paradigm shift, the evolution of the NoSQL world, and how social media is championing NoSQL adoption. You also saw the various alternatives in the NoSQL world and how they equate. Finally, you saw how Redis maps up in the NoSQL ecosystem.

In the next chapter, we will take a plunge into the world of Redis.
Where to buy this book
You can buy Learning Redis from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.