6.4 Understanding the types of big data problems

There are many types of big data problems, each requiring a different combination of NoSQL systems. After you've categorized your data and determined its type, you'll find there are different solutions. How you build your own big data classification system might be different from this example, but the process of differentiating data types should be similar.

Figure 6.5 is a good example of a high-level big data classification system.

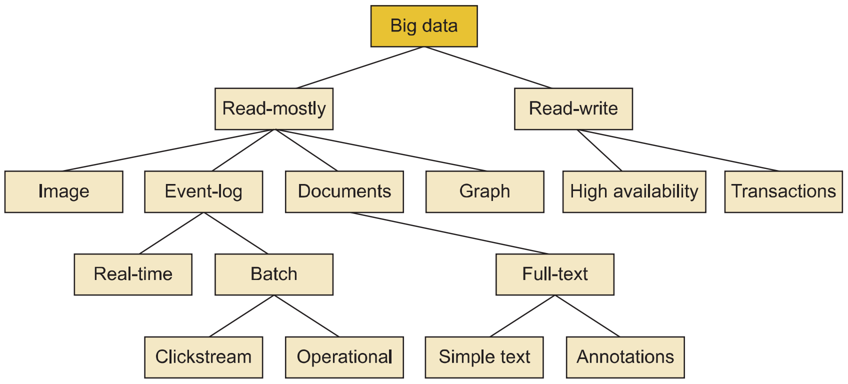


Figure 6.5 A sample of a taxonomy of big data types. This chapter deals with read-mostly problems. Chapter 8 has a focus on read/write big data problems that need high availability.

Let's take a look at some ways you classify big data problems and see how NoSQL systems are changing the way organizations use data.

- *Read-mostly*—Read-mostly data is the most common classification. It includes data that's created once and rarely altered. This type of data is typically found in data warehouse applications but is also identified as a set of non-RDBMS items like images or video, event-logging data, published documents, or graph data. Event data includes things like retail sales events, hits on a website, system logging data, or real-time sensor data.
- *Log events*—When operational events occur in your enterprise, you can record it in a log file and include a timestamp so you know when the event occurred. Log events may be a web page click or an out-of-memory warning on a disk drive. In the past, the cost and amount of event data produced were so large that many organizations opted not to gather or analyze it. Today, NoSQL systems are changing companies' thoughts on the value of log data as the cost to store and analyze it is more affordable.

  The ability to cost-effectively gather and store log events from all computers in your enterprise has lead to BI *operational intelligence* systems. Operational intelligence goes beyond analyzing trends in your web traffic or retail transactions. It can integrate information from network monitoring systems so you can detect problems before they impact your customers. Cost-effective NoSQL systems can be part of good operations management solutions.

- *Full-text documents*—This category of data includes any document that contains natural-language text like the English language. An important aspect of document stores is that you can query the entire contents of your office document in the same way you would query rows in your SQL system.

  This means that you can create new reports that combine traditional data in RDBMSs as well as the data within your office documents. For example, you could create a single query that extracted all the authors of titles of PowerPoint slides that contained the keywords *NoSQL* or *big data*. The result of this list of authors could then be filtered with a list of titles in the HR database to show which people had the title of *Data Architect* or *Solution Architect*.

  This is a good example of how organizations are trying to tap into the hidden skills that already exist within an organization for training and mentorship. Integrating documents into what can be queried is opening new doors in knowledge management and efficient staff utilization.

As you can see, you might encounter many different flavors of big data. As we move forward, you'll see how using a shared-nothing architecture can help you with most of your big data problems, whether they're read-mostly or read/write data.

6.5 Analyzing big data with a shared-nothing architecture

There are three ways that resources can be shared between computer systems: shared RAM, shared disk, and shared-nothing. Figure 6.6 shows a comparison of these three distributed computing architectures.

Of the three alternatives, a shared-nothing architecture is most cost effective in terms of cost per processor when you're using commodity hardware. As we continue,

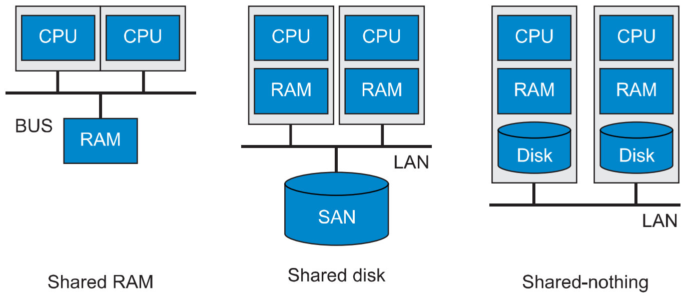


Figure 6.6 Three ways to share resources. The left panel shows a shared RAM architecture, where many CPUs access a single shared RAM over a high-speed bus. This system is ideal for large graph traversal. The middle panel shows a shared disk system, where processors have independent RAM but share disk using a storage area network (SAN). The right panel shows an architecture used in big data solutions: cache-friendly, using low-cost commodity hardware, and a shared-nothing architecture.

you'll see how each of these architectures works to solve big data problems with different types of data.

Of the architectural data patterns we've discussed so far (row store, key-value store, graph store, document store, and Bigtable store), only two (key-value store and document store) lend themselves to cache-friendliness. Bigtable stores scale well on shared-nothing architectures because their row-column identifiers are similar to key-value stores. But row stores and graph stores aren't cache-friendly since they don't allow a large BLOB to be referenced by a short key that can be stored in the cache.

For graph traversals to be fast, the entire graph should be in main memory. This is why graph stores work most efficiently when you have enough RAM to hold the graph. If you can't keep your graph in RAM, graph stores will try to swap the data to disk, which will decrease graph query performance by a factor of 1,000. The only way to combat the problem is to move to a shared-memory architecture, where multiple threads all access a large RAM structure without the graph data moving outside of the shared RAM.

The rule of thumb is if you have over a terabyte of highly connected graph data and you need real-time analysis of this graph, you should be looking for an alternative to a shared-nothing architecture. A single CPU with 64 GB of RAM won't be sufficient to hold your graph in RAM. Even if you work hard to only load the necessary data elements into RAM, your links may traverse other nodes that need to be swapped in from disk. This will make your graph queries slow. We'll look into alternatives to this in a case study later in this chapter.

Knowing the hardware options available to big data is an important first step, but distributing software in a cluster is also important. Let's take a look at how software can be distributed in a cluster.

6.6 Choosing distribution models: master-slave versus peer-to-peer

From a distribution perspective, there are two main models: master-slave and peer-to-peer. Distribution models determine the responsibility for processing data when a request is made.

Understanding the pros and cons of each distribution model is important when you're looking at a potential big data solution. Peer-to-peer models may be more resilient to failure than master-slave models. Some master-slave distribution models have single points of failure that might impact your system availability, so you might need to take special care when configuring these systems.

Distribution models get to the heart of the question *who's in charge here?* There are two ways to answer this question: one node or all nodes. In the master-slave model, one node is in charge (master). When there's no single node with a special role in taking charge, you have a peer-to-peer distribution model.

Figure 6.7 shows how these models each work.

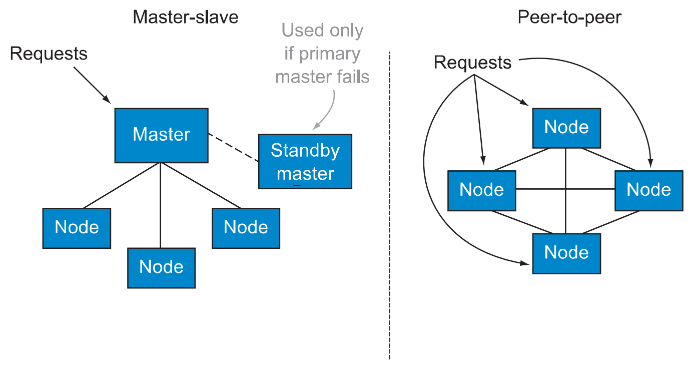


Figure 6.7 Master-slave versus peer-to-peer—the panel on the left illustrates a master-slave configuration where all incoming database requests (reads or writes) are sent to a single master node and redistributed from there. The master node is called the NameNode in Hadoop. This node keeps a database of all the other nodes in the cluster and the rules for distributing requests to each node. The panel on the right shows how the peer-to-peer model stores all the information about the cluster on each node in the cluster. If any node crashes, the other nodes can take over and processing can continue.

Let's look at the trade-offs. With a master-slave distribution model, the role of managing the cluster is done on a single master node. This node can run on specialized hardware such as RAID drives to lower the probability that it crashes. The cluster can also be configured with a standby master that's continually updated from the master node. The challenge with this option is that it's difficult to test the standby master without jeopardizing the health of the cluster. Failure of the standby master to take over from the master node is a real concern for high-availability operations.

Peer-to-peer systems distribute the responsibility of the master to each node in the cluster. In this situation, testing is much easier since you can remove any node in the cluster and the other nodes will continue to function. The disadvantage of peer-to-peer networks is that there's an increased complexity and communication overhead that must occur for all nodes to be kept up to date with the cluster status.

The initial versions of Hadoop (frequently referred to as the 1.x versions) were designed to use a master-slave architecture with the NameNode of a cluster being responsible for managing the status of the cluster. NameNodes usually don't deal with any MapReduce data themselves. Their job is to manage and distribute queries to the correct nodes on the cluster. Hadoop 2.x versions are designed to remove single points of failure from a Hadoop cluster.

Using the right distribution model will depend on your business requirements: if high availability is a concern, a peer-to-peer network might be the best solution. If you can manage your big data using batch jobs that run in off hours, then the simpler master-slave model might be best. As we move to the next section, you'll see how MapReduce systems can be used in multiprocessor configurations to process your big data.

6.7 Using MapReduce to transform your data over distributed systems

Now let's take an in-depth look to see how MapReduce systems can be used to process large datasets on multiple processors. You'll see how MapReduce clusters work in conjunction with distributed filesystems such as the Apache Hadoop Distributed File System (HDFS) and review how NoSQL systems such as Hadoop use both the map and reduce functions to transform data that's stored in NoSQL databases.

If you've been moving data between SQL systems, you're familiar with the extract, load, and transform (ETL) process. The ETL process is typically used when extracting data from an operational RDBMS to transfer it into the staging area of a data warehouse. We reviewed this process and ETL in chapter 3 when we covered OLAP systems.

ETL systems are typically written in SQL. They use the SELECT statement on a source system and INSERT, UPDATE, or DELETE functions on the destination system. SQL-based ETL systems usually don't have the inherent ability to use a large number of processors to do their work. This single-processor bottleneck is common in data warehouse systems as well as areas of big data.

To solve this type of problem, organizations have moved to a distributed transformation model built around the map and reduce functions. To effectively distribute work evenly over a cluster of processors, the output of a map phase must be a set of key-value pairs where part of the key structure is used to correlate results into the reduce phase. These functions are designed to be inherently linearly scalable using a large number of shared-nothing processors.

Yet, at its core, the fundamental process of MapReduce is the parallel transformation of data from one form to another. MapReduce processes don't require the use of databases in the middle of the transformation. To work effectively on big data problems, MapReduce operations do require a large amount of input and output. In an ideal situation, data transformed by a MapReduce server will have all the input on the local disk of a shared-nothing cluster and write the results to the same local disk. Moving large datasets in and out of a MapReduce cluster can be inefficient.

The MapReduce way of processing data is to specify a series of stepwise functions on uniform input data. This process is similar to the functional programming constructs that became popular in the 1950s with the LISP systems at MIT. Functional programming is about taking a function and a list, and returning a list where the function has been applied to each member of the list. What's different about modern MapReduce is all the infrastructure that goes with the reliable and efficient execution of transforms on lists of billions of items. The most popular implementation of the MapReduce algorithm is the Apache Hadoop system.

The Hadoop system doesn't fundamentally change the concepts of mapping and reducing functions. What it does is provide an entire ecosystem of tools to allow map and reduce functions to have linear scalability. It does this by requiring that the output of all map functions return a key-value pair. This is how work can then be distributed evenly over the nodes of a Hadoop cluster. Hadoop addresses all of the hard parts of distributed computing for large datasets for you so that you can concentrate on writing the map and reduce operations.

It's also useful to contrast the MapReduce way with RDBMSs. MapReduce is a way of explicitly specifying the steps in a transformation, with the key being to use key-value pairs as a method of distribution to different nodes in a cluster. SQL, on the other hand, attempts to shield you from the process steps it uses to get data from different tables to perform optimal queries.

If you're using Hadoop, MapReduce is a disk-based, batch-oriented process. All input comes from disk, and all output writes to disk. Unlike MapReduce, the results of SQL queries can be loaded directly into RAM. As a result, you'd rarely use the result of a MapReduce operation to populate a web page while users are waiting for a web page to render.

The Hadoop MapReduce process is most similar to the data warehouse process of precalculating sums and totals in an OLAP data warehouse. This process is traditionally done by extracting new transactions each night from an operational data store and converting them to facts and aggregates in an OLAP cube. These aggregates allow sums and totals to be quickly calculated when users are looking for trends in purchasing decisions.

NoSQL systems each vary in how they implement map and reduce functions and how they integrate with existing Hadoop clusters. Some NoSQL systems such as *HBase* are designed to run directly within a Hadoop system. Their default behavior is to read from HDFS and write the results of their transforms to HDFS. By taking this approach, HBase can leverage existing Hadoop infrastructure and optimize input and output processing steps.

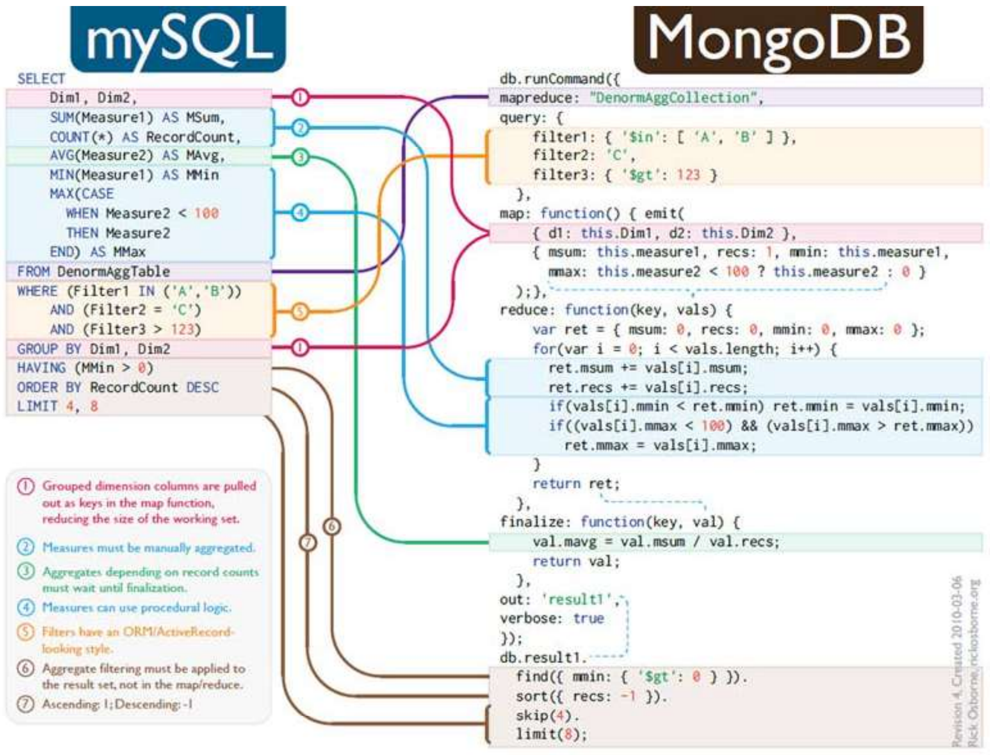
Most other NoSQL systems that target big data problems provide their own ways to perform map and reduce functions or to integrate with a Hadoop cluster. For example, MongoDB provides their own map and reduce operations that work directly on MongoDB documents for both input and output. Figure 6.8 shows an example comparing MongoDB map reduce functions with SQL.

Now that you have an understanding of how MapReduce leverages many processors, let's see how it interacts with the underlying filesystems.

6.7.1 MapReduce and distributed filesystems

One of the strengths of a Hadoop system is that it's designed to work directly with a filesystem that supports big data problems. As you'll see, Hadoop makes big data processing easier by using a filesystem structure that's different from a traditional system.

The Hadoop Distributed File System (HDFS) provides many of the supporting features that MapReduce transforms need to be efficient and reliable. Unlike an ordinary filesystem, it's customized for transparent, reliable, write-once, read-many operations. You can think of HDFS as a fault-tolerant, distributed, key-value store tuned to work with large files.



```
SELECT
    Dim1, Dim2,
    SUM(Measure1) AS MSum,
    COUNT(*) AS RecordCount,
    AVG(Measure2) AS MAvg,
    MIN(Measure1) AS MMin
    MAX(CASE
      WHEN Measure2 < 100
      THEN Measure2
    END) AS MMax
FROM DenormAggTable
WHERE (Filter1 IN ('A','B'))
    AND (Filter2 = 'C')
    AND (Filter3 > 123)
GROUP BY Dim1, Dim2
HAVING (MMin > 0)
ORDER BY RecordCount DESC
LIMIT 4, 8
```

```
db.runCommand({
mapreduce: "DenormAggCollection",
query: {
    filter1: { '$in': [ 'A', 'B' ] },
    filter2: 'C',
    filter3: { '$gt': 123 }
  },
map: function() { emit(
    { d1: this.Dim1, d2: this.Dim2 },
    { msum: this.measure1, recs: 1, mmin: this.measure1,
      mmax: this.measure2 < 100 ? this.measure2 : 0 }
  );},
reduce: function(key, vals) {
    var ret = { msum: 0, recs: 0, mmin: 0, mmax: 0 };
    for(var i = 0; i < vals.length; i++) {
      ret.msum += vals[i].msum;
      ret.recs += vals[i].recs;
      if(vals[i].mmin < ret.mmin) ret.mmin = vals[i].mmin;
      if((vals[i].mmax < 100) && (vals[i].mmax > ret.mmax))
        ret.mmax = vals[i].mmax;
    }
    return ret;
  },
finalize: function(key, val) {
    val.mavg = val.msum / val.recs;
    return val;
  },
out: 'result1',
verbose: true
});
db.result1.
  find({ mmin: { '$gt': 0 } }).
  sort({ recs: -1 }).
  skip(4).
  limit(8);
```

1. Grouped dimension columns are pulled out as keys in the map function, reducing the size of the working set.
2. Measures must be manually aggregated.
3. Aggregates depending on record counts must wait until finalization.
4. Measures can use procedural logic.
5. Filters have an ORM/ActiveRecord-looking style.
6. Aggregate filtering must be applied to the result set, not in the map/reduce.
7. Ascending: 1; Descending: -1

Figure 6.8 A comparison of a mySQL SQL query with MongoDB's map and reduce functions. The queries perform similar functions, but the MongoDB query can easily be distributed over hundreds of processors. (Attribution to Rick Osborne)

Traditional filesystems store data in a single location; if a drive fails, data is restored from a backup drive. By default, files in HDFS are stored in three locations; if a drive fails, the data is automatically replicated to another drive. It's possible to get the same functionality using a fault-tolerant system like RAID drives. But RAID drives are more expensive and difficult to configure on commodity hardware.

HDFSs are different: they use a large (64 megabytes by default) block size to handle data. Figure 6.9 shows how large HDFS blocks are compared to a typical operating system.

HDFS also has other properties that make it different from an ordinary filesystem. You can't update the value of a few bytes in an existing block without deleting the old block and adding an entirely new block. HDFS is designed for large blocks of immutable data that are created once and read many times. Efficient updates aren't a primary consideration for HDFS.

Although HDFS is considered a filesystem, and can be mounted like other filesystems, you don't usually work with HDFS as you would an additional disk drive on your Windows or UNIX system. An HDFS system wouldn't be a good choice for storing typical Microsoft office documents that are updated frequently. HDFS is designed to be a

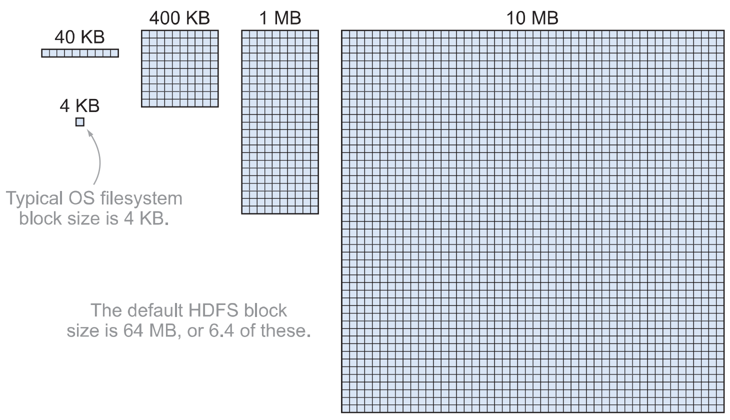

40 KB
400 KB
1 MB
10 MB
4 KB
Typical OS filesystem block size is 4 KB.
The default HDFS block size is 64 MB, or 6.4 of these.

Figure 6.9 The size difference between a filesystem block size on a typical desktop or UNIX operating system (4 KB) and the logical block size within the Apache Hadoop Distributed File System (64 MB), which is optimized for big data transforms. The default block size defines a unit of work for the filesystem. The fewer blocks used in a transfer, the more efficient the transfer process. The downside of using large blocks is that if data doesn't fill an entire physical block, the empty section of the block can't be used.

highly available input or output destination for gigabyte and larger MapReduce batch jobs.

Now let's take a closer look at how MapReduce jobs work over distributed clusters.

6.7.2 How MapReduce allows efficient transformation of big data problems

In previous chapters, we looked at MapReduce and its exceptional horizontal scale-out properties. MapReduce is a core component in many big data solutions. Figure 6.10 provides a detailed look at the internal components of a MapReduce job.

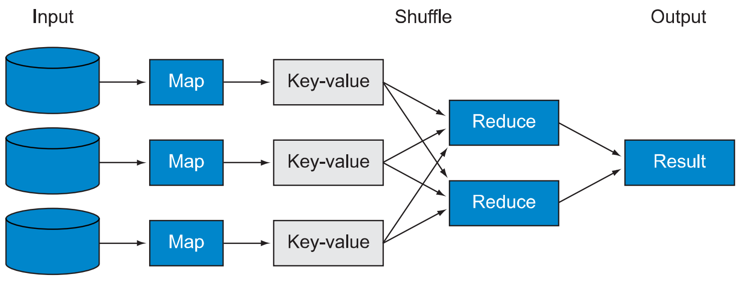


Figure 6.10 The basics of how the map and reduce functions work together to gain linear scalability over big data transforms. The map operation takes input data and creates a uniform set of key-value pairs. In the shuffle phase, which is done automatically by the MapReduce framework, key-value pairs are automatically distributed to the correct reduce node based on the value of the key. The reduce operation takes the key-value pairs and returns consolidated values for each key. It's the job of the MapReduce framework to get the right keys to the right reduce nodes.

The first part of a MapReduce job is the map operation. Map operations retrieve data from your source database and convert it into a series of independent transform operations that can be executed on different processors. The output of all map operations is a key-value structure where the keys are uniform across all input documents. The second phase is the reduce operation. The reduce operation uses the key-value pairs created in the map as input, performs the requested operation, and returns the values you need.

When creating a MapReduce program, you must ensure that the map function is only dependent on the inputs to the map function and that the output of the map operation doesn't change the state of data; it only returns a key-value pair. In MapReduce operations, no other intermediate information can be passed between map functions.

At first glance, it may seem like creating a MapReduce framework would be simple. Realistically, it's not. First, what if your source data is replicated on three or more nodes? Do you move the data between nodes? Not if you want your job to be efficient. Then you must consider which node the map function should run on. How do you assign the right key to the right reduce processor? What happens if one of the map or reduce jobs fails in mid-operation? Do you need to restart the entire batch or can you reassign the work to another node? As you can see, there are many factors to consider and in the end it's not as simple as it appears.

The good news is that if you stick to these rules, a MapReduce framework like Hadoop can do most of the hard work finding the right processor to do the map, making sure the right reduce node gets the input based on the keys, and making sure that the job finishes even if there's hardware failure during the job.

Now that we've covered the types of big data problems and some of the architecture patterns, let's look into the strategies that NoSQL systems use to attack these problems.

6.8 Four ways that NoSQL systems handle big data problems

As you've seen, understanding your big data is important in determining the best solution. Now let's take a look at four of the most popular ways NoSQL systems handle big data challenges.

Understanding these techniques is important when you're evaluating any NoSQL system. Knowing that a product will give you linear scaling with these techniques will help you not only to select the right NoSQL system, but also to set up and configure your NoSQL system correctly.

6.8.1 Moving queries to the data, not data to the queries

With the exception of large graph databases, most NoSQL systems use commodity processors that each hold a subset of the data on their local shared-nothing drives. When a client wants to send a general query to all nodes that hold data, it's more efficient to send the query to each node than it is to transfer large datasets to a central processor. This may seem obvious, but it's amazing how many traditional databases still can't distribute queries and aggregate query results.

This simple rule helps you understand how NoSQL databases can have dramatic performance advantages over systems that weren't designed to distribute queries to the data nodes. Consider an RDBMS that has tables distributed over two different nodes. In order for the SQL query to work, information about rows on one table must all be moved across the network to the other node. Larger tables result in more data movement, which results in slower queries. Think of all the steps involved. The tables can be extracted, serialized, sent through the network interface, transmitted over networks, reassembled, and then compared on the server with the SQL query.

Keeping all the data within each data node in the form of logical documents means that only the query itself and the final result need to be moved over a network. This keeps your big data queries fast.

6.8.2 Using hash rings to evenly distribute data on a cluster

One of the most challenging problems with distributed databases is figuring out a consistent way of assigning a document to a processing node. Using a *hash ring* technique to evenly distribute big data loads over many servers with a randomly generated 40-character key is a good way to evenly distribute a network load.

Hash rings are common in big data solutions because they consistently determine how to assign a piece of data to a specific processor. Hash rings take the leading bits of a document's hash value and use this to determine which node the document should be assigned. This allows any node in a cluster to know what node the data lives on and how to adapt to new assignment methods as your data grows. Partitioning keys into ranges and assigning different key ranges to specific nodes is known as *keyspace management*. Most NoSQL systems, including MapReduce, use keyspace concepts to manage distributed computing problems.

In chapters 3 and 4 we reviewed the concept of hashing, consistent hashing, and key-value stores. A hash ring uses these same concepts to assign an item of data to a specific node in a NoSQL database cluster. Figure 6.11 is a diagram of a sample hash ring with four nodes.

As you can see from the figure, each input will be assigned to a node based on the 40-character random key. One or more nodes in your cluster will be responsible for storing this key-to-node mapping algorithm. As your database grows, you'll update the algorithm so that each new node will also be assigned some range of key values. The algorithm also needs to replicate items with these ranges from the old nodes to the new nodes.

The concept of a hash ring can also be extended to include the requirement that an item must be stored on multiple nodes. When a new item is created, the hash ring rules might indicate both a primary and a secondary copy of where an item is stored. If the node that contains the primary fails, the system can look up the node where the secondary item is stored.

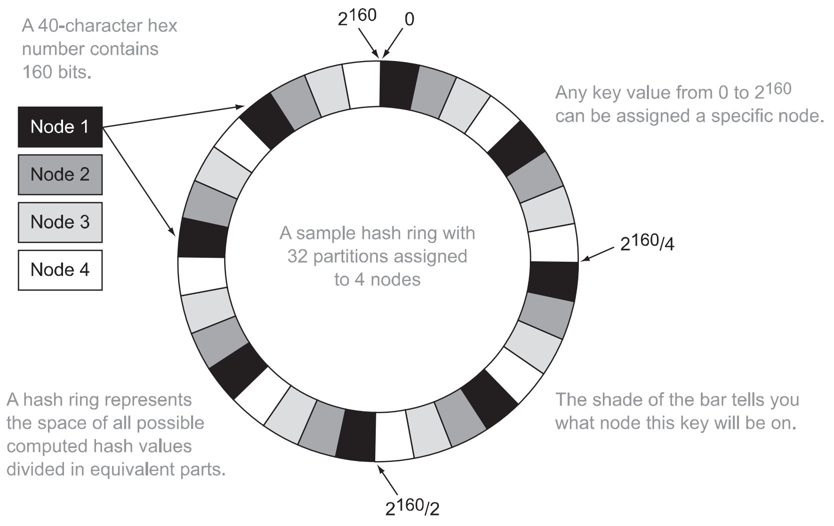

A 40-character hex number contains 160 bits.
Node 1
Node 2
Node 3
Node 4
2160
0
Any key value from 0 to 2160 can be assigned a specific node.
A sample hash ring with 32 partitions assigned to 4 nodes
2160/4
A hash ring represents the space of all possible computed hash values divided in equivalent parts.
The shade of the bar tells you what node this key will be on.
2160/2

Figure 6.11 Using a hash ring to assign a node to a key that uses a 40-character hex number. This number can be expressed in 2^{160} bits. The first bits in the hash can be used to map a document directly to a node. This allows documents to be randomly assigned to nodes and new assignment rules to be updated as you add nodes to your cluster.

6.8.3 *Using replication to scale reads*

In chapter 3 we showed how databases use replication to make backup copies of data in real time. We also showed how load balancers can work with the application layer to distribute queries to the correct database server. Now let's look at how using replication allows you to horizontally scale read requests. Figure 6.12 shows this structure.

This replication strategy works well in most cases. There are only a few times when you must be concerned about the lag time between a write to the read/write node and a client reading that same record from a replica. One of the most common operations after a write is a read of that same record. If a client does a write and then an immediate read from that same node, there's no problem. The problem occurs if a read

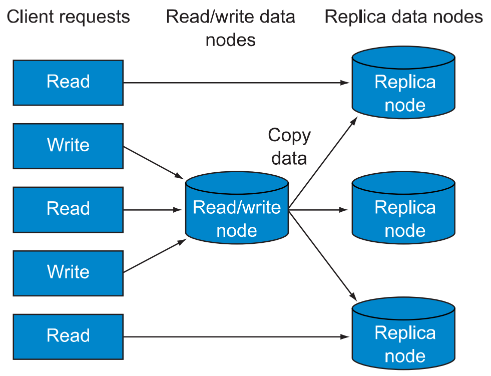


Figure 6.12 How you can replicate data to speed read performance in NoSQL systems. All incoming client requests enter from the left. All reads can be directed to any node, either a primary read/write node or a replica node. All write transactions can be sent to a central read/write node that will update the data and then automatically send the updates to replica nodes. The time between the write to the primary and the time the update arrives on the replica nodes determines how long it takes for reads to return consistent results.

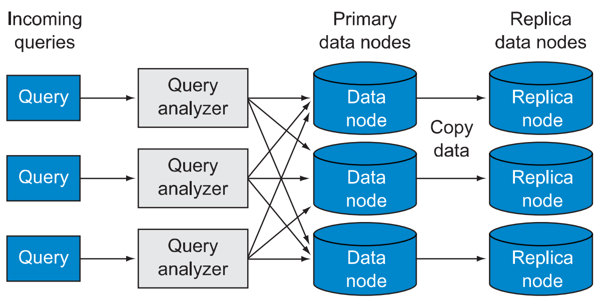


Figure 6.13 NoSQL systems move the query to a data node, but don't move data to a query node. In this example, all incoming queries arrive at query analyzer nodes. These nodes then forward the queries to each data node. If they have matches, the documents are returned to the query node. The query won't return until all data nodes (or a response from a replica) have responded to the original query request. If the data node is down, a query can be redirected to a replica of the data node.

occurs from a replica node before the update happens. This is an example of an inconsistent read.

The best way to avoid this type of problem is to only allow reads to the same write node after a write has been done. This logic can be added to a session or state management system at the application layer. Almost all distributed databases relax database consistency rules when a large number of nodes permit writes. If your application needs fast read/write consistency, you must deal with it at the application layer.

6.8.4 Letting the database distribute queries evenly to data nodes

In order to get high performance from queries that span multiple nodes, it's important to separate the concerns of query evaluation from query execution. Figure 6.13 shows this structure.

The approach shown in figure 6.13 is one of moving the query to the data rather than moving the data to the query. This is an important part of NoSQL big data strategies. In this instance, moving the query is handled by the database server, and distribution of the query and waiting for all nodes to respond is the sole responsibility of the database, not the application layer.

This approach is somewhat similar to the concept of *federated search*. Federated search takes a single query and distributes it to distinct servers and then combines the results together to give the user the impression they're searching a single system. In some cases, these servers may be in different geographic regions. In this case, you're sending your query to a single cluster that's not only performing search queries on a single local cluster but also performing update and delete operations.

6.9 Case study: event log processing with Apache Flume

In this case study, you'll see how organizations use NoSQL systems to gather and report on distributed *event logs*. Many organizations use NoSQL systems to process their event log data because the datasets can be large, especially in distributed environments. As you can imagine, each server generates hundreds of thousands of event