PART II

The Gory Details

CHAPTER 8

Distributed Database Patterns

The manager is to be blamed who distributes parts to his players which they are unable to act.

—Franz Schubert

The speed of communications is wondrous to behold. It is also true that speed can multiply the distribution of information that we know to be untrue.

—Edward R. Murrow

Administrators of web applications have traditionally had two choices when the application demand exceeds database capacity: *scaling up* by increasing the power of individual servers, or *scaling out* by adding more servers. For most of the relational database era, scaling up was the more practical option. Early relational databases did not provide a clustering option, whereas the CPU and memory supplied by a single server was constantly and exponentially increasing in line with Moore's law. Consequently, scaling out was neither practical nor necessary.

However, as database workloads shifted from client-server applications running behind the firewall to web applications with potentially global scope, it became increasingly difficult to support workload and availability requirements on a single server. Furthermore, Internet applications were often subject to unpredictable and massive growth in workload: it became possible for an application to "go viral" and to suddenly experience exponential growth in demand. The economic sweet spot for computer hardware and the imperatives of growth increasingly encouraged clusters of servers rather than single, monolithic proprietary servers. A scale-out solution for databases became imperative.

To address the demands of web-scale applications, a new generation of distributed nonrelational databases emerged. In this chapter, we'll dive deep into the architectures of these distributed database systems.

Distributed Relational Databases

The first database systems were designed to run on a single computer. Indeed, prior to the client-server revolution, all components of early database applications—including all the application code—would reside on a single system: the *mainframe*.

In this centralized model, all program code runs on the server, and users communicate with the application code through dumb terminals (the terminals are "dumb" because they contain no application code). In the client-server model, presentation and business logic were implemented on workstations—usually Windows PCs—that communicated with a single back-end database server. In early Internet applications, business logic was implemented on one or more web application servers, while presentation logic was shared between the web browser and the application server, which still almost always communicated with a single database server.

Figure 8-1 Illustrates the three architectures, showing how each pattern continued to rely on a single, monolithic database server.

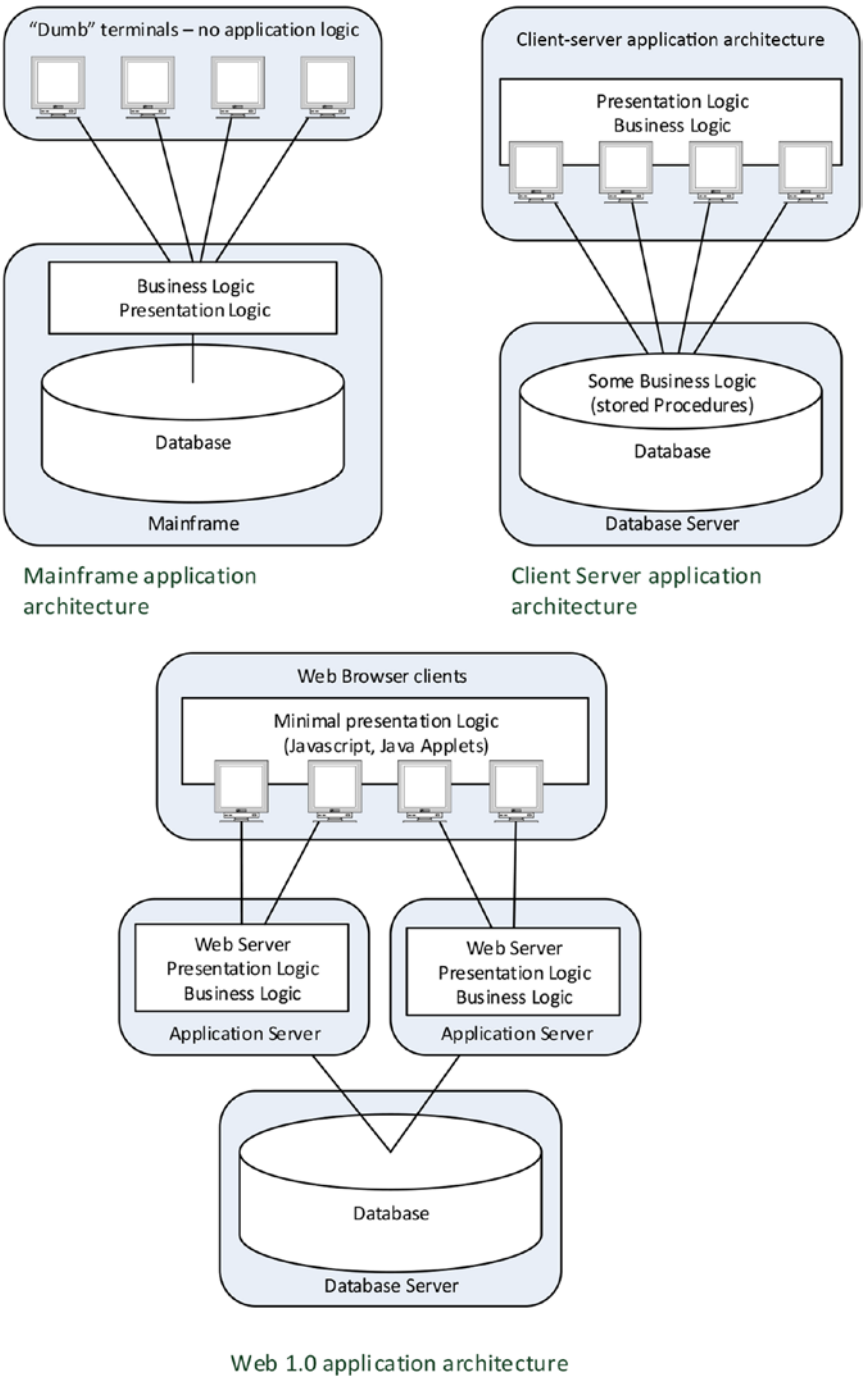


Figure 8-1. *Mainframe, client-server, and early web architectures relied on single, monolithic database servers*

Replication

Database replication was initially adopted as a means of achieving high availability. Using replication, database administrators could configure a *standby database* that could take over for the primary database in the event of failure.

Database replication often took advantage of the *transaction log* that most relational databases used to support ACID transactions. We introduced the transaction log pattern in the context of in-memory databases in Chapter 7. When a transaction commits in an ACID-compliant database, the transaction record is immediately written to the transaction log so that it is preserved in the event of failure. A replication process monitoring the transaction log can apply changes to a backup database, thereby creating a replica.

Figure 8-2 illustrates the log-based replication approach. Database transactions are written in an asynchronous "lazy" manner to the database files (1), but a database transaction immediately writes to the transaction log upon commit (2). The replication process monitors the transaction log and applies transactions as they are written to the read-only slave database (3). Replication is usually asynchronous, but in some databases the commit can be deferred until the transaction has been replicated to the slave.

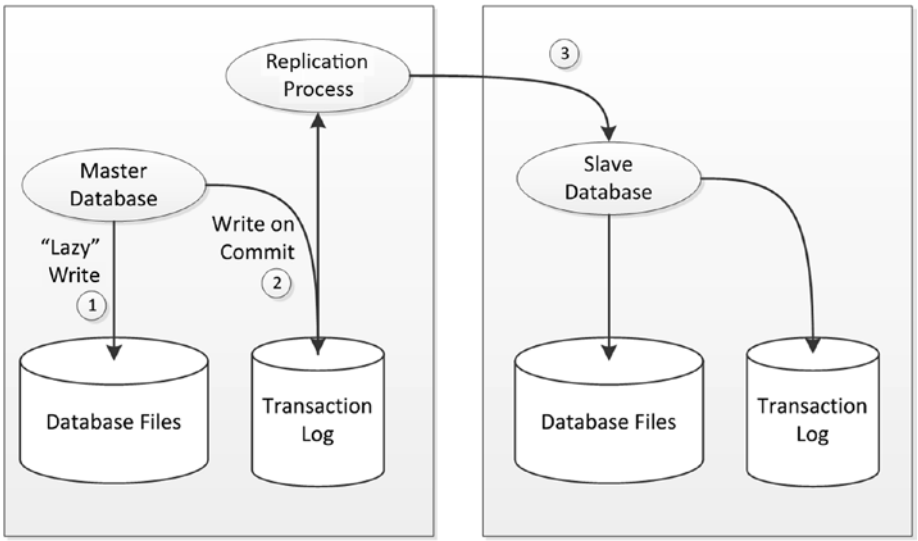


Figure 8-2. *Log-based replication*

As we saw in Chapter 3, replication is typically a first step toward distributing the database load across multiple servers. Using replication, the read workload can be distributed in a scale-out fashion, although database transactions must still be applied to the master copy.

Shared Nothing and Shared Disk

The replication pattern for distributing database workloads works well to distribute read activity across multiple servers, but it does not distribute transactional write loads, which still must be directed exclusively to the master server.

Replication is also of limited value for distributing data warehousing workloads. An OLTP workload typically consists of large numbers of short-duration requests. However, in a data warehousing environment, the workload usually consists of smaller numbers of data-intensive queries. In high-end database servers, these massive queries are executed by multiple processes or threads, each of which can leverage a separate CPU core and take advantage of multiple IO channels.

Parallelizing a query across multiple database servers requires a new approach. Data warehousing vendors provided a solution to this problem by implementing a *shared-nothing* clustered database architecture. Like so many concepts in the relational world, the shared-nothing idea was most notably outlined by Michael Stonebraker in the 1980s. A database server may be classified as:

- **Shared-everything**: In this case, every database process shares the same memory, CPU, and disk resources. Sharing memory implies that every process is on the same server and hence this architecture is a single-node database architecture.
- **Shared-disk**: In this case, database processes may exist on separate nodes in the cluster and have access to the CPU and memory of the server on which they reside. However, every process has equal access to disk devices, which are shared across all nodes of the cluster.
- **Shared-nothing**: In this case, each node in the cluster has access not only to its own memory and CPU but also to dedicated disk devices and its own subset of the database. We've seen several examples of shared-nothing architecture in this book already, including the sharded MySQL design in Chapter 3 and the VoltDB partitioning scheme in Chapter 7.

The shared-nothing model became the basis for several early clustered database systems, such as Teradata. It provides an attractive model for data warehousing workloads because queries can easily be parallelized across the multiple nodes based on the data they wish to access. For a system that wishes to maximize read-centric workloads, it is significantly easier to implement. Databases implementing the shared-nothing model often refer to themselves as *massively parallel processing* (*MPP)* databases. Figure 8-3 illustrates the shared-nothing model.

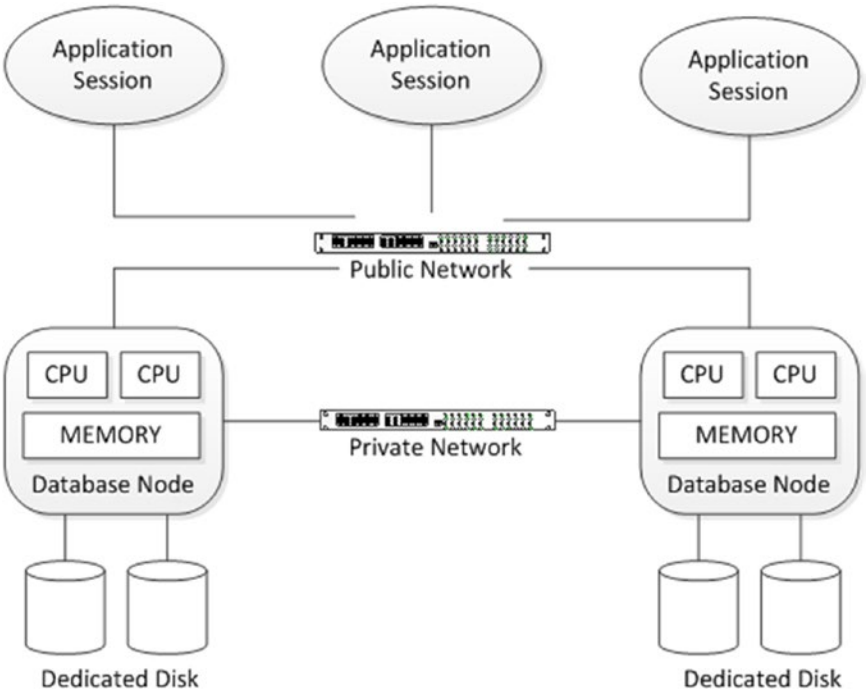


Figure 8-3. *Shared-nothing database architecture*

The shared-nothing architecture tends to break down in transactional scenarios, because of the need to coordinate transactions that may touch data on multiple nodes. Since ACID transactions are "all or nothing," it's necessary for all nodes in the transaction to coordinate closely on transaction execution. This coordination, known as *two-phase commit,* is notoriously difficult to implement and may result in "in doubt" transactional outcomes and poor transactional performance.

The other drawback of the shared-nothing architecture is that without careful partitioning, the cluster workload becomes unbalanced. Maintaining correct partitioning becomes a major operational activity. When nodes are added or removed from the cluster, expensive rebalancing is required.

A *shared-disk* architecture theoretically allows for greater and more elastic scalability, and it removes the need for rebalancing operations. It also provides a more economical high-availability solution, since no node has exclusive responsibility for any particular set of data. In shared-nothing, a node failure results in a portion of the database being unavailable, while in shared-disk, the remaining nodes are able to take over responsibility for the failed node.

The challenge for the shared-disk architecture is the need to coordinate cached data across nodes. Without an in-memory cache, performance for all operations will degrade to disk speed. But to maintain a consistent view of data across all nodes, each node needs to maintain a consistent cache. Maintaining this *cache coherency* puts a strain on the network between the nodes and is difficult to successfully implement.

To date, the only surviving commercially successful shared-disk RDBMS is Oracle's *Real Application Clusters* (*RAC*) cluster database. RAC is the basis for Oracle's Exadata database machine and cloud database offerings. Figure 8-4 illustrates the shared-disk model.

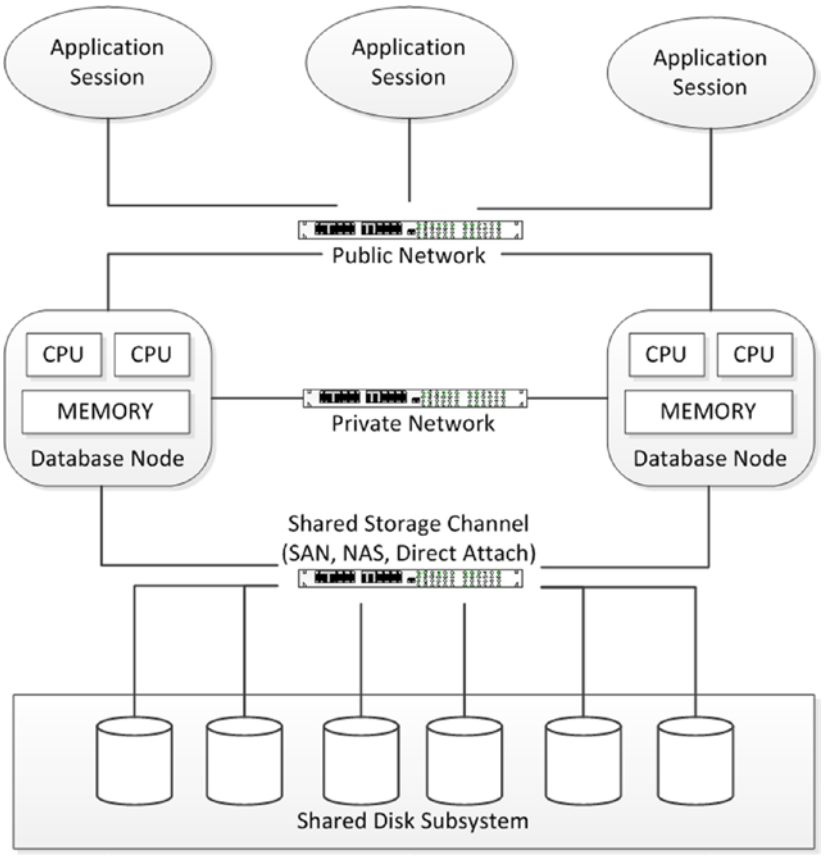


Figure 8-4. *Shared-disk database architecture*

Nonrelational Distributed Databases

Maintaining ACID transactional integrity across multiple nodes in a distributed relational database is a significant challenge. However, in nonrelational database systems, ACID compliance is often not provided. For nonrelational distributed databases, the following considerations become more significant:

- **Balancing availability and consistency**: As we saw in Chapter 3, Brewer's CAP theorem argues that a distributed database that aims to scale beyond a single local network must choose between availability and consistency in the event of a network partition. An ACID-compliant database is obliged to favor consistency over all other factors. However, a nonrelational database without the constraint of strict ACID compliance can strike a different balance.
- **Hardware economics**: Even small differences in the cost of individual servers multiply quickly when a system scales to thousands or hundreds of thousands of nodes. Therefore, an economical database architecture will better leverage commodity hardware so as to take advantage of the best price/performance ratios available. Furthermore, it may become necessary to be able to cope with disparities between server configurations, so that new hardware can be added to the database cluster without requiring all existing nodes to be upgraded to the latest hardware specification.
- **Resilience**:In a massive database cluster, nodes will fail from time to time. In the event of these failures, there can be no data loss, interruption to availability, or maybe even failure at the transaction level.

There have been three broad categories of distributed database architecture adopted by next-generation databases. The three models are:

- Variations on **traditional sharding architecture,** in which data is segmented across nodes based on the value of a "shard key."
- Variations on the Hadoop HDFS/HBase model, in which an "**omniscient master**" determines where data should be located in the cluster, based on load and other factors
- The Amazon Dynamo **consistent hashing model**, in which data is distributed across nodes of the cluster based on predictable mathematical hashing of a key value.

Replication may be inherent within each of these architectures in order to ensure that no data is lost in the event of a server failure, although the replication strategies vary. We look at examples of each of these approaches in the remainder of this chapter. We use MongoDB as an example of a sharding architecture, HBase as the example of an omniscient master, and Cassandra as an example of Dynamo-style consistent hashing.

MongoDB Sharding and Replication

MongoDB supports sharding to provide scale-out capabilities and replication for high availability. Although each can be implemented independently of the other, they are usually both present in a production scenario.

Sharding

A high-level representation of the MongoDB sharding architecture is shown in Figure 8-5. Each shard is implemented by a distinct MongoDB database, which in most respects is unaware of its role in the broader sharded server (1). A separate MongoDB database—the config server (2)—contains the metadata that can be used to determine how data is distributed across shards. A router process (3) is responsible for routing requests to the appropriate shard server.

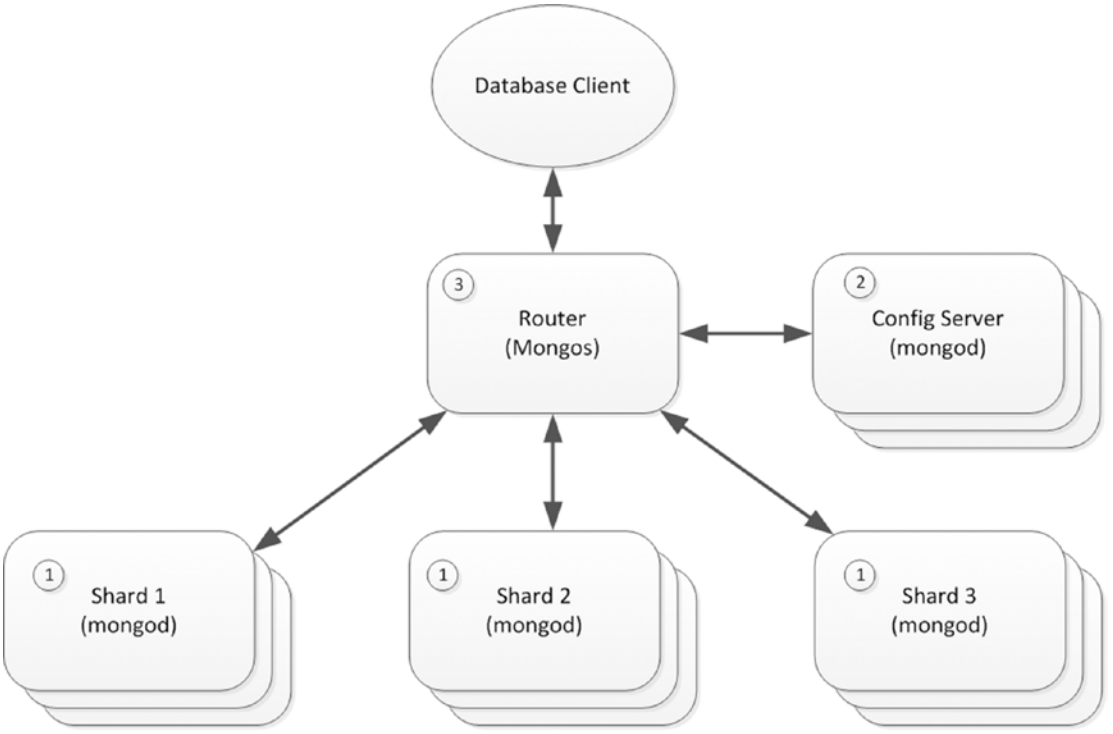


Figure 8-5. *MongoDB sharding architecture*

You may recall that in MongoDB, a collection is used to store multiple JSON documents that usually have some common attributes. To shard a collection, we choose a *shard key,* which is one or more indexed attributes that will be used to determine the distribution of documents across shards. The B-tree structure of the MongoDB index contains the information necessary to distribute keys evenly across shards.

Sharding Mechanisms

Distribution of data across shards can be either *range based* or *hash based*. In range-based partitioning, each shard is allocated a specific range of shard key values. MongoDB consults the distribution of key values in the index to ensure that each shard is allocated approximately the same number of keys. In hash-based sharding, the keys are distributed based on a hash function applied to the shard key.

There are advantages and compromises involved in each scheme. Figure 8-6 illustrates the performance trade-offs inherent in range and hash sharding for inserts and range queries.

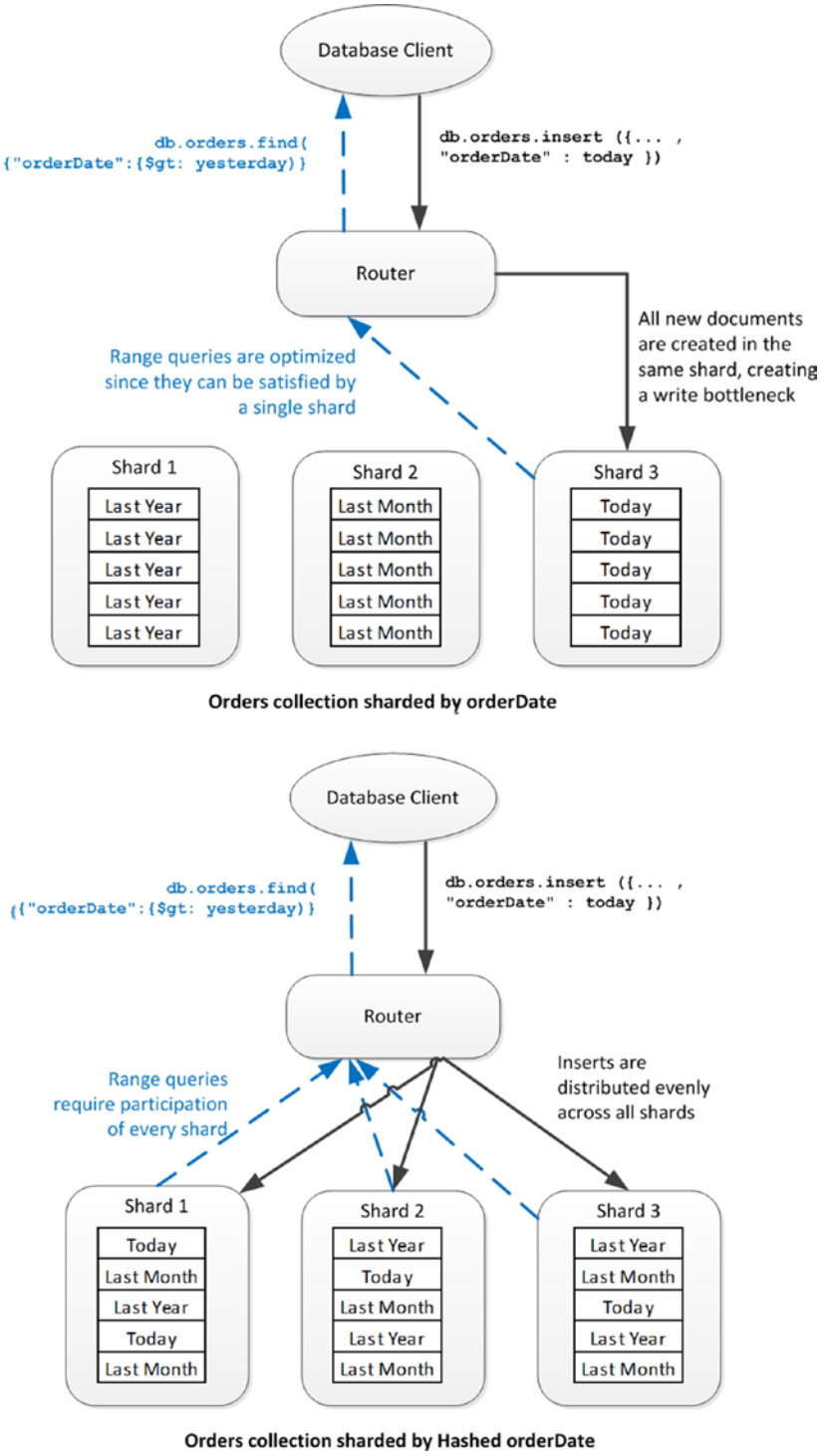


Figure 8-6. *Comparison of range and hash sharding in MongoDB*

Range-based partitioning allows for more efficient execution of queries that process ranges of values, since these queries can often be resolved by accessing a single shard. Hash-based sharding requires that range queries be resolved by accessing all shards. On the other hand, hash-based sharding is more likely to distribute "hot" documents (unfilled orders or recent posts, for instance) evenly across the cluster, thus balancing the load more effectively.

However, when range partitioning is enabled and the shard key is continuously incrementing, the load tends to aggregate against only one of the shards, thus unbalancing the cluster. With hash-based partitioning new documents are distributed evenly across all members of the cluster. Furthermore, although MongoDB tries to distribute shard keys evenly across the cluster, it may be that there are hotspots within particular shard key ranges which again unbalance the load. Hash-based sharding is more likely to evenly distribute the load in this scenario.

Tag-aware sharding allows the MongoDB administrator to fine-tune the distribution of documents to shards. By associating a shard with the tag, and associating a range of keys within a collection with the same tag, the administrator can explicitly determine the shard on which these documents will reside. This can be used to archive data to shards on cheaper, slower storage or to direct particular data to a specific data center or geography.

Cluster Balancing

When hash-based sharding is implemented, the number of documents in each shard tends to remain balanced in most scenarios. However, in a range-based sharding scenario, it is easy for the shards to become unbalanced, especially if the shard key is based on a continuously increasing value, such as an auto-incrementing primary key ID.

For this reason, MongoDB will periodically assess the balance of shards across the cluster and perform rebalance operations, if needed. The unit of rebalance is the *shard chunk*. Shards consist of chunks—typically 64MB in size—that contain contiguous values of shard keys (or of hashed shard keys). If a shard is added or removed from the cluster, or if the balancer determines that a shard has become unbalanced, it can move chunks from one shard to another. The chunks themselves will be split if they grow too large.

Replication

Sharding is almost always combined with replication so as to ensure both availability and scalability in a production MongoDB deployment.

In MongoDB, data can be replicated across machines by the means of *replica sets*. A replica set consists of a primary node together with two or more secondary nodes. The primary node accepts all write requests, which are propagated asynchronously to the secondary nodes.

The primary node is determined by an election involving all available nodes. To be eligible to become primary, a node must be able to contact more than half of the replica set. This ensures that if a network partitions a replica set in two, only one of the partitions will attempt to establish a primary.

The successful primary will be elected based on the number of nodes to which it is in contact, together with a priority value that may be assigned by the system administrator. Setting a priority of 0 to an instance prevents it from ever being elected as primary. In the event of a tie, the server with the most recent *optime*—the timestamp of the last operation—will be selected.

The primary stores information about document changes in a collection within its local database, called the *oplog*. The primary will continuously attempt to apply these changes to secondary instances.

Members within a replica set communicate frequently via heartbeat messages. If a primary finds it is unable to receive heartbeat messages from more than half of the secondaries, then it will renounce its primary status and a new election will be called. Figure 8-7 illustrates a three-member replica set and shows how a network partition leads to a change of primary.

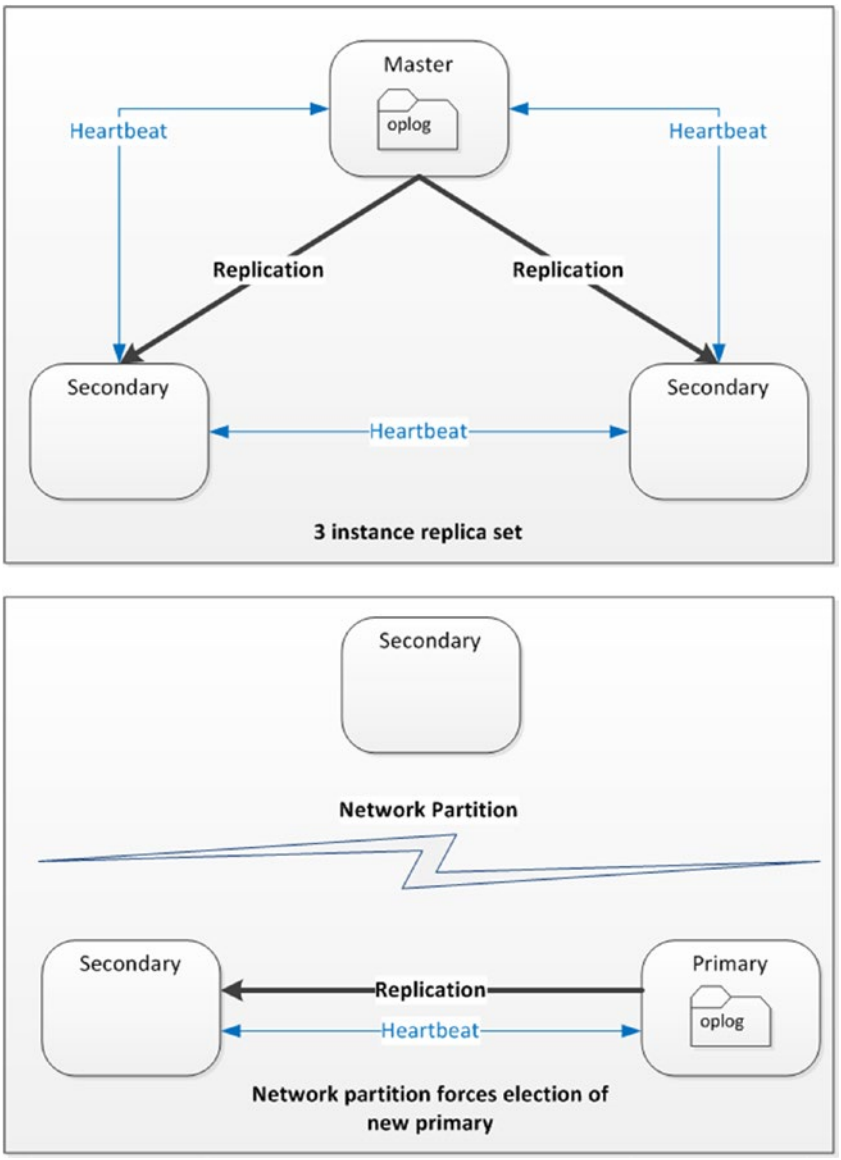


Figure 8-7. *MongoDB replica set and primary failover*

Arbiters are special servers that can vote in the primary election, but that don't hold data. For large databases, these arbiters can avoid the necessity of creating otherwise unnecessary extra servers to ensure that a quorum is available when electing a primary.

Write Concern and Read Preference

A MongoDB application has some control over the behavior of read and write operations, providing a degree of tunable consistency and availability.

- The **write concern** setting determines when MongoDB regards a write operation as having completed. By default, write operations complete once the primary has received the modification. This means that if the primary should fail irrecoverably, then data might be lost. To ensure that write operations have been propagated beyond the primary, the client can issue a blocking call, which will wait until the write has been received by all secondaries, a majority of secondaries, or a specified number of secondaries.
- The **read preference** determines where the client sends read requests. By default, all read requests are sent to the primary. However, the client driver can request that read requests be routed to the secondary if the primary is unavailable, or to secondaries, or to whichever server is "nearest." The latter setting is intended to favor low latency over consistency.

The default settings for read preference and write concern result in MongoDB behaving as a strictly consistent system: everybody will see the same version of a document. Allowing reads to be satisfied from a secondary node results in a more eventually consistent behavior, unless the write concern is configured to block writes until they reach secondary nodes.

We'll look more at MongoDB consistency in the next chapter.

HBase

HBase can be thought of both as the "Hadoop database" and as "open-source BigTable." That is, we can describe HBase as a mechanism for providing random access database services on top of the Hadoop HDFS file system, or we can think of HBase as an open-source implementation of Google's BigTable database that happens to use HDFS for data storage. Both of these descriptions are accurate: although HBase theoretically can be implemented on top of any distributed file system—or, indeed, even a nondistributed file system—it's almost always implemented on top of Hadoop HDFS, and many of HBase's architectural assumptions reflect this. On the other hand, HBase implements real-time random access database functionality, which is essentially distinct from the base capabilities of Hadoop.

In the discussion that follows, we are going to concentrate on the HBase architecture as it is most commonly encountered: as implemented on top of HDFS. The implementation of HBase over HDFS creates a sort of hybrid, a mix of shared-nothing and shared-disk clustering patterns. On the one hand, every HBase node can access any element of data in the database because all data is accessible via HDFS. On the other hand, it is typical to co-locate HBase servers with HDFS DataNodes, which means that in practice each node tends to be responsible for an exclusive subset of data stored on local disk.

In either case, HDFS provides the reliability guarantees for data on disk: the HBase architecture is not required to concern itself with write mirroring or disk failure management, because these are handled automatically by the underlying HDFS system.

We introduced HDFS and Hadoop architecture in Chapter 2; please refer to that chapter if you need a refresher. HDFS implements a distributed file system using disks that are directly attached to the servers—*DataNodes*—that constitute a Hadoop cluster. HDFS automatically manages redundancy of data: by default, data is replicated across three DataNodes, one of which (if possible) is located on a separate server rack.

Tables, Regions, and RegionServers

HBase implements a wide column store based on Google's BigTable specification. We touched on that data model in Chapter 2, and we'll talk more about it in Chapter 10. For now, we can consider HBase tables as potentially massive tabular datasets that are implemented on disk by a variable number of HDFS files called *Hfiles*.

All rows in an HBase table are identified by a unique *row key*. A table of nontrivial size will be split into multiple horizontal partitions called *regions*. Each region consists of a contiguous, sorted range of key values. This resembles the MongoDB range-based sharding scheme we described earlier in this chapter.

Read or write access to a region is controlled by a *RegionServer*. Each RegionServer normally runs on a dedicated host, and is typically co-located with the Hadoop DataNode.

There will usually be more than one region in each RegionServer. As regions grow, they split into multiple regions based on configurable policies. Regions may also be split manually. We'll discuss this more a little later in this chapter.

Each HBase installation will include a Hadoop *Zookeeper* service that is implemented across multiple nodes. Hbase may share this Zookeeper ensemble with the rest of the Hadoop cluster or use a dedicated service.

When an HBase client wishes to read or write to a specific key value, it will ask Zookeeper for the address of the RegionServer that controls the HBase catalog. This catalog consists of the tables -ROOT- and .META., which identify the RegionServers that are responsible for specific key ranges. The client will then establish a connection with that RegionServer and request to read or write the key value concerned.

The HBase *master server* performs a variety of housekeeping tasks. In particular, it controls the balancing of regions among RegionServers. If a RegionServer is added or removed, the master will organize for its regions to be relocated to other RegionServers.

Figure 8-8 illustrates some of these architectural elements. An HBase client consults Zookeeper to determine the location of the HBase catalog tables (1), which can be then be interrogated to determine the location of the appropriate RegionServer (2). The client will then request to read or modify a key value from the appropriate RegionServer (3). The RegionServer reads or writes to the appropriate disk files, which are located on HDFS (4).

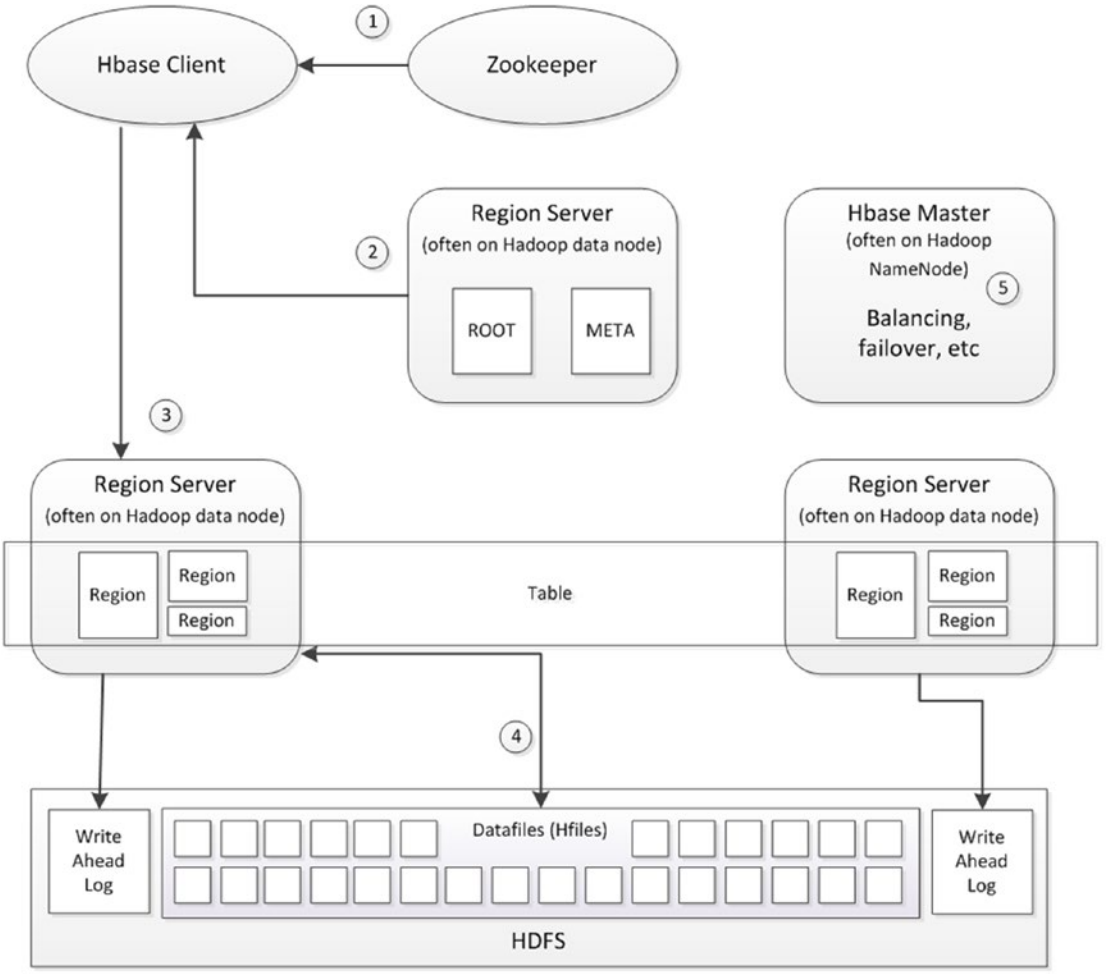


Figure 8-8. *HBase architecture*

Caching and Data Locality

The RegionServer includes a *block cache* that can satisfy many reads from memory, and a *MemStore,* which writes in memory before being flushed to disk. However, to ensure durability of the writes, each RegionServer has a dedicated *write ahead log* (*WAL*), which journals all writes to HDFS. This architecture is an implementation of the *log-structured merge tree* (*LSM*) pattern that is more fully described in Chapter 10.

The RegionServer can act as a generic HDFS client, communicating with the HDFS NameNode to perform read and write operations to files. In the most typical production deployment scenario, each RegionServer is located on a Hadoop NameNode, and as a result, region data will be co-located with the RegionServer, providing good *data locality*. This data locality will be disrupted by rebalance operations and RegionServer failovers, but compactions—which merge HDFS disk files as described in Chapter 10—will restore data locality.

Hadoop and HBase support a mode known as *short-circuit reads,* in which the RegionServer can read directly from local disk, bypassing the NameNode. This, of course, is only possible when the data is stored on a DataNode that also hosts the RegionServer.

The three levels of data locality are shown in Figure 8-9. In the first configuration, the RegionServer and the DataNode are located on different servers and all reads and writes have to pass across the network. In the second configuration, the RegionServer and the DataNode are co-located and all reads and writes pass through the DataNode, but they are satisfied by the local disk. In the third scenario, short-circuit reads are configured and the RegionServer can read directly from the local disk.

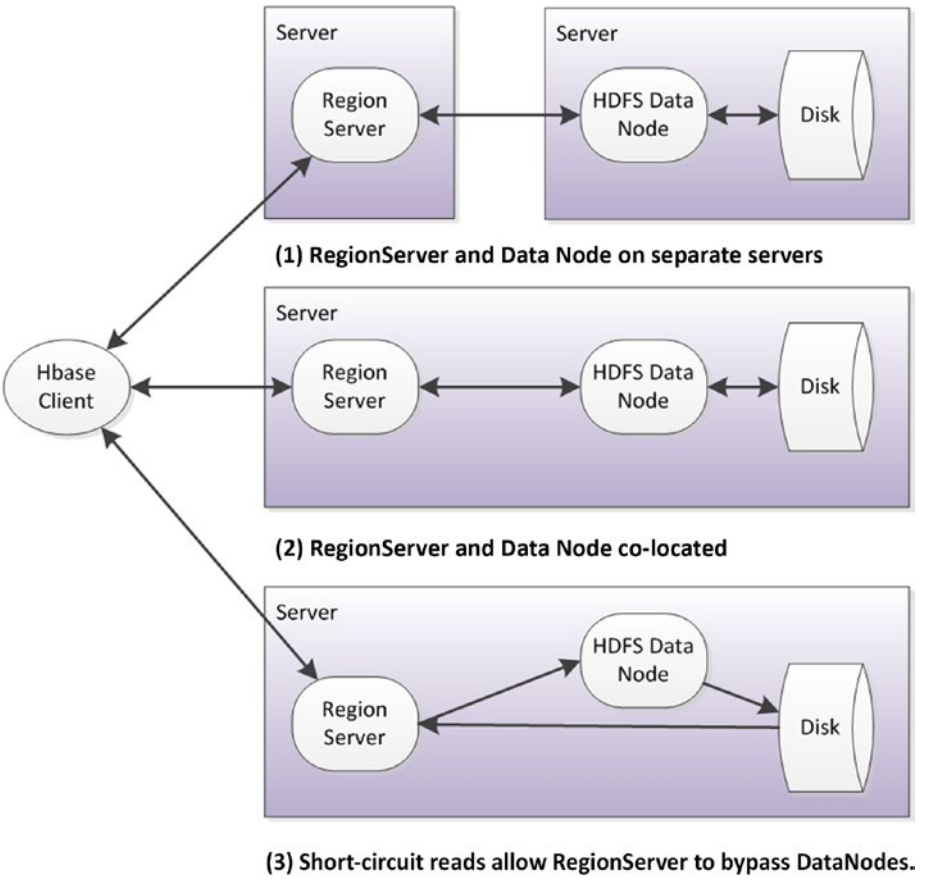


Figure 8-9. *Data locality in HBase*

Rowkey Ordering

The HBase region partitioning scheme requires that regions consist of contiguous ranges of *rowkeys.* This range-based partitioning has a significant impact on performance when the rowkey contains some form of monotonically incrementing value, such as a timestamp or a incrementing counter. In this event, all write operations will be directed to a specific region and hence to a single RegionServer. This can create a bottleneck on write throughput.

HBase offers no internal mechanisms to mitigate this issue. It's up to the application designer to construct a key that is either randomized—a hash of the timestamp, for instance—or is prefixed in some way with a more balanced attribute. In the HBase time series database *OpenTSDB,* the timestamp is prefixed by a metric identifier variable that has a large number of values. Data for a single metric will be located in a specific RegionServer, but data for a specific timestamp will be distributed across all the RegionServers.

RegionServer Splits, Balancing, and Failure

As regions grow, they will be split by the RegionServer as required. The new regions will remain controlled by the original RegionServer—at least initially—but they are eligible for relocation during load-balancing operations. The default region-split policy results in regions of incrementally greater size, with the first split occurring after as little as 128M, while the tenth region will be approximately 10GB in size. However, it is possible to split regions manually or to override the split policy with custom code.

One of the most important responsibilities of the HBase *master node* is to balance regions across RegionServers. The master will periodically evaluate the balance of regions across all RegionServers, and should it detect an imbalance, it will migrate regions to another server. This is a "soft" rebalance—the region's data remains in its original location on HDFS disk, but the responsibility for managing that data is moved to a different RegionServer.

As noted earlier, rebalancing tends to result in a loss of data locality: when the RegionServer acquires responsibility for a new region, that region will probably be located on a remote data node—at least until the next major compaction.

Region Replicas

In earlier versions of HBase, a failure of a RegionServer would require a failover to a new RegionServer. Because the RegionServers don't actually store the data for a region (the data is in HDFS), a failure is not catastrophic. The master would detect the failure and allocate the regions concerned to other RegionServers in a similar way to the balancing operation. However, some interruption of service would result.

Region replicas allow for redundant copies of regions to be stored on multiple RegionServers. Should a RegionServer fail, these replicas can be used to service client requests.

The original RegionServer serves as the master copy of the region. Read-only replicas of the region are distributed to other RegionServers—located in other racks, if possible—which then "follow" the primary RegionServer. Writes to these replicas are asynchronous to primary RegionServer writes, so data in the replicas will not always be up to date. We'll see in the next chapter how the configuration of HBase region replicas affects consistency and availability.

HBase also supports a replication facility that can be used to stand up a duplicated HBase database. This is typically used to duplicate an entire HBase database in another data center.

Cassandra

In Chapter 3, we introduced Amazon's Dynamo database and the concept of consistent hashing. A number of open-source systems have implemented the Dynamo model. In this chapter, we consider the Cassandra implementation.

Gossip

In HBase and MongoDB, we encountered the concept of master nodes—nodes which have a specialized supervisory function, coordinate activities of other nodes, and record the current state of the database cluster. In Cassandra and other Dynamo databases, there are no specialized master nodes. Every node is equal and every node is capable of performing any of the activities required for cluster operation.

Nodes in Cassandra do, however, have short-term specialized responsibilities. For instance, when a client performs an operation, a node will be allocated as the *coordinator* for that operation. When a new member is added to the cluster, a node will be nominated as the *seed node* from which the new node will seek information. However, these short-term responsibilities can be performed by any node in the cluster.

One of the advantages of a master node is that it can maintain a canonical version of cluster configuration and state. In the absence of such a master node, Cassandra requires that all members of the cluster be kept up to date with the current state of cluster configuration and status. This is achieved by use of the *gossip* protocol. Every second each member of the cluster will transmit information about its state and the state of any other nodes it is aware of to up to three other nodes in the cluster. In this way, cluster status is constantly being updated across all members of the cluster.

The gossip protocol is aptly named: when people gossip, they generally tend to gossip about other people! Likewise, in Cassandra, the nodes gossip about other nodes as well as about their own state.

Cluster configuration is persisted in the system *keyspace,* which is available to all members of the cluster. A keyspace is roughly analogous to a schema in a relational database—the system keyspace contains tables that record metadata about the cluster configuration.

This architecture eliminates any single point of failure within the cluster. Although distributed databases with master nodes have strategies to allow for rapid failover, the crash of a master node usually creates a temporary reduction in availability, such as momentarily falling back to read-only mode.

One of the main topics of gossip within a Cassandra cluster is node availability. The traditional mechanism for detecting node failure is to send heartbeats between nodes. However, in a widely distributed system, the heartbeats may be lost because of network issues rather than actual node failure. For this reason, Cassandra failure detection is more probabilistic: if you like, nodes in the cluster become increasingly "worried" about other nodes. If it seems likely that a node is down, then the operations will be directed to "known good" nodes.

Consistent Hashing

Cassandra and other dynamo-based databases distribute data throughout the cluster by using consistent hashing. The rowkey (analogous to a primary key in an RDBMS) is hashed. Each node is allocated a range of hash values, and the node that has the specific range for a hashed key value takes responsibility for the initial placement of that data.

In the default Cassandra partitioning scheme, the hash values range from -2^{63} to $2^{63}-1$. Therefore, if there were four nodes in the cluster and we wanted to assign equal numbers of hashes to each node, then the hash ranges for each would be approximately as follows:

Node	Low Hash	High Hash
Node A	-2^{63}	$-2^{63}/2$
Node B	$-2^{63}/2$	0
Node C	0	$2^{63}/2$
Node D	$2^{63}/2$	2^{63}

We usually visualize the cluster as a ring: the circumference of the ring represents all the possible hash values, and the location of the node on the ring represents its area of responsibility. Figure 8-10 illustrates simple consistent hashing: the value for a rowkey is hashed, which determines its position on "the ring." Nodes in the cluster take responsibility for ranges of values within the ring, and therefore take ownership of specific rowkey values.

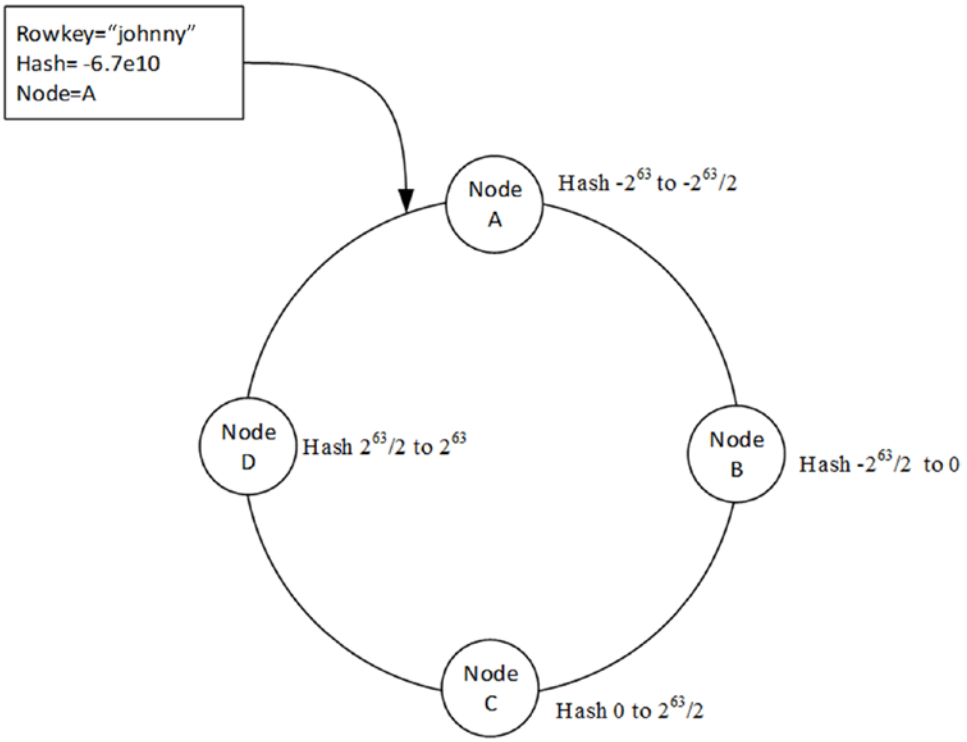


Figure 8-10. *Consistent hashing*

The four-node cluster in Figure 8-10 is well balanced because every node is responsible for hash ranges of similar magnitude. But we risk unbalancing the cluster as we add nodes. If we double the number of nodes in the cluster, then we can assign the new nodes at points on the ring between existing nodes and the cluster will remain balanced. However, doubling the cluster is usually impractical: it's more economical to grow the cluster incrementally.

Early versions of Cassandra had two options when adding a new node. We could either remap all the hash ranges, or we could map the new node within an existing range. In the first option we obtain a balanced cluster, but only after an expensive rebalancing process. In the second option the cluster becomes unbalanced; since each node is responsible for the region of the ring between itself and its predecessor, adding a new node without changing the ranges of other nodes essentially splits a region in half. Figure 8-11 shows how adding a node to the cluster can unbalance the distribution of hash key ranges.

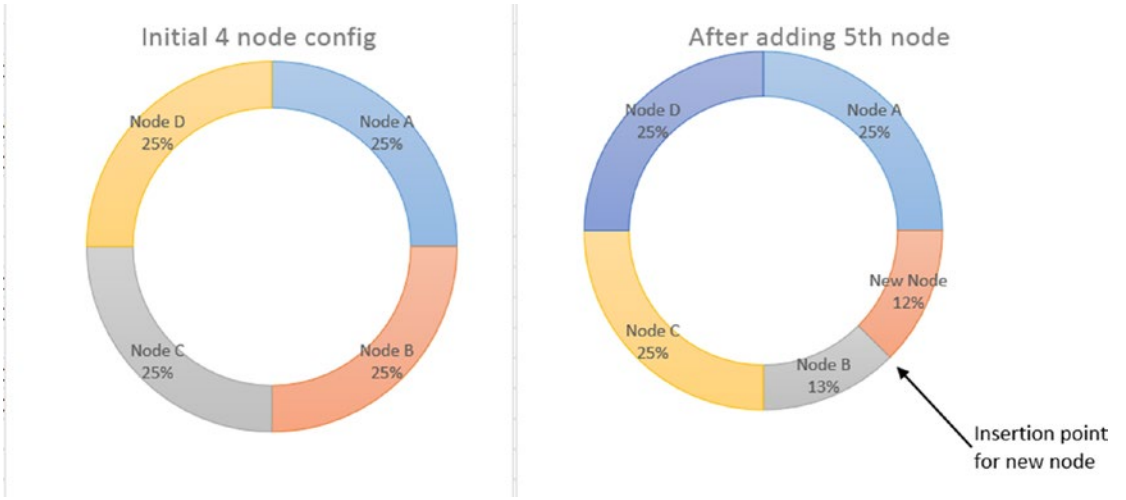


Figure 8-11. *Adding a node to a Cassandra cluster (without virtual nodes)*

Virtual nodes, implemented in Cassandra, Riak, and many other Dynamo-based systems, provide a solution to this issue. When using virtual nodes, the hash ranges are calculated for a relatively large number of virtual nodes—256 virtual nodes per physical node, typically—and these virtual nodes are assigned to physical nodes. Now when a new node is added, specific virtual nodes can be reallocated to the new node, resulting in a balanced configuration with minimal overhead. Figure 8-12 illustrates the relationship between virtual nodes and physical nodes.

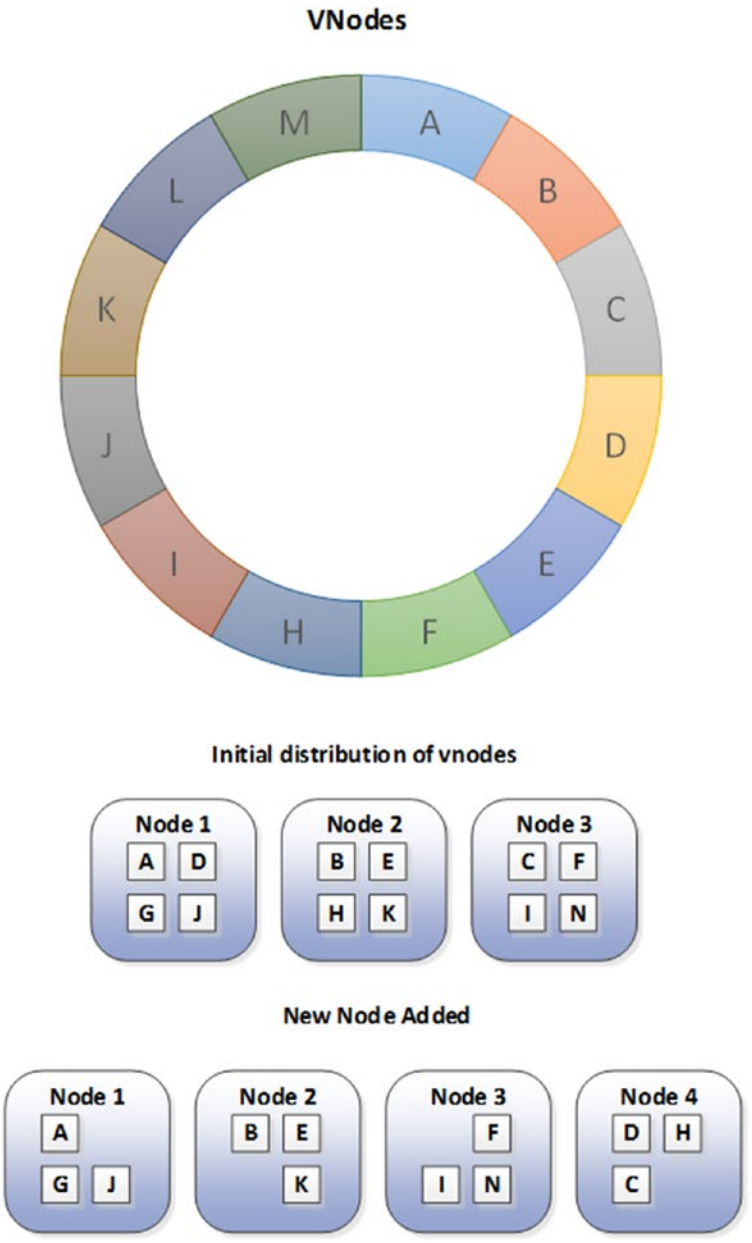


Figure 8-12. *Using virtual nodes to partition data among physical nodes*

Virtual nodes have some other advantages. For instance, it is easier to balance a cluster made up of heterogeneous systems, since you can allocate more virtual nodes to more powerful new machines and fewer virtual nodes to underpowered older machines. Also, if a node dies it can be reconstituted from a larger number of physical machines, thus sharing the overhead of recovery more equitably across the cluster.

Order-Preserving Partitioning

The Cassandra *partitioner* determines how keys are distributed across nodes. The default partitioner uses consistent hashing, as described in the previous section. Cassandra also supports *order-preserving partitioners* that distribute data across the nodes of the cluster as ranges of actual (e.g., not hashed) rowkeys. This has the advantage of isolating requests for specific row ranges to specific machines, but it can lead to an unbalanced cluster and may create hotspots, especially if the key value is incrementing. For instance, if the key value is a timestamp and the order-preserving partitioner is implemented, then all new rows will tend to be created on a single node of the cluster.

In early versions of Cassandra, the order-preserving petitioner might be warranted to optimize range queries that could not be satisfied in any other way; however, following the introduction of secondary indexes, the order-preserving petitioner is maintained primarily for backward compatibility, and Cassandra documentation recommends against its use in new applications.

Replicas

So far, we have seen how Cassandra allocates the initial copy of a data item to a node. The consistent hashing algorithm also determines where replicas of data items are stored.

The node responsible for the hash range that equates to a specific rowkey value is called the *coordinator* node. The coordinator is responsible for ensuring that the required number of replica copies of the data are also written. The number of nodes to which the data must be written is known as the *replication factor*, and is the "N" in the NWR notation that we first encountered in Chapter 3.

By default, the coordinator will write copies of the data to the next N-1 nodes on the ring. So if the replication factor is 3, the coordinator will send replicas of the data item to the next two nodes on the ring. In this scenario, each node will be replicating data *from* the previous two nodes on the ring and replicating *to* the next two nodes on the ring. This simple scheme is referred to as the *simple replication strategy*.

Cassandra also allows you to configure a more complex and highly available scheme. The *Network Topology Aware replication strategy* ensures that copies will be written to nodes on other server racks within the same data center, or optionally to nodes in another data center altogether. Figure 8-13 illustrates these two replication strategies.

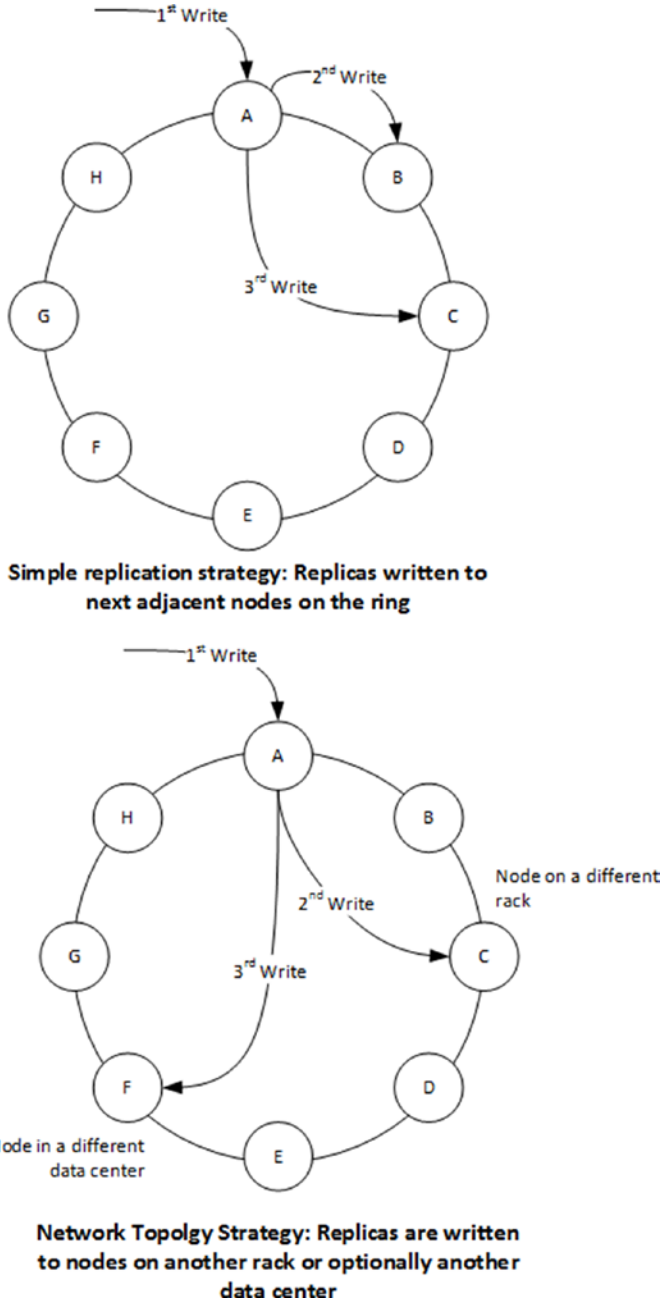


Figure 8-13. *Replication strategies in Cassandra*

Snitches

Cassandra uses *snitches* to help optimize read and write operations. A variety of snitches may be configured.

- The **simpleSnitch** returns only a list of nodes on the ring. This is sufficient for the simple replication strategy we discussed in the previous section.
- The **RackInferringSnitch** uses the IP addresses of hosts to infer their rack and data center location. This snitch supports the network aware replication strategy.
- The **PropertyFileSnitch** uses data in the configuration file rather than IP addresses to determine the data center and rack topology.

In addition, all snitches monitor the read latency for requests and use this to build a statistical model that can route requests to the best-performing nodes.

Specialized snitches exist that understand the networked topology inside various cloud platforms, such as Amazon EC2.

Summary

In this chapter we've reviewed distributed database patterns for traditional relational databases and for several nonrelational systems. Relational database architecture was developed in an era of large, monolithic database servers, and most relational databases still run as a single instance. However, shared-nothing clustering is commonplace in massively parallel data warehouses, and Oracle has a commercially successful shared-disk clustered RDBMS.

We looked in detail at three distributed nonrelational database systems. MongoDB uses a combination of sharding and replication to enable distributed processing. HBase leverages the distributed file system of the Hadoop Distributed File System together with a range-partitioning strategy to achieve a highly scalable solution. Cassandra uses the consistent hashing scheme pioneered in Amazon's Dynamo system to create a symmetrical clustering solution in which no master servers are required.

In a distributed database, multiple copies of data are typically maintained across the cluster. In the next chapter, we'll see how these databases manage data consistency within such a distributed system.

CHAPTER 3

Sharding, Amazon, and the Birth of NoSQL

Step 1 - Shard database. Step 2 - shoot yourself.

—Twitter user @Dmitriy, 2009

"Bob: So, how do I query the database?

IT guy: It's not a database. It's a Key-Value store. . . .

You write a distributed map-reduce function in Erlang.

*Bob: Did you just tell me to go **** myself?*

IT guy: I believe I did, Bob."

—Fault Tolerance cartoon, @jrecursive, 2009

The last time we saw a major new brand of relational database was around 1995, with the first release of MySQL. In 1995, the World Wide Web in the United States was barely two years old—the Netscape browser had been released only the year before. In terms of computer systems, it was a different era.

In the 10 years between 1995 and 2005, the Internet was transformed from a dial-up curiosity to arguably the most important communication system in our civilization, a foundation for international commerce, and soon to be a centerpiece of our social lives. Despite that, the database systems used in 2005 had the same names as those used in 1995. Surveying the software landscape in 2005, you would be excused for thinking that there was nothing new under the database sun.

Behind the scenes, however, the ability of the relational database to sustain the needs of web applications had been stretched to the breaking point. Out of this pressure arose a new breed of web-scale transactional database systems—what we now call NoSQL.

Scaling Web 2.0

The World Wide Web was initially conceived and implemented as a global collection of linked static documents. Indeed, the vast majority of the web still consists of read-only static content. Google developed many of the technologies introduced in the last chapter to provide an index and search capability across these documents.

But to retailers and other businesses, the web promised to deliver much more than simply a place to store online catalogs and white papers. The idea of web-based retail outlets promised to revolutionize modern commerce. And although this concept of e-commerce resulted in the biggest boom and bust of our generation, the promise was eventually realized and today you can purchase virtually anything online.

The World Wide Web of static pages is often referred to as *Web 1.0*, and the World Wide Web of dynamically created content with transactional capability is referred to as *Web 2.0*. However, version 2.0 was not the result of a controlled architectural redevelopment; rather, it resulted from web developers scrambling to cope with ever-increasing demands for functionality, performance, and scale.

How Web 2.0 was Won

The first web servers provided accessed to hyperlink documents written in HTML. There were no database systems involved, and there was no ability to conduct business or any transactional activity.

Early websites that wanted to provide some form of user interaction—Amazon.com, for instance—used the *Common Gateway Interface* (*CGI*). CGI allowed an HTTP request to invoke a script rather than display a HTML page. Early dynamic webpages would then invoke scripts written in the Perl language, which would connect to a database and generate HTML code on the fly based on database contents. In this way, a website could display a catalog based on data held within a database or could personalize a page based on a user's profile.

CGI-based approaches gave way to more elegant and cohesive frameworks such as Java J2EE and ASP.NET, though huge numbers of websites based on the PHP language still follow the CGI model. However, regardless of the framework employed, the common pattern entailed a web application server displaying information dynamically generated from database content.

It's easy to scale up the web layer in this architecture. Just as there can be many clients for every database in the client/server architecture, there can be as many web servers as you like communicating with a single back-end database. So a bottleneck in the web server layer can be fixed simply by adding more web servers.

However, fixing bottlenecks at the database layer was not so simple. While there were a few database clustering solutions available at the beginning of the twenty-first century, it was generally difficult to achieve linear scalability with these solutions, and none had ever demonstrated scalability at the level required by the larger e-commerce sites.

During the early stages of Web 2.0, the solution to database performance was simply to buy a more powerful database server. Database servers were getting more capable every year, and storage servers could provide databases with massive IO capacity. So during the initial Internet bubble, companies like EMC and Oracle did very well, because it made sense for these early Web 2.0 companies to buy the most powerful database possible and thereby sustain the absurdly optimistic growth curves they expected.

Two factors led to the abandonment of this *scale-up* solution. First, the dot.com crash brought financial reality back into the equation, and the surviving web companies needed financially prudent solutions. Businesses wanted a solution that could start small and grow as required. Second, as Web 2.0 companies reached global scale, they found that even the most massive centralized database server could not meet their needs. Scaling up had run out of steam.

Furthermore, even if the scale-up solution had delivered the capacity required, it still represented a potential single point of failure, and it could not provide equitable response time across a global market.

The Open-source Solution

Following the dot.com crash, open-source software became increasingly valued within Web 2.0 operations. Linux supplanted proprietary UNIX as the operating system of choice, and the Apache web server became dominant. During this period, MySQL overtook Oracle as the Database Management System (DBMS) of choice for website development.

MySQL was then and is still now far less scalable than Oracle; it generally runs on less powerful hardware and is less able to take advantage of multicore processors. However, Web developers came up with a couple of tricks to get MySQL to go further.

First, they used a technology called *Memcached* to avoid database access as much is possible. Memcached is an open-source utility that provides a distributed object cache. Object-oriented languages could cache an object-oriented representation of database information across many servers. By reading from these servers rather than the database, the load on the database could be reduced.

Second, web developers took advantage of MySQL replication. Replication allows changes to one database to be copied to another database. Read requests could be directed to any one of these replica databases. Write operations still had to go to the master database however, because master-to-master replication was not possible. However, in a typical database application—and particularly in web applications—reads significantly outnumber writes, so the read replication strategy makes sense.

Figure 3-1 illustrates the transition from single web server and database server to multiple web servers, Memcached servers, and read-only database replicas.

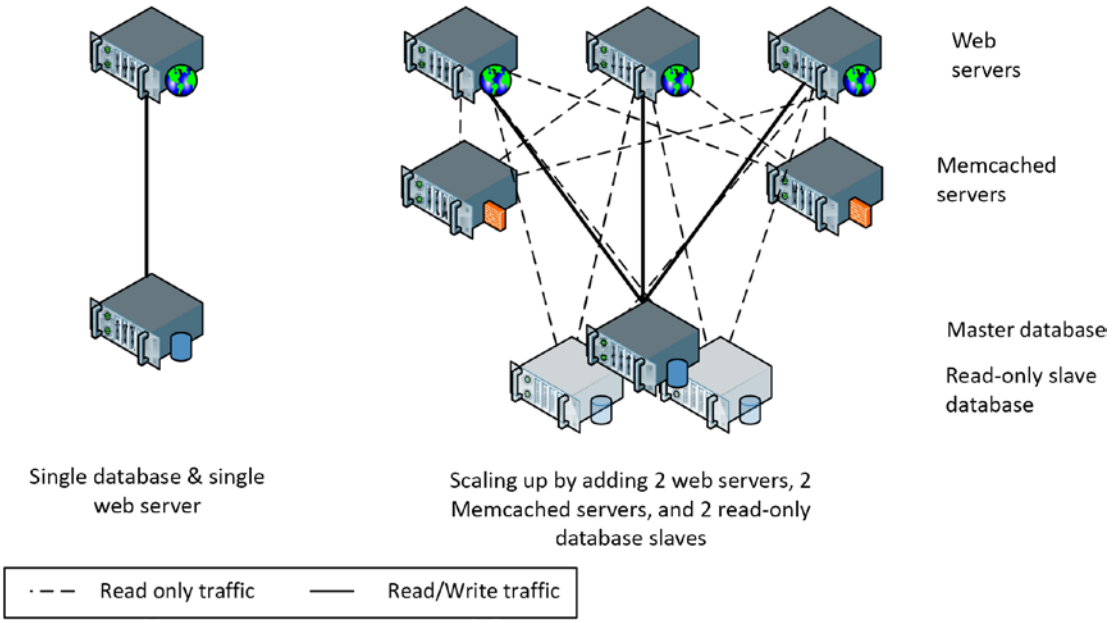


Figure 3-1. *Scaling up with Memcached servers and replication*

Memcached and read replication increase the overall capacity of MySQL-based web applications dramatically. However, both these techniques can only increase the read capability of the system. When the system reaches a bottleneck on database write activity, a more dramatic solution is required.

Sharding

Sharding allows a logical database to be partitioned across multiple physical servers.

In a sharded application, the largest tables are partitioned across multiple database servers. Each partition is referred to as a *shard*. This partitioning is based on a Key Value, such as a user ID. When operating on a particular record, the application must determine which shard will contain the data and then

send the SQL to the appropriate server. Sharding is a solution used at the largest websites; Facebook and Twitter are the most well-known examples. At both of these websites, data that is specific to an individual user is concentrated in MySQL tables on a specific node.

Figure 3-2 illustrates the Memcached and replication configuration shown earlier in this chapter with sharding added. In this example, there are three shards, and for simplicity's sake, the shards are labeled by first letter of the primary key. As a result, we might imagine that rows with the key GUY are in shard 2, while key BOB would be allocated to shard 1. In practice, it is more likely that the primary key would be hashed to ensure even distribution of keys to servers.

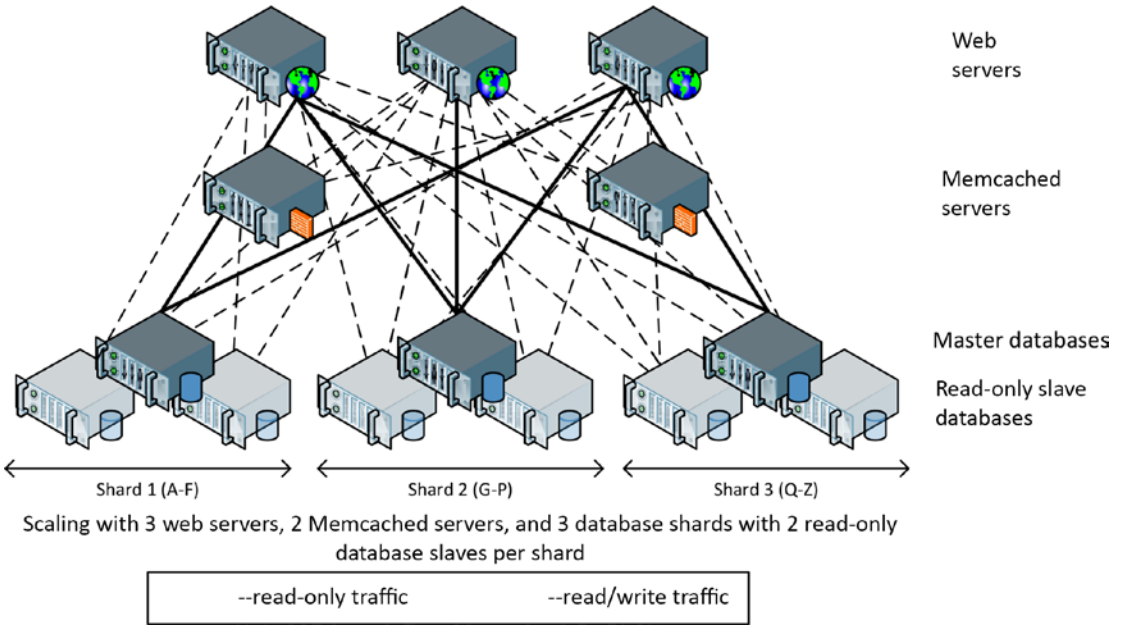


Figure 3-2. *Memcached/replication architecture from Figure 3-1, with sharding added*

The exact number of servers being used at Facebook is constantly changing and not always publicly disclosed, but in around 2011 they did reveal that they were using more than 4,000 shards of MySQL and 9,000 Memcached servers in their configuration. This sharded MySQL configuration supported 1.4 billion peak reads per second, 3.5 million row changes per second, and 8.1 million physical IOs per second. As we will see, sharding involves significant operational complexities and compromises, but it is a proven technique for achieving data processing on a massive scale.

Sharding is simple in concept but incredibly complex in practice. The application must contain logic that understands the location of any particular piece of data and the logic to route requests to the correct shard. Sharding is usually associated with rapid growth, so this routing needs to be dynamic. Requests that can only be satisfied by accessing more than one shard thus need complex coding as well, whereas on a nonsharded database a single SQL statement might suffice.

Death by a Thousand Shards

Sharding—together with caching and replication—is arguably the only way to scale a relational database to massive web use. However, the operational costs of sharding are huge. Among the drawbacks of a sharding strategy are:

- **Application complexity**. It's up to the application code to route SQL requests to the correct shard. In a statically sharded database, this would be hard enough; however, most massive websites are adding shards as they grow, which means that a dynamic routing layer must be implemented. This layer is often in addition to complex code being required to maintain Memcached object copies and to differentiate between the master database and read-only replicas.
- **Crippled SQL**. In a sharded database, it is not possible to issue a SQL statement that operates across shards. This usually means that SQL statements are limited to row-level access. Joins across shards cannot be implemented, nor can aggregate GROUP BY operations. This means, in effect, that only programmers can query the database as a whole.
- **Loss of transactional integrity**. ACID transactions against multiple shards are not possible—or at least not practical. It is possible in theory to implement transactions across databases in some database systems—those supporting *Two Phase Commit* (*2PC*)—but in practice this creates problems for conflict resolution, can create bottlenecks, has issues for MySQL, and is rarely implemented.
- **Operational complexity**. Load balancing across shards becomes extremely problematic. Adding new shards requires a complex rebalancing of data. Changing the database schema also requires a rolling operation across all the shards, resulting in transitory inconsistencies in the schema. In short, a sharded database entails a huge amount of operational effort and administrator skill.

Relational database vendors—Oracle, in particular—tried to create a relational database implementation that could provide the scalability of a sharded database without the ACID and relational compromises or operational headaches. Oracle's *Real Application Clusters* (*RAC*) is the most significant example of a transparently scalable, ACID compliant, relational cluster.

In Oracle RAC databases, each database node works with data located on shared storage devices. This *shared disk* clustering is in contrast to the *shared nothing* model employed by other clustered databases, which are more suited to data warehousing workloads. (We'll compare shared disk and shared nothing architectures in more detail in Chapter 8.)

New database nodes in RAC can be added without any data rebalancing, and a sort of distributed memory cache is implemented across these database nodes. Oracle RAC showed a lot of promise, and indeed is widely implemented. However, it failed as an alternative to the MySQL sharded model, for three reasons: First, it was too expensive. Second, it failed to demonstrate the level of scalability required at the biggest websites. Third, it became apparent that no ACID compliant database could ever satisfy the needs of the world's biggest websites. This last restriction was a sort of "laws of physics" constraint articulated in what has come to be known as *CAP theorem*.

CAP Theorem

In 2000, Eric Brewer outlined the "CAP" conjecture, which was later granted theorem status when a mathematical proof was provided. The CAP theorem says that in a distributed database system, you can have at most only two of Consistency, Availability, and Partition tolerance. *Consistency* means that every

user of the database has an identical view of the data at any given instant. *Availability* means that in the event of a failure, the database remains operational. *Partition tolerance* means that the database can maintain operations in the event of the network's failing between two segments of the distributed system.

In 2000, the issue of partition tolerance was somewhat theoretical. Most systems resided in a single data center, and redundant network connectivity within that data center prevented any partition from ever occurring. If the data center failed, perhaps a failover data center would be bought online. However, there were almost no true multiple data center applications.

But as web systems became global in scope and aspired to continual availability, partition tolerance became a real issue. Consider the distributed application shown in Figure 3-3. In the event of the network partition shown, the system has two choices: either show each user a different view of the data, or shut down one of the partitions and disconnect one of the users.

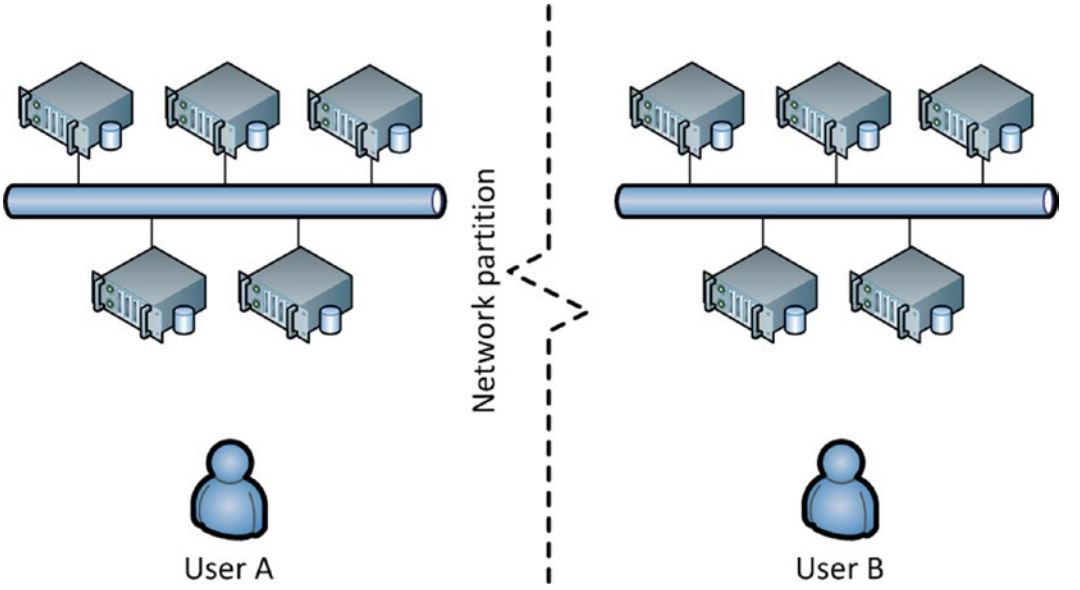


Figure 3-3. *Network partition in a distributed database application*

Oracle's RAC solution, which of course supported the ACID transactional model, would choose consistency. In the event of a network partition—known in Oracle circles as the "split brain" scenario—one of the partitions would choose to shut down. However, in the context of a global social network application, or a worldwide e-commerce system, the desired solution is to maintain availability even if some consistency between users is sacrificed.

Eventual Consistency

CAP theorem provides a stark choice: if you want your system to be undisturbed by network partitions, you must sacrifice strict consistency between partitions.

However, even without considerations of CAP theorem, ACID transactions were increasingly untenable in large-scale distributed websites. This relates more to performance than to availability. In any highly available database system, multiple copies of each data element must be maintained in order to allow the system to continue operating in the event of node failure. In a globally distributed system, it becomes increasingly desirable to distribute nodes around the world to reduce latency in various locations. To ensure

strict consistency, though, it becomes necessary to ensure that a database change is propagated to multiple nodes synchronously and immediately. When one of those nodes is on the other side of the planet, this creates an unavoidable increase in latency.

For banks, this sort of latency penalty is unavoidable. However, for many websites, including social networks and certain e-commerce operations, this worldwide synchronous consistency is unnecessary. It doesn't matter if my friend in Australia can see my tweet a few seconds before my friend in America. As long as both friends can see the tweet eventually, I'm happy.

This concept of *eventual consistency* has become a key characteristic of many NoSQL databases. The concept was most notably outlined by Werner Vogels, CTO of Amazon, and was implemented in Amazon's *Dynamo* key-value store.

Amazon's Dynamo

Amazon had pioneered many of the web technologies used in early Web 2.0, particularly the use of the Perl language to glue together databases and web front ends. Early Amazon used Oracle databases as the primary repository for catalogs, customer details, and orders. The load placed on the Oracle database was tremendous, and several notable Amazon outages were associated with database failures.

Amazon tried splitting the website into multiple functional areas, each of which could have its own dedicated database. They were also very early adopters of *service-oriented architecture* (*SOA*), in which logical services such as get-product-details would be resolved not by SQL queries but by web service calls. This abstracted the database from the application layer and allowed Amazon to experiment with alternative database technologies.

In 2007, Amazon revealed details of an alternative nonrelational system that had been developed internally to address the requirements of their massive online website.[1] This system—called Dynamo—was built with the following requirements in mind:

- **Continuous availability**: Even the shortest application outage was incredibly costly for Amazon. The data store simply had to remain available under all foreseeable circumstances.
- **Network partition tolerant**: As a global e-commerce vendor with customers and data centers all around the world, Amazon was most concerned that a network partition should not force a loss of availability, even if that loss of availability was isolated to a particular geography.
- **No-loss conflict resolution**: Another of Amazon's key requirements was that no order or shopping cart update should ever be lost. So for instance, if the user added items to his or her shopping cart from two different computers, both items should show up in the final cart. Furthermore, there should be no circumstances under which someone was blocked from adding an item to his or her cart, which implied that there be no exclusive write locks on objects.
- **Efficiency**: The system needed to respond quickly, since it was well understood that even small delays in website response time resulted in a significant reduction in online sales. Online customers were notoriously fickle and impatient.
- **Economy**: This system needed to be able to run on commodity hardware.
- **Incremental scalability**: It should be possible to grow the system by adding servers in small increments without manual maintenance or downtime.

Amazon was willing to compromise on a lot of features of existing databases in order to achieve these goals.

Principally, the data store should relax consistency—within limits—if necessary in order to ensure availability. The phrase "within limits" is important here: the system should favor availability over consistency, but in a predictable, controllable, and manageable way. Also, the trade-off between consistency and availability should be configurable—the application should be able to choose what happens if there is a network partition, for instance.

Additionally, the data store need only support primary key-based access and need not support a data model: the values retrieved by a key lookup would be unstructured binary objects. Unlike Google's BigTable, which had the design goal of storing massive files, the assumption for Dynamo was that most objects would be small—under 1 MB.

Dynamo—and many of the systems that it inspired—explicitly attempted to achieve a different outcome in terms of CAP theorem. Rather than try to always achieve consistency at the expense of network partition tolerance, Dynamo would allow (though not require) consistency to be sacrificed instead. See Figure 3-4.

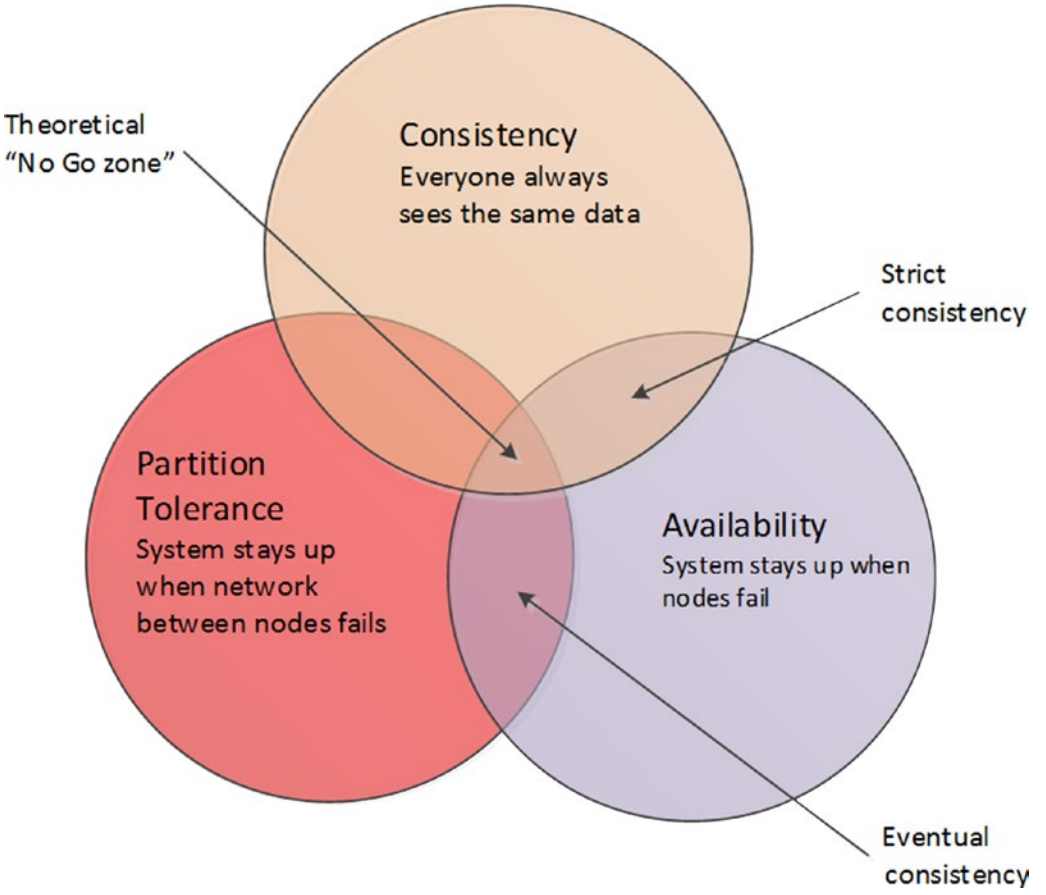


Figure 3-4. *Dynamo and ACID RDBMS mapped to CAP theorem limits*

Dynamo has served as an architectural model for quite a few nonrelational databases. (We'll look into the gory details of Dynamo internals in Chapters 8 and 9.) Some of the key architectural characteristics of Dynamo include:

- **Consistent hashing**: Consistent hashing is a scheme that uses the hash value of the primary key to determine the nodes in the cluster responsible for that key and that allows nodes to be added or removed from the cluster with minimal rebalancing overhead. See below for more details.
- **Tunable consistency**: The application can specify trade-offs between consistency, read performance, and write performance. It's possible in Dynamo to specify strong consistency, eventual consistency, or weak consistency. See below for more details.
- **Data versioning**: Since write operations will never be blocked, it is possible that there will be multiple versions of an object in the system. Sometimes these can be merged by the data store itself, but sometimes they will need to be resolved by the application or user. For instance, if the buyer updates his or her shopping cart from two computers, the resulting cart may have duplicate items that he or she may need to remove.

There are lots of complex design features in the Dynamo system; many of these will be discussed in Chapters 8 and 9. However, it would be remiss to continue without at least a little bit of elaboration on two of the key features: consistent hashing and tunable consistency.

Consistent Hashing

When we *hash* a key value, we perform a mathematical computation on the key value and use that computed value to determine where to store the data. One reason to use hashing is so that we are able to evenly distribute the data across a certain number of slots. The most simple example is to use the *modulo* function, which returns the remainder of a division. If we want to hash any number into 10 buckets, we can use modulo 10; then key 27 would map to bucket 7, key 32 would map to bucket 2, key 25 to bucket 5, and so on.

Using this method, we could map keys evenly across 10 servers. When we want to determine which node should store a particular item, we would calculate its modulo and use the result to locate the node. In practice, hashing functions are more complex than a simple modulo function, and a good hash function always distributes the hash values evenly across nodes, regardless of any skew in key values.

Hashing works great as a way of distributing data evenly across a fixed number of nodes. But we have a problem if we add or remove a node—we have to recalculate the hash values and redistribute all the data. For instance, if we wanted to add a new server in the modulo 10 example above, we would recalculate hashes using modulo 11 and then we would have to move almost every data item accordingly. *Consistent hashing* works by hashing key values and applying a consistent method for allocating those hashed values to specific nodes.

By convention and possibly federal law, consistent hashing schemes are represented as rings— because the hash values "loop around" to 0. Figure 3-5 shows what happens when a node is added to an existing cluster. Only those keys currently mapped to the "neighbors" of the new node need remapping.

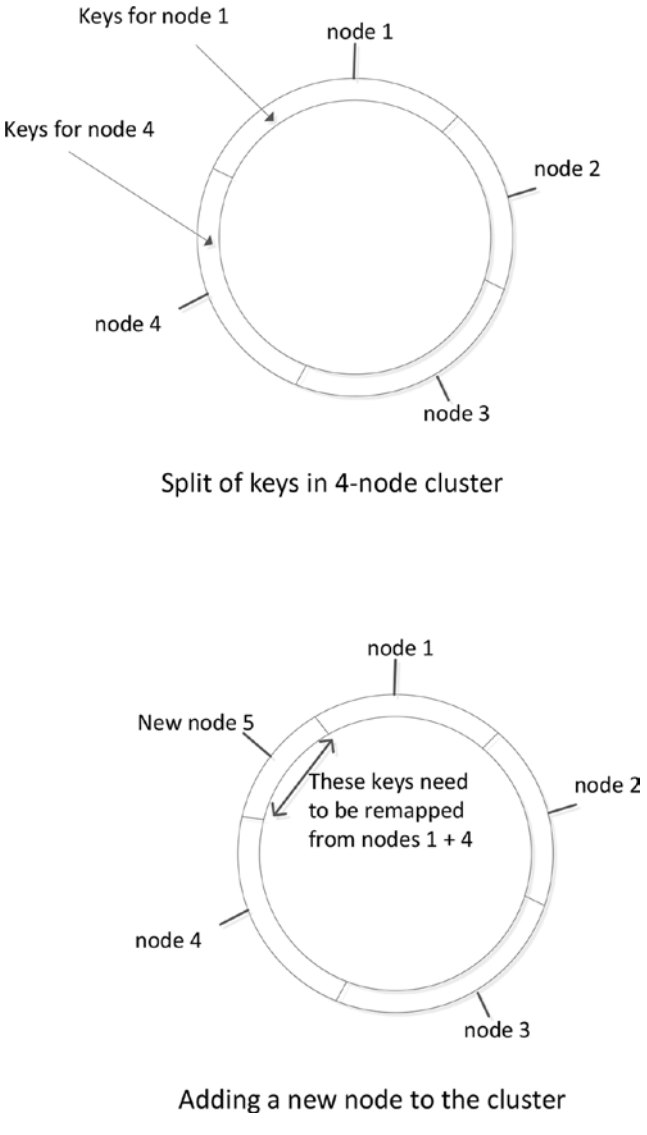


Figure 3-5. *Adding a new node to the consistent hashing scheme*

The remapping process when a new node is added is still an expensive operation, and in practice Dynamo-based databases often employ a "virtual nodes" workaround to further reduce the overhead. This mechanism is explained in detail in Chapter 8.

Tunable Consistency

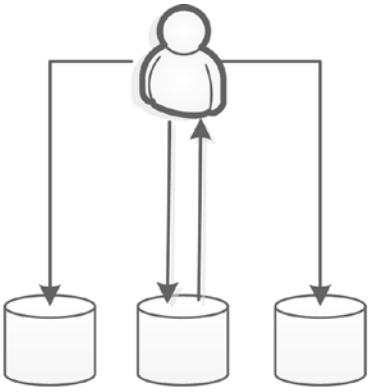
Dynamo allows the application to choose the level of consistency applied to specific operations. *NWR notation* describes how Dynamo will trade off consistency, read performance, and write performance:

- **N** is the number of copies of each data item that the database will maintain.
- **W** is the number of copies of the data item that must be written before the write can complete.
- **R** is the number of copies that the application will access when reading the data item.

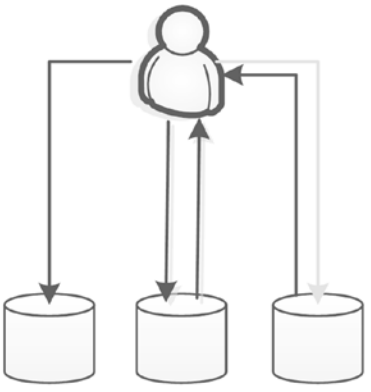
When W = N, Dynamo will always write every copy before returning control to the application—this is what ACID databases do when implementing synchronous replication. If the application is more concerned about write performance than read performance, then it could set W = 1, R = N. Then each read must access all copies to determine which is correct, but each write only has to touch a single copy of the data before returning control (other writes propagate to all copies as a background task).

Probably the most common configuration is N > W > 1. More than one write must complete, but not all nodes need to be updated immediately. Another common setting is W + R > N; this ensures that the latest value will always be included in a read operation, even if it is mixed in with "older" values. This is sometimes referred to as *quorum assembly*.

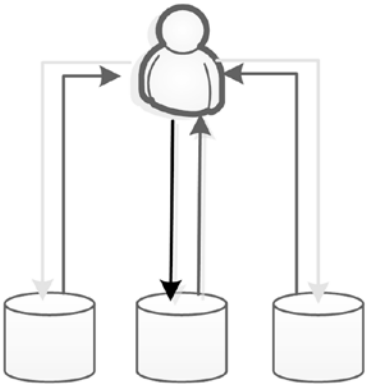
Figure 3-6 shows examples of various NWR settings. Depending on the settings, Dynamo can trade off consistency, reliability, and performance.



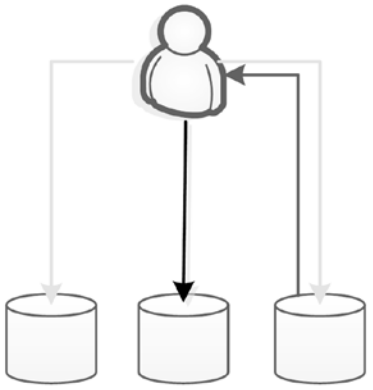
N=3 W=3 R=1
Slow writes, fast reads, consistent
There will be 3 copies of the data.
A write request only returns when all 3 have written to disk.
A read request only needs to read one version.



N=3 W=2 R=2
Faster writes, still consistent (quorum assembly)
There will be 3 copies of the data.
A write request returns when 2 copies are written – the other can happen later.
A read request reads 2 copies make sure it has the latest version.



N=3 W=1 R=N
Fastest write, slow but consistent reads
There will be 3 copies of the data.
A write request returns once the first copy is written – the other 2 can happen later.
A read request reads all copies to make sure it gets the latest version.
Data might be lost if a node fails before the second write.



N=3 W=1 R=1
Fast, but not consistent
There will be 3 copies of the data.
A write request returns once the first copy is written – the other 2 can happen later.
A read request reads a single version only: it might not get the latest copy.
Data might be lost if a node fails before the second write.

Figure 3-6. *Tunable consistency in Dynamo*

Dynamo and the Key-value Store Family

Systems that implement one of the primary characteristics of Dynamo—the idea of a binary value retrieved by primary key only—became generically known as *key–value stores*. Just as Google's GFS and MapReduce papers became the blueprint for Hadoop, Amazon's Dynamo paper became the blueprint for many key-value stores.

Web developers who were struggling with the operational complexity entailed by sharding and other heroic database techniques had already started experimenting with various nonrelational designs. However, with the release of Amazon's Dynamo paper, these developers had a proven architectural model to build on. The result was a relatively sudden burst of Dynamo-inspired systems in 2008–2009. These systems were the first recognizable NoSQL databases.

It's true that not all key–value stores were based explicitly on the Dynamo model. However, the list of Dynamo-inspired systems is impressive, and it includes active databases such as Riak, LinkedIn's Voldemort, Cassandra, and of course, Amazon's own DynamoDB. Some of these systems, such as Riak and Voldemort, are more or less exact copies of the Dynamo architecture, while others use Dynamo in conjunction with other concepts. For instance, Apache's Cassandra implements Dynamo's consistent hashing and tunable consistency models, combined with a variation on the BigTable data model.

Although Dynamo represents the most popular and well-articulated key-value store architecture, there are certainly key-value stores that owe little or nothing to the Dynamo design. These include systems such as Redis and Oracle NoSQL.

Conclusion

We've seen that the challenge of maintaining highly available global websites proved inconsistent with the ACID transaction model. Brewer's CAP theorem—as well as practical experience—argue that a system cannot aspire to both strong consistency and global availability in the event of an imperfect network like the Internet. For most massive websites, continual availability may be more important than perfect consistency.

Attempts to deploy relational databases on the scale required by the largest web properties involved the use of caching (Memcached, in particular), read-only replication, and sharding. This architectural pattern effectively broke both the relational and the ACID properties of the database: once a database has been sharded, ACID consistency and ad hoc SQL query access are lost. Nevertheless, the sharding solution has proved to be effective at sites such as Twitter and Facebook.

Amazon—the pioneer of Internet retailers—abandoned the RDBMS as the core database in favor of an internally developed, nonrelational key-value store called Dynamo. Dynamo implements eventual consistency rather than strict consistency; updates to data are eventually guaranteed to be propagated throughout the system, but may not be seen by every user instantaneously.

Dynamo has been a strong influence on the design of many other key-value stores, such as Riak and Cassandra, and is the basis for Amazon's cloud-based database DynamoDB. Systems such as Dynamo enforce no structure on their payload. This makes them impenetrable to all but programmers. As we will see in the next chapter, another class of databases—the document databases—extend the key-value concept by requiring that values be structured in a self-describing format such as XML or JSON.

Note

1. http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

CHAPTER 4

Document Databases

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

—Object-oriented data community, mid-1990s

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

—David Ensor, same period

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

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

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

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

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

FIGURE 15.12 PHP CODE TO QUERY THE VENDOR TABLE

```
1  <HTML>
2  <HEAD>
3  <TITLE>Rob & Coronel -  PHP Example</TITLE>
4  </HEAD>
5  <BODY BGCOLOR="LIGHTBLUE">
6  <H1><CENTER><B>Simple Query using PHP and ODBC functions</B></CENTER>
7  <CENTER><B>(Vertical Output)</B></CENTER></H1>
8  <BR>
9  <HR>
10 <?php
11 $dbc = odbc_connect("RobCor","","");
12 $sql = "SELECT * FROM VENDOR ORDER BY VEN_CODE";
13 $rs  = odbc_exec( $dbc, $sql );
14
15 while (odbc_fetch_row( $rs ))
16 {
17 $VEN_CODE          = odbc_result($rs,"VEN_CODE");
18 $VEN_NAME          = odbc_result($rs,"VEN_NAME");
19 $VEN_CONTACT_NAME  = odbc_result($rs,"VEN_CONTACT_NAME");
20 $VEN_ADDRESS       = odbc_result($rs,"VEN_ADDRESS");
21 $VEN_CITY          = odbc_result($rs,"VEN_CITY");
22 $VEN_STATE         = odbc_result($rs,"VEN_STATE");
23 $VEN_ZIP           = odbc_result($rs,"VEN_ZIP");
24 $VEN_PH            = odbc_result($rs,"VEN_PH");
25 $VEN_FAX           = odbc_result($rs,"VEN_FAX");
26 $VEN_EMAIL         = odbc_result($rs,"VEN_EMAIL");
27 $VEN_CUS_ID        = odbc_result($rs,"VEN_CUS_ID");
28 $VEN_SUPPORT_ID    = odbc_result($rs,"VEN_SUPPORT_ID");
29 $VEN_SUPPORT_PH    = odbc_result($rs,"VEN_SUPPORT_PH");
30 $VEN_WEB_PAGE      = odbc_result($rs,"VEN_WEB_PAGE");
31
32 echo "<BR>";
33 echo "VENDOR CODE:     ". $VEN_CODE . "<BR>";
34 echo "VENDOR NAME:     ". $VEN_NAME . "<BR>";
35 echo "CONTACT PERSON:  ". $VEN_CONTACT_NAME . "<BR>";
36 echo "ADDRESS:         ". $VEN_ADDRESS . "<BR>";
37 echo "CITY:            ". $VEN_CITY . "<BR>";
38 echo "STATE:           ". $VEN_STATE . "<BR>";
39 echo "ZIP:             ". $VEN_ZIP . "<BR>";
40 echo "PHONE:           ". $VEN_PH . "<BR>";
41 echo "FAX:             ". $VEN_FAX . "<BR>";
42 echo "E-MAIL:          ". $VEN_EMAIL . "<BR>";
43 echo "CUSTOMER ID:     ". $VEN_CUS_ID . "<BR>";
44 echo "SUPPORT ID:      ". $VEN_SUPPORT_ID . "<BR>";
45 echo "SUPPORT PHONE:   ". $VEN_SUPPORT_PH . "<BR>";
46 echo "VENDOR WEB PAGE: ". $VEN_WEB_PAGE . "<BR>";
47 echo "<HR>";
48 }
49 odbc_close($dbc);
50 ?>
51 </BODY>
52 </HTML>
```

In the figure, note that PHP uses multiple tags to query and display the data returned by the query. Take a closer look at the PHP functions:

- The **odbc_connect** function (line 11) opens a connection to the ODBC data source. A handle to this database is set in the $dbc variable.
- The **odbc_exec** function (line 13) executes the SQL query stored in the $sql variable against the $dbc database connection. The query's result set is stored in the $rs variable.
- The **while** function (line 15) loops through the result set ($rs) and uses the ODBC_FETCH_ROW function to get one row at a time from the result set. Notice that PHP variables start with the dollar sign ($).
- The **odbc_result** function (lines 17–30) gets a column value from a row in the result set and stores it in a variable. This function extracts the different values for each field to be displayed and stores them in variables.
- The **echo** function (lines 32–47) outputs text to the webpage using the variables defined in the previous lines. You can also combine text (HTML code) and PHP variables (lines 33–46) using the "." delimiter.
- The **odbc_close** function closes the database connection.

The previous examples are just two of the many ways you can interface webpages and databases to web applications. These examples only scratch the surface of the multiple features that web application servers provide.

Current-generation systems involve more than just the development of web-enabled database applications. They also require applications that can communicate with each other and with other systems not based on the web. Clearly, systems must be able to exchange data in a standard-based format. That is the role of XML.

15-3 Extensible Markup Language (XML)

Companies use the Internet to generate business transactions and integrate data to increase efficiency and reduce costs. These transactions are known as electronic commerce (e-commerce); it enables all types of organizations to sell products and services to a global market. E-commerce transactions—the sale of products or services—can take place between businesses (business-to-business, or B2B) or between a business and a consumer (business-to-consumer, or B2C).

Online Content

To learn more about e-commerce, consult Appendix I, Databases in Electronic Commerce, at *www.cengagebrain.com*.

Most e-commerce transactions take place between businesses. Because B2B e-commerce integrates business processes among companies, it requires the transfer of business information among different business entities. However, the way in which businesses represent, identify, and use data tends to differ substantially from company to company. As a simple example, some companies use the term *product code*, while others use *item ID*.

Until recently, a purchase order traveling over the web was expected to be in the form of an HTML document. The HTML webpage displayed on the web browser would include formatting as well as the order details. HTML **tags** describe how something *looks* on the webpage, such as typefaces and heading styles, and they often come in pairs to start and end formatting features. For example, the following tags in angle brackets would display FOR SALE in bold Arial font:

FOR SALE

tag
In markup languages such as HTML and XML, a command inserted in a document to specify how the document should be formatted. Tags are used in server-side markup languages and interpreted by a web browser for presenting data.

Extensible Markup Language (XML)
A meta-language used to represent and manipulate data elements. Unlike other markup languages, XML permits the manipulation of a document's data elements. XML facilitates the exchange of structured documents such as orders and invoices over the Internet.

If an application needs to get the order data from the webpage, there is no easy way to extract details such as the order number, date, customer number, product code, quantity, or price from an HTML document. The HTML document can only describe how to display the order in a web browser; it does not permit the manipulation of the order's data elements. To solve that problem, a new markup language known as Extensible Markup Language was developed.

Extensible Markup Language (XML) is a meta-language used to represent and manipulate data elements. XML is designed to facilitate the exchange of structured documents, such as orders and invoices, over the Internet. The World Wide Web Consortium (W3C) published the first XML 1.0 standard definition in 1998, setting the stage for giving XML the real-world appeal of being a true vendor-independent platform. It is not surprising that XML has rapidly become the data exchange standard for e-commerce applications.

The XML meta-language allows the definition of new tags, such as <ProdPrice>, to describe the data elements used in an XML document. This ability to *extend* the language explains the *X* in XML; the language is said to be *extensible*. XML is derived from the Standard Generalized Markup Language (SGML), an international standard for the publication and distribution of highly complex technical documents. For example, documents used by the aviation industry and the military services are too complex and unwieldy for the web. Just like HTML, which was also derived from

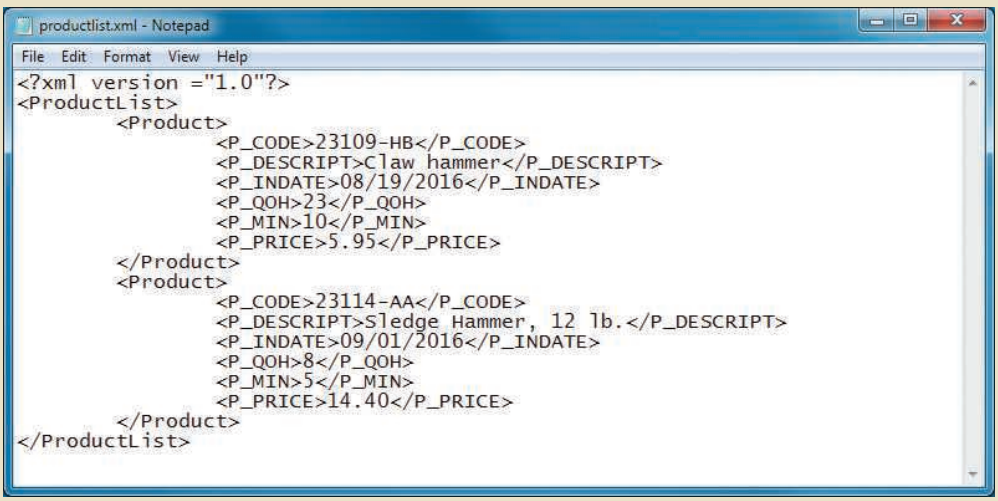
SGML, an XML document is a text file. However, it has a few important additional characteristics:

- XML allows the definition of new tags to describe data elements.
- XML is case sensitive: <ProductID>is not the same as <Productid>.
- XML must be well formed; that is, tags must be properly formatted. Most openings also have a corresponding closing. For example, a product's identification would require the format <ProductId>2345-AA</ProductId>.
- XML must be properly nested. For example, properly nested XML might look like this: <Product><ProductId>2345-AA</ProductId></Product>.
- You can use the <-- and --> symbols to enter comments in the XML document.
- The *XML* and *xml* prefixes are reserved for XML only.

XML is *not* a new version or replacement for HTML. XML is concerned with the description and representation of the data, rather than the way the data is displayed. XML provides the semantics that facilitate the sharing, exchange, and manipulation of structured documents over organizational boundaries. XML and HTML perform complementary functions rather than overlapping functions. Extensible Hypertext Markup Language (XHTML) is the next generation of HTML based on the XML framework. The XHTML specification expands the HTML standard to include XML features. Although it is more powerful than HTML, XHTML requires strict adherence to syntax requirements.

To illustrate the use of XML for data exchange purposes, consider a B2B example in which Company A uses XML to exchange product data with Company B over the Internet. Figure 15.13 shows the contents of the productlist.xml document.

FIGURE 15.13 CONTENTS OF THE PRODUCTLIST.XML DOCUMENT



The preceding example illustrates several important XML features:

- The first line represents the XML document declaration, and it is mandatory.
- Every XML document has a *root element*. In the example, the second line declares the ProductList root element.

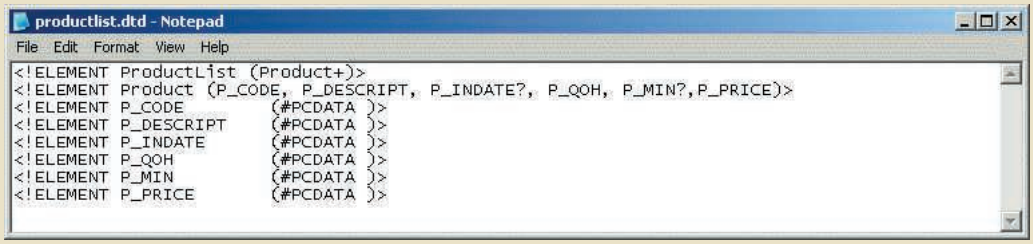
- The root element contains *child elements* or subelements. In the example, line 3 declares Product as a child element of ProductList.
- Each element can contain *subelements*. For example, each Product element is composed of several child elements, represented by P_CODE, P_DESCRIPT, P_INDATE, P_QOH, P_MIN, and P_PRICE.

Once Company B receives productlist.xml, it can process the document, assuming that it understands the tags created by Company A. The meaning of the XML in Figure 15.13 is fairly self-evident, but there is no easy way to validate the data or to check whether the data is complete. For example, you could encounter a P_INDATE value of "25/14/2016," but is that value correct? What happens if Company B expects a Vendor element as well? How can companies share data descriptions about their business data elements? The next section shows how document type definitions and XML schemas are used to address such concerns.

15-3a Document Type Definitions (DTD) and XML Schemas

Companies that use B2B transactions must have a way to understand and validate each other's tags. One way to accomplish that task is through the use of document type definitions. A **document type definition (DTD)** is a file with a .dtd extension that describes XML elements—in effect, a DTD file provides the composition of the database's logical model and defines the syntax rules or valid elements for each type of XML document. (The DTD component is similar to having a public data dictionary for business data.) Companies that intend to engage in e-commerce transactions must develop and share DTDs. Figure 15.14 shows the productlist.dtd document for the productlist.xml document shown earlier in Figure 15.13.

FIGURE 15.14 CONTENTS OF THE PRODUCTLIST.DTD DOCUMENT



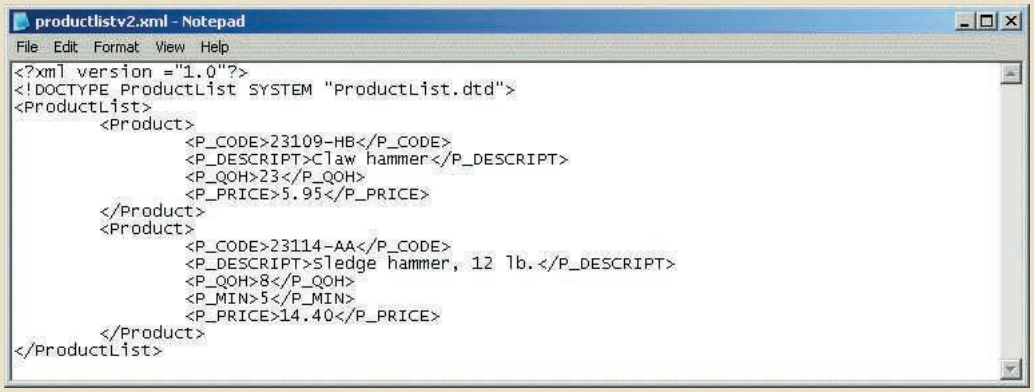
In Figure 15.14, the productlist.dtd file provides definitions of the elements in the productlist.xml document. In particular, note the following:

- The first line declares the ProductList root element.
- The ProductList root element has one child, the Product element. The second line describes the Product element.
- The plus symbol (+) indicates that Product occurs one or more times within ProductList.
- An asterisk (*) would mean that the child element occurs zero or more times.
- The question mark (?) after P_INDATE and P_MIN indicates that they are optional child elements.
- The third through eighth lines show that the Product element has six child elements.
- The #PCDATA keyword represents the actual text data.

document type definition (DTD)
A file with a .dtd extension that describes XML elements; in effect, a DTD file describes a document's composition and defines the syntax rules or valid tags for each type of XML document.

To be able to use a DTD file to define elements within an XML document, the DTD must be referenced within that XML document. Figure 15.15 shows the productlistv2.xml document that includes the reference to productlist.dtd in the second line.

FIGURE 15.15 CONTENTS OF THE PRODUCTLISTV2.XML DOCUMENT



In Figure 15.15, note that P_INDATE and P_MIN do not appear in all Product definitions because they were declared to be optional elements. The DTD can be referenced by many XML documents of the same type. For example, if Company A routinely exchanges product data with Company B, it will need to create the DTD only once. All subsequent XML documents will refer to the DTD, and Company B will be able to verify the data being received.

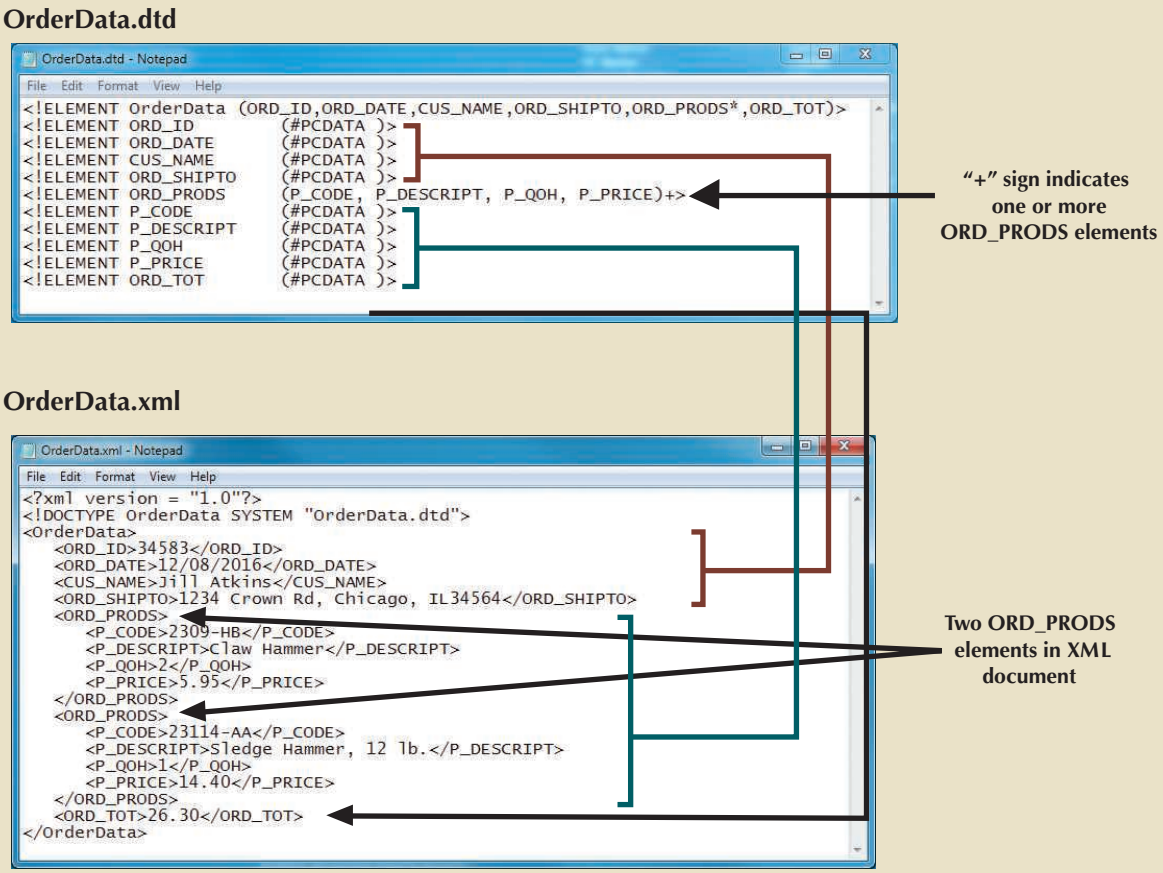
To further demonstrate the use of XML and DTD for e-commerce data exchanges, consider the case of two companies exchanging order data. Figure 15.16 shows the DTD and XML documents for that scenario.

Although the use of DTDs is a great improvement for data sharing over the web, a DTD only provides descriptive information for understanding how the elements—root, parent, child, mandatory, or optional—relate to one another. A DTD provides limited additional semantic value, such as data type support or data validation rules. That information is very important for database administrators who are in charge of large e-commerce databases. To solve the DTD problem, the W3C published an XML schema standard that better describes XML data.

The **XML schema** is an advanced data definition language that is used to describe the structure of XML data documents. This structure includes elements, data types, relationship types, ranges, and default values. One of the main advantages of an XML schema is that it more closely maps to database terminology and features. For example, an XML schema can define common database types such as date, integer, or decimal; minimum and maximum values; a list of valid values; and required elements. Using the XML schema, a company would be able to validate data for values that may be out of range, have incorrect dates, contain invalid values, and so on. For example, a university application must be able to specify that a GPA value is between 0 and 4.0, and it must be able to detect an invalid birth date such as "14/13/2016." (There is no 14th month.) Many vendors are adopting this new standard and are supplying tools to translate DTD documents into XML schema definition documents. It is widely expected that XML schemas will replace DTD as the method to describe XML data.

XML schema
An advanced data definition language used to describe the elements, data types, relationship types, ranges, and default values of XML data documents. One of the main advantages of an XML schema is that it more closely maps to database terminology and features.

FIGURE 15.16 DTD AND XML DOCUMENTS FOR ORDER DATA



Unlike a DTD document, which uses a unique syntax, an **XML schema definition (XSD)** file uses a syntax that resembles an XML document. Figure 15.17 shows the XSD document for the OrderData XML document.

The code shown in Figure 15.17 is a simplified version of the XML schema document. As you can see, the XML schema syntax is similar to the XML document syntax. However, the XML schema introduces additional semantic information for the OrderData XML document, such as string, date, and decimal data types; required elements; and minimum and maximum cardinalities for the data elements.

15-3b XML Presentation

One of the main benefits of XML is that it separates data structure from its presentation and processing. By separating the two, you can present the same data in different ways—which is similar to having views in SQL. The Extensible Style Language (XSL) specification provides the mechanism to display XML data. *XSL* is used to define the rules by which XML data is formatted and displayed. The XSL specification is divided into two parts: Extensible Style Language Transformations (XSLT) and XSL style sheets.

XML schema definition (XSD)
A file that contains the description of an XML document.

- *Extensible Style Language Transformations (XSLT)* describes the general mechanism that is used to extract and process data from one XML document and enable its transformation within another document. Using XSLT, you can extract data from an XML document and convert it into a text file, an HTML webpage, or a webpage that is formatted

FIGURE 15.17 THE XML SCHEMA DOCUMENT FOR THE ORDER DATA

OrderData.xsd - Notepad

File Edit Format View Help

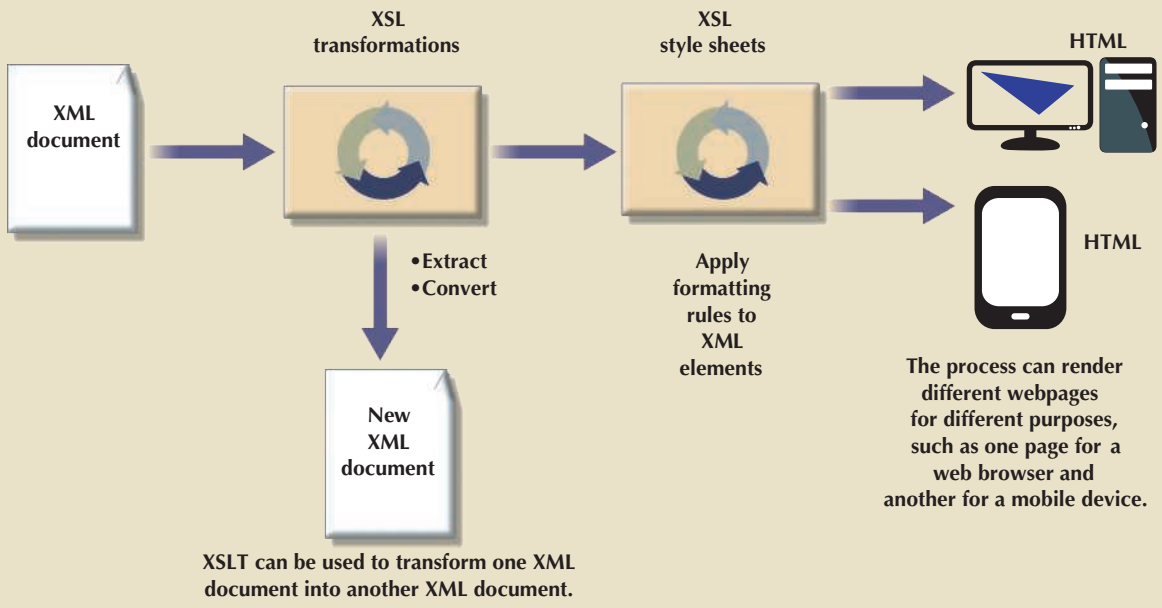
```
<xsd:schema xmlns:xsd="http://www.company.com/xmlschema">

<xsd:element name="orderData" type="order"/>

<xsd:complexType name="order">
        xsd:element name="ORD_ID"       type="xsd:string/>
        xsd:element name="ORD_DATE"     type="xsd:date/>
        xsd:element name="ORD_NAME"     type="xsd:string/>
        xsd:element name="CUS_NAME"     type="xsd:string/>
        xsd:element name="ORD_SHIPTO"   type="xsd:string/>
        xsd:element name="ORD_PRODS"    type="xsd:productlist/>
        xsd:element name="ORD_TOT"      type="xsd:decimal/>
</xsd:complexType name="order">

<xsd:complexType name="prodlist"
        <xsd:element name="product"     type="aproduct" minOccurs="1" maxOccurs="unbounded"/>
</xsd:complexType>

<xsd:complexType name="aproduct">
        xsd:element name="P_CODE"       type="xsd:string"              use="required"/>
        xsd:element name="P_DESCRIPT"   type="xsd:string"              use="required"/>
        xsd:element name="P_QOH"        type="xsd:positiveInteger"     use="required"/>
        xsd:element name="P_PRICE"      type="xsd:decimal              use="required"/>
</xsd:complexType>

</xsd:schema>
```

for a mobile device. What the user sees in those cases is actually a view (or HTML representation) of the XML data. XSLT can also be used to extract certain elements from an XML document, such as product codes and product prices, to create a product catalog. XSLT can even be used to transform one XML document into another.

- *XSL style sheets* define the presentation rules applied to XML elements—somewhat like presentation templates. The XSL style sheet describes the formatting options to apply to XML elements when they are displayed on a browser, smartphone, tablet screen, and so on.

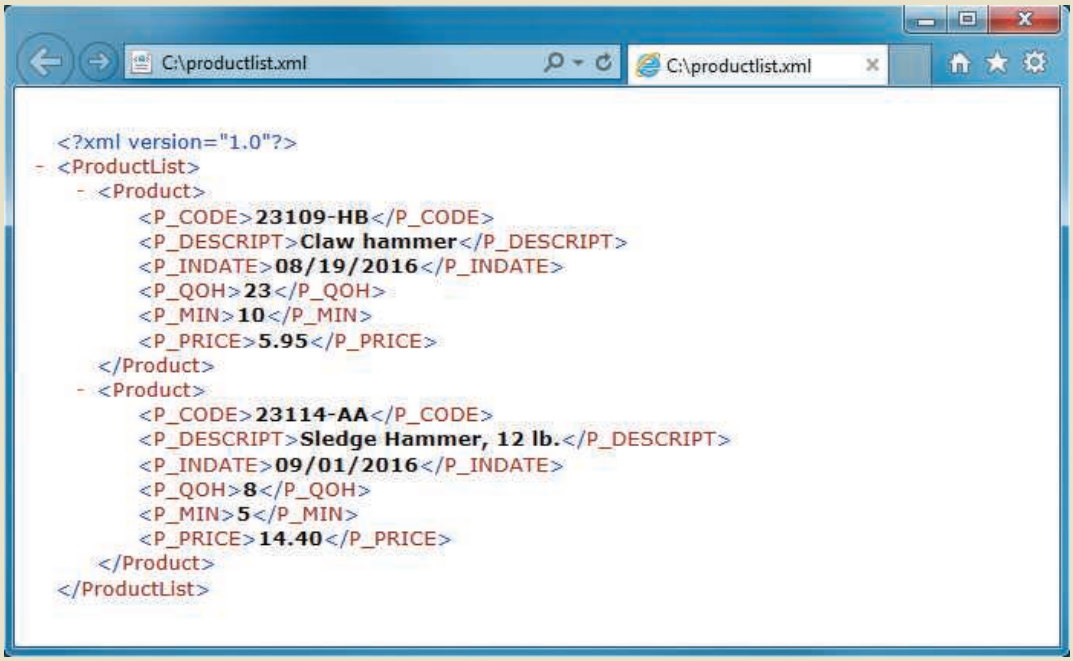
Figure 15.18 illustrates the framework used by the various components to translate XML documents into viewable webpages, an XML document, or some other document.

FIGURE 15.18 FRAMEWORK FOR XML TRANSFORMATIONS



To display the XML document with Windows Internet Explorer (IE), enter the URL of the XML document in the browser's address bar. Figure 15.19 is based on the product-list.xml document created earlier. As you examine Figure 15.19, note that IE shows the XML data in a color-coded, collapsible, tree-like structure. (Actually, this is the IE default style sheet that is used to render XML documents.)

FIGURE 15.19 DISPLAYING XML DOCUMENTS



15-3c XML Applications

Now that you have some idea what XML is, how can you use it? What kinds of applications lend themselves particularly well to XML? This section lists some of the uses of XML. Keep in mind that the future use of XML is limited only by the imagination and creativity of developers, designers, and programmers.

- *B2B exchanges*. XML enables the exchange of B2B data, providing the standard for all organizations that need to exchange data with partners, competitors, the government, or customers. In particular, XML is positioned to replace EDI as the standard for automation of the supply chain because it is less expensive and more flexible.
- *Legacy systems integration*. XML provides the "glue" to integrate legacy system data with modern e-commerce web systems. Web and XML technologies could be used to inject some new life into old but trusted legacy applications. Another example is the use of XML to import transaction data from multiple databases to a data warehouse database.
- *Webpage development*. XML provides several features that make it a good fit for certain web development scenarios. For example, web portals with large amounts of personalized data can use XML to pull data from multiple external sources (such as news, weather, and stock sites) and apply different presentation rules to format pages on desktop computers as well as mobile devices.
- *Database support*. A DBMS that supports XML exchanges can integrate with external systems such as the web, mobile data, and legacy systems, thus enabling the creation

of new types of systems. These databases can import or export data in XML format or generate XML documents from SQL queries while still storing the data using their native data model format. An example is the use of the FOR XML clause in the SQL SELECT statement in SQL Server. Alternatively, a DBMS can also support an XML data type to store XML data in its native format—enabling support to store tree-like hierarchical structures inside a relational structure.

- *Database metadictionaries*. XML is also used to create metadictionaries, or vocabularies, for entire industries. Examples of metadictionaries include HR-XML for the human resources industry, the metadata encoding and transmission standard (METS) from the Library of Congress, the clinical accounting information (CLAIM) data exchange standard for patient data exchange in electronic medical record systems, and the extensible business reporting language (XBRL) standard for exchanging business and financial information.
- *XML databases*.[1] Most databases on the market support XML to manage data in some shape or form. The approaches range from simple middleware XML software to object databases with XML interfaces to full XML database engines and servers. XML databases provide for the storage of data in complex relationships. For example, an XML database would be well suited to store the contents of a book. The book's structure would dictate its database structure: a book typically consists of chapters, sections, paragraphs, figures, charts, footnotes, endnotes, and so on. Examples of databases with XML data type support are Oracle, IBM DB2, and MS SQL Server. Fully XML databases examples are Tamino from Software AG *(www.softwareag.com)* and the open source dbXML from *http://sourceforge.net/projects/dbxml-core*.
- *XML services*. Many companies are already working to develop a new breed of services based on XML and web technologies. These services break down the interoperability barriers among systems and companies alike. XML provides the infrastructure that helps heterogeneous systems to work together across the desk, the street, and the world. Services would use XML and other Internet technologies to publish their interfaces. Other services that want to interact with existing services would locate them and learn their vocabulary (service request and replies) to establish a "conversation."

One area in which Internet, web, virtualization, and XML technologies work together in innovative ways to leverage IT services is cloud computing.

15-4 Cloud Computing Services

You have almost certainly heard about the "cloud" from the thousands of publications and TV ads that have used the term over the years, although it has represented different concepts. In the late 1980s, the term *cloud* was used by telecommunication companies to describe their data networks. In the late 1990s, during the peak of Internet growth, the term depicted the Internet itself. Then, in 2006, Google and Amazon began using the term *cloud computing* to describe a new set of innovative web-based services. Google, Yahoo, eBay, and Amazon were early adopters of this new computing paradigm.

But what exactly is cloud computing? According to the National Institute of Standards and Technology (NIST),[2] **cloud computing** is "a computing model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computer resources

cloud computing
A computing model that provides ubiquitous, on-demand access to a shared pool of configurable resources that can be rapidly provisioned.

[1] For a comprehensive analysis of XML database products, see "XML Database Products" by Ronald Bourret at *www.rpbourret.com*.

[2] *Recommendations of the National Institute of Standards and Technology*, Peter Mell and Timothy Grance, Special Publication 800–145 (Draft), January 2011.