This article is taken from the book The Cloud at Your Service. The authors discuss the concept of sharding, a term invented by and a concept heavily employed by Google that is about how to partition databases so they scale infinitely. They show how Yahoo!, Flickr, Facebook, and many other sites that deal with huge user communities have been using this method of database partitioning to enable a good user experience concurrently with rapid growth.

Get 35% off any version of The Cloud at Your Service, with the checkout code fcc35. Offer is only valid through http://www.manning.com.

When you think Internet scale—by which we mean exposure to the enormous population of entities (human and machine) any number of which could potentially access our application—any number of very popular services might appropriately come to mind. Google is the obvious king of the realm, but Facebook, Flickr, Yahoo!, and many others have had to face the issues of dealing with scaling to hundreds of millions of users. In every case, each of these services was throttled at some point by the way they used the database where information about their users was being accessed. These problems were so severe that major reengineering was required, sometimes many times over. That’s a good lesson to learn early as you are thinking about how your Internet scale applications should be designed in the first place.

**Application issues that prevent scaling**

Experience from the very best in the business—many of whom had to completely reengineer their very successful services—and the failed ones who were not able to do so in time highlights the two most impactful issues that prevent applications from achieving Internet scale. The first is about a very large working set size that cannot fit into memory; if very commonly accessed data requires going back to disk it is clear that the application cannot...
scale. The second is about database updates that are so frequent or so large or both that the I/O system throttles the ability to scale.

**Very large working sets**

If the amount of memory required to keep your frequently accessed data loaded in memory exceeds what you can (economically) fit in a commodity machine, your application has a very large working set. Five years ago this was 4GB, today it is 128GB or even 256GB. Note this need not be the same size (or even near) the size of your entire database. That is, assuming good database schema and indexing, it doesn’t.

**Too many writes**

If either the I/O system or a slave working on its behalf cannot keep up with the amount of writes—recording the data—being sent to the server, there are too many writes for the system to scale any further. The I/O system is throttling the application. While the I/O system can be improved with a RAID approach (perhaps by 5X or even 10X), the slave delay problem is actually very hard to solve and it only would delay when the throttle point is hit.

**Solution: Partition the data**

The solution is to partition the data. That is easy to say and extremely hard to do. The simple description of what needs to be done follows:

Split the data between multiple machines and have a way to make sure that you always access data from the right place. For example, consider this simple scheme. Imagine you want to store data about customers, each of whom has a "last name" field in your database. One partitioning scheme is to create 26 identical databases and assign each of them a letter of the alphabet. Then, whenever you want to look up the data about John Smith, you first connect to the "S" database and then fetch the data you want. All of a sudden, your single database solution where all last names are stored just got twenty-six times more capacity because each of 26 databases has all last names of a single letter of the alphabet.

This is the essence of the process of partitioning a database—called *sharding*—and we will delve into this in much more detail next.

**Sharding defined: a parallel database architecture for massive scaling**

The term *sharding* was coined by Google engineers and popularized through their publication of the Big Table architecture. However, the concept of "shared-nothing" database partitioning on which sharding is based has been around for a decade or more and there have been many implementations over this period, especially high-profile in-house built solutions by Internet leaders such as eBay, Amazon, Digg, Flickr, Skype, YouTube, Facebook, Friendster, and Wikipedia.

**DEFINITION: SHARDING**

A decomposition of a database into multiple smaller units (called shards) that can handle requests individually. Related to a scalability concept called *shared nothing* that removes dependencies between portions of the application so that they can run completely independently and in parallel for much higher throughput.

There was a time when we scaled databases by buying ever bigger, faster, and more expensive machines. While this arrangement is great for big iron and database vendor profit margins, it doesn’t work so well for the providers of very popular Web-facing services that need to scale well past what they can afford to spend on giant database servers. This is where sharding comes in. It is a revolutionary new database architecture that, when implemented at Flickr, enabled that service to handle more than one billion transactions per day, responding to requests in less than a few seconds and that scales linearly at low cost. Does sound revolutionary doesn’t it? Let’s look at it more closely.

**Sharding in a nutshell**

In the simplest model for partitioning, shown in figure 5.1, the data for User A is stored on one server and the data for User B is stored on another server. It’s a federated model. In a system like Flickr, groups of 500K users are
stored together in what are called shards. We expand this simple 2-shard design in Figure 5.1 where the criterion for shard selection is odd- versus even-numbered users.

The more we examine sharding the more its advantages just seem to keep on coming:

- **High availability**
  
  If one box goes down the others still operate, so in the simple model, while some users will stop getting service, the rest of the service is unaffected. Normally, there is some replication, so that, even if a box with a shard goes down, those users can be serviced on the replicated server.

- **Faster queries.**
  
  Smaller amounts of data in each user group mean faster querying. Services with a lot of experience with their particular usage patterns and database structures learn how to best partition their database onto shards to keep queries fast enough and keep users satisfied.

- **More write bandwidth.**
  
  With no master database serializing writes, you can write to many shards in parallel. This increases your write throughput. Writing is the major bottleneck for many websites.

  But the bottom line is that sharding allows your application to do more work. It essentially provides a parallel backend, which means your application can do more work simultaneously. Now it can handle higher user loads, especially when writing data because there are parallel paths through your system on something that normally throttles performance, namely database writes. You can load balance web servers that access shards over different network paths processed by separate CPUs, which use separate caches of RAM and separate disk I/O paths to process work. The result is that very few bottlenecks limit your application’s performance. As if these arguments were not motivation enough, let’s look at a real example.

**Why shard (or partition) your database?**

Let’s take Facebook.com as an example. In early 2004, Harvard students still mostly used the site as an online yearbook. It’s not hard to believe in this case that a single beefy server handled the entire storage requirements and query load on the database. Fast forward to 2008, when just the Facebook application-related page views were about 14 billion a month. That translates to over 5,000 page views per second, each of which requires multiple backend queries to satisfy. Besides query load with its attendant IOP, CPU, and memory cost, there’s also storage capacity to consider. Today, Facebook stores 40 billion physical files to represent about 10 billion photos, which is over a petabyte of storage. Even though the actual photo files are likely not in a relational database, their metadata such as identifiers and locations still would require a few terabytes of storage to represent these photos in the database. You can be sure that the original database used by Facebook did not have terabytes of storage.
available just to store photo metadata. And, indeed, it should not have originally been designed that way. That is because you cannot know the exact usage patterns that will drive the best sharding strategy until you can observe user behaviors as they interact with your application.

At some point during the development of Facebook, they reached the physical capacity of their database server and probably suffered with a lot of user dissatisfaction while they worked to rearchitect. The question would have been whether they should scale vertically by buying a more expensive, beefier server with more RAM, CPU horsepower, disk I/O, and storage capacity or spread their data out across multiple relatively cheap database servers. As we discussed earlier, if your service has lots of rapidly changing data (i.e., lots of writes) or is sporadically queried by lots of users in a way that causes your working set not to fit in memory (i.e., lots of reads leading to lots of page faults and disk seeks), then your primary bottleneck will likely be I/O. This is typically the case with social media sites like Facebook, LinkedIn, Blogger, MySpace and even Flickr. In such cases, it is either prohibitively expensive or physically impossible to purchase a single server to handle the load on the site. In such situations, sharding the database, while challenging to design and implement correctly, provides the best performance and highest cost savings relative to the increased complexity of the system.

**How sharding changes an application**

In a well designed application, the primary change sharding adds to the core application code is that, instead of code that just opens a single database and then does a query such as this,

```csharp
string connectionString = ConfigurationSettings.AppSettings["ConnectionInfo"];
OdbcConnection conn = new OdbcConnection(connectionString);
conn.Open();

OdbcCommand cmd = new OdbcCommand("SELECT Name, Address FROM Customers
WHERE CustomerID= ?", conn);
OdbcParameter param = cmd.Parameters.Add("@CustomerID", OdbcType.Int);
param.Value = customerId;
OdbcDataReader reader = cmd.ExecuteReader();
```

the actual connection information about the database to which it should connect depends on the data we are trying to store or access. Notice here that there is a method `GetDatabaseFor()` that opens one of many databases based on a `customerId` parameter. So, instead, you'd have the following:

```csharp
string connectionString = GetDatabaseFor(customerId);
OdbcConnection conn = new OdbcConnection(connectionString);
conn.Open();

OdbcCommand cmd = new OdbcCommand("SELECT Name, Address FROM Customers
WHERE CustomerID= ?", conn);
OdbcParameter param = cmd.Parameters.Add("@CustomerID", OdbcType.Int);
param.Value = customerId;
OdbcDataReader reader = cmd.ExecuteReader();
```

The assumption here is that the `GetDatabaseFor()` method knows how to map a customer ID to a physical database location. Almost everything else should remain the same unless the application uses sharding as a way to parallelize queries. This seemingly very small change actually provides an application with a database architecture that is quite different from a traditional one.

**Sharding in contrast with traditional database architectures**

To begin to really understand how different and how powerful sharding is, this topic will contrast a sharded database with a traditional one.

**Data are denormalized**

Traditionally, we normalize data, i.e., we pull it apart and relate data elements in a new record to the tables that are used to store the range of possible standard values. For example, if a user has a relationship status that only has a fixed set of possible values (e.g., single or married), normalized data would retain an index into the
relationship status table. In other words, data are splayed out into anomalyless tables and then joined back together again when they need to be used. However, in sharding the data are denormalized: you store together data that are used together. So every user record in this example will retain the relationship status with the record and not just an index into a single relationship status table.

This doesn't mean you don't also segregate data by type. You can keep a user's profile data separate from their comments, blogs, email, and media, but the user profile data would be stored and retrieved as a whole. While we don't know the top secret internal workings of Facebook, something like this must be employed based on observed behavior and performance of their service. This is a very fast approach. You just get a blob of data and store a blob of data. No joins are needed and it can be written with one disk write.

**Data are parallelized across many physical instances**

Historically database servers are scaled up as shown in Figure 2. You buy bigger machines to get more power. But it is not difficult to reach the limit of the server, the database, or both.

![Figure 2](http://www.manning.com/rosenberg/)  
*Figure 2* The traditional approach to scaling a database is to get bigger servers that drive bigger databases. But the system is throttled by how fast the server and the disk subsystem can handle writes and it quickly reaches its limits when dealing with Internet scale.

With sharding, the data are parallelized and you scale by scaling horizontally. Using this approach, you can get massively more work done because it can be done in parallel. And, as Figure 3 shows, there is no limit to how many databases can be put to work.

![Figure 3](http://www.manning.com/rosenberg/)  
*Figure 3* The sharding approach to database scaling moves more modest servers (usually based on cheap commodity hardware) with modest databases where each server and its associated database takes on a portion of the database load.

A good partitioning scheme will balance the load and will allow continued expansion as the application continues to need to scale.
**Data are kept small**
The larger a set of data a server handles the harder it is to cache intelligently because you have such a wide diversity of data being accessed. You need huge amounts of RAM that may not even be enough to cache the data when you need it. By isolating data into smaller shards, the data you are accessing is more likely to stay in cache. Smaller sets of data are also easier to backup, restore, and manage.

**Data are more highly available**
Since the shards are independent, a failure in one doesn't cause a failure in another. And, if you make each shard operate at 50% capacity, it's much easier to upgrade a shard in place. Keeping multiple data copies within a shard also helps with redundancy and makes the data more parallelized so more work can be done on the data. You can also set up a shard to have a master-slave replication (where the master database is the authoritative source and the slave databases are synchronized to it) or dual-master replication (where each server functions as both a master and a slave to the other server) to avoid a single point of failure within the shard. If one server goes down, the other can take over.

**Data are not replicated**
Replicating data from a master server to slave servers is a traditional approach to scaling. Data is written to a master server and then replicated to one or more slave servers. At that point read operations can be handled by the slaves, but all writes happen on the master.

Obviously, the master becomes the write bottleneck and a single point of failure. And, as the load increases, the cost of replication increases and affects CPU, network bandwidth, and disk I/O. The slaves fall behind and have stale data. The folks at YouTube had a big problem with replication overhead as they scaled.

Now that the sharding concept and its attributes have been described, we need to explore the various common approaches to partitioning databases into shards.

**Sharding in practice: the most common database partitioning schemes**
Continuing to “peel the sharding onion,” we need to discuss the most common types of sharding. The way the database is partitioned needs to match the characteristics of the application and the usage patterns it experiences. Do we separate out features each to their own database? Should we divide up the segments of users so each has a separate database? Is it best to use an even more sophisticated scheme because our system may need to be repartitioned over and over as it grows? These are choices that have to be made early and they have to be based on a real understanding of how your application is used.

**Vertical partitioning**
A simple way to segment your application database is to move tables related to specific features to their own server. For example, placing user profile information on one database server and friend lists on another and using a third for user-generated content like photos and blogs might make sense. Figure 4 shows a hypothetical social networking site (the real ones guard their inner architectures like state secrets) that employed vertical database partitioning.
Figure 4: A hypothetical social networking site that employed vertical (feature-based) partitioning when it implemented sharding to help it scale its application to ever larger numbers of users.

The key benefit of this approach is that it is straightforward to implement and has low impact to the application as a whole. However, the main problem with this approach is that, if the site experiences additional growth, then it may be necessary to further shard a feature-specific database across multiple servers (e.g., handling metadata queries for 10 billion photos by 300 million users may be more than a single server can handle; but not many services will see Facebook’s growth profile).

**Range based partitioning**

In situations where the entire data set for a single feature or table still needs to be further subdivided across multiple servers, it is important to ensure that the data is split up in a predictable manner. One approach to ensuring this predictability is to split the data based on value ranges in each entity. For example, splitting up sales transactions by the year they were created or assigning users to servers based on the first digit of their zip code. The main problem with this approach is that, if the value whose range is used for partitioning isn’t chosen carefully, then the sharding scheme leads to unbalanced servers. In the previous example, splitting up transactions by date means that the server with the current year gets a disproportionate amount of read and write traffic. Similarly, partitioning users based on their zip code assumes that your user base will be evenly distributed across the different zip codes, but this fails to account for situations where your application is popular in a particular region and the fact that human populations vary across zip codes.

**Key- or hash-based partitioning**

This is often a synonym for user-based partitioning for Web 2.0 sites. With this approach, each entity has a value that can be used as input into a hash function whose output is used to determine which database server to use. A simple example is to imagine you had ten database servers and your user IDs were a numeric value that was incremented by 1 each time a new user was added. In this example, the hash function could be to perform modulo operation on the user ID with the number 10 and then pick a database server based on the remainder value. This approach would ensure a uniform allocation of data to each server. The key problem with this approach is that it effectively fixes your number of database servers, since adding new servers means changing the hash function,
which, without downtime, is like being asked to change the tires on a moving car. This example illustrates the critical importance of thinking ahead when making sharding design decisions.

**TIPS FOR AVOIDING UNBALANCED SHARDING**

Avoid bad hashing algorithms: you don’t want to shard based on the first character of a username because there are a lot more ‘M’ than ‘Z’ names in our culture.

Avoid the problem of inequality among users: the day Sarah Palin was announced as the VP pick started down this path of a user being much more active than any others in any type of social networking service.

**Directory-based partitioning**

A loosely coupled approach to this problem is to create a lookup service that knows your current partitioning scheme and abstracts it away from the database access code. This means the `GetDatabaseFor()` method actually hits a web service or a database that stores/returns the mapping between each entity key and the database server it resides on. This loosely coupled approach means you can perform tasks like adding servers to the database pool or change your partitioning scheme without having to impact your application.

Consider the previous example, where there are ten servers and the hash function is modulo operation—the remainder after division of one number by another. In spite of modulo’s sheer simplicity, it has a very uniform distribution. Let’s say we want to add five database servers to the pool without incurring downtime. We can keep the existing hash function, add these servers to the pool, and then run a script that copies data from the ten existing servers to the five new servers based on a new hash function implemented by performing the modulo operation on user IDs using the new server count of fifteen. Once the data are copied over—although this is tricky since users are always updating their data—the lookup service can change to using the new hash function. None of the calling applications would be any wiser that their database pool just grew 50% and the database they went to for accessing John Doe’s pictures five minutes ago is different from the one they are accessing now.

Like any solution that creates a highly efficient layer of abstraction (or indirection) this is highly scalable, and once you write scripts to be able to migrate users to/from shards you can tweak and rebalance to make sure that all your hardware is utilized efficiently. The downside of this approach is that it is complicated. Speaking of complicated, in the next subtopic we will explore challenges and problems with sharding and begin to make it clear that while extremely powerful, sharding should not be used too early or too often.

**Sharding challenges and problems**

Sharding isn’t perfect. It does have a few problems not the least of which is that, as we have said, it is very complicated.

Once a database has been sharded, new constraints are placed on the operations that can be performed on the database. These constraints primarily center around the fact that operations across multiple tables or multiple rows in the same table no longer will run on the same server. Following are some of the constraints and additional complexities introduced by sharding.

**Rebalancing data**

What happens when a shard outgrows your storage and needs to be split? Let’s say some user has a particularly large friends list that blows your storage capacity for the shard. You need to move the user to a different shard. This can be a very big problem. If this is not designed very carefully, moving data from shard to shard may require a service shutdown.

Rebalancing has to be built in from the start. Google’s shards automatically rebalance. For this to work, data references must go through some sort of naming service so they can be relocated. Additionally, references must be invalidateable so the underlying data can be moved while you are using it. A simple example of this is shown in Figure 5.
Initially, the GetDatabaseFor() function pushed requests for partition A to the center server. But that server’s database shard has become too large and needs to be rebalanced. So all or a portion of database A is moved to the server on the left and once that data is successfully moved, GetDatabaseFor() is simply modified so that future requests for shard A now are directed to the leftmost server.

Using a scheme-like directory-based partitioning does make rebalancing a more palatable experience at the cost of increasing the complexity of the system and creating a new single point of failure (i.e., the lookup service/database).

**Joining data from multiple shards**

To create a complex friends page, or a user profile page, or a thread discussion page, you usually must pull together lots of different data from many different sources. With sharding, you can't just issue a query and get back all of the data. You have to make individual requests to your data sources, get the responses, and then build the page. Fortunately, because of caching and fast networks, this process is usually fast enough that your page load times can be excellent and, in the types of examples we are using, human response time is a pretty forgiving lower bound.

**Referential integrity**

As you can imagine, it is extremely difficult to enforce data integrity constraints such as foreign keys in a sharded database. Most relational database management systems do not support foreign keys across databases on different database servers. This means that sharded applications that require referential integrity often have to enforce it in application code and run regular SQL jobs to clean up dangling references. Consequently, dealing with data inconsistency issues due to denormalization and lack of referential integrity can become a significant development cost to the service.

**Little support**

Finally, the biggest problem with sharding may be the lack of experience and expertise you will find about it. People have experience with traditional RDBMS tools so there is a lot of help out there. There are thousands of books, experts, tool chains, and discussion forums when something goes wrong or you are wondering how to implement a new feature. The Eclipse IDE won't have a shard view any time soon and you won't find any automated backup and restore programs for your shard. With sharding, you are on your own although there are signs of hope coming. LiveJournal makes their tool chain available. Hibernate has a library under development. MySQL has added support for partitioning. But, in general, sharding is something you must implement yourself.

**Sharding in real life: how Flickr’s sharding works**

How Flickr implemented sharding will be a good way to understand the concept more deeply. This material is drawn from materials produced by Flickr’s CTO Cal Henderson and the Web site High Scalability at
http://highscalability.com, which is an excellent resource for sharding in practice and other topics relating to scaling to enormous levels. We begin by examining the overall profile of the Flickr service as outlined in Figure 6.

![Flickr](image)

- More than 4 billion queries per day.
- ~35M photos in squid cache (an open source Web delivery optimizing system)
- ~2M photos in squid's RAM
- ~470M photos, 4 or 5 sizes of each
- 38k requests/sec to memcached (open source distributed memory object caching system)
- 2 PB raw storage
- Over 400,000 photos being added every day

Figure 6 The profile of the Flickr photo sharing site.

Clearly, this service has a large number of registered users and a lot of data.

**Flickr’s database partitioning scheme**

Flickr’s equivalent to the `GetDatabaseFor()` method assigns a random number for new accounts and uses this number as an index into the correct shard for this new user using

\[
\text{ShardToUse} = \text{RandomNumber mod NumberofShards}
\]

My data is stored on my shard but the record of my performing an action on your comment is stored on your shard. Shards contain a slice of the main database. The main database employs 100% replication in dual-master architecture. Migration of certain users is done manually from time to time. A very small percentage of extreme power users destroy the nice balance between shards. It’s important to restore that balance by migrating these types of users off to a different area of the database. Even on a site as big and active as Flickr, migration can be done manually.

Each shard is designed to hold approximately 400K+ of user data (not the photos themselves). A lot of data is stored twice. For example, a comment is part of the relation between the commenter and the commentee. Where is the comment stored? Both places. This is a good tradeoff between performance and disk use.

Certain operations, such as clicking on a favorite, access several shards. First, the photo owner’s account is pulled from the cache to get the shard location for this user. Then, it pulls my information from the cache to get my shard location. Next a “distributed transaction” is started to answer a question like “who favorited my photo.”

**Flickr’s reliability strategy**

To get rid of replication lag on every page load, the user is assigned to a bucket. If a host is down, go to the next host in the list; if all hosts are down, display an error page. Flickr doesn’t use persistent connections; they build connections and tear them down. Every page load thus tests the connection. Flickr architecture is shown in Figure 7.
The first level is load balancing as the incoming connection requests arrive. Squid caches are open source web delivery systems that operate as reverse proxies for HTML pages and images. The PHP App Servers connect to the shards and keep the data consistent. The storage managers do the actual mapping from an index to the correct shard. NatApps are for mass storage of photos. Their fundamental database architecture uses dual-masters for each shard. This gives them resilience in case of failure. The Dual Tree structure is a custom set of changes to MySQL that allows scaling by incrementally adding masters without ring architecture. The central database includes data like the users table, which includes primary user keys (a few different IDs) and a pointer to which shard a user’s data can be found on. The Big Search Engine is a replication of the database they want to search.

Each server in a shard is 50% loaded. They can shut down a half of the servers in each shard. It is designed so that one server in the shard can take the full load if a server of that shard is down or in maintenance mode. To upgrade, all they have to do is shut down half the shard, upgrade that half, and then repeat the process.

**Summary**

We covered sharding to scale very large applications with databases because databases frequently become the barrier to scaling. As we have seen, sharding is not only a powerful strategy for building high-scalability applications but it is a very common one as well. Google, Yahoo!, Flickr, Facebook, and many other sites that deal with huge user communities have found common ground that database partitioning in this way is a must to enable a good user experience as the site grows without bound.