Database sharding is the process of splitting up a database across multiple machines to improve the scalability of an application. The justification for database sharding is that *after a certain scale point* it is cheaper and more feasible to scale a site horizontally by adding more machines than to grow it vertically by adding beefier servers.

**Why Shard or Partition your Database?**

Let's take [**Facebook.com**](http://www.facebook.com/) as an example. In early 2004, the site was mostly used by Harvard students as a glorified online yearbook. You can imagine that the entire storage requirements and query load on the database could be handled by a single beefy server. Fast forward to 2008 where just the Facebook application related page views are about [**14 billion a month**](http://developers.facebook.com/news.php?blog=1&story=174) (which translates to over 5,000 page views per second, each of which will require multiple backend queries to satisfy). Besides query load with its attendant [**IOPs**](http://en.wikipedia.org/wiki/IOPS), 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**](http://www.facebook.com/note.php?note_id=30695603919). 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. Do you think the original database used by Facebook had terabytes of storage available just to store photo metadata?

At some point during the development of Facebook, they reached the physical capacity of their database server. The question then was whether to scale vertically by buying a more expensive, beefier server with more RAM, CPU horsepower, disk I/O and storage capacity or to spread their data out across multiple relatively cheap database servers. In general, 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 which 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 provides excellent bang for the buck with regards to cost savings relative to the increased complexity of the system.

Now that we have an understanding of when and why one would shard a database, the next step is to consider how one would actually partition the data into individual shards. There are a number of options and their individual tradeoffs presented below – Pseudocode / Joins

**How Sharding Changes your Application**

In a well-designed application, the primary change sharding adds to the core application code is that instead of code such as

//string connectionString = @"Driver={MySQL};SERVER=dbserver;DATABASE=CustomerDB;"; <-- should be in web.config

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 connect to depends on the data we are trying to store or access. So, you'd have the following instead

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 being that the GetDatabaseFor() method knows how to map a customer ID to a physical database location. For the most part everything else should remain the same unless the application uses sharding as a way to parallelize queries.

### A Look at a Some Common Sharding Schemes

There are a number of different schemes one could use to decide how to break up an application database into multiple smaller DBs. Below are four of the most popular schemes used by various large-scale Web applications today.

1. 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, friend lists on another and a third for user generated content like photos and blogs. The key benefit of this approach is that is straightforward to implement and has low impact to the application as a whole. 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 140 million users may be more than a single server can handle).
2. 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 values ranges that occur within each entity. For example, splitting up sales transactions by what 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 which fails to account for situations where your application is popular in a particular region and the fact that human populations vary across different zip codes.
3. 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 simplistic example is to consider if you have ten database servers and your user IDs were a numeric value that was incremented by 1 each time a new user is added. In this example, the hash function could be performing a [**modulo operation**](http://en.wikipedia.org/wiki/Modulo_operation) on the user ID with the number ten and then pick a database server based on the remainder value. This approach should 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.
4. Directory Based Partitioning: A loosely couples approach to this problem is to create a lookup service which 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 which actually 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 a modulo operation. 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 is copied over (although this is tricky since users are always updating their data) the lookup service can change to using the new hash function without any of the calling applications being 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.

### Problems Common to all Sharding Schemes

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. Below are some of the constraints and additional complexities introduced by sharding

* Joins and Denormalization – Prior to sharding a database, any queries that require joins on multiple tables execute on a single server. Once a database has been sharded across multiple servers, it is often not feasible to perform joins that span database shards due to performance constraints since data has to be compiled from multiple servers and the additional complexity of performing such cross-server.

A common workaround is to de-normalize the database so that queries that previously required joins can be performed from a single table. For example, consider a photo site which has a database which contains a user\_info table and a photos table. Comments a user has left on photos are stored in the photos table and reference the user's ID as a foreign key. So, when you go to the user's profile it takes a join of the user\_info and photos tables to show the user's recent comments.  After sharding the database, it now takes querying two database servers to perform an operation that used to require hitting only one server. This performance hit can be avoided by de-normalizing the database. In this case, a user's comments on photos could be stored in the same table or server as their user\_info AND the photos table also has a copy of the comment. That way rendering a photo page and showing its comments only has to hit the server with the photos table while rendering a user profile page with their recent comments only has to hit the server with the user\_info table.

Of course, the service now has to deal with all the perils of denormalization such as data inconsistency (e.g. user deletes a comment and the operation is successful against the user\_info DB server but fails against the photos DB server because it was just rebooted after a critical security patch).

* Referential integrity – As you can imagine if there's a bad story around performing cross-shard queries it is even worse trying 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 applications that require referential integrity often have to enforce it in application code and run regular SQL jobs to clean up dangling references once they move to using database shards.

Dealing with data inconsistency issues due to denormalization and lack of referential integrity can become a significant development cost to the service.

* Rebalancing **(Updated 1/21/2009)** – In some cases, the sharding scheme chosen for a database has to be changed. This could happen because the sharding scheme was improperly chosen (e.g. partitioning users by zip code) or the application outgrows the database even after being sharded (e.g. too many requests being handled by the DB shard dedicated to photos so more database servers are needed for handling photos). In such cases, the database shards will have to be rebalanced which means the partitioning scheme changed AND all existing data moved to new locations. Doing this without incurring down time is extremely difficult and not supported by any off-the-shelf today. 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)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
  
Scalability Strategies Primer: Database Sharding

## **Introduction**

We’ve entered an era of application development that increasingly requires scalable solutions to everyday problems. By and large, the methods used to scale an application to massive proportions are fairly standardized. With a modicum of time spent researching database scalability methods, one can get the gist of the strategies involved in solving many common scalability problems. The actual implementations of such methods, however, have nary seen the light of day beyond the doors of the companies and developers that develop them.

This article is a primer, intended to shine some much-needed light on the logical, process oriented implementations of these scalability strategies in the form of a broad introduction. More specifically, my intention is to teach the majority of these implementations by example. Examples will be formatted as imagined scenarios modeling real life scalability challenges.

The technology focus of this article is intended to be as non-specific as possible. However, in scenarios or example implementations where it is necessary to define a specific software stack, I have chosen the typical LAMP (Linux, Apache2, MySQL, PHP) web development stack.

This article is intended for intermediate developers familiar with the intricacies of a data access layer and its role within the typical web application.

## **Brief Author’s Note**

Most industry standard development practices, frameworks, and design pattern fundamentals have inherent in their intended usage the requirement that the system on which they are running will be of sufficient performance to provide a consistently reliable environment. Now there are obvious exceptions to this horrifyingly broad generalization, such as any distributed design pattern for one, but I argue that the above holds true for what much of the software industry’s development is focused upon. I have found, through a combination of personal experience and fairly extensive research that most of what is taught to and learned by the typical developer is based on the absence of scale. As a result of this, the solutions to many problems of scale are often counter-intuitive.

Before delving further into the subject at hand, I encourage the reader to keep an open mind and attempt to overcome what might at first be an initial sense that you are a bit like Alice in Wonderland – that what you are reading is counter to much of what you’ve previously been taught or learned. Keeping in mind that there is no silver bullet solution, no solution that can solve all your scalability problems without requiring that you compromise on other portions of your application, will help to hammer home the reasoning behind many of the theories presented here.   
Simply stated, the generally accepted development solutions to most business requirements break down at scale and must be approached from an often counter-intuitively different angle.

## **Database Scalability**

It’s important to have an understanding of some of the basic scalability terminology used throughout this article to explain the more complex concepts and exacting scalability solutions presented later.

Let’s start by defining some of that basic terminology, and discerning some of it from their often-misused forms.  The addition of brackets to some of the definitions below is used to specify a more database centric form of the word or phrase.

***Performance*** – The speed at which a system can service a single transaction.

***Scalability*** – The ability of a system to maintain its quality of service as overall system load increases.

***Scaling Vertically*** – To scale a system vertically is to increase the scalability of the system by adding resources to each individual component within the system.

***Scaling Horizontally*** – To scale a system horizontally is to increase the scalability of the system via the addition of individual components to the system.

***Vertical Dataset Partitioning*** – To partition a [database table] dataset vertically is to partition a dataset by field [or table column].

***Horizontal Dataset Partitioning*** – To partition a [database table] dataset horizontally is to partition a dataset by data entry [or table row].

***System*** – A system, as it is being used in the context of this article, is a general-purpose term used to describe a collection of components that are interconnected in both software and hardware, and that combine to create an integrated whole.

***Shard*** – A single instance of a database, often run on a commodity server, which houses a subset of the system’s total dataset. Each shard within a system typically shares the same topically based schema, though the actual data on each shard is unique to that shard.

Did you notice the differentiation made between the definition of Performance and Scalability? Both words are often used interchangeably to mean one or the other, but they are in fact starkly different in definition. I thought it wise to make a concerted effort to bring these two terms to attention, as it is important to realize that this article attempts to tackle the problems specific to rolling out a solution to the Scalability of a system and not the Performance of a system. Also, please note how each definition below Scalability uses one or more previously defined terms within its own definition. We’re building upon concepts here in a way that I hope is easy to follow.

I’d like to get even more specific about the kinds of scalability problems we’re trying to solve with the strategies presented here. This article focuses on scaling horizontally by partitioning horizontally (by row). You may alternatively decide to scale horizontally by partitioning vertically (by column) as necessary, but in my experience, you end up needing to partition horizontally at some point along the path to massive scalability regardless of your decision to partition vertically. Because of this inevitability, I’ve chosen to focus my efforts on an article that details the horizontal partitioning of a database schema.

## **Partitioning Your Dataset**

To better explain the reasoning behind partitioning your dataset, we need to first answer the following fundamental two-part question:

Why are we partitioning our dataset and how does it help us to achieve scalability of our application?

It is difficult, if not nearly impossible, to massively scale your data layer when the data is limited to residing on a single server. Whether the limiting factor is a hardware cost issue, or you’ve simply equipped your server with the highest performing hardware possible, we ultimately find ourselves up against a wall – there are inherent limitations to what is currently possible by vertical scaling our hardware, it is a simple matter of fact. If we instead take our dataset schema, duplicate it onto multiple servers (shards), and split (or partition) the data on the original single server into equal portions distributed amongst our new set of servers (shards), we can parallelize our query load across them. Adding more servers (shards) to our existing set of servers results in near limitless scalability potential. The theory behind how this works is simple, but the execution is a fair bit more complex thanks to a series of scale-specific issues one encounters along the way.

We’ll continue with the concepts behind the partitioning of the actual dataset. There are numerous dimensions, or data subsets (often topically organized tables), to which you may choose to partition your dataset, and the method to follow is highly dependent on your particular problem domain’s requirements. This process is often referred to as denormalizing your schema. At small scale, having a normalized schema is ideal, but at large scale, having your data denormalized makes scalability much simpler – we’ll get into why that is shortly. Now I’m going to be completely honest here – how one decides to partition their dataset involves some serious thought, so let’s break down a few of the more commonly encountered, real-life dataset scenarios and walk through each by example.

I’ve divided each dataset scenario first by its lookup type, then by its partitioning method. I’ve done this for a couple of reasons. First, the way in which your web application will find where a specific piece of data (commonly a table row) resides within a partitioned system is different depending on the partitioning method used. Secondly, each lookup type has several trade-offs associated with its use, and therefore I think it appropriate that each lookup type be discussed at length.

## **Algorithmic Lookups**

To perform an algorithmic lookup is to use a rule to retrieve the shard location of the data based upon a simple algorithm seeded by some portion of the data itself.

### ****Date and Time Partitioning****

Partitioning by a dataset entry’s relevant date and time is one of the simplest ways to perform an algorithmic lookup. Often the relevant date and time that you are performing the lookup on is the date and time of the entry’s insertion into the dataset, though this is not always the case.

Take, for example, a common billing scenario involving invoices. If you are working with a large dataset that is overpowering a single server within your system, you may opt to partition that server’s dataset over multiple servers, or shards. In doing so, you relieve much of the contention on the origin server’s resources. In our example, we’re presented with the following problem:

“Our system has a single database server. That database component is over utilized, nearly to the point of failure, by intermittent but long running queries. The primary issue is the sheer amount of data within the dataset. To put this in more specific terms, we are working with approximately 25 million invoices added evenly over a period of 5 years.”

Given the above scenario, we can say the following:

* ***Low Volume of Reads/Writes*** – we aren’t working with a high volume of reads or writes
* ***Resource Intensive Reads*** – although the database reads are intermittent, they are long running and therefore each utilize an intensive amount of both CPU and Disk resources
* ***High Volume of Data*** – we’re working with a large volume of data
* ***Even Distribution of Data*** – our dataset is large, but its evenly distributed over 5 years at about 5 million entries per year

Now that we’ve broken this problem down by its core characteristics, we can begin to apply our sharding solution!

In this scenario, we’re fortunate enough to be working with a predictable dataset. Based upon historical precedents, our dataset will continue to grow over time at a rate of 5 million invoices per year. The typical commodity server is capable of handling a table of 5 million records without much difficulty (the actual number of records is arbitrary in the context of this example). Because of this, we can easily partition our dataset of 25 million records by relocating each year’s worth of records onto a separate database shard. If we apply this strategy we end up with five commodity servers, or shards, each running a separate instance of MySQL. Within each instance of MySQL is a single Invoices table containing one year’s worth of invoices or five million records.

Each year going forward, we’ll add another shard to our system, and on it will reside a single instance of MySQL with its table containing that years’ worth of records. Assuming we continue to experience linear growth and fairly evenly distributed and consistent query execution across all shards, we’ve now positioned ourselves for long-term scalability.

Determining where to insert a new row is simple. Assuming that each shard is labeled starting with zero, the following pseudo-code expression will produce the shard to insert into.

currentDate.Year - startDateOfFirstRecord.Year = shardId to insert into

The above expression can also be used to determine which shard we need to query in order to retrieve the data that we need. If we need to retrieve invoices from a year ago, we would use the following pseudocode.

(currentDate.Year - 1) - startDateOfFirstRecord.Year = shardId to query

This strategy is fairly straightforward and lends itself well to parallelization of queries that span multiple shards. For example, if we need to query records over two years, we would split the query up into two separate queries, each running on the shard that contains the specified year’s data. We’re now running these queries in parallel and taking advantage of the increase in resources that our new system provides us – running a query on two shards is effectively twice as fast as running nearly the exact same query on a single database server.

### ****Row Count Partitioning****

Another common alternative to Date and Time partitioning is Row Count Partitioning. Row Count Partitioning utilizes the same basic concepts as Date and Time partitioning, with the following exceptions:

* We aren’t limited to a single type of dataset field as our criteria. We can simply use a common, albeit simplistic, benchmark for determining dataset size – the dataset’s row count.
* We don’t need to worry about the growth rate at which our dataset grows, because the row count is a measurable result of our dataset’s growth itself.

Let’s apply the Row Count Partitioning strategy to the same scenario we used in the Date and Time Partitioning strategy. I’ve restated the scenario here so you don’t have to navigate backwards:

“Our system has a single database server. That database component is over utilized, nearly to the point of failure, by intermittent but long running queries. The primary issue is the sheer amount of data within the dataset. To put this in more specific terms, we are working with approximately 25 million invoices added evenly over a period of 5 years, though this can’t be guaranteed.”

Given the above scenario, we can say the following:

* **Low Volume of Reads/Writes** – we aren’t working with a high volume of reads or writes
* **Resource Intensive Reads** – although the database reads are intermittent, they are long running and therefore each utilize an intensive amount of both CPU and Disk resources
* ***High Volume of Data*** – we’re working with a large volume of data
* ***Even Distribution of Data*** – our dataset is large, but its evenly distributed over 5 years at about 5 million entries per year, though this can’t be guaranteed

We no longer have to worry about whether we’re working with a predictable dataset. Based upon historical precedents, our dataset will continue to grow over time at a rate of 5 million invoices per year, but this now can’t be guaranteed. The typical commodity server is capable of handling a table of 5 million records without much difficulty. Because of this, we can easily partition our dataset of 25 million records by relocating five million records at a time onto a separate database shard. If we apply this strategy we end up with five shards, each running a separate instance of MySQL. Within each instance of MySQL is a single table containing five million records worth of invoices.

Each year going forward, we’ll add another shard to our system, and on it will reside a single instance of MySQL with a table containing five million records. Now that we don’t need to worry about whether our dataset maintains its apparent even distribution, we are now positioned for long-term scalability, in addition to non-linear growth.

Determining where to insert a new row is simple. We simply set a configuration variable, within our application, indicating which of our shards is currently actively taking on additional rows. As the active shard reaches its maximum row count threshold, our application simply needs to increment (or otherwise change) the configuration variable to the next shard ready for use.

Now here’s where the flexibility we’ve received as a function of working with a row count tends to make querying the dataset slightly more difficult. The above rule cannot also be used to determine which server we need to query in order to retrieve the data that we need. If we need to retrieve invoices from a year ago, we need to query all of our shards, since we don’t have the luxury of being able to determine where the records that fell within the last year were inserted.

This isn’t necessarily as bad as it sounds. What we’ve done here is build a highly parallelized version of our existing system. Now, instead of one server querying through our dataset to find what we need, we have five shards doing the work in parallel, over smaller subsets of the same dataset. We would then take the results from each shard and combine them in our application layer, resulting in a significantly faster method of querying a large dataset.

It’s important to understand that each method of partitioning your dataset is better suited to certain problems, as opposed to others. In the case of the Row Count Partitioning strategy, we’ve explored a method of partitioning that is best used in conjunction with a large dataset that requires queries to be run often across a majority of the data within our dataset.

This strategy is fairly straightforward and lends itself well to parallelization of queries that span multiple shards. For example, if we need to query records over all five years, we would split the query up into five separate queries, each running on a separate shard. We’re now running these queries in parallel and taking advantage of the increase in resources that our new system provides us – running a query on five shards is effectively five times as fast as running nearly the exact same query on a single server.

## **Master Index Lookups**

Using a Master Index Lookup isn’t nearly as simple or intuitive as an Algorithmic Lookup. It is, however, necessary for scenarios that require partitioning along a dimension that is specific to a particular dataset field that is not inherently consecutively ordered.

### ****Domain Partitioning****

One of the more complex partitioning methods, Domain Partitioning provides the highest amount of flexibility in choosing one’s partitioned dimension.

Determining what dimension to partition on is highly dependent on the nature of one’s database schema. However, in order to properly illustrate how you might do so, I will use a typical user-centric database pseudo-schema as our working example. I’m hoping that this example will clearly describe the process involved in determining a dimension to partition in such a way that you will be able to infer how to alter this strategy for your particular application.

The key to understanding how to partition a database schema is to first determine if you are working with a schema that can be appropriately partitioned in the first place. A schema that is well suited for partitioning along a particular dimension will have the following properties:

* The schema’s tables are naturally topically segregated as opposed to topically cross-related. For example, most tables relate to a User table, and contain rows that are particular to a single User row.
* The schema’s most significant topical tables, if split, are unlikely to require many joins across them. For example, if we are partitioning along the User table, it should be unlikely that we will need to join multiple Users that each reside on a different shard.
* The number of satellite tables, and hence their contained data, related to a single hierarchical table is small enough in size, and will stay small enough in size, so as to not become a performance bottleneck.

Please note that although the above properties make for a best-case scenario dataset, failure of your dataset to have all of them does not imply that your dataset cannot be partitioned using the Domain Partitioning strategy.

Domain Partitioning is comprised of two main server types: An Index Shard and one or more Domain Shards.

The Index Shard serves as the primary lookup mechanism. At a minimum, it will contain a single index of a dataset’s partitioned dimensions and their Domain Shard locations. This can be easily represented as such – an application utilizing the Domain Partitioned strategy will first connect to and query the Index Shard to retrieve the shard location of the requested data. Once that location is returned from the Index Shard, another connection will be made to the Domain Shard storing the requested data, and the original query will then be executed, returning its result to the application.

You may be wondering if there is a high amount of overhead involved in always connecting to the Index Shard and querying it to determine where the second data retrieving query should be executed. You would be correct to assume that there is some overhead, but that overhead is often insignificant in comparison to the increase in overall system performance, as a result of this strategy’s granted parallelization. It is likely, independent of most dataset scenarios encountered, that the Index Shard contains a relatively small amount of data. Having this small amount of lookup data means that the database tables holding that data are likely to be stored entirely in memory. That, coupled with the low latencies one can achieve on a typical gigabit LAN, and also the connection pooling in use within most applications, and we can safely assume that the Index Shard will not become a major bottleneck within the system (have fun cutting down this statement in the comments, I already know it’s coming :)

Let’s proceed through our user-centric scenario by detailing it out further:

“Our system has a single database server. That database server is over utilized, nearly to the point of failure, by intermittent but long running queries. The primary issue is the sheer amount of growth the dataset is experiencing. To put this in more specific terms, we are working with approximately 5 million users added non-linearly over the last 2 years. Each user is made up of a user table, a user\_profile table, a user\_blog table, and a user\_blog\_entry table. Each row within the user\_profile table is related to a single row within the user table. Each row within the user\_blog table is related to a single row within the user table. Each row within the user\_blog\_entry table is related to a single row within the user\_blog table.”

Given the above scenario, let’s start by choosing our partitioning dimension. We know that all of the data within our dataset is directly or indirectly topically related to a single entity – a user.  We also know that users being added to the system are the primary source of our system’s enormous growth. Since users are defined by their user row within the user table, we can confidently say that by partitioning the user table across multiple shards, we can distribute load in a manageable manner. The user table, and its rows, is our chosen partition dimension.

### ****Defining the Index Shard schema****

Now that we know our partition dimension is a user, we need a unique identifier for each user so that we can use it to lookup our user. This is easily done by incorporating a userId key column within the user table. Having the userId key is important, but it isn’t very useful in looking up a user in and of itself. Most users are queried using their username. We can even take this a step further and say that authenticating a user is nearly as important as identifying them by their username. Authenticating a user requires both a username and a password. Armed with this common knowledge, we can begin to specify the schema of our Index Shard’s first couple of tables.

The most obvious required table within the Index Shard lookup database is a shard table. This table has the following columns:

***shardId*** – used as the unique identifier for a shard

***connectionString***– a connection string used to connect to a shard

***status*** – used to signify a shard’s status as online, offline, or in active insert mode

***createdDate*** – the date the shard was added to the system, used for historical purposes

The second table we need to define is the user\_lookup table. This table has the following columns:

***userId*** – used to uniquely identify a user. Is the same userId used to identify a user within the user table located on each shard.

***shardId***–used to uniquely identify the current shard that a user is located on

***username*** – the username corresponding with the userId. Is the same username used within the user table located on each shard.

***password***– the password corresponding with the userId. Is the same password used within the user table located on each shard.

Now let’s run through the four common SQL command types and apply their usage to our sharded system, making sure to note the differences in procedures used. These command types are the SELECT, INSERT, UPDATE, and DELETE.

Insert Scenario: A new user signs up.

1. Connect to the Index Shard using an application configuration-level connection string.
2. Query the shard table and retrieve the shard row that represents the current shard with a status of active insert mode.
3. Disconnect from the Index Shard.
4. Connect to the Domain Shard as specified by the previously retrieved shard row’s connectionString.
5. Insert the user’s sign up information into the user table. Retrieving the userId as a result.
6. Insert the user’s remaining creation information in to the user table’s related tables as necessary (i.e. user\_profile, user\_blog, etc).
7. Disconnect from the Domain Shard.
8. Connect to the Index Shard using an application configuration-level connection string.
9. Insert the new user’s lookup information into the user\_lookup table, using the shardId from the retrieved shard table and the userId from the Domain Shard’s user table, for the new location of the user’s information.
10. Disconnect from the Index Shard.

Update Scenario: A user changes their password.

1. Connect to the Index Shard using an application configuration-level connection string.
2. Query the user\_lookup table, using the username and current password of the user, and retrieve the user\_lookup row that contains the user’s lookup information.
3. Query the shard table and retrieve the shard row that represents the user’s Domain Shard location.
4. Update the retrieved user\_lookup row, changing the password field to the user’s new password.
5. Disconnect from the Index Shard.
6. Connect to the Domain Shard as specified by the previously retrieved shard row’s connectionString.
7. Update the user’s user row, found using the userId as retrieved earlier from the user\_lookup table, changing the password field to the user’s new password.
8. Disconnect from the Domain Shard.

Delete Scenario: A user closes their account.

1. Connect to the Index Shard using an application configuration-level connection string.
2. Query the user\_lookup table, using the username and current password of the user, and retrieve the user\_lookup row that contains the user’s lookup information, saving it for later use.
3. Query the shard table and retrieve the shard row that represents the user’s Domain Shard location.
4. Delete the user’s user\_lookup row, using the user’s username and password, or userId to find the user’s row.
5. Disconnect from the Index Shard.
6. Connect to the Domain Shard as specified by the previously retrieved shard row’s connectionString.
7. Delete the user’s user row, found using the userId as retrieved earlier from the user\_lookup table.
8. Disconnect from the Domain Shard.

Select Scenario: A system visitor views a user’s profile page.

1. Connect to the Index Shard using an application configuration-level connection string.
2. Query the user\_lookup table, using the username or userId of the user, and retrieve the user\_lookup row that contains the user’s lookup information.
3. Query the shard table and retrieve the shard row that represents the user’s Domain Shard location.
4. Disconnect from the Index Shard.
5. Connect to the Domain Shard as specified by the previously retrieved shard row’s connectionString.
6. Query the user\_lookup table to retrieve the user’s basic information, using the previously retrieved userId.
7. As necessary, query the user’s additional profile information and blog entries via the user\_profile, user\_blog, and user\_blog\_entry tables respectively.
8. Disconnect from the Domain Shard.

Our system has now been successfully Domain Partitioned. As a result, we have effectively enabled our system to scale along the User dimension far into the future. By simply adding additional shards to our system, and updating the necessary shard table row’s, we can easily increase the capacity of our system with little effort.

## **Dealing with Hotspots and Rebalancing Shards**

It is nearly inevitable, that at some point within the lifecycle of any one of the previously discussed scalability strategies, we will encounter a scenario that requires a redistribution of our data as a measure to prevent continued performance degradation within a single overloaded shard.

Examples of real-life scenarios where this kind of behavior may be observed are numerous. Using our user-centric theme, one might notice a hotspot originating around a specific user that is getting a higher than usual query hit-rate due to some level of user importance. That user may be a prominent blogger or some other similarly popular individual. As a result of their prominence, we can imagine their user profile receiving pageviews of multiple orders of magnitude higher than the average user’s profile.

Building upon our Domain Partitioning strategy and by enumerating a further set of specifics, we can find a suitable solution to the hotspot problem. Let’s imagine that each row within our user table contains a unique identifier. This unique identifier is a Globally Unique Identifier (GUID) and its uniqueness is guaranteed throughout the system. If we also require that any related tables, such as the user\_profile, user\_blog, and user\_blog\_entry tables also contain GUID’s serving as their table keys, we can easily move any data associated with a single user amongst any of our user Domain Shards by migrating the data and updating the user\_lookup table. These migrations could be automated, but are likely to be rare enough that a manual migration may be preferred given that it is sometimes difficult to estimate the required resources needed to dedicate to a hotspot user or group of users.

A simple automated migration process might look something like the following:

1. Connect to the Index Shard using an application configuration-level connection string.
2. Query the user\_lookup table, using the username and current password of the user, and retrieve the user\_lookup row that contains the user’s lookup information.
3. Query the shard table and retrieve the new shard’s shard row that represents the user’s new Domain Shard location.
4. Update the retrieved user\_lookup row, changing the shardId field to the user’s new shard.
5. Disconnect from the Index Shard.
6. Connect to the user’s new Domain Shard as specified by the previously retrieved shard row’s connectionString.
7. Insert the user’s original user row data into the user’s new shard’s user table.
8. Insert the user’s remaining information in to the user table’s related tables as necessary (i.e. user\_profile, user\_blog, etc).
9. Disconnect from the Domain Shard.
10. Connect to the user’s old Domain Shard as specified by the user’s originally retrieved shard row’s connectionString.
11. Delete the user’s user row, found using the userId as retrieved earlier from the user\_lookup table.
12. Disconnect from the Domain Shard.

It’s important to realize that dimension entity migrations using the above method is a storage and performance trade-off. A true GUID is a relatively large column type, usually somewhere between 16 and 36 bytes depending on whether compression is used. Working with keys of this size makes migrations simple, but can have a negative effect on queries running through the dataset. Be sure to weigh your options and plan accordingly based upon your particular business requirements.

## **Adding High-Availability**

This article has, up until this point, discussed solutions to database scalability challenges solely. However, if one is to design both a highly scalable and highly available database solution, there are a few other points that need to be applied to the design principles presented so far. Please note that the subject of high-availability is worthy of its own article and that this section is merely a brief introduction to one of the more common failover strategies.

Providing high-availability, or failover functionality to ensure continued operation of our system, is a concept that is independent of the scalability of a system. Therefore, we can add high-availability to any scenario previously used to explain the application of our scalability strategies. There are also numerous availability strategies that one can use, similar to the differences in how we apply scalability strategies. I’ll be reviewing one of the more common availability strategies used in MySQL – master to master replication.

### ****Master to Master Replication****

Master to Master Replication involves utilizing MySQL’s replication capabilities to create redundancy of data between two servers, effectively creating a small redundant cluster.

This strategy requires two MySQL servers, which together form a single shard as defined in any one of our previously imagined scenarios. As is typical in a MySQL replication implementation, there is a master and a slave replication instance on each server within the shard’s cluster. As mentioned earlier, each shard is partitioned along a specific topical dimension. This topical dimension, implemented via a database table, must contain a unique identifier. This unique identifier must be unique not just on a single machine, but either globally or between the two servers being used to create our highly-available shard. This uniqueness is really the only data-level requirement necessary for this availability strategy.

In an effort to explain this as simply as possible, let’s assume that our unique identifier is a Globally Unique Identifier (GUID). Upon inserting a row into our primary dimension table on our first Master, our replication configuration will replicate that command to its slave system which in our case is the second Master. Because this same configuration is setup on our second Master, inserting a row into our primary dimension table on our second Master will replicate the command to our first Master. This replication setup effectively results in a high-availability, hot-failover solution.

Any table that we wish to replicate is required to have a GUID as its key so that it may be properly replicated. In practice, one should provide every table within a shard with a GUID key so as to ensure complete global independence of the dataset contained within the shard.

Now, our application layer needs to be aware of the availability of both servers within the shard cluster. In doing so, we’ve provided our application with two endpoints to which it may connect, execute commands, and retrieve data. A common method of utilizing both servers equally is to implement a simple round-robin algorithm to spread the connections between the two servers fairly evenly.

The occurrence of a single server failure within the shard cluster does not result in the loss of any significant amount of data, thanks to the replication mechanism continuously copying commands from the failed server to the running server. In order for our system to take advantage of this failover, we need to make our application layer aware of the behavior that occurs as a result of one server failing. Simply put, if one server fails within a shard cluster, our application layer needs to identify this occurrence and redirect its SQL commands to the remaining server while the failed server is being recovered via human intervention.

## **Wrap-Up**

Originally this article’s length was meant to be no more than a few pages, coupled with a coded implementation of a simple PHP sharding framework. About half way though, I realized that the concepts themselves were so important to the complete understanding of how a sharding implementation might work, and the subject itself so immense, that I wouldn’t be doing the reader justice by cutting their explanations so short and not getting as logically detailed as is reasonable for an introduction. Hence, I rewrote much of the original article into a format that espoused real-life scenarios to explain many of the more advanced partitioning strategies. I hope you, the reader, agree that I made the correct choice – please let me know if you don’t.

I feel confident saying that we’ve covered much of what I consider to be the bare conceptual basics of horizontally scaling your application’s database layer, and that was the intent – to provide a solid primer and no more. This subject is easily worthy of a book, and in fact, the absence of such literature was largely the motivation for this article. I hope to expound further on this subject over a series of future posts, but in the meantime, I’d like to hear some feedback on the format I’ve chosen. Did you find the “teach by example” theme easy to follow? Did the article provide a good base from which to build out from? Where there any particularly confusing sections that weren’t clear? Please comment and let me know so that I may further refine my writing style for this type of technical article.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*