[An Unorthodox Approach To Database Design : The Coming Of The Shard](http://highscalability.com/blog/2009/8/6/an-unorthodox-approach-to-database-design-the-coming-of-the.html)

**Update 4:** [Why you don’t want to shard.](http://www.mysqlperformanceblog.com/2009/08/06/why-you-dont-want-to-shard/) by Morgon on the MySQL Performance Blog. Optimize everything else first, and then if performance still isn’t good enough, it’s time to take a very bitter medicine.   
**Update 3:** [Building Scalable Databases: Pros and Cons of Various Database Sharding Schemes](http://www.25hoursaday.com/weblog/2009/01/16/BuildingScalableDatabasesProsAndConsOfVariousDatabaseShardingSchemes.aspx) by Dare Obasanjo. Excellent discussion of why and when you would choose a sharding architecture, how to shard, and problems with sharding.  
**Update 2:** [Mr. Moore gets to punt on sharding](http://www.37signals.com/svn/posts/1509-mr-moore-gets-to-punt-on-sharding) by Alan Rimm-Kaufman of 37signals. Insightful article on design tradeoffs and the evils of premature optimization. With more memory, more CPU, and new tech like SSD, problems can be avoided before more exotic architectures like sharding are needed. Add features not infrastructure. [Jeremy Zawodny](http://jeremy.zawodny.com/blog/archives/010841.html)says he's wrong wrong wrong. we're running multi-core CPUs at slower clock speeds. Moore won't save you.  
**Update:** Dan Pritchett shares some excellent [Sharding Lessons](http://www.addsimplicity.com/adding_simplicity_an_engi/2008/08/shard-lessons.html): Size Your Shards, Use Math on Shard Counts, Carefully Consider the Spread, Plan for Exceeding Your Shards  
  
Once upon a time we scaled databases by buying ever bigger, faster, and more expensive machines. While this arrangement is great for big iron profit margins, it doesn't work so well for the bank accounts of our heroic system builders who need to scale well past what they can afford to spend on giant database servers. In a extraordinary two article series, Dathan Pattishall, explains his motivation for a revolutionary new database architecture--sharding--that he began thinking about even before he worked at Friendster, and fully implemented at Flickr. Flickr now handles more than 1 billion transactions per day, responding in less then a few seconds and can scale linearly at a low cost.

What is sharding and how has it come to be the answer to large website scaling problems?

# Information Sources

* [Unorthodox approach to database design Part1:History](http://mysqldba.blogspot.com/2006/10/unorthodox-approach-to-database-design.html)
* [Unorthodox approach to database design Part 2:Friendster](http://mysqldba.blogspot.com/2006/11/unorthodox-approach-to-database-design.html)

# What Is Sharding?

While working at Auction Watch, Dathan got the idea to solve their scaling problems by creating a database server for a group of users and running those servers on cheap Linux boxes. In this scheme 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. Groups of 500K users are stored together in what are called shards.  
  
The advantages are:

 **High availability**. If one box goes down the others still operate.

 **Faster queries**. Smaller amounts of data in each user group mean faster querying.

 **More write bandwidth**. With no master database serializing writes you can write in parallel which increases your write throughput. Writing is major bottleneck for many websites.

 **You can do more work**. A parallel backend means you can do more work simultaneously. You can handle higher user loads, especially when writing data, because there are parallel paths through your system. You can load balance web servers, which access shards over different network paths, which are processed by separate CPUs, which use separate caches of RAM and separate disk IO paths to process work. Very few bottlenecks limit your work.

# How Is Sharding Different Than Traditional Architectures?

Sharding is different than traditional database architecture in several important ways:

 **Data are denormalized**. Traditionally we normalize data. Data are splayed out into anomaly-less tables and then joined back together again when they need to be used. In sharding the data are denormalized. You store together data that are used together.   
  
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, media, etc, but the user profile data would be stored and retrieved as a whole. This is a very fast approach. You just get a blob and store a blob. 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. You buy bigger machines to get more power. With sharding the data are parallelized and you scale by scaling out. Using this approach you can get massively more work done because it can be done in parallel.

 **Data are kept small**. The larger a set of data a server handles the harder it is to cash intelligently because you have such a wide diversity of data being accessed. You need huge gobs 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 making the data more parallelized so more work can be done on the data. You can also setup a shard to have a master-slave or dual master relationship within the shard to avoid a single point of failure within the shard. If one server goes down the other can take over.

 **It doesn't use replication**. 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 load increases the cost of replication increases. Replication costs in CPU, network bandwidth, and disk IO. The slaves fall behind and have stale data. The folks at [YouTube](http://highscalability.com/youtube-architecture)had a big problem with replication overhead as they scaled.  
  
Sharding cleanly and elegantly solves the problems with replication.

# Some Problems With Sharding

Sharding isn't perfect. It does have a few problems.

 **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.  
  
On some platforms I've worked on this is a killer problem. You had to build out the data center correctly from the start because moving data from shard to shard required a lot of downtime.  
  
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. This is what Flickr does. And your references must be invalidateable so the underlying data can be moved while you are using it.

 **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 the data. You have to make individual requests to your data sources, get all the responses, and the build the page. Thankfully, because of caching and fast networks this process is usually fast enough that your page load times can be excellent.

 **How do you partition your data in shards?** What data do you put in which shard? Where do comments go? Should all user data really go together, or just their profile data? Should a user's media, IMs, friends lists, etc go somewhere else? Unfortunately there are no easy answer to these questions.

 **Less leverage**. People have experience with traditional RDBMS tools so there is a lot of help out there. You have books, experts, tool chains, and discussion forums when something goes wrong or you are wondering how to implement a new feature. Eclipse won't have a shard view and you won't find any automated backup and restore programs for your shard. With sharding you are on your own. 

 **Implementing shards is not well supported**. Sharding is currently mostly a roll your own approach. [LiveJournal](http://highscalability.com/livejournal-architecture) makes their tool chain available. Hibernate has a [library](http://highscalability.com/product-hibernate-shards) under development. MySQL has added support for [partioning](http://dev.mysql.com/doc/refman/5.1/en/partitioning.html). But in general it's still something you must implement yourself.

# Appedix: Unorthodox approach to database design Part1:History

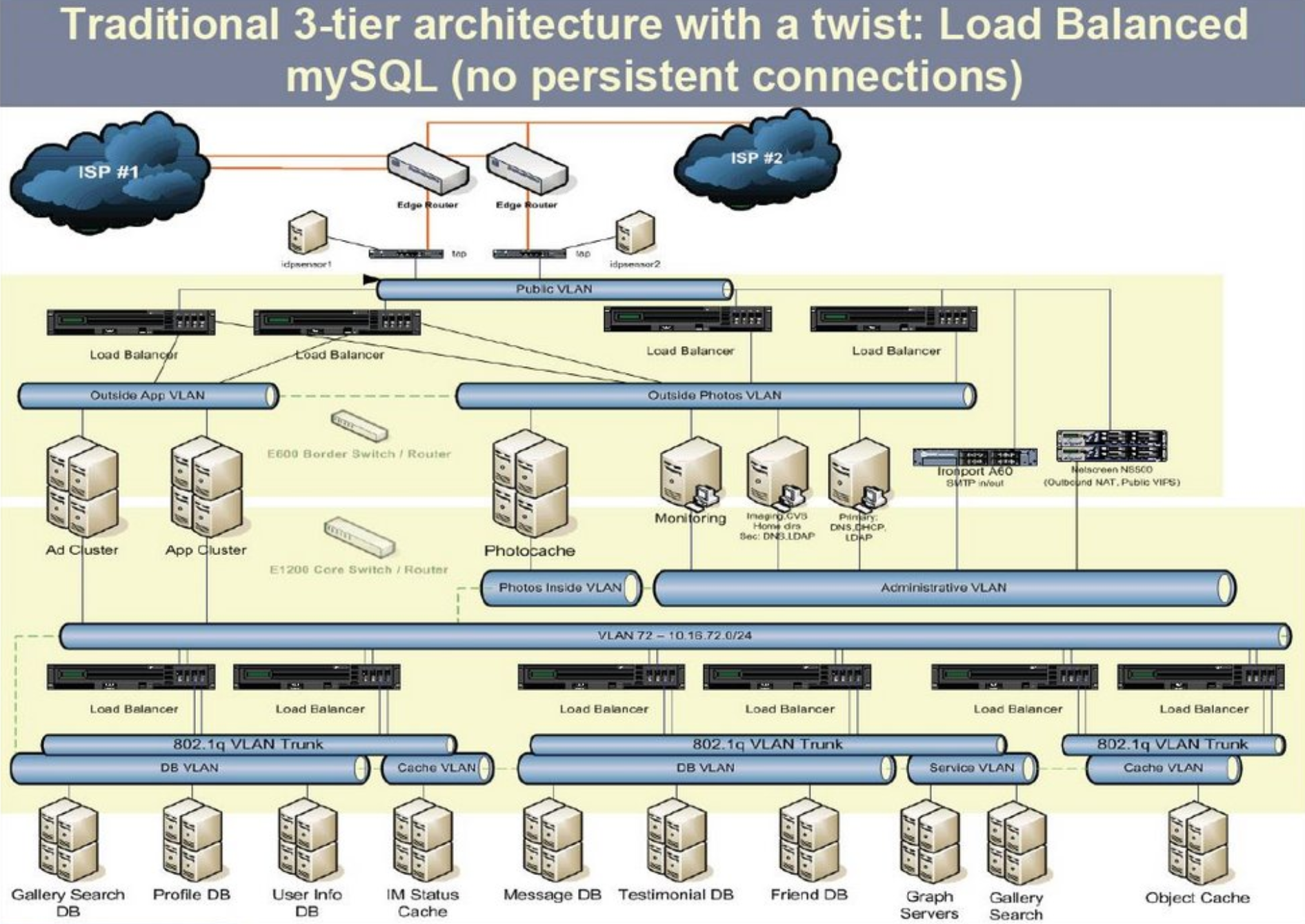
Back in 1999 I worked for a company called AuctionWatch. AuctionWatch created tools for power sellers to sell their goods with one easy interface to all 3 Major Marketplaces - Amazon, EBay, and Yahoo. The primary tool was the Auction Control Panel known as ACP. Think of a webtool that allowed power sellers to manage 1000s of auctions in one easy interface. Allowing them to schedule auctions, launch auctions and handle shipping from 1 page.   
  
During this time in 1999, we had 2 database servers named Godzilla and Mothra. I believe they where Sun E4000s with a super expensive diskpack in raid-5. Since mySQL was only in Gamma back then and replication was just a concept, we couldn't use replication to handle the onslaught of load from our users. Getting 200 queries per second would saturate AuctionWatch back then.  
  
So, what was done? Well more database boxes on more very expensive Sun boxes that handled different parts of the site was the 1st approach. The problem with this was if any mySQL instance crashed or a ISAM (yes we used ISAM back then) table went corrupt, the entire site went down. Also more boxes meant more points of failure.  
  
So, I came up with a plan and ran it past my managers. What if we could do ACP in a box? Meaning what if we could split up all sellers' data across many very in-expensive Linux boxes. I sold the idea buy producing this reasoning

If one box went down then only a small population of users would be affected.  
The application can query data faster when there is less of it.  
The application can run more updaters since it would have multiple write points.  
The application can launch auctions faster since the backend could support many more instances.

My boss loved the idea and the team worked on ways we could make it work for the site.  
  
Then the dot com bubble burst and I got laid off so the project was only partially rolled out. I have no idea if they are still using this today, but if so great!  
  
This was the beginnings of my unorthodox approach to database design.

* Denormalized data.
* Spread the data across many physical instances.
* Keep data small.
* Allow for any question to be asked.
* Allow for parallelism.
* Allow for absolute redundancy so downtime does not exist or is minimal.

# Appendix2: Unorthodox approach to database design Part 2:Friendster



Above is a diagram of the Friendster backend at the time I was there. The entire presentation can be found [here](http://www.mysqluc.com/cs/mysqluc2005/view/e_sess/6447)at mysqluc.com. Due to all the [turmoil](http://www.nytimes.com/2006/10/15/business/yourmoney/15friend.html?ei=5070&en=c428512ee52c5388&ex=1163221200&adxnnl=1&adxnnlx=1163099138-vginmAXfYiGXiz7Mhd6CBQ)at Friendster, even with the coolest and most competent VP of Engineering in the business Jeff Winner we could not push my intended design. This design is now used at Flickr with huge success.  
  
What I wanted to do was introduce a system that allowed Friendster to partition the user data. This was the same system that I started to implement at AuctionWatch in 1999 and was not able to complete.   
  
Each 64bit AMD server would house 500K distinct users and all their data. I provided a system to increase capacity, scale linearly on a fraction of the servers, at the fraction of the cost. There would be no need to cache any data, and everything could be fetched real time. In fact when I was discussing this with Jeff who loved the idea and saw the potential in walks Kent the CFO (now the 5th CEO) and sees 500K users on the white board. So, Kent goes hey what's a Sook user? We all chuckled and that became the project name, project 500k.  
  
Now here was my argument. What Friendster was using was simple old replication. Replication is a batch oriented system, when SQL is applied on the master it's logged to a file, slaves then read that file (think of tailf) then takes the SQL that is written and applies that same SQL on a slave server. When the application runs out of read capacity, the solution is to add more slave servers. In this model there is an inherent limitation.

Below is a list of some:

* There is a single point of failure on the master
* ALL write SQL operation done to the master must be replicated to the slaves
* When IO bandwidth is low replication lags, causing slave lag

Now many people to get around these issues build other masters to take over or come up with clever tricks in the application to get around these problems.  
  
I wanted to solve the issue without a trick. What was very expensive for Friendster was the user generated data. There was a crap load of it, about 500GB of it at the time. Replicating all that data was the cause of my heart-ache. Since the application was very dynamic and required huge ranges, I/O bandwidth was cut on every range query causing replication lag.   
  
Project S00k solved these issues. Here was the argument:

 Split up user data  
With a small amount of data web developers can do full table scans at a fast rate with a high concurrency if the data is small enough. So, bad code could go out and not take down the site.   
On top of that, instead of replicating to an INNODB datafile that is 500GB that is very expensive to support by having multiple copies of, let’s only deal with a datafile or database size that is a few gigs in size. Why is this better?

* The data is faster to backup
* The data is faster to recover
* The data can fit into memory
* The data is easier to manage

Splitting up the user data so that User A exists on one server while User B exists on another server, each server now holds a shard of the data in this federated model.

 Provide High Availability  
A problem at Friendster was if the master goes down, the site goes down. To solve this:

* Take each S00K Shard and put it in a replication ring. This is now a backup of the Shard of data.
* If one server out of the Shard pair goes down so what keep writing to the other.
* If both go down only a small amount of users are affected, show them the outage notification while other users are still able to use the site.

 Provide more write bandwidth  
In single-master many-slave architecture write bandwidth is throttled, in federation you can write to all the masters and read from them.

 Really use replication for what it's design is for  
Replication is a batch system it’s IMHO designed for backup of the database. Each server of the shard replicates it’s data to it’s pair. To remove effects of replication lag, stick the shard viewer to a single server in the ring. All operations are done there. So, if there is replication lag, or replication is broken users would not notice. Read and write to the same place.

But, because of the environment in Friendster getting this done was not possible. So, I worked on the same old system and we where never able to solve the root of the capacity issues.

# Appendix3: Mr. Moore gets to punt on sharding

Sharding is a database technique where you break up a big database into many smaller ones. Instead of having 1 million customers on a single, big iron machine, you perhaps have 100,000 customers on 10 different, smaller machines.

The general advise on sharding is that you don’t until you have to. It’s similar to [Martin Fowler’s](http://www.ddj.com/showArticle.jhtml?articleID=184414966) First Law of Distributed Object Design: Don’t distribute your objects! Sharding is still relatively hard, has relatively poor tool support, and will definitely complicate your setup.

Now I always knew that the inevitable day would come where we would have no choice. We would simply have to shard because there was no more vertical scaling to be done. But that day seems to get pushed further and further into the future.

* Bigger caches, more reads

Our read performance is in some aspect being taken care of by the fact that you can get machines with 256GB RAM now. We upgraded the [Basecamp](http://www.basecamphq.com/)database server from 32GB to 128GB RAM a while back and we thought that would be the end of it.

The box was maxed out and going beyond 128GB at the time was stupid expensive. But now there’s 256GB to be had at a reasonable price and I’m starting to think that by the time we reach that, there’ll be reasonably priced 512GB machines.

So as long as Moore’s law can give us capacity jumps like that, we can keep the entire working set in memory and all will be good. And even if we should hit a ceiling there, we can still go to active read slaves before worrying about sharding.

* The bigger problem is writes

Traditionally it hasn’t been read performance that caused people to shard anyway. It has been write performance. Our applications are still very heavy on the reads vs writes, so it’s less of a problem than it is for many others.

But with the rise of SSD, like Fusion-IO’s ioDrive that can do 120K IOPS, it seems that we’re going to be saved by the progress of technology once again by the time we’ll need it.

* Punt on sharding

So where does that leave sharding? For us, we’re in the same position we’ve been in for the past few years. We just don’t need to pay the complexity tax yet, so we don’t. That’s not to say that sharding doesn’t have other benefits than simply allowing that which otherwise couldn’t be, but the trade is not yet good enough.

One point of real pain we’ve suffered, though, is that migrating a database schema in MySQL on a huge table takes forever and a day. That’s a very real problem if you want to avoid an enterprisey schema full of kludges put in place to avoid adding, renaming, or dropping columns on big tables. Or avoid long scheduled maintenance windows.

I really hope that the clever chaps at MySQL comes up with something more reasonable for that problem, though. I’m told that PostgreSQL is a lot more accommodating in this regard, so hopefully competition will rise all boats for that.

* Don’t try to preempt tomorrow

I guess the conclusion is that there’s no use in preempting the technological progress of tomorrow. Machines will get faster and cheaper all the time, but you’ll still only have the same limited programming resources that you had yesterday.

If you can spend them on adding stuff that users care about instead of prematurely optimizing for the future, you stand a better chance of being in business when that tomorrow finally rolls around.

# Appendix4: Building Scalable Databases: Pros and Cons of Various Database Sharding Schemes

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 perform 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 denormalize 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 denormalizing 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).

# Appendix5: Why you don’t want to shard.

The Percona training workshop will *not* cover sharding. If you follow our blog, you’ll notice we don’t talk much about the subject; in some cases it makes sense, but in many [we’ve seen](https://www.percona.com/blog/2009/03/01/kiss-kiss-kiss/) that it causes architectures to be prematurely complicated.

*So let me state it:* **You don’t want to shard.**

Optimize everything else first, and then if performance still isn’t good enough, it’s time to take a *very* bitter medicine. The reason you need to shard basically comes down to one of these two reasons:

1. **Very large working set** – The amount of memory you require to keep your frequently accessed data loaded exceeds what you can (economically) fit in a commodity machine. 5 years ago this was 4GB, today it is [128GB](https://www.percona.com/blog/2008/08/04/128gb-or-ram-finally-got-cheap/) or even 256GB.Â  Defining “working set” is always an interesting concept here, since with good schema and indexing it normally doesn’t need to be the same size as your entire database.
2. **Too many writes** – Either the IO system, or a slave can’t keep up with the amount of writes being sent to the server.Â  While the IO system can be improved with a RAID 10 controller w/battery backed write cache, the slave delay problem is actually very hard to solve. [Maatkit](http://www.maatkit.org/doc/mk-slave-prefetch.html) has a partial-solution (via Paul Tuckfield), but it doesn’t work for all workloads.

*(Yes, I am simplifying some of the scalability issues with MySQL on big machines, but I have faith that Yasufumi is making this better).*

### What types of Sharding are there?

Despite my cautions, if you have established that *you need to shard* there are quite a few options available to you:

1. **Sharding Partitioning by Application Function** – This is usually the **best** way to fix any of the problems mentioned above. What you do is pick a few very busy tables, and move them onto their own MySQL server.Â  Partition-by-function keeps the architecture still simple, and should work for most cases unless you have a single table which by *itself* can’t fit into the above constraints.
2. **Sharding by hash or key** – This method works by picking a column on a table and try and divide up your data based on it.Â  You can choose any column to hash on, you just need to make sure that it will equally distribute the data equally. In practice this method can be really hard to get working right, since even if each shard has the same amount of ‘customers’, demanding users tend to by far exceed average users and some servers are overloaded while others are not.

*(Tip: There are a few famous cases of both (a) bad hashing algorithms and (b) users becoming unequal all of the sudden;Â  You don’t want to shard based on the first character of a username – as there will be a lot more ‘M’ than ‘Z’.Â  For users becoming unequal all of the sudden, it’s always interesting to think of what scaling challenges Flickr would have had for the official Obama photographer in the lead up to the 08 election.)*

1. **Sharding via a Lookup Service**– This method works by having some sort of directory service which you query first to ask “what shard number will this users data exist on?”.Â  It’s a highly scalable architecture, and once you write scripts to be able to migrate users to/from shards you can tweak and rebalanced to make sure that all your hardware is utilized efficiently.Â  The only problem with this method is what I stated at the start: *it’s complicated*.

*(***Note:***I’ve left out some of the more complicated sharding architectures.Â  For example; another solution is to have shards all store fragments of data, and to cross backup those fragments across shards.)*

### Why is it so complex?

The reason it’s complex comes down to two reasons:

1. The application developer has to write more code to be able to handle sharding logic (this is actually lessened with projects such as [HiveDB](http://www.hivedb.org/).)
2. Operational issues become more difficult (backing up, adding indexes, changing schema).

I think that a lot of people remember (1), but (2) can be a real pain point.Â  It can take a lot of work to build an application that works correctly when you are rolling through an upgrade where the schema will not be the same on all nodes.Â  A lot of these tasks remain only semi-automated, so from an operations perspective there can often be a lot more work to be done.

*This concludes Part 1 – I hope I’ve justified why we are not covering sharding.Â  In Part 2, I will write about something that is going to be in the course – “XtraDB: The top 10 enhancements”, and in Part 3 “XtraDB: The top 10 parameters”.*

### Related

[Choosing a good sharding key in MongoDB (and MySQL)](https://www.percona.com/blog/2015/03/19/choosing-a-good-sharding-key-in-mongodb-and-mysql/)March 19, 2015In "MongoDB"

[What's new in TokuMX 1.4, Part 4: Smaller, faster sharded clusters](https://www.percona.com/blog/2014/02/20/whats-new-in-tokumx-1-4-part-4-smaller-faster-sharded-clusters/)February 20, 2014In "Tokutek"

[MongoDB 3.4: Sharding Improvements](https://www.percona.com/blog/2017/01/04/mongodb-3-4-sharding-improvements/)January 4, 2017In "MongoDB"

[Morgan Tocker](https://www.percona.com/blog/author/morgan/)

Morgan is a former Percona employee. He was the Director of Training at Percona. He was formerly a Technical Instructor for MySQL and Sun Microsystems. He has also previously worked in the MySQL Support Team, and provided DRBD support.

**sudhir** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-805849)

“(Tip: There are a few famous cases of both (a) bad hashing algorithms and (b) users becoming unequal all of the sudden; You don’t want to shard based on the first character of a username – as there will be a lot more ‘M’ than ‘Z’. For users becoming unequal all of the sudden, it’s always interesting to think of what scaling challenges Flickr would have had for the official Obama photographer in the lead up to the 08 election.)” – Probably I am reading an old article –  
can sharding on reverse of username work. It should be more random?

<http://marksverbiage.blogspot.com/> [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625292)

I couldn’t agree more. Sharding is complicated and is highly likely to introduce bugs into your application. These bugs will be subtle, edge-cases, for example, where a customer has moved to a different server and then want to report on data which cross the boundary when they moved. These are going to be difficult bugs to find and very hard to fix.

[morgan](https://www.percona.com/) [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625306)

@Mark – I didn’t mention bugs, but I’m with you on this one. With sharding (or complicated replication setups) having an application error/disaster mangling your data is not fun. Trying to sift through and re-piece it together is like pulling teeth.

[Ori Lahav](http://olahav.typepad.com/) [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625325)

thanks Morgan for the great post.  
I would like to hear your opinion on MySql Cluster – isn’t that the way to do sharding?

**Morgan Tocker** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625334)

@Ori – thank you.

In response to MySQL Cluster, I throw this into a different category of sharding (although from a technical perspective it is method 2: sharding by hash or key). But my reason why it is different is that by managing the sharding internally some of these pains have been taken out for you.

Having said that: I’d still consider “migrating to MySQL Cluster” as a similar decision to “migrating to another database” – there are a lot of workloads it isn’t that well suited for, and it can be difficult to simply retrofit into an existing application. I’d like to see join performance (via hash joins) improved for example, but there are a few more issues than that.

**Brooks Johnson** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625336)

I agree with the overall statement that sharding is often overkill, I’m not as fond of partitioning by application as it can often lead to complex issues. For example, if you need to join two large tables that are on separate machines one ends up writing the join in the application server, which is far more complex than a simple sql statement. Effectively, one is no longer using the database to full potential and it isn’t that much different from a key/value store. If what you effectively have is a key/value store, why not use a key/value store optimized data store (one with range scans)?

Alternatively, with sharding, one can still join various tables other if the sharding was done by a good key. Sharding keeps much of the coding flexibility, but as pointed out, with a heavy, often too heavy, operational cost. I’m not saying that application partitioning doesn’t make sense in many or even most cases, but that sharding can make more sense in a relationship rich application where the joining of those relationships is important.

**Morgan Tocker** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625338)

@Brooks – It would depend on the application if it’s possible – but you’re right, even with Partition by function you loose some functionality (although less than the other sharding types). Many applications have different requirements that allow you to separate out this functionality reasonably elegantly;

– Logging and searching are two problems that are normally pretty expensive, and a lot of applications have to solve them.  
– If we look at something like this wordpress install, the comments could reasonably safely be moved to another server/group servers.  
– If we look at something like Facebook/Flickr/Livejournal the friendship system is usually expensive enough that it can be moved too.

I think that you can strike a good middle ground before having to throw out the relational model completely.

[David Holoboff](http://mysqlhints.blogspot.com/) [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625339)

Great article. We are indeed working on a large working set, well beyond 256G, and I can relate to the issues you listed.

Interesting point you made about the first character of usernames… I have a question — what about creating a seperate table for each user?

**Morgan Tocker** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625348)

@David: That would depend on the number of users

It’s a common question with no answer that is correct 100% of the time.

If there are too many, you end up with a lot of small files. In MySQL-terms that means at least 1 file per table (.frm), and a lot of pressure on your filesystem to have that many in one directory (a good rule of thumb is probably aim for less than 1000 per directory, but each filesystem implementation is different). There’s also a lot of problems with opening tables scalability in MySQL. See <http://www.mysqlperformanceblog.com/2006/11/21/opening-tables-scalability/> and <http://www.mysqlperformanceblog.com/2009/06/16/slow-drop-table/> for example.

**Morgan Tocker** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625350)

@David: I think I hit ‘submit’ too quickly:

Another problem that you need to be careful of, is how much memory InnoDB allocates for data dictionary (it’s unlimited in size!). We have a patch that addresses it here:  
<https://www.percona.com/docs/wiki/patches:innodb_dict_size_limit>

**Peter van Dijk** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625498)

For those of us dealing with enormous working sets, there really isnt any other option than to shard, but I dont think sharding needs to be as doom and gloom as you point out.

I think that “complexity” as you mention it, is probably too broad of a term to use in reference to the structure of sharding systems.  
Yes, they are complex, but that’s not necessarily a bad thing. Mysql is complex. that doesnt make it bad.  
The real danger of implementing sharding is bad design. Bad design is ‘bad’ complexity.

In relation to your article, there’s a number of things which i wanted to mention, in no parcticular order:

– I disagree with the assumption that putting your entire database on a single server is a good idea, for the exact same reason that, even though you could have a single web server with 1000 xeon’s in it and a TB of memory, doesnt mean that it’s a good idea. There are a lot of people out there that cant see out from the confines of their database infrastructure. I’d much rather have 4 webservers than 1. Same applies to database infrastructure.

– Complexity (read: difficulty of use) for application developers depends on the application and framework design, not the fact that it’s sharded. If you design your sharding infrastructure with simplicity for developers as a primary goal, it’s possible to implement frameworks where the developer only needs to have a rudimentary understanding of the database infrastructure (even though a full understanding of how things work is desired. Still, our new developers are able to use the system straight out of the induction process). It’s just as easy to have a bad unsharded framework as it is a bad sharded framework.

– Backups arent all that scary, just that you have to do them from more than one server. If you’re considering sharding, you may want to ensure that you have good automation for these sorts of things well in advance, not to mention things like good monitoring systems.

– Schema changes and other migrates are not always that much of a problem. If your entire development team is on the mysql command line making live schema changes, yes, you’re going to have issues. We found that the best approach is to treat database changes as a part of the application. Our developers write code to change the database, not queries. This gives error handling, logging, full historys of everything that gets changed, and a number of other benefits. In terms of shards, it allows us to keep all of the shards consistant in a much easier and managable way.

– In organisations of varying skill level, it’s important to have development processes in place to cope with the fact that not everyone is going to understand everything that happens at the database level. This can potentially become more important once sharding is implemented.

– Some days i wish i had a bigger budget. There’s some really cool things you can do in terms of sharding solutions that you could implement at the mysql or storage engine level, but they’re a bit to complex to discuss here

In summary, I think what I’m trying to say is that:  
– Like you said, most people shouldnt be sharding. It’s really only a solution for the high end. Make sure you’re capable of fully understanding your entire architecture.  
– Complexity isnt evil, but bad design is. If you do have to shard, 90% of your work should be in the design, not the implementation, otherwise you’re going to end up with all those problems you mentioned in the article.  
– There are also some other benefits to sharding which you dont list here. Shards can be an amazing tool when dealing with clusters in different geographical locations.

[morgan](https://www.percona.com/) [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625554)

@Peter: It’s good to have disagreements – it sparks discussion!

There’s a difference between saying “you don’t want to shard” and “you shouldn’t shard”. I think my wording is very careful here, and the point is that there will be some situations where you still have to. There are people with working sets larger than 256G (including yourself), but a large number of people will also have less than this number.

Having your data in one place does not necessarily mean having no redundancy. In my point about “too many writes” I explicitly mentioned slaves, which means I still want replication. Where you said “Iâ€™d much rather have 4 webservers than 1. Same applies to database infrastructure.”, this is something I disagree on. Yes – I’d rather have more webservers, but that is because webservers have no affinity. Replication is like RAID1, Sharding\* is like RAID0. If you have x10 as many shards, then you actually have a much higher chance of failure.

\* In the examples I provided, I would typically use replication as well.

**Peter van Dijk** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625563)

Yeah, i wasnt disagreeing with you on that first one at all. I definately agree that most people ‘shouldnt’, ‘dont want to’ and ‘shouldnt want to’ shard. same thing. different words. That being said, i’m still a great proponent of their use where it’s warranted.

RE: “In the examples I provided, I would typically use replication as well.”, and “Having your data in one place does not necessarily mean having no redundancy.”,

I couldnt agree more, but this isnt specific to either centralised database or to shards.

Having 4 database servers or shards shouldnt mean no redundancy. In an ideal world, i’d typically assign a hot spare that can take the role of an existing shard in the case of a failure. Again, this is a design level problem. Using sharding as a tool doesnt prohibit you from designing a redundant, fault tollerant infrastructure.

I think your comments on failure rates are a little too simplified and dont really show the entire problem. Even though it increases the chance of a failure, it also lessens the impact by an equal amount. For example, in the case were you do have 10 shards, if one fails, only 10% of your customers would be affected at all. If you factor replication and/or failover to a hot spare on top of this, it actually looks more like RAID10 rather than RAID0.

Of course, this can add complexity, but it doesnt need to be badly designed. You can automate these kinds of processes. It’s like anything else though: how much do you want to spend, and how much time do you want to spend doing it. Horses for courses.

Those points aside,  
I think what it really comes down to, is that once you start doing the scalability research for your application, you’ll find that there are any number of different approaches that you can take. Sharding may not be appropriate for any of the points that either of us have made,  
but I like it because it gives me the opportunity to do many things that a centralised database infrastructure does not – such as partitioning shards geographically while still maintaining centralised management, let alone things like performance and fault tollerance, possible increases in data integrity, and loads of other stuff i cant remember

**Andy** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625572)

Morgan,

You mentioned HiveDb in your post. Is HiveDB still alive? They made some announcements a long time ago and there has been no activity on their site for over a year. No code released.

So HiveDB seems pretty dead. So does Hibernate Shard.

What are some good frameworks that support sharding?

[morgan](https://www.percona.com/) [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625776)

@Andy – I included HiveDB as an example, since I’ve known people have used it in the past… but over 1 year of inactivity could be death in open source terms. I’m not sure of any other good examples – most of the stock frameworks are actually pretty bad at supporting read/write splitting let alone sharding.

**TS** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-625886)

@morgan:

I agree that applications should avoid sharding for as long as possible.

Recent development in the MySQL and File Systems point toward migration to ZFS hybrid storage pool for accelerating synchronous writes and L2ARC read accleration.

Smugmug already migrated to ZFS hybrid storage pool, although I disagree with the NFS methodology. Since NFS uses 10Gbit switches and adapters, which are about 500 dollars a port and you need dual adapters and redundant switching…that’s 8 ports total to connect two machines with redundancy = $4000 total. That is far more expensive than dual path SAS HBAs for DAS JBODs. (You needed DAS JBODs on the ZFS NFS server too).

I dream of one day you can run MySQL with perfect scalability with this machine setup:  
<http://www.flickr.com/photos/gavinmroy/2871736448/>

That’s the standard DB config that PostgreSQL can scale to. 32 cores with 100 spindles. If we now use SSDs in the caching tier on top of it with X25-M/Es, then you can buy 500K IOPs easily.

**Dave** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-627491)

I’ve not seen much mention of partitioning using MySQL 5.1 and above – how does that relate to the sharding issue, and is it a good intermediate step to work with before having to restructure applications to handle “proper” sharding across multiple data servers? Especially if you have budget constraints!

[morgan](https://www.percona.com/) [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-627503)

@Dave – partitioning \*may\* help buy you some time. Lets look at the two reasons that make us shard:

1. Very large working set:  
– Partitioning may change some execution plans so that less indexes need to be loaded in memory. How true this is would depend on the queries.

2. Too many writes:  
– There’s a chance partitioning might make some index updates cause a little less random IO, but this hopefully is already negated by the InnoDB insert buffer feature.

There are some annoying limitations which make MySQL Partitioning nothing like in other databases (Oracle), but having said that I’ve also managed to find a few suitable use cases every now and then as well. Maybe there’s room for a followup blog post some time.

**Dave** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-628427)

@Morgan,  
Thanks for the info – our working set is quite large, with a few tables hitting 150Gb (including indexes), and there are frequent data replenishes which delete a large chunk of information and then replace it with updated information. In addition, queries against this information are frequent and very varied!

Unfortunately, InnoDb is a non starter here due to budget (and therefore storage) constraints, so we’re stuck with MyISAM for the time being.

Roll-on a nice conclusive blog post re. partitioning (including advancements in MySQL 5.4), InnoDb, Memory usage, storage requirements, etc that I can take to the powers that be and get them to hand over some more cash! LoL!

[Baron Schwartz](https://www.percona.com/) [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-628588)

Dave, I honestly doubt that partitioning is the miracle solution for you. It is no silver bullet.

I think if the powers that be understood the real cost of MyISAM, that equation would flip on its head.

Maybe you can think about archiving.

**Morgan Tocker** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-628653)

@Baron – thanks for being a little less politically correct than me  I re-read my comment and realized I said “it may help”, but I really meant “It may help, but I don’t think it will fix it”.

@Dave – When Baron was talking about archiving, he was probably implying mk-archiver – <http://www.maatkit.org/doc/mk-archiver.html>. It’s an excellent tool if you find you don’t need older records.

**Dave** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-629104)

@Baron and @Morgan,  
Thanks again for the info, we actually archive our data yearly, and only store about 4 rolling years of data – unfortunately, most (if not all) of it is in regular use…

I have got some interesting numbers back from my analysis though;

My main area of concern was the replacement of data with refreshed information. This involved DELETEing large chunks and INSERTing new data. There is a single field that is primarily used when doing the DELETE, which contains roughly 200 unique entries with a fairly even distribution (eg DELETE FROM table\_name WHERE field\_name = 5).

After partitioning the data into 10 chunks based on the HASH of this field, we have experienced a four-fold improvement in the DELETE command, which is now able to DELETE 8.2m records from a 1.5b record table faster than a similar DELETE of 3.6m records from a 500m record table.

Maybe our application just happens to be one of those lucky ones that partitioning \_may\_ help!

I have yet to get some proper performance numbers on the SELECT side of the data, but I shouldn’t imagine it will be any slower than the current layout, and the data replace section is such a key one, I’m prepared to forego a small performance hit for the increases I’ve already mentioned.

[peter](http://www.mysqlperformanceblog.com/) [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-630016)

Brooks Johnson,

The functional partitioning makes sense under 2 conditions

1) The functional partitions are independent enough, hence you do not need to join data frequently between them at all. Putting different tables on different hosts is not the idea, putting “Forum” on one database host “Wiki” on another and “Bug System” on the third is.

2) The gain you’re looking for is relatively small. It is often easy to find 3 independent functions with one of them responsible for 50% of the load (and hence split giving you double capacity) but getting 10x this way is rarely possible

[peter](http://www.mysqlperformanceblog.com/) [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-630023)

Peter,

We’re not against sharding. In fact we help a lot of people how to shard properly. The problem is now it is such a buzz word so people with 1GB data set start sharing even if it is never going to grow over 10GB

The bad design is one issue the other however simply working with sharded data. Really it is a lot depended on how tightly coupled is your data – for example hosting million of separate blogs is very easy to shard because there is no interdependencies.

The large data also indeed causes operational concerns – databases in TB range are often not fun in MySQL due to challenges with backups and expecially things as alter table.<http://www.mysqlperformanceblog.com/2006/10/08/small-things-are-better/>

With backups – the concern is cross box consistency. With single box you can restored backup from yesterday and it will be consistent (even though not up to date) – in sharded envinronment backups will correspond to different point in time and so would not be consistent.

[Log Buffer](http://pythian.com/blogs/about-log-buffer) [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-631747)

“On the MySQL Performance Blog, Morgan Tocker explains why you donâ€™t want to shard. (It has nothing to do with The Dark Crystal, I already checked.) […]”

[Log Buffer #158](http://www.pythian.com/news/3561/log-buffer-158-a-carnival-of-the-vanities-for-dbas)

[Dathan Vance Pattishall](http://mysqldba.blogspot.com/) [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-650159)

I do agree you should only use it if you need to do realtime queries that are user facing across a very large dataset (10’s of TBs)

Sharding is super easy if you know what you’re doing. 1,2 are not an issue at all for me. I can isolate all traffic for super powers users to an in memory DB which will not be overrun if done correctly.

**Anthony Berglas** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-682354)

MASTER/SLAVE REPLICATION

You forgot to mention that if there are many more reads than writes (common case) then running slave, read only databases off a master provides scalability without having to resort to sharding.

Also, Sharding only works if the shards are largely independent, eg. GMail user accounts. But sharding and an integrated system such as ERP is likely to slow it down as the shards need to communicate.

Some databases (Oracle) can horizontally partition a table (and I hope thus a database) automatically based on key values. That is the right approach. Keep the logical/physcial separation. Google style sharding and big table are a hack that ignores 50 years of database theory.

You also forgot to mention that if you take a couple of big tables out of a database, you loose locking and transactions. Not a good option. (Oh, I forgot, MySql does not have locking anyway ;).)

**Peter van Dijk** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-682446)

@Anthony,  
I think it might be helpful to consider that sharding can be used as another level of abstraction in a complex system, specifically, (and obviously this is a fairly gross oversimplification, but probably still valid)

Where a raw disk has a filesystem placed on top of it to aid in organisation of the underlying data,  
a database server typically will use table structures on top of a filesystem to further abstract the low level operations of storing information in files into something that can be searched, modified and more easily maintained in a structured form.

Similarly, shards, when implemented in a useful way, are able to abstract a given system in such a way that you’re able to distribute storage across an arbitrary number of machines. In our case, we have shards in different physical locations, where things like replication are completely impractical.

By extension, the reason that sharding isnt really a good idea for most people is the same reason that, for example, if you want to copy your holiday photos onto a usb thumb drive, you dont use a database to do it. In many cases, that extra level of abstraction is completely useless and simply adds complexity.

There are a lot of people who have spent a lot of time researching this area, and, particularly in the web world, it is an invaluable tool for dealing with enormous data sets. I think the notion that it’s ‘hack that ignores 50 years of database theory’ probably just indicates the need for better education and understanding of how it can be used as a tool.

**Morgan Tocker** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-682806)

@ Anthony – I wrote under bullet point 2, that sharding was often a response to being write heavy (“too many writes”). I didn’t forget about replication, this article just has a specific purpose

Most applications can be broken down into shards (see my comment #6 for examples), but I don’t dispute this can be difficult in others. The example I often give for an application that won’t shard is IMDB’s database. I don’t think there are many good ways to divide actors up, and the movies they star in.

A small correction to your point about locking: Readers don’t block writers in InnoDB because of MVCC, but MySQL does have locking. Related to your point though is cross-box consistency, and it is an issue. Peter wrote about this in comment #23.

[Clement Huge](http://www.clementhuge.overblog.com/) [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-771733)

Hello,

I am also working with massive amount of data and I love sharding for a lot of reasons.  
The main one is that even though it complexify the operations, it gives you a lot more flexibility and scalability.

I have worked with traditional approach and sharding approach. Right now, I work with a traditional company that has so much money that they would rather buff the hardware as crazy as possible to fit everything in a few boxes.  
I also work with less traditional and they prefered having a lot of boxes to serve shards.

When I was called to the rescue to the non traditional company, it was primarily because of the complex operations: how to publish new objects to all the shards (keeping versioning as well) and how to publish or replicate scripts as well. True! it was a challenge but it was fun to find the solution.  
The second challenge was about datawarehousing transactional data to purge data on the servers.  
Bottom line, we had very few indexes and had pretty much servers about 16GB of data, representing each one shard while we were server billions of transactions per month!  
The more traditional management (the datawarehouse) was then the most difficult administration part (with defragmentation/archiving/partitioning, and getting datamart and datamining part.  
based on my experience, I definitely prefer sharding which gives you a more intuitive solution for high availability and high performing having active/active nodes… oops shards (;-)).  
Well you got my point  It competes seriously peer-to-peer replication, mirroring, log shipping and clustering!

**Joe Dugan** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-774199)

This is a good article, and gives some very good points on why not to shard. But there are many reasons to shard as said in the earlier posts. I have seen some dramatic performance increases on mySql and Postgresql. There is a very good company called dbshards.com that has some very impressive benchmarks. I would recommend people look at all the options before they decide not to shard.

I also read their article on reliable replication. It seems much better than the standard mySql replication.

Sharding often solves scalability issues without much headache.

**Glenn** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/" \l "comment-883124)

You forgot a major use case: locality. Having the backend server in the USA is bad for users in Italy, yet you often don’t want completely distinct backends–users in one place should be able to talk to users elsewhere, and users should be able to be transparently migrated if their locale changes.

**James** [Reply](https://www.percona.com/blog/2009/08/06/why-you-dont-want-to-shard/#comment-892400)

Locality: Glen this use case is fine if your application is not interested in data in the other countries. If you can seperate the data in your application to geographical regions then you will obtain a performance gain, but with a lot of applications this is not possible. Users in England are interested in the data in Italy, so sharding your data based on country will not help as you will have many cross domain/shard queries. Mongo does not provide any locality functionality to meet the use case you describne, that I know of. Does anyone know if mongo solves the issue of locality?