THE ART OF DATABASE SHARDING

Maxym Kharchenko, Amazon.com

Contents

What is Database Sharding

Database Sharding is a technique of breaking a big database into smaller pieces (called: Shards) and then running each piece on dedicated commodity hardware in a *shared nothing* architecture.

The term *"Sharding"* was popularized by Google after publication of Big Table architecture in the mid 2000s, however the concept of shared nothing databases has been around for a while. It has now become a de facto standard for managing huge data sets with traditional databases.

In this whitepaper I will discuss why data sharding makes sense when load grows beyond the capabilities of reasonable hardware, how to plan and build your shards as well as what you can expect (both good and bad) after sharding your data.

Big Data and the Scaling challenge

If you are managing data today, chances are, you are going to be managing *more* data tomorrow. And, no, not a smidge more, a lot more! You can say what you want, but it's just a fact of life.

What's the point of managing data? In a nutshell, it's to make the data useful. I.e. data has to be available, data operations (i.e. searches) must be reasonably fast and their timing (or latencies) must be predictable.

Managing data obviously becomes more difficult as the size of the data grows. More data very often means more load: more users connecting, more queries running at the same time and more transactions hitting the database, often, at the most inopportune times. And since hardware that runs the database has more or less constant resources, it is virtually certain that at some point it will be *insufficient* to handle the increased load.

Fortunately, in most cases, data and load do not grow very fast. It takes *years* for a reasonably sized piece of hardware to be completely overrun. In addition to that, when it finally does get overrun, the march of technology and its peculiar manifestation, known as the Moore's law, assures that you are not going to pay through the roof to replace it, even though the new hardware will need to deal with a much bigger load.

As you probably remember, Moore's law loosely states that hardware capabilities double every 2 years [(1)]

That means that as long as your you can size the system to hold a few years worth of data and your data/load growth is *"reasonable"*, then you can just replace the hardware every so often and you can do it on the cheap.

This is the essence of traditional scaling. And it works for many, many people.

But to every rule there are always exceptions.

Your software might be enhanced with new capabilities that require a lot more data and processing power ... or people might just like what you are doing and go viral with the orders.

Suddenly, your load grows much faster than hardware powers under the Moore's law and you have a big scaling problem.

[(1)] Technically, Moore's law states that the number of transistors placed inexpensively in an integrated circuit doubles every 18 months, but there are similar laws that describe other hardware resources as well. I.e. Kryder's Law describes similar relationship with the disk storage, Nielsen's Law covers network bandwidth etc Bottom line is that so far, computer capabilities double every 1.5 to 2 years and this is likely to continue for a while

THROWING HARDWARE AT THE PROBLEM OR VERTICAL SCALING

The traditional way of solving this scaling challenge is to move the database *to a bigger machine*, officially known as *vertical scaling* and unofficially as *throwing hardware at the problem*.

The beauty of this solution is that it's very simple. Apart from a few system administrators installing a new hardware box and a DBA spending a couple of hours moving data there, nobody needs to do a thing! No need to develop new software or spend hours testing new algorithms. Just replace the hardware and you can double or triple system capacity in a matter of hours!

However, there are a few issues with this approach.

First of all, this process is not infinite, in other words you cannot *"go big"* forever. Yes, you can move from 2 CPU/2 Core machine to 4 CPU/8 Core and then, maybe to 8 CPU/10 Core. But at some point, you are going to hit the wall as your current hardware will be at the bleeding edge of what's possible.

In addition to that, there is a matter of cost.

Upgrading a small system to medium size will probably give you adequate bang for your money. But as you move into more advanced hardware, your end results will get more and more diminished.

I.e. this is a simple SPEC2006 Rate [2] comparison [3] between 3 Intel processors, each of which supports 10 cores with the difference being the number of processors on the motherboard.

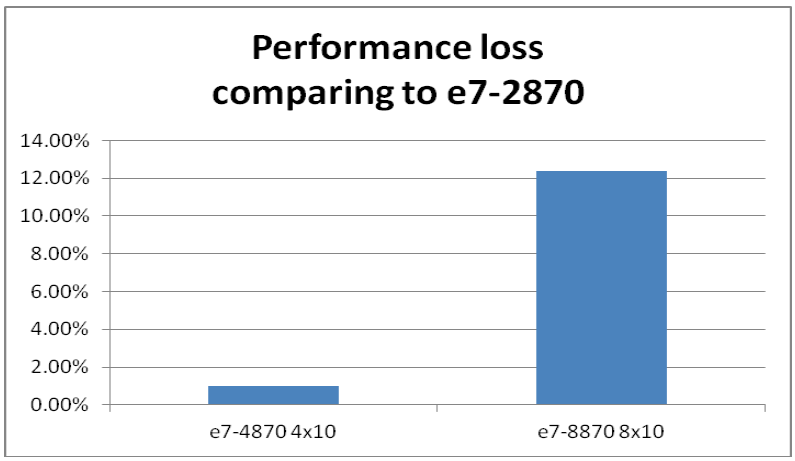
Processor	CPUs	Cores per CPU	Total Cores	CINT2006 Rate	$$ per CPU	$$ Total	Potential CINT2006 Rate	Performance loss	$$ / per CINT2006 Rate
e7-2870 2x10	2	10	20	505	$4,227	$8,454			$16.74
e7-4870 4x10	4	10	40	1000	$4,394	$17,576	1010	0.99%	$17.58
e7-8870 8x10	8	10	80	1770	$4,616	$36,928	2020	12.38%	$20.86

As you can see from the table: as the number of cores goes up, throughput goes up as well (in other words, more cores = more CPU power), but the increase is not linear and gets smaller as we move further along. In other words, with more cores, each core contributes less to the total, likely due to synchronization and scheduling issues.

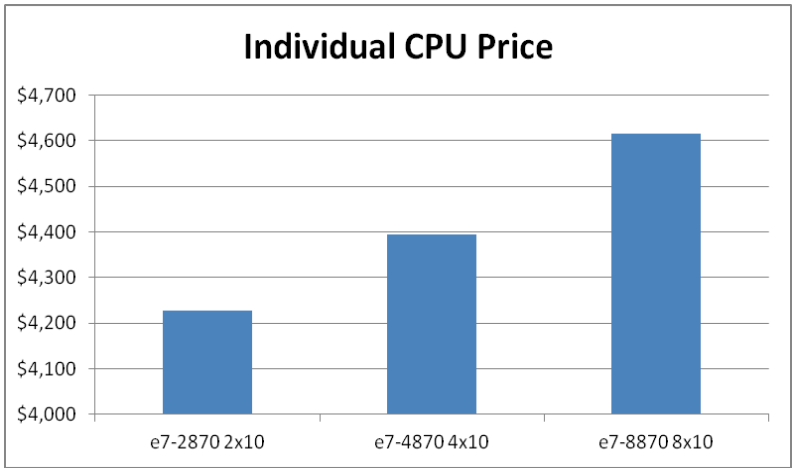
And this gap gets bigger as we increase the number of cores:

[2] SPEC2006 INT Rate can loosely be translated as "system wide CPU throughput"

[3] Comparison is done between IBM BladeCenter servers



On the other hand, the price for *"processors supporting more processors"* certainly does not get diminished:



The bottom line is that when you are moving into advanced systems, you get hit with a double whammy:

1. You get less of the system overall

   and

2. You also pay more for it

Please, note that this unfortunate situation is not a result of our choice of particular technology. There is nothing inherently wrong with e7-8870 or 8 CPUs on a motherboard. In fact, I have no doubt that we will be doing the same comparisons, using e7-8870 or a similar processor as a 'good choice' baseline a few years down the road.

It is rather the fact that at the moment, this technology is *advanced*, which has both technical (i.e. not all the kinks have been worked out) and economic (i.e. less demand, higher prices, bigger margins) implications.

COMMODITY VS. STATE-OF-THE-ART SYSTEMS

If scaling UP to bigger systems is not the answer, what is?

Well, the rather straightforward alternative is to scale OUT, that is: use *more* machines, not *bigger* machines.

This principle:

More > Bigger

is not unique to computer systems by the way. Many projects nowadays use commodity components to achieve better results at a fraction of the cost of traditional approaches.

For example, space junkies (such as me) would probably recognize this image:

It is one of relatively few NASAs Deep Space Network (DSN) dishes that are used to track and communicate with satellites all over the Solar system (and beyond). Each dish is a magnificent, state-of-the-art, custom built and very expensive piece of equipment.

The picture below is a more recent radio telescope project, Allen Telescope Array (ATA), which uses a lot of inexpensive *off the shelf* dishes.

The interesting thing about these two projects is that they are roughly equivalent in radio sensitivity (that is, their usefulness). And yet, ATA manages to achieve that at 1/3 of the cost of DSN.

But there is more:

- *Downtime*: Going down is not a big deal for any ATA dish as it is only a small part of the whole. Downtime for any DSN dish on the other hand is a *huge* deal that threatens the entire service.

- *Maintenance:* Since ATA dishes are cheap, they are also *expendable*. If they go bad, they can be simply thrown away and replaced (or, at least, they can be repaired *leisurely*). DSN dishes are huge investments that have to be regularly maintained and repaired.

- *Extendibility:* For the same reason, it is much easier and faster to extend ATA network by adding a few more dishes. Extending DSN network will take time and require huge capital expenditures.

As you can see, replacing a few expensive components with lots of inexpensive ones makes a lot of sense, both technically and economically.

And this, in a nutshell, is what *data sharding* is about.

DATA SHARDING =

Splitting your data into small manageable chunks
AND
Running each chunk on cheap commodity hardware.

Designing Shards

As we just established, the idea of sharding is to take huge monolithic data and chop it up into small manageable pieces. It begs several questions:

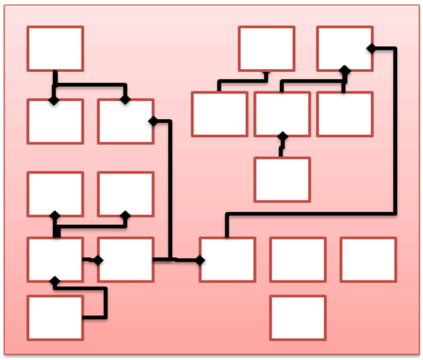
- What is it that we are chopping?
- How do we chop?
- Into how many pieces?

Let's take these questions one at a time.

What is your 'Big Data' exactly?

We all tend to think of data as a singular term, a big entity that we can work with. Something that looks like this:

But the reality, at least in a relational world, is far messier. I'll bet that the data schema that you have has many tables, so it probably looks like this:



The question now becomes, how do you shard (or split) that?

Vertical Partitioning

Sharding, technically, is splitting of things that are *similar*. So, before we can start *sharding*, we need to determine whether the data that we have describes *similar* or *different* things. In other words, we need to classify our data and split it into groups or subsystems.

How to classify the data

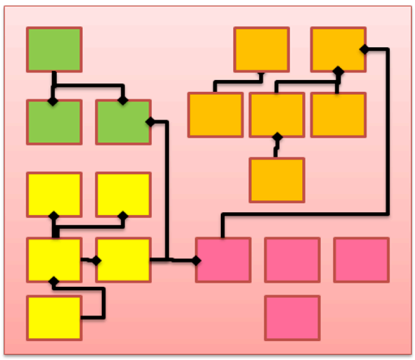
There are, actually, several common techniques:

You can split the data by *function*. I.e. if you support e-commerce web site, your buyers can register, make orders, write reviews etc, while sellers can list their items, set prices, ship merchandise etc ... This is commonly called *splitting by verbs.*

Alternatively, you can split your data by *resource*. In the same example, your web store can have a catalog of items, users registry, orders database etc ... This is also called *splitting by nouns.*

If you want to learn more, Martin Abbot's excellent book on scalability describes this process in fairly exhaustive detail.

When classification is completed, the end result should look something like this:



Breaking subsystems apart

The next step is to split subsystems (that we just identified), so that each can run independently on its own separate box(es).

This process is called *vertical partitioning* and is usually a logical first step in splitting your data.

It is interesting to think of what we gain and what we lose by splitting data vertically.

Benefits should be fairly obvious: data pieces are now *smaller* (remember, that's why we started the process in the first place). This, along with the fact that the data is more *"focused",* means that data is also much better *cached*. This should do wonders for data queries.

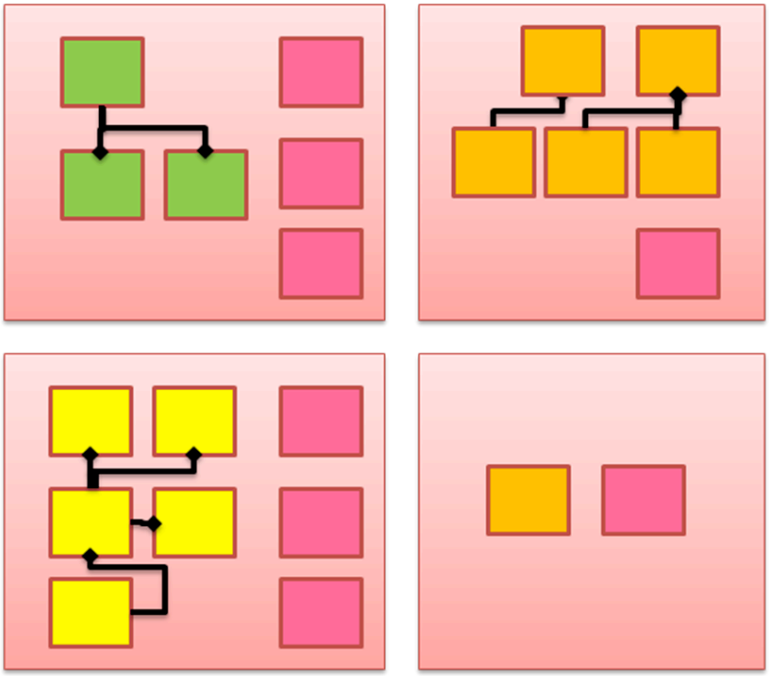
An additional benefit is that developers, who support, say, user registration, will now be able to focus exclusively on their own isolated data piece, thus becoming much deeper domain experts. This generally leads to better code.

On the other hand, a vertical split is not without its problems. The main thing that gets lost here is: enforced relations between data, or, to put it simply: *foreign keys*.

While it is possible to envision a system where foreign keys will be externally maintained, such systems are usually exceedingly difficult to build and, besides, you can only rely on such "distributed constraint" if you accept the fact that your data may *not* always be available. This is a consequence of a CAP theorem.

Thus, when designing data schema that *might* undergo vertical splits, one needs to essentially *forget* (or, at least *not rely on*) foreign keys constraints between tables, as they can be "ditched" at any time. This makes many people cringe.

Nevertheless, considering the alternatives, vertical splits are a good thing and a necessary first step to make your data small and manageable. After completing the split, your system should look like this:



Vertical Partitioning: Special Cases

As with everything, there are always a few special cases with vertical splits: things that are logically different but do not get split and, alternatively, things that should belong together and yet are still separated.

First category often includes data, such as setup tables, domain dictionaries etc. This data is often relevant to many parts of the application, thus making it difficult to break it apart. On the other hand, setup data is usually small and static and might not warrant a separate hardware partition or service. Instead, full copies of this "common" data can be simply included along into each vertical "partition" [(4)]

On the other hand, there might be data that functionally belongs together and yet individual pieces are so active that it still might make sense to break them apart. A good example here would be: queue tables.

[(4)] In reality, not a lot of data can be classified as truly static. Thus, these full copies of common data will still need to be occasionally maintained, either by replicating them from a common source or scheduling occasional data pushes.

DATA SHARDING - Step 1: Partition vertically, first

Horizontal Partitioning or Sharding

Even after data is split vertically, chances are, we are still not done.

Consider, for example the subsystem that manages customers placing orders. Even though it is completely separated now, the load might still be too high for any reasonable piece of hardware to handle (you've got to admit that this is a good problem to have!)

So, we have to divide data even further.

The good news right now is that our data looks much simpler. It describes '*similar things*', so it probably is a small group of related tables that share some common part [(5)].

For simplicity, let's assume that are only left with one table that looks like this:

[(5)] ... and thus *can* have foreign keys between them

```
CREATE TABLE Orders (
  order_id number PRIMARY KEY,
  customer_fname varchar2(30),
  customer_lname varchar2(30),
  order_date date
```

Since the goal of the split is to reduce load (and make it predictable), it makes sense to split things evenly.

That is:

- The resulting pieces must be equal in size
- They also need to experience similar load

So, how should we go about the split to achieve these goals?

To narrow done on the best approach, let's consider several *bad* ones first.

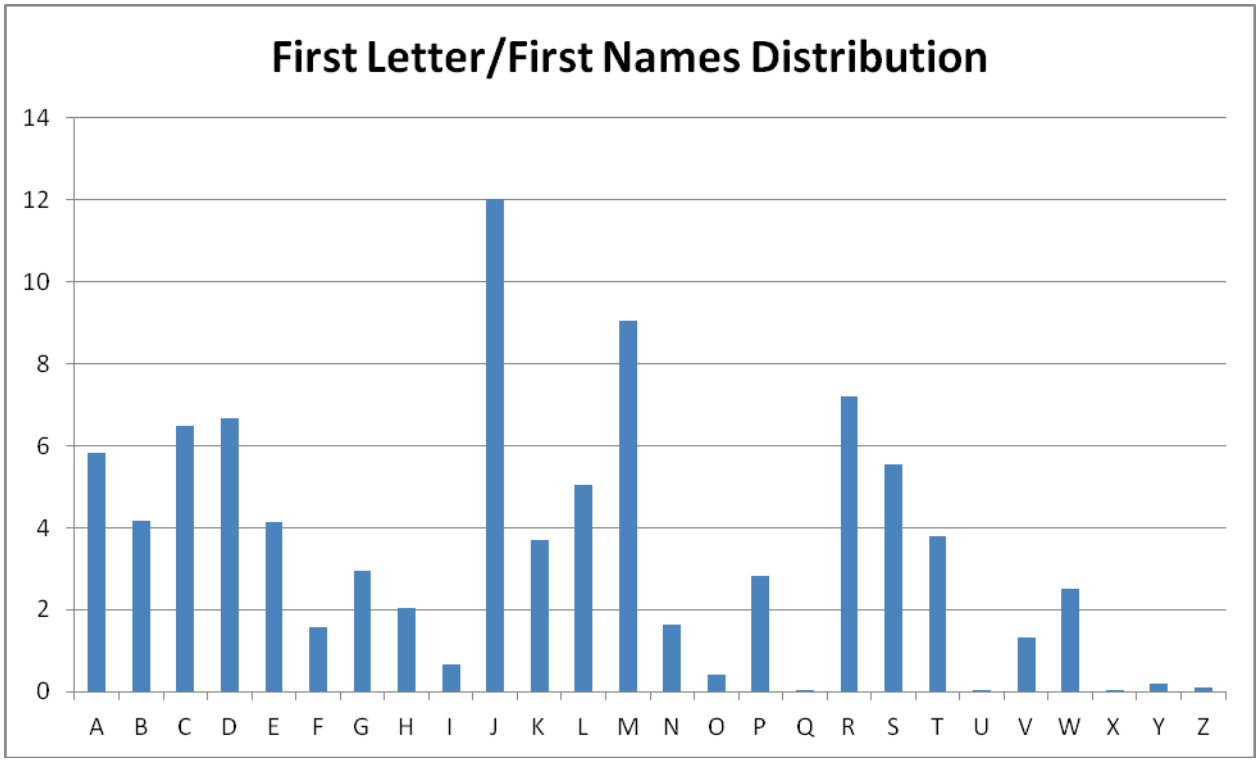
BAD SPLIT: EXAMPLE 1

The first naïve attempt might be to use a first or last letter of customer's name to map our orders into N ranges. I.e. if we break ORDERS table into 4 pieces, our ranges might look like this:

1. A-F
2. G-M
3. N-S
4. T-Z

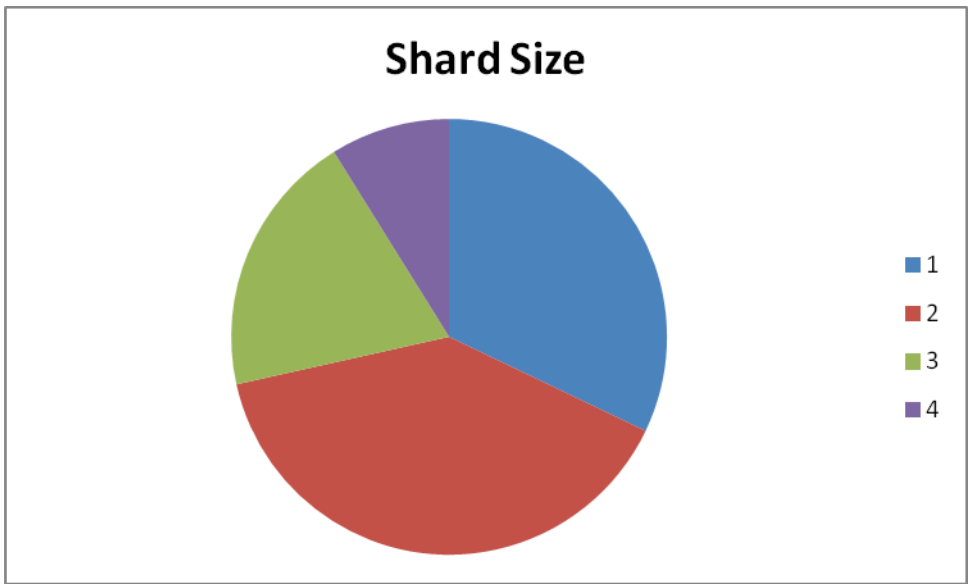
This setup is very simple but it falters on the assumption that names are uniformly distributed by the first letter.

In reality, names distribution curve looks like this:

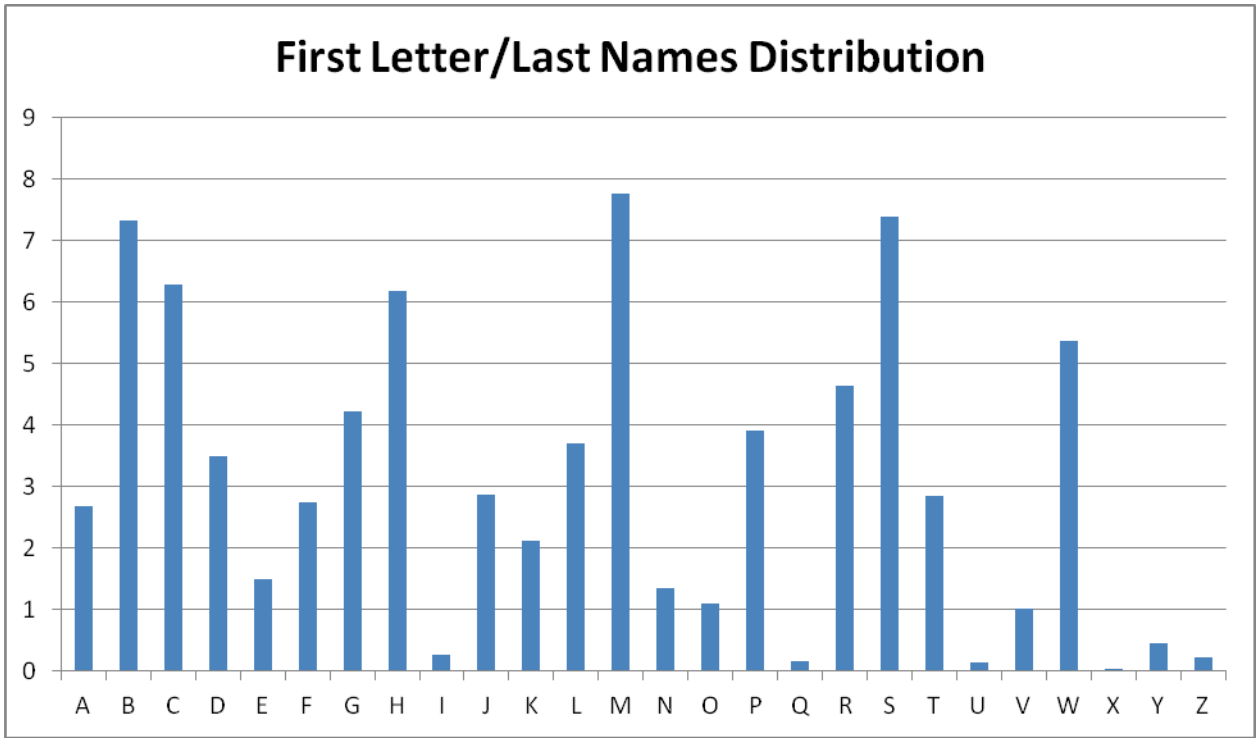


Source: http://www.census.gov/genealogy/names/

And as a result, the size of our shards would be very uneven:



And if you think that this skew only applies to the first names, think again:



Source: http://www.census.gov/genealogy/names/

So, the bottom line is that if we implement this simple partitioning strategy, we will not be able to achieve *even data distribution*.

BAD SPLIT: EXAMPLE 2

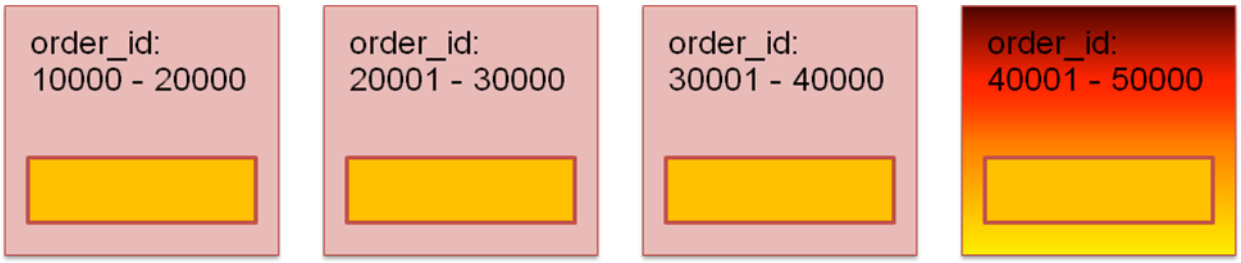
Perhaps, we just selected a bad partition key as name data tends to congregate together. What if we use an *order_id* instead?

It is simply a number and thus has no correlation to any customer in particular. Perhaps, we can divide all orders into ranges, such as:

1. 1-1000
2. 10001-20000
3. 20001-30000
4. 30001-<latest>

Unfortunately, this is also a bad idea.

First of all, *order_id* is probably sequential, so there is a strong correlation between order_id and the time when order was created. Ordering systems tend to work mostly with a very recent data, so in this split, partition 4 will receive almost all of the traffic, while other 3 partitions will be almost idle (who really needs orders that are 90 days old ?!)



Some people might counter that this problem can be solved by applying a *modulus* function to order_id instead of using order_id directly. In this scenario, 4 latest orders 30000,30001,30002,30003 will be assigned to 4 different partitions in perfect round robin fashion.

Does this solve the problem?

Not really!

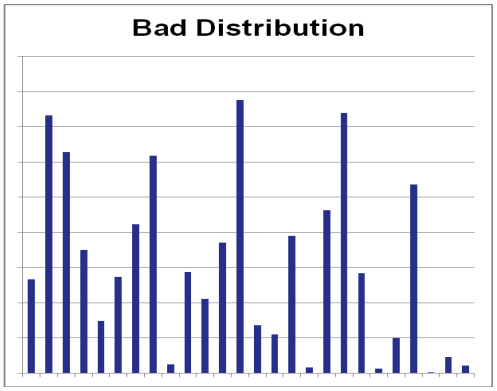
The assumption here is that order_ids are always sequential and always increment by 1. But this might not necessarily be the case forever. For example, in the future, you might want to split order ranges between different workers to simplify data load or assign certain order ranges to *"subsystems"* to simplify data warehouse merges. Suddenly, your order_id sequence is increased by 10 instead of 1 and your entire partitioning scheme is screwed (I'll let the reader figure out why…)

Sharding Step 1: Make distribution uniform

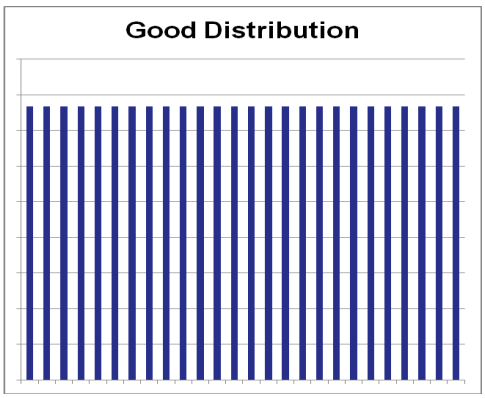
So, is there a way around it?

Yes, and it is fairly trivial. There is a class of mathematical functions that exhibit Avalanche Effect. These functions randomize input data, taking input values that are close together and distributing them far apart. More importantly, the resulting distribution is fairly uniform if the input set is big enough.

In other words, avalanche functions can transform this:



Into this:



A good example of functions with avalanche effect is cryptographic hash functions and, probably, the best known example is: *md5*.

Of course, if you *sharding key* is already distributed relatively uniformly, then a regular hash function might be sufficient.

The goal here is to find the best hashing function that makes sense for *your* data so that the resulting data distribution is as balanced and uniform as possible.

DATA SHARDING - Step 2: Make data distribution uniform (find the best hashing function)

Sharding Step 2: Decide on the number of shards

Shards are physical machines that will run your data partitions. You should have a fairly good idea how many shards to use given the size of your data and load.

And, since data is now uniformly distributed, you can apply a modulus function to map particular data record to a physical shard.

Something like this:

```
hashed = hash_func(customer_name)

shard = mod(hashed, N)
```

What should the N be?

Well, let's say we need 3 machines right now to process our load … so N should be equal to 3, right?

Wrong!

The problem here is that we assume that N will *always* be 3. But if we had to split the data once, the chances are we'll probably have to do it again…

In other words, our design does not take *re-sharding* into account.

Let me show you why using N=3 is bad if re-sharding is potentially in the picture.

Let's say that our ORDERS table has 12 records and after sharding, data distribution looks like this:

Hashed_id	Shard: mod(hashed_id, 3)
1	1
2	2
3	0
4	1
5	2
6	0
7	1
8	2
9	0
10	1
11	2
12	0

Several years (or months) down the road, it is apparent that 3 shards are NOT enough to handle the load, and so we need to add another one. Since we now have 4 shards, we need to remap our data by *mod(hashed_id, 4)* function:

Hashed_id	Old Shard: mod(hashed_id, 3)	New Shard: mod(hashed_id, 4)
1	1	1
2	2	2
3	**0**	**3**
4	**1**	**0**
5	**2**	**1**
6	**0**	**2**
7	**1**	**3**
8	**2**	**0**
9	**0**	**1**
10	**1**	**2**
11	**2**	**3**
12	0	0

But notice that even though we only need to logically redistribute 25% of the data, our new sharding scheme requires us to move 75%!

And moving data is very expensive.

Logical Shards

The common solution is to make N *relatively big* and *independent* on the number of actual physical machines.

That is: split the data into many *logical* shards and then have another layer of indirection to assign logical shards to physical.

This additional layer can be as simple as a lookup table that maps logical shards to machines, either directly (by list) or by ranges. I.e., in our case, we can divide the range into 6 logical shards and then map them to 3 physical shards like this:

Logical Shard	Physical Shard
(1,2)	1
(3,4)	2
(5,0)	3

Since mapping table is usually small and very static, every process can simply read it during startup and keep in memory for efficient shard routing.

The often asked question is: how many logical shards are *enough*?

While the real answer is: it depends ... there are several considerations that can help you calculate this number better.

- First of all, this number, obviously, should be big enough to account for ALL future re sharding endeavors. I.e. 4 is probably not a good number as you'll have difficulty going beyond 4 shards.
- It also needs to be (fairly) evenly divisible by all likely 'physical machine numbers'. I.e. 60 is a good number here as it can be evenly divided between 2,3,4,5,6,10,12,15,20 or 30 machines.
- It should be big enough to avoid the chance of collisions between 'big' data items. For example, your order table might contain orders from individuals as well as large organizations. Organizations will tend to place A LOT more orders than individuals and since you want your shards uniformly sized, you do not want many 'big producers' winding up in the same shard [(6)] In this regard, 60 is probably not a very good number.
- Finally the number should be small enough so that the size of static routing table is reasonable

[(6)] See Birthday paradox

DATA SHARDING - Step 3: Chose reasonable number of logical shards

DATA SHARDING - Step 4: Decide how to map logical shards to physical machines

A FEW WORDS ABOUT SHARDING KEY

One of the things that we, sort of, missed is: how to choose columns that data is split on?

The logic here is very similar to how you chose a partition key for "regular" table partitions.

Here are a few rules of thumb:

- Sharding key must be *immutable* (we really do not want to move data around between different shards)
- It must *exist in every table* (so that we do not need to run secondary lookups to determine it, or worse, run cross shard queries)
- It must be *granular* (so that we can use a large number of logical shards)
- And finally it needs to be *independent* (we cannot assume that any other shard will be in close proximity)

DATA SHARDING - Step 5: Chose the best sharding key

A QUICK RECAP – HOW TO SHARD YOUR DATA LOGICALLY:

Just to recap, these are the rules to logically shard your data:

- Split off "different" data (vertical partitioning)
    - If not enough:
        - Find the best hashing function to uniformly distribute remaining data
        - Determine the number of logical shards
        - Decide how to map logical shards to physical
        - Chose sharding key

IMPLEMENTING SHARDS

Now that logical data split is "architected", let's discuss what needs to happen to implement it in practice.

The solutions will obviously depend on:

- The actual storage engine (loosely known as: *database*) being used
- How much downtime we can afford during sharding process

Let's begin.

DECIDE ON THE NUMBER OF PHYSICAL SHARDS:

This number would be obviously dependent on several factors:

- Current data size and load
- Projected MAX data size and load
- Type of commodity hardware that you want to use (definition of *sweet spot* hardware varies significantly between different organizations)
- When are you planning to replace the hardware or re-shard data again?

The driving force here is to implement solution that will hold up for a while (as re-sharding is always painful) …

DATA SHARDING - Step 6: Decide on the number of physical shards

READY, SET, SHARD!

With logical design finalized and number of physical shards chosen, we are finally ready to shard our data *for real*.

I'm going to discuss a few basic approaches how to do it but note that these are just examples and they are certainly not the only ones available. There is nothing magic about the sharding process: it is simply a "data move" and any experienced DBA should be able to come up with plenty of workable solutions.

The examples below cover ORACLE database and they also assume that a very short downtime (a few minutes) is acceptable.

Sharding the entire database

This is probably the simplest approach as long as you are ok with wasting a bit of space.

The idea here is to setup a number of physical standby databases that feed off the monolithic database that you need to shard.

When ready to shard:

1. Turn off writes on primary
2. Make the last sync
3. Break replication
4. Re-make standbys into their own primaries
5. Once new primaries open up, re-connect *new shard aware* applications to these new databases and enable writes.

It is extremely critical to make sure that only applications that are *aware of the new sharding scheme* can connect to the new shards.

Let me re-emphasize it again:

You cannot have any applications that read (or, worse, write) data to/from the wrong shard.

Doing so is a recipe for disaster as wrongly placed data can accumulate quickly and will be extremely difficult to repair.

Thus, it is highly recommended to create an additional verification of incoming data in the database itself (i.e. in the form of triggers). Fortunately, most sharding algorithms can be easily coded in PL/SQL.

Once re-sharding process is completed, and application "goodness" is verified, you can start thinking about removing non-qualifying data.

I.e. if you split your master database in 4, the end result of sharding by physical standbys will be that ~75% of data on each shard will not "belong" (remember, physical standby is a full replica of the original data).

However, this is only a "wasted space" problem; it does not affect data quality as non qualifying data is "unseen" by the applications. Thus, "waste" removal does not need to happen immediately and can be done slowly, in background and take a fairly long time.

Sharding a few tables

If you only have a few tables that you need to split off or re-shard, making a copy of the entire database will probably be overkill.

Instead, you can use any of the methods that ORACLE offers to replicate individual objects, i.e. materialized views, multi master replication, Streams or Golden Gate. You can also come up with your own tools.

The basic idea here is the same: set replicas of master tables elsewhere and when ready to re-shard:

1. Turn writes off
2. Do the last sync
3. Activate writes with new *shard aware* applications.

As in the previous example, your main concern should be to make sure that ALL new applications are aware of the new sharding scheme.

Moving "head" of data

Finally, in some cases, you can shard your data without the need to move any data at all (how cool is that!)

The idea here is to only shard (or re-shard) *new* data, while keeping existing data where it is already.

I.e. in our ORDERS table example, we can adjust mapping layer:

Logical Shard	Physical Shard
(1,2)	1
(3,4)	2
(5,0)	3

To also include the timestamp:

Logical Shard	Time	Physical Shard
(1,2)	2011	1
(3,4)	2011	2
(5,0)	2011	3

This way, re-sharding can be done by a simple *routing change*, such as:

Logical Shard	Time	Physical Shard
(1,2)	2011	1
(3,4)	2011	2
(5,0)	2011	3
(2,0)	2012	4
(1)	2012	1
(5)	2012	3
(3,4)	2012	2

Of course, it assumes that your applications either *never* need to go across shards or can suffer slight performance degradation (and more complex code) for the time interval when cross shard queries are necessary.

Benefits and Downsides of Sharded Databases

So, what can you expect when you shard your data?

Sharding Benefits

"Limitless" Scaling: Sharding allows you to scale your system *horizontally*, adding (albeit, with some difficulty) data partitions as you go. This, *in theory*, allows unlimited data scaling.

"Reasonable" hardware cost: Sharding lets you buy hardware at *"sweet spot"* prices, not forcing you to overpay for hardware if your data grows rapidly.

"Reasonable" load: Sharding reduces the load on your database and allows it to run under *normal* conditions. This is usually underappreciated, but databases, just like the laws of physics, can and do break down with unpredictable consequences when ~~energy~~ load gets too high.

Data is better cached: Because shards are smaller, data is much better cached in memory, which generally helps queries.

Improved availability: Sharding tends to increases data availability (as long as you agree with the new availability definition)

I.e. let's consider a scenario where one monolithic database is broken into 4 shards.

On the one hand, each shard now contains 1/4 of the data, so if any shard goes down, 75% of data is still available. That's a good thing!

On the other hand, we've just quadrupled the chances that something will go wrong with at least one shard. And, that's a bad thing!

I would argue that the "good thing" will win in most cases as having data *mostly* available is better than being completely down. In addition to that, keep in mind that the chance of system failure is probably getting lower as load is reduced.

Better monitoring: A somewhat non-obvious thing is that while shards are not exactly identical, they are "similar" enough so that you can make "apples-to-apples" comparisons and get some interesting data out.

I.e. if the load on one of the shards is consistently higher, you can use other shards as reference on what the load *should* look like (as theoretically, load should be roughly similar ... at least in the long run).

Or, if you are running "the same" SQLs in all shards (you really should), you can cross check performance metrics to see if SQLs are executed efficiently everywhere.

Easier maintenance: This is purely for DBAs: databases are smaller and thus management operations, such as backups, restores or statistics collections are faster. Hey, you might not need *days* to complete full backups after all!

Safer upgrades, patches and parameter changes: Not sure how your system will behave after ORACLE 12.1 upgrade? No need to risk all of it! Start with one shard and go from there.

Sharding Downsides

If sharding had only benefits, people would choose to do it all the time. Since most people don't, it must have some significant downsides. What are they?

Applications are more complex: Applications that work with sharded databases are necessarily more complex as they have to now keep track of a physical location of each data piece and connect to the proper database to work with it. This adds another layer to data access and may affect latencies.

On the other hand, once written, this hashing/bucketing/database connection piece can be abstracted in a library and reused by many programs, so, at least, developing this layer is a one-time job.

More systems: This should be obvious: sharding increases the number of systems to manage. More systems = more power, more rack space, more maintenance and, in general, more chances that something will go wrong somewhere.

It does not necessarily mean that more DBAs are needed to support sharded databases, but automation does become very critical.

Some data operations are either gone or become too difficult:

- Referential constraints across shards - Gone!
- Joins across shards - Gone!
- Cross shard queries or reports - Gone!

I'm probably being too harsh here, but the above operations do become very difficult to the point that doing them, at least, frequently, becomes impractical.

In other words, "shared nothing" architecture does mean that *nothing can be (efficiently) shared.*

DIY everything: This is probably the biggest show stopper. While sharding is probably considered "mainstream" now and there are some third-party tools that help with sharding to some extent, major RDBMS vendors (i.e. ORACLE) still do not support sharding out of the box.

In other words, if you want to shard your data, you are essentially on your own and have to DIY everything.

Is Sharding right for you?

To shard or not to shard … I hope this white paper helped you answer this question better *for your system*.

Sharding can be a wonderful tool that solves your load problems and gives you an (almost) unlimited scalability. It can also be a drag, making your system more complex than it needs to be. As with any tools it all depends *on the circumstances*.

So, what circumstances make a data system *just right* for Sharding?

In my mind there are three and I summarized them as the questions below.

Is your system in danger of overrunning reasonable hardware? If NOT, sharding is clearly overkill.

Is your workload mostly relational? Do you need to use relational databases? There are better options now if your load is "specialized". I.e., for simple key/value lookups, Amazon DynamoDb is probably a better option (yes, I'm biased :))

Are you willing to pay the price? In the features lost. As well as in time and effort to develop the new architecture. Remember, sharding is still mostly DIY experience.

If you answered NO to any of the questions, sharding might not be a best fit for your system *right now*.

But if you answered YES on all three, congratulations … and strap yourself on, as you are in for quite a ride!