Database Sharding

Database Sharding

The Rise of Database Sharding

The concept of Database Sharding has been gaining popularity over the past several years, due to the enormous growth in transaction volume and size of business application databases. This is particularly true for many successful online service providers, Software as a Service (SaaS) companies, and social networking Web sites.

Database Sharding can be simply defined as a “shared-nothing” partitioning scheme for large databases across a number of servers, enabling new levels of database performance and scalability achievable. If you think of broken glass, you can get the concept of sharding – breaking your database down into smaller chunks called “shards” and spreading those across a number of distributed servers.

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

The focus of this paper is on the need for Database Sharding, the options available for database partitioning, and the key considerations for a successful sharding implementation.

What Drives the Need for Database Sharding?

Database Sharding is a highly scalable approach for improving the throughput and overall performance of high-transaction, large database-centric business applications. Since the inception of the relational database, application engineers and architects have required ever-increasing performance and capacity, based on the simple observation that business databases generally grow in size over time. Adding to this general trend is the extreme expansion of business data due to the evolution of the Internet economy, the Information Age, and the prevalence of high-volume electronic commerce.

As any experienced database administrator or application developer knows all too well, it is axiomatic that as the size and transaction volume of the database tier incurs linear growth, response times tend to grow logarithmically. This is shown in the following diagram:

(http://agildata.com/database-sharding)

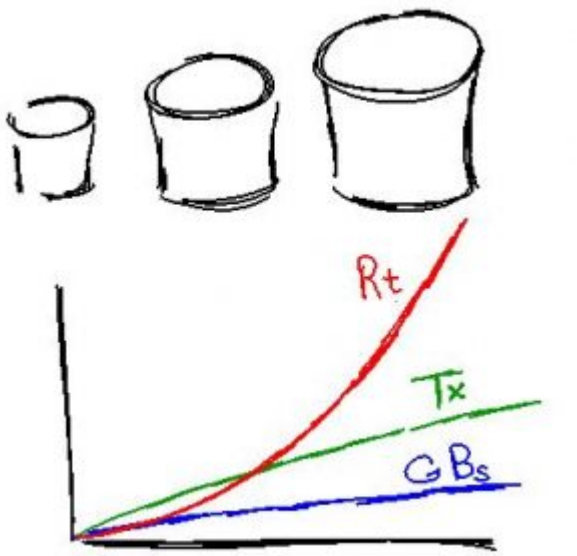


Figure 1. The growth in database transactions and volumes has a large impact on response times.

The reasons for the performance and scalability challenges are inherent to the fundamental design of the database management systems themselves. Databases rely heavily on the primary three components of any computer:

- CPU
- Memory
- Disk

Through benchmark tests that we have performed, we know that each of these elements on a single server can only scale to a given point, and then other measures must be taken. While it is clear that disk I/O is the primary bottleneck, as database management systems have improved they also continue to take greater advantage of CPU and memory. In fact, we have observed that it is the matching of these three factors that determines maximum performance. In other words, you cannot add an unlimited number of CPUs (or processing cores) and see a commensurate increase in performance without also improving the memory capacity and performance of the disk drive subsystem. It is also common to see a diminishing return as resources are added to a single database server. These factors are especially true in mixed-use business transaction systems; systems that perform a high volume of read and write transactions, as well as supporting generalized business reporting tasks.

Therefore, as business applications gain sophistication and continue to grow in demand, architects, developers and database administrators have been presented with a constant challenge of maintaining database performance for mission critical systems. This landscape drives the need for Database Sharding.

Database Partitioning Options

It has long been known that database partitioning is the answer to improving the performance and scalability of relational databases. Many techniques have been evolved, including:

- **Master/Slave:** This is the simplest option used by many organizations, with a single Master server for all write (Create Update or Delete, or CRUD) operations, and one or many additional Slave servers that provide read-only operations. The Master uses standard, near-real-time database replication to each of the Slave servers. The Master/Slave model can speed overall performance to a point, allowing read-intensive processing to be offloaded to the Slave servers, but there are several limitations with this approach:
    - The single Master server for writes is a clear limit to scalability, and can quickly create a bottleneck.
    - The Master/Slave replication mechanism is “near-real-time,” meaning that the Slave servers are not guaranteed to have a current picture of the data that is in the Master. While this is fine

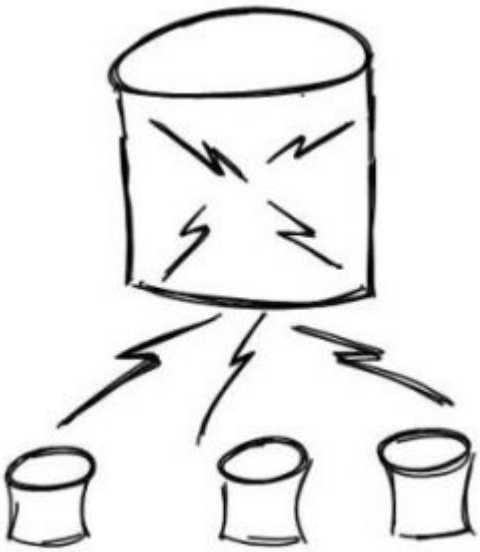
for some applications, if your applications require an up-to-date view, this approach is unacceptable.
    - Many organizations use the Master/Slave approach for high-availability as well, but it suffers from this same limitation given that the Slave servers are not necessarily current with the Master. If a catastrophic failure of the Master server occurs, any transactions that are pending for replication will be lost, a situation that is highly unacceptable for most business transaction applications.
- **Cluster Computing:** Cluster computing utilizes many servers operating in a group, with shared messaging between the nodes of the cluster. Most often this scenario relies on a centralized shared disk facility, typically a Storage Area Network (SAN). Each node in the cluster runs a single instance of the database server, operating in various modes:
    - For high-availability, many nodes in the cluster can be used for reads, but only one for write (CRUD) operations. This can make reads faster, but write transactions do not see any benefit. If a failure of one node occurs, then another node in the cluster takes over, again continuing to operating against the shared disk facility. This approach has limited scalability due to the single bottleneck for CRUD operations. Even the reads will ultimately hit a performance limit as the centralized shared disk facility can only spread the load so much before diminishing returns are experienced. The read limitations are particularly evident when an application requires complex joins or contains non-optimized SQL statements.
    - More advanced clustering techniques rely on real-time memory replication between nodes, keeping the memory image of nodes in the cluster up to date via a real-time messaging system. This allows each node to operate in both read or write mode, but is ultimately limited by the amount of traffic that can be transmitted between nodes (using a typical network or other high-speed communication mechanism). Therefore, as nodes are added, the communication and memory replication overhead increases geometrically, thus hitting severe scalability limits, often with a relatively small number of nodes. This solution also suffers from the same shared disk limitations of a traditional cluster, given that a growing, single large database has increasingly intensive disk I/O.
- **Table Partitioning:** Many database management systems support table partitioning, where data in a single large table can be split across multiple disks for improved disk I/O utilization. The partitioning is typically done horizontally (separating rows by range across disk partitions), but can be vertical in some systems as well (placing different columns on separate partitions). This approach can help reduce the disk I/O bottleneck for a given table, but can often make joins and other operations slower. Further, since the approach relies on a single server instance of the database management system, all other CPU and memory contention limitations apply, further limiting scalability.
- **Federated Tables:** An offshoot of Table Partitioning is the Federated Table approach, where tables can be accessed across multiple servers. This approach is necessarily highly complex to administer, and lacks efficiency as the federated tables must be accessed over the network. This approach may work for some reporting or analytical tasks, but for general read/write transactions it is not a very likely choice.

The common drawback with each of these approaches is the reliance on shared facilities and resources. Whether relying on shared memory, centralized disk, or processor capacity they each suffer with scalability limitations, not to mention many other drawbacks, including complex administration, lack of support for critical business requirements, and high availability limitations.

Database Sharding, The “Shared-Nothing” Approach

Database Sharding provides a method for scalability across independent servers, each with their own CPU, memory and disk. Contrasted with other traditional methods of achieving greater database performance, it does not suffer from many of the typical limitations posed by these other approaches. The concept of a “shared-nothing” database implementation has been under research or discussion for 15+ years, but it appears that the business application market is just now finding the more general need for such capability due to the exponential increase in data volumes over the past several years.

The basic concept of Database Sharding is very straightforward: take a large database, and break it into a number of smaller databases across servers. The concept is illustrated in the following diagram:



(http://www.agildata.com/wp-content/uploads/2016/05/database-sharding-figure-2.jpg)

Figure 2. Database Sharding takes large databases and breaks them down into smaller databases.

The obvious advantage of the shared-nothing Database Sharding approach is improved scalability, growing in a near-linear fashion as more servers are added to the network. However, there are several other advantages of smaller databases, which should not be overlooked when considering a sharding solution:

- *Smaller databases are easier to manage.* Production databases must be fully managed for regular backups, database optimization and other common tasks. With a single large database these routine tasks can be very difficult to accomplish, if only in terms of the time window required for completion. Routine table and index optimizations can stretch to hours or days, in some cases making regular maintenance infeasible. By using the sharding approach, each individual “shard” can be maintained independently, providing a far more manageable scenario, performing such maintenance tasks in parallel.
- *Smaller databases are faster.* The scalability of sharding is apparent, achieved through the distribution of processing across multiple shards and servers in the network. What is less apparent is the fact that each individual shard database will outperform a single large database due to its smaller size. By hosting each shard database on its own server, the ratio between memory and data on disk is greatly improved, thereby reducing disk I/O. This results in less contention for resources, greater join performance, faster index searches, and fewer database locks. Therefore, not only can a

sharded system scale to new levels of capacity, individual transaction performance is benefited as well.

- *Database Sharding can reduce costs.* Most Database Sharding implementations take advantage of lower-cost open source databases, or can even take advantage of “workgroup” versions of commercial databases. Additionally, sharding works well with commodity multi-core server hardware, far less expensive than high-end multi-CPU servers and expensive SANs. The overall reduction in cost due to savings in license fees, software maintenance and hardware investment is substantial, in some cases 70% or more when compared to other solutions.

There is no doubt that Database Sharding is a viable solution for many organizations, supported by the number of large online vendors and SaaS organizations that have implemented the technology (giants such as Amazon, eBay, and of course Google).

Practicalities of Database Sharding

If Database Sharding is highly scalable, less costly, and improves performance, why hasn’t adoption of the technology been more widespread? Is it feasible for your organization?

The reality is that Database Sharding is a very useful technology, but like other approaches, there are many factors to consider that ensure a successful implementation. Further, there are some limitations and Database Sharding will not work well for every type of business application. This chapter discusses these critical considerations and how they can be addressed.

Database Sharding Challenges

Due to the distributed nature of individual databases, a number of key elements must be taken into account:

- *Reliability.* First and foremost, any production business application must be reliable and fault-tolerant, and cannot be subject to frequent outages. The database tier is often the single most critical element in any reliability design, and therefore an implementation of Database Sharding is no exception. In fact, due to the distributed nature of multiple shard databases, the criticality of a well-designed approach is even greater. To ensure a fault-tolerant and reliable approach, the following items are required:
    - Automated backups of individual Database Shards.
    - Database Shard redundancy, ensuring at least 2 “live” copies of each shard are available in the event of an outage or server failure. This requires a high-performance, efficient, and reliable replication mechanism.
    - Cost-effective hardware redundancy, both within and across servers.
    - Automated failover when an outage or server failure occurs.
    - Disaster Recovery site management.
- *Distributed queries.* Many types of queries can be processed far faster using distributed queries, performing parallel processing of interim results on each shard server. This technique can achieve

order-of-magnitude improvements in performance, in many cases 10X or more. To enable distributed queries in a seamless manner for the application, it is important to have a facility that can process a segment of the query on each individual shard, and then consolidate the results into a single result set for the application tier. Common queries that can benefit from distributed processing are:

  - Aggregation of statistics, requiring a broad sweep of data across the entire system. Such an example is the computation of sales by product, which ordinarily requires evaluation of the entire database.
  - Queries that support comprehensive reports, such as listings of all individual customers that purchased a given product in the last day, week or month.

- *Avoidance of cross-shard joins.* In a sharded system, queries or other statements that use inner-joins that span shards are highly inefficient and difficult to perform. In the majority of cases, it has been found that such inner-joins are not actually required by an application, so long as the correct techniques are applied. The primary technique is the replication of Global Tables, the relatively static lookup tables that are common utilized when joining to much larger primary tables. Tables containing values as Status Codes, Countries, Types, and even Products fall into this category. What is required is an automated replication mechanism that ensures values for Global Tables are in synch across all shards, minimizing or eliminating the need for cross-shard joins.
- *Auto-increment key management.* Typical auto-increment functionality provided by database management systems generate a sequential key for each new row inserted into the database. This is fine for a single database application, but when using Database Sharding, keys must be managed across all shards in a coordinated fashion. The requirement here is to provide a seamless, automated method of key generation to the application, one that operates across all shards, ensuring that keys are unique across the entire system.
- *Support for multiple Shard Schemes.* It is important to note that Database Sharding is effective because it offers an application specific technique for massive scalability and performance improvements. In fact it can be said that the degree of effectiveness is directly related to how well the sharding algorithms themselves are tailored to the application problem at hand. What is required is a set of multiple, flexible shard schemes, each designed to address a specific type of application problem. Each scheme has inherent performance and/or application characteristics and advantages when applied to a specific problem domain. In fact, using the wrong shard scheme can actually inhibit performance and the very results you are trying to obtain. It is also not uncommon for a single application to use more than one shard scheme, each applied to a specific portion of the application to achieve optimum results. Here is a list of some common shard schemes:
  - Session-based sharding, where each individual user or process interacts with a specific shard for the duration of the user or process session. This is the simplest technique to implement, and adds virtually zero overhead to overall performance, since the sharding decision is made only once per session. Applications which can benefit from this approach are often customer-centric, where all data for a given customer is contained in a single shard, and that is all the data that the customer requires.

    - Transaction-based sharding determines the shard by examining the first SQL Statement in a given database transaction. This is normally done by evaluating the “shard key” value used in the statement (such as an Order Number), and then directing all other statements in the transaction to the same shard.
    - Statement-based sharding is the most process intensive of all types, evaluating each individual SQL Statement to determine the appropriate shard to direct it to. Again, evaluation of the shard key value is required. This option is often desirable on high-volume, granular transactions, such as recording phone call records.
- Determine the optimum method for sharding the data. This is another area that is highly variable, change from application to application. It is closely tied with the selection of the Database Shard Scheme described above. There are numerous methods for deciding how to shard your data, and its important to understand your transaction rates, table volumes, key distribution, and other characteristics of your application. This data is required to determine the optimum sharding strategy:
    - Shard by a primary key on a table. This is the most straightforward option, and easiest to map to a given application. However, this is only effective if your data is reasonably well distributed. For example, if you elected to shard by Customer ID (and this is a sequential numeric value), and most of your transactions are for new customers, very little if anything will be gained by sharding your database. On the other hand, if you can select a key that does adequately and naturally distribute your transactions, great benefits can be realized.
    - Shard by the modulus of a key value. This option works in a vast number of cases, by applying the modulus function to the key value, and distributing transactions based on the calculated value. In essence you can predetermine any number of shards, and the modulus function effectively distributes across your shards on a “round-robin” basis, creating a very even distribution of new key values.
    - Maintain a master shard index table. This technique involves using a single master table that maps various values to specific shards. It is very flexible, and meets a wide variety of application situations. However, this option often delivers lower performance as it requires an extra lookup for each sharded SQL Statement.

As you can see, there are many things to consider and many capabilities required in order to ensure that a Database Sharding implementation is successful and effective, delivering on its objectives of providing new levels of scalability and performance in a cost-effective manner.

When Database Sharding is Appropriate

Database Sharding is an excellent fit for many types of business applications, those with general purpose database requirements. It can also be used effectively for Data Warehousing applications, and as there are many available products and technologies to accomplish this, we will not focus on this element here.

The general purpose database requirements that are a fit for sharding include:

- High-transaction database applications

- Mixed workload database usage
    - Frequent reads, including complex queries and joins
    - Write-intensive transactions (CRUD statements, including INSERT, UPDATE, DELETE)
    - Contention for common tables and/or rows
- General Business Reporting
    - Typical “repeating segment” report generation
    - Some data analysis (mixed with other workloads)

To determine if Database Sharding is applicable to your specific application or environment, the most important thing to evaluate is how well your database schema lends itself to sharding. In essence, Database Sharding is a method of “horizontal” portioning, meaning that database rows (as opposed to columns) for a single schema table are distributed across multiple shards. To understand the characteristics of how well sharding fits a given situation, here are the important things to determine:

- Identify all transaction-intensive tables in your schema.
- Determine the transaction volume your database is currently handling (or is expected to handle).
- Identify all common SQL statements (SELECT, INSERT, UPDATE, DELETE), and the volumes associated with each.
- Develop an understanding of your “table hierarchy” contained in your schema; in other words the main parent-child relationships.
- Determine the “key distribution” for transactions on high-volume tables, to determine if they are evenly spread or are concentrated in narrow ranges.

With this information, you can rapidly gain an assessment of the value and applicability of sharding to your application. As an example, here is a simple Bookstore schema showing how the data can be sharded:

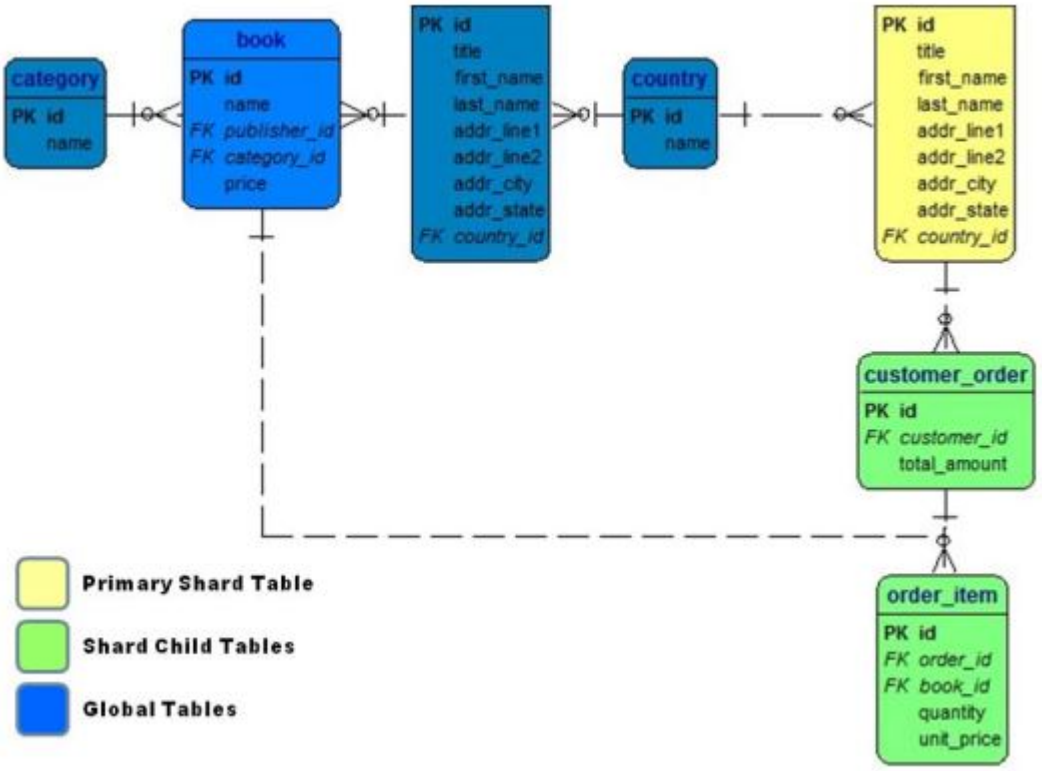
View the BookStore demo here. (http://www.codefutures.com/img/dbs-bookstoreschema.gif)

(http://www.agildata.com/wp-content/uploads/2016/05/database-sharding-figure-3.jpg)

Figure 3. Example Bookstore schema showing how data is sharded.

In the Bookstore example, the Primary Shard Table is the ‘customer’ entity. This is the table that is used to shard the data. The ‘customer’ table is the parent of the shard hierarchy, with the ‘customer_order’ and ‘order_item’ entities as child tables. The data is sharded by the ‘customer.id’ attribute, and all related rows in the child tables associated with a given ‘customer.id’ are sharded as well. The Global Tables are the common lookup tables, which have relatively low activity, and these tables are replicated to all shards to avoid cross-shard joins.

While this example is very basic, it does provide the basic considerations for determining how to shard a given database application. By using this evaluation approach, you can determine if sharding is applicable to your particular environment, and what benefits can be realized by implementing Database Sharding.



Read More about the AgilData's Scalable Cluster for MySQL

Get the Free Database Sharding White Paper

- **First Name***
- **Last Name***
- **Email***
- **Company***
- **Phone**

Submit