Horizontal, vertical, and functional data partitioning

11/04/2018 • 17 minutes to read • Contributors all

In this article

In many large-scale solutions, data is divided into *partitions* that can be managed and accessed separately. Partitioning can improve scalability, reduce contention, and optimize performance. It can also provide a mechanism for dividing data by usage pattern. For example, you can archive older data in cheaper data storage.

However, the partitioning strategy must be chosen carefully to maximize the benefits while minimizing adverse effects.

> **Note**
>
> In this article, the term *partitioning* means the process of physically dividing data into separate data stores. It is not the same as SQL Server table partitioning.

Why partition data?

- **Improve scalability**. When you scale up a single database system, it will eventually reach a physical hardware limit. If you divide data across multiple partitions, each hosted on a separate server, you can scale out the system almost indefinitely.
- **Improve performance**. Data access operations on each partition take place over a smaller volume of data. Correctly done, partitioning can make your system more efficient. Operations that affect more than one partition can run in parallel.
- **Improve security**. In some cases, you can separate sensitive and nonsensitive data into different partitions and apply different security controls to the sensitive data.
- **Provide operational flexibility**. Partitioning offers many opportunities for fine-tuning operations, maximizing administrative efficiency, and minimizing cost. For example, you can define different strategies for management, monitoring, backup and restore, and other administrative tasks based on the importance of the data in each partition.
- **Match the data store to the pattern of use**. Partitioning allows each partition to be deployed on a different type of data store, based on cost and the built-in features that data store offers. For example, large binary data can be stored in blob storage, while more structured data can be held in a document database. See Choose the right data store.

- **Improve availability**. Separating data across multiple servers avoids a single point of failure. If one instance fails, only the data in that partition is unavailable. Operations on other partitions can continue. For managed PaaS data stores, this consideration is less relevant, because these services are designed with built-in redundancy.

Designing partitions

There are three typical strategies for partitioning data:

- **Horizontal partitioning** (often called *sharding*). In this strategy, each partition is a separate data store, but all partitions have the same schema. Each partition is known as a *shard* and holds a specific subset of the data, such as all the orders for a specific set of customers.

- **Vertical partitioning**. In this strategy, each partition holds a subset of the fields for items in the data store. The fields are divided according to their pattern of use. For example, frequently accessed fields might be placed in one vertical partition and less frequently accessed fields in another.

- **Functional partitioning**. In this strategy, data is aggregated according to how it is used by each bounded context in the system. For example, an e-commerce system might store invoice data in one partition and product inventory data in another.

These strategies can be combined, and we recommend that you consider them all when you design a partitioning scheme. For example, you might divide data into shards and then use vertical partitioning to further subdivide the data in each shard.

Horizontal partitioning (sharding)

Figure 1 shows horizontal partitioning or sharding. In this example, product inventory data is divided into shards based on the product key. Each shard holds the data for a contiguous range of shard keys (A-G and H-Z), organized alphabetically. Sharding spreads the load over more computers, which reduces contention and improves performance.

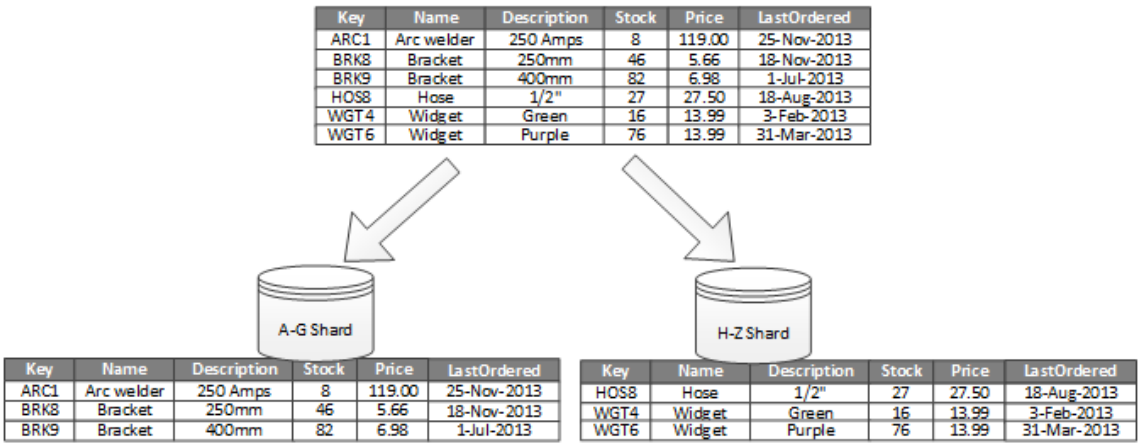

Figure 1. Horizontally partitioning (sharding) data based on a partition key.

The most important factor is the choice of a sharding key. It can be difficult to change the key after the system is in operation. The key must ensure that data is partitioned to spread the workload as evenly as possible across the shards.

The shards don't have to be the same size. It's more important to balance the number of requests. Some shards might be very large, but each item has a low number of access operations. Other shards might be smaller, but each item is accessed much more frequently. It's also important to ensure that a single shard does not exceed the scale limits (in terms of capacity and processing resources) of the data store.

Avoid creating "hot" partitions that can affect performance and availability. For example, using the first letter of a customer's name causes an unbalanced distribution, because some letters are more common. Instead, use a hash of a customer identifier to distribute data more evenly across partitions.

Choose a sharding key that minimizes any future requirements to split large shards, coalesce small shards into larger partitions, or change the schema. These operations can be very time consuming, and might require taking one or more shards offline while they are performed.

If shards are replicated, it might be possible to keep some of the replicas online while others are split, merged, or reconfigured. However, the system might need to limit the operations that can be performed during the reconfiguration. For example, the data in the replicas might be marked as read-only to prevent data inconsistences.

For more information about horizontal partitioning, see [Sharding pattern].

Vertical partitioning

The most common use for vertical partitioning is to reduce the I/O and performance costs associated with fetching items that are frequently accessed. Figure 2 shows an example of vertical partitioning. In this example, different properties of an item are stored in different partitions. One partition holds data that is accessed more frequently, including product name, description, and price. Another partition holds inventory data: the stock count and last-ordered date.

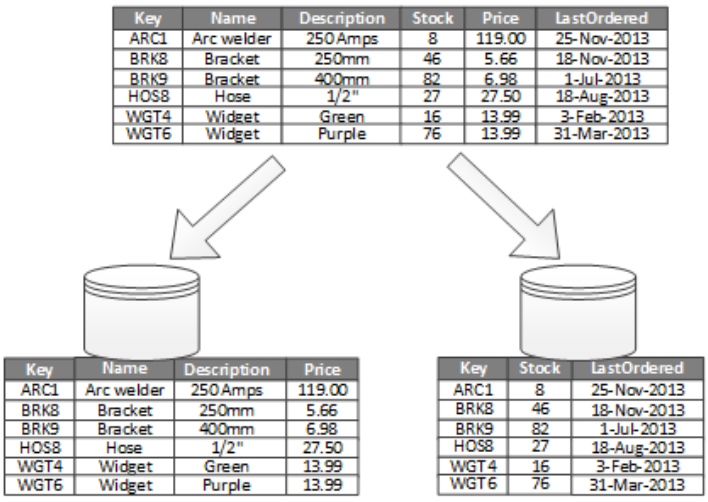

Figure 2. Vertically partitioning data by its pattern of use.

In this example, the application regularly queries the product name, description, and price when displaying the product details to customers. Stock count and last- ordered date are held in a separate partition because these two items are commonly used together.

Other advantages of vertical partitioning:

- Relatively slow-moving data (product name, description, and price) can be separated from the more dynamic data (stock level and last ordered date). Slow moving data is a good candidate for an application to cache in memory.
- Sensitive data can be stored in a separate partition with additional security controls.
- Vertical partitioning can reduce the amount of concurrent access that's needed.

Vertical partitioning operates at the entity level within a data store, partially normalizing an entity to break it down from a *wide* item to a set of *narrow* items. It is ideally suited for column-oriented data stores such as HBase and Cassandra. If the data in a collection of columns is unlikely to change, you can also consider using column stores in SQL Server.

Functional partitioning

When it's possible to identify a bounded context for each distinct business area in an application, functional partitioning is a way to improve isolation and data access performance. Another common use for functional

partitioning is to separate read-write data from read-only data. Figure 3 shows an overview of functional partitioning where inventory data is separated from customer data.

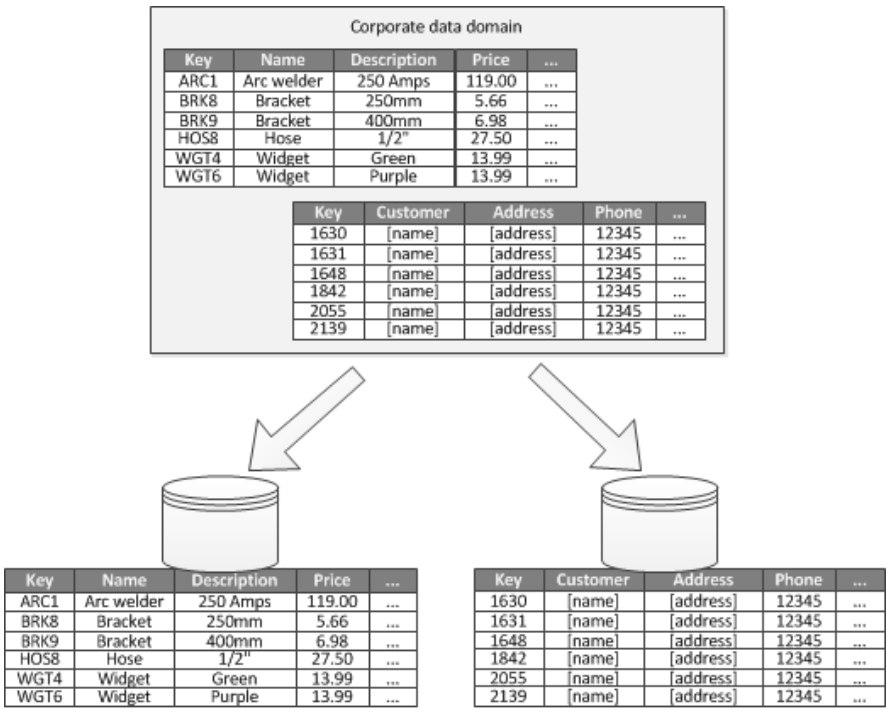

Figure 3. Functionally partitioning data by bounded context or subdomain.

This partitioning strategy can help reduce data access contention across different parts of a system.

Designing partitions for scalability

It's vital to consider size and workload for each partition and balance them so that data is distributed to achieve maximum scalability. However, you must also partition the data so that it does not exceed the scaling limits of a single partition store.

Follow these steps when designing partitions for scalability:

1. Analyze the application to understand the data access patterns, such as the size of the result set returned by each query, the frequency of access, the inherent latency, and the server-side compute processing requirements. In many cases, a few major entities will demand most of the processing resources.
2. Use this analysis to determine the current and future scalability targets, such as data size and workload. Then distribute the data across the partitions to meet the scalability target. For horizontal partitioning, choosing the right shard key is important to make sure distribution is even. For more information, see the [Sharding pattern].
3. Make sure each partition has enough resources to handle the scalability requirements, in terms of data size and throughput. Depending on the data store, there might be a limit on the amount of storage space, processing power, or network bandwidth per partition. If the requirements are likely to exceed these limits, you may need to refine your partitioning strategy or split data out further, possibly combining two or more strategies.
4. Monitor the system to verify that data is distributed as expected and that the partitions can handle the load. Actual usage does not always match what an analysis predicts. If so, it might be possible to rebalance the partitions, or else redesign some parts of the system to gain the required balance.

Some cloud environments allocate resources in terms of infrastructure boundaries. Ensure that the limits of your selected boundary provide enough room for any anticipated growth in the volume of data, in terms of data storage, processing power, and bandwidth.

For example, if you use Azure table storage, there is a limit to the volume of requests that can be handled by a single partition in a particular period of time. (See Azure storage scalability and performance targets.) A busy shard might require more resources than a single partition can handle. If so, the shard might need to be repartitioned to spread the

load. If the total size or throughput of these tables exceeds the capacity of a storage account, you might need to create additional storage accounts and spread the tables across these accounts.

Designing partitions for query performance

Query performance can often be boosted by using smaller data sets and by running parallel queries. Each partition should contain a small proportion of the entire data set. This reduction in volume can improve the performance of queries. However, partitioning is not an alternative for designing and configuring a database appropriately. For example, make sure that you have the necessary indexes in place.

Follow these steps when designing partitions for query performance:

1. Examine the application requirements and performance:

    - Use business requirements to determine the critical queries that must always perform quickly.
    - Monitor the system to identify any queries that perform slowly.
    - Find which queries are performed most frequently. Even if a single query has a minimal cost, the cumulative resource consumption could be significant.

2. Partition the data that is causing slow performance:

    - Limit the size of each partition so that the query response time is within target.
    - If you use horizontal partitioning, design the shard key so that the application can easily select the right partition. This prevents the query from having to scan through every partition.
    - Consider the location of a partition. If possible, try to keep data in partitions that are geographically close to the applications and users that access it.

3. If an entity has throughput and query performance requirements, use functional partitioning based on that entity. If this still doesn't satisfy the requirements, apply horizontal partitioning as well. In most cases, a single partitioning strategy will suffice, but in some cases it is more efficient to combine both strategies.

4. Consider running queries in parallel across partitions to improve performance.

Designing partitions for availability

Partitioning data can improve the availability of applications by ensuring that the entire dataset does not constitute a single point of failure and that individual subsets of the dataset can be managed independently.

Consider the following factors that affect availability:

How critical the data is to business operations. Identify which data is critical business information, such as transactions, and which data is less critical operational data, such as log files.

- Consider storing critical data in highly available partitions with an appropriate backup plan.

- Establish separate management and monitoring procedures for the different datasets.

- Place data that has the same level of criticality in the same partition so that it can be backed up together at an appropriate frequency. For example, partitions that hold transaction data might need to be backed up more frequently than partitions that hold logging or trace information.

How individual partitions can be managed. Designing partitions to support independent management and maintenance provides several advantages. For example:

- If a partition fails, it can be recovered independently without applications that access data in other partitions.

- Partitioning data by geographical area allows scheduled maintenance tasks to occur at off-peak hours for each location. Ensure that partitions are not too large to prevent any planned maintenance from being completed during this period.

Whether to replicate critical data across partitions. This strategy can improve availability and performance, but can also introduce consistency issues. It takes time to synchronize changes with every replica. During this period, different partitions will contain different data values.

Application design considerations

Partitioning adds complexity to the design and development of your system. Consider partitioning as a fundamental part of system design even if the system initially only contains a single partition. If you address partitioning as an afterthought, it will be more challenging because you already have a live system to maintain:

- Data access logic will need to be modified.
- Large quantities of existing data may need to be migrated, to distribute it across partitions.
- Users expect to be able to continue using the system during the migration.

In some cases, partitioning is not considered important because the initial dataset is small and can be easily handled by a single server. This might be true for some workloads, but many commercial systems need to expand as the number of users increases.

Moreover, it's not only large data stores that benefit from partitioning. For example, a small data store might be heavily accessed by hundreds of concurrent clients. Partitioning the data in this situation can help to reduce contention and improve throughput.

Consider the following points when you design a data partitioning scheme:

Minimize cross-partition data access operations. Where possible, keep data for the most common database operations together in each partition to minimize cross-partition data access operations. Querying across partitions can be more time-consuming than querying within a single partition, but optimizing partitions for one set of queries might adversely affect other sets of queries. If you must query across partitions, minimize query time by running parallel queries and aggregating the results within the application. (This approach might not be possible in some cases, such as when the result from one query is used in the next query.)

Consider replicating static reference data. If queries use relatively static reference data, such as postal code tables or product lists, consider replicating this data in all of the partitions to reduce separate lookup operations in different partitions. This approach can also reduce the likelihood of the reference data becoming a "hot" dataset, with heavy traffic from across the entire system. However, there is an additional cost associated with synchronizing any changes to the reference data.

Minimize cross-partition joins. Where possible, minimize requirements for referential integrity across vertical and functional partitions. In these schemes, the application is responsible for maintaining referential integrity across partitions. Queries that join data across multiple partitions are inefficient because the application typically needs to perform consecutive queries based on a key and then a foreign key. Instead, consider replicating or de-normalizing the relevant data. If cross-partition joins are necessary, run parallel queries over the partitions and join the data within the application.

Embrace eventual consistency. Evaluate whether strong consistency is actually a requirement. A common approach in distributed systems is to implement eventual consistency. The data in each partition is updated separately, and the application logic ensures that the updates are all completed successfully. It also handles the inconsistencies that can arise from querying data while an eventually consistent operation is running.

Consider how queries locate the correct partition. If a query must scan all partitions to locate the required data, there is a significant impact on performance, even when multiple parallel queries are running. With vertical and functional partitioning, queries can naturally specify the partition. Horizontal partitioning, on the other hand, can make locating

an item difficult, because every shard has the same schema. A typical solution to maintain a map that is used to look up the shard location for specific items. This map can be implemented in the sharding logic of the application, or maintained by the data store if it supports transparent sharding.

Consider periodically rebalancing shards. With horizontal partitioning, rebalancing shards can help distribute the data evenly by size and by workload to minimize hotspots, maximize query performance, and work around physical storage limitations. However, this is a complex task that often requires the use of a custom tool or process.

Replicate partitions. If you replicate each partition, it provides additional protection against failure. If a single replica fails, queries can be directed toward a working copy.

If you reach the physical limits of a partitioning strategy, you might need to extend the scalability to a different level. For example, if partitioning is at the database level, you might need to locate or replicate partitions in multiple databases. If partitioning is already at the database level, and physical limitations are an issue, it might mean that you need to locate or replicate partitions in multiple hosting accounts.

Avoid transactions that access data in multiple partitions. Some data stores implement transactional consistency and integrity for operations that modify data, but only when the data is located in a single partition. If you need transactional support across multiple partitions, you will probably need to implement this as part of your application logic because most partitioning systems do not provide native support.

All data stores require some operational management and monitoring activity. The tasks can range from loading data, backing up and restoring data, reorganizing data, and ensuring that the system is performing correctly and efficiently.

Consider the following factors that affect operational management:

- **How to implement appropriate management and operational tasks when the data is partitioned**. These tasks might include backup and restore, archiving data, monitoring the system, and other administrative tasks. For example, maintaining logical consistency during backup and restore operations can be a challenge.
- **How to load the data into multiple partitions and add new data that's arriving from other sources**. Some tools and utilities might not support sharded data operations such as loading data into the correct partition.
- **How to archive and delete the data on a regular basis**. To prevent the excessive growth of partitions, you need to archive and delete data on a regular basis (such as monthly). It might be necessary to transform the data to match a different archive schema.
- **How to locate data integrity issues**. Consider running a periodic process to locate any data integrity issues, such as data in one partition that references missing information in another. The process can either attempt to fix these issues automatically or generate a report for manual review.

Rebalancing partitions

As a system matures, you might have to adjust the partitioning scheme. For example, individual partitions might start getting a disproportionate volume of traffic and become hot, leading to excessive contention. Or you might have underestimated the volume of data in some partitions, causing some partitions to approach capacity limits.

Some data stores, such as Cosmos DB, can automatically rebalance partitions. In other cases, rebalancing is an administrative task that consists of two stages:

1. Determine a new partitioning strategy.
    - Which partitions need to be split (or possibly combined)?
    - What is the new partition key?
2. Migrate data from the old partitioning scheme to the new set of partitions.

Depending on the data store, you might be able to migrate data between partitions while they are in use. This is called *online migration*. If that's not possible, you might need to make partitions unavailable while the data is relocated (*offline migration*).

Offline migration

Offline migration is typically simpler because it reduces the chances of contention occurring. Conceptually, offline migration works as follows:

1. Mark the partition offline.
2. Split-merge and move the data to the new partitions.
3. Verify the data.
4. Bring the new partitions online.
5. Remove the old partition.

Optionally, you can mark a partition as read-only in step 1, so that applications can still read the data while it is being moved.

Online migration

Online migration is more complex to perform but less disruptive. The process is similar to offline migration, except the original partition is not marked offline. Depending on the granularity of the migration process (for example, item by item versus shard by shard), the data access code in the client applications might have to handle reading and writing data that's held in two locations, the original partition and the new partition.

Related patterns

The following design patterns might be relevant to your scenario:

- The sharding pattern describes some common strategies for sharding data.
- The index table pattern shows how to create secondary indexes over data. An application can quickly retrieve data with this approach, by using queries that do not reference the primary key of a collection.
- The materialized view pattern describes how to generate prepopulated views that summarize data to support fast query operations. This approach can be useful in a partitioned data store if the partitions that contain the data being summarized are distributed across multiple sites.

Next steps

- Learn about partitioning strategies for specific Azure services. See Data partitioning strategies