Data Partitioning Guidance

08/26/2015 • 18 minutes to read

In this article

In many large-scale solutions, data is divided into separate partitions that can be managed and accessed separately. The partitioning strategy must be chosen carefully to maximize the benefits while minimizing adverse effects. Partitioning can help to improve scalability, reduce contention, and optimize performance.

Why Partition Data?

Most cloud applications and services store and retrieve data as part of their operations. The design of the data stores that an application uses can have a significant bearing on the performance, throughput, and scalability of a system. One technique that is commonly applied in large-scale systems is to divide the data into separate partitions.

ⓘ Note

The term *partitioning* used in this guidance refers to the process of physically dividing data into separate data stores. This is not the same as SQL Server Table Partitioning, which is a different concept.

Partitioning data can offer a number of benefits. For example, it can be applied in order to:

- **Improve scalability**. Scaling up a single database system will eventually reach a physical hardware limit. Dividing data across multiple partitions, each of which is hosted on a separate server, allows the system to scale out almost indefinitely.
- **Improve performance**. Data access operations on each partition take place over a smaller volume of data. Provided that the data is partitioned in a suitable way, this is much more efficient. Operations that affect more than one partition can execute

in parallel. Each partition can be located near the application that uses it to minimize network latency.

- **Improve availability**. Separating data across multiple servers avoids a single point of failure. If a server fails, or is undergoing planned maintenance, only the data in that partition is unavailable. Operations on other partitions can continue. Increasing the number of partitions reduces the relative impact of a single server failure by reducing the percentage of the data that will be unavailable. Replicating each partition can further reduce the chance of a single partition failure affecting operations. It also enables the separation of critical data that must be continually and highly available from low value data (such as log files) that has lower availability requirements.
- **Improve security**. Depending on the nature of the data and how it is partitioned, it may be possible to separate sensitive and non-sensitive data into different partitions, and therefore different servers or data stores. Security can then be specifically optimized for the sensitive data.
- **Provide operational flexibility**. Partitioning offers many opportunities for fine tuning operations, maximizing administrative efficiency, and minimizing cost. Some examples are defining 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 could be stored in a blob data store, while more structured data could be held in a document database. For more information see Building a Polyglot Solution in the patterns & practices guide Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence on MSDN.

Designing Partitions

Data can be partitioned in different ways: horizontally, vertically, or functionally. The strategy you choose depends on the reason for partitioning the data, and the requirements of the applications and services that will use the data.

ⓘ **Note**

The partitioning schemes described in this guidance are explained in a way that is independent of the underlying data storage technology. They can be applied to many types of data stores, including relational and NoSQL databases.

Partitioning Strategies

The three typical strategies for partitioning data are:

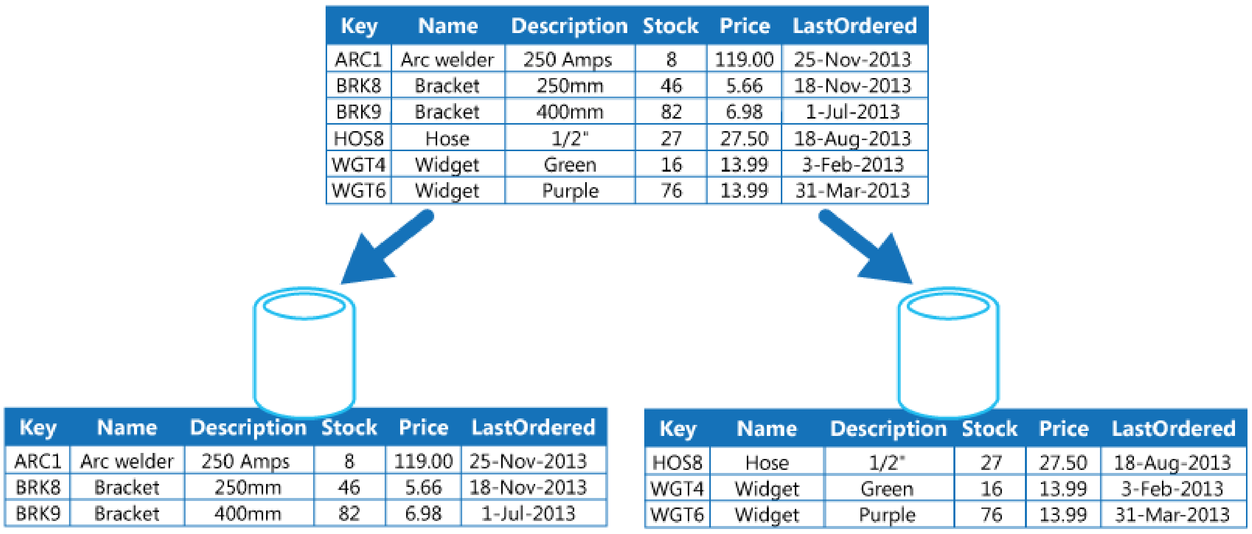
- **Horizontal partitioning** (often called *sharding*). In this strategy each partition is a data store in its own right, but all partitions have the same schema. Each partition is known as a *shard* and holds a specific subset of the data, such as all the orders for a specific set of customers in an ecommerce application.
- **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, such as placing the frequently accessed fields in one vertical partition and the 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 ecommerce system that implements separate business functions for invoicing and managing product inventory might store invoice data in one partition and product inventory data in another.

It's important to note that the three strategies described here can be combined. They are not mutually exclusive and you should 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. Similarly, the data in a functional partition may be split into shards (which may also be vertically partitioned).

However, the differing requirements of each strategy can raise a number of conflicting issues that you must evaluate and balance when designing a partitioning scheme that meets the overall data processing performance targets for your system. The following sections explore each of the strategies in more detail.

Horizontal Partitioning (Sharding)

Figure 1 shows an overview of 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, organized alphabetically).



Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013

Key	Name	Description	Stock	Price	LastOrdered
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Figure 1 - Horizontal partitioning (sharding) divides the data based on a partition key

Sharding enables you to spread the load over more computers; reducing contention, and improving performance. You can scale the system out by adding further shards running on additional servers.

The most important factor when implementing this partitioning strategy is the choice of sharding key. It can be difficult to change the key after the system is in operation. The key must ensure that data is partitioned so that the workload is as even as possible across the shards—some shards may be very large but each item is the subject of a low number of access operations, while other shards may be smaller but each item is accessed much more frequently. It is also important to ensure that a single shard does not exceed the scale limits of the data store being used to host that shard.

The sharding scheme should also avoid creating hotspots (or hot partitions) that may affect performance and availability. For example, using a hash of a customer identifier instead of the first letter of a customer's name will prevent the unbalanced distribution that would result from common and less common initial letters. This is a typical technique that helps to distribute the data more evenly across partitions.

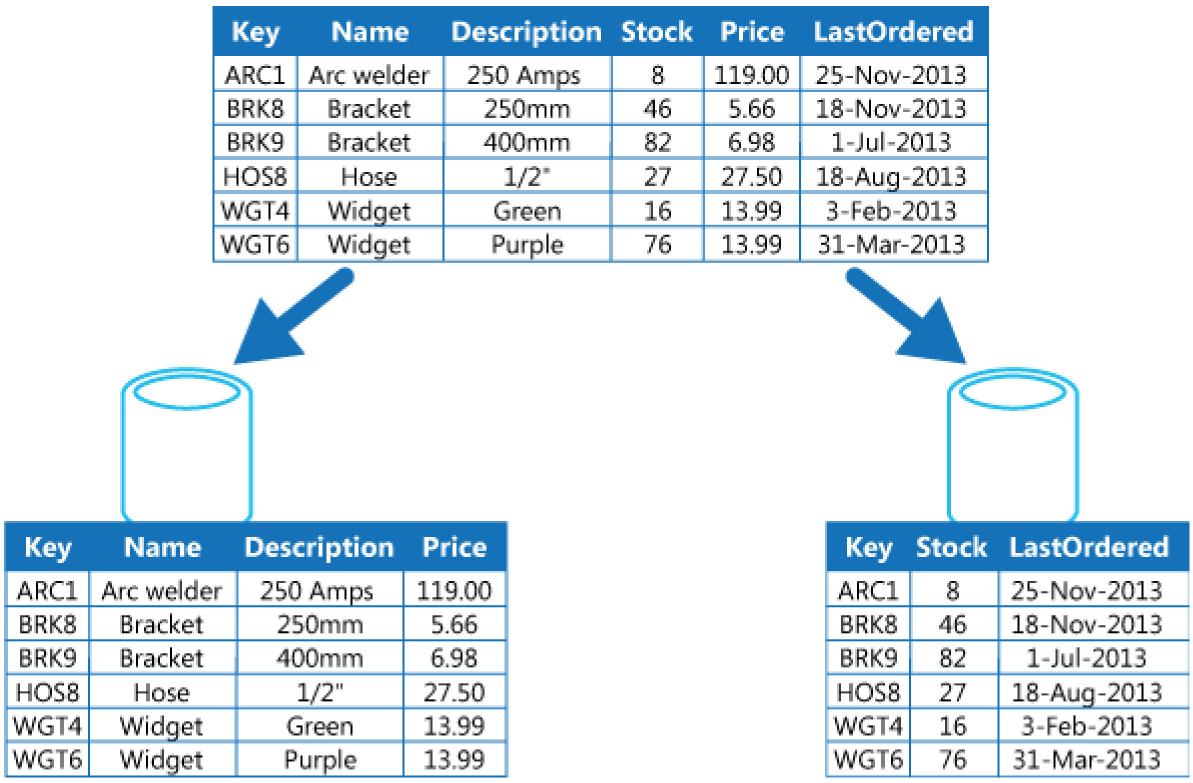
The sharding key you choose should minimize any future requirements to split large shards into smaller pieces, coalesce small shards into larger partitions, or change the schema that describes the data stored in a set of partitions. These operations can be very time consuming, and may require taking one or more shards offline while they are performed. If shards are replicated, it may be possible to keep some of the replicas online while others are split, merged, or reconfigured, but the system may need to limit the operations that can be performed on the data in these shards while the reconfiguration is taking place. For example, the data in the replicas could be marked as read-only to limit the scope of any inconsistences that could otherwise occur while shards are being restructured.

ⓘ Note

For more detailed information and guidance about many of these considerations, and good practice techniques for designing data stores that implement horizontal partitioning, see the Sharding Pattern.

Vertical Partitioning

The most common use for vertical partitioning is to reduce the size of the items that are accessed most frequently in the queries performed by the application. Figure 2 shows an overview of an example of vertical partitioning, where different properties for each data item are held in different partitions.



Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Key	Name	Description	Price
ARC1	Arc welder	250 Amps	119.00
BRK8	Bracket	250mm	5.66
BRK9	Bracket	400mm	6.98
HOS8	Hose	1/2"	27.50
WGT4	Widget	Green	13.99
WGT6	Widget	Purple	13.99

Key	Stock	LastOrdered
ARC1	8	25-Nov-2013
BRK8	46	18-Nov-2013
BRK9	82	1-Jul-2013
HOS8	27	18-Aug-2013
WGT4	16	3-Feb-2013
WGT6	76	31-Mar-2013

Figure 2 - Vertical partitioning organizes data by its pattern of use

In this example, the application regularly queries the product name, description, and price together when displaying the details of products to customers. The stock level and date when the product was last ordered from the manufacturer are held in a separate partition because these two items are commonly used together. This partitioning scheme has the added advantage that the relatively slow-moving data (product name, description, and price) is separated from the more dynamic data (stock level and last ordered date). An application may find it beneficial to cache the slow-moving data in memory.

Another typical scenario for this partitioning strategy is to maximize the security of sensitive data. For example, by storing credit card numbers and the corresponding card security verification numbers in separate partitions.

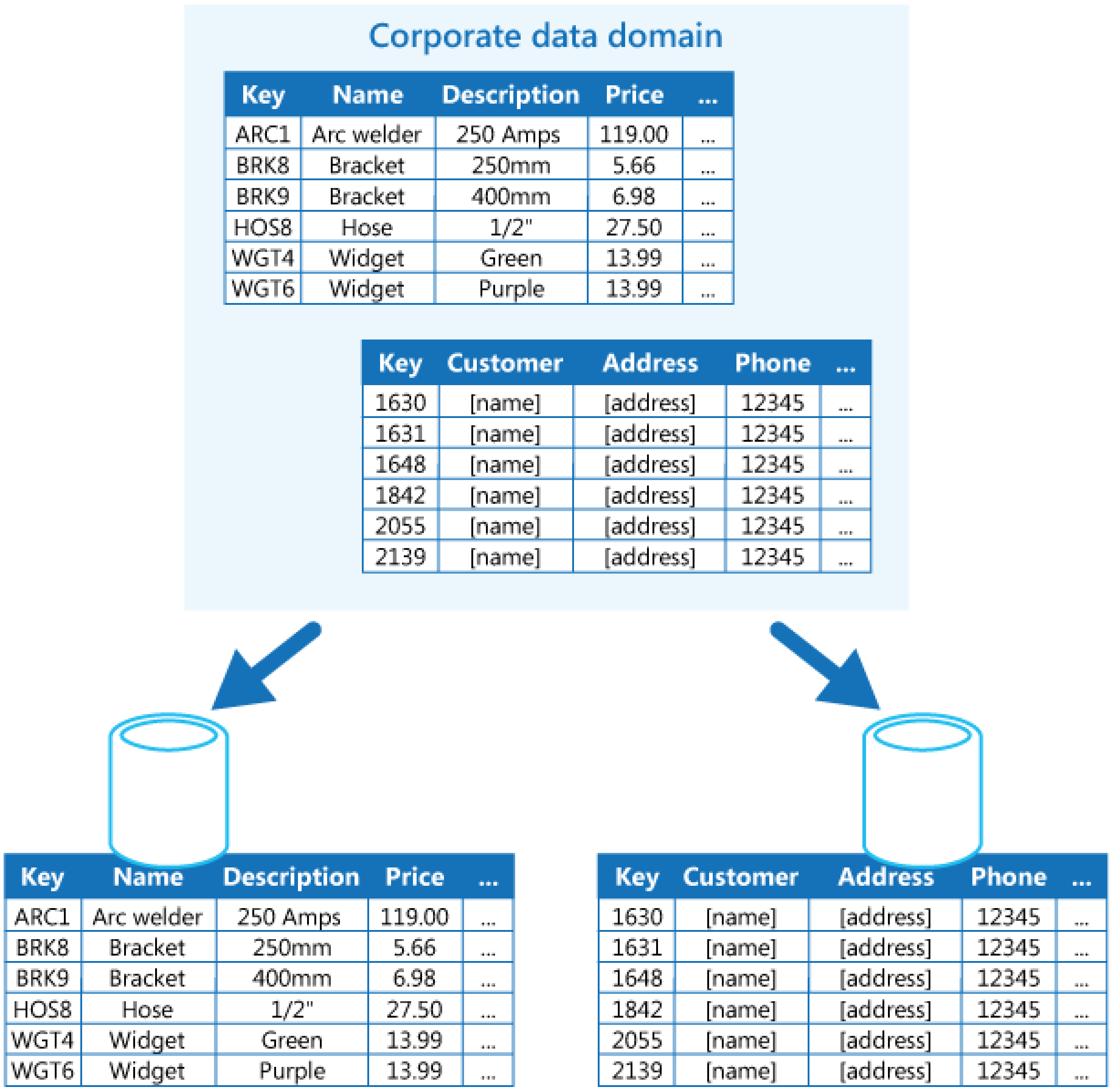
Vertical partitioning can also reduce the amount of concurrent access required to the data.

Note

Vertical partitioning operates at the entity level within a data store, partially normalizing an entity that comprises multiple fields into multiple entities with fewer fields.

Functional Partitioning

For systems where it is possible to identify a bounded context for each distinct business area or service in the application, functional partitioning provides a technique for improving isolation and data access performance. Figure 3 shows an overview of functional partitioning where inventory data is separated from customer data.



Key	Name	Description	Price	...
ARC1	Arc welder	250 Amps	119.00	...
BRK8	Bracket	250mm	5.66	...
BRK9	Bracket	400mm	6.98	...
HOS8	Hose	1/2"	27.50	...
WGT4	Widget	Green	13.99	...
WGT6	Widget	Purple	13.99	...

Key	Customer	Address	Phone	...
1630	[name]	[address]	12345	...
1631	[name]	[address]	12345	...
1648	[name]	[address]	12345	...
1842	[name]	[address]	12345	...
2055	[name]	[address]	12345	...
2139	[name]	[address]	12345	...

Key	Name	Description	Price	...
ARC1	Arc welder	250 Amps	119.00	...
BRK8	Bracket	250mm	5.66	...
BRK9	Bracket	400mm	6.98	...
HOS8	Hose	1/2"	27.50	...
WGT4	Widget	Green	13.99	...
WGT6	Widget	Purple	13.99	...

Key	Customer	Address	Phone	...
1630	[name]	[address]	12345	...
1631	[name]	[address]	12345	...
1648	[name]	[address]	12345	...
1842	[name]	[address]	12345	...
2055	[name]	[address]	12345	...
2139	[name]	[address]	12345	...

Figure 3 - Functional partitioning separates data by bounded context or subdomain

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

Designing Partitions for Scalability

It is 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 vertical scaling limits of a single partition store.

Follow these steps when designing the partitions for scalability:

1. Analyze the application to understand the data access patterns, such as size of each query, the frequency of access, the inherent latency, and the compute processing requirements such as stored procedures. In many cases, a few major entities will demand most of the processing resources.

2. Based on the analysis, determine the current and future scalability targets such as data size and workload, and distribute the data across the partitions to meet the scalability target. In the horizontal partitioning strategy, choosing the appropriate shard key is important to make sure distribution is even. For more information see the Sharding pattern.
3. Make sure that the resources available to each partition are sufficient to handle the scalability requirements in terms of data size and throughput. For example, the node hosting a partition might impose a hard limit on the amount of storage space, processing power, or network bandwidth that it provides. If the data storage and processing requirements are likely to exceed these limits it may be necessary to refine your partitioning strategy or split data out further. For example, one scalability approach might be to separate logging data from the core application features by using separate data stores to prevent the total data storage requirements exceeding the scaling limit of the node. If the total number of data stores exceeds the node limit, it may be necessary to use separate storage nodes.
4. Monitor the system under use to verify that the data is distributed as expected and that the partitions can handle the load imposed on them. It could be possible that the usage does not match that anticipated by the analysis, and it may be necessary to redesign some parts of the system to gain the balance that is required.

Note that some cloud environments allocate resources in terms of infrastructure boundaries, and you should 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 Microsoft Azure table storage, a busy shard might require more resources than are available to a single table to handle requests (there is a limit to the volume of requests that can be handled by a single table in a given period of time). In this case, the shard may need to be split across multiple tables to spread the load. If the total size of these tables exceeds capacity of a storage account, it may be necessary to create additional storage accounts and spread the tables across these accounts. If the number of storage accounts exceeds the number of accounts that are available to a subscription, then it may be necessary to use multiple subscriptions.

Designing Partitions for Query Performance

Query performance can typically be boosted by using smaller data sets and parallel query execution. Each partition should contain a small proportion of the entire data set, and 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 if you are using a relational database.

Follow these steps when designing the partitions for query performance:

1. Analyze the application to identify:

   - Queries that perform slowly.
   - Critical queries that must always perform quickly.

2. Partition the data that is causing slow performance. Ensure that you:

   - Limit the size of each partition so that the query response time is within target.
   - Design the shard key in a way that the application can easily find the partition if you are implementing horizontal partitioning. This prevents the query needing to scan through every partition.
   - Consider the location of a partition on the performance of queries. 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 is still not able to 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 using parallel queries across partitions to improve the 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. When designing and implementing partitions, consider the following factors that affect availability:

- 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 affecting instances of applications that access data in other partitions.
  - Partitioning data by geographical area may allow scheduled maintenance tasks to occur at off-peak hours for each location. Ensure that partitions are not too big to prevent any planned maintenance from being completed during this period.

- How critical the data is to business operations. Some data may comprise critical business information such as invoice details or bank transactions. Other data might simply be less critical operational data, such as log files, performance traces, and so on. After identifying each type of data, consider:
    - Storing critical data in highly-available partitions with an appropriate back up plan.
    - Establishing separate management and monitoring mechanisms or procedures for the different criticalities of each dataset. Place data that has the same level of criticality in the same partition so that it can backed up together at an appropriate frequency. For example, partitions holding data for bank transactions may need to be backed up more frequently than partitions holding logging or trace information.
- Whether to replicate critical data across partitions. This strategy can improve availability and performance, although it can also introduce consistency issues. It takes time for changes made to data in a partition to be synchronized with every replica, and during this period different partitions will contain different data values.

Issues and Considerations

The considerations for designing data partitioning are:

- 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 only within a single partition, but optimizing partitions for one set of queries might adversely affect other sets of queries. To minimize the query time across partitions where this cannot be avoided, execute parallel queries over the partitions and aggregate the results within the application. However, this approach may not be possible in some cases, such as when it is necessary to obtain a result from one query and use this in the next query.
- If queries make use of relatively static reference data, such as postal code tables or product lists, consider replicating this data in all of the partitions to reduce the requirement for a separate lookup operation in a different partition.
- Where possible, minimize requirements for referential integrity across vertical and functional partitions. In these schemes, the application itself is responsible for maintaining referential integrity across partitions when data is updated and consumed. Queries that must join data across multiple partitions run more slowly than queries that join data only within the same partition because the application will typically need to perform consecutive queries based on a key and then on a foreign key. Instead, consider replicating or de-normalizing the relevant data. To

minimize the query time where cross-partition joins are necessary, execute parallel queries over the partitions and join the data within the application.

- Consider the effect that the partitioning scheme might have on the data consistency across partitions. You should evaluate whether strong consistency is actually a requirement. Instead, a common approach in the cloud is to implement eventual consistency. The data in each partition is updated separately, and the application logic can take responsibility for ensuring that the updates all complete successfully—as well as handling the inconsistencies that can arise from querying data while an eventually consistent operation is running. For more information about implementing eventual consistency, see the Data Consistency Primer.
- Consider how queries will locate the correct partition. If a query must scan all partitions to locate the required data there will be a significant impact on performance, even when using multiple parallel queries. Queries used with the vertical and functional partitioning strategies can naturally specify the partitions. However, when using horizontal partitioning (sharding), locating an item can be difficult because every shard has the same schema. A typical solution for sharding is to maintain a map that can be used to look up the shard location for specific items of data. This map may be implemented in the sharding logic of the application, or maintained by the data store if it supports transparent sharding.
- When using a horizontal partitioning strategy, consider periodically rebalancing the shards to 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.
- Replicating each partition provides additional protection against failure. If a single replica fails, queries can be directed towards a working copy.
- If you reach the physical limits of a partitioning strategy, you may need to extend the scalability to a different level. For example, if partitioning is at the database level it may mean locating or replicating partitions in multiple databases. If partitioning is already at the database level, and physical limitations are an issue, it may mean locating or replicating 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 it 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:

- Consider executing a periodic process to locate any data integrity issues and either attempt to fix these issues automatically or raise an alert.
- Consider how you will implement appropriate management and operational tasks when the data is partitioned, such as 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 the data can be loaded into multiple partitions, and how new data arriving from other sources might be added. Some tools and utilities may not support sharded data operations such as loading data into the correct partition, and so this may require creating or obtaining new tools and utilities.
- How the data will be archived and deleted on a regular basis (perhaps monthly) to prevent excessive growth of partitions. It may be necessary to transform the data to match a different archive schema.

Related Patterns and Guidance

The following patterns and guidance may also be relevant to your scenario when implementing data partitioning in your applications and data stores:

- Sharding Pattern. Sharding enables a data store to scale more easily by distributing partitions across storage nodes. This pattern describes how to divide a data store into horizontal partitions.
- Data Consistency Primer. Managing and maintaining data consistency across partitions is an important concern, particularly in terms of the concurrency and availability issues that can arise. You frequently need to trade strong consistency for performance by adopting an eventual consistency model. This primer discusses the advantages and limitations of the two consistency models.
- Data Replication and Synchronization Guidance. Data can be replicated across partitions in different locations, and it may be necessary to ensure that the replicas are synchronized periodically to ensure that they remain consistent. This guidance summarizes the issues related to replicating data that is distributed across multiple locations, and describes solutions for resolving these issues.
- Index Table Pattern. This pattern describes how to create indexes that enable data to be retrieved quickly in a partitioned data store.
- Materialized View Pattern. This pattern describes how to generate pre-populated views that summarize data to support fast query operations. This approach can be useful in a partitioned data store if the partitions containing the data being summarized are distributed across multiple sites.

More Information

- The section Building a Polyglot Solution in the patterns & practices guide Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence on MSDN.
- The page Real World: Designing a Scalable Partitioning Strategy for Microsoft Azure Table Storage on MSDN.

Next Topic | Previous Topic | Home | Community

patterns & practices
proven practices for predictable results