Sharding by Hash Partitioning
A Database Scalability Pattern to Achieve Evenly Sharded Database Clusters

Caio H. Costa, João Vianney B. M. Filho, Paulo Henrique M. Maia
and Francisco Carlos M. B. Oliveira
Universidade Estadual do Ceara, Fortaleza, Brazil

Keywords: Database Sharding, Hash Partitioning, Pattern, Scalability.

Abstract: With the beginning of the 21st century, web applications requirements dramatically increased in scale. Applications like social networks, ecommerce, and media sharing, started to generate lots of data traffic, and companies started to track this valuable data. The database systems responsible for storing all this information had to scale in order to handle the huge load. With the emergence of cloud computing, scaling out a database system has became an affordable solution, making data sharding a viable scalability option. But to benefit from data sharding, database designers have to identify the best manner to distribute data among the nodes of shared cluster. This paper discusses database sharding distribution models, specifically a technique known as hash partitioning. Our objective is to catalog in the format of a *Database Scalability Pattern* the best practice that consists in sharding the data among the nodes of a database cluster using the hash partitioning technique to nicely balance the load between the database servers. This way, we intend to make the mapping between the scenario and its solution publicly available, helping developers to identify when to adopt the pattern instead of other sharding techniques.

1 INTRODUCTION

With the beginning of the 21st century, web applications requirements dramatically increased in scale. Web 2.0 technologies made web applications much more attractive due to improvements on their interactivity. A whole new set of applications has emerged: social networks, media sharing applications and online office suites, for example. Those new applications attracted a huge number of users, many of which have migrated from local to online applications. Concurrently, many companies developed SOA-based applications making easier the integration between different systems and increasing the reuse of functionalities through services. With this scenario, new integration functionalities have been developed and applications have began to exchange lots of data.

The amount of data to be persisted and managed grew in the same proportion as the data traffic of this new environment grew. Nowadays, large e-commerce sites have to manage data from thousands of concurrent active sessions. Social networks record the activities of their members for later analysis by recommending systems. Online applications store the preferences of millions of users. Coping with the increase in data and traffic required more computing resources. Consequently, the databases responsible for storing all that information had to scale in order to handle the huge load without impairing services and applications performance.

Database systems can scale up or scale out. *Scaling up* implies bigger machines, more processors, disk storage, and memory. Scaling up a database server is not always possible due to the expensive costs to acquire all those resources and to physical and practical limitations. The alternative is *scaling out* the database system, which consists of grouping several smaller machines in a cluster (Sadalage and Fowler, 2013). To increase the cluster capacity, commodity hardware is added on demand. The relatively new but quickly widespread cloud computing technology has turned more affordable the necessary infrastructure for scaling out systems.

Once the database has the necessary infrastructure for running on a large cluster, a distribution model has to be adopted based on the application requirements. An application may require read scalability, write scalability, or both. Basically, there are two distribution models: *replication* and *sharding*. The former takes the same data and copies it into multiple

H. Costa C., Vianney B. M. Filho J., Henrique M. Maia P. and Carlos M. B. Oliveira F..
Sharding by Hash Partitioning - A Database Scalability Pattern to Achieve Evenly Sharded Database Clusters.
DOI: 10.5220/0005376203130320
In *Proceedings of the 17th International Conference on Enterprise Information Systems* (ICEIS-2015), pages 313-320
ISBN: 978-989-758-096-3

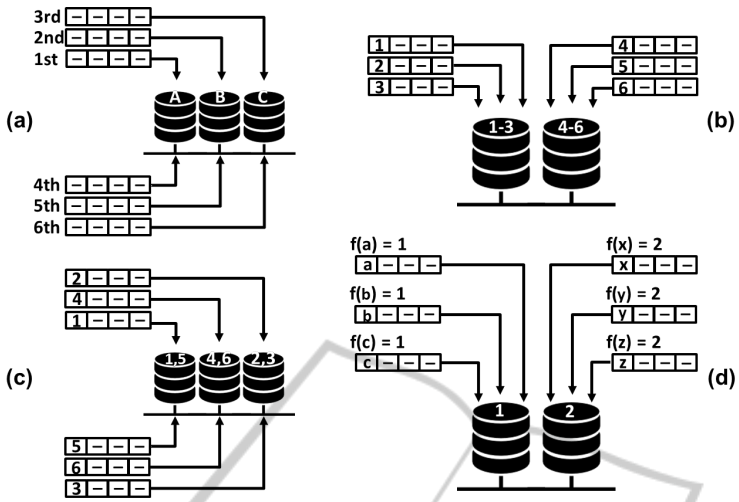

Figure 1: The four partitioning strategies: (a) round-robin, (b) range, (c) list, and (d) hash (DeWitt and Gray, 1992).

nodes, while the latter puts different data on different nodes (Sadalage and Fowler, 2013). Both techniques are orthogonal and can be used together.

This paper discusses database sharding distribution models, specifically a technique known as hash partitioning. The goal of this work is to catalog in the format of a *Database Scalability Pattern* the best practice that consists in sharding the data among the nodes of a database cluster using the hash partitioning technique to nicely balance the load among the database servers. The pattern avoids the creation of hot spots, which means data ranges that concentrate the read and write operations in one node or in a group of nodes. This sharding technique is embedded in some NoSQL databases like AmazonDB (DeCandia et al., 2007) and MongoDB (Boicea et al., 2012), and in ORM frameworks for relational databases, e.g., EclipseLink (Kalotra and Kaur, 2014) and Hibernate Shards (Wu and Yin, 2010). Our main contribution is making publicly available the mapping between the scenario and its solution, helping developers to identify when to adopt hash partitioning in sharded clusters instead of other sharding techniques.

The remainder of the paper is structure as follows. Section 2 presents the background and Section 3 discusses the main related work. Section 4 describes the pattern using the format presented by Hohpe and B. Woolf (Hohpe and B.Woolf, 2003) which is used to name the subsections of the pattern. Finally, Section 5 brings the conclusion and future work.

2 BACKGROUND

As data traffic and data volume increase, it becomes more difficult and expensive to scale up a database server. A more appealing option is to scale out, that is, run the database on a cluster of servers where several nodes can handle the requests. With a distributed database architecture, the load is distributed among the servers that constitute the cluster, thus increasing the performance and availability of the system. There are two architectures for scaling out a database system: distributed database systems and parallel database systems. Both approaches are used in high performance computing, where there is a need for multiprocessor architecture to cope with a high volume of data and traffic.

The work done by Elmasri and Navathe (Elmasri and Navathe, 2011) defines distributed database systems as a collection of multiple logically interrelated databases distributed over a computer network, and a distributed database management system as a software system that manages a distributed database while making the distribution transparent to the user. In this architecture, there are no shared hardware resources and the nodes can have different hardware configurations. Parallel database management systems link multiple smaller machines to achieve the same throughput as a single, larger machine, often with greater scalability and reliability than a single-processor database system (Connolly and Begg, 2005). However, multiple processors share either memory (disk storage and primary memory) or only disk storage (in this case, each processor has its own primary memory).

Many database vendors offer scalability solutions based on the shared disk approach, like Oracle RAC (Abramson et al., 2009) does. These kind of solutions link several servers through a high speed net-

work, but they still have a limiting device. They share a storage device which acts like a system bottleneck. In the nothing shared parallel architecture, processors communicate through a high speed network and each of them has its own primary and secondary memory. That type of architecture requires node symmetry and homogeneity, but in pure distributed database, system homogeneity is not required. To achieve real horizontal scalability, a pure distributed database system architecture must be adopted.

2.1 Distribution Models

Once the hardware resources, server nodes, for deploying a distributed database are available, a distribution model should be chosen to leverage the cluster capacity. Roughly, there are two paths to data distribution: replication and sharding.

2.1.1 Replication

Replication can be performed in two ways: master-slave and peer-to-peer. In a master-slave scheme, one node is responsible for processing any updates to the data and a background process is responsible for synchronizing the data across the other nodes, the slaves. That kind of replication is recommended for intensive data read applications. To increase the cluster capacity of processing read requests, more slaves can be added. However, the write throughput is limited by the capacity of processing write requests of the master node. In peer-to-peer replication, there is no master node. All the replicas can accept write requests, thus improving the system capacity of handling write requests. On the other hand, peer-to-peer replication clusters have to deal with inconsistency problems that may arise.

2.1.2 Sharding

Sharding is a way of scaling out a database via horizontal fragmentation. A horizontal fragment of a relation (table) is a subset of the tuples in that relation. The tuples that belong to the horizontal fragment are specified by a condition on one or more attributes of the relation (Elmasri and Navathe, 2011). Often, only a single attribute is involved. That is, the horizontal scalability is supported by putting different parts of the data onto different servers of a cluster. The objective is making different clients talk to different server nodes. Consequently, the load is balanced out nicely among the servers. Sharding is particularly valuable for performance since it can improve both read and write performance. Replication can greatly improve read performance but does little for applications that have several write operations. Sharding provides a way to horizontally scale those operations.

2.1.3 Partitioning Strategies

There are four different strategies for partitioning data across a cluster: round-robin, range, list, and hash partioning (DeWitt and Gray, 1992). The simplest partitioning strategy is the round-robin, which distributes the rows of a table among the nodes in a round-robin fashion (Figure 1a).

For the range, list, and hash strategies, an attribute, known as *partitioning key*, must be chosen among the table attributes. The partition of the table rows will be based on the value of the partitioning key. In the range strategy, a given range of values is assigned to a partition. The data is distributed among the nodes in such a way that each partition contains rows for which the partitioning key value lies within its range (Figure 1b).

The list strategy is similar to the range strategy. In the former each partition has a list of values assigned one by one. A partition is selected to keep a row if the partitioning key value is equal to one of the values defined in the list (Figure 1c). In the latter, the mapping between the partitioning key values and its nodes is based on the result of a hash function. The partitioning key value is used as parameter of the hash function and the result determines where the data will be placed (Figure 1d).

3 RELATED WORK

A catalog of software design patterns was first described by Gamma et al. (Gamma et al., 1994), a pioneer work in the computer science field. Subsequently, Fowler et al. (Fowler et al., 2002) and Hohpe and B. Woolf (Hohpe and B.Woolf, 2003) identified and published software engineering architectural patterns.

Shumacher et al. (Shumacher et al., 2006) present, in the format of patterns, a set of best practices to turn applications more secure at different levels. Hafiz (Hafiz, 2006) describes four design patterns applicable to the design of anonymity systems and can be used to secure the privacy of sensitive data. Shumacher (Shumacher, 2003) introduces an approach for mining security patterns from security standards and presents two patterns for anonymity and privacy. Strauch et al. (Strauch et al., 2012) describe four patterns that address data confidentiality in the cloud.

The authors in (Eessaar, 2008) propose a pattern-based database design and implementation approach

Table 1: Common partitioning hash keys and their efficiency.

Hash Key	Efficiency
User id in applications with many users.	Good
Status code in applications with few possible status codes.	Bad
Tracked device id that stores data at relatively similar intervals.	Good
Tracked device id, where one is by far more popular than all the others.	Bad

that promotes the use of patterns as the basis of code generation. In addition, they present a software system that performs code generation based on database design patterns. However, their work does not list the database design patterns that the proposed tool is based on and does not mention design patterns related to distributed database systems.

The paper (Fehling et al., 2011) proposes a pattern-based approach to reduce the complexity of cloud application architectures. The pattern language developed aims to guide the developers during the identification of cloud environments and architecture patterns applicable to their problems. Fehling et al. (Fehling et al., 2011) also gives an overview of previously discovered patterns. In that list, there are six patterns related to cloud data storage, but none of them is related to data sharding techniques.

Pallman (Pallmann, 2011) presents a collection of patterns implemented by the Microsoft Azure Cloud Platform and describes five data storage patterns which can be used to store record-oriented data. In (Go, 2014), the Partition Key pattern describes Azure's data distribution at a high level, without specifying the strategy used to actually partition the data. The pattern described in our paper is focused in the strategy used to distribute data and is not oriented to any vendor's platform.

The white paper (Adler, 2011) suggests a reference architecture and best practices to launch scalable applications in the cloud. The suggested best practices, which are not organized as patterns, were derived from the wealth of knowledge collected from many different industry use cases. Although Adler (Adler, 2011) discusses database scalability, it only addresses master-slave replication.

Stonebraker and Cattel (Stonebraker and Cattell, 2011) present ten rules to achieve scalability in datastores that handle simple operations are listed. During the discussion of the rules, the authors presents important drawbacks about sharding data in specific situations. The observations made by Stonebraker and Cattel (Stonebraker and Cattell, 2011) were important for the construction of the pattern presented in this work.

In addition to the four confidentiality patterns, Strauch et al. (Strauch et al., 2012) also present two patterns related to horizontal scalability of the data access layer of an application: Local Database Proxy and Local Sharding-Based Router. The Local Sharding-Based Router pattern suggests the extension of the data access layer with the addition of a router responsible for distributing the read and write requests among the database servers. Each database server in the cluster holds a portion of the data that was partitioned using some data sharding technique. Nonetheless, that pattern does not suggest any particular data sharding strategy. It can be used as a complement to our proposed pattern in order to implement a read and write strategy after the deployment of the pattern described here.

4 SHARDING BY HASH PARTITIONING

The goal of the database scalability pattern entitled Sharding by Hash Partitioning is to distribute, as evenly as possible, a functional group among the nodes of a cluster using the sharding distribution method. The desired result is an homogeneous distribution of the load generated by client requests among the databases servers.

4.1 Context

Scaling out techniques are used to increase the performance of database systems. The master/slave architecture provides read scalability but does not help much for write intensive applications. To obtain general read and write scalability, the sharding technique should be used. But nothing shared systems scale only if data objects are partitioned across the system's nodes in a manner that balances the load (Stonebraker and Cattell, 2011).

Clusters that use the sharding method to partition the data, distribute the data portions among several database servers to balance queries and update requests nicely. On the other hand, a bad distribution can originate two issues: concentration spots that hold the most requested registries, and servers that concentrate the majority of update requests. These points of concentration, known as hot spots (Stonebraker and Cattell, 2011), can be a data set kept in only one server

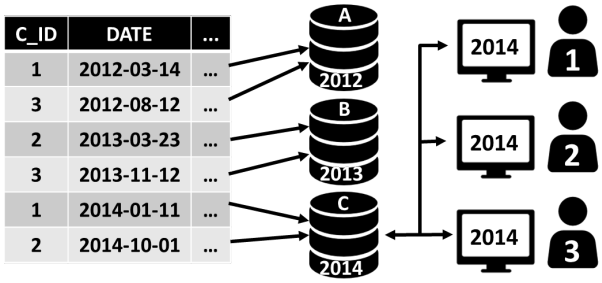

Figure 2: The transaction log table partitioned by range.

or data sets distributed between a few servers.

For instance, suppose a table containing a date field is chronologically distributed between three database servers: A, B and C. Server A is responsible for storing the oldest data, server B is responsible for storing the intermediate data, and server C is responsible for storing the most recent data. If the client application keeps generating new data, the server C will receive the majority of the update requests. In addition, there is a great chance of server C receiving the majority of the queries because, generally, application users are more interested in recent information. In this scenario, the attempt to distribute the load across the three database servers by sharding the data using the range sharding strategy will fail.

4.2 Challenge

How to distribute data avoiding the creation of hot spots to obtain a sharded cluster that nicely balance the requests load across its nodes?

4.3 Forces

When using sharding, partitioning the data among the nodes to avoid hot spots is not a trivial task. The appropriate partitioning strategy must be chosen depending on the application data access pattern. Most of the times, choosing the wrong sharding strategy will decrease the system's performance.

Excess communication can happen in those database systems where data placement was not carefully chosen by the database designer (Zilio et al., 1994). When the round-robin sharding strategy is used, data is distributed without considering its attributes. Registries that are commonly aggregated to compose a query result may be scattered across the nodes generating data traffic overhead.

It is natural to think of distributing the data based on natural partitioning keys like date fields. Therefore, the obvious choice would be the range partitioning strategy. However, applications in which the users are interested only in recent information, range partitioning can generate hot spots like in the example shown at the end of Section 4.1.

In addition to balance the load among the database servers in the cluster nicely, a partitioning strategy should generate no significant overhead when discovering in which node a registry must reside.

4.4 Solution

The hash partitioning can be used to achieve an even distribution of the requests load among the nodes of a distributed database system. A field whose values are particular to a group of registries must be chosen as the partitioning key. The choice on the partitioning should take into account the fact that registries that are commonly accessed together will share the same value for the partitioning key.

Queries interested in data that shares the same partitioning key value will hit the same node. Consequently, when two or more requests ask for data with different values for the partitioning key, they will be directed to different nodes. The same happens when saving new data in the sharded cluster. A hash function is applied to the value of the partition key in the new registry. The result will determine in which node the data will be stored. Hence, if two new registries present different values for their partitioning key, they will be stored in different nodes of the cluster.

Once the partitioning key field is chosen, an index can be created based on another field to order the registries with the same partitioning key value. With the existence of that secondary index, results from queries that search for registries with the same value for the partitioning key can be ordered.

4.5 Results

Data placement is very important in nothing shared parallel database systems. It has to ease the parallelism and minimize the communication overhead (Zilio et al., 1994). Systems that aggregate the necessary information in only one registry are best suited

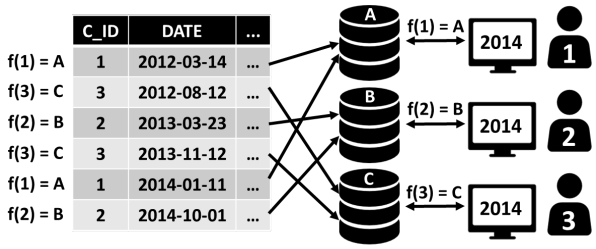

Figure 3: The transaction log table partitioned by hash key.

for that approach since it will minimize the network traffic. Thus, it is important to avoid multi-shard operations to the greatest extent possible, including queries that go to multiple shards, as well as multi-shard updates requiring ACID properties (Stonebraker and Cattell, 2011). Therefore, sharding data among clusters nodes is best suited for applications that does not require joins between database tables.

If the hash partitioning needs to be completely implemented in the data access layer of an application, an algorithm and a hash function need to be chosen to realize the hash partitioning method. For instance, the MD5 (Rivest, 1992) can be the hash function implementation and the Consistent Hashing algorithm (Karger et al., 1997) can be used to distribute and locate data across the cluster. Tools that implement the pattern, like some NoSQL datastores that originally support sharding and some persistence frameworks, already have an implemented algorithm.

Even if the datastore or persistence framework being used already implement the pattern, choosing the right partitioning hash key is a task of the database designer. Choosing the right partitioning key is very important when using the hash partitioning method. The database designer must choose the right partitioning key to keep the workload evenly balanced across partitions. For instance, if a table has a very small number of heavily accessed registries, even a single one, request traffic is concentrated on a small number of partitions. To obtain a nicely balanced workload, the partitioning hash key must have a large number of distinct values, which are requested fairly uniformly, as randomly as possible. Table 1 shows common partitioning hash keys and their efficiency.

4.6 Next

Cloud datastores provide the necessary infrastructure to scale at lower costs and using a simplified management. However, the majority of cloud relational databases does not implement the Sharding by Hash Partitioning pattern. Indeed, they do not support sharding natively. In such cases, the datastore functionalities can be extended by implementing the pattern in the application data access layer. Strauch et al. (Strauch et al., 2012) describe the pattern Local Sharding-Based Router which complements the pattern described in this paper. The pattern Local Sharding-Based Router proposes an extra layer, deployed in the cloud, responsible for implementing sharding in cloud datastores that do not support sharding natively. The Local Sharding-Based Router pattern does not suggest any sharding strategy.

4.7 Sidebars

There are NoSQL datastores, like MongoDB (Liu et al., 2012) and DynamoDB (DeCandia et al., 2007), that implement the Sharding by Hash Partitioning pattern. Those datastores can be used if an application access pattern does not require joins between different tables, and requires the scalability offered by data sharding and the workload balance offered by hash partitioning. That is, if the application will benefit from the Sharding by Hash Partitioning pattern and does not need the relation constraints offered by relational databases, the mentioned datastores can be used. The paper (DeCandia et al., 2007) describes the hash partitioning algorithm used by DynamoDB. It is a variation of the Consistent Hashing algorithm (Karger et al., 1997) and can be used as a reference for implementing the Sharding by Hash Partitioning pattern during a framework development or in an application data access layer.

There are few relational databases which support sharding in a real nothing shared architecture. The ones that do support, generally, are more complex to maintain. If an application needs to shard data by hash partitioning, but the relational database used does not provide this feature, a framework that implements the Sharding by Hash Partitioning pattern can be used. EclipseLink implements a data sharding by hash key policy and can provide this feature to relational databases that do not support sharding. Alternatively, it can simply be used if developers do not know how to configure databases that support sharding.

In on-premise applications that stores data in the

cloud, if the Sharding by Hash Partitioning pattern has to be implemented in the data access layer, this must be done at the cloud side to avoid network latency if more than one node needs to be queried (Strauch et al., 2012).

4.8 Example

To improve the understanding of the Sharding by Hash Partitioning pattern, a system that logs each transaction realized by customers of a bank will be used as an example. A single table stores all the transaction log registries. Each registry has a field that describes any common bank transaction that can be performed by a client of the bank, such as withdrawal, deposit, or transfer. As expected, the table has a field that holds the transaction date.

Over time the table becomes very large. The IT staff decides to shard the table data across nodes of a cluster to improve performance and obtain scalability. The staff creates a cluster composed of three database servers. In the first attempt, the table is chronologically partitioned, that is, a range partitioning based on the transaction date is configured. Server A stores the oldest transactions, and server C stores the more recent transactions (Figure 2). This partitioning scheme generates hot spots. All new transaction log registries are stored in server C and most of the bank customers consult their latest transactions, which are also stored in server C.

In this case the use of hash key partitioning is recommended. The bank IT staff decides to shard the transaction logs table using the customer ID as the partitioning key. Furthermore, they create an index based on the transaction date field. Now, the result of a hash function applied to the customer ID determines where the transaction registry will be stored (Figure 3). Due to the large amount of customers, probabilistically, the data is more evenly partitioned. When customers consult their latest transactions, the requests will be distributed across the nodes of the cluster. The index based on the transaction date will keep the transaction log registries ordered within a customer query result.

5 CONCLUSIONS

The data sharding based on hash key partitioning, identified and formalized as a database scalability pattern in this work, efficiently provides read and write scalability improving the performance of a database cluster. Data sharding by hash key partitioning, however, does not solve all database scalability problems. Therefore, it is not recommended to all scenarios. The formal description of the solution as a pattern helps in the task of mapping the data sharding by hash key partitioning to its recommended scenario.

As future work we intend to continue formalizing database horizontal scalability solutions as patterns, so we can produce a catalog containing a list of database scalability patterns that aims to solve scalability problems.

REFERENCES

Abramson, I., Abbey, M., Corey, M. J., and Malcher, M. (2009). *Oracle Database 11g. A Beginner's Guide*. Oracle Press.

Adler, B. (2011). Building scalable applications in the cloud. reference architecture and best practices.

Boicea, A., Radulescu, F., and Agapin, L. I. (2012). Mongodb vs oracle - database comparison. In *Proceedings of the 2012 Third International Conference on Emerging Intelligent Data and Web Technologies*, pages 330–335. IEEE.

Connolly, T. M. and Begg, C. E. (2005). *DATABASE SYSTEMS. A Practical Approach to Design, Implementation, and Management*. Addison-Wesley, 4th edition.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA. ACM.

DeWitt, D. and Gray, J. (1992). Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98.

Eessaar, E. (2008). On pattern-based database design and implementation. In *Proceedings of the 2008 International Conference on Software Engineering Research, Management and Applications*, pages 235–242. IEEE.

Elmasri, R. and Navathe, S. B. (2011). *Fundamentals of Database Systems*. Addison-Wesley, 6th edition.

Fehling, C., Leymann, F., Retter, R., Schumm, D., and Schupeck, W. (2011). An architectural pattern language of cloud-based applications. In *Proceedings of the 18th Conference on Pattern Languages of Programs*, number 2 in PLoP '11, pages 1–11, New York, NY, USA. ACM.

Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., and Stafford, R. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition.

Go, J. (2014). Designing a scalable partitioning strategy for azure table storage. http://msdn.microsoft.com/en-us/library/azure/hh508997.aspx.

Hafiz, M. (2006). A collection of privacy design patterns. In *Proceedings of the 2006 Conference on Pattern Languages of Programs*, number 7 in PLoP '06, pages 1–7, New York, NY, USA. ACM.

Hohpe, G. and B.Woolf (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 1st edition.

Kalotra, M. and Kaur, K. (2014). Performance analysis of reusable software systems. In *2014 5th International Conference on Confluence The Next Generation Informatino Technology Summit*, pages 773–778. IEEE.

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. (1997). Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA. ACM.

Liu, Y., Wang, Y., and Jin, Y. (2012). Research on the improvement of mongodb auto- sharding in cloud environment. In *Proceedings of the 7th International Conference on Computer Science and Education*, pages 851–854. IEEE.

Pallmann, D. (2011). Windows azure design patters. http://neudesic.blob.core.windows.net/webpatterns/index.html.

Rivest, R. (1992). The md5 message-digest algorithm. IETF RFC 1321.

Sadalage, P. J. and Fowler, M. (2013). *NoSQL Distilled. A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 1st edition.

Shumacher, M. (2003). Security patterns and security standards - with selected security patterns for anonymity and privacy. In *European Conference on Pattern Languages of Programs*.

Shumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., and Sommerlad, P. (2006). *Security Patterns: Integrating Security and Systems Engineering*. Wiley.

Stonebraker, M. and Cattell, R. (2011). 10 rules for scalable performance in 'simple operation' datastores. *Communications of the ACM*, 54(6):72–80.

Strauch, S., Andrikopoulos, V., Breitenbuecher, U., Kopp, O., and Leymann, F. (2012). Non-functional data layer patterns for cloud applications. In *2012 IEEE 4th International Conference on Cloud Computing Technology and Science*, pages 601–605. IEEE.

Wu, P. and Yin, K. (2010). Application research on a persistent technique based on hibernate. In *International Conference on Computer Design and Applications*, volume 1, pages 629–631. IEEE.

Zilio, D. C., Jhingran, A., and Padmanabhan, S. (1994). Partitioning key selection for a shared-nothing parallel database system. Technical report, IBM Research Division, Yorktown Heights, NY.