Universiteit Antwerpen

Finding efficient Shard Keys with a learning process on query logs in Database Sharding

Research internship report submitted on 27 August 2014.
in the faculty of Mathematics and Computer Science, at
the University of Antwerp.

Promotor:
Prof. Dr. Floris Geerts

Emad Heydari Beni

ADReM

Research Group Antwerp
Advanced Database Research and Modelling

Contents

List of Figures

List of Tables

Author

The Author grants to University od Antwerp the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

Finding efficient Shard Keys with a learning process on query logs in Database Sharding

EMAD HEYDARI BENI, August 2014
Email: emad7105@hotmail.com
emad.heydaribeni@student.uantwerpen.be

Promoter: Prof. Dr. Floris Geerts

University of Antwerp
Department of Mathematics and Computer Science
ADReM Research Group
Antwerp
Belgium

Acknowledgements

I would like to express the deepest appreciation to Prof. dr. Floris Geerts, who helped me to find and start this subject for my second research internship. Without his guidance and persistent support this literature would not have been possible.

Abstract

Nowadays industry has been experiencing a dramatic data growth. Not only this data has to be processed properly, but also it must be stored somewhere with smart strategy to be able to write and read with highest possible speed. Over the past decades, vendors have been motivated to migrate their brown-field database solutions to a distributed version through Partitioning/Sharding concepts. (In)appropriate shard keys have a great impact on the future performance of the whole application. The improper choices may cause SLA violation for enterprises and end up with business failure. In the first chapter, we introduce various approaches of data partitioning along with challenges that you may face. The second chapter explores the Sharding strategy utilised by two famous vendors. And, lastly we propose an automatic approach for detection of efficient Sharding Scheme and Sharding keys with a learning process on existing query logs of a database.

Keywords: Database Sharding, Partitioning, Sharding, Shard Key

CHAPTER 1

Introduction

Most applications have been in need of storage means to keep data. Data may be produced by either users input or applications themselves. Over the past few decades, the most known and commonly-used type of database has been Relational Database Management System (RDBMS) which is based on Structured Query Language (SQL). With advent of Cloud Computing and boom of data in all ranges of applications such as everyday web or scientific applications, NoSQL[1] type of databases has started to play a crucial role in storage systems. Various types of this young storage mechanism have been around such as[2]:

- Key-Value Store
- Column-Oriented Store
- Document-Oriented Store
- Graph Database

One of the recent controversial topics among data scientists is which mechanism is superior to the other one. This comparison is fundamentally inappropriate, since each approach has its own advantages. According to [3], "The NoSQL Discussion has nothing to do with SQL.". This comparison is not in the scope of our literature.

In traditional applications, data usually resides in a single instance of a database (machine) with replication instances to ensure high-availability and redundancy. Although hardware advancement with considerable storage abilities in recent years

[1]Not SQL. This type of databases does not follow Relational structuring of data in shape of relations, so called Tables.

might enable us to handle data of medium-sized applications on a single-machine[2], data growth, so called data explosion, is the way faster than hardware technology progress which obliges database specialists to think about commodity machines in order to store the data.

1.1 Database Partitioning

Partitioning is about logically splitting tables or entities into smaller independent pieces which brings manageability, performance and high-availability if it is done properly. Although database elements division done by partitioning strategies conceptually are supposed to be performed from logical perspective, this idea is widely being utilised in physical fashion as well. According to [4], database partitioning comprises several methods: *Master/Slave Partitioning*, *Vertical Partitioning*, *Horizontal Partitioning*, and *Shared-Nothing/Sharding*.

Master/Slave Partitioning. This is the traditional way of improving the response time of the whole application by adding Master instances for all data updating purposes along with Slave nodes for read requests. All updates are continuously being dispatched to slaves to catch up with latest changes done by application layer. Basically there is no data partitioning here, except forwarding requests to various nodes for improvement in performance and availability.

Vertical Partitioning.[3] Vertical partitioning subdivides attributes into groups and assigns each group to a physical object [5]. In other words, this method is about row/entity/column-set splitting which concludes with partitioning a row with some columns in one database, and the other columns in another database.

Vertical partitioning is somehow similar to Normalization concepts in relational database design with more focus on performance aspects. In some scenarios, a table may contain, as a case in point BLOB[4], unstructured data columns along with other simpler columns. If the structured and simple columns of this type of tables get queried a lot in production, the database system is obliged to read and update all columns which lead to expensive IO operations. As mentioned in [7], by applying vertical partitioning, one can store structured columns in one tablespace, and the large unstructured columns which may be queried for update less than others in another tablespace. These two tablespaces may be located in different machines with various hardware capabilities.

[2]Monolithic Databases

[3]There is a misunderstanding and confusion between Vertical partitioning and Vertical scaling. Vertical scaling is about improving the machine hardware with more memory and processing unit which is somehow irrelevant to Vertical partitioning and its logical concepts.

[4]Binary Large Object

For example in Fig.1.1, the employee table contains a BLOB column to store picture of the entity. But in most cases, users tend to change the other attributes of the entity rather than picture all the time. As illustrated in [6], experiments on 3691 affected rows with and without vertical partitioning represent that CPU Execution Time is 250ms for normal Employee table, and 31ms for vertically partitioned table as shown in Fig.1.1.

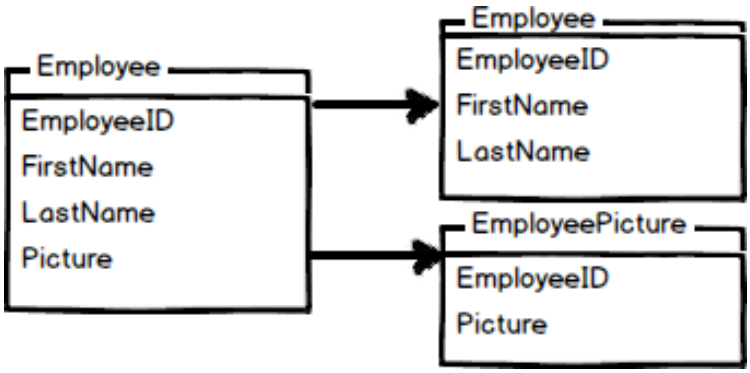


Figure 1.1: Example of vertical partitioning [6]

Horizontal Partitioning. This method is about logically separating rows of a table by one or multiple columns and store them in different databases with the same table schema in order to improve scalability, faster data retrieval, high-availability, and somehow better manageability. In a simple Twitter-like application, for example, Tweet messages can be separated by their Dates in general, and their published month in specific. In other words, Tweets of January resides in January Database, Tweets of February resides in February database, and so on. This approach bring benefit of having smaller index size to database management systems which leads to faster data retrieval with less search efforts.

Database Sharding[5] or Shared Nothing. This method is conceptually quite similar to Horizontal Partitioning; But it not only uses logical entity separation, but it also takes physical aspect into account. The difference is that Database Sharding places the separated rows in different machines. Sharding is discussed in following section with more details.

1.2 Database Sharding

Sharding is a kind of horizontal partitioning. In other words, it is a method of storing data across multiple machines in order to support very large data sets and ensure high throughput operations. Every shard is a single machine, which is independent

[5]The term 'Shard' has been introduced by Google.

from other shards. And, all shards logically are one database which is sharded (i.e. partitioned) and distributed among some real machines under the hood. Sharding considerably reduces the amount of data that each server needs to store in one single machine. In other words, instead of using very large machine that sometimes it is not feasible to prepare, we can use multiple smaller machines. In addition, we can always scale out/in by adding/removing machines. From performance perspective, we reduce the number of operations that every machine(shard) is going to handle comparing to classic database architectures. Because, data is distributed among shards, and there are only few machines responsible for a particular operation.

Shard. A shard is an instance of database which holds a portion of rows. A distributed database with Sharded architecture contains shards with independent machines. Every shard has own memory, processing unit and networking features. From application perspective, there is only one database and all shards are running transparently. Since sharding is an specific type of horizontal partitioning with shared nothing instances, every database instance, so called shard, maintains the same database schema[6].

Shard Key. The shard key, depending on the database vendor, might be one or multiple columns of a table which is(are) responsible to determine the target shard to store the rows. The selected shard key, which is the main topic of this literature, should meet some necessary requirements that we will discuss them in following sections.

Some database vendors support having replicated rows in other shards which is typically called *replication factor*. If database specialist choose a high replication factor, the degree of availability gets higher along with slower update operations. This trade-off introduces new concepts such as *eventually consistency* which is not in the scope of this literature.

Sharding Coornidantor[7]. Most of the developed sharding strategies in production use some nodes to coordinate the shards by application data queries based on shard Key(s) as well as keeping necessary metadata of existing data within shards. In some intelligent sharded architectures, coordinators acts somehow similar to an advanced query optimizer by producing efficient query execution plans as well as performing necessary joins requested by user.

1.2.1 When To Use Sharding?

The necessity of having scalable storage strategies are undeniable with massive growth of data in last decade. The performance of every database implementation is based on local processing unit, memory and local/network disk. Scalability of typical storage approaches are tied to these local or cluster-wise metrics. But

[6]Sharding is not only about relational databases. NoSQL types also have considerable support for sharding and partitioning in general.

[7]This node is also called *Aggregator*, *Shard Manager*, *Shard Coordinator*, *Query Router*, etc.

according to [8] in fact, these components can only get scaled to a given point in order to have a decent degree of performance. Table 1.1 is a synopsis of [8] regarding other storage options except database sharding.

Table 1.1: Other Storage options shortcomings in case of scalability, gist of [8].

Storage option	
Master/Slave	It is about having a single Master server receiving CUD[8], and replicated Slaves for Read operations. Master node has Write scalability problem. Since slaves continuously receive the latest changes from Master node, in case of disaster recovery of the application while Master is down, it is likely to loose the latest changes which is not acceptable for some enterprises.
Cluster Computing	It is formed in a group of database instances in cluster of machines with shared messaging platform as well as shared disk facilities (SAN[9]). One node is responsible for CUD and the other nodes for READ. If the CUD node gets down, another node takes the responsibility of CUD operations. This ensure the high-availability, but not yet Write scalability. More advanced clustering techniques using real-time memory replication enable all nodes to handle CRUD operations. In case of scalability and node extension, there might be a heavy memory replications which have influence on Disk and network I/O's with order of magnitude.

Data size. When one or some table(s) in a database are notably getting larger in size is the time to think about database sharding and shrinking the tables into smaller tables in different machines. If specially the indexed data is increasing monotonically and a bigger part of your dataset are only going to be read in future, it is highly recommended to take sharded architecture into consideration.

Write Scalability. Master/Slave strategy meets the need of *Read Scalability* by having enough slaves to respond to users Read requests. But, there will be one Master responsible for CUD[10] operations which can reduce the **Write Scalability**. If an application turns out to be the target of considerable number of CUD operations, Sharding architecture should be take into consideration.

Other resources constraints. According to general definition of Sharding, when there is no enough resource in one of database machines, a new node should be added. But this statement does not only point to storage resources. It's about

[10]Create Update Delete

various factors such as disk storage, RAM, CPU, and network. Since data does no longer resides in one machine, the data size of each shard is relatively smaller than the whole database. Then, databases can benefit from keeping all indexes along with big portion of data on the RAM and boost the performance of the operations. But what if there is plenty of space on disk storage in a node but RAM is almost full? Most databases start relocating the less hot data blocks to the disk which is quite slower for random access operations. Other situation may be the lack of processing unit in a machine. There might be sufficient RAM and storage capacity on a node. But since an application might execute CPU intensive operations more often, this metric could be a bottleneck of the whole application performance. Furthermore, network IO may also have influence on system performance among those applications in which operations are triggering considerable amount of data movement in between nodes. All these measures should be, therefore, taken into account when one wants to scale out by adding a new shard.

1.2.1.1 Advantages of having Database Sharding

In this section we consider some of key drivers of exploiting Database Sharding including *Manageability*, *Availability*, and *Cost Reduction*.

Manageability. As matter of fact that Sharded architecture is composed of shards with smaller datasets, managing smaller instances, including maintenance and backup creation, with less amount of data is the way easier and faster than traditional huge single database instance.

Availability. Each shard takes responsibility of storing a portion of the whole application data in an independent machine. If any of shards stops working due to any kind of reasons, it is not likely to have the whole application outage since users may query other shards. Then, disaster recovery time becomes quite short because of smaller amount of data within every shard. In case of having replication mechanism for every shard, then the probability of having service outage gets closer to zero which is suitable for mission-critical applications.

Cost Reduction [8]. There is no longer need of having cutting-edge servers with very quality hardware along with very expensive disk solutions. On the other hand, cheap commodity computers can be utilized instead. Besides, most of free open-source databases are viable of having sharding mechanism in addition to commercial vendors which is cost-cutting for enterprises.

1.2.2 Sharding Strategies

Sharding strategies can be categorized as four approaches: *Range-based*, *List-based*, *Hash-based*, and *Composite* sharding Which are briefly discussed in Table 1.2

Table 1.2: Database Sharding strategies.

Strategy	**Description**
Range-Based Sharding	Non-overlapping value ranges are assigned to every shard. Sharding manager determine the right shard from shard key value inside query. For instance, Date field of a table is selected as shard key, we might have months as shard ranges. Then, queries with 'February' as shard key value gets redirected to the shard which is responsible for February.
List-based Sharding	This is almost similar to Range-based Sharding with slightly different features. Instead of having ranges, every shard is responsible for a list of values. For example, data of ['Norway', 'Finland', 'Sweden'] reside on a shard. Besides data of ['Belgium', 'Netherlands', 'Luxembourg'] store on another shard and so on. This country names play the shard key role in this strategy.
Hash-Based Sharding	Sometimes there is no meaningful field(s) to be chosen as Shard key(s) and it is intended to have evenly distributed data among shards. Then, field with appropriate cardinality is chosen as shard key. This shard key passes through a pre-defined Hash algorithm in sharding manager and the result of this hashing will be the key of that shard.[11]
Composite Sharding	This approach comprises two or more of above strategies. For instance, List-based strategy can be used along with Hash-based approach. More specifically, a data query may target ['Norway', 'Finland', 'Sweden'] shard. Afterwards, when it gets to that cluster which is responsible for Scandinavia, it may be redirected to a specific machine by Hashing the row ID.

1.2.3 Challenges in Database Sharding

What have been discussed so far represent a big picture of Database Sharding which is kind of Utopia in theory. But, when it comes to implementation and deployment, and maintenance, the real difficulties and challenges may rise based on one-decade-experiences of market leaders like Facebook, Google, Twitter, etc. In this section some of the challenges are going to be introduced briefly.

Data Skew. The input data should get distributed evenly among shards. In other words, it is better and more controllable when you have roughly same amount of data on each node. But, it is not the case all the time due to various types of user requests, application types, your selected shard key and other factors. In a social network application, for instance Facebook, if shard coordination has ben organized

by users ID's and every shard is responsible for a small set of users, what might happen to a user who is celebrity with hundreds of thousands relevant data like comments, photos, and likes which are going to redirected to that particular Shard? How much capacity should that machine have? This may cause performance related issues and violate SLA's. In order to avoid these problems, an appropriate Sharding schema as well as proper shard keys should be used. Besides, the whole cluster should be monitored regularly with some database administrator notifiers.

Inappropriate Sharding schema. Sometimes developers and database designers cannot predict the real load of the system beforehand, and they might choose an improper Sharding scheme. In most cases, changing the strategy is quite difficult task which usually causes downtime. If the newly selected approach is viable, other problems such as slow migration with massive amount of data movement along with slow disk and network IO operations may cause service outage or even in some cases data loss. In order to cope with this difficulties, database designers should consider all aspects of the application before production deployment. There should be the possibility of turning the database into READ-ONLY mode, so called *Dark mode*, while data migration is happening and drop latest DML[12] operations.

Referential Integrity [9]. Single instance databases take care of referential integrity when there are foreign keys between data entities. But, when it comes to a sharded architecture, most of vendors does not support this feature. Then, this responsibility is handed over to the application and developers which introduces new challenges.

Join operations [9]. It is a classical problem in distributed databases including Database Sharding. Traditionally, join operations are being executed in one machine. But with Sharded architecture, tables might be distributed in multiple machines, and a simple join operation may force several machines to produce output and send it back to an aggregator node[13]. There have been no clear solution for this matter so far. But there are some workarounds to either eliminate the impact of these joins or decrease the amount of relevant data distribution by using appropriate shard key for data distribution. In other words, more relevant data should reside on one or closer machines. In addition, a decent caching strategy could be helpful too.

Hard schema change. Changing the schema such as adding new columns to a relatively large sharded architecture is a challenging task.

By reviewing maintenance reports of companies in which Sharding is being used, very useful information can be found. Difficult backup strategies, low-speed data migration because of poor network connectivity among shards, long disaster recov-

[12]Data Manipulation Language

[13]Based on the vendor, it can be either the Sharding coordinator or one of the shards in p2p topologies.

ery RTO[14], locking transactions, single point of failures, etc. are probable challenges of this road.

[14]Recovery Point Objective, It is the maximum tolerable period in which data might be lost from an IT service due to a major incident (Wikipedia)

CHAPTER 2

Database Sharding: State of art

In this chapter we want to skim through several Database sharding solutions by various vendors, and see how they handle different things.

2.1 Case study: Sharding at Tumblr

The main ideas of this section are based on reports published on Tumblr's Engineering Blog [10], and the presentation of Evan Ellias, the Database Engineer at Tumblr corporation, at Velocity Europe 2011 [11] conference.

2.1.1 Drivers to have Database Sharding

Prior to Sharded architecture, they had traditional Master/Slave strategy to store the data. Tumblr's key drivers are listed as follows:

- Sudden growth of their data in just one year: 1.5 Billion posts becomes 12.5 billions as well as increasing number of Blogs from 9 Million to 33 Million.
- Former Master/Slave MySQL strategy had introduced *Write scalability* issue on Master nodes.
- Data growth causes several issues as follows:
  - Full disk on database instances.
  - Full RAM.
  - Slow and poor SSD performance when it is getting full.

- Various single-point-of-failure's in the database cluster.
- Very slow backup procedure.
- Longer to spin up new slaves due to huge data size within Masters.

2.1.2 Requirements

There are set of requirements which are taken into account by Tumblr varying from design decisions to operational requirements.

Sharding Decisions. As briefly discussed in previous and more details in following chapter, an appropriated Shard Key(s) should be chosen in first place. The proposed Shard Key in [11] for their Blog systems are Blog ID's which show that in Sharded architecture, Post ID's are not sufficient anymore for coordinating the appropriate shard.

Then, an appropriate Sharding scheme is quite important. According to [11], Tumblr uses *Range-Based* scheme by Blog ID's. In addition, database designers should be generous in selecting number of initial shards to have small data size on each shard.

MySQL instances features. There should be a support for read-only and off-line shards for any planned maintenance and unexpected failures. There should be also support for reading and writing to different MySQL instances for the same range (i.e. in range-based architecture), not for scaling Reads, but for rebalancing purposes that we will see them in rest of this section.

Centralized tool/service for handling common needs. Since when we have Sharded architecture, there are numerous machines and shards working simultaneously, there should be a tool to manage these machines and instances. For example, for Querying multiple shards at the same time, persistent connections, Shard coordination by parsing SQL's, executing UNIX or MySQL commands on some or all nodes, and other things. This tool could make some Sharding tasks like adding new shards easier and faster for the Database administrator. Currently Tumbler is using their own developed tool called *JetPants*, which is available as an Open-source tool on GitHub. Facebook also recently has introduced a tool called *Presto*[1].

2.1.3 Sharding

Usually Sharding contains two major concerns: firstly *initial Sharding of Tables*, and secondly *how to add new Shards*. In this section, we discuss how these two have been done by Tumblr database specialists based on [11], a presentation in Velocity

[1]Presto is mainly used for running analytics distributed queries. But, it could be used in these types of Sharding tools with some tunings.

Europe 2011 conference.

2.1.3.1 How to initially shard a table?

There are two options for this goal to be accomplished: *Transitional Migration with legacy database*, or *All at once approach.*

Transitional Migration with legacy database. Firstly a *cutoff* ID must be chosen.[2] This ID should be a bit higher than latest generated ID. When database reaches the cutoff ID, new rows must be written to shards instead of legacy database. In addition, if there is any row update among legacy rows, the updated row should be written to the shards too. Furthermore, there should be a slow background process responsible for migration of rows in legacy database to Shards and decreasing the cutoff ID.

All at once. As discussed in previous chapter, there should be a *Dark mode* to send all CUD[3] operation to both legacy and shards at the same time, but still legacy database is responsible for Read requests. Then, a process should exist for migrating data from legacy database to the appropriate shard. Eventually at some point in time, all Read request must be forwarded to Shards, and stop writing data to legacy database. Advices from Tumbr:

2.1.3.2 how to add a new Shard?

The other definition of adding new shard is splitting a shard. In other words, specially in range-based Sharding, one of the shards may reach your data threshold and you want to split this shard into two shards. According to Tumblr database architecture, 2/3 of the shard disk is currently full, and normally you have two standby slaves per shard pool. During the shard splitting, the table schema is not allowed to be touched. Necessary steps to achieve this goal is described as follows:

1. Create N new slaves in parent shard pool; these will soon become masters of their own shard pools. See figures 2.1 and 2.2. All Reads/Writes operation are still being forwarded to the Master node. This has to be as fast as possible; so you should utilise an efficient file copy approach for mirror slave creations.

[2]This ID has nothing to do with Shard key, and it has to be unique and incremental.
[3]Create Update Delete

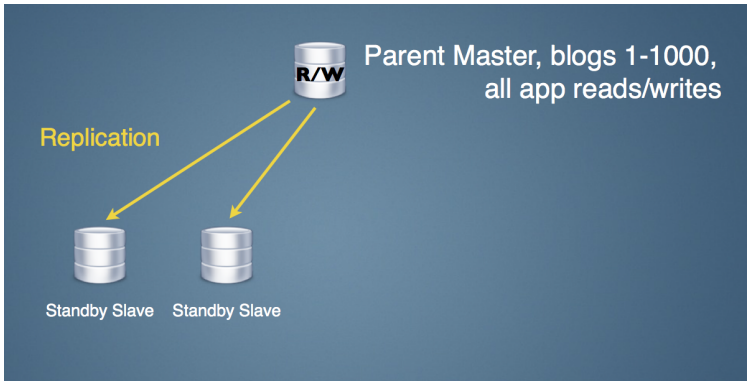


Figure 2.1: Tumblr: A shard with its two slaves. [11]

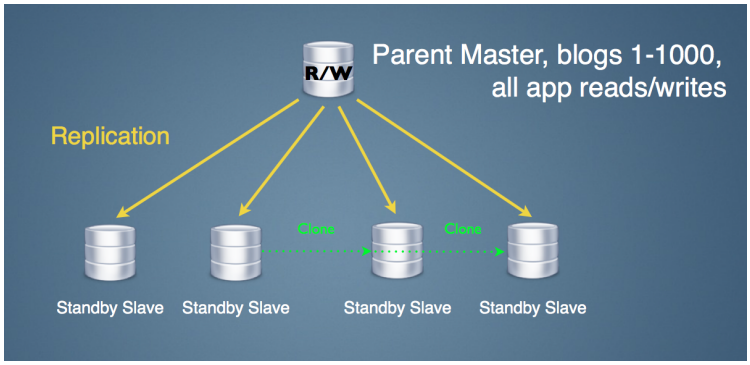


Figure 2.2: Tumblr: Cloning another two slaves. [11]

2. Reduce the data set on those slaves so that each contains a different subset of the data. This step can be done by exporting/importing the table with different ranges. This, like previous step, has to be performed as fast as possible by taking various measures into account, such as disabling binary logging, benchmarking that your hardware can support which concurrency level for import/export queries. And according to Tumblr strategy, every shard needs to have two slaves. Then, spin up two slaves for every proposed shard. See figures 2.3, 2.4 and 2.5

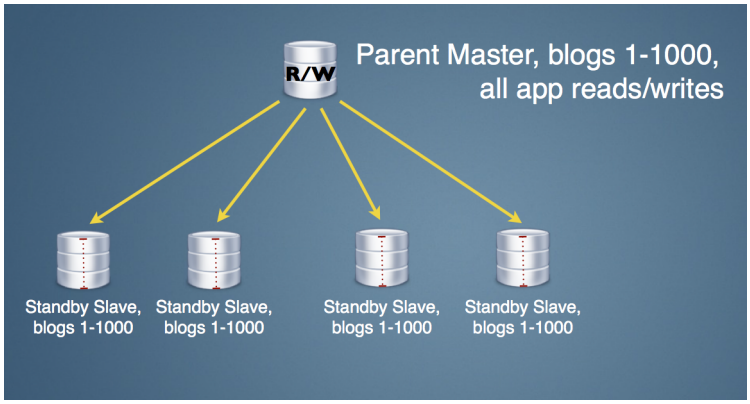


Figure 2.3: Tumblr: A shard with its four slaves (two recently added). [11]
22

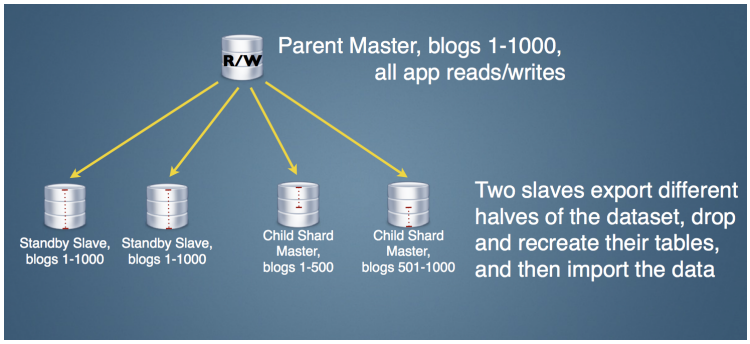


Figure 2.4: Tumblr: Two recently added slaves with reduced data. [11]

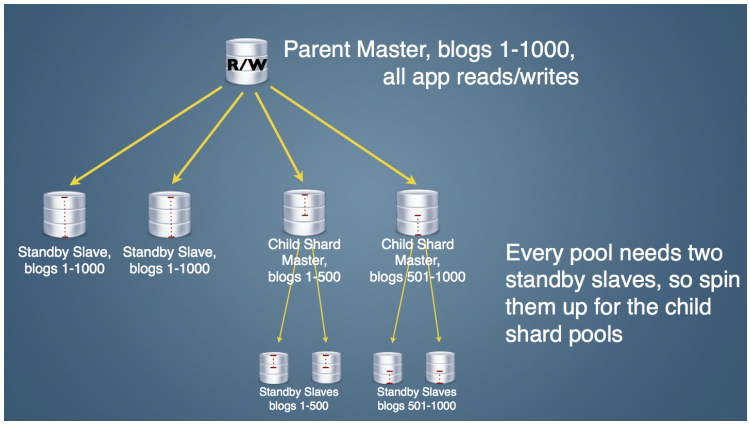


Figure 2.5: Tumblr: Adding two slaves for future shards. [11]

3. Move app Reads from the parent to the appropriate children. Since we have a distributed application, otherwise it is not possible for web/app servers to

receive new configurations at the same time, we might face some inconsistency issues due to stale configurations on some nodes. So it is not appropriate to move Reads and Writes the same time to the shards. When each configuration update is received, Write requests then can be forwarded to the child-shards. See figure 2.6 (i.e. but still note that writes are being replicated to the sub-shards by parent-shard assuming that they are his slaves without knowledge of their smaller range responsibilities. It means that sub-shard may receive and persist out-of-the-range rows. We will take care of those rows later on.).

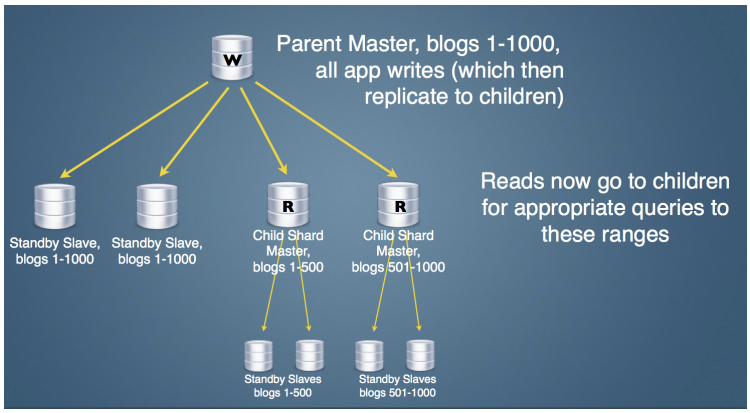


Figure 2.6: Tumblr: Move Reads to the sub-shards. [11]

4. Move app writes from the parent to the appropriate children. Now, app/web servers have confirmed their latest configuration updates, and we can hand over the Writes responsibility to the sub-shards. See figure 2.7.

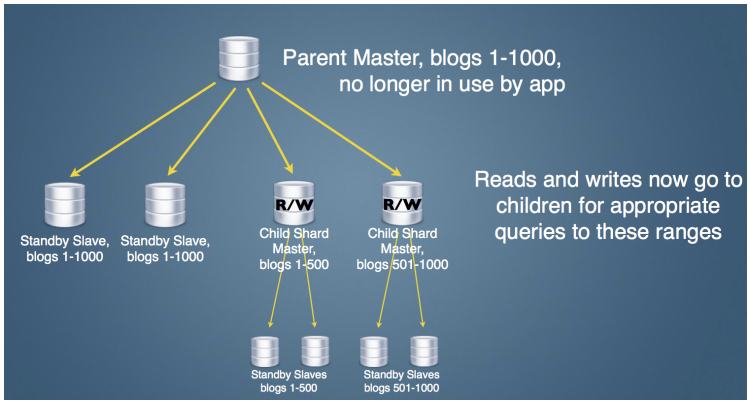


Figure 2.7: Tumblr: Move Writes to the sub-shards. [11]

5. Stop replicating writes from the parent; take the parent pool offline. Now the parent node is no longer in use. So we can drop it along with its two replicas.

See figure 2.8.

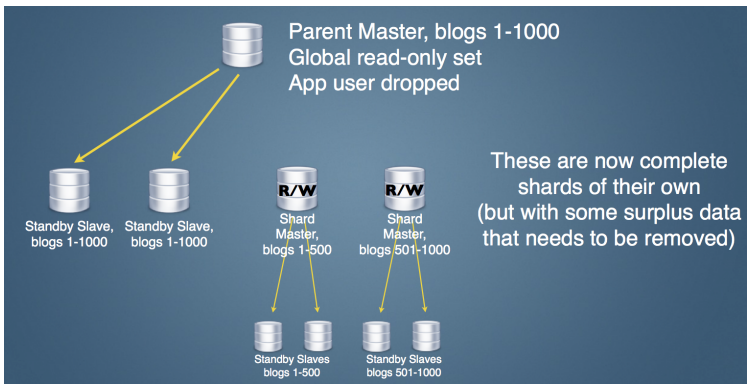


Figure 2.8: Tumblr: Drop the parent Shard along with its slaves. [11]

6. A simple process should go through these two new shards, and remove and clean up those wrong replicated rows, mentioned in step 3, based on their blog ID's.

2.2 Case study: Sharding using MongoDB

MogoDB is a document-oriented open-source database. Documents are somehow similar to JSON objects. In this section, it is assumed that the reader has fundamental knowledge about MongoDB in specific, and Document databases in general. MongoDB in sharded architecture comprises three main actors: *Shards (Mongod)*, *Query Routers (Mongos)*, and *Config Servers*. Most of the information in this section regarding Sharded MongoDB cluster are derived from MongoDB documentation [12] of release 2.6.3.

Mongos (Mongod). This node is a shard which contains part of our data. Besides, it is a replica set, which is almost similar to Master/Slave approach. There is one Primary node responsible for Write requests and Secondary members for Read operations. If Primary nodes fails, one of Secondary nodes can become Primary.

Query Routers (Mongos). This node is like a routing module or application. User queries are forwarded to the proper shard by this node. In addition, Mongos also takes the responsibility of splitting the insertion data which exceeds maximum chunk size into smaller chunks of data (i.e. Could be also done manually).

Through this way, data can be distributed evenly among shards. Chunks are kind of logical concept in the system. If you go through one of these shards (mon-

god) machines, you can only see collections of data (i.e. a collection of data is a big set of documents.) that every portion of these collections may belong to a chunk.

User thinks that Mongos is a single database machine and Mongos thinks that a shard is a single database machine. We can have multiple Mongos as well as having replicas for every shards (i.e. for every shard we can have primary, secondary, arbiter, etc.).

Config Servers. These machines hold all meta data about the cluster and keep the mapping information clusters data set to the shards. See figure 2.9. Every Sharded cluster has exactly three Config servers.

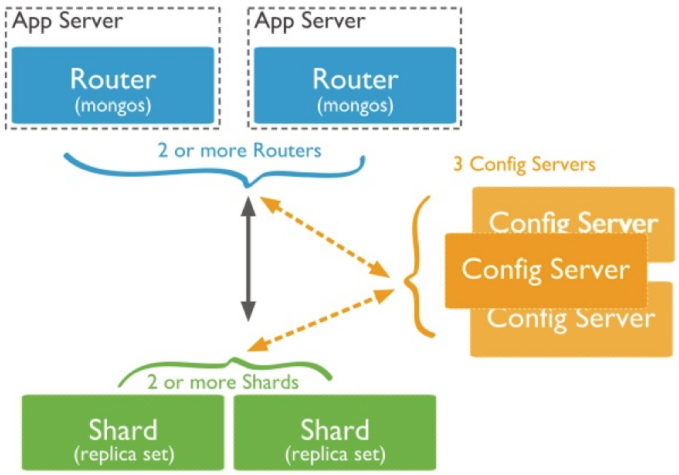


Figure 2.9: MongoDB: Shareded Database in MongoDB. [12]

2.2.1 Data Partitioning

In order to distribute the data across shards, we need to have a shard key. A shard key can be an index field or index compound field that exists in every document of a collection. MongoDB provides two methods of data partitioning into chunks for the purpose of even data distribution across shards: *Range based partitioning*, and *hash based partitioning*.

Range based partitioning. Dividing a data set into chunks determined using the shard keys, such as numeric range partitioning. There is no overlapping between chunks. Pros: decent for range based queries. Cons: high probability of uneven data distribution. See figure 2.10.

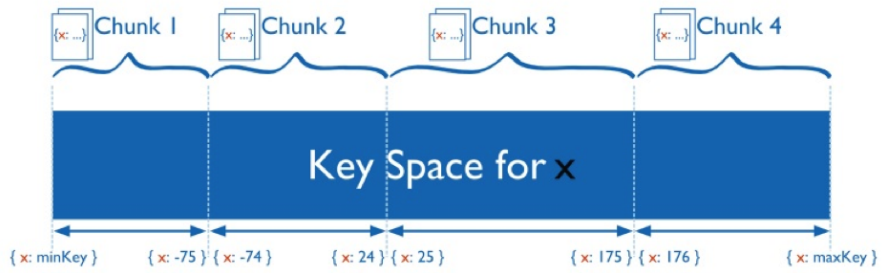


Figure 2.10: MongoDB: Range based partitioning. [12]

Hash based partitioning. MongoDB computes the hash of a field's value and then uses these hashes to create chunks. Therefore, two documents with close shard keys are more likely to be far from each other. In this method, we have more even data distribution among shards. Pros: even data distribution. Cons: less efficient for range queries. See figure 2.11. In case of range queries, it is likely to query all shards to respond just one Read operation.

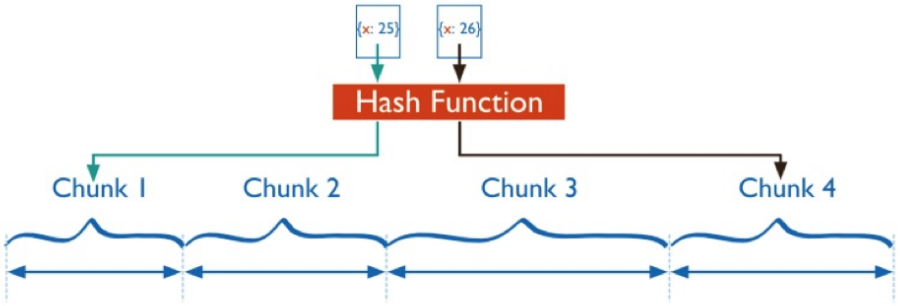


Figure 2.11: MongoDB: Hash based partitioning. [12]

2.2.2 Maintain a balanced data distribution

Since nowadays the amount of data in enterprises is growing rapidly, adding new servers or probability of having an uneven shard is unavoidable. MongoDB with two background process tries to handle this issue by re-balancing the data among shards: *Splitting*, and *Balancing*.

Splitting. When the amount of data in a chunk within a shard exceeds predefined limits, the chuck is divided in two halves. But, all happens locally in one shard. See figure 2.12.

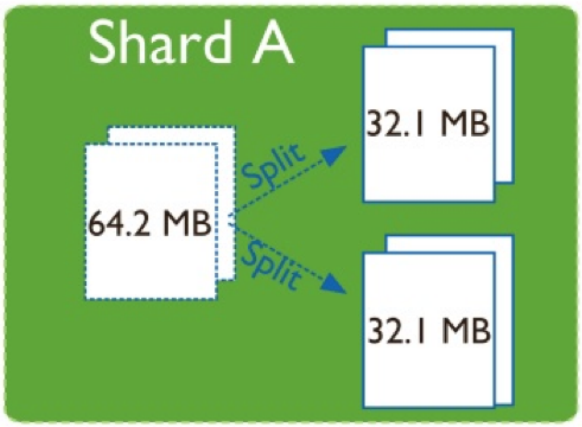


Figure 2.12: MongoDB: Splitting process within one shard. [12]

Balancing. This background process is always running in all query routers (Mongos). The task of this process is to migrate chunks from a shard containing largest number of chunks to the one with least number of chunks. Firstly, data starts moving from original shard to the destination shard. During this journey, all coming updates regarding this data are being stored on the original shard. As soon as the migration is completed, destination shard receives all updates, and apply them on the data. Then, all metadata which is located in config servers are being updated by destination shard. At this point, MongoDB removes all pre-migrated data which is still located in the origin shard. In case of any failure in data migration, Balancer process aborts the everything.

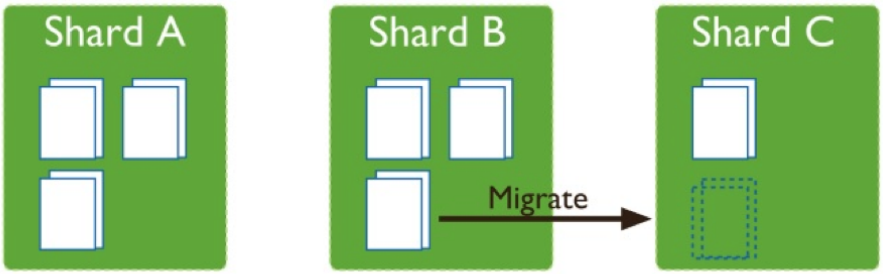


Figure 2.13: MongoDB: Balancing process. [12]

By adding new shard to the cluster, there will be definite balancing data migration. Because, the new shard is holding no chunks and balancing process comes into play as above.

CHAPTER 3

Proposed approach

In this chapter we introduce a decision-based approach for an automatic estimation/suggestion of Sharding scheme with appropriate Shard keys based on SQL query logs.

3.1 Shard Keys

3.1.1 Considerations

Cardinality/Granularity. The chosen field as a Shard key should have enough different values in order to enable the Shard selector to distribute the data easily across the cluster. Entity ID's, timestamps, phone numbers and these kinds of fields might be the options for Shard keys due to their high cardinality.

In all queries. The selected Shard key should exist in all/most queries. Because of the fact that Shard selectors are obliged to forward the queries to an appropriate Shard holding the data, this field should appear in most queries, which introduces Query Isolation.

Compound Shard Keys. Sometimes, it is not easy to find a field repeating in all queries or with very low cardinality if exists. In these cases, one should select several keys as compound Shard Key.

According to [12], The most performant queries in Sharded databases are those which are forwarded to just a shard. In most of the databases, if a query does not contain the shard key, they forward the query to all shards and wait for response from any of them. It is better to use 'compound key'; because if only one of the

fields among compound keys exists in our query, shard manager may query smaller range of shard instances to fetch the data.

The *Primary Keys* are potential choice for Shard Keys. These keys have very high cardinality due to their uniqueness. Sometimes these keys are random which escalate the difficulties for query routers in Shard selection process.

Foreign Keys. These types of keys should be taken into consideration when one wants to choose an appropriate Shard Key. By monitoring the SQL queries in a database, sometimes some queries come with *WHERE* clauses, which may trigger the database to create joined tables. Since target rows might be resided on different machines in a very large Sharded clusters, even a simple Join forces the Query router to send sub-queries to several shards to collect data and join this data from various nodes and returns it to the query initiator, which introduces latency due to current load of the cluster, network traffic, unnecessary data movements. It is more convenient to detect mostly used Join queries by learning process on users' queries and select the Shard Keys with respect to Foreign Keys. In other words, one should desing the Sharded cluster and select widely-used Foreign Keys as shard keys. This may guarantee that rows with same primary key values and with same Foreign key values are stored in one machine, which limits the workload of result preparation of a repeated Join query to a single shard, not several shards.

3.1.2 Sharding Schemes

[1]

3.1.2.1 Range-based Sharding

This scheme is easy to implement and deploy; but maintenance is not easy to do. Because, it is likely to have uneven data distribution across shards, so called *Data Skew*, and this has to be rebalanced which is sometimes not easy.

Choosing an appropriate Shard key in Range-based Sharding is also not trivial. For instance, if a timestamp field is selected for the Shard Key of a table, it may introduce *Write Scalability* issues in some write-intensive scenarios. Because the query routers are oblige to forward the Write operations to the partition holding the latest updates, and this database instance might get easily overloaded by dozens of DML[2] operations while other instances are responsible for older data. (e.g. in case of applications such as *Twitter*, those old instances may not even be read by users in due course.).

Any increasing-value field as a Shard key may create such problems. These shards should be continuously monitored with appropriate Administrator notifications in order to let them split the shard properly and on time.

[1] These Sharding/Partitioning schemes are explained in chapter 1.

[2] Data Manipulation Language

3.1.2.2 Hash-based Sharding

This approach promises that the Sharded cluster ends up with evenly distributed data across all partitions, which eliminates the data rebalancing. But, this Sharding scheme is prone to considerable performance issues in case of scenarios with numerous *Range* queries. Because, Shards are determined by hashed value of Shard Keys, and the probability of having two contiguous row on the same machine is very low. Query router therefore is obliged to query several shards for a very simple Range query, which is violating Query Isolation measures. Furthermore, it is not simple to use compound keys in such schemes.

3.2 Automatic Selection

Initially, we have a big set of given SQL queries noted with Q along with database schema information including relations (tables), attributes (columns) with data types, and primary keys. There should be a learning tool responsible for reading queries one by one and constructing our information vector. We have following data listed in table 3.1.

Table 3.1: Available data about the database.

	Available Information
Query log	$Q_1 \dots Q_n$
Database schema	Database relations (tables), Foreign Keys, Primary Keys, Column Data Types
Input	Some constant values as input for our algorithm.

3.2.1 Cost Vector

While the learning tool is processing all queries, it constructs a *Cost vector* holding some information about the queries. As matter of fact that numerous forms of SQL queries are being used in applications, we just focus on very simple queries without JOIN's and other functions like GROUP BY, COUNT, etc. For simplification, we consider small queries with simple WHERE clauses as follows.

```
SELECT column_name, ...
FROM table_name
WHERE column_name operator value operator ...
```

Every typical relational database contains several tables that we call them R[3], and each R is composed of multiple attributes[4]. The cost vector is based on the

[3]Relation
[4]Columns

occurrence of attributes in the Where clause.[5]

$$\begin{cases} R_i: \text{Relation i} \\ n(R_i): \text{Number of rows stored in Relation i.} \\ A_{i,j}: \text{Attribute j of Relation i.} \\ n(A_{i,j}): \text{Number of occurence of } \mathrm{A_{i,j}} \text{ in queries.} \\ dv(A_{i,j}): \text{Number of distinct values of } \mathrm{A_{i,j}} \text{ as operand in Where clauses.} \\ op(A_{i,j}, operator): \text{Number of occurence of } \mathrm{A_{i,j}} \text{ as operand with a particular operator.} \\ n_{update}(A_{i,j}): \text{Number of occurence of } \mathrm{A_{i,j}} \text{ as operand in Update queries.} \\ pr(A_{i,j}, A_{i,m}): \text{Number of occurence of } \mathrm{A_{i,j}} \text{ along with } \mathrm{A_{i,m}} \text{ in a predicate.} \end{cases}$$

$\text{cost}(A_{i,j}) = (\text{n}(A_{i,j}), \text{dv}(A_{i,j}), \text{op}(A_{i,j}, =), \text{op}(A_{i,j}, \textit{range}), \text{n}_{\text{update}}(A_{i,j}), \text{pr}(A_{i,j}, A_{i,1}), \ldots, \text{pr}(A_{i,j}, A_{i,m}))$

We have several type of operators in SQL queries which are common among most vendors, we only consider operators like (in)equality, range (i.e. BETWEEN, >, <, >=, <=) in this state of literature.

Excluding large attributes. Since we know the attributes data types, we can exclude some of them from the calculation of cost vectors. Some attributes with types such as Text, (Var)char(n>100), Blob, etc. that are too large should not be considered in the algorithm. Because those attributes hold data which are in appropriate for Shard key selection.

Excluding reference tables. Reference tables are those type of table holding small number of rows and are necessary for lots of queries. These tables are being updated not so often. If the Sharding vendor supports having reference tables, these tables should be replicated completely to all Shards. We will propose a simple automatic approach to detect these tables and exclude them from cost vector calculations.

3.2.2 Sharding Schema

We should introduce two important challenges in distributed databases with Partitioning schemes: *Query Isolation*, and *Write Scalability*.

Query Isolation. Shard coordinators are responsible for forwarding the queries to the appropriate Shard(s) based on Shard keys in order to fetch the data and send it back to users who are the query initiators. If the Shard key is not present in the query, the coordinator is obliged to send the query to all or some of the Shards for finding the data. But, queries containing Shard keys are more efficient due to the

[5]In the cost vector, we only care about the occurrence of attributes and their values in Where clauses.

fact that they are sent to less number of Shards.

Write Scalability. As discussed before, Write scalability is about the ability of the coordinator to forward Write queries to various machines. For example, if the Sharding scheme is range-based and timestamps as our Shard key, all Write queries are pointing the last shard holding the recent data, in which we may end up with various difficulties with that Shard such as capacity shortage, IO latency, processing overload. It is therefore better to distribute our Write queries to the whole cluster.

To recap, we can somehow infer that one may have better Query Isolation while using *range-based sharding*, and better Write Scalability while using *hash-based Sharding*. These two concepts are the main fundamentals of our following calculations.

In this part we want to estimate that which Sharding scheme is more appropriate for our database based on existing query logs: Range-based or Hash-based. Firstly, we want to figure out from query logs that our application is whether *Write-oriented*, *Read-oriented*, or *Balanced*.

In order to achieve this goal, we divide all queries in two types: *Read* and *Write*. Read queries are assumed to be SELECT queries, and consequently Write queries comprises UPDATE, INSERT and DELETE. So when our tool is running and reading the queries, it should count the number of occurrence of write's and read's. Eventually, we will have following information.

$$\begin{cases} n(write): \text{Number of Write queries.} \\ n(read): \text{Number of Read queries.} \end{cases}$$

The value of following fraction will be useful in rest of the section for Sharding scheme suggestion approach.

$$r_{write} = \frac{n(write)}{n(read)} \tag{3.1}$$

This is the initial collected information in our algorithm for auto-suggestion of Sharding scheme. If the query logs are produced over a long period of time, we may realize that whether the given application is write-oriented or read-oriented. If we have a very high rate of writes, the Sharding architecture should definitely respect Write scalability measures in order to avoid hammering any particular machine with probable massive amount of request, which may prevent unbalanced shards, IO overhead, and slow response time. In other words, if the rate of writes is considerably larger than read rate, *Hash-based* sharding seems to be a decent choice for this case. Because, the Shard coordinator determine the Shard machine based on hashed Shard key(s) of each write query, which is promised to be a unique and random value.

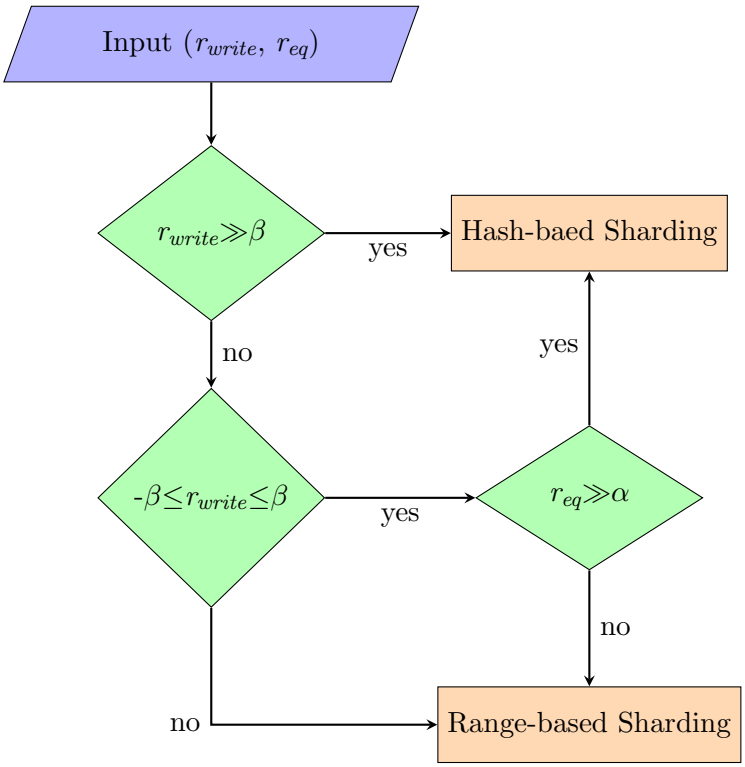
On the other hand, if the read rate is much higher than the other rate, the application is read-oriented. The Query Isolation measures therefore are clearly significant. In contrast to Write-oriented, it is not an easy decision to choose any Sharding scheme at this point in this case. In addition, there are applications with relatively close rates of writes and read that we call them Balanced query patterns. Other metrics are going to be introduced to leverage the idea to get closer to a decent estimation. Knowing the fact that we have p relations, and m_i attributes for each R_i, we can calculate following values.

$$\begin{cases} \sum_{i=1}^{p} \sum_{j=1}^{m_i} op(A_{i,j}, =): \text{ Sum of all occurences of all attributes appearing in (in)equality predicates for the whole database.} \\ \sum_{i=1}^{p} \sum_{j=1}^{m_i} op(A_{i,j}, range): \text{ Sum of all occurences of all attributes appearing in range predicates for the whole database.} \end{cases}$$

Then we can form another rate or fraction by division of these two values that will be useful in a short while.

$$r_{eq} = \frac{\sum_{i=1}^{p} \sum_{j=1}^{m_i} op(A_{i,j}, =)}{\sum_{i=1}^{p} \sum_{j=1}^{m_i} op(A_{i,j}, range)} \tag{3.2}$$

With respect to r_{write} and r_{eq} calculated in 3.1 and 3.2, we are able draw following conclusion.



According to the flowchart, α and β are some constants given by the database

architect to be used as thresholds.

3.2.3 Shard Key

As discussed in previous section, we have calculated the cost vectors, and the suggested Sharding scheme has been decided by far. Now it is time to nominate one or several attributes per relation as our Shard key(s). Firstly, challenges and features of future Shard keys should be considered.

Reference tables. As explained earlier in this chapters, some database vendors support unpartitioned tables along with Sharded tables. In other words, these tables are replicated to all nodes, and are accessible to all database instances locally. These relations should contains small quantity of rows with low rate of Write queries. Because, every CUD[6] operation has to be replicated and synchronised to all machines, which may introduce either inconsistency or latency. These relations have to be waived from our cost vector calculations as well as Shard keys determination decision tree. One may select the reference tables manually before the execution of the learning tool. So if the $r_{read}(R_i)$ is much higher than $r_{wite}(R_i)$, this table can be a candidate to be treated as reference table. The ϵ is a constant value defined by the database architect.

$$\begin{cases} n(R_i) < \epsilon\text{: small number of rows} \\ n_{read}(R_i) \gg n_{write}(R_i) \end{cases}$$

Particular value of an attribute is dramatically getting requested. Sometimes some values are getting queried in Where predicate of SQL operations more than others. For instance, if we have a social network application, some celebrities are target for this type of situations. If we Shard our data based on user ID's, the Shard responsible for those celebrities might get hammered by dozens of queries. In order to cope with Write queries, it is suggested to use *Compound Shard Keys* to split the data in smaller partitions, not only with the user ID. The other workaround in case of massive number of Read queries is to use advanced caching mechanism in database or application level in order to avoid unnecessary IO operations.

Predicates consisting of several sub-predicates. Predicates are expressions that evaluates to a *boolean* value, which are commonly being used in Where clauses. We have formed our Cost vectors based on occurrence of attributes in predicates of queries. If a particular attribute frequently appears with another attribute(s) in the predicates of queries, *Compound Shard keys* should be taken into consideration. Because, compound keys let the query coordinator ask less number of Shards for data retrieval due to absence of some of the keys, which has positive impact on Query Isolation measures.

The value of the existing Shard keys should not get updated. Once a row is inserted to a Shard using its Shard key, the value of the attribute playing

[6]Create Update Delete

the Shard key role should not be touched. This update might be very expensive if it occurs frequently. Because, the row has to be fetched and removed from the shard, and the new shard must be coordinated again using the new value of the Shard key. Meaning that we may perform several IO operations.

High cardinality of Shard key values. A proposed Shard keys should have enough distinct values, so called cardinality. The *Hash-based Sharding* scheme requires the value cardinality as high as possible to enable the query coordinator for even distribution of the rows among machines.

Frequent occurrence of Shard keys in Where predicates with range operators. If the candidate Shard keys frequently appears in Where predicate of operations with range operators (i.e. BETWEEN, >, <, >=, <=), they might get nominated for the final selected Shard key in *Range-based* Sharding scheme. In other words, given the fact that we are going to use range-based Sharding, this feature should be treated as a positive point for our estimation algorithm and decision tree.

3.2.3.1 Decision Algorithm

In this section we want to select some candidate Shard keys for each table based on the calculated cost vectors. The *cost($A_{i,j}$)* stands for the vector of values for the attributes A_j of the relation R_i. We need to have some cost values formed by these vectors of values with respect to the necessary requirements of our Sharding Schemes.

Hash-based Sharding. If the Sharding Scheme is selected to be Hash-based method, we need to meet some requirements for calculation of the cost value assigned to every attributes of each relation. We call it $X_{i,j}$, meaning the cost value of attribute A_j of the relation R_i. The formula 3.2.3.1 describes this value.

$$X_{i,j} = n(A_{i,j}) + \alpha dv(A_{i,j}) + \underbrace{\beta op(A_{i,j}, =) - \gamma op(A_{i,j}, range)}_{\text{R}_{\text{eq}} \text{ is important}} - \overbrace{\mu n_{update}(A_{i,j})}^{\text{negative point}} \quad (3.3)$$

As illustrated in 3.3, α, γ and μ are constant values representing the importance of the particular metric in cost value calculation from database architect perspective. The *n($A_{i,j}$) + α dv($A_{i,j}$)* shows the quantity of the repetition of the attribute within predicates as well as the level of distinct values. Since we are considering the Hash-based scheme, range queries will be treated as negative point in our estimations; in contrast, equality predicates along with distinct values are positive points.

The highest value of X is important factor in our decision tree for selecting the candidate Shard key(s). But what if that all $X_{i,j}$ are high and close to each other? How can we realize the statistical distance between these values? We propose to use *Variance* of these values. Formally, the Variance shows that how far a set of numbers is spread out. In other words, if the *Var(X)* is quite large, it means that the

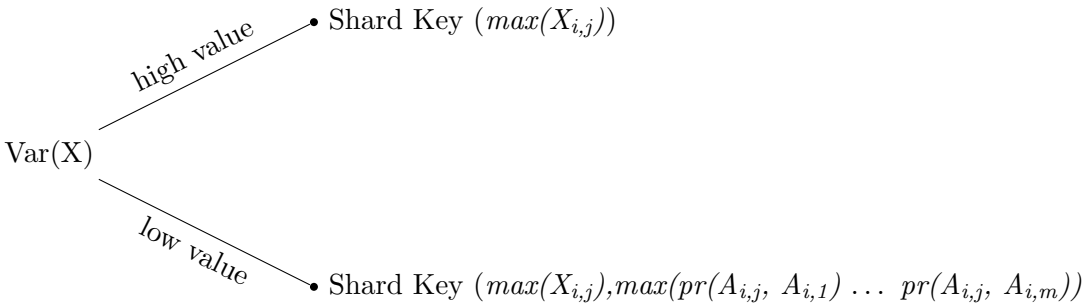
$X_{i,j}$ values are not close to each other and the maximum cost value among attributes is going to promise that its attribute could be the candidate of the Shard key. But, if the *Var(X)* holds a small value, it means that the $X_{i,j}$ values are close to each other, which leads us to think about *Compound Shard keys*. In this case, we just select two keys for simplifications; but in production you choose more.

$$\begin{cases} (X_{i,1} \dots X_{i,j} \dots X_{i,m}) \\ Var(X_i) : \text{Variance of cost values for the relation i.} \end{cases}$$

The following notation , *max($X_{i,j}$)*, returns the attributes owning the maximum of all cost values of its relation. The explanation is as follows.

$$\begin{cases} A_{i,k} \leftarrow max(X_{i,j}) \\ \text{The cost value of attribute k from relateion i} \\ \text{has the highest cost value among others.} \end{cases}$$

The decision tree for the selection of Shard keys with Hash-based scheme is as follows. *max($X_{i,j}$)* means that the attribute which has the highest $X_{i,j}$. If the *Var(X)* has a low value, we choose the mostly repeated attribute within predicates along with our first Shard Key.

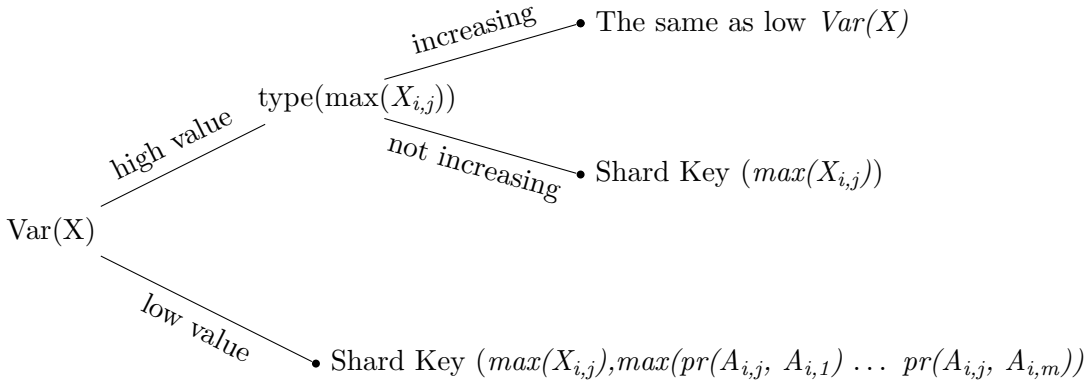


Range-based Sharding. If this approach is selected for our Sharding scheme, we need to additionally take range queries into account. In order to achieve this goal, *op($A_{i,j}$, range)* values within cost vectors should be counted as positive point. Then, we can form our cost values for all attributes of each relation as follows:

$$X_{i,j} = n(A_{i,j}) + \alpha dv(A_{i,j}) + \underbrace{\beta op(A_{i,j}, =) + \gamma op(A_{i,j}, range)}_{\text{range-based predicates are also important}} - \overbrace{\mu n_{update}(A_{i,j})}^{\text{negative point}} \quad (3.4)$$

As illustrated in 3.4, α, γ and μ are constant values representing the importance of the particular metric in cost value calculation from database architect perspective, which may be different from selected values for cost value calculation of Hash-based approach. Again, the Shard key value should not get updated once it is stored on of the Shards since this operation is too expensive. Therefore, as shown, it should be counted as negative point in our cost value.

Monotonically increasing attributes. As matter of fact that we initially know the data type of all attributes of each relation, we know that which column value is going to get increased over the time. This may cause some trouble if those attributes is selected as our Shard key. For example, if that column chosen as our Shard key is of a timestamp data type, we will be inserting new rows on a Shard responsible for recent data. This may cause latency due to IO limitations. In order to cope with this issue, our proposed workaround is again to use *Compound Shard Keys* to be sure that all new rows will not reside on a particular node all the time. We should use these metrics in our decision tree.



Again, we are using *Var(X)* in order to see how the set of cost values are spread out. As illustrated in the decision tree, we limit our Shard key selection to two keys for simplification.

CHAPTER 4

Conclusion

We studied most challenges of Database Partitioning in general, Sharding in specific. Furthermore, the customised approaches of two successful vendors have been reviewed. As matter of fact that choosing a set of decent Shard keys in the beginning stage of database designing is unavoidably significant for the future performance of the whole application, we introduced a way of auto-detecting the candidate Shard Keys. That was done by considering query logs and forming the *cost vectors* holding information about attributes of all relations within predicates. Finally, with a decision-tree based approach, we presented candidate single or compound Shard keys. In addition, we proposed a relatively same approach for detection of appropriate Sharding scheme, either Range-based or Hash-based one.

4.1 Future Work

As explained, our proposed approach is based on small and simple range of SQL queries. We should leverage our strategy for consideration of other SQL operations containing other concepts. In addition, read operations in databases have various processing requirements and also the amount of returning data is different from each other, which have influence on the performance of the Shards. The measures should be added to the algorithm to have better outcome.

Eventually, a learning tool should be developed in order to analyze the result of this paper in practice. That could help us to improve the decision tree for better estimations.

Appendices

Bibliography

[1] Y. Stephen "NoSQL is a Horseless Carriage," *NorthScale*, Retrieved 2014-06-26.

[2] B. Scofield "NoSQL - Death to Relational Databases," *SlidesShare*, 2010.

[3] M. Stonebraker "SQL databases v. NoSQL databases," *Communications of the ACM*, 2010.

[4] Encyclopedia "http://www.pcmag.com/encyclopedia/term/65630/database-partitioning," *PC Magazine*, 2010.

[5] S. Navathe, S. Ceri, G. Wiederhold, J. Dou "Vertical Partitioning Algorithms for Database Design," *ACM Transactions on Database Systems, Stanford University*, 1984.

[6] M. Medic "Database table partitioning in SQL Server," *SQL Shack*, 2014.

[7] Oracle "Database VLDB and Partitioning Guide 11g Release 1 (11.1)," *Oracle documentations*, 2007.

[8] CodeFutures "Database Sharding," *White Paper*, 2014.

[9] D. Obsanjo "Building Scalable Databases: Pros and Cons of Various Database Sharding Schemes," *Obsanjo's technical blog*, 2009.

[10] Tumblr Engieering Blog "http://engineering.tumblr.com," *Tubmlr corporation.*

[11] E. Elias "Tumblr: Massively Sharded MySQL," *Velocity Europe*, 2011.

[12] MongoDB Documentation Project "MongoDB Documentation," *Release 2.6.3*, 2014.