LOCALITY-AWARE DISTRIBUTED STATE PARTITIONING FOR STREAM PROCESSING SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Muhammed Yağmur Şahin
October 2016

Locality-Aware Distributed State Partitioning for Stream Processing Systems
By Muhammed Yağmur Şahin
October 2016

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Buğra Gedik(Advisor)

Can Alkan

Gültekin Kuyzu

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

LOCALITY-AWARE DISTRIBUTED STATE PARTITIONING FOR STREAM PROCESSING SYSTEMS

Muhammed Yağmur Şahin
M.S. in Computer Engineering
Advisor: Buğra Gedik
October 2016

Today, there are many applications that deal with high-volume data streams. These distributed stream processing applications process data on-the-fly and provide real-time distributed computing for big data. Due to the volume of data they process, some of these applications make use of data parallel nodes. The state management for distributed nodes in these applications is an important task to handle, because of different use cases such as: dealing with node failures, checkpointing, data enrichment, and re-partitioning. Therefore, distributed stream processing applications need a state management mechanism. In this thesis, we present a locality-aware state management mechanism for distributed stream processing applications. The proposed mechanism provides a transparent locality-aware data partitioning and state management system for distributed stream processing applications. The mechanism partitions data while preserving locality and handles state transfer among nodes transparently, in order to adapt to potential changes in the partitioning. In addition to this, it provides operators with a high-performance state management facility that can tackle check-pointing scenarios. The idea is implemented as a pluggable library for the open-source, distributed stream-processing engine, Apache Storm.

Keywords: Locality-aware state partitioning, Consistent Hash, Apache Storm.

ÖZET

VERİ KATARI İŞLEME SİSTEMLERİ İÇİN VERİ YERELLİĞİ FARKINDALIĞI OLAN DAĞITIK DURUM BÖLÜMLENDİRMESİ

Muhammed Yağmur Şahin
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Buğra Gedik
Ekim 2016

Günümüzde çok sayıda orta katman uygulaması veri katarlarını işlemede kullanılmaktadır. Bu uygulamaların işlediği verinin boyutu düşünüldüğünde, bu uygulamaların veriyi paralel olarak işleyen düğümlere sahip olması kaçınılmazdır. Bu dağıtık veri katarı işleme uygulamaları veriyi anlık olarak çalışma sırasında işlemektedirler. Diğer bir deyişle bu uygulamalar büyük veri için gerçek zamanlı ve dağıtık işleme imkanı sağlarlar. Bu dağıtık düğümlerin durum yönetimi ile ilgili işlemler, bir düğümun bozukluğu, sağlama noktaları oluşturulması, verinin zenginleştirilmesi ve verinin tekrar bölümlere ayrılması gibi kullanım senaryolarından dolayı ele alınması gereken önemli bir görevdir. Bu tezde, dağıtık veri katarı uygulamaları için saydam, veri yerelliği farkındalığında bir durum yönetimi mekanizması sunulmaktadır. Önerilen mekanizma dağıtık veri katarı uygulamaları için veri yerelliği farkındalığı olan, saydam bir veri bölümlendirme ve durum yönetimi sistemi sağlamaktadır. Mekanizma veriyi, veri yerelliğini koruyarak bölümlendirmekte ve bölümlendirme şemasında gerçekleşebilecek olası değişimlerde, veriyi işleyen düğümler arasında durum verisinin saydam olarak aktarımını sağlamaktadır. Buna ek olarak, mekanizma veri ile ilgili saklama ünitesinde sağlama noktaları oluşturmak uzere yüksek performanslı bir durum yönetimi özelliği de sunmaktadır. Bu fikir, açık kaynaklı, dağıtık veri katarı motoru Apache Storm için bir tak-çıkar kütüphane olarak gerçeklenmiştir.

Anahtar sözcükler: Veri Yerelliği Farkındalığı, Veri Bölümlendirme, Consistent Hash, Apache Storm.

Acknowledgement

First of all, I would like to thank my supervisor, Assoc. Prof. Buğra Gedik for his help, guidance, patience and support throughout my studies. I consider myself to be lucky for working under his supervision and I have learned a lot from him.

I am grateful to my jury members, Asst. Prof. Can Alkan and Asst. Prof. Gültekin Kuyzu, for reading and reviewing this thesis.

I thank to the love of my life, the unyielding support behind all my decisions, Gamze. I love you more than you can imagine.

I thank to my mother Gülsen and my father İbrahim for their endless love and support. I know that my debt to you is beyond measure. I am grateful to you.

I thank to my best friend, my brother, Umurcan for his support, selflessness and joy that keep me strong all the time.

Contents

List of Figures

Chapter 1

Introduction

The data that is produced and processed by computer systems is growing exponentially in the era of the Internet. In many applications, data should be processed in near real-time to provide actionable insights. Examples include analyzing live stock ticker data in financial markets, call detail records in telecommunications, video streams in surveillance, production line status feeds in manufacturing, and vital body signals in health-care, to name a few.

Distributed stream processing systems process data on-the-fly, as it arrives continuously. They provide a paradigm for applying a series of operations on data streams, and are suitable for processing high volume of data with low-latency. When the response time and computational power requirements of current applications are considered, the importance of distributed stream processing systems become even more apparent. The application areas of distributed stream processing are wide as the penetration of the Internet in different fields of life.

A distributed stream processing system can consume data from geological sensors, social media, banking services as well as national security services. As data is consumed and emitted to the stream processing system, a series of operations needed to be applied to the data. A stream can be described as a series of tuples that are continuosly fed into the distributed system. In distributed stream

processing, the data tuples are partitioned using a partitioning key and emitted to the operators that could have replicas as well [1]. These operators are referred to as data parallel operators. Many stream processing applications require data parallel operators to keep state. Therefore, the data parallel operators might be stateful. The states of data parallel operators should be managed by the stream processing middleware or the application developer. The state management is an important task to handle in distributed stream processing applications, whether it is done by the middleware or the application developer.

The following use case explains the state management of a data parallel operator. Consider a distributed stream processing system, which calculates some statistics over the visits of pages in a web site. The data stream for the system consists of tuples containing the page id and the the visit count. The partitioning function partitions the tuples according to their page id and directs the tuples to their corresponding data parallel operators. In such a systems, if a failure occurs for a node that a data parallel operator is working on, then the visit statistics maintained for the page ids it was processing are lost, unless a state management mechanism is used. When the failed node is started to work again, the visit statistics for that page will start from scratch. In this simple example, the visit statistics for the pages would not have been lost, if periodic check pointing to a persistent unit was available for the state that was being stored in the memory of the data parallel operators.

When the data is partitioned and transferred to the data parallel stateful operators, it is exposed to different operations, which may maintain state. In certain scenarios, this state needs to be persisted to disk. Example scenarios include check-pointing to disk or data enrichment from disk [2]. Persistence of data comes with the burden of disk access, which is a costly operation for a real-time system. The issue of locality of the data rises here. In certain cases, the locality of this data can significantly impact the performance of writing to and reading from the storage unit. While developing our state management mechanism, we consider two use cases: partitioned state management with check-pointing, and enrichment using dynamic mapping.

First, the partitioned state management case is about the adaptability of the state management mechanism to the changing nature of the distributed stream processing systems. In these systems, elasticity is an important issue, because the topology of the system could change due to any reason such as failures or workload spiked. Therefore, a re-partition of the data among the nodes can be required, and that causes a change in the partitioning scheme as well [3]. This also leads to state transfer among the data parallel nodes in the distributed system, so the state management mechanism should handle this transparently without affecting performance.

Second, the enrichment using dynamic mapping case is relevant when the streaming data needs to be enriched using lookup data that resides in a database. There are often multiple partitioned stateful operators in a streaming application. These operators typically use a partitioning key to perform enrichment, aggregation, trigger processing, etc. [3]. Since enrichment is a common task, its performance is important. To increase enrichment performance, we need to have each replica of the enrichment operator get the necessary information from the database for itself.

In both of the cases, the problems to be considered are transparent re-partitioning and reducing disk access times. For the sake of transparency in data migration, our mechanism handles re-partitioning on-the-fly and does not rely on disk. The mechanism also reduces disk access cost by utilizing locality of the data when partitioning.

The locality of the data means having similar data points near to each other [4]. When the data is partitioned according to locality, then the data stored or retrieved in each data parallel operator will be close to each other in terms of their keys. Storing data according to locality provides an ability to perform batch stores and retrieves and reduces the disk access cost [4]. Therefore, our mechanism utilizes locality when partitioning the data.

We develop our mechanism as a plug-in for open-source distributed stream processing engine Apache Storm [5]. The mechanism provides Storm a locality-aware

partitioned state management support for its data parallel operators. Due to the characteristics of stream processing systems, the state management mechanism should not negatively impact application performance [6]. To increase transparency and throughput, our mechanism does not involve any disk read/writes when creating or updating the partitioning scheme, i.e. the state transfer is handled on-the-fly. Our plug-in can be easily used by any Apache Storm application. We also developed a demo application using this mechanism for test and demonstration purposes.

The rest of this thesis is organized as follows. Chapter 2 discusses the related work and provides a background on Apache Storm. Chapter 3 introduces the system design and implementation for the locality-aware partitioned state management system for the Apache Storm. Chapter 4 explains the how locality aware partitioner design is developed and the idea behind the development of locality-aware partitioning scheme. Chapter 5 provides analytical results of the evaluation for the locality-aware partitioner for both one-dimensional and two-dimensional data types as well as the locality-aware partitioned state management library for Apache Storm. Finally, Chapter 6 concludes this thesis.

Chapter 2

Background on Apache Storm

Apache Storm is an open source, distributed, real-time computation system for large volumes of streaming data [5]. It is a fast, scalable, fault-tolerant and reliable system for processing real-time, high-velocity data. It is also easy to operate and can be used with any programming language. Its performance was benchmarked at processing one million 100 byte messages per second per node with commodity hardware. It restarts a worker in case it dies or if the node dies Apache Storm assigns workers to a running node to provide fault-tolerance. The incoming tuples are guaranteed to be processed by Apache Storm, and the messages are replayed if any failure occurs.

The features provided by Apache Storm makes it a prominent middleware for distributed stream processing. As a distributed stream processing engine Apache Storm re-defines stream processing concepts in its own terminology as well. The details about this terminology is provided in the following sections.

2.1 Topologies

The topology is the real-time application that is running on Apache Storm. The applications are packaged as topologies in terms of Apache Storm concepts. A

topology runs forever unless it is terminated and it can be described as a graph of Spouts, Bolts, and Stream Groupings.

2.2 Streams

The stream is the abstraction in the Apache Storm terminology that describes the series of tuples that is processed in a parallel and distributed fashion. Tuples can contain any primitive types as well as custom types whose own serializers are defined.

2.3 Spouts

The spouts are the sources of streams in an Apache Storm topology. Spouts provide tuples to the Apache Storm topologies as streams, generally by reading from external sources such as Twitter API [5]. A spout can emit more than one stream, and it could be reliable or unreliable. Reliable spouts guarantee the message processing, in other words, replays the tuples in the case of a failure.

2.4 Bolts

Bolts are main processing units in the Apache Storm. Bolts can handle tasks ranging from filtering, projection, aggregation, joining, talking to databases, and more. Bolts execute many tasks across the cluster. Each task corresponds to one thread of execution, and stream groupings define how to send tuples from one set of tasks to another set of tasks [5]. This provides parallelism in the bolts execution. Apache Storm Bolts can be considered as data parallel operators, since they are processing the data using many tasks in parallel.

2.5 Stream Groupings

Another component of Apache Storm topologies is the stream groupings. Stream groupings specify which bolts will be dealing with which streams and they also define how the tuples of the stream should be partitioned among the tasks of a bolt [5]. There are built-in stream groupings in Apache Storm, but stream groupings could also be extended by implementing the CustomStreamGrouping interface that Apache Storm provides.

2.6 Proposed Solution for Apache Storm

Our proposed solution consists of two building blocks: the locality-aware partitioning and the state management mechanisms.

The partitioning mechanism is made available to any application using Apache Storm's pluggable architecture for grouping mechanisms. The locality-aware partitioning provides grouping of incoming tuples according to their locality and directs those tuples to the relevant operators in the application. The state management mechanism is provided with the help of a standalone application that the Apache Storm operators are communicating with via a custom interface. This interface was developed by extending existing Apache Storm components and can be configured by the application developers.

There are pluggable grouping mechanisms already available for Apache Storn, which manage particular task assignment and priority issues. However, there is no grouping available for locality-aware partitioning, In this work, we are presenting a new pluggable, locality-aware partitioned state management mechanism for Apache Storm. Our service can be easily integrated into any Apache Storm topology, and it can be used transparently without changing the regular course of the application.

Since the distributed stream processing systems handle real-time data, preserving a node's state is crucial. Without this capability, the processing could become disrupted after recovering from failure. The state that needs to be preserved is the data which is stored in the operator memory, typically representing a summary of large amount of tuples that were processed earlier. It is important not to lose this data in case of a failure, because when the system restarts the previous calculation becomes lost. This might cause to misleading analytical results after a restart.

Our solution for state management is designed as a pluggable library for Apache Storm. It can be easily used with existing applications via the provided API. The aim of this mechanism is to augment Apache Storm with a state management service, while at the same time using an underlying locality-aware partitioning mechanism to utilize benefits of data locality for handling disk operations. The details about the proposed mechanism are provided in the next chapter: 'System Design and Implementation'.

Chapter 3

System Design and Implementation

In this section, we explain the details of the designed system. The system is designed to be easily integrated into Apache Storm, and it consists of a locality-aware partitioning and a state management service components.

Our system proposes a locality-aware state management service for data parallel distributed stream processing operators. The system is designed as a bundle for open-source, distributed stream processing engine Apache Storm using its extendible architecture [5]. Our system is designed as an extension for Apache Storm and it provides an additional service for Apache Storm and an API for its users to manage the states of data parallel operators in their topology. We designed our system for easy integration into existing applications and working with different types of data.

While designing the system, the first consideration was to build a transparent system that provides easy integration with new and existing topologies. Transparency of the system means that handling all operations related to locality-aware partitioning and state management without transforming the application

architecture. The system performs the state management operations in the background, and the distributed stream processing topology does not change, and the performance of the whole system is not affected.

Since the system is designed for Apache Storm and Storm provides an extensible interface, the system uses this facility. Related to that point, the system itself also does not lay a burden on application developers with a steep learning curve about the new API and to write many lines of code for state management in their implementation. The locality-aware partitioned state management mechanism's integration to any Apache Storm application can be done in a trivial way. Any operator that wants to use the state management service can simply extend from an interface we provide. And to take advantage of the locality-aware partitioning, the grouping mechanism we provide can be used while constructing a topology. The state management service is going to handle all the related tasks, in a transparent manner. Any additional configurations related to state management service can be made using a configuration file, which locality-aware partitioned state management mechanism uses to make necessary arrangements for the particular cases configured in that file.

The second concern for our system is to support different data types. This issue is important since it is also related to the transparency of the system. The system is designed and implemented using Generics in Java to support different data types flowing through the distributed stream processing application.

3.1 Use of the API

While designing the system, it is primarily considered that current development practices should not be changed, as transparency implies that the use of the API should also be simple to learn and implement. The API provides two main components: one for locality-aware partitioning and another for state management. The state management part has also two main interfaces; one for the data parallel operators, aka Apache Storm Bolts, and the other for the partitioned

state management service. These components constitute our state management mechanism's interfaces.

The use of the API and its interfaces can be explained as follows:

- Locality-Aware Grouping: This component is implemented using Apache Storm's pluggable grouping API. Apache Storm's groping API is utilized to provide locality-aware partitioning of tuples to distribute them to the copies of data parallel operators according to locality. This partitioning regarding locality helps operators to deal with 'local data', which results in better read and write performance on the disk. It is easy to use the API's locality aware grouping via Apache Storm's pluggable groping architecture.
- State Service Bolt: The operators in the stream topology are based on this component. The basic Storm operators (Bolts) are inherited from this parent class, and they become stateful operators whose state is managed by the state management service. This new Bolt type is created via extending the Apache Storm's Bolt interface. State management mechanism features transparent state transfer among data parallel stateful operators and state checkpointing to a storage unit using this new type of Bolts. The necessary methods that are used to provide state management mechanism are implemented under this parent class. This helps the API users to make use of facilities of the API without requiring major changes in their design.
- Partitioned State Management Service: This component resides beyond the Apache Storm applications. This part is developed independently from Apache Storm as a partitioned state management service that enables data parallel operators to transfer their states among them. This component also keeps the states of data parallel operators while they are exchanging their states. This component is designed and implemented as a server application to provide object transfer among different instances of the nodes in the cluster.

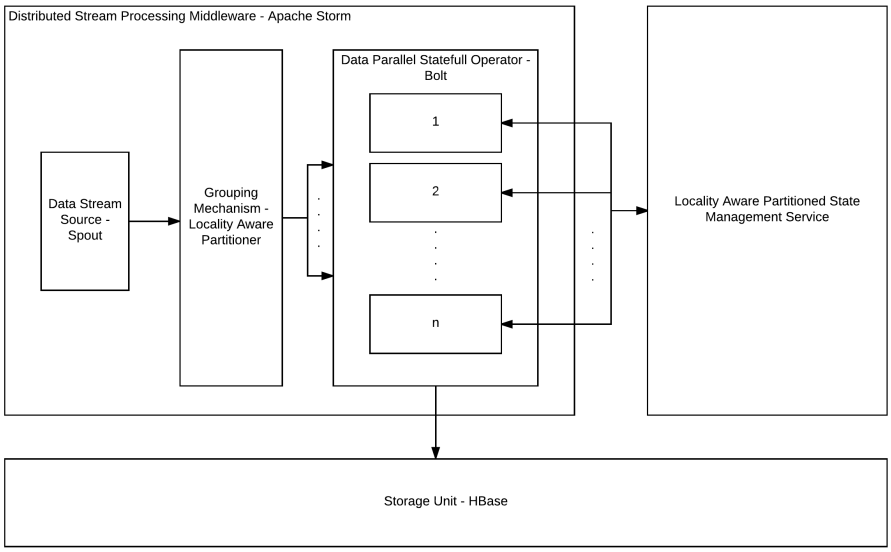

Figure 3.1: The Workflow of the Locality Aware Partitioned State Management Service Architecture

Storm does not enable changing the topology design during run-time. Therefore, the API's facilities should be used while developing the topologies. In other words, locality-aware grouping schemes and operators should be integrated into the application during topology design time.

3.2 Architectural Design

The system is composed of two components. These components are the locality-aware partitioner and the state management service. The partitioner's job is to distribute data to the operators according to locality. State management service, however, is responsible for providing state management functionality for data parallel operators.

In Apache Storm's stream processing topology, the partitioner has the job of grouping mechanism. Since the data is emitted between operators according to predefined grouping schemes in an Apache Storm topology, the partitioner is responsible for grouping the tuples with preserving data locality and directing

them to appropriate copy of the operators.

The partitioner can have different mechanisms while distributing data to the different copies of the operators. In the proposed API, the tuples are planned to be grouped in such a way that the data items that are close to each other will be grouped together. The partitioner, therefore, processes tuples and emits them to the same copy of the data parallel stateful operator with respect to data locality. This approach constitutes the locality-aware partitioning mechanism. The partitioner can be integrated into any Apache Storm topology by using extensible grouping mechanism of Apache Storm. The partitioner designed as a custom grouping scheme and easily integrated into topologies while designing them.

State management service is the component that provides partitioned state management functionality for Apache Storm applications. This service is used to persist, read and re-distribute the state that is kept within the data parallel operators. The service works as follows: After significant amount of data is processed by the topology and data parallel operators have accumulated summary state in their memory using state management services, this state is repartitioned across operators after the partitioning scheme is updated to be locality-aware. State management service is also responsible for writing operator state into a persistent storage space, as well as reading operator state from a persistent storage space. These three features constitute a locality-aware state management service for Apache Storm. The state management service could rely on any database or file system, however, Apache HBase[7] is chosen in this work due to its fast read/write in terms of locality. The workflow of the state management architecture can be seen in Figure 3.1.

The state management service is planned as a separate program rather than as an Apache Storm application. However, an interface is designed and implemented for Storm bolts to make them use the state management service easily. The features of the state management service can be easily taken advantage of by using the provided interface while implementing the Apache Storm applications.

The interface for state management service is extending Apache Storm's Bolt

interface. This new interface is also implemented by a class that can be inherited by Bolts. This class provides the details for exchanging information between the state management service application and the Apache Storm topologies. The topologies that are using the locality-aware partitioned state management service facility use this server application to transfer and migrate state among their data parallel operators.

3.3 System Implementation

The implementation is realized considering the system as a pluggable a library, as an extension for any Apache Storm topology. Therefore, Apache Storm's extensible architecture is utilized, while implementing the library. Apache Storm's extension grouping interface is a direct gate for the locality-aware partitioning mechanism. The locality-aware bolt, however, is implemented as an additional bolt type that should be extended to provide locality-aware state management for Apache Storm topologies. In addition to this, a server is developed in order to help data transfer among copies of the bolts. The partitioning design requires copies of the bolts to transfer data among them to increase locality of the partitioned state that is kept in every single copy of a bolt. This server is used by the new storm bolt interface, so the users of the library would not be burdened with implementing details of the data transfer tasks.

3.3.1 Locality-Aware Grouping Mechanism for Apache Storm

The locality-aware partitioner is implemented as a custom storm grouping mechanism. The idea is to extend Apache Storm to provide locality-aware data partitioning among copies of bolts. This locality-aware data partitioning is obtained by utilizing a new partitioning approach we developed called *histogram hashing*. As any Apache Storm grouping mechanism, this mechanism is directing data

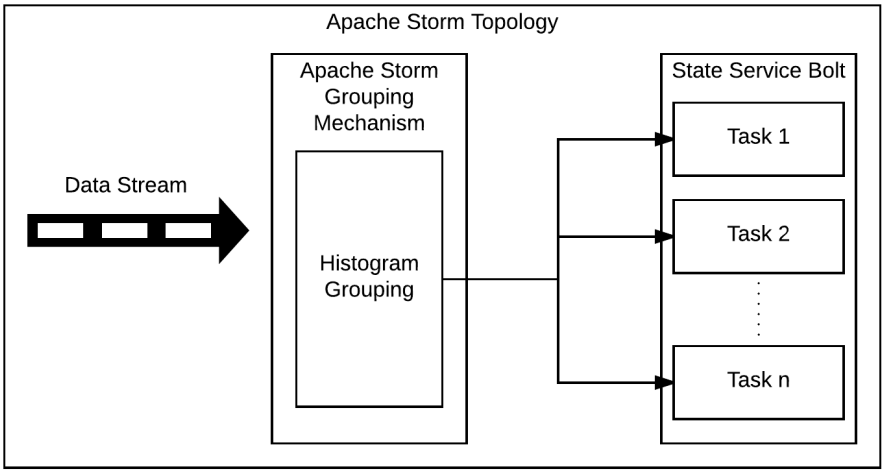

Figure 3.2: Component Diagram Representing Locality-Aware Grouping Mechanism

to copies of bolts. However, this mechanism is also considering locality of data while doing this job. This is achieved using the histogram hash, which is based on consistent hash [8] as its baseline. The locality-aware grouping mechanism is implemented to allow changes of the partitioning of state across copies of storm bolts. This increases locality at any time by changing the partitioning scheme according to the data stream that passes through the topology. The component diagram in Figure 3.2 shows how the custom locality-aware grouping mechanism is implemented and integrated into Apache Storm topologies.

3.3.2 Locality-Aware Stateful Bolt for Apache Storm

A new class of operator is developed for Apache Storm to reduce the overhead of using the library because it would be painful for the users to implement state transfer functionality. Instead, a new Apache Storm interface for Bolts is developed, and any Bolt that inherits from this new class of Bolt gets the state transfer abilities among its copies. The interface also provides each copy a state management service, via communicating with the partitioned state management

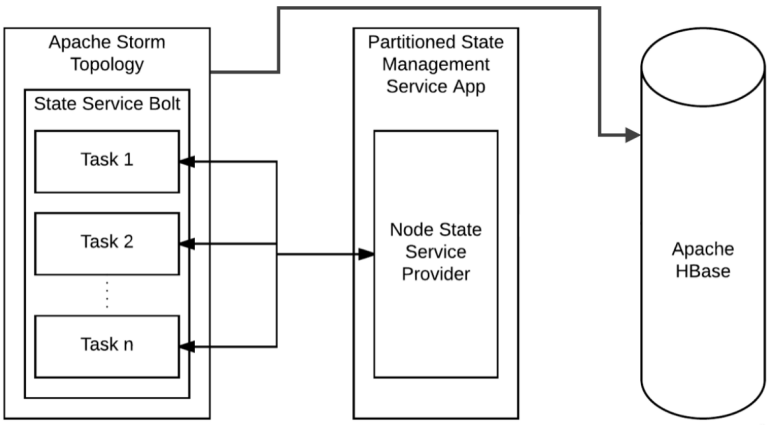

Figure 3.3: Component Diagram Representing Locality-Aware Bolts

service application. The locality-aware bolt interface has all the details implemented such as: networking, data transfer, data filtering before data transfer, persisting the data into storage and reading the data from storage. Therefore, the users of the library should only use the required functions according to their need. The component diagram in Figure 3.3 shows how the locality-aware bolts are designed and integrated into Apache Storm topologies.

3.3.3 Partitioned State Management Service

A server application is implemented for transferring state between copies of Bolts, because there is no interface provided by Apache Storm to transfer data between copies of the same Bolt. Storm architecture allows only the longitudinal movement of data. Therefore, a different mechanism was needed and a server application was implemented to transfer state on-the-fly. The server has a state service, which keeps track of state transfers among copies. The mechanism is working as follows: a copy of the Bolt filters the data in its state service and sends the non-local data to the server where all the non-local data is transferred to state management service of the server. After that, all the non-local data on the state management service of the server is re-distributed among copies of the bolt according to the new data partitioning scheme, which is based on data locality. The server provides this feature without needing to stop the stream, and

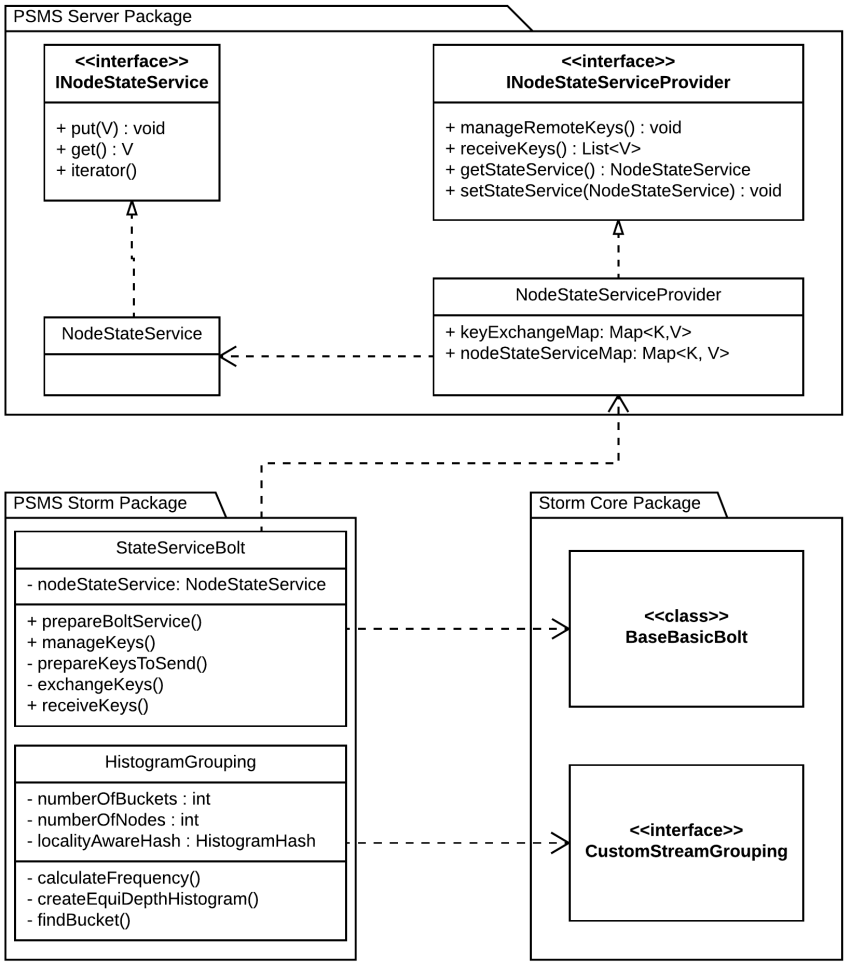

Figure 3.4: Class Diagram Representing Architecture of Locality-Aware Partitioned State Management Service

it handles all the transfer on-the-fly. The diagram given in Figure 3.4 presents the class hierarchy that provides the details about the integration of partitioned state management service into Apache Storm topologies.

Chapter 4

Partitioner Design

Providing a locality-aware data partitioning mechanism is one of the goals of the proposed partitioned state management system. There are other requirements of the partitioning mechanism, such as load balance and migration cost. The load balance is about the balanced distribution of data among the operator replicas. The migration cost is defined as the amount of data that should be migrated in case of a change in the partitioning scheme. These three characteristics are considered for both 1-dimensional and 2-dimensional data sets, while developing the locality-aware partitioning mechanism.

For 1-dimensional data, two alternatives approaches are taken: Consistent Hash and Histogram Hash. Consistent Hash is a hashing mechanism from the literature, known for its performance in terms of load balance and migration. Histogram Hash is a new approach that we developed, which performs well in terms of locality, load balance, and migration cost. Histogram Hash also uses Consistent Hash in the background.

Hilbert-curves and k-d trees are used for partitioning 2-dimensional data. Hilbert-curves are used to obtain a 1-dimensional representation of the 2-dimensional data to apply techniques developed for the 1-dimensional data to

the 2-dimensional data. The k-d tree is used to partition data directly in 2-dimensional space, by specializing the basic k-d tree implementation to provide locality-aware partitioning, while preserving load balance among the operator replicas.

4.1 1-Dimensional Case

4.1.1 Consistent Hash

Typical hashing based schemes spread the load through a known, fixed collection of servers [9, 8]. Cloud systems, however, do not have a fixed collection of machines. Instead, machines become up and down as they crash or are brought into the network. For most of the hashing mechanisms, in the case of a re-size, nearly all of the keys need to be remapped. In other words, most of the keys need to be migrated when the size of the hash table is changed. The power of Consistent Hash resides here. It is a particular hashing mechanism such that the number of keys to be remapped is K/n, where K denotes the number of keys and n denotes the number of slots, on average when the hash needs to be re-sized [9, 8]. This enables consistent hashing to perform well when the hash table is resized. In addition to this point, balanced distribution of load is another important property of Consistent Hash.

A consistent hash is composed of partitions over a keyspace. When a new node joins the cluster, it picks a random number from the keyspace, and that number forms a partition that represents the data it is going to handle. To provide load balance, Consistent Hash assigns multiple partitions to each node.

4.1.2 Histogram Hash

Consistent Hash is a good partitioning mechanism when considering load balance and migration cost. However, it is not locality-aware. In theory, Consistent Hash could not preserve locality, since it distributes key on a keyspace in a random fashion, and it is not ensured that close by keys are closer in the keyspace.

Since the performance of Consistent Hash with respect to load balance and migration cost is adequate, it is better to improve Consistent Hash to provide locality-awareness. Histogram Hash mechanism is a result of such an effort.

The idea behind Histogram Hash is to override the bucketing mechanism of Consistent Hash by creating a histogram of the data distribution. The histogram created is an equi-depth histogram having multiple buckets. The boundaries of buckets in the histogram are arranged according to the frequency of the elements. The total frequency of elements that will be used in the histogram is calculated and then divided by the number of buckets. This gives the frequency per bucket.

To form the histogram, the elements of a sample stream data set are sorted. According to the frequency per bucket property, each data item is placed in the current bucket as long as the frequency per bucket property is not exceeded. When frequency per bucket is exceeded for one bucket, the next one is started to be filled by incoming elements. The smallest and the largest element values give the boundaries of buckets in the histogram. This is basically how the histogram is prepared.

After having prepared the histogram for use, the data items to be partitioned are first processed according to histogram bucket boundaries. Rather than directly hashing data items using Consistent Hash, the bucket of the histogram that an item should be placed according to the histogram bucket boundaries is calculated. Then, instead of hashing the real data item by Consistent Hash, the histogram bucket number is hashed by the underlying Consistent Hash. Therefore, it is ensured that the local items will be in the same bucket of the histogram and consequently will be assigned to the same operator replica. Figure 4.1 shows

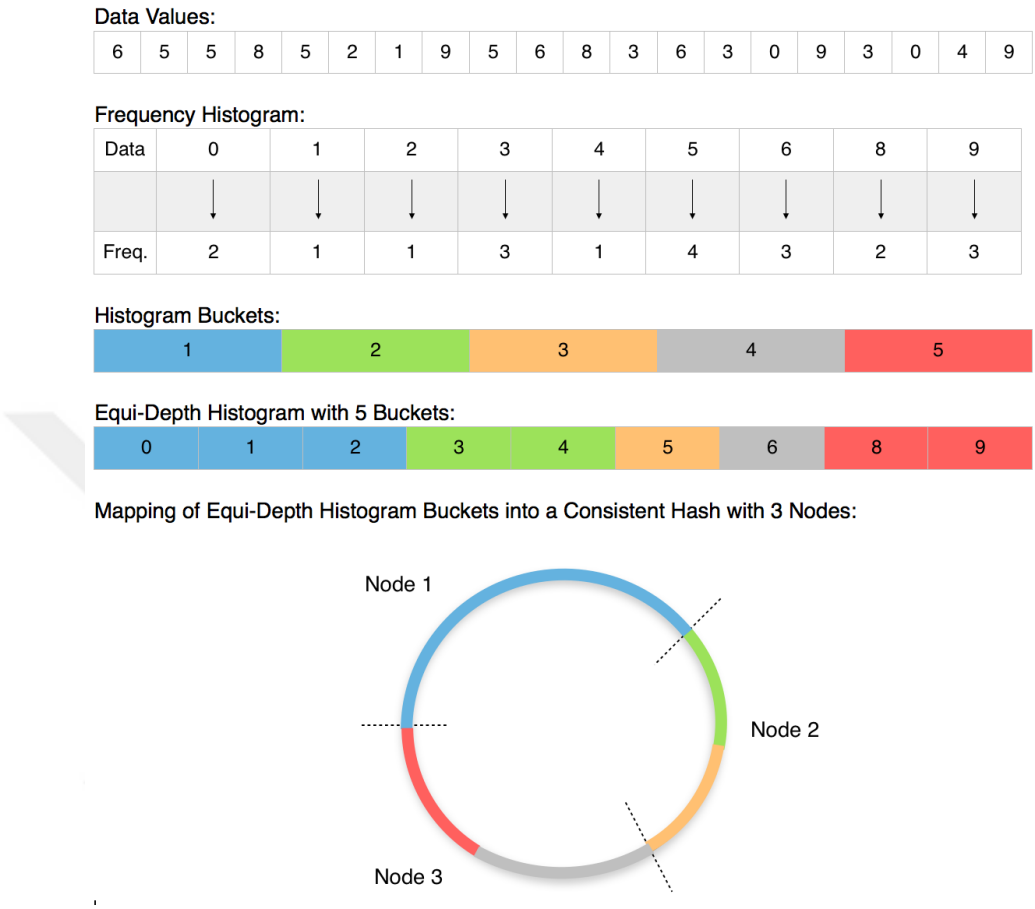

Figure 4.1: An illustration about how Histogram Hash is obtained using Consistent Hash.

the mapping of an example data stream into the consistent hash using an equi-depth histogram, which is called as Histogram Hash.

As stated above, there are two building blocks while creating the histogram that is used in the Histogram Hash. The first one is calculating frequencies of data points, and the second one is creating the equi-depth histogram using the frequencies. The algorithm for calculating the frequencies (Algorithm 1) uses the data set and the workload generator as the parameters and returns a frequency distribution, which is a map that contains the keys in the data set and their frequencies. The frequency of a key defines how many times it appears in the data set. The frequency distribution is used to determine which data points will be placed in which histogram buckets, since the total frequencies in each histogram

Algorithm 1 calculateFrequencies(D, W)

```
1: Param: D, data-set
2: Param: W, number-generator
3: f ← {}
4: for all d ∈ D do
5:     k ← W.generate();
6:     if f contains k then
7:         f[k] ← f[k] + 1
8:     else
9:         f[k] ← 1
10:    end if
11: end for
12: return f
```

bucket is set to be equal. This is done in order to keep the load balanced among histogram buckets.

The equi-depth histogram is created from the frequency distribution. This histogram is used in Histogram Hash. When a new data is needed to be hashed, the histogram bucket for that data is calculated according to the lower and upper boundaries of the histogram buckets. After the histogram bucket is obtained for the data, its histogram bucket index is used in the underlying Consistent Hash to determine the operator replica it will map to. This provides a locality-aware hashing mechanism, while preserving the load-balance across the operator replicas. The algorithm for creating the equi-depth histogram (Algorithm 2) uses the frequency distribution, the frequency per bucket, and the histogram bucket count as the parameters and returns an equi-depth histogram as the result. The resulting histogram is a list of buckets, when the buckets contain the data items. The data items are sorted before the creation of the histogram. Each data item is added to a bucket's list of data while the frequency per bucket property is not exceeded for that bucket. When the frequency for a bucket is exceeded, the data is started to be added a new bucket until the number of buckets become zero. When a data item is going to be hashed using the Histogram Hash, its bucket in the histogram is found via binary search and its bucket index is hashed using the underlying Consistent Hash.

Algorithm 2 createHistogram(F, P, B)

```
1: Param: F, frequency distribution
2: Param: P, frequency per bucket
3: Param: B, bucket count
4: e ← [[]]
5: B ← B − 1
6: i ← 0
7: l ← 0
8: for all k ∈ F do
9:     f ← F[k]
10:    if |(l + f) − P| < |(l − P)| or B = 0 then
11:        e[i] ← e[i] ∪ {k}
12:        l ← l + f
13:    else
14:        e ← e ∪ [[k]]
15:        l ← f
16:        B ← B − 1
17:    end if
18:    i ← i + 1
19: end for
20: return e
```

Histogram Hash mechanism has a payoff. While preserving locality, the load balance property of Consistent Hash is altered. That could cause in performance changes in migration as well. However, the evaluations show that migration is not affected by the alteration. Related to that point, there is a point that the locality measure and load balance measure are reasonably well performing. This shows that the Histogram Hash could be an effective alternative when the locality measure of data is important to reduce disk access latency.

4.2 2-Dimensional Case

The design of the partitioner in a multi-dimensional space is rather difficult when compared to the one-dimensional case, because there is no ordering that preserves spatial locality. This requires a different approach that considers the multi-dimensional space. Two alternative approaches are developed for 2-dimensional

data: dimension reduction via space filling curves and k-d trees.

4.2.1 Dimension Reduction

For the 2-dimensional space, if a linear representation could be obtained, it would be easier to partition the space as in the case for the one-dimensional space. In the literature, there are different alternatives for this situation. Space-filling curves which provide a linear mapping for multidimensional spaces constitute the baseline for this problem [10]. Space-filling curve is a continuous curve that passes through every point of a closed square [11].

The Hilbert curves is a continuous fractal space-filling curve, which is a variant of the space-filling curves. Space filling curves, particularly Hilbert Curves, maintains locality in a linear ordering of multi-dimensional data [12]. The Hilbert curve is useful because it gives a mapping between 1-dimensional and 2-dimensional space that fairly well preserves locality [11]. On a Hilbert Curve, which provides a linear ordering of multi-dimensional data, each point in the data set is represented by a point on the curve. This is called the Hilbert Index. Formally [12]:

Consider a 2-Dimensional lattice with 2^m *points per dimension*

$$P = B^m * B^m$$

where $B^m = 0, 1^m$

A standard Hilbert Index is a function

$$H : P -> B^{2m}$$

which maps each point to its index

on the Hilbert curve as it passes through the lattice

Due to locality property of Hilbert Index, the Hilbert curve is used in our work to provide a locality-aware mapping between 2-dimensional and 1-dimensional spaces.

The Hilbert Index for 2-dimensional data is obtained using a library called

Algorithm 3 buildKdTree(S, D, M, P)

```
1: Param: S, data-set
2: Param: D, depth of the tree
3: Param: M, minimum size
4: Param: P, split strategy                          ▷ according to size or variance
5: if D.size() == 0 or D.size() < M then
6:     return null
7: else
8:     mid ← null
9:     if D / 2 = 0 then
10:        mid ← split(S, true, P)                   ▷ true for split on x-axis
11:    else
12:        mid ← split(S, false, P)                  ▷ false for split on y-axis
13:    end if
14:    x-coordinate ← (S.get(mid).x-coor+S.get(mid − 1).x-coor)/2
15:    y-coordinate ← (S.get(mid).y-coor+S.get(mid − 1).x-coor)/2
16:    root ← {x-coordinate, y-coordinate}
17:    root.right(buildKdTree(S[0, mid], D+1, M, P))
18:    root.left(buildKdTree(S[mid, S.size()], D+1, M, P))
19:    return root
20: end if
```

Uzaygezen. It is an open source project, which provides multi-dimensional indexing using Hilbert Curves. Uzaygezen supports mapping from a multi-dimensional space into one-dimension space via calculating the Hilbert Index.

4.2.2 A Special Purpose K-D Tree

The k-d tree is used in the partition of a two-dimensional data set. The k-d tree is a binary tree in which every node is a k-dimensional point [13]. Every non-leaf node implicitly generates a splitting hyper-plane that divides the space into two parts. This produces two sub-trees in the left and the right of that node. The hyper-plane direction is chosen in the following way: for a particular split, if the "x" axis is chosen, all points with a smaller "x" value than the node will appear on the left sub-tree and all points with larger "x" value will be on the right sub-tree.

In our work, k-d tree is used for partitioning purpose, rather than for range searching. The points that fall into the same leaf node of the k-d tree are considered to be close. This property provides a locality-aware partitioning for multi-dimensional spaces.

We developed a special k-d tree implementation in order to have a balanced and a locality-aware partition of the data set. While partitioning the 2-dimensional space, the data variance in the splitting hyper-plane is calculated in order to preserve balance and locality. The algorithms for building the k-d tree is provided in Algorithm 3.

Chapter 5

Evaluation

5.1 Evaluation Metrics for Data Partitioning

5.1.1 Load Balance

Load Balance is the term that is used to define the load balance among the nodes. The balance calculation algorithm works in the following way: the method gets the data size for each node as an array. It then calculates the average size. Next, it calculates the sum of distances of each node's data size to the average size, and divides it into the number of nodes.

5.1.2 Locality

Locality is defined as how close the data values are in a node. The calculation of locality in a node is done in the following way: all the numbers in a data node are sorted first. Next, the distance of consecutive numbers are calculated. The sum of the distances is then divided into the size of data in the node. For each node, the locality is measured in this way, and the per-node locality values are averaged across nodes to get the final locality value. If the numbers in a node are

all consecutive, then locality for that node is perfect, that is equal to 1. Locality values are in the range of (0,1].

5.1.3 Migration

If a new node is added to the distributed system or if a node is removed from the distributed system, the data stored in the whole distributed system is needed to be re-distributed according to the new number of nodes. Migration is measured in the following way: the number of nodes is increased by one. After that, the same data points are placed again for the increased number of nodes. The number of data points that are not placed in the same node for both of the cases is divided by the new node count to obtain the migration measure.

5.2 Performance in 1-Dimensional Data

5.2.1 Generation of Test Data

The test data is generated using Zipf distribution in tests for 1-dimensional data. Zipf is a probability density function, sometimes referred to as the zeta distribution [14]. Zipf's law states that given some corpus of natural language expressions, the frequency of any word is inversely proportional to its rank in the frequency table [15]. In other words, the terms that are more often encountered in a set has a lower rank than the terms that are encountered less often. Thus the most frequent word will occur approximately twice as often as the second most frequent word [15].

The generated data is composed of integers in the range of (0,100000). The first test benchmark is created to measure the effect of changing node sizes on the locality, balance, and migration. For the other two benchmarks, the test data set size and distribution skew is modified to figure out their impact on performance.

5.2.2 Analytical Results and Illustrations

The performance of data partitioning method is compared under three changing parameters, the node size, the data set size and the data skew. The performance of Histogram Hash is compared to the performance of Consistent Hash and Random Hash. The figures presents the Histogram Hash's locality preserving property. In the figures; HH represents the hHistogram Hash, CH represents the Consistent Hash and RH represents the Random Hash.

5.2.2.1 Tests Under Changing Node Sizes

The node size is defined as the number of copies of the data parallel operators. Changing the node size means adding a new node to the system or removing a node from the system. In other words, changing node size is changing the number of copies of a data parallel stateful operator. Three metrics; locality, migration, and balance are inspected under varying node size. Figure 5.1, 5.2, and 5.3 show that Histogram Hash performs better over the Consistent Hash and Random Hash regarding locality. As the node size increases the locality preserving property is kept in the Histogram Hash, but the performance regarding the balance property is decreasing due to histogram buckets' unbalanced mapping on the Consistent Hash keyspace. The Histogram Hash has far better performance than the Consistent Hash in terms of locality. Although the relationship of locality and balance could be seen as a trade-off, Histogram Hash is one step ahead of Consistent Hash, since there is an optimal point that the performance of both locality and balance is accurate enough.

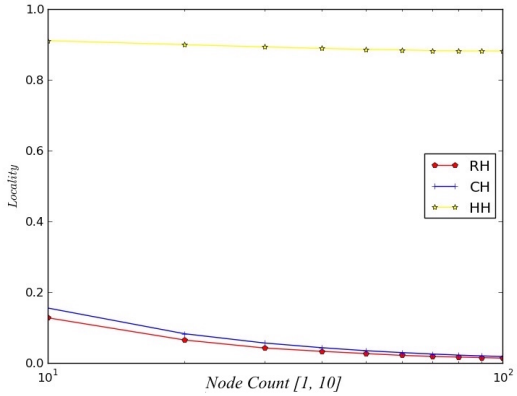

Figure 5.1: Node Size vs. Locality

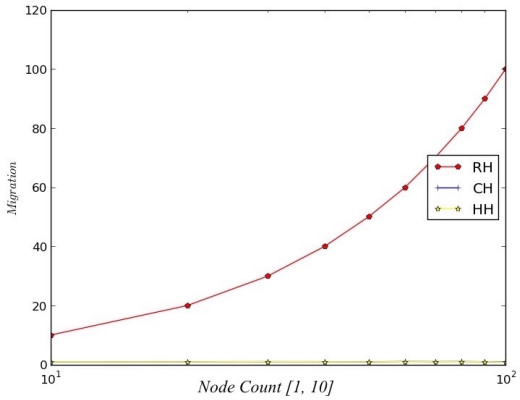

Figure 5.2: Node Size vs. Migration

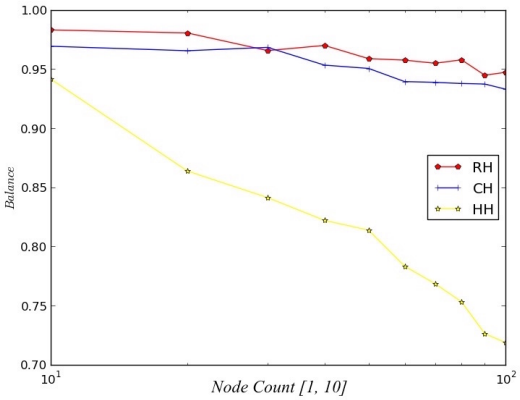

Figure 5.3: Node Size vs. Balance

5.2.2.2 Tests Under Changing Data Set Sizes

The data set size indicates the number of data items that pass through the stream. Changing the data set size means increasing or decreasing the amount of data that the system will be consuming during the tests. Three metrics; locality, migration, and balance are inspected under varying data set size. Figure 5.4, 5.5, and 5.6 show that the Histogram Hash performs better over the Consistent Hash and Random Hash regarding locality. As the data set size increases, the locality preserving property of Histogram Hash increases as well. The performance of balance property is again better as data set size gets larger. Increasing data set size helps the Histogram Hash to provide better results in both locality and balance, since Histogram Hash mechanism uses a part of the data to create a histogram for the data set and partitions the data using this histogram and an underlying Consistent Hash. The migration performance is again, as expected, close for both the Histogram Hash and Consistent Hash. However, the Random Hash performs worse than these two in terms of migration.

5.2.2.3 Tests Under Changing Skew on Data

Three metrics; locality, migration, and balance are inspected under varying skew in the data distribution. Figure 5.7, 5.8, and 5.9 show that the Histogram Hash performs better over the Consistent Hash and Random Hash regarding locality. As the data skew increases the locality preserving property increases as well in the Histogram Hash. The balance and the migration properties vary as the data skew increases. The performance of balance especially decreases due to the skewed mapping between the histogram buckets and the Consistent Hash keyspace. Increasing data skew ruins the balanced composition of histogram buckets and it results in decreased load balance. The migration performance is again, as expected, close in both the Histogram Hash and Consistent Hash. However, the Random Hash performs worse than these two.

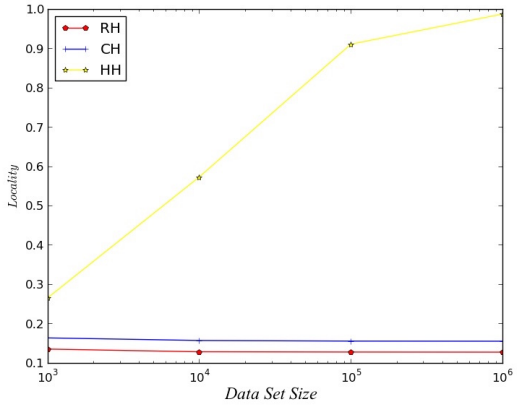

Figure 5.4: Data Set Size vs. Locality

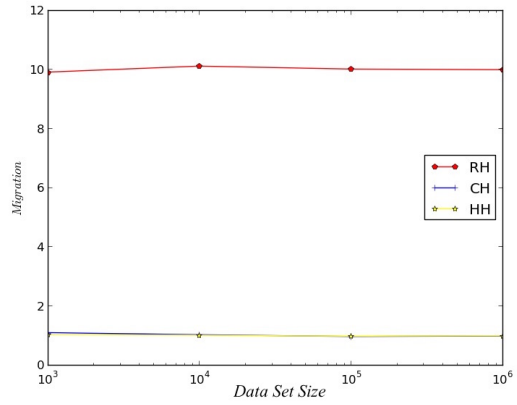

Figure 5.5: Data Set Size vs. Migration

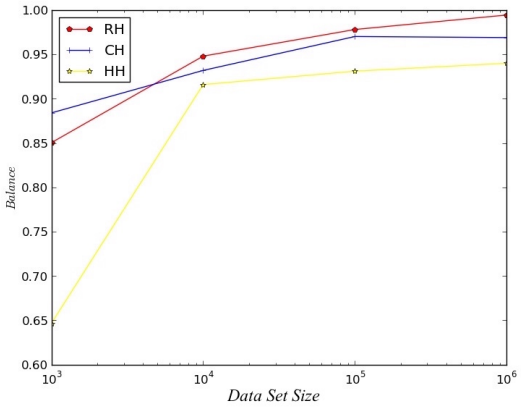

Figure 5.6: Data Set Size vs. Balance

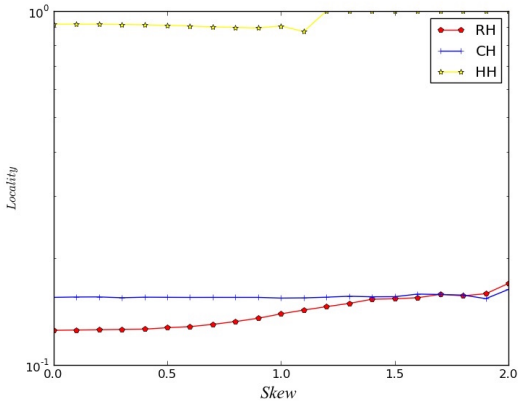

Figure 5.7: Skew vs. Locality

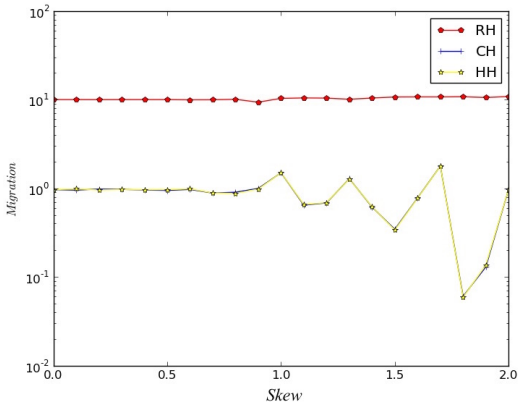

Figure 5.8: Skew vs. Migration

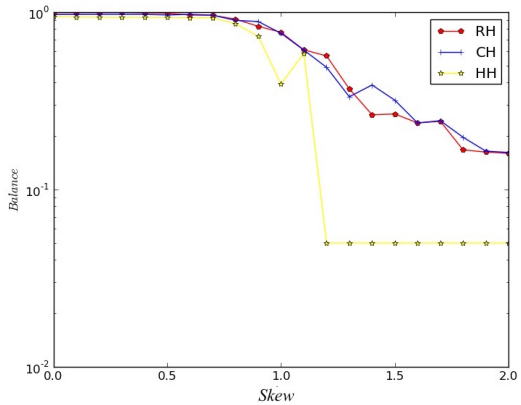

Figure 5.9: Skew vs. Balance

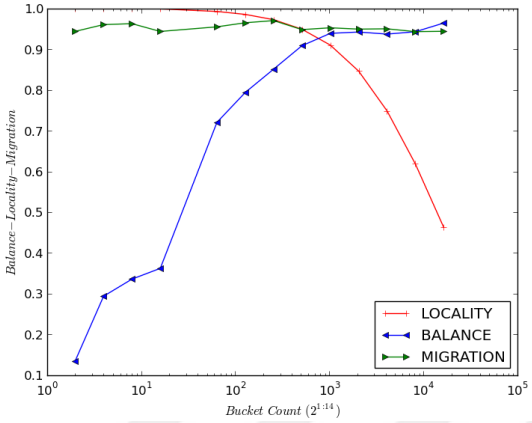

Figure 5.10: Changes in Balance, Locality and Migration Under Varying Histogram Bucket Count

5.2.2.4 Relation Between Histogram Bucket Count and Balance-Locality-Migration

Evaluating the performance if Histogram Hash mechanism presents a new concept that needs further study. Histogram Hash uses buckets and distributed data items into buckets with respect to data frequency and locality. Since the Histogram Hash uses an underlying Consistent Hash to hash the bucket indices, changing the number of buckets impacts balance, locality and migration metrics. Therefore, we tested the Histogram Hash's performance in terms of balance, locality and migration under changing bucket count. As Figure 5.10 shows, while the bucket count is increasing the performance regarding load balance, it decreases it in terms of locality. The reason behind this behavior of the Histogram Hash is due to the following: When the bucket count is decreased the locality increases because the most of the data is placed in the same bucket and the same Consistent Hash node as well. However, this causes a lower performance in terms of balance, since the most of the data is placed in the same node of the Consistent Hash. While increasing the bucket count, the situation becomes reversed. There is also an optimum point where a sweet spot between locality and balance could be obtained.

5.3 Performance in 2-Dimensional Data

5.3.1 Generation of Test Data

Gaussian distribution or normal distribution is a continuous probability density function [16]. It is also called informally as bell curve because of the shape of its distribution graphic. The Gaussian distribution is symmetric about its mean, and it is non-zero over the entire real line. The probability density of the normal distribution is given as follows [16]:

$$f(x \mid \mu, \sigma^2) \;=\; \frac{1}{\sqrt{2\sigma^2 \Pi}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Gaussian distribution has a wide and variety range of use to represent many phenomena in different areas of science. Gaussian distribution is used in this work to represent 2-dimensional spatial data generation. It is considered to be useful due to its ability to model skewed spatial data.

5.3.2 Analytical Results and Illustrations

In the tests with 2-dimensional data, an open source library is used to generate Hibert Curve for the 2-dimensional data. The library is called Uzay Gezen [17]. Uzay Gezen library provides a mapping from a multi-dimensional space into the one-dimensional space via the Compact Hilbert Index. During the tests, two metrics are used: the grid size and the leaf count. The grid size describes the number of grids in the 2-dimensional plane. The leaf count represents the number of leaves in the k-d tree. These two are used to measure changes in locality, migration, and balance properties. In Figures 5.11, 5.12, and 5.13; it can be seen that as the grid size and the leaf count increases, the locality decreases. This is because when the grid size increases, the plane is divided into more partitions, and this causes local data to be put in different histogram buckets. The balance

and migration, however, are not affected by the grid size. Since the k-d tree is a binary tree, the balance and the migration properties are not affected by increasing or decreasing the grid size. The leaf count, however, causes changes in balance and the migration because while partitioning the 2-dimensional space increasing the leaf count results in more divisions and could lead to an unbalanced distribution of data among the divisions.

5.4 Case Studies on Apache Storm

Apache Storm is an open source distributed stream processing middleware. Its architecture consists of spouts, bolts, and grouping mechanisms. The running stream processing programs are called storm topologies in Apache Storm.

Spouts define the data sources that provide streaming data into a storm topology. Bolts are the main components where the analytical tasks are handled. The grouping mechanisms provide strategies for emitting tuples between Apache Storm components. Both Spouts and Bolts emit tuples between them using the grouping mechanisms. Bolts could have multiple tasks, which are copies of the same bolt that process tuples in parallel. Bolts that have more than one task for processing tuples of a data stream are called data parallel operators. These type of operators provide parallel processing of data streams. Apache Storm has built-in and custom grouping mechanism for managing data partitioning among tasks of a bolt. The proposed custom grouping mechanism uses Histogram Hash to provide local data items to be processed by the same task of the bolt. Together with the state management service, this constitutes the locality-aware state management mechanism for Apache Storm.

The performance of the Histogram Hash mechanism is tested under different data sizes and different dimensions using balance, migration, and locality metrics. Having these metrics in hand, testing this mechanism in a real-world environment would be useful for obtaining real performance measures. For this

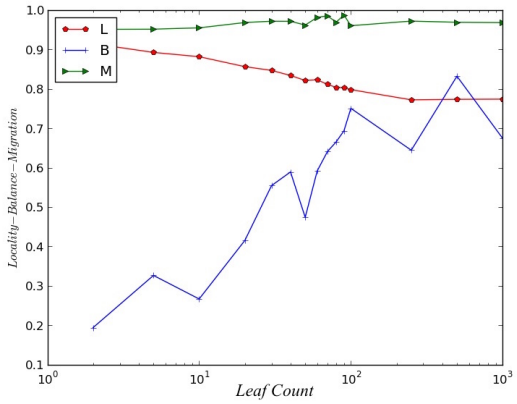

Figure 5.11: Changing Leaf Count with Grid Size 12

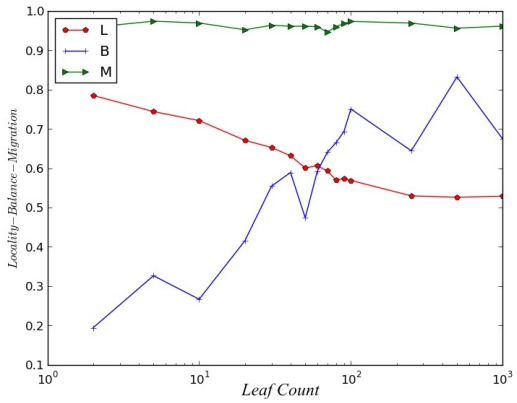

Figure 5.12: Changing Leaf Count with Grid Size 13

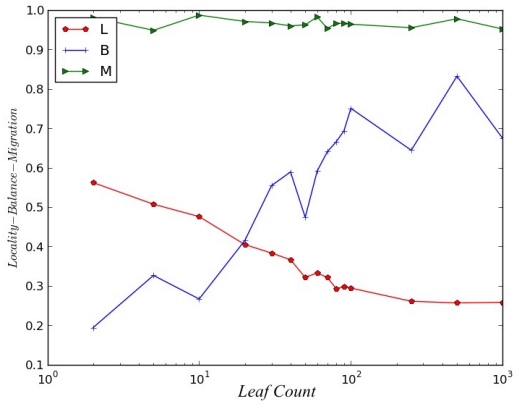

Figure 5.13: Changing Leaf Count with Grid Size 14

purpose, a locality-aware state management mechanism for Apache Storm is developed. Since the proposed mechanism is a locality-aware state management system, check-pointing the states of data parallel operators into a persistent storage and enrichment of the keys from the persistent storage are the prominent use cases.

The motivation behind the first use case is that, when an failure occurs and the topology is needed to be restarted, the state of the storm bolt could be persisted to keep the state when restarting the topology. While writing data into disk utilizing batch queries will reduce the time for writing. The idea that is tested in the benchmark is, if the locality of the data in an operator is high, it will result in faster write time due to the storage's tendency to keep local items together. This is tested under varying skew, data size, and node sizes.

The motivation behind second use case is that, when the keys stored in the data parallel operators are needed to be enriched with the information stored on the disk, the read time from the persistent storage should be as fast as possible to meet the requirements of stream processing applications. The idea that is tested in the benchmark is, if the locality of the keys stored in an operator is high, it will result in faster read time due to the storage's tendency to keep local items together and bringing local items together in the queries. This use case is also tested under varying node size, data size, and skew.

5.4.1 Test Environment

Evaluating the scalability of our solution requires multiple machines. We use kernel-based virtual machines in amd64 architecture with 4 cores and 4GB memory. The machines have CentOS 6.6 operating system, 1.7.25 version of Java, 2.6.0 version of Apache Hadoop [18], 1.0.0 version of Apache HBase [7], and 0.8.2 version of Apache Storm [5] installed.

5.4.2 Test Results

While testing the locality-aware partitioned state management library for Apache Storm, the write time into a storage system is considered. The partition mechanism is tested and its performance is inspected under different conditions such as varying node size, data set size, and data skew. The main purpose of the partitioning method is to increase the performance in storage read/write time via utilizing locality. Apache HBase is an open source, distributed No-SQL database [7]. Apache HBase considers locality of the data when reading or writing [7]. That is, for instance, putting local data instances into the database is faster because of the local data's layout in Apache HBase. Since our partition mechanism, Histogram Hash, is locality-aware, using Apache HBase is expected to result in better performance result compared to other approaches; Consistent Hash and the Random Hash.

5.4.2.1 Checkpointing into the Disk

The stream processing systems process the data on-the-fly. This requires processing the streaming data in real-time. These systems are sometimes required to handle disk operations. In this use case, the disk operations are defined as checkpointing into the disk. The data parallel operators may need to persist their states into a storage unit. The motivation behind this use case is to show that the write time performance is getting better with the use of the Histogram Hash mechanism as the custom grouping mechanism for data parallel stateful operators. The test benchmark shows that locality-aware partitioning mechanism provides better write time for stream processing systems.

The tests are conducted by varying three different parameters: the node size, the data set size, and the data skew. For these parameters, the write time into HBase is measured. In Figures 5.14, 5.15, and 5.16; HH represents the Histogram Hash, CH represents the Consistent Hash, and RR represents the Random Hash partitioning mechanism that is used by state management service.

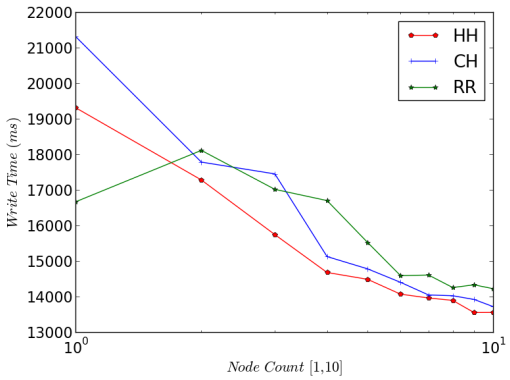

Figure 5.14: Node Size vs. Write Time

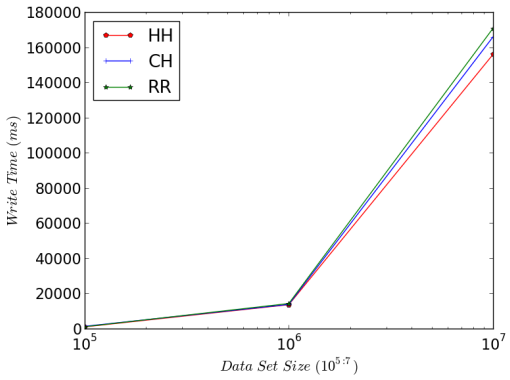

Figure 5.15: Data Set Size vs. Write Time

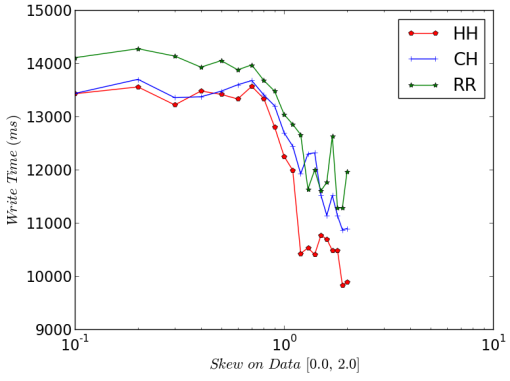

Figure 5.16: Data Skew vs. Write Time

Figure 5.14 shows the performance of Histogram Hash compared to the Random Hash and the Consistent Hash under changing node sizes. The histogram hash grouping shows lower write times when the consistent hash and the random hash groupings are considered. Increasing the node size provides writing data into the disk in parallel, and thus provides better performance for the write time.

In Figure 5.15, the performance of Histogram Hash compared to the Random Hash and the Consistent Hash is seen under changing data set size. The Histogram Hash grouping performs better than the Consistent Hash and the Random Hash groupings. As the data set size increases, the difference between write time of the Histogram Hash, Consistent Hash, and Random Hash groupings increases. The histogram hash grouping provides better write time as the data set size increases.

Figure 5.16 presents the result of changing data skew. As the data skew increases the write performance enhances. While the skew increases, the same data items are started to appear more frequent. This causes to better performance while writing on database. The histogram hash grouping again performs better than the consistent hash and the random hash groupings.

5.4.2.2 Enrichment from the Disk

Although the stream processing systems handle the data on-the-fly, they are sometimes required the enrich that data from some other sources. One of these sources might be the storage units. According to this use case the stream processing systems are required to enrich the keys, passing through the system, from the data stored on the disk. These operations are defined as enriching data from the disk. The motivation behind this use case is to show that the read time performance is getting better with the use of Histogram Hash mechanism as the custom grouping mechanism for data parallel stateful operators. The test benchmark presents that the locality-aware partitioning mechanism provide better read time for stream processing systems.

The tests are conducted varying three different properties: the node size, the data set size, and the data skew. For these parameters, the read time from HBase is measured. In Figures 5.17, 5.18, and 5.19; HH represents the Histogram Hash, CH represents the Consistent Hash, and RR represents the Random Hash partitioning mechanism that is used by the state management service.

Figure 5.17 shows the read time performance of the Histogram Hash approach compared to the Random Hash and the Consistent Hash under varying node sizes. The Histogram Hash grouping presents lower read times compared to the Consistent Hash and the Random Hash groupings. Increasing the node size provides reading data from the disk in parallel, so provides better performance on read time.

In Figure 5.18, the performance of the Histogram Hash approach over the Random Hash and the Consistent Hash is seen under changing data set size. The Histogram Hash performs better than the Consistent Hash and the Random Hash. When the data set size increases, the performance of the Histogram Hash grouping improves with respect to the performance of the Consistent Hash and the Random Hash groupings, because the histogram's partitioning scheme performs better as more data is partitioned via the mechanism.

Figure 5.19 presents the result of changing data skew. As the data skew increases the read performance increases, because the frequency of the same item started to increase on the disk. When the skew in at significant values, the performance of Histogram Hash groping is better than both Consistent Hash and Random Hash groupings.

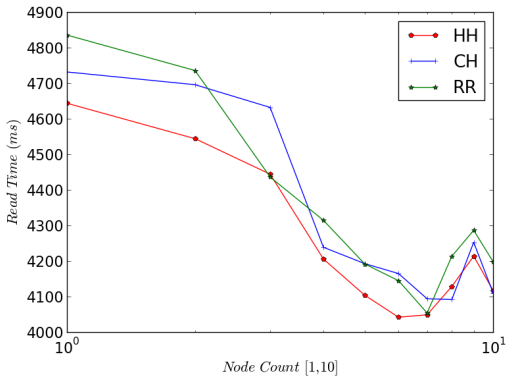

Figure 5.17: Node Size vs. Read Time

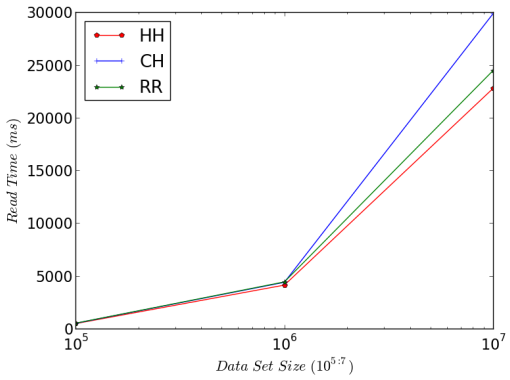

Figure 5.18: Data Set Size vs. Read Time

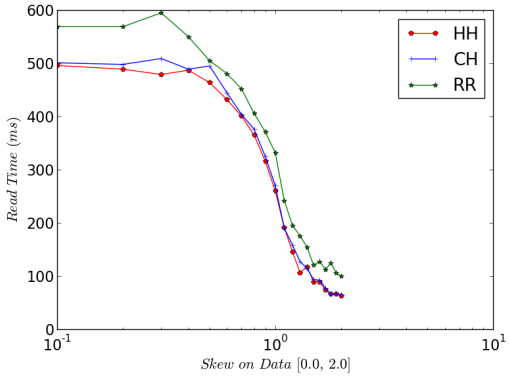

Figure 5.19: Data Skew vs. Read Time

Chapter 6

Conclusion

In this thesis, we introduced the problem of locality-aware state management for distributed stream processing applications. We proposed a solution that provides a locality-aware state management library for Apache Storm. The solution consists of two components: the locality-aware data partitioning mechanism and the state management service for data parallel stateful operators. Histogram Hash was developed and utilized for locality-aware data partitioning for the 1-dimensional data. Histogram Hash works by observing the stream of data and creating an equi-depth histogram from that data. After creating the histogram, it uses Consistent Hash to partition the data, by passing the histogram bucket indices of the data items to the Consistent Hash, rather than the data items themselves. k-d trees and space-filling curves were used for locality-aware data partitioning for two-dimensional data.

The state management service provides a locality-aware state management facility, which depends on the locality-aware partitioning mechanism provided by the Histogram Hash. The locality-aware state management service provides APIs for data parallel stateful operators to persist their state into the storage system. We developed our state management solution for the open source, distributed stream processing middleware Apache Storm.

We evaluated the performance of our work in two parts: the performance of the Histogram Hash and the performance of the locality-aware state management library for Apache Storm. While evaluating the performance of the Histogram Hash, we considered three performance metrics: the load balance of the data, the locality of the data, and the migration cost of the data under three changing variables, namely: data size, data skew, and parallel node size. The experiments compared the Histogram Hash with the Random Hash and the Consistent Hash. The results show that the Histogram Hash performs similar to the Consistent Hash regarding load balance and migration cost, yet performs better in terms of locality.

After showing that Histogram Hash improves locality, we created the experimental setup for testing its impact using Apache Storm. Using Apache HBase as the storage backend, we created a test benchmark for writing state of the data parallel operators into the persistent storage and enriching the keys stored in data parallel operators via reading information from the persistent storage. The purpose was to show that if the locality of the data was higher, the time required for writing and reading the data will be shorter, because the local data is written into and read from Apache HBase faster. Finally, the demo application that uses locality-aware state management library has shown that the locality-aware partitioning performs better for Apache Storm, as the write and read times to and from the storage decreased with locality-aware partitioning.

Bibliography

[1] S. Schneider, M. Hirzel, B. Gedik, and K. Wu, "Safe data parallelism for general streaming," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 64, no. 2, pp. 504 – 517, 2015.

[2] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *The VLDB Journal*, vol. 23, no. 4, pp. 517 – 539, 2014.

[3] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu, "Auto-parallelizing stateful distributed streaming applications," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 53–64, ACM, 2012.

[4] T. S. F. X. Teixeira, G. Teodoro, E. Valle, and J. H. Saltz, "Scalable locality-sensitive hashing for similarity search in high-dimensional, large-scale multimedia datasets," *CoRR*, vol. abs/1310.4136, 2013.

[5] "Apache storm." http://storm.apache.org. [Online; accessed 19-September-2016].

[6] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2009.

[7] "Apache hbase." http://hbase.apache.org. [Online; accessed 19-September-2016].

[8] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web caching with consistent hashing," *Comput. Netw.*, vol. 31, pp. 1203–1213, May 1999.

[9] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "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, (New York, NY, USA), pp. 654–663, ACM, 1997.

[10] H. Sagan, *Space-filling curves*. Universitext Series, Springer-Verlag, 1994.

[11] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of hilbert space-filling curve," tech. rep., University of Maryland at College Park, 1996.

[12] C. H. Hamilton and A. Rau-Chaplin, "Compact hilbert indices for multi-dimensional data," in *Complex, Intelligent and Software Intensive Systems, 2007. CISIS 2007. First International Conference on*, pp. 139–146, IEEE, 2007.

[13] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, Sept. 1975.

[14] M. E. J. Newman, "Power laws, Pareto distributions and Zipf's law," *Contemporary Physics*, vol. 46, pp. 323–351, Sept. 2005.

[15] D. M. Powers, "Applications and explanations of zipf's law," in *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*, pp. 151–160, Association for Computational Linguistics, 1998.

[16] J. K. Patel and C. B. Read, *Handbook of the normal distribution*, vol. 150. CRC Press, 1996.

[17] "Uzaygezen." https://github.com/aioaneid/uzaygezen. [Online; accessed 19-September-2016].

[18] “Apache hadoop.” http://hadoop.apache.org. [Online; accessed 19-September-2016].