Méthodes parallèles pour le traitement des flux de données continus

Ge Song

NNT : 2016SACLC059

THESE DE DOCTORAT
DE
L'UNIVERSITE PARIS-SACLAY
PREPAREE A
L'UNIVERSITE CENTRALESUPELEC

ÉCOLE DOCTORALE N°573
Interfaces : approches interdisciplinaires / fondements, applications et innovation

Spécialité de doctorat : Informatique

Par

Mme Ge Song

Méthodes parallèles pour le traitement des flux de données continus

Thèse présentée et soutenue à Châtenay-Malabry, le 28 septembre 2016 :

Composition du Jury :

M. Johan Montagnat, CNRS, Rapporteur
M. Abdelkader Hameurlain, IRIT Institut de Recherche en Informatique de Toulouse, Rapporteur
M. Lei Yu, Ecole Centrale de Pékin, Examinateur
Mme. Bich-Liên Doan, CentraleSupélec, Examinateur
M. Frédéric Magoulès, CentraleSupélec, Directeur de thèse
M. Fabrice Huet, I3S Université Nice Sophia Antipolis, Co-Encadrant de thèse

I would like to dedicate this thesis to my family ...

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Ge SONG
November 2016

Acknowledgements

I am grateful to all the people who help and support me during these three and a half years.

Firstly, I would like to thank my two supervisors Professor Frédéric Magoulès and Professor Fabrice Huet. Without their support and help, this thesis can not be finished. They gave me many valuable ideas, suggestions and criticisms with their rich research experience and background. I have learned many things from them such as algorithm analysis, benchmark design, dissertation writing etc.

Secondly, I wish to acknowledge the reviewers of this thesis: Mr. Johan Montagnat and Mr. Abdelkader Hameurlain, and the juries of my oral defence: Mrs. Bich-Liên Doan and Mr. Lei Yu, for their time to review this thesis and their good advices of making it better.

Thirdly, I would like to express my thanks to all my colleagues in the team for the relax and happy environment they establish to work. Especially, I would like to thank my friends Justine Rochas and Oleksandra Kulankhina with whom I shared the office during these years. I am very grateful for their company in my dark days. Besides, It was very nice co-writting papers with Justine and cooking with Oleksandra. I would also like to thank all the interns and students I have co-worked with during my study.

Finally, I would like to express my thanks to my family. My gratitude first goes to my lovely parents for their support and encouragement. I also want to thank my younger sister Rui Song, who is smart and talented and works very hard. Her attitude always affects me. My family is always the source of my efforts.

Abstract

We live in a world where a vast amount of data is being continuously generated. Data is coming in a variety of ways. For example, every time we do a search on Google, every time we purchase something on Amazon, every time we click a 'like' on Facebook, every time we upload an image on Instagram, every time a sensor is activated, etc., it will generate new data. Data is different than simple numerical information, it now comes in a variety of forms. However, isolated data is valueless. But when this huge amount of data is connected, it is very valuable to look for new insights. At the same time, data is time sensitive. The most accurate and effective way of describing data is to express it as a data stream. If the latest data is not promptly processed, the opportunity of having the most useful results will be missed.

So a parallel and distributed system for processing large amount of data streams in real time has an important research value and a good application prospect. This thesis focuses on the study of parallel and continuous data stream Joins. We divide this problem into two categories. The first one is Data Driven Parallel and Continuous Join, and the second one is Query Driven Parallel and Continuous Join.

In a Data Driven Join, the query never changes, but the type (format, dimension, etc.) of data does change. We use kNN Join (k Nearest Neighbor) as our use case for this category of Join. The Data Parallel model should be used for this kind of Join. The difficulties lie on the preprocessing and the partitioning of data. We review five different kinds of parallel and distributed processing methods for kNN Join. We then summarize the workflow into the following three steps: data-preprocessing, data-partitioning and data-computation. The pre-processing stage is used to either select the pivot points or reduce the dimensionality of data. We introduce two different types of partition method, one size-based, the other distance-based. In the computation step, we also present two strategies, one which produces directly the global result, and one which firstly produces a local result. We then analyze the methods theoretically from Load Balance, Accuracy and Complexity aspects. We have implemented every method in Hadoop MapReduce to do Benchmarks, thereby quantitatively determine the application scenarios for each method. After having evaluated the parallel methods, we extend some of them to continuously process data streams. We use the Sliding

Window model for processing the most recent data, and update the results periodically. We design two data re-partition strategies — a naive one and an advanced one based on Naive Bayes. Finally, we implement the algorithms on Apache Storm, and evaluate its real-time performance.

In a Query Driven Join, the format of data does not change. Queries, however, can be arbitrary, because it is written by the users, which is unpredictable. A Task Parallel model should be applied for a Query Driven Join. We use a Join on RDF data (Semantic Web) as an example for this kind of Join. The difficulties lie in the efficient decomposition of queries for parallel and distributed processing. We design several strategies to decompose queries according to their structure and generate a parallel query plan. We use Bloom Filters for data transmission among nodes in order to minimize the communication overhead. We propose a Query Topological Sort algorithm to determine the order of information exchange among different sub-queries. After the design for the parallel parts, we also extend it to continuously process data streams. We introduce strategies for data re-evaluation and expiration. We also introduce a method for using Sliding Bloom Filters. We then theoretically analyze the efficiency of the algorithm, and we calculate the dominating parameters for the system. The probability of having false positive results is also discussed. Finally, we implement the whole system on Apache Storm platform to evaluate the algorithms from the parallel aspect and the streaming aspect using both synthetic data and LUBM Benchmarks.

Abstract

Nous vivons dans un monde où une grande quantité de données est généré en continu. Par exemple, quand on fait une recherche sur Google, quand on achète quelque chose sur Amazon, quand on clique en 'Aimer' sur Facebook, quand on upload une image sur Instagram, et quand un capteur est activé, etc., de nouvelles données vont être généré. Les données sont différentes d'une simple information numérique, mais viennent dans de nombreux format. Cependant, les données prisent isolément n'ont aucun sens. Mais quand ces données sont reliées ensemble, on peut en extraire de nouvelles informations. De plus, les données sont sensibles au temps. La façon la plus précise et efficace de représenter les données est de les exprimer en tant que flux de données. Si les données les plus récentes ne sont pas traitées rapidement, les résultats obtenus ne sont pas aussi utiles.

Ainsi, un système parallèle et distribué pour traiter de grandes quantités de flux de données en temps réel est un problème de recherche important. Il offre aussi de bonne perspective d'application. Dans cette thèse nous étudions l'opération de jointure sur des flux de données, de manière parallèle et continue. Nous séparons ce problème en deux catégories. La première est la jointure en parallèle et continue guidée par les données. La seconde est la jointure en parallèle et continue guidée par les requêtes.

Pour une jointure guidée par les données, la requête ne change jamais, contrairement au type (format, dimension, nature, etc.) des données. Nous utilisons l'opération de kNN (k plus proche voisins) comme cas d'utilisation pour cette catégorie de jointure. Un modèle est nécessaire pour paralléliser les données pour ce type de jointure. La difficulté se situe dans le prétraitement et le partitionment des données. Nous examinons cinq types de méthodes en parallèles et distribuées pour traiter l'opération de jointure dans le kNN. Nous décomposons l'algorithme en trois étapes de traitement: le prétraitement des données, leur partitionnement et leur traitement. L'étape de prétraitement est utilisée soit pour organiser les données ou réduire leur dimensionnalité. Nous introduisons deux types de méthodes de partitionnement, l'une basée sur la taille et l'autre sur la distance. Dans l'étape de calcul, nous présentons également deux stratégies, l'une donne directement les résultats finaux, et l'autre produit des résultats intermédiaires. Nous analysons de manière théorique les méthodes du point de vue de l'équilibre, de la précision et de la complexité. Nous implémentons toutes les

méthodes en Hadoop MapReduce afin de les évaluer, et ainsi déterminer quantitativement les scénarios d'application pour chacune. Après avoir évalué les méthodes parallèles, nous les étendons pour traiter des flux de données en continu en temps réel. Nous utilisons le modèle de 'Fenêtre Glissante' pour traiter les données les plus récentes, et mettre à jour périodiquement les résultats. Nous proposons deux stratégies pour le repartitionnement. Enfin, nous implémentons les algorithmes sur la plateforme Apache Storm, et évaluons leur performances. Concernant la jointure guidée par les requêtes, le format de données ne change pas, mais la requête, écrite par les utilisateurs, est arbitraire. Un modèle de parallélisme de tâche peut être appliqué pour ce type de jointure. Nous étudions la jointure sur les données RDF comme un cas d'utilisation. La difficulté se situe dans la décomposition des requêtes en sous-requêtes pouvant être distribuées et parallélisées. Nous proposons plusieurs stratégies pour décomposer les requêtes en fonction de leur structure et générer un plan d'exécution. Nous utilisons des Bloom Filters pour la transmission de données entre les nœuds afin de minimiser les coûts de communication. Nous proposons un algorithme, appelé 'Query Topological Sort', pour déterminer l'ordre d'échange d'informations entre les différentes sous-requêtes. Nous étendons ce mécanisme ensuite au cas continu des flux de données. Nous introduisons des stratégies pour la réévaluation et l'expiration des données. Nous présentons également la méthode de 'Sliding Bloom Filters'. Nous analysons ensuite théoriquement l'efficacité de notre méthode, et nous calculons les paramètres dominants du système. La probabilité d'avoir des faux positifs est aussi discutée. Enfin, nous implémentons l'ensemble du système sur la plateforme Apache Storm en utilisant à la fois des données synthétiques et LUBM Benchmarks.

My Table of Contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Big Data Processing

We are awash in a flood of data today. With a rapid growth of applications like social network analysis, semantic web analysis and bio-information analysis, a large amount of data needs to be processed to witness this quick increase. For example: Google deals with more than 24 PB of data every day (a thousand times the volume of all the publications contained by the US National Library); Facebook updates more than 10 million photos and 3 billion "likes" per day; YouTube accommodates up to 800 million visitors monthly (a video more than an hour uploaded every second); and Twitter is almost doubling its size every year, etc.

Data is being collected every second in a broad range of areas. Information which previously relied on guesswork and experience, can now be discovered directly from the data. The rising of internet search engine technology and the development of social networks, have given the data more power. It also makes the related technologies which depend on the processing of large amount of data such as machine learning, data mining, recommendation systems etc, a hot topic in both IT industry and academic.

Big data has four features:

- **Volume:**
  The amount of data is very large, usually in an order (such as TB and PB) that can not be stored and processed by a single machine. So, data need to be distributed stored in a cluster, then processed in a parallel and distributed way. Therefore, big data processing and cloud computing technologies are closely linked.

- **Variety:**
  The data types are variety. There are low-dimensional data (such as geographic coordinates), and also high-dimensional data (such as audio, video, pictures, etc.) as

well. The formats of data could also be different. So when processing big data, the model needs to adapt to different kinds of data, or transform them into a unified format.

- **Value:**
  Information is everywhere, but the density of data value is relatively low. So before embarking on a high-density computing, a pre-processing step needs to firstly be used to clean and extract information from the original data.

- **Velocity:**
  The timeliness is high, so the processing speed should also be high. Generally, recent data is more valuable, so old data should be withdrawn when new data arrives. At the same time, the system needs to respond in a real-time or quasi real-time manner.

The revolution of big data has brought a lot of benefits to us, but at the same time, it requires an upgrading for the relevant technologies.

1.2 Issues with Big Data

While the potential benefits of big data is really significant, there remain many challenges that should be addressed.

The most significant issue comes from the size of Big Data.The flip side of size is speed. The larger the data set to be processed, the longer it will take to analyze. Processing large and rapidly increasing amounts of data has been a target for many decades. In the past, this target was relying on the computation capacity of the processors, following Moore's law [16]. But now the data volume is scaling faster than the resources, and the capacity of CPU increases slowly. Thus, parallel and distributed solutions have been searched for, from Open MPI [15], to the famous Hadoop ecosystem [2], until the latest Apache Spark [3]. To meet the increasingly growing large amount of data and to optimize the performance, the parallel and distributed computing platforms are also engaged in constantly upgrading.

The second dramatic issue is the cost of network communication in transferring data. The transmission of data always became the bottleneck of a parallel platform according to our previous study in paper [130]. So when designing a method to compute an algorithm in parallel, minimizing the data transmission cost is a very important aspect.

The last but not least important issue comes from the dynamics of data. After the revolution of the Internet technologies and the popularity of social networks, data is now more dynamic. Users add new data to the network every second. And often, the latest data is the most valuable. This process can be described as a stream of data which constantly

updates. Traditional distributed processing platforms process the data in batch. Each data update will trigger a re-calculation. This obviously can not meet the real-time processing requirement.

1.3 Real-Time processing for data stream

Recently a new type of data-intensive computation mode has been widely recognized: applications where the data is not processed as persistent static relations but rather as transient dynamic data streams. These applications include financial analysis, network monitoring, telecommunications data management, traffic data monitoring, web applications, manufacturing, sensor networks, and so on. In this model, individual data items are represented by a record with a time-stamp. The data streams continuously arrive in multiple, rapid, time-varying, unpredictable and unbounded manners.

For this type of data, the shortcoming and drawbacks of batch-oriented data processing are obvious. Real-time query processing and in-stream processing is the immediate need in many practical applications. An important feature for a data stream processing system is that it needs to cope with the huge dynamicity in data streams in the near future, both at the architecture and the application level. At the architecture level it should be possible to add or remove computational nodes based on the current load. At the application level, it should be able to withdraw old results and take new coming data into account. However, current parallel processing systems for big data, being tailored to optimal realization of predefined analytics (static monitoring), are lacking the flexibility required for sensing and dynamic processing of changes in the data.

That is why, in recent years, many solutions for stream processing like Twitter's Storm [5], Yahoo's S4 [19], Cloudera's Impala [12], Apache Spark Streaming [21], and Apache Tez [22] appeared and joined the group of Big Data and NoSQL systems. Wherein Twitter Storm is the most successful and most widely used one.

1.4 Objectives and Contributions

The join operation is a popular problem in the big data area. It needs more than two inputs data sets, and outputs one or more required information. It has different kinds of applications, varying from kNN join (a join process for two data sets to find the k nearest neighbors for every elements in the query set) to semantic join (a join process in the semantic web area). It is a simple but not trivial and often used operation.

The challenges for designing a good parallel and distributed join method are:

- How to partition data and distribute the partitions to the computing cluster. Since more than one input data sets is involved, we need to make sure that the merge of the calculation of each partition is identical to the join of the whole data set.
- How to minimize the intermediate data communicated through network. Data transmission is unavoidable. So some clustering or classification method needs to be used to avoid unnecessary data transfers. Or some advanced data structures need to be used in order to reduce the size of data to be transferred.

More challenges for designing a good streaming join method are:

- How to withdraw old data.
- How to partition data when we do not have a global knowledge of the whole data set.

In this thesis, we are going to address the streaming join operation for data streams in a parallel and distributed way, both for *data driven join* and for *query driven join*, using two representative applications *k nearest neighbor join* and *semantic join* respectively.

The major contributions of this thesis are the following:

- **A survey of all the method for processing k nearest neighbor join in a parallel way**, including the pre-processing step, the data partitioning step and the main computation step. We analyze the performance in both a theoretical way and an experimental way.
- **Design parallel processing of k nearest neighbor join to a continuous manner for processing data streams**, after summarizing and analyzing all the technologies used for parallel processing kNN join, a parallel and continuous manner of processing kNN join on data stream is proposed. We separate the scenarios into 3 different categories, and we focus on two of them. We design data re-partition and re-computation strategies. And we implement the methods proposed on Apache Storm in order to evaluate their performance.
- **Design a parallel and distributed way to process semantic RDF joins,** both for distributing data and for decomposing queries. We use Bloom Filters as the media to transfer intermediate data. This method minimizes the communication cost among nodes. We propose a Query Topological Sort method to determine the order of communications. We then extend this method to process RDF joins in a continuous way for streams. Data expiration and re-evaluation strategies are proposed. We analyze

the performance in a theoretical way, for false positive rate, parameter tuning and efficiency. In the end, we implement the whole design on Apache Storm and design benchmarks in order to evaluate the performance.

1.5 Organization of Dissertation

This thesis studies the parallel and continuous join operation for data stream. Overall, this thesis is organized as follows:

- **Chapter 2:** introduces the state of the art of the related domains. It begins from the background of parallel computing, and firstly introduces two main forms of parallelism: data parallelism and task parallelism. Then it presents the most popular Big Data management systems, mainly systems based on the MapReduce paradigm, including Hadoop [2] and its improvements, Spark [3] and YARN [4]. The second part of the background introduces the concepts of stream processing. It first talks about the rules in data stream processing, followed by the introduction of the Sliding Window model and Sliding Window join. Then it analyzes the history about data stream management systems, focuses on the comparison of the 3 most used parallel streaming processing systems: Storm [5], Spark Streaming [21] and S4 [19]. In the end we detail the use of Apache Storm. Another important aspect of this Chapter is to introduce the use cases which will be presented in the following Chapters. Section 2.2.1 introduces the definition of k Nearest Neighbor algorithm, its applications and the traditional way of computing this algorithm. Section 2.2.2 presents the concepts of Semantic Web, including the purpose of Semantic Web, RDF data model and its corresponding query language SPARQL.

- **Chapter 3:** presents the techniques about processing data driven streaming join in a parallel and distributed manner. We choose kNN and its applications as our key use case to study. This Chapter begins with a short introduction about the methods we have evaluated, followed by an introduction of kNN join for centralized environment, parallel kNN join and continuous kNN join. In the main part of this Chapter, we first decompose the workflow of processing kNN in three steps: data preprocessing, data partitioning and main computation. Some corresponding methods are proposed for each steps. For data preprocessing, we introduce the pre-processing for reducing the dimension of data and to select the central points of data clusters respectively. For data partitioning, we discuss two different types of partition: size based partition which intends to gain a better load balance and distance based partition which tries to gather

the most relevant data inside each partition. We separate the main computation steps into two types: the ones with an intermediate computation to get local results, and those which directly calculate the global results. We then theoretically analyze each method from the perspective of load balance, accuracy and complexity. In Section 3.4, we extend the parallel methods presented previously to adapt to a computation for streams. We separate the streaming scenario into three types. New strategies about re-partition and re-computation of the streams are proposed for each type respectively. In the end, Section 3.5 presents an extensive experimental evaluation for each method, first on MapReduce and from the parallel point of view, then on Storm for evaluating the streaming characteristics.

- **Chapter 4:** introduces our techniques for processing query driven streaming join in a parallel and distributed manner. We choose RDF data and its query as our key use case to study. This Chapter begins with an introduction of the background and explains the motivation and the goal of this work, followed by a detailed state of the art on RDF processing technologies. In the main parts, we first describe query decomposition, and data distribution. We then explain our method for generating the query plan in 4 rules in Section 4.3. In Section 4.4, we extend the method proposed in Section 4.3 to work in a continuous way with sliding windows. We then analyze our methods in Section 4.5 from the aspects of main parameters of Bloom Filters, dominating parameters, and complexity points of view. The implementation issues are discussed in Section 4.6, where the algorithms for finding the join vertices, judging the category of join vertex, Query Topological Sort and Sliding Bloom Filters, are shown. Finally, we evaluate our method in Section 4.7 with both synthetic and LUBM [14] benchmarks.

- **Chapter 5:** reviews the contributions and presents some research and development perspectives that may arise from this thesis.

Chapter 2

Background and Preliminaries

2.1 Background

2.1.1 Parallel Computing

Parallel computing is a computation model which uses two or more processors (cores or computers) in combination to perform multiple operations concurrently. The basic condition for parallel computing is that in general, a large problem can be divided into a limited number of smaller problems, and these small problems can be handled simultaneously.

Unlike the traditional serial computation, parallel computing uses multiple resources simultaneously to solve a problem. The problem should first be cut into a series of instructions, which will later be executed simultaneously on different processors. This model is much more suitable for explaining, modeling and solving complex real world phenomena. It does not only speed up the time spent to perform a large task, but also makes it possible to process large-scale data sets or complex problems which cannot be handled by a single machine.

In parallel computing, there are mainly two forms of parallelism:

- Data Parallelism
- Task Parallelism

We will introduce them separately in the coming two sections.

2.1.1.1 Data Parallelism

Data parallelism focuses on distributing data across different parallel computing resources, in which the same computation is applied to multiple pieces of data. This is usually used for data-intensive tasks. Fig. 2.1 shows the schematic of data parallelism.

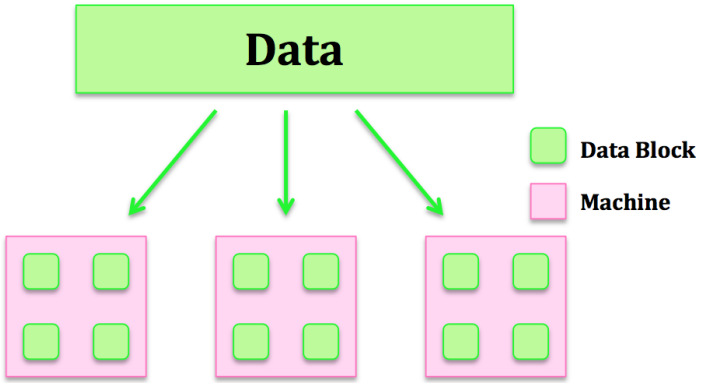


Fig. 2.1 Data Parallelism Schematic

In a data parallelism process, the data set is first divided into partitions, which will later be processed by different processors using the same task. This simple idea makes the storing and handling of big data possible. For example, Facebook has several million photos uploaded each day. But these photos are too large to be stored in a single machine. Then a data parallelism strategy is suitable for this problem.

However, because each machine only has a subset of data, gathering the results together is a problem that this model needs to address. At the same time, the main factor affecting the performance of this model is the transmission of intermediate data, hence reducing the amount of data to be transferred is another problem to face.

Since data parallelism emphasizes the parallel and distributed nature of data, when the size of data is growing, it is inevitable to use this model in parallel computing. Examples of Big Data frameworks that uses data parallelism are: Hadoop MapReduce [2], Apache Spark[3], YARN[4], and Apache Storm[5].

2.1.1.2 Task Parallelism

Task parallelism focuses on distributing tasks concretely performed by processors across different parallel computing resources, in which the same data (or may be different data in a hybrid system) is processed by different tasks. This is usually used for computation-intensive tasks.

In a task parallelism process, the parallelism is organized around the functions to be run rather than around the data. It depends on task decomposition. This idea makes it possible to handle a complex problem. For example, in a semantic join, task 1 needs to save the data which meets a certain condition in a specific data structure, and task 2 needs to use the data which meets another condition to probe this data structure. This process can be considered as a task parallel process.

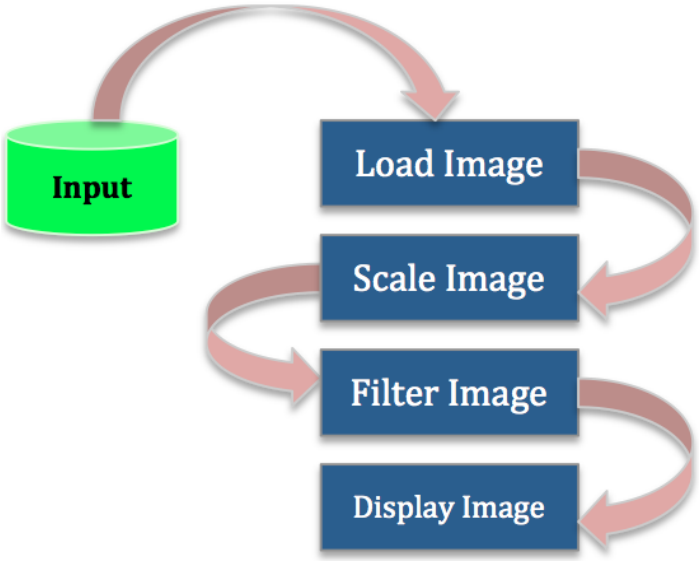


Fig. 2.2 Image Pipeline Processing

The difficulties of this type of process lies first on the decomposition of the work, specifically the decomposition of queries in a join process. Also task parallelism processes usually suffer from bad load balancing, since it is not easy to divide tasks with equal complexity. The communication among tasks is another problem. Synchronization is the most important communication approach in task parallelism processes, and can be divided into thread synchronization and data synchronization. Thread synchronization focuses on determining the order of execution in order to avoid Data Race Condition problems. Data synchronization is mainly used to ensure the consistency among multiple copies of data.

The most common task parallelism is pipelining. Suppose you have multiple tasks, task I, task II and task III, instead of having each one operating on the data independently, pipelining takes the data and first give it to task I, then task II and finally task III. Image processing often chooses to use a pipeline. Images are coming in a stream, some of the processing starts with the first task, and applies a certain filter on the images, then passes on to the second task, and so on. This is a very common combination of task parallelism and data parallelism. An example of pipeline processing of images is shown in Fig. 2.2.

Recently, the most popular task parallelism example is deep learning. Deep learning is a branch of machine learning which is based on a set of algorithms which attempt to model high-level abstractions in data by using multiple processing layers. The difference between deep learning and traditional machine learning is that in deep learning instead of having one model to train all the data, we separate the model into layers, and each layer can be considered as a sub-task of the whole model.

Task parallelism and data parallelism complement each other, and they are often used together to tackle large-scale data processing problems. The Big Data frameworks that uses task parallelism are: Apache YARN [4] and Apache Storm [5]. They are hybrid systems which support both task and data parallelism.

2.1.1.3 Big Data Management Systems

MapReduce is a flexible and scalable parallel and distributed programming paradigm which is specially designed for data-intensive processing. It was initially introduced by Google [70] and popularized by the Hadoop framework.

The concept of MapReduce has been widely known since 1995 with the message passing Interface (MPI) [78] standard, having reduce [1] and scatter operations [2]. The MapReduce programming model is composed of a Map procedure and a Reduce procedure. The Map task is usually used for performing some preliminary and cleaning work such as filtering and sorting. For example we can use a Map task to sort the students by alphabetical order of their surname, and then filter the students whose score is below a certain level.The Reduce task is used to perform a summary operation such as count or aggregation. For example we can use a Reduce task to count the number of students whose score is above a given level.

The idea of the MapReduce paradigm comes from high-order functional programming, where Map and Reduce are two primitives. In this paradigm every record is represented by a $< key, value >$ pair. The Map function processes a fragment of $< key, value >$ pairs in order to generate a list of intermediate $< key, value >$ pairs. Each $< key, value >$ pair is processed by the same map function on different machines without depending on other pairs. The output keys of the Map tasks could be either the same as the input keys or different from them. The output $< key, value >$ pairs have an information of partition which indicates to which Reduce task this pair needs to be sent. The partition information makes sure that all pairs with the same key can be later sent to the same Reduce task. The Reduce function gathers the outputs of the same partition from all map tasks together through a Shuffle phase and merges all the values associated with the same key, then produces a list of $< key, value >$ pairs as output.

Hadoop [2] is an open-source framework written in Java for distributed storing and processing large scale data sets. The core of Hadoop contains a distributed storage named Hadoop Distributed File System (HDFS), the MapReduce programming paradigm. HDFS is a distributed, scalable, and portable file-system written in Java. It stores large files across multiple machines on a cluster. Its reliability is achieved by replicating the data among

[1] http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/
[2] http://mpitutorial.com/tutorials/performing-parallel-rank-with-mpi/

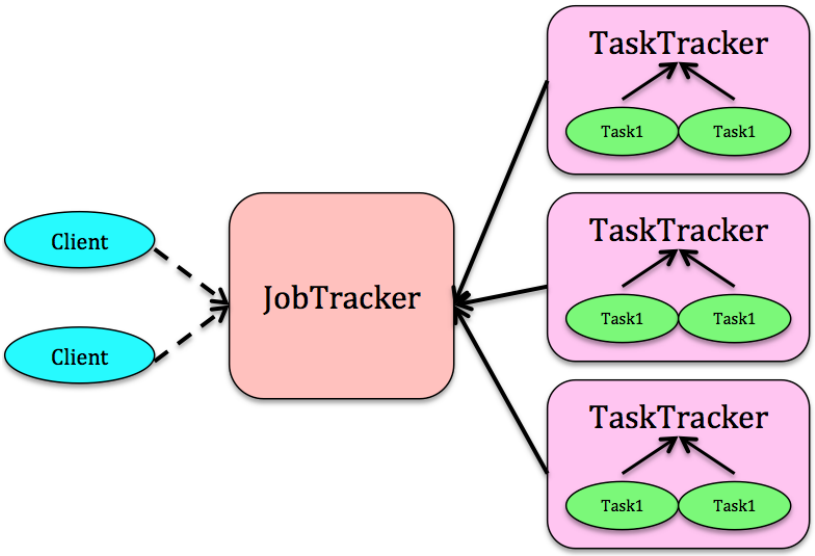


Fig. 2.3 Physical Structure of Hadoop System

multiple nodes. The default number of replications is set to 3, which means the same piece of data is stored on three nodes. It is very popular not only among the academic institution but also in many companies such as web search, social network, economic computation and so on. A lot of research work focuses on optimizing Hadoop performance and its efficiency in many different aspects [77] [138] [124].

The whole system of Hadoop works in a master-slave manner, with JobTracker as the master, and the other nodes as slaves. A TaskTracker daemon runs on each slave node. The JobTracker daemon is responsible for resource allocation (e.g. managing the worker nodes), tracking (e.g. resource consumption or resource availability) and management (e.g. scheduling). The TaskTracker has much more simple responsibilities. It is in charge of launching tasks with an order decided by the JobTracker, and sending the task status information back to JobTracker periodically. The schematic of this process is shown in Fig. 2.3.

When running a Hadoop job, input data will first be divided into some splits (64M by default). Then each split will be processed by a user-defined map task.

So the whole process of a Hadoop job as shown in Fig. 2.4 can be summarized as follow:

- **Step 1:** Split data into blocks (64M by default)
- **Step 2:** Map Phase: Extract information from data (filter, sort)
- **Step 3:** Shuffle Phase: Exchange data through network from Map Phase to Reduce Phase

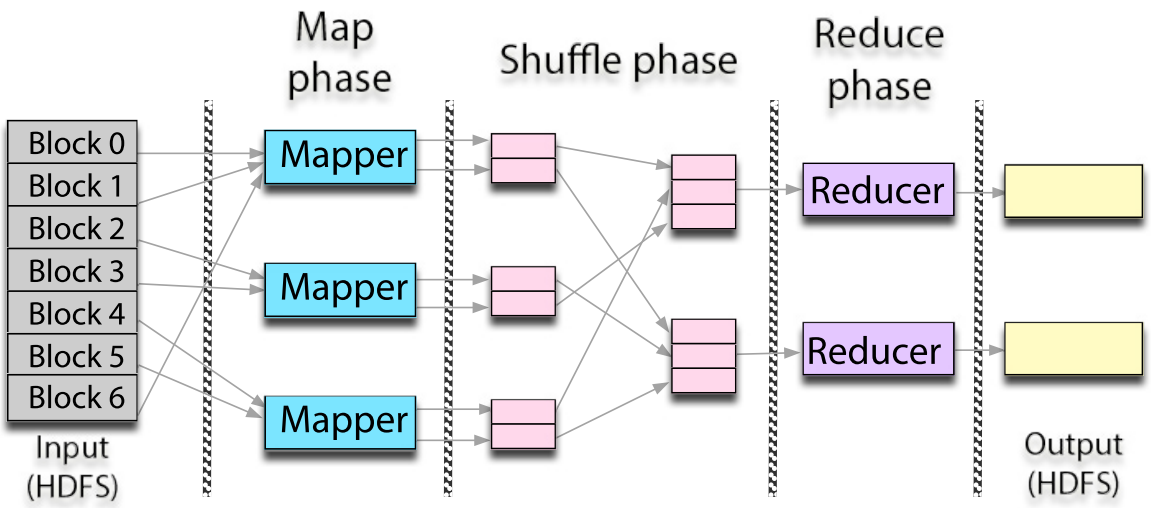


Fig. 2.4 Logical View of Hadoop Framework

- **Step 4:** Reduce Phase: Summary operation (count, aggregation)

The previous introduction on Hadoop shows that a Hadoop MapReduce job has some special characteristics as shown below,

- **Execution Similarity:** According to the programming model, users only have to provide a map function and a reduce function. And the execution for each Map task (or Reduce task) is very similar to others. In other words, all data will be processed by these functions repeatedly. Thanks to this design, we only need to study how each $< key, value >$ pairs are processed for a particular job, as reading, sorting, transferring and writing data are independent of these two functions.

- **Data Similarity:** MapReduce is well suited for off-line batches processing. And it is usually used to do repeated work in which the input data has very similar format, such as log analysis, inverted index and so on. We can just take a look at a small sample and then we can estimate the whole dataset.

Hadoop is now a very mature system, with specific application and user groups. However, due to the limitation of the MapReduce paradigm and the Hadoop implementation, it has performance limitations in some application scenarios. In order to better integrate Hadoop in the applications, many works have been done from the very beginning to extend Hadoop and to improve its performance. We discuss a limited number of them.

The first type of effort intends to improve the performance of Hadoop by predicting its performance and tuning the parameters.

In our previous work [130], we have introduced a Hadoop performance prediction model. This model is implemented through 2 parts, a job analyzer and a prediction module. The job analyzer is in charge of collecting the important properties related to the jobs for the prediction module. Then, the prediction module will use this information to train a locally linear model based on locally weighted regression. This work can predict job performance metrics such as map task execution time, reduce task execution time, network communication etc. The overhead (i.e. the cost of computing the prediction) is low. Moreover, when the amount of data augments the overhead decreases. This work can not only help us to tune the parameters when writing a MapReduce job, but also help us to find out the bottlenecks of the Hadoop framework.

WaxElephant [127] is a Hadoop simulator. It proposes a solution to address two challenging issues for a large-scale Hadoop clusters. The first one is to analyze the scalability. The second one is to identify the optimal parameters of configurations. WaxElephant has 4 main capabilities. It can firstly load real MapReduce workloads derived from the historical log of Hadoop clusters and replay the job execution history. It can then synthesize workflows and execute these workflows based on some statistical characteristics of the workfloads. Its main functionality is to identify the optimal parameters of the configurations. Finally, it can analyze the scalability of the cluster.

The second type of effort aims at extending Hadoop to have database-like operations.

Hive [134] plays a role of data warehouse on top of Hadoop. It originates from Facebook. It provides a SQL-like interface to operate the data stored in a Hadoop cluster. It offers operations like select, join, etc.

HBase [81] is a column-oriented No-SQL database running on HDFS. It fills up the lack of immediately reading and writing operations in HDFS.

Yahoo! used Hadoop clusters to do data analysis tasks. They created a new data processing environment called Pig [120], and its associated query language Pig Latin, whose target is to provide a MapReduce style programming, a SQL style programming as well as the ability to control the execution plan together. It offers high-level data manipulation primitives such as projection and join in a much less declarative style than in SQL.

The third type of work proposes to combine with other programming language or model.

Ricardo [67] is a scalable platform for deep analytics. It decomposes data analysis algorithms into parts executed by the R statistical analysis system and parts handled by the Hadoop cluster. This decomposition attempts to minimize the transfer of data across the system. It avoids re-implementing either statistical or data-management functionality, and it can be used to solve complex problems.

Paper [144] presents three contributions towards tight integration of Hadoop and Teradata EDW. They provide fast parallel loading of Hadop data to Teradata EDW. And they also make MapReduce programs efficient and direct parallel access to Teradata EDW data without external steps of exporting and loading data from Teradata EDW to Hadoop.

HadoopDB [29] combines MapReduce with existing relational database techniques. It is therefore a hybrid system of parallel DBMS and Hadoop approaches for data analysis, achieving the performance and efficiency of parallel databases and yielding the scalability, fault tolerance and flexibility at the same time. It is flexible and extensible for performing data analysis at large scales.

The Nephele/PACTs programming model [33] is a generalization of the MapReduce paradigm and extends MapReduce by adding additional second-order functions with additional plug-in points for developing parallel applications.

Apache Spark[3] is another popular parallel computing framework after Hadoop MapReduce. Spark provides an application programming interface in Java, Scala, Python and R on a data structure called the resilient distributed dataset (RDD). Spark also uses the MapReduce paradigm but it overcomes the limitations in MapReduce. Hadoop MapReduce forces a particular linear data flow, it reads input data from disk, maps a function across the data, reduces the results of the map, and stores reduction results on disk. The resilient distributed dataset structure works as a working set for distributed programs, it offers a restricted form of distributed shared memory. Unlike Hadoop jobs, the intermediate data of Spark can be saved in memory, which avoids the unnecessary reading and writing from HDFS. Therefore Spark is better for data mining and machine learning algorithms, which require iterations. RDD facilitates the implementation of iterative algorithms which need to visit the dataset multiple times in a loop. It also makes it easy to do interactive or exploratory data analysis, like repeated database-style querying of data.

Spark requires a manager which is in charge of the cluster, and a distributed file system. Spark also supports a pseudo distributed mode (local mode), which is usually needed for testing.

Compared with Hadoop, Spark has the following advantages:

- Store intermediate data into memory, providing a higher efficiency for iterative operations. So Spark is more suitable for Data Mining and Machine Learning algorithms containing a lot of iterations.

- Spark is more flexible than Hadoop. It provides many operators like: map, filter, flatMap, sample, groupByKey, reduceByKey, union, join, cogroup, mapValues, sort,

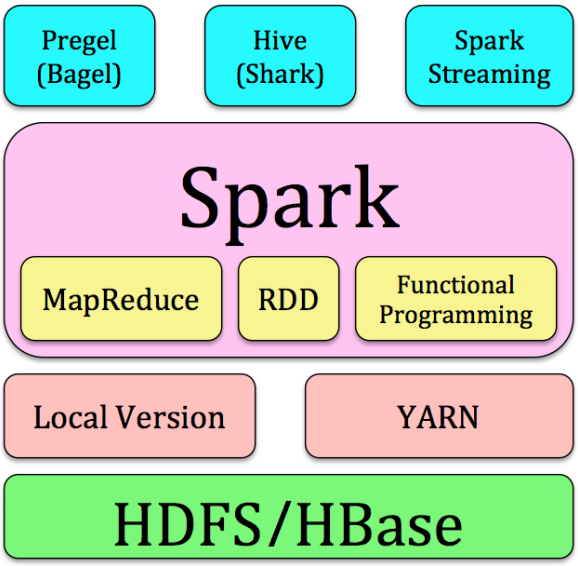


Fig. 2.5 Structure of Spark System.

partionBy, while Hadoop only provides Map and Reduce. However, due to the characteristics of RDD, Spark does not perform well on the fine-grained asynchronous update applications [152] or the applications with incremental changes, such as the web crawlers with updates.

- By providing a wealth of Scala, Java, Python APIs and interactive Shell API, Spark has a higher availability with different programming languages and different modes to use.

According to the characteristics of Spark, its applicable scenarios are:

- Iterative calculations requiring multiple operations
- Applications that require multiple operations on a specific data set

And the benefit increases with the amount of data and the number of operations. But the benefit is smaller in applications with a small amount of data and intensive computations.

The structure of a Spark system is shown in Fig. 2.5.

YARN [4] is an attempt to take Apache Hadoop beyond MapReduce for data-processing. As we explained above, in Hadoop, the two major responsibilities of the JobTracker are resource management and job scheduling or monitoring. As there is only one JobTracker in the whole system, it becomes a bottleneck. The fundamental idea of YARN is to split these functions into two separate daemons — a global ResourceManager (RM) and a per-application ApplicationMaster (AM).

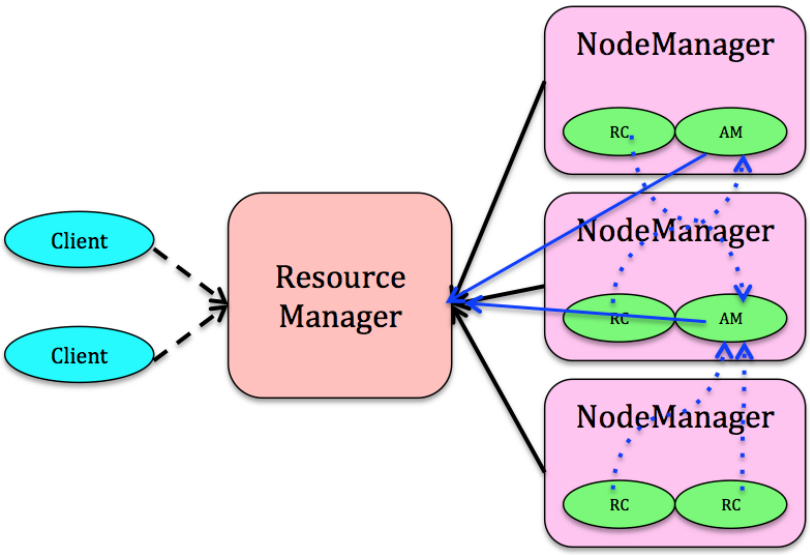


Fig. 2.6 Architectural View of YARN

The ResourceManager together with the per-node slave daemon NodeManger forms a new generic system for managing tasks in a distributed manner. Moreover, the ResourceManager is the ultimate authority that arbitrates resources among all applications in the system, while the per-application ApplicationMaster is a framework specific entity and is used to negotiate resources from the master ResourceManager and the slaves NodeManagers to execute and monitor the tasks. A pluggable Scheduler is used in the ResourceManager to allocate resources to jobs. The Scheduler works using an abstract concept of Resource Container (RC) which incorporates resource elements such as CPU, Memory, Disk, Network etc. The NodeManager is a per-node slave daemon, and its responsibility is to launch the tasks and to monitor the resources (CPU, Memory, Disk, Network). From the system perspective, the ApplicationMaster runs as a normal container. An architectural view of YARN is shown in Fig. 2.6.

Hadoop, **Spark** and **YARN** all use the **MapReduce** paradigm as their abstract computational concept. The ecosystem of MapReduce and its derivative methods are mature and very good for parallel processing of big data. But most of them are still an 'offline' processing platform, which means that they can not handle dynamic data streams.

2.1.2 Stream Processing

As we introduced in Section 1.1, the characteristics of Big Data contains 4 Vs, among which Volume, Variety and Velocity are the most important. The relations among these 3 Vs are shown in Fig. 2.7.

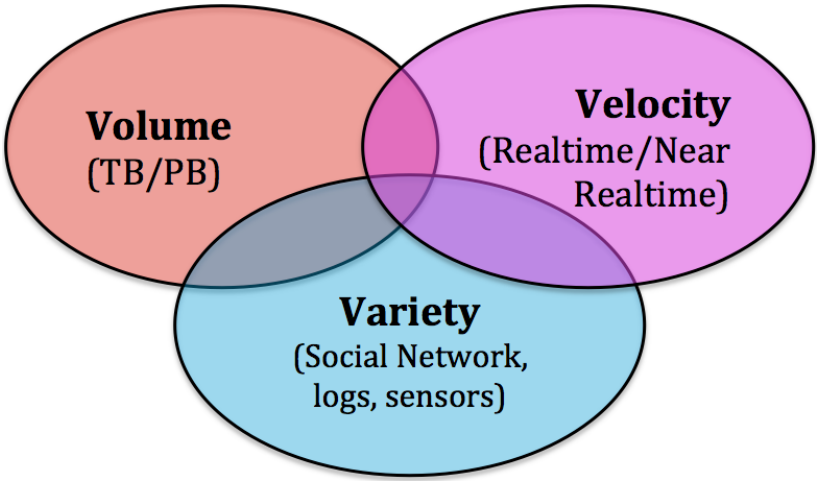


Fig. 2.7 The 3 Vs of Big Data

The need of velocity requires to process the data fast, so that the system can react to the changing conditions in real time. This requirement of processing high-volume data streams with low-latency becomes increasingly important in the areas of trading, fraud detection, network monitoring, and many other aspects, thus increasing the demand for stream processing. Under this requirement, the capacity of processing big volumes of data is not enough, we also need to react as fast as possible to the update of data.

Stream processing is a programming paradigm, which is also called dataflow programming or reactive programming. A stream is a sequence of data and a series of operations will be applied to each element in the stream. Data items in streams are volatile, they are discarded after some time. Since stream processing often involves large amount of data, and requires the results in real-time, the stream processing platforms (e.g. Apache Storm) often work in parallel. Besides, this paradigm is a good complement of parallel processing, and allows applications to more easily exploit a limited form of parallel processing. It simplifies parallel processing by restricting the parallel computation that can be performed.

Traditional popular big data processing frameworks like Hadoop and Spark assume that they are processing data from a database, i.e. that all data is available when it is needed. And it may require several passes over a static, archived data image. But when data arrives in a stream or streams, data will be lost if it is not processed immediately or stored. Moreover, in the streaming scenarios, usually the data arrives so rapidly that it is not feasible to store it all in a conventional database, to process it when needed. So stream processing algorithms often rely on concise, approximate synopses of the input streams in real time computed with a simple pass over the streaming data.

Below, we will first present the issues in data stream processing, then introduce an approach to summarize a stream with only looking at fixed-length windows, and in the end we will present some Stream management systems.

2.1.2.1 Rules in Data Stream processing

Definition 2.1 *A **data stream** is a real-time, continuous, ordered (explicitly by timestamp or implicitly by arrival time) sequence of items.*

In general, we need to follow some rules for processing low-latency and high-volume data streams [131]. The most important rules are:

- **Rule 1: Keep the data moving**
  The first requirement for a real-time high-volume data stream processing framework is to process data "on the fly", without storing everything.
- **Rule 2: Handle Stream Imperfections**
  The second requirement is to provide resilience against "imperfections" in streams, including delay, missing and out-of-order data.
- **Rule 3: Generate Predictable Outcomes**
  A stream processing engine must guarantee predictable and repeatable outcomes.
- **Rule 4: Integrate Stored and Streaming Data**
  A stream processing system also needs to efficiently store, access, and modify state information, and combine it with new coming streaming data.
- **Rule 5: Guarantee Data Safety and Availability**
  Fault-tolerance is another important point for such a system.
- **Rule 6: Partition and Scale Applications Automatically**
  In order to meet the real-time requirement for high-volume and fast data streams, the capability to distribute processings across multiple machines to achieve incremental scalability is also important. Ideally, the system should automatically and transparently distribute the data and queries.
- **Rule 7: Process and Respond Instantaneously**
  The last but most important requirement is to have a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications.

When designing a stream processing algorithm, we need to keep two things in mind:

- It can be more efficient to get an approximate answer than an exact solution.
- A variety of techniques related to hashing turn out to be very useful. Because, these techniques introduce useful randomness, which produces an approximate answer that is very close to the exact one.

The approximation of a streaming algorithm comes from two aspects: (1) from limiting the size of states maintained for the process; (2) from reducing the precision of the result. An approximate solution is defined as follows:

Definition 2.2 *An* ε***-approximation*** *solution is a deterministic procedure that, given any positive* $\varepsilon < 1$*, computes an estimation* $\widehat{X}$ *for X which worst case relative error is at most* ε*.*

Definition 2.3 *An* (ε, δ)***-approximation*** *solution for a quantity X is a randomized procedure that, given any positive* $\varepsilon < 1$ *and* $\delta < 1$*, computes an estimate* $\widehat{X}$ *which is within a relative error of X with the probability at least 1 -* δ*.*

The algorithms for processing streams usually involve summarization of the stream in some ways. Summary data structures such as : wavelets, sketches, histograms and samples have been widely used especially for streaming aggregation [30][61][83] [79][58][139] [114][85][71]. These algorithms always begin by considering how to make a useful sample or how to filter tout most of the undesirable elements. Another important approach to summarize a stream is to process within a fixed-length window [133][66][57], then query the window as if it were a relation in a database. This model is called "sliding window model", the details about this model will be presented in the coming section.

A lot of prior work on stream processing focused on developing space-efficient, one-pass algorithms for performing a wide range of centralized, one-shot computations over massive streams. These applications involve: (1) computing quantiles [88]; (2) estimating distinct values [82]; (3) counting frequent elements [60][65]; (4) estimating join sizes and stream norms [34][64].

As the size of data is getting larger, some recent efforts have concentrated on distributed stream processing and proposing communication efficient streaming frameworks to handle a number of query tasks such as aggregation, quantiles and join (such as Apache Storm, Spark Streaming, Yahoo! S4 etc.) which we will introduce Section 2.1.2.4.

2.1.2.2 Sliding Window Model

Through the analysis above, we can list the 3 most important issues arising in stream processing:

(1) Unbounded streams can not be wholly stored in bounded memory.

(2) New items in a stream are often more relevant than older ones, because streams are temporally ordered. So outdated data should be withdrawn and no longer used when evaluating queries.

(3) The query plans for streams may not use blocking operators that must consume the entire input before any result is produced.

To solve these issues a common solution is to restrict the range of continuous queries to a sliding window [87]. This process can be considered as maintaining a moving window of the most recent elements in the stream. A sliding window protocol is a packet-based data transmission protocols. It is used for reliable in-order delivery of data. Conceptually, each portion of the transmission of data is assigned a unique consecutive sequence number, which is later used to reorder the data received.

There are two types of sliding windows:

- **Count-Based Sliding Window:** also called sequence-based sliding window, which contains the last T items.
- **Time-Based Sliding Window:** which contains the items that have arrived in the last t time units.

Computing all queries within a sliding window allows continuous queries over unbounded data streams to be executed with finite memory. This execution generates new results incrementally as new items arrive. Furthermore, a windowed process over streams is practical and useful in many applications.

Performing a continuous process over sliding windows poses two strategies need to be well designed, which will affect the performance and efficiency of the algorithms:

- Re-execution Strategies
- Data Invalidation Strategies

Each strategy has two choices:

- **Eager Re-execution Strategies:** generates new results after each new data arrives. This strategy may be not feasible when streams have a high arrival rate.

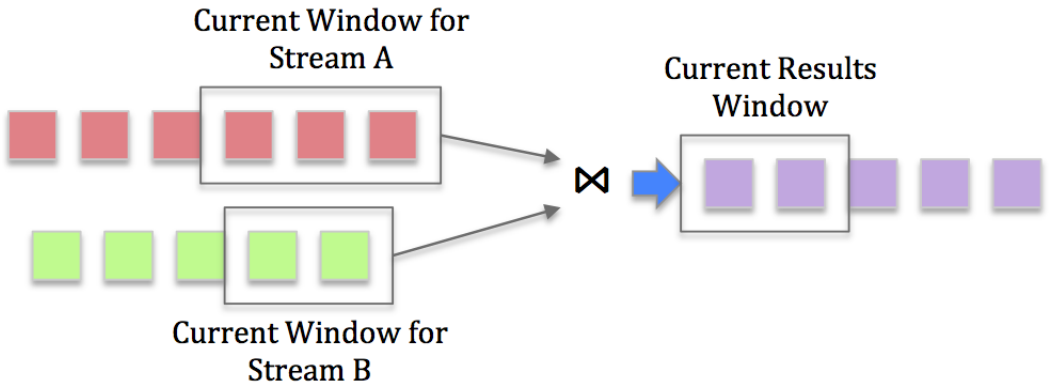


Fig. 2.8 Streaming Join over Sliding Window

- **Lazy Re-execution Strategies:** is a more practical solution, which re-executes the query periodically. The weak point of this strategy is that it causes an increased delay between the arrival of new data and the generation of new results based on this new data.

- **Eager Expiration Strategies:** proceed by scanning the sliding window, moving forward upon arrival of new data and removing old data at the same time.

- **Lazy Expiration Strategies:** involve removing old data periodically and require more memory to store data waiting for expiration.

Several algorithms have been proposed for maintaining different types of statistics over data streams within a sliding window while requiring time and space that is sublinear (typically, poly-logarithmic), in the same or different window sizes [68][84][125][135]. Clustering problems over sliding windows [53] [121] is a new trend of research in the machine learning area.

2.1.2.3 Sliding Window Join

Sliding window joins have been widely studied. A sliding window join uses two (or more) streams of data as input, with window sizes for each stream, as shown in Fig. 2.8. The output is also a stream, containing all pairs (a, b), where a $\in$ CurrentWindow(A), b $\in$ CurrentWindow(B), such that:

(i) a and b satisfy the join predicate.

(ii) a is in the current active window of A as well as b is in the current active window of B.

In a stream system, joins are only allowed when the state does not grow indefinitely. The join condition must ensure that a data of one input stream only joins with a bounded range of data from the other input streams. At the same time, a streaming join must have an equality or band predicate [112] between progressing attributes of its inputs. Having n data streams, and n corresponding sliding windows for each stream, the object of a sliding window join is to evaluate the join operation of the n windows. The semantics of a sliding window join is a monotonic query over append-only relations. For each newly arrived data s, the join operation needs to probe all unexpired data and present in the sliding window at the arrival time of s, and return all the results that satisfy all the join predicates. And s will be kept in the window until it expires, and joined with other new coming data.

A binary incremental Nested Loop Join example is introduced in [105]. In this example two sliding windows S_1 and S_2 are joined. For each newly arrived data in window S_1, a scan of S_2 will be done to return the matching results. The same procedure is applied to each newly arrived data in window S_2. Paper [87] generalizes this model to more than two windows. In this paper, for each newly arrived data t, an execution of join sequence according to the query plan will be done. For example, suppose 3 sliding windows S_1, S_2 and S_3 need to be joined according to the query plan $S_1 \bowtie (S_2 \bowtie S_3)$. Upon arrival of new data in S_1, after invalidating expired data in S_2 and S_3, we need to probe all the data in $S_2 \bowtie S_3$. If a new S_2 (or S_3) data arrives, then we need firstly to withdraw the old data in S_1 and S_3 (or S_2), and compute the join of the newly arrived data with the data in the current window of S_3, then probe the results for each data in the current window of S_1.

Parallel and distributed sliding window models have also been proposed. Paper [109] presents a randomized $(\varepsilon, \delta)-$approximation scheme for counting the number of 1 in a sliding window on the union of distributed streams. Paper [122] uses the sliding window model for answering complex queries over distributed and high dimensional data streams. They proposes a compact structure ECM-sketches, which combines the state-of-the-art sketching technique for data stream summarization with deterministic sliding window synopses. This structure provides probabilistic accuracy guarantees for the quality of the estimation, for point queries and self-join queries, which can be applied for finding heavy hitters, computing quantiles, or answering range queries over sliding windows. Paper [84] proposes a Parallel Sliding Windows (PSW) method, for processing very large graphs. PSW requires only a very small number of non-sequential accesses to the disk. And it naturally implements the asynchronous model of computation. Paper [146] studies the processing of asynchronous event streams within sliding windows. They characterize the lattice structure of event stream snapshots within the sliding window, then propose an only algorithm to maintain Lat-Win at runtime.

But to the best of our knowledge, a parallel and distributed sliding window model for processing data driven complex Nested Loop Join such as kNN join, or for processing query driven semantic join have not been studied yet.

2.1.2.4 Data Stream Management System

Some works have extended the MapReduce paradigm to process dynamic data.

ASTERIX [45] is a data-intensive storage and computing platform which deals with data-feeds from multiple sources of data. It follows a "web warehousing" philosophy [35] where social network or web data is ingested into and analyzed in a single scalable platform. It also builds a logical plan for each query, which is a directed acyclic graph (DAG) of algebraic operators that are similar to what one might expect to find in a nested-relational algebra such as select, project, join, groupby, and other operations over streams of data.

Hadoop Online Prototype (HOP) [63] is an approach for getting early results from a job while it is being executed by a flush API.

HaLoop [55] [56] is a novel parallel and distributed system that supports large-scale iterative data analysis applications. It is built on top of Hadoop and extends it with a new programming model and several important optimizations that include (1) a loop-aware task scheduler, (2) a loop-invariant data chaching, and (3) caching for efficient fixpoint verification.

Twister [75] is a distributed in-memory MapReduce runtime optimized for iterative MapReduce computations. It performs and scales well for many iterative MapReduce computations.

PIC [76] stands for Partitioned Iterative Convergence. It was designed for processing iterative algorithms on clusters.

Continuous Hadoop [140] is a framework to support continuous MapReduce applications. In this framework, jobs registered by the users can be automatically re-executed when new data is added to the system. New data is identified using a time stamping mechanism. The new coming data is produced by a function called carry in the reduce phase and is automatically added as an input for the subsequent run.

But these frameworks are not suitable for processing data streams and returning the results in real-time because of the nature of MapReduce paradigm.

To address the limitation of database-based solutions, some stream processing engines have been proposed. Here we list some of the most notable ones.

Aurona [25] is a Stream-Oriented Database to manage data streams for monitoring applications. It provides storage organization, real-time scheduling, introspection, and load shedding.

Borealis [26] is a distributed stream processing engine. It is based on Aurora for the core stream processing functionality and Medusa for the distribution functionality. It modifies and extends both systems in non-trivial and critical ways to provide advanced capabilities that are commonly required by newly-emerging stream processing applications.

StreamCloud [90] is a scalable and elastic stream processing engine for processing large data streams. It uses a novel parallelization technique that splits queries into subqueries that can be allocated to independent sets of nodes while minimizing the distribution overhead. It works in a shared nothing cluster.

STREAM [37] is the STandford stREam datA Manager. It provides a general-purpose system for processing continuous queries over multiple continuous data streams and stored relations. It is designed to deal with high-volume and bursty data streams operated by complex continuous queries. It introduces a declarative language(CQL [38]) to specify queries.

TelegraphCQ [59] provides a suite of novel technologies for continuously adaptive query processing. It is focused on meeting the challenges that arise in handling large streams of continuous queries over high-volume, highly-variable data streams. It is based on individual modules that communicate using the Fjord API. Its modules are generic units that produce and consume data.

SPADE (IBM) [80] is a large scale, distributed data stream processing middleware under development at IBM. It provides a programming language for flexible composition of parallel and distributed data-flow graphs, a toolkit of stream processing operators and a a rich set of stream adapters.

Spark Streaming is an extension of the Spark framework. It enables scalable, high-throughput and fault-tolerant stream processing for dynamic data streams. Spark Streaming receives input data streams and divides them into batches, which will later be processed by the Spark engine to generate final stream in batches. It has APIs for Scala, Java and Python. Spark Streaming is a data parallelism framework. It follows the same ideas as MapReduce batch processing paradigm. It is not a real "real-time processing" framework: the incoming events are cached and processed as a batch, resulting in a larger delay than the real streaming processing frameworks such as Twitter Storm.

Yahoo! S4 [118] is a general-purpose, distributed, scalable, partially faut-tolerant, pluggable platform for data stream processing that allows users to easily develop applications for processing continuous unbounded streams of data. It is a java based solution which relies on the user defined classes to process and produce stream data. It uses Apache Zookeeper [97] to maintain the state of a distributed job. It allows a parallel execution of data streams. But it does not provide a dynamic load balancing protocol, which is left for users to define.

Table 2.1 A comparison of Storm, Spark Streaming and Yahoo! S4

	Storm	**Spark Streaming**	**S4**
Original Design	Twitter	UC Berkeley	Yahoo!
Implemented In	Clojure	Scala	Java
APIs	Java (and others)	Scala, Java	Java
Processing Model	Record at a time	Mini Batches	Record at a time
Latency	Sub-Second	Few Seconds	Sub-Second
Data Units	Tuple	Java Object	Java Object
Hadoop Distribution	Hortonworks HDP	Cloudera, MapR	None
Resource Manager	Mesos/Zookeeper	YARN/Mesos	Zookeeper
Distributing Work	User Specifies	MapReduce	Evenly
Fault Tolerance (Ever Record Processed:)	At Least Once	Exactly Once	No Guarentee
Dynamic Deployment	Yes	No	No

S4 uses "Plain Old Java Objects" (POJO) mode as its communication protocol and User Datagram Protocol (UDP) as its underlying protocol, which has an impact on reliability. Besides, it does not support dynamic deployment of the cluster or add or delete nodes during run time.

Twitter Storm [137], is a distributed, parallel and fault tolerance framework for data streams processing. Queries can be expressed in using the boxes and arrows model. A Storm job, which is called a Topology, consists in two components: Spout and Bolt . Spout nodes are responsible for generating the system input streams. Bolt nodes are in charge of processing those streams and generate output results. It relies on Zookeeper servers to maintain the state of distributed setups. Storm supports task parallelism. In a Storm cluster, the data is flowing while the tasks do not. Storm is a real "flow processing" framework: every incoming data will be handled as an event, which has a smaller delay than the mini-batch processing frameworks such as Spark Streaming. It also supports dynamic deployment of the cluster, and add or delete nodes during run time.

Among these frameworks, **Spark Streaming**, **Yahoo! S4** and **Twitter Storm** are the most widely used. Table. 2.1 shows a comparison of these three frameworks.

Through the above comparison, we think Storm is the best choice for a real-time and low latency data stream processing problem. The reasons are:

- Storm processes data in form of streams. And it processes intensive queries in parallel. (Latency)
- It is scalable. As the amount of data increases, we can simply increase the number of nodes for processing. (Dynamic Deployment)
- It has a good reliability, which can ensure that each event can be at least processed once. (Fault Tolerance)
- It provides good fault tolerance. Once a node fails, the task on this node is re-assigned to the other nodes. (Fault Tolerance)
- It provides a simple programming model, which reduces the complexity of real-time processing. (Distributing Work)
- It supports multiple programming languages such as Clojure, Java, Ruby and Python. (APIs)

For the above reasons, we chose to use Storm to evaluate the algorithms designed in this thesis.

2.1.2.5 Introduction to Apache Storm

Storm is a distributed, reliable, fault-tolerant framework for data streams processing. It was firstly developed in Twitter, and is now an Apache open source project.

A Storm job is called a **Topology**. A Topology is made of different types of components. Each component is responsible for a simple specific task. The component for handling and distributing the input streams is called a **Spout**. The component for processing the streams is called a **Bolt**. A Spout passes data to Bolt(s), which transforms it in some ways. A Bolt is a user defined task, it can either persist the data into different sort of storages, or pass it to some other Bolts, or process it through some user specified functions. A Storm cluster can be seen as a chain of a Spout followed by several Bolts, where each Bolt makes some kind of transformation on the data. An example of a Topology is shown in Fig. 2.9.

A Storm cluster works in a master slave manner. A daemon called Nimbus runs on the Master node. It is responsible for distributing the user specified code to the cluster: assigning tasks to worker nodes, and monitoring the cluster for failures. A daemon called Supervisor runs on the Worker nodes. It executes the task on this node and is part of the Topology. Usually, a Topology runs across many worker nodes. Storm keeps the states of the cluster in Zookeeper or on local disk. So the daemons are stateless. They can fail or restart without

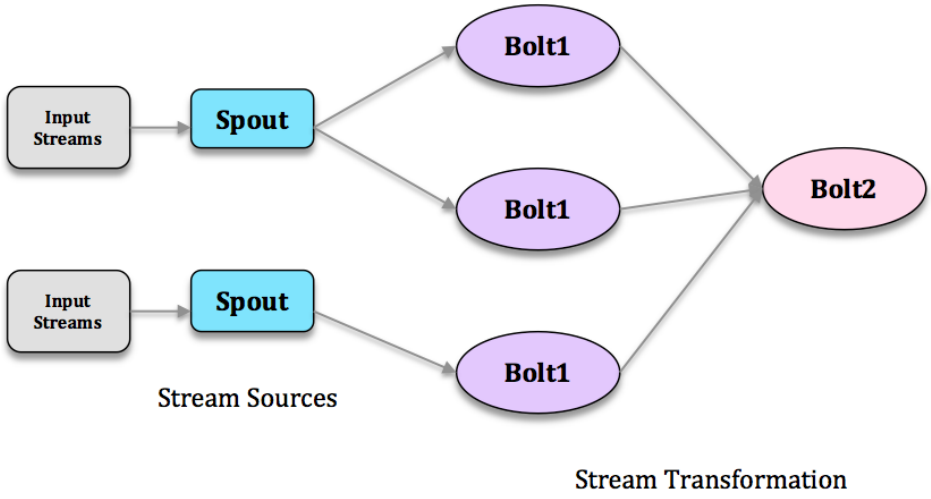


Fig. 2.9 An example of a Topology

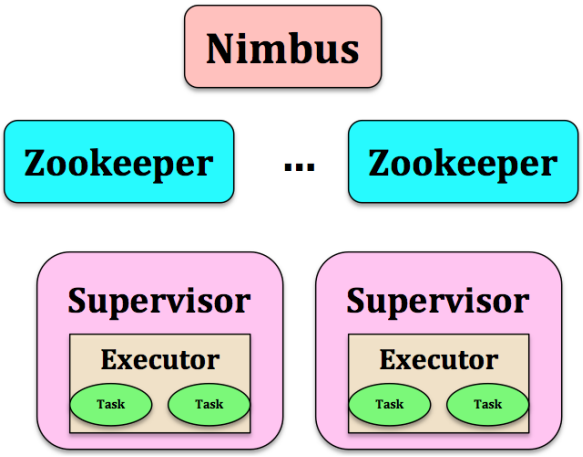


Fig. 2.10 A physical view of a Storm cluster

affecting the health of the whole system. The physical view of a Storm cluster is shown in Fig. 2.10

2.2 Concepts of use cases

2.2.1 k Nearest Neighbor

Given a set of query points R and a set of reference points S, a k **nearest neighbor join** is an operation which, for each point in R, discovers the k nearest neighbors in S.

k nearest neighbor join (short for kNN) is frequently used as a classification or clustering method in machine learning or data mining areas. The primary application of a kNN join

is k-nearest neighbor classification. Some data points are given for training, and some new unlabeled data is given for testing. The aim is to find the class labels for new points. For each unlabeled data, a kNN query on the training set will be performed to estimate its class membership. This process can be considered as a kNN join of the testing set with the training set. The kNN operation can also be used to identify similar images. To do that, description features (points in a dataspace of dimension 128 [44]) are first extracted from images using a feature extractor technique. Then, the kNN operation is used to discover the points that are close, which should indicates similar images. kNN join, together with other methods, can be applied to a large number of fields, such as multimedia [111][108], social network [41], time series analysis [126] [31], bio-information and medical imagery [100][107].

The basic idea to compute a kNN join is to perform a pairwise computation of distance for each element in R and each element in S. The difficulties mainly lie in the following two aspects: (1) Data Volume (2) Data Dimensionality. Suppose we are in a d dimension space, the computational complexity of this pairwise calculation is $O(d \times |R| \times |S|)$. Finding the k nearest neighbors in S for every r in R boils down to finding the smallest k distances, and leads to a minimum complexity of $|S| \times log\,|S|$. As the amount of data or their complexity (number of dimensions) increases, this approach becomes impractical. This is why a lot of work has been dedicated to reducing the in-memory computational complexity [102][49][62][149][46]. These works mainly focus on two points: (1) Using indexes to decrease the number of distances need to be calculated. However, these indexes can hardly be scaled on high dimension data. (2) Using projections to reduce the dimensionality of data. But this results in a loss of accuracy. Despite these efforts, there are still significant limitations to process kNN on a single machine when the amount of data increases.

For large datasets (can not be processed in reasonable time on a single machine), only distributed and parallel solutions prove to be powerful enough. The MapReduce paradigm is the most used solution to parallel and distributed execution of kNN joins. Writing an efficient kNN in MapReduce is also challenging for many reasons. First, classical algorithms as well as the index and projection strategies have to be redesigned to fit the MapReduce programming model and its share-nothing execution platform. Second, data partition and distribution strategies have to be carefully designed to limit communications and data transfer. Third, load balancing is a new problem to address. Not only the number of distances to be sorted needs to be reduced, but also the number of MapReduce jobs and tasks. Finally, parameter tuning remains a key point to improve performance.

In this section we will present the preliminaries about processing a kNN join both on centralized environments and on parallel environments, both for static data and for dynamic data streams.

2.2.1.1 Definition

Given two data sets R and S in $\mathbb{R}^d$, each record $r \in R$ and $s \in S$ can be considered as a d-dimensional point in $\mathbb{R}^d$. Suppose the L^2 norm (Euclidean norm) is considered, the similarity between two points is measured by their Euclidean distance $d(r,s)$. Then, a nearest neighbors query of r in S returns the set of k nearest neighbors of r from S. It consists in finding k points in S with the smallest distance to the query point r.

More formally, given two data sets R and S in $\mathbb{R}^d$, and given r and s, two elements, with $r \in R$ and $s \in S$, we have:

Definition 2.4 *Let $d(r,s)$ be the distance between r and s. The **kNN query** of r over S, noted $kNN(r,S)$ is the subset $\{s_i\} \subseteq S$ ($|\{s_i\}| = k$), which are the k nearest neighbors of r in S, where $\forall\, s_i \in kNN(r,S)$, $\forall\, s_j \in S - kNN(r,S)$, $d(r,s_i) \leq d(r,s_j)$.*

This definition can be extended to a set of query points:

Definition 2.5 *The **kNN join** of two datasets R and S, $kNN(R \ltimes S)$ is:*
$kNN(R \ltimes S)$={(r,kNN(r,S)), $\forall\, r \in R$}

Depending on the use case, it might not be necessary to find the exact solution of a kNN query, and that is why approximate kNN queries have been introduced. The idea is to have the k^{th} approximate neighbor not far from the k^{th} exact one, as shown in the following definition.

Definition 2.6 *The $(1+\varepsilon)$-**approximate kNN query** for a query point r in a dataset S, $AkNN(r,S)$ is a set of approximate k nearest neighbors of r from S, if the k^{th} furthest result s^k satisfies $s^{k*} \leq s^k \leq (1+\varepsilon)s^{k*}$ ($\varepsilon > 0$) where s^{k*} is the exact k^{th} nearest neighbor of r in S.*

And as with the exact kNN, this definition can be extended to an approximate kNN join AkNN($R \ltimes S$).

The approximation of a kNN join mainly comes from the projection used for reducing the dimensionality of data in a centralized method. It might also come from the partitioning method used for a parallel and distributed method. And for a continuous kNN join, the approximation may also come from the summarization method.

2.2.2 Semantic Web

The earlier data formats defined by W3C [3] such as HTML, XML etc. are mainly human-readable contents. The object is to make the web pages pleasant to read and easy to navigate

[3] https://www.w3.org/TR/NOTE-rdfarch

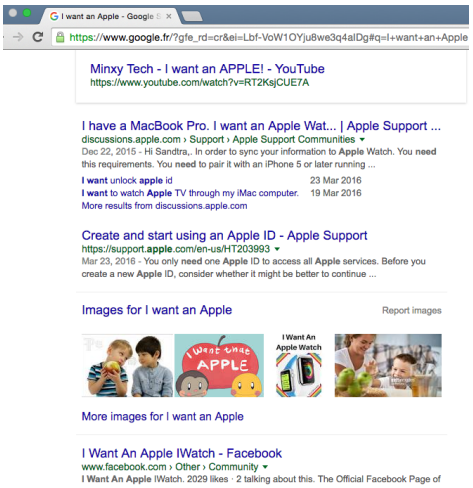


Fig. 2.11 I want an Apple!

by users. But on the other side, these data are not easy to process by machines to discover interesting knowledge because of their underlying representation. The direct effect of this problem is the difficulty to manipulate each data. For example, when one wants to search a specific subject on the internet, the search engine needs to first crawl the web pages and index the information by keywords. Although many advanced machine learning and nature language processing methods have been used, the results is still based on words, but not on the context. An example Google query "I want an apple" is shown in Fig. 2.11. The search engine gives answers about a song called "I want an apple", some results about the Apple watch, Apple ID etc.

The problem comes from the web of documents design shown in Fig. 2.12. Under this design, the machines can not understand the meaning of the content of data.

That's why Tim Bernes-Lee expressed the concept of Semantic Web in 1998 as follows[47]:

"The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

In brief, the Semantic Web aims at filling the gap between machines and humans. The purpose is to integrate data at Web Scale. It is a framework for integrating multiple sources to draw new conclusions and an architecture for describing all kinds of things. The Semantic Web is also called Web of Data. It provides machine-understandable information by linking everything together as shown in Fig. 2.13. This concept makes it possible to add a meaning (semantic) to every data on the web, and link the data items through their semantics. The search engines based on Semantic Web are more powerful to answer some natural questions

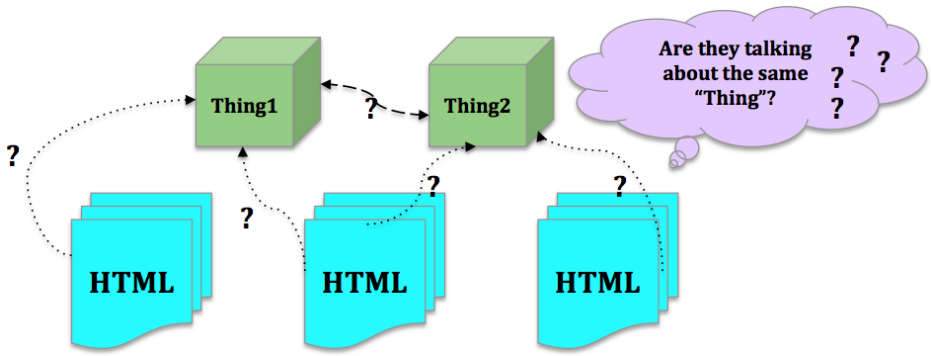


Fig. 2.12 Web of Document

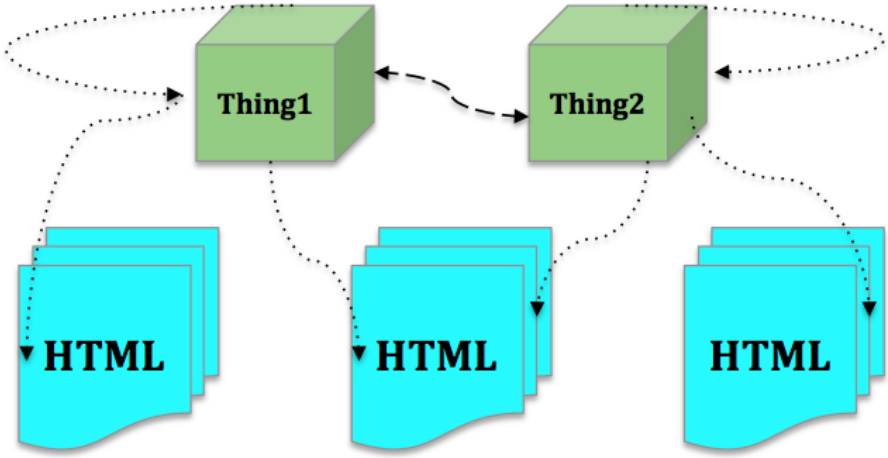


Fig. 2.13 Web of Data

in a concise and relevant manner. For example, when submitting a question like "Who is the president of France", these semantic search engines will not only return a list of results related to some of the keywords contained in the question, instead, they will search for the existing facts extracted and combined from different structured documents to give a direct answer along with relevant statements. Some search engines based on Semantic Web technologies are: Google Knowledge Graph[4], Ask Jeeves [5], Wolfram Alpha [6] and etc. An example from Wolfram Alpha shown in Fig. 2.14.

However, compared to other web technologies, the Semantic Web is still young. It requires many technologies to handle complex problems, including the processing of Big Data and Data Streams. In the coming sections, we will introduce the RDF data model which is a W3C data format designed for Semantic Web, and its corresponding query language

[4]https://www.google.com/intl/es419/insidesearch/features/search/knowledge.html
[5]http://fr.ask.com/
[6]https://www.wolframalpha.com/

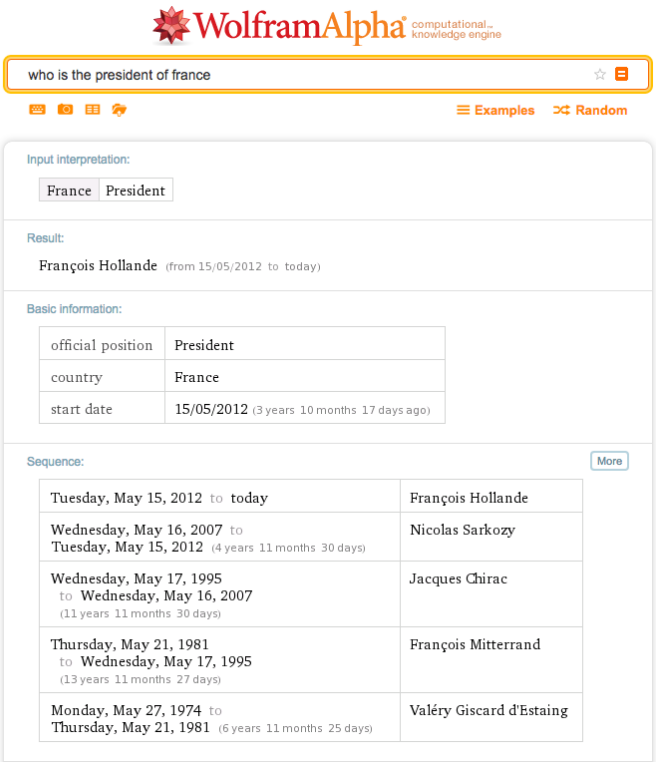


Fig. 2.14 Wolfram Alpha: a Semantic Search Engine

SPARQL. Then we will present the recent researches about parallel queries of RDF data and continuous queries of RDF streams.

2.2.2.1 RDF data model

The Resource Description Framework (RDF) [7] is a data standard proposed by W3C. It aims at representing semantic data in a machine understandable manner. It provides an abstract data model for representing structured knowledge into independent statements. This representation is used to describe semantic relations among data. In RDF format, data items are expressed as triples in form of <subject, predicate, object>. The subject of a triple indicates the resource that this triple is about; the predicates refers to the property of the subject; and the object denotes to the projection value of the subject by the predicate. An example of RDF triple in the XML format is shown in Fig. 2.15

There are three different kinds of values in an RDF triple:

- **IRIs:** IRI is short for Internationalized Resource Identifier. They are a complement of URIs. They preserve all the benefits from URIs, which are global unique identifiers

[7] https://www.w3.org/RDF/

```
<rdf:Description rdf:about="subject">
        <ex:predicate>
                <rdf:Description rdf:about="object"/>
        </ex:predicate>
</rdf:Description>
```

Fig. 2.15 An RDF Triple

```
Triple 1 : <Sophie, hasSister, Ray>
Triple 2 : <Sophie, hasDaughter, Amelie>
Triple 3 : <Ray, hasDaughter, Yume>
```

Fig. 2.16 RDF data naturally has semantics

of resources on the Web. The main advantage of using IRIs is that with the unique identifier feature, anyone can "link to it, refer to it, or retrieve a representation of it".

- **Literals:** they are a convenient alternative of IRIs for identifying some fix values such as strings, numbers or dates etc. Anything represented by a literal could also be represented by a URI, but it is often more convenient or intuitive to use literals. There are two different types of literals. The first one is plain literals which are unicode strings combined with an optional language tag. The other one is typed literals which are consist of unicode strings with a data type.

- **Blank nodes:** they are anonymous resources which name or identifier is not known or not specified. They can be described as existential variables.

Note that, the subject can be either an IRI or a Blank node, the predicate can only be an IRI and the object can be any of the 3 kinds of values.

One of the advantages that RDF format has is that it can link data together. The Predicate in a triple acts as a link between subject and object. RDF can also be considered as a directed graph, with predicate as edges in the graph, and subjects and objects as vertices in the graph. These links bring semantics to data. And this semantic can not only be understood by humans but also by machines. For example in Fig. 2.16, the 3 RDF triples shown provide more information beyond their literal values because of the semantic relation among them. From the semantic meaning of data, the machine can infer relationships of aunt, niece and cousins.

The corresponding graph representation of these RDF triples is shown is Fig. 2.17.

To help machines to understand the relations among data, a structure of data must be pre-defined. In Semantic Web, this pre-defined structure is called an Ontology. An ontology of data is a set of knowledge about a particular domain (for ex. Family, Profession, Bio-

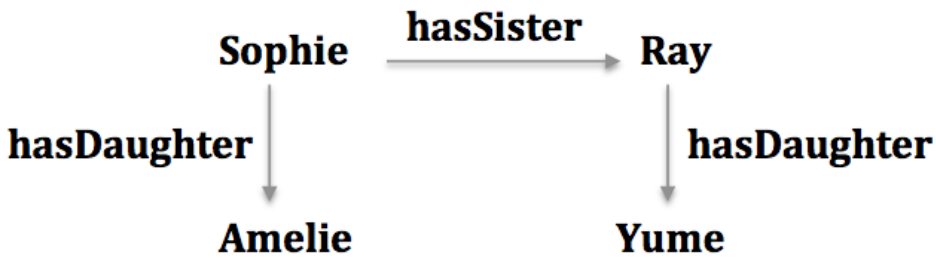


Fig. 2.17 Graph representation of RDF triples

```
SELECT ?S ?O1 ?O2
WHERE{
      ?S "workAt" ?O1 .
      ?S "hasDiplome" ?O2 .
      ?S "hasPaper" "kNN"
}
```

Fig. 2.18 A SPARQL Query: Triple Pattern Representation

Information etc.). The most common used standards to represent an Ontology are RDF Schema (RDFS) [6] and Web Ontology Language (OWL) [7].

2.2.2.2 SPARQL Query Language

SPARQL [20] is a W3C recommendation query language for querying RDF data. The basic component of a SPARQL query is the Basic Graph Pattern (BGP) [8]. A BGP is a conjunction of triple patterns. A triple pattern is a special kind of triple where S, P and O can be either a literal or a variable. The variable part in a triple pattern is used to retrieve unknown values; or to link a triple pattern with others; or both. Two triple patterns are connected if they share a common variable part. In this case, a conjunctive join is formed on this variable part.

A SPARQL query example is shown in Fig. 2.18. SPARQL syntax is similar to SQL. The SELECT clause states the variables to be retrieved. The WHERE clause contains all the triple patterns to be applied on RDF data. The conjunction is specified by a "." character. In the example in Fig. 2.18, the SELECT clause indicates 3 variables to be returned, and 3 triple patterns joined on ?S in the WHERE clause. SPARQL queries can also be represented by a directed graph as shown in Fig. 2.19 or in a relational representation as shown in Fig. 2.20

Depending on its shape, a SPARQL query can be star-shaped or chain-shaped. Depending on the processing mode, SPARQL join can be divided into hash join and nested loop join etc. There are many different expressions for SPARQL queries, the most used are: triple patterns, directed graphs and relational representation.

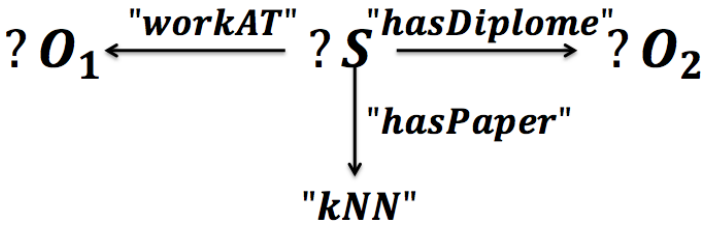


Fig. 2.19 A SPARQL Query: Graph Representation

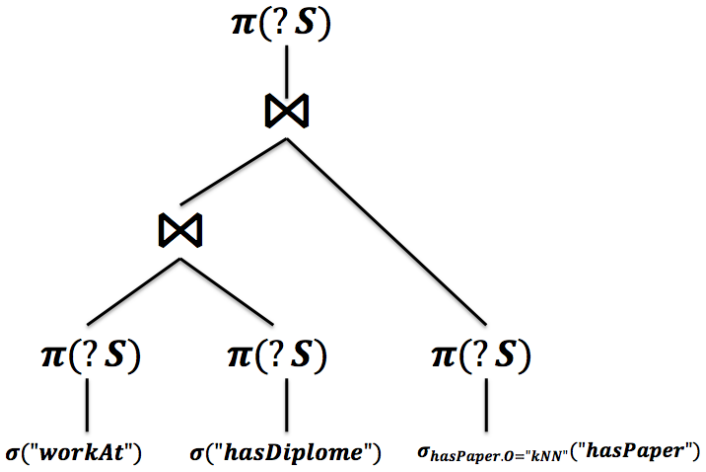


Fig. 2.20 A SPARQL Query: Relational Representation

Chapter 3

Data Driven Continuous Join (kNN)

3.1 Introduction

kNN computation is both a data and computation intensive job. The query applied over the data to compute the k nearest neighbors is always the same. The difficulty comes from the different types of data. The data for this problem may vary a lot, from the number of records (this use case yields both small and big data sets), the number of dimensionality (e.g. GPS data in 2 dimensions, twitter data in 77 dimensions, images feature data in 128 dimensions etc.), and the data formats.

We call this kind of join a **"data driven join"**. In this Chapter, we introduce the technologies for processing a continuous kNN join over data streams in a parallel and distributed manner. We first evaluate all the technologies used for processing a parallel and distributed kNN join. The methods evaluated are divided into two categories: (1) Exact Solutions: **H-BkNNJ** the basic kNN method, **H-BNLJ** [154] which is a parallel nested loop method for computing kNN join, and **PGBJ** [113] based on Voronoi Diagrams; (2) Approximate Solutions: **H-zkNNJ** [154] based on z-value and **RankReduce** [132] based on LSH. Then we summarize a general work flow composed by **data preprocessing**, **data partitioning** and **kNN join computation** for processing a parallel and distributed kNN join. We then give a theoretical analysis about every technologies through **load balancing**, **accuracy** and **complexity** aspects. After the summary for parallel and distributed kNN join processing method, we introduce new technologies for processing Continuous kNN join for data streams in a parallel and continuous manner. In the end we evaluate both parallel and distributed kNN processing on Hadoop MapReduce [1] and continuous kNN processing on

[1] We only evaluate the parallel and distributed processing of kNN join on Hadoop MapReduce, but the ideas presented in this Chapter can also be implemented on other parallel and distributed computing platform like Spark, because they share the same workflow.

Apache Storm. The evaluation and benchmarks of the parallel methods has been done in collaboration with two other Ph.D students, Justine Rochas and Lea El Beze.

3.2 Related Work

3.2.1 kNN Join for centralized environment

The basic solution to compute kNN adopts a nested loop approach, which calculates the distance between every object r_i in R and s_j in S and sorts the results to find the k smallest ones. This approach is computational intensive, making it unpractical for large or complex datasets. Two strategies have been proposed to work out this issue.

The first one consists in reducing the number of distances to compute, by avoiding scanning the whole dataset. This strategy focuses on indexing the data through efficient data structures. For example, a one-dimension index structure, the B^+-Tree, is used in [101] to index distances; [49] adopts a multipage overlapping index structure R-Tree; [62] proposes to use a balanced and dynamic M-Tree to organize the dataset; [150] introduces a sphere-tree with a sphere-shaped minimum bound to reduce the number of areas to be searched; [36] presents a multidimensional quad-tree in order to handle large amounts of data; [46] develops a kd-tree which is a clipping partition method to separate the search space; and [103] introduces a loose coupling and shared nothing distributed Inverted Grid Index structure for processing kNN queries on MapReduce.

Among them, the R-Tree index structure is the most widely used. R-Tree can be considered as an extension of B^+-Tree for multidimensional data. It is also a balanced search Tree. The general idea of R-Tree is to group close points and represent them in minimum bounding rectangles (MBRs). The letter "R" is short for "Rectangles". The points are recursively grouped into the MBRs. An example of R-Tree is shown in Fig. 3.1. R-Tree can not guarantee worst-case performance, but it can give a good performance in average cases for range queries and nearest neighbor queries. The idea of using an R-Tree structure for indexing data in a kNN problem is to adopt branch-and-bound search techniques. The R-Tree is usually traversed either in a depth-first manner or a breadth-first manner. The distances between the query point and the MBRs are calculated, and later used for pruning the search tree. However, R-Tree is proved to be unefficient for high-dimensional data in real applications.

Reducing the searched dataset might not be sufficient: because for data in high dimension space, even computing distances is very costly. That is why a second strategy focuses on projecting the high-dimension dataset onto a low-dimension one, while maintaining the

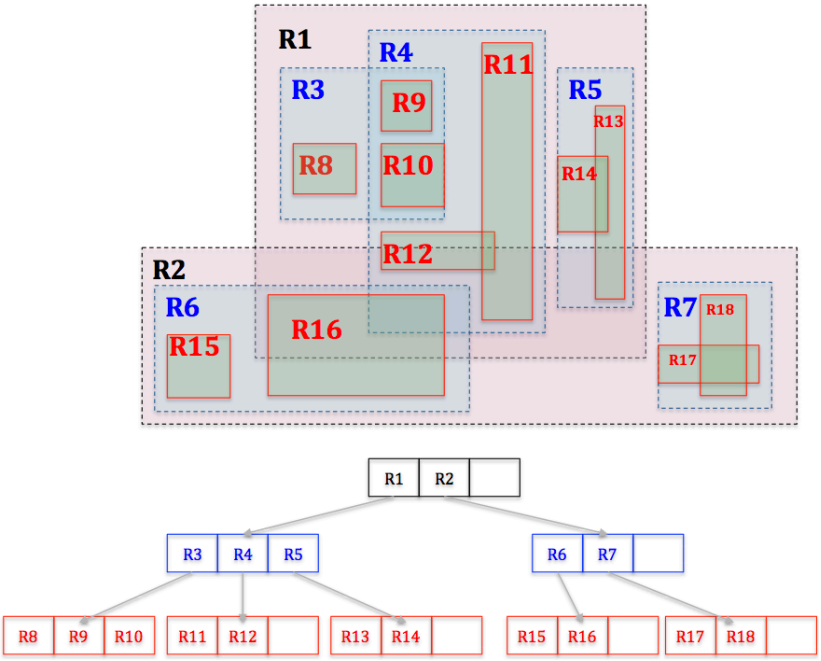


Fig. 3.1 An Example of R-Tree

locality relationship among data. Representative efforts are LSH (Locality-Sensitive Hashing) [86] and Space Filling Curve [147].

Locality Sensitive Hashing supposes that if two points are close to each other, then after a projection operation these two points should remain close together. Based on this idea, LSH uses hash collisions to reduce the dimensionality of high-dimensional data. Different from the conventional hash methods, LSH aims at maximizing the probability of a "collision" for similar items. An example of the process of LSH is shown in Fig. 3.2. Essentially, LSH uses a hash family[2] to randomly project data from a high dimension space to a lower dimensional one. An LSH function should make sure that nearby points will be projected to the same hash value with high probability. This hash value is represented by a bucket in Fig. 3.2. Each data is represented by a point with different colors. In this figure, the close points are projected in the same or nearby buckets by 4 hash functions.

Space Filling Curve maps data from high dimensional spaces to a 1 dimensional space while preserving locality of the data points. It is based on the idea that a continuous curve in a high dimension (more than 2) space can be considered as the path of a continuously moving point. There are many different kinds of Space Filling Curves, such as: Dragon Curve [1], Gosper Curve [10], Hilbert Curve [11], Z-Value [23] etc. Among them, Z-Value is proved to be the one which can better preserve the locality information. So it is the most frequently

[2] A hash family is a set of hash functions generated through a common rule

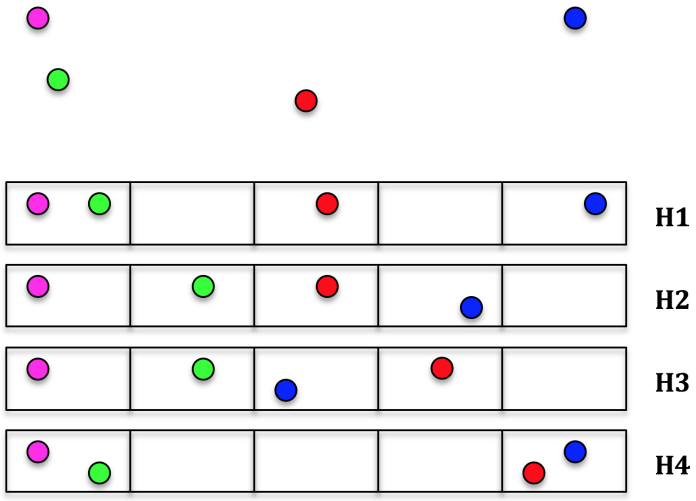


Fig. 3.2 An Example of LSH

used for solving kNN problems. The Z-Value of a point is calculated by interleaving the binary representations of its coordinate values. An example showing the calculation of Z-Value is shown in Fig. 3.3. In this figure, the points are in 2 dimensions, with X and Y its coordinates. X is written in blue and Y is written in red. The z-value is a value which binary representation is written by a red and blue figure, and which decimal representation is written in black. The "Z" in the figure indicates the neighborhood after projection.

3.2.2 Parallel kNN Join

With the increasing amount of data, these centralized methods still can not handle kNN computation on a single machine efficiently. Experiments in [39] suggest using GPUs to significantly improve the performance of distance computation, but this is still not applicable for large datasets (over TB) that cannot reasonably be processed on a single machine.

There are only very few existing works on parallel and distributed kNN join compared with the extensive research on traditional, centralized and single-threaded kNN join. In general, optimizing a parallel join is more complex than a centralized one. Early studies about parallel join algorithms in a shared-nothing and multi-core environment come from paper [106] and paper [128]. However, these studies focus on relational join operations such as equi-join or θ-join, which can not be applied directly to a parallel kNN join problem.

A problem similar to kNN join is kNN graph. It can be considered as a special case of a general kNN join. kNN graph is a self-join, where only one data set is used. The goal of a kNN graph is to generate kNNs for every data item in the data set. It is a graph in which every node is connected to its k nearest neighbors. Some studies about processing the kNN

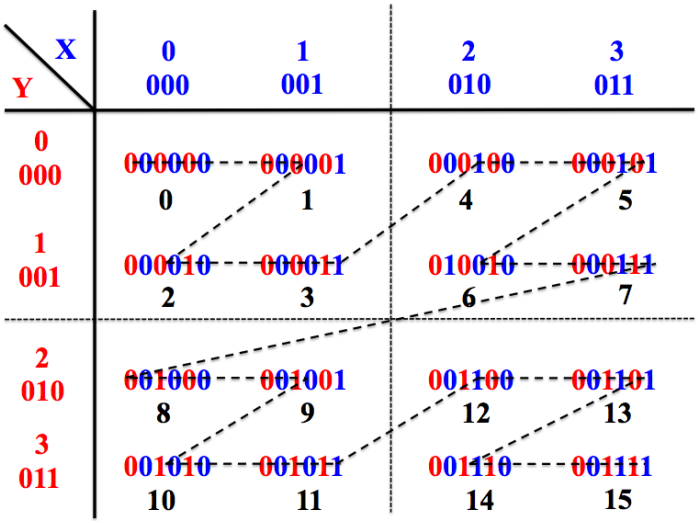


Fig. 3.3 An Example of Z-Value

graph problem with MPI is proposed in paper [123]. A kNN graph construction method on MPI and OpenMP is presented in paper [141]. An approximate kNN graph generation method on MapReduce is introduced in paper [73]. Although these papers propose very good ideas to solve a kNN related problem, they can not be easily generalized for solving the kNN join problem in parallel, since they only use one data set while a kNN join needs two. Two data sets require more complex partitioning strategy than only one data set.

More recent papers have focused on providing efficient distributed implementations of the kNN problem. Some of them use ad hoc protocols based on well-known distributed architectures [119, 91]. But most of them use the MapReduce model.

Writing an efficient kNN join on MapReduce is challenging for many reasons:

- First, classical algorithms as well as the index and projection strategies have to be redesigned to fit the MapReduce programming model and its share-nothing execution platform.
- Second, data partition and distribution strategies have to be carefully designed to limit communications and data transfer.
- Third, the load balancing problem which is new compared to the centralized version should also be considered.
- Fourth, not only the number of distances needed to be reduced, but a balance between the number of MapReduce jobs and map/reduce tasks has to be found.

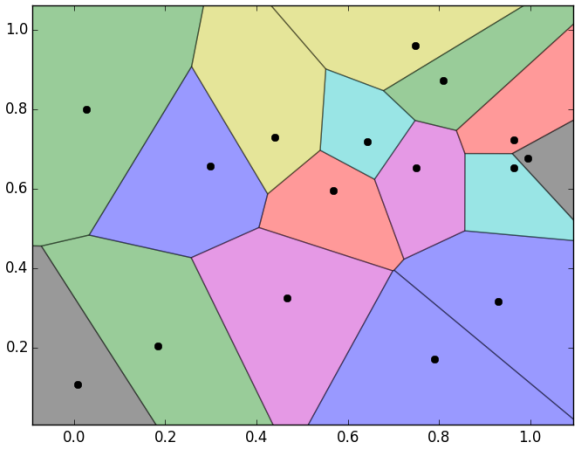


Fig. 3.4 An Example of Voronoi Diagram

- Finally, the parameter tuning of some methods remains a key point to improve performance.

The outstanding contribution for parallel and distributed processing of kNN join are the following works:

Paper [132] uses LSH to reduce the dimensionality of data, and implement it on MapReduce.

Paper [113] proposes to use Voronoi diagrams to partition data, then distribute each partition to different worker nodes of a Hadoop cluster. Voronoi diagrams are a partitioning of a plane. They divide the plane into regions based on distance to a specific subset of pivots. Each pivot is the leader of the region consisting of the points closer to that pivot than to any other. An example of Voronoi Diagrams generated by Python is shown in Fig. 3.4.

Paper [154] first proposes a Nested Loop execution of a kNN Join on MapReduce. Then it uses Z-Value to reduce the dimensionality of data, and tries to partition data into equal sized blocks in order to get a good load balance.

We will analyze these frameworks in Section 3.3.

3.2.3 Continuous kNN Join

Continuous kNN join for data streams has also been widely studied. Different from a static kNN join where the two join sets are fixed, in a Continuous kNN join, data is dynamic in at least one data set.

Paper [136] designs the join algorithms for data streaming systems, where memory is not large enough to hold all data to be processed.

Paper [50] proposes an efficient technique for processing continuous k nearest neighbor queries on data streams. They prove that this method can be used on high throughput data streams using only very limited storage. Their method is mainly based on 3 ideas:

- Select the exact objects in the stream which can possibly become the nearest neighbor of one or more continuous query points. Store them in a skyline data structure. Skyline is a data structure used to summarize multidimensional data sets. Given a data set P containing data points $p_1, p_2, ..., p_n$, the Skyline of P is the set of all p_i in P, where no p_j dominates p_i. For example, when assisting a user to find a set of restaurants from a larger set of candidate sets. Each restaurant is identified by two attributes: a distance from the user and the rank. To help the user to narrow down the choices, the Skyline structure can be used to find the set of all restaurants that are not dominated by another restaurant. Restaurant A dominates restaurant B if A is at least as close as B and has a higher rank than B.

- Index the query points.

- Delay the process for the points who are not immediately possible to be a nearest neighbor for any query point.

These methods are only suitable for low dimension data sets, but not efficient for high dimension data sets. To solve the problem of the "curse of dimensionality", paper [149] proposed the kNNJoin+ method, which supports efficient incremental computation of kNN join for dynamic high-dimensional data. This method is based on a Sphere-Tree index structure which is shown in Fig. 3.5. Sphere-Tree is used to deal with the update of data. Sphere-Tree is based on R-Tree and does not have a high pruning capability in high-dimensional space. Moreover, the computation of distance in high-dimensional space has a very high cost. They store every data on disk leading to a high disk I/O cost, which can not meet the real-time requirement in a stream processing scenario.

Paper [145] addresses the problem of real-time continuous kNN join processing in the context of content-based recommendation. As in a social network use case, data is always represented by features of hundreds of dimensions, so the main solution proposed in this paper is to maintain the high dimensional kNN join as data evolve. They proposed an in-memory index structure called HDR-Tree, which combines a clustering technique and PCA (Principal Component Analysis) [18].

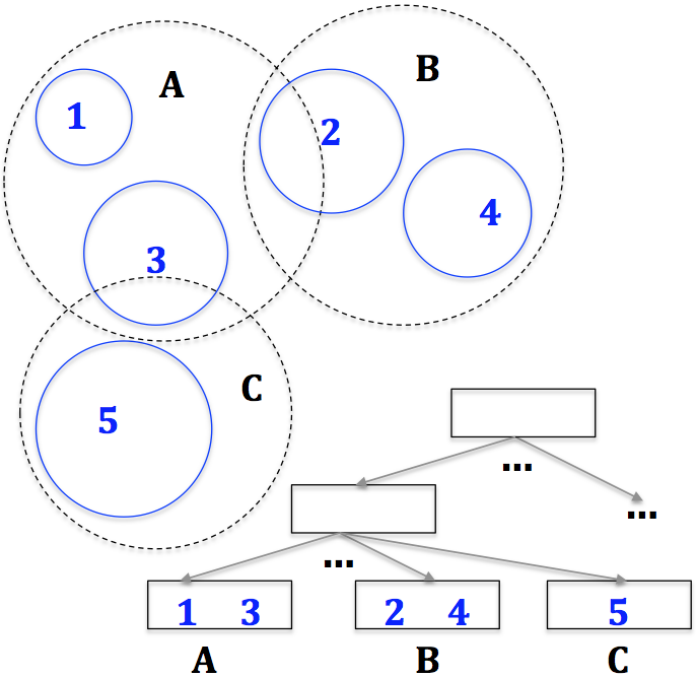


Fig. 3.5 Sphere-Tree

Unfortunately, these methods are all centralized. Since a "Tree" indexing structure is very hard to extend in a parallel and distributed environment, these methods may only be used for a local improvement in a parallel environment.

So far, to the best of our knowledge, no work can directly solve the kNN join in a parallel and distributed manner for continuous update data streams.

3.3 Parallel kNN

3.3.1 Workflow

In this section, we introduce the workflow for processing a kNN join in parallel. It consists in three ordered steps: (i) data pre-processing, (ii) data partitioning and (iii) kNN computation. These three steps are analyzed in the coming sections.

3.3.1.1 Data Preprocessing

The purpose of data preprocessing is to transform the original data to benefit from particular properties. This step is done before the partitioning of data to pursue two different goals:

(1) to reduce the dimension of data;

(2) to select central points of data clusters.

To reduce the dimension, data from a high-dimensional space will be mapped to a low-dimensional space by a linear or non-linear transformation. In this process, the challenge is to maintain the locality of the data in the low dimension space. In this thesis, we focus on two methods to reduce data dimensionality.

The first method is based on **space filling curves**. Paper [154] proves that z-value is the best space-filling curve to keep the locality information. The z-value of a data is a one dimensional value that is calculated by interleaving the binary representation of data coordinates from the most significant bit to the least significant bit, as presented in Section 3.2.2.

However, due to the loss of information during this process, this method can not fully keep the spatial location of data. In order to increase the accuracy, we can use several shifted copies of data and compute their z-values respectively. The shifted data set is generated by moving the original data set into the direction of a random vector. But this increases the computation cost, and occupies more disk space.

The second method to reduce data dimensionality is the **locality sensitive hashing (LSH)** [86][69] method. This method maps high-dimensional data into low-dimensional one, with L families of M locality preserving hash functions $\mathscr{H} = \{\ h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{W} \rfloor\ \}$, where a is a random chosen vector, b is a real number chosen uniformly from the range $[0, W]$, and W is the size of the buckets into which transformed values will fall. The principle of LSH is to make sure that $\forall h \in \mathscr{H}$:

$$if\ d(x,y) \leq d_1, Pr[h(x) = h(y)] \geq p_1 \tag{3.1}$$

$$if\ d(x,y) \geq d_2, Pr[h(x) = h(y)] \leq p_2 \tag{3.2}$$

where Pr is short for probability, d(x, y) is the distance between two points x and y, $d_1 < d_2$ and $p_1 > p_2$.

As a result, the closer two points x and y are, the higher the probability the hash values of these two points h(x) and h(y) in the hash family $\mathscr{H}$ (the set of hash functions used) are the same. The accuracy of LSH (how well it preserves locality) depends on the tuning of its parameters L, M, and W. The parameter L impacts the accuracy of the projection: increasing L increases the number of hash families that will be used, it thus increases the accuracy of the positional relationship by avoiding fallacies of a single projection. But in return, it also increases the processing time because of the duplication of data. The parameter M impacts the probability that the adjacent points fall into the same bucket. The parameter W reflects

the size of each bucket and thus, impacts the number of data in a bucket. All three parameters are important for the accuracy of the result. Basically, the key concept of LSH for computing *k*NN is to generate some collisions to find enough accurate neighbors.

The reference RankReduce paper [132] does not highlight enough the cost of setting the right value for all parameters, and show only one specific setup that allows the authors to have an accuracy greater than 70%.

Another aspect of the preprocessing step is to select central points of data clusters. Such points are called *pivots*. Paper [113] proposes 3 methods to select pivots. The *Random Selection* strategy generates a set of samples, then calculates the pairwise distance of the points in the sample, and the sample with the biggest sum of distances is chosen as the set of pivots. It provides good results if the sample is large enough to maximize the chance of selecting points from different clusters. The *Furthest Selection* strategy randomly chooses the first pivot, and calculates the furthest point to this chosen pivot as the second pivot, and so on until having the desired number of pivots. This strategy ensures that the distance between each selected point is as large as possible, but it is more complex to process than the random selection method. Finally, the *K-Means*[3] *Selection* applies the traditional k-means method on a data sample to update the centroid of a cluster as the new pivot at each step, until the set of pivots stabilizes. With this strategy, the pivots are ensured to be in the middle of a cluster, but it is the most computational intensive strategy as it needs to converge towards the optimal solution. The quality of the selected pivots is crucial, for effectiveness of the partitioning step, as we will see in the experiments.

The pre-processing of data is summarized in Table 3.1. These methods not only can be used in kNN join problems, but also can be used for any parallel and distributed data processing problem.

Table 3.1 The summary about the pre-processing step

Purpose	**Method**
Reduce the dimensionality	Space Filling Curve (z-value) Locality Sensitive Hashing (LSH)
Select the pivot points	Random Selection Furthest Selection K-Means Selection

[3]https://en.wikipedia.org/wiki/K-means_clustering

3.3.1.2 Data Partitioning

In order to process data on a shared-nothing parallel and distributed environment, we need to divide the dataset into independent pieces, which is called partitions. When computing a kNN join, we need to divide R and S respectively. As in any parallel and distributed process, the data partition strategy will strongly impact CPU, network communication and disk usages, which in turn will impact the overall processing time [130]. Besides, a good partition strategy could help reducing the number of data replications, thereby reducing the number of distances needed to be calculated and sorted.

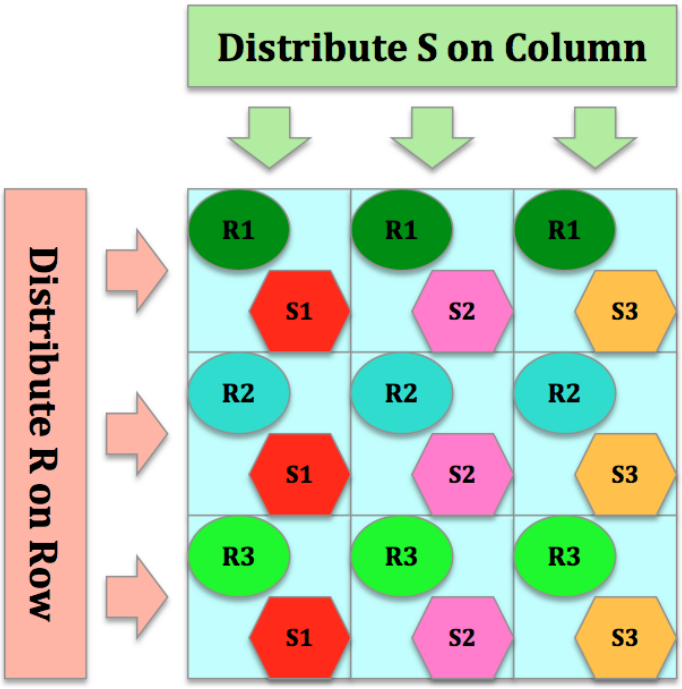


Fig. 3.6 Random Partition without any special strategies

We can choose different partition strategies to partition data. The simplest one is a Random partition strategy. H-BNLJ simply divides R into rows and S into lines, making each subset of R calculate with every subset of S. The process of this random partition is shown in Fig. 3.6 This ensures the distance between each object r_i in R and each object s_j in S will be calculated. But this way of dividing datasets generates a lot of data replications. For example, in H-BNLJ, each piece of data is duplicated n times where n is the number of subsets of R and S, resulting in a total of n^2 tasks for calculating pairwise distances. This method wastes a lot of hardware resources, and ultimately leads to low efficiency.

The key to improve performance is to preserve spatial locality of objects when decomposing data for tasks [155]. This means making a **coarse clustering** in order to produce a

reduced set of neighbors that only contains the candidates for the final result. Intuitively, the goal is to have a partition of data such that an element in a partition of R can find its k nearest neighbors in only one partition of S.

More precisely, what we want is: for every partition R_i ($\cup_i R_i = R$), find a corresponding partition S_j ($\cup_j S_j = S$), where:

$$kNN(R_i \ltimes S) = kNN(R_i \ltimes S_j)$$

And,

$$kNN(R \ltimes S) = \bigcup kNN(R_i \ltimes S_j)$$

which means that, not only it is possible to compute kNN for each element of R_i in a single S_j, but also the concatenation of the results for all R_i is equal to the global kNN join. Two partitioning strategies that enable to separate the datasets into independent partitions, while preserving locality information, have been proposed. They are the distance-based partitioning strategy and the size-based partitioning strategy. We introduce these two strategies respectively in the coming sections.

Distance-Based Partitioning Strategy

The distance-based partitioning strategy we study in this section is based on Voronoi diagrams, a method to divide the space into disjoint cells as presented in Section 3.2.2. We can find other distance-based partitioning methods in the litterature, such as in [103], but we chose Voronoi diagrams because it can be applied to data in any dimension. The main property of Voronoi diagrams is that every point in a cell is closer to the pivot of this cell than to any other pivot. More formally, the definition of a Voronoi cell is as follow:

Definition 3.1 *Given a set of disjoint pivots:*

$$P = \{p_1, p_2, ..., p_i, ..., p_n\}$$

then, the Voronoi Cell of p_i $(0 < i \leq n)$ *is:*

$$\forall\, i \neq j,\, VC(p_i) = \{p \| d(p, p_i) \leq d(p, p_j)\}.$$

Paper [113] gives a method to partition datasets R and S using Voronoi diagrams. The partitioning principles are illustrated in Fig. 3.7.

After having identified the pivots p_i in R (c.f. Section 3.3.1.1), the distances between elements of each dataset and the pivots are computed. The elements are then put in the cell of the closest pivot, giving a partitioning of R (resp. S) into P_i^R (resp. P_i^S). For each cell, the

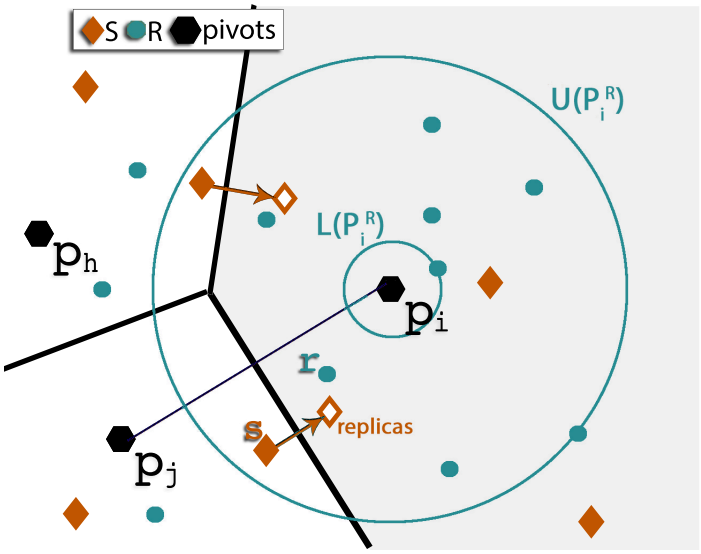


Fig. 3.7 Distance Based Partitioning Strategy : Voronoi Diagram

upper bound $U(P_i^R)$ (resp. the lower bound $L(P_i^R)$) is computed as a sphere determined by the furthest (resp. nearest) point in P_i^R from the pivot p_i. The boundaries and other statistics are used to find candidate data from S in the neighboring cells. This data is then replicated in cell P_i^S. For example, in Figure 3.7, the elements s of P_j^S fall inside $U(P_i^R)$ and are thus copied to S_i as a potential candidate for the kNN of r in P_i^R.

The main issue with this method is that it requires computing the distance from all elements to the pivots. Also, the distribution of the input data might not be known in advance. Hence, pivots have to be recomputed if data change. More importantly, there is no guarantee that all cells have an equal number of elements because of potential data skew. These disadvantages may lead to a load imbalancing problem, which will later give a negative impact on the overall performance. To alleviate this issue, the authors propose two grouping strategies, which will be discussed in Section 3.3.2.1.

Size Based Partitioning Strategy

Another type of partitioning strategy aims at dividing data into equal size partitions. Paper [154] proposes a partitioning strategy based on the z-value described in the previous section.

In order to have a similar number of elements in all n partitions, the authors first sample the dataset and compute the n quantiles. These quantiles are an unbiased estimation of the bounds for each partition. Figure 3.8 shows an example for this method. In this example data items are only shifted once. Then, data is projected using the z-value method. The “Z"

in the figure indicates the neighborhood after projection. Data sets are projected into a one dimension space, represented by Z_i^R and Z_i^S in the figure. Z_i^R is divided into partitions using the sampling estimation explained above. For a given partition R_i, its corresponding S_i is defined in Z_i^S by copying the nearest k preceding and the nearest k succeeding points to the boundaries of S_i. In Fig. 3.8, four points of S_i are copied in partition 2, because they are considered as candidates for the query points in R_i^2.

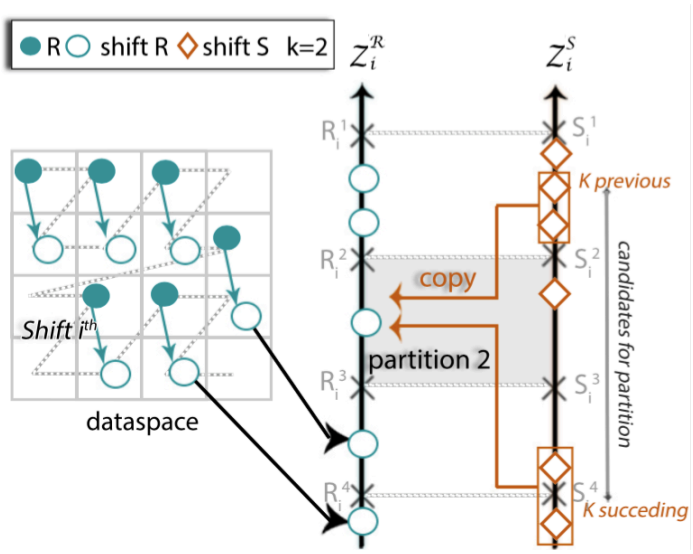


Fig. 3.8 Size-Based Partitioning Strategy : Z-Value

This method is likely to produce a substantially equivalent number of objects in each partition, in order to naturally achieve load balancing. However, the quality of the result depends solely on the quality of the z-curve, which might be an issue for high dimension data.

Another similar size-based partitioning method uses Locality Sensitive Hashing to first project data into low dimension space as illustrated in Fig. 3.9. In this example, data is hashed twice using two hash families $a1$ and $a2$. Each hashed data is then projected in the corresponding bucket. Ideally the data initially close in the high dimension space should be hashed to the same bucket with a high probability, if the bucket size (parameter W in LSH) is large enough to receive at least one copy of close data.

The strategy of partitioning directly impacts the number of tasks and the amount of computation. Distance based methods aim at dividing the space into cells that are driven by distance rules. Size based methods create equal size zones in which the points are ordered.

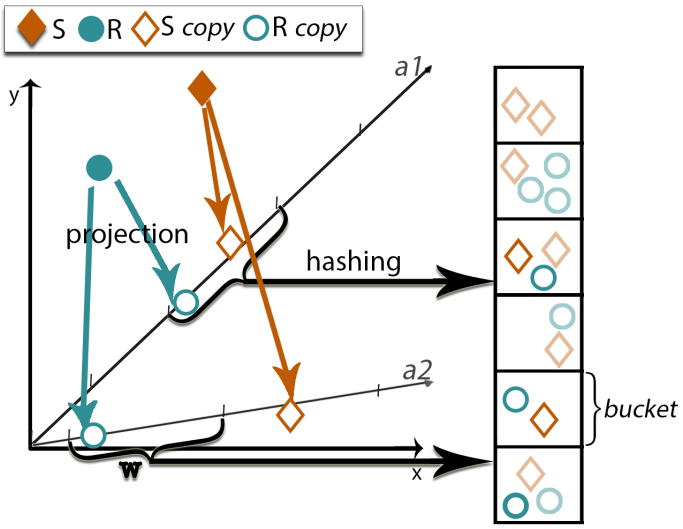


Fig. 3.9 Size Based Partitioning Strategy : LSH

Although it's more probable for the related objects to have the same hash value than the distance ones, normally one hash function can not guarantee the accuracy. We often need a group of hash functions to generate multiple hash tables to avoid the conflict probability of distance objects.

3.3.1.3 Computation

The main principle to compute a kNN join, is to (i) calculate the distance between r_i and s_j for all i, j, and (ii) sort these distances in ascending order to pick the first k results. The number of jobs for computing and sorting also impacts the global performance of the kNN computation, because of the complexity of MapReduce tasks and the amount of data transmitted. The preprocessing and partitioning steps impact on the number of MapReduce tasks that are further needed for the core computation. In this section, we review different strategies used to finally compute and sort distances efficiently. These different strategies can be divided into two categories, depending on the number of jobs they require. Those categories can themselves be divided into two subcategories: the ones that do not require special preprocessing and partition steps before computation and the ones that implement the preprocessing and partitioning steps.

One MapReduce Job

Without preprocessing and partitioning strategies.

The naive solution (**H-BkNNJ**) only uses one MapReduce job to calculate and sort the distances, and only the Map Phase is done in parallel. The Map tasks will cut datasets into splits, and label each split with its original dataset (R or S). The Reduce task then takes one object r_i and one object s_j to form a key-value pair $< r_i, s_j >$, calculates the distance between them, and for each key r_i sorts the distances with every objects in S, resulting in $|S|$ distances to be sorted. Since only the Map phase is parallel, and only one Reduce task is used for calculating and sorting, when the datasets becomes large, this method will quickly exceed the processing capacity of the computer. Therefore, it is only suitable for small datasets.

With preprocessing and partitioning strategies.

PGBJ [113] uses a preprocessing step to select the pivot of each partition and a distance based partitioning strategy to ensure that each subset R_i only needs one corresponding subset S_i to form a partition where the kNN of all $r_i \in R_i$ can be found. Therefore, in the computation step, Map tasks find the corresponding S_i for each R_i according to the information provided by the partitioning step. Reduce tasks then perform the kNN join inside each partition of $< R_i, S_i >$.

Overall, the main limitation of these two approaches is that the number of values to be sorted in the Reduce task can be extremely large, up to $|S|$, if the preprocessing and partitioning steps have not significantly reduced the set of searched points. This aspect can limit the applicability of such approaches in practice.

Two Consecutive MapReduce Jobs

To overcome the previously described limitation, multiple successive MapReduce jobs are required. The idea is to have the first job compute the local top k nearest neighbors for each pair (R_i, S_j). Then, the second job is used to merge all the top k values for a given r_i and to merge and sort all local top k values (instead of all values) producing the final global top k.

Without preprocessing and partitioning strategies.

H-BNLJ does not have any special preprocessing or partitioning strategy. The Map Phase of the first job distributes R into n rows and S into n columns. The n^2 Reduce tasks output the local kNN for each object r_i in the form of $(r_{id}, s_{id}, d(r,s))$.

Since each r_{id} has been replicated n times, the Map Phase of the second MapReduce job will pull every candidate of r_i from the n pieces of R, and form $(r_{id}), list(s_{id}, d(r,s))$. Then each Reduce task will sort $list(s_{id}, d(r,s))$ in ascending order of $d(r,s)$ for each r_i, and finally, give the top k results.

Moreover, in order to avoid the scan of the whole dataset of each block, some index structures like R-Tree [154] or Hilbert R-Tree [74] can be used to index the local S blocks.

With preprocessing and partitioning strategies.

In **H-zkNNJ** [154] the authors propose to define the bounds of the partitions of R and then to determine from this the corresponding S_i in a preprocessing job. So here, the preprocessing and partitioning steps are completely integrated in MapReduce. Then, a second MapReduce job takes the partitions R_i and S_i previously determined, and computes for all r_i the candidate neighbor set, which represents the points that could be in the final kNN[4]. To get this candidate neighbor set, the closest k points are taken from either side of the considered point (the partition is in dimension 1), which leads to exactly $2k$ candidate points. The third MapReduce round determines the exact result for each r_i from the candidate neighbor set. So in total, this solution uses three MapReduce jobs, and among them, the last two are actually devoted to the kNN core computation. As the number of points that are in the candidate neighbor set is small (thanks to the drastic partitioning resulting from preprocessing), the cost of computation and communication is extremely reduced.

In **RankReduce** [132][5], the authors first preprocess data to reduce the dimentionality and partition data into buckets using LSH. In our implementation, like in **H-zkNNJ**, one MapReduce job is used to calculate the local kNN for each r_i, and a second one is used to find the global ones.

3.3.1.4 Summary Work Flow

So far, we have studied different kNN computation workflows with three main steps. The first step focuses on data preprocessing, either for selecting dominating points or for projecting data from high dimension to low dimension. The second step aims at partitioning and organizing data such that the following kNN core computation step is lighten. The last step can use one or two MapReduce jobs depending on the number of distances we want to calculate and sort.

Figure 3.10 summarizes the workflow we have gone through in this section and the techniques associated with each step.

[4]Note that the notion of candidate points is different from local top k points.

[5]Although RankReduce only computes kNN for a single query, it is directly expandable to a full kNN join.

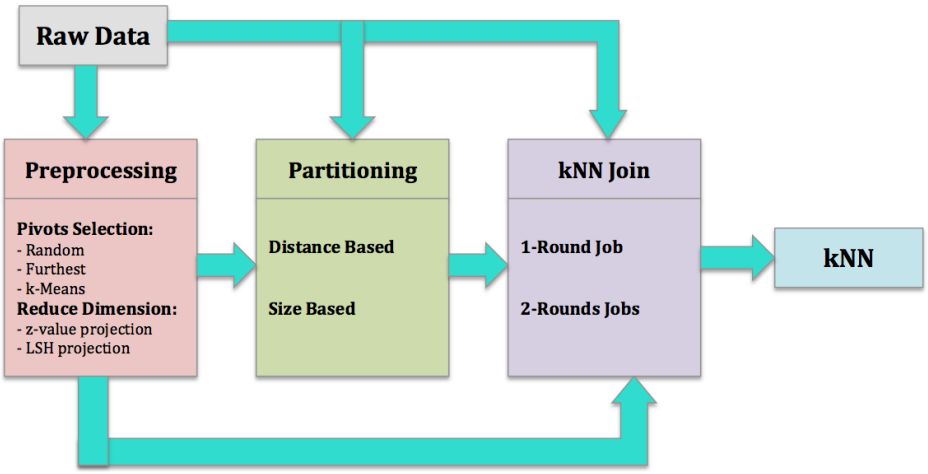


Fig. 3.10 General Workflow for processing a parallel and distributed kNN Join

3.3.2 Theoretical Analysis

3.3.2.1 Load Balance

In a parallel and distributed environment like MapReduce, the Map tasks or the Reduce tasks will be processed in parallel, so the overall computation time of each phase depends on the completion time of the longest task. Therefore, in order to obtain the best performance, it is important that each task performs substantially the same amount of computation. When considering load balancing in this section, we mainly want to have the same time complexity in each task. Ideally, we want to calculate roughly the same number of distances between points in each task.

For **H-BkNNJ**, there is no load balancing problem. Because in this basic method, only the Map Phase is treated in parallel. In Hadoop each task will process 64M data by default.

H-BNLJ cuts both the dataset R and the dataset S into p equal-size pieces, then those pieces are combined pairwise to form a partition of $< R_i, S_j >$. Each task will process one block of data so we need to ensure that the size of the data block handled by each task is roughly the same. However, **H-BNLJ** uses a random partitioning method which can not exactly divide the data into equal-size blocks.

PGBJ uses Voronoi diagrams to cut the data space of R into cells, where each cell is represented by its pivot. Then data items are assigned to the cell whose pivot is the nearest from them. For each R cell, we need to find the corresponding pieces of data in S. Sometimes, the data in S may be potentially needed by more than one R cell, which will lead to the duplication of some elements of S. Thus the number of distances to be calculated in each task, i.e. the relative time complexity of each task is:

$$\mathscr{O}(Task) = \left|P_i^R\right| \times \left(\left|P_i^S\right| + |RepS_c|\right)$$

where $\left|P_i^R\right|$ and $\left|P_i^S\right|$ represents the number of elements in cell P_i^R or P_i^S respectively, and $|RepS_c|$ the number of replicated elements for the cell. Therefore, to ensure load balancing, we need to ensure that $\mathscr{O}(Task)$ is roughly the same for each task. **PGBJ** introduces two methods to group the cells together to form a bigger cell. On one hand, the *geo grouping* method supposes that close cells have a higher probability to replicate the same data. On the other hand, the *greedy grouping* method estimates the cells whose data are more likely to be replicated. This approximation gives an upper bound to the complexity of the computation for a particular cell, which enables grouping of the cells that have the most replicated data in common. This leads to a minimization of replication and to groups that generate the same workload.

The **H-zkNNJ** method assumes:

$$\begin{gathered}\forall \text{ i} \neq \text{j}, \\ \text{if } |R_i| = \left|R_j\right| \text{ or } |S_i| = \left|S_j\right|, \\ \text{then } |R_i| \times |S_i| \approx \left|R_j\right| \times \left|S_j\right|\end{gathered}$$

That is to say, if the number of objects in each partition of R is equivalent, then the sum of the number of k nearest neighbors of all objects in each partition can be considered approximately equivalent, and vice versa. So an efficient partitioning should try to enforce either (i) $|R_i| = \left|R_j\right|$ or (ii) $|S_i| = \left|S_j\right|$. In paper [154], the authors give a short proof which shows that the worst-case computational complexity for (i) is equal to:

$$\mathscr{O}(|R_i| \times log\,|S_i|) = \mathscr{O}\left(\frac{|R|}{n} \times log\,|S|\right) \tag{3.3}$$

and for choice (ii), the worst-case complexity is equal to:

$$\mathscr{O}(|R_i| \times log\,|S_i|) = \mathscr{O}\left(|R| \times log\frac{|S|}{n}\right) \tag{3.4}$$

where n is the number of partitions. Since $n \ll |S|$, the optimal partitioning is achieved when $|R_i| = \left|R_j\right|$.

In **RankReduce**, a custom partitioner is used to load balance tasks between reducers. Let $W_h = \mid R_h \mid \times \mid S_h \mid$ be the weight of bucket h. A bin packing algorithm is used such that each reducer ends up with approximately the same amount of work. More precisely, let $\mathscr{O}(R_i) = \sum_h W_h$ the work done by reducer Ri, then this methods guarantees that

$$\forall i \neq j, \mathscr{O}(R_i) \approx \mathscr{O}\left(R_j\right) \tag{3.5}$$

Because the weight of a bucket is only an approximation of the computing time, this method can only give an approximate load balance. Having a large number of buckets compared to the number of reducers significantly improves the load balancing.

3.3.2.2 Accuracy

Usually, the lack of accuracy is the direct consequence of techniques to reduce the dimensionality with techniques such as z-values and LSH. In [154] (**H-zkNNJ**), the authors show that when the dimension of the data increases, the quality of the results tends to decrease. This can be counterbalanced by increasing the number of random shifts applied to the data, thereby increasing the size of the resulting dataset. Their experiments show that three shifts of the initial dataset (in dimension 15) are sufficient to achieve a good approximation (less than 10% of errors measured), while controlling the computation time. Furthermore, paper [148] processes a detailed theoretical analyses showing that, for any fixed dimension, by using only $\mathcal{O}(1)$ random shifts of data, the z-value method returns a constant factor approximation in terms of the radius of the k nearest neighbor ball.

For LSH, the accuracy is defined by the probability that the method will return the real nearest neighbors. Suppose that the points within a distance $d = |p-q|$ are considered as close points. The probability [129] that these two points end up in the same bucket is:

$$p(d) = Pr_{\mathscr{H}}[h(p) = h(q)] = \int_0^W \frac{1}{d} f_s(\frac{x}{d})(1 - \frac{x}{W})dx \tag{3.6}$$

where W is the size of the bucket and f_s is the probability density function of the hash function $\mathscr{H}$. From this equation we can see that for a given bucket size W, this probability decreases as the distance d increases. Another way to improve the accuracy of LSH is to increase the number of hashing families used. The use of LSH in **RankReduce** has an interesting consequence on the number of results. Depending on the parameters, the number of elements in a bucket might be smaller than k. Overall, unlike z-value, the performance of LSH depends a lot on parameter tuning.

3.3.2.3 Complexity

Carefully balancing the number of jobs, tasks, computation and communication is an important part of designing an efficient distributed algorithm. All the kNN algorithms studied in this Chapter have different characteristics. We will now describe them and outline how they can impact the execution time.

(1) **The number of MapReduce jobs:** Starting a job (whether in Hadoop [104] or any other platform) requires some initialization steps such as allocating resources and copying data. Those steps can be very time consuming.

(2) **The number of Map tasks and Reduce tasks used to calculate kNN$(R_i \ltimes S)$:** The larger this number is, the more information is exchanged through the network during the shuffle phase. Moreover, scheduling a task also incurs an overhead. But the smaller this number is, the more computation is done by each machine.

(3) **The number of final candidates for each object r_i:**

We have seen that advanced algorithms use pre-processing and partitioning techniques to reduce this number as much as possible. The goal is to reduce the amount of data transmitted and the computational cost.

Together these three points impact two main overheads that affect the performance:

- Communication overhead, which can be considered as the amount of data transmitted over the network during the shuffle phases.
- Computation overhead, which is mainly composed of two parts: 1). computing the distances, 2). finding the k smallest distances. It is also impacted by the dimension of the data.

Suppose the dataset is d dimensional, the overhead for computing the distance is roughly the same for every r_i and s_j for each method. The difference comes from the number of distances to sort for each element r_i to get the top k nearest neighbors. Suppose that the dataset R is divided into n splits. Here n represents the number of partitions of R and S for **H-BNLJ** and **H-zkNNJ**, the number of cells after using the grouping strategy for **PGBJ** and the number of buckets for **RankReduce**. Assuming there is a good load balance for each method, the number of elements in one split R_i can be considered as $\frac{|R|}{n}$. Finding the k closest neighbors efficiently for a given r_i can be done using a **Priority Queue** [24], which is less costly than sorting all candidates.

Since all these algorithms uses different strategies, their steps cannot be directly compared. Nonetheless, to provide a theoretical insight, we will now compare their complexity for the last phase, which is common to all of them.

The basic method **H-BkNNJ** only uses one MapReduce job, and requires only one Reduce task to compute and sort the distances. The communication overhead is $\mathcal{O}(|R| + |S|)$. The number of final candidates for one r_i is $|S|$. The complexity of finding the k smallest

distances for r_i is $\mathcal{O}(|S| \cdot log\,(k))$[6]. Hence, the total cost for one task is $\mathcal{O}(|R| \cdot |S| \cdot \log\,(k))$. Since R and S are usually large datasets, this method quickly becomes impracticable.

To overcome this limitation, **H-BNLJ** [154] uses two MapReduce jobs, with n^2 tasks to compute the distances. Using a second job significantly reduces the number of final candidates to nk. The total communication overhead is $\mathcal{O}(n\,|R| + n\,|S| + kn\,|R|)$. The complexity of finding the k elements for each r_i is reduced to $(n \cdot k) \cdot log\,(k)$. Since each task has $\frac{|R|}{n}$ elements, the total sort overhead for one task is $\mathcal{O}(|R| \cdot k \cdot log(k))$.

PGBJ [113] performs a preprocessing phase followed by two MapReduce jobs. This method also only uses n Map tasks to compute the distances and the number of final candidates falls to $|S_i|$. Since this method uses a distance based partitioning method, the size of $|S_i|$ varies, depending on the number of cells required to perform the computation and the number of replications ($|RepS_c|$, see Section 3.3.2.1) required by each cell. As such, the computational complexity cannot be expressed easily. Overall, finding the k elements is reduced to $\mathcal{O}(|S_i| \cdot log\,|k|)$ for each r_i, and $\mathcal{O}(\frac{|R|}{n} \cdot |S_i| \cdot log\,|S_i|)$ in total per task. The communication overhead is $\mathcal{O}(|R| + |S| + |RepS_c| \cdot n)$. In the original paper the authors gives a formula to compute $|RepS_c| \cdot n$, which is the total number of replications for the whole dataset S.

In **RankReduce** [132], the initial dataset is projected by L hash families into buckets. After finding the local candidates in the second job, the third job combines the local results to find the global k nearest neighbor. For each r_i, the number of final candidates is $L \cdot k$. Finding the k elements takes $\mathcal{O}(L \cdot k \cdot log(k))$ per r_i, and $\mathcal{O}(|R_i| \cdot L \cdot k \cdot log(k))$ per task. The total communication cost becomes $\mathcal{O}(|R| + |S| + k \cdot |R|)$.

H-zkNNJ[154] also begins by a preprocessing phase and uses in total three MapReduce jobs in exchange for requiring only n Map tasks. For a given r_i, these tasks process elements from the candidate neighbor set $C\,(r_i)$. By construction, $C\,(r_i)$ only contains $\alpha \cdot k$ neighbors, where α is the number of shifts of the original dataset. The complexity is now reduced to $\mathcal{O}((\alpha \cdot k) \cdot log\,(k))$ for one r_i, and $\mathcal{O}(\frac{|R|}{n} \cdot (\alpha \cdot k) \cdot log\,(k))$ in total per task. The communication overhead is $\mathcal{O}(\frac{1}{\varepsilon^2} + |S| + k \cdot |R|)$, with $\varepsilon \in (0,1)$, a parameter of the sampling process.

From the above analysis we can infer the following. **H-BkNNJ** only uses one task, but this task needs to calculate the entire data set. **H-BNLJ** uses n^2 tasks to greatly reduce the amount of data processed by each task. However this also increases the amount of data to be exchanged among the nodes. This should prove to be a major bottleneck. **PGBJ**, **RankReduce** and **H-zkNNJ** all use three jobs which reduce the number of tasks to n, and thus reduces the communication overhead.

[6] thanks to the priority queue, the complexity is smaller than sorting $|S|$

Although the computational complexity of each task depends on various parameters of the preprocessing phases, it is possible to outline a partial conclusion from this analysis. There are basically three performance brackets. First, the least efficient should be **H-BkNNJ**, followed by **H-BNLJ**. **PGBJ**, **RankReduce** and **H-zkNNJ** are theoretically the most efficient. Among them, **PGBJ** has the largest number of final candidates. For **RankReduce** and **H-zkNNJ** , the number of final candidates is of the same order of magnitude. The main difference lies in the communication complexity, more precisely in $\frac{1}{\varepsilon^2}$ compared to $|R|$. As the dataset size increases, we will eventually have $|R| \gg \frac{1}{\varepsilon^2}$. Hence, **H-zkNNJ** seems to be the theoretically more efficient for large query sets.

3.3.2.4 Wrap up

Although the workflow for computing kNN in a parallel and distributed manner is the same for all existing solutions, the guarantees offered by each of them vary a lot. As load balancing is a key point to reduce completion time, one should carefully choose the partitioning method to achieve this goal. Also, the accuracy of the computing system is crucial: are exact results really needed? If not, then one might trade accuracy for efficiency, by using data transformation techniques before the actual computation. Complexity of the global system should also be taken into account for particular needs, although it is often related to the accuracy: an exact system is usually more complex than an approximate one. Table 3.2 shows a summary of the systems we have examined and their main characteristics.

Methods	**Preprocessing**	**Partitioning**	**Accuracy**	**Complexity**			
				Jobs	**Tasks**	**Final Candidate (per r_i)**	**Communication**
H-BkNNJ (Basic Method)	None	None	Exact	1	1	$\|S\|$	$\mathcal{O}(\|R\|+\|S\|)$
H-BNLJ [154] (Zhang et al.)	None	None	Exact	2	n^2	nk	$\mathcal{O}(n\|R\| +n\|S\|+kn\|R\|)$
PGBJ [113] (Lu et al.)	Pivots Selection	Distance Based	Exact	3	n	$\|S_i\|$	$\mathcal{O}(\|R\| +\|S\|+\|RepS_c\|\cdot n)$
RankReduce [132] (Stupar et al.)	LSH	Size Based	Approximate	3	n	$L\cdot k$	$\mathcal{O}(\|R\|+\|S\| +k\cdot\|R\|)$
H-zkNNJ [154] (Zhang et al.)	Z-Value	Size Based	Approximate	3	n	$\alpha\cdot k$	$\mathcal{O}(\frac{1}{\varepsilon^2}+\|S\|+k\cdot\|R\|)$

Table 3.2 Summary table of kNN computing systems with MapReduce

Due to the multiple parameters and very different steps for each algorithm, we had to limit our complexity analysis to common operations. Moreover, for some of them, the complexity

depends on parameters set by a user or some properties of the dataset. Therefore, the total processing time might be different, in practice, than the one predicted by the theoretical analysis. That is why it is important to have a thorough experimental study.

3.4 Continuous kNN

The parallel and distributed technologies introduced above can only be applied to static data. For a static kNN join computation, the re-computation of the whole join is needed if there is any insertion, deletion or change of a data point. If there is any change or update in data, the expensive kNN join computation needs to be performed again. This drawback limits the efficiency in many real applications where updates are inevitable. This restriction also prevents these technologies to be applied directly for processing data streams. A parallel and distributed kNN join for data streams follows a workflow similar to the one presented in Section 3.3.1. But since data gets frequently updated, a more dynamic adaptation of the computation is essential. We need to partition and compute new data based on previous results, rather than re-compute everything. Furthermore the platform requires time for initialization when a new job is launched.

According to the aforementioned factors, the following two strategies need to be designed:

(1) Re-partition strategy

(2) Re-computation strategy

In order to describe the impact of the dynamic nature of data, we extend the definition given in Section 2.2.1.1 to account for the dynamicity of the dataset:

Definition 3.2 *Given two data sets R and S, the kNN join of R and S is :*

$$R \ltimes_{kNN} S = \{(r, kNN(r)) | r \in R \wedge kNN(r) \subseteq S\}$$

The changes (insertion or/and deletion) in R or/and in S have individual impacts on the join result, and also on the procedure of the join.

We can divide the continuous kNN joins into 3 types according to the dynamicity of R and S:

(1) **Static R and Dynamic S (SRDS):** This kind of join rarely exists in real applications. As the previous partition strategies are all based on R, and R will not change, we just need to use the previous method to partition the corresponding S every time we got

enough new values of s. We can use the method introduced in Section 3.2.3 (Sphere-Tree) to avoid the re-computation of the existing ones as a local improvement on each machine.

(2) **Dynamic R and Static S (DRSS):** This is the most used scenarios in real applications. As R can be considered as the Query Set, and S as the Searching Set, usually the Query Set is dynamic. For example, R is the dataset of user's locations, and S is the set of restaurants, and each time we want to search the top k nearest restaurants close to a given person. In this case, R is dynamic, and S can be considered as static since the opening or closing of restaurant is rare compared to our moving speed .

(3) **Dynamic R and Dynamic S (DRDS):** This is the general situation for a kNN stream join. In the previous example, if at the same time we consider also restaurants opening and closing, then S becomes dynamic.

Since (1) is trivial to be treated, we will only discuss (2) and (3) in the following sections.

3.4.1 Dynamic R and Static S kNN Join for Data Streams (DRSS)

3.4.1.1 Workflow for DRSS

When R is dynamic and S is static, the partitioning strategy will be different from what we introduced in Section 3.3.1.2. Because, with static data, for efficiency reasons, all the advanced partitioning strategies are based on R (please refer to the proof in Section 3.3.1.2.) The size-based or distance based partitioning strategies will first partition R, then find the appropriate S for each R. These partitioning strategies are not suitable when R is dynamic. In this case, we need to re-partition data when new data arrives. As shown in Section 3.3.1.2, the partition step requires a large amount of computation. Besides, the re-partition not only involves the new data, but also the previous data which has already been partitioned. This process will cause a transmission of data, which then results in huge network and disk overhead. Hence, in the DRSS scenario, the size-based and distance-based partitioning strategies described in the previous section. We can only use the random partition strategy.

The straightforward method for processing DRSS kNN join on a parallel and distributed streaming processing platform (here we use Apache Storm as an example) is to adopt our **Sliding Block Nested Loop Join (SBNLJ)** methodology.

Data update in SBNLJ:

For efficiency reasons, we choose to use lazy re-execution and lazy expiration strategies for our sliding window. We remove old data and re-compute new data periodically, and we call each period a **generation**. We update the data and results after each generation.

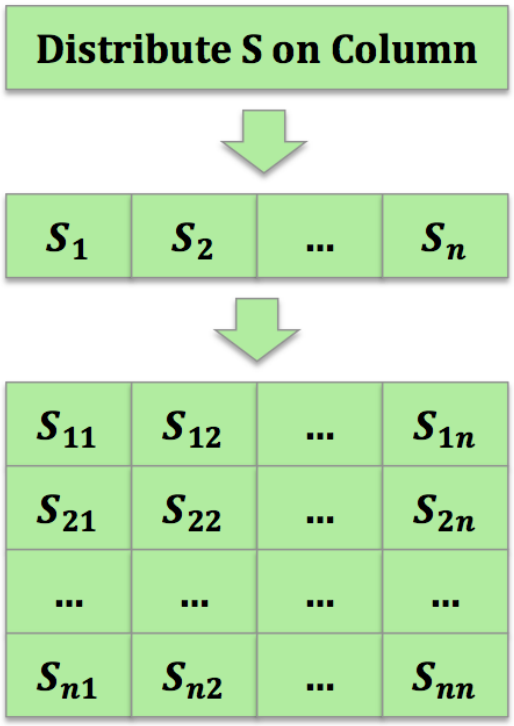


Fig. 3.11 The process for partition S in SBNLJ

Partition process in SBNLJ:

The basic idea is to first divide S into n equal-sized blocks. This step is easy to achieve by a linear scan, putting every $\frac{|S|}{n}$ records into one block. Since the Random Partition strategy will generate n^2 buckets to make sure every r_i can meet every s_j to be further computed, we need to vertically replicate each S blocks n times. This process is shown in Fig. 3.11. In this figure, one block of S is represented by a lattice and the blocks in the same column hold the same piece of data (with $S_{1i} = S_{2i} = S_{3i} = ... = S_{ni}$).

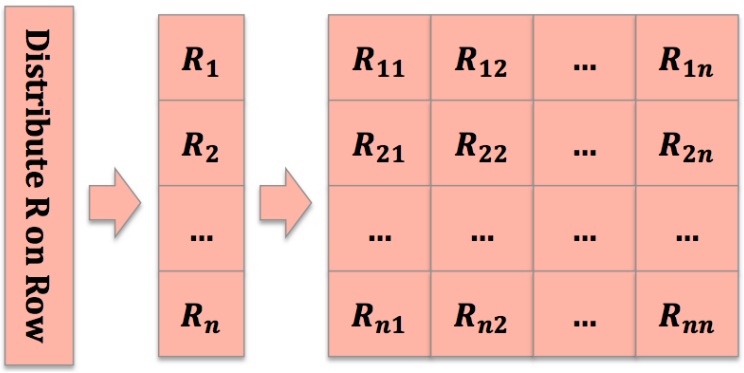


Fig. 3.12 The process for partition each generation of R in SBNLJ

Since *S* is static, the partition will never change. We can use one *Spout* and n^2 *Bolts* to achieve the partition for S. The *Spout* needs to divide S into n blocks, replicate each block n times, and give each partition of S an ID as the subscript in Fig. 3.11. Then n^2 *Bolts* will be launched. Each will receive one partition of S and save it in the local disk of this Bolt.

Each time we receive a new generation of R, we will do the same partition process as for S. Firstly, the current generation of R will be cut into n equal-sized blocks. To make each *Bolt* receive a different pair of R and S, we horizontally replicate the blocks n times. The whole process is shown in Fig. 3.12. In this figure, the blocks of R are represented by a lattice and the lattices in the same row are identical. Then an output field is declared in *Spout* with the subscript of each partition (one lattice in Fig. 3.12) as the IDs. Each partition will be an input of the *Bolt* in the next step which holds the partition of S with the same ID.

Computation process in SBNLJ:

To improve the efficiency, we choose to use two rounds of Computation, the first round is called kNNLocalBolt. It has n^2 tasks. Each processes a nested loop for the local R and S on this node, then emits the local top k nearest neighbors of each r as results in form of:

$$< \tau_k, < r_i, < s_j, d(r_i, s_j) >>>$$

Where τ_k indicates the generation number. The emitted Field ID is set as the indicator of rows in Fig. 3.11 or Fig. 3.12 (the first part of the partition ID), because the lattices in the same row hold the same piece of R; and in the next step, we would like to gather all the results for each r_i on one single machine, in order to merge the local results to have the global results. The second round has n nodes. Each receives one field as input. This input contains one partition of R along with all the local top k results with every partitions of S. In the second round of computation, we just sort the n local top k results to get the global results.

Re-Computation in SBNLJ:

As *S* is static, we do not need to re-compute the r we already computed. We only need to compute the new r. After emitting the results of each generation, the results from the oldest generation will expire, and the rest of the results will be stored in a temporal list.

3.4.2 Dynamic R and Dynamic S kNN Join for Data Streams

Dynamic R and dynamic S kNN join is the general case. To deal with this type of processes, we need first to partition the data, then calculate and update. In order to improve the efficiency and to reduce the network and disk overhead, we want to partition and transmit partitioned data only once.

3.4.2.1 Basic Method

The basic method is a random partition, we use the random partition method for each generation of data. Unlike DRSS, S is also dynamic. The idea is to divide R and S into n blocks, and replicate each block n times, then combine each different R and S into pairs, using the subscript of R and S blocks as the ID of each partition. In each generation, the partition with the same ID will be sent to the same node to be processed, to ensure that each r_i and s_j in the same validity period can meet each other. The process is shown in Fig. 3.13.

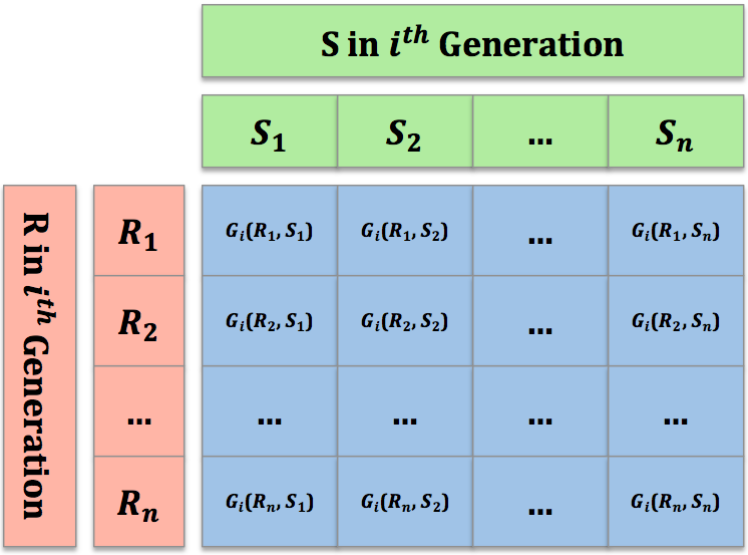


Fig. 3.13 Sliding Radom Partition for DRDS

The partition step is performed on Spout. The current generation of R and S is divided into n blocks separately, and each block will be replicated n times, then pairwise combined. Each partition's ID consists of two parts, the first part is the ID of an R block and the other part is the ID of an S block. n^2 fields will be declared, with each partition ID as the field name.

The computation part is similar to the process of DRSS. The first step is also to find local results. But in this scenario, S is also dynamic, so every time after a new generation is received, every record in the current valid sliding window should be considered when emitting new results. We call the already existing generations in the current valid Sliding Window R_{old} and S_{old}, and the new coming one R_{new} and S_{new} respectively. In order to avoid multiple calculation, we only calculate $\{ R_{new} \ltimes_{kNN} (S_{old} \cup S_{new}) \}$ and $\{ R_{old} \ltimes_{kNN} S_{new} \}$ in each generation. The output of the first stage is in form of:

$$< \tau_r, < r_i, < \tau_s, < s_j, d(r_i, s_j) >>>>$$

Where τ_r indicates the generation of r and τ_s represents the generation of s. The second step for computation needs also to use n nodes to gather together all the local results for each block of R, and merge them in order to get the global top k nearest neighbors. It will first remove the previous r according to τ_r. Then for each valid r, remove the invalid results in accordance with τ_s. Finally, it will sort all the potential distances for each r to get the final results. The results will be temporarily stored in the Bolt for further comparison in the next generation.

3.4.2.2 Advanced Method

The basic method requires n^2 nodes, which leads to a lot of repeated calculation and network and disk overhead. That is why an advanced partition strategy which produces only n partitions (as for the static data) is highly required. But at the same time, we would like to apply the partition strategy only to the new generations, and avoid moving the data which has already been partitioned.

Suppose we already have n partitions for the first Sliding Window using any size-based or distance-based partitioning method presented in Section 3.3.1.2. The partitions are represented by:

$$N = \{N_i | i \in [1,n]\} \tag{3.7}$$

In order to partition the new generation of data without destroying existing partitions, we use the Naive Bayes method to find the corresponding partition for each data record in the new coming generations. Every new r_i can still find its nearest neighbors inside the partition, thereby limiting the number of partitions to n.

Theorem 3.1 ***Bayes' Theorem:*** *Given two independent events A and B, the conditional probability of given B and A occurs is:*

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{3.8}$$

The re-partitioning strategy is based on Naive Bayes classification. We consider the n partitions from N_1 to N_n as n different classes and the already partitioned data as training set. The probability that a new point x belongs to N_i is:

$$P(N_i|x) = \frac{P(x|N_i) \cdot P(N_i)}{P(x)} \tag{3.9}$$

And x should be assigned to the partition N_y which has the biggest probability:

$$x \in N_y, \ where\ P(N_y|x) = max\{P(N_1|x), P(N_2|x), ..., P(N_n|x)\} \tag{3.10}$$

So the partitioning problem for new coming data x can be transferred to the calculation of the probability $P(N_i|x)$.

Suppose we use p different Locality Sensitive Hashing functions $\{l_1, l_2, ..., l_p\}$ to calculate the LSH value $\{l_1(x), l_2(x), ..., l_p(x)\}$ of each data x [7], then these p LSH values can be considered as p features of x. And:

$$P(N_i|x) = P(N_i|(l_1(x), l_2(x), ..., l_p(x))) \tag{3.11}$$

According to Bayes' Theorem:

$$= \frac{P((l_1(x), l_2(x), ..., l_p(x))|N_i)}{P(l_1(x), l_2(x), ..., l_p(x))} \tag{3.12}$$

Since l_1, l_2, ..., l_p are independent, we have:

$$= \frac{P(l_1(x)|N_i) \cdot P(l_2(x)|N_i) ... \cdot P(l_p(x)|N_i) \cdot P(N_i)}{P(l_1(x)) \cdot P(l_2(x)) ... \cdot P(l_p(x))} \tag{3.13}$$

In order to gain load balance, we suppose that the number of records in each partition are the same. That is to say, each partition has the same probability of being chosen with:

$$P(N_1) = P(N_2) = ... = P(N_n) = \frac{1}{n} \tag{3.14}$$

And $P(l_1(x)) \cdot P(l_2(x)) ... \cdot P(l_p(x))$ is independent of partitions, so it can be considered as a constant for every partition. Then, the comparison of $P(N_i|x)$ values is identical to the comparison of $P(l_j(x)|N_i)$ values.

$P(l_j(x)|N_i)$ is the probability of the appearance of $l_j(x)$ on N_i, and this probability is decided by the distribution of data on N_i. The naive method to compare this probability is to compare the difference between $l_j(x)$ and the average value of l_j on N_i. The larger this difference, the smaller the probability that this data belongs to N_i.

More precisely, it can be decided by the probability density function [8] (PDF). A Probability Density Function(PDF) is a function which describes the relative likelihood for a random variable to have a given value. The probability of the random variable falling within

[7] Here we can also use p different shifts to get p different z-values for each data as its features.

[8] https://en.wikipedia.org/wiki/Probability_density_function

a particular range of values is defined by the integral of this variable's density over that range. $PDF(x)$ can be considered as the probability of having x in the current distribution of data.

Suppose we have Gaussian Distribution (Normal Distribution)[17] with its Probability Density Function as follows:

$$PDF(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.15}$$

Where μ is the mean of the distribution, σ^2 is the variance and σ is the standard deviation. We can continuously calculate the expectation and the standard deviation of each feature on each partition, to estimate the Probability Density Function parameters, and use them to compute $P(l_j(x)|N_i)$.

The partition phase is done in a Bolt, this partition Bolt receives data from Spouts and also the computation Bolts. The computation Bolts will update the expectation and variance for their current data periodically, and send this information to the partition Bolt. The partition Bolt then partitions each generation according to the method presented before.

Since the results for each r can be found in a single node, we only need to use n nodes for computation. Each node processes a nested loop for all the r_i and s_j pairs in the current validate window on this nodes. For avoiding multiple computations, we also temporally store the result in form of:

$$< \tau_r, < r_i, < \tau_s, < s_j, d(r_i, s_j) >>>>$$

and compute only $\{ R_{new} \ltimes_{kNN} (S_{old} \cup S_{new}) \}$ and $\{ R_{old} \ltimes_{kNN} S_{new} \}$ in each generation. The update process is the same as in the basic method.

3.5 Experiment Result

In this section, we present an extensive experimental evaluation for the methods described in the previous sections. For a parallel and distributed kNN stream join, the most important influence on performance comes from the parallel processing method presented in Section 3.3 including data pre-processing strategies, data partitioning strategies and the computation stage. So we will first evaluate the parallel methods used for processing a static kNN join on Hadoop MapReduce [9]. This evaluation will guide us through choosing the methods for processing kNN join on streams.

[9]These parallel methods can also be processed on Apache Storm or other parallel and distributed processing platform using the same methodology

The experiments were run on two clusters of Grid'5000[10], one with Opteron 2218 processors and 8GB of memory, the other with Xeon E5520 processors and 32GB of memory, using Hadoop 1.3, 1Gb/s Ethernet and SATA hard drives. We follow the default configuration of Hadoop: (1) the number of replications for each split of data is set to 3; (2) the number of slots of each node is 1, so only one map or reduce task is processed on the node at one time.

We mainly evaluate 5 approaches.

For **H-zkNNJ** and **H-BNLJ**, we took the source code provided by the authors as a start[11]. We also added some modifications to combine intermediate data in order to reduce the size of intermediate files. The other approaches were implemented from scratch according to the description provided in their respective papers.

When implementing **RankReduce**, we added a reduce phase in the first MapReduce job to collect some statistical information of each bucket. This information is used for achieving good load balance. Moreover, to improve the precision, we choose to use multiple families and hash functions depending on the dataset. Finally, our version of **RankReduce** uses three MapReduce jobs instead of two.

Most of the experiments were ran using two different datasets:

- **OpenStreetMap:** we call it the *Geographic - or Geo - dataset*. The Geo dataset contains geographic XML data in two dimensions[12]. This is a dataset containing real location and description of objects. The data is organized by region. We extract $256 * 10^5$ records from the region of France.

- **Catech 101:** we call it the *Speeded Up Robust Features - or SURF - dataset*. It is a public set of images[13], which contains 101 categories of pictures of different objects, and 40 to 800 images per category. SURF [43] is a detector and descriptor for points of interest in images, which produces image data in 128 dimensions. We extract 32 images per category, each image has between 1000 and 2000 descriptors.

In order to learn the impact of dimension and dataset, we use 5 additional datasets: **El Nino:** in 9 dimensions; **HIGGS:** in 28 dimensions; **TWITTER:** in 77 dimensions; **BlogFeedBack:** in 281 dimensions; and **Axial Axis:** in 386 dimensions. These data sets are all downloaded from the UCI Machine Learning Repository[14].

[10] www.grid5000.fr
[11] http://ww2.cs.fsu.edu/~czhang/knnjedbt/
[12] Taken from: http://www.geofabrik.de/data/download.html
[13] Taken from: www.vision.caltech.edu/Image_Datasets/Caltech101
[14] Taken from: https://archive.ics.uci.edu/ml/

We use two equal-sized data-sets for R and S with $|R| = |S|$, in all our experiences. The number of records of each dataset is varied from $0.125 * 10^5$ to $256 * 10^5$. For all experiments, we have set $k = 20$ except when evaluating its impact.

We evaluate the methods through the following metrics:

- The impact of the size of data
- The impact of k
- The impact of the dimension of data and the nature of dataset

We record the following information: the processing time, the disk space required, the recall and precision, and the communication overhead.

To assess the quality of the approximation algorithms, we compute two commonly used metrics and use the results of the exact algorithm **PGBJ** as a reference. First, we define the recall as $recall = \frac{|A(v) \bigcap I(v)|}{|I(v)|}$, where $I(v)$ are the exact kNN of v and $A(v)$ the kNN found by the approximate methods. Intuitively, the recall measures the ability of an algorithm to find the correct kNNs. Another metric, the precision is defined by $precision = \frac{|A(v) \bigcap I(v)|}{|A(v)|}$. It measures the fraction of correct kNN in the final result set. By definition, the following properties holds: (1) $recall \leq precision$ because all the tested algorithms return up to k elements. (2) if an approximate algorithms outputs k elements, then $recall = precision$.

Each algorithm produces intermediate data so we compute a metric called *Space requirement* based on the size of intermediate data ($Size_{intermediate}$), the size of the result ($Size_{final}$) and the size of the correct kNN ($Size_{correct}$). We thus have $space = \frac{Size_{final} + Size_{intermediate}}{Size_{correct}}$.

We start by evaluating the most efficient number of machines to use (hereafter called *nodes*) in terms of resources and computing time. For that, we measure the computing time of all algorithms for three different data input sizes of the geographic dataset. The result can be seen on Figure 3.14. As expected, the computing time is strongly related to the number of nodes. Adding more nodes increases parallelism, reducing the overall computing time. There is however a significant slow down after using more than 15 machines. Based on those results, and considering the fact that we later use larger datasets, we conducted all subsequent experiments using at most 20 nodes.

3.5.1 Geographic dataset

For all experiments in this section, we used the parameters described in Table 3.3. Details regarding each parameter can be found in sections 3.3.1.1 and 3.3.1.2. For RankReduce, the value of W was adapted to get the best performance from each dataset. For datasets up to

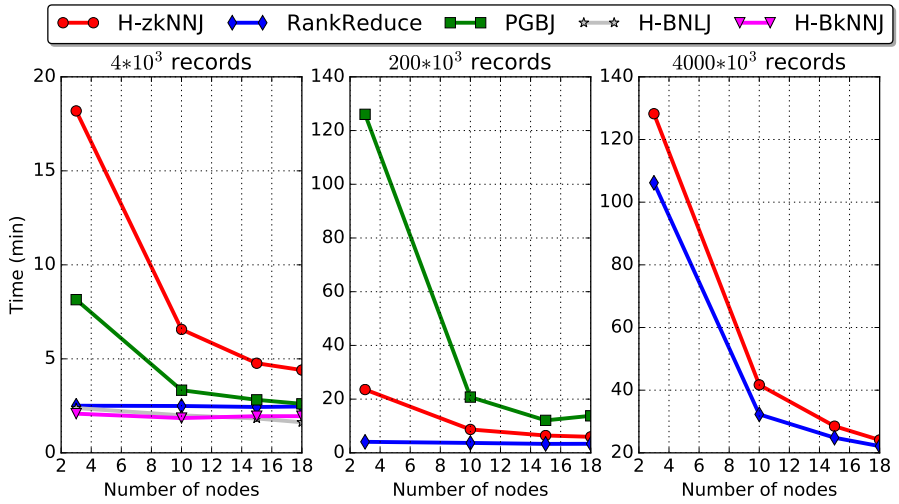


Fig. 3.14 Impact of the number of nodes on computing time

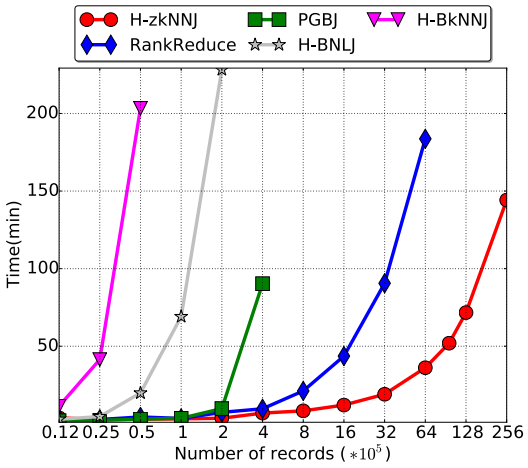
$16*10^5$ records, $W = 32*10^5$, up to $25*10^5$ records, $W = 25*10^5$ and finally, $W = 15*10^5$ for the rest of the experiments.

Algorithm	Partitioning	Reducers	Configuration
H-BNLJ	10 partitions	100 reducers	
PGBJ	3000 pivots	25 reducers	k-means + greedy
RankReduce	$W = \begin{cases} 32*10^5 \\ 25*10^5 \\ 15*10^5 \end{cases}$	25 reducers	L = 2 M = 7
H-zkNNJ	10 partitions	30 reducers	3 shifts, p=10

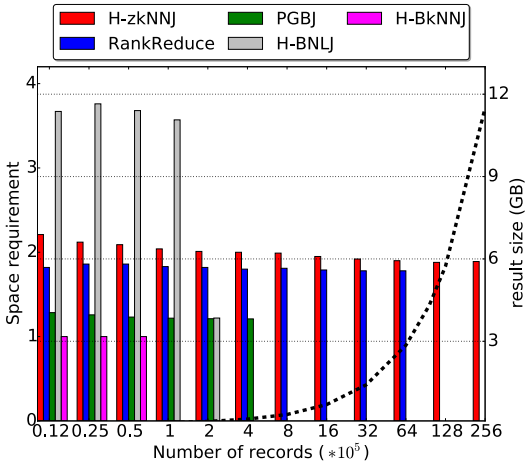
Table 3.3 Algorithm parameters for geographic dataset

3.5.1.1 Impact of input data size

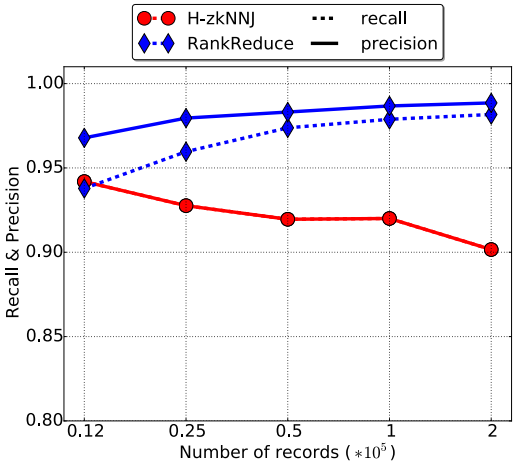
Our first set of experiments measures the impact of the data size on execution time, disk space and recall. Figure 3.15a shows the global computing time of all algorithms, varying the number of records from $0.125*10^5$ to $256*10^5$. The global computing time increases more or less exponentially for all algorithms, but only **H-zkNNJ** and **RankReduce** can process medium to large datasets. For small datasets, **PGBJ** can compute an exact solution as fast as the other algorithms.



(a) Time



(b) Result size and Disk Usage



(c) Recall and Precision

Fig. 3.15 Geo dataset impact of the data set size

Figure 3.15b shows the space requirement of each algorithm as a function of the final output size. To reduce the footprint of each run, intermediate data is compressed. For example, for **H-BNLJ**, the size of intermediate data is 2.6 times bigger than the size of output data. Overall, the algorithms with the lowest space requirements are **RankReduce** and **PGBJ**.

Figure 3.15c shows the recall and precision of the two approximate algorithms, **H-zkNNJ** and **RankReduce**. Since **H-zkNNJ** always returns k elements, its precision and recall are identical. As the number of records increases, its recall decreases, while still being high, because of the space filling curves used in the preprocessing phase. On the other hand, the recall of **RankReduce** is always lower than its precision because it outputs less than k elements. It benefits from larger datasets because more data end up in the same bucket,

increasing the number of candidates. Overall, the quality of **RankReduce** was found to be better than **H-zkNNJ** on the Geo dataset.

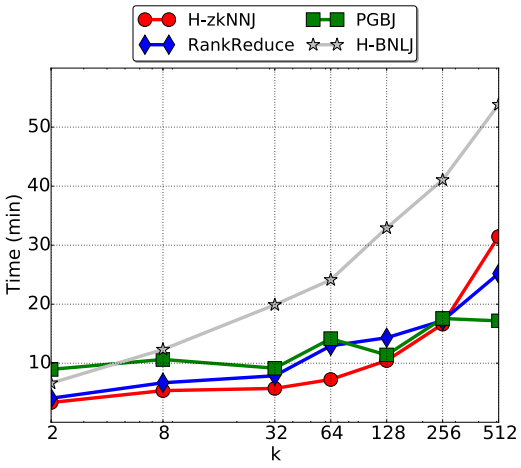
3.5.1.2 Impact of k

Changing the value of k can have a significant impact on the performance of some of the kNN algorithms. We experimented on a dataset of $2 * 10^5$ records (only $5 * 10^4$ for **H-BNLJ** for performance reasons) with values for k varying from 2 to 512. Results are shown in Figure 3.16 using a logarithmic scale on the x-axis.

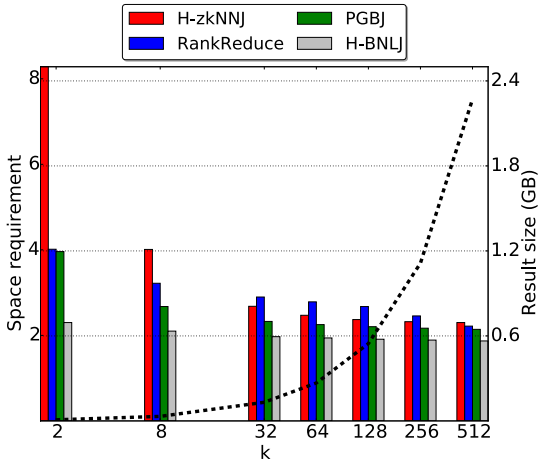
First, we observe a global increase in computing time (Figure 3.16a) which matches the complexity analysis performed earlier. As k increases, the performance of **H-zkNNJ**, compared to the other advanced algorithms, decreases. This is due to the necessary replication of the z-values of S throughout the partitions to find enough candidates: the core computation is thus much more complex.

Second, the algorithms can also be distinguished considering their disk usage, visible on Figure 3.16b. The global tendency is that the ratio of intermediate data size over the final data size decreases. This means that for each algorithm the final data size grows faster than the intermediate data size. As a consequence, there is no particular algorithm that suffers from such a bottleneck at this point. **PGBJ** is the most efficient from this aspect. Its replication of data occurs independently of the number of selected neighbors. Thus, increasing k has a small impact on this algorithm, both in computing time and space requirements. On this figure, an interesting observation can also be made for **H-zkNNJ**. For $k = 2$, it has by far the largest disk usage but becomes similar to the others for larger values. This is because **H-zkNNJ** creates a lot of intermediate data (copies of the initial dataset, vectors for the space filling curve, sampling...) irrespective of the value of k. As k increases, so does the output size, mitigating the impact of these intermediate data.

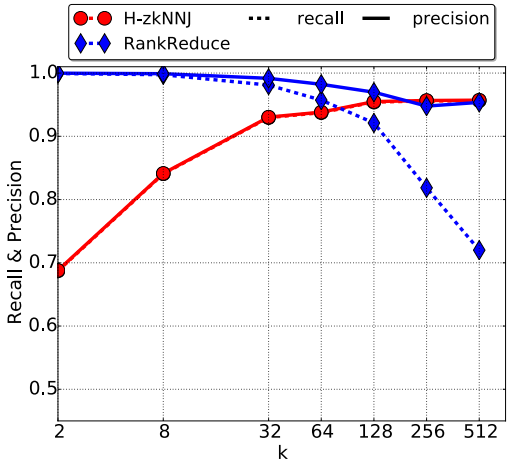
Surprisingly, changing k has a different impact on the recall of the approximate kNN methods, as can be seen on Figure 3.16c. For **RankReduce**, increasing k has a negative impact on the recall which sharply decreases when $k \geq 64$. This is because the window parameter (W) of LSH was set at the beginning of the experiments to achieve the best performance for this particular dataset. However, it was not modified for various of k. Thus it became less optimal as k increased. This shows there is a link between global parameters such as k and parameters of the LSH process. When using **H-zkNNJ**, increasing k improves the precision: the probability to have incorrect points is reduced as there are more candidates in a single partition.



(a) Time



(b) Result size and Disk Usage



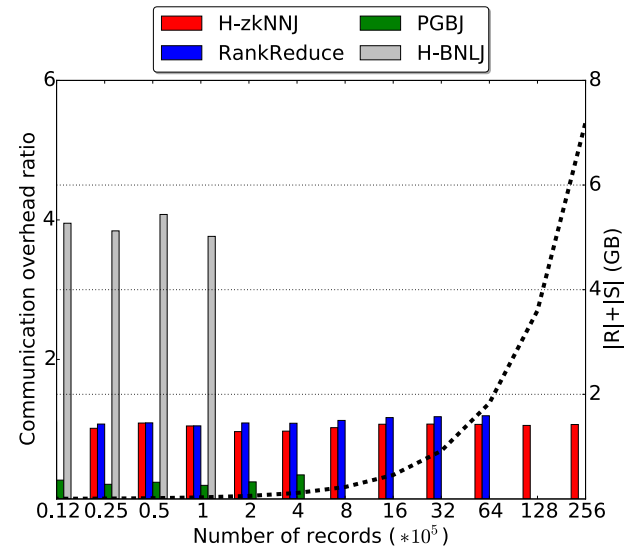
(c) Recall and Precision

Fig. 3.16 Geo dataset with 200k records (50k for H-BNLJ), impact of k

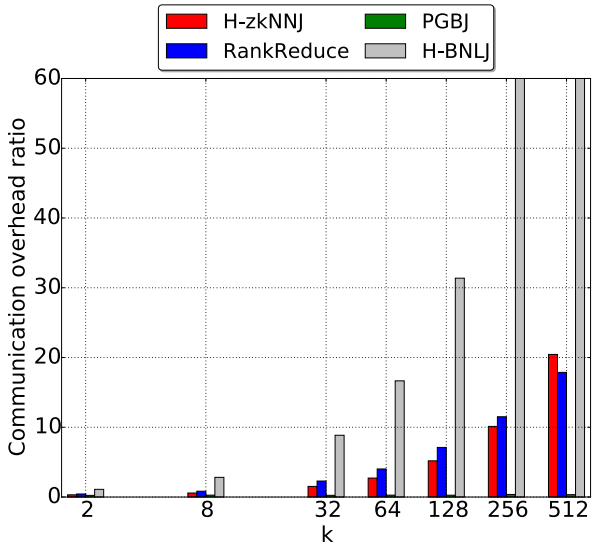
3.5.1.3 Communication Overhead

Our last set of experiments looks at inter-node communication by measuring the amount of data transmitted during the shuffle phase (Figure 3.17). The goal is to compare these measurements with the theoretical analysis in Section 3.3.2.3,

Impact of data size. For Geo dataset (Figure 3.17a), **H-BNLJ** has indeed a lot of communication. For a dataset of $1 * 10^5$ records, the shuffle phase transmits almost 4 times the original size. Both **RankReduce** and **H-zkNNJ** have a constant factor of 1 because of the duplication of the original dataset to improve the recall. The most efficient algorithm is



(a) Impact of the data set size



(b) Impact of k with $2 * 10^5$ records ($0.5 * 10^5$ for H-BNLJ)

Fig. 3.17 Geo dataset, communication overhead

PGBJ for two reasons. First it does not duplicate the original dataset and second, it relies on various grouping strategies to minimize replication.

Impact of k. We have performed another set of experiments, with a fixed dataset of $2 * 10^5$ records (only $0.5 * 10^5$ for **H-BNLJ**). The results can be seen in Figure 3.17b. For different values of k, we have a similar pattern as with the data size. For **RankReduce** and **H-zkNNJ**, the shuffle increases linearly because the number of candidates in the second phase depends

on k. Moreover **H-zkNNJ** also replicates k previous and succeeding elements in the first phase, and because of that, its overhead becomes significant for large k. Finally in **PGBJ**, k has no impact on the shuffle phase.

3.5.2 Image Feature Descriptors (SURF) dataset

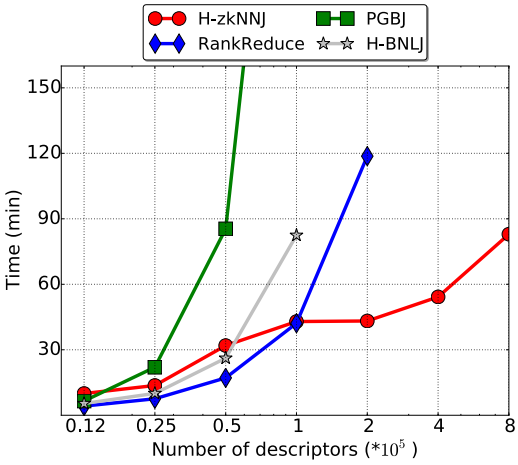
We now investigate whether the dimension of input data has an impact on the kNN algorithms using the SURF dataset. We used the Euclidian distance between descriptors to measure image similarity. For all experiments in this section, the parameters mentioned in Table 3.4 are used.

Algorithm	Partitioning	Reducers	Configuration
H-BNLJ	10 partitions	100 reducers	
PGBJ	3000 pivots	25 reducers	k-means + geo
RankReduce	W = 10^7	25 reducers	L = 5 M = 7
H-zkNNJ	6 partitions	30 reducers	5 shifts

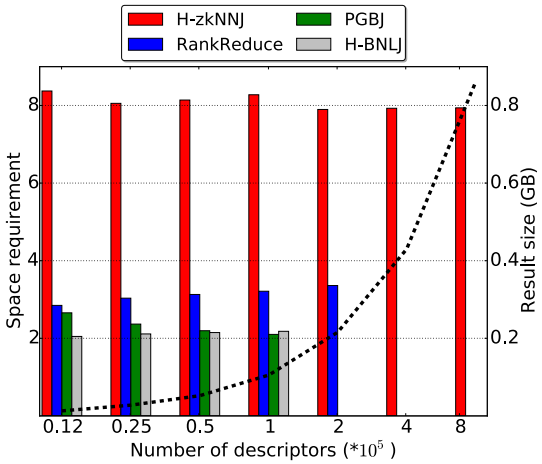
Table 3.4 Algorithm parameters for SURF dataset

3.5.2.1 Impact of input data size

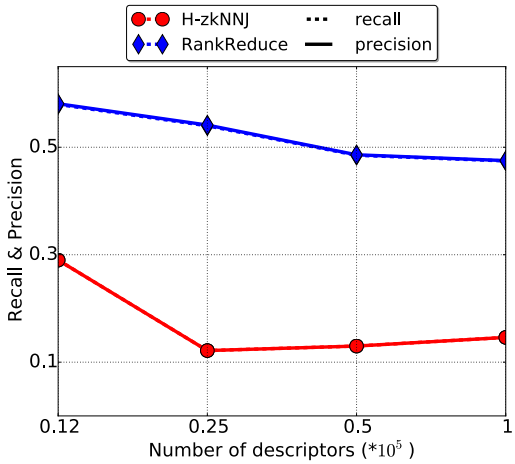
Results of experiments when varying the number of descriptors are shown in Figure 3.18 using a log scale on the x-axis. We omitted **H-BkNNJ** as it could not process the data in reasonable time. In Figure 3.18a, we can see that the execution time of the algorithms follows globally the same trend as with the Geo dataset, except for **PGBJ**. It is a computationally intensive algorithm because the replication process implies calculating a lot of Euclidian distances. When in dimension 128, this part tends to dominate the overall computation time. Regarding disk usage (Figure 3.18b), **H-zkNNJ** is very high because we had to increase the number of shifted copies from 3 to 5 to improve the recall. Indeed, compared to the Geo dataset, recall is very low (Figure 3.18c). Moreover, as the number of descriptors increases, **H-zkNNJ** goes from 30% to 15% recall. As explained before, the precision was found to be equal to the recall, which means the algorithm always returned k results. This, together with the improvement using more shifts, proves that the space filling curves using in **H-zkNNJ** are less efficient with high dimension data.



(a) Time



(b) Result size and Disk Usage

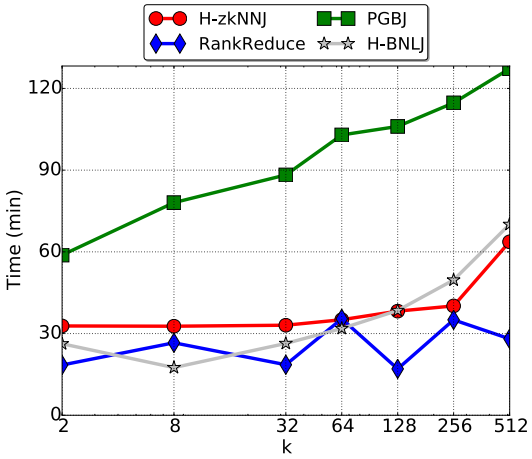


(c) Recall and Precision

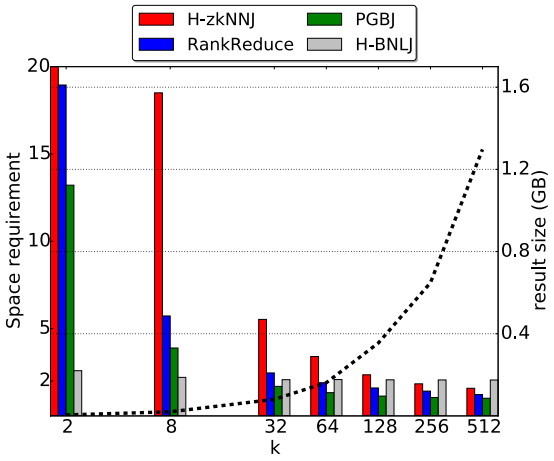
Fig. 3.18 Surf, impact of the dataset size

3.5.2.2 Impact of k

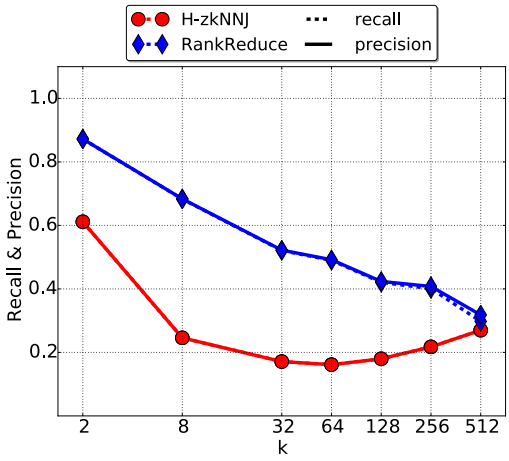
Figure 3.19 shows the impact of different values of k on the algorithms using a logarithmic scale on the x-axis. Again, since for **H-BNLJ** and **H-zkNNJ**, the complexity of the sorting phase depends on k, we can observe a corresponding increase of the execution time (Figure 3.19a). For **RankReduce**, the time varies a lot depending on k. This is because of the stochastic nature of the projection used in LSH. It can lead to buckets containing different numbers of elements, impacting the load balance and some values of k naturally lead to a better load balancing. **PGBJ** is very dependent on the value of k because of the grouping phase. Neighboring cells are added until there are enough elements to eventually identify



(a) Time



(b) Result size and Disk Usage



(c) Recall and Precision

Fig. 3.19 Surf dataset with 50k records, impact of k,

the k nearest neighbors. As a consequence, a large k will lead to larger group of cells and increase the computing time.

Figure 3.19b shows the effect of k on disk usage. **H-zkNNJ** starts with a very high ratio of 74 (not showed on the Figure) and quickly reduces to more acceptable values. **RankReduce** also experiences a similar pattern to a lesser extend. As opposed to the Geo dataset, SURF descriptors cannot be efficiently compressed, leading to large intermediate files.

Finally, Figure 3.19c shows the effect of k on the recall. As k increases, the recall and precision of **RankReduce** decreases for the same reason as with the Geo dataset. Also, for large k, the recall becomes lower than the precision because we get less than k results.

The precision of **H-zkNNJ** decreases but eventually shows an upward trend. The increased number of requested neighbors increases the number of preceding and succeeding points copied, slightly improving the recall.

3.5.2.3 Communication Overhead

With the SURF dataset, we get a very different behavior than with the Geo dataset. The shuffle phase of **PGBJ** is very costly (Figure 3.20a). This is an indication of large replications incurred by the large dimension of the data and a poor choice of pivots. When they are too close to each other, entire cells have to be replicated during the grouping phase.

For **RankReduce** the shuffle is decreased but stay important, essentially because of the replication factor of 5. Finally, the shifts of original data in **H-zkNNJ** lead to a large communication overhead.

Considering now k, we have the same behavior we observed with the Geo dataset. The only difference is **PGBJ** which now exhibits a large communication overhead (Figure 3.20b). This is again because of the choice of pivots and the grouping of the cells. However, this overhead remains constant, irrespectively of k.

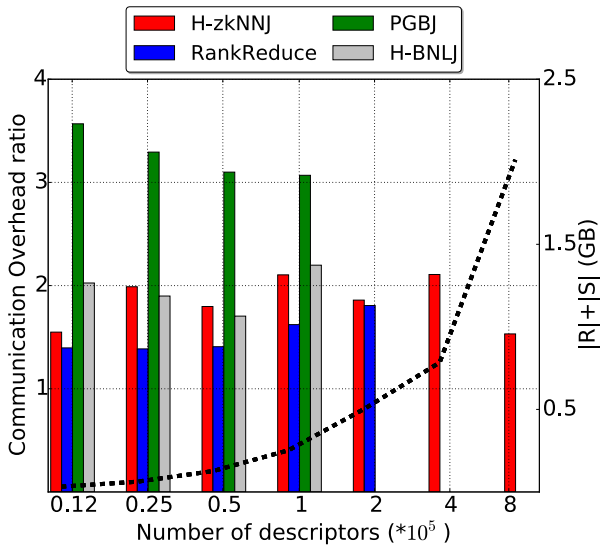
3.5.3 Impact of Dimension and Dataset

We now analyze the behavior of these algorithms according to the dimension of data. Since some algorithms are dataset dependent (i.e the spatial distribution of data has an impact on the outcome), we need to separate data distribution from the dimension. Hence, we use two different kinds of datasets for these experiments. First, we use real world data of various dimensions[15]. Second, we have built specific datasets by generating uniformly distributed data to limit the impact of clustering. All the experiments were performed using $0.5 * 10^5$ records and $k = 20$.

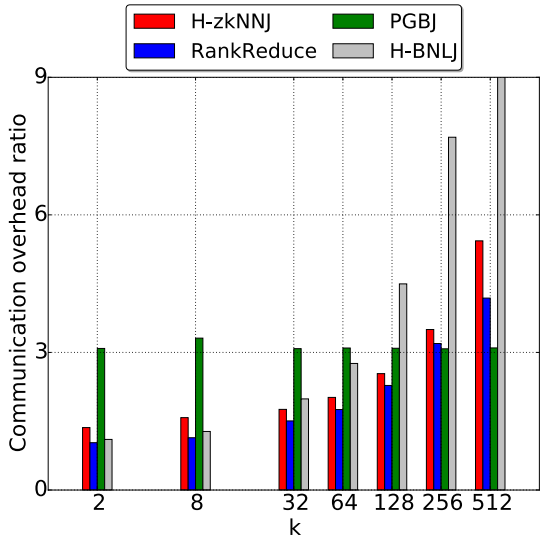
Since **H-BNLJ** relies on the dot product, it is not dataset dependent and its execution time increases with the dimension as seen on Figures 3.21a and 3.22a.

PGBJ is heavily dependent on data distribution and on the choice of pivots to build clusters of equivalent size which improves parallelism. The comparison of execution times for the datasets *128-sift* and *281-blog* in Figure 3.21a shows that, although the dimension of data increases, the execution time is greatly reduced. Nonetheless, the clustering phase of the algorithm performs a lot of dot product operations which makes it dependent on the dimension, as can be seen in Figure 3.22a.

[15] archive.ics.uci.edu/ml/datasets.html



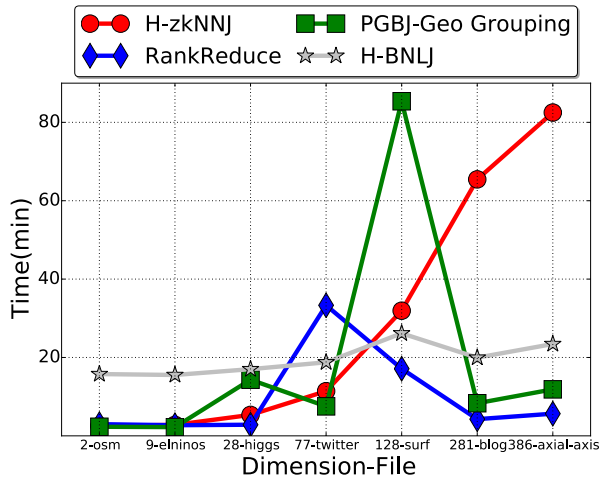
(a) Surf, impact of the dataset size



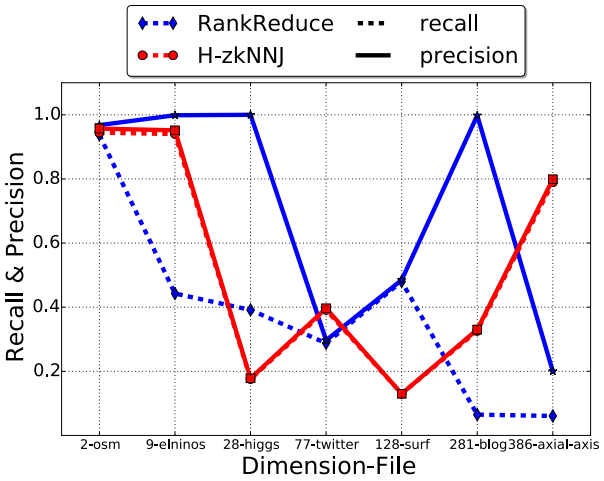
(b) Surf dataset with 50k records, impact of k,

Fig. 3.20 Communication overhead for the Surf dataset

H-zkNNJ is an algorithm that depends on the spatial dimension. Very efficient for low dimension, its execution time increases with the dimension (Figure 3.22a). A closer analysis shows that all phases see their execution time increase. However, the overall time is dominated by the first phase (generation of shifted copies and partitioning) whose time complexity sharply increases with dimension. Data distribution has an impact on the recall which gets much lower than the precision for some datasets (Figure 3.21b). With generated



(a) Execution time

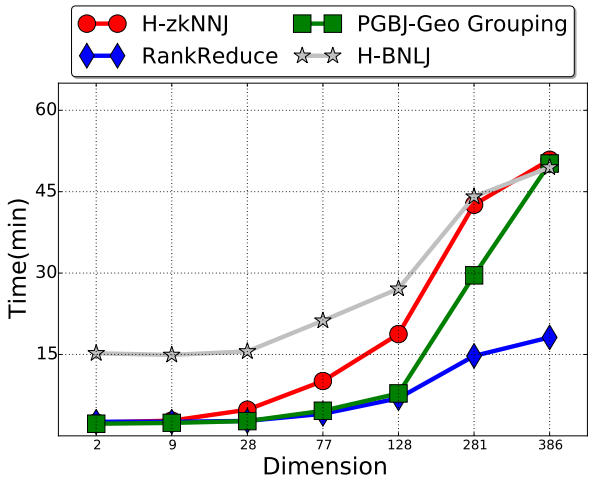


(b) Recall and Precision

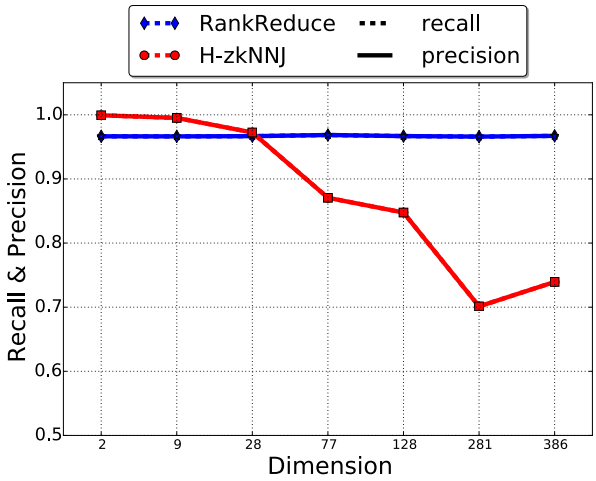
Fig. 3.21 Real datasets of various dimensions

dataset (Figure 3.22b), both recall and precision are identical and initially very high. However as dimension increases, the recall decreases because of the projection.

Finally, **RankReduce** is both dependent on the dimension and distribution of data. Experiments with the real datasets have proved to be difficult because of the various parameters of the algorithm to obtain the requested number of neighbors without dramatically increasing the execution time (see discussion in Section 3.5.4.5). Despite our efforts, the precision was very low for some datasets, in particular *28-higgs*. Using the generated datasets, we see that its execution time increases with the dimension (Figure 3.22a) but its recall remains stable (Figure 3.22b).



(a) Execution time



(b) Recall and Precision

Fig. 3.22 Generated datasets of various dimensions

3.5.4 Practical Analysis

In this section, we analyze the algorithms from a practical point of view, outlying their sensitivity to the dataset, the environment or some internal parameters.

3.5.4.1 H-BkNNJ

The main drawback of **H-BkNNJ** is that only the Map phase is in parallel. In addition, the optimal parallelization is subtle to achieve because the optimal number of nodes to use is defined by $\frac{input\ size}{input\ split\ size}$. This algorithm is clearly not suitable for larger datasets but because of its simplicity, it can, nonetheless, be used when the amount of data is small enough.

3.5.4.2 H-BNLJ

In **H-BNLJ**, both Map and Reduce phases are in parallel, but the optimal number of tasks is difficult to find. Given a number of partitions n, there will be n^2 tasks. Intuitively, one would choose a number of tasks that is a multiple of the number of processing units. The issue with this strategy is that the distribution of the partitions might be unbalanced. Figure 3.23 shows an experiment with 6 partitions and $6^2 = 36$ tasks, each executed on a reducer. Some reducers will have more elements to process than others, slowing the computation.

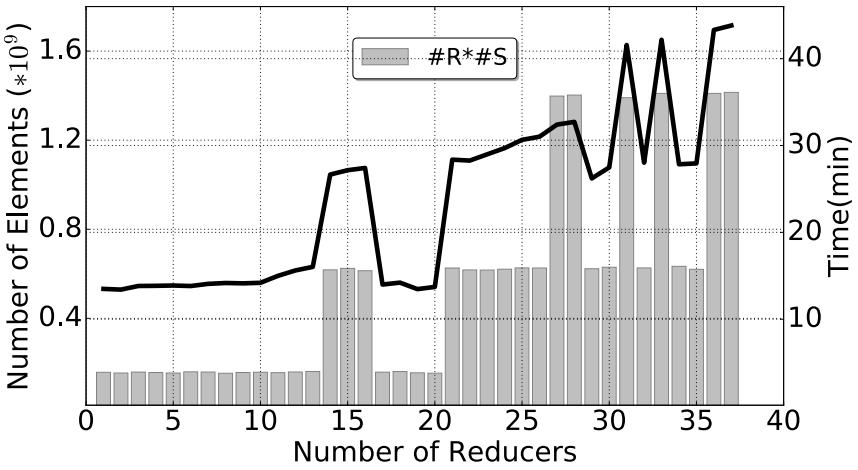


Fig. 3.23 H-BNLJ, candidates job, 10^5 records , 6 partitions, Geo dataset

Overall, the challenge with this algorithm is to find the optimal number of partitions for a given dataset.

3.5.4.3 PGBJ

A difficulty in **PGBJ** comes from its sampling-based preprocessing technique because it impacts the partitioning and thus the load balancing. This raises many challenges. First, how to choose the pivots from the initial dataset. The three techniques proposed by the authors, farthest, k-means and random, lead to different pivots and different partitions and possibly different executions. We found that with our datasets, both k-means and random techniques gave the best performance. Second, the number of pivots is also important because it impacts the number of partitions. A too small or too large number of pivots decreases performance. Finally, another important parameter is the grouping strategy used (Section 3.3.2.1). In Figure 3.24, we can see that the greedy grouping technique has a higher grouping time (bars) than the geo grouping technique. However, the global computing time (line) using

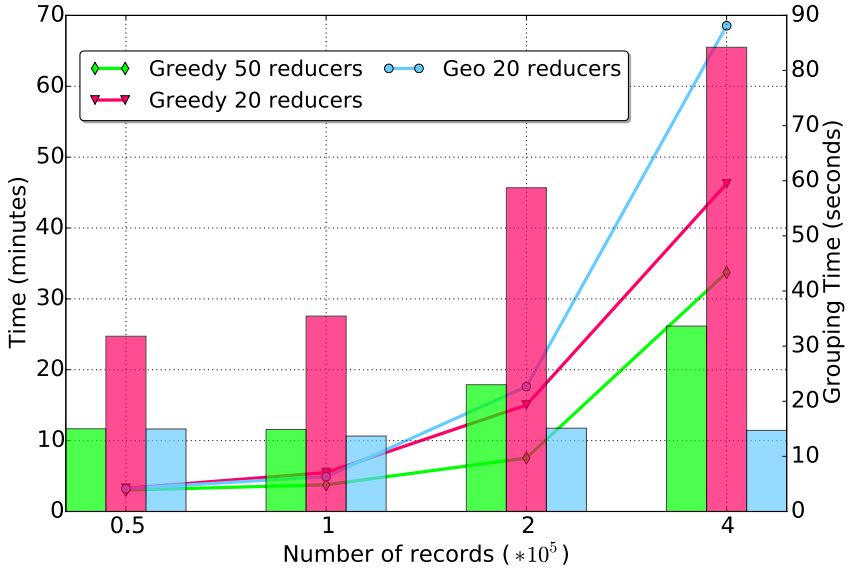


Fig. 3.24 PGBJ, overall time (lines) and Grouping time (bars) with Geo dataset, 3000 pivots, KMeans Sampling

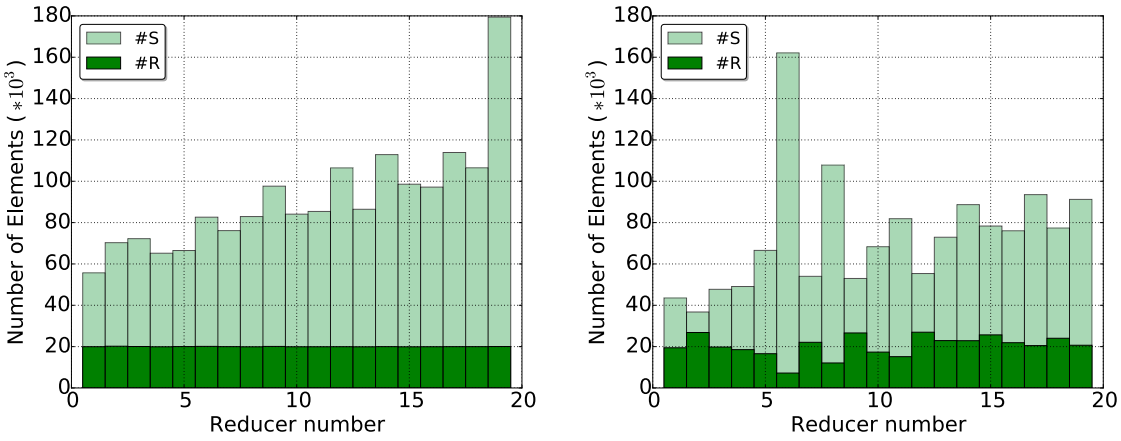
this technique is shorter thanks to the good load balancing. This is illustrated by Figure 3.25 which shows the distribution of elements processed by reducers when using geo grouping (3.25a) or greedy grouping (3.25b).

3.5.4.4 H-zkNNJ

In **H-zkNNJ**, the z-value transformation leads to information loss. The recall of this algorithm is influenced by the nature, the dimension and the size of the input data. More specifically, this algorithm becomes biased if the initial data is very scattered, and the more input data or the higher the dimension, the more difficult it is to draw the space filling curve. To improve the recall, the authors propose to create duplicates in the original dataset by shifting data. This greatly increases the amount of data to process and has a significant impact on the execution time.

3.5.4.5 RankReduce

RankReduce, with the addition of a third job, can have the best performance of all, provided that it is started with the optimal parameters. The most important ones are W, the size of each bucket, L, the number of hash families and M, the number of hash functions in each family. Since they are dependent on the dataset, experiments are needed to precisely tune them. In [72], the authors suggests this can be achieved with a sample dataset and a theoretical model.



(a) Geo (b) Greedy

Fig. 3.25 PGBJ, load balancing with 20 reducers

The first important metric to consider is the number of candidates available in each bucket. Indeed, with some poorly chosen parameter values, it is possible to have less than k elements in each bucket, making it impossible to have enough elements at the end of the computation (there are less than k neighbors in the result). On the opposite, having too many candidates in each bucket will increase too much the execution time. To illustrate the complexity of the parameter tuning operation, we have run experiments on the Geo and SURF datasets. First, Figure 3.26 shows that, for the Geo dataset, increasing W improves the recall and the precision at the expense of the execution time, up to an optimal before decreasing. This can be explained by looking at the number of buckets for a given W. As W increases, each bucket contains more elements and thus their number decreases. As a consequence, the probability to have the correct k neighbors inside a bucket increases, which improves the recall. However, the computational load of each bucket also increases.

A similar pattern can be observed with the SURF dataset (Figure 3.27, left), where increasing W improves the recall (from 5% to 35%) and the precision (from 22% to 35%). Increasing the number of families L greatly improves both the precision and recall. However, increasing the number of hash functions M, decreases the number of collisions, reducing execution time but also the recall and precision. Overall, finding the optimal parameters for the LSH part is complex and has to be done for every dataset

After finishing all the experiments, we found that the execution time of all algorithms mostly follows the theoretical analysis presented in Section 3.3.2. However, as expected, the computationally intensive part, which could not be expressed analytically, has proved to be very sensitive to a lot of different factors. The dataset itself, through its dimension and the

Algorithm	Advantage	Shortcoming	Typical Usecase
H-BkNNJ	Trivial to implement	1. Breaks very quickly 2. Optimal parallelism difficult to achieve a priori	Any tiny and low dimension dataset ($\sim$ 25000 records)
H-BNLJ	Easy to implement	1. Slow 2. Very large communication overhead	Any small/medium dataset ($\sim$ 100000 records)
PGBJ	1. Exact solution 2. Lowest disk usage 3. No impact on communication overhead with the increase of k	1. Cannot finish in reasonable time for large datasets 2. Poor performance for high dimension data 3. Large communication overhead 4. Performance highly depends on the quality of a priori chosen pivots	1. Medium/large dataset for low/medium dimension 2. Exact results
H-zkNNJ	1. Fast 2. Does not require a priori parameter tuning 3. More precise for large k 4. Always give the right number of k	1. High disk usage 2. Slow for large dimension 3. Very high space requirement ratio for small values of k	1. Large dataset of small dimension 2. High values of k 3. Approximate results
RankReduce	1. Fast 2. Low footprint on disk usage	1. Fine parameter tuning required with experimental set up 2. Multiple hash functions needed for acceptable recall 3. Different quality metrics to consider (recall + precision)	1. Large dataset of any dimension 2. Approximate results 3. Room for parameter tuning

Table 3.5 Summary table for each algorithm in practice

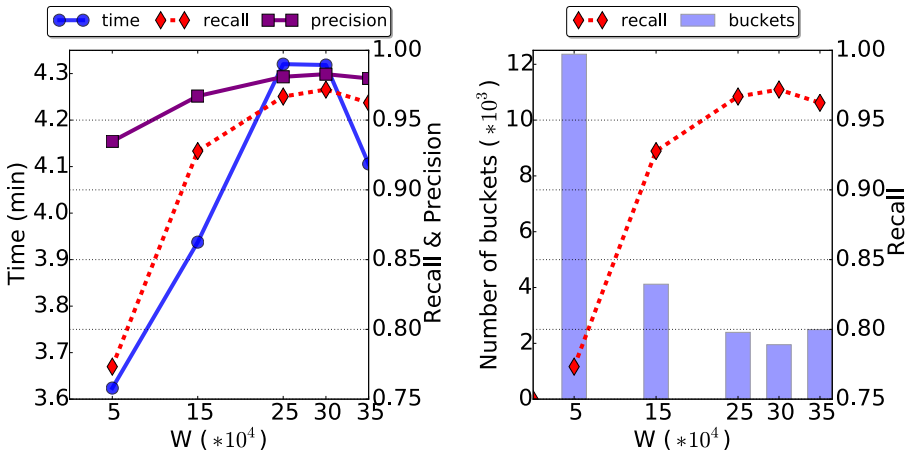


Fig. 3.26 LSH tuning, Geo dataset, 40k records, 20 nodes

data distribution, but also the parameters of some of the pre-processing steps. The magnitude of this sensitivity and its impact on metrics such as recall and precision could not have been inferred without thorough experiments.

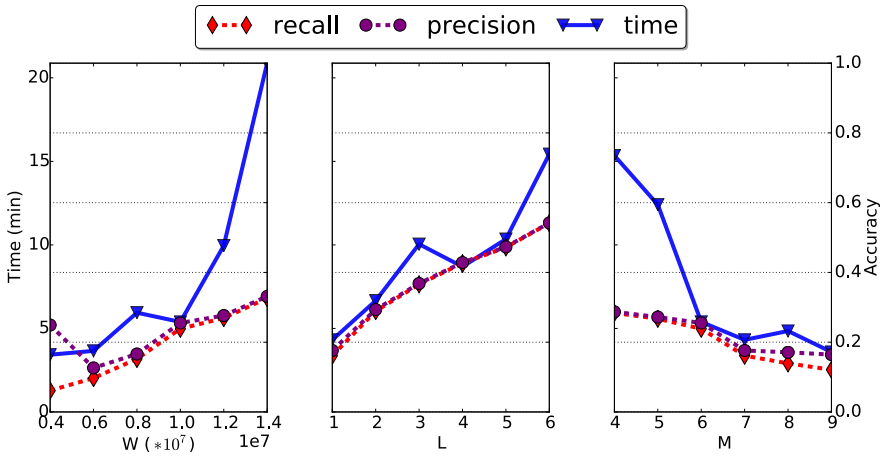


Fig. 3.27 LSH tuning, SURF dataset, 40k records, 20 nodes

3.5.5 Streaming Evaluation

In the parallel kNN method for processing streaming data, the performance follows the same discipline as in a parallel kNN method for processing static data. The only difference is, in a streaming processing platform like Storm, the output of one phase of calculation can be directly used as the input of the next phase of calculation. Thereby, eliminating the time for submitting different jobs and the time for initializing the platform for each job. Normally, for the same amount of calculation, Storm needs less time than Hadoop, especially when multiple jobs are needed.

The evaluation is run on Grid 5000 [16], with 15 nodes. Among them, one is reserved to Nimbus, and the other 14 nodes are used for computing. The setting of the cluster is: 15 nodes with 2 CPUs Intel Xeon E5-2660 v2; 10 cores/CPU; 126GB RAM; 5x558GB SATA hard drives; 10Gbps ethernet; Opteron 2218 Processors; and 8GB Memory. Storm Version is 1.0, and we only use one slot on each machine to avoid the competition of disks (since disk IO affects the performance of Hadoop the most).

In this part of experiment, we will evaluate the streaming feature of the algorithm. The criteria for judging a streaming algorithm are mainly the **Execution Latency** and the **Process Latency**. The execution latency is the time between the data being read by the system, and the data finishing to be processed by the system (the time that we get all the kNN). The process latency is the time between the data being read by the system, and the data starting to be processed by the system. If the number of machines is fixed, these two latencies depend on two parameters, which are: the Sliding Window Size and the Number of Generation. These two parameters together determine the number of elements in a generation. Since we update

[16]https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home

the calculation by generation, the number of elements inside each generation determines the processing time of each generation, thus the latency of the process.

In this experiment, we implement the basic method (Sliding Block Nested Loop presented in Section 3.4.2.1) for processing DRDS on Apache Storm. In this implementation, we use two Spouts to generate GPS data (in two dimension), one for R and another for S. For each Sliding Window, we partition each data streams into three partitions. So in total, we need 9 LocalBolts to process each combination of the partitions to get the local results. And three GlobalBolts are set to gather the local results and obtain the global results. The Topology of this process is shown in Fig. 3.28 as displayed by Storm. In this topology we have four different roles, they are getR (Spout), getS (Spout), local (Bolt) and global (Bolt). The color represent the computation density. Red means the density is high, which leads to a high latency. Green has a median computation density and latency. Blue has a low computation density and latency. In this figure, each role has a different parallelism (number of processors). Normally, the roles with a high computation latency should be assigned more nodes to process.

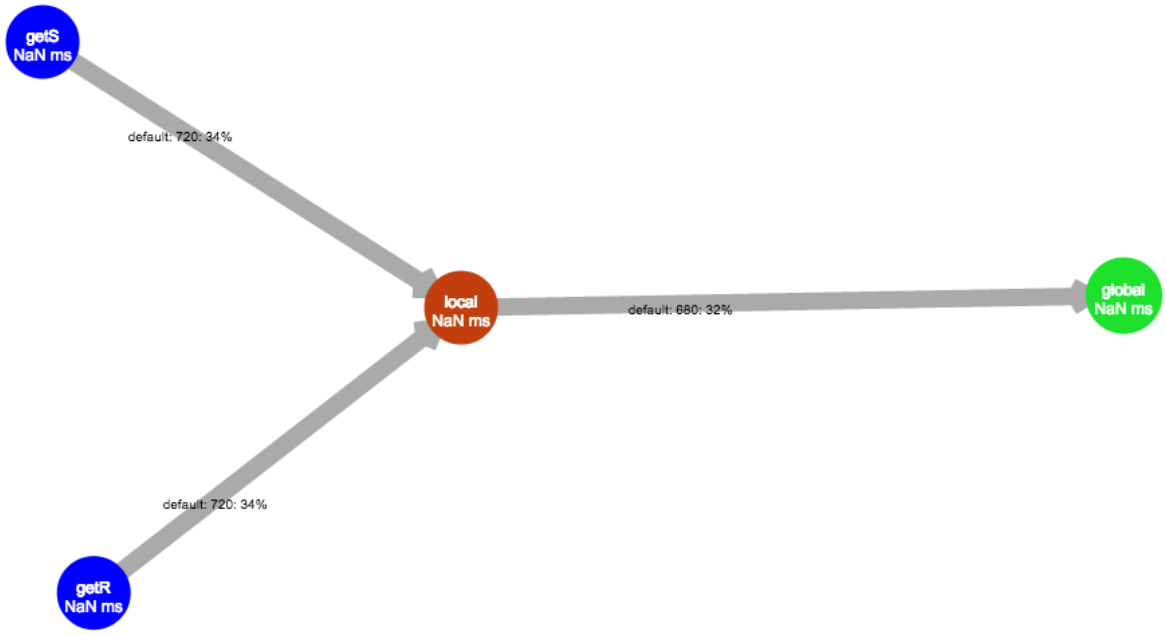


Fig. 3.28 The Topology for the Basic Method of DRDS

First, we fix the size of the Sliding Window to 200 (both for R and S). Then we vary the number of generations as 2, 4, 6, 8, 10 respectively. We measure the execution latency and the process latency after 300 seconds of processing of the Topology. The results are shown in Fig. 3.29 and Fig. 3.30.

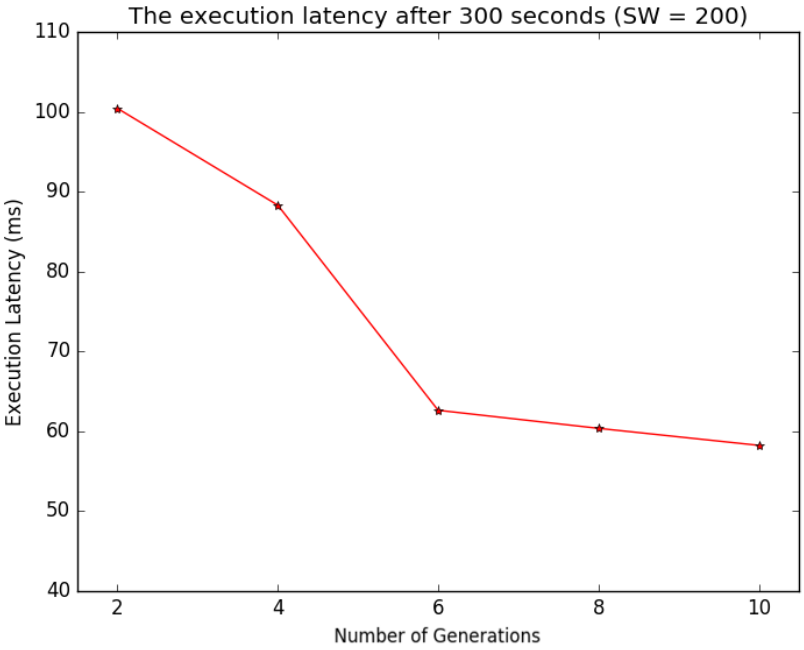


Fig. 3.29 Execution latency after 300 seconds of process (SW=200)

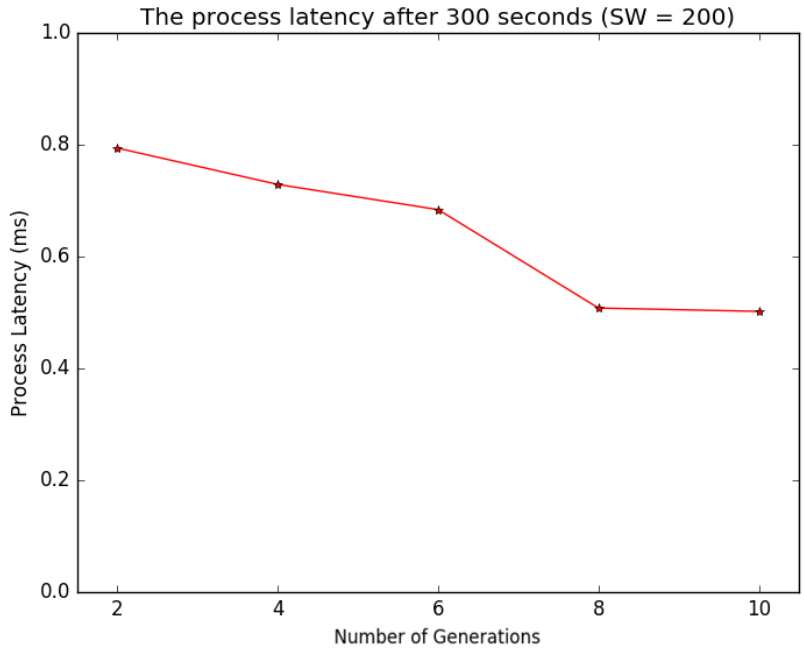


Fig. 3.30 The process latency after 300 seconds of process (SW=200)

For the execution latency, generally, when we increase the sliding window size, the execution latency tends to decrease. This decrease is very fast at the beginning, then it

becomes slower. The ideal number of generations inside a Sliding Window is the value when the curve flattens, as it indicates an equilibrium has been reached in execution latency. Another remarkable point is that, the execution latency is mainly due to the LocalBolt (the Bolt which calculates the local results). That is because comparing with the LocalBolt, the GlobalBolt (the Bolt which combine the local results into the global ones) has lower computation complexity, and less elements to compute than the LocalBolt. Concerning the process latency, it is negligible, which means that the data can be processed in time once it is emitted.

In the second evaluation, we fix the number of generations to 5, then we change the Sliding Window size to 200, 300, 400, 500 and 600 (both for R and S) respectively. We record the execution latency and the process latency after 300 seconds of processing of the Topology. The results are shown in Fig. 3.31 and Fig. 3.32.

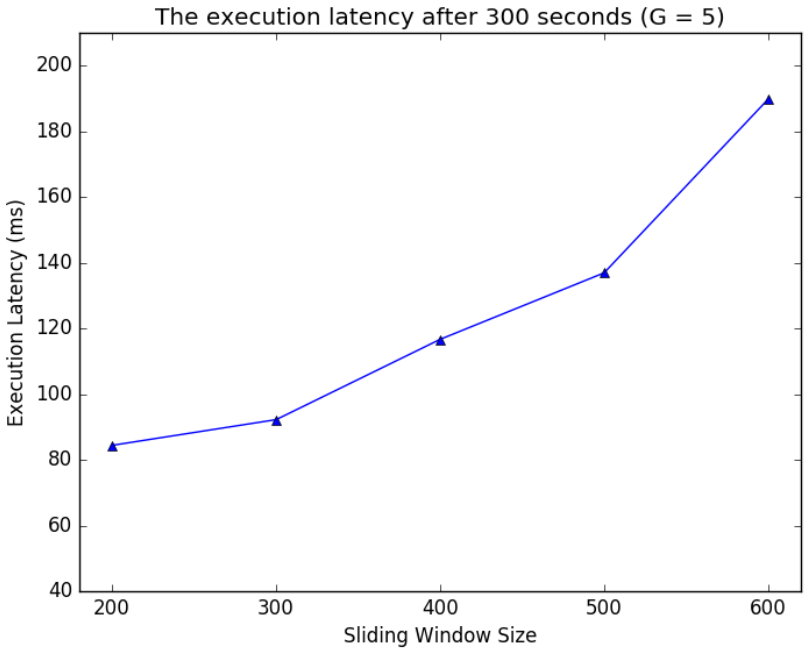


Fig. 3.31 The execution latency after 300 seconds of process (G=5)

In this evaluation, the execution latency usually increases when the Sliding Window size increases. And since, inside of each generation, the new $r \in R$ should be processed with all the $s \in S$ of the current Sliding Window, when the size of the Sliding Window increases, the execution latency shows a sharp increase. This is consistent with the $(N \cdot log(N))$ theoretical complexity of the Local Bolt processing. The process latency is still less than 1 ms. But it also increases with the Sliding Window size.

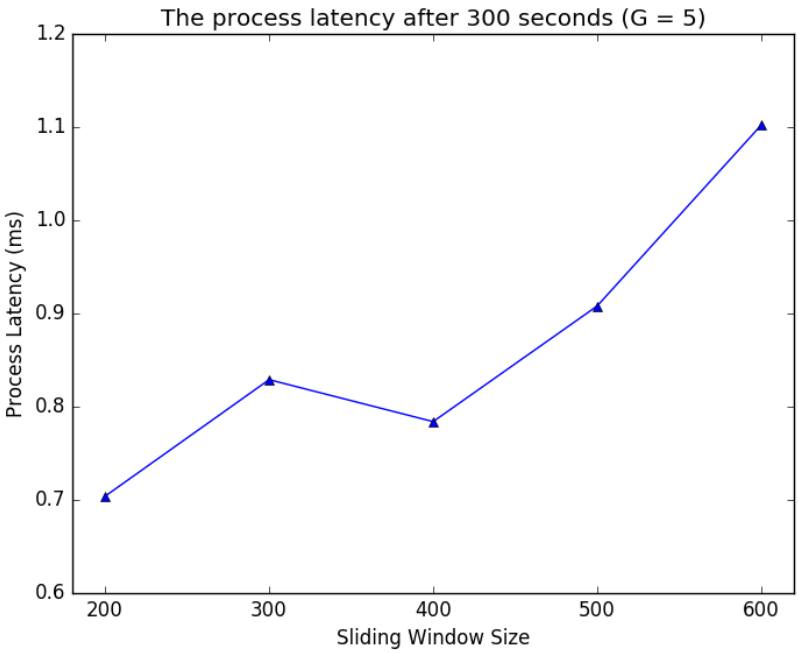


Fig. 3.32 The process latency after 300 seconds of process (G=5)

3.5.6 Lessons Learned

The first aspect is related to load balancing. **H-BNLJ** actually cannot guarantee load balancing, because of the random method it uses to split data. For **PGBJ**, Greedy grouping gives a better load balance than Geo grouping, at the cost of an increased duration of the grouping phase. At the same time, our experiments also confirm that **H-zkNNJ** and **RankReduce**, which use size based partitioning strategies, have a very good load balance, with a very small deviation of the completion time of each task.

Regarding disk usage, generally speaking, **PGBJ** has the lowest disk space requirement, while **H-zkNNJ** has the largest for small k values. However, for large k, the space requirement of all algorithms becomes similar.

The communication overhead of **PGBJ** is very sensitive to the choice of pivots.

The data is another important aspect affecting the performance of the algorithms. As expected, the performance of all algorithms decreases as the dimension of data increases. However, what exceeded the prediction of the theoretical analysis is that the dimension is really a curse for **PGBJ** . Because of the cost of computing distances in the pre-processing phase, its performance becomes really poor, sometimes worse than **H-BNLJ**. **H-zkNNJ** also suffers from the data dimension, which decreases its recall. However, the major impact comes from the distribution of data.

In addition, the overall performance is also sensitive to some specific parameters, especially for **RankReduce**. Its performance depends a lot on some parameter tuning, which requires extensive experiments.

Based on the experimental results, we summarize the advantages, disadvantages and suitable usage scenarios for each algorithm, in Table 3.5.

For the streaming part, it is important to balance the Number of Generations and the Sliding Window Size. Generally, it is good to have a smaller Sliding Window in order to have less execution latency. The number of generations depends on the parameter tuning: the balance should be achieved when the curve becomes flat. But to counterbalance the latency and the Sliding Window size, one can also increase the number of nodes for processing the topology.

3.6 Conclusion

In this Chapter we introduced our methods for processing kNN stream join in a parallel and distributed manner. We have first approached the parallel methods from a workflow point of view. We summarized the main processing steps as data preprocessing, data partitioning and actual computation. We introduced and explained the different algorithms which could be used for each step, and developed their pros and cons, in terms of load balancing, accuracy of results, and overall complexity. We then extended the parallel methods for processing data streams continuously, by designing the re-partition and re-computation strategies. We split the streaming joins in 3 different scenarios: Dynamic R and Static S, Static R and Dynamic S, Dynamic R and Dynamic S, and introduced our strategies for processing each kind of streaming join respectively.

We then performed extensive experiments to compare the performance, disk usage and accuracy of all the parallel algorithms in the same environment, mainly using two real world datasets, a geographic coordinates-based one (in 2 dimensions) and an image-based one (in 128 dimensions). Moreover, we performed a fine grained analysis, outlining, for each algorithm, the importance and difficulty of fine tuning some parameters to obtain the best performance. We also evaluated for the streaming part by comparing the execution latency and the process latency.

Overall, our work first gives a clear and detailed view of the parallel and distributed processing methods for a kNN join. It also clearly exhibits the limits of each of them in practice and shows precisely the context in which they best perform. We then extended the methods for processing a streaming kNN join in a parallel and distributed manner, which makes it possible to deal with large-scale data streams and return the results in real-time.

Chapter 4

Query Driven Continuous Join (RDF)

4.1 Introduction

The continuous popularity of RDF data leads to a growth of the volume of RDF data at a very high speed every year. By now, the RDF data format is one of the data formats with the highest potential to achieve the goal of organizing the world's information and making it universally accessible and useful. But in order to accomplish this target, the amount of data we need to handle has already far exceeded the processing and storage capacity of a single machine. Moreover, static data are increasingly combined to streaming information, leading to streaming reasoning [42]. In such a context, a parallel and distributed system for processing RDF streams and returingn the results in real-time is strongly required.

In this Chapter, we present our work about the design of technologies for processing continuous RDF join in a parallel and distributed manner. As opposed to the Data Driven Join presented in Chapter 3, the conjunctive join for RDF data is a **Query Driven Join**. Unlike kNN join, in an RDF join the format and the dimension of data never changes —- it is always RDF triples. However, the queries written by the users vary a lot. The difficulties of processing this kind of join are different from those of processing Data Driven Joins. For a Query Driven Join, we need less effort to pre-process or partition data, because data is already available in a suitable format. But, we need to prepare the query plan for the user defined queries, and decompose it in order to process it in a parallel and continuous manner.

The rest of this Chapter is organized as follow: Section 4.2 presents the related work, including centralized solutions, parallel and distributed solutions, and the systems for processing continuous RDF streams along with their shortcomings. Section 4.3 introduces our methods about parallel and distributed processing RDF joins. Section 4.4 talks about the strategies for making the methods proposed in Section 4.3 work in a continuous way. Section 4.5 gives the theoretical analyses about accuracy, efficiency and complexity. Section 4.6

shows the implementation issues. The experiment results are shown in Section 4.7. And in the end a conclusion is given in Section 4.8.

4.2 Related Work

4.2.1 Centralized Solutions for Processing Static RDF Data

Most publicly accessible RDF processing systems choose to map RDF triples onto relational tables, such as FORTH RDF Suite [32], Sesame [54], 3store [94], Jena [115] and so on. They choose to use traditional relational databases as their underlying persistent data store. There are two ways of doing this:

(1) Store all triples in a single giant table, with *subject, predicate, object* as generic attributes

(2) Group triples by their predicate, storing triples with the same predicate in the same property table

In scenario (1), RDF data is decomposed into a large number of single statements (triples) which are directly stored in relational tables or hash tables. Then simple statement-based queries can be processed. A statement-based query has one or two variable parts of a triple, and the answer is a set of resources that complement the variables. However, statement-based queries are not the most representative and expressive way of querying RDF data. More complex queries involve multiple filtering steps which are not efficiently supported in relational databases.

Way (2) attempts to create relational-like property tables (for instance Jena [1]). These tables gather together information about multiple properties over a list of subjects. For example in Jena, a property table is defined as a relational database table where each row corresponds to one or more RDF triples. The property URIs for each triples are not stored in the table. A Jena property table has one single column to store the subject. The rest columns store the property values (objects) for the triples. The property URI for the column is stored in the metadata of the table. Here is an example of property tables used by Jena shown in Fig. 4.1.

Still, this strategy does not ensure good performance for queries that can not be answered from one single property table (non-property-bound queries). Another problem of this approach is that it imposes a relational-like structure on semi-structured RDF data. However

[1] https://jena.apache.org/

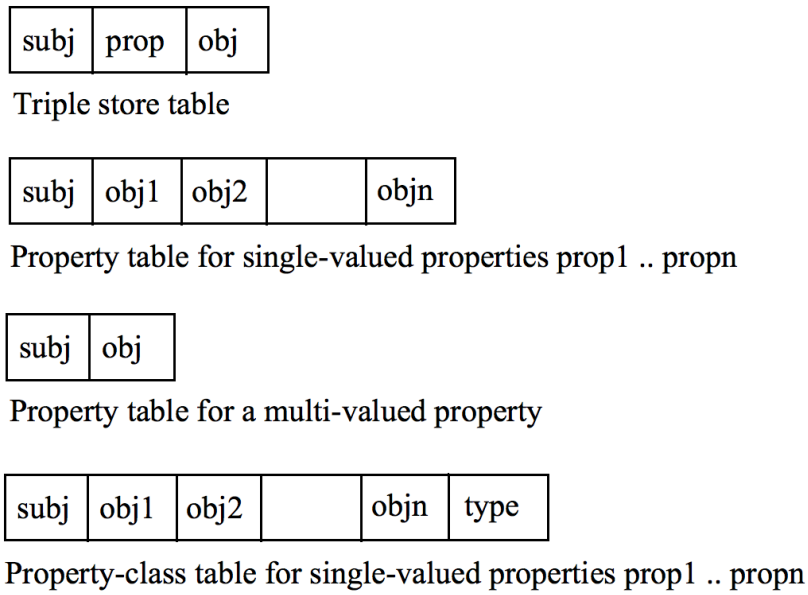


Fig. 4.1 Example of Property Tables (Taken from paper [143])

it results in a lot of NULL values in the property tables, since not all the subjects will use all properties. Handling such sparse tables requests large computational overhead.

There has been a lot of work dedicated to efficiently store and query RDF data. The most frequently used strategies are: Vertical Partitioning and Index Data.

The **Vertical partitioning** approach proposed in paper [27] is an improvement for property tables. In a vertical partitioning scheme, triple tables are rewritten into two-column tables for each property, one column for subjects the other for objects. Each table is sorted by subject in order to quickly locate particular subjects. It has a great advantage for processing queries in which properties act as bound variables. But the authors observe repeatedly the problem of having non-property-bound queries. Actually, this approach suffers from similar scalability problems as the property tables on the queries that are not bound by predicate values.

Another improvement is to **index data**. However, due to the special nature of RDF data — it is based on a triple format, multiple indexes need to be used to ensure the completeness. Here we list some works about indexing RDF data.

Hexastore [142] indexes RDF data in six possible ways. It can be considered as a further improvement of the Vertical Partitioning approach. The difference is that it treats subject, predicate and object equally. In Hexastore, data is indexed in six possible ways, one for each possible ordering of the three RDF elements. Thus each component in an RDF triple has a special index structure. Moreover, every possible pairwise combination of the importance or precedence of the three elements is also indexed. Each RDF instance is associated with

two vectors, each of which gathers elements of one of the other types, and along with lists of the third type resources attached to each vector, which realizes a system that maintains in total six indices. Each index structure in this system is constructed based on one RDF triple component and defines a prioritization between the other two components. According to the authors, this format of index allows quick and scalable general-purpose query processing. But the price is that it occupies five times more in space in the worst-case.

Here is an example of the 'spo' indexing in the Hexastore system shown in Fig. 4.2. In this example, a subject key s_i is associated to a sorted vector of predicate keys $\{p_1^i, p_2^i, ..., p_n^i\}$. Each predicate key p_j^i is, at the same time, linked to an associated sorted list of object keys $\{o_1^{i,j}, o_2^{i,j}, ..., o_k^{i,j}\}$. These objects lists are accordingly shared with the 'pso' index. The same 'spo' pattern is repeated for every subject in the Hexastore. The analogous patterns are realized in the other five indexing schemes.

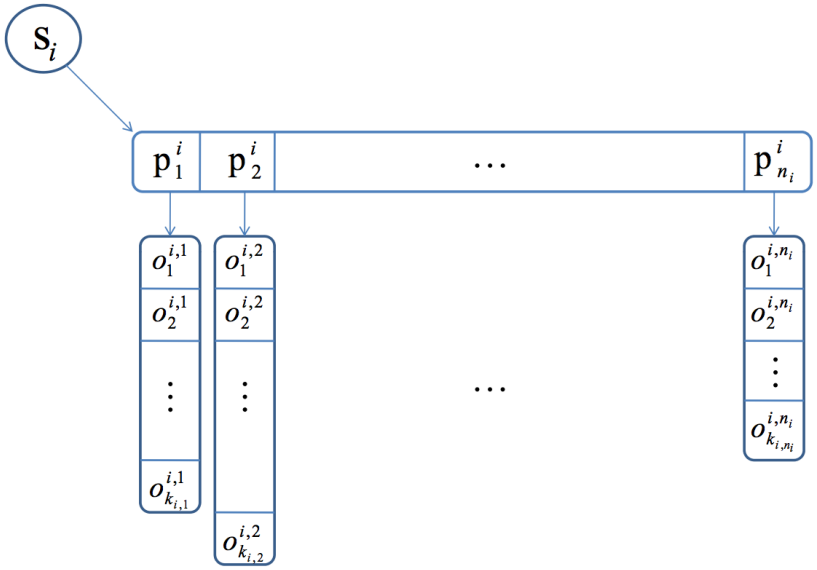


Fig. 4.2 'spo' indexing in Hexastore (Taken from paper [142])

RDF-3X (short for RDF Triple eXpress) [117] is designed for managing and querying RDF data. It designs an architecture for RDF indexing and querying. And it optimizes the join for large RDF sets. In particular, RDF-3X addresses the challenge of schema free data. And it copes well with data exhibiting large diversity of properties.

It stores all triples in a B^+-bree [2] where the triples are sorted lexicographically. This design makes it possible to converse SPARQL triple patterns into range scans. The indexes of RDF-3X are built over all 6 permutations of the three components that compose an RDF triple. More over, every two-dimensional and one-dimensional projections over the three

[2] https://en.wikipedia.org/wiki/B%2B_tree

components have also been indexed, which makes the number of indexes up to 15. Each of these indexes can be compressed to ensure that the total storage space for all indexes together to be less than the size of the primary data. The design of the indexes eliminates the need for physical-design tuning. The principle of this index method is shown in Fig. 4.3.

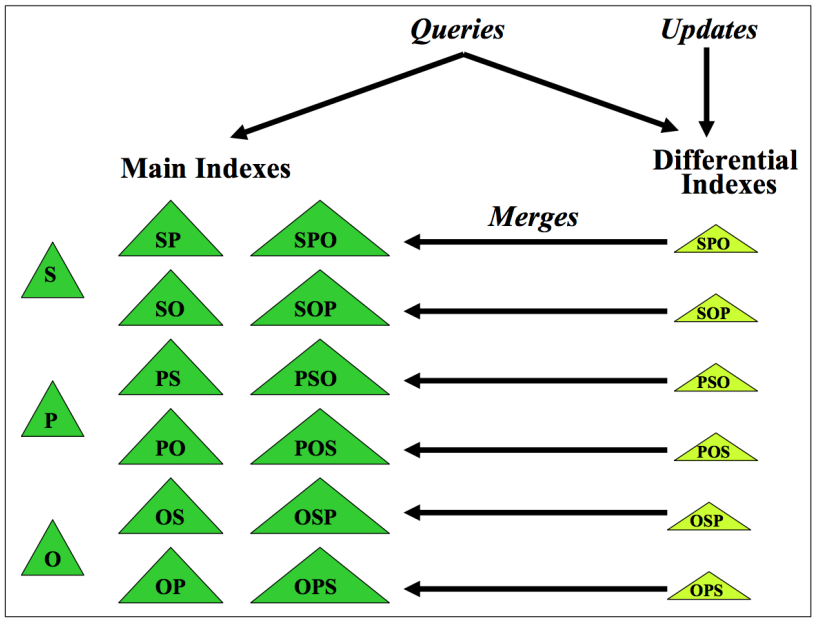


Fig. 4.3 Exhaustive Index of RDF-3X (Taken from paper [117])

The 'exhaustive' index of the triples table makes the query processor of RDF-3X possible to rely mostly on merge joins over sorted index lists. In fact the triples table is a virtual concept, because all processing is index-only. The processing overhead of this approach is much lower than the traditional ones.

The query optimizer focuses on join order in its generation of execution plans. It uses the dynamic programming method for plan enumeration, where the cost model is based on RDF specific statistical synopses such as counters of frequent predicate-sequences.

TripleBit [151] provides a system to store and access RDF data. The system follows a compact design which intends to reduce both the size of stored RDF data and that of the indexes. It provides two auxiliary index structures in order to minimize the cost of index selection during execution. Its query processor can dynamically generate optimal execution orders which aims at improving the join efficiency and to reduce the size of intermediate data. It uses a two dimensional bit matrix to represent data. One dimension (the row) is created based on the union of subject and object, the other dimension (the column) is based on the triple itself. Here we took an example from paper [151]. Suppose we have the following triples:

T1: *person1 isNamed "Tom".*
T2: *publication1 hasAuthor person1.*
T3: *publication1 isTitled "Pub1".*
T4: *person2 isNamed "James".*
T5: *publication2 hasAuthor person2.*
T6: *publication2 isTitled "Pub2".*
T7: *publication1 hasCitation publication2.*
Then the triple matrix is shown in Fig. 4.4.

	isNamed		hasAuthor		isTitled		hasCitation
	T1	T4	T2	T5	T3	T6	T7
person1	1	0	1	0	0	0	0
person2	0	1	0	1	0	0	0
publication1	0	0	1	0	1	0	1
publication2	0	0	0	1	0	1	1
"Tom"	1	0	0	0	0	0	0
"James"	0	1	0	0	0	0	0
"Pub1"	0	0	0	0	1	0	0
"Pub2"	0	0	0	0	0	1	0

Fig. 4.4 Example of Triple Matrix (Taken from paper [151])

In brief, the centralized systems mainly have the following three physical structures:

(1) Triple table. The ideas is to put the triples in a 3 or 4 columns table, where each row represents one RDF statement. This representation is not efficient, because it requires too many self-join operations over this large table for an RDF join.

(2) Property table. It consists of building one or more tables based on common set of attributes which occur frequently. However, not all subjects share the common set of attributes. More over, if a subject has more than one object for a given predicate, then these objects need to be duplicated. In addition, a query may require complex join plans if it needs to search multiple property tables.

(3) Vertical partition. In this strategy, a table with two columns is created for each predicate. The first one contains the subjects of all tuples that share the predicate for this table. And the second one contains the associated object. But this approach has scalability problems.

In order to make the relational representation and management work efficiently, the index strategy is often associated with the above 3 strategies as a forth strategy:

(4) Index data. Many ways of indexing RDF data have been proposed. But due to the special nature of RDF data, which contains 3 parts in each triple, in order to enhance the index completeness and efficiency, multiple indexes need to be used.

However, these methods are not suitable for a parallel and distributed streaming processing system. These duplication or tedious indexes increase the amount of data need to be stored. Thus, it makes the partitioning and distribution of data become very complex. In addition, these platforms typically require a complex query optimization plan, such as the Dynamic Programming method used in RDF-3X to produce logical plans. This also do not fit the requirements for a streaming processing system.

4.2.2 Parallel Solutions for Processing Static RDF Data

Storing huge amount of RDF triples and processing them in a parallel and distributed way is also a challenging problem in the Semantic Web area. Current frameworks like Jena or Virtouso do not scale very well for big data. Because they run on a single machine and cannot process huge amount of triples. For example, we can only load 10 million triples in Jena running in a machine with 2GB of main memory. This is far from the huge amount of triples need to be processed (for example 3 billion RDF triples in DBpedia).

Many works have been done to attempt to process RDF data with MapReduce. Here we list a short introduction about some of these works.

Paper [98] describes a framework built on Hadoop to store and retrieve large amount of RDF triples. The authors store RDF data in HDFS, and they design their algorithms to answer SPARQL queries in a MapReduce manner.

Paper [89] proposes an efficient distributed reasoning engine for the widely-used RDFS and OWL rules. It is built on top of Spark. The authors implement parallel reasoning algorithms with the Spark RDD programming model.

Paper [153] introduces a distributed and memory-based graph engine for processing web scale RDF data. Instead of managing RDF data in triple stores or as bitmap matrices (as explained previously), the authors store RDF data in its native graph form.

We will not discuss the details of the above methods running on MapReduce, because the translation of SPARQL queries into MapReduce workflows is not efficient. For example, the first two methods rely on translating the SPARQL query into a series of SQL joins, then map these SQL join flows directly to MapReduce jobs. But this mapping is actually not efficient. The fundamental reason is that MapReduce can only process parallel data but not independent tasks. It needs to launch an entire MapReduce job for each sub-query which results in costly MapReduce jobs where some subqueries only need a Reduce task. The third

one relies on graph processing. MapReduce is proved to be inefficient in this case, because most of the graph algorithms are iterative, and some of them require a large number of iterations. However, a MapReduce procedure can only conduct one iteration. To implement all these iterations, it involves heavy I/O costs and extra job starting and initialization time.

Generally, in the parallel and distributed platforms for processing RDF data, some works choose to divide data (e.g. [96]), some others want to split queries into subqueries (e.g. [93]), and others want to distribute both (e.g. [95]). Paper [93] classifies the different paradigms for building RDF querying systems into 4 different types, **Q-I** is a centralized one; **Q-II** choose to divide data but not query, which means that every machine stores a piece of data, and processes the entire query; **Q-III** devides both data and query, which means that each machine only processes a sub-query on a sub-set of data; and **Q-IV** choose to divide queries but not data, which means the entire data sets will be stored on every machine but processed by a sub-query, as shown in Fig. 4.5

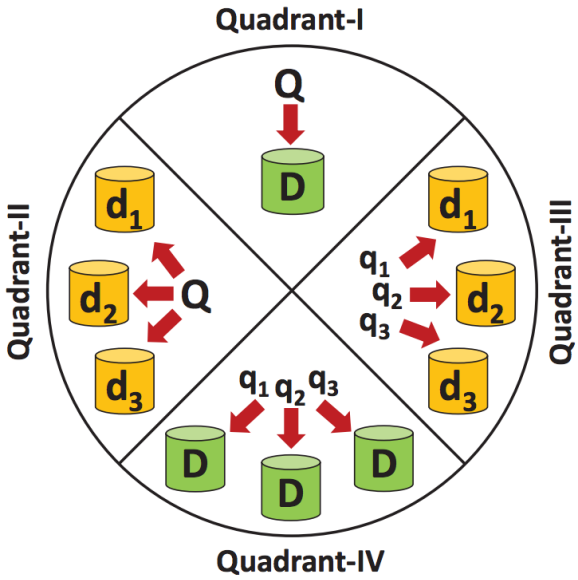


Fig. 4.5 The four different paradigms for building RDF querying systems (Taken from paper [93])

Here we introduce one example for each type of different paradigm.

Q-II In order to solve the bottleneck caused by the transformation of intermediate data, paper [96] has used a graph partition method and replicated the boundary data instead of using a simple hash partition method by s, p, o. However, the authors choose to divide data into different partitions, each partition still has many replications from the other partitions. This requires extra space. Besides, the partition step requires knowledge about the whole dataset, which is not possible for a streaming environment.

Q-III YARS2[95] presents the architecture of a federated distributed graph-structured data repository for a Semantic Web search engine. They divide local data structures with constant seeks and linear throughput, in order to be able to optimize the performance of the individual operations. They also present a general indexing framework for RDF triples which is instantiated through a compressed index structure with near-constant access times (of the index size). They investigate different data placement techniques for distributing the index structure. And they introduce methods for parallel concurrent query processing over the distributed index. Since indexing is not efficient for data streams as explained before, this work cannot be extended for RDF streams neither. In addition, data placement causes huge network communication, which leads to an inability to return the results in real time.

Q-IV DREAM [93] chooses not to distribute the data but only to distribute the sub-queries to avoid the intermediate data transformation, but when the volume of data exceeds the capacity of the storage of one machine, this method will no longer work.

Actually, **Q-II** (distribute only data) may not be efficient in many use cases, especially when the query is complex. Besides, it usually adopts some replication strategies in order to avoid communications among nodes. These replication strategies usually require entire dataset, which is not possible in a dynamic streaming processing system. **Q-IV** (distribute only query) is not scalable when data increases. Because when the size of data is larger than the storing capacity of one machine — which appears in most of the streaming use cases, this kind of method can not work any more. **Q-III** (distribute both data and query) is the best solution that fits a parallel and distributed streaming processing system.

In a **Q-III** method, the most important problem to address is data partitioning. Typically, the partitioning strategies will seriously affect the amount of intermediate data transmission among nodes and load balancing. Unfortunately, data partition is a theoretically NP-Hard problem [92]. It is not obvious to choose an appropriate partitioning strategy. In accordance with the triple and graph features of RDF data, the existing partition strategies for RDF systems can be classified into two categories.

- The first one is based on vertex partitioning methods for graphs. Vertex partitioning is a well-known problem using heuristics and approximates algorithms [28]. Therefore, it is easy to leverage existing graph partitioning theory to partition RDF graphs [96]. Graph partitioning divides RDF Graphs into smaller sub-graphs which share minimum connections among them. However, the high overhead of loading RDF data into an existing graph partitioner will make these methods inefficient on large RDF graphs. Secondly, classic graph partitioning algorithms (such as min-cut) require the entire

graph information in order to make decisions, which is again not possible in stream processing.

- The second method is hash partitioning. It divides RDF triples into smaller and similar sized partitions, in order to get better load balance. Since the random hash partition method will generate an enormous intermediate transmission of data [99], the more commonly used method is to partition triples by hashing their index. The indices are usually based on the permutation of the three parts of RDF triples and their projections [110]. For example, Virtuoso [52] partitions each index of all RDBMS tables; YARS2 [95] uses hashing on the first element of all six alternately indices. Unfortunately, these methods for hashing the index are also not applicable to the stream processing system, in which we do not have enough time and space to calculate and store these indexes.

4.2.3 Continuous Solutions for Processing Dynamic RDF Data

The join approach (such as nested loop join, hash join, merge-sort join etc.) for traditional static data does not work very well for dynamic data. For processing RDF streams, compared to the static join method, the difficulties for stream joins lie in:

(1) The amount of data is too big to be completely stored, so the graph partitioning method or the replication method are not suitable.

(2) It is difficult to decide the query plan in advance, which results in the lack of join order.

The reasoning on RDF data streams is an important step to make logical reasoning in real time for huge and noisy data streams in order to support the decision process of large number of concurrent users. So far, this area has received little attention by the whole Semantic Web community.

The first attempt to extend SPARQL to support streams is Streaming SPARQL [51]. It introduces a syntax for the specification of logical and physical windows (see introductions about sliding window model in Section 2.1.2.2) in SPARQL queries by means of local grammar extensions. However, this work omits talking about some important components, such as aggregation and timestamp functions. It does not follow the established approach where windows are used to transform streaming data into batch processing in order to apply standard algebraic operations. It changes the standard SPARQL operators by making them time-stamp-aware which results in a different query language.

The most significant work for processing streams is probably C-SPARQL [42]. It is an extension of SPARQL designed to express continuous queries for RDF streams. C-SPARQL

queries can be considered as inputs of specialized reasoners to make real-time decisions. In this kind of applications, reasoners deal with the snapshots of knowledge, which are continuously updated by queries.

C-SPARQL extends RDF to RDF streams in form of adding a time stamp. Each data in RDF triple has a time stamp as following:

$$\begin{gathered}
\ldots\ldots \\
(< sub_i, pre_i, obj_i >, \tau_i) \\
(< sub_{i+1}, pre_{i+1}, obj_{i+1} >, \tau_{i+1}) \\
\ldots\ldots
\end{gathered}$$

Time stamps are annotations of RDF triples, and they are created in monotonically non-decreasing order.

C-SPARQL uses a sliding window model, where a window extracts the last data elements from RDF streams. So only part of the stream are considered by one execution of the query. This extraction can be both physical and logical. Physical extraction means the window contains a given number of triples (count-based sliding window). Logical extraction refers to all triples occurring within a given time interval, whose number is variable over time (time-based sliding window).

C-SPARQL is a very interesting attempt to extend traditional query languages like SPARQL to work in a continuous manner. But unfortunately, it works on a single machine, and it does not have any extension for working in a parallel and distributed way.

To the best of our knowledge, much research is still needed to construct a parallel and distributed platform which can process the join of RDF streams continuously. This kind of research has a high prospect of applications.

4.3 Parallel Join on RDF Streams

In this Section, we will introduce our strategies for processing RDF joins on a parallel and distributed environment. The RDF triples will be distributed among several machines to be pended; and the queries will be decomposed into sub-queries, and distributed to the corresponding machines to be processed. Let us see the example shown in Fig. 4.6 in order to clarify this process.

In this example, we want to find the person who has friend *"person1"*, and likes the post *"post1"* and attends to the event *"event1"* at the same time. This query could be an often used query in social networks like Facebook or Twitter. This is a join processing on the subject *?S*. It is composed by 3 triple patterns, which are represented by *Q1*, *Q2* and *Q3* respectively.

```
SELECT ?S
WHERE{
  Q1    ?S “hasFriend” person1 .
  Q2    ?S “likePost” “post1” .
  Q3    ?S “attendEvent” “event1”
}
```

Fig. 4.6 A SPARQL query example

Suppose the triples we need to apply for this join are the following ones shown in Table 4.1.

Table 4.1 Triples

Triple	Subject	Predicate	Object	Triple	Subject	Predice	Object
T1:	person0	"hasFriend"	person1	**T13:**	person1	"likePost"	"post4"
T2:	person2	"hasFriend"	person1	**T14:**	person2	"likePost"	"post2"
T3:	person3	"hasFriend"	person1	**T15:**	person3	"likePost"	"post4"
T4:	person4	"hasFriend"	person1	**T16:**	person8	"likePost"	"post1"
T5:	person5	"hasFriend"	person1	**T17:**	person2	"likePost"	"post5"
T6:	person1	"hasFriend"	person6	**T18:**	person2	"likePost"	"post3"
T7:	person2	"hasFriend"	person7	**T19:**	person2	"likePost"	"post4"
T8:	person8	"hasFriend"	person4	**T20:**	person3	"likePost"	"post1"
T9:	person0	"likePost"	"post1"	**T21:**	person0	"attendEvent"	"event1"
T10:	person0	"likePost"	"post2"	**T22:**	person0	"attendEvent"	"event2"
T11:	person0	"likePost"	"post3"	**T23:**	person3	"attendEvent"	"event1"
T12:	person1	"likePost"	"post1"	**T24:**	person1	"attendEvent"	"event2"

This process is shown in Fig. 4.7. In this figure, each machine is represented by a rectangle, and it needs to process a sub-set of the RDF data using a sub-query.

According to our understanding, in order to complete the target of parallel and distributed processing for RDF data streams, we need to address the following three problems:

(1) Partition the RDF streams, and disperse these sub-streams to the nodes

(2) Decompose the queries into sub-queries and assign these sub-queries to the appropriate nodes

(3) Reply rapidly to the changes of data (the expiration of old data, and the update of new data), and return the results in real-time

The strategies of step (1) and (2) need to ensure that the amount of data transmitted among the nodes is as low as possible so that it will not become the bottleneck of the system. The

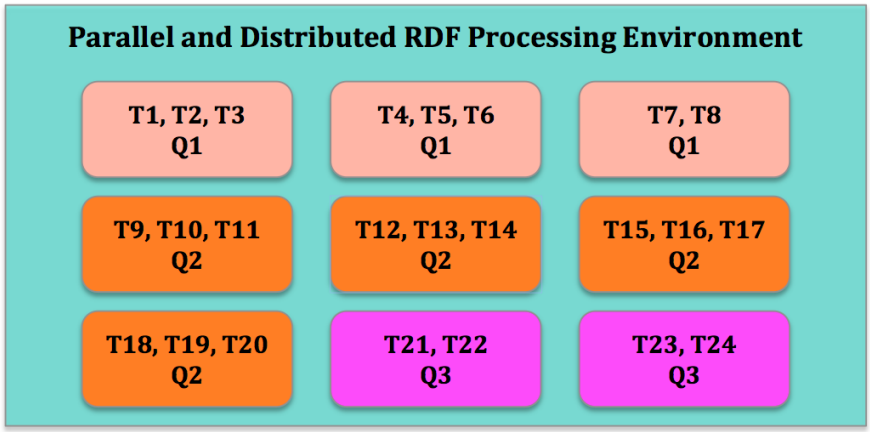


Fig. 4.7 Process RDF joins in a parallel and distributed envirnoment

decomposition method of step (2) should avoid using complex or time-consuming algorithms. The third step could be achieved only if the first two steps are well designed.

In this Section, we introduce a method to process conjunctive joins over massive RDF streams in a parallel and distributed way. In this process, data is dynamic but the query is static. So it is better to first decompose the query, then adapt the data to the sub-queries. The number of machines used to process each sub-query can be dynamically decided by the number of triples this sub-query needs to process in order to achieve load balance.

In this Section, we will introduce the following parts of our work:

- Firstly, we decompose SPARQL queries into triple patterns, and distribute them to the nodes. The number of nodes for processing each triple pattern needs to be dynamically adapted to the number of triples to be processed.
- Secondly, we partition RDF triples according to their predicate. The sub stream of triples with the same predicate needs to be assigned to the nodes that hold the triple pattern with the same predicate.
- Finally, we classify the joins into three types according to their shape. And an appropriate query plan is proposed, taking into account the communication among nodes and data transfer order.

The transmission of information among nodes is carried by a Bloom Filter [48]. Compared to other distributed platforms, we do not directly transfer triples, but transfer Bloom Filters, thereby greatly reducing the amount of data to be transferred. Our method does not need any index, data replication or complex query optimization strategy, which perfectly fit the needs of a parallel stream processing platform.

The rest of this Sections is organized as follows: Section 4.3.1 introduces the query decomposition; Section 4.3.2 talks about data partition and assignment; Section 4.3.3 presents the query planner.

4.3.1 Query Decomposition and Distribution

For a parallel and distributed RDF stream processing system, the decomposition strategy should be simple and lightweight, not requiring any complex method for selecting data. The general idea to decompose the queries is simple: we just divide the queries into triple patterns — which are the basic elements in a query, and send each triple pattern to some corresponding machines.

The reasons for choosing this method are numerous:

- It is simple, and does not require any complicated computations.
- The sub-streams can easily be assigned to the sub-queries.
- The performance or accuracy of this method does not depend on the index or replication of data.
- Among the triples processed by a query, the number of different subjects or objects involved could be tens of thousands; But the number of triple patterns is a limited and fixed number.

In our design, as shown in Fig. 4.7, we both have:

(1) Each triple pattern is processed by multiple machines

(2) Each machine processes multiple triple patterns

Strategy (1) is used to achieve good load balance, because different triple patterns have to process different number of triples. Strategy (2) is naturally applied when there are not enough machines assigned for every predicate. We will introduce these strategies separately.

Each triple pattern processed by multiple machines: To avoid the load imbalance issue caused by different number of triples processed by different triple patterns, a dynamic strategy is used to increase or decrease the number of nodes for processing each triple pattern. This strategy tries to ensure that the number of triples processed by each machine is similar. Thus the processing time for each triple pattern is also similar. The calculation will be introduced in Section. 4.5.2.

Each machine processes multiple triple patterns: Furthermore, one node may also receive multiple triple patterns, especially in the case we do not have enough nodes. To maximize the parallelism of one join, we would like to send triple patterns forming this join to different machines in order to process them in parallel. Another problem is to allow a node to handle multiple triple patterns.

SPARQL Queries can be seen as a sub-graph matching problem. Therefore, the sub-queries should be a sub-graph of an RDF Graph. We use the Edge Coloring method to solve this problem. The edge coloring method is often used to solve scheduling problems. Because of the graph nature of RDF data, this method is suitable for distributing the sub-queries. It is used when we do not have enough nodes for each triple pattern. Furthermore, the coloring of the Ontology [3] can be used for all the queries under this Ontology.

Edge coloring is one type of Graph Coloring. It assigns "colors" as labels to the edges of a graph so that no adjacent edges share the same color.

We can find a minimum number of colors to color the query graph. The minimum required number of colors for the edges of a given graph is called the **chromatic index** of the graph. Suppose the degree of the graph is Δ. Then according to Vizing's theorem[4], the number of colors used for coloring the graph is either Δ or Δ+1. There are many polynomial time algorithms that can construct optimal colorings of a graph, although the general problem of finding an optimal edge coloring is an NP-Complete problem.

An example is shown in Figure 4.8. In this example we omit the information about subjects and objects, and we use the edges to represent predicates. As shown in Figure 4.8, after coloring the edges, the predicates are classified into four categories: $C_1(P_1, P_4, P_9)$, $C_2(P_2, P_3, P_{11}, P_{12})$, $C_3(P_6, P_7, P_8)$ and $C_4(P_5, P_{10})$. The edges with the same color can not form a join (because they are not adjacent), while the edges with different colors may form a join. In order to ensure that the triple patterns sharing the same join variable are processed in parallel, they should be spread over different machines. So the predicates with the same color could be assigned to the same nodes to be processed.

4.3.2 Data Partition and Assignment

For a parallel and distributed RDF stream processing system, the data partition and assignment strategy first needs to be adapted to the query decomposition strategy. It should also be simple and lightweight. In our method, we choose to partition the triples by predicate. The triples with the same predicate will be assigned to the same nodes that hold the triple pattern

[3] https://en.wikipedia.org/wiki/Ontology_(information_science)
[4] https://en.wikipedia.org/wiki/Vizing%27s_theorem

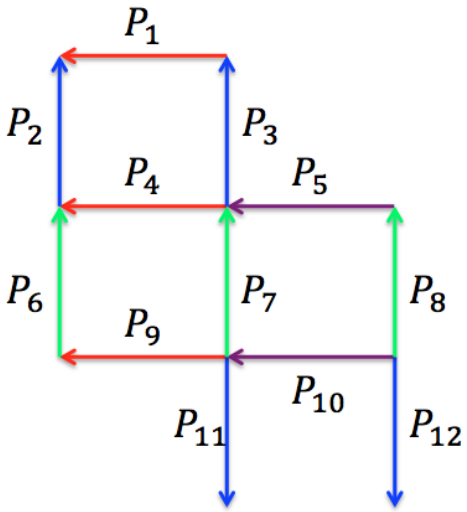


Fig. 4.8 Edges Coloring

with that same predicate. If two or more triple patterns of the query graph use the same predicate, they will receive the same triples.

The reasons for choosing this method are numerous:

- It does not require any index, and will not occupy more disk space.
- It can quickly adapt to changes in data, and it does not require any re-partitioning on the existing data when new data arrives or old data expires.
- Among the triples processed in a query, the number of different types of subjects and objects involved could be tens of thousands; But the number of different predicates is limited, and this number must be smaller than the number of triple patterns that compose this query.

From now till the end of this Chapter, we will use predicates to represent the triples and triple patterns. When we say: we need to treat predicate P_1, it means that we want to process the triples or triple patterns whose predicate is P_1 [5].

4.3.3 Parallel and Distributed Query Planner

There are 3 main problems to be solved to complete this idea and to generate join results efficiently, they are:

[5]Many triple patterns making up the query may have the same predicate. However, since each triple pattern has a different position in the query, they should all be considered as different. If two triple patterns have the same predicate, they will receive the same triples. In the description hereafter, for simplicity, we assume that each triple pattern in a query has a different predicate. But we need to remind the readers that this is not essential.

(1) The communication among nodes

(2) The join of the intermediate results produced by each triple pattern

(3) The order of sending and receiving information

We will introduce our strategies for solving these 3 problems below.

The communication among nodes: The shuffle of the intermediate data will tremendously affect the performance of a parallel and distributed system. In order to avoid the communication through network, some previous works choose not to partition data (e.g. [93]), while some others choose to replicate the boundary data of each sub graph (e.g. [96]), as we introduced in the related works. These two methods are both not suitable in our case. Unlike most other existing systems, in the method proposed in this Chapter, we do not avoid the transmission of data among nodes in order to join the results of each triple patterns. But instead, we use an advanced data structure to minimize the size of the intermediate data need to be transferred.

The communication among nodes in our method is realized through Bloom Filters.

A Bloom Filter is composed of an m-bits array initially set to 0. It uses k hash functions h_i $(1 \leq i \leq k)$ to map elements to this array. Each of the hash functions maps the elements to one of the m positions with a uniform random distribution. To insert an element in the filter, we compute its hashes value with each of the k hash functions, then set the corresponding k bits to 1. To query an element (check whether it is contained by the Bloom Filter), the same hashed values are computed to get k positions. If any of the bits at these positions is 0, then definitely, the element is not contained by the Bloom Filter. Conversely, if they are all 1, then the element is considered to belong to this Bloom Filter.

Because of the collision of hash functions, there exists a small probability of having false positive. This probability is fixed by the number of elements already inserted in the Bloom Filter (n), the number of bits contained by the bit array (m), and the number of hash functions used by this Bloom Filter (k). The calculation of the false positive rate will be presented in Section 4.5.

Figure 4.9 shows the insertion of a dataset $S = \{s_1, s_2, s_3\}$ into a Bloom Filter of 30 bits using 3 hash functions, and the query of three new subjects s_1, s_x, s_y. We can see that s_1 and s_x will be considered as members of S, whereas s_x is a false positive.

The join of the intermediate results of each triple patterns The general idea to join the intermediate results is to use some triple patterns to generate Bloom Filters, and some others to probe these Bloom Filters. First we introduce the notion of join vertices:

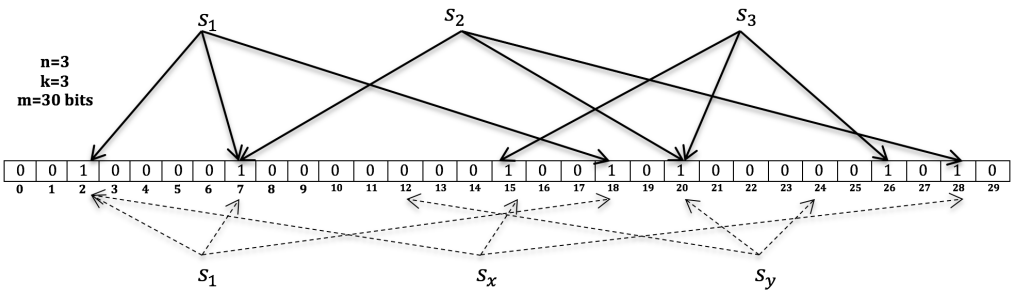


Fig. 4.9 Bloom Filter

Definition 4.1 *The variable vertices with degree greater than 1 in the graph representation of a query are considered as* ***join vertices****. Their degree corresponds to the number of triple patterns involved in this join.*

The nodes which generate Bloom Filters are called **Builders**, and the ones which receive Bloom Filters and use them to probe the triples are called **Probers**. Each join vertex is computed by a combination of several **Builders** and **Probers**. In general, each **Builder** and **Prober** first need to process a triple pattern. Then the **Builders** use the results of the triple pattern to form a Bloom Filter by injecting the projection of the join vertex of each result into the Bloom Filter. The **Probers** check whether the projection on the join vertex of each result is contained by the Bloom Filter.

The difficulty of designing a query plan first lies in how to choose the **Prober** and its corresponding **Builders**.

For the purpose of simplicity, we summarize all the symbols and notations we will use in Table 4.2.

Let J be a join formed by triple patterns $P = \{P_i\}$ (i $\in [1,n]$, with n = $|P|$ the degree of this join). Joins can be divided into three categories according to the number of variables contained by these triple patterns.

The first category is called **1-Variable Join**.

Definition 4.2 *Let J be a join formed by* $P = \{P_i\}$ *with* $i \in [1, n]$, *and* $n = |P|$. *If* $\forall\ i \in [1, n]$ P_i *contains one and only one variable which is the join vertex, join J is called a* ***1-Variable Join****.*

An example of a **1-Variable Join** is shown in Fig. 4.10. In this example, the join vertex is $?S_1$. It contains 3 triple patterns P_1, P_2 and P_3. Each triple pattern has and only has one variable ($?S_1$).

A *1-Variable Join* is the usually encountered *"star-shaped"* join. The join-ordering problem for this kind of join is known or conjectured to be NP-Hard. Since these nodes

Symbol	Definition
π	Projection
σ	Selection
$\bowtie$	Join
$\wedge$	And (Cartesian product)
P_i	A Triple Pattern
$P_{i,l}$	The position of the constant part of Triple Pattern P_i
$P_{i,v}$	The position of the variable part of Triple Pattern P_i
$BF_{P_{i,l}=l_i}(P_{i,v})$	The Bloom Filter containing the results of triple pattern P_i
$BF(P_{i,v})$	Short for $BF_{P_{i,l}=l_i}(P_{i,v})$

Table 4.2 Symbols and their definitions

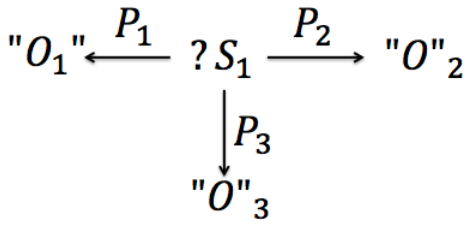


Fig. 4.10 A 1-Variable Join

are connected to many other nodes, it is hard for a graph partitioning method to cut them (because well-connected graph is hard to partition). And since they involve many edges, it is also difficult for the n-hop guarantee methods to classify them [93].

However, the status of all these triple patterns for this join vertex is the same, and the only unknown part of these triple patterns is the join vertex. So these triple patterns can be dispersed on different machines to be processed separately. Then the final results can be considered as the integration of the results of these triple patterns.

For the simplicity of description, the constant part (either IRIs or Literals) of the triple pattern P_i is expressed as $P_{i,l}$ and its variable part as $P_{i,v}$. $BF_{P_{i,l}=l_i}(P_{i,v})$ represents the Bloom filter containing the projection on the variable part ($P_{i,v}$) of the triples whose projection on the constant part is l_i ($P_{i,l} = l_i$). In other words, $BF_{P_{i,l}=l_i}(P_{i,v})$ is the Bloom Filter formed by the results of the triple pattern P_i. To simplify the notion, this Bloom Filter is written as $BF(P_{i,v})$ for short.

Lemma 4.1 *Given a 1-Variable join J with triple patterns* $P = \{P_1, P_2\}$ *and join vertex* ?*v, where* $P_{1,l} = l_1$, $P_{2,l} = l_2$, *and* $P_{1,v} = P_{2,v} =$?*v, then:*

$$\pi(P_{1,v})\sigma_{P_{1,l}=l_1}(P_1) \bowtie \pi(P_{2,v})\sigma_{P_{2,l}=l_2}(P_2) \tag{4.1}$$
$$\approx (\pi(P_{1,v})\sigma_{P_{1,l}=l_1}(P_1)) \wedge \sigma_{P_{1,v}\in BF(P_{2,v})}(P_{1,v}) \tag{4.2}$$
$$\approx (\pi(P_{2,v})\sigma_{P_{2,l}=l_2}(P_2)) \wedge \sigma_{P_{2,v}\in BF(P_{1,v})}(P_{2,v}) \tag{4.3}$$

Equation 4.2 is the Cartesian product of the projection on the variable part of P_1 with the elements of $P_{1,v}$ which belong to the Bloom Filter formed by the elements in $P_{2,v}$. Equation 4.3 can be explained similarly. Equations 4.2 and 4.3 should be an approximate solution of equation 4.1 with $p \cdot n$ more false results where p is the False Positive probability of the Bloom Filter, and n is the number of elements contained by this Bloom Filter.

Lemma 4.1 describes that the final results of the 1-Variable join formed by two triple patterns is approximately equal to the result of one triple pattern filtered by the Bloom Filter containing the results of the other triple pattern. The projections on the variable part of the triples which matches one triple pattern (left part of equations 4.2 and 4.3) filtered by the Bloom Filter containing all the projections on the variable part of the triples which match the other triple pattern (right part of equations 4.2 and 4.3) should be an approximate solution of the 1-Variable join (equation 4.1) with a deviation of the false positive probability of the Bloom Filter.

Proof: Suppose T_1 and T_2 are the sets of triples which match triple patterns P_1 and P_2 respectively.

V_1 and V_2 are two sets of projections on the variable part of the triples in T_1 and T_2 respectively. And,

$$V_1 = \pi(P_{1,v})\sigma_{P_{1,l}=l_1}(P_1),$$
$$V_2 = \pi(P_{2,v})\sigma_{P_{2,l}=l_2}(P_2).$$

Then, $BF(P_{1,v})$ and $BF(P_{2,v})$ are Bloom Filters containing the elements of V_1 and V_2 respectively.

Let $\xi = \{v \| v \in V_1 \wedge V_2\}$, be the result of equation 4.1.

Let $\widehat{\xi} = \{\hat{v} \| \hat{v} \in V_1 \wedge \hat{v} \in BF(P_{2,v})\}$.

By construction of a Bloom Filter we have:

$$\text{if } \hat{v} \in BF(P_{2,v}) \text{ then } \hat{v} \in V_2 \text{ with a probability } p \text{ of false positive.}$$

So ξ is a subset of $\widehat{\xi}$, with $p * n$ less elements, where p is the false positive probability of $BF(P_{2,v})$, and n is the number of elements contained by $BF(P_{2,v})$.

Similarly, equation 4.3 is also an approximate solution of equation 4.1 with $p * n$ more elements. □

Lemma 4.1 can be generalized to Theorem 4.1.

Theorem 4.1 *Let J be a 1-Variable Join formed by n triple patterns $P = \{P_i\}$ with $i \in [1,n]$, $P_{i,v} = ?v$, and $P_{i,l} = l_i$.*

Suppose $\forall\, P_k \in P$, $\widetilde{P} = P - \{P_k\}$, with $\left|\widetilde{P}\right| = n\text{-}1$, then $\forall i \in [1,n]$ and $\forall j \in [1,n-1]$:

$$\bowtie_{i=1}^{n} \pi(P_{i,v})\sigma_{P_{i,l}=l_i}(P_i) \tag{4.4}$$

$$\approx (\pi(P_{k,v})\sigma_{P_{k,l}=l_k}(P_k)) \wedge \sigma_{(P_{k,v})\in\cap_{j=1}^{n-1}BF(P_{j,v})}(P_{k,v}), \tag{4.5}$$

Equation 4.4 represents the join of triple patterns in P. Equation 4.5 denotes the results of the sub-query P_k, which also belongs to the intersection of Bloom Filters formed by the other triple patterns in P.

According to Theorem 4.1, the rule for handling 1-Variable Join can be made as follows:

Rule 4.1 *For the 1-Variable Joins J formed by n triple patterns $\{P_1, ..., P_n\}$, the **Prober** can be chosen as any of these triple patterns, and the **Builders** should be the rest of these triple patterns.*

The **Builders** add the projection on the join vertex of the matching triples to its Bloom Filter. Then the **Prober** pulls all the necessary Bloom Filters, and use them to filter the projection on the join vertex of its matching triples. The results should be those which belong to all these Bloom Filters.

According to Rule 4.1, the distribution of the **1-Variable Join** example in Fig. 4.10 is shown in Fig. 4.11. This is a 1-Variable Join, so we can choose any triple pattern as the **Prober** and the rest as **Builders**. In this example, we choose P_2 as the **Prober**, P_1 and P_3 as **Builders**. The machine holding P_1 needs to first process the sub-query $P_1 <?S_1, P_1, O_1>$, then uses the variable part of the matching triples to form a Bloom Filter $BF(P_{1,v})$. The machine holding P_3 similarly computes a Bloom Filter $BF(P_{3,v})$. The **Prober** P_2 first needs to match with the sub-query P_2, then pull the 2 Bloom Filters $BF(P_{1,v})$ and $BF(P_{3,v})$ and use the projection on the variable part of the matching triples to probe them. In the end, it will return the ones which belong to $BF(P_{1,v})$ and $BF(P_{3,v})$ at the same time as the results of this join.

The second category is called **2-Variable Join**

Definition 4.3 *Let J be a join formed by $P = \{P_i\}$ ($i \in [1, n]$), and the join vertex is ?v. If P contains one and only one triple pattern P_j which has 2 variable parts, join J is called a **2-Variable Join**.*

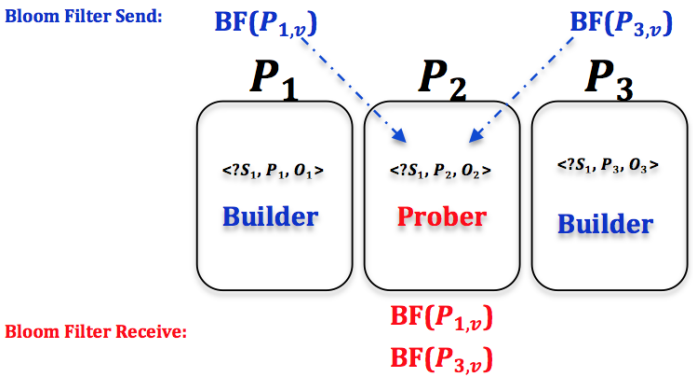


Fig. 4.11 Distribution of 1-Variable Join

An example of a **2-Variable Join** is shown in Fig. 4.12. In this example, the Join vertex is $?S_1$, and it is formed by 3 triple patterns P_1, P_2 and P_3. The triple pattern P_2 has a second variable part $?O_2$ besides the join variable $?S_1$.

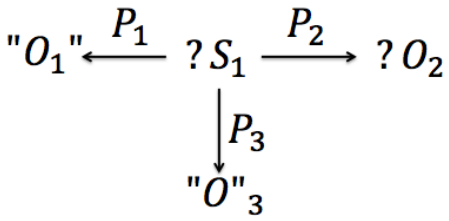


Fig. 4.12 A 2-Variable Join

2-Variable Joins contain two unknown parts, the first one is the join vertex ?v, the second one is the other variable part in P_j. The Bloom Filters used for exchanging information contain only the join vertex ?v, the other parts of the triples can not be carried by the Bloom Filters. So the triple pattern P_j which has two variable parts can only be used as **Prober** but not as **Builder**.

Rule 4.2 *For a 2-Variable Join J formed by n triple patterns $P = \{P_i\}$ (with $i \in [1, n]$) , let $P_{II} = \{P_j\}$ be a subset of P, which contains the only triple pattern with 2 variable parts ($|P_{II}|$=1). Let $P_I = P - P_{II}$ be the set of the remaining triple patterns with only one variable part. Then the **Builders** for this join are all the triples in P_I, and the **Prober** is the only triple in P_{II}.*

Like in 1-Variable Joins, the **Builders** in the 2-Variable Joins add the projection on the join vertex of the matching triples to a Bloom Filter. Then the **Prober** pulls all these Bloom Filters it needs and uses them to filter the projection on the join vertex of its matching triples. The results should be the ones which belong to all these Bloom Filters.

According to Rule 4.2, the solution of the **2-Variable Join** example in Fig. 4.12 is shown in Fig. 4.13.

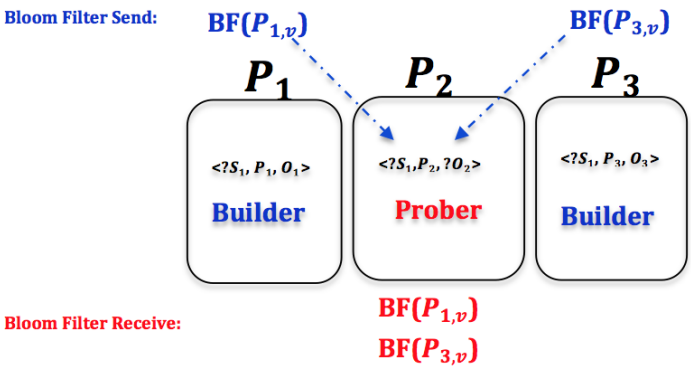


Fig. 4.13 Solution of 2-Variable Join

In this example, the triple pattern P_2 which contains 2 variable parts need to be chosen as the **Prober**. P_1 and P_3 should be the **Builders**. The machines holding P_1 need to first process the sub-query $P_1 <?S_1, P_1, O_1>$. They use the projection on the join vertex of the matching triples to form a Bloom Filter $BF(P_{1,v})$. The machines holding P_3 similarly compute Bloom Filter $BF(P_{3,v})$. The **Prober** P_2 first needs to match with the sub-query $<?S_1, P_2, ?O_2>$, then pulls all these Bloom Filters and uses them to filter the matching triples. In the end, it returns the ones which belong to these two Bloom Filters as the results of this join.

Finally we now study the third category which is called **multiple-Variable Join**

Definition 4.4 *Let J be a join formed by n triple patterns $P = \{P_i\}$ (with $i \in [1, n]$), and the join vertex is ?v. If P contains k (with $1<k\leq n$) triple patterns $P_{II} = \{P_j\}$ (with $j \in [1, k]$) , which contain 2 variable parts (the join vertex plus another), join J is called a* ***multiple-Variable Join****.*

An example of a **multiple-Variable Join** is shown in Fig. 4.14. In this example, the join vertex is $?S_1$. It is formed by 3 triple patterns, P_1, P_2 and P_3. The triple patterns P_2 and P_3 both have two variable parts, the first one is $?S_1$, the other ones are $?O_2$ and $?O_3$ respectively.

As the name suggests, a multiple-Variable join contains multiple unknown parts. The multiple-Variable join is more difficult to treat because it often involves a complex query graph which makes it difficult to determine the communication order.

For simplicity, the variable part other than the join vertex of the triple pattern P_i is expressed as P_{i,v_2}. $BF_{P_{i,v} \in BF_X}(P_{i,v})$ represents the Bloom Filter containing the projection on the join vertex of the matching triples of P_i and filtered by the Bloom Filter BF_X

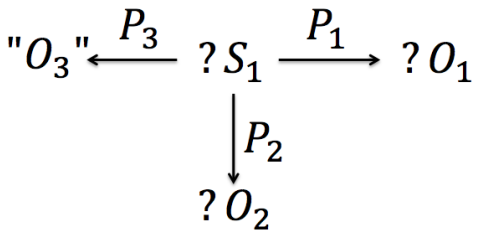


Fig. 4.14 A multiple-Variable Join

Lemma 4.2 *Let J be a multiple-Variable Join with join vertex ?v formed by 3 triple patterns* $P = \{P_1, P_2, P_3\}$*, with* $P_{II} = \{P_1, P_2\}$ *2 triple patterns which have another variable part other than the join vertex, and* $P_I = \{P_3\}$ *the subset containing the triple pattern which only has the join vertex as its variable part.* $P_{1,v} = P_{2,v} = P_{3,v} = ?v$, $P_{1,v_2} = ?v_1$, $P_{2,v_2} = ?v_2$ *and* $P_{3,l} = l_3$.

Let $BF(P_{3,v})$ *(short for* $BF_{P_{3,l}=l_3}(P_{3,v})$*) be a Bloom Filter containing the projection on the join vertex of the matching triples of* P_3.

Let $BF(P_{1,v})$ *(short for* $BF_{P_{1,v} \in BF(P_{3,v})}(P_{1,v})$*) be a Bloom Filter containing the projection on the join vertex of the join results of* $P_{1,v}$ *and* $P_{3,v}$.

Let $BF(P_{2,v})$ *(short for* $BF_{P_{2,v} \in BF(P_{3,v})}(P_{2,v})$*) be a Bloom Filter containing the projection on the join vertex of the join results of* $P_{2,v}$ *and* $P_{3,v}$.

Then,

$$\pi(?v_1)[\pi(?v)\sigma(P_1) \bowtie \pi(?v)\sigma(P_2) \bowtie \pi(?v)\sigma_{P_{3,l}=l_3}(P_3)] \tag{4.6}$$

$$\approx \pi(?v_1)[\pi(?v)\sigma(P_1) \wedge \sigma_{(P_{1,v}) \in BF(P_{2,v})}(P_{1,v})] \tag{4.7}$$

And,

$$\pi(?v_2)(\pi(?v)\sigma(P_1) \bowtie \pi(?v)\sigma(P_2) \bowtie \pi(?v)\sigma_{P_{3,l}=l_3}(P_3)) \tag{4.8}$$

$$\approx \pi(?v_2)[\pi(?v)\sigma(P_2) \wedge \sigma_{(P_{2,v}) \in BF(P_{1,v})}(P_{2,v})] \tag{4.9}$$

Lemma 4.2 states that the query results for the variable parts other than join vertex in a multiple-Variable join is approximately equal to the results of the triple pattern containing this variable part filtered by the Bloom Filter formed by the results of the other triple patterns. In a multiple-Variable join, the process for finding the unknown parts besides ?v is equivalent to finding the qualified triples containing these unknown parts. In Lemma 4.2, the constrain condition for finding the qualified triples is the join vertex ?v. Similarly to the 2-Variable joins, in order to obtain the other unknown part, all the triple patterns containing two variables should be chosen as a **Prober** (the formula $\pi(?v)\sigma(P_1)$ and $\pi(?v)\sigma(P_2)$ in the equations 4.7

and 4.9). Meanwhile, as part of the constraints, they should also be used as a **Builder** at the same time. Nevertheless, the Bloom Filters built by the triple patterns containing 2 variables ($BF(P_{2,v})$ and $BF(P_{1,v})$ in the equations 4.7 and 4.9) are based on the triples already filtered by the Bloom Filter built by the triple pattern containing only one variable part ($BF(P_{3,v})$).
Proof: Suppose T_1, T_2 and T_3 are the sets of triples which match triple patterns P_1, P_2 and P_3 respectively.

Suppose V_1, V_2 and V_3 are the projections on the join vertex of the triples in T_1, T_2 and T_3 respectively.

Then,

$$V_1 = \pi(?v)\sigma(P_1) \tag{4.10}$$

$$V_2 = \pi(?v)\sigma(P_2) \tag{4.11}$$

$$V_3 = \pi(?v)\sigma_{P_{3,l}=l_3}(P_3) \tag{4.12}$$

$BF(P_{3,v})$ is the Bloom Filter containing all the elements in V_3.

Suppose

$$\chi = \{v \| v \in V_2 \wedge V_3\} \tag{4.13}$$

and

$$\widehat{\chi} = \{\hat{v} \| \hat{v} \in V_2 \wedge \hat{v} \in BF_{P_{3,l}=l_3}(P_{3,v})\} \tag{4.14}$$

χ is the results for $\pi(?v)\sigma(P_2) \bowtie \pi(?v)\sigma_{P_{3,l}=l_3}(P_3)$ which is the join of P_2 and P_3, and according to Lemma 4.1, χ is a subset of $\widehat{\chi}$, where $\widehat{\chi}$ also includes some false positive elements.

Now suppose

$$\xi = \{v \| v \in V_1 \wedge V_2 \wedge V_3\} \tag{4.15}$$

and

$$\widehat{\xi} = \{\hat{v} \| \hat{v} \in V_1 \wedge \hat{v} \in BF(P_{2,v})\}. \tag{4.16}$$

Then ξ is the result of $\pi(?v)\sigma(P_1) \bowtie \pi(?v)\sigma(P_2) \bowtie \pi(?v)\sigma_{P_{3,l}=l_3}(P_3)$, which is the projection on the join vertex of the join results of P_1, P_2 and P_3. If $\hat{v} \in BF(P_{2,v})$, then $\hat{v}$ will be in χ with a deviation of false positive probability.

So ξ is a subset of $\widehat{\xi}$. This proves that equation 4.6 is approximately equal to equation 4.7. A similar reasoning can be applied for equations 4.8 and 4.9. □

Lemma 4.2 can be generalized to Theorem 4.2.

Theorem 4.2 *Let J be a multiple-Variable Join formed by n triple patterns P = $\{P_i\}$ (1 < i $\leqslant$ n), where p (1 $\leqslant$ p $\leqslant$ n − 2) triple patterns contain only the join vertex as one variable part*

and $P_I = \{P_j\}$ ($j \in [1, p]$) is the set of such triple patterns, and q (p+q = n) triple patterns contain two variable parts and $P_{II} = \{Q_k\}$ ($k \in [1, q]$) the set of such triple patterns. $P = P_I \cup P_{II}$.

The join vertex is $P_{i,v} = ?v$. The constant part of the triple patterns in P_I is $P_{j,l} = l_j$. The other variable part of the triple patterns in P_{II} is $P_{k,v_2} = ?v_k$.

Suppose $\forall\ Q_x \in P_{II}$, $\widetilde{P_{II}} = P_{II}$ - $\{Q_x\}$, with $\left|\widetilde{P_{II}}\right|$ = q-1, and the other variable part other than join vertex of Q_x is $?v_x$.

Let $BF(P_{j,v}) = BF_{P_{j,l}=l_j}(P_{j,v})$, be the Bloom Filter containing the projection on the join vertex of the matching triples of P_j, and $X = \cap_{j=1}^{p} BF(P_{j,v})$ the intersection of the Bloom Filters built by all the triple patterns in P_I.

Let $BF(Q_{k,v}) = BF_{Q_{k,v} \in X}(Q_{k,v})$, be the Bloom Filter formed by the projection on the join vertex of the matching triples of Q_k and filtered by the intersection of all the Bloom Filters built by the triple patterns in P_I.

Then:

$$\pi(?v_x)(\bowtie_{k=1}^{q} \pi(?v)\sigma(Q_k) \bowtie_{j=1}^{p} \pi(?v)\sigma_{P_{j,l}=l_j}(P_j)), \text{with } Q_k \in P_{II} \text{ and } P_j \in P_I \quad (4.17)$$

$$\approx \pi(?v_x)(\pi(?v)\sigma(Q_x) \wedge \sigma_{Q_{x,v} \in \cap_{k=1}^{q-1} BF(Q_{k,v})}(Q_{x,v}),$$

$$\text{with } Q_k \in \widetilde{P_{II}} \text{ and } P_j \in P_I \quad (4.18)$$

Theorem 4.2 states that the result on the other variable part other than the join vertex of any triple patterns containing 2 variable parts in P_{II}, should be the projection of the join results on this variable part (Equation 4.17). The join results are defined by two constrains. The first one is the Bloom Filters ($\cap_{j=1}^{p} BF(P_{j,v})$) formed by the triple patterns containing only one variable parts in P_I. The second one is the Bloom Filters ($\cap_{k=1}^{q-1} BF_{Q_{k,v} \in \cap_{j=1}^{p} BF(P_{j,v})}(Q_{k,v})$) formed by the results after filtering by the first constrain of the triple patterns with two variable parts in P_{II}.

According to Theorem 4.2, the rule for handling multiple-Variable Joins can be made as following:

Rule 4.3 *For a multiple-Variable Join J formed by n triple patterns $P = \{P_i\}$, with p triple patterns containing only the join vertex $P_I = \{P_j\}$, and q triple patterns containing two variable parts $P_{II} = \{Q_k\}$. Every triple patterns in P_{II} should be **Probers** and **Builders** at the same time. But the elements forming the **Builders** for the triple patterns in P_{II} should be first filtered by the **Builders** formed by the triple patterns in P_I.*

According to Rule 4.3, the solution for the Multiple-Variable Join example in Fig. 4.14 is shown in Fig. 4.15

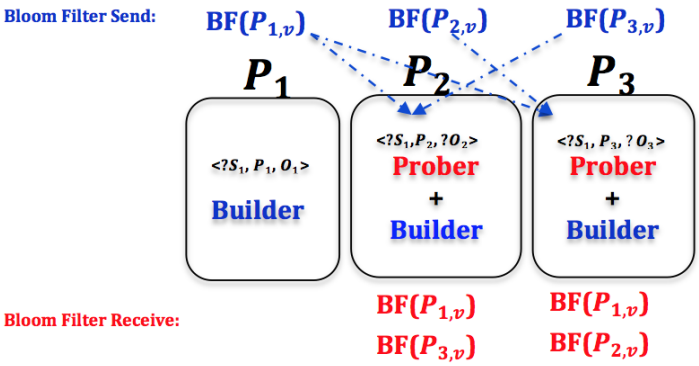


Fig. 4.15 Solution of multiple-Variable Join

In this example, P_2 and P_3 are triple patterns with 2 variable parts, so they need both to be a **Prober** and a **Builder** at the same time. P_1 has only one variable part, so it first needs to process the sub-query $<?S_1, P_1, O_1>$, then use the projection of the matching triples to form $BF(P_{1,v})$. P_2 and P_3 first pull $BF(P_{1,v})$, and use it to filter the matching results of $<?S_1, P_2, ?O_2>$ and $<?S_1, P_3, ?O_3>$ respectively. Then P_2 uses the projection on the join vertex of the filtered results to form $BF(P_{2,v})$ and send it to P_3. P_3 uses the projection on the join vertex of the filtered results to form $BF(P_{3,v})$ and send it to P_2. The results on P_2 should be the remaining ones after filtering by $BF(P_{3,v})$, and those on P_3 should the remained ones after filtering by $BF(P_{2,v})$.

The order of sending and receiving information Until now we introduced our strategies about how to convey information among nodes and how to join the intermediate data of the triple patterns for a join, through the previous three rules. However, in some complex queries (such as nested loop join), there always exist some triple patterns containing two variable parts, which need to be both a **Builder** and a **Prober** (multiple-variable join). The order for transmitting and receiving Bloom Filters is another important issue.

The nature of these three rules is that: for any join vertex J, the dependencies of J are the adjacent nodes around it. If one (or more) dependency is also a variable node, then the dependencies of this variable node are also considered as the dependencies of J.

Thus, a join place can be abstracted into a task, with dependencies. Thereby, a query can be considered as a directed graph. The execution order of this query is a topological sort of this graph. If two nodes of an edge are both join places, they depends on each other. As shown in Lemma 4.2, the direction does not matter so we can choose an arbitrary one.

A topological sort of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex v, u comes before v in the ordering. A

topological sort can be found if and only if the graph is a directed acyclic graph (DAG). Any directed acyclic graph has at least one topological sort.

Different topological sort algorithms exit, such as graph Depth-First Search (DFS), or Kahn's Algorithm. The Kahn's Algorithm first finds a list of starting points whose degree is zero, and insert them in a stack. At least one such kind of points can be found in a DAG. And a solution will be returned in a list. We need to remind the readers that the solution is not necessarily unique. Or we can just use a Depth-First Search to solve the topological sort problem, and the Topological Sort of a graph is the reverse of a Depth-First Search of this graph.

Definition 4.5 *A **query graph** is a graph describing the dependencies of each join place in the query. Each dependency within one triple pattern will only be described once.*

In the topological sort for query graph, we give constant nodes a higher priority over variable ones, which are thus to be sorted after the constant nodes at the same level.

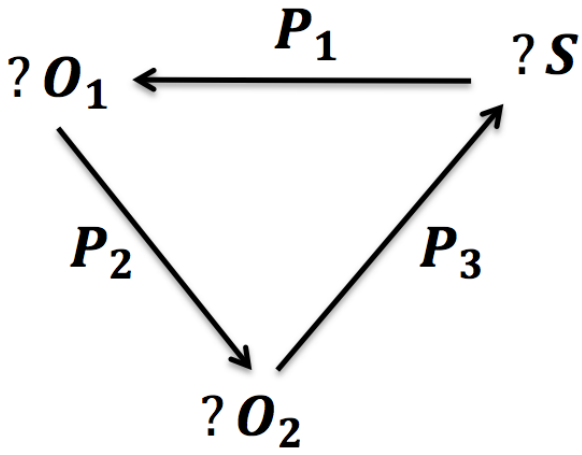


Fig. 4.16 Query with cycle

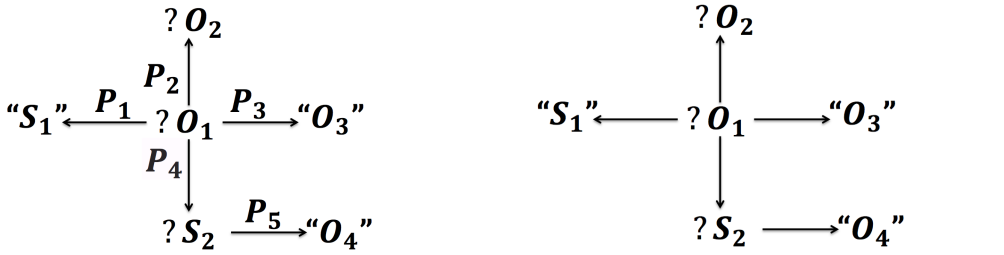
Definition 4.6 ***Query Topological Sort** is a topological sort for the query graphs, where the constant nodes on the graph have higher priority than the variable nodes at the same level.*

Rule 4.4 *Suppose the Query Topological Sort for a query graph of query q is Q. Then, the constrain formed by a variable node n needs to be released after the constrains formed by the constant nodes before n in Q have all been released.*

Since Topological Sort can only be applied for directed acyclic graphs (DAG), unfortunately, our method can not be used for queries whose query graph contains cycles as shown

in Fig. 4.16. It has been proven that this kind of queries can not be processed independently in parallel because a cycle in a graph can not be further partitioned.

An example of the order of sending and receiving Bloom Filters is shown as following. A query q is shown in Fig. 4.17a, and one of its query graph g is shown in Fig. 4.17b.



(a) An example query (b) The query graph of the example

Fig. 4.17 An example sending and receiving order

In this query, there are two join vertices, $?O_1$ and $?S_2$. $?O_1$ is a multiple-Variable join, since P_2 and P_4 both have two variable parts. $?S_2$ is a 2-Variable join, because only P_4 has two variable parts. The **Probers** should be P_2 and P_4. P_1 to P_5 should all be **Builders**. P_2 needs to receive $BF(P_{1,?O_1})$, $BF(P_{3,?O_1})$ and $BF(P_{4,?O_1})$, and build $BF(P_{2,?O_1})$. P_4 needs to receive $BF(P_{1,?O_1})$, $BF(P_{2,?O_1})$, $BF(P_{3,?O_1})$ and $BF(P_{5,?S_2})$, and send $BF(P_{4,?O_1})$.

One query topological sort result from join place $?O_1$ is:

$$\{\text{“}O_4\text{”}, ?S_2, \text{“}S_1\text{”}, \text{“}O_3\text{”}, ?O_2, ?O_1\}.$$

From this result, we can see that $?S_2$ should be released after “O_4”; and $?O_3$ should be released after “O_4”, “S_1” and “O_3”; and the same for $?O_2$. So, P_2 needs to be first filtered by $BF(P_{1,?O_1})$ and $BF(P_{3,?O_1})$ then build $BF(P_{2,?O_1})$ and send it to P_4. P_4 needs to be first filtered by $BF(P_{1,?O_1})$, $BF(P_{3,?O_1})$ and $BF(P_{5,?S_2})$ then build $BF(P_{4,?O_1})$ and send it to P_2. This process vividly describes Theorem. 4.2.

4.4 Continuous RDF Join

In this scenario, we assume that the query needs to process RDF streams which increase over time. To solve this problem, we add the sliding window model and the sliding Bloom Filter model to the parallel RDF query framework we presented previously.

For a streaming system, usually, only the newest updates of the streams will be treated. One of the most commonly used model for processing data streams in this manner is the sliding window model [40].

The sliding window model defines the processing of data within a certain range, which makes sure to process the latest data every time. There are two types of sliding window models. The first one is *time-based sliding window model*, which processes the elements from the last T time units each time. The other type is *count-based sliding window model*, which processes the latest N elements each time. The two main factors affecting the sliding window query efficiency are the *re-evaluation strategies* and *tuple invalidation procedures*. There are usually two re-evalution and expiration mode, the *eager* way and the *lazy* way. The eager way re-executes the query and generates the new results right after each new tuple arrives, then withdraws old tuples upon arrival of each new tuple. However this ideal mode is infeasible in real situations where the streams have very high arrival rates. So real applications (like C-SPARQL) usually run in lazy mode, re-executing the query and removing the old tuples periodically.

In our work, we call the period of re-execution and expiration of the sliding window the **generation**. The generation could be *count-based* or *time-based*, depending on the type of sliding window we choose to use. We can say that the window slides forward in the units of generation, as shown in figure 4.18.

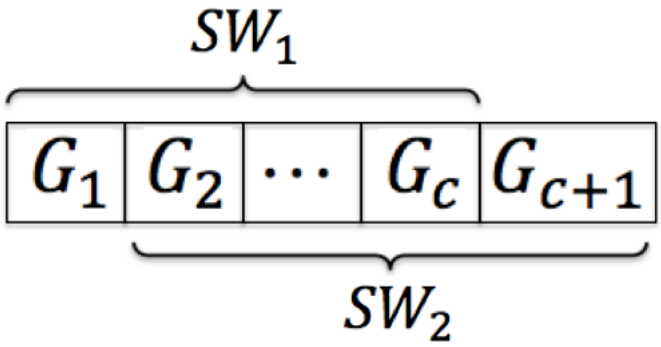


Fig. 4.18 Sliding Window Model with Generation

In a streaming system, a Bloom Filter which is considered as a footprint of data should also be updated for each new generation. To minimize the network communication, we want to send a smaller Bloom Filter without increasing the false positive rate. So, instead of building a Bloom Filter for the whole Sliding Window as done in paper [116], we choose to build a Bloom Filter for each generation. Each time, we only send the Bloom Filter for the current generation to the **Prober**. We store all the Bloom Filters for the current Sliding Window in a list on the **Prober** side. This list is called a Builder list and will be updated when the Sliding Window moves forward. An element x on the **Prober** A will only be considered

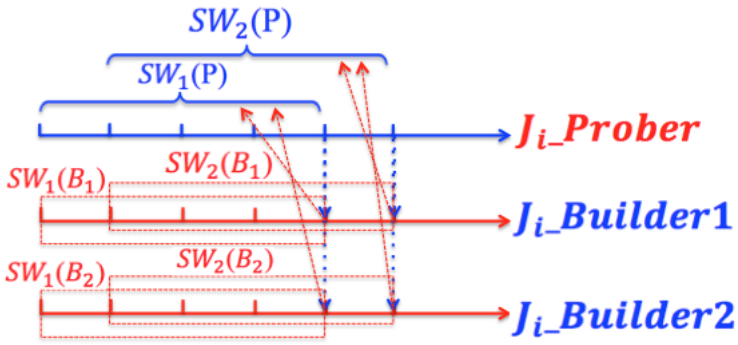


Fig. 4.19 Continuous RDF Join

as contained by the **Builder** B, if it belongs to at least one Generation of the Bloom Filters of **Builder** B.

We call the Sliding Window on the **Probers** the **Prober Window**. The Sliding Bloom Filters are called **Builder Filters**.

Once the stream goes forward a generation, the **Prober Window** will send a request to all the Builders it needs, to pull the desired current Generation of Bloom Filter from the **Builder Filter**, and add it to the Builder list. Then it will check every triples in the current Prober Window to select the triples that belong to all the Builder Filters, and return the results to the user for the current Sliding Window. The processing of the current Prober Windows with the current Builder Filters can be considered as a static RDF Query, which should follow the rules proposed in 4.3.3.

For each join vertex J_i, we use J_i_Prober to represent the sub-queries which need to receive Bloom Filters, and $J_i\_Builder$ to represent the sub-queries which need to build Bloom Filters. The current Prober Window is represented by $SW(P_j)$ and the current Builder Filter is represented by $SW(B_i)$ respectively. The process of the Sliding Window model of a join J_i consisting of two **Builders** and one **Prober** is shown in Fig. 4.19. In this figure, the join J_i consists of one Prober J_i_Prober and two Builders $J_i\_Builder1$ and $J_i\_Builder2$. After finishing one generation, the Prober will request (the blue dashed arrow in the figure) the Builders to send their current generation of Bloom Filter to the Prober (the red dashed arrow in the figure).

All the algorithms presented in this section will be shown in the Section 4.6.

Table 4.3 Symbols and their definitions

Symbol	Definition
k	The number of hash functions used by a Bloom Filter
n	The number of elements need to be inserted in the Bloom Filter
m	The number of bits contained by a Bloom Filter
p	The false positive probability
g	The number of elements contained by a generation
s	The number of elements contained by a Sliding Window
c	The number of generations containing by a Sliding Window

4.5 Analysis

In this Section, we analyze the methods introduced in Section 4.3 and Section 4.4 theoretically. Firstly, we analyze the relations between the number of elements inserted and the dominating parameters of a Bloom Filter such as the number of hash functions and the false positive rate. Then we give the theoretical value of the main parameters in the system, including the number of nodes for processing each triple pattern, and the parameters for constructing each Bloom Filter. Finally, we will analyze the theoretically complexity from the following aspects: the insertion and expiration cost for building a Bloom Filter, the transmitting cost of a Bloom Filter and the probing cost for a Bloom Filter.

4.5.1 Analysis of Bloom Filters

The dominating parameters for a Bloom Filter are shown in table 4.3.

First, we need to calculate the false positive rate of a Bloom Filter.

Lemma 4.3 *Suppose that we use k hash functions to insert n elements into an m bits Bloom Filter, then the probability that a certain bit is 0 should be no more than* $e^{\frac{-kn}{m}}$

Proof: The probability that a certain bit is 0 after the insertion of one elements by one hash function is:

$$1\text{-}\frac{1}{m}$$

So the probability that a certain bit is 0 after the insertion of n elements by k hash function is:

$$(1-\frac{1}{m})^{kn}$$

And:

$$\lim_{m\to\infty}(1-\frac{1}{m})^{kn} = e^{\frac{-kn}{m}}$$

□

Lemma 4.4 *Suppose that we use the simple uniform hashing functions, then the false positive rate p for a standard Bloom Filter is a function of n, m and k, and* $p = (1-e^{-\frac{nk}{m}})^k$

Proof: The simple uniform hashing functions will hash each element to one of the m bits in the Bloom Filter with the same probability. When a certain hash function deals with a certain element, the probability for not setting a certain bit b_x to 1 is:

$$1-\frac{1}{m}$$

So when k hash functions deal with this specific element, the probability for not setting the bit b_x to 1 is:

$$(1-\frac{1}{m})^k$$

And the probability for not setting the bit b_x to 1 when k hash functions deal with n elements is:

$$(1-\frac{1}{m})^{kn}$$

In contrast, the probability for setting this bit b_x to 1 is:

$$1-(1-\frac{1}{m})^{kn}$$

In the query stage, we consider that an element e is in a dataset, if all the hash bits for this element are set to 1 in the Bloom Filter formed by the elements of this dataset. So the false possitive rate is:

$$\mathrm{p} = (1-(1-\frac{1}{m})^{kn})^k$$

$$\begin{gathered}\lim_{x\to 0}(1+x)^{\frac{1}{x}} = e \text{ and } \lim_{m\to\infty}(-\frac{1}{m}) = 0\\ \Rightarrow \lim_{m\to\infty}(1-(1-\frac{1}{m})^{kn})^k\\ = \lim_{m\to\infty}(1-(1-\frac{1}{m})^{-m\times\frac{-kn}{m}})^k\\ =(1-e^{-\frac{nk}{m}})^k\end{gathered}$$

□

Remark 4.1 ***Most of the research works about Bloom Filters consider p =*** $(1-e^{-\frac{nk}{m}})^k$ ***as the false positive rate of a Bloom Filter. Actually, this is not the false positive rate, this is the probability that an element is considered as belonging to the Bloom Filter, so it also contains the true positives. But this probability can be considered as the upper bound of the false positive rate. In this thesis, we choose to use this probability as the probability of false positive. All the calculations are based on this rate.***

Lemma 4.5 *Suppose that we use k hash functions to insert n elements into an m bits Bloom Filter, then the expected number of non zero bits should be* $m \cdot (1-e^{\frac{-kn}{m}})$

Proof: Suppose that X_j is a set of random variables and that X_j=1 when the j^{th} bit is 0, and 0 otherwise, then according to Lemma 4.4

$$E\left[X_j\right] = (1-\frac{1}{m})^{kn} \approx e^{\frac{-kn}{m}}$$

Suppose now that X is a random variable representing the number of the bits which are still 0, then:

$$E\left[X\right] = E\left[\sum_{i=1}^{m} X_i\right] = \sum_{i=1}^{m} E\left[X_i\right] \approx me^{\frac{-kn}{m}}$$

So the expectation of the number of bits which are not 0 should be:

$$m \cdot (1-e^{\frac{-kn}{m}})$$

□

4.5.2 Dominating Parameters

(1) **Dynamically assign the number of nodes for processing triples P_i.**

Generally, the number of triples for each predicate in the data source is different. So the proportion of each predicate in the Sliding Window is also different. If the computation resources are equally assigned to each predicate, it will probably trigger a load imbalance problem caused by data skew (a waste of resources for some predicates, and lack of resources for others). To achieve load balancing, a dynamic allocation of nodes for each predicate is very important.

Suppose the processing capacity of a node is w elements. Suppose the size of a Sliding Window is s, the size of a generation is g. Then the number of generations in the Sliding Window is

$$\mathrm{c} = \frac{s}{g}$$

Suppose $counter_{ij}$ is a counter which counts the number of elements already received for triple pattern P_i in the j^{th} generation, the dynamic allocation strategy will add a new node for predicate P_i when:

$$\sum_{j=1}^{c} counter_{ij} > w$$

Each time a new node is added, the corresponding $counter_{ij}$ will be reduced by w.

(2) **Bloom Filter parameters**

Increasing the number of bits for a Bloom Filter can reduce the chance of hash collisions, which reduces the false positive rate. But the larger the Bloom Filter is, the higher the transmission costs. We assume that if half of the bits in a Bloom Filter is set to 1, then this Bloom Filter reaches the state of equilibrium between space efficiency and hash collision. Under this assumption, we can give the calculations of the relations among the dominating parameters of a Bloom Filter.

Suppose the Bloom Filter contains n elements when it reaches its equilibrium state. Then, the following equation describes the relation among the number of bits of this Bloom Filter, and the number of hash functions it uses:

$$m = \frac{k \cdot n}{50\%} = 2 \cdot k \cdot n \text{ bits}$$

Lemma 4.6 *The false positive p reaches the minimum value, when* $e^{-\frac{nk}{m}} = \frac{1}{2}$. *At this extreme point* $k = ln2 \times \frac{m}{n}$ *and* $p = \frac{1}{2}^k = 2^{-ln2 \times \frac{m}{n}}$

Proof: According to Lemma 4.4:

$$p = (1 - e^{-\frac{nk}{m}})^k$$

So, the false positive rate p can be considered as a function of the number of hash functions k:

$$p = f(k) = (1 - e^{-\frac{nk}{m}})^k$$

Then,

$$f(k) = (1 - b^{-k})^k \; with \; b = e^{-\frac{n}{m}} \tag{4.19}$$

Using the logarithm operator on both sides of Equation 4.19, we get:

$$ln[f(k)] = k \cdot ln(1 - b^{-k}) \tag{4.20}$$

Then calculating the derivation of both sides of Equation 4.20, we get:

$$\frac{1}{f(x)} \cdot f'(x) = ln(1 - b^{-k}) + k \cdot \frac{1}{1 - b^{-k}} \cdot (-1) \cdot (-b^{-k}) \cdot ln(b) = ln(1 - b^{-k}) + k \cdot \frac{b^{-k} \cdot ln(b)}{1 - b^{-k}} \tag{4.21}$$

When Equation 4.21 equals to 0, Equation 4.20 reaches its minimum value. At this time we get:

$$ln(1 - b^{-k}) + k \cdot \frac{b^{-k} \cdot ln(b)}{1 - b^{-k}} = 0 \tag{4.22}$$

$$\Rightarrow (1 - b^{-k}) \cdot ln(1 - b^{-k}) = b^{-k} \cdot ln(b^{-k}) \tag{4.23}$$

According to the symmetrical characteristic of Equation 4.23, we get:

$$1 - b^{-k} = b^{-k} \tag{4.24}$$

$$\Rightarrow e^{-\frac{kn}{m}} = \frac{1}{2} \tag{4.25}$$

$$\Rightarrow k = ln2 \cdot \frac{m}{n} \tag{4.26}$$

And since:

$$p = f(k) = (1 - \frac{1}{2})^k = (\frac{1}{2})^k = 2^{ln2 \cdot \frac{m}{n}} \tag{4.27}$$

□

According to Lemma 4.6, we have the following Theorem:

Theorem 4.3 *Given the false positive rate p, and the maximum number need to be inserted in the Bloom Filter n, The number of bits of this Bloom Filter m should be:*

$$m = -\frac{n \cdot lnp}{(ln2)^2} \tag{4.28}$$

And the number of hash functions k should be:

$$k = ln2 \cdot \frac{m}{n} = log_2 \frac{1}{p} \tag{4.29}$$

In our case, we use the maximum number of elements to be inserted in the Bloom Filter and its false positive rate to construct a Bloom Filter. In all experiences, we set the false positive rate to 0.01. The maximum number of triples that need to be inserted in the Bloom Filter is set to the number of triples of each generation.

4.5.3 Complexity

The cost for the join method proposed in this paper contains the 3 following parts:

1. The insertion and expiration cost for building a Bloom Filter
2. The cost for transmitting a Bloom Filter
3. The cost for probing a Bloom Filter

Suppose there are w triples on a node. Suppose the number of generations contained by a Sliding Window is c. In the worst case we need to insert all w elements into the Bloom Filter. Then the cost for inserting a generation should be

$$O(\frac{wk}{c}) \tag{4.30}$$

And the cost for transmission of a generation should be $O(m)$, where m is the number of bits in a Bloom Filter. When p gets the optimal value, it becomes:

$$m = -\frac{w \cdot ln(p)}{(ln(2))^2} bits. \tag{4.31}$$

Suppose the data transmission rate is μ Mbps, then the cost for transmitting a Bloom Filter is:

$$\tau = \frac{m}{\mu} \tag{4.32}$$

$$= -\frac{w \cdot ln(p)}{\mu \cdot (ln(2))^2} \tag{4.33}$$

$$= O(w) \tag{4.34}$$

In the worst case we need to probe all the elements from the current Sliding Window, the probe cost is:

$$s \cdot k \tag{4.35}$$

The complexity of searching an element in a Bloom Filter is $O(k)$, where k is the number of hash functions. Usually this number is a very small constant, so this cost is $O(1)$. Then the probing cost of a Bloom Filter depends on the number of elements in a Sliding Window, since each time we need to use the whole elements in the Sliding Window to probe the Bloom Filter of the current Sliding Window. So in order to reduce the probing cost, we need to use small generations. However smaller generations cause frequent requests for Bloom Filters, which will increase the overall communication cost.

So in a real application, the performance depends on fine tuning the size of the Sliding Window and the number of generations.

4.6 Implementations

The whole system contains three main parts : **Query Planner**, **Executor** and the Topology (please refer to the introduction about Topology in Section 2.1.2.5) on Storm.

Query Planner

The **Query Planner** analyzes the queries, returns the join vertices and their type, and the results of the Query Topological Sort, as shown in Fig. 4.20.

The main algorithms used for Query Planner are: **Find Join Vertices** in Algo. 1; **Define Join Types** in Algo. 2 and **Query Topological Sort** in Algo. 3.

In Algo. 1, we go through a query graph, and add the variable nodes whose degree is bigger than 1 into a list. This list is used to store the Join Vertices.

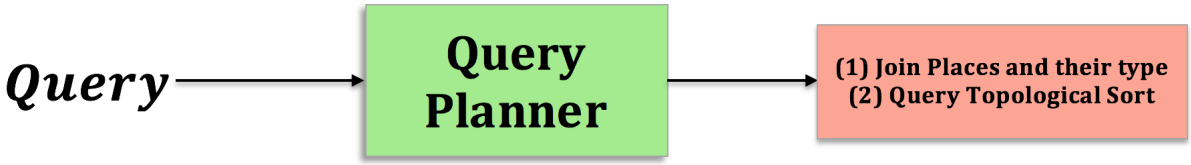


Fig. 4.20 Query Planner

Algorithm 1 Algorithm for finding the Join Vertices

Require:
A SPARQL query Q, with vertices set V ;
Output: A List of Join Vertices *JVList*;

```
for each v in V do
    if v is a variable and d(v) > 1 then
        JVList.add(v);
    end if
end for
return JVList;
```

Algorithm 2 Algorithm for computing the category of the Join Vertex

Require:
A join vertex v, with a set P of connected edges;
3 categories I, II and III of join vertex as defined in Section 4.3.3;
Output: the category of this join vertex v;

```
List 1VTriple = new List();
List 2VTriple = new List();
for each p in P do
    if p has one variable then
        1VTriple.add(p);
    else if p has two variables then
        2VTriple.add(p);
    end if
end for
if 2VTriple.size()==0 then
    return v ∈ Category I;
else if 2VTriple.size()==1 then
    return v ∈ Category II;
else
    return v ∈ Category III;
end if
```

Algo. 2 is based on the the rules presented in Section 4.3.

There are many methods to do a Topological Sort for a Graph. In our system, we choose to use a Depth First Search method. In this method, we use a queue to mark the vertices that we are going to visit, and a list to store the final result. We first add the start Join Place ?Q into the queue. Then, while the queue is not empty, we pop the top element in the queue, and we sort its adjacent nodes starting with fixed vertices and ending with variable vertices. Then we go through its adjacent list and add each node to the queue. Each time we add a node we reduce its degree by 1, and when it reaches 0, we add this node to the result list.

Algorithm 3 Query Topological Sort

Require:
A Query Graph G;
A Multiple-Variable Join Place ?Q to start;
Output: The dependency order of the query graph

```
1:  Queue queue = new Queue();
2:  List result = new List();
3:  queue.enqueue(?Q);
4:  while !queue.isEmpty() do
5:     Vertex v = queue.dequeue();
6:     Collections.sort(v.adjacent, Comparator(Fix > Variable));
7:     for each Vertex adj in v.adjacent() do
8:        queue.enqueue(adj);
9:        if - -adj.indegree==0 then
10:          result.add(adj);
11:       end if
12:    end for
13: end while
14: return result;
```

Executor

The **Executor** is responsible for producing the Topology to be executed on a Storm Cluster. The work flow of an Executor is shown in Fig. 4.21.

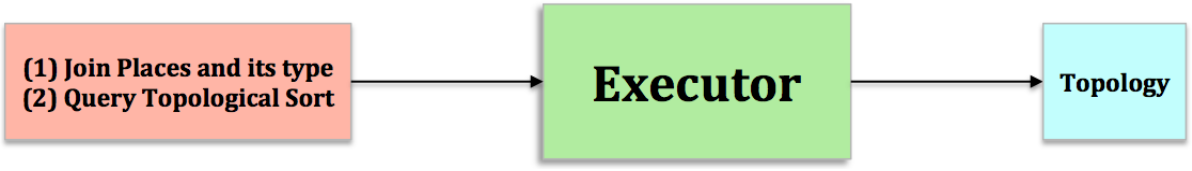


Fig. 4.21 Executor Work Flow

A Storm Topology contains two different components: **Spout** and **Bolt**. The **Spouts** are used to distribute resources. The **Bolts** are used to process each data. We can define different kinds of **Bolt**. In our case, a **Bolt** task is an instance of a **Builder** or/and a **Prober**. We also need to define the input of the **Bolts**, which in our case is the sending and receiving orders of Bloom Filters.

Executor needs to use the output of Query Planner, and define the **Builders** and the **Probers**, and the sending and receiving orders as described in Section 4.3.3. The UML Graph for an Executor is shown in Fig. 4.22.

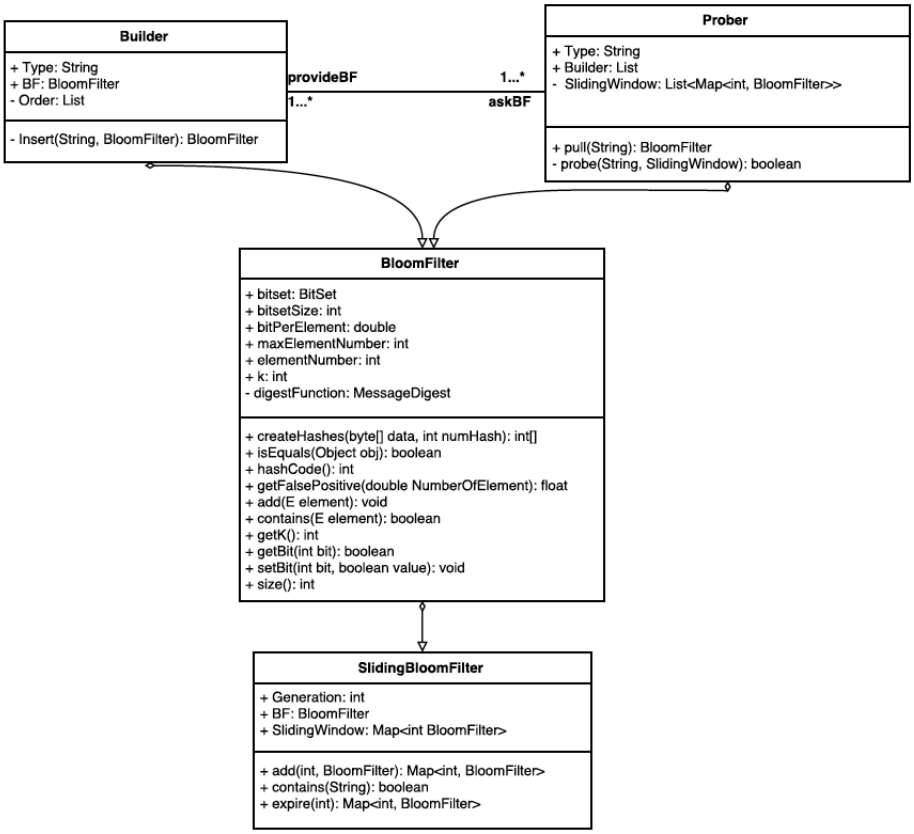


Fig. 4.22 Executor UML Graph

The most important part on an Executor is the Sliding Bloom Filter. We create a Bloom Filter for each generation, and store these Bloom Filters in a temporary hash table at the **Prober** side. We use an integer to indicate the generation index as the hash key associated to the Bloom Filter value. We periodically update the hash table in order to withdraw the old generation of Bloom Filters.

4.7 Experiment Result

In this section, we evaluate the performance and accuracy of the system.

4.7.1 Experiment Setup

We evaluate the system on Grid 5000 [6], with 11 nodes. Among them, one is reserved to be Nimbus, and the rest 10 nodes are used for computing. The setting of the cluster is shown as below.

We use 11 nodes (2 CPUs Intel Xeon E5-2660 v2, 10 cores/CPU, 126GB RAM, 5x558GB HDD, 10Gbps ethernet). The Storm version is 1.0, and we only use one slot on each machine. The Apache Jena[13] API is used for reading triples.

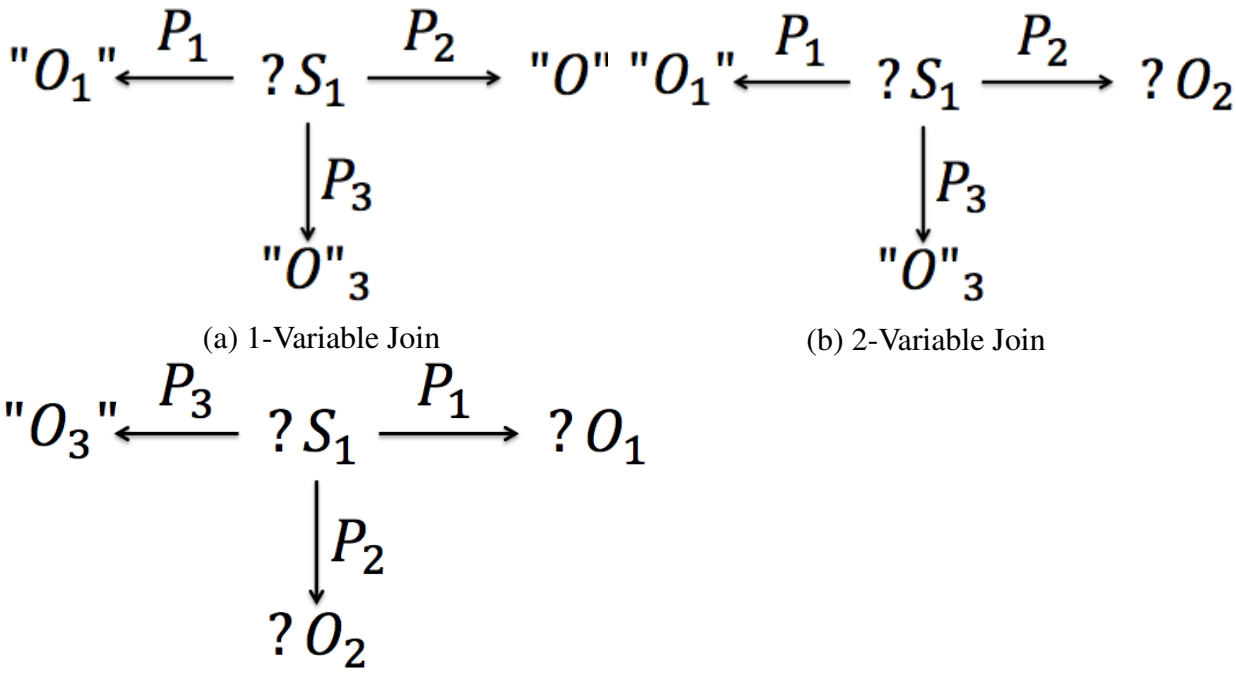
Apache Jena is an open source Semantic Web framework for Java. It provides an API to extract data from, and write to, RDF graphs. The graphs are represented as an abstract "model". A model can be sourced with data from files, databases, URLs or a combination of these. A Model can also be queried through SPARQL. We use both synthetic and real-world benchmarks. The real benchmark we used in the evaluation is LUBM[14]. LUBM is a widely used benchmark for Semantic Web. It consists of a university domain ontology. It is customizable and repeatable Data. It contains a set of test queries and several performance metrics. The original LUBM generator generates triples class by class, which means for example, it will generate first all triples for "Professor", then for "GraduateStudent", etc. In our experiment, we modified the original LUBM generator in order to have a random order of triples directly generated in Spouts.

4.7.2 Evaluation about the 3 basic types of join Using Synthetic data

We first use synthetic data to test the three basic types of Join shown in Fig. 4.23.

The RDF triples generated in Spouts are distributed to the nodes according to their predicate. Two sub-sets of data are generated. The first sub-set contains only matching results whereas the second one contains only non matching ones. For generating the matching sub-set, we choose a common range of subjects for each predicate. Since each predicate has this same set of subjects, this set can be considered as the results of the join. We then generate a different range of subjects for each predicate, making sure all these subjects are different from the ones in the previous set. They form the non matching set. We collect the final result set of the Join, and compare it with the generated matching sub-set, in order to calculate the precision and recall as we defined in Section 3.5 of the system. Besides,

[6]https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home



(a) 1-Variable Join

(b) 2-Variable Join

(c) Multiple-Variable Join

Fig. 4.23 3 Basic Types of Join

we record the generation time of each result triple, and the emitting time of the result, to calculate the **execution latency**.

Another metric we have used is the **process latency**, which is defined as the difference between the time a triple begins to be processed and the time it is generated.

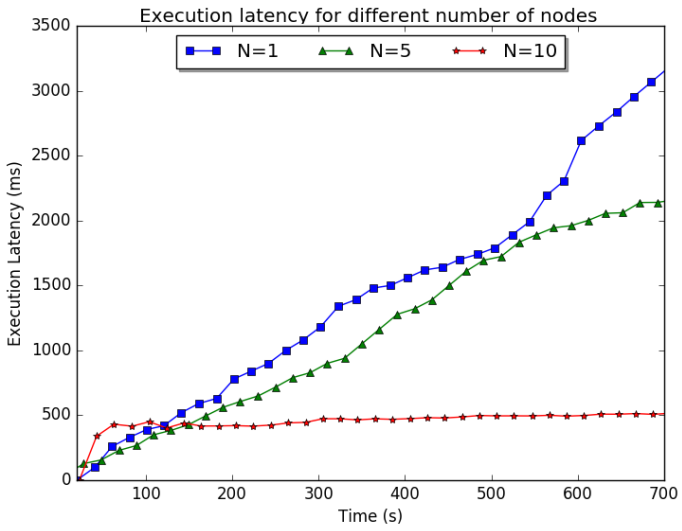
We use Count Based Sliding Window for all the experiences.

4.7.2.1 The evaluation of parallel performance

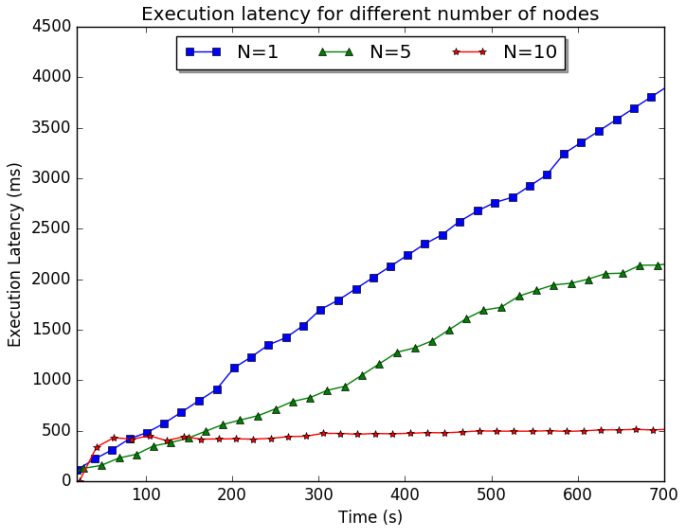
In this evaluation we use synthetic data to evaluate the 3 basic types of join. At this step we want to evaluate the parallel efficiency of our method, so we use different number of nodes to process each type of join, and we record the number of triples, the execution latency and process latency.

Fig. 4.24(a), 4.24(b) and 4.24(c) show the execution latency of each sliding window for each kind of join with a varying number of nodes.

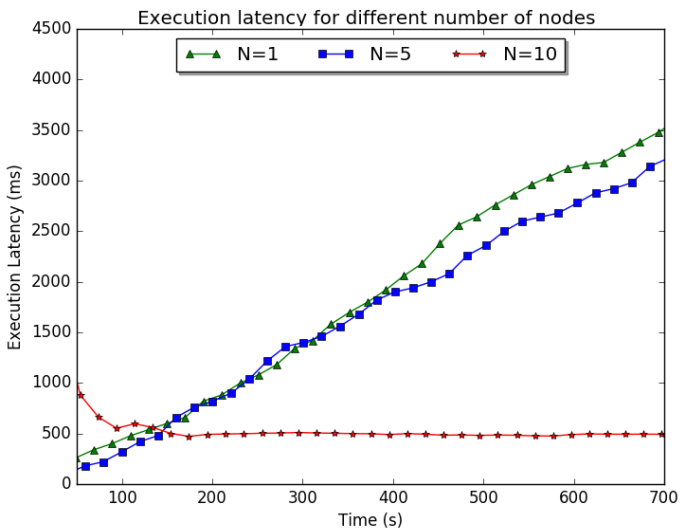
From these 3 figures we can see that the execution latency decreases while the number of nodes increases. When the number of machine for executing is set to 10, the execution latency does not change with time, which means that we have achieved a good balance of parallelism, each sub-query is executed separately on different nodes in parallel. At this time, the execution latency is approximately equal to the execution time of each Sliding Window.



(a) The Execution Latency of 1-Variable Join



(b) The Execution Latency of 2-Variable Join



(c) The Execution Latency of Multiple-Variable Join

Fig. 4.24 Basic Types of Join

Another important metric for evaluating the algorithms is the accuracy, which is presented by **recall** and **precision** the same as we defined in Section 3.5. In our evaluation, we got all the elements from the results sub-set we generated in the final result sets. So for us the accuracy is 100%.

The reason can be explained as follows: as we described in Section 3.3.2, we can use two parameters to determine all the parameters for constructing a Bloom Filter. In our implementation, each time we choose to use the maximum number of elements needed to be inserted in the Bloom Filter and the false positive probability to determine all the parameters for constructing the Bloom Filter. We set the false positive probability to a low value (0.01) for every Bloom Filter. Secondly, we choose to use the number of triples contained by each generation as the maximum number needed to be inserted in the Bloom Filter. But, actually, we only need to insert the matching results in the Bloom Filter, which is much smaller than the generation size. So the real false positive probability of our Bloom Filters should be much less than 0.01 in practice. And that is the reason why we got 100% accuracy for the results.

4.7.2.2 Impact of number of generations

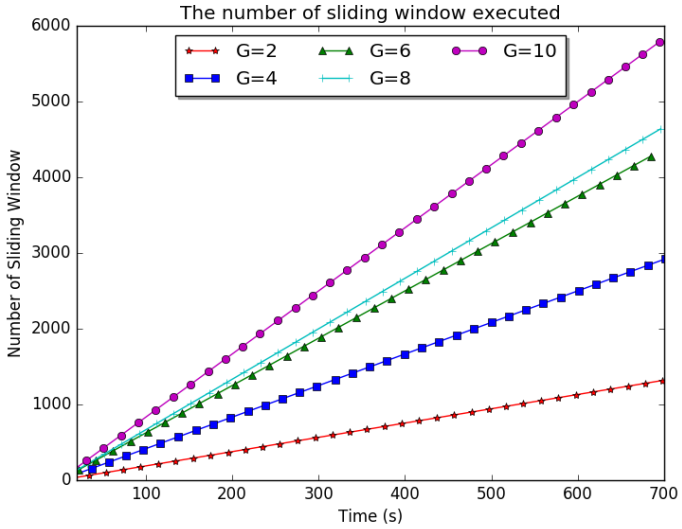
Here we use Storm to deal with data streams. There are two parameters which strongly affect the performance of a streaming process. The first one is the Sliding Window Size, the second one is the number of generations.

We first fix the Sliding Window size to 400 triples, and we vary the number of generations G to 2, 4, 6, 8 and 10. The number of generations in a Sliding Window affects the frequency of execution and transformation of data. We record the number of Sliding Windows executed, the Execution Latency, the Processing Latency and the Data Transferred through network for each type of Join.

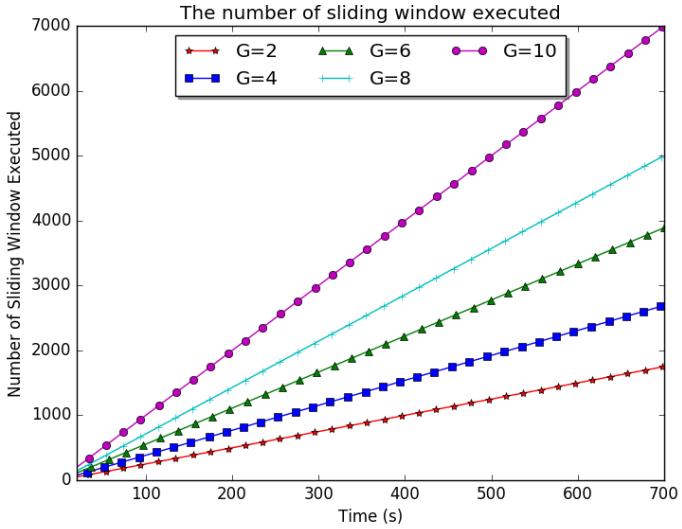
Fig. 4.25(a), Fig. 4.25(b) and Fig. 4.25(c) show the number of Sliding Window executed over time. When we increase the number of generations, the system can execute more Sliding Windows. Because each generation has less triples to process, and we can finish each Sliding Window faster.

Correspondingly, the execution latency decreases while the number of generations increases, as shown in Fig. 4.26(a), Fig. 4.26(b) and Fig. 4.26(c). There is a peak for 2-Variable and Multiple-Variable join at the very beginning, because the system has not reached the balance yet.

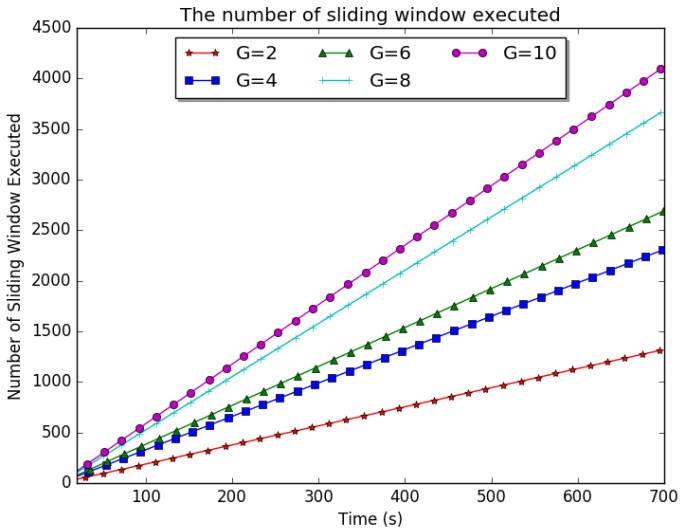
The process latency behaviors is opposite. When the generation size is large, the elements do not need to wait until the right generation for execution. Conversely, when the generation size is small, the elements need to wait until their generation for processing. Anyway, the



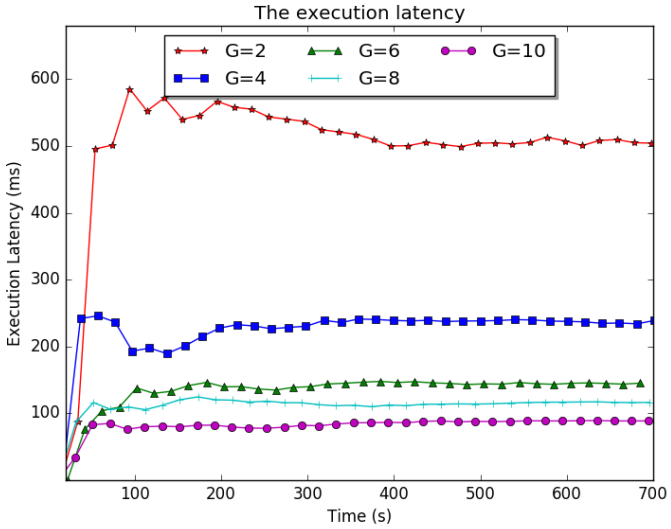
(a) Number of Sliding Window Executed for 1-Variable Join



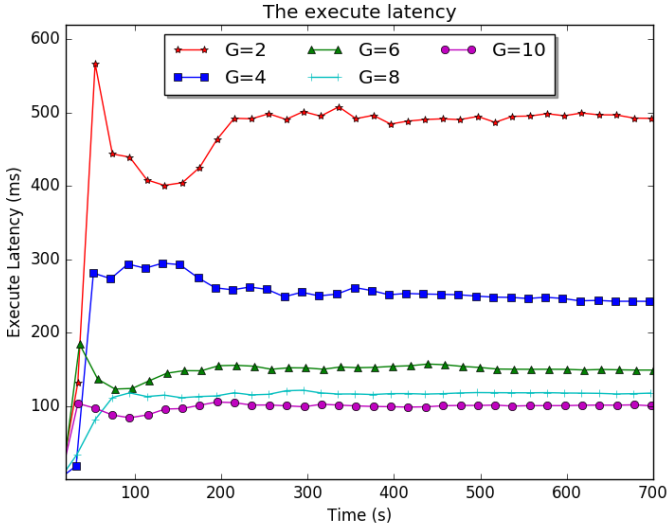
(b) Number of Sliding Window Executed for 2-Variable Join



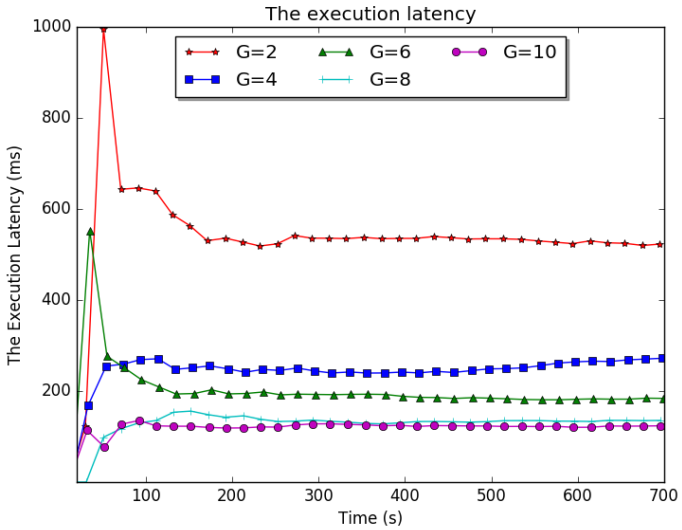
(c) Number of Sliding Window Executed for Multiple-Variable Join



(a) Execution Latency for 1-Variable Join



(b) Execution Latency for 2-Variable Join



(c) Execution Latency for M-Variable Join

Fig. 4.26 Execution Latency (SW = 400)

process latency is very small compared to the execution latency, as shown in Fig. 4.27(a), Fig. 4.27(b) and Fig. 4.27(c).

Another important metric to evaluate our method is the communication overhead. Because, the Bloom Filter size depends on the number of elements to be inserted into the Bloom Filter once the false positive rate is chosen. In this benchmark, we only consider the data transmitted among the Bolts, and we ignore the triples transmitted from the Spouts to the Bolts. The only data transmitted through the network is the Bloom Filters. So we record the size of Bloom Filters with different number of generations. In 2-Variable Join, the amount of data transmitted is similar when G=4 and G=6 (the same for 1-Variable Join when G=6 and G=8). We can consider that at this time the system is balanced from the data transmission point of view. The results for 1-variable join, 2-variable join and multiple-variable join are shown in Fig. 4.28(a), Fig. 4.28(b) and Fig. 4.28(c) respectively.

4.7.2.3 Impact of number of Sliding Window Size

In this part, we evaluate the impact of the Sliding Window size on performance. This time, we fix the number of generations at 4, and we vary the Sliding Window size at 200, 400, 600 and 800.

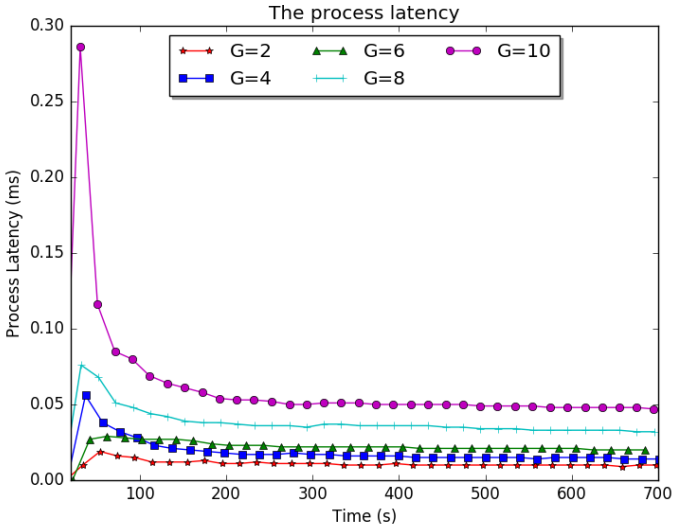
We first evaluate the number of Sliding Window executed over time. Fig. 4.29(a), Fig. 4.29(b) and Fig. 4.29(c) show the results for the three different types of joins respectively. We can see from these figures that, as we increase the size of a Sliding Window, the number of Sliding Windows executed decreases, since the time for executing one Sliding Window increases.

We then evaluate the execution latency. The results are shown in Fig. 4.30(a), Fig. 4.30(b) and Fig. 4.30(c). The execution latency increases with the size of the Sliding Window, because the execution time for one Sliding Window increases with its size.

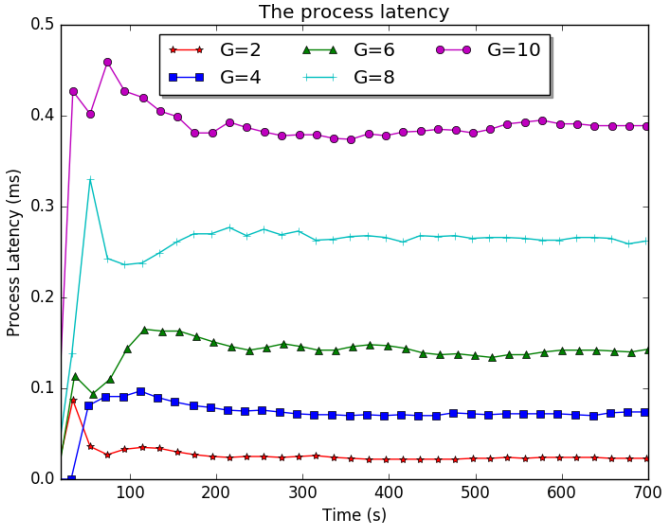
The process latency has a different behavior. The results are shown in Fig. 4.31(a), Fig. 4.31(b) and Fig. 4.31(c). They do not have any regular patterns. But since the process latency is not very high (less than 0.5 ms), they are not considered as a critical factor which affects the performance.

The last thing we want to evaluate is the amount of data transferred through the network. As shown in Fig. 4.32(a), Fig. 4.32(b) and Fig. 4.32(c), the amount of data transferred through the network grows while we increase the Sliding Window size. But when the Sliding Window size is 800, the data becomes inaccurate, because the execution latency is too long, and the Sliding Window can not be finished in time.

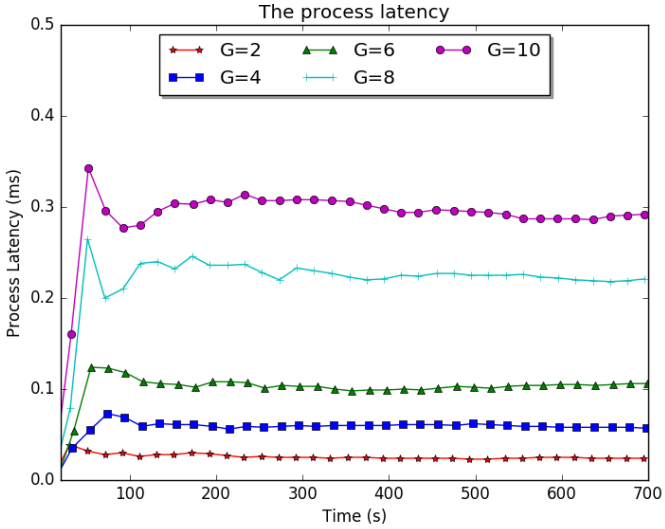
Generally, the execution latency for 1-Variable Join is smaller than that of 2-Variable Join, and Multiple-Variable Join has the highest latency. The number of matching triple patterns is



(a) Process Latency for 1-Variable Join

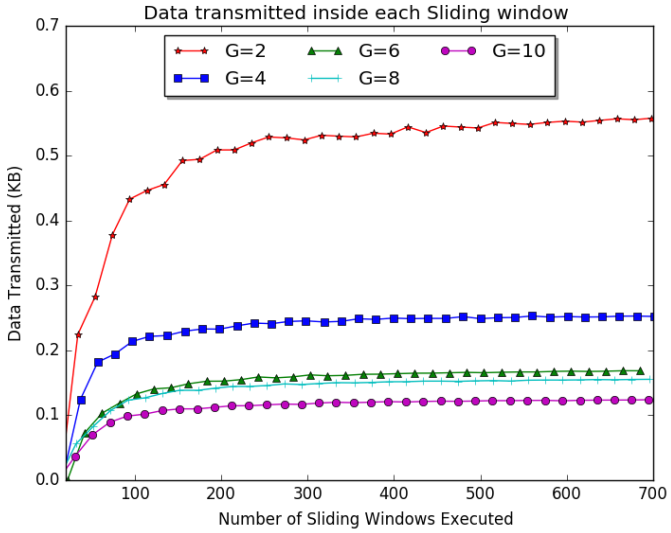


(b) Process Latency for 2-Variable Join

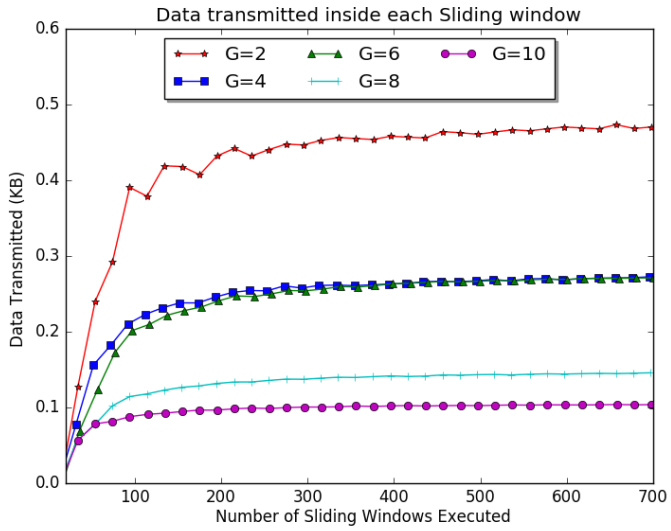


(c) Process Latency for M-Variable Join

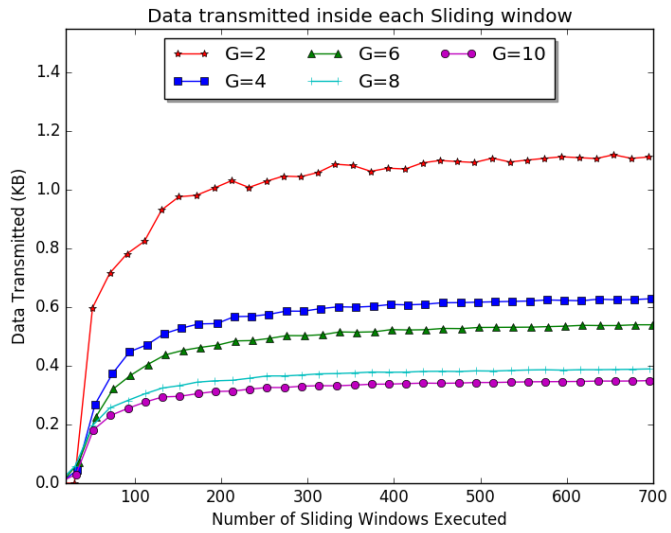
Fig. 4.27 Process Latency (SW = 400)



(a) Communication Overhead for 1-V Join

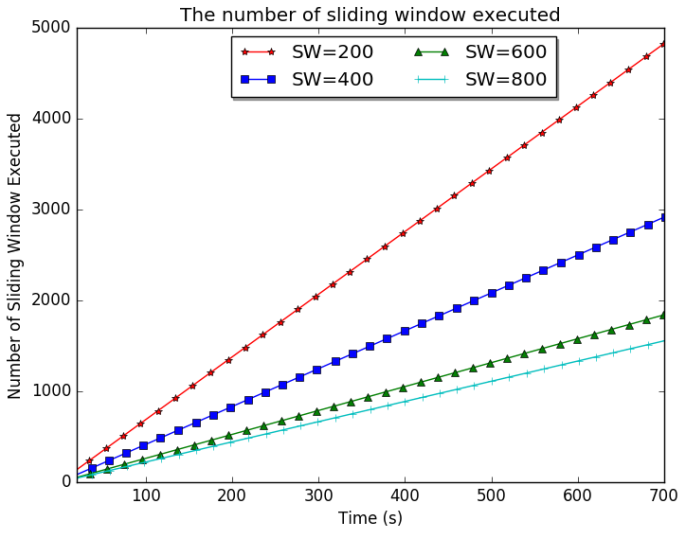


(b) Communication Overhead for 2-V Join

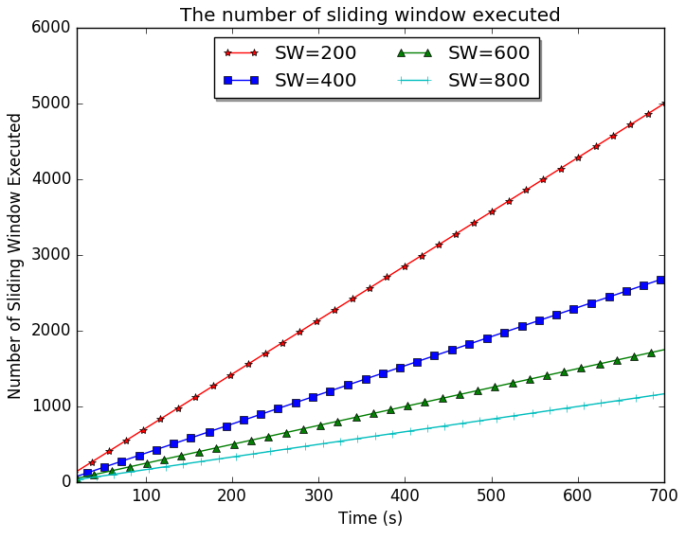


(c) Communication Overhead for Multiple-Variable Join

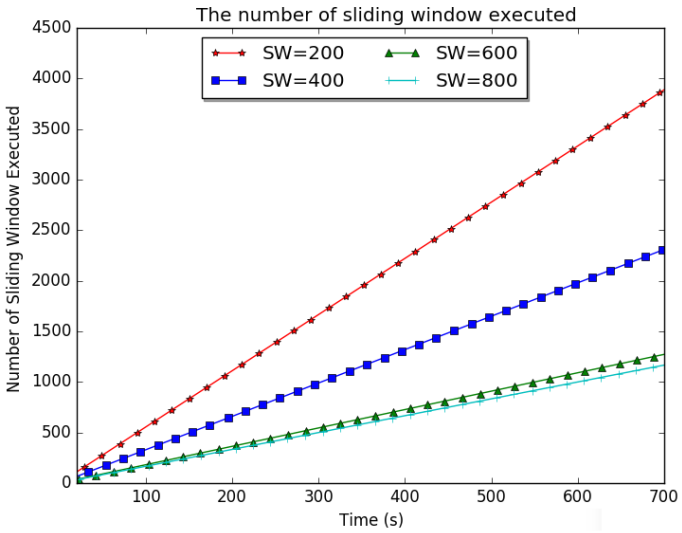
Fig. 4.28 Communication Overhead (SW = 400)



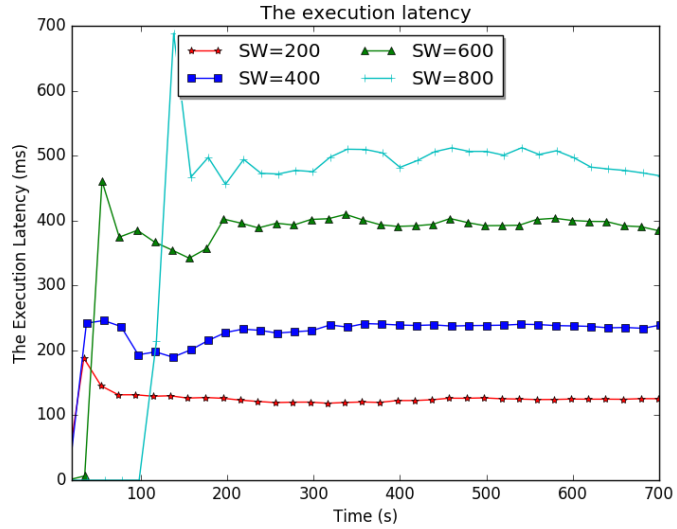
(a) Number of Sliding Window Executed for 1-Variable Join



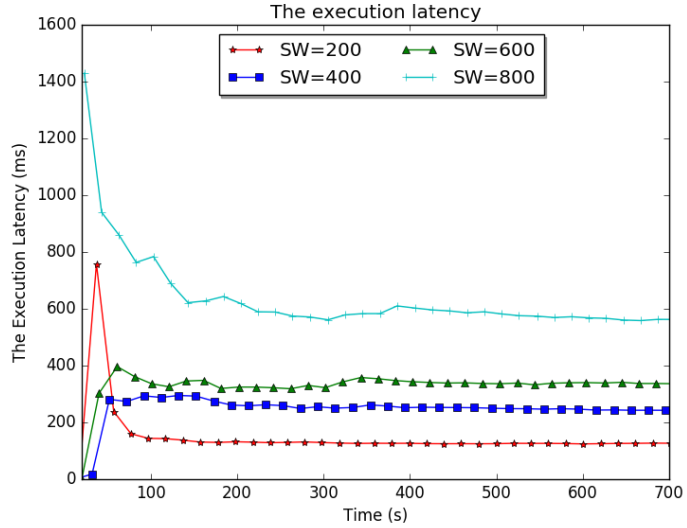
(b) Number of Sliding Window Executed for 2-Variable Join



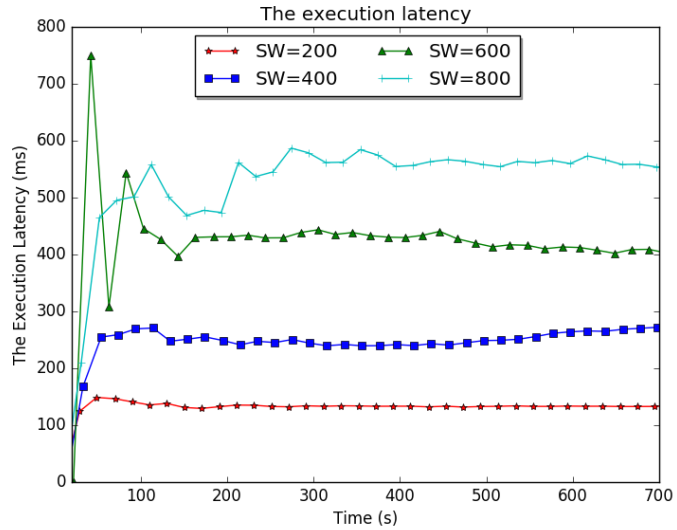
(c) Number of Sliding Window Executed for Multiple-Variable Join



(a) Execution Latency for 1-Variable Join

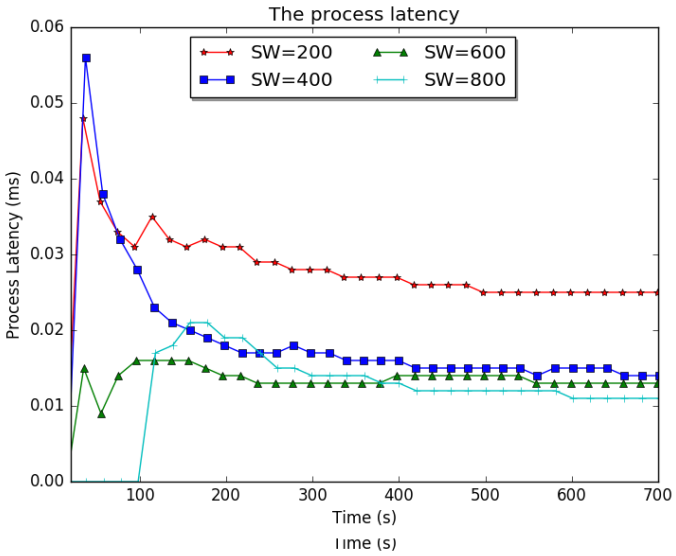


(b) Execution Latency for 2-Variable Join

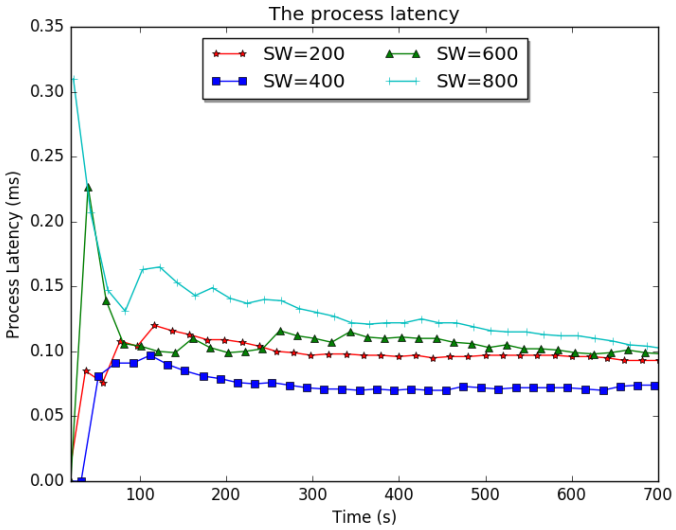


(c) Execution Latency for M-Variable Join

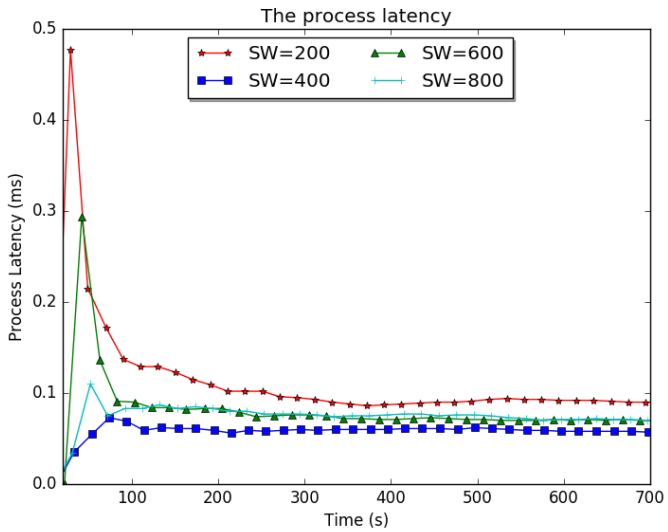
Fig. 4.30 Execution Latency (G = 4)



(a) Process Latency for 1-Variable Join

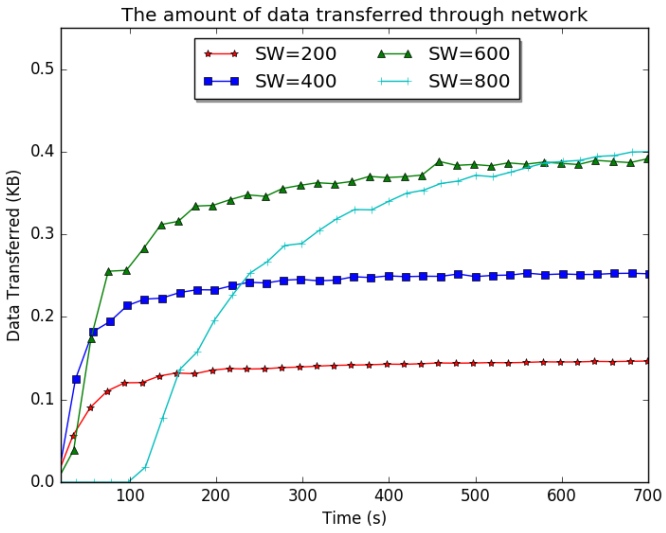


(b) Process Latency for 2-Variable Join

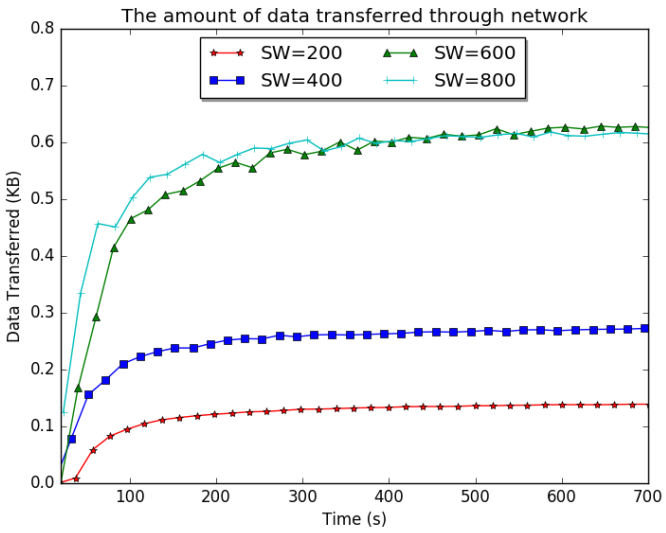


(c) Process Latency for M-Variable Join

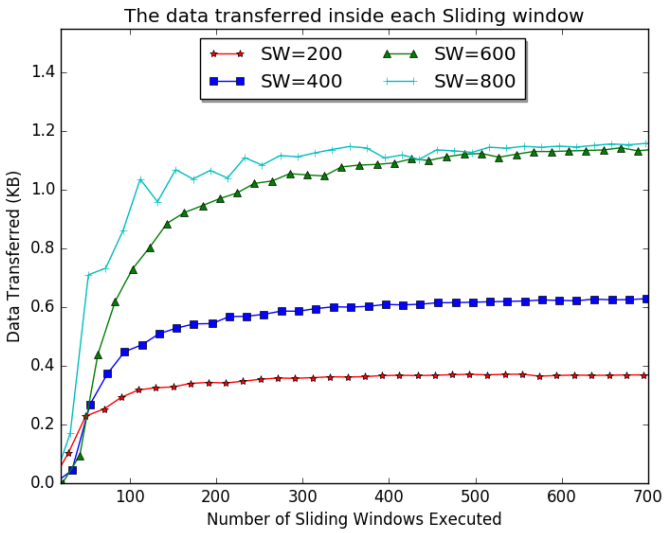
Fig. 4.31 Process Latency (G = 4)



(a) Communication Overhead for 1-V Join



(b) Communication Overhead for 2-V Join



(c) Communication Overhead for Multiple-Variable Join

Fig. 4.32 Communication Overhead (G = 4)

smaller for 1-Variable join than for 2-Variable Join and Multiple-Variable Join, resulting in a shorter probe list (defined in Section 4.4), and a faster execution. But this general result is mitigated by the distribution of triples.

4.7.3 LUBM Benchmark

In this Section, we use the LUBM Benchmarks to evaluate our framework.

LUBM has 14 queries, and we have tested query No.1, query No.3, and query No.4. The remaining queries either didn't have a join or were too complex to implement due to time constraints. LUBM is intended to evaluate the performance of the repositories with respect to extensional queries over a large data set that commits to a single realistic ontology. It consists of a university domain. It is customizable and repeatable.

Query1 bears large input and high selectivity. It only queries one class and one property and does not assume any hierarchy information or inference. Query1 is shown in Fig. 4.33.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE{
  ?X  rdf:type ub:GraduateStudent .
  ?X   ub:takesCourse
       http://www.Department0.University0.edu/GraduateCourse0}
```

Fig. 4.33 Query1 in LUBM

Query3 is similar to Query 1 but class Publication has a wide hierarchy. Query3 is shown in Fig. 4.34.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X
WHERE{
  ?X  rdf:type ub:Publication .
  ?X   ub:publicationAuthor
       http://www.Department0.University0.edu/AssistantProfessor0}
```

Fig. 4.34 Query3 in LUBM

Query4 has small input and high selectivity. It assumes subClassOf relationship between Professor and its subclasses. Class Professor has a wide hierarchy. Another feature is that it queries about multiple properties of a single class. Query4 is shown in Fig. 4.35.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?X, ?Y1, ?Y2, ?Y3
WHERE{
  ?X  rdf:type ub:Professor .
  ?X  ub:worksFor <http://www.Department0.University0.edu> .
  ?X  ub:name ?Y1 .
  ?X  ub:emailAddress ?Y2 .
  ?X  ub:telephone ?Y3}
```

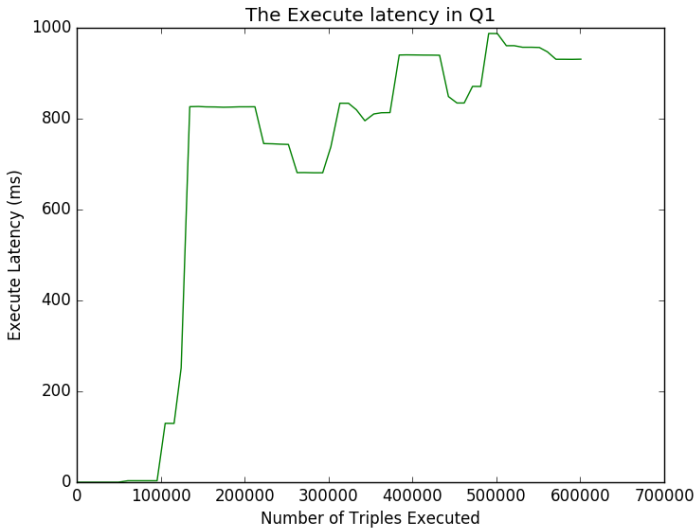
Fig. 4.35 Query4 in LUBM

We generate 6 Million triples for each benchmark. But these triples are filtered: only the triples used by the Bolts (i.e. with a matching predicate) are emitted by the Spout. Hence we only record the number of triples really processed by the Bolts. But we need to remind the readers, that the number of triples processed (including the triples already filtered before arriving at the Bolts) by the system is 6 Million for each query. For this part of experiences, we set the sliding window size to 600, and the number of generations to 4.

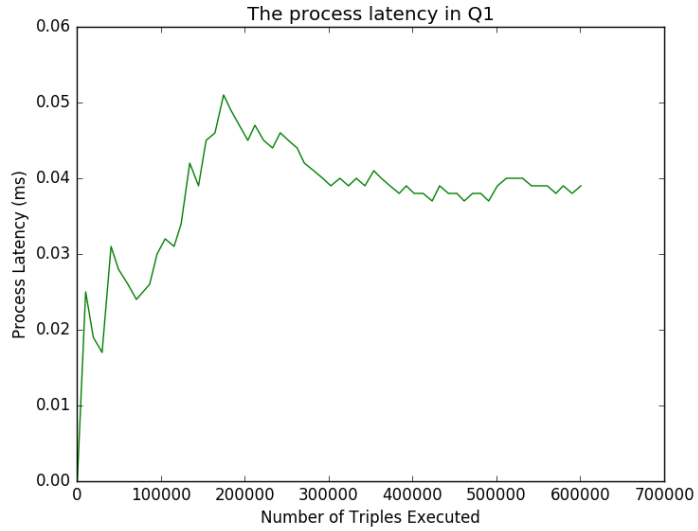
Fig. 4.36(a) and Fig. 4.36(b) show the execution latency and the process latency of query No.1. The process latency begins to increase at the beginning, and it fluctuates up and down around 0.04 ms when the process is stable. The execution latency increases sharply at the beginning, because it waited for the triples to be filtered by the Spouts. Then, when it gets enough triples for one generation, the execution latency starts to increase slowly. At this time, the execution latency can be considered as the execution time for each generation. It does not show the same trends as in the synthetic benchmarks, because the triples generated by the Spouts have first been filtered by the Spouts, and only the useful triples (the triples which match with the predicate rdf:type or ub:takesCourse) have been sent to the Bolts. But the number which really affects the processing time is the number of triples which are really processed by the Bolts, and this number is not predictable.

Fig. 4.37(a) and Fig. 4.37(b) show the execution latency and the process latency of query No.3 respectively. The process latency increases sharply at the beginning for Q3. This is because the Bolts are waiting for the triples filtered by the Spouts to be sent. The execution latency does not follow any pattern, because the number of triples executed by the Bolts can not be predicted.

Fig. 4.38(a) and Fig. 4.38(b) show the execution latency and the process latency of query No.4 respectively. The process latency of Q4 became stable at 0.28 ms after a sharp increase at the begin. The execution latency of Q4 follows waves up and down at 1200 ms after the system reaching a steady state. Because Q4 is a Multiple-Variable join, the latency mainly comes from the execution time of the triple patterns which are both a Prober and a Builder.



(a) Execution Latency for Q1

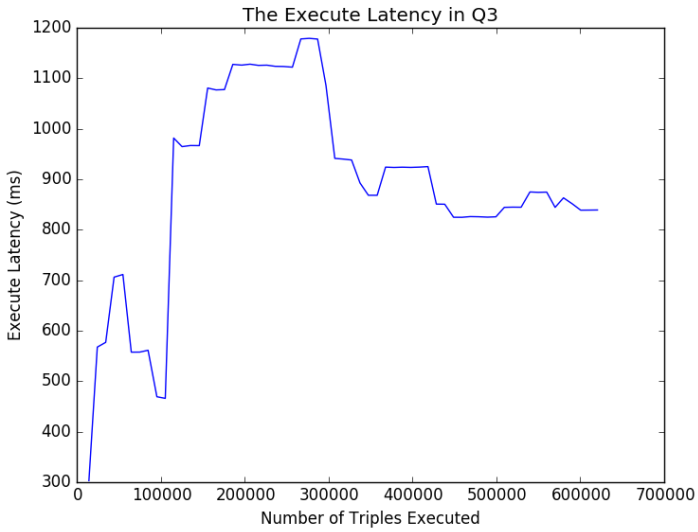


(b) Process Latency for Q1

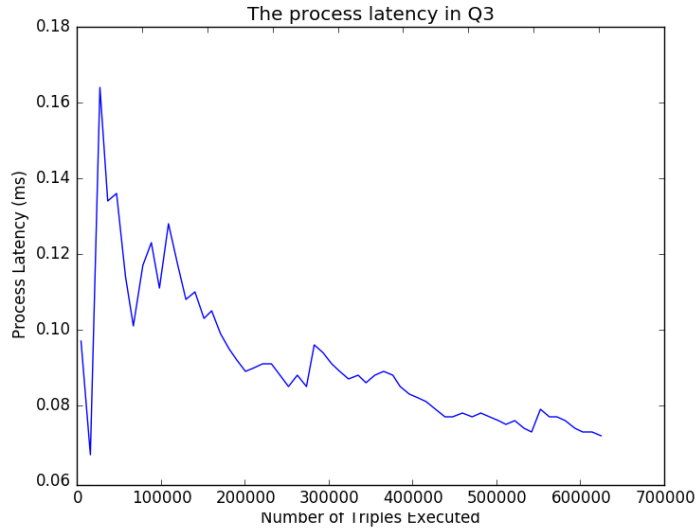
Fig. 4.36 Evaluation for Q1

Besides, it has more different predicates, which means that more triples generated by the Spouts have been executed by the Bolts.

Fig. 4.39(a), Fig. 4.39(b) and Fig. 4.39(c) show the data transferred through network for each query. The amount of data transmitted in Q1 and Q3 is almost the same, because they are both 1-Variable Join formed by 2 triple patterns. They have the same Sliding Window size and number of generations, leading to the same size of Bloom Filter in each generation. Correspondingly, Q4 has approximately 5 times (comparing to Q1 and Q3) the amount of data



(a) Execution Latency for Q3



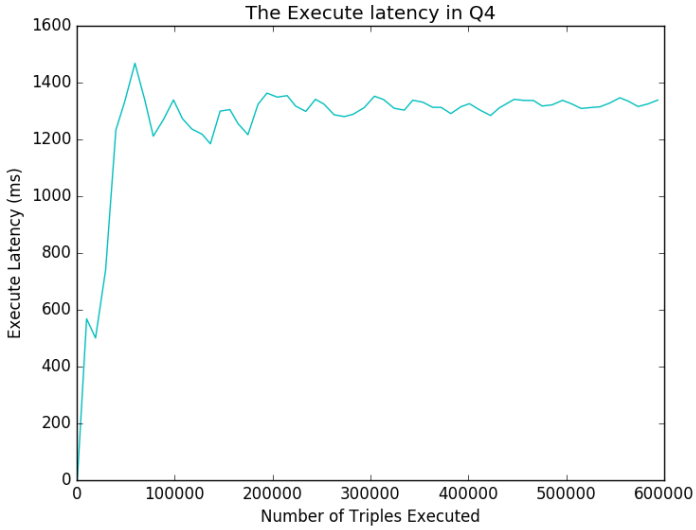
(b) Process Latency for Q3

Fig. 4.37 Evaluation for Q3

transmitted through network inside each Sliding Window, because it is a Multiple-Variable Join, and it has 5 Bloom Filters to be transmitted through the network for each generation.

4.8 Conclusion

In this Chapter we have introduced our methods for processing RDF stream joins in a parallel and distributed manner. We began with an introduction about the background, and

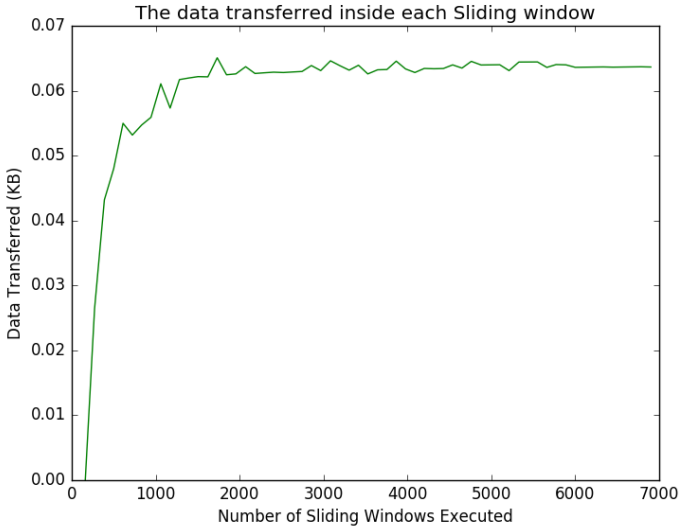


(a) Execution Latency for Q4

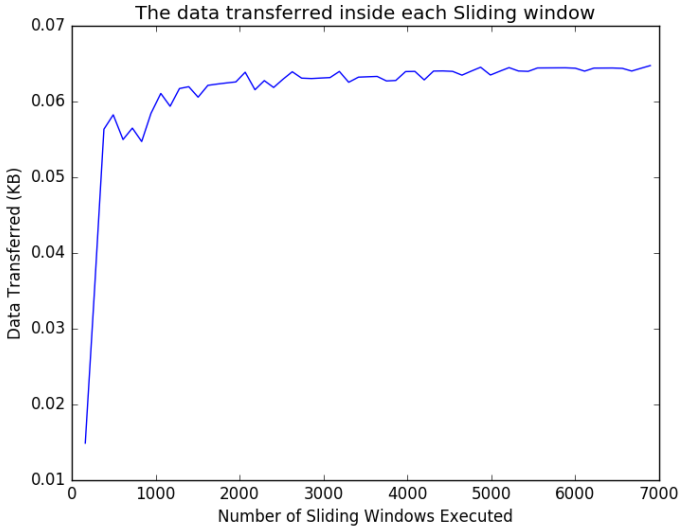
(b) Process Latency for Q4

Fig. 4.38 Evaluation for Q4

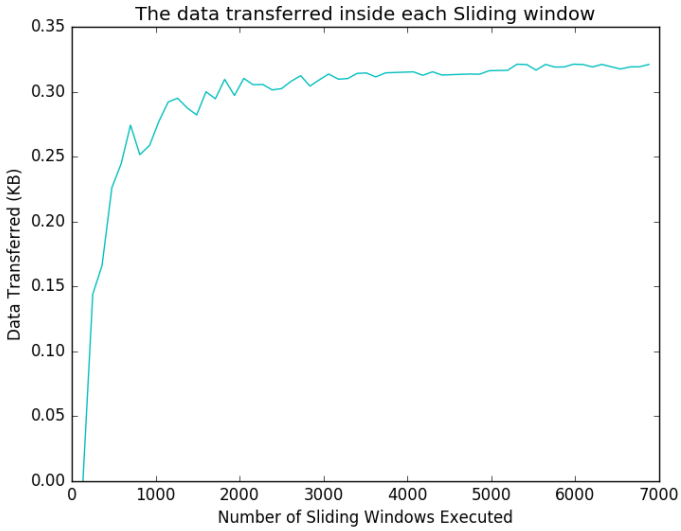
explained the motivation and the goal of our work. Then we showed some related works, and composed our approach to this state of the art. In Section 4.3 we explained our method for processing RDF joins in a parallel and distributed way. Generally, for processing a parallel and distributed RDF join, we need to address two problems: (1) partition RDF triples and distribute them to machines; (2) decompose the queries into sub-queries and assign them to the appropriate machines. We distribute the data according to their predicate. In our method, one predicate can be processed by multiple machines in order to avoid load



(a) Data Transferred for Q1



(b) Data Transferred for Q3



(c) Data Transferred for Q4

Fig. 4.39 Data Transferred

balancing problems. At the same time, one machine can also process multiple predicates, in the case there are not enough machines available. We decompose queries into triple patterns, and send them to the corresponding machines, holding the triples with the same predicate. We use Bloom Filters to minimize the communication among nodes. The joins are classified into 3 categories according to their shape, and we introduce 3 rules to join the intermediate results of each triple pattern. A Query Topological Sort method is proposed to determine the order of sending and receiving information. In Section 4.4, we extended the methods proposed in Section 4.3 to make them work for continuous RDF stream joins. We use a sliding window model and sliding window Bloom Filters. Each window “slides" in the unit of a generation. We then analyzed our methods in Section 4.5, focusing on Bloom Filters, dominating parameters, and complexity. The implementation issues are presented in Section 4.6, where the algorithm for finding the join vertices, the algorithm for judging the category of the join vertex, the algorithm for Query Topological Sort and the algorithm for sliding Bloom Filter are shown. In the end we evaluate our method in Section 4.7.

Our method does not need any tedious index which occupy more disk spaces. It does not need any complicated query plan which requires NP-hard computations neither. Besides, we do not directly transfer triples, but transfer Bloom Filters, thereby greatly reducing the amount of intermediate data transmitted. Although Bloom Filter might increase the probability of having false positive results, we got 100% of correct results in practice when we evaluated our methods. We avoid the strategies which depend on the analysis of the whole dataset to be able to process dynamic data streams. In brief, we proposed a both time and space efficient method to process the join operation on RDF streams in a parallel and distributed way.

Chapter 5

Conclusion and Future Work

The purpose of this thesis is to study parallel and continuous joins for data streams. We divide the joins into two different categories, Data Driven and Query Driven. We chose a classic use case for each type of join to study the technologies for processing them in a parallel and continuous manner. Then, we carefully and exhaustively analyzed and evaluated our methods both theoretically and experimentally. In this Chapter, we summarize our contributions and propose some research perspective.

5.1 Conclusion

5.1.1 Data Driven Join

In a Data Driven Join, the query never changes, but the format of data does. For example, in our use case of kNN (k nearest neighbor), the query is always finding k nearest neighbors. But the data could be GPS data in 2 dimensions, Twitter data in 77 dimensions, image data (SURF) in 128 dimensions, etc. A Data Parallel model can be applied for this problem. The difficulties for processing this kind of join in a parallel way is mainly caused by the data.

For a Data Parallel model, the data should first be pre-processed and then partitioned before it is dispatched to different nodes in order to achieve a better performance. The pre-processing step is mainly used for selecting the pivot points of data as the center of each partition or/and reducing the dimension of data. In this thesis, we studied 3 different techniques, proposed in the literature, for selecting the pivot data: Random Selection, Furthest Selection and K-Means Selection. Random Selection requires less calculation than the other methods, but the results it delivers are worse. We reviewed 2 methods for reducing the dimension of data in the pre-processing step. The first one is based on z-value which is a Space Filling Curve. This method projects high dimensional data into a one dimensional

space while maintaining the locality information with a high probability. Because of the loss of information due to the projection, we usually need to create some more shifts of the original data and compute the z-value multiple times. In practice, 2 shifts of data are sufficient to have a good accuracy. Another method for reducing the dimension is based on LSH (Locality Sensitive Hashing). This method uses a hash family to map high dimension data into lower dimension space. It makes use of the hashing collision to map closer points to the same bucket with a higher probability. In order to increase the accuracy, multiple hash families can be used. But in return, the processing time will also increase because of the duplication of data.

The partitioning step is used for dividing data. The number of partitions and the type of partitions dictates the amount of computation in the following steps. In this thesis, we first introduced a Random partitioning method, which does not apply any particular strategies. This method generates n^2 partitions, which results in too many data duplication and low performance. We then reviewed 2 advanced partitioning strategies in order to produce only n partitions. The first method is called Distance-Based Partitioning. Voronoi Diagrams are the main concept of this strategy. The idea is to group the most relevant points in each partition. The second method is called Size-Based Partitioning. It aims at creating roughly equal size partitions to gain a better load balance. The Size-Based method is based on a Sampling strategy. It is proven that the quantiles in the Gibbs Sampling [9] of the data set is an unbiased estimation of the quantiles for the whole data set.

We summarized two ideas about the main computation step. The first one generates the global data directly in only one processing phase. But the shortcoming is that each task should process a large amount of computation. When the size of each partition is large, this will be a major bottleneck. Another way of processing the main computation step is introduced where local results will first be generated. Then the local results will be merged in the second processing phase in order to produce the global results. Overall, each task has less computation than with the previous method but more computation phases are required.

In order to extend the parallel method to process data streams continuously, we applied a Sliding Window Model. We classified the scenario into 3 different types according to the dynamic of the data set. For the streaming join, we first proposed a basic method for each type of scenario based on a random partitioning strategy. In this basic method, the partitioning will apply the random partition introduced before to every generation. This generates n^2 blocks of data, which occupy more disk space and leads to a lot of unnecessary computation. We then proposed an advanced re-partitioning strategy based on the Naive Bayes method. In this method, the re-partitioning method partitions the new generation of

data without moving the already partitioned data. It computes the conditional probability of having the new data in each existing partition.

We also proposed a theoretical and experimental analysis of all these methods. We evaluated each method from a practical point of view, and summarized the advantage, shortcoming and typical use cases for each method.

Overall, we proposed a comprehensive strategy for processing Data Driven streaming join in a parallel and continuous manner. We offered a variety of methods, in order to adapt different scenarios such as different size of data set, different requirement about accuracy and execution time etc.

5.1.2 Query Driven Join

In a Query Driven Join, the format of data never changes, but the query does. We use RDF join in Semantic Web as the typical use case for this kind of process. In an RDF join, the format of data is always triples ($< subject, predicate, object >$). Conversely, the queries are written by the users and are not predictable. A Task Parallel model is typically used for solving these tasks. The difficulties for processing this kind of join in a parallel way is caused by queries. We proposed to decompose them into sub-queries before assigning them to different processing nodes. There are 3 important issues which are difficult to address when designing a decomposing strategy:

- Minimize the communication among sub-queries
- Minimize the usage of disk space
- Combine the result of the sub-queries

The decomposition of queries is proved to be an NP-hard problem in many research works. It is time consuming when the query graph is complicated. In order to minimize the communication among nodes some previous works just choose to either decompose the query or parallel the data, but not both. However these methods cannot deal with neither large data sets nor complex queries. In our method, we chose to decompose the queries into triple patterns, the smallest part of a query.

In order to minimize the communication, we used Bloom Filters as the footprint of the intermediate results to be joined. Bloom Filters are advanced data structure based on bit arrays for membership querying. In the decomposed query, some triple patterns (sub-queries) are responsible for building the Bloom Filters and are called **Builders**. The rest of triple patterns are used to probe the corresponding Bloom Filters, and are called **Probers**. The

main part of generating a query plan in our method is thus determining the **Probers** and the **Builders**, and deciding the order of communication between them.

We designed several rules to determine the **Probers** and **Builders**. We classify the joins into 3 different types according to their structure. 1-Variable join contains only one variable part which is the join vertex. A 1-Variable join is usually star-shaped which is difficult to decompose according to the Graph Theory. However, this is the simplest type of join to be processed in our rules. Since each triple pattern in an 1-Variable join is in the same position, any of them can be chosen as the **Prober** and the rest as the **Builders**. A 2-Variable join contains two variable parts. Since the Bloom Filters can only carry the information of the join vertex but not the other parts, the triple pattern which has two variable parts can only be chosen as the **Prober**. Thus the other triple patterns act as **Builders**. Multiple-Variable join is the most complex case. It has more than 2 triple patterns with two variable parts. Each of these triple patterns with two variable parts should act both as a **Prober** and as a **Builder**.

To decide the order of communication, we created a Query Topological Sort method where we give a higher priority to the constant vertex over variable ones.

After finishing the design of the parallel part, we extended our method to process data streams in a continuous way. We also used a Sliding Window model. We updated the Sliding Window periodically. We design the data re-evaluation strategy and data expiration strategy for our method. We introduced our strategies of using a Sliding Window Model on **Probers** which we call **Prober Windows**, and a Sliding Bloom Filter strategy on **Builders** which is called **Builder Filters**.

Before evaluating our method experimentally, we analyzed it theoretically. The theoretical analysis aimed at providing the appropriate parameters for the whole system, including the parameters for constructing the Bloom Filters, the number of generations and the size of the Sliding Windows.

In the end, we evaluated the whole system on Apache Storm using both synthetic data and real benchmarks. Despite the use of Bloom Filters, we obtained a very high accuracy in most of the experiments We were able to generate and process 6 million triples in 12 minutes. With 400 elements in each Sliding Window and 10 generations, the execution latency was around 100 ms for each kind of join on 10 machines, and the process latency was less than 0.5 ms.

In conclusion, compared to other distributed platforms, we do not directly transfer triples, but Bloom Filters, thereby greatly reducing the amount of data to be transmitted. Our method does not need any index, data replications or complex query optimization strategies, which perfectly fits the needs of a parallel stream processing platform.

5.2 Future Directions

In this part, we will give some outlooks about the future directions of research with respect to the work presented in this thesis.

5.2.1 Research Part

Some of the work presented in this thesis could be further expanded.

The first thing we need to do is to enrich the advanced re-partitioning strategy for kNN streaming join. We only gave a rough introduction about our method based on Naive Bayes. The theoretical and experimental analysis and proof have not been done due to time limitation. We will continue working on this part to provide a study about the accuracy using this partitioning method, the load balance issue, the computation overhead etc.

Another future research is further improve SPARQL support. There are many more operations besides JOIN for processing RDF data, such as FILTER, OPTIONAL, etc. In order to build a complete query engine, we need also to provide other operations. Supporting these operators will certainly have an impact on the decomposition strategy we have proposed and the use of Bloom Filters.

5.2.2 Use Cases

5.2.2.1 Real Time Recommendation System

Recommendation systems are in widespread use today. For example, e-commerce websites such as Amazon provide recommendations on products; reservation websites such as Booking.com provides recommendations about hotels to their users; telecommunication providers such as Orange usually recommend movies or musics to their users; search engines like Google tailor search results based on the knowledge of the users' past searches, etc.

Recommendation systems usually provide contextual relevant user experiences in order to increase conversion rates and user satisfaction. Traditionally, these recommendations have been processed in offline batches, which generated new recommended results with a nightly, weekly or even monthly delay using *kNN* based algorithms. However, usually it is necessary to react in a much shorter time frame in scenarios such as geo-location-based recommendations, timeliness topics, etc.

The methods proposed in this thesis for processing parallel and continuous Data Driven streaming joins can be applied in such systems.

5.2.2.2 Real Time Nature Language Processing System

The purpose of Semantic Web is to make machines process structured and semantically formalized data. There already exists some Semantic Search Engine. These technologies are also used in question-answering communities to make machines better "understand" the questions posed in a human language. Natural language processing (NLP) systems aim at translating human languages into a way that can be processed by computers. The methods for processing natural languages are divided into two categories: rule-based and machine learning-based. Nowadays, the machine learning one is the most used. There already exist some works which combines RDF and NLP, suche as FRED [1], a tool for automatically producing RDF/OWL ontology. In a natural language processing scenario, in order to improve the users' satisfaction, we usually need to provide the results in real time. But the machine learning methods used here such as Support Vector Machine or Hidden Markov Model usually require a long time to train data, which is not very efficient for a streaming process where data is dynamically updated. So combining RDF and Semantic Web technologies with traditional machine learning based NLP methods is an interesting and challenging area.

The methods proposed in this thesis for processing the continuous RDF join in a parallel manner can be applied to systems doing real time natural language processing for queries submitted by users.

[1] http://wit.istc.cnr.it/stlab-tools/fred

Appendix A

Papers published during this thesis

Journal Papers:

1 **K Nearest Neighbour Joins for Big Data on MapReduce: a Theoretical and Experimental Analysis** — IEEE Trans. Knowl. Data Eng., vol. 28, no. 9, pp. 2376–2392, 2016. — G. Song, J. Rochas, L. E. Beze, F. Huet, and F. Magoulès

2 **Detecting topics and overlapping communities in question and answer sites** — Social Netw. Analys. Mining, vol. 5, no. 1, pp. 27:1–27:17, 2015 — Z. Meng, F. L. Gandon, C. Faron-Zucker, and G. Song

Conference Papers:

1 **Solutions for Processing K Nearest Neighbor Joins for Massive Data on MapReduce** — 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015, 2015, pp. 279– 287. — G. Song, J. Rochas, F. Huet,and F. Magoulès

2 **A Hadoop MapReduce Performance Prediction Method** — 10th IEEE International Conference on High Performance Computing and Communications and 2013 IEEE International Conference on Embedded and Ubiquitous Com- puting, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013, 2013, pp. 820–825 — G. Song, Z. Meng, F. Huet, F. Magoulès, L. Yu, and X. Lin

3 **A Game Theory Based MapReduce Scheduling Algorithm** — New York, NY: Springer New York, 2013, pp. 287–296. — G. Song, L. Yu, Z. Meng, and X. Lin

4 **Empirical study on overlapping community detection in question and answer sites** — 2014 IEEE/ACM International Conference on Advances in Social Networks

Analysis and Mining, ASONAM 2014, Beijing, China, August 17-20, 2014, 2014, pp. 344–348. — Z. Meng, F. L. Gandon, C. Faron-Zucker, and G. Song

Work in progress:

1 **Parallel and Continuous Join Processing on Very Large RDF Streams** — Ge Song, Fabrice Huet, Frédéric Magoulès

References

[1] Dragon Curve:https://en.wikipedia.org/wiki/dragon_curve.

[2] Hadoop:http://hadoop.apache.org.

[3] Spark:http://spark.apache.org.

[4] YARN:http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/yarn.html.

[5] Storm:http://storm.apache.org.

[6] RDFS:https://www.w3.org/tr/rdf-schema.

[7] OWL:https://www.w3.org/2001/sw/wiki/owl.

[8] BGP:https://www.w3.org/tr/rdf-sparql-query/#basicgraphpatterns.

[9] Gibbs Sampling: https://en.wikipedia.org/wiki/gibbs_sampling.

[10] Gosper Curve:https://en.wikipedia.org/wiki/gosper_curve.

[11] Hilbert Curve:https://en.wikipedia.org/wiki/hilbert_curve.

[12] Cloudera Impala: https://www.cloudera.com/products/apache-hadoop/impala.html.

[13] Jena: https://jena.apache.org/.

[14] LUBM: http://swat.cse.lehigh.edu/projects/lubm/.

[15] Open MPI: https://en.wikipedia.org/wiki/open_mpi.

[16] Moore's law: https://en.wikipedia.org/wiki/moore%27s_law.

[17] Normal Distribution: https://en.wikipedia.org/wiki/normal_distribution.

[18] Principal Component Analysis :https://en.wikipedia.org/wiki/principal_component_analysis.

[19] Yahoo S4: http://incubator.apache.org/s4/.

[20] SPARQL:http://www.w3.org/tr/rdf-sparql-query.

[21] Spark Streaming: https://www.cloudera.com/products/apache-hadoop/impala.html.

[22] Apache Tez: https://tez.apache.org/.

[23] Z-Value Curve:https://en.wikipedia.org/wiki/z-order_curve.

[24] Priority Queue: https://en.wikipedia.org/wiki/priority_queue.

[25] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management, 2003.

[26] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[27] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 411–422, 2007.

[28] Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *20th International Parallel and Distributed Processing Symposium IPDPS*, 2006.

[29] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[30] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 574–576, New York, NY, USA, 1999. ACM. ISBN 1-58113-084-8.

[31] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. Efficient similarity search in sequence databases. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, FODO '93, pages 69–84, London, UK, UK, 1993. Springer-Verlag. ISBN 3-540-57301-1.

[32] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *Proceedings of the 2nd International Workshop on the Semantic Web (SemWeb 2001)*, 2001.

[33] Alexander Alexandrov, Dominic Battré, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Massively parallel data analysis with pacts on nephele. *PVLDB*, 3(2):1625–1628, 2010.

[34] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 20–29, New York, NY, USA, 1996. ACM. ISBN 0-89791-785-5.

[35] Sattam Alsubaiee, Alexander Behm, Raman Grover, Rares Vernica, Vinayak Borkar, Michael J. Carey, and Chen Li. Asterix: scalable warehouse-style web data integration. In *Proceedings of the Ninth International Workshop on Information Integration on the Web*, IIWeb '12, pages 2:1–2:4, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1239-4.

[36] Mugurel Ionut Andreica and Nicolae T Ăpus. Sequential and mapreduce-based algorithms for constructing an in-place multidimensional quad-tree index for answering fixed-radius nearest neighbor queries, 2013.

[37] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM. ISBN 1-58113-634-X.

[38] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[39] Ahmed Shamsul Arefin, Carlos Riveros, Regina Berretta, and Pablo Moscato. Gpu-fs-knn: A software tool for fast and scalable knn computation using gpus. *PLOS ONE*, 2012.

[40] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, 2002. ISBN 1-58113-507-6.

[41] Xiao Bai, Rachid Guerraoui, Anne-Marie Kermarrec, and Vincent Leroy. Collaborative personalized top-k processing. *ACM Trans. Database Syst.*, 36(4):26:1–26:38, December 2011. ISSN 0362-5915.

[42] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF streams with C-SPARQL. *SIGMOD Record*, 39(1):20–26, 2010.

[43] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *ECCV'06*. 2006. ISBN 978-3-540-33832-1. doi: 10.1007/11744023_32.

[44] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc J. Van Gool. Speeded-up robust features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, 2008.

[45] Alexander Behm, Vinayak R. Borkar, Michael J. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[46] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 1975. ISSN 0001-0782.

[47] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[48] Burton H. Bloom. Space/Time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[49] Christian Böhm and Florian Krebs. The k-nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, 6(6):728–749, November 2004. ISSN 0219-1377.

[50] Christian Böhm, Beng Chin Ooi, Claudia Plant, and Ying Yan. Efficiently processing continuous k-nn queries on data streams. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 156–165, 2007.

[51] Andre Bolles, Marco Grawunder, and Jonas Jacobi. Streaming SPARQL - extending SPARQL to process data streams. In *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings*, pages 448–462, 2008.

[52] Peter A. Boncz, Orri Erling, and Minh-Duc Pham. Experiences with virtuoso cluster RDF column store. In *Linked Data Management.*, pages 239–259. 2014.

[53] Vladimir Braverman, Harry Lang, Keith Levin, and Morteza Monemizadeh. Clustering problems on sliding windows. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, pages 1374–1390. SIAM, 2016. ISBN 978-1-611974-33-1.

[54] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, pages 54–68, 2002.

[55] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[56] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. The haloop approach to large-scale iterative data analysis. *VLDB J.*, 21(2):169–190, 2012.

[57] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: A new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 215–226. VLDB Endowment, 2002.

[58] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 111–122, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1-55860-715-3.

[59] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, S. Krishnamurthy W. Hong, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow, January 2003.

[60] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, pages 693–703, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43864-5.

[61] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 295–306, New York, NY, USA, 2001. ACM. ISBN 1-58113-332-4.

[62] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. ISBN 1-55860-470-7.

[63] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online aggregation and continuous query support in mapreduce. In *SIGMOD Conference*, pages 1115–1118, 2010.

[64] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, April 2005. ISSN 0196-6774.

[65] Graham Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. *ACM Trans. Database Syst.*, 30(1):249–278, March 2005. ISSN 0362-5915.

[66] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, and Anne Rogers. Hancock: A language for extracting signatures from data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 9–17, New York, NY, USA, 2000. ACM. ISBN 1-58113-233-6.

[67] Sudipto Das, Yannis Sismanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, and John McPherson. Ricardo: integrating r and hadoop. In *SIGMOD Conference*, pages 987–998, 2010.

[68] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows: (extended abstract). In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 635–644, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. ISBN 0-89871-513-X.

[69] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, 2004. ISBN 1-58113-885-7.

[70] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, 2004.

[71] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 61–72, New York, NY, USA, 2002. ACM. ISBN 1-58113-497-5.

[72] Wei Dong, Zhe Wang, William Josephson, Moses Charikar, and Kai Li. Modeling lsh for performance tuning. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM '08, pages 669–678, New York, NY, USA, 2008. ISBN 978-1-59593-991-3.

[73] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 577–586, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0632-4.

[74] Qinsheng Du and Xiongfei Li. A novel knn join algorithms based on hilbert r-tree in mapreduce. In *Computer Science and Network Technology (ICCSNT), 2013 3rd International Conference on*, pages 417–420, 2013.

[75] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *HPDC*, pages 810–818, 2010.

[76] Reza Farivar, Anand Raghunathan, Srimat T. Chakradhar, Harshit Kharbanda, and Roy H. Campbell. Pic: Partitioned iterative convergence for clusters. In *CLUSTER*, pages 391–401, 2012.

[77] Michael J. Fischer, Xueyuan Su, and Yitong Yin. Assigning tasks for efficiency in hadoop: extended abstract. In *SPAA*, pages 30–39, 2010.

[78] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[79] Venkatesh Ganti, Mong-Li Lee, and Raghu Ramakrishnan. Icicles: Self-tuning samples for approximate query answering. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 176–187, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1-55860-715-3.

[80] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *SIGMOD Conference*, pages 1123–1134, 2008.

[81] Lars George. *HBase - The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly, 2011. ISBN 978-1-449-39610-7.

[82] Phillip B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 541–550, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-804-4.

[83] Phillip B. Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 331–342, New York, NY, USA, 1998. ACM. ISBN 0-89791-995-5.

[84] Phillip B. Gibbons and Srikanta Tirthapura. Distributed streams algorithms for sliding windows. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 63–72, New York, NY, USA, 2002. ACM. ISBN 1-58113-529-7.

[85] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 79–88, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-804-4.

[86] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB'99*, 1999.

[87] Lukasz Golab and M Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 500–511, 2003. ISBN 0-12-722442-4.

[88] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 58–66, New York, NY, USA, 2001. ACM. ISBN 1-58113-332-4.

[89] Rong Gu, Shanyong Wang, Fangfang Wang, Chunfeng Yuan, and Yihua Huang. Cichlid: Efficient large scale RDFS/OWL reasoning with spark. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 700–709, 2015.

[90] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, 2012.

[91] Parisa Haghani, Sebastian Michel, and Karl Aberer. Lsh at large distributed knn search in high dimensions. In *WebDB'08*, 2008.

[92] Mohammad Hammoud and Majd F. Sakr. Distributed programming for the cloud: Models, challenges, and analytics engines. In *Large Scale and Big Data - Processing and Management.*, pages 1–38. 2014.

[93] Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed-Mehdi-Reza Beheshti, and Sherif Sakr. DREAM: distributed RDF engine with adaptive query planner and minimal communication. *PVLDB*, 8(6):654–665, 2015.

[94] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk RDF storage. In *PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida, USA, October 20, 2003*, 2003.

[95] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A federated repository for querying graph structured data from the web. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007.*, pages 211–224, 2007.

[96] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.

[97] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010.

[98] Mohammad Farhan Husain, Pankil Doshi, Latifur Khan, and Bhavani M. Thuraisingham. Storage and retrieval of large RDF graph using hadoop and mapreduce. In *Cloud Computing, First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings*, pages 680–686, 2009.

[99] Mohammad Farhan Husain, James P. McGlothlin, Mohammad M. Masud, Latifur R. Khan, and Bhavani M. Thuraisingham. Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Trans. Knowl. Data Eng.*, 23(9):1312–1327, 2011.

[100] K. Inthajak, C. Duanggate, B. Uyyanonvara, S.S. Makhanov, and S. Barman. Medical image blob detection with feature stability and knn classification. In *Computer Science and Software Engineering*, 2011.

[101] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 2005.

[102] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[103] Changqing Ji, Tingting Dong, Yu Li, Yanming Shen, Keqiu Li, Wenming Qiu, Wenyu Qu, and Minyi Guo. Inverted grid-based knn query processing with mapreduce. In *Proceedings of the 2012 Seventh Grid Annual Conference*, CHINAGRID '12, pages 25–32, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4816-6.

[104] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. *PVLDB*, 3(1):472–483, 2010.

[105] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 341–352, 2003.

[106] Masaru Kitsuregawa and Yasushi Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 210–221, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc. ISBN 1-55860-149-X.

[107] Flip Korn, Nikolaos Sidiropoulos, Christos Faloutsos, Eliot Siegel, and Zenon Protopapas. Fast nearest neighbor search in medical image databases. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 215–226, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. ISBN 1-55860-382-4.

[108] Hans-Peter Kriegel and Thomas Seidl. Approximation-based similarity search for 3-D surface segments. *Geoinformatica*, 1998. ISSN 1384-6175.

[109] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6.

[110] Kisung Lee and Ling Liu. Scaling queries over big RDF graphs with semantic hash partitioning. *PVLDB*, 6(14):1894–1905, 2013.

[111] Dingzeyu Li, Qifeng Chen, and Chi-Keung Tang. Motion-aware knn laplacian for video matting. In *ICCV'13*, 2013.

[112] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: a new architecture for high-performance stream systems. In *In Proc. Of the VLDB Endowment*, pages 274–288, 2008.

[113] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proc. VLDB Endow.*, 2012. ISSN 2150-8097.

[114] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Dynamic maintenance of wavelet-based histograms. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 101–110, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1-55860-715-3.

[115] Brian McBride. Jena: Implementing the RDF model and syntax specification. In *SemWeb*, 2001.

[116] Moni Naor and Eylon Yogev. Tight bounds for sliding bloom filters. *Algorithmica*, 73 (4):652–672, 2015.

[117] Thomas Neumann and Gerhard Weikum. RDF-3X: a risc-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[118] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *ICDM Workshops*, pages 170–177, 2010.

[119] David Novak and Pavel Zezula. M-chord: A scalable distributed similarity search structure. In *Scalable Information Systems*, 2006. ISBN 1-59593-428-6.

[120] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[121] George Papandreou, Iasonas Kokkinos, and Pierre-André Savalle. Modeling local and global deformations in deep learning: Epitomic convolution, multiple instance learning, and sliding window detection. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 390–399, 2015.

[122] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *Proc. VLDB Endow.*, 5(10): 992–1003, June 2012. ISSN 2150-8097.

[123] Erion Plaku and Lydia E. Kavraki. Distributed computation of the knn graph for large high-dimensional point sets. *J. Parallel Distrib. Comput.*, 67(3):346–359, March 2007. ISSN 0743-7315.

[124] Jorda Polo, David Carrera, Yolanda Becerra, Malgorzata Steinder, and Ian Whalley. Performance-driven task co-scheduling for mapreduce environments. In *NOMS*, pages 373–380, 2010.

[125] Lin Qiao, D. Agrawal, and A. El Abbadi. Supporting sliding window queries for continuous data streams. In *Scientific and Statistical Database Management, 2003. 15th International Conference on*, pages 85–94, July 2003.

[126] Davood Rafiei and Alberto Mendelzon. Similarity-based queries for time series data. *SIGMOD Rec.*, 26(2):13–25, June 1997. ISSN 0163-5808.

[127] Zujie Ren, Zhijun Liu, Xianghua Xu, Jian Wan, Weisong Shi, and Min Zhou. Waxelephant: A realistic hadoop simulator for parameters tuning and scalability analysis. In *Seventh ChinaGrid Annual Conference, ChinaGrid 2012, Beijing, China, September 20-23, 2012*, pages 9–16, 2012.

[128] Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 110–121, New York, NY, USA, 1989. ACM. ISBN 0-89791-317-5.

[129] M. Slaney and M. Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *Signal Processing Magazine, IEEE*, 25(2):128–131, March 2008. ISSN 1053-5888. doi: 10.1109/MSP.2007.914237.

[130] Ge Song, Zide Meng, Fabrice Huet, Frédéric Magoulès, Lei Yu, and Xuelian Lin. A hadoop mapreduce performance prediction method. In *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE*

International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013, pages 820–825, 2013. doi: 10.1109/HPCC.and.EUC.2013.118.

[131] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

[132] Aleksandar Stupar, Sebastian Michel, and Ralf Schenkel. Rankreduce - processing k-nearest neighbor queries on top of mapreduce. In *In LSDS-IR*, 2010.

[133] Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '98, pages 2–2, Berkeley, CA, USA, 1998. USENIX Association.

[134] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 996–1005, 2010. doi: 10.1109/ICDE.2010.5447738.

[135] Srikanta Tirthapura, Bojian Xu, and Costas Busch. Sketching asynchronous streams over a sliding window. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 82–91, New York, NY, USA, 2006. ACM. ISBN 1-59593-384-0.

[136] Wee Hyong Tok and Stéphane Bressan. Progressive and approximate join algorithms on data streams. In *Advanced Query Processing, Volume 1: Issues and Trends*, pages 157–185. 2013.

[137] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2376-5.

[138] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 235–244, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0607-2. doi: 10.1145/1998582.1998637.

[139] Jeffrey Scott Vitter, Min Wang, and Bala Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of the Seventh International Conference on Information and Knowledge Management*, CIKM '98, pages 96–104, New York, NY, USA, 1998. ACM. ISBN 1-58113-061-9.

[140] T. T. Vu and F. Huet. A lightweight continuous jobs mechanism for mapreduce frameworks. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 269–276, May 2013.

[141] D. Wang, Y. Zheng, and J. Cao. Parallel construction of approximate knn graph. In *Distributed Computing and Applications to Business, Engineering Science (DCABES), 2012 11th International Symposium on*, pages 22–26, Oct 2012.

[142] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[143] Kevin Wilkinson and Kevin Wilkinson. Jena property table implementation. In *In SSWS*, 2006.

[144] Yu Xu, Pekka Kostamaa, and Like Gao. Integrating hadoop and parallel dbms. In *SIGMOD Conference*, pages 969–974, 2010.

[145] Chong Yang, Xiaohui Yu, and Yang Liu. Continuous KNN join processing for real-time recommendation. In *2014 IEEE International Conference on Data Mining, ICDM 2014, Shenzhen, China, December 14-17, 2014*, pages 640–649, 2014.

[146] Y. Yang, Y. Huang, J. Cao, X. Ma, and J. Lu. Design of a sliding window over distributed and asynchronous event streams. *IEEE Transactions on Parallel and Distributed Systems*, 25(10), Oct 2014. ISSN 1045-9219.

[147] Bin Yao, Feifei Li, and P. Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In *Data Engineering*, 2010.

[148] Bin Yao, Feifei Li, and P. Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 4–15, March 2010.

[149] Cui Yu, Rui Zhang, Yaochun Huang, and Hui Xiong. High-dimensional knn joins with incremental updates. *GeoInformatica*, 14(1):55–82, 2010.

[150] Cui Yu, Rui Zhang, Yaochun Huang, and Hui Xiong. High-dimensional knn joins with incremental updates. *Geoinformatica*, 2010. ISSN 1384-6175.

[151] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. TripleBit: a fast and compact system for large scale RDF data. *PVLDB*, 6(7):517–528, 2013.

[152] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[153] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4):265–276, 2013.

[154] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in mapreduce. In *Extending Database Technology*, 2012. ISBN 978-1-4503-0790-1.

[155] Xiaofang Zhou, David J. Abel, and David Truffet. Data partitioning for parallel spatial join processing. *Geoinformatica*, 1998. ISSN 1384-6175.

université PARIS-SACLAY | ÉCOLE DOCTORALE INTERFACES Approches Interdisciplinaires : Fondements, Applications et Innovation

Titre : Méthodes parallèles pour le traitement des flux de données continus (en français)

Mots clés : Big Data, Flux de Données, Calculation en Parallèle, Exploration de Données

Résumé : Nous vivons dans un monde où une grande quantité de données est généré en continu. Par exemple, quand on fait une recherche sur Google, quand on achète quelque chose sur Amazon, quand on clique en 'Aimer' sur Facebook, quand on upload une image sur Instagram, et quand un capteur est activé, etc., de nouvelles données vont être généré. Les données sont différentes d'une simple information numérique, mais viennent dans de nombreux format. Cependant, les données prisent isolément n'ont aucun sens. Mais quand ces données sont reliées ensemble on peut en extraire de nouvelles informations. De plus, les données sont sensibles au temps. La façon la plus précise et efficace de représenter les données est de les exprimer en tant que flux de données. Si les données les plus récentes ne sont pas traitées rapidement, les résultats obtenus ne sont pas aussi utiles.

Ainsi, un système parallèle et distribué pour traiter de grandes quantités de flux de données en temps réel est un problème de recherche important. Il offre aussi de bonne perspective d'application. Dans cette thèse nous étudions l'opération de jointure sur des flux de données, de manière parallèle et continue. Nous séparons ce problème en deux catégories. La première est la jointure en parallèle et continue guidée par les données. La second est la jointure en parallèle et continue guidée par les requêtes.

Title : Parallel and continuous join processing for data stream (in English)

Keywords : Big Data, Data Stream, Parallel Computing, Data Mining

Abstract : We live in a world where a vast amount of data is being continuously generated. Data is coming in a variety of ways. For example, every time we do a search on Google, every time we purchase something on Amazon, every time we click a 'like' on Facebook, every time we upload an image on Instagram, every time a sensor is activated, etc., it will generate new data. Data is different than simple numerical information, it now comes in a variety of forms. However, isolated data is valueless. But when this huge amount of data is connected, it is very valuable to look for new insights. At the same time, data is time sensitive. The most accurate and effective way of describing data is to express it as a data stream. If the latest data is not promptly processed, the opportunity of having the most useful results will be missed.

So a parallel and distributed system for processing large amount of data streams in real time has an important research value and a good application prospect. This thesis focuses on the study of parallel and continuous data stream Joins. We divide this problem into two categories. The first one is Data Driven Parallel and Continuous Join, and the second one is Query Driven Parallel and Continuous Join.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France