COMPLEX QUERY PROCESSING AND RECOVERY IN DISTRIBUTED SYSTEMS

SHEN YANYAN

Bachelor of Science

Peking University, China

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2015

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety.

I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has not been submitted for any degree in any university previously.

Shen Yanyan August 11, 2015

ACKNOWLEDGMENT

I want to express my sincere gratitude to my supervisor, Prof. Beng Chin Ooi, for his continuous guidance and support over the past five years. I knew little about research when I started my PhD study. It was Prof. Ooi who taught me how to become a good researcher and enlightened me on challenging research problems. No matter how busy he is, he has always been available to answer my questions and offer his wise advice. I am very grateful to his encouragement when my papers got rejected and his forgiveness to my poor written English.

I would like to thank Divesh Srivastava, Luna Xin Dong, Laks V.S. Lakshmanan, Luciano Barbosa, my mentors during my summer internships at AT&T Lab in year 2011 and 2012. They taught me valuable research skills and right working attitude. Thank you to Divesh, for innumerable technical discussions, informal chats about life and insightful advice on our research projects. I would also like to thank Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, my internship mentors at Microsoft Research Redmond, for their guidance and support on the search problem. It has been such a pleasure working with all of my mentors. In addition, I would like to thank all the interns I met at AT&T Lab and Microsoft DMX group. Without them, I would not have had such great and productive summers.

I would like to thank my thesis committee members, Prof. Kian-Lee Tan and Prof. Chee-Yong Chan, for their helpful suggestions and insightful comments on this dissertation.

I would like to thank all my colleagues in the database group for their company during my entire PhD life. Special thanks to Prof. Wei Lu, who helped me through all the three thesis works and provided helpful advice. Thanks to my seniors, Sai Wu, Su Chen, Shanshan Ying, Ju Fan, Xuan Liu, Meihui Zhang, Meiyu Lu, Feng Li, Peng Lu, and my junior fellows, Jinyang Gao, Sheng Wang, Qian Lin, for their assistance and support to my research and life.

I am always grateful to my long-term house mates, Jingwen Bian, Chao Chen, Xiao Liu, Guanfeng Wang, Jing Yang and Jie Yang, who have shared many exciting and joyful day and night with me. Thank all of you for the assistance to my life and putting up with my bad temper.

I would like to thank my best friends, Qi Sun, Minhui Xu, Chengyuan Yang and Yiqing Wu, who were shocked by my intention to pursue a PhD degree and missing me all the time when I am in Singapore. We have known each other for over 12 years and I believe our friendship will live forever.

Finally, I want to express my deepest gratitude to my parents for their endless love, support, understanding and encouragement to me.

CONTENTS

ABSTRACT

We live in the era of Big Data, where data is being created, collected and integrated at an unprecedented scale. To uncover the true value of Big Data, distributed system is unquestionably one of the most important and effective solutions. Among all the existing distributed systems, epiC is one of the most elastic and extensible data processing systems proposed for Big Data. epiC adopts a general Actor-like concurrent programming model which is able to handle multi-structured data and execute different kinds of computations in a single system. While epiC provides a simple yet extensible interface to cope with various types of Big Data applications, many challenges still remain to be solved, such as data storage, complex query processing, simplicity management and resilience to failures.

In this thesis, we aim to develop effective and efficient solutions to address two challenging issues in epiC: complex query processing and failure recovery. We employ epiC as our underlying distributed system due to its simplicity, efficiency and extensibility, but our approaches can be implemented in other distributed systems as well. For the query processing, we first focus on the problem of answering k nearest neighbor join queries in epiC. We then introduce our graph processing engine, epiCG, to handle graph-related analytics queries. epiCG is built on top of epiC and supports both edge-cut and vertex-cut partitioning methods. Lastly, we address the recovery problem in epiC/epiCG. The traditional checkpoint-based recovery works well for one-pass jobs such as kNN join, but it incurs long recovery latency for iterative graph applications. We discuss in detail the drawbacks of the checkpoint-based recovery method and

propose a novel parallel recovery mechanism. We also implement our recovery method in epiCG. For all the three pieces of work, we compare our approaches with state-of-the-art solutions and conduct extensive experiments using real datasets and multiple benchmark tasks.

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

In the era of Big Data, data is being created and collected at an unprecedented scale in a broad range of application areas. In social science, for instance, over 100 billion emails were sent and received per day worldwide in 2013 [72]; more than 15TB data were collected daily in Facebook in 2012 [8]; over 500 million tweets were sent to Twitter per day in 2013 [1]; 100 hours of videos were uploaded to YouTube every minute in 2014 [6]. According to a recent report [7], 90% of world's data have been generated over the past two years. Along with the Big Data explosion, tremendous successes have been achieved by analyzing the sheer volume of data being generated. A McKinsey report estimated that data analytics could save U.S. healthcare costs by 300 to 450 billion annually [38]. In [61], it was estimated that services enabled by individual locational data could help consumers to capture over 600 billion dollars in economic surplus.

While the potential benefits of Big Data are significant, it is challenging to uncover the true value of Big Data due to its three V characteristics. The first V is Volume, i.e., data size. The sheer size of data requires the capability to continuously act upon the large-scale growing data. Velocity and Variety are the other two Vs of Big Data. Velocity refers to the high generation speed of data and Variety refers to diverse data types. Recently, Big Data has also been characterized by Veracity, which refers to the noises, biases and abnormality in data. All the Vs in Big Data introduce a large number of challenging issues such as scale, heterogeneity, statistical errors, privacy, data storage, data integration and query processing. Without the ability to address all these crucial issues, the true value of Big Data is locked.

As we will see later, traditional centralized system infrastructures and computation methods are far from satisfactory in terms of supporting Big Data analytics. It is therefore not surprising that traditional solutions that leverage multi-core and multi-thread to speed up data processing even do not have enough space to store data due to its sheer size. Supercomputers employ a massive number of multi-core processors that are collaborated with each other in a complicated way to maximize the computing capability. While supercomputers are indeed powerful and competitive in high-performance computing, they are really expensive and can hardly be afforded by typical IT companies and research communities. Moreover, data is growing at a much faster rate than the performance improvement of supercomputers [11].

To handle the challenges of Big Data, distributed processing over a cluster of community computers has gained attraction in recent years. In general, computers in a distributed system are physically distributed and each computer is associated with its own memory and disk space, and is responsible for a subset of computation tasks. All the computers perform computation in parallel and communicate with each other via network messages. While a single computer has limited capability in terms of both storage and computing power, the collaboration of multiple computers exhibits competitive computing capability compared with a general-purpose computer. More importantly, distributed systems based on clusters of community computers are more affordable for mid-sized companies and research communities.

Figure 1.1 summarizes the state-of-the-art distributed systems proposed for Big Data applications. Among all the existing distributed systems, epiC [47] is one of the most elastic and extensible data processing systems designed for Big Data applications. The core abstraction of epiC is a general Actor-like concurrent programming model which is able to execute different kinds of computations (called *units*), independent of the data processing models. This flexible design allows users to handle multi-structured data in a single system, by processing each data type with the most appropriate data processing model.

While epiC provides a simple yet extensible unit interface to cope with various types of Big Data applications, many challenges still remain to be solved. From the perspective of application design, it should be able to support various

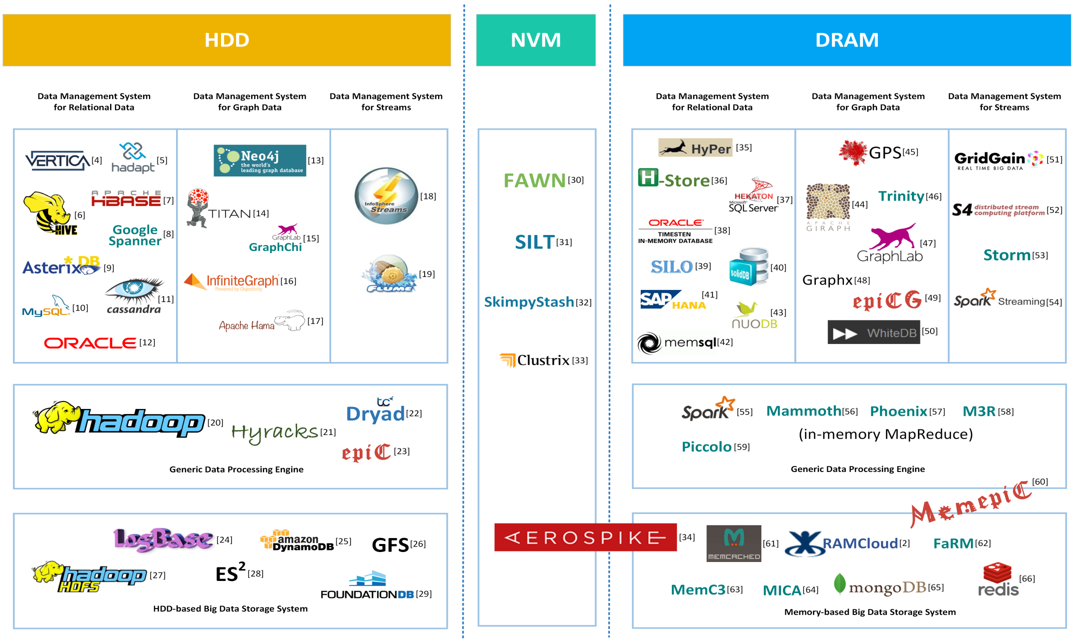


Figure 1.1: Landscape of advanced distributed systems [97]

analytics tasks over Big Data efficiently; from the perspective of system design, it should possess several properties such as simplicity, scalability, elasticity and fault tolerance. In this thesis, we aim to develop effective and efficient solutions to address two challenging issues in epiC: complex query processing and failure recovery.

Various real-life applications, such as data mining, pattern recognition, multimedia and geographic analysis, require to analyze Big Data via *complex* queries, i.e., queries that cannot be easily expressed by standard SQL queries or non-relational (e.g., NoSQL) queries. k nearest neighbor join is an important example of complex queries that combines each object of one dataset with k nearest neighbors of another dataset. As a primitive operation, kNN join serves a broad spectrum of data mining applications. For instance, in each iteration of the well-known k-Means and k-Medoid clustering, a set of cluster centers are computed and each data point will be assigned to its nearest center. This point assignment process corresponds to a $k = 1$ nearest neighbor join between the set of center points and the set of data points. In k nearest neighbor classification, we need to decide the class labels of unclassified data objects based on a set of classified objects (k is a pre-defined parameter). To do this, for each unclassified object, a k nearest neighbor query on the set of classified objects is evaluated. This process corresponds again to a k nearest neighbor join between the set of unclassified objects and the set of classified objects. Other kNN join based applications include (but are not limited to) the following list: sample assessment, sample post-processing, missing value imputation, and k-distance diagrams [15, 16]. While kNN join covers almost all the stages of knowledge discovery process [15], it mainly solves complex queries over high-dimensional data objects and is insufficient to handle complex graph analytics queries that require to perform iterative computation over large graph data.

Recent years have witnessed the emergence of large real-life graphs such as social networks (e.g., Facebook, LinkedIn), spacial networks (e.g., Google Maps, FedEx) and the Web. Querying and mining large graphs are becoming increasingly important in many real applications. Examples include two-hop friend list and influence analysis in social networks [80, 37], traffic analysis and route recommendation over spacial graphs [89, 28, 32], PageRank [67] and reverse link web graph computation over the Web graph. In most applications, the sheer size of graph data creates a critical need for distributed systems to handle various graph analytics queries more efficiently.

While epiC allows us to accomplish heterogeneous analytics tasks in a single system, designing efficient algorithms in epiC to answer the above two complex queries, kNN join and graph analytics queries, is still challenging with the following two reasons.

- kNN join queries can hardly be handled via a single-unit epiC job. Different kinds of units must be implemented and collaborated with each other to process data gradually. The design of each unit has to take two important issues into account. The first issue is to balance the load among the units that process data simultaneously in parallel. The second issue is to reduce the network communication cost between units. Both issues are critical to the performance of complex query processing, especially under Big Data workloads.

- Most graph analytics tasks, such as PageRank and shortest path computation, require iterative computation over large graphs. However, epiC is a disk-based distributed solution which requires to flush the computed results (e.g., computed graph data) to the distributed file system at the

> end of each iteration and reload them to the memory in the beginning of next iteration. This incurs high network cost and degrades the system performance. Hence, it is important to enhance epiC to support iterative computation for graph analytics tasks more efficiently.

The second challenging issue we want to address is the failure recovery problem in epiC. That is, epiC should continue operating properly and quickly when some of the system components fails. Failure recovery is one of the most fundamental problems that must be faced when we run programs in distributed systems. In fact, the increasing data size and analytics complexity inevitably increase the failure probability of machines in the distributed systems. Currently, epiC, as well as other advanced distributed systems, adopts checkpoint-based approach to recover from failures. During the computation, the system will periodically save its runtime status to persistent storage as a checkpoint. When a failure occurs, the system will reload the latest checkpoint and restart the computation since then. For non-iterative analytics tasks such as kNN join, checkpoint-based recovery is efficient and easy to implement [71]. However, for graph analytics tasks, it might incur high recovery latency as all the computers in the cluster have to redo the lost iterations since the latest checkpoint even if a computer has finished its computation task and never failed. This inspires us to develop more efficient recovery mechanism to reduce the recovery overhead.

To address the above two challenging issues (i.e., complex query processing and failure recovery), in this thesis, we first study the problem of answering k nearest neighbor join query in epiC. We then extend epiC and develop an efficient graph processing engine, called epiCG, on top of epiC, to handle graph analytics queries efficiently. For the recovery issue, the traditional checkpoint-based recovery method works well for the non-iterative jobs such as kNN join [71], but incurs long recovery latency for the iterative graph analytics tasks. Hence, we propose a novel parallel recovery mechanism and implement it in epiCG to accelerate the recovery process.

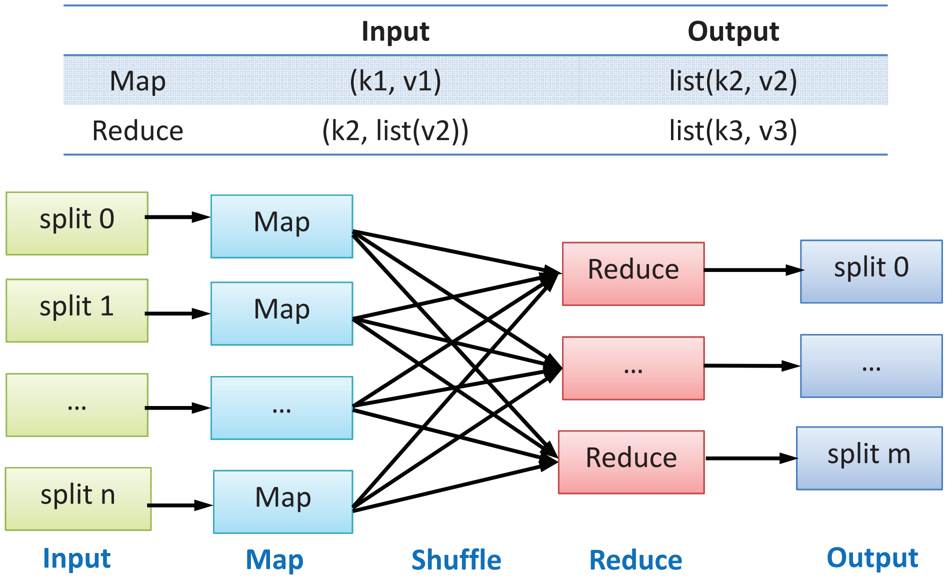
In the remainder of this chapter, we first review several advanced distributed systems. We then present research challenges in distributed systems and provide background of complex query processing and failure recovery in distributed systems. Finally, we outline the objectives of this thesis and provide an outline of the thesis.

1.1 Brief Review of Distributed Systems

MapReduce [31] and its open source implementation Hadoop [3] are undoubtedly an advanced distributed solution proposed for Big Data analytics. MapReduce is a distributed platform with two primitive functions, Map and Reduce, for the purpose of data processing. Figure 1.2 illustrates the processing logic of MapReduce framework. The Map function absorbs a key-value pair as input and transforms it into an intermediate key-value pair. The Reduce function takes all the intermediate key-value pairs with the same key as input and produces a key-value pair in the final result. In MapReduce, programmers are responsible for implementing the Map and Reduce functions, while the system will manage the overall computation and communication process automatically. MapReduce achieves great success and popularity due to its following features.

- Flexibility. Programmers can write simple Map and Reduce functions to process data over a large cluster without the knowledge of how the MapReduce job is performed in the underlying distributed system.
- Efficiency. MapReduce does not require input data to be stored in the database before the processing. Therefore, it is very efficient for the applications that only process data in a small number of passes.
- Scalability. MapReduce supports data parallel partitioned execution. To cope with the increasing size of data and load, MapReduce can easily leverage more computers to execute Map and Reduce functions in parallel and achieve high computing capability.
- Fault tolerance. A MapReduce job is typically processed by a cluster of computers. Once a computer/mapper/reducer fails, MapReduce can recover from the failure automatically and the programmers do not need to worry about the failures during the period of job execution.

To enhance the performance of MapReduce, many extensions based on MapReduce framework have been developed. For example, Sailfish [73] modifies the transportation layer between mappers and reducers in order to reduce the network cost of shuffling intermediate key-value pairs; FileMap [35] is a file-based distributed system, in which data is stored in Unix files and no distributed file system is required; Themis [74] aims to reduce I/O cost for executing MapReduce jobs. More proposals can be found in a recent survey [54].



	Input	Output
Map	(k1, v1)	list(k2, v2)
Reduce	(k2, list(v2))	list(k3, v3)

Figure 1.2: MapReduce framework

One limitation of the above proposed systems is that they are not suitable for iterative computations. This is because most of these systems require expensive I/O operations towards underlying file systems during each iteration of computation. In MapReduce, for example, all the data will be flushed out to the distributed file system (DFS) at the end of one iteration and be retrieved from the DFS at the beginning of next iteration. It is important to note that iterative computations do exist in many real-life analytics jobs such as PageRank, shortest path computation and connected component computing. Recently, a new iterative programming model, called Pregel [59], has been proposed by Google to deal with iterative computation. Pregel aims to handle iteration computations for graph-oriented applications. That is, the input of a Pregel job is typically a directed graph. As shown in Figure 1.3, a Pregel job consists of three phases: an *input* phase to load and distribute graph data among a cluster of compute nodes, followed by a set of *supersteps* for iterative computations, and finally an *output* phase to produce the computed results. Pregel adopts vertex-centric computation model. In each superstep, every vertex executes the *compute* function specified by the programmers and sends messages to other vertices. When all the vertices finish computations and forward messages successfully in one superstep, they proceed to the next superstep synchronously. Pregel eliminates costly I/O operations by maintaining all the

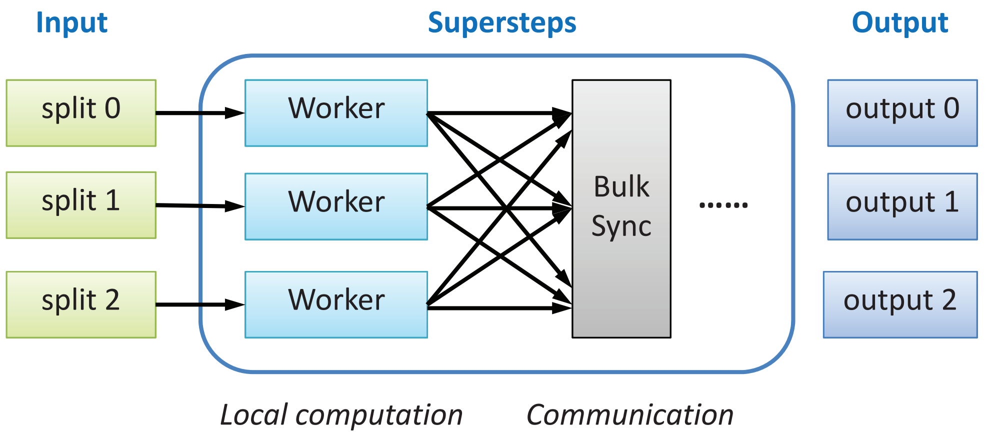


Figure 1.3: Pregel overview

graph data and messages in main memory during the iterative computations. Inspired by Pregel, various vertex-centric distributed systems are developed to support graph-parallel computations. GPS [75], Hama [5] and Giraph [2] provide similar APIs as Pregel. Trinity [76] deploys a distributed memory cloud to support both online graph query processing and offline graph analytics tasks. GraphLab [56] allows vertices to perform computation asynchronously. Unlike Pregel, GraphLab provides three primitives *gather*, *apply* and *scatter* for graph computation. Pregelix [4] aims to support both in-memory and out-of-core graph workloads efficiently. Pregelix is built on top of Hyracks [20] and leverages the out-of-core data management techniques and optimizations from Hyracks to accelerate the processing for extremely large graphs.

As we can see, different distributed systems (e.g., MapReduce, Pregel) have been developed to process data of different types (e.g., key-value pairs, graph data). However, due to the high variety of Big Data, we cannot afford to build a specific distributed system for each particular type of data/task. To address the high variety challenge, Jiang et al. [47] proposed a novel distributed system, called epiC. epiC adopts a general Actor-like programming model and provides a simple yet efficient *unit* interface to support various computation models. Figure 1.4 provides an overview of epiC. In epiC, users can express different computation logics by defining different units. Each processing unit performs computation in parallel with other units and communicates with other units through message passing. For example, MapReduce framework can be easily developed by implementing two units, MapUnit and ReduceUnit; relational

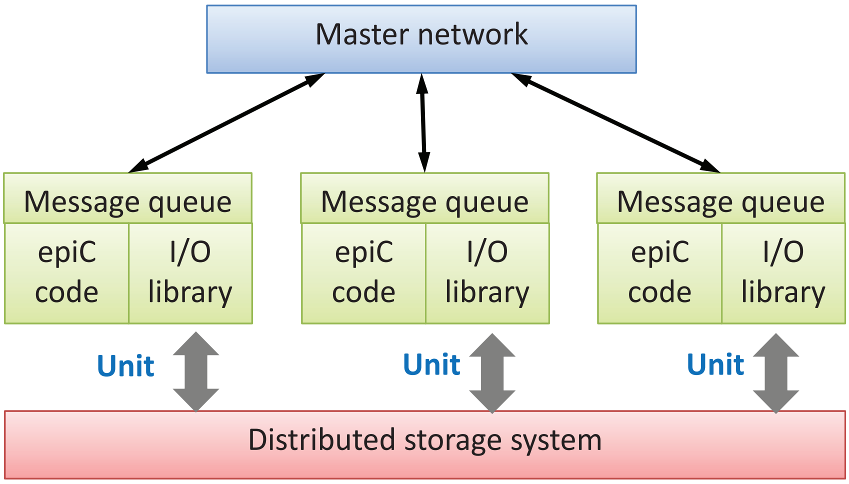


Figure 1.4: epiC overview

model can be implemented by designing SQL-related units such as SingleTableUnit to process a single table, JoinUnit to join two tables based on join keys and AggregateUnit to collect partitions of different groups and calculate the aggregated results for each group. Such kind of unit-based solution allows programmers to process each data type with the most appropriate data processing model. More importantly, thanks to the flexibility and extensibility of epiC, various data types and data analytics tasks can be handled appropriately in a single distributed system.

1.2 Research Challenges in Distributed Systems

Developing a distributed system with promising capability of handling Big Data analytics is a non-trivial task. In this section, we first provide an overview of research challenges in distributed systems. We then elaborate on two important challenges: complex query processing and resilience to failures.

1.2.1 Overview

In order to ensure that distributed systems are efficient, scalable and reliable, we have to address the following challenging issues.

- Storage. Data storage is a fundamental challenge in distributed systems. Data processed by distributed systems can be stored in various types of data storages such as shared data storage, main memory and real-time data streams. Whether the storage is *effective* or not have a great impact on the execution of the upper-level applications. Typically, the *effectiveness* is referred to as long-term duration, provenance, availability, consistency, performance, etc.

- Query processing. Query processing is inevitably a crucial challenge in distributed systems. Typically, we consider two kinds of queries: offline data analytics queries and online transactional queries. In general, query processing in distributed systems has to address several problems: correctness, efficiency, scalability, accuracy and speedup. Noting that there does not exist a distributed system that can fit all requirements with one size, different kinds of distributed systems have been developed and each of them focuses on some particular queries.

- System management. Distributed systems are much harder to manager than stand-alone systems and the complexity stems from the complex collaborations (computation and communication) among multiple computers and complicated infrastructure for the purpose of efficiently processing data in sheer size. To make distributed systems more applicable, an important research challenge is to make the management of distributed system simpler. The management includes system configuration and upgrade, software development, install, update and remove.

- Fault tolerance. Failures are inevitable and a reliable distributed system must have the ability to detect the occurrence of failures automatically. Various types of failures could happen in distributed systems, either from software or from hardware. Once a failure is detected, the system has to perform recovery resiliently such that the overall recovery process is transparent to the users. For the distributed systems that deal with real-time query processing, efficient failure recovery is required to ensure high availability.

- Security. Security issues in distributed systems may come from network vulnerability, erroneous operations performed by the users, malicious software used in distributed systems, etc. The development of distributed systems should be able to guarantee the anonymity of sensitive data and the correctness of computation results.

In this thesis, we mainly focus on two challenging issues in distributed systems: complex query processing and fault tolerance.

1.2.2 Complex Query Processing

To discover the value of Big Data, modern distributed systems such as MapReduce aim to support large-scale data-driven analytics. Therefore, the first and foremost challenge in distributed system design is its ability to answer complex data analytics queries. Broadly, we categorize complex analytics queries into the following three categories.

- SQL-like data processing. SQL-like data processing is to implement basic database operations to process the data. The operations include projection, selection, aggregation and join. Join operations can be further categorized into similarity join, kNN join, equijoin, etc.

- Iterative computation. Many data mining and machine learning applications require to perform computation over data sets iteratively, e.g., social network analysis, web data ranking, clustering. One famous example of iterative computation is PageRank [67], which continuously calculates the PageRanks of all the webpages. Typically, the input of an iterative computation job (e.g., PageRank) consists of an invariant part that will not change in different iterations (e.g., static connection graph for the web pages), and a variant part that will change during the iterations (e.g., PageRank of each web page computed after each iteration).

- Stream and continuous query processing. Many analytics queries such as stream processing [78, 12] and online aggregation [41, 52] cannot retrieve all the data they need before computation, but have to deal with continuous data streams. These queries will be issued once and then logically run continuously over data streams.

Traditional centralized approaches to complex query processing cannot be easily transformed to efficient distributed processing. Hence, various novel approaches have been proposed to answer complex queries efficiently in a distributed environment.

For SQL-like data processing, more attention has been paid to accelerate the process of performing various types of joins such as *theta-join* [64, 99], *equijoin* [14], *similarity join* [9, 62, 85] and *multiway join* [46]. Most of the proposed solutions adopt MapReduce as the underlying distributed system, and the proposed solutions provide particular Map and Reduce functions for different join operations. Various optimizations are provided to balance the workload among mappers and reduces, and reduce the shuffling cost of transmitting intermediate data from mappers to reducers. In addition to the implementations of database operations, several high-level languages such as PigLatin [65] and HiveQL [81] have also been introduced for MapReduce. These high-level languages are well supported by many distributed systems such as Pig [65] and Hive [81]. In these systems, programmers do not have to implement database operations on their own. Instead, they can pose complex SQL queries using the high-level languages, and the system will automatically translate these queries into a sequence of lower-level operations that have already been implemented in the systems.

For queries that involve iterative computation, a straightforward solution is to decompose the query into a sequence of analytics jobs and execute the jobs sequentially. However, such a solution requires to retrieve both invariant and variant input data at the beginning of the execution of each job and flush them out whenever a job finishes. Obviously, this will incur high I/O cost as read and write in distributed systems always require remote data access via network. To address the problem, HaLoop [25], a variant of MapReduce was proposed to support iteration computation efficiently. Specifically, HaLoop caches invariant data across all the iterations via two new primitives AddMap and AddReduce. Furthermore, HaLoop supports automatic termination. That is, if two consecutive sets of reducers' output are identical, HaLoop will terminate the iteration and report the final results. Recently, a new computation model, called Pregel [59], was introduced to perform iterative computation for graph applications. Unlike MapReduce, Pregel only involves two I/O phases. One is the input phase that retrieves input data at the beginning of a Pregel job; another is the output phase that flushes out the final results at the end of the job. During the iterations, all the data, either invariant or variant, will be maintained in the memory.

For stream and continuous query processing, continuous operations will be performed over continuously incoming data. Consider, for example, an aggregation query over a data stream. The key challenge to deal with continuous queries is to effectively pipeline the execution of consecutive operations. MapReduce Online [30] extends MapReduce framework and addresses two kinds of pipelines in MapReduce. First, it supports the pipeline between mappers and reducers. While MapReduce requires reducers to pull data from mappers, MapReduce Online asks mappers to push the output data to the reducers and the reducers will sort incoming data locally. Second, if an application requires multiple MapReduce jobs to be executed sequentially, MapReduce Online also supports the pipeline between two consecutive jobs, which is to transfer output data from reducers directly to the mappers of the next job. For example, consider a sort-merge query that involves two jobs, one job for sorting followed by another job for merging. MapReduce Online allows the mappers for the merging job to start merging once the reducers for the sorting job start to produce results.

While the above three types of complex queries have been well studied in the literature, there still exist a broad spectrum of complex analytics queries that have not been solved yet such as k-nearest neighbor join queries. Further research is needed to support more complex analytics queries in distributed systems.

1.2.3 Resilience to Failures

Failure is an inevitable result of involving more and more computers into one system to cope with the increasing scale of data. In distributed systems, each computer is responsible for a subset of computation tasks. When a computer fails, all the tasks executed on that computer fail as well. If the tasks are correlated with each other, one task failure may result in the failures of the tasks that are being executed in healthy computers. In the worst case, all the tasks fail and have to be redone from the beginning.

Resilience to failures is one of the most important requirements for distributed systems. To fulfill this requirement, a distributed system must be able to detect the occurrence of failures automatically. Furthermore, the system must be able to perform recovery immediately upon any failure and return to normal execution after recovery. In practice, failures may occur at any time, either during the normal execution or during recovery, and the latter makes the overall recovery process more complicated.

In many popular distributed systems such as MapReduce, automatic failure detection is achieved by asking all the slave computers (which are responsible for computation tasks) to send heartbeats periodically to one master computer (which manages the collaborations among slaves). Specifically, the master computer will set up a local servlet to check the heartbeats registered by the slaves periodically. If a slave computer does not send its heartbeat within a predefined time period, the servlet will inform the master computer of the slave's failure. Once a slave fails, all the tasks executed in that slave fail as well and the master will ask some healthy slaves to take over (i.e., re-execute) the failed tasks.

Upon the detection of a failure, the system has to recover from the failure by restarting appropriate execution. One of the most popular recovery mechanisms adopted in distributed systems is checkpoint-based recovery. Intuitively, checkpoint-based recovery requires the system to write a consistent state, i.e., checkpoint, to a reliable storage periodically. Whenever a failure occurs, the system will terminate the current execution, reload the latest checkpoint from the reliable storage and resume the execution since then. For example, in MapReduce, if a mapper fails, the system will create a new mapper. The newly created mapper will retrieve the input data that were processed by the failed mapper and then execute the Map function over the retrieved data again. Similarly, if a reducer fails, the system will launch another reducer to substitute the failed one; the new reducer will retrieve the corresponding output from the mappers and execute the Reduce function over the retrieved data again. In MapReduce, the system does not have to make any checkpoint explicitly due to the fact that both the initial input and the intermediate data produced by mappers are materialized to the reliable distributed file system automatically. Other distributed systems such as Pregel have to perform periodical checkpointing explicitly [59].

Checkpoint is the basic foundation of most existing recovery mechanisms [42]. However, performing checkpointing essentially requires the system (i.e., all the computers involved in the system) to pause its on-going execution and materialize all the necessary information such as the computed data and forwarding messages into a reliable storage. During that period, the execution of the job is paused and no progress will be made. Furthermore, the recovery process based on checkpointing may involve high recovery latency. This is because the overall recovery process includes reloading the latest checkpoint from the reliable storage, rolling back the system to the state maintained in the latest checkpoint and redoing all of the lost computations and communications since then. In order to accelerate the recovery process, we need to develop new recovery mechanisms that are more efficient.

1.3 Objective and Contributions

The objective of this thesis is to develop effective and efficient approaches to two challenging issues in distributed systems: complex query processing and fault tolerance. Most specifically, we first focus on answering a complex analytics query, k nearest neighbor join in a distributed manner. We then propose an efficient graph processing engine to handle graph-related analytics queries. Finally, we address the recovery problem in distributed systems. For all the three problems we study, we choose epiC [47] as our underlying distributed system due to its simplicity, efficiency and extensibility, but our approaches can be implemented in other distributed systems as well. We provide more details in the following sections.

1.3.1 k Nearest Neighbor Join

k nearest neighbor (kNN) join is an important primitive operation that serves a wide range of data mining and analytics applications, such as k-means clustering, k-medoids clustering and outlier detection [22, 51]. Given two sets R, S of data objects, kNN join is defined as: for each object o in R, find k objects in S that are closest to o based on a pre-defined distance measure. All the existing approaches solve kNN join problem in a centralized manner and hence suffer from performance deterioration when the size of dataset increases. In this thesis, the first problem we consider is: how to answer kNN join query in a distributed manner?

We leverage MapUnit/ReduceUnit in epiC which is an implementation of MapReduce framework, and propose a MapReduce-based solution to answer

kNN join query for objects under the metric space. In our solution, we exploit the Voronoi-based partitioning method and divide input objects into groups. We design an effective map function which guarantees that similar objects must be gathered and processed by the same reducer (i.e., ReduceUnit). We then answer kNN join query by examining pair-wise objects within the same group. In order to further accelerate the processing, we provide a theoretical analysis of the computation and shuffling cost involved in our approach. Based on the cost model, we introduce a cost-based grouping strategy to balance the workload among the reducers (i.e., ReduceUnits) and introduce several pruning rules to eliminate the examination of dissimilar object pairs.

Contributions. Our proposed method is the first distributed solution for answering kNN join query. Compared with the existing index-based approaches [15, 16], our distributed solution allows us to perform pair-wise examinations for candidate object pairs in parallel, thus accelerating the processing of kNN join query significantly. Furthermore, our cost-based grouping strategy that keeps similar objects together and our proposed pruning rules that keep dissimilar objects apart can be applied to the existing index-based solutions as well.

1.3.2 Efficient Graph Processing Engine

The second problem we address is: how to answer graph-related analytics queries efficiently? As mentioned before, epiC is a disk-based distributed solution for large-scale data analytics. To support iterative computations for graph applications, epiC implements a class called *graphUnit* to handle the computation task of a subgraph. However, current graphUnit-based graph engine has two drawbacks. First, it is not a memory-based solution for iterative graph applications because the graph data will be flushed to the distributed file system at the end of each iteration and then be reloaded into main memory in the beginning of next iteration. As discussed previously, such kind of I/O operations is time-consuming due to the high network cost. Second, in the current design, if one slave computer wants to communicate with another slave, it has to send a message to the master first and the master will forward the message to the corresponding slave. In other words, all the communications among slave computers are coordinated by the master computer and cannot be performed directly by the slave computers. Hence, the master computer will be the bottleneck when the communications among the slave computers are frequent.

To support memory-based iterative computation, we extend epiC and develop a new graph processing engine, called epiCG. epiCG is implemented as an extension of epiC to avoid deploying a new distributed system in the cluster for graph processing. The design of epiCG addresses several challenges. First, in terms of graph partitioning, epiCG supports both edge-cut based and vertex-cut based graph partitioning methods. Edge-cut partitioning method is easier to implement, but vertex-cut partitioning method is known to be more effective in handling power-law graphs. Second, for any given partitioning, epiCG can distribute the input graph among the slave computers efficiently and perform computation and communication effectively during the iterations. Finally, epiCG allows slave computers to send messages among each other instead of communicating via the master computer.

Contributions. epiCG is developed as one extension of epiC, thus allowing users to execute different types of analytics jobs using the same distributed system. epiCG supports both vertex-cut and edge-cut partitioning methods. For vertex-cut, we propose an efficient greedy strategy to parallelize the process of vertex-cut generation. In terms of fault tolerance, epiCG achieves automatic failure detection and recovery. We compare epiCG with two advanced distributed graph processing systems, Giraph [2] and PowerGraph [36]. The results illustrate the high efficiency and scalability of epiCG.

1.3.3 Recovery in Distributed Graph Processing Systems

In the third piece of this thesis, we focus on the recovery issue in epiC/epiCG. The traditional checkpoint-based recovery works well for one-pass jobs such as kNN join, but it always requires long recovery time for graph-related applications executed in epiCG. The reason is two-fold. First, in distributed graph processing systems like epiCG, each computer is responsible for the computation task of a subgraph. Once a computer fails, checkpoint-based recovery requires the whole system to rollback to the latest checkpoint. All the computations finished by the healthy computers are ignored and will be redone since the latest checkpoint. This is wasteful. Second, for the failures that occur during the recovery period, all the partially recovered workload has to be redone since the latest checkpoint as well. If the frequency of failure occurrence is high, checkpoint-based recovery may repeatedly rollback and re-execute the lost computation since the latest checkpoint.

To address the problem, we study the problem of efficient failure recovery in distributed graph processing systems. We first formalize the failure recovery problem in graph processing systems. We then propose a novel partition-based recovery method to parallelize the failure recovery processing. Different from the traditional checkpoint-based recovery approach, our recovery method distributes recovery tasks to multiple computers such that the tasks can be executed concurrently. We prove that it is NP-hard to find a partitioning for the recovery workload such that the total recovery time is minimized. Hence, we provide a communication and computation cost model to estimate the overall recovery time for a given partitioning and propose a greedy algorithm to split the recovery workload among the computers in a cost-effective way. To further accelerate the recovery process, we require every compute node to log their computed results into local disk periodically. Based on the logs, the computations performed by the healthy computers do not need to be redone during recovery.

Contributions. Our work is the first parallel recovery mechanism proposed for distributed graph processing. Our recovery method can handle failures that occur at any time, either during normal execution or during recovery period. To accelerate the recovery process, we eliminate the high computation cost for the subgraphs residing in healthy computers and distribute the recovery tasks for the subgraphs in failed computers to multiple computers. We implement our recovery method in epiCG for performance evaluation. The results show that our partition-based recovery method is efficient and scalable.

1.4 Synopsis of the Thesis

The remainder of this thesis is organized as follows. In Chapter 2, we first review existing techniques in solving kNN join and the advanced distributed graph processing systems proposed to answer graph-related analytics queries. We then present the recovery mechanisms adopted in distributed systems. Chapter 3 studies the problem of efficiently answering kNN join queries in epiC. Chapter 4 introduces our distributed graph engine epiCG on top of epiC. Chapter 5 addresses the problem of efficient failure recovery in epiCG. We conclude this thesis in Chapter 6.

CHAPTER 2

Literature Review

Various techniques have been proposed to address the challenges in distributed systems. On the one hand, efficient algorithms are designed to support both offline data analytics and online query processing. On the other hand, various distributed systems are developed to handle various real-life data-driven applications more effectively. In this chapter, we review the techniques and systems that are closely related to this thesis. In particular, we first introduce the existing methods of answering kNN join query. We then review the advanced distributed systems proposed for efficient graph processing. Finally, we discuss the recovery mechanisms adopted in distributed systems.

2.1 Answering k Nearest Neighbor Join Query

The goal of kNN join is to produce k nearest neighbors of each object in a data set R from another data set S according to a given distance measure. Instead of solving kNN join for a particular distance measure, in this thesis, we consider to perform kNN join for objects under metric space. In what follows, we first introduce the concept of metric space and then discuss the existing solutions to kNN join.

2.1.1 Objects under Metric Space

A metric space is an ordered pair (S, d) where S is a set of objects and d defines the distances between every two objects in S. Formally, the distance function d in a metric space is presented by $d : M \times M \rightarrow \mathbb{R}$, which have the following three properties:

1. (positivity) $d(x, y) \geq 0$ for all $x, y \in S$;
2. (symmetry) $d(x, y) = d(y, c)$ for all $x, y \in S$;
3. (triangle inequality) $d(x, y) + d(y, z) \geq d(x, z)$ for all $x, y, z \in S$.

There exist a number of examples of metric space. For instance, consider a real number set $\mathbb{R}$ with distance function d satisfying $d(x, y) = |y - x|$ for all $x, y \in \mathbb{R}$. It is easy to see $(\mathbb{R}, d)$ is a metric since all the above three conditions hold. Another popular example is the set of n-dimensional objects with the Euclidean distance function $d(x, y) = \sqrt{(x_1 - y_1)^2 + \cdots + (x_n - y_n)^2}$, where $x = (x_1, \cdots, x_n)$ and $y = (y_1, \cdots, y_n)$ are n-dimensional vectors.

2.1.2 Existing Solutions to kNN Join

Existing solutions to kNN join can be categorized into two groups. The first group contains all the centralized solutions in which input data is stored in local disk and kNN join is performed in main memory using a single computer. The second group contains all the distributed solutions to handle data in sheer size and perform kNN join in parallel.

Centralized Solutions

A naïve centralized approach to kNN join is the following. For each data object o in R, we compute its distance to each of the object in S, and maintain a list of size k to store k objects that are closest to o. Clearly, this approach involves high computation cost as it requires to compute distance for every object pair in $R \times S$. Furthermore, if either R or S is too large to be maintained in memory, this approach will incur high I/O cost caused by repeated disk reads of data objects.

To address the problem, various approaches leverage indexing technique to accelerate the processing of kNN join query. Böhm et al. proposed a R-tree based method to answer kNN join query [15, 16]. In their design, the input data objects are first organized and stored into large-sized pages, and each large page will be further partitioned into a set of small-sized pages, each of which is equipped with a secondary R-tree index. Large-sized pages are used to reduce the I/O cost, i.e., more objects can be escaped from retrieval when the algorithm determines that a large page does not contain any kNN result; secondary R-tree indexes within the small-sized pages try to reduce CPU cost to accelerate the processing further. However, R-tree based solutions to kNN query were found to be inefficient when the dimensionality of data objects increases [88].

Xia et al. [90] proposed a grid partitioning based approach named *Gorder* to answer kNN join query. Gorder applies the Principal Components Analysis (PCA) technique on the input data objects and sorts data objects according to their proposed Grid Order. Specifically, each object will be assigned to a grid and objects in close proximity are always assigned to the same grid. After the assignment, Gorder applies the scheduled block nested loop join on the ordered data objects and outputs the final result. Note that it uses block nested loop join for the purpose of reducing both CPU and I/O costs.

Yu et al. [94] proposed *IJoin*, a B^+-tree based method to answer kNN join query for multi/high-dimensional datasets. In their design, two input datasets are first split into disjoint partitions. IJoin then constructs a B^+-tree for the objects in each dataset using the iDistance technique [45, 95]. iDistance helps to efficiently filter far-away candidate pairs during processing. To further reduce CPU and I/O costs, two variants of IJoin were introduced. The first variant eliminates unnecessary disk accesses and distance computations via approximation bounding cube; the second variant indexes high-dimensional data objects by only considering a subset of dimensions rather than all the dimensions.

Yao et al. [93] proposed *Z-KNN*, a Z-order based method to answer kNN join query. They utilized Z-order to map each multi-dimensional data object into a one-dimensional value and provided both approximate and exact solutions to kNN join. Instead of using spatial databases or stand-along systems, Z-KNN relies on relational databases and performs kNN join via primitive SQL operators. In particular, Z-KNN method transforms a kNN join query into a set of kNN search operations by considering each object in R as a query point. An important advantage of this solution is that no changes are required to be made to the underlying database engine.

All of the above index-based solutions perform kNN join in a single-threaded manner. Another set of centralized solutions to kNN join rely on multiprocessor environment to parallelize the join process [23, 69]. Brinkhoff et al. [23] focused on two-dimensional data objects and exploited R-tree for efficient processing of spatial join. Several optimizations including tunning the parameters for the R-tree and better buffer management are provided to reduce CPU and I/O costs. In [69], the authors first de-clustered spatial data and then stored them into a parallel database system for querying. They proposed various spatial join algorithms for different de-clustering methods. However, none of the parallel algorithms can be easily adapted to handle kNN join for the data objects that are physically distributed across several computers. Furthermore, most parallel algorithms focus on performing kNN join over two-dimensional data objects and the proposed optimized solutions are inappropriate to solve kNN join for multi-dimensional objects.

Distributed Solutions

Recently, more attention has been paid to perform join operation in a distributed environment. Zhang et al. [98] studied the problem of solving spatial join using MapReduce and provided an implementation for the Map and Reduce functions. However, their approach cannot be adapted to handle kNN join.

In [96], the authors answered kNN join query using MapReduce. They first provided a basic solution using block nested loop join. Specifically, in the mapper side, similar objects are forwarded to the same reducer, while dissimilar objects are forwarded to different reducers. In the reducer side, pairwise distance computations are performed to produce the final results. The drawback of this basic solution is the high shuffling cost from mappers to reducers due to the fact that each data object will be forwarded to multiple reducers. To address the problem, they provided another efficient MapReduce algorithm by transforming multi-dimensional data objects into one-dimensional z-values. While the second approach shows high efficiency, it is an approximate algorithm, i.e., the produced results may not be the exact kNN join results.

In this thesis, we focus on answering exact kNN join query for multi-dimensional data objects under metric space. We choose epiC as our underlying distributed system due to its efficiency, simplicity and scalability.

2.2 Advanced Distributed Graph Processing Systems

Distributed graph processing systems (DGPS) are proposed to handle iterative graph computation and machine learning tasks such as PageRank, shortest path computing and connected component computation. Existing distributed graph processing systems can be categorized into two groups based on their computation models. One category is known as synchronous graph processing systems, where all the computers will reach a synchronous barrier at the end of each iteration. Another category is known as asynchronous graph processing systems, in which no global synchronization is required. We now review the existing distributed graph processing systems in each of the categories.

2.2.1 Synchronous Graph Processing

The most famous example of synchronous graph processing systems is Pregel [59]. Pregel follows the Bulk Synchronization Parallel (BSP) model [83]. Typically, the execution of a Pregel job consists of an initialization phase where the graph is distributed among compute nodes, followed by several iterations of computations, called *supersteps*, and finally an output phase to flush out the results. Every vertex carries two states: active and inactive. At the beginning of the job, every vertex is active. During the computations, a vertex can inactivate itself by *voltToHalt* and it will be automatically activated upon receiving a new incoming message. The job terminates when all the vertices become inactive or the number of supersteps reach a predefined upper bound. During each superstep, every active vertex can process messages sent by other vertices in the previous superstep, execute *compute* function, update its value and forward messages to other vertices (to be processed in the next superstep). A global synchronization will be performed at the end of each superstep.

The architecture of Pregel is a master-slave architecture. In particular, each of the slaves is responsible for performing computation for a subgraph and forwarding messages to other slaves, or local aggregation values to the master; the master is responsible for checking the health status of every slave, synchronize the supersteps and performing global aggregation based on the local aggregation values sent by the slaves.

Since the introduction of Pregel, various synchronous graph processing systems have been developed including Giraph [2], GPS [75], Trinity [76], GraphX [91] and Hama [5]. Most of them follow similar or exactly the same computation model as Pregel, hence each of them can be considered an implementation of Pregel.

Compared with MapReduce, Pregel achieves high performance in iterative graph computations. However, a Pregel job may also require long execution time due to the frequent communications among inter-computer vertices. To address the problem, several optimizations are proposed to improve the performance. Tian et al. [82] introduced the idea of "thinking as a graph". Instead of designing compute function for each vertex, they adopted a subgraph-based computation unit and allowed the programmers to define compute function over a subgraph. Basically, in each iteration, computations are first performed within every subgraph *internally* and after that, subgraphs exchange updated information among each other via message forwarding. Such kind of graph-centric programming model allows asynchronous computation in different subgraphs, thus accelerating convergence rates for many graph applications. However, designing compute function for subgraphs introduces extra implementation cost and would be difficult for some complex graph applications.

Pregelix [4] addresses the problem of out-of-core graph computation, i.e., the graph size is too large to be loaded into the main memory of the computers. Pregelix leverages the existing data-parallel platform, Hyracks [20], as its underlying execution engine and supports the full Pregel API by carefully using the operations provided by Hyracks.

Unlike Pregel, asynchronous graph processing systems are developed to perform iterations without doing synchronization, as we will introduce in the next section.

2.2.2 Asynchronous Graph Processing

Distributed GraphLab [55] adopts asynchronous computation model for iterative machine learning and graph applications. Distributed GraphLab consists of three main parts: data graph, update function and the sync operation. Data graph defines mutable data structure; update function defines computation operations to be applied to the data graph; sync operation maintains global aggregation values concurrently. In each iteration, GraphLab loops over the vertices in the data graph and executes the update function for each vertex. The overall iterative computation is performed in parallel, using a cluster of computers. GraphLab performs asynchronous processing by allowing multiple distributed processes to execute update function over different data graphs simultaneously without global synchronization. To ensure that overlapping computations are not conducted simultaneously, GraphLab introduces several consistency models to optimize the parallel computation while maintaining the serialization of the execution.

Later on, PowerGraph [53] was proposed to handle iterative computation. PowerGraph follows the same asynchronous computation model as distributed GraphLab, but focuses on dealing with power-law graphs. Processing power-law graphs is challenging. The main reason is that vertices with high fan-outs have a large number of neighbors that are distributed among all the computers in the cluster; forwarding messages to all these neighbors will incur high network cost. To address the problem, PowerGraph adopts vertex-cut graph partitioning rather than edge-cut graph partitioning. In vertex-cut, when a vertex wants to forward the same message to several neighbors that reside in another computer, it first sends the message to its *mirror* on that computer and the mirror will forward the message to the corresponding neighbors. By doing this, PowerGraph eliminates high communication cost caused by forwarding messages for the vertices having high fan-outs.

Compared with synchronous graph processing systems, asynchronous graph processing systems are more efficient due to the elimination of synchronization cost, but require complex concurrency control to ensure serializability. In this thesis, we focus on synchronous graph processing and leave asynchronous graph processing as our future work.

2.3 Recovery Mechanisms in Distributed Systems

One advantage of distributed systems is the higher computation power achieved by the collaboration of a cluster of computers. However, the increasing number of computers will inevitably increase the probability of the occurrences of failures. Various types of failures may occur in distributed systems, including software anomalies, hardware failures and network failures. Moreover, the failure of one computer may affect other computers because the computation tasks distributed among the computers can be correlated with each other. In the worst case, all the computers in the distributed system may need to redo all the computation tasks since the beginning in order to recover from the failure of a single computer. Hence, resilience to failures is one of the most important requirements of distributed systems. In this section, we first discuss different failure models proposed for distributed systems and then review the existing recovery methods adopted in distributed systems.

2.3.1 Modeling Failures

Typically, a distributed system consists of two components: a fixed number of processes, and communication links that transfer information from one process to another. Consider, for example, a Pregel-like distributed system, where the processes correspond to the slave computers which perform graph computation in parallel, and the communication links correspond to the messages forwarded from one slave to another. Failures may occur in both processes and communication links. In essence, process failures will lead to a certain amount of computation workload being lost and the failures in communication links prevent processes from communicating or communicating correctly.

Failures that occur in processes can be modeled in different ways as follows [77].

- Fail-stop: a process fails by stopping.
- Arbitrary(malicious, byzantine): a process fails without stopping. Specifically, a failed process may show an unexpected behavior that might be malicious and disruptive.

Failure models for communication links are the following [77].

- Send-omission: a process completes a send operation, but the information does not appear in the outgoing buffer.
- Receive-omission: information is put into a process's incoming buffer, but the process does not receive it.

- Channel-omission: information is lost during transmission.

In addition to the above models, failures in distributed systems can also be modeled by their types, say, software fault and hardware fault. Software fault can be further categorized into deadlock, protection fault, dividing by zero, etc; hardware fault can be caused by hard disk failures, input and output device failures, etc.

In this thesis, we focus on the failures of computers rather than analyze a particular failure type. We consider fail-stop failure model. That is, each computer either works properly or fails by stopping. Note that when a computer fails, all the computation tasks executed by that computer fail as well. Furthermore, we mainly consider the scenario where only a subset of computers in the cluster have failed while others are working properly. We refer to this as *partial failures*. In contrast, the scenario that all the computers in the distributed system have failed is referred to as *full failures*, which rarely happens in practice.

2.3.2 Failure Recovery

The objectives [79] of failure recovery in distributed systems include:

- application transparency: the recovery process should be transparent to the upper-level applications.
- application independence: the recovery mechanism should be able to recover the execution of arbitrary upper-level applications.
- high throughput: all the CPU resources should be available when there is no failure.
- maximal fault-tolerance: the recovery mechanism should be able to recover from the failure of any number of processes in the distributed system.
- low latency: the execution of recovery algorithm should be efficient.

With different objectives of failure recovery, the adoption of recovery mechanisms varies from one distributed system to another. In general, recovery methods used in the existing distributed systems fall into three categories:

replication-based recovery, rollback recovery and roll-forward recovery. The details are provided in the following.

- Replication-based recovery. Replication is a basic recovery method used in distributed systems, especially the real-time distributed systems [70]. When a failure occurs in a certain system component (e.g., computer), one replication of the failed component will take over the responsibility and the system can continue its processing. Typically, there are two types of replications.

  - Data replication: store data in multiple places/devices to make data available during processing.
  - Computation replication: execute the same computing task on separate devices.

  Replication-based recovery must ensure that all the replications of a component will change synchronously with the original one [39]. To fulfill this requirement, there are two fundamental replication techniques. The first technique is primary-backup replication, where one of the replicas is called the primary and the others are called the backups [26]. The primary is responsible for receiving information, performing computation and forwarding information to other primaries, and every backup will communicate with its primary. When a primary fails, one of its backups will be selected as the primary and handle the incoming invocations accordingly. The second technique is called active replication. In active replication, all the replicas for a particular component play the same role, i.e., all of them will handle the same sequences of invocations. In contrast to primary-backup replication, all the replicas for the same component in active replication are pair-wise equivalent. While replication-based recovery provides high availability of the system upon a failure, it is worth noting that it incurs high network cost for synchronizing all the replicas periodically.

- Rollback recovery [33]. Traditional rollback recovery treats a distributed system as a collection of processes that communicate between each other via network messages. There are two important types of rollback recovery: checkpoint-based and log-based. In checkpoint-based rollback recovery, all the processes will save their states to a stable storage periodically, either synchronously or asynchronously, during normal execution. Each of the saved states is called a *local checkpoint*. Upon a failure, a failed process will restart from one of its saved states. Furthermore, any process that is dependent on the failed one has to rollback to one of its local checkpoint as well to preserve global consistency. In log-based rollback recovery, every process will write logs (e.g., the executed commands) to a stable storage. During recovery, a failed process will retrieve its logs and replay the lost workload based on the logs. In practice, log-based rollback recovery is always combined with checkpointing method to reduce the log size. There are several protocols proposed for log-based rollback recovery, including pessimistic log-based recovery protocol, optimistic log-based recovery protocol, causal log-based recovery protocol and optimal log-based recovery protocol. Each of them has its own trade-off between runtime efficiency and recovery efficiency.

- Roll-forward recovery. Roll-forward recovery aims to reduce the amount of rollback upon a failure, i.e., quickly correct system state in the presence of failures and resume normal execution. The first work on roll-forward recovery [40] proposed a protocol based on synchronous and incremental checkpoint to maximize roll-forward. Later on, many optimizations [29, 92] were introduced to improve the performance of roll-forward recovery in real systems.

Recovery in Distributed Graph Processing Systems

Designing efficient failure recovery methods has long been a goal of distributed systems. In this thesis, we study the problem of failure recovery in our graph processing engine, epiCG. Our solution can be implemented in other Pregel-like distributed graph processing systems as well. Many existing distributed graph processing systems including Giraph [2], GraphLab [57], PowerGraph [36], GPS [75], Mizan [50] adopt checkpoint-based rollback recovery. GraphX [91] adopts log (called lineage) based recovery, and utilizes the resilient distributed datasets (RDD) to speedup failure recovery. Pregel [59] proposes confined recovery which is a hybrid mechanism of the checkpoint-based and log-based recovery. Specifically, only the newly-added computer that substitutes for the failed one will rollback and redo the computation since the latest checkpoint. One problem of confined recovery is that the recovery workload is performed in a centralized way. During recovery, most healthy computers become idle but wait for the recovery process to finish.

Location and replication independent recovery proposed by Bratsberg et al. employed replicas for recovery [21]. The algorithm partitions the data into fragments and replicates fragments among multiple computers which can takeover in parallel upon failures. However, recovery task for the failed computer is still performed in a centralized manner after the computer finishes internal recovery.

Another recovery method that presents similarities with our proposed solution is introduced by RAMCloud [66]. RAMCloud backs up the data across many distributed computers, and during recovery, it reconstructs in parallel the lost data. However, as RAMCloud is a distributed storage, it does not need to track the dependencies among the scattered data. In contrast, in distributed processing systems, understanding how the program dependencies affect both the communication and the computation time is of utmost importance [10].

Since checkpointing and logging operations are the backbone of recovery methods [13, 18, 19], accelerating them has long been a target for optimizations [33]. Most previous work focuses on accelerating the process of storing the information required for recovery, including incremental checkpointing [63] and command log [60].

2.3.3 Summary

In this chapter, we first reviewed the existing work on complex query processing in distributed systems. We presented the techniques on answering two important complex queries, k-nearest neighbor join queries and graph-related analytics queries. For kNN join queries, most of the work perform kNN join using a single computer [15, 16, 90, 94, 93]. The recent distributed solution [96] is an approximate algorithm that may not produce the exact kNN join results. For graph-related analytics queries, various graph processing distributed systems have been proposed such as Pregel [59] and GraphLab [57]. However, all of these graph processing systems require their own configurations from scratch, which is always a daunting job. We then reviewed related work on fault tolerance in distributed systems. We introduced the failure models and various recovery mechanisms in distributed systems. While most distributed systems adopt checkpoint-based recovery due to its simplicity, the recovery process based on checkpointing may involve high recovery latency as all the computers in the cluster have to rollback to the latest checkpoint and redo all the lost computation.

CHAPTER 3

kNN Join using MapReduce Framework

k nearest neighbor join (kNN join), designed to find k nearest neighbors from a dataset S for every object in another dataset R, is a primitive operation widely adopted by many data mining applications. As a combination of the k nearest neighbor query and the join operation, kNN join is an expensive operation. Given the increasing volume of data, it is difficult to perform a kNN join on a centralized machine efficiently. In this chapter, we investigate how to perform kNN join using MapReduce which is a well-accepted framework for data-intensive applications over clusters of computers. In brief, the mappers cluster objects into groups; the reducers perform the kNN join on each group of objects separately. We design an effective mapping mechanism that exploits pruning rules for distance filtering, and hence reduces both the shuffling and computational costs. To reduce the shuffling cost, we propose two approximate algorithms to minimize the number of replicas. We implement our apporach by leveraging MapUnits/ReduceUnit in epiC, which is an implementation of MapReduce framework. Extensive experiments on our in-house cluster demonstrate that our proposed methods are efficient, robust and scalable.

3.1 Introduction

k nearest neighbor join (kNN join) is a special type of join that combines each object in a dataset R with the k objects in another dataset S that are closest to it. kNN join typically serves as a primitive operation and is widely used in many data mining and analytic applications, such as the k-means and k-medoids clustering and outlier detection [22, 51].

As a combination of the k nearest neighbor (kNN) query and the join operation, kNN join is an expensive operation. The naive implementation of kNN join requires scanning S once for each object in R (computing the distance between each pair of objects from R and S), easily leading to a complexity of $O(|R| \cdot |S|)$. Therefore, considerable research efforts have been made to improve the efficiency of the kNN join [17, 90, 94, 93]. Most of the existing work devotes themselves to the design of elegant indexing techniques for avoiding scanning the whole dataset repeatedly and for pruning as many distance computations as possible.

All the existing work [17, 90, 94, 93] is proposed based on the centralized paradigm where the kNN join is performed on a single, centralized server. However, given the limited computational capability and storage of a single machine, the system will eventually suffer from performance deterioration as the size of the dataset increases, especially for multi-dimensional datasets. The cost of computing the distance between objects increases with the number of dimensions; and the curse of the dimensionality leads to a decline in the pruning power of the indexes.

Regarding the limitation of a single machine, a natural solution is to consider parallelism in a distributed computational environment. MapReduce [31] is a programming framework for processing large scale datasets by exploiting the parallelism among a cluster of computing nodes. Soon after its birth, MapReduce gains popularity for its simplicity, flexibility, fault tolerance and scalability. MapReduce is now well studied [48] and widely used in both commercial and scientific applications. Therefore, MapReduce becomes an ideal framework of processing kNN join operations over massive, multi-dimensional datasets.

However, existing techniques of kNN join cannot be applied or extended to be incorporated into MapReduce easily. Most of the existing work rely on some centralized indexing structure such as the B$^+$-tree [94] and the R-tree [17], which cannot be accommodated in such a distributed and parallel environment directly.

In this chapter, we investigate the problem of implementing kNN join operator by leveraging MapUnit/ReduceUnit in epiC, which is an implementation of MapReduce framework. The basic idea is similar to the hash join algorithm. Specifically, the mapper (i.e., MapUnit) assigns a key to each object from R and S; the objects with the same key are distributed to the same reducer (i.e., ReduceUnit) in the shuffling process; the reducer performs the kNN join over the objects that have been shuffled to it. To guarantee the correctness of the join result, one basic requirement of data partitioning is that for each object r in R, the k nearest neighbors of r in S should be sent to the same reducer as r does, i.e., the k nearest neighbors should be assigned with the same key as r. As a result, objects in S may be replicated and distributed to multiple reducers. The existence of replicas leads to a high shuffling cost and also increases the computational cost of the join operation within a reducer. Hence, a good mapping function that minimizes the number of replicas is one of the most critical factors that affect the performance of the kNN join using MapReduce.

In particular, we summarize the contributions of this chapter as follows.

- We present an implementation of kNN join operator using MapReduce, especially for large volume of multi-dimensional data. The implementation defines the mapper and reducer jobs and requires no modifications to the MapReduce framework.

- We design an efficient mapping method that divides objects into groups, each of which is processed by a reducer to perform the kNN join. First, the objects are divided into partitions based on a Voronoi diagram with carefully selected pivots. Then, data partitions (i.e., Voronoi cells) are clustered into groups only if the distances between them are restricted by a specific bound. We derive a distance bound that leads to groups of objects that are more closely involved in the kNN join.

- We derive a cost model for computing the number of replicas generated in the shuffling process. Based on the cost model, we propose two grouping strategies that can reduce the number of replicas greedily.

- We implement our approach by leveraging MapUnit/ReduceUnit in epiC, whic is an implementation of MapReduce framework. We conduct extensive experiments to study the effect of various parameters using two real datasets and some synthetic datasets. The results show that our proposed methods are efficient, robust, and scalable.

3.2 Preliminaries

In this section, we first define kNN join formally and then describe Voronoi diagram-based partitioning, which is a basis of our kNN join approach. Finally, we provide a brief review of MapReduce framework. Table 3.1 lists the symbols and their meanings used throughout this chapter.

3.2.1 kNN Join

We consider data objects in an n-dimensional metric space $\mathcal{D}$. In general, a metric space $\mathcal{D}$ is a set of objects that has a notion of the distance between every two objects. Given two data objects r and s, let $|r, s|$ represent the distance between r and s in $\mathcal{D}$. For any objects $r, s, t \in \mathcal{D}$, the distance measure $|\cdot, \cdot|$ satisfies the following four conditions:

1. (non-negativity) $|r, s| \geq 0$
2. (positive-definiteness) $|r, s| = 0$ iff $r = s$
3. (symmetry) $|r, s| = |s, r|$
4. (triangle inequality) $|r, s| \leq |r, t| + |t, s|$

Without loss of generality, we use the Euclidean distance (L_2) as our distance measure, i.e.,

$$|r, s| = \sqrt{\sum_{1 \leq i \leq n} (r[i] - s[i])^2}, \tag{3.1}$$

where $r[i]$ (resp. $s[i]$) denotes the value of r (resp. s) along the i^{th} dimension in $\mathcal{D}$. Our methods can be easily applied to other distance measures such as the Manhattan distance (L_1), and the maximum distance (L_∞).

Definition 3.1 (k nearest neighbors)**.** *Given an object r, a dataset S and an integer k, the k nearest neighbors of r from S, denoted as $KNN(r, S)$, is a set of k objects from S that $\forall o \in KNN(r, S), \forall s \in S - KNN(r, S)$, $|o, r| \leq |s, r|$.*

Table 3.1: Notations used throughout Chapter 3

Symbol	Definition
$\mathcal{D}$	an n-dimensional metric space
R (resp. S)	an object set R (resp. S) in $\mathcal{D}$
r (resp. s)	an object, $r \in R$ (resp. $s \in S$)
$\|r, s\|$	the distance from r to s
k	the number of near neighbors
$KNN(r, S)$	the k nearest neighbors of r from S
$R \ltimes S$	kNN join of R and S
$\mathbb{P}$	a set of pivots
p_i	a pivot in $\mathbb{P}$
p_r	the pivot in $\mathbb{P}$ that is closest to r
P_i^R	the partition of R that corresponds to p_i
$p_i.d_j$	the j^{th} smallest distance of objects in P_i^S to p_i
$U(P_i^R)$	$\max\{\|r, p\| \| \forall r \in P_i^R\}$
$L(P_i^R)$	$\min\{\|r, p\| \| \forall r \in P_i^R\}$
T_R	the summary table for partitions in R
N	the number of reducers

Definition 3.2 (kNN join). *Given two datasets R and S and an integer k, kNN join of R and S (denoted as $R \ltimes_{KNN} S$, abbreviated as $R \ltimes S$), combines each object $r \in R$ with its k nearest neighbors from S. Formally,*

$$R \ltimes S = \{(r, s) \mid \forall r \in R, \forall s \in KNN(r, S)\} \tag{3.2}$$

According to Definition 3.2, $R \ltimes S$ is a subset of $R \times S$. Note that kNN join operation is asymmetric, i.e., $R \ltimes S \neq S \ltimes R$. Given $k \leq |S|$, the cardinality of $|R \ltimes S|$ is $k \times |R|$. We assume that $k \leq |S|$. Otherwise, kNN join degrades to the cross join and just generates the result of Cartesian product $R \times S$.

3.2.2 Voronoi Diagram-based Partitioning

Given a dataset O, the main idea of Voronoi diagram-based partitioning is to select M objects (which may not belong to O) as pivots, and then split objects of O into M disjoint partitions where each object is assigned to the partition with its closest pivot [1]. In this way, the whole data space is split into M

[1] In particular, if there exist multiple pivots that are closest to an object, then the object is assigned to the partition with the smallest number of objects.

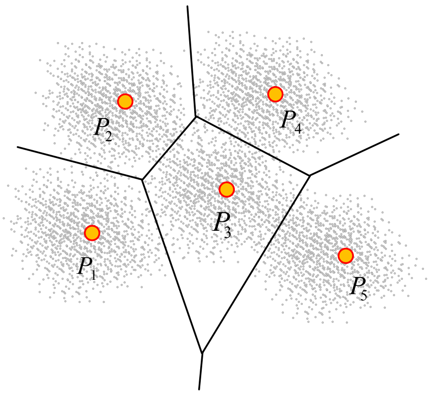


Figure 3.1: An example of data partitioning

"generalized Voronoi cells". Figure 3.1 shows an example of splitting objects into 5 partitions by employing the Voronoi diagram-based partitioning. For the sake of brevity, let $\mathbb{P}$ be the set of pivots selected. $\forall p_i \in \mathbb{P}$, P_i^O denotes the set of objects from O that takes p_i as their closest pivot. For an object o, let p_o and P_o^O be its closest pivot and the corresponding partition respectively. In addition, we use $U(P_i^O)$ and $L(P_i^O)$ to denote the maximum and minimum distance from pivot p_i to the objects of P_i^O, i.e., $U(P_i^O) = \max\{|o, p_i| | \forall o \in P_i^O\}$, $L(P_i^O) = \min\{|o, p_i| | \forall o \in P_i^O\}$.

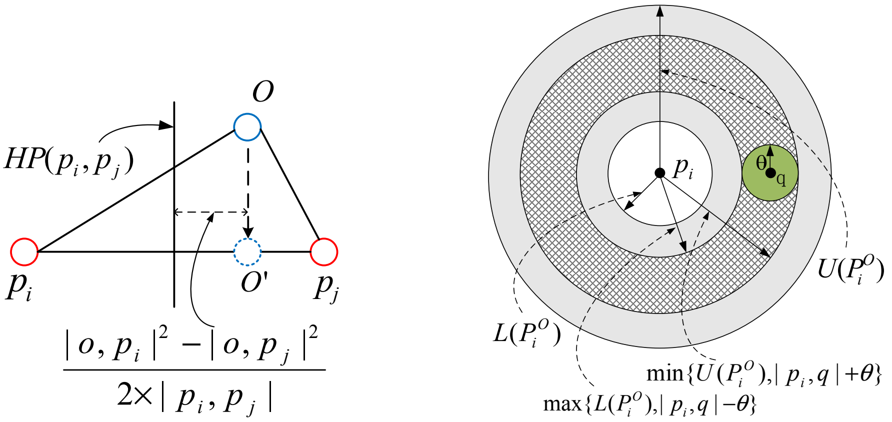
Definition 3.3 (Range selection)**.** *Given a dataset O, an object q, and a distance threshold θ, range selection of q from O is to find all objects (denoted as $\bar{O}$) of O, such that $\forall o \in \bar{O}, |q, o| \leq \theta$.*

By splitting the dataset into a set of partitions, we can answer range selection queries based on the following theorem.

Theorem 3.1. [43] *Given two pivots p_i, p_j, let $HP(p_i, p_j)$ be the generalized hyperplane, where any object o lying on $HP(p_i, p_j)$ has the equal distance to p_i and p_j. $\forall o \in P_j^O$, the distance of o to $HP(p_i, p_j)$, denoted as $d(o, HP(p_i, p_j))$ is:*

$$d(o, HP(p_i, p_j)) = \frac{|o, p_i|^2 - |o, p_j|^2}{2 \times |p_i, p_j|} \tag{3.3}$$

Figure 3.2(a) shows distance $d(o, HP(p_i, p_j))$. Given object q, its belonging partition P_q^O, and another partition P_i^O, according to Theorem 3.1, it is able to compute the distance from q to $HP(p_q, p_i)$. Hence, we can derive the following corollary.


O
HP(p_i, p_j)
p_i
O'
p_j
$\frac{|o,p_i|^2 - |o,p_j|^2}{2\times|p_i,p_j|}$
p_i
θ
q
U(P_i^O)
L(P_i^O)
min{U(P_i^O), |p_i, q| + θ}
max{L(P_i^O), |p_i, q| − θ}

(a) $d(o, HP(p_i, p_j))$

(b) bounding area of Equation 3.4

Figure 3.2: Properties of data partitioning

Corollary 3.1. *Given a partition P_i^O and $P_i^O \neq P_q^O$, if we can derive $d(q, HP(p_q, p_i))$ $> \theta$, then $\forall o \in P_i^O$, $|q, o| > \theta$.*

Given a partition P_i^O, if we get $d(q, HP(p_q, p_i)) > \theta$, according to Corollary 3.1, we can discard all objects of P_i^O. Otherwise, we check partial objects of P_i^O based on Theorem 3.2.

Theorem 3.2. [45, 95] *Given a partition P_i^O, $\forall o \in P_i^O$, the necessary condition that $|q, o| \leq \theta$ is:*

$$\max\{L(P_i^O), |p_i, q| - \theta\} \leq |p_i, o| \leq \min\{U(P_i^O), |p_i, q| + \theta\} \qquad (3.4)$$

Figure 3.2(b) shows an example of the bounding area of Equation 3.4. To answer range selections, we only need to check objects that lie in the bounding area of each partition.

3.2.3 MapReduce Framework and epiC

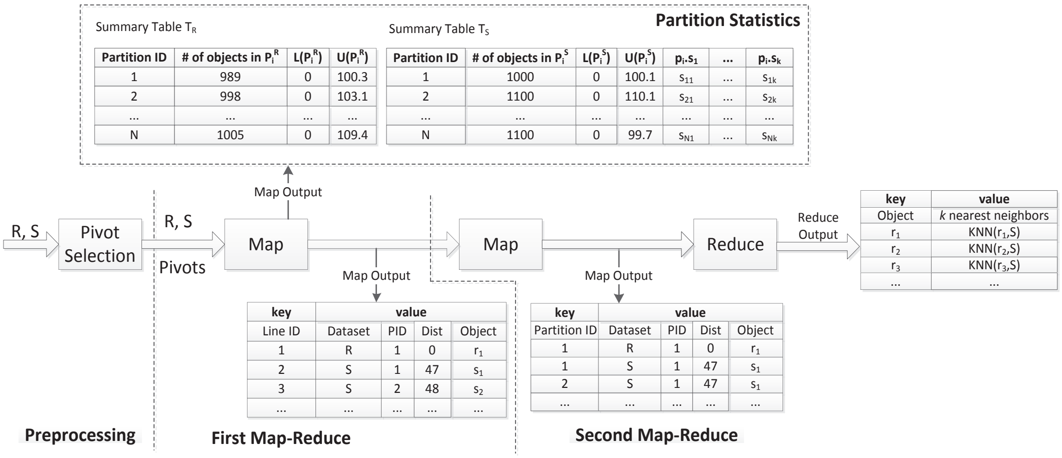
MapReduce [31] is a popular programming framework to support data-intensive applications using shared-nothing clusters. In MapReduce, input data are represented as key-value pairs. Several functional programming primitives including Map and Reduce are introduced to process the data. Map function takes an input key-value pair and produces a set of intermediate key-value pairs. MapReduce runtime system then groups and sorts all the intermediate values associated with the same intermediate key, and sends them to the Reduce function. Reduce function accepts an intermediate key and its corresponding values, applies the processing logic, and produces the final result which is typically a list of values.

epiC provides an implementation of MapReduce framework. In epiC, mappers are referred to as MapUnits and reducers are referred to as ReduceUnits. In this chapter, we may use all these notations alternatively. Data in epiC are stored in HDFS by default. HDFS consists of multiple DataNodes for storing data and a master node called NameNode for monitoring DataNodes and maintaining all the meta-data. In HDFS, imported data will be split into equal-size chunks, and the NameNode allocates the data chunks to different DataNodes. The epiC runtime system establishes two processes, namely MasterNode and WorkerNode. The MasterNode splits a submitted job into MapUnit and ReduceUnit tasks and schedules the tasks among all the available WorkerNodes. WorkerNodes will accept and process the assigned MapUnit/ReduceUnit tasks. For a MapUnit task, the WorkerNode takes a data chunk from HDFS and applies the `map()` function. When all the `map()` functions complete, the epiC runtime groups all the intermediate results and launches a number of ReduceUnit tasks to run the `reduce()` function and produce the final results. Both `map()` and `reduce()` functions are specified by the user.

3.3 An Overview of kNN Join Using MapReduce

In MapReduce, the mappers produce key-value pairs based on the input data; each reducer performs a specific task on a group of pairs with the same key. In essence, the mappers do something similar to (typically more than) the hashing function. A naive and straightforward idea of performing kNN join in MapReduce is similar to the hash join algorithm.

Specifically, the `map()` function assigns each object $r \in R$ a key; based on the key, R is split into disjoint subsets, i.e., $R = \bigcup_{1 \leq i \leq N} R_i$, where $R_i \bigcap R_j = \emptyset, i \neq j$; each subset R_i is distributed to a reducer. Without any pruning rule, the entire set S has to be sent to each reducer to be joined with R_i; finally $R \ltimes S = \bigcup_{1 \leq i \leq N} R_i \ltimes S$.



Partition ID	# of objects in P_i^R	$L(P_i^R)$	$U(P_i^R)$
1	989	0	100.3
2	998	0	103.1
...	...	...	...
N	1005	0	109.4

Partition ID	# of objects in P_i^S	$L(P_i^S)$	$U(P_i^S)$	$p_i.s_1$	...	$p_i.s_k$
1	1000	0	100.1	s_{11}	...	s_{1k}
2	1100	0	110.1	s_{21}	...	s_{2k}
...	...	...	...	...	...	...
N	1100	0	99.7	s_{N1}	...	s_{Nk}

key	value			
Line ID	Dataset	PID	Dist	Object
1	R	1	0	r_1
2	S	1	47	s_1
3	S	2	48	s_2
...	...	...	...	...

key	value			
Partition ID	Dataset	PID	Dist	Object
1	R	1	0	r_1
1	S	1	47	s_1
2	S	1	47	s_1
...	...	...	...	...

key	value
Object	*k* nearest neighbors
r_1	KNN(r_1,S)
r_2	KNN(r_2,S)
r_3	KNN(r_3,S)
...	...

Figure 3.3: An overview of kNN join in MapReduce

In this scenario, there are two major considerations that affect the performance of the entire join process.

1. The shuffling cost of sending intermediate results from mappers to reducers.

2. The cost of performing the kNN join on the reducers.

Obviously, the basic strategy is too expensive. Each reducer performs kNN join between a subset of R and the entire S. Given a large population of S, it may go beyond the capability of the reducer. An alternative framework [96], called H-BRJ, splits both R and S into $\sqrt{N}$ disjoint subsets, i.e., $R = \bigcup_{1 \leq i \leq \sqrt{N}} R_i$, $S = \bigcup_{1 \leq j \leq \sqrt{N}} S_j$. Similarly, the partitioning of R and S in H-BRJ is performed by the `map()` function; a reducer performs the kNN join between a pair of subsets R_i and S_j; finally, the join results of all pairs of subsets are merged and $R \ltimes S = \bigcup_{1 \leq i,j \leq \sqrt{N}} R_i \ltimes S_j$. In H-BRJ, R and S are partitioned into equal-sized subsets on a random basis.

While the basic strategy can produce the join result using one MapReduce job, H-BRJ requires two MapReduce jobs. Since the set S is partitioned into several subsets, the join result of the first reducer is incomplete, and another MapReduce is required to combine the results of $R_i \ltimes S_j$ for all $1 \leq j \leq \sqrt{N}$. Therefore, the shuffling cost of H-BRJ is $\sqrt{N} \cdot (|R| + |S|) + \sum_i \sum_j |R_i \ltimes S_j|$[2],

[2] $\sqrt{N} \cdot (|R| + |S|)$ is the shuffling cost of the first MapReduce. $\sum_i \sum_j |R_i \ltimes S_j|$ is the shuffling cost of the second MapReduce for merging the partial results.

while for the basic strategy, it is $|R| + N \cdot |S|$.

In order to reduce the shuffling cost, a better strategy is that R is partitioned into N disjoint subsets and for each subset R_i, find a subset of S_i that $R_i \ltimes S = R_i \ltimes S_i$ and $R \ltimes S = \bigcup_{1 \leq i \leq N} R_i \ltimes S_j$. Then, instead of sending the entire S to each reducer (as in the basic strategy) or sending each R_i to $\sqrt{N}$ reducers, S_i is sent to the reducer that R_i belongs to and the kNN join is performed between R_i and S_i only.

This approach avoids replication on the set R and sending the entire set S to all reducers. However, to guarantee the correctness of the kNN join, the subset S_i must contain the k nearest neighbors of every $r \in R_i$, i.e., $\forall r \in R_i, KNN(r, S) \subseteq S_i$. Note that $S_i \cap S_j$ may not be empty, as it is possible that object s is one of the k nearest neighbors of $r_i \in R_i$ and $r_j \in R_j$. Hence, some of the objects in S should be replicated and distributed to multiple reducers. The shuffling cost is $|R| + \alpha \cdot |S|$, where α is the average number of replicas of an object in S. Apparently, if we can reduce the value of α, both shuffling and computational cost we consider can be reduced.

In summary, for the purpose of minimizing the join cost, we need to

1. find a good partitioning of R;

2. find the minimal set of S_i for each $R_i \in R$, given a partitioning of R [3].

Intuitively, a good partitioning of R should cluster objects in R based on their proximity, so that the objects in a subset R_i are more likely to share common k nearest neighbors from S. For each R_i, the objects in each corresponding S_i are cohesive, leading to a smaller size of S_i. Therefore, such partitioning not only leads to a lower shuffling cost, but also reduces the computational cost of performing the kNN join between each R_i and S_i, i.e., the number of distance calculations.

3.4 Handling kNN Join Using MapReduce

In this section, we introduce our implementation of kNN join using MapReduce. First, Figure 3.3 illustrates the working flow of our kNN join, which consists of one preprocessing step and two MapReduce jobs.

[3] The minimum set of S_i is $S_i = \bigcup_{1 \leq j \leq |R_i|} KNN(r_i, S)$. However, it is impossible to find out the k nearest neighbors for all r_i apriori.

- First, the preprocessing step finds out a set of pivot objects based on the input dataset R. The pivots are used to create a Voronoi diagram, which can help partition objects in R effectively while preserving their proximity.

- The first MapReduce job consists of a single Map phase, which takes the selected pivots and datasets R and S as the input. It finds out the nearest pivot for each object in $R \cup S$ and computes the distance between the object and the pivot. The result of the mapping phase is a partitioning on R, based on the Voronoi diagram of the pivots. Meanwhile, the mappers also collect some statistics about each partition R_i.

- Given the partitioning on R, mappers of the second MapReduce job find the subset S_i of S for each subset R_i based on the statistics collected in the first MapReduce job. Finally, each reducer performs the kNN join between a pair of R_i and S_i received from the mappers.

3.4.1 Data Preprocessing

As mentioned in previous section, a good partitioning of R for optimizing kNN join should cluster objects based on their proximity. We adopt the Voronoi diagram-based data partitioning technique as reviewed in Section 3.2, which is well-known for maintaining data proximity, especially for data in multi-dimensional space. Therefore, before launching the MapReduce jobs, a preprocessing step is invoked in a master node for selecting a set of pivots to be used for Voronoi diagram-based partitioning. In particular, the following three strategies can be employed to select pivots.

- **Random Selection.** First, T random sets of objects are selected from R. Then, for each set, we compute the total sum of the distances between every two objects. Finally, the objects from the set with the maximum total sum distance are selected as the pivots for data partitioning.

- **Farthest Selection.** The set of pivots are selected iteratively from a sample of the original dataset R (since preprocessing procedure is executed on a master node, the original dataset may be too large for it to process). First, we randomly select an object as the first pivot. Next, the object with the largest distance to the first pivot is selected as the

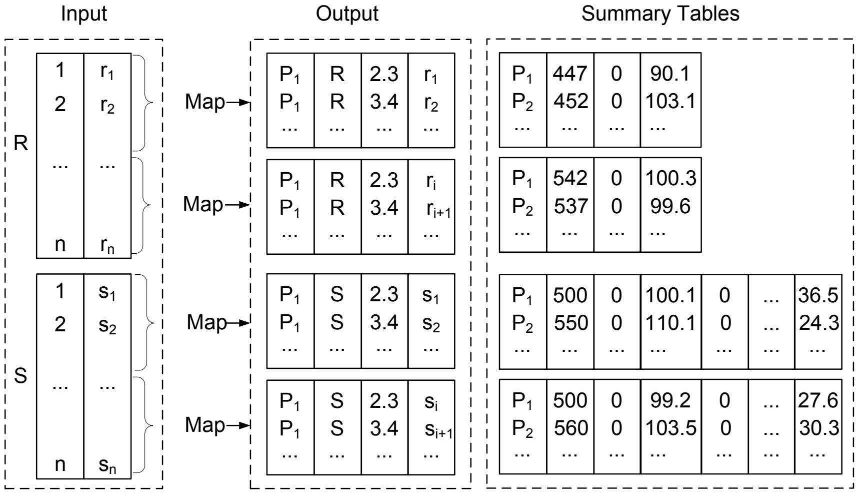


Figure 3.4: Partitioning and building the summary tables

second pivot. In the i^{th} iteration, the object that maximizes the sum of its distance to the first $i-1$ pivots is chosen as the i^{th} pivot.

- ***k*-means Selection.** Similar to the farthest selection, k-means selection first does sampling on the R. Then, traditional k-means clustering method is applied on the sample. With the k data clusters generated, the center point of each cluster is chosen as a pivot for the Voronoi diagram-based data partitioning.

3.4.2 First MapReduce Job

Given the set of pivots selected in the preprocessing step, we launch a MapReduce job for performing data partitioning and collecting some statistics for each partition. Figure 3.4 shows an example of the input/output of the mapper function of the first MapReduce job.

Specifically, before launching the map function, the selected pivots $\mathbb{P}$ are loaded into main memory in each mapper. A mapper sequentially reads each object o from the input split, computes the distance between o and all pivots in $\mathbb{P}$, and assigns o to the closest pivot P. Finally, as illustrated in Figure 3.4, the mapper outputs each object o along with its partition id, original dataset name (R or S), distance to the closest pivot.

Meanwhile, the first map function also collects some statistic for each input data split and these statistics are merged together while the MapReduce job completes. Two in-memory tables called summery tables are created to keep these statistics. Figure 3.3 shows an example of the summary tables T_R and T_S for partitions of R and S, respectively. Specifically, T_R maintains the following information for every partition of R: the partition id, the number of objects in the partition, the minimum distance $L(P_i^R)$ and maximum distance $L(P_i^R)$ from an object in partition P_i^R to the pivot. Note that although the pivots are selected based on dataset R alone, the Voronoi diagram based on the pivots can be used to partition S as well. T_S maintains the same fields as those in T_R for S. Moreover, T_S also maintains the distances between objects in $KNN(p_i, P_i^S)$ and p_i, where $KNN(p_i, P_i^S)$ refers to the k nearest neighbors of pivot p_i from objects in partition P_i^S. In Figure 3.3, $p_i.d_j$ in T_S represents the distance between pivot p_i and its j^{th} nearest neighbor in $KNN(p_i, P_i^S)$. The information in T_R and T_S will be used to guide how to generate S_i for R_i as well as to speed up the computation of $R_i \ltimes S_i$ by deriving distance bounds of the kNN for any object of R in the second MapReduce job.

3.4.3 Second MapReduce Job

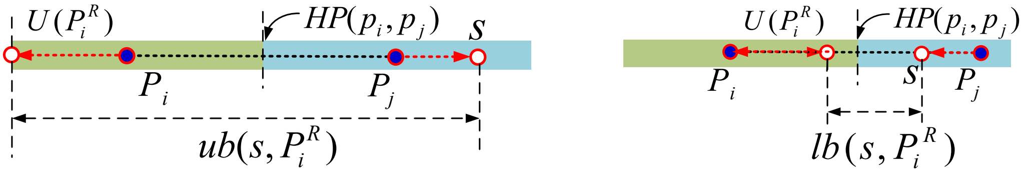
The second MapReduce job performs the kNN join in the way introduced in Section 3.3. The main task of the mapper in the second MapReduce is to find the corresponding subset S_i for each R_i. Each reducer performs the kNN join between a pair of R_i and S_i.

As mentioned previously, to guarantee the correctness, S_i should contains the kNN of all $r \in R_i$, i.e., $S_i = \bigcup_{\forall r_j \in R_i} KNN(r_j, S)$. However, we cannot get the exact S_i without performing the kNN join on R_i and S. Therefore, in the following, we derive a distance bound based on the partitioning of R which can help us reduce the size of S_i.

Distance Bound of kNN

Instead of computing the kNN from S for each object of R, we derive a bound of the kNN distance using a set oriented approach. Given a partition P_i^R (i.e., R_i) of R, we bound the distance of the kNN for all objects of P_i^R at a time based on T_R and T_S, which we have as a byproduct of the first MapReduce.

Theorem 3.3. *Given a partition $P_i^R \subset R$, an object s of $P_j^S \subset S$, the upper*



(a) upper bound

(b) lower bound

Figure 3.5: Bounding k nearest neighbors

bound distance from s to $\forall r \in P_i^R$, denoted as $ub(s, P_i^R)$, is:

$$ub(s, P_i^R) = U(P_i^R) + |p_i, p_j| + |p_j, s| \tag{3.5}$$

Proof. $\forall r \in P_i^R$, according to the triangle inequality, $|r, p_j| \leq |r, p_i| + |p_i, p_j|$. Similarly, $|r, s| \leq |r, p_j| + |p_j, s|$. Hence, $|r, s| \leq |r, p_i| + |p_i, p_j| + |p_j, s|$. Since $r \in P_i^R$, according to the definition of $U(P_i^R)$, $|r, p_i| \leq U(P_i^R)$. Clearly, we can derive $|r, s| \leq U(P_i^R) + |p_i, p_j| + |p_j, s| = ub(s, P_i^R)$. □

Figure 3.5(a) shows the geometric meaning of $ub(s, P_i^R)$. According to the Equation 3.5, we can find a set of k objects from S with the smallest upper bound distances as the kNN of all objects in P_i^R. For ease of exposition, let $KNN(P_i^R, S)$ be the k objects from S with the smallest $ub(s, P_i^R)$. Apparently, we can derive a bound (denoted as θ_i that corresponds to P_i^R) of the kNN distance for all objects in P_i^R as follows:

$$\theta_i = \max_{\forall s \in KNN(P_i^R, S)} |ub(s, P_i^R)|. \tag{3.6}$$

Clearly, $\forall r \in P_i^R$, the distance from r to any object of $KNN(r, S)$ is less than or equal to θ_i. Hence, we are able to bound the distance of the kNN for all objects of P_i^R at a time. Moreover, according to the Equation 3.5, we can also observe that in each partition P_i^S, k objects with the smallest distances to p_i may contribute to refine $KNN(P_i^R, S)$ while the remainder cannot. Hence, we only maintain k smallest distances of objects from each partition of S to its corresponding pivot in summary table T_S (shown in Figure 3.3).

Algorithm 1 shows the details on how to compute θ_i. We first create a priority queue PQ with size k (line 1). For partition P_j^S, we compute $ub(s, P_i^R)$ for each $s \in KNN(p_j, P_j^S)$, where $|s, p_j|$ is maintained in T_S. To speed up the computation of θ_i, we maintain $|s, p_j|$ in T_S based on the ascending order.

Algorithm 1: boundingKNN(P_i^R)

```
1 create a priority queue $PQ$;
2 foreach $P_j^S$ do
3     foreach $s \in KNN(p_j, P_j^S)$ do                  /* set in $T_S$ */
4         $ub(s, P_i^R) \leftarrow U(P_i^R) + |p_i, p_j| + |s, p_j|$;
5         if $PQ.size < k$ then $PQ$.add($ub(s, P_i^R)$);
6         else if $PQ.top > dist$ then
7             $PQ$.remove(); $PQ$.add($ub(s, P_i^R)$);
8         else break;
9 return $PQ$.top;
```

Hence, when $ub(s, P_i^R) \geq PQ.top$, we can guarantee that no remaining objects in $KNN(p_j, P_j^S)$ help refine θ_i (line 8). Finally, we return the top of PQ which is taken as θ_i (line 9).

Finding S_i for R_i

Similarly to Theorem 3.3, we can derive the lower bound distance from an object $s \in P_j^S$ to any object of P_i^R as follows.

Theorem 3.4. *Given a partition P_i^R, an object s of P_j^S, the lower bound distance from s to $\forall r \in P_i^R$, denoted by $lb(s, P_i^R)$, is:*

$$lb(s, P_i^R) = \max\{0, |p_i, p_j| - U(P_i^R) - |s, p_j|\} \tag{3.7}$$

Proof. $\forall r \in P_i^R$, according to the triangle inequality, $|r, p_j| \geq |p_j, p_i| - |p_i, r|$. Similarly, $|r, s| \geq |r, p_j| - |p_j, s|$. Hence, $|r, s| \geq |p_j, p_i| - |p_i, r| - |p_j, s|$. Since $r \in P_i^R$, according to the definition of $U(P_i^R)$, $|r, p_i| \leq U(P_i^R)$. Thus we can derive $|r, s| \geq |p_i, p_j| - U(P_i^R) - |s, p_j|$. As the distance between any two objects is not less than 0, the low bound distance $lb(s, P_i^R)$ is set to $\max\{0, |p_i, p_j| - U(P_i^R) - |s, p_j|\}$. □

Figure 3.5(b) shows the geometric meaning of $lb(s, P_i^R)$. Clearly, $\forall s \in S$, if we can verify $lb(s, P_i^R) > \theta_i$, then s cannot be one of $KNN(r, S)$ for any $r \in P_i^R$ and s is safe to be pruned. Hence, it is easy for us to verify whether an object $s \in S$ needs to be assigned to S_i.

Theorem 3.5. *Given a partition P_i^R and an object $s \in S$, the necessary condition that s is assigned to S_i is that: $lb(s, P_i^R) \leq \theta_i$.*

Algorithm 2: compLBOfReplica()

```
foreach $P_i^R$ do
    $\theta_i \leftarrow$ boundingKNN $(P_i^R)$;
foreach $P_j^S$ do
    foreach $P_i^R$ do
        $LB(P_j^S, P_i^R) \leftarrow |p_i, p_j| - U(P_i^R) - \theta_i$;
```

According to Theorem 3.5, $\forall s \in S$, by computing $lb(s, P_i^R)$ for all $P_i^R \subset R$, we can derive all S_i that s is assigned to. However, when the number of partitions for R is large, this computation cost might increase significantly since $\forall s \in P_j^S$, we need to compute $|p_i, p_j|$. To cope with this problem, we propose Corollary 3.2 to find all S_i which s is assigned to only based on $|s, p_j|$.

Corollary 3.2. *Given a partition P_i^R and a partition P_j^S, $\forall s \in P_j^S$, the necessary condition that s is assigned to S_i is that:*

$$|s, p_j| \geq LB(P_j^S, P_i^R), \tag{3.8}$$

where $LB(P_j^S, P_i^R) = |p_i, p_j| - U(P_i^R) - \theta_i$.

Proof. The conclusion directly follows Theorem 3.5 and Equation 3.7. □

According to Corollary 3.2, for partition P_j^S, objects exactly lying in region $[LB(P_j^S, P_i^R), U(P_j^S)]$ are assigned to S_i. Algorithm 2 shows how to compute $LB(P_j^S, P_i^R)$, which is self-explained.

kNN Join between R_i and S_i

As a summary, Algorithm 3 describes the details of kNN join procedure that is described in the second MapReduce job. Before launching map function, we first compute $LB(P_j^S, P_i^R)$ for every P_j^S (line 1–2). For each object $r \in R$, the map function generates a new key value pair in which the key is its partition id, and the value consists of $k1$ and $v1$ (line 4–6). For each object $s \in S$, the map function creates a set of new key value pairs, if not pruned based on Corollary 3.2 (line 7–11).

In this way, objects in each partition of R and their potential k nearest neighbors will be sent to the same reducer. By parsing the key value pair

$(k2, v2)$, the reducer can derive the partition P_i^R and subset S_i that consists of $P_{j_1}^S, \ldots, P_{j_M}^S$ (line 13), and compute the kNN of objects in partition P_i^R (line 16–25).

$\forall r \in P_i^R$, in order to reduce the number of distance computations, we first sort the partitions from S_i by the distances from their pivots to pivot p_i in the ascending order (line 14). This is based on the fact that if a pivot is near to p_i, then its corresponding partition often has higher probability of containing objects that are closer to r. In this way, we can derive a tighter bound distance of kNN for every object of P_i^R, leading to a higher pruning power. Based on Equation 3.6, we can derive a bound of the kNN distance, θ_i, for all objects of P_i^R. Hence, we can issue a range search with query r and threshold θ_i over dataset S_i. First, $KNN(r, S)$ is set to empty (line 17). Then, all partitions P_j^S are checked one by one (line 18–24). For each partition P_j^S, based on Corollary 3.1, if $d(r, HP(p_i, p_j)) > \theta$, no objects in P_j^S can help refine $KNN(r, S)$, and we proceed to check the next partition directly (line 19–20). Otherwise, $\forall s \in P_j^S$, if s cannot be pruned by Theorem 3.2, we need to compute the distance $|r, s|$. If $|r, s| < \theta$, $KNN(r, S)$ is updated with s and θ is updated accordingly (lines 22–24). After checking all partitions of S_i, the reducer outputs $KNN(r, S)$ (line 25).

3.5 Minimizing Replication of S

By bounding the k nearest neighbors for all objects in partition P_i^R, according to Corollary 3.2, $\forall s \in P_j^S$, we assign s to S_i when $|s, p_j| \geq LB(P_j^S, P_i^R)$. Apparently, to minimize the number of replicas of objects in S, we expect to find a large $LB(P_j^S, P_i^R)$ while keeping a small $|s, p_j|$. Intuitively, by selecting a larger number of pivots, we can split the dataset into a set of Voronoi cells (corresponding to partitions) with finer granularity and the bound of the kNN distance for all objects in each partition of R will become tighter. This observation is able to be confirmed by Equation 3.8. By enlarging the number of pivots, each object from $R \cup S$ is able to be assigned to a pivot with a smaller distance, which reduces both $|s, p_j|$ and the upper bound $U(P_i^R)$ for each partition P_i^R while a smaller $U(P_i^R)$ can help achieve a larger $LB(P_j^S, P_i^R)$. Hence, in order to minimize the replicas of objects in S, it is required to select a larger number of pivots. However, in this way, it might not be practical to provide a single

Algorithm 3: kNN join

```
map-setup                                    /* before running map function */
    compLBOfReplica ();
map (k1,v1)
    if $k1.dataset = R$ then
        $pid \leftarrow$ getPartitionID($k1$.partition);
        output($pid$, $(k1, v1)$);
    else
        $P_j^S \leftarrow k1$.partition;
        foreach $P_i^R$ do
            if $LB(P_j^S, P_i^R) \leq k1.dist$ then
                output($i$, $(k1, v1)$);
reduce (k2,v2)                               /* at the reducing phase */
    parse $P_i^R$ and $S_i$ $(P_{j_1}^S, \ldots, P_{j_M}^S)$ from $(k2, v2)$;
    sort $P_{j_1}^S, \ldots, P_{j_M}^S$ based on the ascending order of $|p_i, p_{j_l}|$;
    compute $\theta_i \leftarrow \max_{\forall s \in KNN(P_i^R, S)} |ub(s, P_i^R)|$;
    for $r \in P_i^R$ do
        $\theta \leftarrow \theta_i$; $KNN(r, S) \leftarrow \emptyset$;
        for $j \leftarrow j_1$ to $j_M$ do
            if $P_j^S$ can be pruned by Corollary 3.1 then
                continue;
            foreach $s \in P_j^S$ do
                if $s$ is not pruned by Theorem 3.2 then
                    refine $KNN(r, S)$ by $s$;
                    $\theta \leftarrow \max_{\forall o \in KNN(r,S)} \{|o, r|\}$;
        output($r$,$KNN(r, S)$);
```

reducer to handle each partition P_i^R. To cope with this problem, a natural idea is to divide partitions of R into disjoint groups, and take each group as R_i. In this way, S_i needs to be refined accordingly.

3.5.1 Cost Model

By default, let $R = \bigcup_{1 \leq i \leq N} G_i$, where G_i is a group consisting of a set of partitions of R and $G_i \cap G_j = \emptyset$, $i \neq j$.

Theorem 3.6. *Given partition P_j^S and group G_i, $\forall s \in P_j^S$, the necessary* *condition that s is assigned to S_i is:*

$$|s, p_j| \geq LB(P_j^S, G_i), \tag{3.9}$$

where $LB(P_j^S, G_i) = \min_{\forall P_i^R \in G_i} LB(P_j^S, P_i^R)$.

Proof. According to Corollary 3.2, s is assigned to S_i as long as there exists a partition $P_i^R \in G_i$ with $|s, p_j| \geq LB(P_j^S, P_i^R)$. □

By computing $LB(P_j^S, G_i)$ in advance for each partition P_j^S, we can derive all S_i for each $s \in P_j^S$ only based on $|s, p_j|$. Apparently, the average number of replicas of objects in S is reduced since duplicates in S_i are eliminated. According to Theorem 3.6, we can easily derive the number of all replicas (denoted as $RP(S)$) as follows.

Theorem 3.7. *The number of replicas of objects in S that are distributed to reducers is:*

$$RP(S) = \sum_{\forall G_i} \sum_{\forall P_j^S} |\{s | s \in P_j^S \wedge |s, p_j| \geq LB(P_j^S, G_i)\}| \tag{3.10}$$

3.5.2 Grouping Strategies

We present two strategies for grouping partitions of R to approximately minimize $RP(S)$.

Geometric Grouping

Geometric grouping is based on an important observation: given two partitions P_i^R and P_j^S, if p_j is far away from p_i compared with the remaining pivots, then P_j^S is deemed to have a low possibility of containing objects as any of kNN for objects in P_i^R. This observation can be confirmed in Figure 3.1 where partition P_5 does not have objects to be taken as any of kNN of objects in P_2. Hence, a natural idea to divide partitions of R is that we make the partitions, whose corresponding pivots are near to each other, into the same group. In this way, regarding group G_i, objects of partitions from S that are far away from partitions of G_i will have a large possibility to be pruned.

Algorithm 4 shows the details of geometric grouping. We first select the pivot p_i with the farthest distance to all the other pivots (line 1) and assign

Algorithm 4: geoGrouping()

```
1  select $p_i$ such that $\sum_{p_j \in \mathbb{P}} |p_i, p_j|$ is maximized;
2  $\tau \leftarrow \{p_i\}$; $G_1 \leftarrow \{P_i^R\}$; $\mathbb{P} \leftarrow \mathbb{P} - \{p_i\}$;
3  for $i \leftarrow 2$ to $N$ do
4      select $p_l \in \mathbb{P}$ such that $\sum_{p_j \in \tau} |p_l, p_j|$ is maximized;
5      $G_i \leftarrow \{P_l^R\}$; $\mathbb{P} \leftarrow \mathbb{P} - \{p_l\}$;$\tau \leftarrow \tau \cup \{p_l\}$;
6  while $\mathbb{P} \neq \emptyset$ do
7      select group $G_i$ with the smallest number of objects;
8      select $p_l \in \mathbb{P}$ such that $\sum_{\forall P_j^R \subset G_i} |p_l, p_j|$ is minimized;
9      $G_i \leftarrow G_i \cup \{P_l^R\}$; $\mathbb{P} \leftarrow \mathbb{P} - \{p_l\}$;
10 return $\{G_1, G_2, \ldots, G_N\}$
```

partition P_i^R to group G_1 (line 2). We then sequentially assign a partition to the remaining groups as follows: for group G_i ($2 \leq i \leq N$), we compute the pivot p_l which has the farthest distance to the selected pivots (τ) and assign P_l^R to G_i (line 3–5). In this way, we can guarantee that the distance among all groups are the farthest at the initial phase. After assigning the first partition for each group, in order to balance the workload, we do the following iteration until all partitions are assigned to the groups: (1) select the group G_i with the smallest number of objects (line 7); (2) compute the pivot p_l with the minimum distance to the pivots of G_i, and assign P_l^R to G_i (line 8–9). In this way, we can achieve that the number of objects in each group is nearly the same. Finally, we return all groups that maintain partitions of R (line 10).

Greedy Grouping

Let $RP(S, G_i)$ be the set of objects from S that need to be assigned to S_i. The objective of greedy grouping is to minimize the size of $RP(S, G_i \cup \{P_j^R\}) - RP(S, G_i)$ when assigning a new partition P_j^R to G_i. According to Theorem 3.6, $RP(S, G_i)$ is able to be formally quantified as:

$$RP(S, G_i) = \bigcup_{\forall P_j^S \subset S} \{s | s \in P_j^S \wedge |s, p_j| \geq LB(P_j^S, G_i)\} \qquad (3.11)$$

Hence, theoretically, when implementing the greedy grouping approach, we can achieve the optimization objective by minimizing $RP(S, G_i \cup \{P_j^R\}) - RP(S, G_i)$ instead of $\sum_{P_j^R \in G_i} |p_i, p_j|$ in the geometric grouping approach. However, it is rather costly to select a partition P_j^R from all remaining partitions with minimum $RP(S, G_i \cup \{P_j^R\}) - RP(S, G_i)$. This is because by adding a new partition P_j^R to G_i, we need to count the number of emerging objects from S that are distributed to the S_i. Hence, to reduce the computation cost, once $\exists s \in P_l^S$, $|s, p_j| \le LB(P_j^S, G_i)$, we add all objects of partition P_l^S to $RP(S, G_i)$, i.e., the $RP(S, G_i)$ is approximately quantified as:

$$RP(S, G_i) \approx \bigcup_{\forall P_j^S \subset S} \{P_j^S | LB(P_j^S, G_i) \le U(P_j^S)\} \tag{3.12}$$

Remark: To answer kNN join by exploiting the grouping strategies, we use the group id as the key of the Map output. We omit the details which are basically the same as described in Algorithm 3.

3.6 Experimental Evaluation

We evaluate the performance of the proposed algorithms on our in-house cluster, Awan[4]. The cluster includes 72 computing nodes, each of which has one Intel X3430 2.4GHz processor, 8GB of memory, two 500GB SATA hard disks and gigabit ethernet. On each node, we install CentOS 5.5 operating system, Java 1.6.0 with a 64-bit server VM, and epiC. All the nodes are connected via three high-speed switches. To adapt the epiC environment to our application, we make the following changes to the default epiC configurations: (1) the replication factor is set to 1; (2) each node is configured to run one MapUnit and one ReduceUnit. (3) the size of virtual memory for each MapUnit and ReduceUnit is set to 4GB. For simplicity, we refer to MapUnit and ReduceUnit in epiC as mapper and reducer in the following content, respectively.

We evaluate the following approaches in the experiments.

- H-BRJ is proposed in [96] and described in Section 3.3. In particular, to speed up the computation of $R_i \ltimes S_j$, it employs R-tree to index objects of S_j and finds kNN for $\forall r \in R_i$ by traversing the R-tree. We used the implementation generously provided by the authors;

- PGBJ is our proposed kNN join algorithm that utilizes the partitioning and grouping strategy;

[4] http://awan.ddns.comp.nus.edu.sg/ganglia/

- PBJ is also our proposed kNN join algorithm. The only difference between PBJ and PGBJ is that PBJ does not have the grouping part. Instead, it employs the same framework used in H-BRJ. Hence, it also requires an extra MapReduce job to merge the final results.

We conduct the experiments using self-join on the following datasets:

- Forest FCoverType[5] (Forest for short): This is a real dataset that predicts forest cover type from cartographic variables. There are 580K objects, each with 54 attributes (10 integer, 44 binary). We use 10 integer attributes in the experiments.

- Expanded Forest FCoverType dataset: To evaluate the performance on large datasets, we increase the size of Forest while maintaining the same distribution of values over the dimensions of objects (like [86]). We generate new objects in the way as follows: (1) we first compute the frequencies of values in each dimension, and sort values in the ascending order of their frequencies; (2) for each object o in the original dataset, we create a new object $\bar{o}$, where in each dimension D_i, $\bar{o}[i]$ is ranked next to $o[i]$ in the sorted list. Further, to create multiple new objects based on object o, we replace $o[i]$ with a set of values next to it in the sorted list for D_i. In particular, if $o[i]$ is the last value in the list for D_i, we keep this value constant. We build Expanded Forest FCoverType dataset by increasing the size of Forest dataset from 5 to 25 times. We use "Forest $\times t$" to denote the increased dataset where $t \in [5, 25]$ is the increase factor.

- OpenStreetMap[6] (OSM for short): this is a real map dataset containing the location and description of objects. We extract 10 million records from this dataset, where each record consists of 2 real values (longitude and latitude) and a description with variable length.

By default, we evaluate the performance of kNN join (k is set to 10) on the "Forest ×10" dataset using 36 computing nodes. We measure several parameters, including query time, distance computation selectivity, and shuffling cost. The distance computation selectivity (computation selectivity for short)

[5] http://archive.ics.uci.edu/ml/datasets/Covertype

[6] http://www.openstreetmap.org

Table 3.2: Statistics of partition size

# of pivots		2000	4000	6000	8000
Random Selection	min.	116	18	24	6
	max.	9062	5383	4566	2892
	avg.	2905.06	1452.53	968.35	726.27
	dev.	1366.5	686.41	452.79	338.88
Farthest Selection	min.	24	14	13	12
	max.	1130678	1018605	219761	97512
	avg.	2905.06	1452.53	968.35	726.27
	dev.	27721.1	13313.56	5821.18	2777.84
k-means Selection	min.	52	17	3	6
	max.	7829	5222	3597	2892
	avg.	2905.06	1452.53	968.35	726.27
	dev.	1212.38	700.20	529.92	338.88

Table 3.3: Statistics of group size

# of pivots		2000	4000	6000	8000
Random Selection	min.	143720	144564	144758	144961
	max.	150531	147180	146617	146118
	avg.	145253	145253	145253	145253
	dev.	1656	560	378	251
Farthest Selection	min.	86805	126635	116656	141072
	max.	1158084	221539	1078712	173002
	avg.	145253	145253	145253	145253
	dev.	170752	20204	149673	6916
k-means Selection	min.	143626	144456	144746	144961
	max.	148111	146521	145858	146118
	avg.	145253	145253	145253	145253
	dev.	1201	570	342	251

is computed as follows:

$$\frac{\#\text{ of object pairs to be computed}}{|R| \times |S|}, \tag{3.13}$$

where the objects also include the pivots in our case.

3.6.1 Study of Parameters of Our Techniques

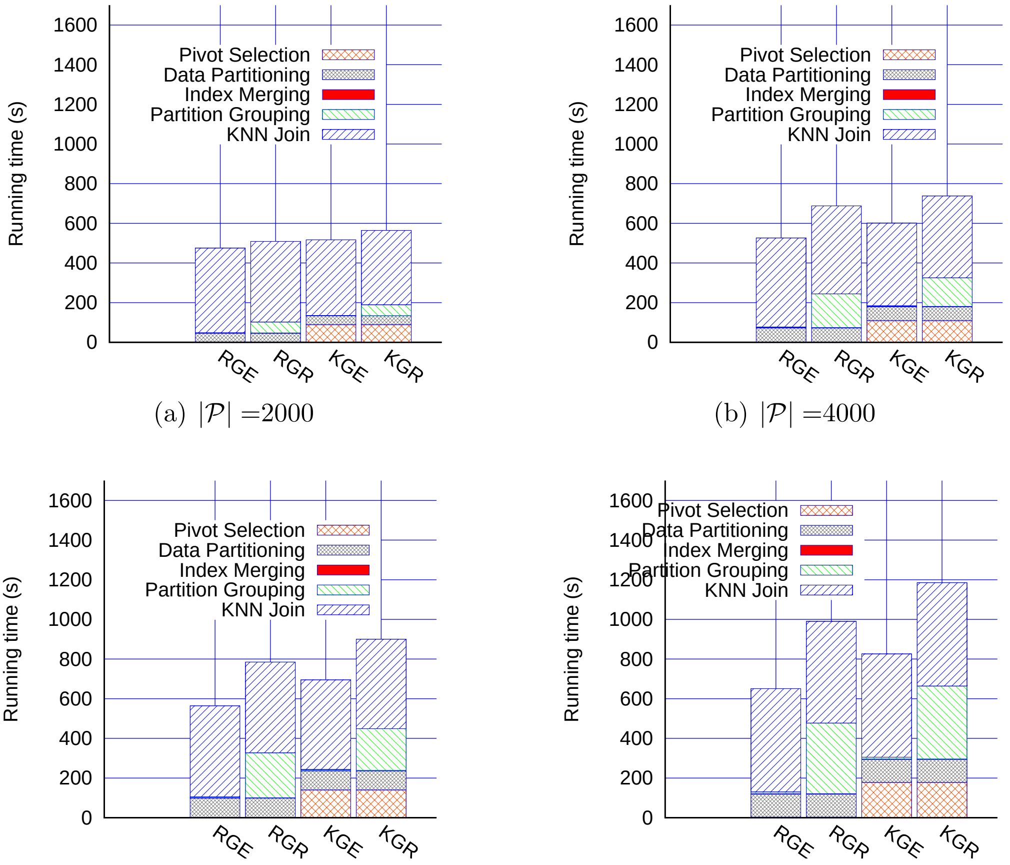
We study the parameters of PGBJ with respect to pivot selection strategy, pivot number, and grouping strategy. By combining different pivot selection and grouping strategies, we obtain 6 strategies, which are: (1) **RGE**, random selection + geometric grouping; (2) **FGE**, farthest selection + geometric grouping; (3) **KGE**, k-means selection + geometric grouping; (4) **RGR**, random selection + greedy grouping; (5)**FGR**, farthest selection + greedy grouping; (6) **KGR**, k-means selection + greedy grouping.

Effect of Pivot Selection Strategies

Table 3.2 shows the statistics of partition sizes using different pivot selection strategies including random selection, farthest selection and k-means selection. We observe that the standard deviation (`dev.`for short) of partition size drops rapidly when the number of pivots increases. Compared to random selection and k-means selection, partition size varies significantly in the farthest selection. The reason is that in the farthest selection, outliers are always selected as pivots. Partitions corresponding to these pivots contain few objects, while other partitions whose pivots reside in dense areas contain a large number of objects. Specifically, when we select 2000 pivots using farthest selection, the maximal partition size is 1,130,678, which is about 1/5 of the dataset size. This large difference in partition size will degrade performance due to the unbalanced workload. We also investigate the group size using geometric grouping approach[7]. As shown in Table 3.3, the number of objects in each group varies significantly using the farthest selection. Again, this destroys the load balance since each reducer needs to perform significantly different volume of computations. However, the group sizes using random selection and k-means selection are approximately the same.

Figure 3.6 shows the execution time for various phases in kNN join. We do not provide the execution time for farthest selection because it takes more than 10,000s to answer kNN join. The reason of the poor performance is: almost all the partitions of S overlap with large-size partitions of R. Namely, we need to compute distances for a large number of object pairs. Comparing **RGE** with **KGE**, and **RGR** with **KGR** in Figure 3.6, we observe that the overall performance using random selection is better than that using k-means selection. Further, when the number of pivots increases, the gap of the overall performance becomes larger. This is because k-means selection involves a large number of distance computations, which results in large execution time.

[7] We omit the results for greedy grouping as they follows the same trend.



(a) $|\mathcal{P}|$ =2000

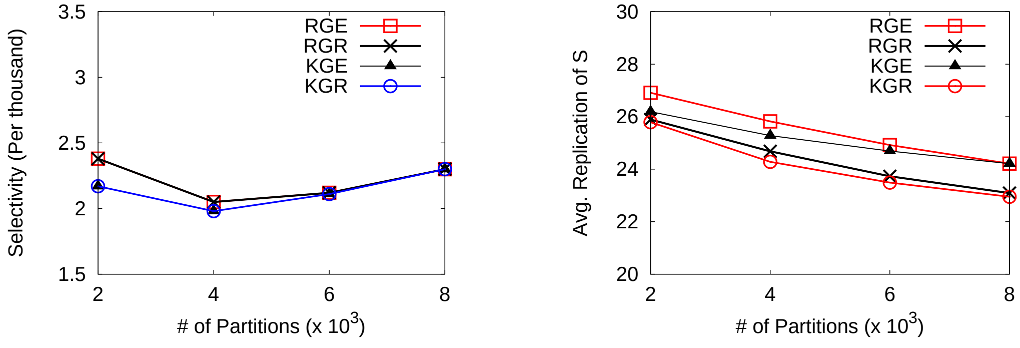
(b) $|\mathcal{P}|$ =4000

(c) $|\mathcal{P}|$ =6000

(d) $|\mathcal{P}|$ =8000

Figure 3.6: Query cost of tuning parameters

Things get worse when k increases. However, during the kNN join phase, the performance of k-means selection is slightly better than that of random selection. To verify the result, we investigate the computation selectivity for both cases. As shown in Figure 3.7(a), we observe that the computation selectivity of using k-means selection is less than that of using random selection. Intuitively, k-means selection is more likely to select high-quality pivots that separate the whole dataset more evenly, which enhances the power of our pruning rules. However, another observation is that the selectivity difference becomes smaller when the number of pivots increases. This is because k-means selection will deteriorate into random selection when the number of pivots becomes larger. It is worth mentioning that the computation selectivity of all the techniques is low, where the maximum is only 2.38‰.



(a) computation selectivity

(b) replication

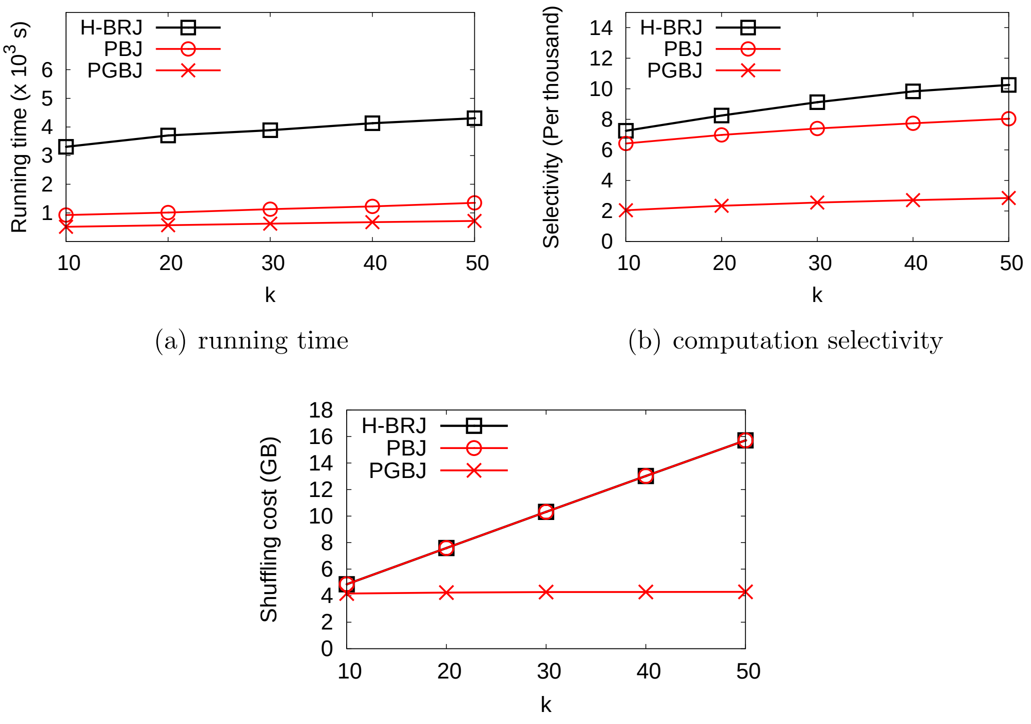
Figure 3.7: Computation selectivity and replication

Effect of the Pivot Number

From Figure 3.6, we observe that the minimal execution time for kNN join phase occurs when $|\mathcal{P}| = 4000$. To specify the reason, we provide the computation selectivity in Figure 3.7(a). From this figure, we find that the computation selectivity drops by varying $|\mathcal{P}|$ from 2000 to 4000, but increases by varying $|\mathcal{P}|$ from 4000 to 8000. As discussed in kNN join algorithm, to compute $KNN(r, S)$, we need to compute the distances between r and objects from S as well as between r and $p_i \in \mathcal{P}$. When the number of pivots increases, the whole space will be split into a finer granularity and the pruning power will be enhanced as the bound becomes tighter. This leads to a reduction in both distance computation between R and S and replication for S. The results for replication of S are shown in Figure 3.7(b). One drawback of using a large number of pivots is that the number of distance computation between r and the pivots becomes larger. On balance, the computation selectivity is minimized when $|\mathcal{P}| = 4000$. For the overall execution time, it arrives at the minimum value when $|\mathcal{P}| = 4000$ for **RGE** and $|\mathcal{P}| = 2000$ for the remaining strategies. The overall performance degrades for all the combination of pivot selection and partition grouping strategies when the number of pivots increases.

Effect of Grouping Strategies

When comparing **RGE** with **RGR**, and **KGE** with **KGR** in Figure 3.6, we find the execution time in the kNN join phase remains almost the same using different grouping strategies. In fact, in our partitioning based approach, for



(a) running time

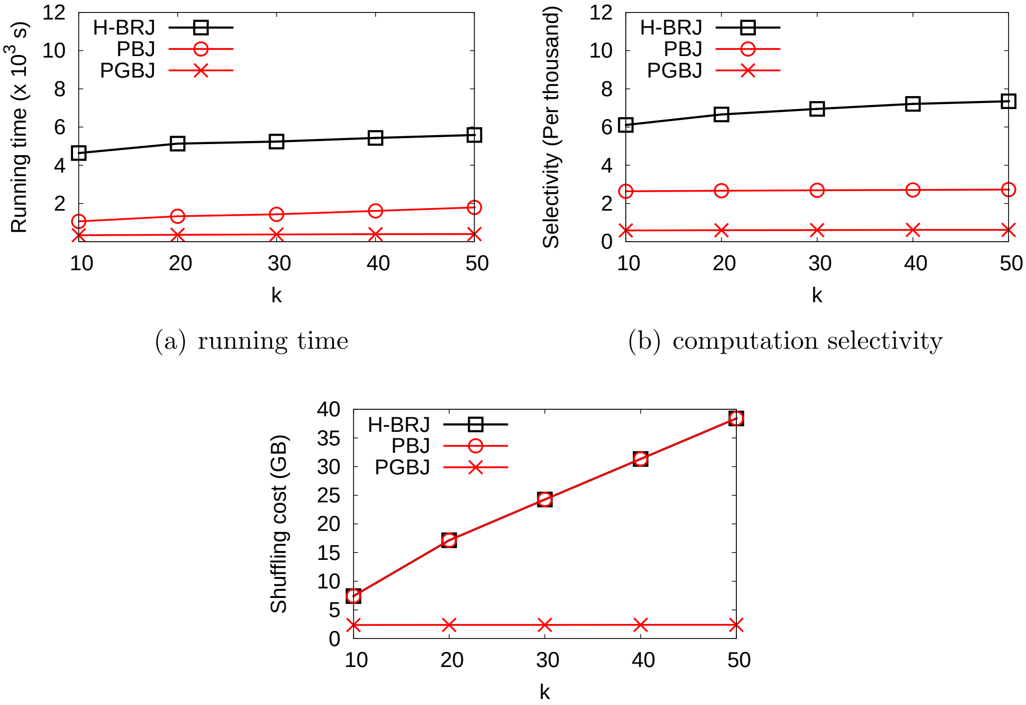
(b) computation selectivity

(c) shuffling cost

Figure 3.8: Effect of k over "Forest × 10"

each object r with all its potential k nearest neighbors, the number of distance computations for r remains constant. This is consistent with the results for the number of object pairs to be computed in Figure 3.7(a). As described above, in PGBJ, $\forall r \in R_i$, we send all its potential kNN from S to the same reducer. Hence, the shuffling cost depends on how to partition R into subsets. From Figure 3.7(b), when $|\mathcal{P}|$ increases, the average replication of S using greedy grouping is slightly reduced. However, the execution time in partition grouping phase increases significantly. This leads to the increment in the overall execution time.

Remark. To summarize the study of the parameters, we find that the overall execution time is minimized when $|\mathcal{P}| = 4000$ and **RGE** strategy is adopted to answer kNN join. Hence, in the remaining experiments, for both PBJ and PBGJ, we randomly select 4000 pivots to partition the datasets. Further, we use geometric grouping strategy to group the partitions for PBGJ.



(a) running time

(b) computation selectivity

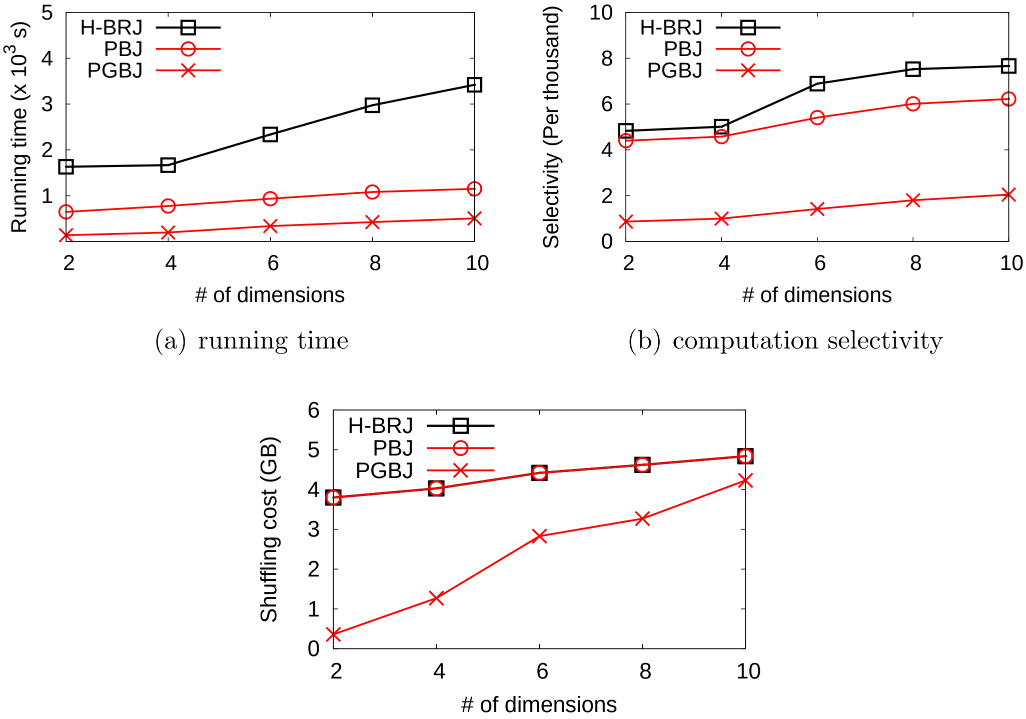
(c) shuffling cost

Figure 3.9: Effect of k over OSM dataset

3.6.2 Effect of k

We now study the effect of k on the performance of our proposed techniques. Figure 3.8 and Figure 3.9 present the results by varying k from 10 to 50 on "Forest × 10" and OSM datasets, respectively.

In terms of running time, PGBJ always performs best, followed by PBJ and H-BRJ.This is consistent with the results for computation selectivity. H-BRJ requires each reducer to build a R-tree index for all the received objects from S. To find the kNN for an object from R, the reducers will traverse the index and maintain candidate objects as well as a set of intermediate nodes in a priority queue. Both operations are costly for multi-dimensional objects, which result in the long running time. In PGJ, our proposed pruning rules allow each reducer to derive a distance bound from received objects in S. This bound is used to reduce computation cost for kNN join. However, without grouping phase, PGJ randomly sends a subset of S to each reducer. This randomness results in a loose distance bound, thus degrading the performance. In addition, Figure 3.8(c) shows the shuffling cost of three approaches on the default dataset.



(a) running time

(b) computation selectivity

(c) shuffling cost

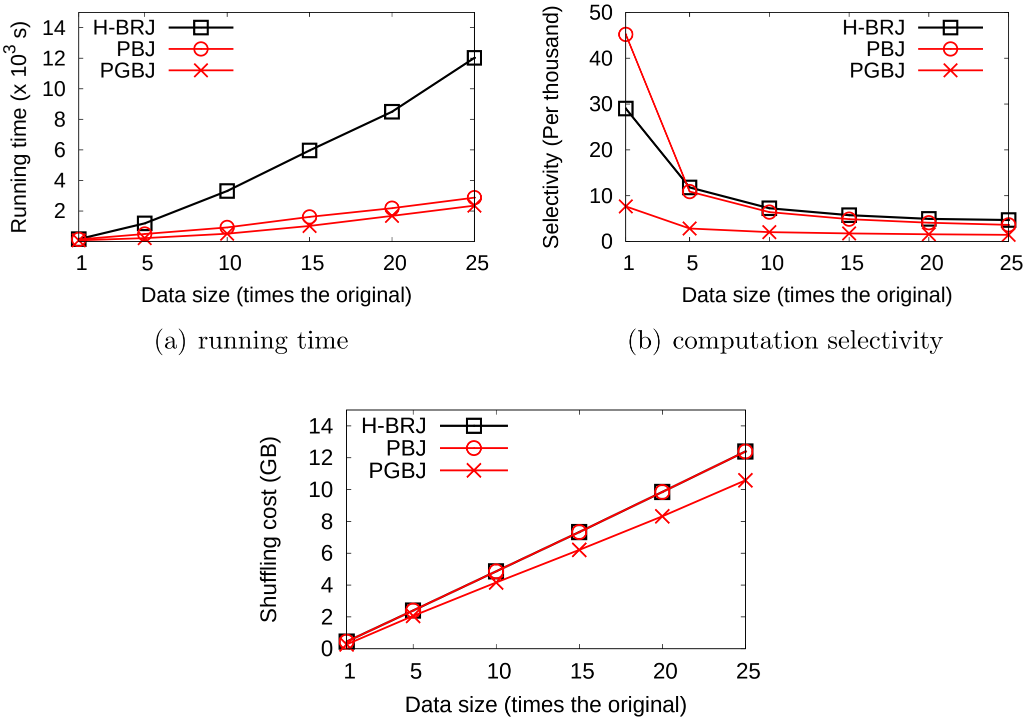
Figure 3.10: Effect of dimensionality

As we can see, when k increases, the shuffling cost of PGBJ remains nearly the same, while it increases linearly for PBJ and H-BRJ. This indicates that the replication of S in PGBJ is insensitive to k. However, for H-BRJ and PBJ, the shuffling cost of $R_i \ltimes S_j$ ($\forall R_i \subset R$, $S_j \subset S$) increases linearly when k varies.

3.6.3 Effect of Dimensionality

We now evaluate the effect of dimensionality. Figure 3.10 presents both the running time and computation selectivity by varying the number of dimensions from 2 to 10.

From the results, we observe that H-BRJ is more sensitive to the number of the dimensions than PBJ and PGBJ. In particular, the execution time increases exponentially when n varies from 2 to 6. This results from the curse of dimensionality. When the number of dimensions increases, the number of object pairs to be computed increases exponentially. Interestingly, the execution time of kNN join increases smoothly when n varies from 6 to 10. To explain



(a) running time

(b) computation selectivity

(c) shuffling cost

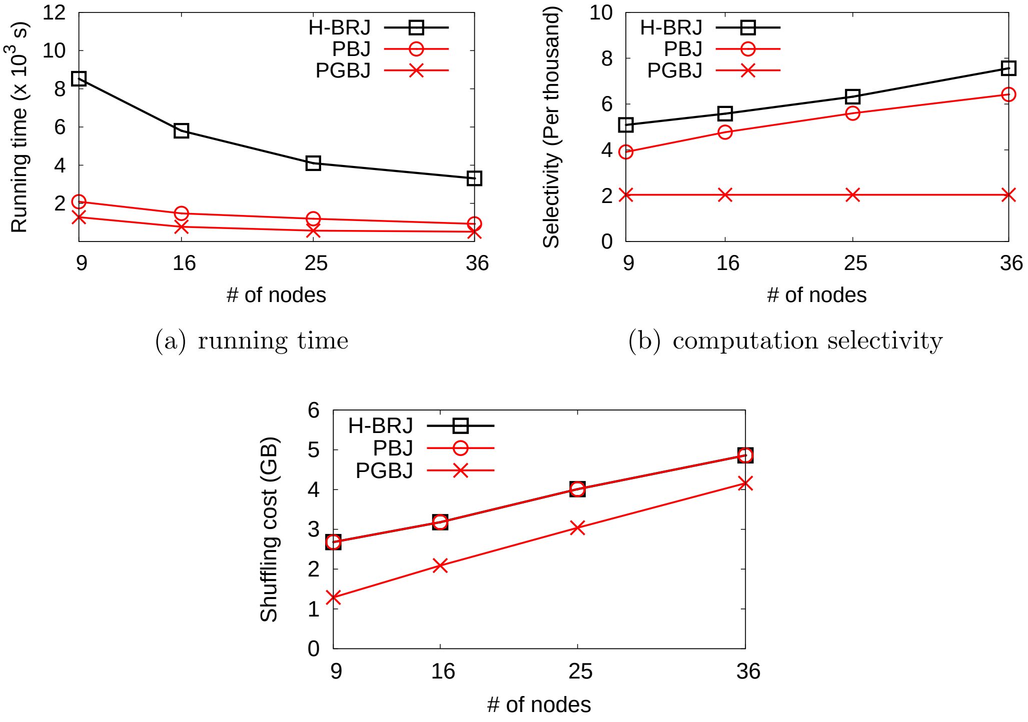
Figure 3.11: Scalability results

this phenomenon, we analyze the original dataset and find that values of 6–10 attributes have low variance, which means the kNN for objects from R do not change too much by adding these dimensions. We show the shuffling cost in Figure 3.10(c). For H-BRJ and PBJ, when the number of dimensions increases, the shuffling cost increases linearly due to the larger data size. However, for PGBJ, when the number of dimensions varies from 2 to 6, the shuffling cost increases exponentially due to the exponential increment of the replication of S. Nevertheless, it will converge to $|R| + N \times |S|$ even at the worst case. Although it may exceed both H-BRJ and PBJ, in that case, PBJ can be used instead of PBGJ if we take the shuffling cost into main consideration.

3.6.4 Scalability

We now investigate the scalability of three approaches. Figure 3.11 presents the results by varying the data size from 1 to 25 times of the original dataset.

From Figure 3.11(a), we can see that the overall execution time of all the



(a) running time

(b) computation selectivity

(c) shuffling cost

Figure 3.12: Speedup results

three approaches quadratically increases when we enlarge the data size. This is determined by the fact that the number of object pairs increase quadratically with the data size. However, PGBJ scales better than both PBJ and H-BRJ. In particular, when data size becomes larger, the running time of PGBJ grows much slower than that of H-BRJ. To verify this result, we analyze the computation selectivity for the three approaches. As shown in Figure 3.11(b), the computation selectivity of PGBJ is always the smallest one. One observation is that when data size increases, the selectivity differences among three approaches tend to be constant. In practice, for large datasets with multi-dimensional objects, a tiny decrease in selectivity will lead to a dramatic improvement in performance. This is the reason that the running time of PGBJ is nearly 6 times faster than that of H-BRJ on "Forest × 25", even if their selectivity does not differ too much. We also present the shuffling cost in Figure 3.11(c). From the figure, we observe that the shuffling cost of PGBJ is still less than that of PBJ and H-BRJ, and there is an obvious trend of increasing returns when the data size increases.

3.6.5 Speedup

We now measure the effect of the number of computing nodes. Figure 3.12 presents the results by varying the number of computing nodes from 9 to 36.

From Figure 3.12(a), we observe that the gap of running time among three approaches tends to be smaller when the number of computing nodes increases. Due to the increment of number of computing nodes, for H-BRJ and PBJ, the distribution of objects over each reducer becomes sparser. This leads to an increment of computation selectivity that is shown in Figure 3.12(b). However, the computation selectivity for PGBJ remains constant. Based on this trend, it is reasonable to expect that PGBJ will always outperform both H-BRJ and PBJ, while the improvement in running time is getting less obvious. We also show the shuffling cost in Figure 3.12(c). From the figure, we can see that the shuffling cost increases linearly with the number of computing nodes. In addition, our approaches cannot speed up linearly, because: (1) each node needs to read pivots from the distributed file system; (2) the shuffling cost will be increased.

3.7 Summary

In this chapter, we study the problem of efficiently answering the k nearest neighbor join using MapReduce framework. By exploiting Voronoi diagram-based partitioning method, our proposed approach is able to divide the input datasets into groups and we can answer the k nearest neighbor join by only checking object pairs within each group. Several pruning rules are developed to reduce the shuffling cost as well as the computation cost. We implement our proposed solution using MapUnit/ReduceUnit in epiC, which is an implementation of MapReduce framework. Extensive experiments performed on both real and synthetic datasets demonstrate that our proposed methods are efficient, robust and scalable.

CHAPTER 4

epiCG: An Efficient Distributed Graph Engine on epiC

In this chapter, we extend epiC and introduce our distributed graph processing engine, epiCG. While existing distributed graph processing systems require to deploy a new framework in the cluster for graph processing, we develop epiCG as an extension of epiC to reduce the effort of system deployment. epiCG supports both edge-cut and vertex-cut graph partitioning. For vertex-cut, we propose a light-weight approach to parallelize the process of generating a vertex-cut partitioning. In terms of fault tolerance, epiCG achieves automatic failure detection and recovery. Extensive experiments show the high efficiency and scalability of epiCG.

4.1 Introduction

Graph is a powerful data structure that models the relationships among objects. Real-life graph examples include social network graphs, road networks and global web graphs. Graph analytics, which aims to extract values from real-life graphs, has attracted more and more attention in various domains such as machine learning and data mining. However, the size of real-life graphs has been growing at a dramatic space in recent years, which inevitably poses challenges in graph analytics to cope with the scale.

MapReduce [31] was proposed as a programming model to handle large-scale data analytics. MapReduce gains popularity due to its efficiency, simplicity and scalability. However, MapReduce is tailored for one-pass data processing, but performs poorly for iterative graph analytics tasks such as shortest path and connected component computation. To address the problem, Pregel [59], a vertex-centric programming model, was proposed to handle iterative graph applications. Pregel follows the Bulk Synchronous Parallel (BSP) model [84]. During each iteration, all the vertices execute a *compute* function defined by the programmer in parallel and communicate with each other via messages. After all the vertices finish their computations and message forwarding, a global synchronization point is reached and all the vertices proceed to the next iteration. Inspired by Pregel, various distributed graph processing systems following the same programming model as Pregel are developed such as GPS [75], Giraph [2], Hama [5], GraphX [91] and Pregelix [4].

4.1.1 Issues and Opportunities

While various implementations of Pregel have been proposed for large-scale graph processing, we find most of them require users to set up a new framework and conduct necessary configuration on their own clusters before running any graph applications. This is always a daunting job for the users who manage the cluster. In other words, regarding various types of data analytics jobs, we should avoid setting up a new system for each type of them.

Opportunity: Is there any unified distributed solution to support various types of analytics applications, for instance, both MapReduce jobs and Pregel jobs?

Another issue about Pregel is the high communication overhead caused by cross-machine message forwarding. Pregel adopts edge-cut based graph partitioning which distributes vertices among the compute nodes and allows edges to cross the nodes. Network communication overhead incurs when a vertex in one compute node wants to send a message to its neighbor in another node. It is worthy noting that the overhead becomes more significant when the degrees of vertices in the graph follow a power-law distribution. Such kind of graph is also referred to as natural graphs. For natural graphs, it can hardly find a good graph partitioning that involves a small number of cross-node edges. Instead of using edge-cut based partitioning method, PowerGraph [36] proposed vertex-cut based graph partitioning. The idea is to randomly distribute edges among the compute nodes and allow vertices to across the nodes. Unlike edge-cut, the communication cost produced by a vertex-cut is restricted to the total number of compute nodes spanned by the vertices. However, PowerGraph does not follow the same framework as Pregel. It abstracts a graph algorithm as the execution of a sequence of *gather()*, *apply()* and *scatter()* functions. Furthermore, the vertex-cut partitioning algorithm used in PowerGraph places the $i+1$ edge after having placed the previous i edges. Therefore, when generating an edge-cut, PowerGraph requires one compute node to load the entire graph into the main memory, execute the partitioning algorithm and forward edges to the nodes accordingly. This has a minimum requirement for the RAM needed to run the edge-cut partitioning algorithm.

Opportunity: Can we implement a light-weight vertex-cut partitioning method in a system that follows the same programming model as Pregel?

The third issue is fault tolerance. Most existing distributed graph processing systems adopt checkpoint-based recovery mechanism. That is, a checkpoint is made periodically. Once a compute node fails or reports an exception, the job will restart its execution from the latest checkpoint. However, to our best knowledge, existing distributed graph processing systems such as GPS, Giraph, Hama cannot recover from failures automatically. Instead, when a failure occurs, the job will be terminated and the system requires users to manually restart the job from the latest checkpoint. Apparently, such kind of implementation violates the requirement of fault tolerance to perform recovery automatically and resume normal execution properly after recovery completes.

Opportunity: Can we achieve automatic failure detection and recovery in distributed graph processing systems?

4.1.2 Our Solution and Contributions

To address the above three challenging issues, we propose our graph processing engine, epiCG. We build epiCG on top of epiC, an elastic data processing system proposed for large-scale data analytics. epiC adopts the Actor-like model as its common programming model and regards any particular programming model as one actor. The common programming model of epiC consists of a set of computation units that can perform computation in parallel and communicate with each other via message passing. Based on this general model, epiC is able to support the implementation of various customized programming models, including MapReduce model, relational model and Pregel model. We implement epiCG as one extension of epiC and obtain the following benefits.

- Reusability: implementation cycle of a graph engine is shortened by leveraging existing components provided in epiC.
- One-size-fit-all: the user does not need to configure the cluster to run a new system for graph applications, which is always a daunting job for system administrators.

While epiC is extensible to support iterative graph applications, the implementation of epiCG on top of epiC is still challenging. The reason is two-fold. First, epiC employs a disk-based generic programming model. That is, the output of a epiC job will be written back to the distributed file system. Consider a PageRank algorithm ran on top of epiC. After the system finishes one iteration of graph processing, it will flush the newly computed graph back to the distributed file system, which will be loaded into memory at the beginning of the next iteration. This incurs high I/O cost when the number of iterations is large. Second, computation units in epiC cannot communication with each other directly. For example, if one computation unit wants to send a messages to another unit, it has to first send the message to the master thread in epiC and the master thread will then forward the message to the corresponding computation unit. Apparently, the master would become the bottleneck as the volume of the messages involved in many graph applications is always large.

To solve both problems, we implement epiCG as an in-memory processing engine on top of epiC. In particular, we allow computation units to maintain their computed data in memory during iterations until they receive a *flush* request from the master. Moreover, we establish connections between every two computation units such that any two units can forward messages between each other directly.

Contributions: the contribution of this work is the following.

- We develop a scalable graph processing engine epiCG as one extension of epiC, making users to execute different types of analytics jobs using the same distributed system. epiCG is an in-memory processing engine that can handle iterative graph applications efficiently. We also establish connections between every two computation units in epiCG for the purpose of direct communication.

- epiCG supports vertex-centric computation using both edge-cut and vertex-cut graph partitioning. For vertex-cut, we propose a light-weight greedy strategy which uses multiple compute nodes to generate a vertex-cut in parallel. In terms of fault tolerance, epiCG allows automatic failure detection and recovery.

- We conduct comprehensive experiments to demonstrate the efficiency and scalability of epiCG, compared with advanced graph processing systems.

4.2 Overview of epiCG

In this section, we present the graph model, computation model and architecture of our distributed graph engine epiCG.

Graph model. The input of epiCG is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of vertices and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ represents the set of directed edges[1]. To distribute the graph among a cluster $\mathcal{N}$of compute nodes, the graph is divided into partitions, where each partition contains a set of vertices and all the edges. Let $\mathcal{P}$ be the set of all the partitions. For any partition $P_i \in \mathcal{P}$, we have $P_i = (\mathcal{V}_i, \mathcal{E}_i)$ where $\mathcal{V}_i \subseteq \mathcal{V}$ and $\mathcal{E}_i = \{(u, v) \in \mathcal{E} \mid u \in \mathcal{V}_i\}$. Typically, the number of partitions is much larger than the number of compute nodes and each compute node will be assigned with multiple partitions.

Computation model. Similar to Pregel, epiCG follows the Bulk Synchronization Parallel (BSP) model [83]. Typically, the execution of an epiCG job starts from an initialization phase where the graph is distributed among compute nodes, followed by several iterations of computations, called *supersteps*, and finally an output phase to flush out the results. Every vertex carries two states: active and inactive. At the beginning of the job, every vertex is active. During the computation, a vertex can inactivate itself by *voltToHalt()* and it will be automatically activated if receiving any new incoming messages.

[1] For indirected graph, we can represent each undirected edge (u, v) by two directed edges $\langle u, v\rangle$ and $\langle u, v\rangle$.

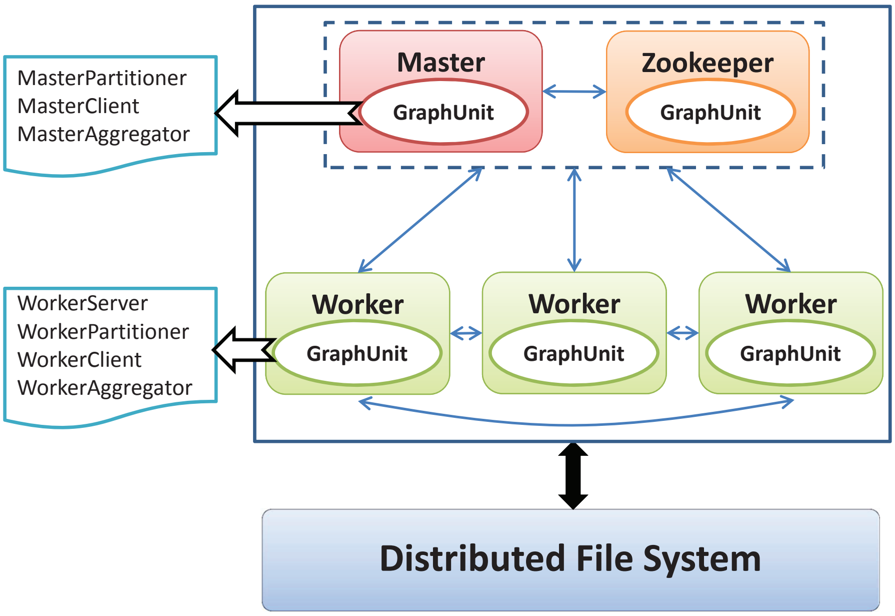


Figure 4.1: The Architecture of epiCG

The program terminates when all the vertices become inactive or the number of supersteps reach a predefined upper bound. During each superstep, every active vertex can process messages sent by other vertices in the last superstep, execute *compute()* function, update its value and forward messages to other vertices (to be processed in the next superstep). A global synchronization will be performed at the end of each superstep.

Architecture. Figure 4.1 shows the architecture of epiCG. epiCG deploys the distributed file system (DFS) as its underlying storage. Typically, DFS includes the initial graph data to be processed by epiCG as well as the final results produced by epiCG. epiCG adopts master-worker architecture which consists of three components. The first component is the worker, which performs graph computation with other workers concurrently. The second component is the master, which coordinates the execution among all the workers. The third component is the zookeeper, which maintains information shared by the master and worker, such as which workers have finished the execution of the current superstep, how many worker have dumped a checkpoint, the health status of each worker and so on. By default, we use two compute nodes to execute the master thread and zookeeper thread respectively. The core for each of the three components is a `GraphUnit`, which defines the execution logic of the worker/master/zookeeper. The details are listed as follows.

- `GraphUnit` in the master side is responsible for two tasks: (1) partition and distribute the input graph among the workers, and (2) coordinate the workers to perform supersteps synchronously. To perform these tasks correctly, `GraphUnit` in the master maintains several important objects: (1) MasterPartitioner: generate the partitioning of the input graph and decide which partition is assigned to which worker; (2) MasterClient: notify the workers on the newly computed global aggregated values; (3) MasterAggregator: retrieve local aggregated values from the workers and generate the global aggregated ones.

- `GraphUnit` in the worker side consists of four parts: (1) WorkerServer: retrieves and manages the graph data that is assigned to the worker; (2) WorkerPartitioner: maintains partition information for the vertices residing in the worker; (3) WorkerClient: forwards messages to the zookeeper and other workers; (4) WorkerAggregator: computes aggregated values and writes to the zookeeper. Based on these parts, each worker can do the following tasks: (1) retrieve its graph data; (2) loop over vertices residing in the slave and execute *compute()* function; (3) forward messages generated during the computation; (4) generate aggregated values and write to the zookeeper.

- `GraphUnit` in the zookeeper side controls the start and stop of zookeeper service and monitors the health and working status of the workers.

Once a graph job is submitted to epiCG, all the `GraphUnits` will be activated immediately. At the beginning of the execution, epiCG sets up pairwise connections between the `GraphUnits` from the master, the workers and the zookeeper. This is different from epiC where units cannot communicate with each other directly, but rely on the message service provided by the master. Specially, if a unit A want to send a message to another unit B, A has to send a message to the master first and tell the master to forward this message to B. As we mentioned before, most graph applications such as PageRank and shortest path computation always involve a large number of messages. Setting up direct connections between units allows them to communicate more efficiently and hence reduces the network burden in the master side.

In what follows, we provide implementation details of epiCG and present how graph job is executed in epiCG.

4.3 Implementation Details

epiCG supports two graph partitioning methods: edge-cut and vertex-cut. Edge-cut partitioning distributes vertices among the compute nodes. When a vertex is assigned to a compute node, all of its outgoing edges are assigned to that node as well. Different from edge-cut, vertex-cut partitioning assigns edges to the compute nodes in a random fashion. For a vertex that spans n nodes, we maintain n copies of the vertex. One copy is elected as the *master* which receives messages, performs computation and forwards updated values to the other copies which are known as *mirrors*. Each copy maintains a subset of edges of the vertex in the original graph. To eliminate network communication cost for sending messages from copies to their neighbors, vertex-cut requires that for any edge $\langle v, u\rangle$, the master version of u and $\langle v, u\rangle$ must be assigned to the same worker.

We next introduce the data structures used for both edge-cut and vertex-cut partitioning in epiCG.

4.3.1 Distributed Graph Structure

Given a graph as input, each worker is responsible for a subgraph, which will be processed by the GraphUnit in that worker. In epiCG, we organize all the vertices into partitions, and hence the subgraph in each worker consists of several partitions. Organizing subgraph into partitions has an important advantage, which is to support dynamic repartitioning. More specifically, each worker can collect statistics such as the execution time for each partition it owns and report this information to the master. The master retrieves all the statistics from the workers and may ask some of them to exchange partitions for the purpose of load balance. Instead of repartitioning the graph in the vertex granularity, partition-based repartitioning is obviously more cost effective.

Table 4.1 lists several important data structures managed by each worker to maintain its subgraph. `Vertex` records the information for a vertex in the input graph, including the identity of the vertex, its value, outgoing edges, status

Table 4.1: Graph-related objects maintained by each worker

Object	Description
`Vertex`	information of a vertex in the input graph
`Partition`	a set of vertex in a partition
`PartitionStore`	a set of partition residing in the worker
`PartitionOwner`	indicating a partition belongs to which worker
`PartitionOwnerList`	partition-worker mapping

(active or inactive). For each vertex, we also attach two variables, isMaster and #allEdges, to support vertex-cut graph partitioning. isMaster is a boolean variable indicating whether the vertex is a master version or a mirror; #allEdges records the total number of edges associated with the vertex in the original graph (see details in Section 4.3.2). `Partition` contains a set of vertices and each partition is associated with an identifier "pid". Typically, every worker is responsible for multiple partitions. We use `PartitionStore` to keep track of all the partitions assigned to the worker. In order to forward messages to the corresponding destination vertices, every worker needs to know in which worker and which partition every destination vertex resides. One way to do this is to maintain two mappings, one vertex-to-partition mapping φ and one partition-to-worker mapping ϕ_{p} in each worker, where φ indicates which vertex belongs to which partition and ϕ_{p} indicates which partition is assigned to which worker. If the worker wants to forward a message to vertex u, it can easily conclude that u belongs to partition $\varphi(u)$ in the worker $\phi_{\mathrm{p}}(\varphi(u))$.

However, the size of φ is proportional to the total number of vertices in the graph. To avoid large memory footprint, epiCG adopts a simple hash mapping function for φ. That is, let $\mathcal{P}$ be the set of all the partitions. For a vertex v and a partition $P \in \mathcal{P}$,

$$\varphi(v) = P \Leftrightarrow v.vid \equiv P.pid \pmod{|\mathcal{P}|} \tag{4.1}$$

Given the total number of partitions, each worker can easily derive which partition a vertex resides in. Although similar hash function can be applied to ϕ_{p}, we allow a better partitioning to be achieved. That is, the assignment of partitions can be decided before graph loading by using advanced graph partitioning tools such as Metis [49]. By default, let $\mathcal{W}$ be the set of all the workers. For a partition $P \in \mathcal{P}$ and a worker $W \in \mathcal{W}$,

$$\phi_{\mathrm{p}}(P) = W \Leftrightarrow P.pid \equiv W.workerid \pmod{|\mathcal{W}|} \tag{4.2}$$

In epiCG, we use `PartitionOwner` to keep information for a partition, i.e., this partition is assigned to which worker. Every worker maintains a list `PartitionOwnerList` (i.e., ϕ_{p}) recording `PartitionOwners` for all the partitions. During the computation, every worker can easily forward a message to the destination vertex by referring to the vertex-partition mapping φ and `PartitionOwnerList`.

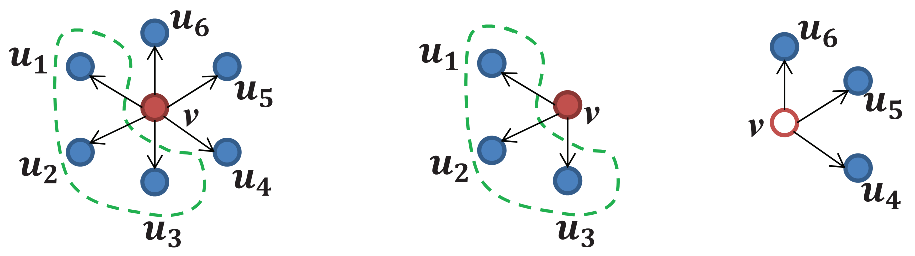
4.3.2 Graph Loading and Output

The execution of an epiCG job consists of three phases: graph loading, iterative computation and output. In this section, we mainly focus on the graph loading and output phases, but leave the discussion on iterative computation in the next section.

Graph Loading Phase

The input graph of an epiCG job is stored as a plain file in the distributed file system. Typically, a graph file consists of a set of lines, where each line contains a vertex id and all of its neighbors' ids representing the outgoing edges for the vertex. In the beginning of graph loading phase, the master will generate the PartitionOwnerList and inform all the workers of the list via zookeeper. All the workers will then retrieve the partitions that are assigned to it based on the list. To achieve this, in epiCG, every worker first reads from the distributed file system several data splits (i.e., several lines) of the input graph. The worker then iterates over the data splits and for each line, it checks whether the vertex belongs to any of its partitions. If so, the worker appends the vertex and its outgoing edges into its local partition. Otherwise, the worker forwards this vertex and the edges to its belonging worker based on the PartitionOwnerList.

To achieve a vertex-cut, we allow vertex with a large number of outgoing edges to span across multiple compute nodes. For example, consider the vertex v in Figure 4.2(a). v has 6 outgoing edges. Instead of assigning v and all its edges to one compute node, we can split its edges into two parts, as shown in Figure 4.2(b). One part contains 3 edges of v, i.e., $v \rightarrow u_1, v \rightarrow u_2, v \rightarrow u_3$



(a) Vertex v

(b) An vertex-cut of v

Figure 4.2: An example of vertex-cut

and the other part contains the rest three edges. We assign these two parts to two different compute nodes, W_1 and W_2 respectively, making vertex v span across two compute nodes. We may decide one replica of v (associated with $v \to u_1, v \to u_2, v \to u_3$) to be the master version, and the other replica (associated with $v \to u_4, v \to u_5, v \to u_6$) to be the mirror. Moreover, since epiCG organizes vertices into partitions, we need to know the two parts actually belong to which partitions in the workers W_1 and W_2, respectively. In summary, to generate vertex-cut partitions, we need to perform the following three steps for each vertex in the input graph.

1. First, we generate a vertex-cut for the vertex, i.e., split the edges into several parts and assign each part to a worker.

2. Second, we choose one copy of the vertex as the master version, while the others are the mirrors.

3. Third, we decide each part in the vertex-cut belongs to which partition of the worker.

We next provide details for the above three steps.

Vertex-cut generation. PowerGraph [36] proposed a greedy strategy for vertex-cut partitioning. However, their approach requires a single node to load the entire graph into main memory, and then place the $i + 1$ edge based on the previously placed i edges. This approach has two drawbacks. First, it has a minimum requirement on the RAM of the single compute node. Second, all the other compute nodes become idle but wait to receive its assigned edges.

In epiCG, we adopt a light-weight vertex-cut generation algorithm which allows all the workers to generate a vertex-cut for a subset of vertices. Recall that in the loading phase, each worker would first load several data splits from the DFS. When a worker reads one line in the split, it can learn from the PartitionOwnerList that the destination vertex of each edge will be assigned to which worker, and then group the edges based on the belonging workers of the destination vertices. Consider the vertex v in Figure 4.2(a). Suppose u_1, u_2, u_3 belong to the same worker W_1 and u_4, u_5, u_6 belong to another worker W_2. We can obtain two groups of edges for v naturally (see Figure 4.2(b)) and both groups compose a vertex-cut for v. We then forward v and three of its edges $v \rightarrow u_1, v \rightarrow u_2, v \rightarrow u_3$ to worker W_1. Similarly, we forward v and three edges $v \rightarrow u_4, v \rightarrow u_5, v \rightarrow u_6$ to worker W_2. More formally, for any edge $\langle v, u\rangle \in \mathcal{E}$, we assign it to worker W iff:

$$\phi_{\mathrm{p}}(\varphi(u)) = W \tag{4.3}$$

It is worth noting that the assignment of the edges is decided by the vertex-to-partition mapping φ and PartitionOwnerList maintained in each worker. Hence, all the workers could decide how to generate a vertex-cut (i.e., assign edges) for the vertices in its data splits independently.

Master selection. Let $\{(v_1, \mathcal{E}_1), \cdots, (v_n, \mathcal{E}_n)\}$ be a vertex-cut for vertex v, where v_i is a copy of v and $\mathcal{E}_i$ is the outgoing edges of v_i, $1 \leq i \leq n$. We denote by W_i the worker to which $(v_i, \mathcal{E}_i)$ is assigned. According to the vertex-to-partition mapping φ and PartitionOwnerList ϕ_{p}, vertex v is assigned to worker $\phi_{\mathrm{p}}(\varphi(v))$. Hence, we choose copy v_i in worker W_i as the master version iff:

$$\phi_{\mathrm{p}}(\varphi(v)) = W_i \tag{4.4}$$

, where φ is defined in Equation 4.1 and ϕ_{p} is defined in Equation 4.2. All of the other copies become the mirrors.

Partition selection. In epiCG, each worker organizes its subgraph into several partitions. Therefore, in addition to the worker information, we need to know each part of a vertex-cut belongs to which partition of the worker. Let $\{(v_1, \mathcal{E}_1), \cdots, (v_n, \mathcal{E}_n)\}$ be a vertex-cut for vertex v and $(v_i, \mathcal{E}_i)$ is assigned to worker W_i, $1 \leq i \leq n$. Suppose $v_j (1 \leq j \leq n)$ is the master version. According to the PartitionOwnerList, we know the set $\mathcal{P}_i$ of partitions that belong to the worker W_i. In fact, $\mathcal{P}_i$ corresponds to the PartitionStore maintained by worker W_i. For the master version v_j, $(v_j, \mathcal{E}_j)$ is assigned to partition $P \in \mathcal{P}_j$ in Worker

W_j iff:

$$P = \varphi(v) \tag{4.5}$$

This is because both φ and ϕ_{p} decide the assignment for all the master vertices. For each mirror $v_i (i \neq j)$, we assign $(v_i, \mathcal{E}_i)$ to partition $P \in \mathcal{P}_i$ in worker W_i iff:

$$P = \mathcal{P}_i[0] \tag{4.6}$$

That is, we assign each mirror v_i to the first partition maintained in the PartitionStore of worker W_i. Here, we may assign $(v_i, \mathcal{E}_i)$ to any one of partitions in worker W_i. We choose the first partition in the PartitionStore only for the implementation purpose.

Another important issue of vertex-cut generation is that we should avoid spanning vertices with low out-degrees (i.e., the total number of the outgoing edges) . This is because the copies of these vertices increase network cost for synchronizing vertex values, while their contributions to reducing the cost of forwarding cross-node message is less significant. As argued in [27], distributing edges in a random fashion will result in all the vertices having several copies, thus degrading system performance. To address the problem, we set a threshold θ on the out-degrees of the vertices. If the out-degree of a vertex is no larger than θ, we assign all the edges to the vertex without generating any copies for the vertex. If the out-degree of a vertex exceeds θ, we decompose it into several parts to obtain a vertex-cut.

Algorithm 5 summarizes the pseudo code of our vertex-cut generation algorithm. Given a vertex v, we first check whether its out-degree exceeds threshold θ (line 3). If not, we assign all the edges to the vertex and the vertex will be forwarded to partition $\varphi(v)$ using Equation 4.1 (line 4). Otherwise, we start to assign the edges of v among the workers (line 5-15). Specifically, we maintain a map function N from workers to the copies indicating which worker owns which copy (line 7) and iterates over all the outgoing edges of the vertex (line 8). For each edge $\langle v, u \rangle$, we first compute the worker W where u resides (line 9). We then check whether N contains worker W (line 10-11). If it contains W, we retrieve from N the copy v' that is assigned to W and attach $\langle v', u \rangle$ to the edge list of v'. If N does not contain worker W, we create a copy v' and assign edge $\langle v', u \rangle$ to the copy. Besides, we add the pair (W, v') to N (line 13). We next select one copy as the master version using Equation 4.4 and for

Algorithm 5: GenerateVertexCut

```
   Input  : $v$, a vertex
            $\phi_{\text{p}}$, PartitionOwnerList
            $n$, the number of partitions
   Output: $M$: list of (copy, partition) pairs
 1 $M \leftarrow \emptyset$;
 2 $\mathcal{E}_v \leftarrow v$.GetEdges ();
 3 if $|\mathcal{E}_v| \leq \theta$ then
 4     $M \leftarrow \{\langle v, \texttt{GetPartition}(v.vid)\rangle\}$;
 5 else
 6     /* generate vertex-cut for $v$ */
 7     $N \leftarrow \emptyset$;
 8     foreach Edge $\langle v, u\rangle \in \mathcal{E}_v$ do
 9         $W \leftarrow \texttt{GetWorkerInfo}(\texttt{GetPartition}(u.vid), \phi_{\text{p}})$;
10         $v' \leftarrow N.get(W)$;
11         if $v' = null$ then
12             $v' \leftarrow \texttt{CreateVertex}(v)$;
13             $N \leftarrow N \cup \{(W, v')\}$;
14         $v'.\texttt{AddEdge}(\langle v', u\rangle)$;
15     /* select master version and assign vertex-cut to workers */
16     $W^* \leftarrow \texttt{GetWorkerInfo}(\texttt{GetPartition}(v.vid), \phi_{\text{p}})$;
17     $v_m \leftarrow null$; $\mathcal{P}_m \leftarrow \emptyset$;
18     foreach $(W, v') \in N$ do
19         if $W = W^*$ then
20             $v'.isMaster \leftarrow true$;
21             $M \leftarrow M \cup \{v', \texttt{GetPartition}(v.vid)\}$;
22             $v_m \leftarrow v'$;
23         else
24             $v'.isMaster \leftarrow false$;
25             $P \leftarrow \texttt{GetAllPartitions}(W).get(0)$;
26             $M \leftarrow M \cup \{(v', P)\}$;
27             $\mathcal{P}_m \leftarrow \mathcal{P}_m \cup \{P.pid\}$;
28         $v'.\#allEdges \leftarrow |\mathcal{E}_v|$;
29     $v_m.\texttt{AddMirrorPartitionIds}(\mathcal{P}_m)$;
```

each pair (W, v') in N, we assign v' to one of the partitions in worker W using Equation 4.5 and 4.6 (line 16-27). Furthermore, for each copy of v, we record the out-degree of v in the original graph (line 28) and for the master version, we record a list of partition ids indicating the locations of the mirrors (line 29).

This is easy to verify that the vertex-cut partitioning produced by Algorithm 5 has the following properties.

Property 4.1. *Consider a vertex v and the set $\mathcal{E}_v$ of its outgoing edges. Let $\{(v_1, P_1), \cdots, (v_n, P_n)\}$ be the list M produced by Algorithm 5 and $\mathcal{E}_i$ be the set of edges associated with v_i, $1 \leq i \leq n$. We have:*

1. $\forall i \neq j \in [1, n]$, $\mathcal{E}_i \cap \mathcal{E}_j = \emptyset$;
2. $\bigcup_{i=1}^{n} \mathcal{E}_i = \mathcal{E}_v$;
3. $\forall \langle v_i, u \rangle \in \mathcal{E}_i$, $\phi_{\mathrm{p}}(P_i) = \phi_{\mathrm{p}}(\varphi(u))$.

The third point guarantees forwarding messages from any copy to its neighbors does not incur any network communication cost.

In the graph loading phase, every worker can apply Algorithm 5 to the vertices in its own data splits in parallel with other workers. After obtaining a vertex-cut for a particular vertex, the worker can forward each copy and its edges to the corresponding partition based on its computed list M. After all the workers finish shuffling vertex copies and edges, every worker obtains the following information:

- for every vertex v in its own partitions, the worker knows the out-degree of the vertex in the original graph based on $v.\#allEdges$ and whether it is a master version or a mirror based on $v.isMaster$.
- for a master vertex v in its own partitions, the worker records every mirror of v resides in which partition in the list of $v.MirrorPartitionIds$ and it can compute every mirror belongs to which worker using Equation 4.2.
- for any vertex in the graph, the worker knows the master version of the vertex resides in which partition and which worker based on Equation 4.1 and 4.2.

Output Phase

When an epiCG job finishes all the supersteps, the master will ask every worker to produce output for its own partitions. Basically, every worker loops over its PartitionStore and for each partition in the store, it iterates over the vertices in the partition. If the vertex is a master version, the worker writes the id and the value of the vertex into its output file; the worker will do nothing for the mirrors.

4.3.3 Iterative Computation

epiCG performs iterative graph computation (i.e., a set of supersteps) following the vertex-centric programming model in Pregel. To cope with vertex-cut partitioning, every worker in epiCG will experience the following three phases in each superstep.

Phase 1: Mirror message delivery.

In epiCG, every master vertex will forward its updated vertex value to all of its mirrors at the end of each superstep. Once a worker receives an updated vertex value, it will store it into a `VertexValueUpdateCache`. In the beginning of a superstep, the worker will first update its mirrors with the new values received in the last superstep. Specifically, every worker will go through all of its residing vertices. If a vertex is a mirror and `VertexValueUpdateCache` contains a new value for the vertex, the worker will update its value accordingly. When all the values in `VertexValueUpdateCache` are processed, the worker will clear this cache.

After a mirror updates its value, it needs to produce and forward messages to its own neighbors. That is, a message that should be forwarded directly from the master vertex will now be produced and forwarded by the mirror instead of the master.

To achieve this, we define a new API, `produceMsg(Edge e)`, in `Vertex` class. This function will be called by the mirrors to send messages to its own neighbors. In epiCG, if a user submits a graph job and chooses vertex-cut as the partitioning method, we require the user to define `produceMsg` function to be executed by the mirrors. Figure 4.3 provides an implementation of `produceMsg` for PageRank computation. A mirror retrieves its value via `getValue()` and computes the share of its neighbor, which is the vertex value divided by the out-degree of the vertex in the original graph.

One limitation of `produceMsge` function is that for every mirror, the message it is going to create can only rely on the value of the vertex, the out-degree of the vertex in the original graph (stored in `Vertex`.#allEdges) and the information

```
public DoubleWritable produceMsg(Edge<LongWritable,NullWritable> edge) {

     int edges = getNumEdges();
     if(edges > 0) {
          double tmp = getValue().get()/edges;
          return new DoubleWritable(tmp);
     }
     else
          return null;
}
```

Figure 4.3: `ProduceMsg(Edge e)` for PageRank

(e.g., value) of the edge. To the best of our knowledge, the above information is sufficient to produce messages for most real-life graph applications including PageRank, breadth first search, graph keyword search, triangle counting, connected component computation, graph coloring, minimum spanning forest computation, k-means, shortest path, minimum cut, clustering/semi-clustering.

Each message produced by a mirror will be forwarded to the master-version destination vertex. Thanks to vertex-cut partitioning, all the messages sent by a mirror will not incur network cost, i.e., the master-version destination vertices reside in the same compute node as the mirror. Hence, each generated message will be appended by the worker to the incoming message store of the destination vertex locally.

Phase 2: master vertex computation and message delivery.

After all the mirrors finish producing and forwarding messages to their neighbors, vertex computation starts. In this phase, every worker checks its partitions in the `PartitionStore`. For each partition, the worker loops over the vertices in it. The worker performs computation for a vertex v iff (1) v is a master vertex; (2) v receives at least one messages sent by other vertices.

When a master vertex performs computation, it may produce messages. Note that a master vertex may only has a subset of the edges and hence can only send messages to its known neighbors. Every worker collects the messages produced by the master vertices during the computation and forwards them to the workers where the destination vertices reside accordingly. If the destination vertex of a message has multiple replicas, the message will only be forwarded to the master vertex, and none of the mirrors will receive any messages. In epiCG, all the workers forward their messages asynchronously.

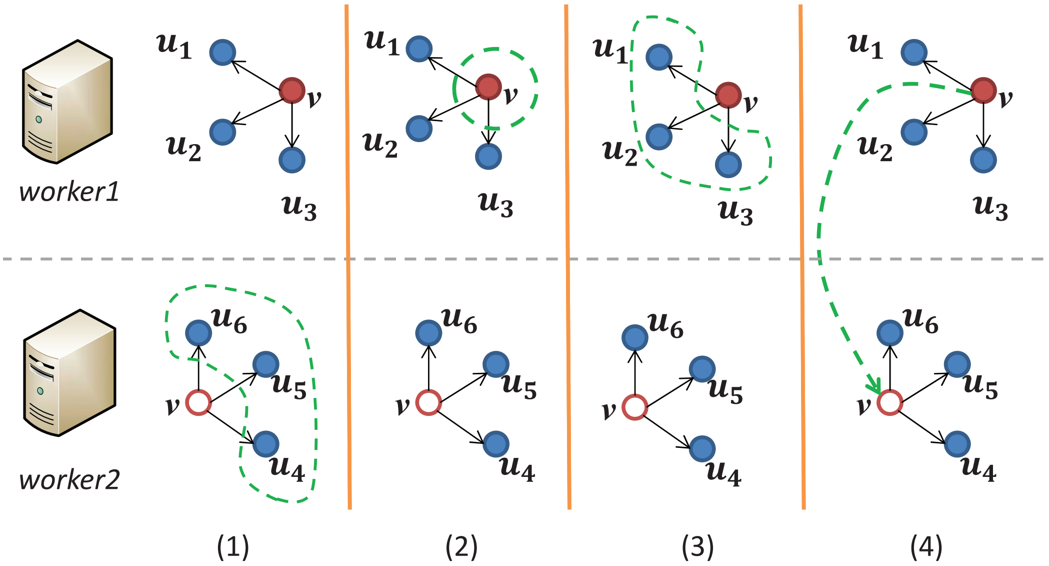


Figure 4.4: Graph computation in epiCG

Phase 3: vertex value synchronization.

When a master vertex finishes its computation, it may produce a new vertex value and all of its mirrors must be informed of this new value. In epiCG, every worker is responsible for sending the updated value of a master vertex (obtained after vertex computation) to all the mirrors. Recall that every master vertex maintains a list `MirrorPartitionIds` recording the partitions where the mirrors reside. By checking this list, the worker forwards the updated vertex value to the mirrors via `syncVertexValueRequest` requests. To ensure the correctness of vertex value synchronization (i.e., every mirror must receive updated vertex value successfully), we adopt the *three-way handshake* protocol to handle the `syncVertexValueRequest` requests. That is, when a worker W_1 forwards a `syncVertexValueRequest` request to another worker W_2, W_1 will wait for a *completion* signal sent from W_2. If W_1 does not receive the completion signal for a while, it will re-send the request. Any worker who receives a `syncVertexValueRequest` request will parse the request and put the updated value into `VertexValueUpdateCache` for the corresponding mirror. Every worker will continue its processing only if all of its requests are sent successfully and all of its received requests are handled properly.

Figure 4.4 illustrates the procedure of graph computation in epiCG. Consider a master vertex v in worker1 and its only mirror in worker2. Each of them is associated with three outgoing edges. During one superstep, worker2 first updates the value of the mirror and the mirror then computes messages by calling `produceMsg` function and forwards to its neighbors u_4, u_5, u_6 (Figure 4.4(1)). Meanwhile, worker1 performs computation for the master vertex v (Figure 4.4(2)) and produces messages to the three neighbors u_1, u_2, u_3 (Figure 4.4(3)). After that, worker1 informs worker2 of the updated value of v (Figure 4.4(4)).

4.4 Fault Tolerance

Failure detection and recovery are two key problems to achieve fault tolerance. epiCG achieves failure detection by asking every worker to register its healthy status periodically. At the end of each superstep, the master will check healthy statuses for all the workers. If a worker does not register its status over a time period, the master will regard it as *failed*.

In most existing distributed graph processing systems such as Giraph and GraphLab, the master will report error messages for the failed workers and terminate the job accordingly. To resume the execution of a failed job, existing systems require users to manually launch a new job that starts from the latest checkpoint. Apparently, such kind of implementation for failure recovery violates the requirement of fault tolerance that the system should be able to recover from failures and resume normal execution automatically.

To achieve automatic recovery, upon worker failures, epiCG leverages the remaining healthy workers to continue the execution instead of terminating the job. Specifically, the master will create a `JobState` indicating: 1) the next superstep (i.e., the latest checkpointing superstep) to perform, and 2) every worker should load graph data from the latest checkpoint in the next superstep. The master then broadcasts this state to all the healthy workers. When the next superstep starts, the master will generate a new partition-to-worker mapping based on the remaining healthy workers and informs the workers of the new mapping via zookeeper; the workers will first load graph data from the latest checkpoint, exchange vertices and edges according to the newly received partition-to-worker mapping and then start execution from the latest checkpointing superstep. By doing this, epiCG is able to resume execution from the latest checkpoint automatically. To accelerate the recovery process, we also implement a novel parallel recovery mechanism in epiCG (see details in Chapter 5).

It is important to note that the job completion time would increase dramatically when too many workers get failed. A user may prefer to terminate the job and restart with a fixed number of workers rather than ask few healthy workers to continue the execution. In epiCG, we provide a threshold `MinWorker` on the minimal number of healthy workers. `MinWorker` is set to 1 by default, but we allow users to set their own values for `MinWorker` via job configuration. Upon worker failures, the master will first check whether the number of remaining healthy workers is smaller than `MinWorker`. If not, it will perform automatic recovery as mentioned before. Otherwise, it will terminate the execution without performing recovery.

4.5 Experimental Evaluation

We evaluate the performance of epiCG by comparing it with the two most popular distributed graph processing systems [34], Giraph [2] and PowerGraph [36]. It has been experimentally shown that PowerGraph is the most efficient graph processing engine on small graphs compared with several other popular graph processing systems such as Hama and GraphX [24]. In all the experiments, we use Giraph version 1.0.0 and PowerGraph version 2.2. Our comparisons include running time, communication cost, scalability and speedup.

4.5.1 Experiment Setup

We ran all the experiments on our in-house clusters. The cluster consists of 72 compute nodes, each of which is equipped with one Intel X3430 2.4GHz processor, 8GB of memory, two 500GB SATA hard disks and gigabit ethernet. For each node in the cluster, we installed CentOS 5.5 operating system, Java 1.7.0 with a 64-bit server VM and Hadoop 0.20.203.0[2]. Giraph runs as a Map-only job on top of Hadoop, hence we made the following changes to the default Hadoop configurations: (1) the replication factor is set to 1; (2) each node is configured to run one map task. (3) the size of virtual memory for each map task is set to 4GB. By default, we chose 30 compute nodes out of the

[2] http://hadoop.apache.org/

Table 4.2: Dataset description

Dataset	Data Size	#Vertices	#Edges	Avg. Degree
Livejournal	1.0GB	3,997,962	34,681,189	8.67
Friendster	31.16GB	65,608,366	1,806,067,135	27.53
Ftiny	1.9GB	12,739,496	221,933,535	17.42
Fsmall	3.7GB	17,694,120	428,192,865	24.19
Fmedium	7.5GB	25,126,704	862,648,522	34.33

72 compute nodes for the experiments. For Giraph, one node was selected as the master running Hadoop's NameNode and JobTracker, while the remaining compute nodes were the slaves running TaskTracker daemons. Similarly, for both epiCG and PowerGraph, we chose one compute node to be the master and all the others to be the slaves; we required every node to execute only one master/slave thread and set the virtual memory size for each thread to 4GB. To make a fair comparison, we ran PowerGraph in the synchronous mode. For all the three systems, we used HDFS as the underlying distributed file system.

4.5.2 Benchmark Tasks and Datasets

We study the performance of different distributed graph processing systems using two benchmark tasks: Shortest path and PageRank.

- **Shortest path**. Shortest path computing is to select one vertex as the *source* and compute the shortest distances to the source for all the vertices. For all the three systems, we always use the same vertex as the source.

- **PageRank**. The PageRank algorithm is an iterative graph processing algorithm. We refer the readers to the original paper [68] for the details of the algorithm.

Without loss of generality, we run all the tasks for 10 supersteps and all the results are averaged over ten runs.

We conduct the experiments using several real-life datasets, downloaded from the website[3]. Table 4.2 provides the details for each of the following datasets.

[3] http://snap.stanford.edu/

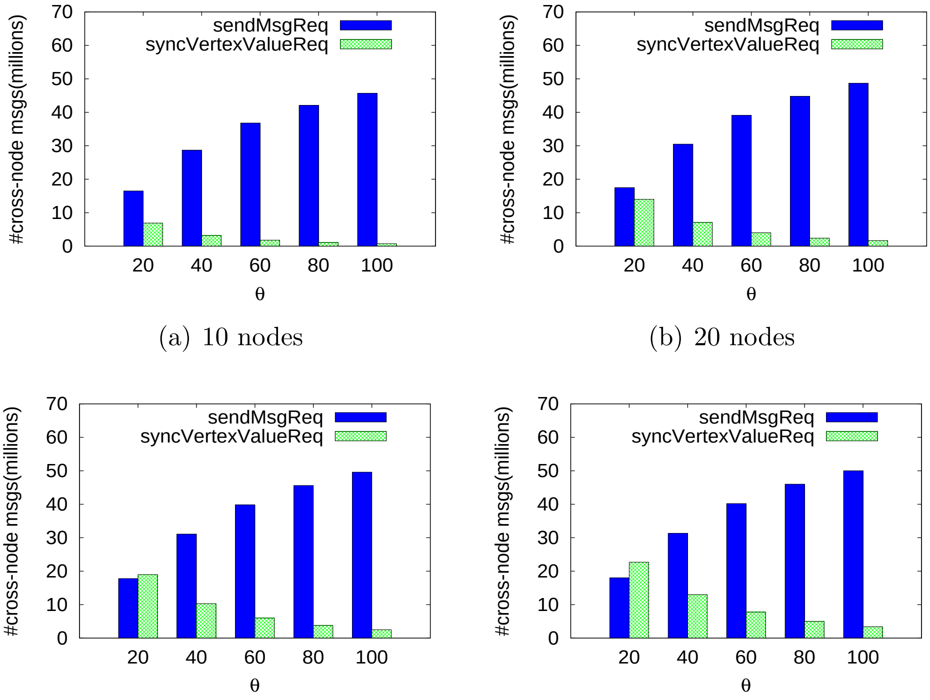
- **Livejournal**. Livejournal is an online social networking and journaling service that enables users to post blogs, journals, and dairies. It contains more than 4 million vertices (users) and over 30 million directed edges (friendships between users). We use this dataset to evaluate the execution of Shortest path tasks.

- **Friendster**. Friendster is an online social networking and gaming service. It contains more than 60 millions vertices and 1 billion edges. We use it to evaluate the execution of PageRank tasks. To evaluate the speedup of various systems, we prepare three down-samples, **Ftiny, Fsmall, Fmedium**, of Friendster by randomly selecting a subset of vertices from the original Friendster dataset and only keeping the edges involving the selected vertices.

For the experiments, we compare the performance of epiCG-E (using edge-cut), epiCG-V (using vertex-cut) with Giraph (using edge-cut) and PowerGraph (using vertex-cut) over two **metrics**: running time and communication cost. For the communication cost, we calculate the number of cross-node messages (i.e., *sendMsgRequests*) to be forwarded during the computation as well as the number of cross-node messages (i.e., *syncVertexValueRequests*) to synchronize vertex values from master vertices to their mirrors.

4.5.3 Effect of Vertex-cut Degree Threshold θ

We first study the effect of vertex-cut degree threshold θ on the performance of epiCG-V (using vertex-cut). Recall that θ decides whether we need to generate mirrors for a vertex. Figure 4.5(a)-4.5(d) show the number of cross-node requests forwarded in Shortest path task over different values of θ using 10, 20, 30, 40 compute nodes, respectively. We mainly consider two kinds of requests: *sendMsgRequest* refers to the messages forwarded during vertex computation and *syncVertexValueRequest* corresponds to the messages from master vertices to their mirrors to synchronize the values. For all the compute node numbers (i.e., 10 to 40 nodes), when θ becomes larger, the number of sendMsgRequests increases. In particular, the number of sendMsgRequests for θ =100 is around 3x larger than that for θ =20. This is because larger values of θ result in more vertices without mirrors and hence more sendMsgRequests will be forwarded



(a) 10 nodes

(b) 20 nodes

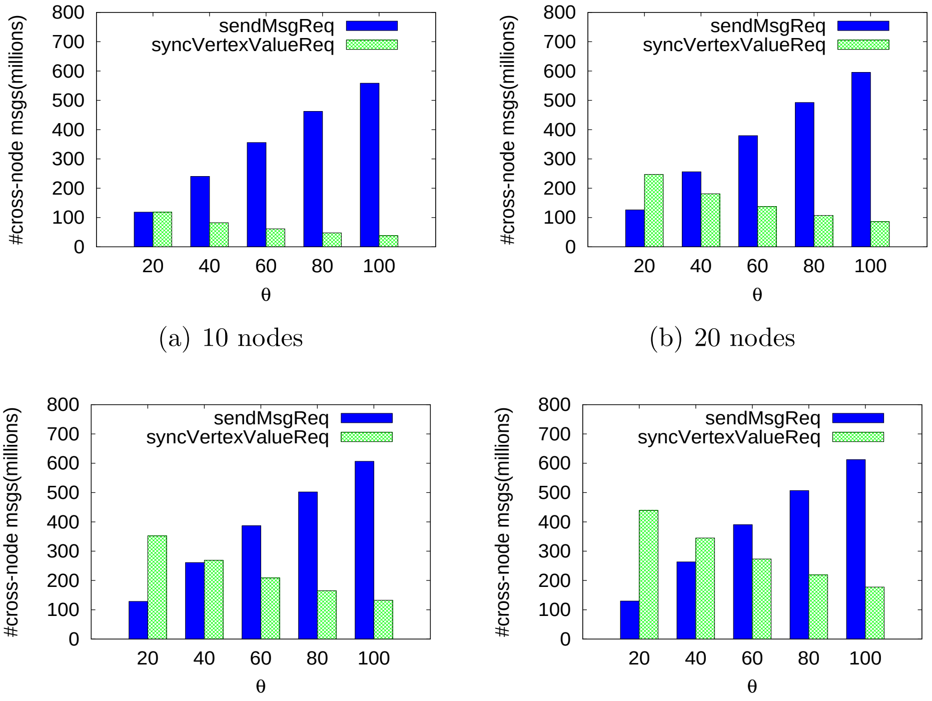
(c) 30 nodes

(d) 40 nodes

Figure 4.5: Effect of vertex-cut degree threshold θ (Shortest path)

directly from vertices to their neighbors. In contrast, the number of syncVertexValueRequests decreases as θ becomes larger, due to the fact that for larger values of θ, more vertices have no mirrors and fewer number of requests is required for vertex value synchronization. When the number of compute nodes increases from 10 to 40, the number of sendMsgRequests for each particular value of θ remains constant while the number of syncVertexValueRequests increases significantly. Specifically, for θ=20, the number of syncVertexValueRequests for 40 nodes is over 4x larger than that for 10 nodes. This is because the neighbors of a vertex can be assigned to multiple distinguished compute nodes. Therefore, the number of mirrors generated for a vertex with more than θ outgoing edges is potentially proportional to the number of compute nodes.

Figure 4.6(a)-4.6(d) show the number of cross-node requests required in PageRank task over various θ using 10, 20, 30, 40 compute nodes, respectively. Compared with Shortest path task, PageRank incurs larger numbers of sendMsgRequests and syncVertexValueRequests for all the cases. This is because we ran PageRank over Friendster dataset, which is 30x larger that Livejournal



(a) 10 nodes

(b) 20 nodes

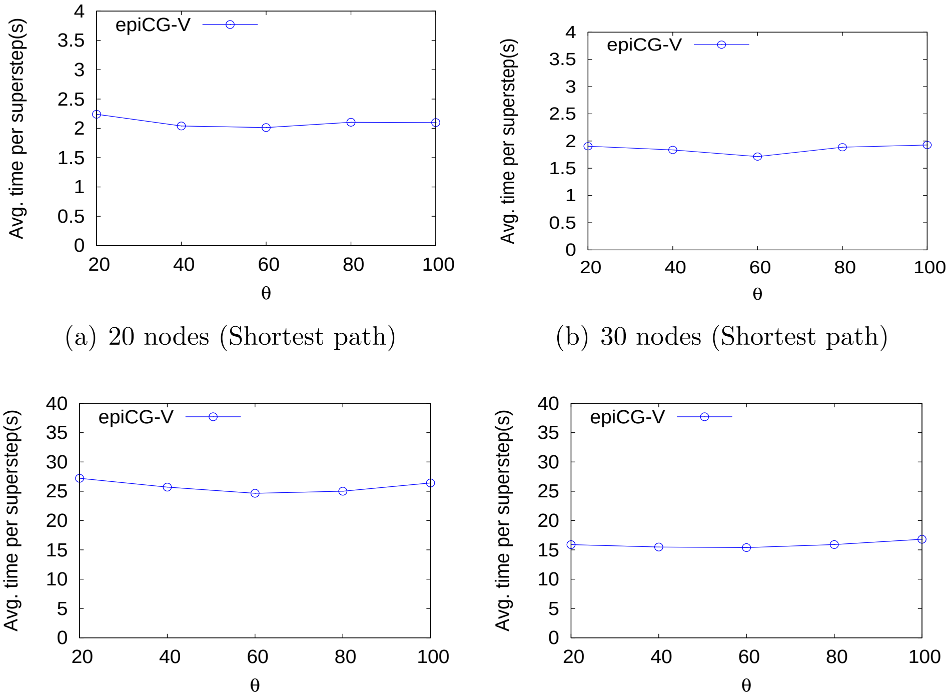
(c) 30 nodes

(d) 40 nodes

Figure 4.6: Effect of vertex-cut degree threshold θ (PageRank)

(used for Shortest path task). For all the compute node numbers, the number of sendMsgRequests increases (e.g., 5x more sendMsgRequests for θ=100 than θ=20 using 10 nodes) and that of syncVertexValueRequests decreases (e.g., 3x fewer syncVertexValueRequests for θ=100 than θ=20 using 10 nodes) as θ becomes larger. This is because larger θ, fewer vertices having mirrors and smaller number of requests used for vertex value synchronization. When the number of compute nodes becomes larger (i.e., from 10 to 40), the number of syncVertexValueRequests increases significantly for all the values of θ. As explained above, the number of mirrors is potentially a linear function of the number of the compute nodes. In the worst case, we may generate mirrors for a vertex in all the compute nodes if each node contains at least one neighbor of the vertex.

Execution time. We now present the execution time of epiCG-V over different values of θ. Figure 4.7(a) provides the execution time per superstep for Shortest path task using 20 compute nodes. As we can see, the execution time does not change too much as θ varies. For smaller values of θ, more sendMsgRequests are forwarded during the computation while the number of vertices with mirrors



(a) 20 nodes (Shortest path)

(b) 30 nodes (Shortest path)

(c) 20 nodes (PageRank)

(d) 30 nodes (PageRank)

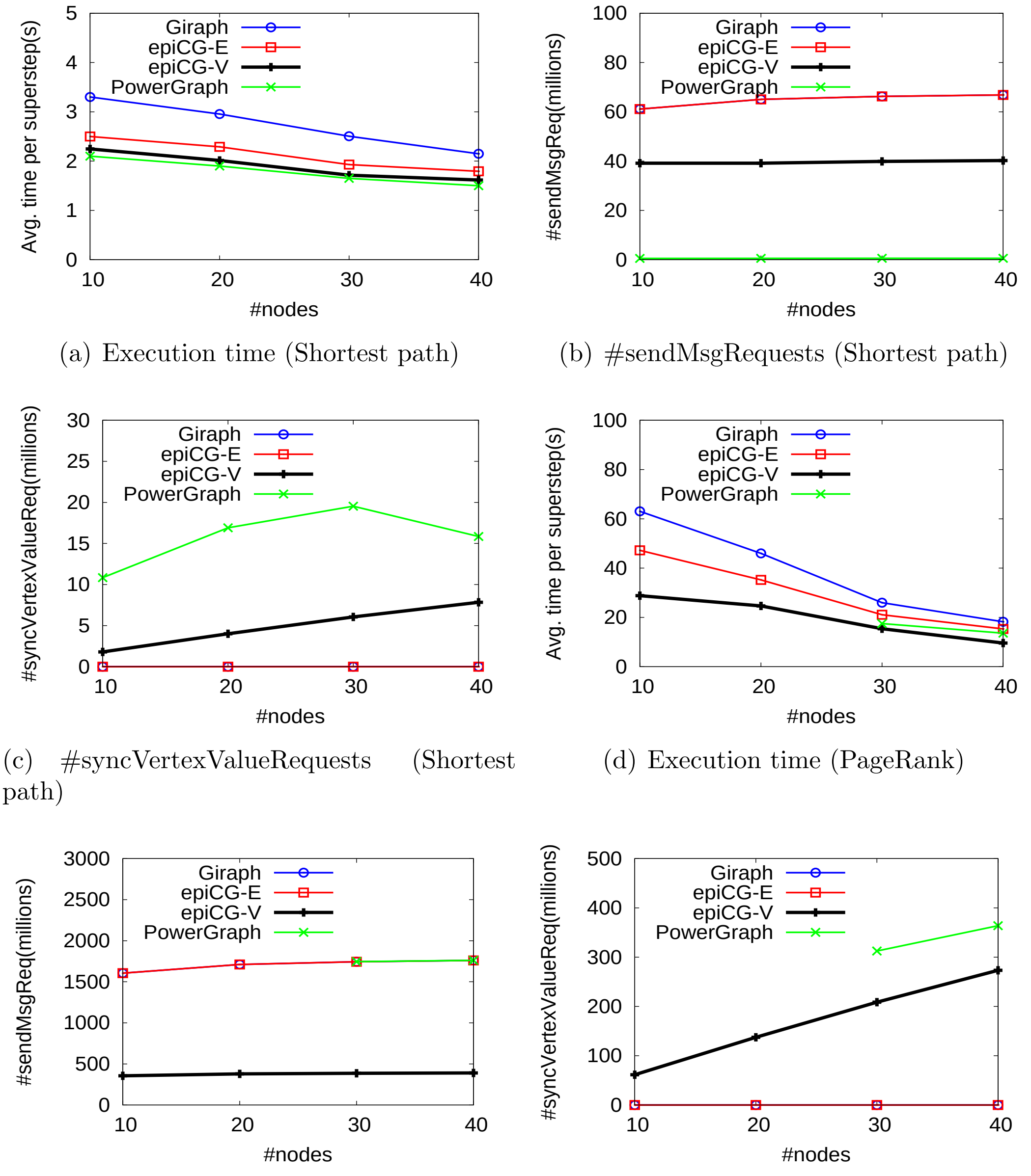
Figure 4.7: Execution time

as well as the cost for synchronizing all the mirrors is reduced. Note that every node has to generate and send messages locally for the mirrors with updated values. Hence, fewer mirrors reduce the execution time caused by the extra message generation and forwarding. Figure 4.7(b) shows the execution time per superstep using 30 nodes. Similar to 20-node cases, the execution time remains constant over various θ. We omit the results for 10 and 40 compute nodes as they exhibit similar behaviors as 20-node and 30-node cases.

Figure 4.7(c) and 4.7(d) provide the execution time per superstep for PageRank task using 20 and 30 compute nodes, respectively. For 20 nodes, the execution time of epiCG-V shows a U-shape curve as θ increases. However, the largest difference between the longest and shortest execution time is less than 5 seconds. For 30 nodes, the execution time over various θ remains constant. We observe similar results for 10 and 40 nodes and omit them to avoid redundancy.

We observe that $\theta = 60$ provides the best performance of epiCG-V in both Shortest path and PageRank tasks. Henceforth, without specified otherwise, we set θ to 60 for both Shortest path and PageRank tasks by default.

4.5.4 Scalability



(a) Execution time (Shortest path)

(b) #sendMsgRequests (Shortest path)

(c) #syncVertexValueRequests (Shortest path)

(d) Execution time (PageRank)

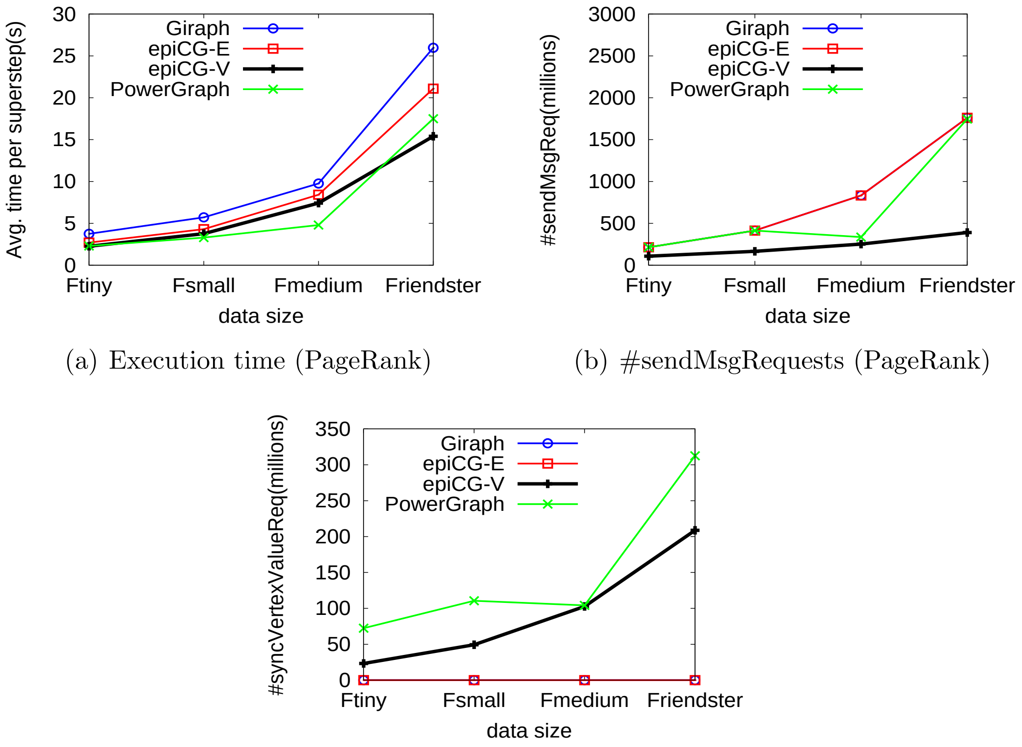
(e) #sendMsgRequests (PageRank)

(f) #syncVertexValueRequests (PageRank)

Figure 4.8: Scalability

We next evaluate the scalability of epiCG using different number of compute nodes. Figure 4.8(a) shows the execution time per superstep in Shortest path task using different number of compute nodes. As we can see, all the graph systems scales well when more compute nodes are used. PowerGraph requires the least execution time over all the compute node numbers, slightly less than epiCG-V. This is because PowerGraph uses C as its programming language, which is more efficient than Java used by epiCG. On average, epiCG-V runs 1.5x faster than Giraph, slightly faster than epiCG-E. This is because epiCG-V leverages vertex-cut to reduce the message forwarding cost during computation. As shown in Figure 4.8(b), the number of sendMsgRequests in epiCG-V is only two-thirds of that in Giraph; epiCG-E and Giraph require the same number of sendMsgRequests as they use the same partitioning function. PowerGraph requires the minimum number of sendMsgRequests over all the number of compute nodes, which is about 6x and 10x lower than that of epiCG-V and Giraph, respectively. However, as we can see in Figure 4.8(c), the number of syncVertexValueRequests forwarded in PowerGraph is over 10x larger than that in epiCG-V. In fact, the sophisticated vertex-cut partitioning approach adopted by PowerGraph balances the cost of message forwarding and that of mirror synchronization. We also observe that the advantage of epiCG-V becomes less significant when more compute nodes are used. This is because the number of mirrors increases linearly with the number of compute nodes, as illustrated in Figure 4.8(c). The cost of vertex value synchronization degrades the performance of epiCG-V.

Figure 4.8(d) provides the execution time per superstep in PageRank task. The execution time in all the graph systems decreases almost linearly as the number of compute nodes increases from 10 to 40. Compared with Shortest path task, PageRank task requires 10x more execution time due to the large size of the Friendster dataset. We only obtain the results of PowerGraph using 30 and 40 nodes. This is because PowerGraph generated a large number of mirrors which exhausted the memory space quickly. epiCG-V requires the least execution time over all the compute node values. On average, epiCG-V runs over 2x and 1.5x faster than Giraph and epiCG-E, respectively. This is because we tested PageRank task on Friendster dataset, which follows power-law degree distribution; vertex-cut reduces the communication cost significantly for natural graphs. For 30 and 40 nodes, epiCG-V runs slightly faster than PowerGraph. This is because the vertex-cut generated by PowerGraph incurs 3x more sendMsgRequests and 1.5x more syncVertexValueRequests than epiCG-V, as shown in Figure 4.8(e) and Figure 4.8(f). We observe that all the systems perform similarly in both Shortest path and PageRank tasks and hence in the following sections, we only report results for PageRank task.



(a) Execution time (PageRank)

(b) #sendMsgRequests (PageRank)

(c) #syncVertexValueRequests (PageRank)

Figure 4.9: Speedup

4.5.5 Speedup

We now study the performance of epiCG using 4 datasets in various sizes, i.e., Ftiny, Fsmall, Fmedium and Friendster (see details in Table 4.2). Figure 4.9(a) shows the execution time per superstep required in all the datasets using 30 compute nodes for PageRank task. All the systems require longer execution time when the size of dataset increases. PowerGraph requires the least execution time on the smaller graphs (i.e., Ftiny, Fsmall, Fmedium), but is less efficient than epiCG-V on the large graph (i.e., Friendster). Furthermore, epiCG-V shows the highest speedup. In particular, the ratio between the time for the largest dataset (i.e., 31GB Friendster) and that for the smallest dataset (i.e., 1.9GB Ftiny) in epiCG-V is 6.7, while the ratios for PowerGraph, epiCG-V and Giraph are 7.6, 7.79 and 6.92, respectively. For all the datasets, epiCG-V requires less execution time than Giraph and epiCG-E and the advantage becomes more significant. For Friendster, epiCG-V runs 1.7x and 1.3x faster than Giraph and epiCG-E. Figure 4.9(b) and 4.9(c) provide the number of sendMsgRequests and that of syncVertexValueRequests over different datasets, respectively. As we can see, PowerGraph requires more sendMsgRequests and syncVertexValueRequests than epiCG-V over all the datasets. The number of sendMsgRequests in both Giraph and epiCG-E increases linearly with the size of the dataset, while epiCG-V requires almost the same number of sendMsgRequests over all the datasets. We also observe that the number of syncVertexValueRequests in epiCG-V becomes larger when the size of dataset increases. More specifically, the number of syncVertexValueRequests in epiCG-V grows linearly with the size of the dataset. This is because larger datasets contain more vertices with over θ (i.e., 60) outgoing edges than the smaller ones.

4.6 Summary

In this chapter, we present our distributed graph processing engine epiCG. We develop epiCG as one extension of epiC to avoid extra configuration for a new system. epiCG supports both edge-cut and vertex-cut partitioning methods. For vertex-cut, we propose a light-weight approach to parallelize the processing of vertex-cut generation. epiCG also allows automatic failure detection and recovery. The experiments on real-life datasets illustrate the high efficiency and scalability of epiCG, compared with two state-of-the-art distributed graph processing systems, Giraph and PowerGraph.

CHAPTER 5

Failure Recovery in epiCG

Distributed graph processing systems increasingly require many compute nodes to cope with the requirements imposed by contemporary graph-based Big Data applications. However, increasing the number of compute nodes increases the chance of node failures. Therefore, provisioning an efficient failure recovery strategy is critical for distributed graph processing systems. In this chapter, we propose a novel recovery mechanism for distributed graph processing systems that parallelizes the recovery process. The key idea is to partition the part of the graph that is lost during a failure among a subset of the remaining nodes. To do so, we augment the existing checkpoint-based and log-based recovery schemes with a partitioning mechanism that is sensitive to the total computation and communication cost of the recovery process. We implement our recovery method and conduct extensive experiments in epiCG to validate the performance of our proposed method.

5.1 Introduction

Graphs capture complex relationships and data dependencies, and are important to Big Data applications such as social network analysis, spatio-temporal analysis and navigation, and consumer analytics. MapReduce was proposed as a programming model for the Big Data about a decade ago, and since then, many MapReduce-based distributed systems have been designed for Big Data applications such as large-scale data analytics. However, in recent years, MapReduce has been shown to be ineffective for handling graph data, and several new systems such as Pregel [59], Giraph [2], GraphLab [36, 57], and Trinity [76] have been recently proposed for scalable distributed graph processing.

With the explosion in graph size and increasing demand of complex analytics, graph processing systems have to continuously scale out by increasing the number of compute nodes, in order to handle the load. But scaling the number of nodes has two effects on the failure resilience of a system. First, increasing the number of nodes will inevitably lead to an increase in the number of failed nodes. Second, after a failure, the progress of the entire system is halted until the failure is recovered. Thus, a potentially large number of nodes will become idle just because a small set of nodes have failed. In order to scale out the performance continuously when the number of nodes increases, it is becoming crucial to provision the graph processing systems with the ability to handle the failures effectively.

The design of failure recovery mechanisms in distributed systems is a nontrivial task, as they have to cope with several adversarial conditions. Node failures may occur at any time, either during normal job execution, or during recovery period. The design of a recovery algorithm must be able to handle both kinds of failures. Furthermore, the recovery algorithm must be very efficient because the overhead of recovery can degrade system performance significantly. To a certain extent, due to the long recovery time, failures may occur repeatedly before the system recovers from an initial failure. If so, the system will go into an endless recovery loop without any progress in execution. Finally, the system must cope with the failures while maintaining the recovery mechanism transparent to user applications. This implies that the recovery algorithm can only rely on the computation model of the system, rather than any computation logic applied for specific applications.

The usual recovery method adopted in current distributed graph processing systems is *checkpoint-based* [56, 59, 91]. It requires each compute node to periodically and synchronously write the status of its own subgraph to a stable storage such as the distributed file system as a checkpoint. Upon any failure, checkpoint-based recovery employs an unused healthy compute node to replace each failed node and requires all the compute nodes to load the status of subgraphs from the most recent checkpoint and then synchronously re-execute all the missing supersteps. A failure is recovered when all the nodes finish the computation of the superstep in which the failure occurs. Note that the recomputation will be replayed again whenever a further failure occurs during recovery.

Although checkpoint-based recovery is able to handle any node failures, it potentially suffers from high recovery latency. The reason is two-fold. First, checkpoint-based recovery replays the missing supersteps for the whole graph, residing in both failed and healthy compute nodes, based on the most recent checkpoint. This could incur high computation cost as well as high communication cost, including loading the whole checkpoint, performing recomputation and passing the messages among all compute nodes in each missing superstep. Second, when a further failure occurs during the recovery, the lost computation caused by the previous failure may have been partially recovered. However, checkpoint-based recovery will forget about all these partially completed workload, rollback every compute node to the latest checkpoint and replay the computation since then. This eliminates the possibility of performing the recovery progressively.

In this chapter, we propose a new recovery scheme to enable fast failure recovery. The key idea is to 1) *restrict the recovery workload to the subgraphs residing in the failed nodes* using locally logged messages; 2) distribute the subgraphs residing in the failed nodes among a subset of compute nodes to *redo the lost computation concurrently.* In our recovery scheme, in addition to global checkpointing, we require every compute node to locally log their outgoing messages at the end of each superstep. Upon a failure, the system first replaces each failed node with a new one. It then divides the subgraphs residing in the failed nodes into partitions, referred to as *failed* partitions, and distributes these partitions among a subset S of compute nodes. During recovery, every node in S will hold its original subgraph and load the status of its newly received partitions from the latest checkpoint. When the system re-executes missing supersteps, the recomputation is confined to the failed partitions by nodes in S concurrently, using logged messages from healthy subgraphs and recalculated ones from failed partitions. To distribute the lost subgraphs effectively, we propose a computation and communication cost model to quantify the recovery time, and according to the model, we split the lost subgraphs among a subset of compute nodes such that the total recovery time is minimized.

To the best of our knowledge, this is the first parallel recovery mechanism proposed for distributed graph processing. In contrast with traditional checkpoint-based recovery, our approach eliminates the high recomputation cost for the subgraphs residing in the healthy nodes due to the fact that failures often occur among a small fraction of compute nodes. Note that the subgraph in a healthy node can include both its original subgraph (whose computation is never lost) and a set of newly received partitions (whose computation is partially recovered) due to previous failures. Furthermore, we distribute the recomputation tasks for the subgraphs in the failed nodes among multiple compute nodes to achieve better parallelism. Thus, our approach is not a replacement for checkpoint-based recovery methods. Instead, it complements them because it accelerates the recovery process through simultaneous reduction of recovery communication costs and parallelization of the recovery computations.

Our contributions of this chapter are summarized as follows.

- We formally define failure recovery problem in distributed graph processing systems and introduce a partition-based failure recovery method that can efficiently handle any node failures, either during normal execution or during recovery period (Section 5.2 and 5.3).

- We formalize the problem of distributing recomputation tasks for subgraphs residing in the failed nodes as a reassignment generation problem: find a reassignment for failed partitions with minimized recovery time. We show the problem is NP-hard and propose a cost-sensitive reassignment algorithm (Section 5.4).

- We implement our proposed parallel recovery method on top of epiCG (Section 5.5) and conduct extensive experiments on real-life datasets using both synthetic and real applications. Our experiments show our proposed recovery method outperforms traditional checkpoint-based recovery by a factor of 12 to 30 in terms of recovery time, and a factor of 38 in terms of the network communication cost using 40 compute nodes (Section 5.6).

5.2 Preliminaries

In this section, we provide some background of epiCG, define our problem and discuss the challenges of failure recovery in epiCG. Table 5.1 lists the symbols

Table 5.1: Notations used throughout Chapter 5

Symbol	Definition
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$
N	compute node
$\mathcal{V}_N$	vertices that reside in node N
$\mathcal{P}$	graph partitions
$\mathcal{N}_{\mathrm{f}}$	failed nodes
s_{f}	superstep that a failure occurs
F	failure
F^i	i-th cascading failure for F
S	state
φ	vertex to partition mapping
ϕ_{p}	partition to node mapping
ϕ_{r}	failed partition to node mapping (reassignment)

and their meaning used throughout this chapter.

5.2.1 Background of epiCG

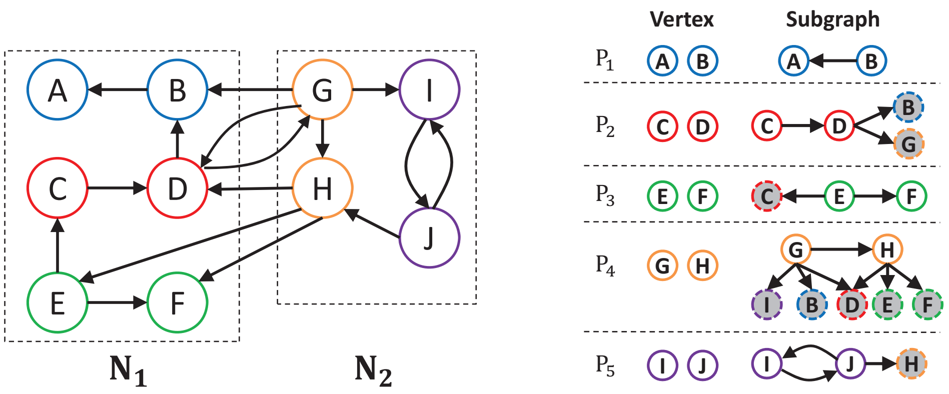
Distributed Graph

The input to epiCG is a directed graph[1] $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ are the sets of vertices and edges, respectively. Every vertex in the graph has a unique vertex identifier. In epiCG, the set of vertices is divided into partitions. A partition of $\mathcal{G}$ is formally denoted by $P_i = (V_i, E_i)$, where $V_i \subseteq \mathcal{V}$ and $E_i = \{\langle v_i, v_j\rangle \in \mathcal{E} | v_i \in V_i\}$. Note that E_i includes all the outgoing edges from vertices in V_i, which may cross partitions. All the partitions are distributed among compute nodes, i.e., physical machines.

Let $\mathcal{P}$ and $\mathcal{N}$ respectively be the set of partitions and the set of compute nodes. Typically, the number of partitions is larger than that of compute nodes (i.e., $|\mathcal{P}| > |\mathcal{N}|$), to achieve a better load balance. For ease of illustration, we denote by $\varphi, \phi_{\mathrm{p}}$ two mappings, where (vertex-partition mapping) $\varphi : \mathcal{V} \rightarrow \mathcal{P}$ records every vertex belongs to which partition and (partition-node mapping) $\phi_{\mathrm{p}} : \mathcal{P} \rightarrow \mathcal{N}$ records every partition resides in which compute node. For any node $N \in \mathcal{N}$, we denote by $\mathcal{V}_N$ the set of vertices residing in N.

Figure 5.1(a) shows a distributed graph $\mathcal{G}$ over two nodes N_1, N_2. $\mathcal{G}$ is

[1] Undirected graphs can be represented as directed graphs where for every edge $\langle u, v\rangle$ there is a corresponding edge $\langle v, u\rangle$



(a) $\mathcal{G}(\mathcal{V}, \mathcal{E})$

(b) $\mathcal{P}$

Figure 5.1: Distributed graph and partitions

divided into 5 partitions P_1-P_5, as shown in Figure 5.1(b). We use colors to differentiate vertices in different partitions.

Computation Model

The computation model in epiCG follows the Bulk Synchronous Parallel (BSP) model [83]. Typically, the computation consists of an *input* phase, where a graph is distributed among the compute nodes, followed by a set of iterations, called *supersteps*, separated by global synchronization points, and finally an *output* phase. Every vertex carries two states: *active* and *inactive*. Initially (at the beginning of superstep 1), all the vertices are active. A vertex can deactivate itself by *voting to halt*. Once a vertex becomes inactive, it has no further work to do in the following supersteps unless activated by incoming messages from other vertices. Within each superstep, only active vertices participate in computation: process messages sent by other vertices in the previous superstep, update its value or the values of its outgoing edges and send messages to other vertices (to be processed in the next superstep). This kind of computation logic is expressed by a user-defined function. All the active vertices in the same compute node execute the function sequentially, while the execution in each compute node is performed in parallel with other nodes. After all the active vertices finish their computation in a superstep, a global synchronization point is reached.

Basic architecture. epiCG follows a master/slave architecture. The master is responsible for coordinating the slaves, but is not assigned any graph partitions. The slaves are in charge of performing computation over its assigned partitions in each superstep.

5.2.2 Failure Recovery in epiCG

Checkpointing Scheme

We consider synchronous checkpointing to be performed every $\mathbb{C}(\in \mathbb{N}^+)$ supersteps. At the beginning of superstep $i\mathbb{C} + 1(i \in \mathbb{N}^+)$, we flush the complete graph status into reliable storage such as distributed file system, including graph structure, vertex values, vertex status(active/inactive), edge values, incoming messages received in the previous superstep, and other auxiliary application-specific information. The saved status is called a *checkpoint*. In short, a checkpoint made in superstep $i\mathbb{C} + 1$ records the graph status after the completion of superstep $i\mathbb{C}$. We assume that no failures occur during checkpointing.

Problem Statement

We consider a graph job that is executed on a set $\mathcal{N}$ of compute nodes from superstep 1 to s_{max}. A compute node may fail at any time during the normal job execution. Let $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$ denote a failure that occurs on a set $\mathcal{N}_\mathrm{f}(\subseteq \mathcal{N})$ of compute nodes when the job performs normal execution in superstep $s_\mathrm{f}(\in [1, s_{max}])$. We associate with F two states S_F and S^*_F, which record the statuses of vertices before and after the recovery for F, respectively.

Definition 5.1 (State). *The state S is a function: $\mathcal{V} \to \mathbb{N}^+$ recording the latest superstep in which the computation is completed by each vertex at certain time.*

After F is detected, all the vertices residing in the failed nodes are lost and their latest statuses are stored in the latest checkpoint. The recovery for F is initiated after all the healthy nodes finish their execution in superstep s_f. Let $c + 1$ be the superstep when the latest checkpoint is made. We have:

$$S_F(v) = \begin{cases} c & v \in \bigcup_{N \in \mathcal{N}_\mathrm{f}} \mathcal{V}_N \\ s_\mathrm{f} & \text{Otherwise} \end{cases} \tag{5.1}$$

In general, the recovery for F is to re-execute the computation from the latest checkpointing superstep to superstep s_{f}. Hence, we have:

$$S_F^*(v) = s_{\mathrm{f}}, \qquad \forall v \in \mathcal{V} \tag{5.2}$$

We now formalize failure recovery problem as follows.

Definition 5.2 (Failure recovery)**.** *Given $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$, the recovery for F is to transform the statuses of all the vertices from S_F to S_F^*.*

Example 5.1:[**Running example**] Consider graph $\mathcal{G}$ distributed over compute nodes N_1, N_2 in Figure 5.1(a) and failure $F(\{N_1\}, 12)$, i.e., N_1 fails during the normal execution of superstep 12. Assume that every vertex is active and sends messages to all its neighbors in normal execution of each superstep, and the latest checkpoint was made in the beginning of superstep 11. S_F and S_F^* are the following.

- $\forall v \in \{A, B, C, D, E, F\}$, $S_F(v) = 10$ and $S_F^*(v) = 12$;
- $\forall v \in \{G, H, I, J\}$, $S_F(v) = 12$ and $S_F^*(v) = 12$;

The recovery for F is to transform the status of each vertex to the one achieved after the completion of superstep 12. □

Challenging Issues

Consider a failure $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$ that occurs during normal execution of a graph job. During the recovery for F, compute nodes may fail at any time. More specifically, multiple failures may occur sequentially before the system achieves state S_F^*. We refer to these failures the cascading failures for F.

Definition 5.3 (Cascading failure)**.** *Given $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$, a cascading failure for F is a failure that occurs during the recovery for F, i.e., after F occurs but before F is recovered.*

Let $\mathbb{F}$ be a sequence of all the cascading failures for F. We denote by F^i the i-th cascading failure in $\mathbb{F}$.

Challenge 1. The key challenge of recovering F is to handle cascading failures for F. To the best of our knowledge, we are not aware of any previous works that provide details on how to handle cascading failures in distributed graph processing systems.

Our goal is to speed up recovery process for F. Informally, the time of recovering F is contributed by three main tasks:

- re-execute computation such that the status of every vertex is updated to the one achieved after the completion of superstep s_{f}.
- forward inter-node messages during recomputation.
- recover cascading failures for F.

A naive recovery method re-runs the job from the first superstep upon the occurrence of each failure. Obviously, such an approach incurs long recovery time as the execution of every superstep can be costly in many real-world graph jobs. In the worst case, the failure occurs during the execution of the final superstep and the system needs to redo all the supersteps. Furthermore, it is more likely that a cascading failure will occur as the recovery time becomes longer.

Challenge 2. Given a failure F, our objective is to recover F with *minimized* the recovery time, i.e., the time span between the start and the completion of recovering F.

In what follows, we first describe how locally logged messages can be utilized for failure recovery, which is the basis of our recovery algorithm, and then discuss its limitations.

Utilizing Locally Logged Messages

Besides checkpoints, we require every compute node to log its outgoing messages at the end of each superstep. The logged messages are used to reduce recovery workload. Consider a failure $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$. Following checkpoint-based recovery method, for any failed node $N \in \mathcal{N}_{\mathrm{f}}$, we employ a new available node to replace N and assign partitions in N to it. We require all the replacements to load the status of received partitions from the latest checkpoint, and all the healthy nodes hold their original computed partitions. In superstep $i \in [c+1, s_{\mathrm{f}}]$ during recovery, only the replacements perform computation for the vertices in failed partitions, while every healthy node forwards locally logged messages to the vertices in failed partitions without any recomputation. For $i \in [c+1, s_{\mathrm{f}})$, the vertices in failed partitions forward messages to each other, but for $i = s_{\mathrm{f}}$, they send messages to those in healthy nodes as well.

Example 5.2: Continue with Example 5.1. To recover $F(\{N_1\}, 12)$, we first employ a new node to replace N_1 and then re-execute superstep $11, 12$. We refer to the new node by N_1. N_1 loads the statuses of P_1, P_2, P_3 from the latest checkpoint. During recovery, only N_1 performs computation for 6 vertices A-F in two supersteps, while the recomputation for vertices G-J is avoided. Superstep 11 incurs 5 logged inter-node messages $G \to B$, $G \to D$, $H \to D$, $H \to E$, $H \to F$. Superstep 12 incurs 6 inter-node messages: the above five logged ones plus a recalculated one $D \to G$. □

Utilizing locally logged messages helps to *confine* recovery workload to the failed partitions (in terms of both recomputation and message passing), thus reducing recovery time. Moreover, the overhead of locally logging is negligible in many graph applications as the execution time is dominated by computation and network message passing (see details in Section 5.6). However, the recomputation for the failed partitions is shared among the nodes that replace the failed ones and this achieves limited parallelism as the computation in one node can only be executed sequentially. This inspires us to *reassign* failed partitions to multiple nodes to achieve parallelism of recomputation.

5.3 Partition-based Recovery

We propose a partition-based method to solve the failure recovery problem. Upon a failure, the recovery process is initiated by the recovery executor. Figure 5.2 shows the workflow of our partition-based failure recovery executor. Let $c + 1$ be the latest checkpointing superstep for the failure. The recovery executor is responsible for the following three tasks.

• **Generating partition-based recovery plan.** The input to this task includes the state before recovery starts and the *statistics* stored in reliable storage, e.g., HDFS. We collect statistics during checkpointing, including:

(1) computation cost of each partition in superstep c.

(2) partition-node mapping ϕ_{p} in superstep c.

(3) for any two partitions in the same node, the size of messages forwarded from one to another in superstep c.

(4) for each partition, the size of messages from an outside node (where the

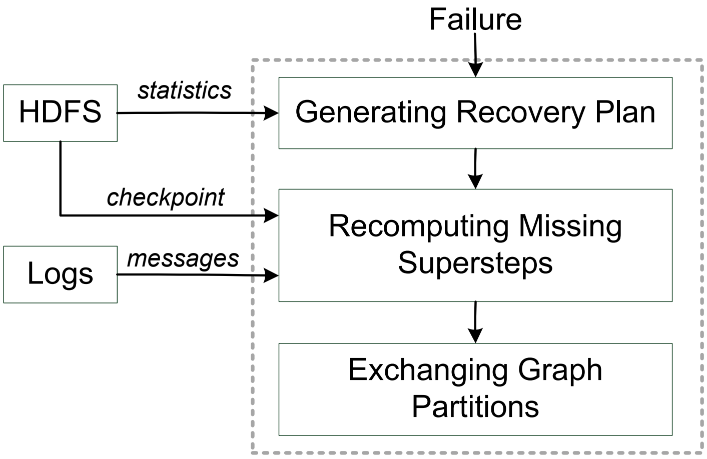


Figure 5.2: Failure recovery executor

partition does not reside) to the partition in superstep c.
The statistics require a storage cost of $O(|\mathcal{P}| + |\mathcal{P}||\mathcal{N}| + |\mathcal{P}|^2)$, which is much lower than that of a checkpoint.

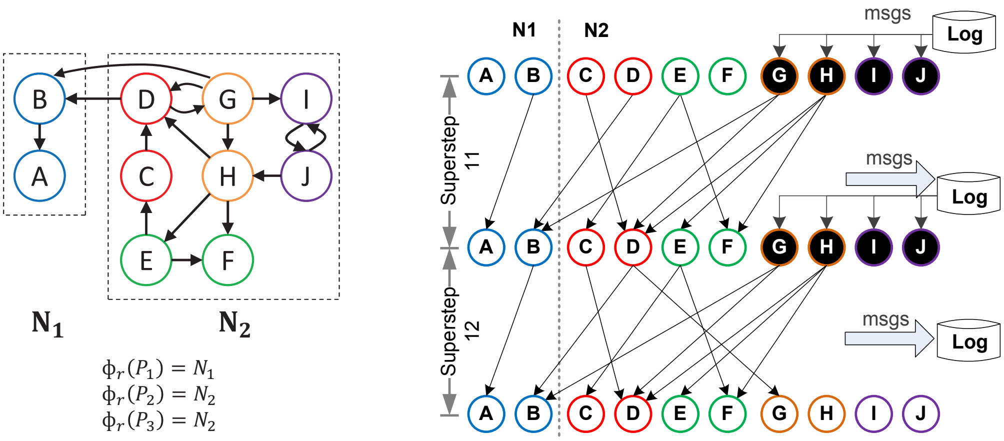
The output recovery plan is represented by a *reassignment* for failed partitions, which is formally defined as follows.

Definition 5.4 (Reassignment)**.** *For any failure, let $\mathcal{P}_\mathrm{f}$ be the set of partitions residing in the failed nodes. The reassignment for the failure is a function ϕ_r: $\mathcal{P}_\mathrm{f} \to \mathcal{N}$.*

Figure 5.3(a) shows a reassignment for $F(\{N_1\}, 12)$ in Example 5.1. We assign P_1 to N_1 (the replacement) and P_2, P_3 to N_2.

• **Recomputing failed partitions.** This task is to inform every compute node of the recovery plan ϕ_r. Each node N checks ϕ_r to see whether a failed partition is assigned to it. If so, N loads the partition status from the latest checkpoint. The status of a partition includes (1) the vertices in the partition and their outgoing edges; (2) values of the vertices in the partition achieved after the completion of superstep c; (3) the status (i.e., active or inactive) of every vertex in the partition in superstep $c + 1$; (4) messages received by the vertices in the partition in superstep c (to be processed in superstep $c + 1$). Every node then starts recomputation for failed partitions. The details are provided in Section 5.3.1.

• **Exchanging graph partitions.** This task is to re-balance the workload among all the compute nodes after the recomputation of the failed partitions



(a) Reassignment

(b) Recomputation

Figure 5.3: Recovery for $F(\{N_1\}, 12)$

completes. If the replacements have different configurations than the failed ones, we allow a new partition assignment (that is different from the one before failure occurs) to be employed for a better load balance, following which, the nodes might exchange partitions among each other.

5.3.1 Recomputing Failed Partitions

Consider a failure $F(\mathcal{N}_{\mathrm{f}}, s_{\mathrm{f}})$ that occurs during normal execution. The recomputation for the failed partitions starts from the most recent checkpointing superstep $c+1$. After all the compute nodes finish superstep j, they proceed to superstep $j+1$ synchronously. The goal of recovery is to achieve state S_F^* (see Equation 5.2). Therefore, the recomputation terminates when all the compute nodes complete superstep s_{f}.

Algorithm 6 provides recomputation details in a superstep during recovery. Consider a node N. In superstep i ($\in [c+1, s_{\mathrm{f}}]$), N maintains a list M of messages that will be sent by vertices residing in N in the current superstep. Initially, M contains all the locally logged outgoing messages for superstep i if any (line 1). N then iterates through all the active vertices residing in it and for each active vertex, N executes its computation and appends all the messages sent by this vertex to M if the vertex value has not been updated to the one achieved in the end of superstep i (line 2-5). After that, N iterates over messages in M. A message m in M is forwarded if m is needed by its destination vertex to perform recomputation in the next superstep (line 6-9).

Algorithm 6: Recomputation

```
   Input: $S$, the state when failure occurs
          $i$, current superstep
          $N$, a compute node
1  $M \leftarrow$logged outgoing messages in superstep $i$;
2  for $v \in \mathcal{V}_N$ do
3      for $v$.Active$= True$ and $S(v) < i$ do
4          Perform computation for $v$;
5          $M \leftarrow M \cup v.Sendmsgs$;
6  for $m \in M$ do
7      $v_s \leftarrow m.Getsrc(); v_d \leftarrow m.Getdst()$;
8      if $S(v_d) < i$ or $(S(v_s) < i \wedge S(v_d) = i)$ then
9          Send $m$ to $v_d$;
10 Flush $M$ into local storage;
```

Finally, N flushes M into its local storage (line 10), which will be used in case there are further cascading failures.

Example 5.3: Figure 5.3(b) illustrates recomputation for $F(\{N_1\}, 12)$, given ϕ_r in Figure 5.3(a). We use directed edges to represent the forwarding messages. In superstep 11, N_1 and N_2 respectively perform 2 and 4 vertex computations for A-F; 2 inter-node messages $D \rightarrow B$, $G \rightarrow B$ are forwarded. N_2 retrieves 4 logged messages sent by G in normal execution of superstep 11 but only re-sends messages to B, D because H, I belongs to healthy partition P_4. Further, N_1, N_2 will log 5 messages sent by A-F locally as they have not yet been included in the log. Superstep 12 performs similarly, except for an additional message $D \rightarrow G$. Compared with the approach in Example 5.2, our algorithm achieves more parallelized recomputation and incurs less network communication cost. □

Note that the messages received by the vertex during recomputation might have a different order compared with those received during normal execution. Therefore, the correctness of our recomputation logic implicitly requires the vertex computation is insensitive to message order. That is, given messages with an arbitrary order, the *effect* of a vertex computation (new vertex value and its sending messages) remains the same. This requirement is realistic since a large range of graph applications are implemented in a message-ordering independent manner. While we are not aware of any graph algorithms that are nondeterministic with respect to the message order, our recovery method can be extended easily to support such algorithms if there is any. Specifically, we can assign a unique identifier to each message. Recall that all the messages to be processed in a superstep must be completely collected by graph processing engine before any vertex computation starts. In each superstep (either during normal execution or recovery), for every active vertex v, we can sort all these messages received by v based on their identifiers, before initiating the computation. The sorting ensures the messages for a vertex computation during normal execution follow the same order as those for recomputation during recovery.

5.3.2 Handling Cascading Failures

We now consider cascading failures for $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$, which occur before F is recovered. A useful property of our partition-based recovery algorithm is that for any failure, the behavior of every compute node *only* relies on the reassignment for the failure and the state after the failure occurs. That is, in our design, given the reassignment and state for the failure, the behavior of every node is independent of what the failure is. The failure can be F itself or any of its cascading failures. Therefore, whenever a cascading failure for F occurs, the currently executing recovery program is terminated and the recovery executor can start a new recovery program for the new failure using the same recovery algorithm.

In practice, the occurrence of failures is not very frequent and hence we expect at least one recovery program to complete successfully. F is recovered when a recovery program exits normally. That is, all the vertices complete superstep s_f and S^*_F is achieved. Further, due to cascading failures, a compute node may receive new partitions during the execution of each recovery program. After recomputation finishes, nodes may exchange partitions to re-balance the workload. The following example illustrates how our recovery algorithm is used to handle cascading failures.

Example 5.4: We start with a recovery program for $F(\{N_1\}, 12)$ in Example 5.3. Suppose a cascading failure F^1 occurs in N_2 when the program is executing in superstep 12. Vertices C-J residing in N_2 are lost due to F^1, while A, B in healthy node N_1 are recovered. Hence, the state S_{F^1} after F^1 occurs

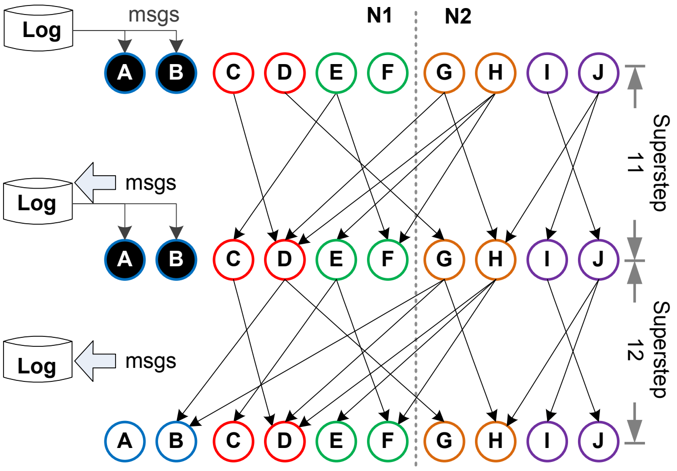


Figure 5.4: Recomputation for cascading failure F^1

satisfying: $S_{F^1}(A) = S_{F^1}(B) = 12$ and $S_{F^1}(v) = 10$ for $v = C\text{-}J$. A new recovery program is initiated for F^1. Suppose the reassignment for F^1 assigns P_2, P_3 to N_1 and P_4, P_5 to N_2 (replacement). N_1, N_2 load the statuses of newly assigned partitions from the latest checkpoint and start recomputation as shown in Figure 5.4.

Since $S_{F^1}(A) = S_{F^1}(B) = 12$, we only perform recomputation for vertices C-J in newly failed partitions P_2-P_5 when re-executing superstep $11, 12$. In superstep 11, C-J forward messages to each other. In superstep 12, these vertice sends messages to A, B as well. Suppose there is no further cascading failure after F^1. The recovery for F is accomplished upon the completion of the new recovery program triggered by F^1. □

Example 5.4 considers cascading failures that occur during recomputation. In practice, failures may occur at any time. If a failure occurs during the period of generating recovery plan for the previous failure, we treat both failures as one *bigger* failure and the union of their failed nodes as the failed node set. If a failure occurs during exchanging phase, we treat it as the one that occurs in superstep s_f. Our recovery approach can be applied to both cases.

Without loss of generality, in the rest of this chapter, we only consider cascading failures that occur in a recomputation phase.

5.3.3 Correctness and Completeness

We first focus on the *correctness* of our recovery method.

Definition 5.5 (Correctness)**.** *Let* $\textsc{Value}(v, i)$ *and* $\textsc{Msg}(v, i)$ *denote the value and the set of received messages for vertex* v *in the end of superstep* i *during normal execution, respectively. A recovery algorithm is* correct if for any failure $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$, after the recovery algorithm finishes, the value of every vertex v equals to $\textsc{Value}(v, s_\mathrm{f})$ and the set of messages receieved by v equals to $\textsc{Msg}(v, s_\mathrm{f})$.

The above definition is under the assumption that the vertex computation is deterministic with respect to message ordering, which is the case in real-world graph applications.

The correctness of our recovery algorithm is based on two properties of Algorithm 6.

Lemma 5.1. *Algorithm 6 has the following properties:*

1. *a vertex performs computation in superstep* j *iff the vertex status has not been updated to the one achieved in the end of superstep* j *during normal execution,* $\forall j \in [c + 1, s_\mathrm{f}]$.

2. *vertex* v *sends messages to* u *in superstep* j *iff* u *will perform computation in superstep* $j + 1$, $\forall j \in [c + 1, s_\mathrm{f}]$.

Lemma 5.1 holds for the recovery program triggered by any failure. In essence, Lemma 5.1 guarantees: i) the input (vertex value and received messages) of a vertex computation in any superstep during recomputation is exactly the same as that during normal execution; ii) for failure $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$, when our recovery algorithm finishes successfully, each vertex completes superstep s_f and receives the same set of messages as it does in the end of superstep s_f during normal execution. These properties ensure the correctness of our approach.

Furthermore, our recovery algorithm is *complete* in that the recovery logic is independent of high-level applications. That is, any node failure can be correctly recovered using our algorithm.

Theorem 5.1. *Our partition-based recovery algorithm is correct and complete.*

5.4 Reassignment Generation

In this section, we present how to generate reassignment for any failure. Consider a failure $F(\mathcal{N}_\mathrm{f}, s_\mathrm{f})$. The reassignment for F is critical to the overall recovery performance, i.e., the time span of recovery. In particular, it decides the computation and communication cost during recomputation. Our objective is to find a reassignment that minimizes the recovery time, denoted by $\Gamma(F)$.

Given the reassignment for F, the calculation of $\Gamma(F)$ is complicated by the fact that $\Gamma(F)$ depends not only on the reassignment for F, but also on the cascading failures for F and the corresponding reassignments. However, the knowledge of cascading failures can hardly be obtained beforehand since F and its cascading failures do not arrive as a batch but come sequentially. Hence, we seek an *online* reassignment generation algorithm that can react in response to any failure, without knowledge of future failures.

Our main insight is that when a failure (either F or its cascading failure) occurs, we prefer a reassignment that can benefit the *remaining* recovery process for F by taking into account all the cascading failures that have already occurred. More specifically, we collect the state S after the failure occurs and measure the minimum time T_{low} required to transform from S to S_F^*, i.e., the time of performing recomputation from superstep $c+1$ to s_{f} without further cascading failures. We then aim to produce a reassignment that minimizes T_{low}. Essentially, S encapsulates all the useful information about previous failures and the corresponding reassignments performed, and T_{low} provides a lower bound of remaining recovery time for F. In what follows, we introduce how to compute T_{low} and then provide our cost-driven reassignment algorithm.

5.4.1 Estimation of T_{low}

For any failure, T_{low} is determined by the total amount of computation cost and network communication cost required during recomputation, which is formally defined as follows.

$$T_{low} = \sum_{i=c+1}^{s_{\mathrm{f}}} \left(\mathbf{T}_{\mathrm{p}}\left[i\right] + \mathbf{T}_{\mathrm{m}}\left[i\right]\right) \tag{5.3}$$

where $\mathbf{T}_{\mathrm{p}}[i]$ and $\mathbf{T}_{\mathrm{m}}[i]$ denote the time for vertex computation and that for inter-node message passing required in superstep i during recomputation, respectively.

Equation 5.3 ignores the *downtime* period for replacing failed nodes and *synchronization* time because they are almost invariant w.r.t. the recovery methods discussed in this chapter. We also assume the cost of *intra-node mes-*

sage passing is negligible compared with network communication cost incurred by inter-node messages.

We now focus on how to compute $\mathbf{T}_{\mathrm{p}}[i]$ and $\mathbf{T}_{\mathrm{m}}[i]$ in Equation 5.3. Let S_i and ϕ_{p_i} denote the state and the partition-node mapping in the beginning of superstep i (during recomputation), respectively. We find that $\mathbf{T}_{\mathrm{p}}[i]$ and $\mathbf{T}_{\mathrm{m}}[i]$ can be computed based on S_i and ϕ_{p_i}. Therefore, we first describe how to compute S_i, ϕ_{p_i}, and then define $\mathbf{T}_{\mathrm{p}}[i]$ and $\mathbf{T}_{\mathrm{m}}[i]$ based on S_i, ϕ_{p_i}.

Compute S_i, ϕ_{p_i}. For $i = c + 1$, S_{c+1} is the state right after the failure (either F or its cascading failure) occurs. Let φ be the vertex-partition mapping, ϕ_{r} be the reassignment for the failure, and $\mathcal{P}_{\mathrm{f}}$ be the set of failed partitions. We have:

$$\phi_{\mathrm{p}_{c+1}}(v) = \begin{cases} \phi_{\mathrm{r}}(v) & \text{If } \varphi(v) \in \mathcal{P}_{\mathrm{f}} \\ \phi_{\mathrm{p}}(v) & \text{Otherwise} \end{cases} \tag{5.4}$$

For any $i \in (c + 1, s_{\mathrm{f}}]$, we have:

$$\phi_{\mathrm{p}_i} = \phi_{\mathrm{p}_{c+1}}, \quad S_i(v) = \begin{cases} S_{c+1}(v) & \text{If } S_{c+1}(v) \geq i \\ i - 1 & \text{Otherwise} \end{cases} \tag{5.5}$$

Compute $\mathbf{T}_{\mathrm{p}}[i], \mathbf{T}_{\mathrm{m}}[i]$. We now formally define $\mathbf{T}_{\mathrm{p}}[i]$ and $\mathbf{T}_{\mathrm{m}}[i]$.

According to the computation model in Section 5.2.1, computation time required in a superstep is determined by the *slowest* node, i.e., maximum computation time among all the nodes. Let $\mathcal{A}(i)$ be the set of vertices that perform computation during re-execution of superstep i. Let $\tau(v, i)$ denote the computation time of v in the normal execution of superstep i, φ be the vertex-partition mapping.

$$\mathbf{T}_{\mathrm{p}}[i] = \max_{N \in \mathcal{N}} \sum_{\tau(v,i)} \{v \in \mathcal{A}(i) \mid \phi_{\mathrm{p}_i}(\varphi(v)) = N\} \tag{5.6}$$

Due to simplicity, we assume computations for vertices in one node are performed sequentially. A more accurate estimation for $\mathbf{T}_{\mathrm{p}}[i]$ can be applied if the computation within a node can be parallelized using machines with multithreaded and multicore CPUs.

To compute $\mathbf{T}_{\mathrm{m}}[i]$, we adopt the Hockney's model [44], which estimates network communication time by the total size of inter-node messages divided by network bandwidth. Let $\mathcal{M}(i)$ be the set of messages forwarded when re-executing superstep i. Let $m.u$, $m.v$ and $\mu(m)$ be the source vertex, destination vertex and size of message m, respectively. Suppose the network bandwidth is B.

$$\mathbf{T}_{\mathrm{m}}\left[i\right] = \sum_{\mu(m)/B} \{m \in \mathcal{M}(i) \mid \phi_{\mathrm{p}_i}(\varphi(m.u))) \neq \phi_{\mathrm{p}_i}(\varphi(m.v))\} \tag{5.7}$$

Note that $\mathcal{A}(i)$, $\tau(v, i)$, $\mathcal{M}(i)$ and $\mu(m)$ in Equation 5.6 and 5.7 can only be obtained during the runtime execution of the application. A *perfect knowledge* of these values requires a detailed bookkeeping of graph status in every superstep, which incurs high maintainence cost. Therefore, we refer to the *statistics* (See Section 5.3) for approximation. Specifically, we can learn from S_i, ϕ_{p_i} whether a partition will perform computation and forward messages to another partition during the re-execution of superstep i, and based on the statistics, we know the computation cost and communication cost among these partitions in superstep c. We then approximate the costs in superstep i by those in superstep c.

Example 5.5: Consider $F(\{N_1\}, 12)$ in Example 5.2 and ϕ_{r} in Figure 5.5(a). Let c_1 and c_2 be the time for each vertex computation and that for sending an inter-node message, respectively. To compute T_{low} under ϕ_{r}, we calculate the re-execution time of superstep $11, 12$ without further cascading failures. In both supersteps, computation time is $4c_1$ caused by P_1, P_2 in N_1. Communication time in superstep 11 is $5c_2$ caused by 5 inter-node messages: 1 from P_2 to P_1, 4 from P_4 to P_2, P_3, and that in superstep 12 is $6c_2$ following the 6 cross-node edges. Hence, T_{low} under ϕ_{r} is $8c_1 + 11c_2$. □

Theorem 5.2. *Given a failure, finding a reassignment ϕ_{r} for it that minimizes T_{low} in Equation 5.3 is NP-hard.*

Theorem 5.2 can be proven by reducing the graph partitioning problem to the problem of finding reassignment with minimized T_{low}.

5.4.2 Cost-Sensitive Reassignment Algorithm

Due to the hardness result in Theorem 5.2, we develop a cost-sensitive reassignment algorithm. Before presenting our algorithm, we shall highlight the differences between our problem and traditional graph partitioning problem. First and foremost, traditional graph partitioning problem focuses on parti-

Algorithm 7: CostSensitiveReassign

```
Input  : $S$, state after the failure occurs
         $\mathcal{P}_{\mathrm{f}}$, the set of failed partitions
         $\mathcal{I}$, statistics
         $\mathcal{N}$, a set of compute nodes
Output: $\phi_{\mathrm{r}}$: reassignment
$\phi_{\mathrm{r}} \leftarrow$ RandomAssign($\mathcal{P}_{\mathrm{f}}, \mathcal{N}$);
$T_{low} \leftarrow$ ComputeCost($\phi_{\mathrm{r}}, S, \mathcal{I}$);
while $true$ do
    ${\phi_{\mathrm{r}}}' \leftarrow \phi_{\mathrm{r}}$; $\mathcal{P} \leftarrow \mathcal{P}_{\mathrm{f}}$; $i \leftarrow 0$;
    while $\mathcal{P} \neq \emptyset$ do
        $i \leftarrow i + 1$;
        $\mathcal{L}_i \leftarrow$ NextChange(${\phi_{\mathrm{r}}}', \mathcal{P}, S, \mathcal{I}$);
        foreach $P \in \mathcal{L}_i.\phi.Keys()$ do
            ${\phi_{\mathrm{r}}}'(P) \leftarrow \mathcal{L}_i.\phi(P)$;
            $\mathcal{P} \leftarrow \mathcal{P} - \{P\}$;
    $l \leftarrow \arg\min_i \ \mathcal{L}_i.Time$;
    if $\mathcal{L}_l.Time < T_{low}$ then
        for $j = 1$ to $l$ do
            foreach $P \in \mathcal{L}_j.\phi.Keys()$ do
                $\phi_{\mathrm{r}}(P) \leftarrow \mathcal{L}_j.\phi(P)$;
        $T_{low} \leftarrow \mathcal{L}_l.Time$;
    else
        break;
```

tioning a static graph into k components with the objective of minimizing the number of cross-component edges. In our case, we try to minimize the remaining recovery time T_{low}. T_{low} is independent of the original graph structure but relies on the vertex states and message-passing during the execution period. Second, graph partitioning outputs k components where k is predefined. On the contrary, our reassignment is required to dynamically allocate the failed partitions among the healthy nodes without the knowledge of k. Further, besides the partitioning, we must know the node to which a failed partition will be reassigned. Third, traditional partitioning always requires k components to have roughly equal size, while we allow imbalanced reassignment, i.e., assign more partitions to one node but fewer to another, if a smaller value of T_{low} can be achieved.

Algorithm 8: NextChange

```
Input  : $\phi_{\mathrm{r}}$, reassignment
         $\mathcal{P}$, a set of partitions
         $\mathcal{I}$, statistics
         $\mathcal{N}$, a set of compute nodes
Output: $\mathcal{L}$: exchange
$\phi \leftarrow \emptyset$; $\mathcal{L}_i.Time \leftarrow +\infty$;
foreach $P \in \mathcal{P}$ do
    foreach $P' \in \mathcal{P} - \{P\}$ do
        ${\phi_{\mathrm{r}}}' \leftarrow \phi_{\mathrm{r}}$;
        Swap ${\phi_{\mathrm{r}}}'(P)$ and ${\phi_{\mathrm{r}}}'(P')$;
        $t' \leftarrow$ComputeCost(${\phi_{\mathrm{r}}}'$, $S$, $\mathcal{I}$);
        if $\mathcal{L}_i.Time > t'$ then
            $\mathcal{L}_i.\phi \leftarrow \{(P, \phi_{\mathrm{r}}(P')), (P', \phi_{\mathrm{r}}(P))\}$;
            $\mathcal{L}_i.Time \leftarrow t'$;
    foreach $N \in \mathcal{N} - \{\phi_{\mathrm{r}}(P)\}$ do
        ${\phi_{\mathrm{r}}}' \leftarrow \phi_{\mathrm{r}}$; ${\phi_{\mathrm{r}}}'(P) \leftarrow N$;
        $t' \leftarrow$ComputeCost(${\phi_{\mathrm{r}}}'$, $S$, $\mathcal{I}$);
        if $\mathcal{L}_i.Time > t'$ then
            $\mathcal{L}_i.\phi \leftarrow \{(P, N)\}$;
            $\mathcal{L}_i.Time \leftarrow t'$;
```

Algorithm 7 outlines our reassignment algorithm. We first generate a reassignment ϕ_{r} by randomly assigning partitions in $\mathcal{P}_{\mathrm{f}}$ among compute nodes $\mathcal{N}$, and calculates T_{low} under ϕ_{r} (line 1-2). We then make a copy of ϕ_{r} as ${\phi_{\mathrm{r}}}'$ and improve ${\phi_{\mathrm{r}}}'$ iteratively (line 3-18). In i-th iteration, the algorithm chooses some partitions and modifies their reassignments (line 7-9). The modification information is stored in $\mathcal{L}_i$. $\mathcal{L}_i$ is in the form of $(\phi, Time)$, where ϕ is a partition-node mapping recording which partition is modified to be reassigned to which node, and $Time$ is T_{low} under the modified reassignment. The selected partitions are removed for further consideration (line 10). The iteration terminates when no more failed partitions left. After that, we check list $\mathcal{L}$ and find l such that $\mathcal{L}_l.Time$ is minimal (line 11), i.e.,

$$l \leftarrow \arg\min_i \quad \mathcal{L}_i.Time$$

If $\mathcal{L}_l.Time$ is smaller than T_{low} achieved by the initial reassignment ϕ_{r}, we update ϕ_{r} by sequentially applying all the modifications in $\mathcal{L}_1, \cdots, \mathcal{L}_l$ (line 12-16), and start another pass of iterations. Otherwise, we return ϕ_{r} as the result.

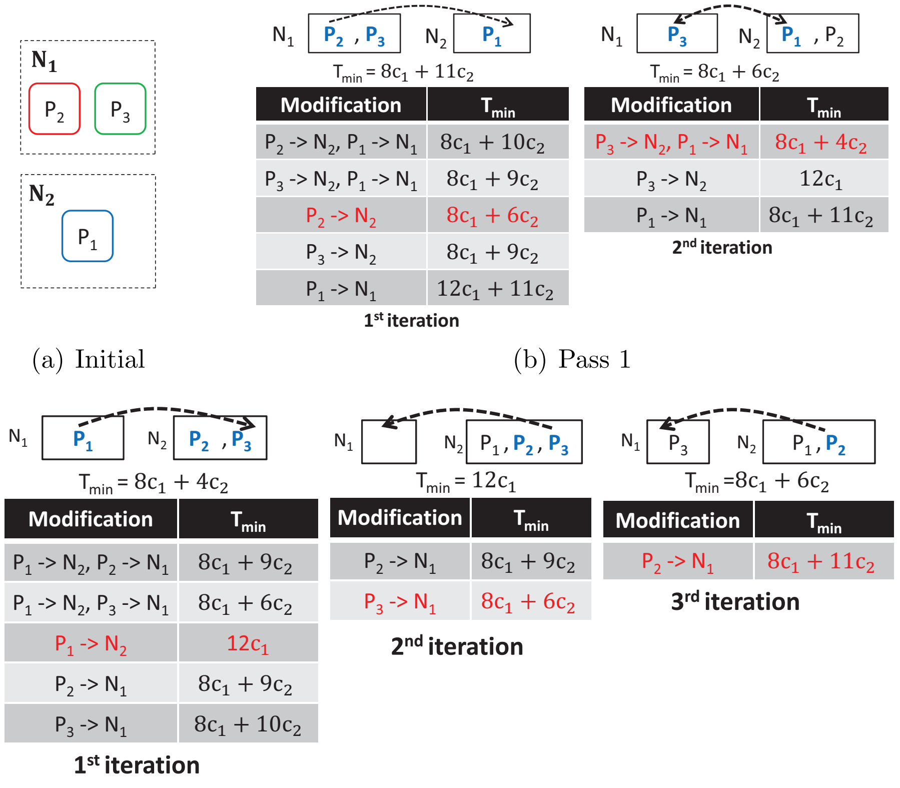
Algorithm 8 describes how to generate modification $\mathcal{L}_i$ (line 7 in Algorithm 7) in i-th iteration. We focus on two types of modifications: i) exchanging the reassignments between two partitions; ii) changing the reassignment for one partition. Given a reassignment ϕ_{r}, NextChange iterates over all the partitions (line 2) and for each partition P, it enumerates all the possibility of modifications, i.e., exchanging the reassignment of P with another partition (line 3-9) as well as assigning P to another node instead of $\phi_{\mathrm{r}}(P)$ (line 10-15). NextChange computes the corresponding T_{low} achieved by each modification and chooses the one with minimized value of T_{low} as the modification $\mathcal{L}_i$.

Example 5.6: Continuing from Example 5.5, suppose $\frac{c_1}{c_2} = 1.1$. Figure 5.5(a) shows the initial reassignment with $T_{low} = 8c_1 + 11c_2$. Figure 5.5(a) provides enumerated modifications and their T_{low} in the first pass. In iteration 1, assigning P_2 to N_2 achieves minimum T_{low}: $8c_1 + 6c_2$. In iteration 2, we only consider modifications for P_1, P_3 as P_1 has been considered. Exchanging reassignments for P_1, P_3 produces T_{low} of $8c_1 + 4c_2$. After that, all the partitions have been considered. We apply the first two modifications to initial the reassignment because the minimal T_{low} (i.e., $8c_1 + 4c_2$) is achieved after the second modification.

Figure 5.5(c) shows enumerated modifications and their T_{low} in pass 2. The minimal T_{low} (i.e., $12c_1$) in three iterations is achieved after the first modification, which is larger than $8c_1 + 4c_2$. Hence, the algorithm terminates and reports the reassignment produced by pass 1, i.e., assigning P_1 to N_1 and P_2, P_3 to N_2. □

5.5 Implementation

We implement our partition-based failure recovery method on top of epiCG. It is worth mentioning that our proposed recovery method can be integrated to other distributed graph processing platforms such as Giraph [2] and Hama [5], in a similar way.


N_1
P_2
P_3
N_2
P_1
N_1 P_2, P_3 N_2 P_1
N_1 P_3 N_2 P_1, P_2
N_1 P_1 N_2 P_2, P_3
N_1 N_2 P_1, P_2, P_3
N_1 P_3 N_2 P_1, P_2

(a) Initial

$T_{min} = 8c_1 + 11c_2$

Modification	T_{min}
P_2 -> N_2, P_1 -> N_1	$8c_1 + 10c_2$
P_3 -> N_2, P_1 -> N_1	$8c_1 + 9c_2$
P_2 -> N_2	$8c_1 + 6c_2$
P_3 -> N_2	$8c_1 + 9c_2$
P_1 -> N_1	$12c_1 + 11c_2$

1st iteration

$T_{min} = 8c_1 + 6c_2$

Modification	T_{min}
P_3 -> N_2, P_1 -> N_1	$8c_1 + 4c_2$
P_3 -> N_2	$12c_1$
P_1 -> N_1	$8c_1 + 11c_2$

2nd iteration

(b) Pass 1

$T_{min} = 8c_1 + 4c_2$

Modification	T_{min}
P_1 -> N_2, P_2 -> N_1	$8c_1 + 9c_2$
P_1 -> N_2, P_3 -> N_1	$8c_1 + 6c_2$
P_1 -> N_2	$12c_1$
P_2 -> N_1	$8c_1 + 9c_2$
P_3 -> N_1	$8c_1 + 10c_2$

1st iteration

$T_{min} = 12c_1$

Modification	T_{min}
P_2 -> N_1	$8c_1 + 9c_2$
P_3 -> N_1	$8c_1 + 6c_2$

2nd iteration

$T_{min} =8c_1 + 6c_2$

Modification	T_{min}
P_2 -> N_1	$8c_1 + 11c_2$

3rd iteration

(c) Pass 2

Figure 5.5: Example of modifications

5.5.1 A Brief Review of epiCG

epiCG distributes a graph processing job to a set of workers. One worker is selected as the *master* that coordinates the other *slave* workers, which perform vertex computations. One of the slaves acts as *zookeeper* to maintain various statuses shared among the master and slaves, e.g., notifying slaves of partitions assigned by the master, doing synchronization after accomplishing a superstep. Figure 5.6 shows the processing logic of workers in one superstep. Initially, the master generates partition assignment indicating which partition is processed by which slave, and writes the partition-to-slave mapping into zookeeper. Slaves fetch the mapping from zookeeper and exchange partitions along with their receiving messages based on the mapping. They then check whether the current superstep is a checkpointing superstep. If so, each slave saves the status of its partitions to a stable storage. After that, every slave performs computation for the vertices residing in it, sends messages and collects messages sent to its

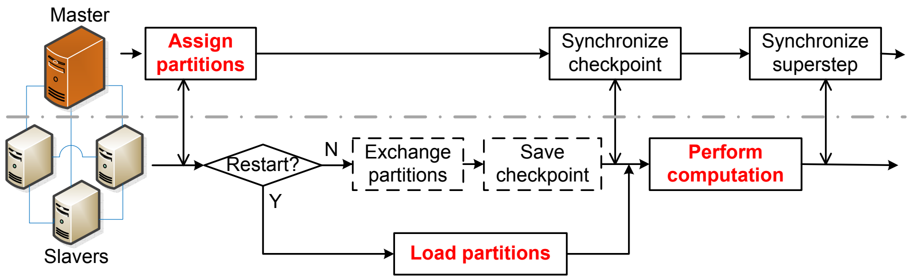


Figure 5.6: Processing a superstep in epiCG

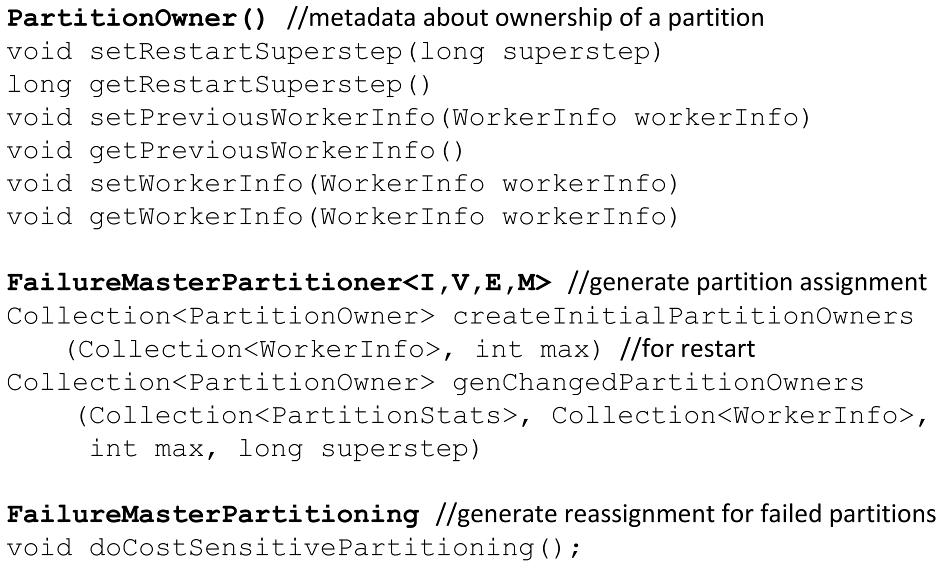
vertices. Finally, the master synchronizes the completion of the superstep.

Failure Recovery

Node failures are detected by the master at the end of each superstep, before synchronization. The master checks the healthy status registered periodically by every slave and considers a slave as failed if it has not registered its status over a specified interval. The default recovery mechanism adopted by epiCG is checkpoint-based recovery. We refer to the first superstep performed upon a failure as *restart* superstep. In the restart superstep, after the master generates the recovery plan and writes it to the zookeeper, slaves will load failed partitions that are assigned to them from the latest checkpoint and start recomputation. Recovery details are omitted to avoid redundancy.

5.5.2 Major APIs

To support partition-based failure recovery, we introduce several APIs to epiCG, as shown in Figure 5.7. We utilize `PartitionOwner` class to maintain the ownership of each partition. `setRestartSuperstep()` sets the next superstep when a partition needs to perform computation; `setWorkerInfo()` and `setPreviousWorkerInfo()` set information (e.g., IP address) for current and previous slaves in which a partition resides, respectively. To shuffle a partition from slave 1 to slave 2, we can simply set the previous, current workers to salve 1 and 2, respectively; the workers can retrieve these information via the three interfaces: `getRestartSuperstep()`, `getPreviousWorkerInfo()` and `getWorkerInfo()`. In order to generate the ownership of every partition, we introduce a new class `FailureMasterPartitioner`. This class will be initialized in the beginning of each superstep. It contains two important func-



```
PartitionOwner() //metadata about ownership of a partition
void setRestartSuperstep(long superstep)
long getRestartSuperstep()
void setPreviousWorkerInfo(WorkerInfo workerInfo)
void getPreviousWorkerInfo()
void setWorkerInfo(WorkerInfo workerInfo)
void getWorkerInfo(WorkerInfo workerInfo)

FailureMasterPartitioner<I,V,E,M> //generate partition assignment
Collection<PartitionOwner> createInitialPartitionOwners
    (Collection<WorkerInfo>, int max) //for restart
Collection<PartitionOwner> genChangedPartitionOwners
     (Collection<PartitionStats>, Collection<WorkerInfo>,
      int max, long superstep)

FailureMasterPartitioning //generate reassignment for failed partitions
void doCostSensitivePartitioning();
```

Figure 5.7: Major APIs

tions: `createInitialPartitionOwners()` generates reassignment for newly failed partitions and retains original ownership for the healthy ones; another is `genChangedPartitionOwners()` which is applied to exchange failed partitions after recovery accomplishes.

5.5.3 Implementation Details in epiCG

As an illustration, we consider a failure (can be a cascading failure) that occurs in executing superstep $s_{\rm f}$ and latest checkpointing superstep is $c+1$. We extend epiCG mainly in the following three aspects.

Partition Assignment

This is performed by master in the beginning of each superstep.

1. During superstep 1 or the restart superstep, the master always invokes `createInitialPartitionOwners()` to generate partition assginment and set current worker for each partition accordingly. In superstep 1, we set the previous worker for a partition to be the same as its current worker and the restart superstep for each partition to 1. In the restart superstep, we set previous worker for each partition to be the one before failure occurs. For newly failed partitions, we set $c+1$ as their restart supersteps; for

the other partitions, their restart supersteps are set to be one after the last superstep in which their computation are performed.

2. In the other supersteps, `genChangedPartitionOwners()` is invoked by the master to dynamically reassign partitions among the slaves. This is achieved by setting the previous worker of a partition as its current one and modifying its current worker to the new one.

Loading Partitions

After the master computes the partition assignment, it writes the partition-to-slave mapping to the zookeeper. Since all slaves are listening to the changes of this mapping information, every slave can fetch and parse this mapping and then load the corresponding failed partitions from the latest checkpoint if necessary. Note that in the checkpoint, partitions residing in the same slave are stored in the same file named with the slave host name, and within each file, there is a pointer to indicate which offset a partition starts. In this way, a slave can quickly load a partition using this implicit two-level index.

Performing Computation

For recomputation, every slave invokes the function `processGraphPartitions()` to execute vertex compute function, and invokes `sendMessageRequest()` to forward messages. During recovery, we adjust these two functions so that unnecessary computation and communication can be avoided, as follows.

1. `processGraphPartitions()` iterates over the partitions and check whether `PartitionOwner.getRestartSuperstep()` is less than the current superstep. If so, the slave loops over all the vertices residing in the partition and perform computation by invoking `Vertex.Compute()`;

2. During the computation from superstep $c + 1$ to $s_{\text{f}} - 1$, a message is omitted if it is sent to a vertex residing in the partition whose restart superstep is less than the current superstep;

3. At end of each superstep, every slave loads its locally logged messages. For supersteps in $[c + 1, s_{\text{f}} - 1]$, only messages to the partitions whose restart supersteps less than the current superstep are forwarded. For

superstep s_f, all the messages are sent via `sendMessageRequest()` to the corresponding slaves.

5.6 Experimental Evaluation

We compare our partition-based recovery method with checkpoint-based method on top of epiCG graph processing engine.

5.6.1 Experiment Setup

The experimental study was conducted on our in-house cluster. The cluster consists of 72 compute nodes, each of which is equipped with one Intel X3430 2.4GHz processor, 8GB of memory, two 500GB SATA hard disks and gigabit ethernet. On each compute node, we installed CentOS 5.5 operating system, Java 1.6.0 with a 64-bit server VM and Hadoop 0.20.203.0[2]. All the nodes are connected via three high-speed switches. We also made the following changes to the default epiCG configurations: (1) the size of virtual memory for each task was set to 4GB; (2) each node was configured to run one GraphUnit. By default, we chose 42 nodes out of the 72 nodes for the experiments and among them, one node acted as the master and zookeeper while the other 41 nodes behaved as slaves.

5.6.2 Benchmark Tasks and Datasets

We study the failure recovery over three benchmark tasks: k-means, semi-clustering [59] and PageRank.

- **k-means[3].** We implement k-means in epiCG. In our experiments, we set $k = 100$.

- **Semi-clustering.** A semi-cluster in a social graph consists of a group of people who interact frequently with each other and less frequently with others. We port the implementation in Hama [5] into epiCG. We use the same parameter values as in Hama, i.e., each cluster contains at most 100

[2] http://hadoop.apache.org/

[3] We does not show results for k-means in Chapter 4 as k-means does not produce messages during computation and hence is not a typical graph analytics query

Table 5.2: Dataset description

Dataset	**Data Size**	**#Vertices**	**#Edges**	**#Partitions**
Forest	2.7G	58,101,200	0	160
LiveJournal	1.0G	3,997,962	34,681,189	160
Friendster	31.16G	65,608,366	1,806,067,135	160

vertices, a vertex is involved in at most 10 clusters, and the boundary edge score factor is set to 0.2.

- **PageRank.** PageRank algorithm is an iterative graph processing algorithm. We refer the readers to the original paper [68] for more details.

Without loss of generality, we run all the tasks for 20 supersteps, and perform a checkpoint at the beginning of superstep 11. For all experiments, the results are averaged over ten runs.

We evaluate benchmark tasks over one vector dataset and two real-life social network graphs (Table 5.2 provides dataset details and the two graph datasets are downloaded from the website[4]).

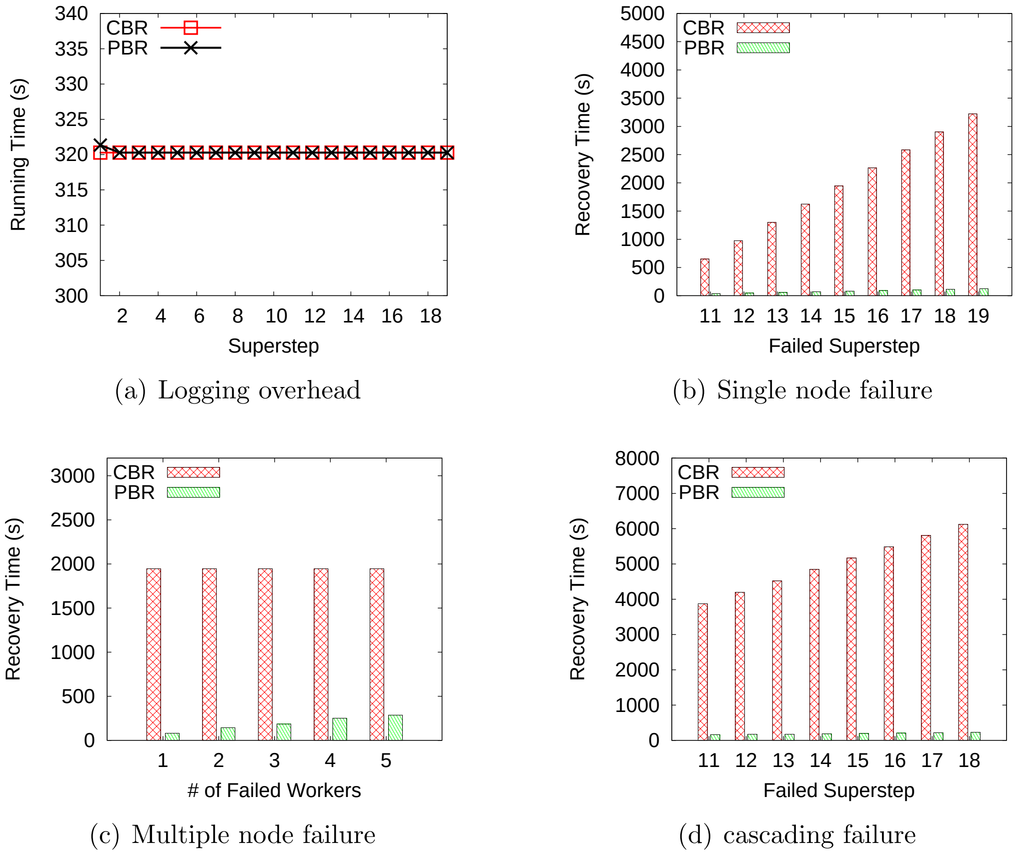
- **Forest.** Forest dataset[5] predicts forest cover type from cartographic variables. It originally contains 580K objects, each of which is associated with 10 integer attributes. To evaluate the performance on large datasets, we increase the size of Forest to 58,101,200 while maintaining the same distribution of values over each dimension using the data generator from [58, 87]. We use this dataset to evaluate the execution of k-means tasks.

- **LiveJournal.** LiveJournal is an online social networking and journaling service that enables users to post blogs, journals, and dairies. It contains more than 4 million vertices (users) and about 70 million directed edges (friendships between users). We use this dataset to evaluate the execution of semi-clustering tasks.

- **Friendster.** Friendster is an online social networking and gaming service. It contains more than 60 millions vertices and 1 billion edges. We use it to evaluate the execution of PageRank tasks.

[4] http://snap.stanford.edu/data/index.html

[5] http://archive.ics.uci.edu/ml/datasets/Covertype



(a) Logging overhead

(b) Single node failure

(c) Multiple node failure

(d) cascading failure

Figure 5.8: k-means results

We compare our proposed partition-based recovery method (PBR) with the checkpoint-based recovery method (CBR) over two **metrics**: recovery time and communication cost.

5.6.3 k-means

We first study the overhead of logging outgoing messages at the end of each superstep in PBR. Figure 5.8(a) shows the running time. PBR takes almost the same time as CBR. The reason is that in k-means tasks, there does not exist any outgoing messages among different vertices, and in this case, PBR performs exactly the same as CBR during normal execution. Another interesting observation is that the checkpointing superstep 10 does not incur higher running time compared with other supersteps. This is because compared with computing the new belonging cluster for each observation, the time of doing checkpointing is negligible.

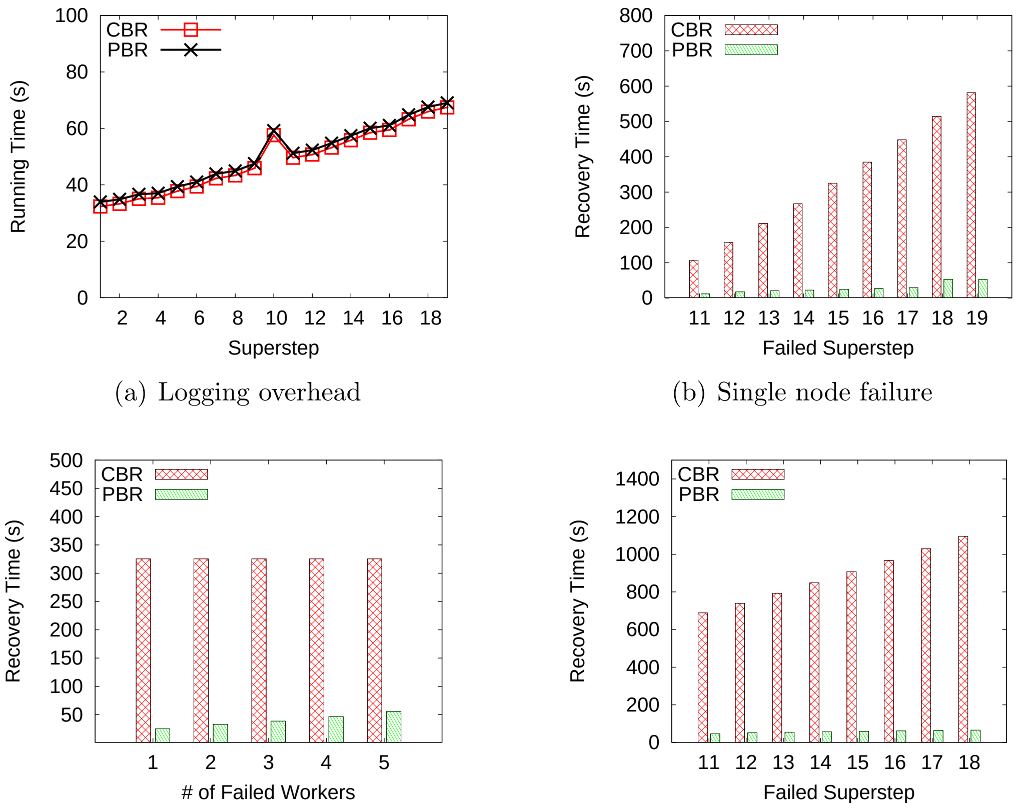
We then evaluate the performance of recovery methods for single node failures by varying the failed superstep from 11 to 19. Figure 5.8(b) plots the results. The recovery time of both CBR and PBR increases linearly when the failed superstep varies. Since there are no messages passing among different workers, computing the new belonging clusters for failed partitions can be accelerated by using all available workers, i.e., recomputation is parallelized over 40 workers for recovery. We find that PBR outperforms CBR by a factor of 12.4 to 25.7 and there is an obvious gain when the failed superstep increases. The speedup is less than 40x due to the overhead of loading the checkpoint in the beginning of a recovery.

Next, we investigate the performance of recovery methods for multiple node failures. The number of failed nodes is varied from 1 to 5 and the failed superstep is set to 15. Figure 5.8(c) plots the results. When the number of failed nodes increases, the recovery time increases linearly for PBR while that remains constant for CBR. No matter how many nodes fail, CBR will redo all computation from the latest checkpoint, while PBR recomputes the new belonging clusters for observations in the failed nodes and hence the recovery time becomes longer for a larger number of failed nodes. On average, PBR outperforms CBR by a factor of 6.8 to 23.9.

Finally, we focus on cascading failure by setting the first failed superstep to 19 and varying the second failed superstep from 11 to 18. Figure 5.8(d) plots the results. When the second failed superstep increases, the recovery time increases linearly for both CBR and PBR. On average, PBR can reduce recovery time by a factor of 23.8 to 26.8 compared with CBR.

5.6.4 Semi-clustering

Figure 5.9(a) plots the running time of each superstep for semi-clustering. PBR takes slightly longer time than CBR during normal execution. This is because compared with the computation and communication costs, the overhead of logging outgoing messages to local disks is relatively insignificant. Moreover, in semi-clustering, the size of each message from a vertex to its neighbors increases linearly with the superstep. Hence, both CBR and PBR runs slower in larger supersteps. In superstep 10, there is an obvious increment in the running time due to performing checkpoint.



(a) Logging overhead

(b) Single node failure

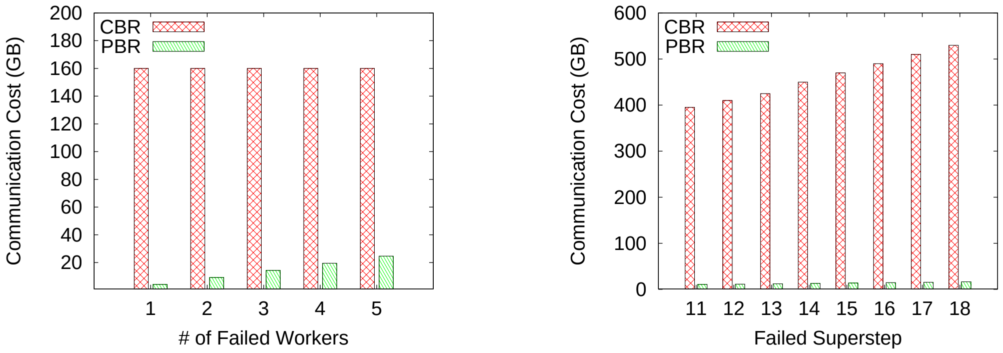
(c) Multiple node failure

(d) Cascading failure

Figure 5.9: Semi-clustering results

We evaluate the performance of recovery methods for single node failures, multiple node failures and cascading failures using the same settings as k-means. Figure 5.9(b), 5.9(c), 5.9(d) show the respective results. Basically, the trends of the running time of PBR and CBR in semi-clustering follows those in k-means. Specifically, PBR outperforms CBR by a factor of 9.0 to 15.3 for single node failures, by a factor of 13.1 to 5.8 for multiple node failures, and by a factor of 14.3 to 16.6 for cascading failures.

Besides the benefit of parallelizing computation, we also show the communication cost incurred by PBR and CBR in Figure 5.10. Since messages sent to the vertices residing in the healthy nodes can be omitted in PBR, we observe that in multiple node failure, PBR incurs 6.5 to 37.9 times less communication cost than CBR. For cascading failures, PBR can reduce communication cost by a factor of 37.1 compared with CBR.



(a) Multiple node failure

(b) Cascading failure

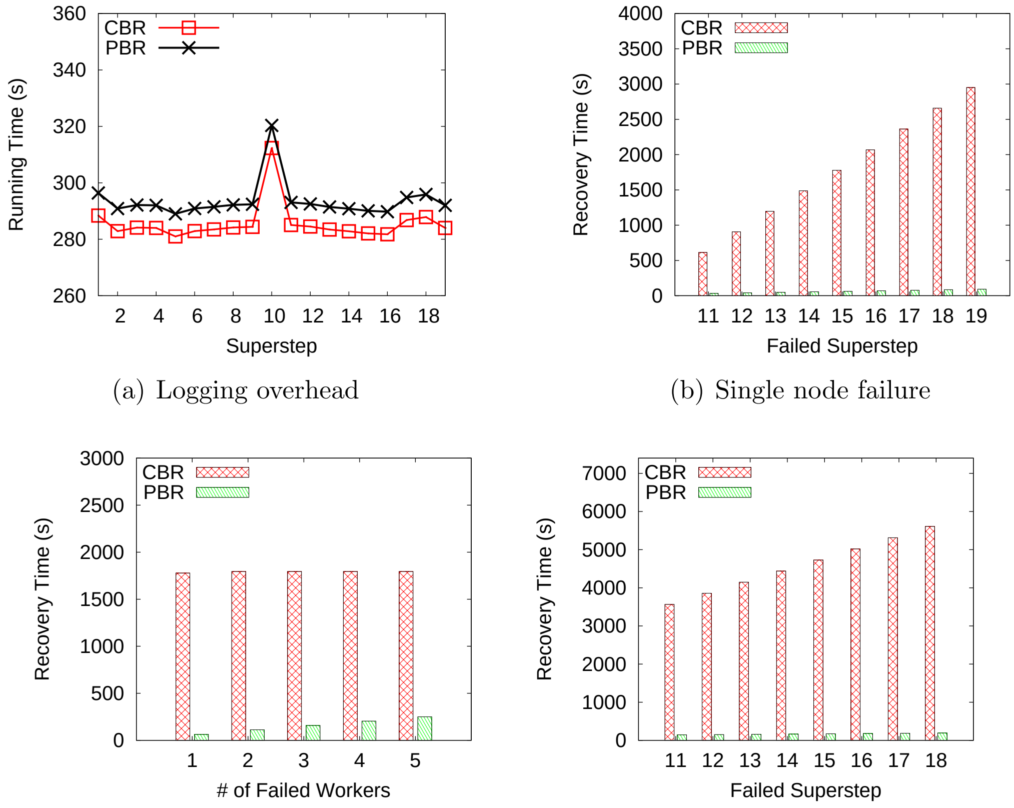
Figure 5.10: Communication cost of semi-clustering

5.6.5 PageRank

To study the logging overhead for PageRank tasks, we report the running time of every superstep in Figure 5.11(a). Compared with k-means and semi-clustering, PBR takes slightly more time than CBR in PageRank. This is because PageRank is evaluated over Friendster dataset which has power-law link distribution and each superstep involves a huge number of forwarding messages that should be logged locally via disk I/O. However, the overhead is still negligible, only 3% increase in running time. Due to doing checkpointing, there is an obvious increment of running time in superstep 10. In each superstep, the worker basically does the same task and hence the running time of each superstep almost remains the same. We also evaluate the performance of recovery methods for single node failures, multiple node failures and cascading failures. Figure 5.11(b), 5.11(c), 5.11(d) provide the recovery time, respectively. Figure 5.12 shows the corresponding communication cost. The performance of PBR and CBR follow the same trends as those in semi-clustering and k-means tasks. This further verifies the effectiveness of PBR, which parallelizes computation and eliminates unnecessary computation and communication cost.

5.6.6 Simulation Study

We perform a simulation study to evaluate the effectiveness and efficiency of our cost-sensitive reassignment algorithm CostSen in partition-based recovery. As a comparison, we consider a random approach Random by balancing computation among the nodes.



(a) Logging overhead

(b) Single node failure

(c) Multiple node failure

(d) Cascading failure

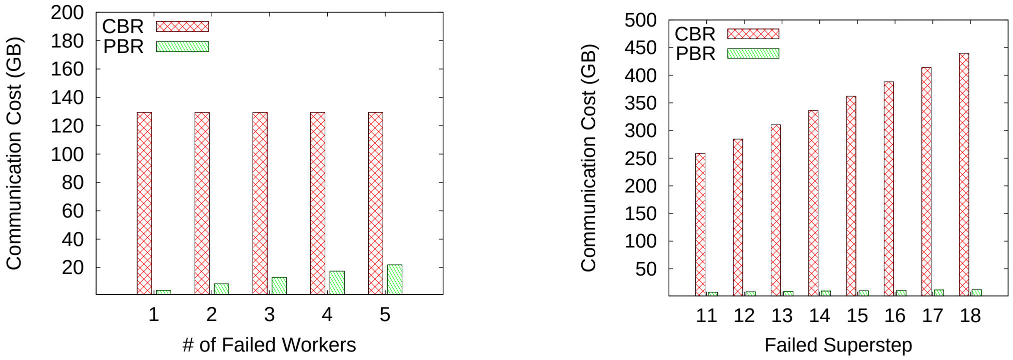
Figure 5.11: PageRank results

Data Preparation

We investigate the effect of the following parameters that potentially affect the performance of the reassignment algorithms:

- n: the number of failed partitions
- m: the number of healthy compute nodes
- computation cost per failed partition during recovery
- communication cost between every two failed partitions during recovery
- communication cost between failed partitions and healthy compute nodes during recovery

We generate communication cost by simulating two categories of graph partitioning, *random-partitioning* and *well-partitioning*. In *random-partitioning*, there is no obvious difference in the connections of two partitions lying in the same node or across two nodes; in *well-partitioning*, the number of edges connecting two partitions within the same node is much larger than that across



(a) Multiple node failure

(b) Cascading failure

Figure 5.12: Communication cost of PageRank

two nodes. For simulation, we generate communication cost using two distributions *uniformly-distributed* and *well-distributed* corresponding to *random-partitioning* and *well-partitioning*, as follows.

1) In *uniformly-distributed*, the communication cost between two failed partitions and that between a healthy node and a failed partition, is uniformly drawn from the range $[1, low]$.

2) In *well-distributed*, for each failed partition, we randomly select k failed partitions. The communication cost from the partition to each of the selected ones is uniformly drawn from range $[1, high]$, and that from the partition to any other failed partition is uniformly drawn from range $[1, low]$. The communication cost between partition and healthy node is generated in the same way. By default, we set p, *low*, *high* to 0.6, 100, 40000, respectively.

We generate comparable computation cost for each failed partition based a *comp-comm-ratio*, γ. Let S_P be the total communication cost from healthy nodes to a failed partition P. We use γ to adjust the ratio between the computation cost of P and S_P. The computation cost of P is randomly drawn from the range $[1, \gamma S_P]$. A larger γ implies that the job is more computation-intensive. Table 5.3 summarizes the ranges of our tuning parameters. Unless otherwise specified, we use the underlined default values.

Measure

We measure the performance of reassignment algorithms via five metrics: maximum computation cost (CompCost), total inter-node communication cost (Comm-

Table 5.3: Parameter ranges for simulation study

Parameter	**Description**	**Range**
n	number of failed partitions	$20, \underline{40}, 50$
m	number of healthy nodes	$20, \underline{40}, 50$
k	number of partitions (or healthy nodes) with high communication cost	$1, \underline{2}, 4, 6, 8$
γ	comp-comm-ratio	$0.1, 0.5, \underline{1}, 5, 10$

Cost), sum of CompCost and CommCost (TotalCost), running time and the number of nodes to which failed partitions are reassigned. All the costs are measured in *seconds* by default.

Effects of Comp-comm-ratio

Table 5.4 shows the results of CostSen and Random by varying γ in *uniformly-distributed* scenario. On average, CostSen produces reassignments with lower TotalCost and CommCost than Random over all the ratios. For $\gamma = 0.1$, CostSen outperforms Random with 2x lower TotalCost and CommCost. As γ increases, the advantage of CostSen in TotalCost and CommCost becomes less significant. The reason is that a larger γ makes the job more computation-intensive; this requires more nodes to parallelize the computation, while CommCost can hardly be reduced due to the uniform distribution. For smaller γ (e.g., 0.1), CostSen assigns failed partitions to a small number of nodes (< 5) due to insignificant CompCost, hence it reports reassignments with higher CompCost than Random. For larger γ (e.g., 10), CostSen performs similarly as Random in terms of the three costs, but it requires 2x fewer nodes for recovery. This saving is desirable in practice. We observe similar results for *well-distributed* scenario and omit the results to avoid duplication.

Effects of High Communication Partition (Healthy Node) Number

Table 5.5 shows the results of both methods when we vary the number of partitions (nodes) with high communication cost (k). For all values of k, CostSen outperforms Random with 2x lower TotalCost and CommCost. CostSen produces reassignments with higher CompCost which is relatively insignificant compared with CommCost. Furthermore, CostSen always involves fewer nodes for recovery. For $k = 10$, it uses 15x fewer nodes than Random.

Table 5.4: Effect of comp-comm-ratio γ (uniformly-distributed)

γ		**CompCost**	**CommCost**	**TotalCost**	**Used nodes**
0.1	Random	0.4	152.6	153	40
	CostSen	8.2	75.2	83.4	1
0.5	Random	2	152.3	154.3	40
	CostSen	36.3	80.9	117.2	2.3
1	Random	3.9	152.4	156.3	40
	CostSen	5.8	143.5	149.3	19.9
5	Random	19.5	152.1	171.6	40
	CostSen	19.5	146.8	166.3	24.4
10	Random	38.9	152.6	191.5	40
	CostSen	38.9	147.1	186	24.1

Table 5.5: Effects of the number of partitions (or healthy nodes) with high communication cost k (well-distributed)

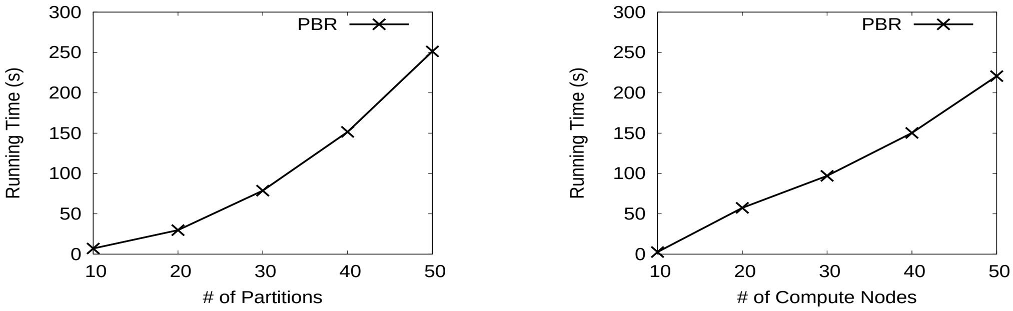
k		**CompCost**	**CommCost**	**TotalCost**	**Used nodes**
1	Random	3.8	902.4	906.2	40
	CostSen	11.5	241.6	253.1	14.23
2	Random	3.9	1603.5	1607.4	40
	CostSen	17.4	690.5	707.9	11.75
4	Random	3.9	2830.1	2834	40
	CostSen	46	1422.5	1468.5	7.63
6	Random	3.9	4032.8	4036.7	40
	CostSen	68.6	1967.4	2036	4.02
8	Random	3.8	5190.9	5194.7	40
	CostSen	76.7	2558.8	2635.5	2.79

Effects of the Number of Failed Partitions

Table 5.6 provides the results by varying the number of failed partitions (n). For each n, CostSen outperforms Random by 2.5x lower TotalCost and CommCost. Again, the reassignments reported by CostSen require higher CompCost, which is much smaller than CommCost. Furthermore, CostSen uses 3x fewer nodes for recovery. Figure 5.13(a) shows the running time of CostSen. It requires less than 250ms to generate reassignments. The running time increases quadratically with the number of failed partitions.

Table 5.6: Effects of the number of partitions n (well-distributed)

n		CompCost	CommCost	TotalCost	Used nodes
20	RANDOM	3.7	775.3	779	20
	COSTSEN	14.7	291.9	306.6	7.05
40	RANDOM	3.8	1605.2	1609	40
	COSTSEN	17.8	689	706.8	11.69
50	RANDOM	4.6	2005	2009.6	40
	COSTSEN	18.8	876.9	895.7	15.19



(a) # of failed partitions (b) # of healthy nodes

Figure 5.13: Running time (well-distributed)

Effects of the Number of Healthy Nodes

Table 5.7 provides the results by varying the number of healthy nodes (m). COSTSEN produces reassignments with 3x lower TotalCost and CommCost over all values of m. Furthermore, it employs fewer healthy nodes for recovery. For larger m (e.g., $40, 50$), the number of nodes involved in the reassignments from COSTSEN is 3x fewer than RANDOM. Figure 5.13(b) shows the running time of COSTSEN. The running time increases linearly with the number of healthy nodes. For $m = 50$, COSTSEN generates reassignments over 40 nodes within 250ms.

Summary of Simulation Study

Our simulation study show that the cost-sensitive reassigning algorithm achieves 2x speedup in our partition-based recovery framework compared with random assignment. It also outperforms random approach by wisely choosing a smaller number of compute nodes to handle failed partitions. Furthermore, our reassigning algorithm is efficient; the running time grows quadratically with the

Table 5.7: Effects of the number of healthy nodes m (well-distributed)

m		**CompCost**	**CommCost**	**TotalCost**	**Used nodes**
20	Random	5.9	1463.3	1469.2	20
	CostSen	20.5	572.1	592.6	10.21
40	Random	3.9	1582	1585.9	40
	CostSen	15.6	686.8	702.4	12.27
50	Random	3.8	1617.9	1621.7	40
	CostSen	16.5	695.8	712.3	12.97

number of failed partitions and linearly with the number of healthy nodes.

5.7 Summary

In this chapter, we present a novel partition-based recovery method to parallelize the failure recovery processing. Different from traditional checkpoint-based recovery, our recovery method distributes the recovery tasks to multiple compute nodes such that the recovery processing can be executed concurrently. Because partition-based failure recovery problem is NP-Hard, we use a communication and computation cost model to split the recovery among the compute nodes. We implement our recovery method in epiCG and observe that our proposed parallel failure recovery method outperforms existing checkpoint-based recovery methods by up to 30 times when using 40 compute nodes.

CHAPTER 6

Conclusion

6.1 Conclusion

epiC is one of the most elastic and extensible data processing systems proposed for Big Data applications. To cope with the 3V features of Big Data, epiC has to address several challenges. From the perspective of application design, it should be able to support various Big Data analytics tasks efficiently; from the perspective of system design, it should possess several properties such as simplicity, scalability, elasticity and fault tolerance. This thesis aims to develop effective and efficient solutions to two challenging issues in epiC: complex query processing and failure recovery. We employ epiC as our underlying distributed system due to its simplicity, efficiency and extensibility, but our approaches can be implemented in other distributed systems as well.

The first problem we study is to efficiently answer k nearest neighbor join query, which is a primitive operation in the domains of machine learning and data mining. While existing solutions to kNN join focus on developing effective index structures to accelerate the join process, all of them are centralized approaches. To cope with the increasing scale of data, we leverage MapUnit and ReduceUnit in epiC and propose a MapReduce-based solution to answer kNN join efficiently. We exploit the Voronoi-based partitioning method and divide the input objects into groups. Our proposed Map and Reduce functions guarantee that similar objects must be gathered into the same group. We then answer kNN join query by examining pair-wise objects within the same group. Our proposed method is the first distributed solution for answering kNN join query. We implement our solution in epiC and the results show that deploying MapReduce framework to answer kNN join query can accelerate query processing significantly compared with the existing index-based solutions. In order to further accelerate the processing, we also provide a theoretical analysis of the computation and shuffling cost involved in our approach. Based on the cost model, we introduce several pruning rules to eliminate the examination of dissimilar object pairs within the same group.

To answer graph analytics queries efficiently, our second work extends epiC and develop a scalable graph engine epiCG. We implement epiCG as one extension of epiC to avoid deploying a new distributed system from scratch. Different from epiC, epiCG is an in-memory processing engine that can handle iterative computations efficiently. In particular, epiCG allows every two units to communicate directly instead of relying on the master computer to transfer the messages; it also eliminates high I/O cost by asking every computation units to maintain their computed data in memory during the iterations. epiCG follows the same programming model as Pregel, but supports both edge-cut and vertex-cut graph partitioning methods. For vertex-cut, we propose an efficient approach to parallelize the vertex-cut generation process. For fault tolerance, epiCG achieves automatic failure detection and recovery. Extensive experiments are conducted to validate the high efficiency and scalability of epiCG.

The third problem we address is efficient failure recovery in epiCG. We first formalize the failure recovery problem and then propose a novel partition-based recovery method to parallelize the recovery tasks. Different from the traditional checkpoint-based recovery approach, our recovery method distributes recovery tasks to multiple compute nodes such that the tasks can be executed concurrently. We implement our recovery method in epiCG for performance evaluation. The experimental results show that our partition-based recovery method outperforms the checkpoint-based approach by up to 30 times when using 40 compute nodes. Furthermore, we prove that it is NP-hard to find a partitioning for the failed subgraph such that the total recovery time is minimized. To address the problem, we provide a cost model to estimate the overall recovery time for a given partitioning and propose a greedy algorithm to split the recovery workload among the compute nodes in a cost-effective way. Based on our simulation study, it is found that our proposed partitioning algorithm outperforms a random partitioning approach by a factor of 2.

6.2 Future Work

Our first work focused on answering kNN join queries in epiC, but there are many other complex queries that need to be solved as well. For instance, people may ask to figure out all the similar object pairs instead of the top-k answers. A promising future work is to develop an effective solution to answer similarity join queries in a distributed manner. Furthermore, in Chapter 3, we mainly considered multi-dimensional data and we would like to extend our work to handle high-dimensional data as well. The key problem here is that voronoi diagram-based partitioning may not work effectively in high-dimensional space, i.e., prune dissimilar object pairs. A more effective partitioning method for high-dimensional objects is needed.

In Chapter 4, we extended epiC and developed a graph processing engine epiCG on top of epiC. Although the results show epiCG is efficient and scalable, it is a synchronous graph processing system. A direct extension of this work is to develop an asynchronous graph processing engine on top of epiC.

Finally, our proposed partition-based recovery in Chapter 5 is a rollback recovery method. That is, the recovery requires to rollback to the latest checkpoint. An interesting future work is to perform recovery without rollback. For example, upon a failure, PageRank job can continue its computation immediately by assigning a random value to each failed vertex. Without rolling back and redoing the lost computation, PageRank still can converge to its final state deterministically. Similar examples include Shortest Path and Connected Component. There are some “outliers” such as k-means, which requires extra information to continue the computation and forward to achieve its final state. The key challenge is to figure out what kind of applications can be recovered without rollback and what cannot.

Bibliography

[1] http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazing-twitter-stats/.

[2] http://giraph.apache.org/.

[3] http://hadoop.apache.org/.

[4] http://hyracks.org/projects/pregelix/.

[5] https://hama.apache.org/.

[6] https://www.youtube.com/yt/press/statistics.html.

[7] http://www.sintef.no/home/press-room/research-news/big-data–for-better-or-worse/.

[8] http://www.slashgear.com/facebook-data-grows-by-over-500-tb-daily-23243691/.

[9] Foto Afrati and Jeffrey Ullman. Matching bounds for the all-pairs mapreduce problem. In *Proceedings of the 17th International Database Engineering &#38; Applications Symposium*, IDEAS '13, pages 3–4, New York, NY, USA, 2013. ACM.

[10] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Blazes: Coordination analysis for distributed programs. In *ICDE*, 2014.

[11] Mark Anderson. Best supercomputers still not best for big data. *IEEE Spectrum*, Jun, 2013.

[12] 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, New York, NY, USA, 2002. ACM.

[13] Joel F. Bartlett. A nonstop kernel. In *SOSP*, 1981.

[14] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.

[15] Christian Böhm and Florian Krebs. Supporting KDD applications by the k-nearest neighbor join. In *DEXA*, pages 504–516, 2003.

[16] Christian Böhm and Florian Krebs. The *k*-nearest neighbour join: Turbo charging the KDD process. *Knowl. Inf. Syst.*, 6(6):728–749, 2004.

[17] Christian Böhm and Hans-Peter Kriegel. A cost model and index architecture for the similarity join. In *ICDE*, pages 411–420, 2001.

[18] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *SOSP*, 1983.

[19] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under unix. In *ACM Trans. Comput. Syst.*, 1989.

[20] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162, Washington, DC, USA, 2011. IEEE Computer Society.

[21] Svein Erik Bratsberg, Svein-Olaf Hvasshovd, and ystein Torbjrnsen. Location and replication independent recovery in a highly available database. In Carol Small, Paul Douglas, Roger G. Johnson, Peter J. H. King, and G. Nigel Martin, editors, *BNCOD*, volume 1271 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 1997.

[22] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: Identifying density-based local outliers. In *SIGMOD*, pages 93–104, 2000.

[23] Thomas Brinkhoff, Hans peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. pages 237–246, 1993.

[24] Yingyi Bu, Vinayak R. Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.

[25] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[26] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Distributed systems (2nd ed.). chapter The Primary-backup Approach, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[27] Rong Chen, Jiaxin Shi, Yanzhe Chen, Haibin Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. Technical report, 2013.

[28] Zaiben Chen, Heng Tao Shen, and Xiaofang Zhou. Discovering popular routes from trajectories. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 900–911, Washington, DC, USA, 2011. IEEE Computer Society.

[29] P. Chimakurthy and Southern Illinois University at Carbondale. Computer Science. *A New Roll-forward Checkpointing/recovery Mechanism for Cluster Federation.* Southern Illinois University at Carbondale, 2007.

[30] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[31] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[32] Yerach Doytsher, Ben Galon, and Yaron Kanza. Storing routes in socio-spatial networks and supporting social-based route recommendation. In *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Location-Based Social Networks*, LBSN '11, pages 49–56, New York, NY, USA, 2011. ACM.

[33] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3), 2002.

[34] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. The case against specialized graph analytics engines. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[35] Michael E. Fisk and Curtis L. Hash, Jr. Filemap: Map-reduce program execution on loosely-coupled distributed systems. In *Proceedings of the Fourth International Workshop on Cloud Data and Platforms*, CloudDP '14, pages 6:1–6:6, New York, NY, USA, 2014. ACM.

[36] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[37] Amit Goyal, Francesco Bonchi, and Laks V.S. Lakshmanan. Learning influence probabilities in social networks. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, WSDM '10, pages 241–250, New York, NY, USA, 2010. ACM.

[38] Peter Groves, Basel Kayyali, David Knott, and V. Steve Kuiken. The big data revolution in healthcare. *McKinsey and Company*, 2013.

[39] Rachid Guerraoui and Andr Schiper. Fault-tolerance by replication in distributed systems. In *IN PROC CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES*, pages 38–57. Springer Verlag, 1996.

[40] B. Gupta and S. K. Banerjee. A roll-forward recovery scheme for solving the problem of coasting forward for distributed systems. *SIGOPS Oper. Syst. Rev.*, 35(3):55–66, July 2001.

[41] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 171–182, New York, NY, USA, 1997. ACM.

[42] Hiroaki Higaki, Kenji Shima, Takayuki Tachikawa, and Makoto Takizawa. Checkpoint and rollback in asynchronous distributed systems. In *INFOCOM*, volume 3, pages 998–1005. IEEE, 1997.

[43] Gísli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[44] Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing*, 20(3), 1994.

[45] 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.

[46] David Jiang, Anthony K. H. Tung, and Gang Chen. Map-join-reduce: Toward scalable and efficient data analysis on large clusters. *IEEE Trans. on Knowl. and Data Eng.*, 23(9):1299–1311, September 2011.

[47] Dawei Jiang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Sai Wu. epic: an extensible and scalable system for processing big data. In *PVLDB*, pages 541–552, 2014.

[48] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of MapReduce: An in-depth study. *PVLDB*, 3(1):472–483, 2010.

[49] George Karypis and Vipin Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[50] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, 2013.

[51] Edwin M. Knorr and Raymond T. Ng. Algorithms for mining distance-based outliers in large datasets. In *VLDB*, pages 392–403, 1998.

[52] Vibhore Kumar, Henrique Andrade, Buğra Gedik, and Kun-Lung Wu. Deduce: At the intersection of mapreduce and stream processing. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 657–662, New York, NY, USA, 2010. ACM.

[53] K. LANG. Finding good nearly balanced cuts in power law graphs. In *Tech. Rep. YRL-2004-036, Yahoo! Research Labs*, 2004.

[54] Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. Distributed data management using mapreduce. *ACM Comput. Surv.*, 46(3):31:1–31:42, 2014.

[55] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[56] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, 2010.

[57] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.

[58] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *PVLDB*, 5(10), 2012.

[59] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[60] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory oltp recovery. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 604–615, March 2014.

[61] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and H. Angela Byers. Big data: The next frontier for innovation, competition, and productivity. *McKinsey Global Institute*, 2011.

[62] Ahmed Metwally and Christos Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proc. VLDB Endow.*, 5(8):704–715, April 2012.

[63] Nichamon Naksinehaboon, Yudan Liu, Chokchai Leangsuksun, Raja Nassar, Mihaela Paun, and Stephen L. Scott. Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. In *CCGRID*, pages 783–788. IEEE Computer Society, 2008.

[64] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 949–960, New York, NY, USA, 2011. ACM.

[65] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[66] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *SOSP*, 2011.

[67] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[68] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[69] Jignesh M. Patel and David J. DeWitt. Clone join and shadow join: Two parallel spatial join algorithms. In *Proceedings of the 8th ACM International Symposium on Advances in Geographic Information Systems*, GIS '00, pages 54–61, New York, NY, USA, 2000. ACM.

[70] Fernando Pedone, Matthias Wiesmann, Andr Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *ICDCS*, pages 464–474. IEEE Computer Society, 2000.

[71] Jorge-Arnulfo Quian-Ruiz, Christoph Pinkel, Jrg Schad, and Jens Dittrich. Rafting mapreduce: Fast recovery on the raft. In Serge Abiteboul, Klemens Bhm, Christoph Koch, and Kian-Lee Tan, editors, *ICDE*, pages 589–600. IEEE Computer Society, 2011.

[72] Sara Radicati and Justin Levenstein. Email statistics report, 2013-2017. 2013.

[73] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 4:1–4:14, New York, NY, USA, 2012. ACM.

[74] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. Themis: An i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 13:1–13:14, New York, NY, USA, 2012. ACM.

[75] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *SSDBM*, 2013.

[76] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, 2013.

[77] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Trans. Softw. Eng.*, 9(3):219–228, May 1983.

[78] Robert Stephens. A survey of stream processing. *Acta Inf.*, 34(7):491–541, 1997.

[79] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3:204–226, 1985.

[80] Jie Tang, Jimeng Sun, Chi Wang, and Zi Yang. Social influence analysis in large-scale networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 807–816, New York, NY, USA, 2009. ACM.

[81] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[82] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". pages 193–204, 2013.

[83] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), 1990.

[84] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 1990.

[85] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 495–506, New York, NY, USA, 2010. ACM.

[86] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD*, pages 495–506, 2010.

[87] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.

[88] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large*

Data Bases, VLDB '98, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[89] Ling-Yin Wei, Yu Zheng, and Wen-Chih Peng. Constructing popular routes from uncertain trajectories. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 195–203, New York, NY, USA, 2012. ACM.

[90] Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jin Hu. Gorder: An efficient method for knn join processing. In *VLDB*, pages 756–767, 2004.

[91] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *GRADES*, 2013.

[92] Jie Xu and Brian Randell. Roll-forward error recovery in embedded real-time systems. In *ICPADS*, pages 414–421. IEEE Computer Society, 1996.

[93] Bin Yao, Feifei Li, and Piyush Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In *ICDE*, pages 4–15, 2010.

[94] Cui Yu, Bin Cui, Shuguang Wang, and Jianwen Su. Efficient index-based knn join processing for high-dimensional data. *Information and Software Technology*, 49(4):332–344, 2007.

[95] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB*, pages 421–430, 2001.

[96] Chi Zhang, Feifei Li, and Jeffrey Jestes. Efficient parallel knn joins for large data in MapReduce. In *EDBT*, 2012.

[97] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. Submitted to TKDE, 2015.

[98] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng. Spatial queries evaluation with mapreduce. In *GCC*, pages 287–292. IEEE Computer Society, 2009.

[99] Xiaofei Zhang, Lei Chen, and Min Wang. Efficient multi-way theta-join processing using mapreduce. *Proc. VLDB Endow.*, 5(11):1184–1195, July 2012.