QUERY-DRIVEN LARGE-SCALE PORTFOLIO AGGREGATE RISK ANALYSIS ON MAPREDUCE

by

Zhimin Yao

Submitted in partial fulfillment of the
requirements for the degree of
Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2014

Table of Contents

List of Figures . . . . **vii**

Abstract . . . . **x**

List of Abbreviations Used . . . . **xi**

Acknowledgments . . . . **xiii**

Chapter 1 Introduction **1**

Chapter 2 Background **5**

2.1 Overview of Insurance and Reinsurance 5
2.1.1 Insurance 6
2.1.2 Reinsurance 7

2.2 Reinsurance Risk Transfer Contracts 9
2.2.1 Reinsurance Treaty Types 9
2.2.2 Reinsurance Portfolio 10

2.3 Quantification of Reinsurance Risk 12
2.3.1 Catastrophe Risk Modeling 12
2.3.2 Monte Carlo Simulation 13
2.3.3 Risk Metrics 15

2.4 Aggregate Risk Analysis Algorithm (ARA) 17
2.4.1 Analysis Input Data 17
2.4.2 Sequential Aggregate Risk Analysis Algorithm 18

2.5 Secondary Uncertainty 20
2.5.1 Calculation Inputs 20
2.5.2 Steps for Combining Standard Deviation 21
2.5.3 Loss Calculation based on Beta Distribution 22

2.6 Financial Term Calculation 23
2.6.1 Beneficial Financial Term Calculation 23

2.6.2 Occurrence Financial Term Calculation 24
2.6.3 Aggregate Financial Term Calculation 24

2.7 Scenario Risk Analysis 25
2.7.1 Probable Maximum Losses (PML) by Line of business (LOB), Class of Business (COB) or Type of Participation (TOP) . . . 25
2.7.2 Region/Peril Losses 25
2.7.3 Multi-marginal Analysis 26
2.7.4 Stochastic Exceedance Probability Analysis 26
2.7.5 Periodic Loss Distribution 27

2.8 MapReduce and BigData Tools 27
2.8.1 Google MapReduce Programming Model 27
2.8.2 Hadoop MapReduce Framework 29
2.8.3 Big Data Tools 31

Chapter 3 QuPARA MapReduce Framework 32

3.1 Hadoop Ecosystem for Reinsurance Analytics 32

3.2 Framework Overview 33
3.2.1 Query Interface 34
3.2.2 Distributed File System 35
3.2.3 Data Filters 35
3.2.4 Core Engine 36

3.3 Data Organization 37
3.3.1 Year Event Table (YET) 37
3.3.2 Portfolio Table (PFT) 38
3.3.3 Layer List Table (LLT) 38
3.3.4 Exposure Data Pool (EDP) 39
3.3.5 Extended Event Loss Table (XELT) 39
3.3.6 Combined Event Loss Table (CELT) 39
3.3.7 Year Event Loss Table (YELT) 40
3.3.8 Event Catalogue Table (ECT) 40
3.3.9 Pruned Event Catalogue Table 40
3.3.10 Year Region Peril Loss Table (YRPLT) 41
3.3.11 Pruned Year Region Peril Loss Table (PYRPLT) 41
3.3.12 Analysis Result Table 41

3.4 Core Engine and MapReduce Algorithms 41
3.4.1 Mapper Algorithm 42
3.4.2 Combine Algorithm 43
3.4.3 Reduce Algorithm 45

3.5 User Query 45
3.5.1 Data Filtering Queries 46
3.5.2 Loss Grouping Query 47
3.5.3 Result Reporting Query 48

3.6 Loss Grouping 48

3.7 Reduce Key Generator 49

3.8 Final Result Production 50

3.9 Scenario Risk Analysis Query Example 50

Chapter 4 QuPARA System Implementation in Hadoop 53

4.1 System Overview 53

4.2 Hadoop Core Engine 54
4.2.1 Batch Job Scheduler 56
4.2.2 Loss Calculation MapReduce Round (LCR) 57
4.2.3 Loss Aggregation MapReduce Round (LAR) 57

4.3 Hive Data Filter 57

4.4 Pentaho Query Interface 58
4.4.1 Analysis Selection and User Input 59
4.4.2 Portfolio Analysis 59
4.4.3 Layer Specified Analysis 60
4.4.4 ELT Specified Analysis 61
4.4.5 Multi-marginal Analysis 61
4.4.6 STEP Analysis 62
4.4.7 Custom Analysis 62
4.4.8 Job Schedule Dashboard 64
4.4.9 Result Report 64

Chapter 5 Data Structure Design and Implementation 66

5.1 XELT Data Generator 67

5.2 CELT Data Structure Design 68
5.2.1 Direct Access Data Structure 68
5.2.2 Hierarchical Data Structure 69

5.3 HashMap CELT Implementation 71
5.3.1 STL HashMap CELT 71
5.3.2 Trove HashMap CELT 71
5.3.3 HPPC HashMap CELT 72

5.4 JVM Parameter Tuning 72
5.4.1 JVM Memory Management 73
5.4.2 GC Techniques 73
5.4.3 JVM Parameter Tuning 74
5.4.4 Relationship between Lookup Time and Data Structure Size . 84

5.5 HashMap CELT Implementations Comparison 85
5.5.1 Insert Performance Comparison 85
5.5.2 Lookup Performance Comparison 86

5.6 Hybrid CELT Implementation 87
5.6.1 Searching Algorithm Selection 88
5.6.2 JVM Parameter Tuning 90

5.7 Final CELT Implementation Selection 93
5.7.1 Insert Performance Comparison 93
5.7.2 Lookup Performance Comparison 95

Chapter 6 System Evaluation 96

6.1 Experiment Data Description 97

6.2 QuPARA System Performance 98
6.2.1 Speed-up Performance 98
6.2.2 Size-up Performance 99
6.2.3 Scale-up Performance 101

6.3 Scenario Analysis Validation 102

6.3.1 Portfolio Aggregate Risk Analysis 103
6.3.2 Specialized Layer Analysis 104
6.3.3 Specialized Exposure Analysis 105
6.3.4 Seasonal Analysis 106
6.3.5 Multi-marginal Analysis 107

Chapter 7 Conclusion and Future Work 110

7.1 Summary and Conclusion 110

7.2 Recommendation for Future Work 111

Bibliography 113

List of Figures

2.1 Risk and premium flows between individuals, primary insurers, and reinsurers 7

2.2 An example two-layer reinsurance program. 12

2.3 The basic structure of catastrophe model 14

2.4 Basic structure of MapReduce programming model 29

2.5 Basic structure of Hadoop MapReduce framework 30

3.1 The Query-Driven Portfolio Aggregate Risk Analysis (QuPARA) framework design 34

4.1 Overview of the final implementation of QuPARA 54

4.2 Implementation of the hadoop core engine 55

4.3 Pentaho portfolio risk analysis page 59

4.4 Pentaho layer specified analysis page 60

4.5 Pentaho ELT specified analysis page 61

4.6 Pentaho multi-marginal analysis page 62

4.7 Pentaho STEP analysis page 63

4.8 Pentaho custom analysis page 63

4.9 Pentaho sample result report page 65

5.1 The basic structure of CELT 69

5.2 Insertion time of STL HashMap CELT using different Xms values 75

5.3 Lookup time of STL HashMap CELT using different Xms values 76

5.4 Insertion time of Trove HashMap CELT using different Xms values 77

5.5 Lookup time of Trove HashMap CELT using different Xms values 77

5.6 Insertion time of HPPC HashMap CELT using different Xms values 78

5.7 Lookup time of HPPC HashMap CELT using different Xms values 78

5.8 Insertion time of STL HashMap CELT using different NewRatio values 80

5.9 Lookup time of STL HashMap CELT using different NewRatio values 81

5.10 Insertion time of Trove HashMap CELT using different NewRatio values 82

5.11 Lookup time of Trove HashMap CELT using different NewRatio values 82

5.12 Insertion time of HPPC HashMap CELT using different NewRatio values 83

5.13 Lookup time of HPPC HashMap CELT using different NewRatio values 83

5.14 Relationship between lookup time and percentage of hits in the primary data structure 84

5.15 Insert time comparison between different HashMap implementations 85

5.16 Memory consumption of the different HashMap implementations 86

5.17 Lookup time comparison between different HashMap implementations 87

5.18 Relationship between lookup time and percentage of hits in the primary data structure 88

5.19 Lookup time comparison between linear search and binary search 89

5.20 Lookup performance comparison between linear search, binary search, and hybrid search 90

5.21 Insertion time of Hybrid CELT using different Xms values . . 91

5.22 Lookup time of Hybrid CELT using different Xms values . . . 91

5.23 Insertion time of Hybrid CELT using different NewRatio values 92

5.24 Lookup time of Hybrid CELT using different NewRatio values 93

5.25 Insert time comparison between the HPPC HashMap CELT and the Hybrid CELT 94

5.26 Memory usage of the HPPC HashMap CELT and the Hybrid CELT 94

5.27 Lookup performance comparison between the HPPC HashMap CELT and the Hybrid CELT 95

6.1 Running time of QuPARA on 1,600 layers using between 4 and 72 cores 99

6.2 Speed-up achieved using QuPARA on 1,600 layers (corresponding to Figure 6.1) 100

6.3 Overall running time comparison between QuPARA implementations using different CELT 100

6.4 Speed-up comparison between QuPARA implementations using different CELT (corresponding to Figure 6.3) 101

6.5 Running time of QuPARA with increasing number of layers on 72 cores 102

6.6 Running time of QuPARA with fixed (N/P = 25) Layers . . . 103

6.7 Result page for portfolio aggregate risk analysis 104

6.8 Loss distribution of layers for commercial buildings with excess of loss treaties 105

6.9 Portfolio loss distribution in earthquake in Canada 106

6.10 Portfolio loss distribution in four seasons 107

6.11 Multi-marginal analysis EP curve results 109

Abstract

Modern reinsurance companies use stochastic simulation techniques for portfolio risk analysis, often referred to as aggregate risk analysis, to support risk management. Their risk portfolios may consist of thousands of reinsurance contracts covering millions of individually insured locations. To quantify risk and to help ensure capital adequacy, each portfolio must be evaluated in large-scaled simulation trials, each capturing a different possible sequence of catastrophic events (e.g., earthquakes, hurricanes, etc.) over the course of a contractual year. In practice, due to the amount of data and computations involved, it is highly attractive to explore high performance parallel computing solutions to accelerate the analysis.

In this thesis, we explore the design of a flexible framework, called QuPARA, which exploits parallelism to perform aggregate risk analysis via distributed computing by using the MapReduce programming model. The goal is to provide a flexible framework that can be used by analysts to answer a wide variety of unanticipated but natural ad hoc queries to help them better understand multiple dimensions of risks that can impact portfolio performance and thus company solvency.

The QuPARA framework was implemented using Apache Hadoop, Apache Hive, and Pentaho. This prototype allows the user to take advantage of large parallel servers in order to answer ad hoc risk analysis queries efficiently even on large data sets. We also present data structure optimizations and tuning that greatly accelerate QuPARA's computation. The performance of the prototype system is competitive with highly tuned production systems that are only capable of answering a narrow set of portfolio queries, in contrast to the wide range of ad hoc queries QuPARA is able to resolve.

List of Abbreviations Used

ARA	Aggregate Risk Analysis
CELT	Combined Event Loss Table
COB	Class of Business
DFS	Distributed File System
ECT	Event Catalogue Table
EDP	Exposure Data Pool
ELT	Event Loss Table
ELTP	Event Loss Table Pool
EP	Exceedance Probability
ETL	Extract, Transform, and Load
GC	Garbage Collection
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
JVM	Java Virtual Machine
LAR	Loss Aggregation MapReduce Round
LCR	Loss Calculation MapReduce Round
LLT	Layer List Table

LOB	Line of Business
MECT	Master Event Catalogue Table
PFT	Portfolio Table
PML	Probable Maximum Loss
PYRPLT	Pruned Year Region Peril Loss Table
QuPARA	Query-Driven Large-Scale Portfolio Aggregate Risk Analysis
STEP	Stochastic Exceedance Probability
TOP	Type of Participation
TVaR	Tail Value at Risk
XELT	Extended Event Loss Table
YELT	Year Event Loss Table
YET	Year Event Table
YRPLT	Year Region Peril Loss Table

Acknowledgments

For the most, I would like to thank my supervisor Dr. Andrew Rau-Chaplin and co-supervisor Dr. Norbert Zeh for their constant motivation, direction and guidance. I would also like to thank my readers Dr. Meng He and Dr. Qigang Gao for their suggestions and comments to improve my thesis.

I would like to thank the industrial experts from Flagstone Re. Dr. Georg Hofmann, Dr. Oliver Baltzer and Duane Wilson for their valuable time and feedback which has provided invaluable insights into reinsurance risk analytics.

Special thanks to my lab members Blesson, Omar and Udaya for their supports and inspirations in my thesis. Many thanks also goes to other members of the Risk Analytics Lab, and the members of the Faculty of Computer Science in Dalhousie University for the support and friendship.

I must present my best regards to my family: my parents Yanshi Yao and Rifen Hu, for giving birth to me at the first place and supporting me spiritually and materially through my life; my wife Taowei Li, for her continued support and encouragement.

Last but not the least, I would like to express my sincere gratitude to all who supported me directly or indirectly.

Chapter 1

Introduction

Risk analysis is a comprehensive methodology, which is used in various areas to determine occurrences of specified events and assess the consequences of such events in financial amount. A typical risk analysis usually involves probabilistic modeling and simulation techniques. In many cases, large-scale simulations will be performed. Parallelism and High Performance Computing (HPC) can be highly attractive for developing risk analysis applications, which can carry out these simulations and analyses quickly.

The financial management of the risk associated with catastrophic events such as earthquakes, hurricanes, and large-scale floods falls largely to reinsurance companies [1]. Their risk portfolios may consist of thousands of reinsurance contracts covering millions of individually insured locations. To quantify risk and to help ensure capital adequacy, reinsurance companies use an analytical pipeline to manage risks. The pipeline is consists of three stages, which are risk assessment, portfolio risk management, and enterprise risk management. In the risk assessment stage, catastrophe models are used to provide scientifically credible loss estimates for individual catastrophe events. In the portfolio risk management stage, risk metrics of the performance of the reinsurance portfolio and contract prices are assessed through portfolio risk analysis. In the enterprise risk management stage, the risk metrics will be combined with liability, assert and other forms of risks to generate an enterprise wide view of risk.

At the heart of the portfolio risk management stage, a stochastic simulation technique for portfolio risk analysis and contract pricing referred to as Aggregate Risk Analysis (ARA) [2, 3, 4, 5]. The ARA is a Monte Carlo simulation performed on a portfolio of risks that a reinsurer holds. At an industrial scale, a reinsurance portfolio may consist of thousands of annual reinsurance contracts covering millions of individually insured locations. To quantify annual portfolio risk, each portfolio must be evaluated with respect to a range of risk metrics that take the uncertainty associated with both event order and magnitude into account[6]. In order to obtain accurate and precise results, each portfolio must be evaluated in up to a million simulation trials, each consisting of a sequence of thousands of catastrophic events, such as earthquakes, hurricanes or floods. Each trial captures one scenario how globally distributed catastrophic events may unfold in a year.

Aggregate risk analysis is computationally intensive as well as data-intensive. Production analytical pipelines exploit parallelism in aggregate risk analysis and ruthlessly aggregate results. The results obtained from production pipelines summarize risk in terms of a small set of standard portfolio metrics that are key to regulatory bodies, rating agencies, and an organization's risk management team, such as Probable Maximum Loss (PML) [7, 8] and Tail Value at Risk (TVaR) [9, 10]. While production pipelines can efficiently aggregate terabytes of simulation results into a small set of key portfolio risk metrics, they are typically very poor at answering the types of ad hoc queries that can help actuaries or underwriters to better understand the multiple dimensions of risk that can impact a portfolio, such as spatial correlation, seasonality, peril features, construction features, and financial terms.

In this thesis, we propose a framework, Query-Driven Large-Scale Portfolio Aggregate Risk Analysis (QuPARA), for aggregate risk analysis that facilitates answering a rich variety of ad hoc queries in a timely manner. A key characteristic of the proposed framework is that it is designed to allow users with extensive mathematical and statistical skills but perhaps limited programming background, such as risk analysts, to pose a rich variety of complex risk queries. The user formulates their query by defining SQL-like filters. The framework then answers the query based on these filters, without requiring the user to make changes to the core implementation of the framework or to reorganize the input data of the analysis. The challenges that arise due to the amounts of data to be processed and due to the complexity of the required computations are largely encapsulated within the framework and hidden from the user.

Our prototype implementation of this framework for QuPARA uses Apache's Hadoop [11, 12] implementation of the MapReduce programming model [13, 14, 15] to exploit parallelism, and Apache Hive [16, 17] to support SQL-like queries. Also, a user interface is developed by using Pentaho to provide easy access and result visualization for the end users.

In our implementation of QuPARA, we discovered that the efficiency of the data structure we used to support exposure loss lookups has a huge impact on the overall system performance. We developed improved implementations of this data structure that greatly accelerate QuPARA's computation over large risk portfolios. Our approach was to combine data structure design with systematic in-depth experimental evaluation and tuning. This allowed us to reduce the size of the core lookup data structure, the Combined Event Loss Table (CELT), by over 50%, which in turn allowed us to double the batch size per QuPARA run within a Hadoop MapReduce environment and reduce the total system overhead by half. Also, the lookup performance is increased by 42% by using the optimized CELT comparing to the baseline implementation. The time reduction on system overhead and lookups resulted in an overall system performance improvement of 31.7%.

Even though QuPARA is not as fast as a production system on the narrow set of standard portfolio metrics, it can answer a wide variety of ad hoc queries in an efficient manner, and achieved performance that is competitive with production systems. For example, our experiments demonstrate that an industry-size risk analysis with 1,000,000 simulation trials, 1,000 events per trial, and on a portfolio consisting of 1,600 risk transfer layers with an average of 5 event loss tables per layer can be carried out on a 72-cores Hadoop cluster in just over 71 minutes. The speed-up achieved with 72-cores is up to 88%.

The remainder of this thesis is organized as follows. Chapter 2 gives an overview of background information about the insurance and reinsurance industry, reinsurance risk analysis, aggregate risk analysis, and the MapReduce programming model. Chapter 3 proposes the design of our new risk analysis framework, QuPARA. Chapter 4 describes the implementation details of our prototype risk analysis system. Chapter 5 discusses our efforts to engineers an efficient CELT implementation. Chapter 6 presents a performance and overall system functionality evaluations of our framework and implementation. Chapter 7 draws conclusion and discusses potential future work related to the QuPARA framework.

Chapter 2

Background

This chapter presents background information and related work for the research presented in this thesis. In Section 2.1, we give a brief overview of the insurance and the reinsurance industry. In Section 2.2, we discuss the risk transfer contract and risk portfolio in the reinsurance industry. In section 2.3, we talk about how risks are quantified in the reinsurance industry. Section 2.4-2.6 show the related work on financial and statistical modeling to address the risk analysis problem to be a sequential algorithm. In Section 2.7, we look beyond the basic procedures of the risk analysis and discuss potential variations to meet the related risk analysis problems under various scenarios. In Section 2.8, we give a review of all the parallel techniques and big data tools we have used to design and implement the QuPARA framework.

2.1 Overview of Insurance and Reinsurance

In this section, we give a brief overview of the insurance and the reinsurance industry. The insurance industry is well known to the public. It provides loss protections through risk transfer contracts to individuals and companies in exchange for financial payments. In addition to the insurance industry, there is a reinsurance industry which helps the insurance companies withstand massive losses that might occur, for example, accumulating catastrophic events like earthquakes [18]. The major insurance/reinsurance businesses can be fitted into two categories, namely property risk protection and life risk protection. We will focus solely on the property insurance/reinsurance industry and associated analytical risks in the remainder of this thesis.

2.1.1 Insurance

Risk, the possibility for financial loss associated an adverse event, is an unavoidable feature of many human activities. From an accounting perspective, risk can be viewed as a financial amount, typically negative, and an associated probability. In some of cases, the risk holders are not willing to or cannot bear the consequences of unexpected losses. Therefore, over time the insurance industry has developed to offer individual and companies the opportunity to manage the risks by entering into risk transfer contracts. Insurance is the transfer of risk from an individual to an insurance company in exchange for an agreed on financial payment. For the remainder of this thesis, we will focus on risk to properties; typically, buildings, although insurance exists for many other types of assets or event life. In property insurance, property risk coverage is specified in a contractual agreement, which is named the *policy*. The individual, who purchases a policy to gain property coverage, is known as the *policyholder*. The insurer, which is a company sells policies to individuals, is called the *primary insurance company*. The monetary payment, which is a policyholder pays to the primary insurance company in exchange of a policy, is called the *premium*.

The foundation of insurance is the uncertainty of the loss to the insured property. At the time a policyholder purchases insurance policy to ensure property coverage, neither the policyholder nor the primary insurance company knows whether, when, and how losses will occur to the insured property [19]. In the insurance business, a primary insurance company collects both risks and premiums while selling policies. The premiums will go into a large cash pool. When a policyholder claims a property loss, the primary insurance company need to pay the recovery amount from the pool. Insurance companies build portfolios of policies expecting that many more people will contribute to the pool than the people actually making claims. Therefore, in any event, the insurance companies should be able to have enough funds from the cash

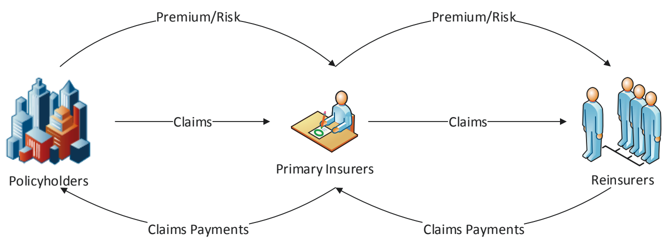


Figure 2.1: Risk and premium flows between individuals, primary insurers, and reinsurers

pool to pay the policyholders.

Most of the loss events, that individual properties experience, are uncorrelated. Meaning that, the probability of loss event in property A is independent of the probability of loss event in property B. Unfortunately, there are rare catastrophe events can affect many properties simultaneously. Natural catastrophes, such as earthquake, hurricane, and flood, are examples of such highly correlated loss. When a catastrophe event occurs, primary insurers, who have policies covering many properties in the same geographical location, may suffer massive losses. These massive correlated losses may threaten the solvency of the insurance companies. Therefore, the primary insurance companies seek other larger risk carriers to split the risks they are carrying in exchange of sharing the premiums they owned. The business, which hedges risks of the primary insurance companies, is called reinsurance. The risk and payment transfer chain between policyholders, primary insurers, and reinsurers is shown on Figure 2.1.

2.1.2 Reinsurance

Reinsurance, in essence, is the insurance for insures [20, 18, 21, 22]. The earliest known contractual reinsurance agreement was made in Genoa in July 1370 to insure a cargo shipment from Genoa to Sluis via sea. Under that contract, the primary insurer transferred the most risky part of the voyage in the shipment, which was from

Cadiz to Flanders, to a secondary insurer [18]. The core concept of reinsurance is to protect the primary insurance companies from insolvency under circumstances when a massive single loss or collective losses occur, for instance, in a large earthquake. In the reinsurance business, primary insurance companies purchase protections from secondary insures to gain coverage on the potential large losses associated typically with a catastrophe event. The reinsurance purchaser, which is the primary insurance company, is called the *ceding company.* The secondary insurer, which is the reinsurance seller, is called the *reinsurer.* The relocation part of the risk, which is transferred from the ceding company to the insurer, is called the *cession* [22].

The primary use of reinsurance is to help a ceding company to improve asset allocation and increase business capacity [22]. In insurance, the policyholders are guaranteed to be indemnified when failures occur on the insured properties. Therefore, due to regulation rules and laws, a primary insurance company has to maintain sufficient amount of capitals and reserves in cash or cash equivalents to proof its claims-paying ability [21]. Based on the amount of capitals and reserves, the primary insurance company will be legally restricted to only issue certain amount of policies. Instead of reserving all assets in cash or cash equivalents, a primary company can build a reinsurance structure to reduce the maximum expected loss amount to meet the statutory solvency requirements. The released capitals can be used to either make investments to gain profits or to reuse internally to increase the policy allowance.

Furthermore, reinsurance can help ceding companies to hedge large individual losses and thereby reduce volatility from individual claims [21]. In primary insurance, the premiums are determined based on the expected losses from the transferred risks on the insured properties. However, sometimes the loss distribution of an individual property, such as a car factory, has low expected value, but an extreme long tail. In this case, there is a huge gap between the expected loss and the largest loss that can occur. When an extreme loss occurs, it will cause a huge impact on the financial status of the company. By purchasing reinsurance protections, ceding companies can limit large individual losses to be acceptable to reduce fluctuations in financial results.

Moreover, reinsurance can save ceding companies from massive collective losses in large catastrophe events [21]. In insurance, the risk of each insured property is usually calculated independently because the failure occurrence of each insured property is statistically independent. However, in a catastrophe event, all the properties in the same region might be seriously damaged at the same time. In such case, based on the amount of policies in the disaster area a primary insurance company carries, the company may suffer huge claim volume. The collective losses from the claims may exceed the insurer's claims-paying ability and may bankrupt the insurance company immediately. As a solution, the ceding companies can purchase protections from reinsurance to cover claims beyond their bearable limit. In summary, the ultimate purpose of using reinsurance for ceding companies is to hedge risks and improve solvency. For the primary insurers, reinsurance is a cost, not a potential profit, but has value because it reduces financial volatility.

2.2 Reinsurance Risk Transfer Contracts

Risk transfer contracts are risk transfer agreements between reinsurance companies and ceding companies. In the reinsurance industry, risk transfer contracts can be made in various forms with very different rules which specify the benefits and obligations. In this section, we discuss the details about the major types of reinsurance contract, the form of insurance portfolio, and the structure of reinsurance portfolio.

2.2.1 Reinsurance Treaty Types

In the modern reinsurance industry, there are three types of contracts are widely used to form agreement between reinsurance companies and ceding companies, which are *Quota Share* treaty, *Surplus* treaty and *Excess of Loss* treaty [18].

Quota Share Treaty

Quota Share is a form of proportional reinsurance which shares the risk and the benefits in certain proportion of the loss. In this treaty, the reinsurance company shares a fixed percentage of every risk, which generate losses, in the ceding company's portfolio. In exchange, the premium the ceding company pays to the reinsurance company is a fixed percentage of the total premium from the insurance portfolio.

Surplus Treaty

Surplus treaty is another form of proportional reinsurance. This treaty is similar to the *Quota Share* treaty except it does not cover all the risks but only for those that exceed a predetermined loss amount.

Excess of Loss Treaty

Unlike the two proportional treaties we discussed above, *Excess of Loss* is a non-proportional treaty which the coverage amount of risks is predetermined. Under the *Excess of Loss* treaty, the reinsurer will only cover losses in a risk when the loss exceeds a pre-determined amount, which is the *loss attachment*, and the coverage will continue until the *loss limit* is reached [23].

In the reinsurance industry, the *Excess of Loss* treaties are most widely used contract type and they are also most challenging to model [24]. In the remainder of this thesis, we will focus solely on the *Excess of Loss* treaty reinsurance and analytical problems.

2.2.2 Reinsurance Portfolio

. A reinsurance company typically holds a *portfolio* of programs that insure primary insurance companies against large-scale losses, like those associated with catastrophic events. Each *program* contains data that describes (1) the buildings to be insured (the *exposure*), (2) the modeled risk to that exposure (the Event Loss Table (ELT)), and (3) a set of risk transfer contracts (the *layers*).

The *exposure* is represented by a table, one row per building covered, that lists the building's location, construction details, primary insurance coverage, and replacement value. The modelled risk is represented by an *event loss table* (ELT). This table lists for each of a large set of possible catastrophic events the expected loss that would occur to the exposure should the event occur. Finally, each *layer* (risk transfer contract) is described by a set of financial terms that includes aggregate deductibles and limits (i.e., deductibles and maximal payouts to be applied to the sum of losses over the year) and per-occurrence deductibles and limits (i.e., deductibles and maximal payouts to be applied to each loss in a year), plus other financial terms.

Consider, for example, a Japanese earthquake program. The exposure might list 2 million buildings (e.g., single-family homes, small commercial buildings, and apartments) and, for each, its location (e.g., latitude and longitude), constructions details (e.g., height, material, roof shape, etc.), primary insurance terms (e.g., deductibles and limits), and replacement value. The event loss table might, for each of 100,000 possible earthquake events in Japan, give the sum of the losses expected to the associated exposure should the associated event occur. Note that ELTs are the output of stochastic region peril models [25] and typically also include some additional financial terms. Finally, a risk transfer contract may consist of two layers as shown in Figure 2.2. The first layer is a per-occurrence layer that pays out a 60% share of losses between 160 million and 210 million associated with a single catastrophic event. The second layer is an aggregate layer covering 30% of losses between 40 million and 90 million that accumulate due to earthquake activity over the course of a year.

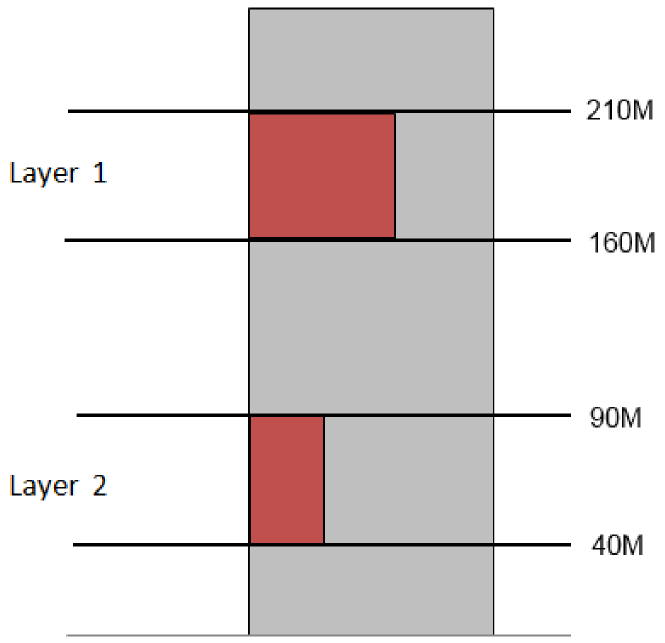


Figure 2.2: An example two-layer reinsurance program.

2.3 Quantification of Reinsurance Risk

At the heart of the analytical pipeline of a modern insurance/reinsurance company is a stochastic simulation technique for portfolio risk analysis and pricing referred to as Aggregate Risk Analysis (ARA). In this section, we firstly introduce catastrophe risks are recognized in the reinsurance industry, and then the loss recognition process is explained. Afterwards, we discuss the use of Monte Carlo simulation in the ARA. At the end of this section, the common risk quantification methods are introduced.

2.3.1 Catastrophe Risk Modeling

The core risk sources, which a reinsurer exposes to, is the catastrophe risk [26, 1]. A major catastrophe event, such as a big earthquake, can cause serious life and property losses. Buildings, bridges, and roads can be damaged or destroyed, and the recovery processes will take very long time. In such disasters, the direct and indirect losses will result a huge financial impact on the insurance industry, and then extend to the reinsurance industry. By using a proper catastrophe risk model, a reinsurer can organize its portfolio to avoid unbearable peak losses by balancing potential risks among different locations. After a disaster occurred, catastrophe risk model can help reinsurers to quickly and accurately estimate economic consequences of the event to perform proper reactions to reduce the impact of the event. Therefore, it is important for a reinsurer to develop and use catastrophe risk models to understand the occurrences and influences of catastrophe events for risk management.

In reinsurance, catastrophe risk modeling involves four important modules, which are hazard, inventory, vulnerability, and loss as shown on Figure 2.3 [26]. The hazard module is responsible for defining the occurrences and features of natural disasters. For example, a hurricane can be described by wind speed and projected path, and its occurrences can be simulated based on historical records. The inventory module produces data to describe physical properties, which may expose to catastrophe risks, as accurate as possible. The description of a property usually includes location, structure, and age. In the vulnerability module, the output from the hazard module is mapped on the inventory data to demonstrate the impact and evaluate the damages of the natural disasters on the properties. The *exposure* is represented by a table, one row per property covered, that lists the property's location, construction details, primary insurance coverage, and replacement value. The modeled risk is represented by an ELT. This table lists for each of a large set of possible catastrophic events the expected loss that would occur to the exposure should the event occur.

2.3.2 Monte Carlo Simulation

A common reinsurer usually holds a portfolio consists of hundreds to thousands layers. Each layer in the portfolio may cover one or more catastrophe events in various locations in a fiscal year. Due to the uncertainties in occurrence, location and severity

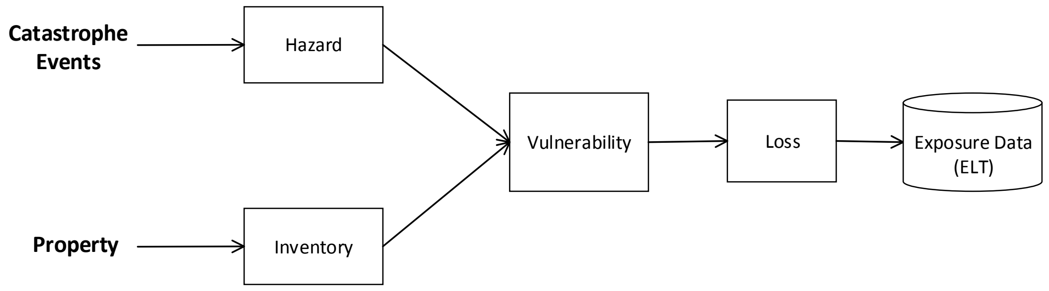


Figure 2.3: The basic structure of catastrophe model

of catastrophe events, it is impossible to predict which and where events will occur in a year, and then it is impossible to evaluate the actual risk amount for the portfolio. Therefore, a Monte Carlo simulation is needed to estimate the expected risks for each contract, and then create a probability distribution for the portfolio to explain possible outcomes in a year.

In reinsurance portfolio risk analysis, we first use sampling techniques to generate simulation trials which contain possible catastrophe events in a year, and then applying the trials to the portfolio to observe the expected risks. Each trial captures one scenario how globally distributed catastrophic events may unfold in a year. To quantify annual risk, the portfolio must be evaluated in up to a million simulation trials, each consisting of a sequence of possible thousands of catastrophic events, such as earthquakes, hurricanes or floods. In practice, the simulation trials is produced by a catastrophe event simulator and stored in the Year Event Table (YET). The simulation is performed by evaluating each event in the YET to against each of the layers in the portfolio to obtain individual layer loss, and then aggregate the layer losses in the same event to be a portfolio loss. Afterwards, all the portfolio losses which are generated by the events from the same trial will be aggregated to be an annual portfolio loss. Recall that, the YET contains millions of trials, and each trial includes thousands of events. The total events in a simulation will be billions. Furthermore, a reinsurance portfolio may carry thousands of layers, and each layer will be evaluated by each of the events in the YET. Therefore, the amount of calculations in one analysis may exceed trillions. Based on the scale of simulation, the size of consumed input data and intermediate results will be very big. Thus, aggregate risk analysis is computationally intensive as well as data-intensive, and it can benefit from exploiting advances in high-performance computing.

2.3.3 Risk Metrics

Risk quantification is the final process of ARA which interprets the analysis results to be meaningful metrics to support decision making. In reinsurance, the generally used portfolio risk metrics are the PML, TVaR, and Exceedance Probability (EP) curve [7, 8, 9, 10, 27].

Probable Maximum Loss (PML)

Probable Maximum Loss (PML) is a chiefly used term in the insurance and reinsurance industry. The PML is an estimated value of the maximum monetary losses affecting properties which can be caused by catastrophe events in a given exceedance probability [7, 8]. In reinsurance, the properties indicate the contracts in a portfolio which exposing to catastrophe risks. The given exceedance probability is defined by a return period which is an estimate recurrence time interval between losses which exceed a certain amount. For example, if an excess of 10 billion dollars loss from a reinsurance portfolio is not expected to occur more than about once in every 500 years, we could say that there is a probability of 99.8% in each year which the loss incurred from such portfolio will be less or equal than 10 billion dollars. The PML is an very important risk metric to evaluate the solvency of a reinsurer. In a given return period, a portfolio has higher PML value indicates higher risk for the reinsurer to fail to pay off the losses. In a portfolio, the PML values can be calculated based on the loss distribution of such portfolio. However, the loss distribution of a large portfolio is usually unknown because of the uncertainties of the natural disasters and their effects. With the available information, it is very difficult for a reinsurer to build the loss distribution for its portfolio through deterministic methods. Therefore, the only feasible way for reinsurers to calculate PML is to build an estimated loss distribution for their portfolios through large scaled stochastic Mote Carlo simulations.

Tail Value at Risk (TVaR)

Tail Value at Risk (TVaR) is a risk measure which quantifies the expected value in a condition which the total loss of a portfolio exceeds the PML in a given return period [9, 10]. Note that, the PML measures the maximum loss value under a return period, for instance, 500 years or probability of 99.8%, for a portfolio. However, there is a chance that the portfolio may suffer losses, which exceed the PML, from events which beyond the given return period. The TVaR is usually used by the reinsurers to adjust their reserve capital to ensure they can withstand more losses to further avoid solvency issues in a severe condition. The TVaR value can be calculated by finding the mean value of all losses beyond a specific return period in the portfolio distribution.

Exceedance Probability (EP) Curve

Exceedance Probability (EP) curve is a graphical representation which gives a detailed overview to explain the loss levels in each exceedance probability from 0 to 1 [27]. The EP curve is valuable for a reinsurer to recognize the size and distribution of the potential losses. Based on the EP curve, a reinsurer can manage risks in its portfolio to meet the acceptance loss level in different return periods. For example, suppose an insurer specifies 100 million dollars as the acceptance maximum loss at a return period of 500 years (99.8%). On the EP curve, if the PML at 99.8% is less than 100 million dollars, the insurer can accepting additional risks from the market to ear n more premium. However, if the PML at 99.8% is already higher than the acceptance level, the insurer would need to decrease risks by either reduce its portfolio or transfer risks to other reinsurers or the capital market. The EP curve can be treated as a continuous curve which each point on the curve represent a PML value at a probability between 0 to 1.

2.4 Aggregate Risk Analysis Algorithm (ARA)

Aggregate risk analysis is a form of Monte Carlo simulation performed on a portfolio of risks that a reinsurer holds rather than on individual risks. In this section, firstly the input data of ARA is explained, followed by the presentation of the sequential aggregate risk analysis algorithm. Afterwards, the statistical and financial calculations, which are secondary uncertainty, retention, occurrence loss, and aggregate loss calculations, are explained.

2.4.1 Analysis Input Data

There are three inputs to the portfolio aggregate risk analysis, which are YET, Portfolio Table (PFT), and a pool of ELTs. The YET is the representation of a pre-simulated occurrence of catastrophe events E in the form of trials T. Each trial captures the sequence of the occurrences of events for a year using time-stamps in the form of event time-stamp pairs. The PFT represents a reinsurance portfolio that includes a group of Programs, P, which in turn represents a set of Layers L, that covers a set of ELTs. The ELT represents the loss information l_E that correspond to an event based on an exposure. An ELT contains exposures in one particular location with certain perils. However, the damage from a catastrophe event, such as earthquake, can spread across different regions. Therefore, one event can appear over different ELTs with different loss information.

2.4.2 Sequential Aggregate Risk Analysis Algorithm

Algorithm 1 shows the sequential analysis of aggregate risk. The algorithm scan through the hierarchy of the portfolio, PF; firstly through the Programs, P, followed by the Layers, L, then the Event Loss Tables, $ELTs$. Line no. 5-9 shows how the loss associated with an Event in the ELT is computed. For this, the loss, l_E associated with an Event, E is retrieved, after which secondary uncertainty and retention financial terms is applied. The computation of secondary uncertainty and financial terms will be explained in the following sections.

```
   Input  : $YET$, $ELT$ pool, $PFT$
   Output: $YLT$
1  for each Trial, $T$ do
2      for each Event, $E$, in $T$ do
3          for each Program, $P$ in $PFT$ do
4              for each Layer, $L$, in $P$ do
5                  for each $ELT$ covered by $L$ do
6                      Lookup $E$ in the $ELT$ and find corresponding loss, $l_E$
7                      Apply Secondary Uncertainty to $l_E$ (Optional)
8                      Apply Beneficial Financial Terms to $l_E$
9                      $l'_E \leftarrow l'_E + l_E$
10                 end
11                 Apply Occurrence Financial Terms to $l'_E$
12                 Apply Aggregate Financial Terms to $l'_E$
13                 $l_L \leftarrow l_L + l'_E$
14             end
15             $l_P \leftarrow l_P + l_L$
16         end
17         $l_{PF} \leftarrow l_{PF} + l_P$
18     end
19     $l_T \leftarrow l_T + l_{PF}$
20 end
21 Populate $YLT$
```

Algorithm 1: Pseudo-code for Sequential Aggregate Risk Analysis

In line no. 11, two Occurrence Financial Terms, namely the Occurrence Attachment and the Occurrence Limit are applied to the loss. Afterwards, two Aggregate Financial Terms, namely the Aggregate Attachment and the Aggregate Limit are applied to the loss in line no. 12. In line no. 13, the net losses, l'_E, are summed up as l_L. The l_L losses correspond to the total loss a layer will suffer from a given event. Occurrence Attachment refers to the retention or deductible of the insured for an individual occurrence loss, whereas Occurrence Limit refers to the limit or coverage the insurer will pay for occurrence losses in excess of the retention. The Occurrence Financial Terms capture specific contractual properties of Excess of Loss treaties as they apply to individual event occurrences only.

In line no. 15, the aggregated layer losses are aggregated from the Layer level to the Program level, l_P. Aggregate Attachment refers to the retention or deductible of the insured for an annual cumulative loss, whereas Aggregate Limit refers to the limit or coverage the insurer will pay for annual cumulative losses in excess of the aggregate retention. The Aggregate Financial terms captures contractual properties as they apply to multiple event occurrences.

In line no. 17 the event losses are aggregated from the Program level to the Portfolio level, l_{PF}, which represents the total losses a portfolio will suffer from a given event. In line no. 19, all the event losses from the same trial will be combined into a trial-portfolio loss, l_T, which represents the yearly loss amount of a portfolio in the simulation trial.

In line no. 21, the trial-portfolio losses are populated in the Year Loss Table YLT which represents the output of the analysis of aggregate risk. Afterwards, financial functions or filters are then applied on the aggregate loss values to generate risk quantification metrics.

2.5 Secondary Uncertainty

When measuring the yearly loss of a reinsurance portfolio, the primary uncertainty we consider is associated with whether a catastrophe event occurs or not in a simulated year. A typical way to measure the risk of an occurred event is to use the $expectedloss$ value in the exposure data. In reality, the losses caused by a catastrophe event can be fluctuating. Therefore, secondary uncertainty is used to incorporates the loss distribution of occurred events to measure the uncertainty of loss amount. In this section, the methodology to compute secondary uncertainty is presented; this method heavily draws on industry wide practices. The inputs and their representations are firstly presented, followed by the sequence of steps for combining independent and correlated standard deviations, and finally computing the losses which are calculated based on the Beta distribution.

2.5.1 Calculation Inputs

There are six inputs required for computing secondary uncertainty for an event, which are obtained from the Extended Event Loss Table (XELT), and are as follows:

i. Program-and-Event-Occurrence-Specific random number, denoted as $z_{(Prog,E)} = P_{(Prog,E)} \in U(0,1)$. Each Event occurrences across different Programs have different random numbers. This number is used to represent the correlation of losses between layers which are grouped in the same program.

ii. Event-Occurrence-Specific random number, denoted as $z_{(E)} = P_{(E)} \in U(0,1)$. Each Event occurrence has an unique number.

iii. Mean loss, denoted as μ_L

iv. Independent standard deviation of loss, denoted as σ_I, which represents the variance within the event-loss distribution.

v. Correlated standard deviation of loss, denoted as σ_C, which represents the error of the event-occurrence dependencies.

vi. Maximum expected loss, denoted as $Loss_{max}$

2.5.2 Steps for Combining Standard Deviation

Given the above inputs, the independent and correlated standard deviations need to be combined to reduce the error in estimating the loss value associated with an event. For this, firstly, the raw standard deviations is produced as $\sigma = \sigma_I + \sigma_C$.

Secondly, the probabilities of occurrences, $z_{(Prog,E)}$ and $z_{(E)}$ are transformed from uniform distribution to normal distribution using, $f(x;\mu,\sigma^2) = \int\limits_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \mathrm{d}x$. This is applied to the probabilities of event occurrences as

$$\begin{aligned} v_{(Prog,E)} &= f(z_{(Prog,E)};0,1) \in N(0,1) \\ v_{(E)} &= f(z_{(E)};0,1) \in N(0,1) \end{aligned} \tag{2.1}$$

Thirdly, the linear combination of the transformed probabilities of event occurrences and the standard deviations is computed as

$$LC = v_{(Prog,E)}\left(\frac{\sigma_I}{\sigma}\right) + v_{(E)}\left(\frac{\sigma_C}{\sigma}\right) \tag{2.2}$$

Then the normal random variable is computed, fourthly, as

$$v = \frac{LC}{\sqrt{\left(\frac{\sigma_I}{\sigma}\right)^2 + \left(\frac{\sigma_C}{\sigma}\right)^2}} \tag{2.3}$$

Finally, the normal random variable is transformed from normal distribution to uniform distribution by using the inverse function as

$$z = \Phi(v) = F_{Norm}(v) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{v} e^{\frac{-t^2}{2}} \mathrm{d}t \tag{2.4}$$

The model used above for combining the independent and correlated standard deviations represents two extreme cases. The first case is when $\sigma_I = 0$ and the second case is when $\sigma_C = 0$. The model also ensures that the final random number, z, is drawn based on both the independent and correlated standard deviations.

2.5.3 Loss Calculation based on Beta Distribution

The loss is calculated based on the Beta distribution as fitting such a distribution allows the representation of risks quite accurately. The Beat distribution is industrial standard used by the large vendor of catastrophe model [24]. The Beta distribution is a two parameter distribution, with an upper bound for the standard deviation, and after normalising in the model above, three parameters are used.

In the Beta-distribution the standard deviation, mean, alpha and beta are defined as

$$\begin{aligned} \sigma_\beta &= \frac{\sigma}{Loss_{max}} \\ \mu_\beta &= \frac{\mu_L}{Loss_{max}} \\ \alpha &= \mu_\beta\Big(\big(\frac{\sigma_{\beta_{max}}}{\sigma_\beta}\big)^2 - 1\Big) \\ \beta &= (1-\mu_\beta)\Big(\big(\frac{\sigma_{\beta_{max}}}{\sigma_\beta}\big)^2 - 1\Big) \end{aligned} \tag{2.5}$$

An upper bound is set to limit the standard deviation using $\sigma_{\beta_{max}} = \sqrt{\mu_\beta(1-\mu_\beta)}$; if $\sigma_\beta > \sigma_{\beta_{max}}$, then $\sigma_\beta = \sigma_{\beta_{max}}$. For numerical purpose in the algorithm a value very close to $\sigma_{\beta_{max}}$ is chosen.

The estimated loss is then obtained by

$$\begin{aligned}
Loss &= Loss_{max} * PDF_{beta}(z;\alpha,\beta) \\
PDF_{beta}(z;\alpha,\beta) &= \int_{-\infty}^{z} \tfrac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} z^{\alpha-1}(1-z)^{\beta-1} \\
&\quad \text{where } \Gamma(z) \text{ is the gamma function.} \\
Loss &= Loss_{max} * \tfrac{1}{B(\alpha,\beta)} z^{\alpha-1}(1-z)^{\beta-1} \\
&\quad \text{where } B \text{ is the normalisation constant.}
\end{aligned} \tag{2.6}$$

2.6 Financial Term Calculation

From algorithm 1, we have seen three different financial terms calculations which are the beneficial financial terms, occurrence financial terms, and aggregate financial terms. In this section, we will discuss these financial term calculations in detail.

2.6.1 Beneficial Financial Term Calculation

In the ARA, the event losses are calculated based on the exposure data, which is the ELTs. The exposure data is created based on potential physical damages and losses. In a multi-layer reinsurance contract, the layer treaties usually take losses in order. Thus, the layers with lower priority gain benefits from the loss deductions generated by the higher prioritized layers. We use l_{ELT} to represent the cumulative loss in a trial which associated with an ELT. $ELTOcc_{att}$ and $ELTOcc_{lim}$ are used to represent the event occurrence beneficial attachment and limit. $ELTAgg_{att}$ and $ELTAgg_{lim}$ are used to represent the exposure aggregate beneficial attachment and limit. d_E is defined as the per-occurrence beneficial loss deduction amount. The calculation in line no. 8 in algorithm 1 is then obtained by

$$\begin{aligned} d_E &= \min(\max(l_{ELT} - ELTAgg_{lim}, 0), \\ &\quad \min(\max(l_E - ELTOcc_{att}, 0), ELTOcc_{lim})) \\ l_E &= l_E - \min(d_E, \max(0, l_{ELT} - ELTAgg_{lim})) \end{aligned} \tag{2.7}$$

2.6.2 Occurrence Financial Term Calculation

The occurrence financial term calculation is based on the Exceed of Loss treaty. In such treaty, there will be a pre-defined per-event loss deductible, Occ_{att}, and per-event loss limit, Occ_{lim}. In a catastrophe event, the reinsurance company only cover the loss portion which is higher than the deductible and is lower than the limit. Thus, the calculation in line no. 11 in algorithm 1 is then obtained by

$$l'_E = \min(\max(l'_E - Occ_{att}, 0), Occ_{lim}) \tag{2.8}$$

2.6.3 Aggregate Financial Term Calculation

The aggregate financial term calculation is based on the Exceed of Loss treaty. In such treaty, there will be a pre-defined cumulative loss deductible, Agg_{att}, and per-event loss limit, Agg_{lim}. Unlike the occurrence terms, the aggregate terms will be applied on the cumulative loss from a contract in year. The reinsurer only provides coverage on the part of cumulative losses which is higher than the deductible and lower than the limit. Thus, the calculation in line no. 14 in algorithm 1 is then obtained by

$$l_T = \min(\max(l_T - Agg_{att}, 0), Agg_{lim}) \tag{2.9}$$

2.7 Scenario Risk Analysis

While computing the basic risk metrics, such as PML, TVaR, and EP curve, for the company's entire portfolio is critical in assessing a company's solvency, analysts are often interested in digging deeper into the data and posing a wide variety of queries with the goal of analyzing such things as cash flow throughout the year, diversity of the portfolio, financial impact of adding a new contract or contracts to the portfolio, and many others.

The following is a representative, but far from complete set of example queries. Note that while all of them involve some aspects of the basic aggregate risk analysis algorithm used to compute EP curves, each puts its own twist on the computation.

2.7.1 Probable Maximum Losses (PML) by Line of business (LOB), Class of Business (COB) or Type of Participation (TOP)

In the reinsurance industry, a layer defines coverage on different types of exposures and the type of participation. Exposures can be classified by class of business (COB) or line of business (LOB) (e.g., marine, property or engineering coverage). The way in which the contractual coverage participates when a catastrophic event occurs is defined by the type of participation (TOP). Decision makers may want to know the loss distribution of a specific layer type in their portfolios, which requires the analysis to be restricted to layers covering a particular LOB, COB or TOP.

2.7.2 Region/Peril Losses

This type of query calculates the expected losses or a loss distribution for a set of geographic regions (e.g., Florida or Japan), a set of perils (e.g., hurricane or earthquake) or a combination of region and peril. This allows the reinsurer to understand both what types of catastrophes provide the most risk to their portfolio and in which regions of the globe they are most heavily exposed to these risks. This type of analysis helps the reinsurer to diversify or maintain a persistent portfolio in either dimension or both.

2.7.3 Multi-marginal Analysis

Given the current portfolio and a small set of potential new contracts, a reinsurer will have to decide which contracts to add to the portfolio. Adding a new contract means additional cash flow but also increases the exposure to risk. To help with the decision on which contracts to add, multi-marginal analysis calculates the difference between the loss distributions for the current portfolio and for the portfolio with any subset of these new contracts added. This allows the insurer to choose contracts or to price the contracts so as to obtain the right combination of added cash flow and added risk.

2.7.4 Stochastic Exceedance Probability Analysis

This analysis is a stochastic approach to the weighted convolution of multiple loss distributions. After the occurrence of a natural disaster not in their event catalogue, catastrophe modeling [17] vendors attempt to estimate the distribution of possible loss outcomes for that event. One way of doing this is to find similar events in existing stochastic event catalogues and propose a weighted combination of the distributions of several events that best represents the actual occurrence. A simulation-based approach allows for the simplest method of producing this combined distribution. To perform this type of analysis, a customized Year Event Table must be produced from the selected events and their weights. In this YET, each trial contains only one event, chosen with a probability proportional to its weight. The final result is a loss distribution of the event, including various statistics such as mean, variance and quantile.

2.7.5 Periodic Loss Distribution

Many natural catastrophes have a seasonal component to them, that is, do not occur uniformly throughout the year. For example, hurricanes on the Atlantic coast occur between July and November. Flooding in equatorial regions occurs in the rain season. As a result, the reinsurer may be interested in how their potential losses fluctuate throughout the year, for example to reduce their exposure through reduced contracts or increased retrocessional coverage during riskier periods. To aid in these decisions, a periodic loss distribution represents the loss distribution for different periods of the year, such as quarters, months, weeks, etc.

2.8 MapReduce and BigData Tools

The ARA involves large-scale simulations which are both data and computationally intensive. In this section, we firstly give an brief overview of the MapReduce programming model which can be used to explore parallel solution ARA. A MapReduce software, Apache Hadoop, which can be used to implement the major component of MapReduce ARA, is discussed as following. Furthermore, the big data tools, which are Apache Hive and Pentaho, are introduced as data management and manipulation tools in the end of this section.

2.8.1 Google MapReduce Programming Model

MapReduce is a programming model which was originally introduced by Jeffrey Dean and Sajay Ghemwat from Google in 2004 [13]. In the original paper, Map-Reduce was described as a programming model and an associated implementation for processing and generating large data sets [13]. A map and a reduce function are adopted in this model to execute a problem that can be decomposed into sub-problems with no dependencies. A classic MapReduce application is the word count problem, which is to calculate world frequency in a given text document. The basic information entity in this model is a ⟨key, value⟩ pair; where the key is an identifier and the value is its corresponding satellite data [28].

The entire MapReduce computation is split into three stages, which are map, shuffle and reduce, and the basic process of executing a MapReduce job is shown on Figure 2.4 [15]. From the beginning, the input will be split into mangy pieces and formatted into a set of ⟨key, value⟩ pairs. In the map stage, each mapper will receive a subset of the ⟨key, value⟩ pairs, and then for each input pair, a map function will be applied to produce one or more intermediate result pairs, which is known as (key', value'). In the shuffle stage, the intermediate ⟨key',value'⟩ pairs will be sorted according to the keys, and all the values attached with a common key will be delivered to a single node to launch the reduce stage. In the reduce stage, each reducer takes all the values, which have a common key, and apply the reduce function on these values to produce the final output. Because the mapper functions and reducer functions are executed independently from each other, the MapReduce model is scalable across large number of computing resources. In addition to the computations, the fault tolerance of the execution, for example, handling machine failures is taken care by MapReduce.

The core elements of a MapReduce algorithm are the map function and the reduce function. If we can solve a problem by first dividing it to be independent sub-problems, and then merge all the results from the sub-problems to produce the answer, we can build a parallel solution to speed up the computation by applying the MapReduce model. In the mapper stage, the processing of each input ⟨key, value⟩ pair is independent, and the map function is always stateless; so that we can parallel the input data processing by employing as may mappers as possible. In the reduce stage, we run a sequential algorithm on all the values with the same key. Also, there should not be any relationship between value groups. Therefore, we can parallel

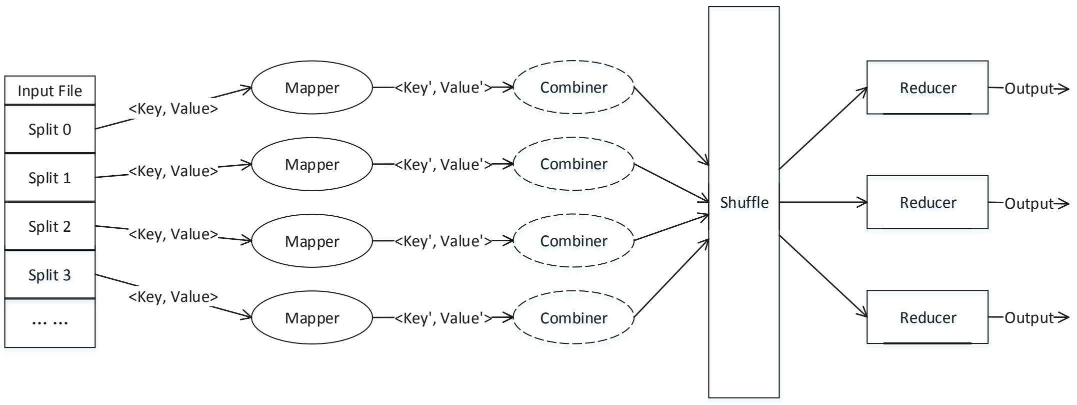


Figure 2.4: Basic structure of MapReduce programming model

the result generation by fire multiple reducers to process different value groups at the same time. Thus, the MapReduce model is most attractive for embarrassingly parallel problems [29].

2.8.2 Hadoop MapReduce Framework

Hadoop MapReduce is an open source software framework for developing and running MapReduce applications on large clusters to processing vast amounts of data [11, 12, 30]. This software framework works on the top of Hadoop Distributed File System (HDFS), which is a distributed file system designed to run on low-cost hardware. HDFS can support high throughput data access and highly fault-tolerant services [31]. Hadoop MapReduce framework is not only providing the basic functionalities, which are described by the basic Map-Reduce model, but also carrying more features to help developing applications. In Hadoop, the data input and output streams are directed by directory paths in HDFS. The basic structure of Hadoop is shown on Figure 2.5.

The Hadoop framework works in the following way for a MapReduce round. First of all, the data files from the HDFS are loaded using the InputFormat interface. The

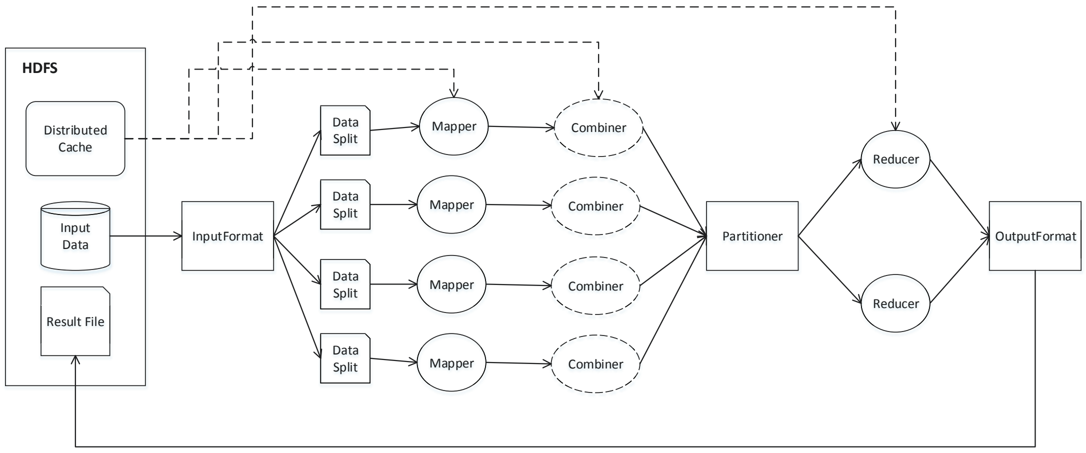


Figure 2.5: Basic structure of Hadoop MapReduce framework

InputFormat interface specifies the input of the Mapper function and splits the input data as required. The Mapper interface receives the partitioned data and emits intermediate ⟨key, value⟩ pairs. Before delivering the mapper results, there is an optional combiner which can be used to do local result aggregation to reduce network traffic. The Partitioner interface receives the intermediate ⟨key',value'⟩ pairs and controls the partitioning of these keys for the Reducer interface. Then the Reducer interface receives the partitioned intermediate ⟨key',value'⟩ pairs and generates the final output of this MapReduce round. The output is received by the OutputFormat interface and provides it back to HDFS. Additional to the original MapReduce programming model, the HDFS provides functionality called distributed cache for distributing small data files which are shared by all the worker nodes. The distributed cache provides local access to the shared data. This feature is very useful for implementing real-world problems.

The Hadoop MapReduce framework provides full functionalities for pure MapReduce problems, and it also has flexibility to build customized solutions around MapReduce. Through the InputFormat, we can define the way the input file is split to adopt the existing data. The Partitioner supports customizing intermediate result grouping which might be very useful. The Outputformat can be used to define the shape of file results. The distributed cache is helpful when supporting information is needed in the worker nodes to do processing. In the study presented in this thesis, we used Hadoop as the core piece of our analysis system.

2.8.3 Big Data Tools

In our study, rather than Hadoop, there were two big data tools which were used to provide additional features for the analysis system, which are Apache Hive, and Pentaho. Apache Hive is a data warehouse which builds on top of HDFS and Hadoop for data query and analysis [16, 17]. It provides a SQL-like language, called Hive Query Language (HQL), to support data query on HDFS through MapReduce process. In our study, Hive is used to store, manipulate, and filter the input data as well as the output results. Pehtaho is an open-source business intelligence software which provides a series of tools to build a work flow [32]. In our study, Pentaho was used to develop the user end interface and job scheduler.

Chapter 3

QuPARA MapReduce Framework

In this chapter, we propose a framework, Query-Driven Large-Scale Portfolio Aggregate Risk Analysis (QuPARA), for aggregate analysis that facilitates answering a rich variety of ad hoc queries in a timely manner. A key characteristic of the proposed framework is to allow users with extensive mathematical and statistical skills but perhaps limited programming background, such as risk analysts, to pose a rich variety of complex risk queries. The user formulates their query by using an SQL-like syntax to a set of filters. The framework then answers the query based on these filters, without requiring the user to make changes to the core implementation of the framework or to reorganize the input data of the analysis. The challenges that arise due to the amounts of data to be processed and due to the complexity of the required computations are completely encapsulated within the framework and hidden from the user.

3.1 Hadoop Ecosystem for Reinsurance Analytics

QuPARA is a framework that fits complex simulation based exploratory reinsurance risk analytics into big data and web analytics techniques. In the very early stage of designing QuPARA, we considered to use online analytical processing (OLAP) techniques to enable large-scaled reinsurance risk analysis. OLAP allows users to selectively extract and view data in various aspects [33]. In reinsurance risk analysis, OLAP is suitable for filtering and extracting input data. However, simulation based analysis does not fit in OLAP due to the complexity of dynamic simulation, stochastic calculations, and varied dataflow. Alternatively, we used a combination of MapReduce and web analytics techniques, which include Hadoop, Hive, and Pentaho, to achieve exploratory reinsurance risk analysis.

The combination of Hadoop, Hive, and Pentaho gives us many of the features we need to achieve a framework for reinsurance risk analysis. The Monte Carlo simulation in the analysis appears to fit in MapReduce programming model because the calculations between simulation trials are highly independent. Comparing to other parallel model, such as shared memory (OpenMP) or message passing (MPI), Hadoop provides a simple model of parallelism with fault-tolerance mechanism to ensure a successful completion of an analysis job even after some components in the system are not working properly. Hive provides a standard way to query and filter large data sets in an ad hoc manner. Pentaho gives ability create a workflow to link and organize all the components in the risk analysis process. QuPARA is not as efficient as the production risk analytic systems; however, with carefully design, implementation, and optimization, the performance can be improved to be competitive to the production systems.

3.2 Framework Overview

This section gives an overview of the QuPARA framework. Figure 3.1 visualizes the design of QuPARA. The framework is split into a front-end offering a *query interface* to the user, and a back-end consisting of a *distributed file system*, *data filters*, and a *core engine*. The *query interface* is built to allow users to specify queries to customize the analysis process. The *distributed file system* is used as the storage of all the data used in QuPARA to guarantee high disk throughput and availability of data. The *data filters* are used to retrieve the data required to answer the query from the portfolio. The different parts of the query entered through the query interface control the behavior of these files. The Aggregate Risk Analysis (ARA) algorithm is

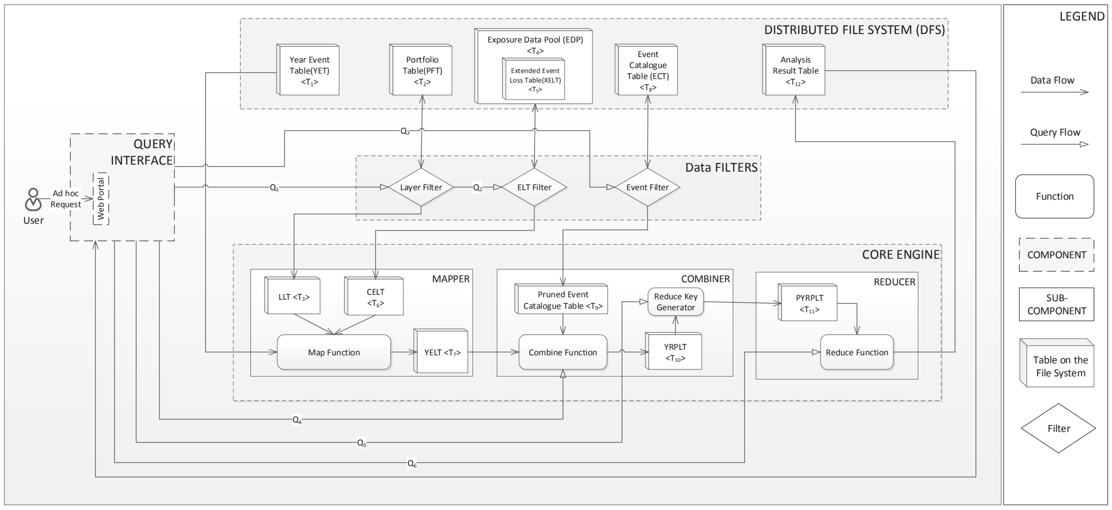


Figure 3.1: The Query-Driven Portfolio Aggregate Risk Analysis (QuPARA) framework design

implemented in the *core engine*, which utilizes the MapReduce programming model, to exploit parallelism to carry out the Monte Carlo loss aggregation calculations.

3.2.1 Query Interface

The *query interface* offers a web-based portal where the user can issue ad hoc queries in a SQL-like syntax. The queries are passed to the *data filters* and the *core engine* to customize access to input data, aggregation rules, and the production of analysis results. When a query has been processed, the user can access the result through the web-based portal. The basic idea of building the *query interface* is to allow the user to access all the features in QuPARA without knowing the actual implementation of the back-end components.

We analyzed a wide range of ad hoc rick analysis queries and discovered that they can all be expressed by specifying a numbers of sub-queries to different components in the QuPARA framework. In this framework, we employed six basic sub-queries, which are explained in detail in Section 3.5.

3.2.2 Distributed File System

The *distributed file system* is the major data storage used by QuPARA supports large files, provides high aggregate throughput and is highly scalable. A distributed file system can be deployed on a cluster with hundreds of nodes and support millions of files. Each node of the cluster can access the data in the file system the same interfaces and semantics as for local files. In QuPARA, the *distributed file system* stores the Portfolio Table (PFT), the Exposure Data Pool (EDP), which contains all portfolio-linked extended event loss tables, as well as the Year Event Table (YET). The raw portfolio data and exposure data can be directly accessed by the workers (mappers, combiners and reducers) in the MapReduce process. The YET is sent through the MapReduce input interface to split and distribute it across all the mappers to achieve parallelism in aggregate loss calculations. At the end of the computation, the result data is written back to the *distributed file system*.

3.2.3 Data Filters

QuPARA incorporates three data filters that allow the user to focus their queries on specific reinsurance layers and exposure data. These filters select the appropriate entries from the data tables they operate on for further processing in the MapReduce components in the *Query Engine*. The three filters are the *layer filter*, the *ELT filter* and the *event filter*.

Layer Filter

The *layer filter*, extracts the set of layers from the portfolio table, PFT, based on the user query and passes this list of layers to the mapper as the "portfolio" to be analyzed. The filter also unions the elt_IDs fields, which is a set of exposure data table identifiers, of the selected layers to construct a set of elt_IDs of required

Extended Event Loss Tables (XELTs) and then passes this set to the *ELT filter* for selection of the relevant XELTs.

ELT Filter

The *ELT filter* is used to select, from the EDP, the set of XELTs to be used in the analysis. The Event Loss Table (ELT) filter extracts from the set of *elt_IDs* received from the *layer filter* a subset of elt_IDs whose corresponding XELTs satisfy certain requirements. For example the *ELT filter* can be used to exclude exposures in certain regions from the analysis. The *ELT filter* then retrieves the corresponding set of XELTs and passes them to the mapper.

Event Filter

The *event filter* selects event features from the Event Catalogue Table (ECT) to provide the grouping information to the combiner. The *event filter* can be used to build different aggregation rules to express the analysis results of various levels of detail.

3.2.4 Core Engine

The *core engine* employs the MapReduce programming model to evaluate the query using a single MapReduce round, which consists of a map/combine step and a reduce step. During the map step, the engine uses one mapper per trial in the YET to calculate the event portfolio loss, l_{PF}, in Algorithm 1 on page 18, to populate the Year Event Loss Table (YELT). The combiner and reducer collaborate to aggregate portfolio losses and produce the final analysis results for the query. There is one combiner per mapper. The combiner pre-aggregates the loss information produced by this mapper, in order to reduce the amount of data to be sent across the network to the reducer(s). The reducer(s) then carry out the final aggregation. In most queries, which require only a single loss distribution as output, there is a single reducer. Multi-marginal analysis is an example where multiple loss distributions, which contain one per subset of the potential contracts to be added to the portfolio, are computed. In this case, we employ a separate reducer to produce each output distribution. In the following sections, we describe the date organization of each table, and then discuss the mapper, combiner, and reducer components of QuPARA in more detail.

3.3 Data Organization

The data used by QuPARA is represented as a number of tables. In the design of the QuPARA framework, we use 12 data tables ($T_1 ... T_{12}$) which include all the input, intermediate and output data. In this section, we discuss the 12 data tables used by QuPARA in detail.

3.3.1 Year Event Table (YET)

The Year Event Table (YET), table T_1 in Figure 3.1, is the input of the *core engine*, T_1 in Figure 3.1. This table is stored in the *distributed file system*. The YET contains tuples ⟨*trial_ID, event_ID, time_Index, z_PE*⟩. The *trial_ID* is a unique identifier associated with each of the trials in the simulation. A typical YET usually contains more than one million trials, and each trial consists of hundreds to thousands simulated catastrophe events. The *event_ID* is a unique identifier associated with each event in the ECT. The *time_Index* determines the time and order of occurrence of the event in the trial. The *z_PE* is a random number specific to the program and event occurrence. Each event occurrence across different programs has a different associated random number. This number is used to represent the correlations of event effects among all the layers in the same program.

3.3.2 Portfolio Table (PFT)

The Portfolio Table (PFT), table T_2 in Figure 3.1, is a meta-data table. This table stores all contract features associated with each layer. The PFT is stored in the *distributed file system*. A simplified PFT contains tuples ⟨*program_ID, layer_ID, cob, lob, top, elt_IDs*⟩. The *layer_ID* is a unique identifier associated with each layer in the portfolio. The class of business, *cob*, is an industry classification according to the perils insured and the related exposure and groups homogeneous risks. The line of business, *lob*, defines a set of one or more related products or services where a business generates revenue. The type of participation, *top*, describes how reinsurance coverage and premium payments are calculated in reinsurance contracts. The *elt_IDs* is a list of (extended) event loss table identifiers that are covered by the layer.

3.3.3 Layer List Table (LLT)

The Layer List Table (LLT), table T_3 in Figure 3.1, is an in-memory list of the *mapper* component in the *core engine*. The LLT contains the financial terms and layer features of a subset of layers in the PFT. The LLT is a product of the *layer filter*, which filters the layers in the PFT according to user query. The basic LLT contains tuples ⟨*layer_ID, occ_Ret, occ_Lim, agg_Ret, agg_Lim, elt_IDs*⟩. Each entry is a simplified contract representation of the layer identified by *layer_ID*. The *occ_Ret* is the occurrence retention or deductible of the insured for an individual occurrence loss. The *occ_Lim* is the occurrence limit or coverage the insurer will pay for occurrence losses in excess of the occurrence retention. The *agg_Ret* is the aggregate retention or deductible of the insured for an annual cumulative loss. The *agg_Lim* is the aggregate limit or coverage the insurer will pay for annual cumulative losses in excess of the aggregate retention. *elt_IDs* is a list of identifiers of (extended) event loss tables that store the exposure data covered by the layer.

3.3.4 Exposure Data Pool (EDP)

The Exposure Data Pool (EDP), table T_4 in Figure 3.1, is a meta data table stored in the *distributed file system*. The EDP contains exposure characteristics of the XELTs. A simplified EDP contains tuples ⟨*elt_ID, region, peril*⟩. Each such entry associates a particular type of peril and a particular region with the XELT with elt_ID.

3.3.5 Extended Event Loss Table (XELT)

The Extended Event Loss Table (XELT), table T_5 in Figure 3.1, is an extension of the original exposure data table ELT and is stored in the *distributed file system*. In contrast to a "normal" ELT, which records only expected loss values for events, the XELT stores information about the loss distribution of each events it contains. This information allows us to use the secondary uncertainty method for stochastic loss estimation of catastrophe events. The XELT contains tuples ⟨*event_ID, z_E, mean_Loss, sigma_I, sigma_C, max_Loss*⟩. The *event_ID* is the unique identifier of an event in the event catalogue. *z_E* is a random number specific to the event occurrence. Event occurrences across different programs have the same random number. The *mean_Loss* denotes the expected loss incurred if the event occurs. The *max_Loss* is the maximum expected loss incurred if the event occurs. *sigma_I* represents the variance of the losses which directly caused by the event. *sigma_C* represents the variance of the correlated losses which occur simultaneously from the event.

3.3.6 Combined Event Loss Table (CELT)

The Combined Event Loss Table (CELT), table T_6 in Figure 3.1, is constructed by each mapper from the extended event loss tables corresponding to the user's query. It combines the information in these XELTs and allows the loss information associated with an event in a particular XELT to be looked up using the corresponding ⟨*event_ID,* *elt_ID*⟩ pair as key. As such, the functionality of the CELT can be achieved by performing lookups directly on XELTs, but we discuss in Chapter 5 how to implement the CELT so that lookups become substantially faster than directly on XELTs.

3.3.7 Year Event Loss Table (YELT)

The Year Event Loss Table (YELT), table T_7 in Figure 3.1, is an intermediate table produced by the mapper for consumption by the combiner. It contains ⟨*trial_ID, event_ID, time_Index, estimated_Loss*⟩ tuples. Each of the tuples in the YELT represents a final portfolio loss in an event. The YELT is not physically created in memory in the mappers since the mapper immediately sends each of the tuples to the combiner. However, the combiner may assemble this in-memory if it has to sort the entries it receives before aggregating them.

3.3.8 Event Catalogue Table (ECT)

The Event Catalogue Table (ECT), table T_8 in Figure 3.1, is the master catalogue of catastrophic events that contains a detailed description of each such event that may occur in the simulation. In our study, we used a simplified ECT that stores only the basic characteristics of the events as ⟨*event_ID, region, peril*⟩ tuples associating a region and a type of peril with each event.

3.3.9 Pruned Event Catalogue Table

The Pruned Event Catalogue Table, table T_9 in Figure 3.1, may contain ⟨*event_ID, region*⟩, ⟨*event_ID, peril*⟩, or ⟨*event_ID, region, peril*⟩ depending on the user query. This table is a product of the *event filter*, which is used to provide grouping information for simulation events. The table is stored in memory in the *combiner*.

3.3.10 Year Region Peril Loss Table (YRPLT)

The Year Region Peril Loss Table (YRPLT), table T_{10} in Figure 3.1, is an intermediate table that is produced by the combiner for consumption by the reducer. The YRPLT contains tuples ⟨*trial_ID, time_Index, region, peril, estimated_Loss*⟩, listing for each trial the estimated loss at a given time, in a given region, and due to a particular type of peril. This table is an aggregated version of the YELT, based on the user query.

3.3.11 Pruned Year Region Peril Loss Table (PYRPLT)

The Pruned Year Region Peril Loss Table (PYRPLT), refers to the T_{11} in Figure 3.1, contains the grouped losses which are used by the *reducer* to generate the final result. This table is stored in the *reducer* component in the *core engine*, and contains ⟨*trial_ID, group_ID, grouped_Loss*⟩ tuples. All the entries in this tale share the same *group_ID*, which is assigned by the *reduce key generator*. In this table, there is only one entry for each $trial\_ID$.

3.3.12 Analysis Result Table

The Analysis Result Table, table T_{12} in Figure 3.1, is stored in the *distributed file system* and contains the final result produced by the *core engine*. The content of the table depends on the user query, which defines the final output from the *reducer*. For example, if we want to create a statistical distribution of the portfolio losses, the Analysis Result Table contains ⟨*trial_ID, loss*⟩ tuples. The user interface then uses these loss values to visualize the loss distribution.

3.4 Core Engine and MapReduce Algorithms

As already discussed, the *core engine* utilizes the MapReduce programming model to process aggregate risk analysis queries in parallel. Thus, its computation is divided into three parts: the mapper, the combiner and the reducer. Each mapper receives one trial in the YET as input, and produces event-layer level loss information. The combiner associated with each mapper aggregates the loss information from the mapper to produce trial-portfolio level loss information. Finally, the reducer generates the analysis output based on the aggregated loss information from the combiners. Depending on the user query, one or more reducers are used to produce the final query output. Next we discuss these components in detail.

3.4.1 Mapper Algorithm

Each mapper retrieves the layer list and the set of required XELTs from the distributed file system using the layer filter and the ELT filter. The layer filter retrieves the layer subset from the PFT and then deliver it to the mapper to create a LLT. The LLT contains all the contract-related financial terms, which are needed in the loss calculations. The layer filter also retrieves the identifiers of the XELTs contained in the subset of layer and passes these identifiers to the *ELT filter*. The XELTs with these identifiers form the base XELT set. If the user query specifies, for example, that the analysis should be restricted to a particular type of peril, the *ELT filter* then extracts the subset of XELT identifiers corresponding to the specified type of peril from the base XELT set, and delivers information about where the corresponding XELTs are stored to the mappers. Given this information, the mapper retrieves the actual XELTs from the storage and constructs an in memory lookup table, called the CELT, to support the lookup of loss information associated with each event in each of the XELTs.

Algorithm 2 shows the computation of the mapper. It takes an entire trial as input, represented as a pair $\langle T, E := \{E_1, E_2, \ldots, E_m\}\rangle$, which T is a unique trial identifier and E represent the set of events in this trial. It then iterates over the sequence of events in its trial, looks up the XELTs recording non-zero losses $l^j_{E_i}$ for each event $E_i \in E$. The loss estimate of the event E_i is done by iterating through every layer L in the LLT to generates the corresponding ⟨trial, event, loss⟩ tuple in the YELT, taking each XELT's and layer's financial terms into account.

```
Input  : $T, E := \{E_1, E_2, \cdots, E_m\}$, where $m$ is the number of events in a trial
Output: A list of entries $\langle T, E_i, l_{PF} \rangle$ of the YELT
1  for each event, $E_i$ in $E$ do
2      Look up $E_i$ in the CELT and find the set $l_{E_i} = \{l^1_{E_i}, l^2_{E_i}, \cdots, l^n_{E_i}\}$ of loss
       values recorded for event $E_i$ in the XELTs covered by the CELT
3      for each layer, $L$, in the LLT do
4          for each ELT $ELT_j$ covered by $L$ do
5              Lookup $l^j_{E_i}$ in $l_{E_i}$
6              Apply Secondary Uncertainty to $l^j_{E_i}$ (Optional)
7              Apply Beneficial Financial Terms of ELT$_j$ to $l^j_{E_i}$
8              $l_L \leftarrow l_L + l^j_{E_i}$
9          end
10         Apply $L$'s Occurrence Financial Terms to $l_L$
11         Apply $L$'s Aggregate Financial Terms to $l_L$
12         $l_{PF} \leftarrow l_{PF} + l_L$
13     end
14     Emit($\langle T$, $E_i$, $l_{PF} \rangle$)
15 end
```

Algorithm 2: Map Function in the Core Engine

3.4.2 Combine Algorithm

The combiner is responsible for aggregating the per-event losses to produce per-trial losses. The aggregation done by the combiner depends on the query. In the simplest case, a single loss distribution for the selected set of layers and XELTs is to be computed. In this case, the combiner sums the loss values in the YELT from the mapper and then sends this aggregate value to the reducer. A more complicated example is the computation of a weekly loss distribution. In this case, the combiner would group the losses corresponding to the events in each week and send each aggregate to a different reducer. Each reducer is then responsible for computing the loss distribution for one particular week. There are many other scenarios in which the event level losses need to be aggregated by characteristics, such as region and peril. Also, the user may want to filter out a particular set of events. Therefore, we construct a sub set of the ECT in the combiner with assistance from the *event filter* to provide event information to the combine function.

The details of the combiner are shown in Algorithm 3. The function receives as input the list of triples $\langle T,\, E_i,\, l_{PF}\rangle$ generated by a single mapper, that is, the list of portfolio loss values for the events in one specific trial. The combiner groups these loss values according to user-specified grouping criteria and outputs one aggregate loss value per group to populate the YRPLT. The entries in the YRPLT pass through the *reduce key generator*, which assigns a key to each entry that identifies the reducer the entry is to be sent to. The *reduce key generator* is implemented as a secondary grouping technique which allows the user to specify queries to reject, combine, or duplicate loss group. For example, we may need to send the loss group of a single combiner to multiple reducers to achieve portfolio combinations in the multi-marginal analysis.

```
Input: A list of YLT entries $\langle T, E_i, l_{PF}\rangle$ for the events in a given trial $T$.
Output: A list of aggregate YLT entries $\langle G_i, T, l_G\rangle$ with key $G_i$ for the event
        groups in trial $T$
1 Join input tuples with event catalogue to annotate events with their attributes
  (region, peril, etc.)
2 Group events in the input list by event features according to the user's query
3 for each group $G_i$ do
4     $l_{G_i} \leftarrow$ sum of the loss values associated with the events in $G_i$ in trial $T$
5     $R_i \leftarrow$ ReduceKeyGenerator($G_i$, T)
6     for each $R \in R_i$ do
7         Emit($\langle R, L_G\rangle$)
8     end
9 end
```

Algorithm 3: Combine Function in the Core Engine

3.4.3 Reduce Algorithm

The reducer is the final component of the *core engine* which formats and outputs the final analysis results. The input of each reducers is a set of trial loss values with the same group key, which we call the PYRPLT. The PYRPLT is a subset of the YRPLT in the *combiner*. The reducer function, shown in Algorithm 4, receives as input the loss values for one specific group and for all trials in the YET. The reducer then aggregates these loss values into the loss statistic requested by the user. The default result query to the combiner is to output the loss distribution of the groups. In this case, the reducer delivers the raw loss data directly to the *distributed file system*, and then the query interface visualizes the data by creating a histogram based on the loss values to express the distribution. However, the user might assign queries to generate special views of the results. For example, to generate Probable Maximum Loss (PML) values, the reducer need to sorts the received loss values in increasing order, and then for each user-specified return period p, report the corresponding probable maximum loss value.

```
   **Input**: A list of loss tuples $\langle R_i, l_{PF} \rangle$ for an event group $G_i$.
   **Output**: Loss statistics for event group $G_i$ based on user's query
1  Based on user query, generate:
   (i) Group loss distribution, or
   (ii) Group loss statistics, or
   (iii) Group Probable Maximum Loss (PML) and/or Tail Value at Risk (TVaR)
```

Algorithm 4: Reduce Function in the Core Engine

3.5 User Query

The motivation for building QuPARA is to be able to answer a rich variety of ad hoc risk analysis queries. The user can specify SQL-like queries to modify the input data, analysis process, and result representation. In QuPARA, there are six queries, referred as $Q_1...Q_6$ in Figure 3.1, which the user can use to influence the analysis process. Based on the functionality of each query provided, we divide the queries into three categories: data filtering queries, loss grouping queries, and result reporting queries. The default analysis scenario in QuPARA is the portfolio aggregate risk analysis, which generates a loss distribution for the whole portfolio. If the users does not specify one of the queries $Q_1 ... Q_6$ above, the default query that is used to analyze this default scenario is used. In this section, we discuss the queries in detail and give the default queries for the default analysis scenario.

3.5.1 Data Filtering Queries

There are three data filtering queries, Q_1, Q_2 and Q_3 in Figure 3.1, that are used by the different filters in QuPARA to extract the entries relevant for the analysis from the data tables.

Q_1 is the query to be processed by the *layer filter*. This query defines the layers in the PFT to be processed in the analysis. In the default analysis, we are interested in performing the analysis on the entire portfolio. Therefore, the default query for Q_1 is:

SELECT $*$ **FROM** PFT

Q_2 is the query to be processed by the *ELT filter* to extract the relevant XELTs from the EDP to provide exposure loss data in the analysis calculations in the *mapper* component in the *core engine*. After the layer filtering process is done in the *layer filter*, a relevant set of *elt_IDs* will be send from to the *ELT filter*. The *ELT filter* will extract all the tuples from the EDP which the *elt_ID* are in the set to form a base XELT set. If there is a user requirement specified, for example, excluding exposures in certain regions from the analysis, the requirement will be applied on the base XELT set to form the final XELT set. Otherwise, the base XELT set will be used as the final XELT set to do the analysis. In the default analysis, there is no additional user requirement specified and then the default query of Q_2 will be pushed to the *layer filter* is:

SELECT $*$ **FROM** EDP
  WHERE elt_ID **IN** {elt_IDs}

Q_3 is the query to be processed by the *event filter* to extract the event features from the ECT to support loss grouping in the *combiner*. The *event filter* also allows us to exclude certain events, such as flood events, from the analysis. This query is not used by default since in the default analysis we group all event losses in a trial together and do not care about the differences of the events. Therefore, the default query for Q_3 is to select only the $event\_ID$ field of all events in the ECT:

SELECT elt_ID **FROM** ECT

3.5.2 Loss Grouping Query

There are two loss grouping queries, Q_4 and Q_5, that are used by the *combiner* to determine which trial-event losses are aggregated in the final result. Q_4 is used to control aggregation in the combiner before sending these aggregated results to the reducer. Q_5 is used by the *reduce key generator* to control which loss values are sent to the same reducers for aggregation.

Q_4 is the query to be processed by the *combine function* to specify the grouping rules for the event losses in the same trial in YELT. The event losses can be grouped by their characteristics, such as region or peril, as well as by time frame in a year. In the default analysis, all the event losses are combined to produce a single trial-level loss. Therefore, the default query for Q_4 is:

SELECT trial_ID, TRIAL_SUM(estimated_Loss)
  FROM YELT

Q_5 defines aggregation groups for the losses in different trials. The *reduce key generator* assigns to each loss group in the YRPLT one or many reduce keys for producing the final result in the *reducer*. For most of the analyzes, only one key is assigned to each grouped trial loss. However, in case of multi-marginal analysis, we need to generate different loss combinations of the base portfolio and joining layers to examine the impact of accepting new contracts into the portfolio. In the default portfolio aggregate risk analysis, we output all the trial losses in one table, and then generate a portfolio loss distribution based on this table. Therefore, the *reduce key generator* assigns the same key to the losses in the YRPLT. This is expressed by the following default query:

```
SELECT trial_ID, estimated_Loss
    FROM YRPLT
        GROUP BY TRIAL
```

3.5.3 Result Reporting Query

The result reporting query, Q_6, is used to define the final output of the analysis. This final output could be a loss distribution, loss statistics, risk quantification metrics, or a Exceedance Probability (EP) curve. In the standard analysis, the result of the analysis is the statistical distribution of the portfolio loss. Thus, no aggregation is required in the reducer, that is, the reducer simply outputs all entries in the PYRPLT is receives. This is expressed by the following default query:

```
SELECT * FROM PYRPLT
```

3.6 Loss Grouping

The *combiner* performs loss grouping based on the user query Q_4 to aggregate per-event losses into trial-level losses. After joining the YELT with table T_9, the event losses may be grouped according to three fields, which are time index, region, and peril. The user query Q_4 may specify rules which use one or more of these fields for grouping. If there is only one grouping rule specified in Q_4, we will firstly sort the YELT by the filed used for grouping and sum the event losses in each group to create a table of grouped losses. For example, if Q_4 request the losses are grouped by region, we sort the event losses by their region fields, and then sum the losses with the same region. In a more complicated scenario, there are multiple grouping rules specified in Q_4. For example, the user may ask for a loss distribution by peril, region and season of the year. In such a scenario, it is necessary to sort the event losses by multiple attributes to produce the event groups. In such a scenario, we always sort the losses in a trial primarily by time index, and then create loss groups to represent seasons. Afterwards, in each season, the losses are sorted by region, and then by peril. After the sorting is done, the event losses in the smallest sub-groups in each season are aggregated to create a grouped loss with a unique combination of event characteristics as the group key G.

3.7 Reduce Key Generator

The *reduce key generator* uses the user query Q_5 to assign a reduce key to each of the loss values produced by the combiner. Usually, since the group key G is built by a unique combination of event characteristics, it can be used directly as the reduce key. However, in certain circumstances, such as multi-marginal analysis, the same loss group may be used many times in different combinations of the base portfolio and additional layers. Assuming we have a base portfolio PF and three additional layers L_1, L_2, and L_3. If we want to add two of the three layers to the portfolio, we need to compare the different combinations, which are PFL_1L_2, PFL_1L_3 and PFL_2L_3, to select the best two layers. The combinations cannot be achieved by the reducer by assigning a single key to each of the loss groups. In multi-marginal analysis, the *reduce key generator* holds a complete ordered set of the additional layers and assigns one or multiple reduce keys R to each loss group value, one per combination of base portfolio and additional layers the loss value participates in. This ensures that the loss value is sent to multiple reducers, each of the reducers produces a loss distribution for one combination of base portfolio added layers.

3.8 Final Result Production

The *reducer* produces the final result based on the output from the *combiner* and then forwards the output to the Distributed File System (DFS) according to the user query Q_6. The input received nu a reducer is a set of loss values with the same reduce key. A typical request to QuPARA is to generate an annual portfolio loss distribution. Therefore, a trivial reducer is used, which outputs all the loss values it receives. For other queries, such as loss statistic or risk quantification metrics, different statistical functions are applied on the loss set to generate the results.

3.9 Scenario Risk Analysis Query Example

In this section, an example of an ad hoc request on QuPARA is given to illustrate the use of queries Q_1 ... Q_6 to express this request. The request is to generate a report on seasonal PML with confidence level of 99% due to hurricanes and floods that affect all commercial properties in different locations in Florida.

Q_1: The first part of processing any user query is the query to be processed by the *layer filter*. In this case, we are interested in all layers covering commercial properties, which translate into the following SQL query:

```
SELECT * FROM PFT
        WHERE lob IN commercial
```

Q_2: The second query is passed to the *ELT filter* to extract the XELTs relevant for the analysis. In this case, we are interested in all XELTs in the ELT set, *elt_IDs*, returned by query Q_1 and which cover Florida (FL) as the region and hurricanes (HU) and floods (FLD) as perils:

```
SELECT * FROM EDP
    WHERE elt_ID IN elt_IDs
        AND region IN FL
        AND peril IN {HU, FLD}
```

Q_3**:** This query is provided to the *event filter* for retrieving event features required for grouping estimated losses in the YELT. In this case, we are interested in grouping losses by different perils. Therefore, we need to extract the region information for each event from the ECT to support loss grouping in the *combiner*.

```
SELECT elt_ID, peril FROM ECT
```

Q_4**:** This query is provided to the combiner for grouping all event losses in a trial in the YELT based on their grouping attributes. In this case, the grouping attributes are the *time_Index* and the *peril* associated with each event. The *combiner* first partitions all event losses into 4 buckets based on their *time_Index* values. In each bucket, event losses, which share the same *peril*, in the same trial are combined together to create a trial-level peril loss in a particular season. In the end, all the zero losses will be discarded. In this example, since we only use the hurricane and flood exposure data in Florida, all event losses in other locations and perils will be evaluated as zero and discarded.

```
SELECT trial_ID, TRIAL_SUM(time_Index, 4, peril, estimated_Loss)
```

Q_5**:** This query is provided to the combiner to define the reduce groups for the estimated losses in loss groups in different trials. The reducer receives all losses with the same reduce key and then produce results. In this case, we have eight reduce groups which are the mix of four seasons and the two peril groups.

```
SELECT trial_ID, estimated_Loss
        FROM YRPLT
                GROUP BY peril, SEASON(4)
```

Q_6**:** This query is provided to the reducer to define the final output of the user request. The seasonal PML with 99% confidence level is estimated by the PML function. As the result, since there are eight reduce groups, we will receive eight PML values, one per season/peril group.

```
SELECT PML(0.01) FROM PYRPLT
```

Chapter 4

QuPARA System Implementation in Hadoop

In this chapter, we present an implementation of QuPARA based on Apache Hadoop, Apache Hive, and Pentaho. Apache Hadoop [12] is an open-source software framework that implements the MapReduce programming model. We used Hadoop to implement the *core engine* in the QuPARA framework. Apache Hive [16] is a data warehouse system built on top of the Hadoop Distributed File System (HDFS) and Hadoop. Hive supports data summarization and ad hoc queries using SQL-like language called Hive-QL. We used Hive to implement the *data filters* in QuPARA. Pentaho [32] is used to build a web-based user interface to interact with the system users and translate the user requirements into ad hoc queries to perform specialized risk analysis in QuPARA.

4.1 System Overview

In Chapter 3, we described the design of QuPARA in terms of the abstract MapReduce programming model. In this Chapter, we describe a practical and efficient implementation of QuPARA that based on the abstract framework design presented in Chapter 3. The implementation addresses the constraints pointed by real system software (Hadoop, Hive, and Pentaho) and hardware. During the implementation, significant work was required to realize the framework design in a practical running analysis system.

Figure 4.1 illustrates the implementation of QuPARA. The communication pipelines (P_1 ... P_6) between subsystems are used to transfer data and queries. The analysis begins when a user fills an analysis request with specifications using the *Pentaho*

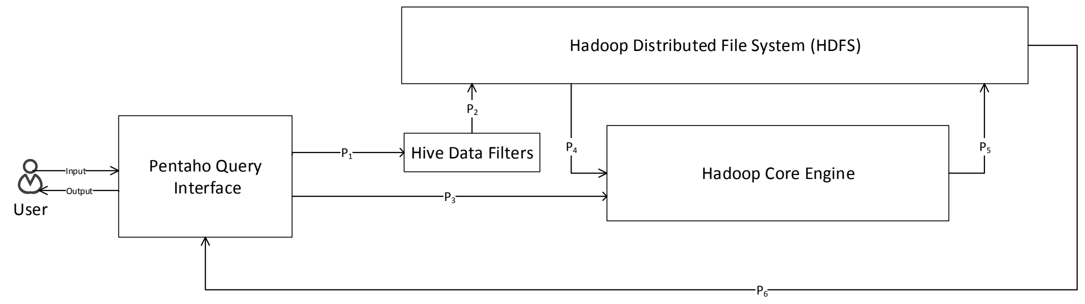


Figure 4.1: Overview of the final implementation of QuPARA

Query Interface. The *Pentaho Query Interface* transforms the specifications into a set user queries, which were introduced as Q_1 ... Q_6 in Section 3.5 on page 45. The user queries are divided into three groups, which are data filtering queries, loss grouping queries, and a result reporting query. The data filtering query is passed into the *Hive Data Filter* through P_1. The *Hive Data Filter* executes the data filtering queries and pass the results through P_2 to HDFS. To initiate the risk analysis, the *Pentaho Query Interface* passes the loss grouping queries and the result report query to the *Hadoop Core Engine* through P_3. The *Hadoop Core Engine* reads input data from HDFS through P_4. At the end of the analysis, the *Hadoop Core Engine* delivers the analysis result to the HDFS through P_5. Finally, the *Pentaho Query Interface* extract the analysis results from the HDFS through P_6 and generate a result report as the output for the user.

4.2 Hadoop Core Engine

The *Hadoop Core Engine* is an implementation of the *core engine*, refer to Figure 3.1 on page 34, in Apache Hadoop. One significant issue we had to address during the implementation of the *core engine* was that a single worker node does not have enough memory to store all XELTs relevant to a large portfolio. In many cases, the complete CELT is too large to be held in the memory on a single worker. In order to address

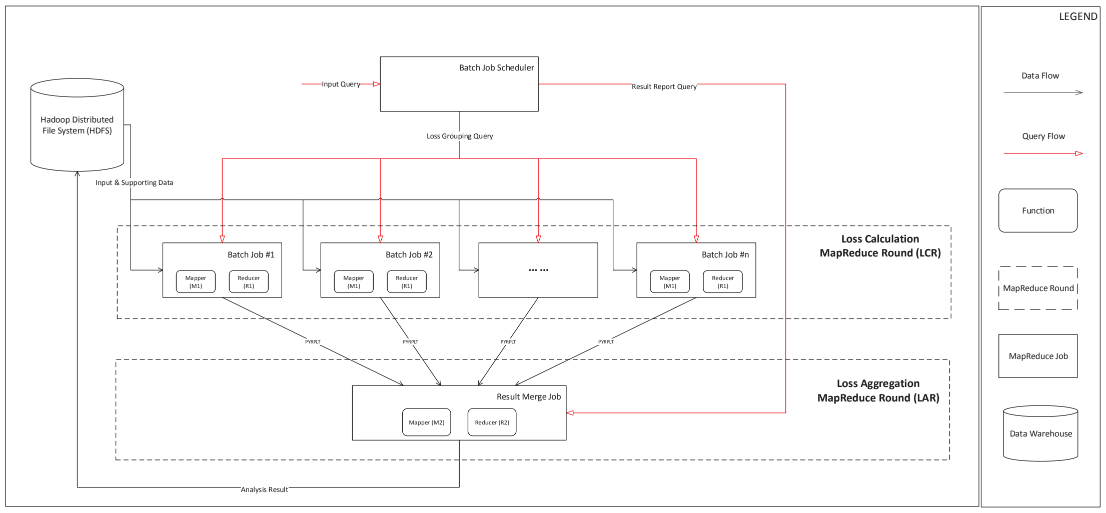


Figure 4.2: Implementation of the hadoop core engine

this constraint, we decided to split the loss calculations in the aggregate analysis by breaking the portfolio into multiple sets of layers, and then distribute these layer sets over multiple loss calculation MapReduce jobs to do loss calculations for each layer. Afterwards, we aggregate all the results from the loss calculation MapReduce jobs to generate the final result. In practice, a two-round MapReduce approach was required in implementing the *core engine* instead of the single round MapReduce approach described in Chapter 3 in order to address the memory constraint.

Figure 4.2 shows the internal structure and workflow of the *Hadoop Core Engine*. The engine is split into three components, which are the *batch job scheduler*, the Loss Calculation MapReduce Round (LCR), and the Loss Aggregation MapReduce Round (LAR). The *batch job scheduler* is responsible for splitting the input portfolio, launching MapReduce jobs, and distributing input queries. The LCR is used to calculate and group losses for each of the layers in the portfolio. The LAR collects and aggregates the layer losses produced from the LCR to produce the final results.

4.2.1 Batch Job Scheduler

The *batch job scheduler* is a program which is used to split the input portfolio, distribute user queries and launch analysis. The goal of the *batch job scheduler* is to divide the analysis job into batch jobs that do not exceed the capacity of the worker nodes. Since there is overhead associated with the execution of each batch job, the *batch job schedule* tries to create the largest individual jobs that can be run. Given that, there is totally n layers in the portfolio, and each worker can process maximum k layers. We calculate the number of batch jobs x we need as $x = \lceil n/k \rceil$.

One approach to schedule the analysis would be to run batched jobs sequentially with each job using all the computing resources. However, the overhead associated with executing each job is significant. If we process x batched jobs one by one, we incur this overhead x times, which degrades the overall performance of QuPARA.

An alternative approach is to run all batched jobs in parallel on p processors, which incurs job execution overhead just $\lceil x/p \rceil$ times. As we described, we need at least x batch jobs to complete the analysis, and we want to equally distribute all the workers to these x jobs to keep load balance. Therefore, the number of processor per batch job y can be calculated as $y = \lceil p/x \rceil$. A concern about splitting computing resources to multiple batched jobs was that it would increase the processing time for a single job since fewer processors are involved in each job. However, since the total computation required to process the entire portfolio was fixed, we expected the total computation time in this parallel approach is the same as in the sequential approach.

The *batch job scheduler* first receives a set of user queries, which includes loss grouping queries, and a result reporting query, from the *Pentaho Query Interface*. Afterwards, it splits the layers in the portfolio equally into x batch jobs, and then launches LCR to execute these jobs with the loss grouping queries. After all the batch jobs in LCR are successfully completed, the *batch job scheduler* will launch the LAR with the *result report query* to produce the final result.

4.2.2 Loss Calculation MapReduce Round (LCR)

Loss Calculation MapReduce Round (LCR) is the first MapReduce round which is used to produce intermediate aggregated losses. In LCR, the batch jobs created by the *batch job scheduler* are executed in parallel. Each of the batch job takes the same YET but different LLT and XELTs as the input.

Batch job in LCR combines the roles of the *mapper* and the *combiner* in the QuPARA framework as described in Chapter 3 (see Figure 3.1). In a batch job, the *mapper* component, *M1* on Figure 4.2, is implemented as same as the *mapper* in the *core engine* as described in Section 3.4.1 on page 42. The *reducer* in the batch job, *R1* on Figure 4.2, is implemented as same as the *combiner* in the *core engine* as described in Section 3.4.2 on page 43. Therefore, the intermediate results produced by the batch jobs are PYRPLT entries as described in Section 3.3.11 on page 41.

4.2.3 Loss Aggregation MapReduce Round (LAR)

Loss Aggregation MapReduce Round (LAR) is the second MapReduce round which collects all the intermediate results from LCR to produce the final result of the analysis. The input of LAR is consumed by a trivial mapper, and then the PYRPLT entries are emit to the reducer(s), *R2* on Figure 4.2, based on their $group/_ID$. The reducer in LAR is implemented based on the concepts and the reduce algorithm described in Section 3.4.3 on page 45.

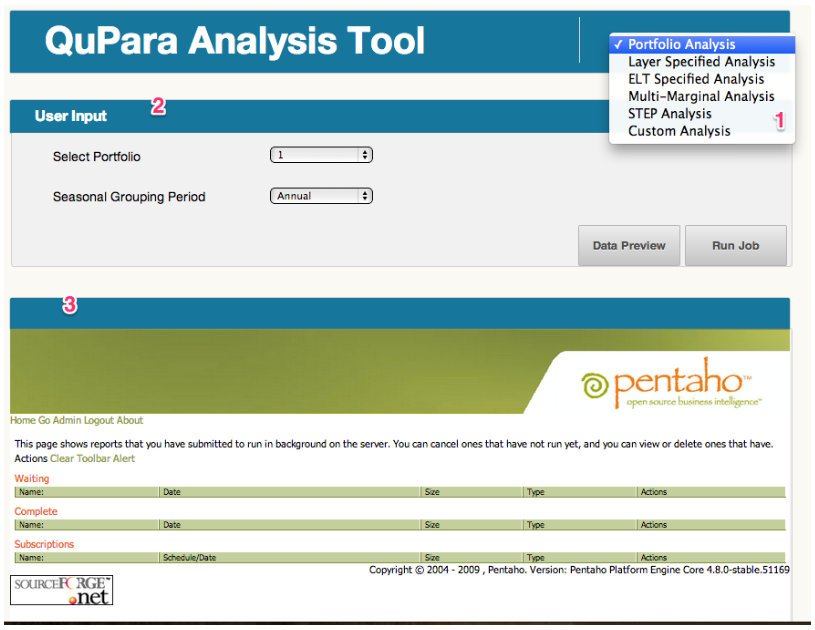
4.3 Hive Data Filter

The component *data filters* in the original QuPARA framework design, refer to the Figure 3.1, was implemented as the *Hive Data Filter* by using Apache Hive. In this implementation, the tables, which are PFT, EDP, and ECT, were not stored as raw data in the HDFS but as the form of data tables in the Hive data warehouse. The layer filter, ELT filter, and event filter, described in the original framework design, implemented using Hive. By using Hive-QL, we could directly execute the *data filtering query* to extract data from the data tables in the Hive data warehouse.

The behaviors of data filters was changed in the final implementation form the original framework design. In Figure 3.1, we perform data filtering in the process of MapReduce job. For example, the layer filter and ELT filter are executed by the *mapper* to extract data from HDFS to build the LLT and the CELT. Also, the *combiner* executes the *event filter* to extract data to form the *Pruned Event Catalogue Table*. However, if we submit the *data filtering queries* to the filters in each mapper and combiner in practice, the huge amount of ad hoc query requests to the Hive-QL engine will cause a significant delay. Since the mapper and reducer cannot progress until all the input data is received, there will be a large time penalty in the overall system performance. We observed that all the mappers and combiners in an analysis execute the same *data filtering queries*, and they will eventually receive the same results from the filters. Therefore, instead of executing the queries multiple times in the MapReduce process, we execute the *data filtering queries* once to produce an intermediate result which is stored in HDFS. Then the mapper and reducer only need to access this intermediate result through the distributed cache, rather than actually running the query.

4.4 Pentaho Query Interface

The *Pentaho Query Interface* was used as a user interface to provide easy access of the QuPARA system. The interface was designed and implemented as a web portal, and it allows users to initiate analysis by either specifying their requirements in terms of a set of parameters to form queries or by writing their own queries in a SQL-like syntax. Figure 4.3 shows the start screen of the *Pentaho Query Interface*. There are



Select Portfolio: 1

Seasonal Grouping Period: Annual

This page shows reports that you have submitted to run in background on the server. You can cancel ones that have not run yet, and you can view or delete ones that have.

Actions Clear Toolbar Alert

Waiting

Name:	Date	Size	Type	Actions

Complete

Name:	Date	Size	Type	Actions

Subscriptions

Name:	Schedule/Date	Size	Type	Actions

Copyright © 2004 - 2009 , Pentaho. Version: Pentaho Platform Engine Core 4.8.0-stable.51169

Figure 4.3: Pentaho portfolio risk analysis page

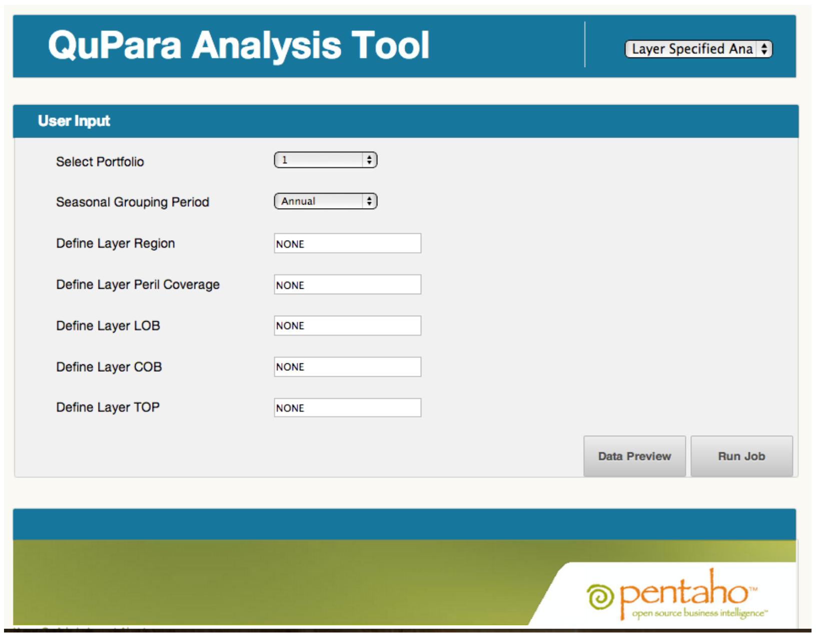
three basic component which are the *analysis selection menu* (1), the *user input field* (2), and the *QuPARA job scheduler dashboard* (3). In this section, we illustrate the *Pentaho Query Interface* and explain each component in the interface in detail.

4.4.1 Analysis Selection and User Input

The *analysis selection manual*, component 1 on Figure 4.3, is used to select a type of analysis from the analysis options the system supports. On the manual, there are seven types of analysis, which are *portfolio analysis*, *layer specified analysis*, *ELT specified analysis*, *multi-marginal analysis*, *STEP analysis*, and *custom analysis*. The *user input field*, component 2 on Figure 4.3, allows users to specify parameters to the selected analysis.

4.4.2 Portfolio Analysis

The default analysis in QuPARA is the aggregate risk analysis which computes a portfolio loss distribution. In the *user input field*, as shown on Figure 4.3, there are



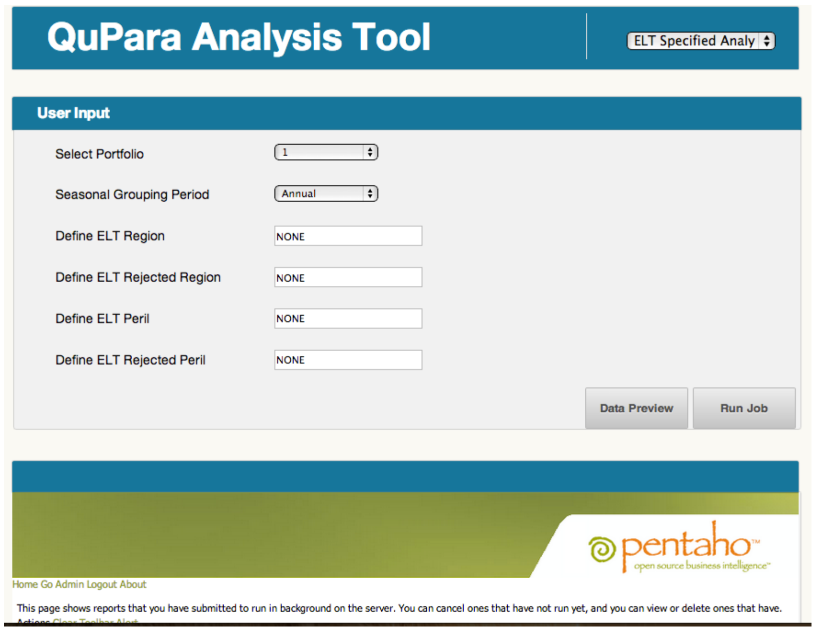
Select Portfolio	1
Seasonal Grouping Period	Annual
Define Layer Region	NONE
Define Layer Peril Coverage	NONE
Define Layer LOB	NONE
Define Layer COB	NONE
Define Layer TOP	NONE

Figure 4.4: Pentaho layer specified analysis page

two parameters which users can specify to initialize the analysis. The first parameter is used to specify which portfolio to be analysis. The second parameter is used to set the seasonal grouping rule.

4.4.3 Layer Specified Analysis

The layer specified analysis is shown on Figure 4.4. This page allows users to analyze a subset of a portfolio. In the *user input field* on this page, there are five layer filtering parameters which define the layer characteristics for the analysis. The default values of these parameters are set to be "NONE", which means all layers will be selected. The layer filtering parameters can be used to select a certain subset of the portfolio. For example, if we only want to analysis the layers associated with earthquake (EQ), we can specify "EQ" in the layer peril coverage input field.



Select Portfolio	1
Seasonal Grouping Period	Annual
Define ELT Region	NONE
Define ELT Rejected Region	NONE
Define ELT Peril	NONE
Define ELT Rejected Peril	NONE

This page shows reports that you have submitted to run in background on the server. You can cancel ones that have not run yet, and you can view or delete ones that have.

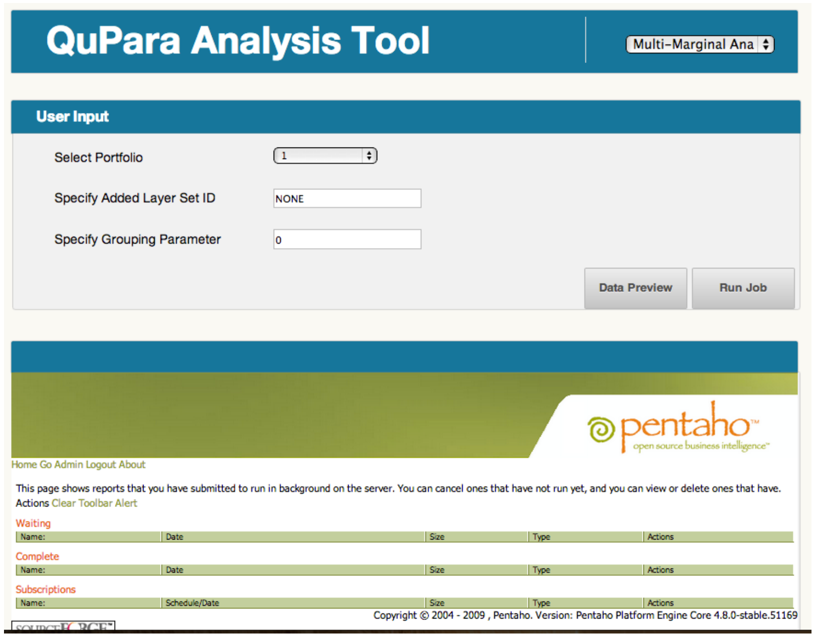
Figure 4.5: Pentaho ELT specified analysis page

4.4.4 ELT Specified Analysis

The ELT specified analysis is shown on Figure 4.5. This page allows users to specify which exposure data will be involvedw in the analysis. In the *user input field* on this page, there are four parameters which help the users to select or exclude certain exposure data involved in the analysis. For example, if we want to exclude all the losses from Japan (JP), we can specify "JP" in the input filed of ELT reject region.

4.4.5 Multi-marginal Analysis

The multi-marginal analysis page is shown on Figure 4.6. In the multi-marginal analysis, users want to exam the impact on the performance of a portfolio by adding new layers into it. There are three parameters on this page, which are the base portfolio, the adding layer set, and the grouping parameter. The base portfolio is the portfolio the user currently holds. The adding layers are the new contracts which the user need to decide to take or not. The grouping parameter specifies the rules of adding layers into the base portfolio. For example, if the user specifies five adding



Select Portfolio: 1
Specify Added Layer Set ID: NONE
Specify Grouping Parameter: 0

This page shows reports that you have submitted to run in background on the server. You can cancel ones that have not run yet, and you can view or delete ones that have.
Actions Clear Toolbar Alert

Waiting

Name:	Date	Size	Type	Actions

Complete

Name:	Date	Size	Type	Actions

Subscriptions

Name:	Schedule/Date	Size	Type	Actions

Copyright © 2004 - 2009 , Pentaho. Version: Pentaho Platform Engine Core 4.8.0-stable.51169

Figure 4.6: Pentaho multi-marginal analysis page

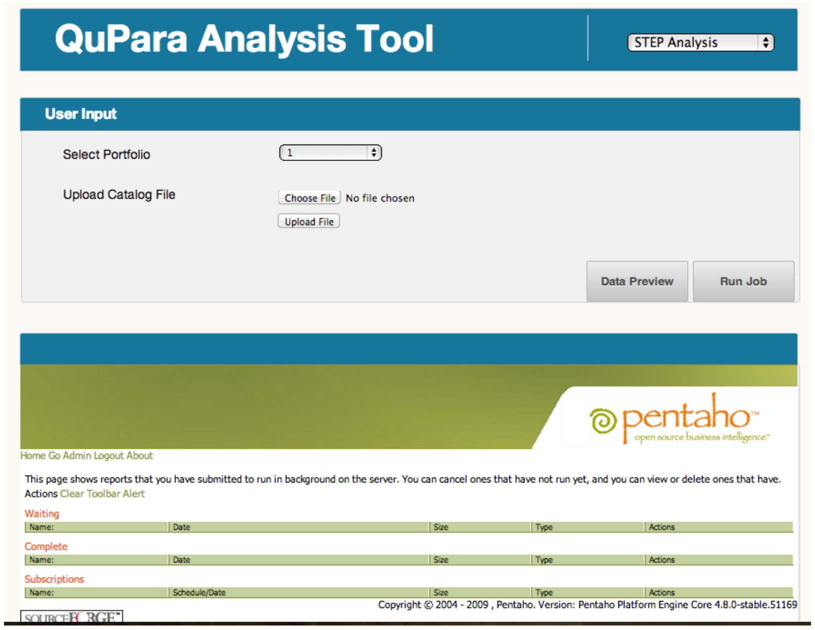
layers, but only want to select the best two of them, the grouping parameter will be set as 2.

4.4.6 STEP Analysis

The STEP analysis page is shown on Figure 4.7. In this analysis, the user is allowed to upload its customized event catalogue to create a loss portfolio for un-known event. The uploaded event catalogue will be used to create a simulation event data, which is a YET. And then a portfolio will be mapped into this simulation data to generate the portfolio loss distribution for the new event.

4.4.7 Custom Analysis

The custom analysis page is shown on Figure 4.8. In this page, the user is allowed to enter the six user queries in SQL-like syntax, which was introduced in Section 3.5 on page 45. On the input fields, if there is no user query specified, the default query will be applied in the analysis.


QuPara Analysis Tool

STEP Analysis

User Input

Select Portfolio: 1

Upload Catalog File: Choose File No file chosen

Upload File

Data Preview | Run Job

Home Go Admin Logout About

This page shows reports that you have submitted to run in background on the server. You can cancel ones that have not run yet, and you can view or delete ones that have.

Actions Clear Toolbar Alert

Waiting

Name:	Date	Size	Type	Actions

Complete

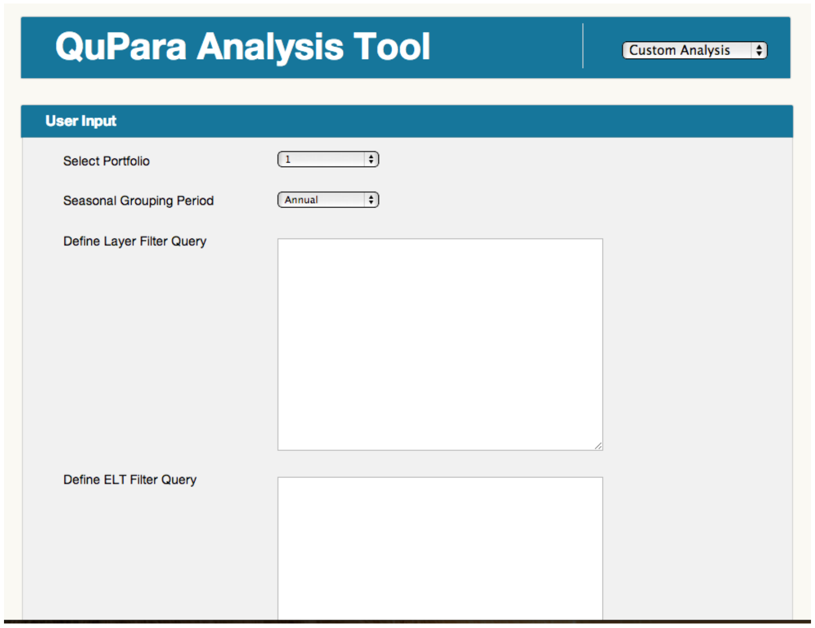
Name:	Date	Size	Type	Actions

Subscriptions

Name:	Schedule/Date	Size	Type	Actions

Copyright © 2004 - 2009 , Pentaho. Version: Pentaho Platform Engine Core 4.8.0-stable.51169

Figure 4.7: Pentaho STEP analysis page


QuPara Analysis Tool

Custom Analysis

User Input

Select Portfolio: 1

Seasonal Grouping Period: Annual

Define Layer Filter Query

Define ELT Filter Query

Figure 4.8: Pentaho custom analysis page

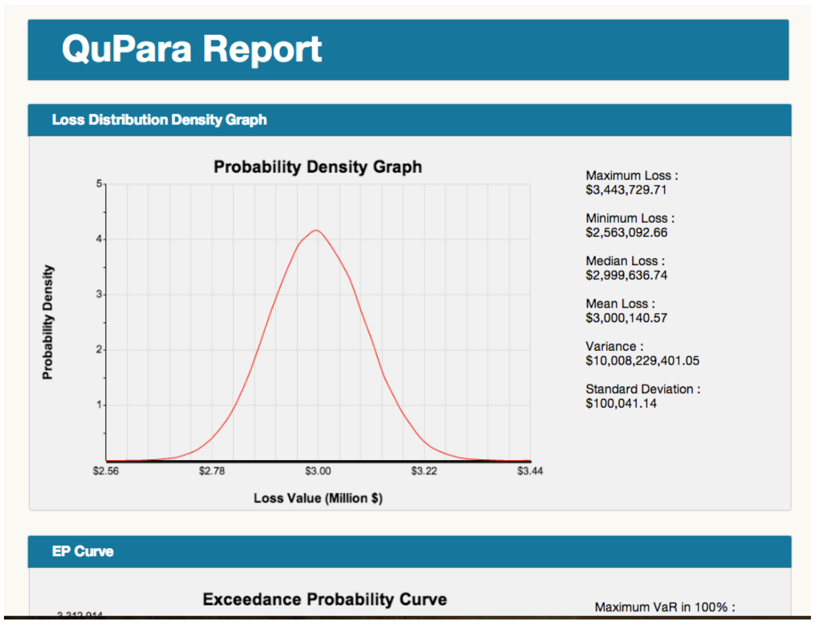
After setting all the parameters for the analysis, there are two options the user can choose, which are the *data preview* and the *run job* options. The data preview option allows user to check the selected layers and exposure data before executing the analysis job. When the user hit the *run job* button, Pentaho will push the job into a queue in the job scheduler.

4.4.8 Job Schedule Dashboard

The job schedule dashboard, component 3 in Figure 4.3, is a job queue system which keeps track of the status of the submitted jobs. When the user hit the *run job* button, an analysis job will be created and push into the job queue to wait for execution. At this time, an analysis job entry will appear under the waiting list. After an analysis is completed, a history record will show on the complete list with a link to the result report page of the completed analysis.

4.4.9 Result Report

The result report page is the final product of the analysis performed by QuPARA. The report visualizes the analysis result to the user using various visualization techniques. A sample result report page, which demonstrates a dummy portfolio distribution, is shown on Figure 4.9.


QuPara Report

Loss Distribution Density Graph

Maximum Loss :
$3,443,729.71

Minimum Loss :
$2,563,092.66

Median Loss :
$2,999,636.74

Mean Loss :
$3,000,140.57

Variance :
$10,008,229,401.05

Standard Deviation :
$100,041.14

EP Curve

Exceedance Probability Curve

Maximum VaR in 100% :

Figure 4.9: Pentaho sample result report page

Chapter 5

Data Structure Design and Implementation

One of the most costly steps in the aggregate risk analysis algorithm is to retrieve loss data from the XELTs of each layer in a portfolio. In order to calculate an aggregated loss for a layer in an event from the YET, we need to look up the event in all XELTs of the layers to retrieve raw loss data. Since there is no correlation between the orders of events in the YET and in the XELTs, this amounts to random entry accesses in random tables. Such random accesses are very time consuming. Our goal in this chapter is to develop a data structure, the CELT, that can speed up such assesses momentously.

We start by introducing a dictionary design the CELT. In order to obtain the most efficient CELT implementation, we carefully select the underlying dictionary data structures and tune the execution environment parameters. The final CELT was assembled step by step according to a set of implementation decisions, based on a series of experiments.

The platform for our sequential experiments was a single node of a Rocks cluster [34], with 2.66Ghz Quad Core Intel Xeon Processor X3350, 4 GB DDR2 RAM, and three 1 TB 7200 RPM SATA disk driver. The node runs CentOS 6.3 [35] with Java version 1.7.0_03 [36]. The maximum Java heap memory size was set to 2 GB for each experiment. Each input XELT contained 10,000 loss entries and each entry consisted of one integer and four doubles.

5.1 XELT Data Generator

Since the choices we made in designing and knowing our CELT implementation was supported by carefully designed experiments to determine the optimal choices of underlying data structure and parameters, we first describe the data set we used in these experiments.

An XELT is an exposure data tables which stores loss information for a certain type of catastrophic event in a specific geographical division. XELTs are essentially dictionaries consisting of key-value pairs in which the key is a unique catastrophic event identifier and the value is the corresponding loss exposure. The number of entries in a single XELT varies from a few hundred to several millions and the average number is approximately ten thousand. The number of XELTs associated with a reinsurance layer may vary and a single XELT may be associated with multiple layers. On average, there are approximately five unique XELTs per layer. A typical reinsurance portfolio may include thousands of layers.

The exposure data in a XELT is generated based on the treaty terms, statistical data associated with each catastrophic event, and property data in different locations. Each reinsurance company has its unique exposure data set, and the data set serves as the main loss information source. The exposure data set is property of the reinsurer and is not transferable. Therefore, we were unable to work with real-world exposure data in our experiments. Instead, we used synthetic exposure data generated and collected from catastrophe model data generators with scientific data to simulate real-world use cases.

The exposure data generation starts by creating a proper Master Event Catalogue Table (MECT), which records the features of every possible catastrophic event, such as region, peril, occurrence rate, and severity data. We generated a set of sample events for each peril from an occurrence distribution that describes this type of peril, and then calculate losses for the sample events by using hazard models. For example, we can use the Poisson distribution to predict the total number of earthquake events may occur in a year [37]. Afterwards, we sample this number of earthquake events from all the earthquake records in the MECT based on their occurrence rates. Afterwards, we run a earthquake hazard model which uses the severity data of each occurring earthquake event to against the property data to estimate the losses.

5.2 CELT Data Structure Design

We use simulation data as the starting point for designing and implementing the Combined Event Loss Table (CELT) to ensure that the final CELT implementation performs well on real-world data. In the input, each XELT has a unique identifier represented as a positive integer value. In each XELT, each catastrophic event is identified by a unique integer value as the unique identifier. The value range of both XELT identifiers and event identifiers is from 0 to the maximum 32 bit unsigned integer value, which is 4,294,967,295.

To help with data structure design, we first consider the type of operations that will be performed on the data structure. The contents of the data structure change only in the construction stage. After the data structure is built, no updates are performed. In the aggregate risk analysis algorithm, we perform a series lookups with different XELT identifiers for each event in the YET to support loss calculations for different layers in a portfolio.

5.2.1 Direct Access Data Structure

A natural design of the CELT data structure that supports fast lookup operations based on event identifiers and ELT identifiers is a two-dimensional array. The table uses ELT identifier as the row index and event identifier as the column index. Insert,

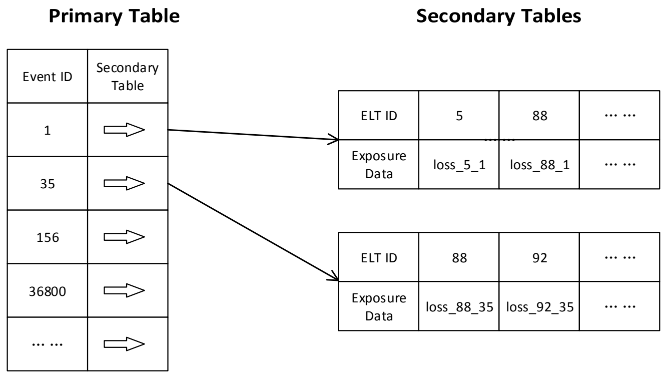


Figure 5.1: The basic structure of CELT

remove and lookup operations in this table are trivial and take constant time. However, according to the data description, both ELTs and events have a wide identifier range. Hence, the table will be very large and most of its entries will be 0. For example, a portfolio with 1000 layers contains approximately 5,000 ELTs, which may include around 50 million entries. In contrast, the size of the direct access table is $(4,294,967,295)^2$. Thus, this data structure would be extremely sparse and have very low memory efficiency. It is not easy to find a machine that can hold a data structure of this size in memory. Thus, the direct access table is not an acceptable choice for our data structure.

5.2.2 Hierarchical Data Structure

We introduce a hierarchical data structure consisting of two lookup layers to implement CELT. The structure of the data structure is visualized on Figure 5.1. The first layer of the CELT is a primary event lookup table. The second layer consists of a pool of secondary ELT lookup tables. A key is an event identifier in the primary table. The associated value is a pointer to a secondary table that contains all ELT entries with this event identifier.

Insert Operation

An insertion into the CELT is split into to two steps. For each event loss entry in each ELT, we first use its event identifier to lookup the corresponding secondary table in the primary table, and then insert the entry into the secondary table. If the in the primary table fails, which indicates that there is no previous loss entry with this event identifier, we create a new secondary table, insert the new table into the primary table with this event identifier as key, and then insert the loss entry into the secondary table. This ensures that only on-empty secondary table are created.

Lookup Operation

To answer a lookup query with a given event identifier and ELT identifier, we first use the event identifier to look up the corresponding secondary table in the primary table. If the secondary table does not exist, we stop searching and report failure of the query. If there is a secondary table associated with the event ID, we use the ELT identifier to find and retrieve the loss data. This secondary lookup can also fail.

In the aggregate risk analysis algorithm, we perform a series lookups in different ELTs for each event. Thus, we can optimize the lookup procedure for query sequence with the same event identifier. In this case, we perform only lookup in the primary table to retrieve the corresponding secondary table. If the secondary table does not exist, we can report failure for all queries in the sequence. If we find a secondary table, we perform a series loss data lookups for the ELT identifiers from in query sequence. In a modern computer, cache memory accesses are significantly faster than accesses in to memory. Therefore, we expect to gain overall performance from this optimized lookup operation.

5.3 HashMap CELT Implementation

The hierarchical CELT implementation we have just described can be realized using different choices of representations of the primary and secondary tables. Hash maps are a normal choice to represent these tables since they support expected constant time insert and lookup operations. Apart from the Java STL, there are a number of high-performance libraries that offer alternative hash map implementations, including Trove and HPPC. This gives us three different CELT implementations based on the choice of the underlying hash map implementation. We refer to these as STL HashMap CELT, Trove HashMap CELT, and HPPC HashMap CELT. We used STL HashMap CELT as the base line implementation. The Trove and HPPC implementations were compared and tested to form an optimized CELT.

5.3.1 STL HashMap CELT

The Java Built-in HashMap is a chained hash table, and usually works well when the number of entries is small. However, as the number of entries grows, the data structure performance degrades. We use the STL HashMap CELT as the base line implementation, and its insert time, memory usage, and lookup time are treated as the baseline for comparisons with other implementations.

5.3.2 Trove HashMap CELT

The GNU Trove library is a free and open-source project that provides high-performance collections in Java. The library consists of a series of fast, lightweight, and easy to use collection data structures for Java. In that CELT, both the primary table and secondary tables used integers as keys and objects as the values associated with these keys, so we used a HashMap implementation called TIntObjectHashMap from

Trove library to implement both tables in the Trove HashMap CELT. The TIntObjectHashMap uses open addressing with double hashing.

5.3.3 HPPC HashMap CELT

The name HPPC stands for high-performance primitive collections for Java. HPPC is a free and open-source project that aims to provide high-performance and memory efficient collections for Java. We built a HPPC HashMap CELT by utilizing the IntObjectOpenHashMap implementation from the HPPC library. The IntObjectOpenHashMap uses open addressing with quadratic collision resolution.

In the following sections, we discuss the performance of the three candidate HashMap CELT implementations and determine the best choice of hash map implementation. We conducted a series of experiments to determine the best environment configurations for each of the candidate data structures and then selected the best one by comparing their performance.

5.4 JVM Parameter Tuning

In Java, the efficiency of an application is highly depending on memory management and garbage collection techniques. While executing an application, the memory management and Garbage Collection (GC) behaviors are driven by the parameters used to initialize the Java Virtual Machine (JVM). In this section, we give an overview of the techniques used in Java to perform memory heap management and garbage collection to understand the basic mechanisms of how JVM utilize the memory. Afterwards, based on the features and behaviors of the CELT, we select and discuss two JVM parameters that could have a significant impact on the performance of the data structures.

5.4.1 JVM Memory Management

In the JVM, the heap memory space is the storage of objects created during runtime. The heap size is initialized with the minimum heap size parameter Xms. As more objects crated, the heap can grow until reaching the maximum heap size, which is specified by the maximum heap size parameter Xmx. Every time the heap size changes, there is a memory management penalty due to GC operations, moving objects around, and other bookkeeping activities.

The JVM uses a generational object memory system, which divides the entire memory heap space into two generations: an old generation and young generation. An object is initially created in the young generation, and it is moved to old generation if it has a long survivor time and survives from GC operations in the young generation.

5.4.2 GC Techniques

Garbage Collection (GC) is the techniques used by JVM to reclaim the memory allocated to objects that are no longer in use. An object is considering live if it is accessible by the application. Any other object is dead. When the percentage of the free heap memory drops below a certain threshold, the JVM runs the garbage collector to reclaim memory allocated to dead objects.

In a GC run, the objects in the young generation are checked first. This is called a minor GC. A minor GC is usually very fast. After going through the young generation, the remaining objects are marked and are moved into the old generation. If the collected memory from a minor GC is not enough to raise the free memory ratio above the threshold, the JVM runs a full GC, which involves the old generation, to seek free memory. The cost of running a full GC is significantly higher than the cost of running a minor GC.

5.4.3 JVM Parameter Tuning

Recall that, we first insert all the entries into the CELT and then perform lookups. This indicates that, in the insertion stage, many objects are created and destroyed. Most of the old generation objects are created in this stage. After all insertions are done, we do not create long-lived objects in the lookup operations. Thus, most of the full GCs will happen in the inserting stage, and only minor GCs should occur in the lookup stage. Thus, we consider to tune JVM parameters to improve the performance of the CELT.

Initial Heap Size (Xms) Selection

Xms is the parameter that determines the initial and minimum heap size used by the JVM. When the maximum allowed heap size and Xms value is different, the memory space used by JVM may grow or shrink the heap during each GC in order to keep the ratio of free space to live objects in a specific range. With a large Xms setting, we can reduce the frequency of overall heap resizing, but the time spent on GC is longer at the beginning. With a small Xms value, GC time is shorter before the heap grows to a large size. However, if the memory usage pattern for an application is fluctuating, the small initial heap size may result in frequent heap resizing and the application performance may suffer from the resize penalties. For the CELT, we expected that a larger Xms leads to better performance since the size of the data structure grows in the insertion stage and stays the same during the rest of the program's execution.

In order to observe the effects of adjusting Xms value on the performance of each candidate CELT implementation, we performed insert and lookup experiments on each of the data structures with three different Xms values: the default value (64 MB), 1.5 GB, and 2GB. In insertion tests, we kept inserting ELTs into the data structure until hitting memory issues. To test the lookup performance of the CELT,

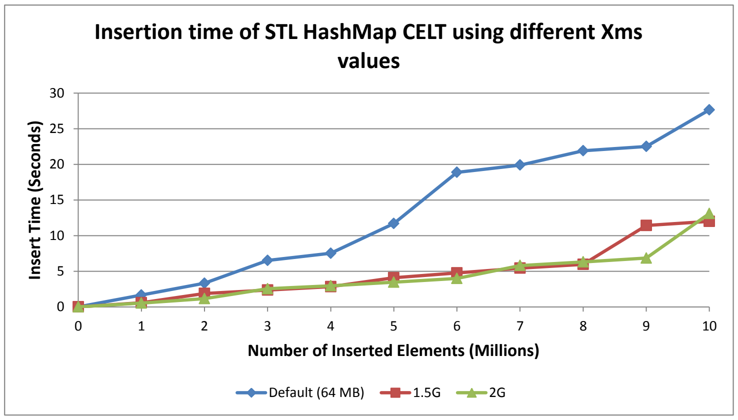


Figure 5.2: Insertion time of STL HashMap CELT using different Xms values

we performed 10 million lookup queries on the data structure constructed by each insertion test, and recorded the total lookup time.

STL HashMap Xms Performance Figure 5.2 shows the comparison of insertion times of STL HashMap CELT using the different minimum heap size settings. Increasing the Xms value decreased the total insert time significantly for each input data. With a higher initial heap size, the need of heap resizing is decreased, so the frequency of GC is decreased.

In order to observe the full effects on the STL HashMap CELT of increasing the value of Xms, we also conducted experiments to test the lookup performance of the STL HashMap CELT. Figure 5.3 shows the lookup time of the STL HashMap CELT using different Xms settings. Again, increasing the value of Xms from the default value improved the lookup performance of CELT significantly. However, the lookup performance did not improve further when changing Xms from 1.5 GB to 2 GB. This is because the heap resizing cost differences between the two Xms settings are insignificant.

Figure 5.4 shows a comparison of the insertion times using different Xms settings

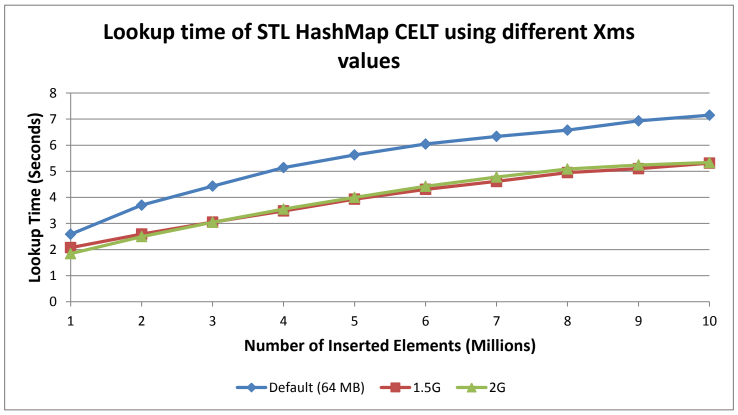


Figure 5.3: Lookup time of STL HashMap CELT using different Xms values

for the Trove HashMap CELT. A larger initial size once again produced better performance by reducing the memory management penalty caused by heap resizing. The Trove HashMap CELT had the best insert performance when the Xms value was set to the maximum allowed heap size of 2G.

Figure 5.5 shows the lookup times of the Trove HashMap CELT using different Xms settings. Increasing the initial heap size improved the lookup time once again, but performance improvement was not as significant as for the STL HashMap CELT. This suggests that the lookup operations in Trove HashMap CELT use less memory than in the STL HashMap CELT.

Figure 5.6 shows the insertion times of the HPPC HashMap CELT using different Xms settings. A larger initial heap size resulted in better insertion performance once again. Thus, by setting the Xms value to 2 GB, we obtained the best insert performance.

Figure 5.7 shows the lookup performance of the HPPC HashMap CELT using different Xms settings. Increasing initial heap size once again improved the lookup performance of the HPPC HashMap CELT. The performance difference between Xms settings of 1.5 GB and 2 GB was still not significant.

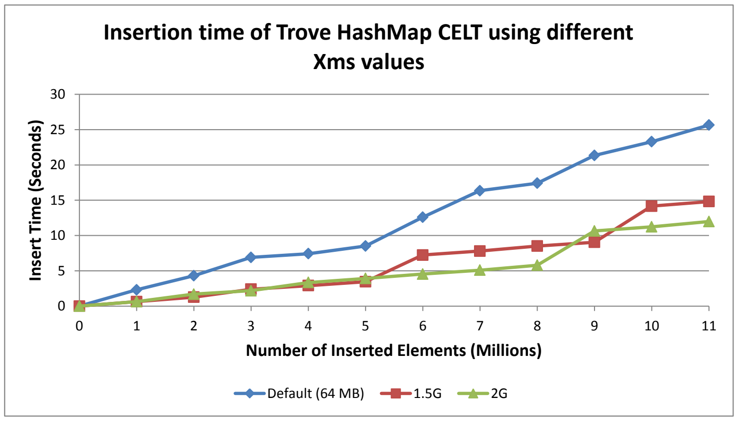


Figure 5.4: Insertion time of Trove HashMap CELT using different Xms values

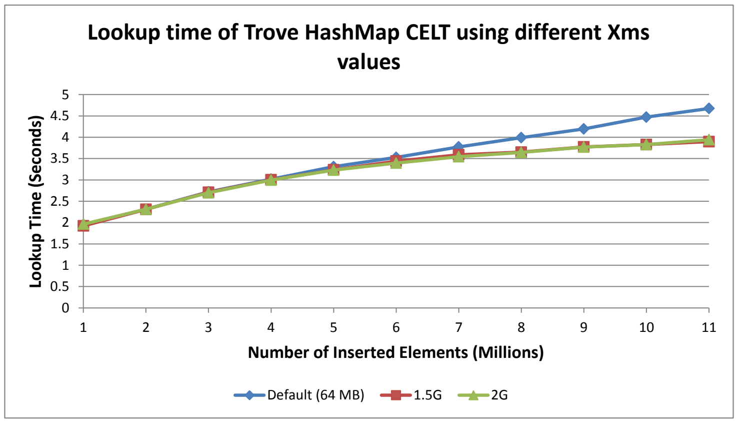


Figure 5.5: Lookup time of Trove HashMap CELT using different Xms values

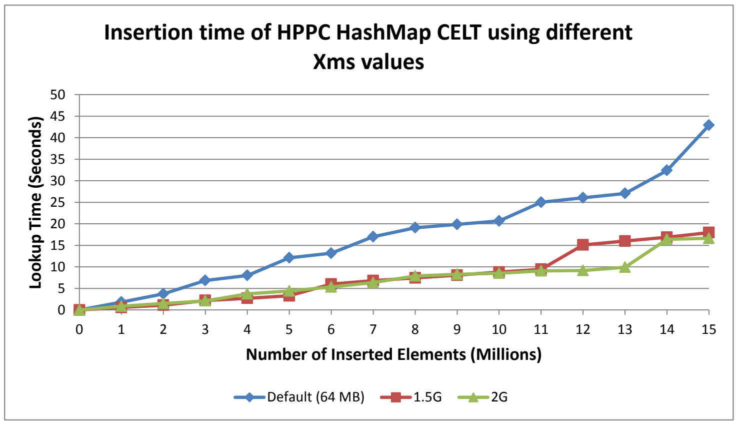


Figure 5.6: Insertion time of HPPC HashMap CELT using different Xms values

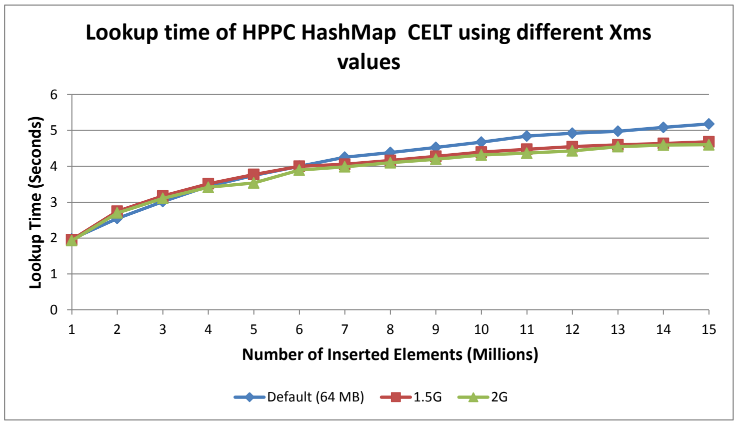


Figure 5.7: Lookup time of HPPC HashMap CELT using different Xms values

In summary, our experiments show that all three CELT implementations achieve the best insertion and lookup performance using the maximum possible heap size of 2 GB.

Young Generation Ratio (NewRatio) Selection

The young generation size in the JVM can affect the frequency and cost of minor and full GC runs. The bigger the young generation, the less often minor GC occurs. Since the total available heap memory is fixed, a larger young germination implies a smaller old generation. If the old generation is filled, the old objects will stick around as young objects in the young generation and these objects are involved in minor GCs.

In the JVM, there are several ways to define the size of young generation. Adjusting the young generation ratio parameter, NewRatio, is the most common and easiest way. The value of NewRatio indicates the ratio between the old generation size and the young generation size. For example, if NewRatio is 1, the heap is divided into two portions of equal size one for the old generation and one for the new generation. When the NewRatio value is set to 3, the old generation will occupy 3/4 of the total heap size, and the remaining is used for the new generation.

In order to determine the effect of the young generation size on the insert and lookup performance of the candidate CELTs, we tested the NewRatio values of 1, 2, 3, and 4, referred to as NewRatio configurations R1, R2, R3, and R4 in the remainder of this section of the optimal initial heap size.

Figure 5.8 shows the insertion times of the STL HashMap CELT using different NewRatio settings. When the number of elements in the data structure is small, a lower NewRatio yields better performance. However, when the number of elements in the data structure exceeded 4 million, the performances of the data structure with smaller NewRatio deteriorated and a larger NewRatio value resulted in better

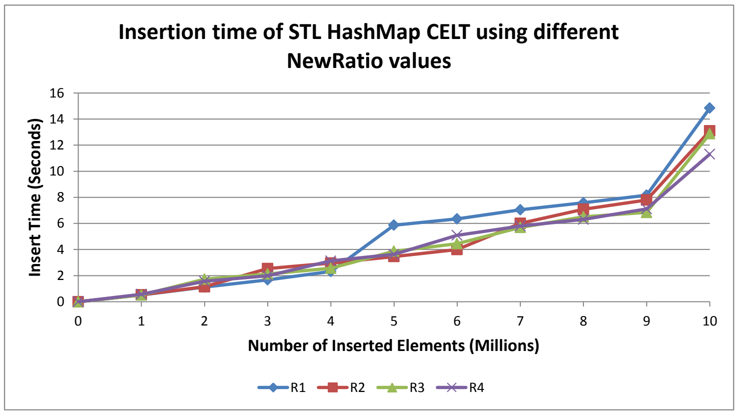


Figure 5.8: Insertion time of STL HashMap CELT using different NewRatio values

performance. Most of the objects created during the construction of the CELT would eventually be moved into the old generation. With a small NewRatio value, the old generation is likely too small to hold the long-lived objects, which forces us to leave some of them in the young generation. In this case, the remaining long-lived objects in the young generation will be re-examined in each minor GC. This is likely the reason why the STL HashMap CELT performed better with a larger NewRatio value if the input size is big.

Figure 5.9 shows the lookup performance of the STL HashMap CELT using different NewRatio settings. We observe that there is no significant performance difference in lookup operations of the STL HashMap CELT using different NewRatio settings. However, using NewRatio equal to 4, the lookup time increases as the input size increase from 9 million to 10 million elements. This is likely because, with a NewRatio value of 4, the young generation is given only 1/5 of the heap. During lookup operations, short-lived objects are created. With a small young generation size, these objects fill the young generation more quickly, which triggers minor GCs more frequently. Therefore, it would be better for us not to set the young generation size too small to avoid frequent GCs.

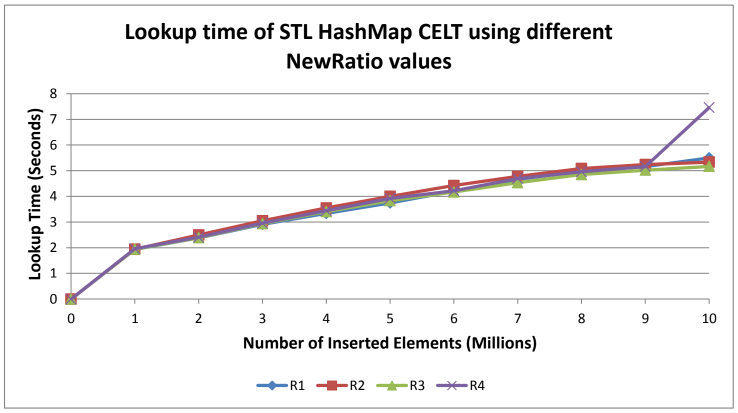


Figure 5.9: Lookup time of STL HashMap CELT using different NewRatio values

Figure 5.10 shows the insert performance of the Trove HashMap CELT using different NewRatio settings. Most of the time, a higher NewRatio value leads to better performance. However, when NewRatio was set to 4, the program performed GC between the 9 million and the 10 million insertions. With the NewRatio equals to 2 or 3, this GC occurred later. This could be caused by the small young generation size.

Figure 5.11 shows the lookup performance of the Trove HashMap CELT using different NewRatio settings. There is no significant difference between all four configurations. Also, we did not observe any GC activity in each of the lookup experiments. This suggests that the Trove HashMap CELT had smaller memory footprints while performing lookup operations than the STL HashMap CELT.

Figure 5.12 shows the insert time performance of the HPPC HashMap CELT using different NewRatio configurations. The data structure performed better when the NewRatio value was set to 3 or 4. There was no significant performance difference between setting the NewRatio parameter to 3 or 4.

Figure 5.13 shows the lookup performance of HPPC HashMap CELT using different NewRatio value settings. A NewRatio of 3 or 4 resulted in the best performance.

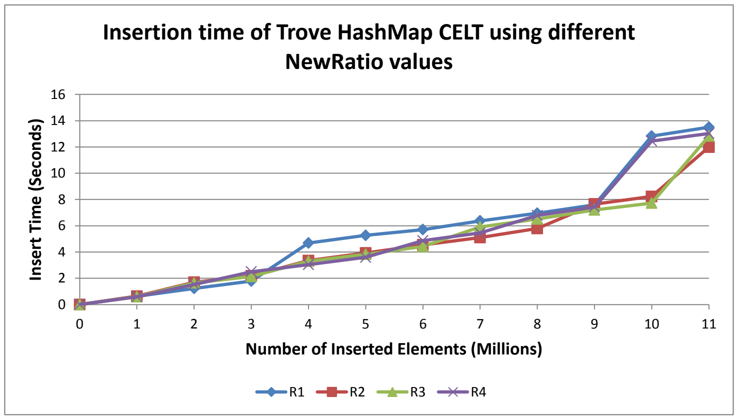


Figure 5.10: Insertion time of Trove HashMap CELT using different NewRatio values

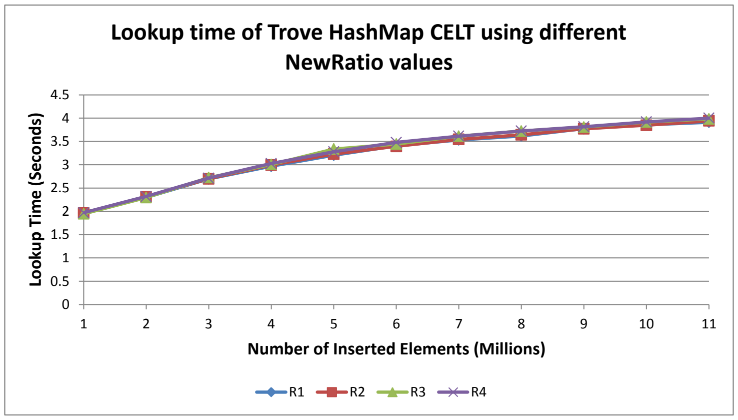


Figure 5.11: Lookup time of Trove HashMap CELT using different NewRatio values

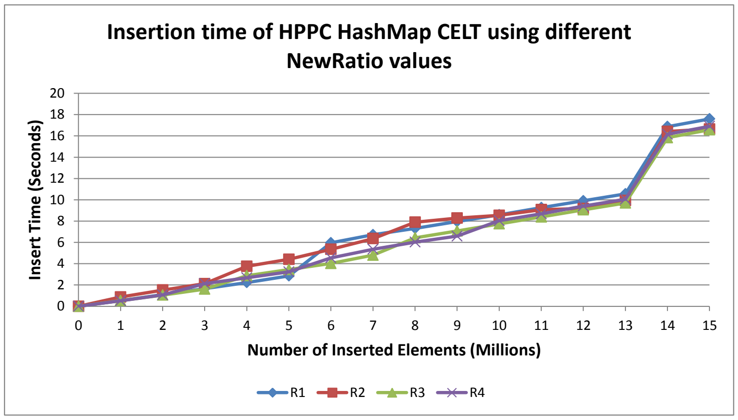


Figure 5.12: Insertion time of HPPC HashMap CELT using different NewRatio values

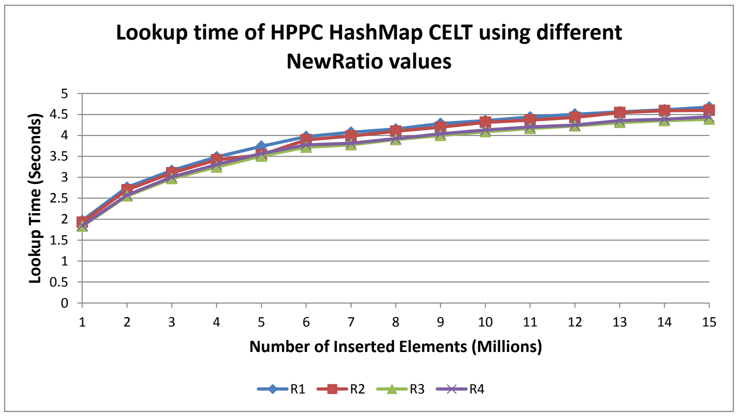


Figure 5.13: Lookup time of HPPC HashMap CELT using different NewRatio values

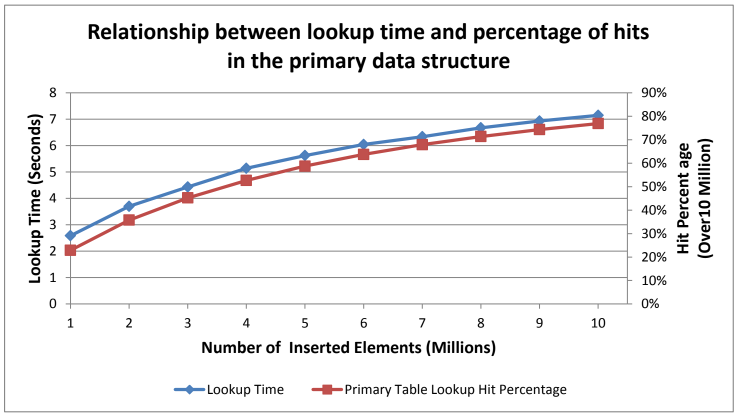


Figure 5.14: Relationship between lookup time and percentage of hits in the primary data structure

From these results, we conclude that the best NewRatio value is 3 for the STL HashMap CELT, 2 for Trove HashMap CELT, and 3 for the HPPC HashMap CELT.

5.4.4 Relationship between Lookup Time and Data Structure Size

From the lookup experiments discussed in the previous section, the lookup cost per element increased with the number of elements in the CELT, no matter which GC settings we chose. A lookup cost per element increased with the primary table contains no secondary table with the given event ID because it can skip the second phase of the operation. Therefore, we suspected that, this increase in lookup time correlates with the frequency of hit in the primary data structure.

Figure 5.14 shows the relationship between total lookup time and percentage of hits in the primary data structure. The horizontal axis is the total number of elements inserted into the data structure; the left vertical axis is the total time spend on executing 10 million lookups; and the right vertical axis is the percentage of hits in the primary data structure over all 10 million lookup operations. The lookup time and hit percentage curves match perfectly which confirms our hypothesis.

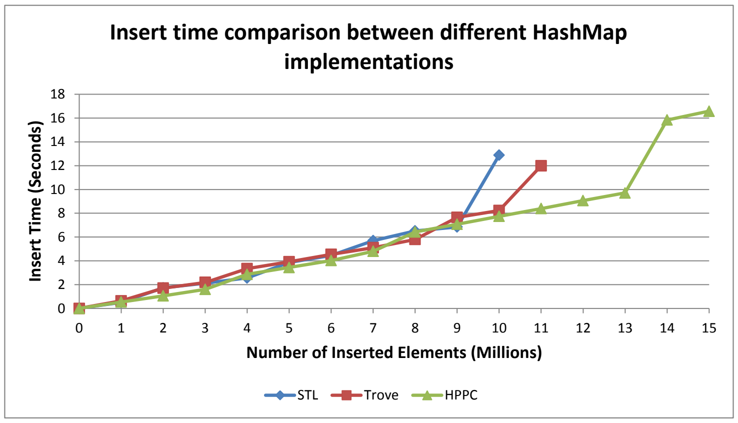


Figure 5.15: Insert time comparison between different HashMap implementations

5.5 HashMap CELT Implementations Comparison

In this section, we choose the best HashMap implementation to use for our CELT implementation by comparing their relative performance for each the optimal JVM parameters determined in Section 5.4.

5.5.1 Insert Performance Comparison

Figure 5.15 compares the insertion times of the three hash map CELT implementations. Figure 5.16 compares the memory consumption. The three implementations achieved the same insert performance. However, the STL HashMap CELT had the highest space overhead and could not handle more than 10 million insertions. The Trove HashMap CELT was able to handle up to 11 million insertions. The HPPC HashMap CELT was the most space efficient and was able to handle up to 15 million insertions, a 50% increase over the STL HashMap CELT and a 36% increase over the Trove HashMap CELT.

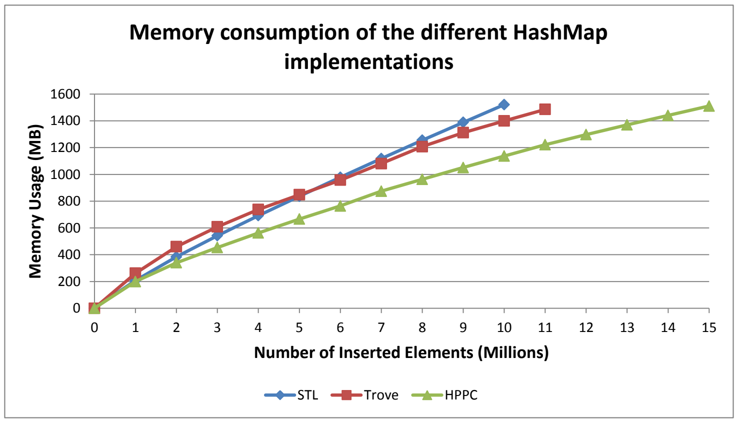


Figure 5.16: Memory consumption of the different HashMap implementations

5.5.2 Lookup Performance Comparison

Figure 5.17 compares the lookup times of the three hash map CELT implementations. The STL HashMap CELT was slowest. The Trove HashMap CELT was the fastest, approximately 24% faster than the STL HashMap CELT and 4% lower than the HPPC HashMap CELT.

Since the STL HashMap CELT was not competitive in terms of size, insertion time as lookup time, our selection was narrowed down to the HPPC HashMap CELT, which used less space, or the Trove HashMap CELT, which was slightly faster for lookup operations. In our QuPARA implementation, the smaller CELT data structure allows more XELTs to fit in the memory of a single mapper, which allow us to reduce the number of bathed jobs accordingly. Since the HPPC implementation uses 36% less space than the Trove implementation, we can reduce the number of batches by 36%. Since we expect this to have a greater system performance impact than improving raw lookup times by 4%, we choose the HPPC HashMap CELT as our final HashMap CELT implementation.

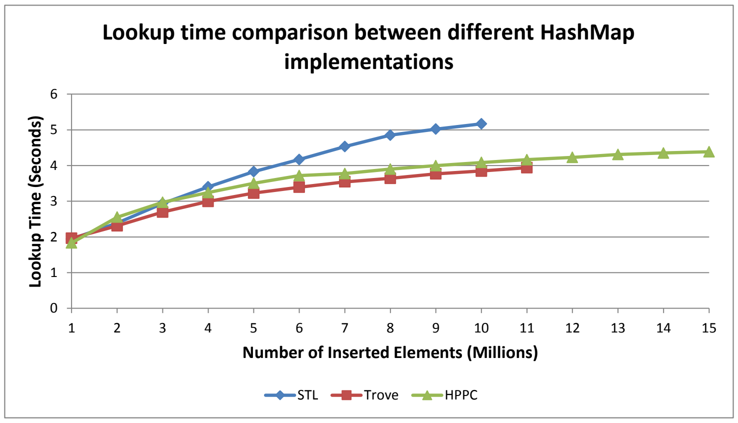
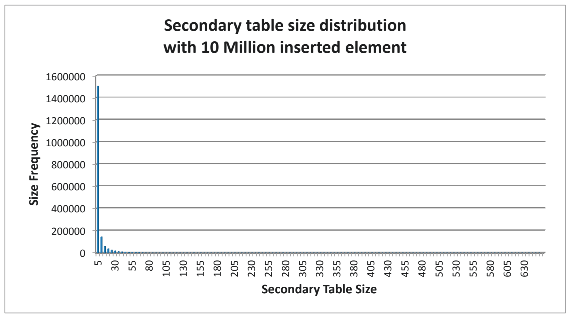


Figure 5.17: Lookup time comparison between different HashMap implementations

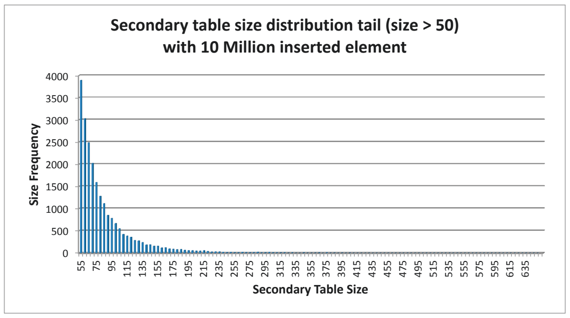
5.6 Hybrid CELT Implementation

CELT is a representation of an extremely sparse 2-dimensional lookup table, the average size of the secondary tables is very small. Figure 5.18a shows the size distribution of secondary tables in a CELT with 10 million elements. The bucket size used in the graph is 5. Figure 5.18b shows the tail of the distribution. As we can see, the distribution has a very long tail, and the larger secondary tables occur more infrequently. According to the distribution, 82.1% secondary tables contain equal or less than 5 elements.

Since more than 80% of the secondary tables in a CELT with 10 million elements contain no more than 5 elements, a has map may be unnecessarily complicated as an implementation of the secondary tables. In particular, a simple array promise to be more space efficient and offer the same on better lookup performance, even using linear search. Since lookups are performed only after all insertions are finished, we can do event better by first inserting the elements in constant amortized time per element need, then sorting the secondary tables once all insertions are done. This allows us to perform binary search during lookup operations. We call this CELT



(a) Secondary table size full distribution



(b) Secondary table size distribution tail

Figure 5.18: Relationship between lookup time and percentage of hits in the primary data structure

implementation that uses an HPPC IntObjectOpenHashMap for its primary table and sorted ArrayLists as secondary tables a hybrid CELT implementation. Our goal in this section is to optimize this implementation as much as possible and compute its performance to that of the HPPC HashMap CELT.

5.6.1 Searching Algorithm Selection

In the same way that a hash table, with its constant lookup cost, is theoretically faster than linear or binary search, binary search is faster than linear search in theory. However, it is also more complicated. Thus, it makes sense to determine the minimum input size necessary for binary search to outperform linear search in practice. To do so, we performed a series of experiments with different list sizes. For each experiment, we first inserted a number of XELT loss entries into an ArrayList and then sorted the entries in the list by their ELT identifier. Afterwards, we performed 10 million lookup queries using the two search algorithms and recorded the wall clock time spent on lookups.

Figure 5.19 shows the cost of 10 million lookup queries using linear search and binary search on between 1 and 130 elements. The figure shows that linear search

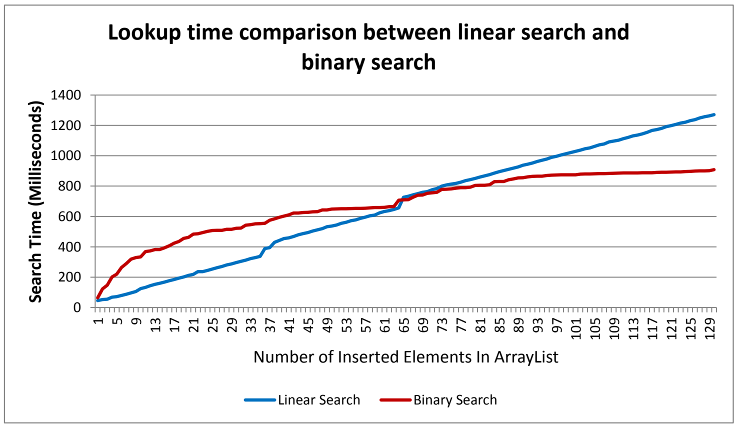


Figure 5.19: Lookup time comparison between linear search and binary search

performs better for less than 64 elements while binary search is more efficient for more than 64 elements.

We can take advantage of both linear search and binary search by building a hybrid search with a threshold value. If the number of elements in the array is less than the threshold, we use linear search. Otherwise, we apply binary search until the size of the sub array left to be searched drops below the threshold value, at which point we switch to linear search. Based on the previous experiments, we select 64 as the threshold value.

Figure 5.20 compares the lookup cost of the hybrid search algorithm to that of linear and binary search. As expected, the hybrid search algorithm matches the cost of linear search up to 64 elements; it may seem surprising that the algorithm performs even better than binary search for more than 64 elements. The reason is the switch to linear search once the number of remaining elements chops before 64, of which point linear search is more efficient than binary search. This also explains the drop in the search cost of the hybrid search when increasing the array size from 64 to 65 elements. For 64 elements, it performs a standard linear search. For 65 elements, it performs one binary search step to reduce the number of elements left to be searched to 32. It then

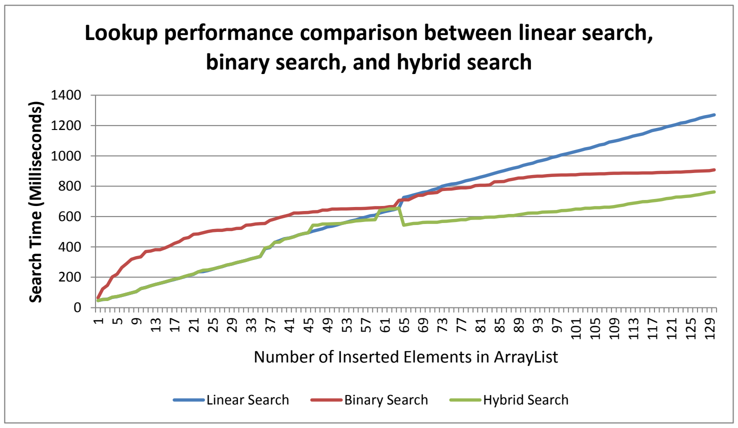


Figure 5.20: Lookup performance comparison between linear search, binary search, and hybrid search

perform a linear search of only 32 elements in contrast to the 64 elements searched when the input size is 64. In conclusion, we choose the hybrid search algorithm as the lookup algorithm for our hybrid CELT implementation.

5.6.2 JVM Parameter Tuning

In the same way as we tuned the JVM parameters for the different candidate HashMap CELT implementations, we discuss the tuning of these parameters for the hybrid CELT implementation in this section.

Initial Heap Size Tuning

Figure 5.21 shows the insert performance of the Hybrid CELT using different Xms values. A larger initial heap size led to better performance. The best Xms value for insert performance was 2 G, which is the same as the memory limit we set for the JVM.

Figure 5.22 shows the lookup costs of the Hybrid CELT using different Xms settings. A higher Xms value once again leaded to better performance.

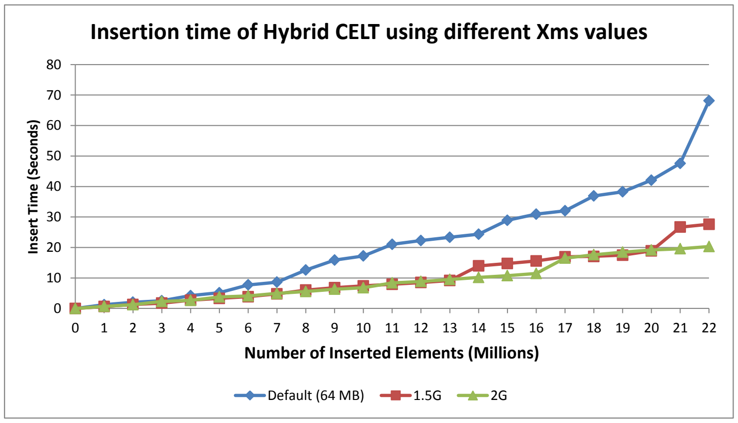


Figure 5.21: Insertion time of Hybrid CELT using different Xms values

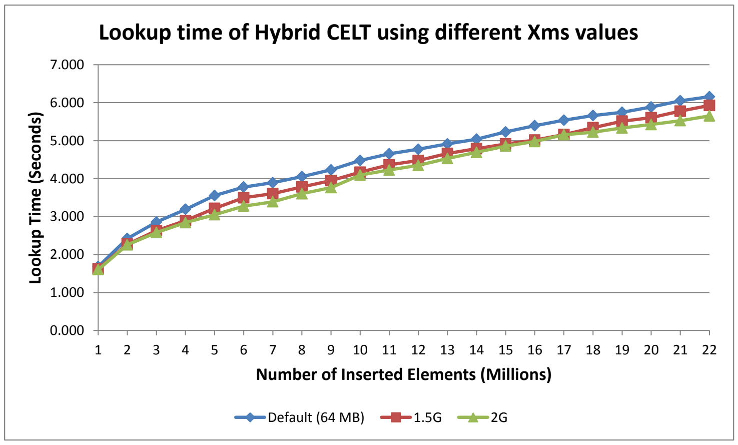


Figure 5.22: Lookup time of Hybrid CELT using different Xms values

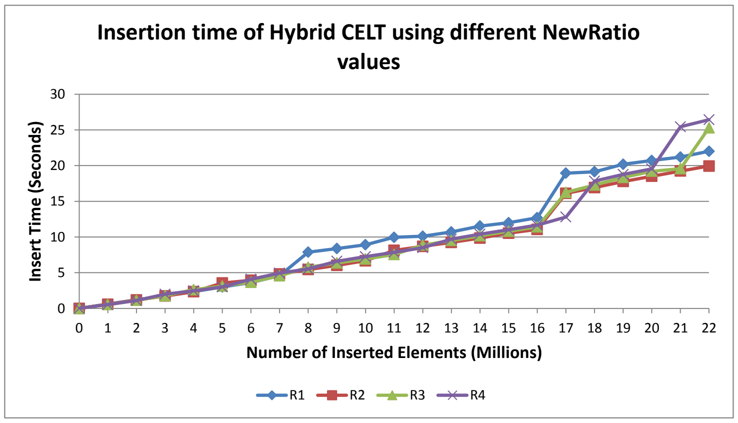


Figure 5.23: Insertion time of Hybrid CELT using different NewRatio values

Since both insert and lookup performance improve with large initial heap size, we decided to use the maximum Xms value of 2 GB for the Hybrid CELT.

Young Generation Ratio Tuning

In order to observe the effect of different young generation sizes on the Hybrid CELT performance, we performed insert and lookup experiments on the data structure with three different NewRatio values.

Figure 5.23 shows the insert performance of the Hybrid CELT using different NewRatio values. The data structure performed better with a NewRatio value of 4 up to 17 million insertions. Beyond 17 million insertions, a NewRatio value of 2 yield the best performance. This suggest that a NewRatio of 2 or 4 should be chosen for the Hybrid CELT.

Figure 5.24 shows the lookup performance of the Hybrid CELT in different NewRatio values. There was no significant performance difference between these different settings.

Thus, for the Xms configuration, we select the best value as 2 GB. For the NewRatio value, we select 4 as the best configuration for the Hybrid CELT.

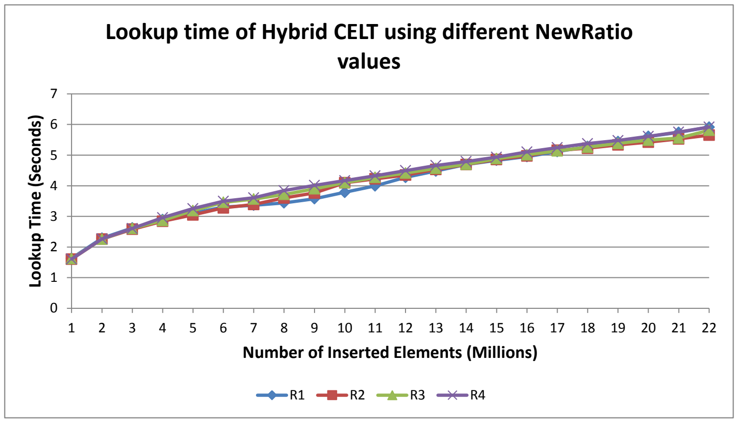


Figure 5.24: Lookup time of Hybrid CELT using different NewRatio values

5.7 Final CELT Implementation Selection

In this section, we compare the performance of the hybrid CELT implementations and the most efficient HashMap CELT implementation, the HPPC HashMap CELT, to decide whether to use the hybrid or HashMap implementation to maximize QuPARA's performance.

5.7.1 Insert Performance Comparison

Figure 5.25 shows the insert time performance of the HPPC HashMap CELT and the Hybrid CELT. Figure 5.26 shows the memory consumption. We observe that, the two implementations had almost equal insertion cost. However, the Hybrid CELT used much less space. This was expected since the memory overhead of an ArrayList is much lower than that of a HashMap. With the same amount of memory, the Hybird CELT can store up to 22 million elements, which is 46% more than the 15 million the HPPC HashMap CELT can hold.

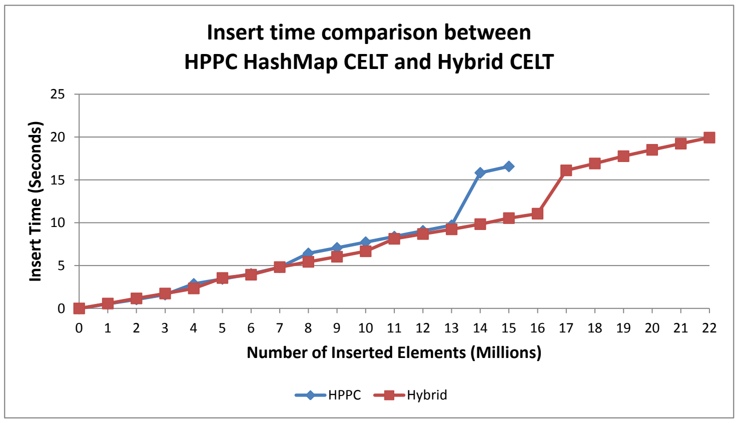


Figure 5.25: Insert time comparison between the HPPC HashMap CELT and the Hybrid CELT

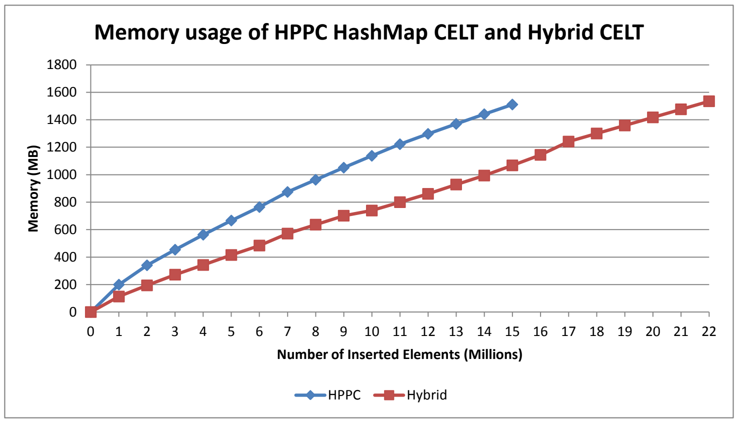


Figure 5.26: Memory usage of the HPPC HashMap CELT and the Hybrid CELT

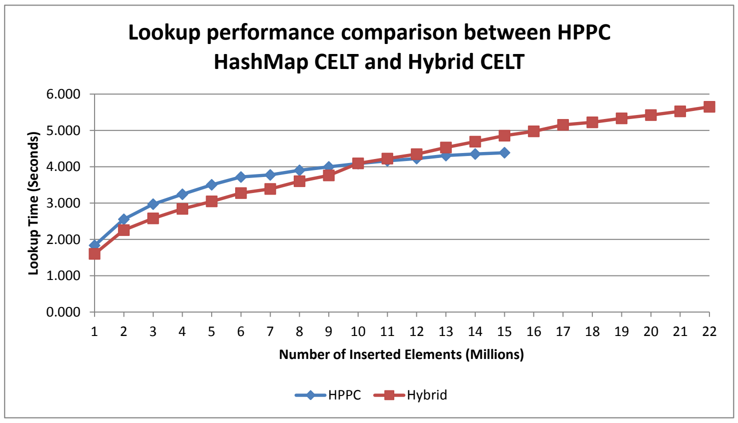


Figure 5.27: Lookup performance comparison between the HPPC HashMap CELT and the Hybrid CELT

5.7.2 Lookup Performance Comparison

Figure 5.27 compares the lookup performance of the HPPC HashMap CELT and the Hybrid CELT. Up to 14 million elements, the Hybrid CELT performed better. Beyond this point, the HPPC HashMap CELT showed better performance. This slight drop in lookup performance of the hybrid CELT is likely outweighed by the layer batch size the greater space efficiency of the Hybrid CELT allows.

In conclusion, the Hybrid CELT implementation is our final choice as it has smallest size, achieved the best insertion time, and its slightly higher lookup time compared to the HPPC HashMap CELT is outweighed by the reduction in size.

Chapter 6

System Evaluation

In this chapter, we discuss our experimental setup for evaluating the performance and functionality of QuPARA, and the results obtained. Throughout the experiments, we used our final aggregate analysis engine implementation which is based on the highest performing underlying data structures. We evaluate the performance of the system using the default portfolio aggregate risk analysis setup for one reinsurance portfolio comprising 1,600 layers. Each of the layers carries 5 unique XELTs. The Year Event Table has 1,000,000 trials, with each trial comprising 1000 Events. We evaluate the system by executing the standard queries to perform speed-up test, size-up test and scale-up test.

We evaluate QuPARA on the Hugh cluster, which belongs to the Faculty of Computer Science in Dalhousie University. The platform was a 19-node Rocks cluster [34], and each of which was built with 2.66Ghz Quad Core Intel Xeon Processor X3350, 4 GB DDR2 RAM, and three 1 TB 7,200 RPM STAT disk drivers. The nodes were connected via Gigabyte Ethernet. The nodes were running CentOS 6.3 with Java version 1.7.0_03.

In the cluster, we deployed the standard Cloudera BigData platform version 4.7.3 [38] by using one node as the master node and 18 nodes as the worker nodes. HDFS was built by assigning the master node as the major name node, and the worker nodes as the data nodes. The maximum capacity of HDFS was 20TB. The Hadoop version was 2.0.0-cdh4.5.0. In Hadoop, the jobtracker and job queue were running on the master node, and there were 18 worker nodes which gives totally 72 cores to run MapReduce jobs. The HIVE version was 0.10.0 [16]. The Pentaho version was 4.8-CE [32].

6.1 Experiment Data Description

In QuPARA, there are five types of input data sets, which are Year Event Table (YET), Portfolio Table (PFT), Extended Event Loss Table (XELT), Exposure Data Pool (EDP), and Event Catalogue Table (ECT). In the reinsurance industry, these data sets are properties of reinsurance companies and not transferable. Therefore, we were unable to work with real-world exposure data. Instead, we used synthetic data sets, which were carefully prepared with help from experts from the reinsurance industry, to match live industrial data.

YET: The YET, as described in Section 3.3.1, is a simulation data table describes a large number of trials, each representing one possible sequence of catastrophic events that might occur in a given year. The YET contains 1 million trials, each consisting of 1,000 catastrophic events.

PFT: The PFT, as described in Section 3.3.2, stores all the layers in a portfolio to be analyzed. In our experiments, the PFT contains 1,600 layers, and each covers 5 unique XELTs.

XELT: The XELT, as described in Section 3.3.5, is an exposure data tables which stores loss information for a certain type of catastrophic event in a specific geographical division. There are totally 8,000 XELT, and each XELT contains 1,000 events.

EDP: The EDP, as described in Section 3.3.4, contains exposure characteristics of the XELTs. This table contains 8,000 records in which each record describes a unique XELT.

ECT: The ECT, as described in Section 3.3.8, is the master catalogue of catastrophic events we used as the foundation in all the data generators. This table contains a detailed description of each such event that may occur. The ECT we used in our experiments contains 500 million unique event records in different regions and perils.

6.2 QuPARA System Performance

In this section, we evaluate the performance of QuPARA system by considering a series of experiments, which include speed-up test, size-up tests and scale-up tests. In the speed-up tests, we fixed the number of input size as 1,600 layers and increased the number of cores from 4 to 72 to check how much efficiency we can get from adding computational power in our system. In the size-up test, we fixed the number of cores as 72 and increased the size of input from 100 layers to 1,600 layers. The size-up test aimed to evaluate the system behavior differences between small inputs and large inputs. The scale-up tests fixed the number of layers per core and then increased the nodes to check the effects of adding computing resources into the QuPARA system.

6.2.1 Speed-up Performance

Figure 6.1 shows the decrease in running time for aggregate analysis using the optimized hybrid CELT described in Chapter 5 when the number of cores is increased but the input size is kept fixed at 1,600 layers. Figure 6.2 shows the relative speed-up achieved in this experiment, that is, the ratio between the running time achieved on 4 cores (a single node) and the running time achieved on up to 72 cores (18 nodes). The graph shows that we gained 88.38% speed-up while using 72 cores to process the analysis. Up to 24 cores (6 nodes), the speed-up is almost linear. Beyond 24 cores (6 nodes), the speed-up starts to decrease. As we observed, the decreasing trend was

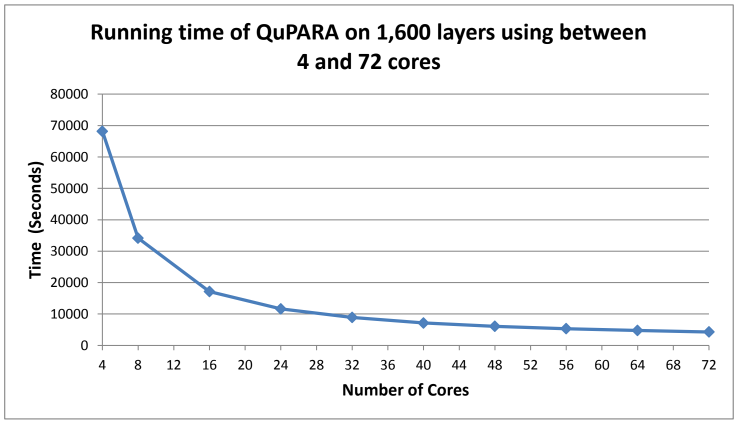


Figure 6.1: Running time of QuPARA on 1,600 layers using between 4 and 72 cores

caused by the increasing weight of the system overhead in the total running time.

In Chapter 5 we described an optimized hybrid CELT data structure. Figure 6.3 shows the effects of the hybrid CELT on the overall system performance. We observed there was a significant difference in the overall system running time between the original implementation and the improved optimization. With the hybrid CELT, we gained 34% performance improvement when 4 cores are used, and 31.7% performance improvement when 72 cores are used. The experiment results show the benefit of the hybrid CELT. Figure 6.4 shows the relative speed-up we achieved with different CELT implemented in QuPARA. On 72 cores, the speed-up of QuPARA with the original STL CELT, which we used as the baseline implementation, is 92.7%, which is better than 88.38% with the hybrid CELT. This is because the system overhead time is fixed, when the computation time decreases, the weight of the system overhead in the overall time increases, and then the speed-up performance decreases.

6.2.2 Size-up Performance

Figure 6.5 shows the increase in running time when the number of layers is increased from 100 to 1600 while keeping the number of cored fixed as 72. Once again, each

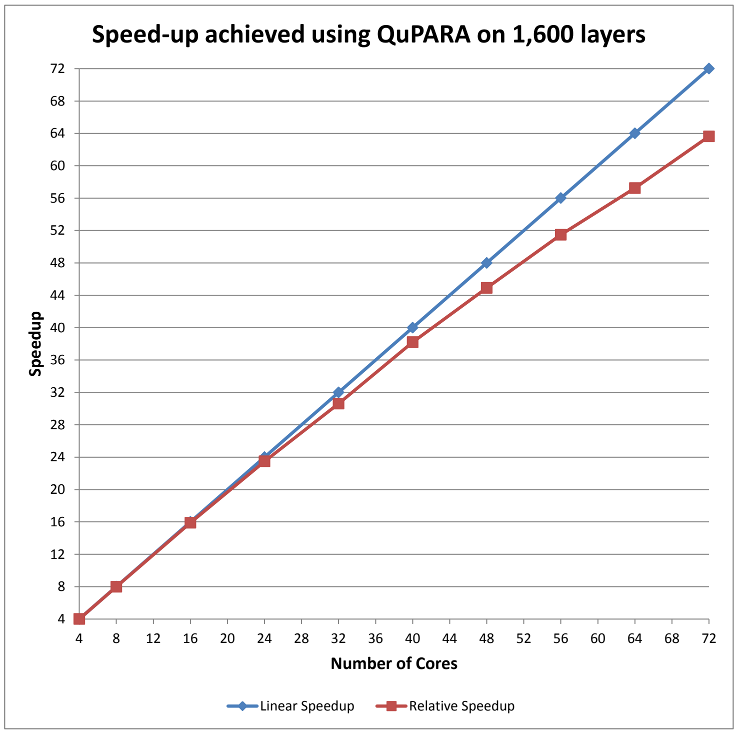


Figure 6.2: Speed-up achieved using QuPARA on 1,600 layers (corresponding to Figure 6.1)

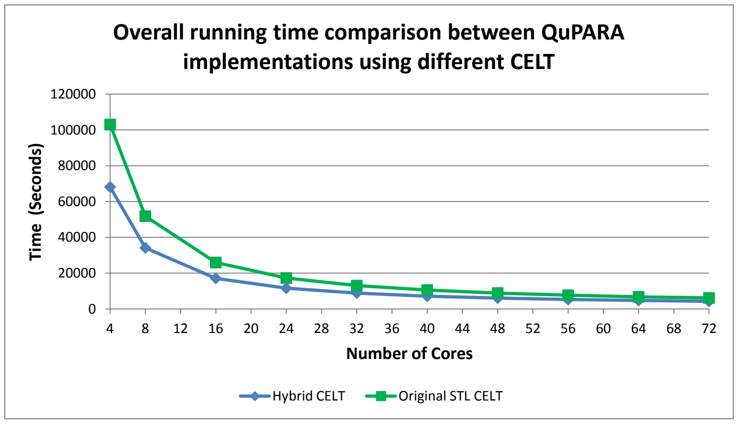


Figure 6.3: Overall running time comparison between QuPARA implementations using different CELT

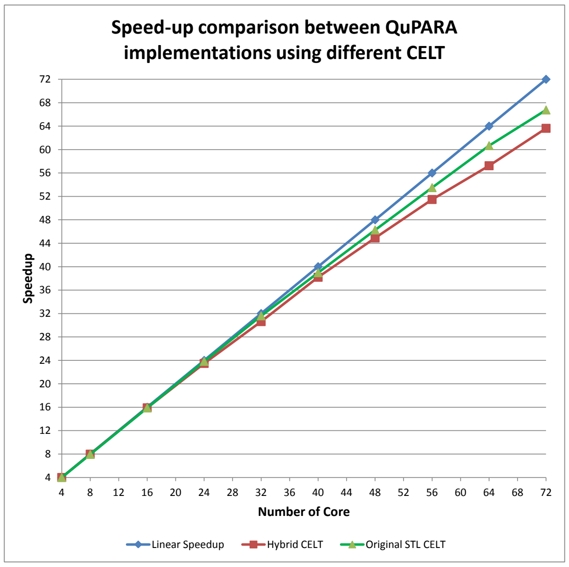


Figure 6.4: Speed-up comparison between QuPARA implementations using different CELT (corresponding to Figure 6.3)

layer covered 5 ELTs, the YET contains 1 million trials, each consisting of 1 thousand events. 72 cores, which are 18 nodes, were used in this experiment. As expected, the running time of QuPARA increases linearly with the input size as we increase the number of layers. With the increase in the number of layers, the time taken for setup, I/O time, and the time for all numerical computations scale in a linear fashion. The time taken for building data structures cleanup are a constant.

6.2.3 Scale-up Performance

Figure 6.6 shows the total time taken in seconds for performing aggregate risk analysis on our experimental platform. Up to 72 cores in 16 nodes were used in the experiment. Each four cores processed one job with 100 layers, each covering 5 unique XELTs. Thus, up to 8,000 XELTs were considered. The YET in our experiments contained 1 million trials, and each trial consists 1,000 events. The graph shows a very slow increase in the total running time, in spite of the constant amount of computation to be performed by each node (because every node processes the same number of

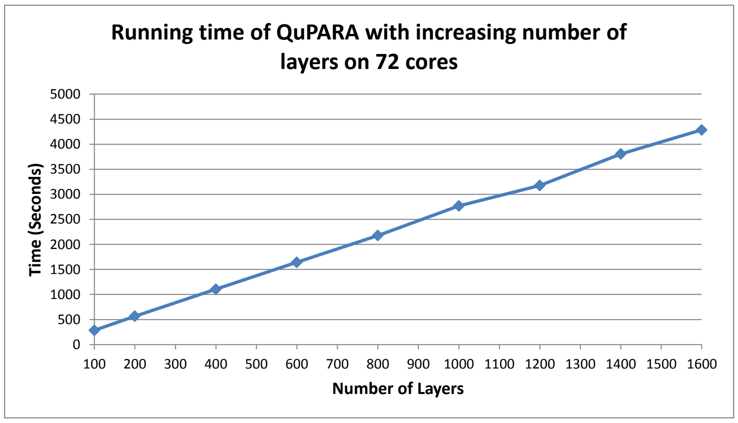


Figure 6.5: Running time of QuPARA with increasing number of layers on 72 cores

layers, XELTs, and YET entries). The gradual increase in the running time is due to the increase in the setup time required by the Hadoop job scheduler and the increase in network traffic (reflected in an increase in the time taken by the reducer). Nevertheless, this scheduling and network traffic overhead amounted to only 0.85% and 2.54% of the total computation time. Overall, this experiment demonstrates that, if the hardware scales with the input size, QuPARA's processing time of a query remains nearly constant.

6.3 Scenario Analysis Validation

An motivation of designing and implementing the QuPARA is to answer the user queries to solve various scenario risk analysis problems. Based on the scenarios described in Section 2.7, we list a set of possible scenario analyzes as following:

1. Portfolio aggregate risk analysis
2. Loss distribution of contracts in selected Line of Business (LOB)
3. Loss distribution of contracts in selected Class of Business (COB)
4. Loss distribution of contracts in selected Type of Participation (TOP)

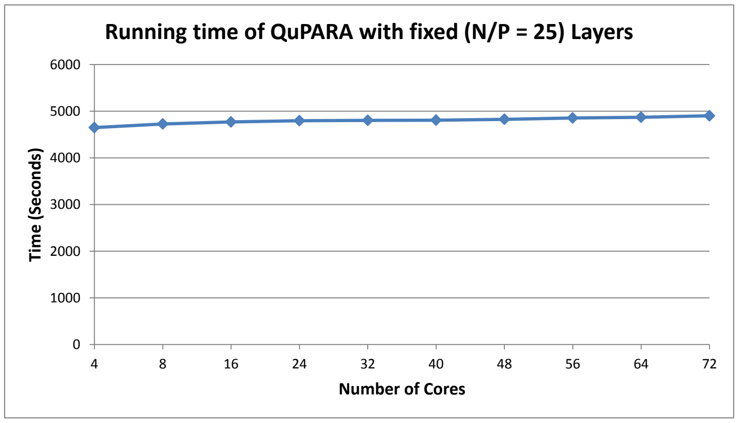


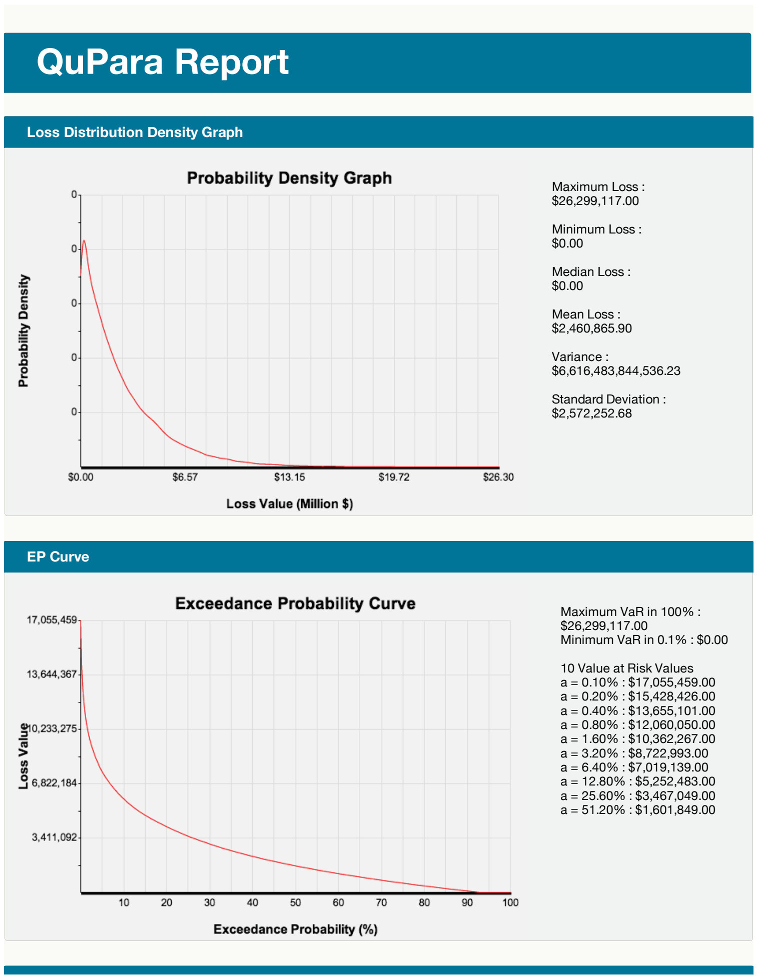
Figure 6.6: Running time of QuPARA with fixed (N/P = 25) Layers

5. Portfolio loss distribution in selected regions
6. Portfolio loss distribution in selected perils
7. Seasonal portfolio loss distribution
8. Multi-marginal analysis
9. Stochastic Exceedance Probability (STEP) analysis

In the rest of this section, we describe use case examples, which cover the analyzes we listed above, and present the results.

6.3.1 Portfolio Aggregate Risk Analysis

The basic use case for the QuPARA system is to perform the portfolio aggregate risk analysis by using the default data filtering and loss grouping queries as described in Section 3.5. In this experiment, the scenario analyses no.1 and no.9 are addressed. To express the result of this use case, we set the *result reporting query* to report the analysis results in different forms, which include the portfolio loss distribution, loss statistics, exceedance probability curve, and a series of PML values.



QuPara Report

Loss Distribution Density Graph

Maximum Loss :
$26,299,117.00

Minimum Loss :
$0.00

Median Loss :
$0.00

Mean Loss :
$2,460,865.90

Variance :
$6,616,483,844,536.23

Standard Deviation :
$2,572,252.68

EP Curve

Maximum VaR in 100% :
$26,299,117.00
Minimum VaR in 0.1% : $0.00

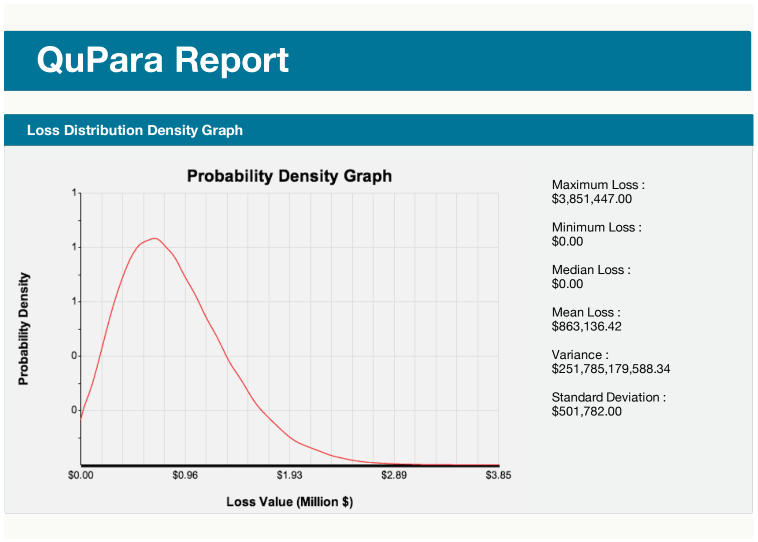
10 Value at Risk Values
a = 0.10% : $17,055,459.00
a = 0.20% : $15,428,426.00
a = 0.40% : $13,655,101.00
a = 0.80% : $12,060,050.00
a = 1.60% : $10,362,267.00
a = 3.20% : $8,722,993.00
a = 6.40% : $7,019,139.00
a = 12.80% : $5,252,483.00
a = 25.60% : $3,467,049.00
a = 51.20% : $1,601,849.00

Figure 6.7: Result page for portfolio aggregate risk analysis

Figure 6.7 shows the result reporting page in this use case. The result page suggests that the portfolio loss distribution has very long right tail, and the most probable loss, which is the peak of the probability density graph, is relatively small.

6.3.2 Specialized Layer Analysis

In QuPARA, the *layer filter* is used to extract a subset of layers from the portfolio to perform analysis. In this experiment, we address the analyses no.2, no.3 and no.4 by creating an analysis on layers which cover commercial buildings with excess of loss treaties. In order to perform this analysis, we need to modify the *data filtering query* Q_1, as described in Section 3.5.1, as following:



Maximum Loss :
$3,851,447.00

Minimum Loss :
$0.00

Median Loss :
$0.00

Mean Loss :
$863,136.42

Variance :
$251,785,179,588.34

Standard Deviation :
$501,782.00

Figure 6.8: Loss distribution of layers for commercial buildings with excess of loss treaties

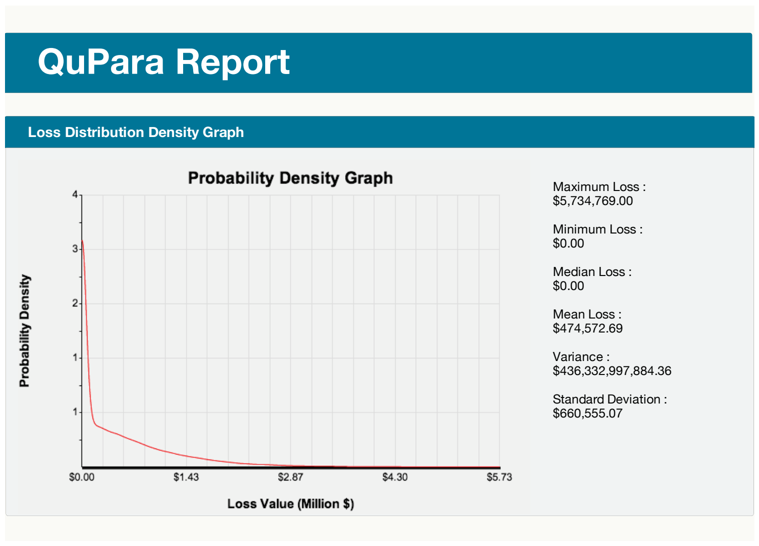
SELECT $*$ **FROM** PFT
    WHERE lob **IN** commercial
    AND WHERE cob **IN** building
    AND WHERE top **IN** exl

The 6.8 shows the result loss distribution of layers for commercial buildings with excess of loss treaties.

6.3.3 Specialized Exposure Analysis

In QuPARA, the *ELT filter* is used to extract and filter the exposure data used in the analysis. In this experiment, we address the analyses no.5 and no.6 by creating an analysis to show the portfolio loss distribution in earthquake in Canada. In order to perform this analysis, we need to modify the *data filtering query* Q_2, as described in Section 3.5.1, as following:

SELECT $*$ **FROM** EDP
    WHERE elt_ID **IN** {elt_IDs}
    AND WHERE peril **IN** {EQ}
    AND WHERE region **IN** {CA}



Maximum Loss : $5,734,769.00

Minimum Loss : $0.00

Median Loss : $0.00

Mean Loss : $474,572.69

Variance : $436,332,997,884.36

Standard Deviation : $660,555.07

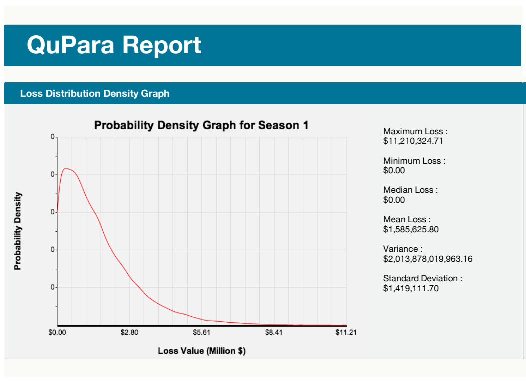
Figure 6.9: Portfolio loss distribution in earthquake in Canada

The 6.9 shows a portfolio loss distribution with losses from earthquake and region in Canada. In this analysis, we filtered out all the exposure data and events, which are not related with earthquake and not in Canada. Then, the QuPARA produces result shows a very sharp and long tailed loss distribution. In this distribution, the most probable loss is close to zero. This is reasonable since notable earthquake happens rarely in a year. However, an earthquake might cause massive damage, which destroy properties completely, when it occurs.

6.3.4 Seasonal Analysis

QuPARA can produce seasonal portfolio loss distribution instead of a single annual portfolio loss distribution. The seasonal distributions can help the analysts understand their portfolio performance in time periods. In this experiment, we address the analysis no.7 by specifying 4 seasons to the loss grouping query Q_5, as described in Section 3.5.2, to group event losses in a trial by time index. The query Q_5 is specified as following:

```
SELECT trial_ID, estimated_Loss
    FROM YRPLT
```



Maximum Loss :
$11,210,324.71

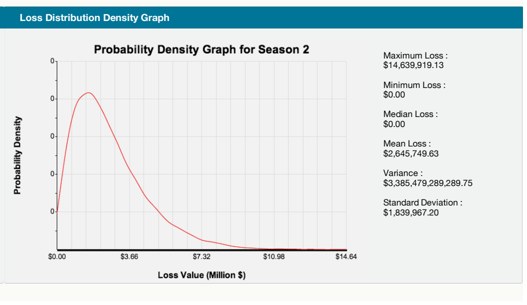
Minimum Loss :
$0.00

Median Loss :
$0.00

Mean Loss :
$1,585,625.80

Variance :
$2,013,878,019,963.16

Standard Deviation :
$1,419,111.70


Loss Distribution Density Graph
Probability Density Graph for Season 2
Probability Density
Loss Value (Million $)
$0.00
$3.66
$7.32
$10.98
$14.64

Maximum Loss :
$14,639,919.13

Minimum Loss :
$0.00

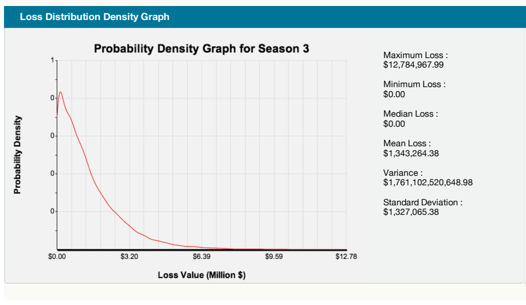
Median Loss :
$0.00

Mean Loss :
$2,645,749.63

Variance :
$3,385,479,289,289.75

Standard Deviation :
$1,839,967.20

(a) Portfolio Loss Distribution in Season 1 (b) Portfolio Loss Distribution in Season 2


Loss Distribution Density Graph
Probability Density Graph for Season 3
Probability Density
Loss Value (Million $)
$0.00
$3.20
$6.39
$9.59
$12.78

Maximum Loss :
$12,784,967.99

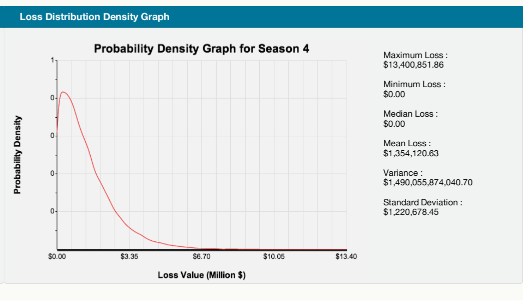
Minimum Loss :
$0.00

Median Loss :
$0.00

Mean Loss :
$1,343,264.38

Variance :
$1,761,102,520,648.98

Standard Deviation :
$1,327,065.38


Loss Distribution Density Graph
Probability Density Graph for Season 4
Probability Density
Loss Value (Million $)
$0.00
$3.35
$6.70
$10.05
$13.40

Maximum Loss :
$13,400,851.86

Minimum Loss :
$0.00

Median Loss :
$0.00

Mean Loss :
$1,354,120.63

Variance :
$1,490,055,874,040.70

Standard Deviation :
$1,220,678.45

(c) Portfolio Loss Distribution in Season 3 (d) Portfolio Loss Distribution in Season 4

Figure 6.10: Portfolio loss distribution in four seasons

GROUP BY SEASON(4)

Figure 6.10 shows the results of the seasonal analysis which contains one loss distribution and its basic statistics in each season. On the graph, the loss distribution in season 2 is flatter than others, and the expected loss in this season is much higher than in other seasons. This is due to the major catastrophe events, such as hurricanes and floods, usually occurs on the second season in a year. The loss distributions of season no. 1 and season no. 4 are similar. This is reasonable since the catastrophe events, such as storms, may occur in these two seasons are similar in this two seasons.

6.3.5 Multi-marginal Analysis

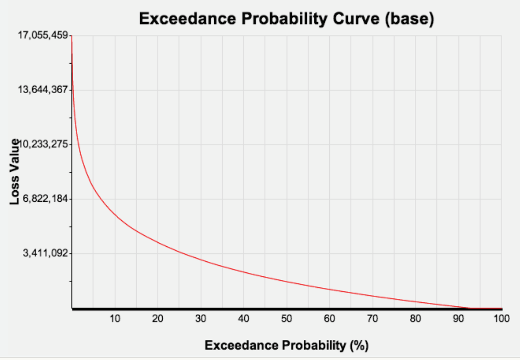
Multi-marginal analysis, analysis no.8, is used to observe the effects of adding more layers into a portfolio for layer pricing purposes. In this experiment, we add 4 layers into a portfolio and try to check the influence of adding all 4 layers in the portfolio EP curve and PML values. To perform this analysis, rather than specifying the 4 additional layers, the loss grouping query Q_5, as described in Section 3.5.2, is modified as following:

```
SELECT trial_ID, estimated_Loss
    FROM YRPLT
        GROUP BY TRIAL
        AND GROUP BY Margin(4)
```

Figure 6.11 shows the results of the multi-marginal analysis we executed. The result suggests that, after adding the additional 4 layers, the shape of EP curve does not change. However, some of the PMLs values in the new portfolio is slightly increased at risk level 1.6% and 6.4%.

QuPara Report

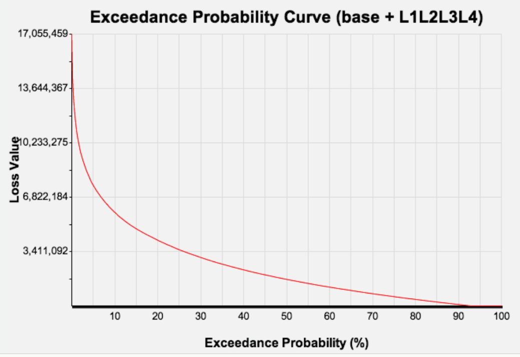
EP Curve



Maximum VaR in 100% :
$26,299,117.00
Minimum VaR in 0.1% : $0.00

10 Value at Risk Values
a = 0.10% : $17,055,459.00
a = 0.20% : $15,428,426.00
a = 0.40% : $13,655,101.00
a = 0.80% : $12,060,050.00
a = 1.60% : $10,362,267.00
a = 3.20% : $8,722,993.00
a = 6.40% : $7,019,139.00
a = 12.80% : $5,252,483.00
a = 25.60% : $3,467,049.00
a = 51.20% : $1,601,849.00

EP Curve



Maximum VaR in 100% :
$26,299,117.00
Minimum VaR in 0.1% : $0.00

10 Value at Risk Values
a = 0.10% : $17,055,459.00
a = 0.20% : $15,428,426.00
a = 0.40% : $13,655,101.00
a = 0.80% : $12,060,050.00
a = 1.60% : $10,393,353.80
a = 3.20% : $8,722,993.00
a = 6.40% : $7,026,158.14
a = 12.80% : $5,252,483.00
a = 25.60% : $3,467,049.00
a = 51.20% : $1,601,849.00

Figure 6.11: Multi-marginal analysis EP curve results

Chapter 7

Conclusion and Future Work

In this final chapter, we first present the summary and conclusion of this thesis, and then discuss possible directions towards future work.

7.1 Summary and Conclusion

QuPARA is an extensible framework to facilitate ad hoc analysis of catastrophic risk-based portfolios. Such an extensible framework can be used for performing analysis of portfolios by taking into account the finer level of detail which is not supported by production-based risk management systems. The proposed framework considers the aggregate risk analysis algorithm and supports the layering of in-depth analysis on top of the basic algorithm that can capture finer level of detail of different loss aggregation levels.

The prototype implementation of QuPARA uses the Apache Hadoop implementation of the MapReduce programming model, Apache Hive for expressing ad hoc queries in an SQL-like language, and Pentaho for building user-friendly interface.

Within the Hadoop implementation of MapReduce, careful engineering of the underlying data structures can lead to further significant performance improvements of QuPARA, thereby further reducing the performance gap to hand-crafted risk analytic systems. The performance-critical data structure in QuPARA is the CELT. We obtained our final implementation of this data structure using systematic in-depth experiments and tuning, leading to a 31.7% reduction in running time.

The performance results for the standard portfolio analysis show that within the

Hadoop implementation, the QuPARA system can achieve high efficiency on distributed parallel environment. The capability and the scalability of the QuPARA system are also proved in the experiments. Moreover, the scenario analysis experiments demonstrate the feasibility of answering ad hoc queries on industry-size data sets efficiently using QuPARA. In this prototype system, a full portfolio risk analysis run consisting of a 1 million trial simulation, with 1,000 event per trial, and 1,600 risk transfer contracts can be completed on a 19-node Hadoop cluster in 80 minutes. This performance is competitive with highly tuned production systems that are only capable of answering a narrow set of portfolio queries, in contrast to the wide range of ad hoc queries QuPARA is able to resolving [39].

7.2 Recommendation for Future Work

While the QuPARA showed good performance and ability on reinsurance portfolio aggregate risk analysis under various scenarios, there are several areas where the design, implementation, and experiment could be further enhanced. In this section, we discuss possible directions for future work:

Extensible Aggregate Procedure

The proposed framework provides great flexibility on data customization through the data filters. However, the aggregation procedure modifications are limited in switching between a limited set of aggregation functions and parameters. A possible improvement on the design of QuPARA framework is to allow the users to directly define aggregation algorithms using high level programming languages, such as R, to modify the analysis.

Evaluation on Live Industrial Data

In the experiments presented in this thesis, we used synthetic data which generated and collected from data generators with scientific data via simulation approaches. The synthetic data is close to but not perfectly match the industrial data which reinsurance companies use. In order to further understand the system performance and the result quality of QuPARA, we need to apply QuPARA on real industrial data and compare the results with production systems reinsurance companies use.

Location Based Aggregate Analysis

QuPARA uses event based aggregate data, XELT, to address losses in layers in aggregate risk analysis. In order to perform more detailed analysis, location based exposure data should be used. Comparing to the per-event aggregate losses in XELT, the location based exposure data contains per-event-region loss data, which provides detailed losses in each region effected by an event. In the future, we may extend the filtering and aggregation abilities of QuPARA to make use of the location based exposure data to offer more analytical options in various detail levels to the users.

Customized Treaty Classification

In QuPARA, the *layer filter* allows users to select a subset of layers based on the layer attributes, such as Line of Business (LOB), Class of Business (COB), and Type of Participation (TOP), to perform grouping and aggregation in the analysis. A further improvement on this process is to allow users to define grouping rules, instead of using existing attributes, to classify layers to perform analysis.

Bibliography

[1] H. Castella, G. de Montmollin, and E. Rüttener, *Catastrophe portfolio modeling: a complete view*, S. Thomas and PartnerRe, Eds. Pembroke: PartnerRe, 2009.

[2] R. Anderson and W. Dong, “Pricing catastrophe reinsurance with reinstatement provisions using a catastrophe model,” *Casualty Actuarial Society Forum*, pp. 303–322, 1988.

[3] G. G. Meyers, F. L. Klinker, and D. A. Lalonde, “The aggregation and correlation of insurance exposure,” *Casualty Actuarial Society Forum*, pp. 60–152, 2003.

[4] W. Dong, H. Shah, and F. Wong, “A rational approach to pricing of catastrophe insurance,” *Journal of Risk and Uncertainty*, vol. 12, no. 2-3, pp. 201–218, May 1996.

[5] R. M. Berens, “Reinsurance contracts with a multi-year agguegate limit,” *Casualty Actuarial Society Forum*, pp. 289–308, 1997.

[6] E. d. Alba, J. Ziga, and M. A. R. Corzo, “Measurement and transfer of catastrophic risks,” *ASTIN Bulletin*, vol. 40, no. 2, pp. 547–568, 2010.

[7] G. Woo, “Natural catastrophe probable maximum loss,” *British Actuarial Journal*, vol. 8, no. 05, pp. 943–959, 2002.

[8] M. E. Wilkinson, “Estimating probable maximum statistics loss with order statistics,” *Casualty Actuarial Society Forum*, pp. 195–209, 1982.

[9] A. A. Gaivoronski and G. Pflug, “Value-at-risk in portfolio optimization: properties and computational approach,” *Journal of Risk*, vol. 7, no. 02, pp. 1–31, 2005.

[10] P. Glasserman, P. Heidelberger, and P. Shahabuddin, “Portfolio value-at-risk with heavy-tailed risk factors,” *Mathematical Finance*, vol. 12, no. 3, pp. 239–269, 2002.

[11] T. White, *Hadoop: the definitive guide*, 1st ed. O’Reilly Media, Inc., 2009.

[12] The Apache Software Foundation. (2012) Apache hadoop. [Online]. Available: http://hadoop.apache.org/

[13] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[14] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, p. 21.

[15] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with mapreduce: a survey," *Special Interest Group on the Management of Data Rec.*, vol. 40, no. 4, pp. 11–20, Jan. 2012.

[16] The Apache Software Foundation. (2014) Apache hive. [Online]. Available: http://hive.apache.org/

[17] E. Capriolo, D. Wampler, and J. Rutherglen, *Programming Hive*. O'Reilly Media, 2012.

[18] Swiss Re., "An introduction to reinsurance," 2002. [Online]. Available: http://fa2f2.voila.net/intro_reinsurance.pdf

[19] InvestorGuide Staff. (2013) Introduction to insurance policies, premiums and claims. [Online]. Available: http://www.investorguide.com/article/11605/introduction-to-insurance-policies-premiums-and-claims-igu/

[20] D. Holland, "A brief history of reinsurance," *ACLI Reinsurance Executive Roundtable: Munich Re.*, no. 6, 2008.

[21] Institue of Insurance Sciences, *An introduction to reinsurance*. Madrid: Fundacion Mapfre, 2013.

[22] P. Baur and A. Breutel-O'Donoyhue, "Understanding reinsurance," *Swiss Re Technical Publishing*, pp. 1–131, 2004.

[23] A. J. Mata, "Pricing excess of loss reinsurance with reinstatements," *Astin Bulletin*, vol. 30, no. 2, pp. 349–368, 2000.

[24] Risk Management Solutions. (2014) Rms-risk management solutions. [Online]. Available: http://www.rms.com/

[25] P. Grossi, H. Kunreuther, and C. C. Patel, *Catastrophe modeling: a new approach to managing risk*, ser. Huebner International Series on Risk, Insurance and Economic Security. Springer, 2005.

[26] P. Grossi, H. Kunreuther, and C. Patel, *Catastrophe modeling: a new approach to managing risk*, ser. Catastrophe Modeling. Springer, 2005.

[27] Bermuda Monetary Authority, "Catastrophe risk return guidelines," Bermuda Monetary Authority, Tech. Rep., 2011.

[28] H. Karloff, S. Suri, and S. Vassilvitskii, “A model of computation for mapreduce,” in *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’10. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2010, pp. 938–948.

[29] R. Choy, A. Edleman, J. R. Gilbert, V. Shah, and D. Cheng, “Star-p: High productivity parallel computing,” DTIC Document, Tech. Rep., 2004.

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 2010, pp. 1–10.

[31] The Apache Software Foundation. (2007) The hadoop distributed file system: architecture and design. [Online]. Available: http://hadoop.apache.org/docs/r0.18.3/hdfs_design.html

[32] Pentaho Community. (2014) Pentaho. [Online]. Available: http://community.pentaho.com/

[33] A. Berson and S. J. Smith, *Data Warehousing, Data Mining, and Olap*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[34] R. C. Distribution. (2014) About rocks cluster. [Online]. Available: http://www.rocksclusters.org/

[35] T. C. Project. (2014) About centos. [Online]. Available: https://www.centos.org/about/

[36] Oracle. (2014) Java sdks and tools. [Online]. Available: http://www.oracle.com/technetwork/java/javase/downloads/index.html

[37] C. Lomnitz, “Poisson processes in earthquake studies,” *Bulletin of the Seismological Society of America*, vol. 63, no. 2, pp. 735–735, 1973.

[38] Cloudera. (2014) Cloudera, ask bigger questions. [Online]. Available: http://www.cloudera.com/

[39] Flagstone Re, “Production risk analytics system performance,” 2014, private communication.