Efficient Data Structures for Risk Modelling in Portfolios of Catastrophic Risk Using MapReduce

Andrew Rau-Chaplin, Zhimin Yao and Norbert Zeh

Faculty of Computer Science, Dalhousie University, Halifax, Nova Scotia, Canada
arc@cs.dal.ca, zhimin.yao@dal.ca, nzeh@cs.dal.ca

Abstract

The QuPARA Risk Analysis Framework [18] is an analytical framework implemented using MapReduce and designed to answer a wide variety of complex risk analysis queries on massive portfolios of catastrophic risk contracts. In this paper, we present data structure improvements that greatly accelerate QuPARA's computation of Exceedance Probability (EP) curves with secondary uncertainty.

1 Introduction

The financial management of the risk associated with catastrophic events such as earthquakes, hurricanes, and large-scale floods falls largely to insurance and reinsurance companies [6]. 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 with respect to a range of risk metrics that take the uncertainty associated with both event order and magnitude into account [1]. The QuPARA Risk Analysis Framework, which was introduced in [18], is an analytical environment implemented using MapReduce [8, 13] and designed to answer a wide variety of complex risk analysis queries on portfolios of catastrophic risk. Given a reinsurance company's portfolio, QuPARA's core analytical function is to compute Exceedance Probability (EP) curves [23, 24], which represent, for each of a set of user-specified loss values, the probability that the total claims a reinsurer will have to pay out exceeds this value. Not surprisingly, there is no computationally feasible closed-form expression for computing such an EP curve over hundreds of thousands of events and millions of insured properties. Consequently, a computationally very intensive simulation approach must be taken that evaluates each portfolio in up to a million simulation trials and then aggregates the expected losses calculated for each trial to obtain a loss distribution. If QuPARA is to be useful in practice, it must be able to quantify portfolio-level risk on large portfolios efficiently.

In this paper, we present data structure enhancements that greatly accelerate QuPARA's computation of EP curves with secondary uncertainty over large risk portfolios [19]. Our approach is to combine data structure design with systematic in-depth experimental evaluation and tuning. We have succeeded in reducing the memorty useage of the core lookup data structure (CELT) by over 50%, which allows us to double the batch size per QuPARA run within a Hadoop/MapReduce environment. The construction time of the data structure has been improved by 40%, and the total lookup time was decresed by 42%. Since there are certian Hadoop system overheads and mathmatical calculations involved in the analysis process, we achieve an overall performance improvement of 31.7%.

2 Portfolio-Level Catastrophic Risk Modelling

Portfolio-level catastrophic risk modelling [2, 10, 12, 15, 24] is the central analytical concern of modern reinsurance companies. A reinsurance company typically holds a *portfolio* of catastrophic risk programs that insure primary insurance companies against large-scale losses, like those associated with earthquakes, hurricanes, and large-scale floods [6]. Each *program* contains data that describes (1) the buildings to be insured (the *exposure*), (2) the modelled risk to that exposure (the *event loss tables*), and (3) a set of risk transfer contracts (the *layers*) [3, 18].

The *exposure* is represented by a table, one row per covered building, 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) [2]. 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. See [2, 4, 10, 12] for details.

As an example, the exposure of a Florida hurricane program might list 2 million buildings including their locations, construction details, primary insurance terms, and replacement values. The ELT might, for each of 100,000 possible hurricane events in Florida, give the sum of the losses expected to the associated exposure should the event occur. ELTs are the output of stochastic region peril models [12] and typically also include some additional financial terms. Finally, the risk transfer contract might consist of multiple layers (or subcontracts). The first layer might be 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 might be an aggregate layer covering 30% of losses between 40 million and 90 million that accumulate due to hurricane activity over the course of a year.

Given a reinsurance company's portfolio described in terms of exposure, event loss tables, and layers, the most fundamental type of analysis query computes an Exceedance Probability (EP) curve, which represents, for each of a set of user-specified loss values, the probability that the total claims a reinsurer will have to pay out exceeds this value [5]. Not surprisingly, there is no computationally feasible closed-form expression for computing such an EP curve over hundreds of thousands of events and millions of individual exposures. Consequently, a simulation approach must be taken. The idea is to perform a stochastic simulation based on a *year event table* (YET). This table describes a large number of trials, each representing one possible sequence of catastrophic events that might occur in a given year. This YET is generated by an event simulator that uses the expected occurrence rate of each event plus other hazard information like seasonality.

In this paper, we focus on the computationally intensive task of computing the expected loss distribution (i.e., EP curve) for a given portfolio, given a particular YET [14]. The loss distribution is computed from the portfolio and the YET in two phases. The first phase computes a *year loss table* (YLT). For each trial in the YET, each event in this trial, and each ELT that includes this event, the YLT contains a tuple $\langle$trial, event, ELT, loss$\rangle$ recording the loss incurred by this event, given the layer's financial terms and the sequence of events in the trial up to the current event. The second phase then aggregates the entries in the YLT to compute the final loss distribution.

Note that the loss value for each event is not a simple mean value because there are a multitude of possible loss outcomes for any given event, modelled as a probability distribution of loss values associated with the event rather than a single value. This probability distribution captures that we are unsure about certain exposure and hazard parameters and their interactions. We refer to this as *secondary uncertainty* [14], in contrast to the *primary uncertainty* whether an

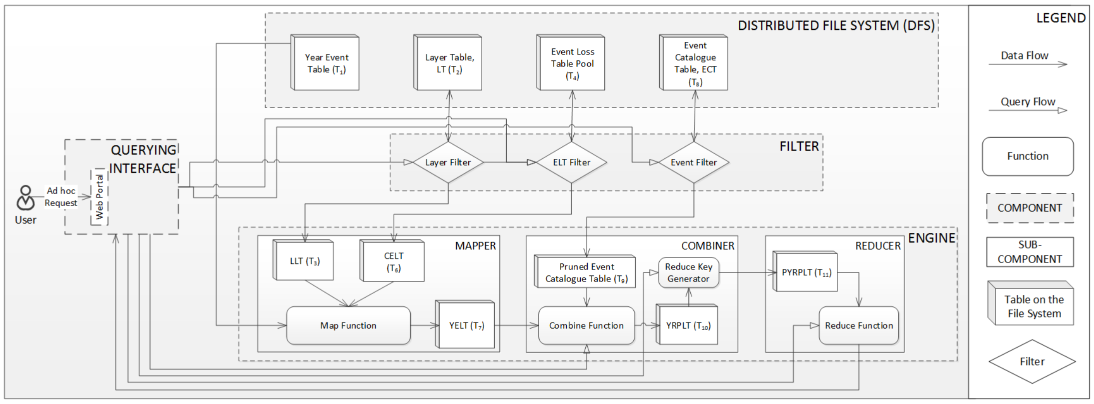

Figure 1: The Query-Driven Portfolio Aggregate Risk Analysis (QuPARA) Framework

event occurs, which is modelled by the YET.

3 The QuPARA Risk Analysis Framework

The QuPARA Risk Analysis Framework, which was introduced in [18], is an analytical environment designed to answer a wide variety of complex risk analysis queries on portfolios of catastrophic risk. In this section, we provide a brief overview of its basic architecture (see Figure 1) so we can explore efficient data structures to enhance its performance in Section 4.

In order to answer ad hoc portfolio risk queries in a timely manner, the QuPARA framework supports a parallel implementation of such queries using the MapReduce programming model. The computation of the YLT is the responsibility of the mapper, while the computation of the final loss distribution(s) is done by the reducer [18, 19].

QuPARA employs the MapReduce framework [8, 13, 21, 22] to evaluate each query using a single round consisting of a map/combine step and a reduce step. During the map step, the engine uses one mapper per trial in the YET, in order to construct a YLT from the YET. The combiner and reducer collaborate to aggregate the loss information in the YLT into the final loss distribution. 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) during the shuffle step. The reducer(s) then carry out the final aggregation.

Each mapper retrieves the set of ELTs required for the query from the distributed file system using a layer filter and an ELT filter. Specifically, the query may specify a subset of the layers in the portfolio to be the subject of the analysis. The layer filter retrieves the identifiers of the ELTs contained in these layers from the layer table. If the query specifies, for example, that the analysis should be restricted to a particular type of peril, the ELT filter then extracts from this set of ELTs the subset of ELTs corresponding to the specified type of peril. Given this set of ELT identifiers, the mapper retrieves the actual ELTs and, in memory, constructs a *combined ELT* or *CELT* associating a loss with each ⟨event, ELT⟩ pair. It then iterates over the sequence of events in its trial, looks up the ELTs recording non-zero losses for each event, and generates the corresponding ⟨trial, event, ELT, loss⟩ tuple in the YLT, taking each ELT's financial terms into account. The aggregation to be done by the combiner depends on the risk query. A reducer, finally, receives one loss value per trial. It sorts these loss values in increasing order and uses this sorted list to generate a final loss distribution.

The following is a more detailed description of the mapper, combiner, and reducer used in QuPARA with a particular focus on the Mapper and the use of the *CELT* data structure, which is the critical component in terms of analysis efficiency.

Mapper: The mapper, shown in Algorithm 1, takes as input an entire trial and the list of its events, represented as a pair $\langle T, E := \{E_1, E_2, \ldots, E_m\}\rangle$. Algorithm 1 does not show the construction of the CELT and layer list (LLT) performed by the mapper before carrying out the steps in lines 1–9. The loss estimate of an event in a portfolio is computed by scanning through every layer L in the LLT, retrieving and summing the loss estimates for all ELTs covered by this layer, and finally applying the layer's financial terms.

Combiner: The combiner receives as input the list of triples $\langle T,\ E_i,\ l_{PF}\rangle$ generated by a single mapper, that is, the list of 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.

Reducer: The reducer 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. For example to generate an exceedance probability curve, the reducer sorts the received loss values in increasing order and, for each loss value v in a user-specified set of loss values, reports the percentage of trials with a loss value greater than v as the probability of incurring a loss higher than v.

Algorithm 1: Mapper in parallel aggregate risk analysis

```
Input: $\langle T, E := \{E_1, E_2, \cdots, E_m\}\rangle$ where $m$ is the number of events in a trial, a set of
       event loss tables ELT$_1$, ELT$_2$, …, ELT$_n$, and layer list (LLT).
Output: A list of entries $\langle T, E_i, l_{PF}\rangle$ of the YLT
1  Construct from event loss tables ELT$_1$, ELT$_2$, …, ELT$_n$ an in-memory combined event loss
   set data structure (CELT) as illustrated in Figure 2.
2  for each event, $E_i$ in $E$ do
3      Look up $E_i$ in the CELT and find corresponding losses, $l_{E_i} = \{l^1_{E_i}, l^2_{E_i}, \cdots, l^n_{E_i}\}$, where
       ELT$_1$, ELT$_2$, …, ELT$_n$ are the ELTs in the CELT
4      for each layer, $L$, in the LLT do
5          for each ELT ELT$_j$ covered by $L$ do
6              Lookup $l^j_{E_i}$ in $l_{E_i}$
7              $l_L \leftarrow l_L + l^j_{E_i}$
8          Apply $L$'s financial terms to $l_L$
9          $l_{PF} \leftarrow l_{PF} + l_L$
10     Emit($\langle T,\ E_i,\ l_{PF}\rangle$)
```

In practice, the construction of the CELT at line 1 of Algorithm 1 and lookups in this massive random access data structure at line 3 of Algorithm 1 dominate the running time of this algorithm. Another major bottleneck is the size of the CELT, which exceeds the RAM size on our hardware. This forces us to split the computation of the YLT into batches. Each batch computes the YLT for a subset of layers and needs to be implemented as a separate MapReduce job. Once all batches have been processed, the final job carries out the work of the reduces in the above description. Thus, major improvements of the efficiency of QuPARA can be achieved by reducing the construction and lookup costs of the CELT and, since starting a MapReduce job incurs a non-trivial overhead, by increasing the number of layers that can be stored by the CELT in the same amount of memory and thereby reducing the number of batches required for a portfolio-level analysis. This is the focus of the remainder of this paper.

4 An Efficient CELT Data Structure

In this section, we describe a thorough experimental evaluation of different candidate implementations of the CELT and choose the one most suitable for the performance evaluation of the whole QuPARA system, which is presented in Section 5. First we outline our experimental setup.

4.1 Experimental Setup

The experiments in the remainder of this paper were performed on a 2.66GHz Quad Core Intel Xeon X3350 processor with 4GB DDR2 RAM and three 1TB 7,200rpm SATA disk drives. The operating system was CentOS 6.3 with Java version 1.7.0_03. The Java heap size was limited to 2GB in these experiments. The system evaluation in Section 5 was performed on a cluster of 19 of these machines configured into a Rocks cluster [9] connected using Gigabit Ethernet. For the performance evaluation of QuPARA, the Java heap size per node in the cluster was limited to 1GB, in order to set aside memory needed for the Hadoop runtime system [20, 22]. Hadoop and HDFS was provided as part of the Cloudera BigData platform version 4.7.3 [7], which provides Hadoop version 2.0.0-cdh4.5.0, HIVE version 0.10.0, and Pentaho version 4.8-CE. One of the 19 nodes in the cluster was configured as a master node running the job tracker and job queue and serving as the major name node for HDFS, while the remaining nodes were worker nodes (and data nodes for HDFS) with a total of 72 cores available to run MapReduce jobs. The maximum capacity of HDFS on our system was 20TB.

The data sets used in our experiments were sampled uniformly at random from a real-world portfolio consisting of 1,600 layers with 5 ELTs per layer and 10,000 loss entries per ELT. Each loss entry consisted of one integer and four doubles representing the event id and various loss perspectives, respectively. The YET used for the experiments consisted of 1,000,000 trials, each containing 1,000 events.

4.2 High-Level Data Structure

An ELT stores the loss information associated with each event it covers. Each ELT has a unique identifier. The CELT combines a number of ELTs into one table. The basic lookup operation to be performed on the CELT is to access the loss information for a given (ELT, event) pair. Thus, the CELT is a matrix that stores loss information indexed by ELT ID and event ID.

In theory, the fastest access time is achieved by storing the CELT as a 2d array, as it allows trivial constant-time lookup of the loss information associated with a given (ELT, event) pair using simple index arithmetic. However, since the valid range of event and ELT IDs is the range of 32-bit integers, the size of such a representation would be astronomical. Even if we perform index mapping on the row and column IDs of the array so we store only columns corresponding to ELTs included in the CELT and only rows corresponding to events included in at least one of these ELTs, the table is extremely sparse, as the loss value associated with most (ELT, event) pairs is zero. Thus, even this representation is wasteful.

A simple space-efficient representation of a sparse matrix with expected constant lookup time is a two-level structure consisting of a primary hash table indexed by row indices; the value

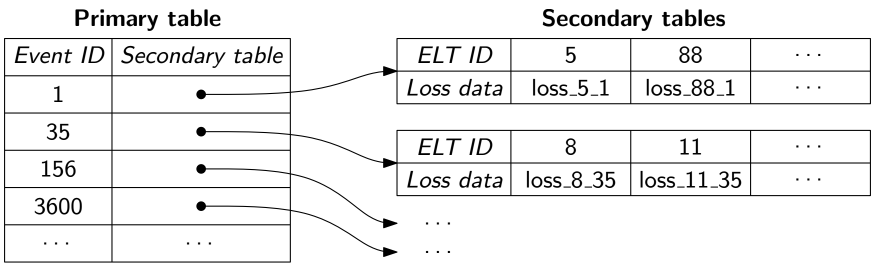

Figure 2: Two-level structure of the CELT

associated with each row index is a reference to a secondary hash table storing the non-zero entries in this row indexed by their column indices. Since the mapper in QuPARA iterates over event IDs and in each such iteration, iterates over the ELTs that record non-zero values for the current event ID, cache locality is improved by choosing event IDs as the row indices used to index into the primary hash table and ELT IDs as the column indices used to index into the hash tables representing the individual rows (see Figure 2). We refer to this two-level data structure implemented using Java STL HashMaps [16] as our *baseline implementation*, which we aim to improve on in terms of space usage, construction time, and lookup time.

An insertion of a value v with key (e, t) into this CELT data structure first looks up the event ID e in the primary table. If this event ID is found, its associated value is a secondary table, into which v is inserted with key (ELT ID) t. If e is not found in the primary table, we first create a new secondary table and insert it into the primary table with key e. Then we insert v with key t into the secondary table just created.

Similarly, a lookup operation with key (e, t) first looks up the event ID e in the primary table. If the event ID is not found, the operation returns immediately and reports that there is no value associated with key (e, t) in the CELT. If the event ID is found, its associated value is a secondary table and we return the result of the lookup with key t in this table (which may be that no value is associated with key t in this table).

In QuPARA, all CELT lookups with the same event ID e (but different ELT IDs) are consecutive. Thus, we can optimize lookups further by performing only a single lookup with key e in the primary table for all these lookup operations. If this lookup fails, we can report failure for the entire batch of lookup operations. If it succeeds, we use the returned secondary table for lookups using the ELT IDs in this query batch. This optimization reduces the total lookup cost in the primary table during a QuPARA run to a minimum, so the lookup cost in the secondary tables dominates the total time spent on CELT lookups. Given that the space usage of the CELT representation is also dominated by the space occupied by secondary tables, our focus is mostly on optimizing the insertion and lookup times and the space usage of the secondary tables.

4.3 Choice of HashMap Implementation

There exist a number of open-source alternatives to the Java standard library (STL) whose data structures have been optimized for performance. The HashMaps in the two-level structure we have just described can be implemented using the HashMap implementations provided by any of these libraries. In our experiments, we considered the Java STL HashMap [**?**], the TIntObjectHashMap class provided by the GNU Trove library [11], and the IntObjectOpenHashMap class provided by the high-performance primitive collections (HPPC) library [17].

In order to ensure that each implementation achieves its maximum performance in terms of construction and lookup time, we tuned various JVM parameters that impact the garbage collection overhead involved in these operations. These parameters include the initial heap size and the ratio between the amounts of space allocated to the young and old generations in Java's generational garbage collector.

A higher initial heap size makes the program use more memory initially but reduces the number of times the heap needs to be resized as the heap space currently allocated becomes insufficient to hold newly created objects. Since QuPARA processes layers in batches chosen to fully utilize the available memory, the heap will always eventually fill up the entire memory, so there is no space penalty to allocating the maximum available heap space also as the initial heap size, but the performance benefit of avoiding heap resize operations altogether is significant. We verified experimentally that setting the initial heap size to the maximum heap size (2GB on our system) results in the best performance for all three HashMap implementations.

A higher ratio between the young and old generation's space allocations reduces the frequency of minor garbage collection runs (which sweep only the young generation) but may lead to old objects remaining in the young generation's space if there is no room left in the old generation's space. When this happens, old objects are swept repeatedly during minor GC runs, which hurts performance. In QuPARA, a very large number of permanent objects (that all eventually become old) are created as part of the CELT. Thus, allocating more space to the old generation should be beneficial. We verified experimentally that the STL HashMap achieves the best overall performance (lookup and insertion performance) with a young-old ratio of 1:3. For the Trove and HPPC HashMaps, the optimal ratios were 1:2 and 1:3, respectively.

In our performance comparisons, we ran each implementation using its optimal JVM parameters as determined above. We compared the space used by each implementation needed to store the same number of elements, and the times taken to process batches of insertions and lookups. Since we first construct the CELT by inserting loss values into it one by one and then perform lookups on the constructed CELT without changing it further, these experiments are representative of the performance characteristics of these HashMap implementations as part of a QuPARA run.

Figures 3, 4, and 5 respectively compare the total insertion time as a function of the number of insertions performed, the total time taken to process 10m lookups as a function of the number of elements inserted into the HashMap, and the memory footprint of the HashMap as a function of the number of inserted elements. The insertion cost of all three implementations differed little up to 9m elements. However, the higher memory footprint of the STL and Trove HashMaps compared to the HPPC HashMap made the garbage collector thrash as these two implementations ran out of memory at 10m and 11m insertions, respectively. The HPPC HashMap also runs out of space eventually, but this happens only after 15 million insertions. The Trove HashMap implementation achieves the lowest lookup cost, which is 4% lower than the lookup cost of the HPPC HashMap, and 24% lower than the lookup cost of the STL HashMap. Given that a smaller memory footprint enables us to process fewer, larger batches of layers in QuPARA, the substantially higher number of elements that can be handled by the HPPC HashMap implementation in the 2GB of memory we had available compared to the Trove HashMap implementation outweighs the 4% increase in lookup performance, and we conclude that the HPPC implementation is the best choice of HashMap implementation to be used in our CELT implementation.

4.4 Hybrid CELT Implementation

An obvious approach to further reduce the space used by the secondary tables (and, hence, of the entire CELT), in order to increase the size of the batches QuPARA can process, is to implement the secondary tables as ArrayLists rather than HashMaps. Since we continue to use an HPPC

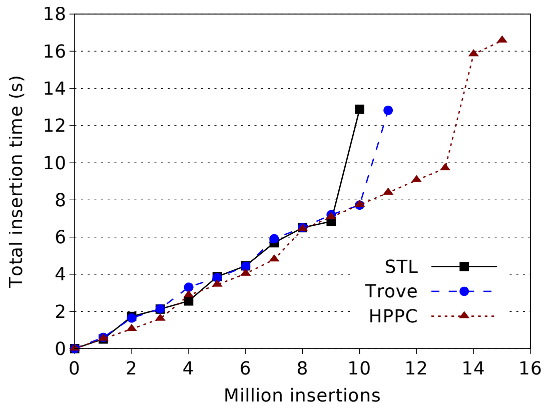

Figure 3: Insertion times of the different HashMap implementations

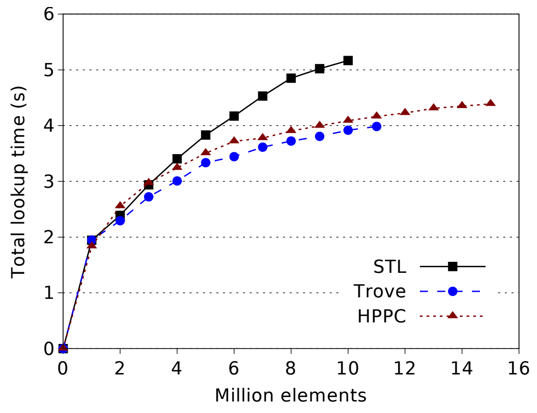

Figure 4: Total time for 10m lookups using the different HashMap implementations

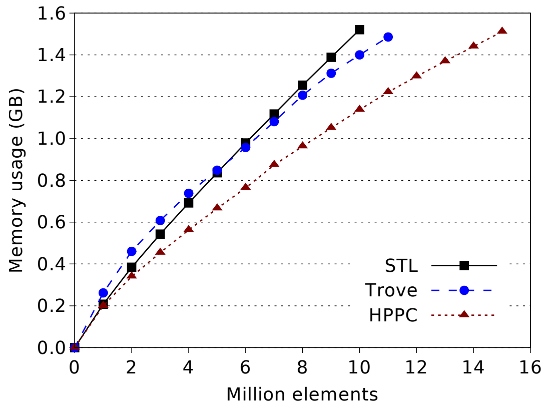

Figure 5: Memory usage of the different HashMap implementations

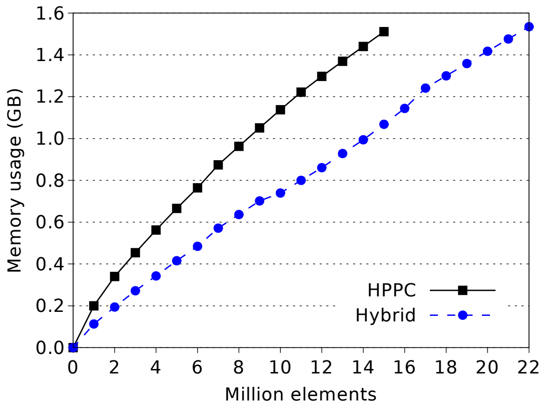

Figure 6: Memory usage of the HPPC HashMap CELT and the hybrid CELT

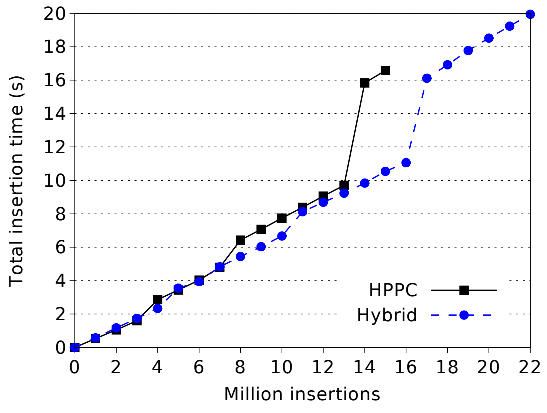

Figure 7: Insertion times of the HPPC HashMap CELT and the hybrid CELT

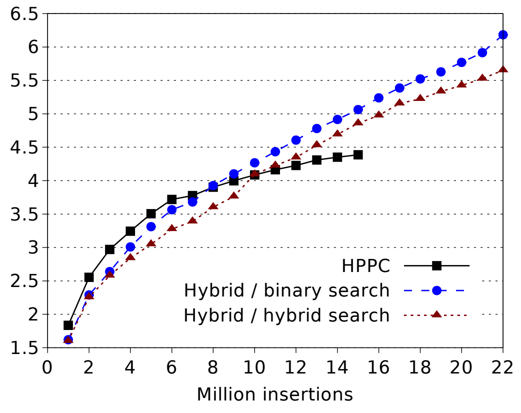

Figure 8: Total time for 10m lookups on the HashMap CELT and the hybrid CELT

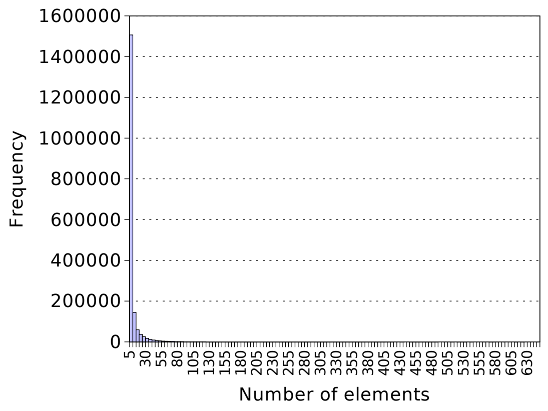

Figure 9: Distribution of secondary table sizes

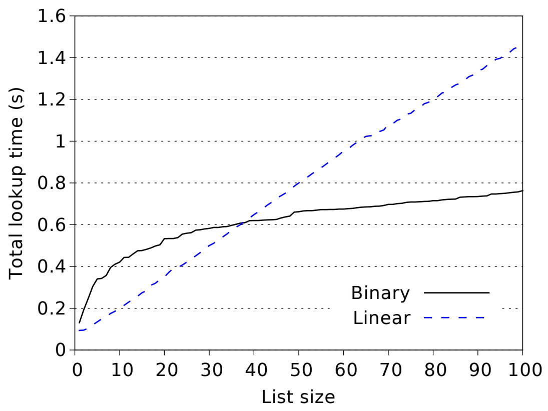

Figure 10: Lookup times using binary and linear search

HashMap to implement the primary table, we refer to this as a *hybrid* CELT implementation. As can be seen in Figure 6, the hybrid CELT implementation uses substantially less space than the CELT implementation using only HPPC HashMaps, which we refer to as the HPPC CELT implementation. The construction of the hybrid CELT proceeds in two phases: The first phase inserts elements one by one as in the HPPC CELT but simply appends each key-value pair inserted into a secondary table to the end of this table. Once all loss values have been inserted into the CELT, we sort the entries in each secondary table by their keys to enable lookups using binary search in these tables. Lookup operations on the hybrid CELT differ from the lookups on the HPPC CELT only in that they use binary search on the secondary tables.

Figure 9 shows the distribution of secondary table sizes. Since over 80% of the secondary tables store 5 or fewer elements, searching them using binary search takes constant time. While the lookup cost of the HPPC CELT is lower than that of the hybrid CELT, as can be seen in Figure 8, the penalty is outweighed by the increase in the batch size that can be processed by using the hybrid CELT implementation (22m vs. 15m elements, a 46% increase). As can be seen in Figure 7, the insertion times of the hybrid CELT and the HPPC CELT differed only insignificantly until the HPPC CELT started thrashing at its limit of 15m elements.

The performance comparisons in Figures 6, 7, and 8 were again performed using the optimal JVM parameters for each implementation. For the hybrid CELT implementation, we verified experimentally that an initial heap size of 2GB once again yielded the best performance, as did a young-old ratio of 1:2.

The heavy bias towards small secondary tables in the CELT suggests another performance improvement we can apply. As shown in Figure 10, for tables up to around 35 elements, linear search is faster than binary search, that is, linear search is faster for most of the secondary tables in our CELT. Thus, we implemented a hybrid search strategy that employs linear search for secondary tables of up to 35 elements and binary search for larger secondary tables. Figure 8 compares the lookup times achieved by the two CELT implementations and the two search strategies (binary or hybrid) for the hybrid CELT implementation. As expected, the hybrid search strategy improves the competiveness of the hybrid CELT implementation in terms of its lookup cost. Combined with the substantially larger batch size enabled by the hybrid CELT and its competitive construction time, this leads us to conclude that the hybrid CELT implementation with a hybrid search strategy is the best choice of CELT implementation to be used in QuPARA.

5 QuPARA Performance Evaluation

We evaluated the speed-up, size-up, and scale-up of our QuPARA implementation on the Hadoop cluster and data sets described in Section 4.1.

Speed-up. The *speed-up* of a parallel program is the ratio between the running time it achieves on a single core and the running time it achieves on P cores. *Linear speed-up* means that the speed-up for P cores is P, that is, the work is perfectly balanced across the cores. For the speed-up test, we fixed the input size at 1,600 layers and increased the number of worker nodes from 1 to 18, that is, the number of cores from 4 to 72. Figures 11 and 12 show the running times and speed-up values achieved, respectively. Up to 24 cores (6 nodes), the speed-up is almost linear. Beyond 24 cores (6 nodes), the speed-up starts to decrease. Due to the substantially decreased overall computation time, the fixed overhead involved in starting Hadoop jobs starts to account for a greater fraction of the total running time at this point. Even so, our implementation achieved a speed-up of 64 with 72 nodes, an efficiency of 88%. This can be considered a very good speed-up result, particularly given that a MapReduce implementation is less fine-tuned than a carefully handcrafted parallel risk modelling system.

Size-up. The *size-up* shows how the running time increases with the input size for a fixed number of cores. Ideally one would aim for this increase to be linear. For the size-up test, we fixed the number of cores at 72 and increased the input size from 100 to 1,600 layers. Figure 13 shows the running time as a function of the number of layers and demonstrates that the running time increases linearly with the input size.

Scale-up. The *scale-up* measures the running time of the system while keeping the ratio between input size and cores fixed. If the running time remains constant, this demonstrates that the system is able to scale to larger input sizes, given a proportionally increased amount of resources. For the scale-up test, we fixed the number of layers per node (4 cores) at 100 layers and increased the number of nodes from 1 to 18. Thus, up to 9,000 ELTs were considered. Figure 14 shows a very slow increase in the total running time, despite the constant amount of computation to be performed by each node. This is due to the increase in the setup time required by the Hadoop job scheduler and the increase in network traffic. Nevertheless, the increase in running time was very slight, so if the hardware scales with the input size, QuPARA is able to process large inputs.

References

[1] E. d. Alba, J. Ziga, and M. A. R. Corzo. Measurement and transfer of catastrophic risks. *ASTIN Bulletin*, 40(2):547–568, 2010.

[2] RR Anderson and Wemin Dong. Pricing catastrophe reinsurance with reinstatement provisions using a catastrophe model. *Casualty Actuarial Society Forum*, pages 303–322, 1988.

[3] A. K. Bahl, O. Baltzer, A. Rau-Chaplin, and B. Varghese. Parallel Simulations for Analysing Portfolios of Catastrophic Event Risk. In *Proceedings of the International SuperComputing Conference (SC12), Workshop on High Performance Computational Finance*, pages 1176–1184, Salt Lake City, Utah, USA, 2012.

[4] Regina M. Berens. Reinsurance Contracts with a Multi-Year Agguegate Limit. *Casualty Actuarial Society Forum*, pages 289–308, 1997.

[5] Bermuda Monetary Authority. Catastrophe Risk Return Guidelines. Technical report, Bermuda Monetary Authority, 2011.

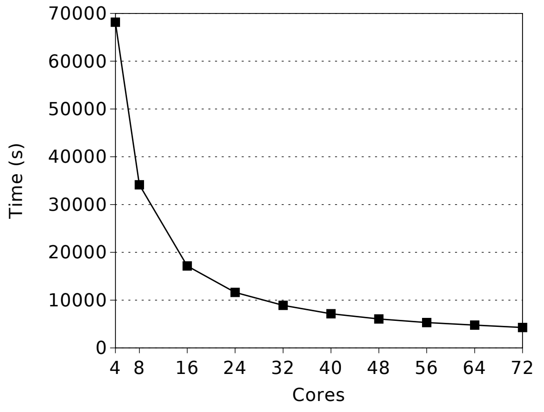

Figure 11: Running time of QuPARA on 1,600 layers using between 4 and 72 cores

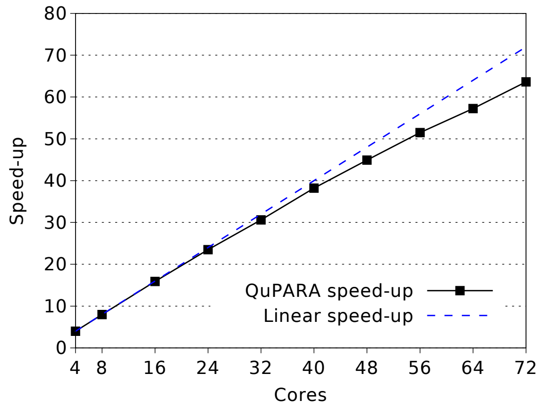

Figure 12: Speed-up of QuPARA on 1,600 layers using between 4 and 72 cores

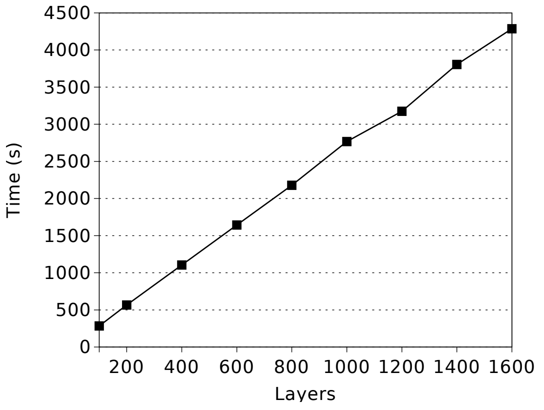

Figure 13: Running time of QuPARA on 100–1,600 layers using 72 cores

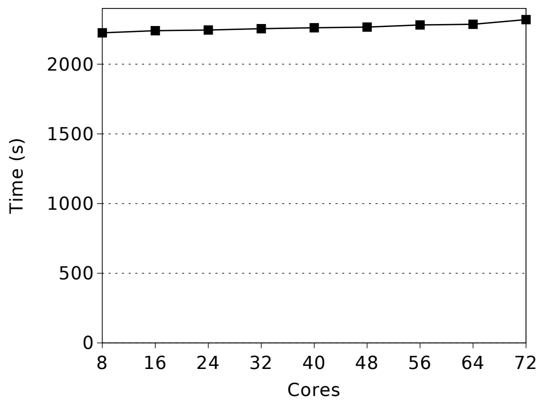

Figure 14: Running time of QuPARA on 25 layers per core using between 4 and 72 cores

[6] Hervé Castella, Gautier de Montmollin, and Erik Rüttener. *Catastrophe Portfolio Modeling: A Complete View*. PartnerRe, Pembroke, 2009.

[7] Cloudera. Cloudera, Ask Bigger Questions, 2014.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[9] Rocks Cluster Distribution. About Rocks Cluster, 2014.

[10] Weimin Dong, Haresh Shah, and Felix Wong. A Rational Approach to Pricing of Catastrophe Insurance. *Journal of Risk and Uncertainty*, 12(2-3):201–218, May 1996.

[11] Rob Eden. GNU Trove: High performance collections for Java, 2013.

[12] P Grossi, H Kunreuther, and C C Patel. *Catastrophe Modeling: A New Approach to Managing Risk*. Huebner International Series on Risk, Insurance and Economic Security. Springer, 2005.

[13] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: A survey. *SIGMOD Rec.*, 40(4):11–20, January 2012.

[14] Dag Lohmann and Feng Yue. Correlation, simulation and uncertainty in catastrophe modeling. In *Proceedings of the Winter Simulation Conference*, WSC '11, pages 133–145. Winter Simulation Conference, 2011.

[15] Glenn G. Meyers, Fredrick L. Klinker, and David A. Lalonde. The Aggregation and Correlation of Insurance Exposure. *Casualty Actuarial Society Forum*, pages 60–152, 2003.

[16] Oracle. HashMap (Java Platform SE 7), 2013.

[17] Stanisaw Osiaski and Dawid Weiss. HPPC: High Performance Primitive Collections for Java, 2013.

[18] A. Rau-Chaplin, B. Varghese, D. Wilson, Z. Yao, and N. Zeh. Qupara: Query-driven large-scale portfolio aggregate risk analysis on mapreduce. In *2013 IEEE International Conference on Big Data*, pages 703–709, 2013.

[19] Andrew Rau-Chaplin, Blesson Varghese, and Zhimin Yao. A MapReduce Framework for Analysing Portfolios of Catastrophic Risk with Secondary Uncertainty. *Procedia Computer Science*, 18(0):2317–2326, 2013.

[20] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, 2010.

[21] The Apache Software Foundation. Apache Hadoop, 2012.

[22] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[23] Margaret E Wilkinson. Estimating probable maximum statistics loss with order statistics. *Casualty Actuarial Society Forum*, pages 195–209, 1982.

[24] Gordon Woo. Natural Catastrophe Probable Maximum Loss. *British Actuarial Journal*, 8(05):943–959, 2002.