Natural Catastrophe Modeling:

Enhancement and extension of a preexisting model

Mark Monroe
markjmonroe@yahoo.com
August 27, 2007

Associated research groups:
Institute of Organic Chemistry, University of Zurich, Kim Baldridge Group
Primary advisor: Dr. Wibke Sudholt
Institute of Environmental Sciences, University of Zurich
Advisor: Prof. Dr. Bernhard Schmid

Summary

Natural catastrophes can have huge impacts on human-made structures as well as on the natural environment. This is why it is of high importance to predict the effects of such events with the help of computer models. Of particular interest are such models in the reinsurance industry, to estimate financial losses from catastrophes. Swiss Re is a large reinsurance company, which is developing and operating a natural catastrophe (NC) modeling software. The Swiss Re NC application is written in Java, and both computation and data intensive. As part of a research collaboration with Swiss Re, the software was provided to the Grid Computing Team within the research group of Prof. Dr. Kim Baldridge (University of Zurich, Institute of Organic Chemistry). The aim of the collaboration is to improve the functioning of the NC application, applying a grid computing infrastructure, that is, distribution of the computations over several computers.

The present thesis was performed in the context of this research project. The two main goals of this thesis were:

- Improve the performance and scalability of the Swiss Re NC model through more efficient data handling.
- Investigate ways to use the NC model framework to calculate the risk of wind damage to forests.

First, a new grid data distribution algorithm was developed and successfully implemented. Prior algorithms would not allow reasonable distribution of the application to more than four or five computation nodes. Through that pervious work, it was discovered that distributing the application was resulting in a large increase of requests to the underlying database, subsequently overloading the database. The new combination algorithm developed reduced the load on the database, allowing a nearly linear decrease in execution times, tested for up to eight computation nodes. In subsequent work, duplicate data in the database was targeted for elimination. To tackle this task, several SQL scripts were developed. By reducing duplicate data at the worst offending places, an application speed increase of up to 9.3% was attained, and database hard drive storage was reduced by 18.5%.

Through the above computational exercises, enough knowledge about the NC model was gained to allow the focus to shift to model development. The main target of such development are so-called “vulnerability curves”, which connect the characteristics of the catastrophe events with the resulting damage for certain types of exposed assets. Forest windthrow was selected as an important example, and good candidate to be modeled using this approach. Windthrow occurs when strong winds cause tree trunks to break or trees to be uprooted. Empirical models have extensively been used to predict the relative risk of different plots of land. Also mechanistic models for the behavior of trees during storms have been developed. However, both model types suffer from problems related to finding relationships between wind speed and windthrow. The predominant issue is the chaotic wind fields. Experiments usually suffer from insufficient remote sensing of wind speed, and no large area multi storm datasets are available. Although a mechanistic model can predict the exact wind speed when a tree will be windthrown, it is hard to ascertain whether that wind speed did or did not occur near the tree, and therefore to validate the model. More experimental data would allow the mechanistic models to be validated as well as the creation of better empirical models. This also means that further research will be needed before realistic vulnerability curves can be generated.

Contents

1 Introduction

Natural catastrophes, such as earthquakes and severe windstorms are relatively rare occurrences. However, when they do occur they often cause large economic and human losses. For example, in 2005 hurricane Katrina caused USD (U.S. Dollar) 135 billion in damages to the southern United States [27]. Although Katrina was an exceptionally large storm, it is not uncommon for a single storm to cause extensive damage. Such damage is increasing due to higher population densities, more population in endangered areas, more insured assets, and higher concentrations of assets. The 2004 storm season was a record year for global property insurance payouts from natural catastrophes at USD 46 billion [28]. While total global damage was estimated at USD 120 billion. Over 300'000 were killed by the same events. The high losses can be attributed to 13 hurricanes and ten typhoons, many impacting the U.S. and Japan. Seasons as bad as 2004 may become more common and in fact the 2005 storm season was even worse in terms of property insurance payouts, which amounted to USD 78 billion [27]. Storm frequency is forecast to increase due to global warming [12]. However, the extent of the increase is unknown.

An increase in storm frequency will affect ecosystems as well as humans. On Kuiu Island, off the coast of Alaska, for instance, forest structure was different between areas with high and low levels of windthrown trees [22]. Windthrown trees are trees that are uprooted or have their trunk broken by strong winds. Large areas of the forest canopy can be opened allowing shade intolerant species to grow. A higher storm frequency might change forest structure by favoring tree species and age mixes that are less prone to windthrow. Any alteration in selection pressure can result in structure changes. For example, in California, U.S.A, models predict increasing ozone pollution to change forest species mixes by affecting growth rates [31].

Many groups including foresters, homeowners, governments, and insurance companies are interested in predicting the frequency, intensity, locations, and damages of natural catastrophes. Due to the potentially huge claims directed at them in case of the occurrence of large catastrophes, reinsurance companies are very interested in such predictions. Reinsurance is insurance for insurance companies. Often an insurance company will have many individual real estate and similar insurance contracts in the same area, for example in the southeast United States. When a large hurricane like Andrew or Katrina moves across the area, many of the insurer's clients will be affected. In fact, so many that the insurance company might have trouble meeting the contracts. By purchasing reinsurance, an insurance company ensures that it can meet all its contracts should a large event occur.

Before a reinsurance company can offer such catastrophe coverage it must be priced. To price the coverage, the risk of an event occurring and the resulting damage must be estimated. Catastrophes by definition do not routinely occur and are unpredictable beyond a short time horizon. In lieu of prefect prediction, it still is possible to predict event trends through time. Although one does not know exactly when an event will occur, one can statistically estimate how often an event with certain magnitude and characteristics will likely occur in a given time span. Such a probabilistic model is the basis for the present thesis work.

As part of a research collaboration with the University of Zurich under the title "Distributed Computing for Reinsurance Related Calculations – Design, Algorithms and Runtime Experience", a Natural Catastrophe (NC) computer model has been

provided by Swiss Re to the research group of Prof. Dr. Kim Baldridge in the Institute of Organic Chemistry. Swiss Re is one of the world's largest reinsurance companies, and is headquartered in Zurich [26]. The goals of the overall project are to increase the performance of the application. Execution speed, efficient allocation of computational resources, application scalability, and stability all fall under the performance criteria. Those goals can be met in a number of ways. In this case it was decided to use a distributed computing approach, which will be discussed shortly.

Within this collaboration project, the present Master's thesis is mainly concerned with two aspects:

1. Enhancement of the performance and scalability of the Swiss Re NC model by improving the data handling: Previous work identified less than optimal handling of the huge amounts of data needed for the risk calculations by the current implementation of the NC model [13]. This leads to unnecessary load on the components of the involved computers such as the CPU (Central Processing Unit), and too much data being sent via the network connections between the computers. This thesis will in particular target the removal of redundancies and the minimization of the data transfer. By this, one hopes to increase the execution speed of the application, and make it easier to distribute the calculations over many machines. Furthermore, this part of the thesis serves as introduction into the NC model and software.
2. Outlining of ways in which the Swiss Re NC model can be extended to calculate risks in the environment, such as the risk of wind damage to forests: The current NC model only calculates risks from natural catastrophes for human-made structures such as buildings. However, the principles of the model are very general, and can thus also be applied to calculate the risks of damage to natural entities. This is of particular interest because the method has not been applied to natural entities and may give new insight into how such systems are impacted by catastrophes. As an initial example, how the windthrow of trees could be modeled within this framework will be analyzed.

After an overview of the Swiss Re NC model and the used computational methodology in Chapter 2, this thesis is split into two parts mirroring the above goals: Chapters 2 to 6 address Goal 1, and are focused mainly on model computational aspects. Chapter 7 focuses on Goal 2, the extension of the risk model. The thesis closes by an outlook and conclusions in Chapter 8.

2 Background of Model and Methodology

2.1 Functional Overview of the Natural Catastrophe Model

A detailed description of how the NC model works in principle can be found in the Swiss Re Natural Catastrophes and Reinsurance publication [29]. This publication will serve as the basis for the following short description. The overview here will concentrate on the modeling of storm damages, as they are important in Chapter 7. However, very similar procedures can be applied to model other natural catastrophes such as earthquakes.

The NC model is divided into two parts: The first part is a set of climate models that generate catastrophes of a given type for a given area. Climatological data from past events is used to estimate the event paths, intensity, and frequency. However, the known climatological record is too short to give statistically valid results. Therefore, it is augmented by simulated events based on the historical data. The generated cyclones have never occurred in reality, but try to provide a realistic and statistically sufficient ensemble of events that could occur. The problem with using past data is that it is only valid if the climate is constant through time. Unfortunately, global warming models predict an increase in frequency and intensity of some types of storms [12]. Adding this predicted rise in events should improve the predictive power of the model, and is thus already employed at Swiss Re.

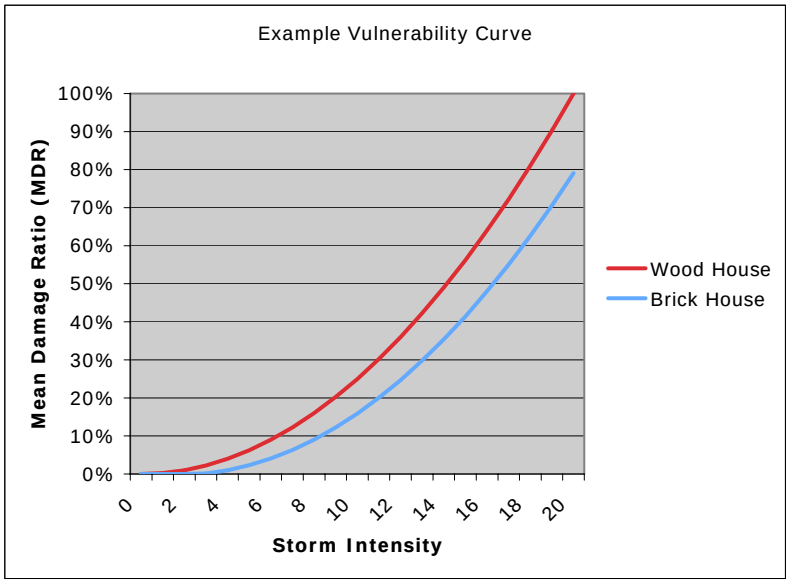


Figure 1: Example vulnerability curve.

The second part of the model applies the simulated catastrophes to entities of interest, and calculates the incurred damage. Each entity type is assigned a so-called “vulnerability curve”. A vulnerability curve defines the relationship between the event intensity or a similar characteristic of the catastrophe events, and another variable that describes the resulting damage (Figure 1). In this case it is the mean damage ratio (MDR), or the percent of total value lost due to the storm. Different entity types have different vulnerability curves. As an example, there are a number of building attributes that will alter a house’s ability to withstand a storm. Construction material is one such parameter that can define a type of building. In our example, wood and brick houses respond differently to storms and therefore have different vulnerability curves. A storm

of intensity 19 would completely destroy a wood house, while the brick house only loses 80% of its monetary value.

2.2 Computational Overview of the Natural Catastrophe Model

Of the two model parts, event simulation and application of those events to entities, only the latter is the focus of the present thesis. The technical aspects of simulating the events will not be detailed here. The events are calculated periodically and subsequently stored in a database. They are not a continuous burden on computation resources, and are taken as persistent parameters here. Also the vulnerability curves are constructed in preparative model building steps, and provided as persistent parameters via the database. Their development for the example of forest windthrow will be the topic of Chapter 7. The most intense usage of computation resources actually occurs when applying the events to the entities. Such calculations are routinely performed on a day-to-day basis. This is where the performance improvements described in Chapters 3 to 6 will concentrate on.

The collection of entities that describes a particular scenario is contained in an input file called a “portfolio”. In the current model implementation, this information consists among other things of building locations, types of buildings, and insurance and reinsurance conditions. A four-level tree structure is used to arrange the data in a portfolio (Figure 2) [13]:

- L3 is the highest level and contains global information regarding the portfolio. There is one per portfolio, and it is the parent of all L2s.
- L2 contains insurance information. There can be many L2s in a portfolio. Each L2 has L1 children.
- L1 contains insurance and building information. Each L1 is associated with one L1Info. Each L1 has L0 children.
- L0 contains insurance and building information. The L0 level is the lowest level.
- L1Info contains location information. Each L1Info is associated with one or more L1s.

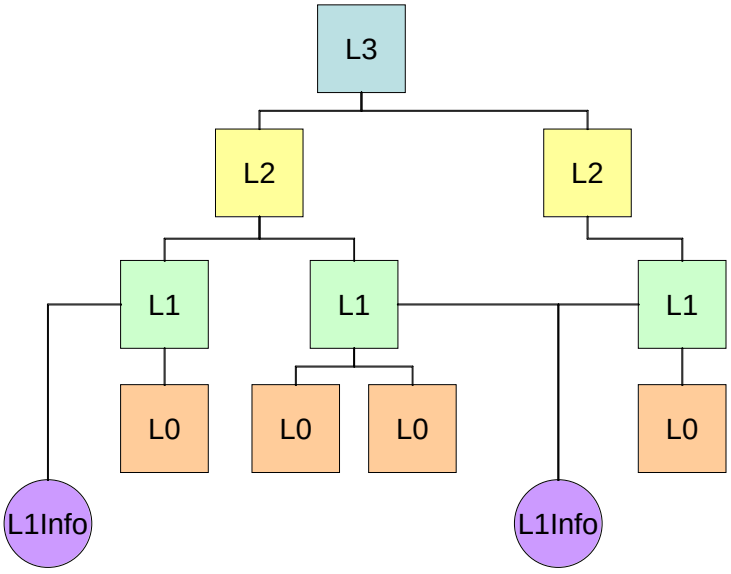


Figure 2: Portfolio tree structure.

The NC model is comprised of three unique sequential processes. Each process must be run prior to executing the next process. Having individual processes allows checking of intermediate data and recovery if a specific process fails. Depending on the portfolio, the individual processes can take quite a while:

1. The “Import” process reads in the selected portfolio. The data is checked for consistency and written out to a database. This process is not CPU intensive, just database intensive. After this step, the portfolio file is no longer used. Instead, the subsequent processes query the database when portfolio data is required.
2. The “Encode” process uses the portfolio data stored in the database. Data are checked for consistency and sums of various child values are generated for their parent node. Entity locations are matched to the nearest known location where event intensity data exists. This process is database intensive, not CPU intensive.
3. The “Rate” process estimates the resulting damage done to a portfolio for a given catastrophe type. Simulated catastrophes are applied to each building in a portfolio, insurance and reinsurance conditions are figured in, and the results are summed. This process is both CPU and database intensive.

The Import and Encode processes only run on individual computation nodes because neither process is CPU bound. Rather, the database is the bottleneck, as large amounts of data are sent to and received from the database. In contrast, the Rate process is CPU intensive. Therefore, in previous work it has been enabled to run on a distributed computing infrastructure (see below) [16]. A single rate job is split up into a number of partial jobs, which run in parallel on the computation nodes. The partial jobs are independent of each other, that is, they do not communicate among themselves. Inner job communication was avoided due to possible negative affects of network latency and data transfer times on the execution time.

2.3 Distributed and Grid Computing

There are a number of ways to increase the execution speed of an application. Simply purchasing a more powerful computer is one option. However, after a certain point, it becomes very expensive to upgrade to the next more powerful machine. Supercomputers and mainframes are examples of very large single machines, although even these usually internally consist of several processors and thus often require parallelization of the application codes to run on them efficiently [25]. Generally, the high expense and complex maintenance associated with extremely large computers makes them cost prohibitive. For that reason, among others, academia as well as industry have been moving away from the single all-powerful machine to systems where many smaller machines work together lately.

Small and medium sized machines are inexpensive and readily available. As more computation power is needed, additional machines can be dynamically added to the infrastructure. Multiple machines also provide a buffer against equipment failure: If five of twenty machines are not functioning, the application can still execute on the other fifteen, while the five non-functional machines are repaired or replaced. How to build systems of interconnected computers, and to create or modify applications so that they can use such infrastructures is the topic of distributed computing [5].

This thesis deals with two types of distributed computing infrastructures, “clusters” and “grids”: Clusters are comprised of nearly identical computers, located at the same facility, and connected by high-speed network interconnects [3]. Grids can be made up

of radically different computers, located at different facilities and even different institutions or countries, and may be connected by a slow or unreliable network such as the Internet [8]. The differences between a cluster and a grid are partly a matter of degree, although the heterogeneity, networking, security, and policy issues make a grid considerably more complex than a cluster. In the present study we are mainly interested in distributing a software application. For this, as an initial approach, a grid can be modeled by a cluster. Therefore, for our purposes we will use the terms grid and cluster interchangeably.

The NC application is well suited for a distributed computing environment: As will be discussed in more detail later, the NC algorithms allow independent and simultaneous execution of tasks, both requirements for distributed computing. Different execution units can be sent to a number of computers, there execute in parallel, and then return a result. Those results are then compiled, and a final result is given to the user. It is expected that such a setup increases, among other things, the following characteristics of the NC calculations:

- Performance, the overall execution speed of the application
- Scalability, the ability of the application and infrastructure to be enlarged to handle an increase in the number of system users
- Stability, the systems ability to recover and continue to operate when there is a hardware or software error
- Fairness, distributing the computational resources equally among the applications and jobs running on the system

Therefore, a grid or cluster distributed environment was chosen as testing and execution infrastructure for the NC application.

2.4 Computer Infrastructure

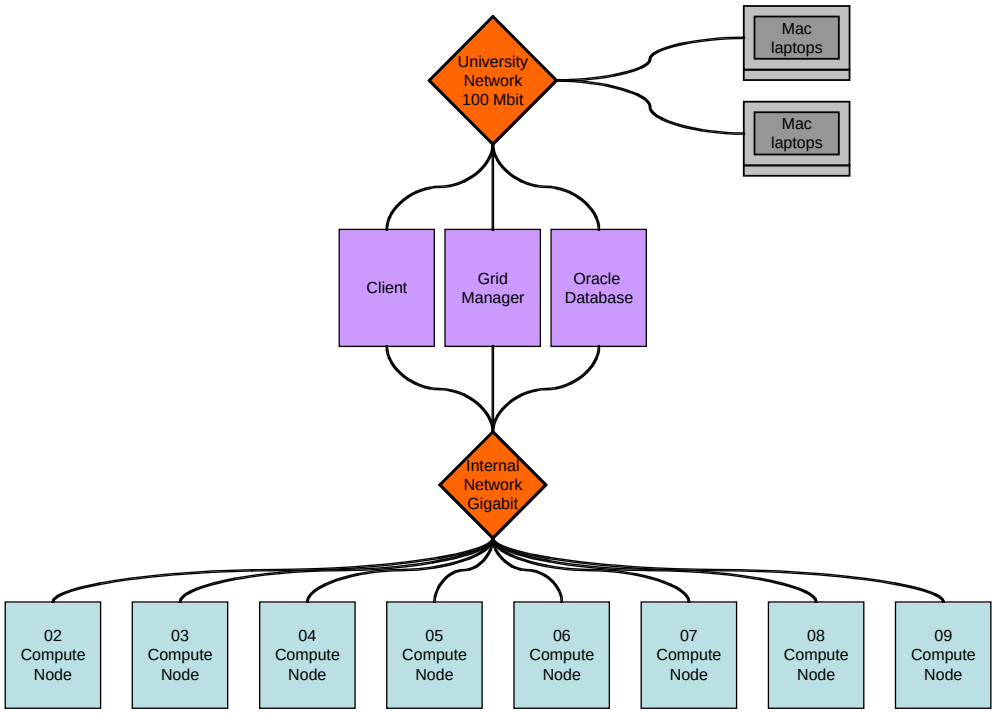


Figure 3: Grid structure [13].

The cluster infrastructure that is modeling a grid at the University of Zurich is detailed in Figure 3. The grid consists of up to eight computation nodes. The corresponding hardware is listed in Table 1. The client node, which is identical in hardware and operating system to the compute nodes, is used for non-grid enabled processes. It is also used to start grid-enabled jobs. The grid manager administers the grid resources: Nodes can be added or subtracted from the grid, memory allocated to the computers' Java Virtual Machines (JVM) may be changed, software can be deployed to the resources, compute jobs are assigned to machines, and other such management tasks. An Oracle database stores the required data, such as the events, vulnerability curves, and imported portfolios, on a separate database server, from which two alternative ones were available in this project. The Oracle database hardware for the system used is presented in Table 2. Table 3 details the versions of the pertinent software.

Two application development environments were principally used. Eclipse, an open source development platform, version 3.1 was used as the coding environment [6]. Oracle SQL Developer version 1.0, a free graphical database tool, was used when manually working with the database.

During this thesis a major software change occurred. The grid middle ware was changed from DataSynapse Grid Server to IBM's WebSphere 6 [30]. Chapter 6 is the only section that was completed under the new WebSphere setup. A WebSphere installation on a single desktop Linux machine was used. The timings in Chapter 6 cannot be compared to the other chapters. However, the interest is only in before and after timings after having modified data in the database. The platform used to run the application does not alter the results. Therefore, the details of the WebSphere single machine setup will not be presented.

Table 1: Grid computation node hardware.

Hardware	Setup
Processors	Dual Xeon 3.20GHz, 2048 KB l2 cache, hyper threading on
Memory	6GB RAM
Hard disks	1x SATA 143 GB drive
Network	Gigabit Ethernet

Table 2: Database hardware.

Hardware	Setup
Processors	Dual Xeon 3.20GHz, 1024 KB l2 cache, hyper threading on
Memory	6GB RAM
Hard disks	6x 143GB SCSI hard drives, of which 4 are in a RAID 0+1 274GB volume for database use
Network	Gigabit Ethernet

Table 3: Grid software versions.

Software	Version
Operating System	SuSE Linux Enterprise Server 9
Grid Middleware	DataSynapse GridServer versions 4.x [1]
NC Model Source Code	Sun Java 1.4.2.10, ~160,000 lines
Java Virtual Machine	Sun Java HotSpot Client VM 1.4.2.10
Database	Oracle Standard Edition 9.2

2.5 Testing Methods

Throughout the computation portion of this thesis, a number of different portfolios have been used for testing purposes. Swiss Re has supplied these portfolios. They are of different size and structure, contain different types of insurance conditions, and assess risks for different types of catastrophes. Most tests have been run with a subset of the supplied portfolios. Subsets were chosen based on a number of factors, including length of portfolio execution time and format requirements. Using the same portfolios repeatedly allows the software changes to be isolated as the cause of any change to system performance.

To have comparable process timings, only one portfolio could be run on the system at a time. A queuing system on the client node was used to assure that this was the case. While a test was running, a number of system parameters were monitored such as database and CPU usage. At the end of a test, the Rate process results were compared with expected results from Swiss Re to verify the run completed successfully. The preceding Import and Encode processes do not return a numerical result, only processed data, and were thus not separately checked. It should be noted that the data provided by Swiss Re within this research project were partly disguised for confidentiality reasons, so the result values have no actual meaning and are not further discussed here.

Overall, running a test on the infrastructure consisted of the following steps, simplified here for readability:

1. The user connects to the client node, in this case through a secured shell.
2. The user starts the job on the client node.
3. Resource monitoring starts, and the client node does any initial non-grid enabled processing.
4. Grid enabled jobs are sent to the grid manager.
5. The grid manager starts the distributed jobs on the compute nodes, and the partial jobs run, if possible in parallel, on the assigned nodes.
6. Once all computation nodes have finished, the client node does any final processing.
7. The result is checked against the expected result to verify the run was successful, and the job and the resource monitoring stops.

3 Combination of the ImportCSV and Encode Processes

3.1 Introduction

3.1.1 Overview

The purpose of this part of the thesis was to improve the performance of the Import and Encode processes of the NC software. Previously, the University team had been predominantly focused on the Rate portion of the application. This was the first extension of the original work towards a different part of the NC application. Two performance measures were selected to improve: execution speed and number of database rows affected.

3.1.2 Benefits

Decreasing database communication as well as increasing execution speed has been a goal for all phases of the Swiss Re-University collaboration project. It is usually thought that by deceasing database accesses the execution speed of the application will increase. However, this will only be the case if the database accesses are the bottleneck.

Anytime a database row is selected, inserted, updated, or deleted, the structured query language (SQL) statement returns the number of rows that were affected by that statement. This number is therefore a good indicator of how much processing the database had to do during that particular run. The more rows are affected, the greater is the load on the database. Assuming the database is the limiting resource, reducing its load would allow more simultaneous executions of the same application. Remote database access is a relatively slow process compared with accessing local storage devices, thus reducing the necessity of it may decrease the execution time of the overall job.

3.1.3 Original Implementation

In the NC code from Swiss Re, the Import process is composed of two separate algorithms based on the type of file imported. ImportXLS uses Microsoft Excel workbooks (XLS) as the input file. The data is read from the Excel file directly into memory. Since all of the data is held in memory at one time, it can only be applied for small and medium sized portfolios. Due to its predominant use for such non-data intensive and short running portfolios, the ImportXLS process was not selected for optimization here.

The ImportCSV process uses comma separated value (CSV) text files generated by a separate application as input. The files come zipped into a single compressed archive. Each file in the archive holds all the instances of one type of Lx object. For example, all of the L2s would be in a text file called L2.txt. Since the Lx objects are in a tree structure (see Section 2.2), each line in a file has a key that links it to its parent and children. The keys are then used to build the Lx tree structure from the text files.

The ImportCSV process opens the zip archive and processes each file in a separate thread. Data is read in a few lines at a time and minimally processed before it is batch-inserted into the database using JDBC (Java Database Connectivity) prepared statements. Unlike the ImportXLS process, the data is never parsed into objects or held in memory for long. ImportCSV is predominantly used for data intensive and long running medium and large size portfolios.

The Encode process is a completely separate process from importing. No data is directly taken over from the Import process and used during encoding. Encoding first queries the database to get the necessary Lx objects. The Lx objects are then encoded. Encoding includes parsing the L1Info location information, summing up values, and error checking, after which the updated objects are batch-saved back into the database. The process thus utilizes the database extensively.

3.2 Approach and Implementation

3.2.1 Combined ImportCSV/Encode

By combining the Import and Encode processes, a back-and-forth transfer of data to and from the database can be eliminated. This should reduce the number of SQL statements, and therefore the database load and hopefully the overall execution time. A new combined ImportCSV/Encode process that imports and encodes the data simultaneously was created. The new algorithm reads the portfolio data, encodes it, and only then sends the Lx objects to the database.

3.2.2 Algorithm

Data from a CSV file is read into a record. A record is comprised of a raw line of text from one of the text files that comprises a CSV zip archive. Each object, L2, L1, L0, and L1Info, has a record. An L2 record holds all of the records of its children. The records are used to separate the reading of the files from the processing of the data they contain. An Lx object of the appropriate type is created using the record. The Encoding functions work on Lx objects. Once the Lx objects have been encoded, they are then batch-inserted into the database.

The pseudo code for the algorithm implemented is outlined below. Each block represents a thread of execution:

- **ImportCSV thread starts**
- Start the processL2Data threads (see below)
- Wait for them to complete
- Fill L3 with generated totals
- Insert L3 into the database

- **ProcessL2Data thread starts**
    - Create a complete L2Record
        - Read a single L2 from L2.txt
        - Read all of the L1s for this L2 from L1.txt
            - Read all of the L0s for an L1 from CONDITION.txt
            - Read the L1Info for an L1 from L1INFO.txt
- Use the L2Record to create the L2 object
- Start X processL1Data threads (see below)
- Wait for them to complete
- Fill L2 with generated totals
- Insert L2 into the database

- **ProcessL1Data thread starts**
- Get X number of L1s and L1Info Records
- Encode the L1Infos for those L1s
- Encode L1s

- Encode L0s
- Insert L0s into the database
- Insert L1s into the database
- Repeat until all L1s for this L2 are processed

3.2.3 The Three Thread Groups

The ImportCSV thread starts the Importing process. It does little but initializes the needed classes and spawns the processL2Data threads.

The processL2Data threads generate the L2Record, which holds all of the text data needed to create its child objects. After the text data is gathered, the L2 is created from it. Next it spawns the processL1 data threads and hands over the L1Records for them to process. After all L1s are processed, final L2 processing is done. The process repeats. When all L2s have been processed, the thread exits.

The processL1Data threads use the L1Records to generate the L1, L1Info and L0 objects that are used during the Encoding process. Encoding and object insertion are done. When all of the L1s are assigned to a processL1Data thread, the thread exits.

3.2.4 Ordering of Input Files

The algorithm that reads the text files assumes that the lines are ordered by their parent keys. The ImportFileReader holds a CSVPushbackReader that points to a line in each file. First it reads a line from L2.txt. Next, the key linking that L2 with the L1s it owns is extracted and used to gather all of the L1s. Each L1 key is used to gather all of its associated L0s. If the keys are not in the right order, the pointers will not be pointing to the correct record when advancing to the next line. The code will then believe that there are no records for the upper level object. The files thus need to be sorted so that the keys are ordered by the owner of the record, that is, order L1s by L2 key, order L0s by L1s. This does not just mean sorting by a particular key, it means that the order of the records in each file must match. Currently this is done separately using a Unix shell script.

3.2.5 Comparison

The original Import process was very simple. It read each file comprising a CSV zip archive in a separate thread and used prepared statements for database insertion. Little data processing was done. The data processing was done during the Encoding step. The Encoding process operated on nearly complete objects. Those objects were created using an SQL persistency layer.

In the new combined ImportCSV/Encode algorithm, to use the Encoding functions already present, a file persistency layer was created that uses records to create the objects. Although the Encode and Import processes have been combined, the result is a simplified algorithm. Threads of identical nature are used to process the data in parallel, of which there are only two thread levels. Previously, a complex multi-threaded structure was required to handle the extensive database interactions during the Encoding process

3.3 Tests and Results

3.3.1 Measurements

The reference values used for comparison in the following came from running test cases with the original University version of the NC code on the client machine [13]. The new algorithm was run on the client node as well. All parameters were identical to the original version, except for the number of Import threads, which the new algorithm required to be increased. Five test cases were chosen for testing the code: large_dlm_no_inuring_1, large_dlm_no_inuring_2, large_dlm_no_inuring_5, large_dlm_with_inuring_4, and medium_dlm_with_inuring. The selected test cases had to be in CSV format and to complete execution in a reasonable amount of time.

3.3.2 Execution Time

The execution times must be compared using the combined total times for the Import and Encode processes. Table 4 shows the times for the NC code from the original versus the ImportCSV/Encode algorithm. The new algorithm does not affect the execution times of the Import and Encode processes uniformly. For the large_dlm_no_inuring_2 case the execution time was reduced by 36.08%, while for the large_dlm_no_inuring_5 test case it increased by 12.58%. Compared to the other test cases, large_dlm_no_inuring_5 caused the lowest average CPU load, 5.13% (see Table 5). Looking at the CPU usage of the database for the original code runs, it held steadily at 85%, higher than for the other test cases that showed improvements in execution time. This leads to the conclusion that under high database load, the new algorithm might not be as efficient as the original one.

Table 4: Comparison of execution time, in milliseconds (delta calculated = new algorithm – original algorithm).

Test Case Name	Import/Encode Time Delta	Percentage change from original
large_dlm_no_inuring_1	-68865	-6.77%
large_dlm_no_inuring_2	-604809	-36.08%
large_dlm_no_inuring_5	676041	12.58%
large_dlm_with_inuring_4	-16611	-4.47%
medium_dlm_with_inuring	-8649	-13.94%
Total	-22893	-0.27%

Table 5: Average CPU usage for combined ImportCSV/Encode processes, in percent (taken at one-minute intervals).

Test Case Name	User/%	System/%	IO wait/%	Idle/%
large_dlm_no_inuring_1-0nodes	26.77	0.70	0.42	72.12
large_dlm_no_inuring_2-0nodes	26.45	0.86	0.50	72.19
large_dlm_no_inuring_5-0nodes	4.14	0.50	0.49	94.87
large_dlm_with_inuring_4-0nodes	17.85	0.76	0.49	80.90
medium_dlm_with_inuring-0nodes	16.50	0.94	0.75	81.82

Table 6: Time needed to sort CSV files, in seconds (measured on client node).

Test Case Name	Sorting Time
large_dlm_no_inuring_1	21.173
large_dlm_no_inuring_2	27.217
large_dlm_no_inuring_5	28.568
large_dlm_with_inuring_4	31.091
medium_dlm_with_inuring	3.589

It should be noted that the algorithm requires the sorting of the CSV input files. The time taken to sort the files is given in Table 6. The longest sort time was for the large_dlm_with_inuring_4 case at 31 seconds. For that particular test case, including the time it took to sort the file would negate the gains in speed from the new algorithm.

3.3.3 Database Rows Affected

As expected, there is an overall reduction in the number of affected database rows per job, as shown in Table 9. The percentage difference is calculated by adding all of the rows affected that occur through the Import, Encode and Rate processes. The values for the original code and the ImportCSV/Encode code are then compared. The reduction varies greatly for the select statements, but not as much for the updates. The percentage change for the select statements is rather low, pointing to the Encode process as not generating a high percentage of the total rows affected by selects. The Rating process is generating most of the selects. Out of a total of 82 million rows affected for the large_dlm_no_inuring_5 test case, 72 million of those result from selects against the T_INTENSITY and T_EVENT tables (see Table 7). Those tables are used during the Rate process. To reduce the number of database rows affected for the select statements, the Rate process algorithm would need to be modified. Updates have almost been eliminated with the new algorithm. Most of the updates occurred during the original Encoding process.

Table 7: Original code database rows affected (Other selects: number of rows affected not including the T_EVENT/T_INTENSITY tables, T_EVENT select: number of rows affected in the T_EVENT/T_INTENSITY tables).

Test Case Name	OTHER SELECT	T_EVENT SELECT	INSERT	UPDATE	DEL	SEQ	SEQ-INC	Totals
large_dlm_no_inuring_1	10048771	45790331	4927108	1110560	2	2204	17	61878993
large_dlm_no_inuring_2	10814459	60019058	6501740	1787850	2	3055	17	79126181
large_dlm_no_inuring_5	10246022	71726924	4868690	1241659	2	2148	16	88085461
large_dlm_with_inuring_4	4348147	25191204	2525608	953999	2	1187	17	33020164
medium_dlm_with_inuring	681005	1539043	379156	157926	2	190	17	2757339
Totals	36138404	204266560	19202302	5251994	10	8784	84	264868138

Table 8: Difference of database rows affected from the original code, in absolute numbers (Negative numbers mean that the new algorithm reduced the number affected. DELETE and SEQ-INC omitted because the change was 0).

Test Case Name	SELECT	INSERT	UPDATE	SEQ	Totals
large_dlm_no_inuring_1	-2334209	-3059	-1108559	0	-3445827
large_dlm_no_inuring_2	-6224081	-195602	-1468239	-97	-7888019
large_dlm_no_inuring_5	-3384275	-2408	-1235526	-1	-4622210
large_dlm_with_inuring_4	-1884072	-335	-953602	-11	-2838020
medium_dlm_with_inuring	-288712	-121	-157889	-2	-446724
Totals	-14115349	-201525	-4923815	-111	-19240800

Table 9: Difference of database rows affected from the original code, in percent (Negative numbers mean that the new algorithm reduced the number affected. DELETE and SEQ-INC omitted because the change was 0.00%).

Test Case Name	SELECT	INSERT	UPDATE	SEQ	Totals
large_dlm_no_inuring_1	-4.18%	-0.06%	-99.82%	0.00%	-5.57%
large_dlm_no_inuring_2	-8.79%	-3.01%	-82.12%	-3.18%	-9.97%
large_dlm_no_inuring_5	-4.13%	-0.05%	-99.51%	-0.05%	-5.25%
large_dlm_with_inuring_4	-6.38%	-0.01%	-99.96%	-0.93%	-8.59%
medium_dlm_with_inuring	-13.00%	-0.03%	-99.98%	-1.05%	-16.20%
Totals	-5.87%	-1.05%	-93.75%	-1.26%	-7.26%

3.4 Conclusions

As was the original goal, the total number of affected database rows was reduced by combining the Import and Encode processes. The total number of select statements decreased by 4% to 13%, depending on the test case. Combing the Rate and ImportCSV/Encode processes would decrease the number of select statements even more.

The other main goal of increasing speed had mixed results. Reducing the number of database rows affected does not necessarily increase the speed of the Encoding process. The current multi-threading structure is not optimal. It is possible, at the end of a job, to only have one thread processing data. This is due to the way that the L1 threading level is locked down to a specific number of threads. A dynamic approach that increases, or decreases, the number of threads processing the L1s dependent on the number of them, would be more efficient. The large_dlm_no_inuring_5 test case needs to be analyzed in more detail. The low CPU usage and high database load of this case differentiates it from the other cases that were run.

4 Combination and Distribution of the ImportCSV, Encode, and Rate Processes

4.1 Introduction

4.1.1 Overview

The new ImportCSV/Encode/Rate algorithm is a modified version of the previously described ImportCSV/Encode combination algorithm (Chapter 3). Along with the ImportCSV/Encode/Rate algorithm, a new distribution approach was created. Through combining all three processes, even more of the database back-and-forth will be eliminated. The same performance measures used previously were once again selected for improvement: execution speed and number of database rows affected.

4.1.2 Original Implementation

The originally used event set-based distribution algorithm works by assigning each grid compute node a set of events (simulated catastrophes), which are then applied to the entire portfolio. When all nodes have finished their calculations, the client node sums all of the values returned by the compute nodes. This approach requires loading of the full portfolio on each compute node. This produces a high load on the database, which limits the performance and scalability of the algorithm. Furthermore, such a distribution technique requires all portfolio data to be present in the database. Since the ImportCSV, Encode and Rate processes will now occur at the same time, the necessary data is only available at the completion of the new algorithm.

4.1.3 New Implementation

The new L2 set-based distribution algorithm splits a portfolio at the L2 level, meaning that each compute node calculates the partial results for a subset of L2s in a portfolio. This decreases the load on the database by eliminating the need for each node to request the entire portfolio. A reduced database load leads to better scalability. More compute nodes can be used to process a portfolio, which is expected to improve the execution time. Also, portfolios previously too large to be run with the available memory can now be split up into smaller pieces. The partial portfolios can be executed individually on a single machine or on a grid, thereby removing the pervious limit to a portfolio's maximum size.

Unfortunately, for this initial implementation of the L2 distribution algorithm, in agreement with Swiss Re we had to ignore some of the more specialized business requirements. For example, there are a number of business rules that require the global portfolio structure to meet certain guidelines before Rating. For these to be checked, the entire portfolio must first be encoded. Since the portfolio is simultaneously imported, encoded, and rated now, such rules cannot be applied. Most importantly, the portfolio structure is not a true tree, but a few levels are mathematically coupled in the Rating algorithm. Therefore, in its current form the algorithm will not necessarily calculate the correct results. However, for the test cases used here the results were always within the correct range.

4.1.4 Benefits

Before, all algorithms for distributing the Rate process onto a cluster or grid infrastructure in the Swiss Re-University of Zurich collaboration project were based on

the event set-based approach [13]. The main result of these studies has been that the distributed Rate process shows a decrease in execution time as one adds more nodes, until one adds about six nodes or more (the actual number depends on the test case). Above five nodes, the execution times increase, meaning that scalability beyond this number is not achieved. This can be explained by the finding that as the number of nodes increases, the resulting data transfer and load on the database increase as well. The database seems to be overloaded by the extra requests for Lx objects, since the event set-based algorithm requires every compute node to load the entire portfolio from the database. Several improvements of this distribution algorithm were tried out to reduce the data transfer, for example by caching the Lx objects, but only with limited success [13].

Originally, the event set-based distribution algorithm was chosen to avoid partitioning the portfolio tree across different machines. Using full copies of the tree for a certain sub-set of events (which are uncoupled) on each node provides an embarrassingly parallel way to distribute the calculations. This prevents any problems with tree-wide couplings of objects that occur during the Rating process. However, as explained above, this approach is not scalable to larger numbers of nodes. Furthermore, there is hope that by changing the underlying risk processing model and eliminating the coupling or doing it in a different way, distribution can be performed by actually splitting the tree itself. This reduces the database load by allowing the nodes to only be concerned with the Lx objects they are processing, rather than all Lx objects in the portfolio. Very large portfolios that do not successfully run due to memory issues could be split into many smaller partial portfolios. These could then be run one at a time, either in parallel or serially. In theory, there would be no limit to the size of portfolio that could be processed, as long as its L2s can be split into sufficient pieces.

In the pervious chapter, we could show that a combination of the Import and Encode processes considerably reduces the number of database rows affected. The reason for this is that one back-and-forth transfer of portfolio data between application and database is eliminated. This principle can be extended to also include the Rate part, resulting in the portfolio data being transferred from the application to the database only once. The ImportCSV and Encode processes are distributed along with the Rate process. Database queries are kept to a minimum, since the Lx objects are read from local files as far as possible and only processed and transferred to the database once. A simpler multithreading architecture can be used, since threads do not need to wait on calls to the database to get the Lx objects. A distributed version of such a fully combined algorithm, with cutting of the tree at the L2 level, is one of many alternative approaches to NC distribution that one could envision to be feasible once the issue with tree-wide coupling is solved.

4.2 Approach and Implementation

4.2.1 Underlying Distribution Implementation

The implementation of the distributed ImportCSV/Encode/Rate algorithm presented here is based upon the modified version of the original Swiss Re NC scheduler. This modified scheduler provides a means of distribution that is easy to work with and independent from the underlying grid middleware.

The modified scheduler roughly works as follows: First, the “pre process” creates the input data for the jobs that will be run on the compute nodes. Next the scheduler sets up the corresponding partial jobs and submits them to the grid middleware, which

subsequently runs the jobs on the grid. A "partial process" executes on each assigned compute node, using the job input data to produce a partial result output that is sent to the database. After all partial processes have successfully completed, a final "post process" runs, which combines the output data of all partial processes into a final result. The specific implementations of the individual processes for the ImportCSV/Encode/Rate algorithm are outlined below.

4.2.2 Pre Process (ImportCSV)

The pre process is called only once, usually on the client node. It is used to create the partial jobs that are sent to the compute nodes. A single portfolio is split at the L2 level into the appropriate number of partial jobs, which currently is given by a user configuration parameter in the database. The L3 level is recreated for each new partial portfolio with correct Lx object counts.

There are a number of ways to decide how many and which L2s should be assigned to a partial portfolio. The goal is to put quantities and sizes of L2s into each partial portfolio that will result in equal processing times, which optimizes the total runtime for parallel execution. The current code uses the L0 count at the L2 level to assign L2s to different partial portfolios. This is expected to result in each partial portfolio having roughly equal L0 counts.

4.2.3 Partial Process (PartialImportCSV)

The partial process is executed on each of the compute nodes that the partial jobs are assigned to by the manager node load balancing. The partial processes cannot communicate between themselves. Each compute node receives a partial portfolio to process, which it computes for all affected events. When the job-specific L2 set has been processed, the loss results are sent to the database along with the L3 data generated.

4.2.4 Post Process (PostImportCSV)

The post process runs only once, usually on the client node, after all the partial processes have been completed. The loss results created by the partial processes are combined at this stage. Since each partial portfolio has its own L3, all of the L3s for each partial portfolio have to be combined as well. The combined loss result and L3 are saved into the database.

4.2.5 Details

ImportCSV

- Split the sorted ImportCSV portfolio file into a user-specified number of partial portfolios
- File is split at the L2 level
    - Read in an L2 and find out how many L0s it has
    - Continue to read in L2s until the L0 bucket count is reached
    - Save L2s to file
    - Repeat for each partial portfolio
- L3 is recreated with appropriate data and counts
- Send partial portfolios to the database

PartialImportCSV

- Get the partial input data
- Parse L1INFO.txt
- Start the processL2Data threads (see below)
- Wait for them to complete
- Send L3 and partial loss results to the database

- **ProcessL2Data thread starts**
  - Create a complete L2 Record
    - Read a single L2 from L2.txt
    - Read all of the L1s for this L2 from L1.txt
      - Read all of the L0s for a L1 from CONDITION.txt
      - Read the L1Info for a L1 from L1INFO.txt
  - Use the L2 Record to create the L2 object
  - Start the processL1Data threads (see below)
  - Wait for them to complete
  - Fill L2 with generated child totals
  - Complete the L2 processing
  - Insert L2 into database

- **ProcessL1Data thread starts**
  - Get X number of L1s and L1Info Records
  - Encode the L1Infos for those L1s
  - Encode L1s
  - Encode L0s
  - Insert L0s into the database
  - Rate the current set of L0s
  - Wait for all L0s below a L1 to be rated
  - Rate the L1
  - Insert L1
  - Repeat until all L1s for this L2 are processed

PostImportCSV

- Get the partial loss results
- Combine all partial loss results
- Update L3 with global portfolio counts data
- Insert combine loss result

4.3 Tests and Results

4.3.1 Description

The same five standard tests cases used previously were again tested using the new algorithm. To test node scalability, tests using one, two, four, and eight compute nodes (plus the client node) were undertaken. Perfect scalability would mean a linear decease of execution time as the number of nodes is increased.

A further test was conducted using two new “extremely large” test cases from Swiss Re to see if the code is able to run very big test cases. The original algorithm was unable to handle the extremely large test cases due to memory constraints. As the portfolios

increase in size, the application's memory requirements increase as well. Therefore system memory sets an upper limit on the portfolio size.

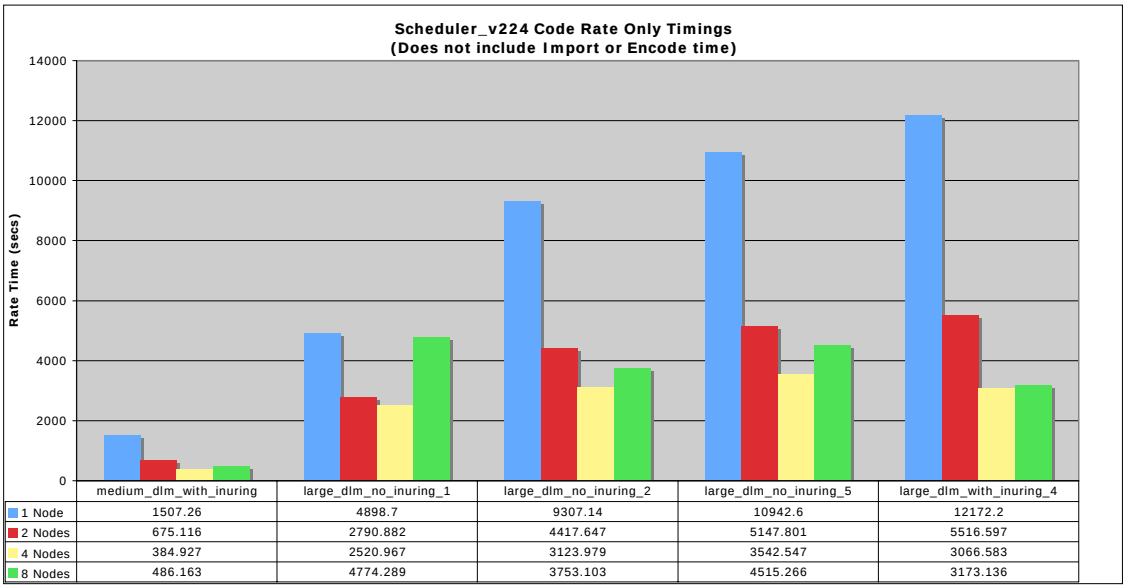
4.3.2 Execution Time and Scalability

During earlier project phases, the scalability of the event set-based distribution algorithm was looked at. The data for Figure 4 and Figure 5 comes from the corresponding tests that were previously run. These used the original event set-based algorithm, but within the same generalized implementation of distribution as used here (see Section 4.2). Only the Rate process was distributed, not the Import and Encode processes. Thus also the timings are only for the Rate process, whereas the timings of the combined ImportCSV/Encode/Rate algorithm in Figure 6 include the entire Import, Encode and Rate processes in one step, as described earlier in this chapter.

The combined ImportCSV/Encode/Rate distribution algorithm is faster than the corresponding event set-based distribution approach for almost all test cases and node counts. This can be attributed to the elimination of the need to get Lx objects from the database. For all test cases except large_dlm_no_inuring_1 and large_dlm_no_inuring_5, the execution time is about linearly reduced as the number of nodes that process the portfolio increases. This means that scalability is now much better than for the old distribution algorithm. Sometimes the performance gain is even higher than linear, potentially due to the associated higher number of parallel database connections for the in total same amount of portfolio data. For the two other mentioned test cases, the change between four and eight nodes is either not large or the timings get worse.

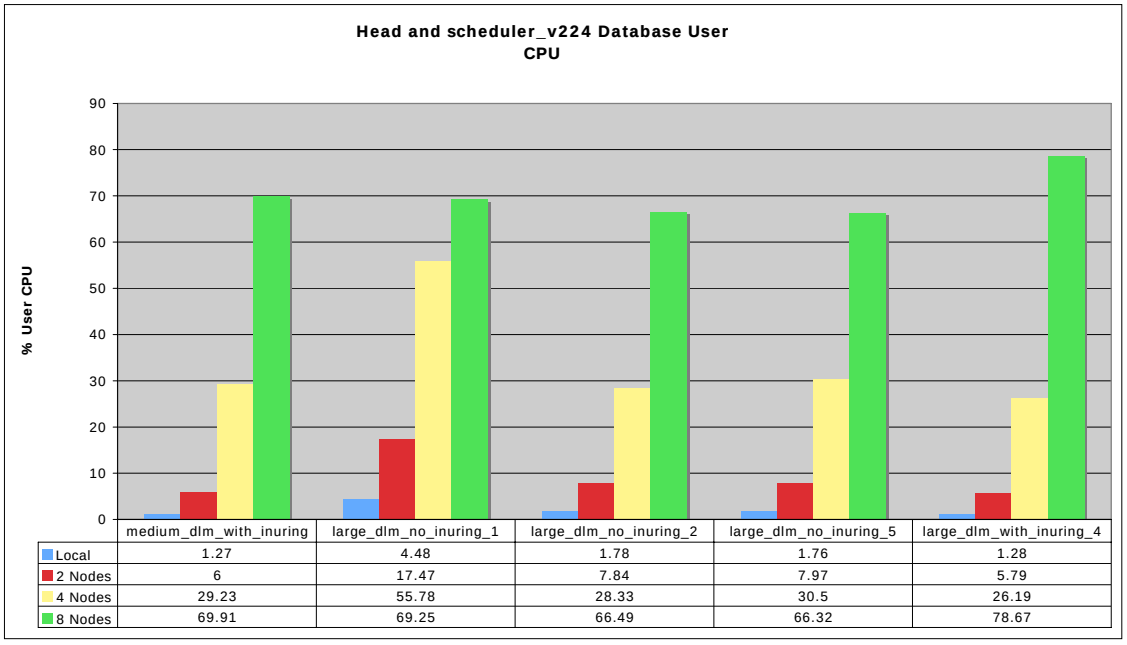
As can be seen from Figure 7, this behavior is – similar to the event set-based algorithm (Figure 5) – correlated with the load on the database. For most test cases, the user database load rises with the number of distributed nodes, but stays very low. However, although the portfolio tree processing is now distributed and the Lx data are read from local files as far as possible, the two exceptional test cases still have a considerably higher database load than all the others. It is clear that the database load reaching high percentages is preventing the large_dlm_no_inuring_5 test case from scaling to eight nodes. Most of this load results from SQL insert or select statements, which need to be optimized or eliminated to improve the scalability of the affected test cases. There are only a few SQL statements left, since event IDs and Lx objects are not retrieved from the database anymore.

One potential problem here might be duplicate processing. As was previously mentioned, the portfolio tree is not completely decoupled. Some objects, in particular the L1Infos, are shared across the whole portfolio tree, and will now need to be reprocessed on each compute node if required. The shared objects that are not portfolio dependent, such as the events and vulnerability curves, are cached to prevent having to reprocess them. However, caching will be less efficient since each compute node must process the first instance of a cached object. If that object instance occurs in multiple partial portfolios, more processing occurs than is necessary.



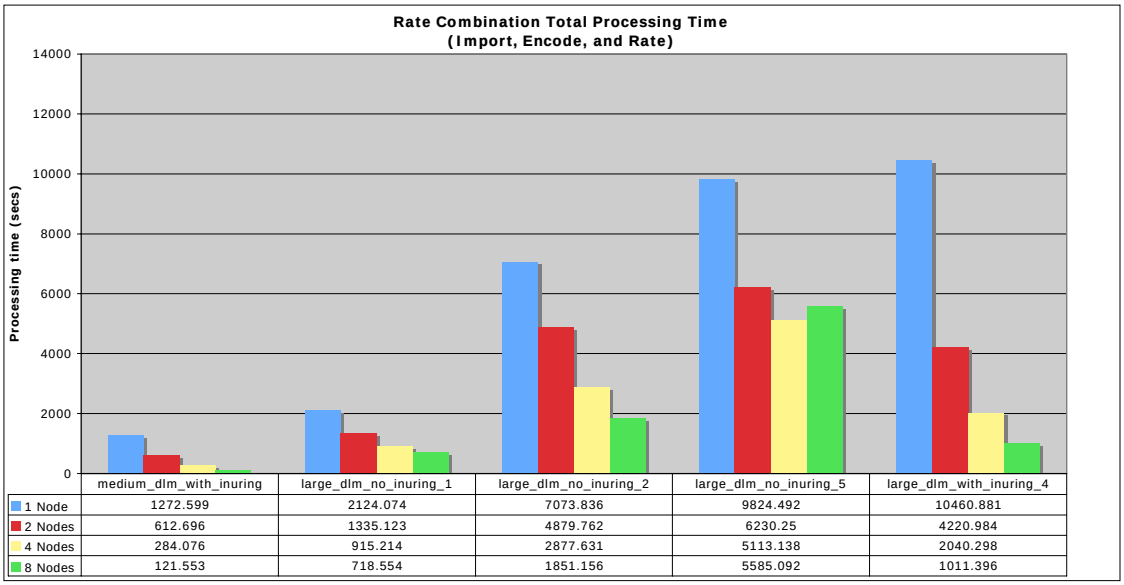
	medium_dlm_with_inuring	large_dlm_no_inuring_1	large_dlm_no_inuring_2	large_dlm_no_inuring_5	large_dlm_with_inuring_4
1 Node	1507.26	4898.7	9307.14	10942.6	12172.2
2 Nodes	675.116	2790.882	4417.647	5147.801	5516.597
4 Nodes	384.927	2520.967	3123.979	3542.547	3066.583
8 Nodes	486.163	4774.289	3753.103	4515.266	3173.136

Figure 4: Execution times for the event set-based distribution algorithm (excludes Import and Encode process times) running on different numbers of distributed nodes [13].



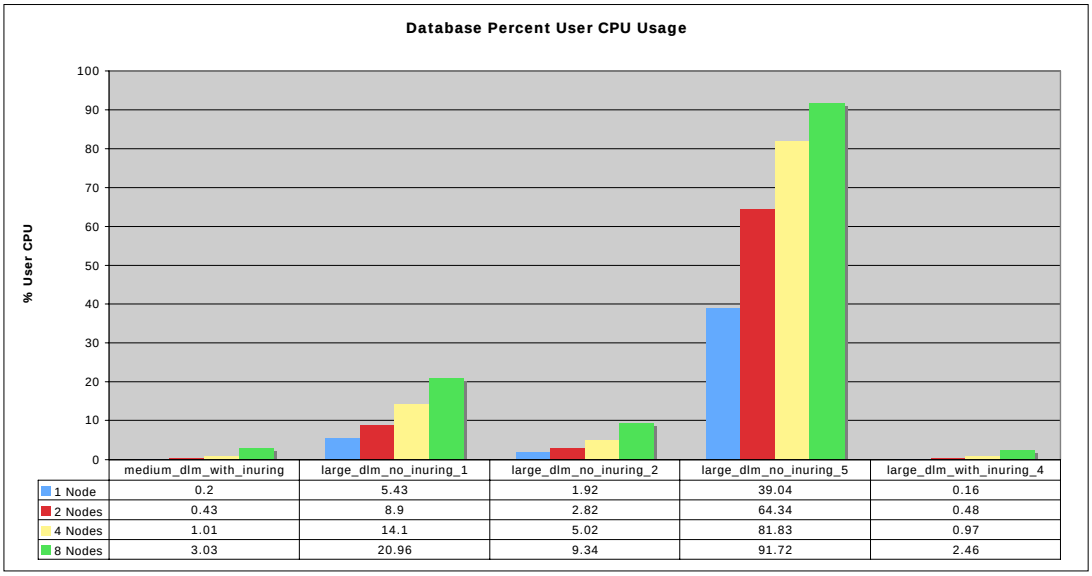
	medium_dlm_with_inuring	large_dlm_no_inuring_1	large_dlm_no_inuring_2	large_dlm_no_inuring_5	large_dlm_with_inuring_4
Local	1.27	4.48	1.78	1.76	1.28
2 Nodes	6	17.47	7.84	7.97	5.79
4 Nodes	29.23	55.78	28.33	30.5	26.19
8 Nodes	69.91	69.25	66.49	66.32	78.67

Figure 5: Database CPU user usage for the event set-based distribution algorithm (excludes Import and Encode process times) running on different numbers of distributed nodes [13].



	medium_dlm_with_inuring	large_dlm_no_inuring_1	large_dlm_no_inuring_2	large_dlm_no_inuring_5	large_dlm_with_inuring_4
1 Node	1272.599	2124.074	7073.836	9824.492	10460.881
2 Nodes	612.696	1335.123	4879.762	6230.25	4220.984
4 Nodes	284.076	915.214	2877.631	5113.138	2040.298
8 Nodes	121.553	718.554	1851.156	5585.092	1011.396

Figure 6: Execution times for the combined ImportCSV/Encode/Rate distribution algorithm (includes Import, Encode and Rate process times) running on different numbers of distributed nodes.



	medium_dlm_with_inuring	large_dlm_no_inuring_1	large_dlm_no_inuring_2	large_dlm_no_inuring_5	large_dlm_with_inuring_4
1 Node	0.2	5.43	1.92	39.04	0.16
2 Nodes	0.43	8.9	2.82	64.34	0.48
4 Nodes	1.01	14.1	5.02	81.83	0.97
8 Nodes	3.03	20.96	9.34	91.72	2.46

Figure 7: Database CPU user usage for the combined ImportCSV/Encode/Rate distribution algorithm running on different numbers of distributed nodes. Please note that the y-axis scale is different from Figure 5.

In the current testing scenario the parallel processing is most efficient if all partial processes run for the same amount of time, because the overall runtime is determined by the longest running partial job. Unfortunately, we later found a bug in the file splitting algorithm that allowed some partial portfolios to have a disproportional large share of L0s in some L2 sets. This bug is the more pronounced the more the file is split into pieces. Our analysis showed that during the large_dlm_no_inuring_1 eight-node test, one of the computation nodes only executed the partial process for 60 seconds, while the longest one took 718 seconds. The large difference between these execution times points to partial portfolios with very different numbers of L0s. This could explain

why the large_dlm_no_inuring_1 test case did not show a large improvement when the number of distributed nodes was increased from four to eight. This adds to this test case having the second highest database load, which could also have contributed to the result.

4.3.3 Extremely Large Test Cases

The goal of the following tests was to see if the new distribution algorithm could handle extremely large test cases. To check this, Swiss Re provided us the CSV_ER and CSV_LM test cases, which they were not able to successfully run in their own infrastructure. Three minor modifications to our NC code and one to the grid configuration had to be completed before these tests could be executed:

- A function that verified if a portfolio was too large for processing was eliminated.
- Modifications to a function that loaded a CSV archive into a byte array were needed to reduce the amount of memory required to load the file. The original code would cause the JVM to run out of memory.
- The combined ImportCSV/Encode/Rate algorithm requires that the CSV files are sorted. The normal Linux sort command used for other portfolios ran out of memory. The files were simply too large for that particular implementation of the sort command. The portfolios were subsequently sorted on a Macintosh iBook G4 laptop computer without incident. The sort command implemented there could handle the large files. The command to sort inside the application code was then disabled.
- The grid manager ran out of memory while distributing the extremely large portfolios across the grid. To fix the problem a few configuration parameters had to be changed. No major effects except slightly improved data transfer rates are expected from this.

Both test runs were conducted using eight nodes. Also eight partial portfolios were used for the CSV_LM test case, whereas the CSV_ER case was broken up into 16 partial portfolios. The reason for this is a memory issue with the file splitting routine. As programmed, it could not handle splitting the larger CSV_ER portfolio into eight pieces without running out of memory. As long as it is possible to split up a large portfolio into smaller partial ones, though, despite producing a bit overhead there is no theoretical limit as to the maximum size portfolio that can be processed.

The CSV_LM test case took 32,635 seconds to complete, whereas CSV_ER took 12,004 seconds. Since no reference values as to what constitutes a right answer were known for these files, we could not check if the result values are correct.

4.4 Conclusions

The ImportCSV, Encode and Rate processes have been combined and distributed on the L2 level. The results for scalability and extremely large test cases show that this new distribution algorithm is a considerable improvement over the previous event set-based approach. The main difference between the original algorithm and this one is the elimination of many database accesses, which previously made the database a bottleneck for the computations. This is achieved by eliminating back-and-forth data transfer, reading from local files where possible, and cutting of the portfolio tree between compute nodes (so that the total tree is processed only once overall). What the

Import, Encode and Rate processes do has not been modified, except of a few special business requirements that were ignored for now (see Section 4.1.3 for more details).

Continued focus on the database with the goal of further reducing its use is expected to give additional gains in execution time. Reducing how often the main execution path blocks on database access should result in a similar affect. Only inserting what is absolutely required for later reference into the database and handling the rest locally will assure a minimum number of generated inserts. For the inserts that are required, their execution should not block the main threads like they currently do. One could achieve this by one or more separate database access threads, for example.

How the partial portfolios are balanced is also an area that could further improve performance and scalability. In our test setup, the overall execution time is the result of the longest running partial job. If the partial portfolios are unbalanced in the length of time it takes to execute them, the overall timing may not look very good. It must be noted that it is the length of execution that has to be balanced, not the number of L2s per partial portfolio. What is needed is a good way to estimate the partial job runtime from the data contained in a portfolio. However, it should be pointed out here that this might not so much be an issue if more than one portfolio is computed at the same time (as it will be the case in a production scenario at Swiss Re). There the total computation time used added together over all nodes, not the real time, will be a better criterion for an efficient usage of the infrastructure. Nevertheless, perhaps an optimal combination of L2 sets can also improve the application of shared object, local caching, etc., and thus lead to better efficiency.

To see whether the ImportCSV/Encode/Rate algorithm scales above eight nodes, tests were conducted using twenty nodes. The file splitting bug, mentioned in Section 4.3.2, prevented some test cases from being properly split up. Sometimes the last partial portfolios were empty, due to an over-allocation of L2s to the preceding partial portfolios. The nodes added were also not identical to the current nodes in CPU speed or secondary cache and other system parameters. Because the longest running job is used as the timing measure at the moment, the slower computers could negate the gain in time attained through the additional nodes. These two issues were large enough to prevent the inclusion of the tests into this thesis.

5 Optimization of the Distributed ImportCSV/Encode/Rate Algorithm

5.1 Introduction

Eliminating the database as a bottleneck in processing large test cases was the reason for combing the previously separate Import, Encode and Rate processes. The initial algorithm and testing were described in Chapter 4. A bug was found in the file splitting algorithm (Section 4.3.2) as well as a potential portfolio tree coupling. That coupling might explain why some of the test cases increased the database load, while others did not. Those issues were further investigated, and the results are detailed below.

The ImportCSV/Encode/Rate distribution algorithm cannot handle any interdependencies at or below the L2 level. This distribution algorithm splits the Lx tree structure at the L2 level, and thus each L2 is assumed to be independently processable. In the original code, there are two places where that assumption is not met: Firstly, insurance conditions can run across L1 and L2 objects. Secondly, L1Info objects can be assigned to more than one L1 object, and those L1s can be assigned to different L2s.

Swiss Re has developed a way to decouple the insurance conditions between separate L2 and L1 objects, eliminating the main obstacle to using this distribution approach in their production environment. The second issue, L1Infos spanning L2s, has a number of possible remedies.

The solution to the L1Info spanning issue is dependent on how the data will ultimately be distributed to the nodes (see also Section 5.4). Currently, the new algorithm sends portfolio data to each node using a CSV zip archive. Other methods of data distribution can be envisioned that would change the dynamics of the problem, or eliminate it altogether. A global cache where all the Lx objects are stored would dictate a different solution to the L1Info spanning issue than the current CSV file passing. Global caching had been tried before, albeit with only limited success [13]. At this time the future direction of portfolio distribution is not known. Therefore, it was decided to work on reducing the time and database CPU usage associated with encoding an L1Info.

5.2 Approach and Implementation

The queries associated with L1Info processing were looked at, and most had a low cost associated with them as assigned by Oracle. There was one with a high cost, which is detailed in Table 10. By removing the “UPPER” statements and eliminating some redundant “WHERE” conditions the cost was greatly reduced. A few new table indexes were also created. However, the new indexes had no effect on the test results. The index changes and results will therefore not be reported here.

The first release of the splitting algorithm had a number of bugs that are addressed by a new L2-L0 splitting algorithm. The new algorithm correctly distributes the L2s among the nodes based on the number of L0s present. As will be seen, although the algorithm eliminates bugs found in the original L2 splitting algorithm, the test runs were sometimes slower. That was due to the algorithm enhancing the L1Info spanning issue.

Table 10: Original and new SQL statements used to select data from the GIS T_ZIP table.

Original SQL (Cost: 104)	SELECT LOCID_ZIP, NAME_E, NAME_N, DOCUMENTATION, LATITUDE, LONGITUDE, ZIP, LOCID_ADMIN0, ZIP_NAME FROM GIS.T_ZIP WHERE ISCURRENT=1 AND LOCID_ADMIN0=339 AND (UPPER(NAME_N)='07083' OR UPPER(NAME_E)='07083' OR LOCID_ZIP=7083 OR UPPER(ZIP)='07083')
New SQL (Cost: 3.416)	SELECT LOCID_ZIP, NAME_E, NAME_N, DOCUMENTATION, LATITUDE, LONGITUDE, ZIP, LOCID_ADMIN0, ZIP_NAME FROM GIS.T_ZIP WHERE ISCURRENT=1 AND LOCID_ADMIN0=339 AND (LOCID_ZIP= 07083 OR ZIP='07083')

5.3 Tests and Results

The optimization changes thus include the new L1Info query, the L2-L0 balance splitting algorithm, two new indexes, and an upgraded database. As mentioned previously, the indexes and database upgrade did not increase the execution speed. Therefore, they will not be separately addressed here. To isolate the effects of the new L2-L0 splitting algorithm, as example the large_dlm_no_inuring_5 test case was run with and without the new GIS SQL code (see Table 11). The test case has 95,826 more L1s than L1Infos. As can bee seen from Table 11, the new GIS SQL improves the performance greatly. It covers up the increase in execution time due to the new L2-L0 splitting algorithm. With the original algorithm, running on eight nodes the large_dlm_no_inuring_5 test case had an execution time of 5,585 seconds (see Figure 8). Without the GIS SQL change, the new L2-L0 splitting algorithm increased the time to 8,892 seconds (see Table 11).

The new L2-L0 splitting algorithm changes the ordering of all the L2s, and does not simply split the L2s up into the different partial portfolios as the pervious version did. This reordering increases the chance that L1Infos spanning multiple L2s will be assigned to different nodes for processing. There appears to be some L2 ordering in the ImportCSV file that places L2s that use the same L1Infos close to each other. The new algorithm also takes slightly longer to execute than the original. That is because of the temporary files that need to be created to hold the data before it is all combined into one partial portfolio. These temporary files are compressed to save space.

For the large_dlm_no_inuring_5 test case, the GIS SQL change not only decreased execution time, but decreased the database load as well. The database CPU usage of the new code in Figure 11 is only 12.68%, compared with 39.04% for the original combination algorithm in Figure 10. These numbers are for a single node, so the problems associated with spanning L1Infos are no issue. The improved L1Info query is responsible for the reduced database usage. This test case must rely heavily on this specific query to have gained so much from the change. The large_dlm_no_inuring_2 test case showed an increase in database CPU usage for all node counts. That test case has 115,775 more L1s than L1Infos. For this case, the L1Info spanning issue increased the database CPU usage more than the improved SQL decreased it.

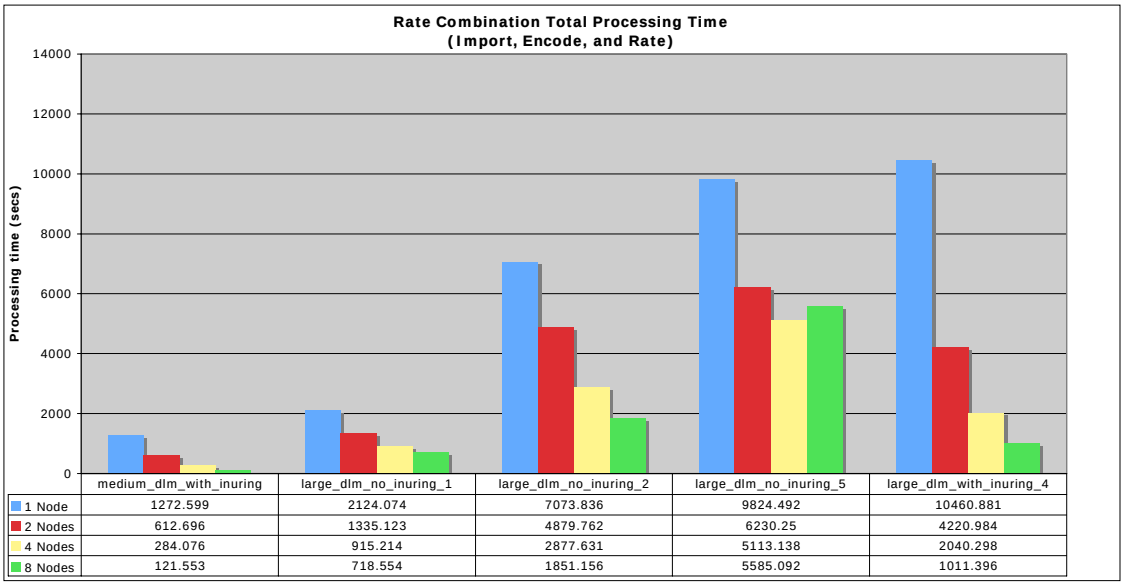


Figure 8: Total processing time of the original ImportCSV/Encode/Rate combination code (repeat of Figure 4).

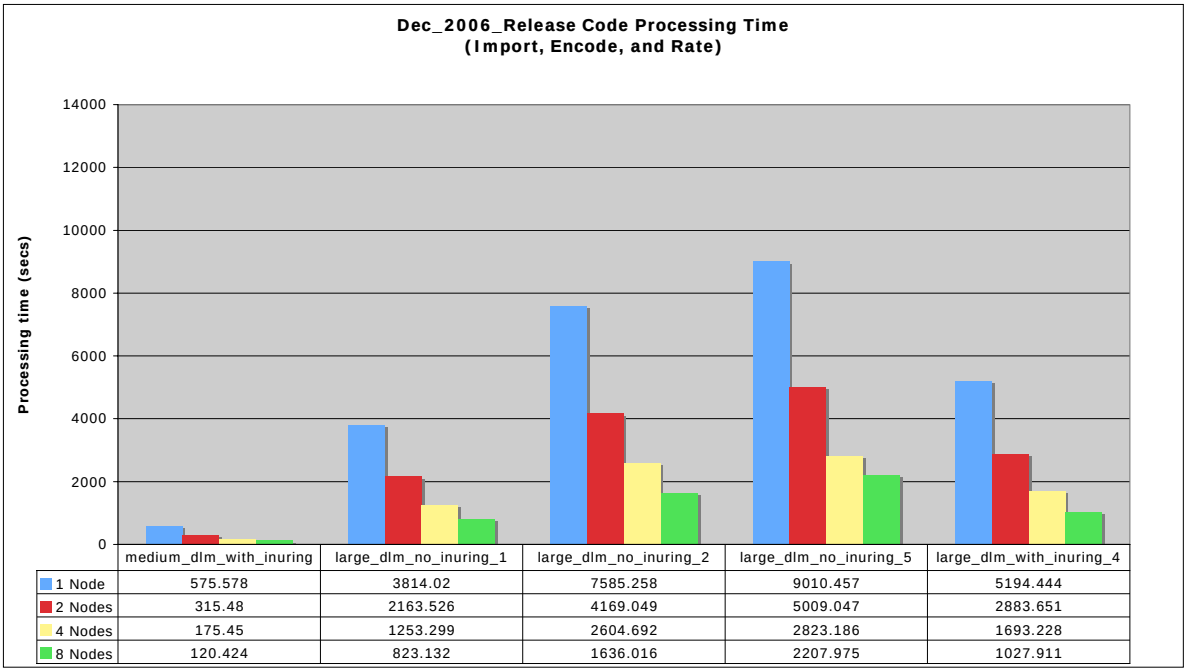
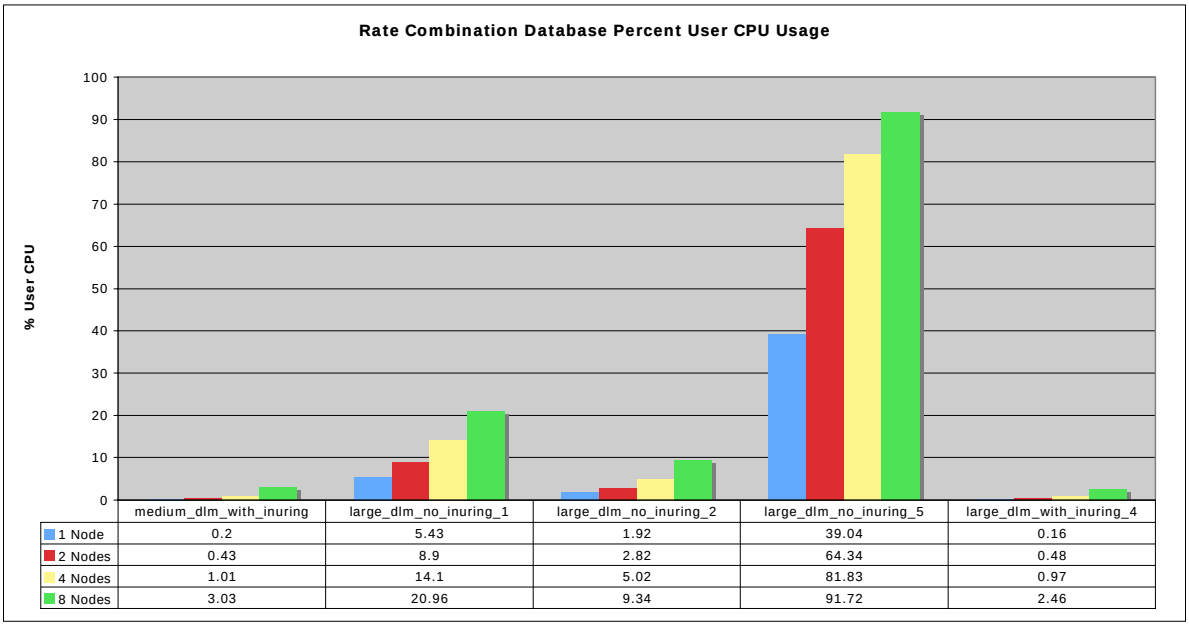
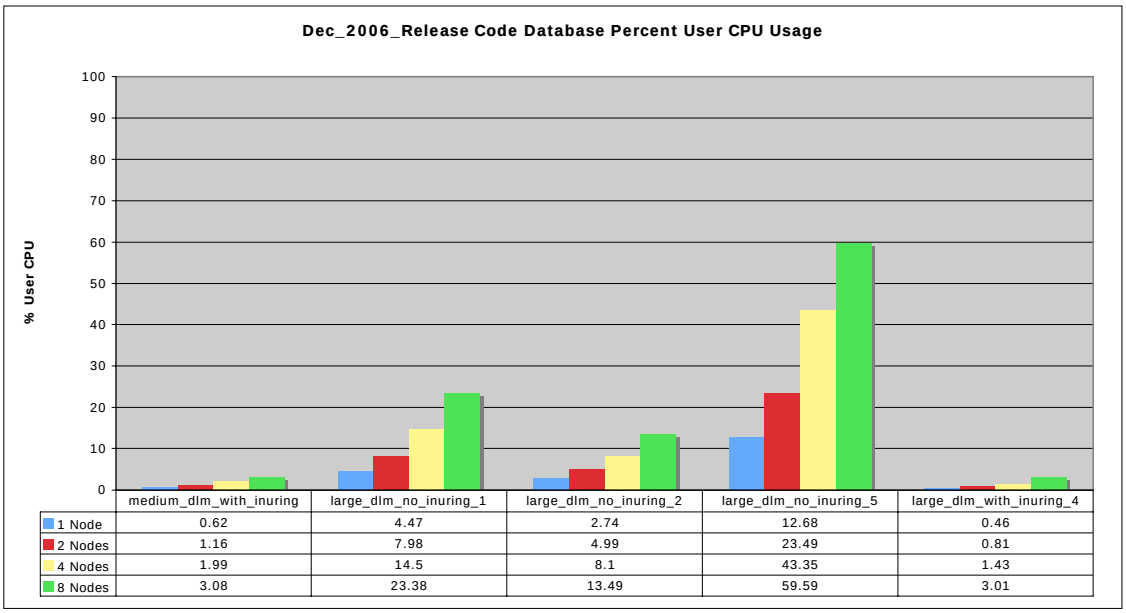


Figure 9: Total processing time of the optimized ImportCSV/Encode/Rate code (version Dec_2006_Release).



	medium_dlm_with_inuring	large_dlm_no_inuring_1	large_dlm_no_inuring_2	large_dlm_no_inuring_5	large_dlm_with_inuring_4
1 Node	0.2	5.43	1.92	39.04	0.16
2 Nodes	0.43	8.9	2.82	64.34	0.48
4 Nodes	1.01	14.1	5.02	81.83	0.97
8 Nodes	3.03	20.96	9.34	91.72	2.46

Figure 10: Percent database CPU usage of the original ImportCSV/Encode/Rate combination code (repeat of Figure 7).



	medium_dlm_with_inuring	large_dlm_no_inuring_1	large_dlm_no_inuring_2	large_dlm_no_inuring_5	large_dlm_with_inuring_4
1 Node	0.62	4.47	2.74	12.68	0.46
2 Nodes	1.16	7.98	4.99	23.49	0.81
4 Nodes	1.99	14.5	8.1	43.35	1.43
8 Nodes	3.08	23.38	13.49	59.59	3.01

Figure 11: Percent database CPU usage of the optimized ImportCSV/Encode/Rate code (version Dec_2006_Release).

Table 11: Comparison between the large_dlm_no_inuring_5 test case with and without new L1Info SQL.

Job ID	Test Case Name	Nodes	Description	Total Execution Time in sec	Average database CPU usage
3077	large_dlm_no_inuring_5	8	Old L1Info SQL	8,892.48	94.08 %
3072	large_dlm_no_inuring_5	8	New L1Info SQL	2,283.81	59.59 %
Difference between test cases				-6,608.67	-34.49 %

5.4 Conclusions

The new L2-L0 splitting algorithm solved the bugs in the previous algorithm. However, the issue with L1Infos spanning L2s on different nodes is now worse. An algorithm that packs L2s in such a way as to limit L1Info spanning would improve the situation. Using a Monte Carlo algorithm [13], for example, would be one approach. It would shuffle the L2s among the baskets looking for a combination where few if any L1Infos in a particular basket are also in another basket.

Unlike the L2-L0 splitting algorithm, the L1Info SQL change had a large, positive effect on the performance of some test cases. Once again it is clear that anytime database usage is reduced, execution time decreases. Now that the Lx objects are not transferred between the database server and the computation nodes anymore, the L1Info encoding process is becoming a bottleneck.

In the present version, the combined Import/Encode/Rate distribution algorithm is only working for CSV format input files. Also, file sorting is still done using a Unix shell script solution. To be applicable in a production scenario at Swiss Re, both issues need to be solved in a more general way that allows to import and process all possible types of input formats (CSV files, Excel files) via a convenient common intermediate format and/or class. This question is very closely related to the approach in which format the data will actually be distributed to the compute nodes, as this currently employs sorted partial CSV files, which need to be created first.

There are a number of different ways to tackle this problem. The following list gives an overview regarding some possible approaches:

- Files: One could develop an extension of the current CSV format that includes any extra Excel file features. The original CSV files would then be a special case of the new file format. Or one could invent a new, for example Extensible Markup Language (XML) based, file format, which would probably be quite extended, though. Files could either be split and/or directly sent to each node, or accessed via a global file system, which might limit performance.
- Distributed caching: For this, a number of different Java implementations (such as a shared file system cache) are available. Given the experience of other University team members, performance might be an issue here, though [13].

- Local databases: Instead of accessing the remote Oracle database, one could store and access items to and from a lightweight database on each compute node. Several potentially suitable Java-based database systems (such as Sun Java DB, Apache Derby, etc.) exist, or one could use a standard database implementation (such as MySQL, PostgreSQL, etc.). These could be ran either embedded with the corresponding dynamic database files consisting of the relevant portfolio sub-trees being sent around, or in client-server mode where the client fills the local databases on each node.

The exploration of the requirements, suitability, performance, and scalability of these different possibilities and the implementation of a solution that provides both a common Import hook as well as a consistent data distribution solution is a very worthwhile topic for further research.

6 Removal of Duplicated Data from Selected Database Tables

6.1 Introduction

Through separate work in the Swiss Re-University collaboration project, it was discovered that a number of database tables storing portfolio data were not normalized [13]. Normalized tables do not include any duplicate rows. It is standard practice to have a normalized table structure in relational databases such as Oracle. However, in this case it turned out that a partly un-normalized database structure is present. The reason for this is the way data processing and storing is performed within the NC application.

There are a number of reasons why minimizing the number of rows containing the same data through normalization is beneficial. Smaller tables require less storage space, are faster to search, and ease maintenance. As has been stated in pervious chapters, the database is a critical bottleneck. Any reduction in database performance could adversely affect the NC application performance and scalability.

This chapter details the results of the removal of duplicate rows from the following tables in the USER table space: T_Ded, T_Place, T_Haz, T_Limit, T_Peril, T_Modifier, and T_Exact_Place. Those tables were chosen because of their high rate of duplicated rows. Three scripts were created to normalize the selected tables; duplReplace.sql, duplTestReplace.sql, and duplDelete.sql. They are described below.

It should be noted that the version of the NC application used in the chapter is considerably different from the previously applied versions. The base version of the NC code was used, which does not contain any of the modifications detailed in the pervious chapters. In addition, there had been extensive changes performed to this base version of code. In particular, it now uses IBM WebSphere Application Server rather than DataSynapse GridServer as its distribution and execution environment [4,30]. The performed changes will not be detailed here, since they do not impact the results of the tests described here.

6.2 Approach and Implementation

6.2.1 Overview

To achieve the removal of duplicate data in the selected database tables, three SQL scripts were developed. The duplReplace.sql script was written in PL/SQL to allow conditional statements and looping. The other two scripts were written in standard SQL. There may be more efficient ways to realize these scripts. However, because the length of time needed to execute the scripts was not a determining factor here, the simplest algorithm was implemented.

The algorithms are described below without any reference to the tables that were updated. The algorithms to update each table are all similar. However, there is a separate script for each table that is updated due to differences in table column names. For the algorithm descriptions below, the generic tables are referred to as either a “duplicate table” or a “reference table”. A duplicate table is a table with duplicate rows, and a reference table is a table that references the duplicate table.

6.2.2 Script duplReplace.sql

Before the script is executed, a one-to-one relationship exists between the duplicate table and its referencing table. This script creates a one-to-many relationship between the tables, resulting in a normalized table. This is done through successively comparing two rows in the duplicate table. Since the rows are sorted first, all duplicate rows will be adjacent to each other. If two adjacent rows have equal column contents, then the foreign ID in the reference table that points to the second row is replaced with the ID from the first row. Now both rows in the reference table point to the same duplicate table row.

1. Declare variables to hold row contents of duplicate table
2. Declare second set of variables to hold row contents of duplicate table
3. Declare variable to hold foreign reference ID used in reference table
4. Declare a curser that is the result of a join between the duplicate table and reference table. Sort on all columns in the duplicate table excluding the primary key.
5. Replace the reference table's foreign ID if two rows in the duplicate table are identical
    a. Open the curser
    b. Populate the first set of variables with a row from the curser
    c. Populate the second set of variables with a row from the curser
    d. Compare the first and second rows
    e. If they match, replace the second row's referencing table's foreign ID with the first row's ID
    f. Stop if curser contains no more rows
    g. Otherwise repeat steps b to g
6. Close curser
7. Commit changes

6.2.3 Script duplTestReplace.sql

This script is used to find out how many rows the reference table now references in the duplicate table. It must be run before the duplDelete.sql script to give meaningful results. If ran after the duplDelete.sql script, there would be no difference between the unique rows and number of rows in the duplicate table because all none-unique rows would have been deleted.

Before the duplReplace.sql script was run, all rows in the reference table pointed to a different row in the duplicate table. By comparing the total number of rows in the reference table to the number of duplicate table rows now referenced, the number of unique duplicate table rows now referenced can be calculated.

1. Count the duplicate table foreign IDs in the referencing table
2. Count the unique duplicate table foreign IDs in the referencing table
3. Subtract the unique foreign ID count from the none-unique foreign ID count

6.2.4 Script duplDelete.sql

This script deletes all rows from the duplicate table that are not referenced by the reference table. If the duplReplace.sql script has not been run yet, all of the rows in the duplicate table will be referenced and no rows will be deleted. This script will free up the space used by the duplicated rows.

1. Create a temporary table USER.T_MJM_TEMP
2. Populate it with all of the duplicate table rows that are referenced by the reference table
3. Truncate and drop storage of the duplicate table
4. Insert everything from T_MJM_TEMP into the now empty duplicate table
5. Drop T_MJM_TEMP

6.3 Tests and Results

Oracle SQL Developer was used to execute the scripts against a specially created Oracle database instance. The timings for the Rate process were run on a development Linux box, not on any of the cluster nodes as before. After each set of scripts was executed, the Rate process was rerun to verify that the scripts did not alter the results, and to measure the Rate timings. In no case any of the results returned by the Rate Process were found to be incorrect, that is, they were not different from the previous run by more than the normal tiny numerical fluctuations, if at all. All caches were filled prior to running test timings.

The number and type of portfolios present in the database will alter the table space size and duplicate row count results. Therefore, the number of portfolios and how many times they were imported into the database are listed in Table 12.

Table 12: Number and types of portfolios present in the database.

Test case name	Number of times portfolio present in database
small_alm_no_inuring	1
small_alm_with_inuring	1
small_dlm_no_inuring	1
small_dlm_with_inuring	1
medium_alm_no_inuring	1
medium_alm_with_inuring	1
medium_dlm_with_inuring_3	2
large_dlm_no_inuring_1	1
large_dlm_no_inuring_2	1
large_dlm_no_inuring_5	1
large_dlm_with_inruing_4	1

It had been purposed that reducing the number of foreign key relationships in the database could increase the speed of queries executed against those tables. Tables that reference one of the following tables – T_Ded, T_Place, T_Haz, T_Limit, T_Peril, T_Modifier, and T_Exact_Place – had their foreign key relationship altered so that they all referenced only unique rows in one of the listed tables. The time required to execute the Rate process before and after executing the duplReplace.sql script was noted.

Normalizing the relationships between the tables without actually deleting the duplicate rows already affected the times it took to execute the Rate process to some extend, as shown in Table 13. As can be seen, the difference between the Rate times before and after the table normalization was small and not in a uniform direction for the small and medium cases. However, the large test cases all ran faster by up to 6.7%, which represents a first promising result.

Table 13: Difference in Rate process times before and after normalization of tables with duplicated rows.

Test case name	Before script time (seconds)	After script time (seconds)	Difference (seconds)
small_alm_no_inuring	15.18	21.581	+6.401
small_alm_with_inuring	10.133	10.077	-0.056
small_dlm_no_inuring	40.661	36.203	-4.458
small_dlm_with_inuring	35.892	41.224	+5.332
medium_alm_no_inuring	614.348	604.219	-10.129
medium_alm_with_inuring	587.385	598.764	+11.379
medium_dlm_with_inuring_3	325.236	332.271	+7.035
large_dlm_no_inuring_1	4120.069	4014.766	-105.303
large_dlm_no_inuring_2	5576.339	5204.781	-371.558
large_dlm_no_inuring_5	6450.430	6043.291	-407.139
large_dlm_with_inruing_4	2301.349	2256.053	-45.296

Furthermore, the duplReplace.sql script eliminated the need for a large number of rows, as can bee seen from Table 14. These numbers were obtained by the duplTestReplace.sql script. Over three million non-unique rows can be removed from the T_Ded table, which is referenced by T_Con. The duplDelete.sql script then actually deleted the rows that are not needed anymore. By deleting the unneeded rows, the USER table space size was reduced from 3’192 MB to 2’600 MB, corresponding to a saving of 592 MB or 18.5%.

Table 14: Difference in unique row counts before and after normalization of tables with duplicated rows.

Reference table	Duplicate table	Referencing Column	Before script rows	After script rows	Difference
T_Con	T_Limit	Limit	3347974	12988	-3334986
T_Con	T_Ded	Deductible	3347974	3042	-3344932
T_L1Info	T_Place	Quality	852823	847205	-5618
T_L1Info	T_Place	Admin0	842118	48	-842070
T_L1Info	T_Place	Admin1	190945	116	-190829
T_L1Info	T_Place	Admin2	209659	34842	-174817
T_L1Info	T_Place	Place	180151	26255	-153896
T_L1Info	T_Place	Zip	130323	19174	-111149
T_L1Info	T_Exact_Place	Exact_Position	750277	359824	-390453
T_L1	T_Haz	Hazard	1141247	484664	-656583
T_L1	T_Modifier	Second_Modifier	1141247	491710	-649537
T_L2	T_Peril	Peril	52849	6	-52843

We also measured the Rate times again after executing the duplDelete.sql script and thus deleting all unnecessary rows. This run was executed two times, to get an idea about the statistical fluctuations of the results. The measured timings are listed in Table 15. Deleting the duplicate rows from the database deceased the execution time of the large test cases by roughly another 100 seconds compared to simply normalizing the tables. In total, the Rate times are speeded up by up to 9.3% for the large test cases. Table 15 also shows that the time savings of the large test cases due to the removal of duplicate data are clearly beyond the expected range of statistical fluctuations between different test case runs.

Table 15: Rate process times for two runs after the deletion of duplicated rows, and difference to times before normalization and deletion.

Test case name	First run time (seconds)	Difference (seconds)	Second run time (seconds)	Difference (seconds)
small_alm_no_inuring	25.211	+10.031	15.429	+0.249
small_alm_with_inuring	15.367	+5.234	10.1	-0.033
small_dlm_no_inuring	46.195	+5.534	31.324	-9.337
small_dlm_with_inuring	45.904	+10.012	41.175	+5.283
medium_alm_no_inuring	616.762	+2.414	566.6	-47.748
medium_alm_with_inuring	593.457	+6.072	598.35	+10.965
medium_dlm_with_inuring_3	326.142	+0.906	331.616	+6.38
large_dlm_no_inuring_1	3915.026	-205.043	3920.124	-199.945
large_dlm_no_inuring_2	5104.605	-471.734	5148.021	-428.318
large_dlm_no_inuring_5	5849.114	-601.316	5977.858	-472.572
large_dlm_with_inruing_4	2188.079	-113.27	2207.778	-93.571

6.4 Conclusions

Normalization and deletion of duplicate rows in selected tables of the NC database led to slight reductions in Rate times for the large test cases of up to 9.3%. The small and medium test cases behaved less uniform, probably due to the database not being such an important factor for their performance. Also, Oracle is heavily optimized to handle joins, thus reducing the number of foreign key relationships did not speed up the Rate process more significantly. Normalizing a larger number of tables, modifying the Java code of the NC application, not just the database, and running distributed jobs, which are more data-intensive, could result in larger benefits. However, running the created database scripts and then deleting the duplicated rows already considerably reduced the USER table space requirements by 18.5%. The scripts thus offer an easy way to reduce the hard drive space required to store the data.

The scripts removed the duplicate data after it had been inserted into the database. The total amount of data sent and retrieved from the database and kept in memory was not changed. The next step in eliminating duplicate data would be to alter the code in an effort to reduce the total amount of data transferred to and from the database and kept in memory. One way to do so would be to change the Import process so that it recognizes when two rows are identical, and does not insert the row a second time. Also, the object representation of the portfolio tree might be altered to not load and process data items redundantly. Such changes would require large, but straightforward modifications to the NC code. Database scripts like those tested here would not be needed anymore.

7 Extension of Model to Cover Biological Entities

7.1 Introduction

The previous chapters were predominately concerned with the computational aspects of the NC model. A solid grounding in model functionality, as detailed in Chapter 2, is a prerequisite for extending the model to cover other entities besides buildings. In our case, the interest is in extending the model to cover biological and environmental entities at risk from natural catastrophes.

There are many possibilities for such extensions, to name just a few: Floods damage crops; storm surges erode beaches; hurricanes damage wetlands; severe winds damage forests. The NC model framework imposes a number of requirements that a modeled entity must meet. Those requirements are nicely meet by trees severely damaged by wind, or "windthrow":

- The NC model is already used to calculate losses to buildings from windstorms. Peak gust speeds and sustained wind speed are taken into account [29].
- Entities must be independent. A windthrown tree or stand – depending on the resolution of the model – must not increase the likelihood that neighboring trees or stands will also be windthrown. Although this requirement is rarely meet in biological systems, windthrow models with entity independence can reasonably predict risk [24].
- Values predicted by applying event intensities to entities must be additive. The model calculates the predicted loss value for a large collection of individual entities. Those individual values are then added together to get a single number for the collection. This condition is valid both for houses and trees.
- The event intensities and predicted values must be on a similar scale. It makes no sense to calculate the risk of windthrow to a stand of trees of five square kilometers if the resolution of the event intensities is only 50 square kilometers. The local variations in event intensity will affect the stand of trees, but would not be captured in the model, probably resulting in a bad prediction. However, the subjects in both the reinsurance (buildings) as well as the windthrow (tree/stand) NC model are about the same order of magnitude in size.

Besides meeting the requirements of the current model, there are a number of reasons why a windthrow model would be interesting from both a reinsurance and scientific perspective:

- Quantifying the risk of windthrow is a prerequisite for insurability. Forest risk insurance is not common in Central Europe, which might be due to lack of reliable quantification of risks [11].
- The NC model would be a novel way to predict windthrow risk using methods not currently used. In high-risk areas, forests could be managed in a way to reduce the odds of windthrow.
- There has been some interest in looking at windthrow and its impacts on forest structure. Forests with high windthrow risk may have different forest structure than forests that house similar species and grow under otherwise similar conditions regarding climate, soil, etc., but with a low windthrow risk [14].

Wind can severely damage a tree in many ways. Our interest centers around two major forms (see Table 16): (1) The main tree trunk is broken at some point along its axis, or (2) the tree is uprooted. These two types of damage will lead to different amounts of salvageable wood, but for our current purpose will not be discussed separately, except where needed.

Table 16: Windthrow pictorial representations.

	
Wind Snap: Wind breaks main trunk of tree [21].	Uprooting: Root systems fails from excessive wind loading [18].

7.2 Model Building Steps

As detailed in Section 2.1, the NC model provides a framework that uses vulnerability curves and a set of storm events to predict long-term risk. Goal 2 of this thesis is to explore ways in which this model could be extended to cover forest windthrow damage. Since storm events are already included in the original model from Swiss Re, the main task here is thus to construct vulnerability curves for the windthrow of trees. However, depending on which wind properties turn out to be required for this, such a model extension might also potentially later have consequences for how storm events should best be described.

There are a number of steps involved in creating the vulnerability curves: First, a relationship between two variables needs to be defined, an explanatory variable defining an event intensity and a response variable defining the resulting damage. Of course, this is a simplification of reality, in that both multi-dimensional explanations as well as multi-dimensional consequences of forest wind damage can be expected.

Therefore, the main driving forces in both variables have to be identified:

- In our case, the model framework requires the explanatory variable to be a measure of wind strength. Typical descriptions of wind strength include the maximum, mean, and mode of the wind speeds for the duration of the storm. Indexes, such as the frequency of wind speed above a specified amount are sometimes used to quantify wind speed variation over time. There are many ways to create an index. It also has to be taken into account that wind has a directional component. Depending on tree growth and landscape topology, storms coming from different directions might thus have different consequences.
- Windthrow represents the response variable. Rather than trying to predict the risk to an individual tree, groups of similar trees, or forest stands will be the entity of interest here. Damage measures would then be expressed in percentage form. Some common estimates of stand damage include: percent of open canopy, percent of damaged trees, and percent of wood volume damaged.

Next, any variations of this basic relationship need to be defined. Each variation will lead to a new vulnerability curve. For example, different species of trees may vary in their susceptibility to wind damage. Each such tree species would then need a different vulnerability curve. Similarly, tree age, height, and diameter, stand elevation, structure, and exposure, tree density and mixture, as well as the soil type could also impact the relationship, requiring different vulnerability curves.

For a biological or ecological model of forest windthrow the information above would be sufficient. However, for the additional usage of the model for economic and insurance/reinsurance purposes, one needs to finally translate the physical damage of the stands to the corresponding financial losses. For this, a number of different factors would have to be considered: Windthrow wood valuation methods, market prices, estimations of extra work costs to remove the damaged trees, reforestation expenses, changes in long-term value of the forest due to windthrow, etc. The windthrow damage also has to be related to the insurance and reinsurance contracts, which will later serve as input for the NC model. Details on any deductibles, coverage, etc., for example, to which wood value or work costs they refer, need to be taken into account in order to estimate the insurance payouts. Depending on the insurance contracts, such losses might have to be split into different vulnerability curves. However, it should be noted that the insurance and reinsurance conditions themselves will be part of the portfolios, not of the vulnerability curves.

The following sections attempt to address the different questions that need to be resolved before a completed model can be created. The emphasis here will be on the initial establishment of the wind intensity – windthrow relationship, as this represents the core of the vulnerability curve construction. Shortly, the translation into financial losses as well as the usage of the model for other than reinsurance purposes will be explored.

7.3 Establishing the Wind Intensity - Windthrow Relationship

7.3.1 Using Empirical Models

First the vulnerability curve relating wind intensity to windthrow must be created. This can be done either with mechanistic models, physical models of wind-tree interaction, or through empirically gathering windthrow data and creating a statistical model. The latter method will be addressed in this section.

There are a number of empirical studies that relate stand variables to windthrow risk, resulting in a map of high and low risk stands. Normally some sort of stand wind exposure parameter is calculated. “Topex” is a popular one that measures whether a stand is in a position where surrounding hills and mountains may block the wind of a storm (sum of maximum angle to ground within X kilometers for each of the eight cardinal directions) [15]. Site-specific attributes such as slope, elevation, and soil type are also used. Moreover, stand specific attributes such as species composition, tree density, mean tree height and diameter are often applied.

Surprisingly, using only site elevation, slope, soil stability, and an index of storm exposure Kramer et al. were able to correctly identify, in 66%-72% of cases, areas where windthrow has and has not occurred for several Alaskan islands [14]. They used a logistic model testing for the presence or absence of windthrow. In this case the storm wind measure was based on the topography and direction of the prevailing storm winds. The authors applied a mathematical model to categorize different areas as to how protected or exposed they would be to severe winds from storms. Individual storms and their wind speeds were not related to windthrow. Instead, a general model predicting relative risk of windthrow for different areas was created.

Lanquaye-Opoku et al. were able to correctly predict in which 25m-by-25m forest segment windthrow would occur 67-83% of the time [15]. They selected three costal and three inland locations in British Columbia to model. To account for the wind exposure, the Topex measure previously discussed was utilized. The authors also took the sites’ mean annual wind speed into account. Although only windthrow presence/absence was modeled, and not amount of damage, the areas used are small enough to give a reasonable idea of the extent of windthrow damage for a given location.

Neither of these studies thus considered single instances of a windstorm, only aggregated wind related parameters were employed: The Lanquaye-Opoku et al. model used mean annual wind speed and Topex, whereas Kramer et al. used an exposure index to categorize areas as having higher or lower probability of high winds based on topology and prevailing storm direction.

Few empirical studies have been done that relate forest windthrow to a particular storm’s wind field. Instead, as was done above, general measures of storm exposure are used. Unfortunately, to create the vulnerability curves, we need a direct wind intensity – windthrow relationship. A unique study that tried, but failed, to create the relationship we need is detailed below.

7.3.2 Empirical Model Data Set Reanalysis

In December 1999 a large winter storm, Lothar, hit western Central Europe causing large amounts of windthrow. Schütz et al. used the opportunity to try and relate wind speed as captured by a Doppler radar station to a number of forest sites with varying amounts of windthrow for the northern part of Switzerland [22].

It should be noted that Doppler radar measures wind speeds at an increasing elevation as you move away from the radar station. Therefore, Schütz et al. had to extrapolate the wind speeds from the measured values to estimate wind speeds at different elevations. For the statistical analyses estimated wind speeds at 600m and 1000m was used.

They used stepwise multiple linear regression to relate a number of measured variables to a damage index. The damage index represents the amount of windthrow as measured by canopy gap size. The model created for pure Spruce stands had an adjusted

R^2=0.0668, n=93. None of the variables representing wind speeds were significant. While the model created for pure Beech stands had an adjusted R^2 = 0.273, n = 99. In the Beech model the maximum 600m wind speed measurement explained 27% of the variance.

Upon request, Schütz provided us the raw data used in the original paper to allow a further analysis to be done [23]. Throughout this section these raw data will be reevaluated in an effort to attribute why the model that was produced had such low explanatory power.

In the original paper, Schütz et al. concluded that the Lothar wind field was very chaotic both spatially and temporally. Good evidence comes from comparing wind speed as obtained from the Doppler radar to that from ground stations at the same locations. Figure 12 shows a comparison for a point in time between two nearby ground stations and what the Doppler radar reported. In both cases, wind speed as extrapolated from the Doppler radar measurements was inconsistent with wind speed measured by the two ground stations [23]. This calls into question how well Doppler radar wind speeds can be translated into actual ground wind speeds.

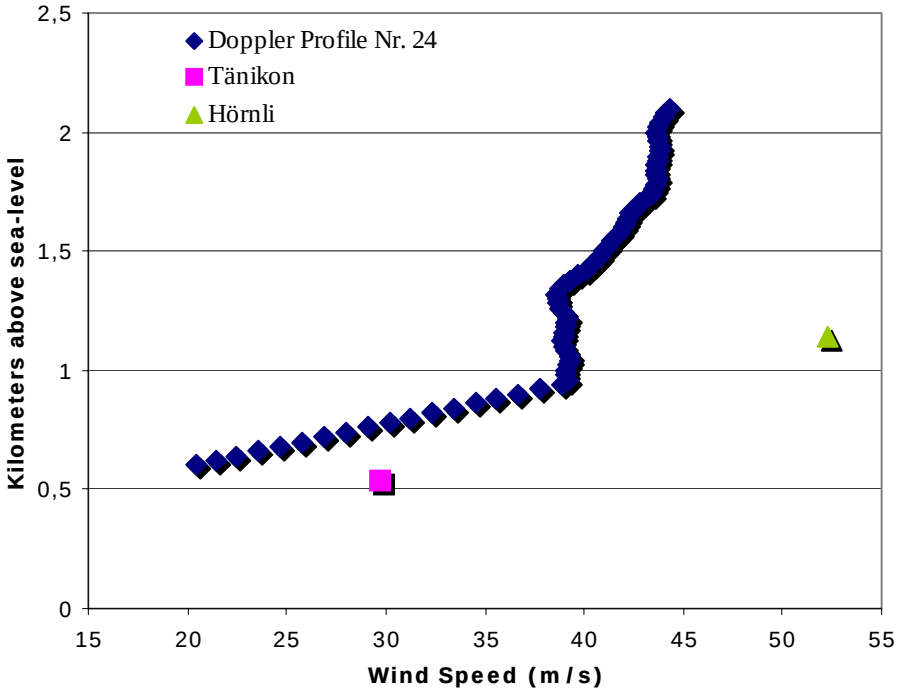


Figure 12: Ground truthing of Doppler radar wind speeds. Tänikon and Hörnli are grounds stations located closely together.

Another problem stems from the fact that wind speed and direction vary with time. Although a tree will experience all variations in the wind, current ground sensors and Doppler radar only periodically capture wind speed and direction. That allows aggregated values like maximum and mean wind speeds to be calculated. However, it might be just as important to know the frequency of wind gusts, speed and direction. It has been found that tree swaying can be an important component in windthrow [17]. If gust frequency matches tree sway, lower wind speeds can cause unexpectedly high damage.

To try to account for the gustiness of an area during the storm, and thus to try out a different measure for wind intensity, an integral of the wind speed for a particular location was calculated. Throughout the storm the radar reported wind speeds in regular intervals of every five minutes. By summing the speeds, an index was created that represents the overall amount of wind at a location during the storm. It appeared that Schütz et al. never tried such an approach. Rather, different wind measures such as maximum, mean, and frequency of large gusts were used. Replacing the wind speed parameter with the new wind integral index for the Beech model and adding it to the spruce model, did not improve either model (Beech adjusted R2=0.049 n=98 Table 19, Spruce adjusted R2=0.016 n=122, Table 20). It should be noted here that the regressions used for this reanalysis did not result in the same model values as reported in the original paper. The number of sample points was higher for the current reanalysis. It is not known what filtering mechanism the publication authors used to reduce the number of sample points. The extra data points did not change the general results of the regressions as reported in the original paper, however (Beech adjusted R^2=0.19 n=98 Table 17, Spruce adjusted R^2=0.024, n=122 Table 18).

Table 17: Reanalysis of original pure beech stand data using the same multiple regression model used in Schütz et al. [22].

Response: LIL, n:98, Residual standard error: 2.158 on 94 degrees of freedom Multiple R-Squared: 0.2194, Adjusted R-squared: 0.1945					
	Df	Sum Sq	Mean Sq	F value	Pr(>F)
V6	1	100.59	100.59	21.5932	1.096e-05
EGK	1	21.87	21.87	4.6943	0.03279
HD	1	0.62	0.62	0.1334	0.71579 n.s.
Residuals	94	437.91	4.66		
LIL: Log transformed damage index, V6: Maximum wind speed at 600m, EGK: number of years since last thinning, HD: tree height divided by width (slenderness)					

Table 18: Reanalysis of original pure spruce stand data using the same multiple regression model used in Schütz et al. [22].

Response: LIL, n:122, Residual standard error: 2.405 on 118 degrees of freedom Multiple R-Squared: 0.04855, Adjusted R-squared: 0.02436					
	Df	Sum Sq	Mean Sq	F value	Pr(>F)
EXKL	1	27.59	27.59	4.7688	0.03096
HO	1	5.01	5.01	0.8656	0.35408 n.s.
DGv	1	2.24	2.24	0.3869	0.53513 n.s.
Residuals	118	682.68	5.79		
LIL: Log transformed damage index, EXKL: aspect class, HO: Tree height, DGv: Tree cover before Lothar					

Table 19: Reanalysis of original pure beech stand data with V6 wind speed replaced by the wind speed integral. The rest of the model is the same as in Schütz et al. [22].

Response: LIL, n:98, Residual standard error: 2.346 on 94 degrees of freedom
Multiple R-Squared: 0.07815, Adjusted R-squared: 0.04873

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
WindIntegral	1	7.10	7.10	1.2906	0.25882 n.s.
EGK	1	36.32	36.32	6.6020	0.01176
HD	1	0.42	0.42	0.0764	0.78283 n.s.
Residuals	94	517.15	5.50		

LIL: Log transformed damage index, WindIntegral: Sum of 600m wind speeds at stand location, EGK: number of years since last thinning, HD: tree height divided by width (slenderness)

Table 20: Reanalysis of original pure spruce stand data with the wind speed integral added to the multiple regression model used in Schütz et al. [22].

Response: LIL, n:122, Residual standard error: 2.416 on 117 degrees of freedom
Multiple R-Squared: 0.04857, Adjusted R-squared: 0.01604

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
WindIntegral	1	0.002795	0.002795	0.0005	0.98258 n.s.
EXKL	1	27.60	27.60	4.7309	0.03164
HO	1	5.01	5.01	0.8594	0.35581 n.s.
DGv	1	2.23	2.23	0.3822	0.53765 n.s.
Residuals	117	682.67	5.83		

LIL: Log transformed damage index, WindIntegral: Sum of 600m wind speeds at stand location, EXKL: aspect class, HO: Tree height, DGv: Tree cover before Lothar

By looking at larger areas, local variance in the chaotic wind field may be evened out and more general patterns might emerge. In the same study, Schütz et al. analyzed a forest area of 65’000 ha [22]. They used satellite images to locate gaps in the forest canopy caused by the winter storm Lothar as it moved through Switzerland. Then the authors tried to find a relationship between the size of the gap and the wind speed measured at that place. A relationship between wind speed and canopy gap was not found, most likely due to the accuracy issues with the Doppler radar wind speed data explained above. However, a relationship between classes of aspect and slope gradients was detected. The classes might be analogous to storm exposure ratings used in the previously discussed studies, and could actually intrinsically represent the local modulations in wind field we were looking for. Or it might have to do with wind direction, which was not monitored in the study. Wind is a directional force, and therefore can be blocked by terrain features. Doppler radar is looking at the wind speed above the mountains, not the wind that actually impacts the trees. Coupling wind direction, with terrain features, might allow a better estimate of the tree level wind speeds.

It could be argued that canopy gap size, which the study used as a measure of windthrow severity, is not a good measure of windthrow damage. For a canopy gap to occur, a high percentage of trees in a small area must all be windthrow. The same

number of trees could fall, but over a larger area. Although the number of fallen trees is the same, dispersed windthrow will not show up as gaps in the canopy.

It is obvious that high winds are the cause of windthrow. Whether or not an empirical relation can be determined between wind intensity and windthrow, though, remains an area for further research. It would be possible to locate large numbers of ground wind sensors that log wind speed and direction every few seconds in forests. That would give a better idea of the local wind fields around the trees. However, it is very hard to predict which stands to place the sensors in as well as when a large enough storm will occur. It could be many years before a windthrow event happens in the stands where the sensors are located.

7.3.3 Using Mechanistic Models

Besides empirically relating wind and windthrow, mechanistic models have been developed. They can be used to directly calculate the wind speed at which a stand of trees will experience damage. Mechanistic models work with the physics of tree-wind behavior. Through modeling how a tree behaves for a specific wind speed, the corresponding stress forces can be calculated. These are then used to estimate the wind speed at which an individual tree is windthrown.

The parameters needed for the model can be put into three categories:

1. Wind parameters define how the wind behaves in and over the stand. They include items like a wind gust factor and surface roughness.
2. Site specific parameters like soil firmness specify the abiotic conditions of where the stand is located.
3. Lastly, tree species parameters specify things like how strong a tree’s trunk and roots are.

Figure 13 shows a graphical representation of the principles behind mechanistic models. A tree is analogous to a tapered beam (trunk) with a weight on top (canopy), Figure 13 (A). The canopy catches the wind and those forces are delivered to the trunk causing the trunk to bend, Figure 13 (B). The weight of the canopy as well as the force of the wind are used to calculate the stresses placed on the trunk and root system, Figure 13 (C). If the forces are too large, the trunk or root system fails, and the tree is windthrown. The point at which this occurs is called the “critical wind speed”.

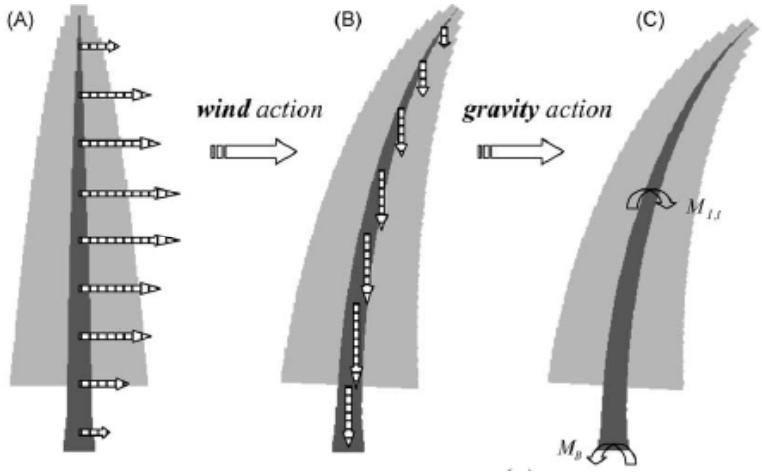


Figure 13: The basic principle in mechanistic modeling. Picture taken from Ancelin et al.[1].

The mechanistic models HWIND and GALES calculate the stand's mean tree's critical wind speed [9]. When the mean tree in a stand is windthrown, it is highly likely that there will be a good amount of wind damage to the whole stand. Similarly to the empirical models, the amount of windthrow is not calculated. The two mechanistic models are useful for relative windthrow risk rankings. To create a vulnerability curve, the full relation between wind speed and damage is needed.

Ancelin et al. created a mechanistic model, FOREOLE, to explain this relation [1]. Rather than calculating the critical wind speed for the mean tree, it is calculated for every tree in a stand. This approach follows the principles behind individual based models, or IBMs. By calculating all critical wind speeds, vulnerability curves relating wind speed to damage are produced. Figure 14 shows an output from the model. Seez, VRS, and VSS represent different stands of trees. The parameters for the Seez stand are directly taken from a real stand in the French Alps, whereas VRS and VSS are modifications of that stand. The mean tree's critical wind speed is also labeled and can be used to compare the model against GALES and HWIND. All three models produce similar mean tree critical wind speeds [1].

Mechanistic models are very attractive because they can be used to directly output vulnerability curves. Little model validation has been done, however. As was the case with the empirical models, similar problems will be encountered in trying to relate wind speed to stand damage. Such a relation is required to validate the mechanistic models. Even if a mechanistic model is completely correct, predicting a storm's local wind field at each stand becomes the next challenge. As can be seen from Figure 14, there is a large difference in the amount of damage that occurs at 15 m/s and 20 m/s. The typical maximum wind speeds measured by Schütz et al. during the Lothar storm (on average 25.6 m/s, adjusted to 600m elevation) [23] are high enough to cause a large amount of damage, Figure 14.

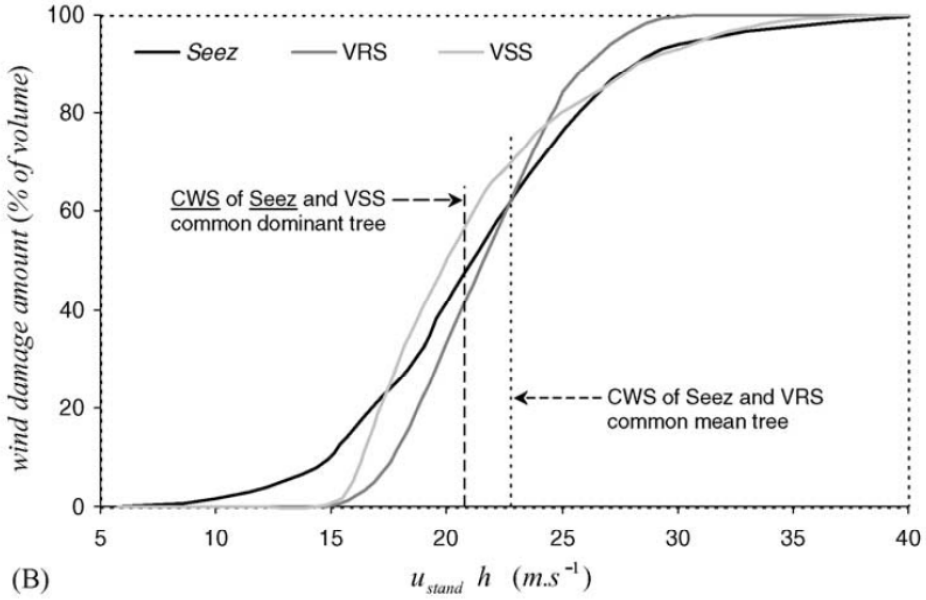


Figure 14: Wind speed in meters per second and subsequence percentage of wood damaged. Stand data for simulation taken from one real Norway spruce stand (Seez) located in the French Alps and two versions with modified trunk diameter distributions. Graph taken from Ancelin et al. [1].

7.4 Converting Windthrow Into Economic Loss

Calculating the risk of windthrow for a stand of trees is the first task. Once that is done, the amount of windthrow damage for a specific risk level is needed. The above sections discussed those two issues. This section is concerned with the last step, converting the windthrow damage into an economic cost.

The most obvious lead to follow here is the direct financial loss associated with the selling of wood. There is little literature comparing the value of a windthrown tree to an analogous standing one. Windthrow timber will be worth less depending on how badly damaged the wood is. There is also a limited time to salvage the damaged trees before the value of the wood is completely lost. The time limit can result in a market glut, and lower lumber prices following a catastrophe [20]. If there was a substantial loss in standing timber, an increase in lumber prices can follow the decrease [20]. Lumber is a commodity item, and therefore it closely follows market supply and demand. Ultimately, the individual insurance and reinsurance contracts will dictate the payout.

Unfortunately, with the limited amount of resources at hand for this thesis, the various economic aspects of forest windthrow could not be further explored. It can be guessed, though, that Swiss Re should have information about economic losses through windthrow from previous storm events, as well as details about corresponding insurance contracts available. These could then be used to relate windthrow damage to the resulting financial burdens.

7.5 Beyond Reinsurance Applications

7.5.1 Ecological Consequences

Besides application in the reinsurance industry, a windthrow model based on the NC framework could be used in studying stand composition. For example, Kramer et al.

created an empirical windthrow model for several southeast Alaskan islands [14]. They found that canopy gaps caused by large windstorms are an important process in driving forest development on the islands. Large tracks of forest can be windthrow, allowing light in the under story and a new generation of trees to germinate. Similar results may occur in other forests.

It can be expected that there should be differences between unmanaged stands where windthrow is a dominant structuring force, compared to those where it is not. Windthrow resistant tree specie communities would be selected for in very high-risk areas. Tree ages could also be different. The storm return interval would tend to put a limit on how old a stand of trees can get before the next catastrophic wind storm occurs.

This short list of examples already shows that windthrow might play an important role for the ecology of forests. A corresponding NC model could thus help to scientifically understand the measured consequences of previous storm events, as well as to predict the expected consequences of future storm events. Such a model would also allow to play with virtual “what if” scenarios, such as what the results would look like if one tree species is substituted by another one, or if storm frequency, intensity, and localization changes with global warming.

7.5.2 Forest Management

The empirical and mechanistic models discussed above are also often used by foresters to assess stand windthrow risk. The Forestry Commission of Great Britain has produced ForestGALES, which is based on the GALES windthrow mechanistic model [7]. It has a graphical user interface and a plug-in for the geographical information system software ArcGIS. Foresters in Britain use ForestGALES to estimate windthrow risk [10].

Once the windthrow risk is known, there are a number of strategies to reduce such risk, a few of which shall be mentioned here: Mixing stands of wind-firm species of trees with more vulnerable species [16], retaining at least 20% of original stand density when selectively harvesting [24], and preferentially retaining trees with low height-diameter ratios [24] are all ways to reduce windthrow.

The NC model could be used in a similar fashion. Furthermore, insurance contracts would be more expensive the higher the risk. Foresters could thus reduce insurance costs through windthrow risk reduction strategies.

7.6 Future Research Areas

There are distinct differences between the goals of most scientific research in windthrow modeling and those of interest to the reinsurance industry. They center on the scale and detail of the predictions. Although the more detailed the predictions are the better, Swiss Re is principally interested in large scale (country wide) and long-term predictions used to quantify risk [2]. The scientific community has looked predominantly at stand level risk prediction. Those differences follow from the ultimate uses of the models: The scientific community has mainly been providing tools for predicting windthrow risk to timber companies and foresters. In contrast, the reinsurance industry is not concerned if a specific stand is damaged, but only what the chances are of a large scale windthrow event occurring.

Data that has been collected and published in the literature therefore tends to be concerned with individual stand level predictions of risk. Enlarging the study areas in the hope that local variations would be evened out might lead to the ability to predict

risk over a large area. In essence we want to know whether two storms of the same intensity will cause the same amount of windthrow, or whether each storm is so unique that ultimately the amount of windthrow it causes cannot be predicted.

To accomplish that one could compare windthrow in the same area from storms of similar intensity. Or compare different areas damaged by the same intensity storms while accounting for any variations between the two areas such as topography, tree species, etc. If the wind speeds and stand parameters were similar, the hope would be that the windthrow the storm caused would also be similar. This also means that one could then try to directly create vulnerability curves from the relative damage caused by storms of different intensity at places with similar characteristics.

If such approaches are applied over a large area, the influence of the local chaotic wind patterns should be reduced, since the focus would shift from individual data points to overall means. The problem of how accurately remote sensors can log wind speeds, as previously discussed, would still be an issue. Only improvements in technology or more ground-based wind sensors can fix that problem.

Wind direction, and the rate of its change might also play an important role. A stand could be protected from wind in only one direction. Therefore, tracking only wind speed could give misleading results. Wind from one direction would have little or no affect on the stand, while wind from another direction could hit the stand at full force. Future empirical windthrow studies should monitor wind direction as well as wind speed.

8 Conclusions

This thesis is part of a research collaboration between the University of Zurich and Swiss Re. The partnership centers on an application that performs reinsurance loss calculations for natural catastrophes developed by Swiss Re. The goals of this work were to (1) improve the performance and scalability of the Swiss Re NC model through more efficient data handling, as well as to (2) investigate ways to use the NC model framework to calculate the risk of windthrow to forest stands.

The preceding chapters were ordered in the same way as they developed through time. Initially work was done on the computational aspects of the model. It had been realized before that the large number of database interactions were limiting application performance. Combing the separate ImportCSV, Encode, and Rate processes was a logical way to reduce the number of database requests, by eliminating back-and-forth data transfer. Rather than trying to develop a completely combined algorithm from the start, first just the Import and Encode processes were combined. The Import and Encode combination algorithm reduced the number of database requests, resulting in an improvement in execution speed and a reduction in database usage. Through this task it became clear that it was both worthwhile as well as possible to develop a fully combined algorithm.

Subsequently, the full ImportCSV, Encode and Rate combination algorithm was developed. To implement the algorithm a number of business rules had to be ignored: Any pre-rating checks, such as data validation, were turned off. The algorithm also required each L2 part of the portfolio tree to be independently processable. While the NC model at that time did not meet this requirement, the rule was nevertheless ignored. After the combined algorithm was functioning, Swiss Re modified the model to allow independence between L2s. This shows that business rules should not always be considered as written in stone, as sometimes a change in a rule can enable far better algorithms to be developed. In this case, the new combination algorithm allowed the use of eight nodes instead of just four nodes to process a single portfolio (see Section 4.3.2). This resulted in a large increase in application performance. Additional optimization efforts were only partially able to further improve performance and scalability of the distributed model.

Instead of continuing to focus on how the application interfaces with the database, the decision was made to look directly at the database. Through work of other team members it was discovered that a number of database tables had large amounts of duplicated data [13]. Less duplicate data stored leads to more efficient use of both database caches as well as hard drive space. SQL scripts were developed to delete the duplicated data for a selected number of tables. A maximum improvement of 9.3% in execution time and 18.5% savings in hard drive space was the result (see Section 6.4). Although not large, the gains in storage and execution time come only at the cost of running a few SQL scripts. The next step would be to remove the duplicated data before it is inserted into the database. This could be done during the Importing part of the code. Also the subsequent Encode and Rate steps should be modified to only process non-redundant data.

The above computational work gave enough background about the NC application to allow work to proceed into the identification and exploration of catastrophe risk assessment for biological entities. Windthrow of trees was selected as a good focus area. It is of interest to both the reinsurance and forestry communities. A procedure was outlined how to expand the current NC model to forest windthrow, including the

development of vulnerability curves as the most important aspect. Existing empirical and mechanistic approaches were evaluated regarding their applicability for this task. A large empirical data set was obtained to analyze the relation between wind intensity and windthrow [22,23]. Unfortunately, it turned out to be difficult to establish such a relationship, which is necessary to develop realistic vulnerability curves. Mechanistic models can be used to create vulnerability curves, but without model validation the relevance of the results is questionable.

For empirical models, first improvements in remote sensing technologies are required to get accurate wind speeds to relate to windthrow. Until this happens, it appears that validating mechanistic model predictions may be the best way to proceed. The critical wind speed for windthrow can be calculated for forests near established weather stations. Then when the critical wind speed is reached, the amount of windthrow present can be checked. Although not a costly experiment, it might take a while before that required wind speed is reached. Another possibility is to gather a data set that includes the relationship between wind speed and windthrow for a large number of individual storms over a large area. It would be best to have multiple storms for the same area. One could then try to find macro patterns over the landscape. The hope is that by looking at large forest areas, the influence of the local chaotic wind field might be reduced, such that a relation between windthrow and wind speed could be found.

Acknowledgments

I thank the RATOS Team at Swiss Re for collaboration, in particular for providing the NC model source code, test cases, documentation, and support. Thanks also go to Mike Packard for the setup and maintenance of the project cluster and database infrastructure in the Baldridge Group at the University of Zurich. Wibke Sudholt, as project leader, has provided much needed support throughout the project. Jean-Philippe Schütz was gracious enough to provide the data used in the paper; Vulnerability of spruce (*Picea abies)* and beech (*Fagus sylvatica)* forest stands to storms and consequences for silviculture [22].

References

1. Ancelin P., Courbaud, B., Fourcaud, T., Development of an individual tree based mechanical model to predict wind damage within forest stands, Forest Ecology and Management, 203, 2004.
2. Bresch, D., Hector, A., Monroe, M., Pfister, P., Philipson, C., Schmid, B., Sudholt, W., Wunderlich, S., Windthrow modeling workshop, May 23, 2007.
3. Computer Cluster, Wikipedia, June 20, 2007, http://en.wikipedia.org/wiki/Computer_cluster/.
4. DataSynapse grid middleware, June 20, 2007, http://datasynapse.com/.
5. Distributed Computing, Wikipedia, June 20, 2007, http://en.wikipedia.org/wiki/Distributed_computing/.
6. Eclipse development platform, June 20, 2007, http://www.eclipse.org/.
7. ForestGALES software, Forestry Commission of Great Britain, June 20, 2007 http://www.forestry.gov.uk/.
8. Grid Computing, Wikipedia, June 20, 2007, http://en.wikipedia.org/wiki/Grid_computing/.
9. Gardiner, B., Peltola, H., Kellomäki, S., Comparison of two models for predicting the critical wind speeds required to damage coniferous trees, Ecological Modeling, 129, 2000.
10. Gardiner, B., Suárez, J., Achim, A., Hale, S., Nicoll, B., Forest Gales: A PC-based wind risk model for British Forests, Forestry Commission, Ver. 2.0, June 2004.
11. Holecy, J., Hanewinkel, M., A forest management risk insurance model and its application to coniferous stands in southwest Germany, Forest Policy and Economics 8, 2006.
12. Intergovernmental Panel on Climate Change, Climate Change 2001: Impacts, Adaptation, and Vulnerability.
13. Internal documents.
14. Kramer, M. G., Hansen, A. J., Taper, M. L., Kissinger, E. J., Abiotic controls on long-term windthrow disturbance and temperate rain forest dynamics in southeast Alaska, Ecology, 82(10), 2001.
15. Lanquaye-Opoku, N., Mitchell, S.J., Portability of stand-level empirical windthrow risk models, Forest Ecology and Management, 216, 2005.
16. Mahmood, H., Application of distributed computing to reinsurance natural catastrophe calculation software, Master's thesis, Royal Institute of Technology Stockholm, Sweden, 2006.
17. Mayer, H., Windthrow, Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences, 324, 1989.
18. Ministry of Forests and Range, British Columbia, Canada, June 20, 2007, http://www.for.gov.bc.ca/hfp/.

19. Oracle SQL Developer, June 20, 2007, http://www.oracle.com/technology/products/database/sql_developer/index.html.
20. Prestemon, J.P., Pye, J.M., Holmes, T.P., Timber economics of Natural Catastrophes, Pelki, M. (ed.), Proceedings of the 2000 Southern forest economics workshop, March 2000, Lexington, Kentucky, 2001.
21. Reid Priedhorsky, June 20, 2007, http://reidster.net/.
22. Schütz, J. P., Götz, M., Schmid, W., Mandallaz, D., Vulnerability of spruce (*Picea abies)* and beech (*Fagus sylvatica)* forest stands to storms and consequences for silviculture, European Journal of Forest Research, 125, 2006.
23. Schütz, J. P., Götz, M., Schmid, W., Mandallaz, D., Data used for: Vulnerability of spruce (*Picea abies)* and beech (*Fagus sylvatica)* forest stands to storms and consequences for silviculture, European Journal of Forest Research, 125, 2006.
24. Scott, R. E., Mitchell, S. J., Empirical modeling of windthrow risk in partially harvested stands using tree, neighborhood, and stand attributes, Forest Ecology and Management, 218, 2005.
25. Supercomputer, Wikipedia, June 20, 2007, http://en.wikipedia.org/wiki/Supercomputer/.
26. Swiss Re, June 20, 2007, http://www.swissre.com/.
27. Swiss Re, Natural catastrophes and man-made disasters in 2005, Sigma No. 2, 2006.
28. Swiss Re, Natural catastrophes and man-made disasters in 2004, Sigma No. 1, 2005.
29. Swiss Re, Natural catastrophes and reinsurance, 2003.
30. WebSphere, IBM, June 20, 2007, http://www-306.ibm.com/software/websphere/.
31. Weinstein, D.A., Laurence, J.A., Retzlaff, W.A., Kern J.S., Lee, E.H., Hogsett, W.E., Weber, J., Predicting the effects of trospoheric ozone on regional productivity of ponderosa pine and white fir, Forest Ecology and Management, 205, 2005.