Modification of the Data Handling to Grid-Enable Reinsurance Natural Catastrophe Calculations

Internship Report

by

Daniel Dönni
(0170426)

Submitted to the
Computer Science Department (IFI)
University of Zurich

In Partial Fulfillment of the Requirements for the
Diploma Degree in Computer Science

Supervisors

Dr. Wibke Sudholt
Institute of Organic Chemistry (OCI)
University of Zurich

and

Dr. Burkhard Stiller
Computer Science Department (IFI)
University of Zurich

November 2007

Table of Contents

1. Introduction ... 3
2. Project Description ... 4
   2.1. Background ... 4
   2.2. History ... 4
   2.3. Portfolio Tree ... 4
   2.4. Current Architecture ... 5
      2.4.1. Client Tier ... 5
      2.4.2. Server Tier ... 5
      2.4.3. Database Tier ... 5
      2.4.4. Processes ... 5
   2.5. Prospective Architecture ... 6
      2.5.1. Client and Database Tier ... 6
      2.5.2. Server Tier ... 6
      2.5.3. Grid Tier ... 7
3. Goal ... 8
   3.1. Query Isolation ... 8
   3.2. Query Optimization ... 8
4. Implementation ... 9
   4.1. Query Isolation ... 9
   4.2. Query Optimization ... 9
      4.2.1. The New Persistence Layer ... 9
      4.2.2. AbstractRetriever ... 10
      4.2.3. Usage ... 10
      4.2.4. Functional Correctness ... 11
5. Evaluation ... 13
   5.1. Expectations ... 13
   5.2. Experimental Results after Query Isolation ... 14
      5.2.1. Rating ... 14
      5.2.2. Rating Details ... 14
      5.2.3. File Sizes ... 15
      5.2.4. Database Throughput ... 16
   5.3. Experimental Results With The New Persistence Layer ... 16
      5.3.1. Data Retrieval ... 16
      5.3.2. Rating ... 17
      5.3.3. Overall Processing Time ... 18
   5.4. Query Analysis ... 19
6. Conclusion ... 21
7. Appendix ... 22
   7.1. Settings ... 22
   7.2. Queries ... 22
8. Bibliography ... 24

1. Introduction

Moore's Law states that the number of transistors on a computer chip doubles every other year [1]. Although the original wording was altered and interpretations changed in the course of time [2], most computer scientists still consider it to be true today. The increase in computing power rendered many applications possible that could not have been realized a couple of years ago.

Despite this rapid growth there are still countless applications in academia and industry that require computing power, processing speed, or data storage that go far beyond the one provided by a single computer. These challenges can only be mastered by deploying supercomputers equipped with numerous processors or by aggregating resources of many individual standard machines.

The two major resource aggregation concepts are known as Cluster Computing and Grid Computing. Computer clusters focus on bundling resources of dedicated machines that typically reside at one single location. Conversely, grid systems seek to combine distributed resources, which may not only traverse geographical but also organizational boundaries.

The type of system is dictated by application characteristics (e.g. process interdependencies) as well as hard- and software availability and accessibility. Swiss Re and University of Zurich jointly work on the project described in this report aspiring to find out whether and if so to what extent the cluster-like setup at Swiss Re can be adapted to a grid setup.

This report is organized as follows: Chapter 2 outlines the project including history, current status, and future prospects. The task I had to accomplish during the internship is defined in Chapter 3 and followed by a discussion of vital implementation details in Chapter 4. The results are presented in Chapter 5. Finally, Chapter 6 summarizes the findings, draws conclusions, and outlines areas of future work.

2. Project Description

This chapter outlines the NatCat project, the one I dealt with during my internship. This chapter briefly summarizes the project's background and history, and outlines its current and prospective architecture.

2.1. Background

NatCat is a system that seeks to predict the financial losses arising from natural catastrophes, such as earthquakes, tropical cyclones, floods, and windstorms [4]. The software is developed, maintained, and used at Swiss Re, one of the world's largest reinsurance companies. Together with associated test data, it has been provided to the Baldridge Group at the University of Zurich within a collaborative research project running since September 2005. The present internship is part of this project.

2.2. History

Originally, NatCat used to be a single-computer application [3]. Due to the development of new business areas [4], the growing demand for natural disaster insurance products [4] and the application's popularity at Swiss Re, it evolved into a large and complex system. The system currently runs on a small computer cluster at Swiss Re and on a similar setup at the University, which allows running several tasks at the same time. As part of the collaboration project, distributed versions of the code were developed. These split tasks into partial jobs, which can run in parallel on the different machines and thus increase processing performance. The next goal is now to migrate NatCat to a grid infrastructure, to make further computer resources accessible. This internship represents one step into this direction.

2.3. Portfolio Tree

A portfolio is a hierarchical collection of policies, insured objects, and coverage types arranged in a tree-like structure. The so-called portfolio tree is depicted in Figure 2.1 and explained below. A site (L1) refers to an insured object. It might be covered by one or several insurance contracts against various perils, e.g. physical destruction, business interruption, etc. In this context, the term coverage (L0) defines the insurer's financial liability with respect to this particular object. A policy (L2) contains several sites and specifies the details of the insurance contract. Finally, all policies are part of a portfolio (L3), which represents the root of the so-called portfolio tree.

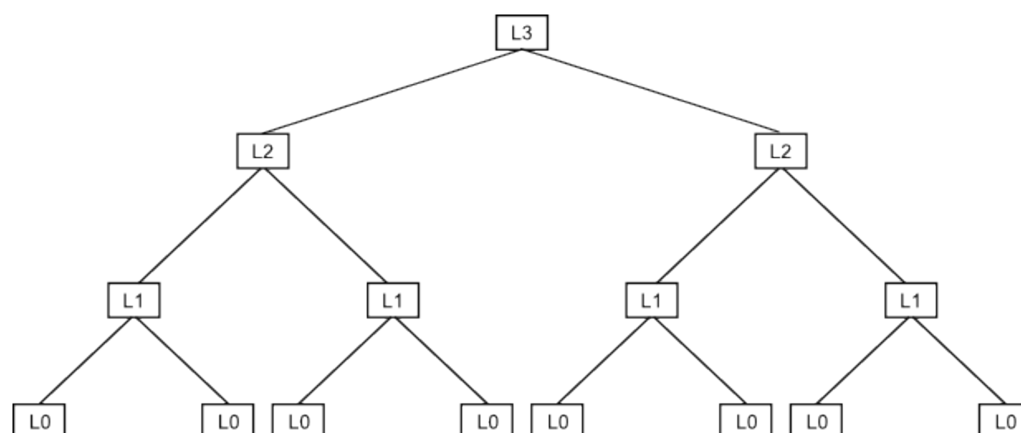

Figure 2.1 Example of a portfolio tree.

2.4. Current Architecture

Figure 2.2 shows the current architecture of the NatCat application. It consists of a client tier, a server tier, and a database tier.

2.4.1. Client Tier

The client tier consists of the hosts having the permission to invoke calculations. In order to do so, a client submits a file to the server tier. Depending on the processing state a user may perform different actions (import, encode, rate), which are then processed by the server tier. Clients are not involved in the calculation process, they only submit tasks and present results. The clients are programmed in J2SE 1.4 or 1.5.

2.4.2. Server Tier

The server tier consists of the hosts executing the calculation. After having received input data from a client, they check the data for consistency, associate it with geographical data, execute the calculation and store the data and results in the database. This process is explained in more detail in the next section.
The NatCat server sources are programmed in J2EE 1.4. The code currently runs on top of WebSphere 6.0 ND, the J2EE 1.4 compliant application server by IBM. It is not clear yet if in a prospective grid infrastructure WebSphere will be suited to provide the required middleware functionalities on all grid sites. Therefore, evaluating the deployment of other products is under consideration.

2.4.3. Database Tier

The database tier is responsible for providing and storing calculation-related data. The database hard- and software are relatively powerful: a dedicated 32-processor system at Swiss Re and a dedicated two-processor or shared four-processor system at the University, all running Oracle 9i. Nevertheless, earlier studies within the collaborative research project [5,6] have shown that due to the large data amounts, the database's performance is one of the major factors limiting the scalability of NatCat.

2.4.4. Processes

The processes "import", "encode", and "rate" constitute the basic framework of the NatCat application and are explained below.

2.4.4.1 Import

The import process is responsible for reading all relevant data from files and storing them in the main database. Errors resulting from the import process are reported to the client tier. Letter 'I' in Figure 2.2 represents the import process. The red arrows show the data flow.

2.4.4.2 Encode

The encode process checks the imported data for consistency and associates them with geographical data from a GIS database. Once again, the resulting data is stored in the database and the client is notified in case of errors. The encode process is represented by letter 'E' in Figure 2.2.

2.4.4.3 Rate

The rate process accesses the data produced by the encode process, calculates the expected losses, and stores the results in the database. The rate process is the one that performs the number crunching. It is by far the most time-consuming task, which

explains why most of the optimization efforts are put into this part. Letter 'R' in Figure 2.2 represents the rate process. Note that "rate" and "calculate" are used interchangeably in this report.

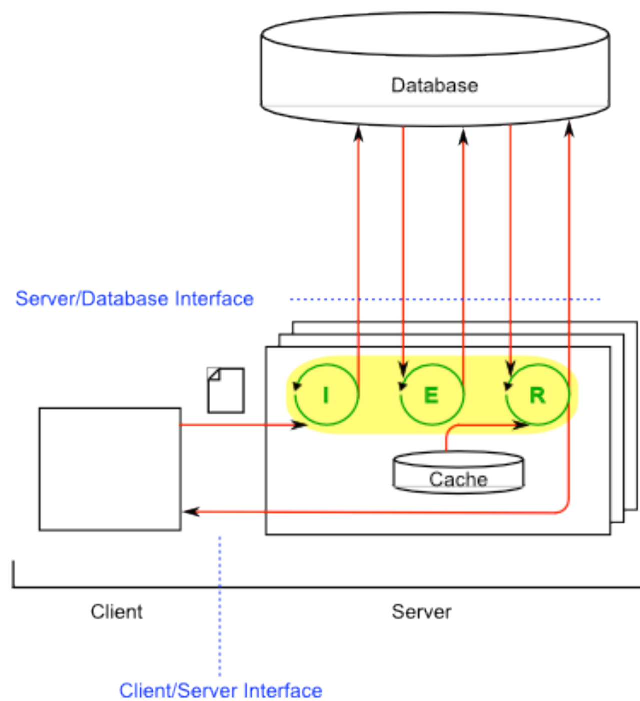

Figure 2.2 Architecture of the original application.

2.5. Prospective Architecture

The current architecture works fine for many calculations. However, it takes up to three days to calculate results for bigger calculations, and there are even cases the system cannot handle because they are too large. Since the scalability of the current architecture is limited, another architecture must be found that is able to cope with future capacity requirements, and if possible increases performance and scalability of the system. Figure 2.3 depicts a possible prospective architecture. The major differences with respect to the current architecture are the additional grid tier and the modified process structure.

2.5.1. Client and Database Tier

While the client tier will basically remain the same, the database tier might undergo several changes. The most important ones concern alterations or the redesign of the database schema, integration of application logic into the database as well as the removal of data redundancy.

2.5.2. Server Tier

The server tier will probably be altered significantly. One change under consideration is to merge the import, encode, and rate processes to limit back-and-forth transfer with the database [6]. However, the most important modification affecting the studies in this internship is the plan to redesign the rate process in a way that allows for transparent local or remote execution. The purpose of the corresponding grid tier is to reduce the load on local nodes.
Another major change concerns the way data is handled in the system. To accelerate data access, it will be necessary to decide which data must be stored in the database: In the

current application, several intermediate results are stored as well, even though they have never been used so far after the result has been calculated. Not storing them anymore would reduce the amount of transferred and stored data. However, as it is currently not known whether these intermediate results will ever be needed, it is unclear whether this potential can be exploited. In addition, due to the current structure of the portfolio tree, considerable amounts of data duplication have been identified in earlier phases of the collaboration project [5]. To reduce these, additional restructuring of application and database is required (cf. Section 2.5.1), which is currently developed and tested in a prototype setup.

2.5.3. Grid Tier

The grid tier is introduced in order to unburden the server tier from some calculation load. It might be used all the time, but it might just as well serve as a buffer that reduces the load on the local nodes in case the capacity demand exceeds the one available on local nodes. Details are not known yet because the interdependencies and performance behavior of individual code and hardware parts are not sufficiently known yet, and also policy issues for remote processing of data remain to be defined. It is clear though that the grid tier will play a major role in the revised architecture.

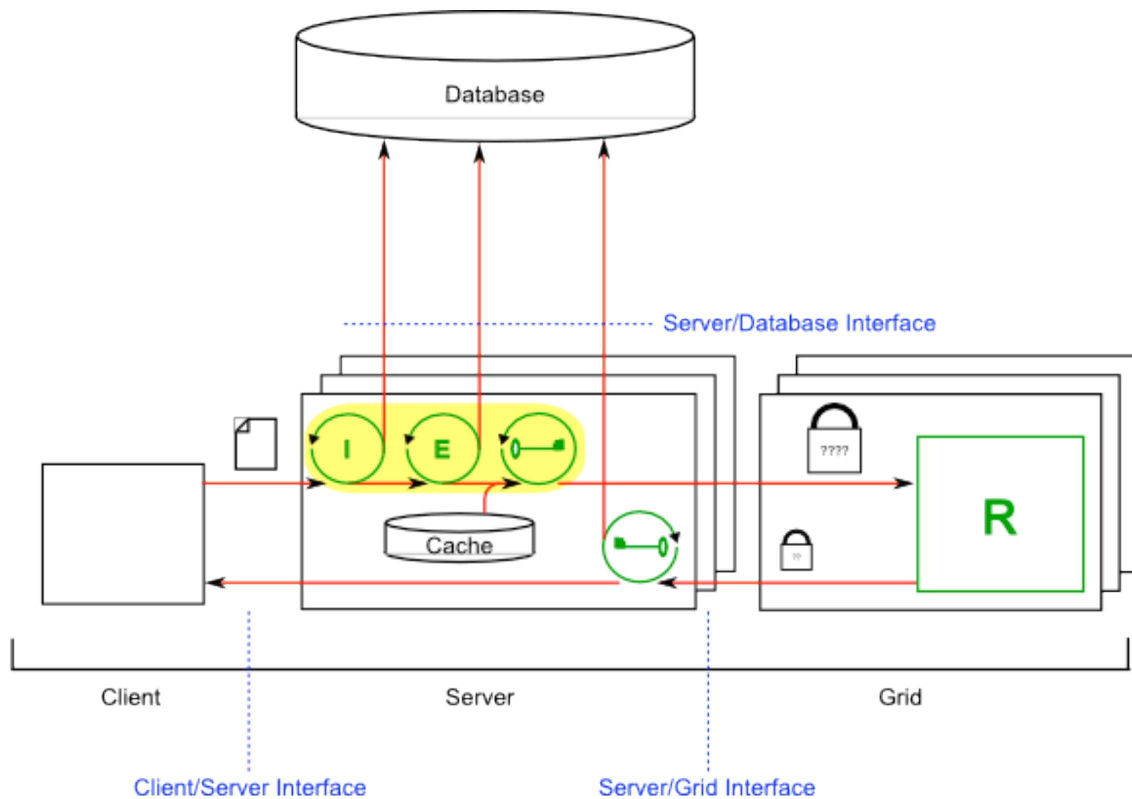

Figure 2.3 Architecture of the prospective application.

3. Goal

As outlined in Section 2.5, the NatCat code has to be restructured to make it suitable to be run on a grid infrastructure. Such a transformation requires major architectural changes, some of which have already been made while others are pending.
One important requirement that has to be fulfilled by NatCat rate jobs running on the grid tier is that they should be independent from the presence of the database server, to allow running in isolation. This reduces the dependence on the network speed and increases the application's reliability. Moreover, it raises hope for performance gains, improved error resilience, and better means to control job and data security. As displayed in Figure 2.3, such a setup makes it necessary to bundle all required and produced data together with the corresponding rate job. The goal of this internship was to provide a prototype solution for this issue.
The work was divided into several subtasks that could be accomplished independently. The first task was to isolate data retrieval from the code performing the calculation. The second task consisted of enhancing the database's performance.

3.1. Query Isolation

The existing code allowed distributing work units among several machines. However, the data associated with a work unit was only retrieved during the calculation process, which led to a strong coupling between the calculating nodes and the database.
In order to loosen this coupling, work units had to be equipped with all necessary data before being assigned to a node, resulting in a complete separation of data handling and calculation. The main task of this internship was to realize this in the source code.

3.2. Query Optimization

Isolating the data retrieval from the calculation process opened possibilities to optimize its performance. Since data was not retrieved on an as-needed basis anymore, it was possible to merge several small queries to a single big query, resulting in diminished overhead and higher performance.

4. Implementation

This chapter describes the implementation of the novel data retrieval mechanism. The focus lies on the isolation of database queries and the performance enhancements of the database, which primarily consisted of replacing the legacy persistence layer with a new one.

4.1. Query Isolation

In order to isolate queries a separate class named `DataRetrieverImpl` has been created which is responsible for retrieving data and storing it in appropriate data structures. The `DataRetriever` interface is shown in Figure 4.1.

```
public interface DataRetriever {

	/**
	 * Retrieves the L3s and everything underneath
	 */
	public L3Array getL3Data(){};

	/**
	 * Retrieves the L2s and everything underneath
	 */
	public L2Array getL2Data(){};

}
```

Figure 4.1 The DataRetriever interface.

The method `getL3Data()` returns an array of `L3` objects including all subordinate data in the portfolio tree. `getL2Data()` does the same but with `L2` objects. The reason for the method's existence is the fact that `L3` objects of bigger test cases cannot be held in memory. In these cases, the `L2` objects are retrieved and processed sequentially.
The actual implementation is not very spectacular; the task mainly consisted of moving existing code parts from the calculation process into the newly created retriever class and of implementing the functionality to wrap and unwrap the data objects (cf. Section 5.1).

4.2. Query Optimization

The original rating process used to fetch data on an as-needed basis, which resulted in numerous database queries returning little data. Once the data retrieval had been isolated from the rating process it was possible to merge several small queries in bigger ones reducing the generated overhead.

4.2.1. The New Persistence Layer

Since it was likely that the existing persistence layer would be replaced in the near future, there was no point in optimizing it. The database access classes were therefore rewritten from scratch, paying particular attention to create a simple, understandable but yet extensible design. The primary purpose was to extract the essential parts from the legacy persistence layer and to put them into new classes such that developers would be able to quickly read and understand them.

The result of my efforts was a new persistence layer consisting of some 1'700 lines of code, which is considerably less than the over 25'000 lines of code of the legacy persistence layer. It must be said though that the newly created persistence layer only contains the functionality required during the rating process. The import and encode processes would have to be added. However, since adding queries requires just a few lines of code, the code is unlikely to grow significantly.
Moreover, data in the legacy persistence layer was retrieved in chunks of fixed size, which made it necessary to issue several queries against the database in order to restore bigger objects. The goal was to get rid off this overhead. Additionally, the results of such a query series were not stored in a single data structure but in as many data structures as queries were executed, resulting in a higher number of data objects to be processed.

4.2.2. AbstractRetriever

`AbstractRetriever` constitutes the core class of the new persistence layer. It is responsible for generating queries, establishing connections to the database, and returning the result to the caller. The most important elements of `AbstractRetriever` are depicted in Figure 4.2.

```
public abstract class AbstractRetriever {

      abstract LinkedList getAttributes();

      abstract String[] getTableNames();

      String getConditionAttribute(){
            return "ID";
      }

      abstract Object createResult(ResultSet rs) throws SQLException;

      public Object retrieveObject(long[] ids) {
            //Implementation
      }
}
```

Figure 4.2 The core methods in AbstractRetriever. Methods for assembling, executing, connecting, and disconnecting from the database are left away as they are irrelevant for understanding the persistence layer usage.

4.2.3. Usage

An example illustrates the usage of the new persistence layer. The structure of a typical query as it appears in the NatCat code is shown in Figure 4.3.

```
SELECT FOO1, FOO2
FROM BAR1, BAR2
WHERE ID IN (1,4,7)
```

Figure 4.3 A simple SQL query.

It consists of the following four components:

- The list of attributes to be retrieved (`FOO1, FOO2`)
- The name of the table storing the respective data (`BAR1, BAR2`)
- The name of the attribute which must fulfill the query condition (`ID`)
- The query condition (`IN (1,4,7)`)

In order to create a new query a new class extending AbstractRetriever (cf. Figure 4.2) must be created. `AbstractRetriever` exposes the following methods:

- `getAttributes()` returns a `LinkedList` containing the attributes to be selected.
- `getTableNames()` returns the name of the table that contains the data.
- `getConditionAttribute()` returns the name of the attribute which must meet the query condition. This method is not abstract as in the vast majority of the cases the condition attribute is named `ID`.
- `createResult(ResultSet rs)` is the method which creates and assigns the values to the data structure representing the data object.
- `retrieveObject(long[] ids)` is the method which initiates the retrieval procedure.

The class corresponding to the query in Figure 4.3 looks as follows:

```
public Object ExampleRetriever extends AbstractRetriever {

      LinkedList getAtrributes() {
            LinkedList attributes = new LinkedList();
            attributeNames.add("FOO1");
            attributeNames.add("FOO2");
            return attributeNames;
      }

      String[] getTableNames() {
            return new String[]{"BAR1", "BAR2"};
      }

      Object createResult(){
            // Query-specific functionality to create the result
object
            return result;
      }
}
```

Figure 4.4 The class corresponding to the query in Figure 4.3.

Retrieving an object from the database is very simple and completely transparent to the caller of the method:

```
Object o = new ExampleRetriever().retrieveObject(new long[]{1,4,7});
```

Figure 4.5 Statement to retrieve data from the database

4.2.4. Functional Correctness

Getting a fine grasp of the inner workings of the legacy persistence layer was a tedious task. It was a mixture of code fragments, manual database key comparisons, and intuitive guesses that helped writing the new persistence layer.

Applying such an approach raises the question of whether the new persistence layer is really equivalent to the legacy one. While there is no way to prove it, the probability that the layers are actually equivalent is very high. This is because the application always returns exactly one numerical value as a result, unless an exception is thrown at runtime.

For the existing test cases the result is known in advance. A test is only successful if the result matches the known result with of precision of 4 digits after the decimal point. In some test cases, the values of several additional parameters must match the known ones with the same precision.
The procedure therefore bears some resemblance with the calculation of the hash value of a downloaded file, which is typically used to verify its integrity. It is possible, albeit unlikely, that the result and the parameters exactly match the known ones by coincidence. Much more likely are scenarios where the test cases are unknowingly chosen such that some code parts are never executed or that the data contains special cases where the bug does not occur, suggesting the code was correct, although it is not.

5. Evaluation

In this chapter, the experimental results are presented and evaluated. The internship was divided into two distinct phases, thus there are also two major measurement series. Before going into further detail though, expectations towards the new code are discussed.

5.1. Expectations

In the first phase of the internship, which mainly consisted of isolating queries, the performance of NatCat is initially expected to decrease, mostly due to the following reasons:

- The portfolio tree must now be traversed twice. While the data was previously accessed during the rating process on an as-needed basis, retrieving all data before starting the calculation requires the application to traverse the portfolio tree twice: Once to retrieve the data and once to perform the calculation (cf. Figure 5.1).
- The data must be wrapped and unwrapped. It cannot only be retrieved; it must also be stored in a suitable data structure. Creating and allocating memory space for this data structure as well as filling it during data retrieval and reading from it during the calculation phase takes time (cf. Figure 5.1).
- Parallelism is reduced. Because the calculation only starts once the data has been completely retrieved, the CPU is underutilized during data retrieval.
- The data must be written to disk and reread – and later, in a real grid scenario, also transmitted over a wide-area network. Once the data is retrieved from the database, it must be placed on a storage device where the nodes performing the calculation reread the data in order to compute results.

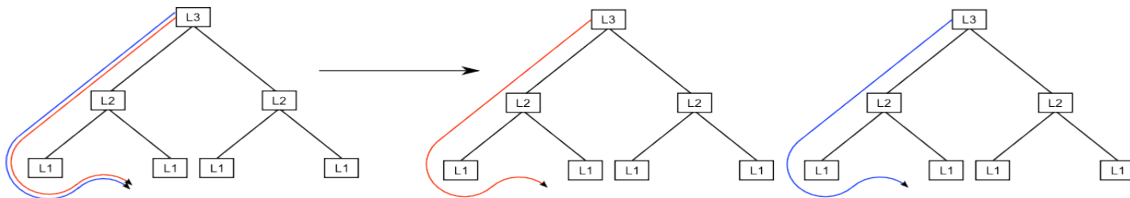

Figure 5.1 In the original implementation, data retrieval and calculation are performed simultaneously (left) while they are separated in the new one (right). For the sake of simplicity, L0 is not shown in the portfolio tree.

Despite the mentioned drawbacks, there are good reasons to retrieve the data beforehand, in particular:

- Parallelism can be regained. While the antecedent data retrieval leads to low CPU usage, it is supposedly high during the calculation process. This might be exploited by reading the data of the next calculation while the result of the existing one is still being calculated. Because the next calculation can immediately be started, the CPU usage is likely to remain high.
- Data can be retrieved more efficiently. Retrieving data only when it is needed results in a considerable amount of database queries returning little data and producing considerable overhead. Retrieving data beforehand allows for merging such queries, resulting in fewer queries returning bigger amounts of data.

- Data can be processed in isolation. While the calculation process is running, data just needs to be read from and written to the local disk. This is both faster and more reliable than accessing data via a wide-area network or even the Internet. Furthermore, eliminating the necessity of a permanent connection to the database server is expected to increase the performance, scalability, and security of the system.

5.2. Experimental Results after Query Isolation

This section presents the details of the performance tests conducted after the first phase of the internship.

5.2.1. Rating

As described in Section 5.1, a performance degradation was expected. Figure 5.2 shows how long it took to rate four small and one medium test cases. (The application configuration for this and all other timings can be found in Section 7.1).
The blue bars show the timings with the original code, the red ones the timings with the modified code. As expected, the performance degraded. However, the difference between the two code versions was small, raising hope that code optimizations would eventually overcompensate the loss.

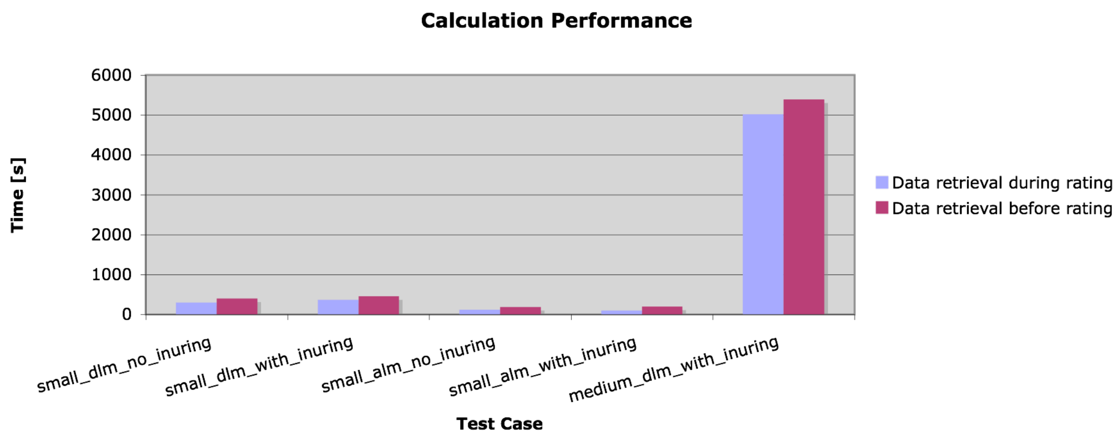

Figure 5.2 The above graph shows the performance of the original code (blue) and the one of the altered code (purple).

5.2.2. Rating Details

The previous section showed the overall performance. For further optimizations it was necessary to know how much time was spent on specific code parts.
Figure 5.3 shows the timings during the rate process. The blue bars represent the time spent on retrieving data; the purple ones the duration of the actual calculation (excluding post-rating).
Obviously, DLM cases are dominated by number crunching, as opposed to ALM cases. The acronym DLM refers to test scenarios where precise location information is available, while this is not the case in ALM scenarios. Therefore, in ALM scenarios some approximations are required. The important thing to know is just that optimization efforts focus on DLM scenarios, because they make up the majority of the large test cases.

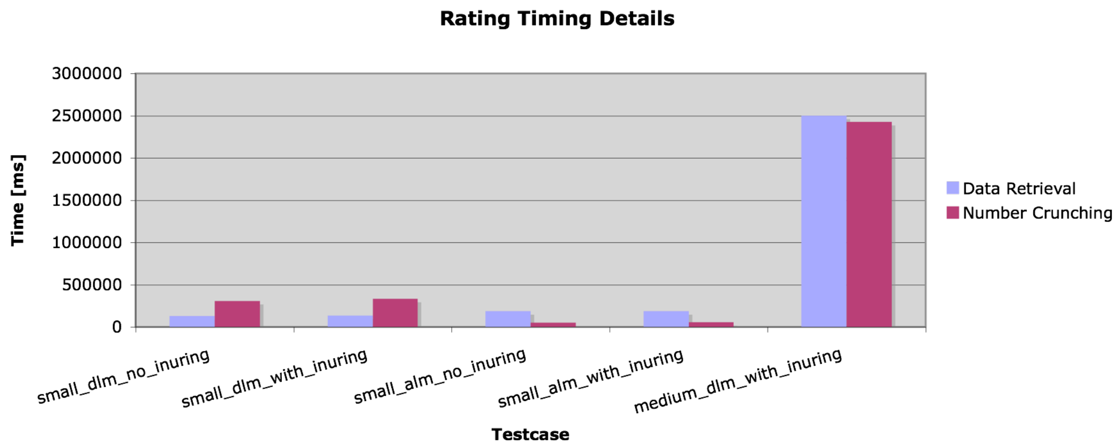

Figure 5.3 The bars show the time it takes to retrieve the data (blue) and to calculate the result (purple). Note that number crunching and data retrieval times do not accumulate to the total rating time. The latter includes additional procedures, such as post-rating.

5.2.3. File Sizes

In many globally distributed applications the network bandwidth constitutes a major bottleneck, hence the size of a work unit is of critical importance. Figure 5.4 shows the size of serialized work units in four small and one medium test case. The blue bars represent uncompressed files; the red ones were created using the GZIP compression algorithm provided by the Java API.
The compressed files are significantly smaller than the uncompressed ones. An interesting observation is the fact that creating compressed files is faster than uncompressed ones, even for the small test cases. The performance penalty of writing a larger file to disk is apparently more severe than the one of the compression.

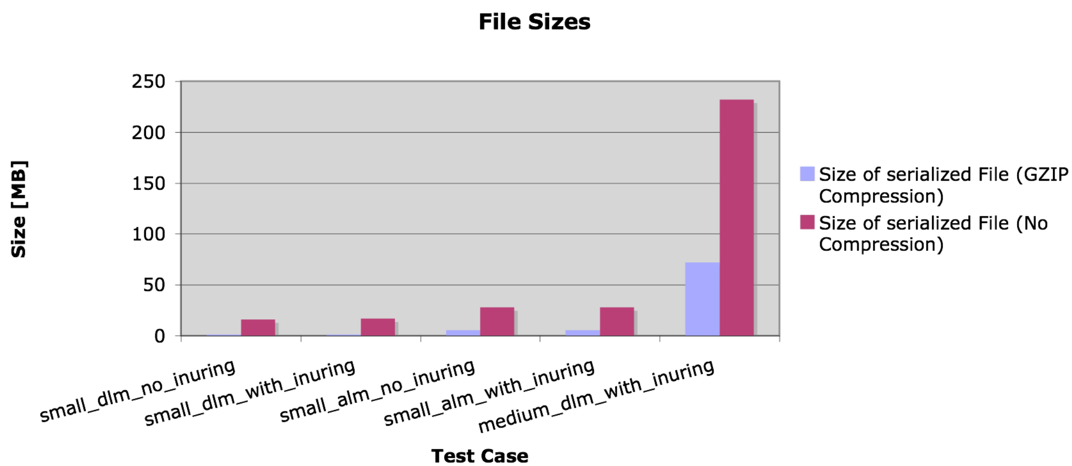

Figure 5.4 Size of compressed and uncompressed serialized work units

The above results look promising. Even if the Internet is used to transmit work units to external resources, the sizes of the compressed files are unlikely to have a significant impact on the performance of the application. What is more, the files were created with the Java default serialization. It is well possible that a customized serialization mechanism produces even smaller amounts of data.

5.2.4. Database Throughput

In earlier phases of the collaboration project [5] it was determined that reading data from the database was one of the major performance bottlenecks. Measuring it precisely was difficult though because data retrieval and calculation were intertwined. After having isolated the code accessing the database this was not very difficult anymore. Running the same test cases yielded the results depicted in Figure 5.5.

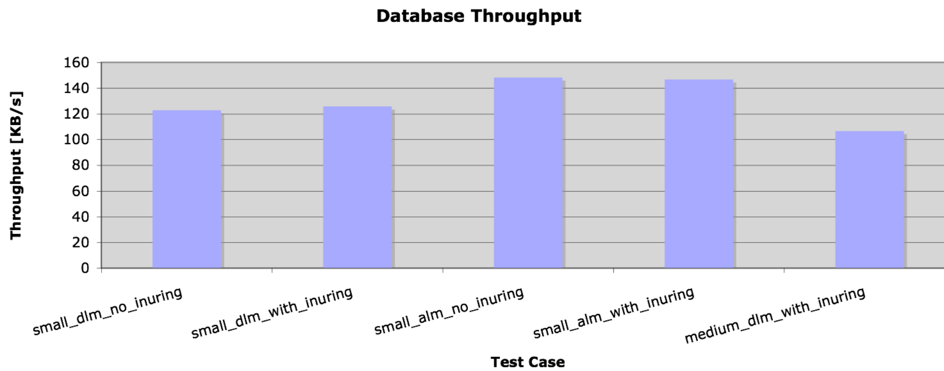

Figure 5.5 The chart shows the estimated average throughput of the database in the original implementation.

For the estimation the new implementation was run and the time of the data retrieval as well as the size of the serialized data file were measured. Due to the fact that the database queries are identical to the ones in the original implementation, the throughput of the implementations is approximately the same. The real throughput of the original implementation is probably a little higher because antecedent data retrieval requires additional data packaging (cf. Figure 5.1). However, the difference is assumingly small (cf. Figure 5.2).

The performance was expected to be poor, somewhere in the one- or possibly the two-digit MBit/s range. The upside of the fact that the actual performance is a lot worse is the fact that this cannot only be caused by the database, which implies that there must be a huge optimization potential in the code and the retrieval mechanism.

5.3. Experimental Results With The New Persistence Layer

This section presents the details of the performance tests conducted after the second phase of the internship. Note that the new persistence layer makes use of WebSphere's connection pool, thus the following timings are directly comparable to ones presented in the previous section.

5.3.1. Data Retrieval

Figure 5.6 compares the performance of the legacy persistence layer with the new one. While the latter is faster in all test cases, the performance gain varies significantly depending on the test case.

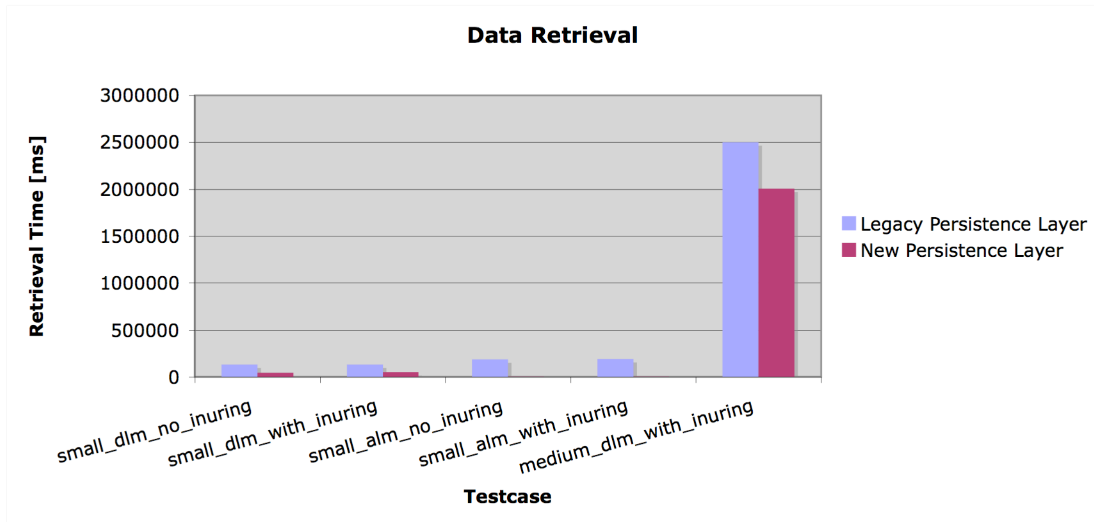

Figure 5.6 Comparison of retrieval times of the legacy and the new persistence layer.

The exact values are shown in Table 5.1. The ALM cases yield the best results, reducing the time to a mere 3% while in the worst case – the medium test case – "only" 20% could be gained. It is difficult to say what this huge variation is caused by; profiling might shed some light on this behavior though.

Test case	Time Consumption (Legacy layer = 100%)
small_dlm_no_inuring	34.4 %
small_dlm_with_inuring	36.4 %
small_alm_no_inuring	2.6 %
small_alm_with_inuring	3.5 %
medium_dlm_with_inuring	80.2 %

Table 5.1 Time consumption of the new persistence layer compared to the legacy one.

It is important to note that the new persistence layer is basically an excerpt of the essential parts of the legacy layer and thus follows the same retrieval strategy. Consequently, the performance gain is primarily caused by the usage of more efficient data structures, and enhanced object handling. Some minor changes have been made in the queries though; however, they hardly scratch the surface of the performance gain resulting from stringent optimization efforts (cf. Section 5.4).

5.3.2. Rating

Figure 5.7 shows the results of the rating part. On the one hand, it was expected that the rating time would increase due to the necessity of unwrapping the data (cf. Section 5.1). On the other hand, data retrieval is not chunked anymore, thus, the amount of objects that must be processed is smaller (cf. Section 4.2.1).

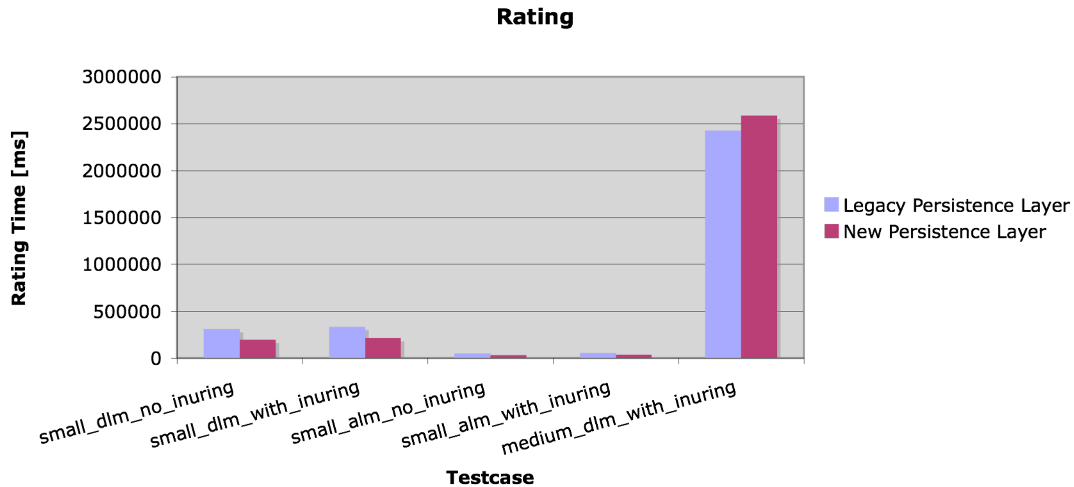

Figure 5.7 Comparison of the rating times with the legacy and the new persistence layer.

As depicted in Figure 5.7, the rating process is faster in the small test cases and slightly slower in the medium test case. The difference is marginal though (cf. Table 5.1).

Test case	Time Consumption (Legacy layer = 100%)
small_dlm_no_inuring	63.5 %
small_dlm_with_inuring	64.0 %
small_alm_no_inuring	64.3 %
small_alm_with_inuring	64.7 %
medium_dlm_with_inuring	106.4 %

Table 5.2 Time consumption of the new persistence layer compared to the legacy one.

5.3.3. Overall Processing Time

Figure 5.8 compares the timings of the complete rating process (including postrating). The new persistence layer is slightly faster in all but one test case. It is surprising that the overall processing time does not reflect the performance enhancements as clearly as one would expect after having seen the previous results. WebSphere's JIT compiler could cause this because some processes (eg. postrating) still make use of the legacy persistence layer, whose classes must be compiled although they are hardly ever used.

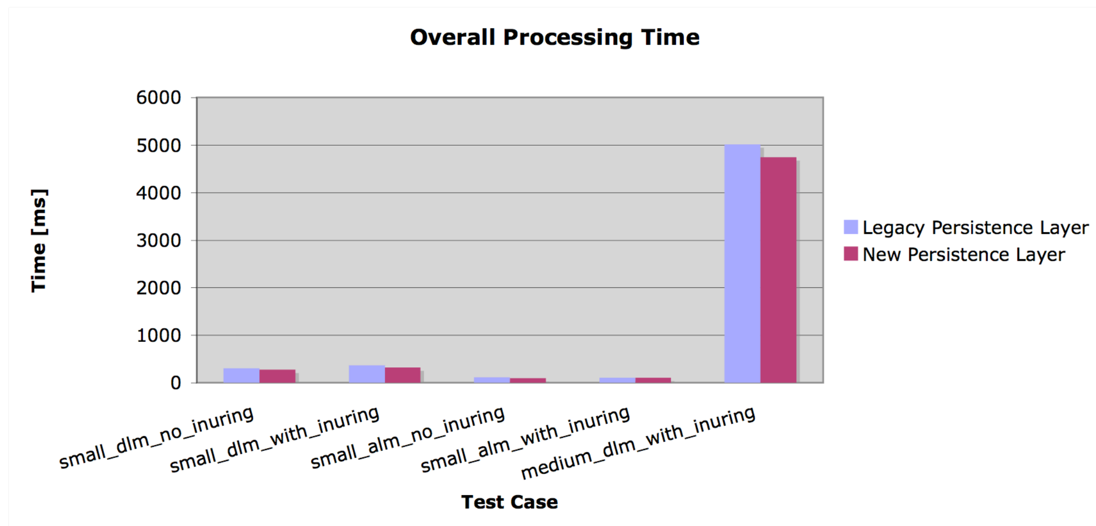

Figure 5.8 Comparison of the overall processing time.

The exact values for the complete rating process are shown in Table 5.3.

Test case	Time Consumption (Legacy layer = 100%)
small_dlm_no_inuring	90.9 %
small_dlm_with_inuring	86.6 %
small_alm_no_inuring	80.2 %
small_alm_with_inuring	100.9 %
medium_dlm_with_inuring	94.5 %

Table 5.3 Time comparison.

5.4. Query Analysis

The purpose of this section is to illustrate the current status of queries as well as some of the remaining inefficiencies contained in the code by means of the medium test case.
In the medium test case a total of 125'070 queries would have been executed without any optimizations. A minor code improvement reduced the amount to 100'452 (-19.7%). This is the code underlying the measurements in Section 5.3.
It turns out that there is a tremendous amount of similar queries. Table 5.4 shows how often queries differing only in the argument of the `WHERE` clause are executed. Apparently, issuing the queries L to T over and over again retrieves the vast majority of the data. Increasing the number of arguments is likely to result in a considerable performance leap.

Query (cf. Section 7.2 for key)	Amount
A – J	1
K	17
L, M	8'967
N, O, P, Q	8'968
R	10'708
S, T	17'935

Table 5.4 Execution frequency of queries

In Section 5.3 it was shown that the performance is better in all but one test case. However, the code did not make use of database hints, hence adding them would further accelerate the retrieval process.

Further ways to improve the performance include multithreaded data retrieval, more efficient handling of the database connection pool, reducing the scope of synchronized methods, and the deployment of a library providing an associative storage accepting primitive data types as keys.

Finally, in this study only four small and one medium test case have been analyzed. The behavior of the new implementation with large test cases was not looked at. It thus remains to be seen in particular how far the memory usage scales.

6. Conclusion

In the course of the internship the data retrieval and rating processes had to be separated. This has been achieved and proves that the two processes can indeed be decoupled and run independently. Thus, it constitutes a first step towards a fully grid-enabled application.

The legacy persistence layer has been replaced with a new one. The results clearly show that it is faster in virtually all test cases, even though it is far from being fully optimized. Some performance gains might be achievable with little effort.

Nevertheless, it is questionable whether that is the way to go. The simple reason is that the creation, execution, and the composition of the result by the database will always consume some time, no matter how stringently the queries are optimized. As a consequence, even the performance of the best optimization will lag behind the one that could be achieved by reading the data directly from a data file, as in that case no queries would have to be executed at all.

7. Appendix

This appendix contains the configuration settings that were used to measure the results.

7.1. Settings

The list below shows the configuration settings that were used to measure the performance. For the sake of simplicity, only the values differing from the defaults and chunk sizes are specified. Note that the new persistence layer never retrieves data in chunks. Therefore, the respective settings only affect test cases that have been run with the legacy persistence layer.

```
batch.save.chunk.size              100
cu.save.chunk.size                 100
file.cache.active                  false
file.cache.purge.on.startup        true
rate.cu.chunk.size                 100
rate.dist.eventset.size.min        1
rate.dist.jobs.max                 1
rate.dist.weight.min               0
rate.distribute                    true
rate.l1.chunk.size                 100
rate.l2.chunk.size                 100
```

Figure 7.1 The list shows the configuration settings used while measuring the application's performance.

7.2. Queries

Key	Query
A	SELECT A.L3_ID, A.ID, A.RANK, A.REFERENCE, A.NAME, A.STATUS, A.PERIL, A.TOTAL_TIV, A.ENCODED_TIV_SHARE, A.L1_COUNT, A.ENCODED_L1_SHARE, A.L0_COUNT, A.ENCODED_L0_SHARE, A.CONDITION FROM MSP_USERDATA.T_L2 A, MSP_USERDATA.T_PERIL_DATA B WHERE A.L3_ID IN () AND A.PERIL=B.ID AND A.STATUS>0 AND B.PERIL='EQA' ORDER BY A.ID
B	SELECT COPULAMIXFACTOR, INTENSITYCOV, MAXCOVFACTOR FROM MSP_EQA.T_LOSS_UNCERTAINTY_PARAM
C	SELECT DISTRIBUTIONPERCENTILE, FREQUENCYSHARE, LOSSSAMPLENUMBER FROM MSP_EQA.T_LOSS_UNCERTAINTY_SAMPLES
D	SELECT ID, DEDUCTIBLE, LIMITID, COINSURANCE, UNDER_COV_AMOUNT FROM MSP_USERDATA.T_CONDITION WHERE ID IN ()
E	SELECT ID, L1_ID, COINSURANCE, COVERAGE_TYPE, RANK, REFERENCE, STATUS, TSI, DEDUCTIBLE, LIMITID FROM MSP_USERDATA.T_CONDITION WHERE L1
F	SELECT ID, L3_ID, L2_ID, L1_ID, RANK, REFERENCE, NAME, RETYPE, IS_SWISS_RE, INCEPTION, EXPIRATION, PRIORITY, NUM_REINST, PER_RISK_LIMIT, OCC_LIMIT, ATTACH_PT, P_SHARE, TIV_WXL_L1 FROM MSP_USERDATA.T_RE_CONDITION WHERE L2_ID IN ()
G	SELECT ID, PERIL, DURATION, MIN_INTENSITY FROM MSP_USERDATA.T_PERIL_DATA WHERE ID IN ()
H	SELECT ID, TYPE, MINIMUM, STANDARD, MAXIMUM FROM MSP_USERDATA.T_DEDUCTIBLE WHERE ID IN ()
I	SELECT ID, TYPE, STANDARD FROM MSP_USERDATA.T_LIMIT WHERE ID IN ()
J	SELECT RC.REFERENCE, RC.NAME, RC.PRIORITY, MIN(RC.ID), MIN(RC.L3_ID), MIN(RC.L2_ID), MIN(RC.L1_ID), MIN(RC.RANK), MIN(RC.RETYPE), MIN(RC.IS_SWISS_RE), MIN(RC.INCEPTION), MIN(RC.EXPIRATION), MIN(RC.NUM_REINST), MIN(RC.PER_RISK_LIMIT),

	MIN(RC.OCC_LIMIT), MIN(RC.ATTACH_PT), MIN(RC.P_SHARE), MIN(RC.TIV_WXL_L1) FROM MSP_USERDATA.T_L1 L1, MSP_USERDATA.T_L2 L2, MSP_USERDATA.T_L3 L3, MSP_USERDATA.T_RE_CONDITION RC WHERE L3.ID IN () GROUP BY RC.REFERENCE, RC.PRIORITY, RC.NAME, RC.RETYPE
K	SELECT VULNCURVEID, INTENSITY, MDR, VARCOEFF, PPA FROM MSP_VULNERABILITY.T_VULN_CURVE_VALUE WHERE VULNCURVEID IN ()
L	SELECT ID, L1_ID, COINSURANCE, COVERAGE_TYPE, RANK, REFERENCE, STATUS, TSI, DEDUCTIBLE, LIMITID FROM MSP_USERDATA.T_CONDITION WHERE L1_ID IN () ORDER BY L1_ID
M	SELECT ID, L3_ID, L2_ID, L1_ID, RANK, REFERENCE, NAME, RETYPE, IS_SWISS_RE, INCEPTION, EXPIRATION, PRIORITY, NUM_REINST, PER_RISK_LIMIT, OCC_LIMIT, ATTACH_PT, P_SHARE, TIV_WXL_L1 FROM MSP_USERDATA.T_RE_CONDITION WHERE L1_ID IN ()
N	SELECT HAZARD, SECOND_MODIFIER, CONDITION, ID, L2_ID, RANK, REFERENCE, STATUS, L1_INFO_ID, L1_INFO_RANK, L1_INFO_REFERENCE, IS_AGGREGATE, RISKS FROM MSP_USERDATA.T_L1 WHERE L2_ID IN () AND STATUS>0 ORDER BY ID
O	SELECT ID, AGE, QUALITY, FLOOD_PROTECTION_LEVEL, BASEMENT FROM MSP_USERDATA.T_SECOND_MODIFIER WHERE ID IN ()
P	SELECT ID, DEDUCTIBLE, LIMITID, COINSURANCE FROM MSP_USERDATA.T_CONDITION WHERE ID IN ()
Q	SELECT ID, SOIL_CONDITION, DISTANCE_COAST, ROUGHNESS, TOPO_INDEX, FLOOD_ZONE_NAME FROM MSP_USERDATA.T_HAZARD WHERE ID IN ()
R	SELECT ID, TICKETID, VULNCURVEID, VULNMOD, REPORTINGUNITID, TSI, RISKS, EFF_RISKS, SCENARIOZONEID, RATINGZONEID, CALCULATIONUNITID, PERILID, HAZARDINCREMENT, HAZARDFACTOR, ALMVULNMOD, SHARED1, SHARED2, SHARED3, ADMIN0LOCID, ISRESIDENTIAL, TOPOINDEX, PTHPARAM1, PTHPARAM2, PTHPARAM3, PTHPARAM4, PTHPARAM5, MARSPVALUE FROM MSP_USERDATA.T_SRX_CALCULATION_UNIT WHERE REPORTINGUNITID IN ()
S	SELECT ID, TYPE, MINIMUM, STANDARD, MAXIMUM FROM MSP_USERDATA.T_DEDUCTIBLE WHERE ID IN ()
T	SELECT ID, TYPE, STANDARD FROM MSP_USERDATA.T_LIMIT WHERE ID IN ()

8. Bibliography

[1] Intel Corporation. Moore's Law. http://www.intel.com/technology/mooreslaw, June 2007. Retrieved: June 14, 2007.

[2] Wikipedia. Mooresches Gesetz – Wikipedia, die freie Enzyklopädie. http://de.wikipedia.org/wiki/Mooresches_Gesetz, 2007. Retrieved: June 14, 2007.

[3] Distributed Computing for Reinsurance related calculations – Design, Algorithms and Runtime experience. http://www.research-projects.uzh.ch/p8487.htm, March 2007. Retrieved: September 20, 2007.

[4] Natural Catastrophes and Reinsurance, Peter Zimmerli et al., http://www.swissre.com/resources/15a16b80462fc16c83aed3300190b89fvNat_Cat_en.pdf, April 2003. Retrieved: May 2, 2007.

[5] Internal Project Documentation.

[6] Natural Catastrophe Modeling: Enhancement and Extension of a Preexisting Model, Mark Monroe, Master of Science Thesis, Institute of Environmental Sciences, University of Zurich, July 2007.