DESIGNING AND DEVELOPING INTERACTIVE BIG DATA DECISION SUPPORT SYSTEMS FOR PERFORMANCE, SCALABILITY, AVAILABILITY AND CONSISTENCY

by

Neil Burke

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
April 2021

Table of Contents

List of Tables

List of Figures

Abstract

Big data decision support systems are used to interpret meaning from extremely large data sets. The users of such systems rely on decision support systems to provide short, human-readable summarizations to aid the user in the decision making process. An *interactive* big data decision support system must do all of this within seconds of a user request. This short response window promotes interactivity between the system and its user, enabling the user to make several ad hoc or follow-up queries to the system shortly after receiving a response.

In this thesis, we explore the design and development of interactive big data decision support systems that satisfy four key useful characteristics: *performance*, *scalability*, *availability* and *consistency*. We do this within the context of two applications.

We first design and develop a novel interactive reinsurance portfolio analytics system. Our system runs on a cloud architecture and efficiently distributes work to achieve excellent *scalability*, scaling up to thousands of cores. In order for our system to be *highly performant*, we design our system to process all data entirely in memory. Our system is made *consistent* by a decentralized data storage service that guarantees strong consistency for all input data. A queuing system that automatically retries failed tasks ensures that the system is *highly available*. In a comparison with one of the leading commercial portfolio analytics systems, our system performed approximately 50 times faster.

Later, we further improve *performance* by caching intermediate results between portfolio analyses, allowing extremely complex location-level analytics queries to be processed in only 11 seconds. Without caching, the same queries would have to process hundreds of millions of transformations over terabytes of data.

Our second application is Online Analytical Processing (OLAP), where we focus solely on data *consistency*. We describe a method for quantifying consistency in distributed OLAP systems and present a corresponding Monte Carlo simulation to approximate the level of consistency for quorum-replicated OLAP systems, allowing users to explore their system's level of consistency under different usage scenarios. In a case study, we validate the accuracy of our simulation on a real, interactive OLAP system.

Chapter 1

Introduction

As the volume of data collected by computer systems grows greater and greater every year, businesses, governments and organizations are increasingly turning to big data decision support systems in order to help them interpret meaning from extremely large data sets. We define a big data decision support system as a computer system that operates on a large, structured data set to produce an output small enough for human consumption, representing an aggregate view of the large data set. Examples of a big data decision support system could be a platform that processes the historical sales data of a large chain of retail stores to aid in deciding what kinds of product and how much should be ordered at each retail store, or a system that analyzes stock trends and financial news stories and aggregates them in real-time into a short, human-readable report to help financial investors to decide which assets to buy or sell.

1.1 Interactive Big Data Decision Support Systems

Put simply, a decision support system is a computer system that provides a decision maker (the user of the system) with data to make well-informed decisions [80, 103]. Decision support systems do not dictate to the user what decision should be made. Rather, they interpret and aggregate data in ways that would be hard for the user to do otherwise, and produce a high-level human-readable response. By itself, decision support systems are an extremely broad topic. Applications can be found in agriculture [50], disease prevention [78], finance [49] and logistics [96]. In the context of this thesis, *big data* refers to processing data sets at the terabyte scale and above. Thus, a *big data decision support system* is a decision support system that processes terabytes of data to generate a summarized output to aid in the process of decision making.

This thesis focuses on the design, development and algorithm engineering of solutions for problems found within applied *interactive big data decision support systems.* Interactive big data decision support systems focus on answering decision support queries in a timely manner, such that interactive user workflows can be supported.

An interactive user workflow is a cyclical pattern where the output of a decision support query is utilized by the user to write the next query. We describe the steps in an interactive workflow in more detail below:

- The user submits a query to the decision support system, potentially after submitting new data to the system.
- The system computes and returns the results of the query to the user.
- Using the results computed by the system, the user makes adjustments to the query or a subset of the data stored on the system and submits a new query, starting the cycle anew.

A key characteristic to interactive workflows is the immediacy in which the decision support system computes and returns the result to the user. Delays longer than 10 seconds can inhibit the user's focus on the task at hand and can result in them switching to a new task while they wait for a response [66, 93]. The time in which a response is returned is also critically important for workflows where a response is needed as soon as possible (e.g., negotiating the terms of a deal over the phone).

In order for a big data decision support system to be fast enough to support interactive workflows, it must be *scalable* and *highly performant*. To ensure the system is accessible at all times, it must be *highly available*; that is, the system must be able to answer queries and ingest new data without any downtime, especially in the event of network, hardware and software failure. If a system is highly available, special considerations must be made to make sure the system is sufficiently *consistent*, meaning that the system uses the most recent version of the data ingested by the system when computing the result of a query.

In this thesis we discuss engineering scalable, highly performant, available and consistent interactive big data decision support systems. We explore these concepts and how they can be achieved within the context of two different applications of interactive big data decision support systems. We begin by briefly describing *performance*, *scalability*, *availability* and *consistency* and how they fit into interactive big data decision support systems before describing the two applications at the core of this thesis and how they fit into these concepts.

1.2 Performance, Scalability, Availability and Consistency

Performance. The performance of any high performance computing system is measured using two metrics *latency* and *throughput* [83]. Latency describes the time taken for a user to receive a response for a submitted query. Throughput describes the number of queries or insertions a system can process in a given unit of time.

Latency is a key metric of any decision support system, as it effectively determines how long a user must wait until they receive a response to a given query. Latency is of particular importance in applications where the user has a limited amount of time to make a decision. For example, consider the scenario where a user of a decision support system is negotiating the terms of a contract over the phone. As the user and the person over the phone negotiate, the terms of the contract change, and thus new data is introduced. If the decision support system supports low-latency insertions and queries, the user can insert the new data, submit a new query, and receive analytics from the system, all in a couple seconds, allowing the user to factor in the latest results from the system as they negotiate over the phone.

Throughput is also an important metric, particularly for decision support systems that must ingest a constant stream of data, and for systems that are used by multiple users in a company. If a system cannot support the throughput required by a company, additional latency can be introduced to insertions and queries, as the system "catches up" and processes operations.

For a system to have low latency and high throughput, careful design and engineering is necessary at both "high" architectural level and the "low" component programming level. By designing a *scalable* cloud architecture, a system can make use of additional compute resources to run multiple insertions or queries concurrently, increasing throughput. In decision support systems where queries are especially complex and require several steps to compute, latency can also be reduced by splitting up the work of a single query to run on multiple compute nodes in parallel.

At the lower level, optimizing the running time performance of the individual components within the cloud architecture is also necessary to minimize latency and maximize throughput. This can depend on many factors, including choice of algorithms, data structures, programming implementation and software stack. In particular, in-memory data structures can drastically improve performance by minimizing data accesses to high-latency persistent storage.

Scalability. Scalability describes a system's ability to effectively utilize additional compute resources to reduce query latency or increase its throughput. For a system to scale well, it must be able to distribute its work across a large number of compute nodes, with all compute nodes working in parallel. Although no system can scale arbitrarily, a system that could scale to an arbitrarily large number of compute nodes could in theory scale to an arbitrarily low latency, or an arbitrarily high throughput.

For big data decision support systems, scalability is critical as the scale of data that must be scanned, analyzed or processed to perform an analysis or answer a query is almost always too large to process on a single compute node. As discussed above, scalability is also extremely important in achieving high performance in a decision support system.

For a system to be scalable, it must be able to efficiently distribute its workload across many compute resources. For systems that process small numbers of complex queries that require processing terabytes of data, individual queries must be split up to be processed in parallel on multiple compute nodes. Efficient distribution of work is typically not trivial, and is largely determined by how the queries are processed. In general, overall system utilization across all workers should be maximized throughout the duration of the query. This can be done by minimizing communication and synchronization between compute nodes, and evenly balancing the processing load across all compute nodes.

Availability. Availability refers to a system's ability to answer queries and ingest new data without any downtime, especially in the event of network, hardware or software failure. This is extremely important in cloud-based systems, as system failures and network interruptions are to be expected in cloud environments, particularly as the system scales to more resources.

In a decision support system, downtime means users can no longer submit new queries or data for the system to process or ingest until the failure has been addressed. This can result in loss of new data, and leaves users without analytics to aid their decision making process. For businesses licensing their decision support system under the software as a service (SaaS) model, excessive downtime can be especially costly as it may frustrate clients and violate service level agreements, resulting in expensive rebates or loss of clients.

Systems are made highly available by being resilient to failure. The first steps towards a high-availability system is to ensure that the system's cloud architecture has no single point of failure. This means no individual compute node is necessary for the successful completion of a request, and that all data necessary to complete a request must be replicated and accessible at multiple different compute nodes. A system's availability can be further improved by ensuring no group of any n compute nodes is necessary for completion of a request, and by increasing the replication factor of all data necessary to complete any request. Availability can also be improved by hosting data and compute nodes across multiple data centers. This way, in the rare event that an entire data center goes offline, the data and compute nodes hosted in the other centers can continue to serve requests.

Consistency. Consistency refers to a system's ability to use the most recent version of the data ingested by the system when computing the result of a given query. The problem of data consistency arises as a direct result of the combination of data replication (to support availability) and latency. If data is replicated on multiple different compute nodes or servers, then each data update operation must be sent to all replicas. Because each operation has latency, and because latency can vary depending on several factors (location, load, etc.), each replica will be updated at different points in time. Thus, if a query is processed while the data update is still in-transit on some replicas, the query's result may be computed from stale, out-of-date data, at least on some replicas.

In decision support systems, inconsistency means that the result of a query may not factor in the most recent view of the data. The significance of this depends on the application and the specifications of the system, and can be difficult to gauge. For applications where the most recent data is especially important, strong consistency guarantees are critical. Once again, consider the scenario where a user of a decision support system is negotiating the terms of a contract over the phone. Say the user inputs the changes to the contract proposed by the client over the phone and submits a query to aid in deciding if the changes are acceptable. A system with poor consistency may return query results based on the contract before the discussed changes, or with only a portion of the changes. Worse yet, if the system has no guarantees on the level of consistency within the system, even if the query did use the most recent changes during processing, the user would have no guarantee that the most recent data was used. This can reduce confidence in the results generated from the decision support system in general. Conversely, missing the past minute of insertions or updates is likely not important for decision support systems or queries that are meant to analyze trends over a much longer period of time.

In distributed systems, consistency is very difficult to guarantee without making significant sacrifices in terms of latency and availability [35]. Consequently, many popular distributed data stores only guarantee that their writes will eventually be readable after an unspecified amount of time [17, 27, 54]. In Chapter 5, we give a background of consistency in greater detail, discussing different types of consistency, how consistency can be guaranteed by sacrificing latency and throughput, and how consistency can be measured probabilistically for simple key-value data stores.

1.3 Contributions of this Thesis

1.3.1 Reinsurance Analytics

Chapters 2 to 4 are focused the design, development and algorithm engineering of a novel interactive decision support system for reinsurance analytics.

Natural disasters, such as earthquakes, floods or hurricanes, can expose an insurance company to *catastrophic* losses that result in the company's bankruptcy or, worse, its inability to reimburse its clients. *Reinsurance* companies act as insurers for insurance companies. A reinsurance contract between an insurance company and a reinsurance company protects the insurance company against such losses in exchange for a premium to be paid to the reinsurance company. The reinsurance industry is capitalized at $500 billion per year and has annual gross written premiums of more than $260 billion.

Reinsurers can significantly reduce the risk of their portfolios by insuring small shares of insurance treaties all over the world. By diversifying their insurance treaty shares geographically, reinsurers can minimize their portfolio's "tail risk" caused by rare catastrophic events like major earthquakes or hurricanes. Consequently, the portfolio of a single reinsurer may indirectly insure hundreds of millions of individual properties, each with their own contracts and terms, across several different countries.

As we describe in greater detail in Chapter 2, reinsurance analytics systems allow users to model the risk distribution complex reinsurance portfolios. Reinsurance analytics systems differ from other decision support systems in that they must apply millions of financial transformations to millions of simulated insurance claims generated by a Monte Carlo simulation to compute a result. Reinsurance analytics systems are critical to the success of reinsurance companies, and the global insurance market as a whole. In particular, the portfolio risk distributions generated by reinsurance analytics systems are especially important in informing underwriters of their portfolio's tail risk. If a portfolio's tail risk is too high, one or two extreme catastrophic events can result in the reinsurer going bankrupt and failing to fulfill its financial obligations to its insurees. With a reinsurance analytics system, underwriters can quantify their tail risk, and take steps to "de-risk" their portfolio.

Because the amount of data required to compute a portfolio's risk distribution is very large, reinsurance analytics systems typically pre-aggregate their simulated insurance claims by region, county or province. This results in a significant reduction of data and processing time at the cost of a less accurate portfolio risk distribution. In Chapter 3 of this thesis, we describe a reinsurance analytics decision support system capable of computing portfolio risk distributions using the original, non-aggregated simulated insurance claims. Our system uses an algorithm to find a space-efficient execution plan that depends on the size and structure of the given portfolio. Because of this, our system is able to compute portfolio risk distributions entirely in memory, making our system over 50 times faster than one of the leading commercial reinsurance analytics decision support systems running on roughly equivalent hardware.

Furthermore, through a cloud architecture that splits the work into small pieces that can be processed independently, our system exhibits excellent scalability. In one experiment, our system scaled up to 2880 vCPUs to process 4 terabytes of input data in approximately 17 minutes. Our system makes use of stateless workers that load input data from a highly available distributed storage service to ensure that no single compute node is necessary to process a request. Since all input to our system is uploaded and stored on a distributed storage service, our system's consistency is determined by the level of consistency guaranteed by the storage service.

In Chapter 4 we augment our reinsurance analytics decision support system with a caching system that allows us to significantly reduce the running time of subsequent analyses. Our caching system intelligently selects a small number of intermediate results to cache (from a pool of hundreds of millions) to strike a balance between cache size and the running time of subsequent analyses. Our experiments show that the introduction of caching can reduce the running time of an analysis by factor of 90, bringing total running time for a reinsurance analytics job down from 17 minutes to approximately 10 seconds, while still maintaining the original system's availability, scalability and consistency. With this, our reinsurance analytics system can support interactive workflows, making it a scalable, consistent, highly performant and highly available interactive big data decision support system.

The core contributions of Chapter 3 were published in the proceedings of the 30th Annual International Conference on Computer Science and Software Engineering [20]. We plan to submit the core contributions of Chapter 4 to the 31th Annual International Conference on Computer Science and Software Engineering for review in the Summer of 2021. The research in both chapters was conducted with support from AnalyzeRe [12], a reinsurance analytics software company based in Halifax, and is being used to develop the next generation of its portfolio analytics platform.

1.3.2 Real-Time OLAP

In Chapters 5 and 6, we explore another application of interactive big data decision processing systems: real-time online analytical processing (OLAP). As is described in more detail in Chapter 5, OLAP systems allow users to insert large amounts of numerical measure data within a multi-dimensional, hierarchical space. Users can query multidimensional subspaces within the dataset to obtain aggregations of the measure data within the subspaces much faster than would be possible on a traditional transactional or key-value database.

In Chapter 6 we present Aggregate Probabilistically Bounded Staleness (A-PBS), a model for describing bounded staleness on OLAP queries. The model, along with a simulation, is used to estimate the probability that a query will return a result consistent with a stream of recent write operations. In a case study, we apply our model and simulation to place probabilistic bounds on OLAP queries for *VOLAP* [29], a real-time distributed OLAP system. We later verify these bounds by running queries in *VOLAP* and observing the actual consistency of the result. We focus solely on consistency in this chapter; since *VOLAP* was designed from the ground-up with scalability, performance and availability (but not consistency) in mind, we refer the reader to the original *VOLAP* paper [29] for a discussion of scalability, performance and availability in real-time OLAP systems.

The core contributions of Chapter 6 were published in the proceedings of the 21st International Database Engineering and Applications Symposium [21].

1.4 Thesis Outline

The remaining chapters of this thesis are organized as follows. Chapter 2 gives a brief overview of the reinsurance industry and related reinsurance analytics systems. In Chapter 3, we describe a reinsurance analytics decision support system capable of efficiently computing portfolio risk distributions at location-level resolution. Chapter 4 expands upon the system described in Chapter 3 by introducing a method wherein a very small percentage of intermediate results are cached, resulting in a significant reduction in the running time of subsequent analyses. Chapter 5 gives an introduction to Online Analytical Processing (OLAP) and staleness in distributed data stores. In Chapter 6 we present Aggregate Probabilistically Bounded Staleness (A-PBS), a model for describing bounded staleness on OLAP queries. Finally, in Chapter 7, we offer concluding remarks and present possible directions for future work.

Chapter 2

Background: Reinsurance Analytics

Insurance companies sell insurance to property owners and thereby expose themselves to the risk of financial losses when the insured files a claim. An insurance contract protects a house owner from catastrophic losses in the event of a fire, water main break or natural disaster. If none of these events occurs, the insurance company profits by charging an insurance premium without having to cover any losses. Natural disasters, such as earthquakes, floods or hurricanes, can expose an insurance company to *catastrophic* losses that result in the company's bankruptcy or, worse, its inability to reimburse its clients. *Reinsurance* companies act as insurers for insurance companies. On the surface, the model is similar to primary insurance: In case of a catastrophic event, a primary insurer incurs losses by meeting its insurance obligations to its clients. A reinsurance contract between a primary insurer and a reinsurance company protects the primary insurer against such losses in exchange for a premium to be paid to the reinsurance company. Unlike primary insurers, reinsurers can significantly reduce the risk of their portfolios by insuring small shares of primary insurance treaties all over the world. By diversifying their primary insurance shares geographically, reinsurers can minimize their portfolio's "tail risk" caused by rare catastrophic events like major earthquakes or hurricanes.

It is because of this diversification that a typical reinsurance portfolio indirectly insures several million locations all over the world. Another key difference between primary insurance and reinsurance is that reinsurance treaties have much more complex structures, often covering only a portion of the losses up to a given limit and under very specific conditions. The reinsurance industry is capitalized at $500 billion per year and has annual gross written premiums of more than $260 billion.

The relationship between reinsurer and insurer is mutually beneficial. By paying a yearly premium to reinsurers, the primary insurer is able to remain in business in the event of a catastrophe, while reinsurers who wisely invest in reinsurance treaties are able to profit from premiums.

In essence, insurers and reinsurers both play a game of chance: In case of a catastrophic event, the reimbursement to be paid to the client far exceeds the premium paid by the client (for a reinsurance company, the client is a primary insurer), but the probability of this happening is low. Or at least, this is the hope of the reinsurer: the probability to make a profit from premiums and no or low claims should be (substantially) higher than the probability of a loss from claims that exceed the premium. Mathematically, we are dealing with a probability distribution over a range of possible losses, called the *loss distribution* throughout the remainder of this thesis.

Each reinsurer holds a portfolio of thousands of complex reinsurance treaties (contracts) with primary insurance companies, which makes it challenging to determine the reinsurer's exposure to risk. Most importantly, the interactions between different reinsurance treaties are sufficiently complex that a closed-form description of the probability distribution of the combined losses is impossible to derive without simplifying the distribution to an unacceptable degree. As a result, the reinsurance industry relies on decision support systems to determine if a particular portfolio is likely to earn or lose money. These decision support systems use Monte Carlo simulation to obtain a sufficiently accurate estimate of the loss probability distribution. Since this is a computationally costly process, most reinsurance decision support systems attempt to reduce the complexity of modeling portfolios by limiting the "granularity" of the analysis, that is, by aggregating loss information to the level of entire counties or states. This results in less detailed analyses, but modeling a portfolio at the level of individual insured properties, called *location level* in this thesis, on current commercial systems is an involved, manual process that requires waiting for weeks or months to obtain a result.

2.1 Portfolio Risk Analysis Using Monte Carlo Simulation

In order to make informed decisions about what treaties to invest in, reinsurers determine the expected losses (payout to primary insurers), worst-case losses, seasonal distributions of losses, and many more metrics of their portfolios. The interactions between different risks each insured property is exposed to and between individual reinsurance treaties make it impossible to derive closed-form expressions describing a portfolio's loss distribution. As a result, reinsurers use risk analytics systems based on Monte Carlo simulation [15] to obtain a discrete approximation of this distribution.

Seismologists, meteorologists, other scientists, and engineers develop catastrophe models that predict the regional likelihood of different events, their severity, and the resulting type and amount of damage to insured properties. Based on these models, it is possible to sample a sequence of events throughout a year. Such a sequence is often referred to as a *trial*. The sequence of events causes a sequence of losses for each insured property. which are recorded in a *year event loss table* (YELT) for the property. Each entry in the YELT is referred to as an *occurrence* and describes an instance of a catastrophic event. Each occurrence is represented by the *trial* (sometimes referred to as *year*) in which it happens, the simulated time of year at which it happens (hereafter referred to as *sequence*), the type of event the occurrence represents (for example, earthquake), usually represented as an integer ID, and the monetary damages (*loss*) caused by the occurrence. A small sample YELT is shown in Table 2.1.

Trial	Sequence	Event	Loss
1	25.5	100	$543,304
1	362.1	55	$40,104
2	68.9	4000	$68,346

Table 2.1: A short sample YELT

Each input YELT represents only the monetary values of the physical damages to a single insured property. The insurance contracts between property owners and primary insurers determine the payouts to be made to property owners and thus the losses of primary insurers. These losses are then covered through a potentially complex network of reinsurance treaties that determine the portion of losses the primary insurer recovers from the reinsurer under these treaties. These payouts from reinsurer to primary insurer constitute the losses of the reinsurer. The sequence of these payouts from reinsurer to all of its clients throughout a year is the *portfolio YELT* for this year. By sampling typically 10,000 trials from the catastrophe models and computing the portfolio YELT for each, we obtain a discrete sample of portfolio YELTs drawn from the probability distribution over these YELTs. By aggregating the portfolio YELTs for all trials, we obtain rather accurate approximations of the different characteristics of the portfolio's loss distribution.

In summary, the input of a location-level risk analytics system is the reinsurance portfolio to be analyzed and a set of trials, each represented as a collection of YELTs, one per insured location. This is illustrated in Figure 2.1. The structure of a typical location-level portfolio is illustrated in Figure 2.2. Contracts at the primary insurer and reinsurer level (represented by triangles) are themselves composed of complex networks of dependent contracts and

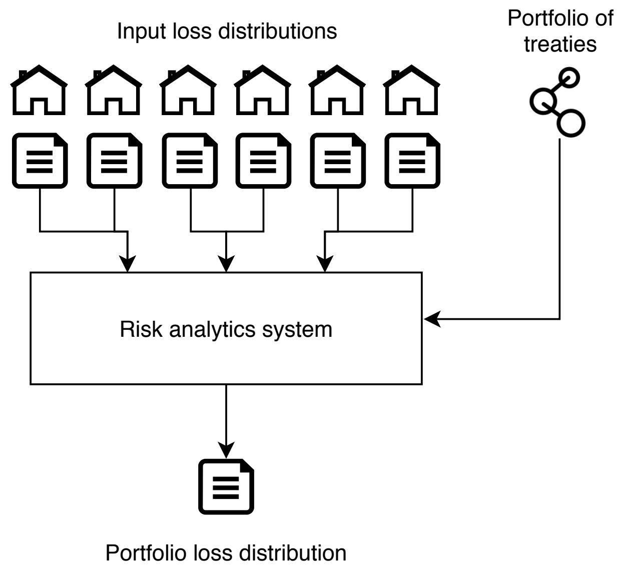

Figure 2.1: A simple overview of the reinsurance modeling process

terms. The result is a large dependency graph, where several million source vertices (one for each location) pass their loss distributions along their edges through a network of primary insurance contracts and treaties, which in turn pass their output loss distributions into a network of reinsurance treaties, until everything converges into one loss distribution for the whole portfolio.

Performing a location-level portfolio analysis is computationally costly. A typical reinsurance portfolio covers hundreds of millions of insured locations; modeling the resulting risk exposure requires a model of the reinsurance portfolio that includes the details of the individual insurance contracts for all locations insured by primary insurers and of the often complex network of reinsurance treaties between primary insurers and reinsurer. The largest part of the input is the YELTs for all insured locations, one per trial, which amount to terabytes of data.

Most commercial risk analytics systems are incapable of performing an analysis of this scale or would take weeks or months to produce a result. Thus, to reduce the computational cost, most risk analytics systems use input YELTs pre-aggregated to the county or state level. This reduces the time to perform a portfolio risk analysis substantially but makes certain types of analyses difficult or impossible. For example, a treaty may only apply to properties within 10 kilometers of the shoreline. At location level, it is essentially trivial to determine which locations fall within this range. At the county level, modeling a treaty like this is impossible,

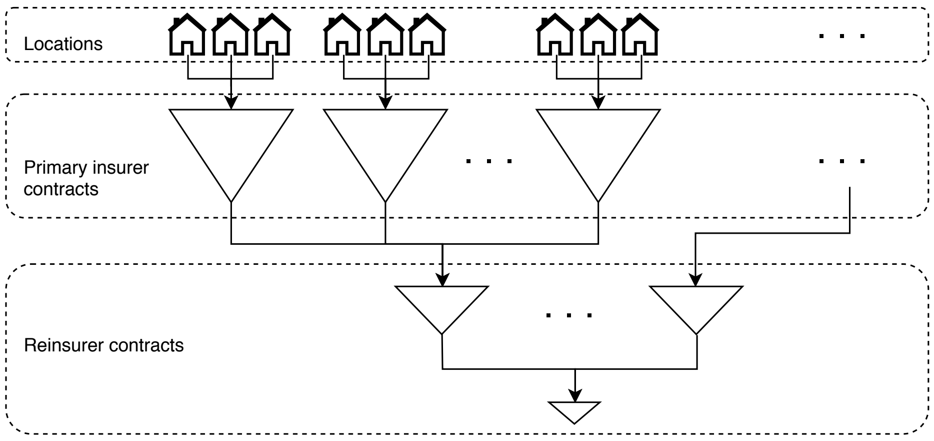

Figure 2.2: The typical structure of a location-level reinsurance portfolio. Triangles correspond to embedded structures within the portfolio, where the size of the triangle roughly corresponds to the size of the structure.

as the losses in each county may be a combination of losses close to and far away from the coast. Location-level analysis also enables reinsurers to measure the marginal impact of each individual property, allowing them to more accurately price the return of a primary insurance treaty. For reinsurance companies that can make binary selections on which properties to include in their portfolio, location-level analytics allows them to select the properties that best fit their desired risk profile, and exclude the ones that do not.

2.2 Related Work

Location-level insurance analytics is a mostly unstudied topic. In this section, we focus on the existing academic literature on reinsurance analytics systems in general, as well as current commercial systems on the market today.

2.2.1 Reinsurance Analytics in the Literature

For reinsurance analytics at coarser granularities (county, state, province), the academic research that does exist focuses on a simplified version of the problem.

Work has been done on creating a distributed risk analytics engine using Hadoop [72], but it assumes the portfolio structure is strictly a tree of depth 3; it is assumed that a portfolio is composed of a set of “programs”, a program is composed of a set of “layers” (or contracts), and a layer is composed of a set of input loss distributions. Not only does this assumption make location-level analyses impossible, it also severely restricts the practical use of the system even for coarser-grained analyses, as actual reinsurance portfolios are very rarely composed of strictly delineated layers and the interactions between treaties are more complex than what can be modeled as a tree.

Other works use optimizations on specialized hardware to achieve fast, single-node running times [22, 28] but make similar simplifications regarding the application of treaty terms.

2.2.2 Reinsurance Analytics in the Industry

There are a number of commercially available products on the market that implement part of the functionality required to solve the problem of location-level reinsurance analytics.

Catastrophe modeling software from vendors such as AIR [4] and RMS [74] can compute the loss distributions of a group of locations up to the primary insurer level. However, since these software packages are primarily meant for primary insurers, the system is designed for modeling a much smaller number of locations than what would be seen in a reinsurance portfolio; the number of locations modeled rarely exceeds 2 million. Both AIR's product Touchstone, and RMS's product RiskLink are fundamentally built around Microsoft SQL Server as both their data storage and computational platform. As such, these applications are bound to the performance and scalability limitations that come with SQL servers [91].

For analytics on reinsurance portfolios, the solutions on the market today [5, 12, 87] are only able to consume data at an aggregated geographic level (e.g., county level). The most flexible of these systems allow the user to "nest" contracts within certain other contracts. This allows for some flexibility in defining simple dependent relationships between contracts, but it does not offer the same flexibility and ease of expressing relationships between contracts as a graph-based portfolio representation.

A third class of commercially available solutions are so-called *Dynamic Financial Analysis* (DFA) products (RiskExplorer [88], MetaRisk [40], ReMetrica [13], Tyche [73]). DFA tools allow for the modeling of complex cash flows and enterprise risk scenarios that go beyond catastrophe risk, such as investment risk and operational risk. These tools provide the greatest level of flexibility in terms of structuring and modeling features. However, they consume data at a very coarse level of detail, typically just at the trial level or sometimes by country, and are typically limited to a small number of trials.

Chapter 3

A Scalable System for Efficient Location-level Analytics

In this chapter, we describe a highly performant, highly available, scalable and consistent decision support system for reinsurance analytics. Unlike other reinsurance systems, our system has been designed from the ground up for computing the risk exposure of a reinsurance portfolio at location-level granularity. As discussed in Section 1.3, this is the first step towards developing an interactive risk analytics system.

This chapter makes two important contributions:

- A simple and natural, yet powerful and flexible representation of reinsurance portfolios as directed acyclic graphs. This representation is flexible enough to model essentially arbitrarily complex portfolios and allows reinsurance companies to build highly tailored and customized reinsurance solutions. In this representation, each vertex represents an individual term or clause of a contract contained within a portfolio, and each edge represents the flow of losses from one term or clause to another.

- A scalable cloud-based analysis platform capable of performing a location-level analysis of a reinsurance portfolio containing 70 million individual locations in approximately 17 minutes (when run on 40 compute instances each with 72 threads). The design uses stateless processing engines to facilitate availability, consistency and scalability, and makes extensive use of cloud services to reduce implementation complexity.

While improvements in Chapter 4 further reduce the running time of a location-level portfolio analysis from 17 minutes to approximately 10 seconds, the system described in this chapter already outperforms existing commercial analytics system by orders of magnitude—many of these systems are unable to perform less fine-grained analyses in 17 minutes—and, as we describe in Section 3.2, is significantly more flexible due to its graph-based representation of reinsurance portfolios.

As we discuss in detail in Section 3.3, a central technical challenge in designing our analytics platform was the development of an evaluation engine that inspects the vertices of the portfolio graph in an order that limits the amount of intermediate data any thread needs to hold in memory at any time. This ensures high computational throughput because it enables each thread on each compute instance to evaluate one trial (one of up to 10,000 portfolio evaluations as part of the Monte Carlo simulation) completely in memory, without interaction with other compute nodes and threads and without the need to access external storage during the evaluation process.

The remainder of this chapter is organized as follows: Section 3.1 discusses related work on large graph processing systems. Section 3.2 presents our graph-based portfolio representation. Section 3.3 discusses our implementation of a cloud-based risk analytics system based on the graph representation discussed in Section 3.2. Section 3.4 discusses experimental results that demonstrate the performance of our system and presents a comparison against a major vendor's commercial system. We offer concluding remarks in Section 3.5.

3.1 Related Work: General-Purpose Graph Modeling Frameworks

The modeling of complex dataflow problems as directed graphs, as we do in our graph-based portfolio representation in Section 3.2, and the development of distributed systems to evaluate such graph-based dataflow representations efficiently, as we do in Section 3.3, has been the focus of many previous works.

The idea of modeling computer programs as directed graphs began to gain popularity in the early 1970s as an attempt to describe a system that would make parallel computer programs easy to write while still highly scalable and efficient [30, 82]. Such dataflow systems would avoid the synchronization and data-race problems found on traditional systems by describing all instructions in terms of their data dependencies between each other, rather than by describing a particular order in which each instruction must be executed. In a dataflow system, each instruction in the program can only be executed if the data from the instructions it depends on have already been computed and stored. By describing a program in this way, instructions can be executed across multiple machines or cores in parallel in any order, so long as their input exists in memory. Conceptually, each instruction forms a vertex in a directed graph, and each data dependency between any two instructions forms a directed edge.

From this, several works emerged suggesting different computer architectures for dataflow processing systems [14, 31, 39, 70]. The token-based dataflow architecture presented in [31] proposes using "tokens" for storing the inputs and outputs for each function. Each token is associated with an edge (i.e. data dependency), and each edge may only have one token associated with it at any time. An instruction may only execute if a token is available for all of its incoming edges. When an instruction does execute, it deletes the tokens on its incoming edges and adds a token containing the result of the instruction on its out edge. Since each edge can only hold one token at a time, multiple iterations of a single loop cannot be processed in parallel, significantly reducing parallelism and consequently, performance [82].

The tagged-token dataflow architecture [14, 39] addressed this problem by allowing an unlimited number of tokens to be stored on each edge. Tokens are also tagged to describe the iteration the token is associated with. Once an instruction has a token for each edge with the same tag, it consumes the tokens and writes a token to its outgoing edge, tagged with the same iteration context.

In practice, tagged-token dataflow systems suffered from poor performance due to the overhead introduced by matching the tags of tokens at every instruction [82]. As a result, interest began growing in hybrid dataflow systems [43, 45, 67]. Instead of treating each instruction as an individual task to be scheduled, hybrid dataflow systems bundle mostly sequential series of instructions into tasks. This way, sequential strings of instructions can be executed efficiently on a von Neumann-like architecture, while the parallel execution and communication of these tasks can be planned using a dataflow system.

The core ideas of the hybrid dataflow approach can be found in several popular, relatively recent works regarding dataflow processing [7, 36, 37, 44, 57, 61, 65]. These systems forgo specialized hardware architectures, instead opting to use software to coordinate tasks in a dataflow graph across a cluster of commodity compute nodes. Like traditional dataflow systems, vertices represent computational tasks and edges represent data dependencies. Vertices are scheduled to run on the next available compute node or thread when all preceding vertices have been computed. Data is transferred from vertex to vertex through shared memory and network sockets. Also like traditional dataflow systems, these systems are designed to simplify writing highly performant parallel code by making parallelism implicit; users only need to submit a directed graph describing the program they want executed, and the dataflow software automatically schedules the work on compute nodes in a scalable way.

Dryad [44], one of the first of such systems, focused on improving scalability and availability over the existing hybrid dataflow model. It relies on the user to describe how individual instructions should be bundled together in the same vertex.

Systems like PowerGraph [36] and Pregel [61] operate on fine-grained directed graphs, where each vertex represents either a single instruction or a very simple operation. They reduce communication overhead by partitioning the graph into subgraphs that can be executed in parallel.

GraphX [37] attempts to bring the dataflow processing model to more general MapReduce-style systems, thereby retaining the flexibility of general dataflow systems while yielding performance similar to specialized graph processing systems for graph processing tasks. It is built as a library for Spark [99], which itself is a popular dataflow framework that expands upon MapReduce [26] by implementing a distributed data structure that can be cached for fast subsequent reads.

What all these systems have in common is that they are designed to schedule a single large computation (expressed as a directed graph) across multiple machines while optimizing load balancing and the amount of communication between compute nodes. In spite of their effectiveness for such problems, their focus on sophisticated scheduling and communication strategies introduces overhead that is unnecessary in the context of portfolio analysis. Since a portfolio analysis consists of running tens of thousands of trials *completely independently*, the reinsurance analytics problem is trivial to parallelize (barring memory constraints; see Section 3.3).

In response to the communication overhead in distributed graph processing systems, several recent works have described graph processing systems that run on a single compute node. GraphChi [51] is one such system that supports processing graphs with billions of vertices on a single compute node. It assumes that graphs are too large to fit into memory, and partitions the graph into multiple shards to be stored on disk. By intelligently ordering the edges within each partition, GraphChi attempts to minimize the number of non-sequential disk accesses as it iterates through the vertices of the graph during processing. By eliminating communication overhead, GraphChi is able to come close to the performance of distributed graph processing engines on large clusters. Notably, for computing 5 iterations of PageRank [69] on a social media graph, the authors observed that Spark was only twice as fast running on a 100 CPU cluster than GraphChi running on 2 CPUs.

Other papers present similar systems on the same premise, fast graph processing on a single machine with a disk-backed graph representation [42, 76, 81, 102, 104]. In some cases, these single-machine systems claim to outperform their distributed counterparts [102, 104].

These approaches vary primarily in how they handle the high cost of accessing external storage.

TurboGraph [42] uses thread-based parallelism to handle I/O and graph processing at the same time.

FlashGraph [102] attempts to improve performance by reducing disk I/O. For example, instead of storing the graph entirely on disk, FlashGraph stores vertex state in memory and only the graph's edge lists on disk.

GridGraph [104] reduces I/O by introducing a preprocessing step that partitions the graph into chunks small enough to be processed entirely in memory.

X-Stream [76] lays out its graph representation so that its edge lists are read with sequential memory accesses during processing, exploiting the performance benefits of sequential over random memory access.

Ligra [81] stores its graph entirely in main memory. Unlike the other systems, which are designed to run on small workstations with limited memory, Ligra is designed to run on more expensive servers with hundreds of gigabytes of memory.

All of these single-machine graph processing systems are focused on iterative processing, where the amount of data flowing across each edge of the graph is relatively small (e.g., PageRank). Location-level reinsurance modeling is different in that the data flowing across edges is much larger and not uniform. This makes memory an even scarcer resource than for general graph processing systems.

While our system differs significantly from the works above, it incorporates core aspects from both single-machine and distributed graph processing systems. Like the distributed graph processing systems discussed above, in order to maximize scalability and parallelism, our system runs multiple graph processing compute nodes in parallel. Unlike the distributed graph processing systems, each of our individual compute nodes stores its own complete copy of the entire graph, and independently processes an instance of the entire graph without any communication or synchronization with other workers, much like the single-machine graph processing systems. Because there is no communication between workers, each worker can compute their result with no idle cycles spent on network I/O or synchronization. This way, our system reaps the benefits of the distributed graph model (high scalability) and single-machine model (excellent individual worker performance) without the detriments of either (poor utilization of individual worker resources and limited parallelism, respectively). This is possible only because the work of a reinsurance analysis is broken into many Monte Carlo trials, each of which can be computed independently of each other.

3.2 Graph-Based Model for Reinsurance Analytics

As described in Section 2.1, the output of a reinsurance analytics job is a list of portfolio YELTs, one per trial, that make up the portfolio's loss distribution. Because trials can be evaluated independently, the core of the problem is computing the portfolio YELT of a single trial from the individual location YELTs.

This process can be represented quite naturally as a directed acyclic graph. The sources of the graph represent the inputs of the computation, that is, there is one source per location YELT. There is a single sink representing the output of the computation, that is, the computed portfolio YELT. Internal vertices represent terms and clauses of individual contracts and treaties. The output of each such vertex v is the sequence of losses incurred under this (part of a) contract or treaty, once again an YELT. This YELT becomes an input edge of every vertex that represents another (part of a) contract or treaty that covers the losses incurred under the treaty represented by v. Essentially, each vertex representing a term, contract or treaty takes a set of YELTs as input and produces a YELT containing the losses incurred under the contract for each relevant occurrence as output. This model provides a flexible tool for constructing complex reinsurance contracts and portfolios from a small number of basic building blocks:

Generator vertices have no input edges and output occurrences for insured locations. These occurrences are read from storage and are produced outside the analysis framework, e.g., using catastrophe modeling.

Commutative processing vertices apply a simple transformation to each occurrence, independent of the values of past occurrences (e.g., multiply each occurrence's loss by 0.5 to indicate that the treaty covers 50% of each incurred loss).

Aggregate processing vertices apply a transformation to each occurrence's loss dependent on the losses of past occurrences within the trial (e.g., when the sum of all processed losses exceeds a set dollar limit, discard the remaining losses.)

Filter vertices output occurrences from the input stream that match a specific condition (e.g., only output occurrences with sequence > 180—the second half of the year—or that represent losses due to hurricanes).

Merge vertices combine inputs from multiple YELTs and output them as a single YELT.

In practice, there are other, more specialized vertex types that are used infrequently or situationally. Within the context of this thesis, these offer little additional insight. Therefore, we ignore them for the sake of simplicity.

For some transformations, the order in which occurrences are processed influences the transformation (for example, an aggregate deductible may only deduct losses from the first ten occurrences). Therefore, YELTs must arrive in sorted order (that is, sorted by sequence), and be output in sorted order in order to be suitable for later transformation. We assume that the input YELTs of the portfolio are already sorted in this order. Each vertex ensures that its output YELT is in sorted order by processing each of its input occurrences in order and writing the resulting occurrences to the output YELT in the order in which they are produced. Merge vertices need to be more careful, taking occurrences from their input YELTs and merging them in sequence order. In the event that a merge vertex needs to merge two or more occurrences with the same sequence and event, they are merged into a single occurrence with their loss values summed.

Figure 3.1 illustrates a simple example portfolio modeled using our graph framework. The generator vertices each read a YELT from storage (according a file path assigned to them during the construction of the portfolio) and send the occurrences across their output edges. The merge vertices merge occurrences from their in-neighbors. In this example, the middle merge vertex combines the occurrences with `Sequence=2.0` and `Event=14` by summing their losses. The scale vertex scales each occurrence's loss according to its `rate` field. The final vertex in the graph merges everything together, and outputs the portfolio's final YELT.

While using a directed graph for modeling a reinsurance portfolio is natural, it has not been done before. As discussed in Section 2.2, previous reinsurance risk analytics systems have constrained themselves to nested or tiered structures. Many complex portfolios cannot be modeled using those systems. Directed graphs, on the other hand, allow the modeler to define arbitrarily complex structures from the elementary vertex types defined above. A typical financial contract can be modeled using 5–10 vertices.

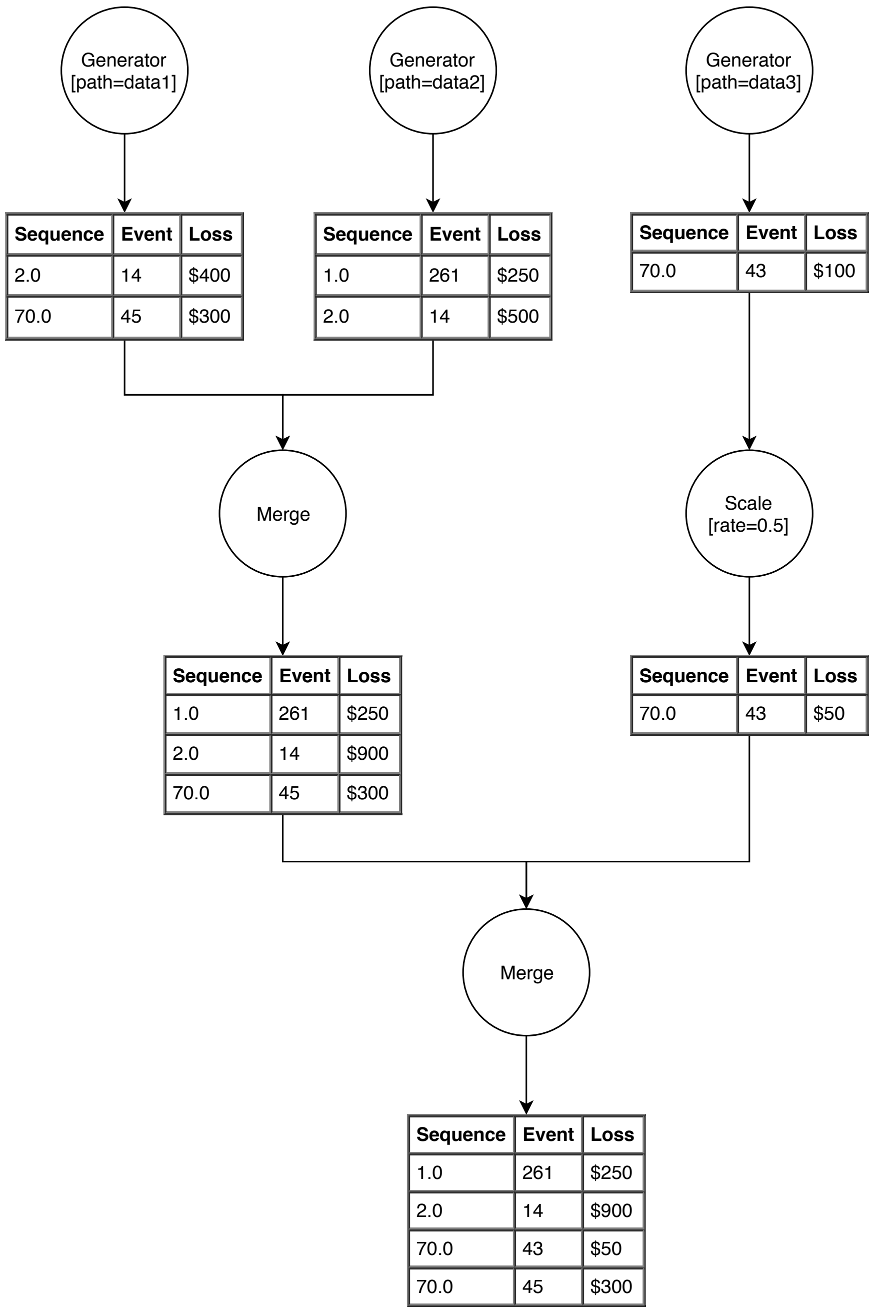

Figure 3.1: Processing YELT occurrences through a very simple example graph for one trial

3.3 A Cloud-Based System for Location-Level Risk Analytics

3.3.1 System Design

A typical reinsurance portfolio covers tens of millions of insured locations; each such insured location is represented by one input YELT per trial. These locations are covered by tens of millions of primary insurance contracts, which in turn are reinsured by thousands of reinsurance treaties. This is illustrated in Figure 2.2. Representing such a portfolio as a graph as in Section 3.2 results in a graph with hundreds of millions of vertices. Performing a location-level analysis of such a portfolio using the Monte Carlo simulation approach discussed in Section 2.1 requires processing several terabytes of data. Thus, to perform such an analysis quickly requires significant computational resources.

We chose to implement a cloud-based solution that distributes the computation across a large number of compute nodes. This provides scalability, elasticity, and availability of our system. Figure 3.2 gives an overview of the system's architecture. The client (a frontend through which the user interacts with the system) uploads the portfolio graph and input YELTs to a *distributed storage system*. To initiate an analysis, the client partitions the analysis into groups of trials to be analyzed and submits each trial group to the *occurrence processor queue*. This queue is responsible for assigning each trial group to the next available *occurrence processor*. Each occurrence processor runs on its own compute node, independently of other occurrence processors. This allows them to operate with *zero* communication between them. Occurrence processors write their results to distributed storage, from where they can be retrieved by the client.

We let the distributed storage system handle data consistency. In our implementation of this platform, we use Amazon S3 [11]. S3 guarantees read-after-write consistency [58] for all writes or updates to both new and existing objects [10]. Since our system reads all of the data required for analysis directly from S3, all reads in our implementation are consistent.

3.3.2 Overview of Optimizations

While this system design allows us to perform a portfolio analysis by evaluating individual trials completely independently on different occurrence processors, processing even a single trial through a portfolio graph of hundreds of millions of vertices and edges can require more memory than is available on a commodity compute node if done naïvely. Our solution is to

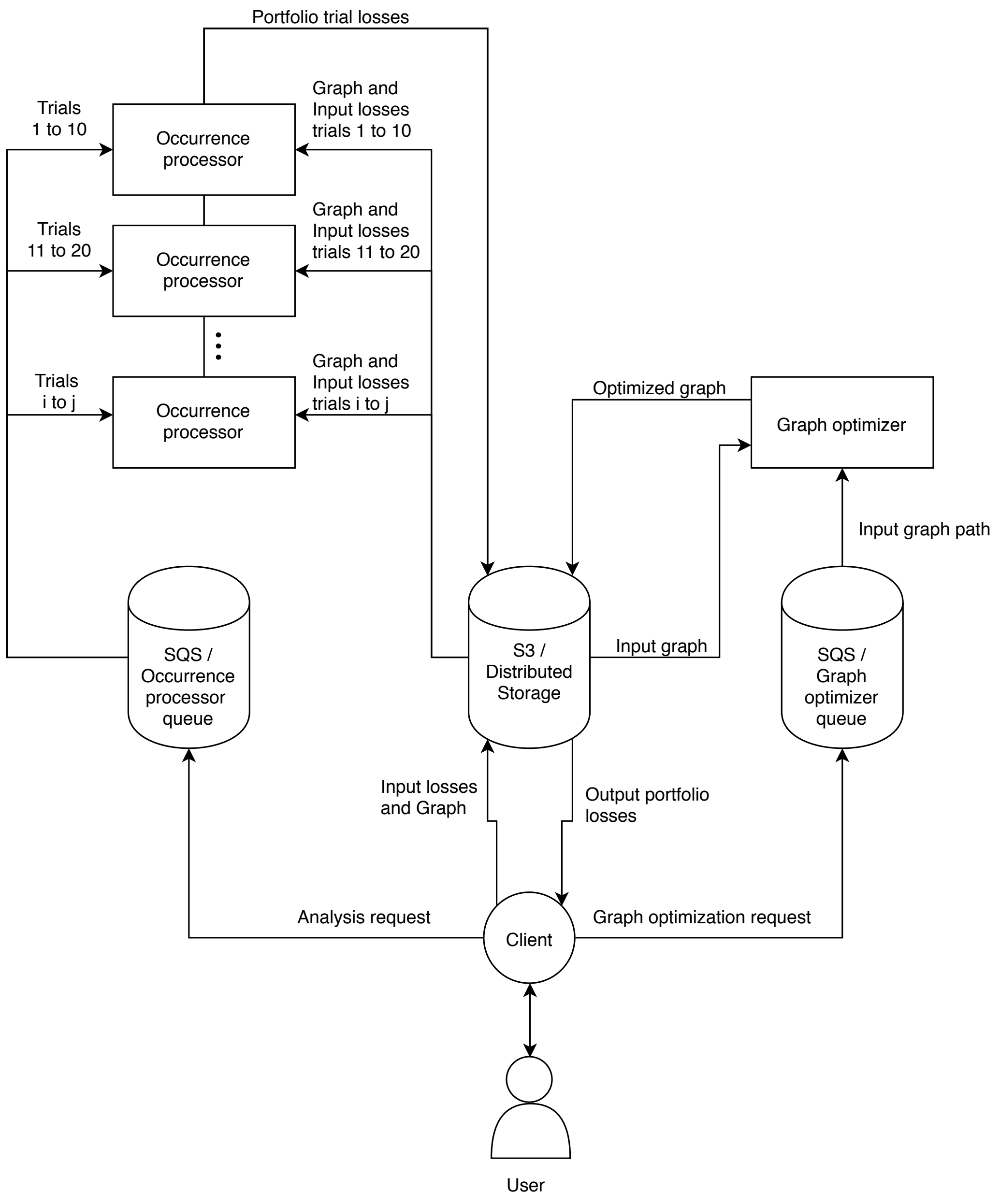

Figure 3.2: System architecture overview. Users upload data through a client to a distributed data store, and submit jobs to the system through two different message queues. Running a portfolio analysis involves submitting groups of trials to be analyzed to the occurrence processor queue, which then assigns these groups to available occurrence processors. Output is written back to the data store, where it can be read by the client. The efficient execution of the occurrence processors depends on the vertices of the portfolio graph being arranged in an order that allows them to be processed using little memory. Whenever the portfolio graph changes substantially due to the addition of new insured locations, new treaties, etc. the client can initiate the reoptimization of the graph layout by submitting a request to the graph optimizer queue. The graph optimizer then reads the graph from distributed storage, optimizes it, and writes it back to distributed storage. This graph optimization step is *not* part of the portfolio analysis but is an offline process to be run periodically.

implement a *graph optimizer* that ensures that vertices in the portfolio graph are evaluated in an order that minimizes the amount of memory needed for storing the YELTs sent between vertices. The result is that we can process 8 trials (one per core) entirely in memory on a single compute node using less than 32 gigabytes of memory.

The graph optimization is performed only periodically whenever the portfolio changes substantially due to the addition of new insured locations or treaties. Thus, it is an offline task that is *not* part of the portfolio analysis itself. Whenever the user wants to reoptimize the graph, the client submits a request to the graph optimizer through the graph optimizer queue in Figure 3.2. The graph optimizer then reads the graph from distributed storage, optimizes its vertex ordering, and writes the resulting rearranged graph back to distributed storage, for use by future analysis runs.

Since location-level graphs are very large and contain hundreds of millions of vertices and edges, and because we would like to store our graph entirely in-memory for performance reasons, we also optimize our memory footprint through careful selection of a space efficient graph data structure.

We split the discussion in the remainder of this section into four parts: Subsection 3.3.3 discusses the design of the occurrence processor. Subsection 3.3.4 discusses the implementation of the graph optimizer. Subsection 3.3.5 discusses our space-efficient graph representation. Subsection 3.3.6 offers some final remarks concerning the scalability, elasticity, and availability of our system design.

3.3.3 Occurrence Processor

The occurrence processor is the primary processing engine of the system. It is responsible for computing the portfolio YELTs of a group of trials from the location YELTs that form the input of the analysis for these trials. The occurrence processor starts one thread per each assigned trial to be evaluated. Typically, the number of trials assigned to an occurrence processor is at least the number of cores on the compute node running the occurrence processor, thereby allowing each core to run at least one thread.

The occurrence processor starts by loading the representation of the portfolio graph into memory. Since this representation is static, it can be shared by all threads analyzing individual trials and concurrent accesses to the graph by different threads do not require locking of the graph. Once the graph is loaded, each thread begins processing a single trial, traversing the vertices in the portfolio graph and producing the output YELT of each visited node from its input YELTs based on the vertex type (see Section 3.2).

Occurrence processors are stateless in the sense that their computations are completely determined by the trial groups they receive from the occurrence processor queue. Nothing is stored on local disk, so any occurrence processor can be scheduled to work on any group of trials. This is important from an availability perspective, as it ensures that no single occurrence processor is required to complete an analysis. Given a trial group, the occurrence processor reads the input graph and the relevant input YELTs for the trials in this group from distributed storage and, after completing its computation, writes the computed portfolio YELTs back to distributed storage.

Since the total amount of input data needed for a typical 10,000 trial location-level portfolio analysis is on the order of terabytes, and the combined size of intermediate YELTs generated by such an analysis is on the order of hundreds of terabytes, computing the portfolio's YELT is only possible through careful use of memory resources. For example, let us assume a portfolio graph of 300 million edges, where each edge represents an intermediate YELT with an average of only 10 occurrences (160 bytes) per trial. Storing all intermediate YELTs for a single trial would therefore have a memory footprint of 48GB. If we conservatively assume that storing the portfolio graph in memory consumes another 10GB, then the total memory usage required to process t trials with t threads would be $t * 48 + 10$ gigabytes. Aggressive deallocation of intermediate YELT data is therefore required to complete a location-level analysis without running out of memory. We considered two approaches to this problem, which we discuss next.

Edge-Buffered Approach

Our first approach avoids allocating space for a full YELT for each edge in the graph. Similar to using tokens to handle input and output from vertices in traditional dataflow processing [31], we allocate a small buffer per edge to hold only a portion of the YELT represented by this edge. The right mental model is that the buffer associated with an edge holds up to the next k occurrences in this YELT that have not been processed yet, for some small parameter k. To perform a portfolio analysis using this representation now requires a "pull-based" approach: To produce the output YELT of the portfolio, we compute each of the occurrences in this YELT from the corresponding occurrences in the input YELTs of the sink s of the portfolio graph.

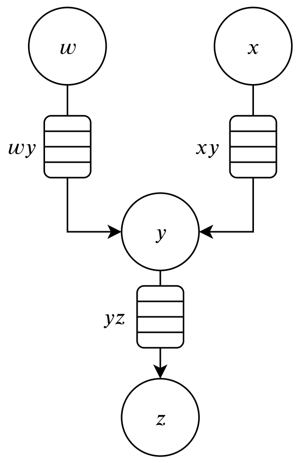

Figure 3.3: A simple graph of four vertices with a buffer on each edge

If the input occurrences necessary to compute the next output occurrence of s are available in the input buffers of s, we can remove them from the input buffers and compute the output occurrence from them. Once all occurrences in the YELT corresponding to an edge have been processed, the edge's buffer is considered to be *exhausted*. Since occurrences must be processed in order, all input buffers to a vertex must have at least one occurrence or be exhausted in order for the vertex to compute the next output occurrence. If not, at least one of the input buffers is empty but not exhausted and must be refilled before the vertex can consume any input occurrences. The input buffer corresponding to each in-edge is an output buffer of another vertex v in the graph, so we can apply this procedure recursively to v to produce enough occurrences in its output YELT to fill its output buffer. Once v's output buffer is full or exhausted, we can switch back to s to fill its output buffer. This process of recursively filling edge buffers *on demand*, as the next occurrence from a buffer is needed as input to a different vertex, continues until all of s's input buffers have been exhausted, and the final portfolio YELT has been computed.

Figure 3.3 illustrates a simple graph with a buffer at each edge. To compute the output occurrences of this portfolio, we pull occurrences from the terminal vertex z. Since z's input edge buffer *yz* is initially empty, this initiates a pull from vertex y. Vertex y pulls from the generator vertices w and x to fill its input edge buffers *wy* and *xy*. Vertices w and y generate their occurrences by reading them from disk. Once edge buffers *wy* and *xy* are full, vertex y consumes the occurrences from its input buffers, transforms them, and writes them to its output buffer *yz* until this buffer is full. At this point, the pull of vertex y has completed and

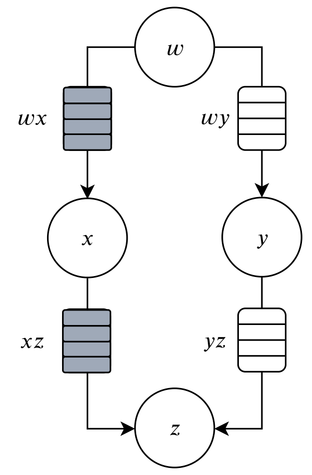

Figure 3.4: An example of deadlocking. Buffers *wx* and *xz* are full, while buffers *wy* and *yz* are empty.

vertex z can consume the input from buffer *yz*, transform it and write the output to disk. We continue to pull from vertex z until the generators are exhausted and all of their occurrences have passed through the graph.

This method bounds the maximum memory required to process a single trial by the number of edges multiplied by the buffer size of each edge. With a small enough buffer size, this method is efficient enough to meet our memory requirements. However, small edge buffers are likely to introduce deadlocks during processing.

Since occurrences must be processed in the order of their sequence field at each vertex, a vertex v cannot process occurrences in one input buffer *wv* if there are no occurrences in its other input buffer *uv* (unless the buffer of edge *uv* is exhausted). This is because v has no knowledge of the sequence field of the next occurrence in the YELT corresponding to *uv* and thus cannot decide whether to process the next occurrence in *uv* or the next occurrence in *wv* first. Thus, a vertex v can process occurrences in its input buffers only as long as none of these buffers is empty. Once an input buffer of v becomes empty, it has to be filled recursively before v can continue processing occurrences.

As Figure 3.4 illustrates, this can lead to deadlock. Vertex z cannot consume occurrences from buffer *xz* because buffer *yz* is empty. Buffer *yz* cannot be filled because buffer *wy* is empty. To fill buffer *wy* with occurrences requires w to produce more occurrences in its output YELT, but this cannot happen before there is room for more occurrences in buffer *wx*. Since buffer *xz* is full, x cannot remove any occurrences from buffer *wx* to make room for more occurrences in *wx*.

This could be addressed by allowing edge buffers to grow when necessary. However, this introduces performance overhead and weakens the bound on the memory usage of the occurrence processor guaranteed if all edge buffers have a fixed small size.

Our experiments discussed in Section 3.4.2 demonstrate that this potential deadlock is not just a hypothetical limitation of the edge-buffered approach. Deadlocking is common for graphs that contain filter vertices, as filtering occurrences has a tendency to make the number of occurrences in nearby buffers imbalanced. The edge-buffered approach with buffers small enough to fit in memory was not able to process our test portfolio graph with filter vertices due to deadlocking. Our second approach overcomes this limitation.

Low-Cutwidth Ordering

If each edge buffer uv is large enough to hold the entire YELT produced by u, no switching back and forth between vertices is required; vertices can be processed in topologically sorted order, which guarantees that the input YELTs of each vertex are available when visiting this vertex to produce its output YELT, thus making deadlocks impossible.

Once produced, the output YELT of a vertex u needs to be kept in memory until the last vertex is processed that has this YELT as one of its input YELTs. Thus, a natural strategy to minimize the memory requirements of the occurrence processor is to allocate space for the output YELT of u when processing u and to discard the YELT once the last vertex has been processed that has it as one of its input YELTs.

The effectiveness of this strategy depends on the order in which the vertices of the portfolio graph are processed. As an example, consider a portfolio graph that is a complete binary tree. There are many valid topological orderings of this graph. If vertices are visited by decreasing distance from the sink, then the output YELTs of all source vertices have to be held in memory simultaneously because they are all evaluated before any of their out-neighbors. Since half of the vertices in a complete binary tree are source vertices, this means that half of all YELTs must be in memory simultaneously. On the other hand, if vertices are visited in postorder (all vertices in each subtree are visited consecutively), only $\log_2 n$ YELTs need to be in memory at any point in time, where n is the number of vertices in the graph. This is because at most one input YELT per ancestor of the currently visited vertex needs to be held in memory.

Formally, if the topological ordering of the graph arranges the vertices in the order $v_1, \ldots, v_n$,

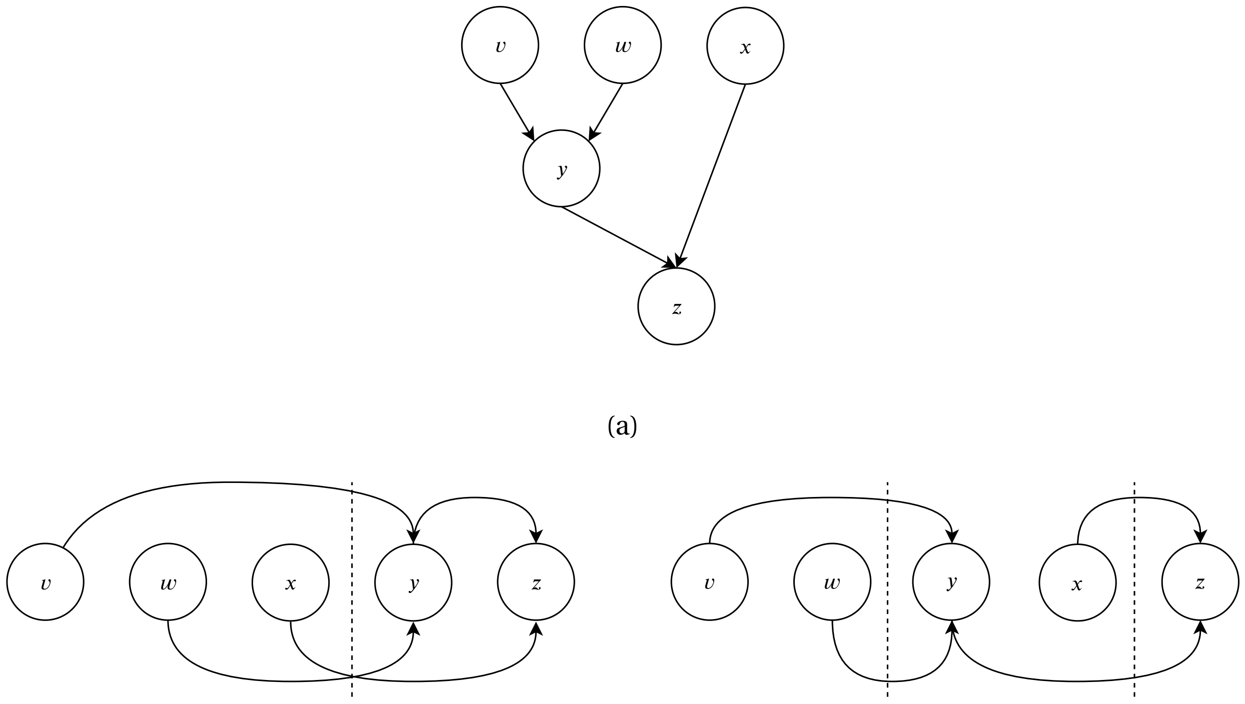

Figure 3.5: Illustration of the cutwidths of topological orderings. Both orderings in (b) and (c) are valid topological orderings of the graph in (a). The ordering in (b) has cutwidth 3, as indicated by the dashed line, which is crossed by the edges *vy*, *wy*, and *xz*. The ordering in (c) has cutwidth only 2, as indicated by the two dashed lines, which are crossed by edges *vx* and *wx*, and *xz* and *yz*, respectively.

then the YELTs that need to be in memory immediately after processing the ith vertex v_i are the ones corresponding to edges $v_j v_k$ with $j \leq i$ and $k > i$. The maximum number of YELTs to be held in memory at any time over the course of the algorithm is then

$$\max_{1 \leq i < n} |\{v_j v_k \in E \mid j \leq i < k\}|.$$

We call this the *cutwidth* of the topological ordering in analogy to the cutwidth of an undirected graph [48] and say that an edge $v_j v_k$ with $j \leq i < k$ "crosses the cut between v_i and v_{i+1}." Figure 3.5 illustrates that different topological orderings of the same graph may have different cutwidths.

Our strategy to minimize the memory requirements of evaluating a single trial is to find a topological ordering of low cutwidth and then process the vertices of the graph in this order. Finding such a low-cutwidth ordering is the task of the graph optimizer, discussed next.

3.3.4 Graph Optimizer

The graph optimizer reads the portfolio graph from distributed storage, computes a low-cutwidth topological ordering of the graph, modifies the structure of the graph to further reduce cutwidth (see below), and writes the result back to storage, to be read by the occurrence processors. Because all of the inputs needed by the graph optimizer are received from the message queue and distributed storage, the graph optimizer is stateless, which allows the graph optimizer to be provisioned on-demand, just as the occurrence processors.

Finding a vertex ordering of *minimum* cutwidth is NP-hard even for undirected graphs [48]. Fixed-parameter algorithms for computing the cutwidth of an undirected graph [34, 86] and a polynomial-time algorithm to approximate the cutwidth of an undirected graph [56] have been proposed in the literature. However, the running times of these algorithms are far from linear. Thus, even if we were able to extend these algorithms to directed graphs, they would not be efficient enough to apply them to portfolio graphs with hundreds of millions of vertices. Instead, we use a heuristic approach that exploits the structure of portfolio graphs to compute low-cutwidth topological orderings for these graphs. This heuristic is not guaranteed to find a *minimum*-cutwidth topological ordering of the portfolio graph nor does it give any approximation guarantee of the cutwidth of the computed topological ordering. It does, however, find topological orderings of sufficiently low cutwidth to lead to low memory requirements of the occurrence processor, and it does find them quickly (in linear time).

The heuristic proceeds in two phases: The first phase finds an initial topological ordering of the given portfolio graph. The second phase modifies the graph and the topological ordering computed in the first phase to reduce the cutwidth of the ordering further. The modifications made to the portfolio graph in the second phase do not alter the portfolio structure it represents.

The structure of a typical portfolio. A typical location-level portfolio covers tens of millions of locations, each represented by a YELT containing the losses incurred for this location. Each of these locations is covered by *one* of tens of millions of primary insurance contracts. The losses incurred under these contracts are covered by a complex network of several thousand reinsurance contracts.

Modeling the connections between insured locations, primary insurance contracts, and reinsurance treaties results in a very tree-like graph, with a densely connected graph of a few thousand vertices representing reinsurance treaties at the root and large trees of primary insurance contracts and individual locations attached to it.

Each individual primary insurance contract or reinsurance treaty can be modeled as a graph of typically 5–10 of the basic vertex building blocks described in Section 3.2. Even if these graphs are densely connected internally, the resulting portfolio graph remains very tree-like; it has a large 2-edge-connected component close to the sink, composed of several thousand vertices, and is otherwise composed of fairly small 2-edge-connected components containing at most a few dozen vertices. A *2-edge-connected component* of a graph is a maximal subgraph that cannot be disconnected by removing a single edge.

While our hope is that the flexible graph-based representation of reinsurance portfolios introduced in this thesis will allow users to model more complex and fine-tuned portfolio structures than are in use in the industry today, we believe that the structure of reinsurance portfolios will remain largely hierarchical, as discussed above, so portfolios should continue to be composed of many fairly small 2-edge-connected components and one or *very* few larger 2-edge-connected components close to the sink. This is the portfolio structure our graph optimizer exploits.

The initial topological ordering. Recall the example of a low-cutwidth ordering of a complete binary tree in Section 3.3.3. Arranging the vertices in postorder resulted in an ordering with cutwidth $\log_2 n$, a significant improvement over the cutwidth $n/2$ achieved by the worst-case ordering (a random ordering will be close to the worst case). Given the tree-like structure of reinsurance portfolios, this suggests the following simple strategy for computing a low-cutwidth topological ordering of a portfolio graph: reverse the directions of all edges in the portfolio graph and perform a depth-first traversal (DFS) of the graph starting at the sink; arrange the vertices in postorder of the resulting DFS tree, that is, in the order the DFS backtracks from them.

The cost of computing a topological ordering in this fashion is linear in the size of the portfolio graph; indeed, this DFS-based approach to topological ordering is one of the classical linear-time topological sorting algorithms [25, p. 612–613]. If the tree of 2-edge-connected components is fairly balanced, which tends to be true for reinsurance portfolios, then the computed ordering ensures that, for any point in the ordering, there are only logarithmically many 2-edge-connected components that are “active” (that is, have at least one vertex both prior and subsequent to the current point in the ordering). More precisely, for every vertex v_i in the computed ordering $v_1, \ldots, v_n$, the set of edges $v_j v_k$ that satisfy $j \le i < k$ are the in-edges of vertices in only a logarithmic number of 2-edge-connected components. Since almost all 2-edge-connected components are small, this implies that the set of edges $v_j v_k$ with $j \le i < k$ are the in-edges of roughly a logarithmic number of vertices in the portfolio graph. If these vertices have low in-degree, the topological ordering thus has low cutwidth. Some vertices, however, can have very high in-degree. The second phase of the graph optimizer now reduces the degrees of high-degree vertices without changing the portfolio structure represented by the graph, in order to obtain the final topological ordering of low cutwidth.

Degree reduction. Since the cutwidth of the DFS-based topological ordering can be significantly impacted by the graph's maximum vertex degree, the degree reduction phase of the graph optimizer transforms the graph so that its maximum in and out-degree is less than or equal to some parameter d by adding trees of mergers around high degree vertices. The optimal choice of d is determined experimentally in Section 3.4. The graph optimizer does this without changing the semantics of the portfolio; that is, the YELT produced at the sink from a degree-reduced graph is the same as the YELT produced by a non-degree-reduced graph.

To see how this can be done, consider a vertex v with high out-degree. A vertex with high out-degree must create a copy of its output YELT for each of its outdeg(v) successors. However, instead of immediately creating outdeg(v) copies once v's YELT has been constructed, copies could be made in a tree-like manner, making few copies initially and replicating each copy further as needed. High in-degree vertices can be reduced in a similar manner. Considering the vertex types used as building blocks discussed in Section 3.2, the only vertices that can have high in-degree (or any in-degree greater than 1) are merge vertices. Thus, a single high-degree merge vertex can be replaced with a tree of d-way merge vertices. In fact, the efficient implementation of a merge vertex becomes easier if it merges a small number d of YELTs instead of a potentially large number of YELTs. For this to reduce the cutwidth of the topological ordering, however, the construction of the merge tree needs to be informed by the current topological ordering, and the vertices this introduces need to be placed carefully into this ordering.

For every high-degree vertex v_i, we replace its in-edges with an "in-tree" that has some

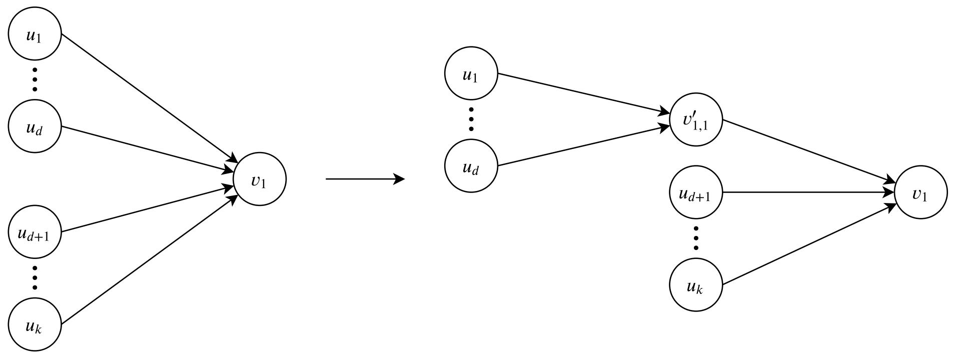

Figure 3.6: Introduction of a single vertex of the in-tree of v_1

new vertices $v'_{i,1},\dots,v'_{i,k}$ as internal vertices. Every in-edge of v_i gets connected to a leaf of this in-tree. At the root of the tree is v_i, and every vertex $v'_{i,j}$ has its parent in the in-tree as its only out-neighbor. This in-tree can have many different shapes; we discuss our choice of its shape below, after discussing how we modify the topological ordering of the graph to reflect the addition of the vertices $v'_{i,1},\dots,v'_{i,k}$.

To update the topological ordering, we can think about the construction of a vertex v_i's in-tree as repeatedly taking a group of d in-neighbors $u_1,\dots,u_d$ of v_i and replacing their out-going edges to v_i with out-going edges to a new vertex $v'_{i,j}$. The vertex $v'_{i,j}$ is then given an out-going edge to v_i. See Figure 3.6 for a graphical representation where a single vertex is added to reduce v_i's in-degree from k to $k-(d-1)$. Every time a vertex is added during degree reduction, it must be placed within the ordering so that the ordering is still topological, and so that the placement minimizes the cutwidth of the ordering.

To ensure that the ordering is still topological after placement of the new vertex $v'_{i,j}$, $v'_{i,j}$ must succeed its in-neighbors $u_1,\dots,u_d$ and precede its out-neighbor v_i. If the in-neighbor that appears last in the topological ordering is u_d, then placing $v'_{i,j}$ anywhere after u_d but before v_i maintains the topological ordering. This is illustrated in Figure 3.7.

Next assume that u_d is the hth vertex in the topological ordering. Then before adding $v'_{i,j}$ to the graph, the edges between $u_1,\dots,u_d$ and v_i cross the cut between $v_{i'}$ and $v_{i'+1}$ for all $h \le i' < i$. If we add $v'_{i,j}$ between v_k and v_{k+1}, where $h \le k < i$, then the number of edges crossing the cut between $v_{i'}$ and its successor is not changed for $h \le i' \le k$. For $k < i' < i$, the d edges between $u_1,\dots,u_d$ and v_i crossing the cut between $v_{i'}$ and its successor are replaced with a single edge between $v'_{i,j}$ and v_i. Thus, the number of edges crossing the cut between $v_{i'}$ and its successor is reduced by $d-1$. This is also illustrated in Figure 3.7. To maximize the

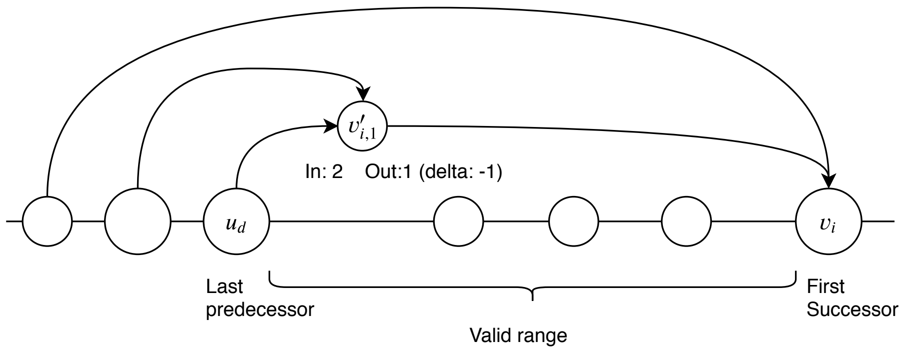

Figure 3.7: An illustration of the placement of an in-degree reduction vertex $v'_{i,1}$ with $d = 2$. The number of edges between $v'_{i,1}$ and the vertex it is placed next to will always be $d-1$ edges smaller than the number of edges between $v'_{i,1}$ and the vertex preceding it.

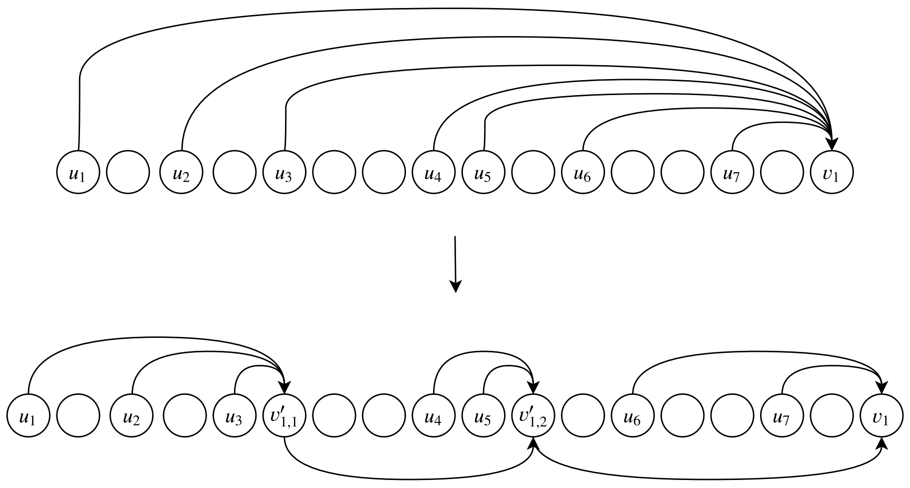

Figure 3.8: A caterpillar contributes at most d to the cutwidth at any point between v_i's first in-neighbor and v_i in the topological ordering. Here, $d = 3$.

benefit of adding the vertex $v'_{i,j}$, we should therefore add $v'_{i,j}$ immediately after u_d.

With this strategy for updating the topological ordering, the best tree topology is a caterpillar (Figure 3.8): Order the in-neighbors $u_1, \ldots, u_k$ of v_i in the order they appear in the current topological ordering. Then take the first d of them and make them children of a new vertex $v'_{i,1}$, inserted in the ordering right after u_d. Next take $v'_{i,1}$ and the next $d-1$ vertices $u_{d+1}, \ldots, u_{2d-1}$ and make them children of a new vertex $v'_{i,2}$, inserted right after u_{2d-1}. Continue in this fashion, making the last vertex $v'_{i,j}$ and the next $d-1$ in-neighbors of v_i children of the next vertex $v'_{i,j+1}$, until all in-neighbors of v_i have been consumed. This guarantees v_i's in-tree contributes at most d to the cutwidth at any point in the ordering between the first predecessor of v_i and v_i (and zero everywhere else).

However, we did not choose a caterpillar in our implementation and decided to make each vertex's in-tree a balanced d-ary tree. Observe that the cutwidth of the topological ordering is only a proxy for estimating the memory requirements of processing the portfolio graph; the underlying assumption is that all YELTs have roughly the same size or at least that there exists a modest upper bound on the size of every YELT. When merging many YELTs into one, this assumption holds true only if many occurrences in the merged YELTs refer to the same event and thus are combined into a single aggregate occurrence in the output. If this happens, the caterpillar is the best possible topology for the in-tree of a high-degree vertex. We choose a balanced d-ary tree as the topology of each in-tree in order to guard against the possibility that only few occurrences merge during the merge process. In this case, degree reduction does not help at all to reduce this vertex's contribution to the memory requirements of processing the portfolio graph; the final merged YELT uses just as much space as the input YELTs combined, the same as holding all these input YELTs in memory at the same time. In such a scenario, the caterpillar topology hurts performance because it does not reduce the memory requirements but forces most occurrences to participate in a linear number of merge steps (in the in-degree of v_i). A d-ary tree reduces the number of merge steps each occurrence participates in to logarithmic while at the same time achieving a cutwidth that is by at most a logarithmic factor in the maximum degree greater than the cutwidth achieved using caterpillars.

3.3.5 In-Memory Graph Data Structure

Since our goal is to run each occurrence processor on a commodity compute node and the simultaneous processing of multiple trials on an occurrence processor requires substantial amounts of memory, even using the low-cutwidth topological ordering discussed so far in this section, we need a highly compact representation of the portfolio graph to allow us to use most of the available memory for storing intermediate YELT data.

In order to maximize performance of the graph optimizer and occurrence processor, the data structure used to represent the graph must be entirely in-memory. We use a modified version of the *Compressed Sparse Row* (CSR) graph representation [77, p. 84–85]. A CSR graph representation is essentially a space efficient adjacency list using two contiguous arrays in memory: an array of vertices and an array of successors. Each element in the successor array is a reference to a vertex in the vertex array. Each vertex contains a pointer to a position in the successor array, which indicates the beginning of the vertex's successor subarray. The vertex's

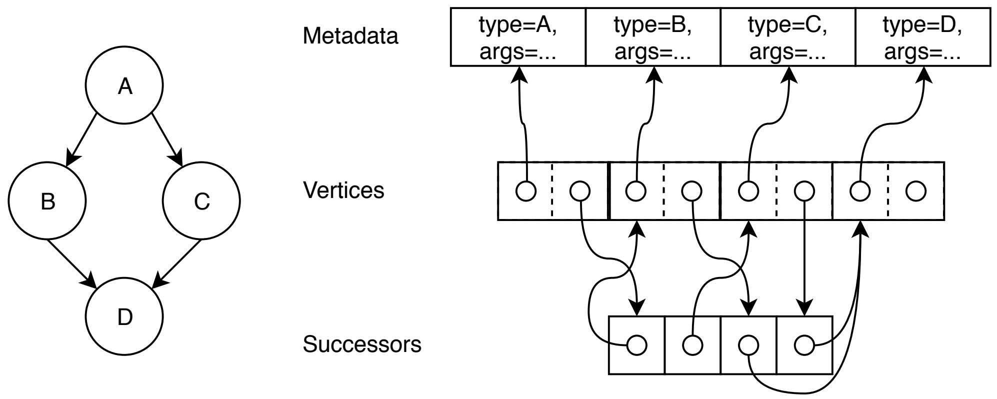

Figure 3.9: A sample graph and in-memory representation. The predecessor array is omitted for readability.

successor subarray ends where the following vertex's successor subarray begins.

In our modified CSR representation, we add a metadata vector, which describes the type of financial transformation each vertex performs (e.g., scale each incoming occurrence by a given participation rate), and the parameters needed by this vertex (the participation rate in this example). The blocks of metadata information associated with all vertices are packed tightly into this vector and each element in the vertex array points to the start position of the vertex's metadata block in this byte vector. Since different vertices store different amounts of metadata, the size of the metadata byte vector is not known in advance (in contrast to the sizes of the vertex and successor arrays, which can be computed from the numbers of vertices and edges in the graph). We allocate a modest amount of space for the metadata byte vector initially and reallocate it, doubling its capacity, every time we do not have enough space for the next metadata block.

Since the graph optimizer needs to be able to efficiently traverse edges in both directions, we also add a predecessor array to our modified CSR graph representation. This results in all edges being stored twice.

A simplified version of our modified CSR graph data structure is illustrated in Figure 3.9. In this figure, the predecessor array is omitted for readability.

3.3.6 Scalability, Availability and Hiding I/O Cost

The system design described in this section facilitates scalability and availability. In theory, we can allocate a number of occurrence processors up to the number of trials we need to evaluate, having each occurrence processor evaluate only one trial. This provides practically unlimited scalability—in theory.

The implementation of occurrence processors and graph optimizers as stateless processes without any communication between them supports availability. To respond to changing system loads, for example when performing more than one portfolio analysis simultaneously, it is easy to provision additional occurrence processors and graph optimizers and deprovision them when they are no longer needed. Similarly, by monitoring the time it takes for each occurrence processor to produce its output after receiving a trial group from the occurrence processor queue, we can detect unresponsive occurrence processors and resubmit their work to a different occurrence processor, assuming the unresponsive occurrence processor has failed. This allows for high-availability without significant implementation effort. In fact, queuing services like Amazon Simple Queue Service (SQS) provide this as a built-in feature.

The statelessness of the occurrence processors and graph optimizers comes at the cost of downloading all required input at run time. Each graph optimizer must download the entire graph, while each occurrence processor must download the graph and all of the relevant input loss data. This limits the scalability of the system as we discuss next.

Provisioning additional occurrence processors beyond a certain point increases the cost (charged by the cloud service provider for renting the required compute instances) of running a portfolio analysis while yielding only a minimal decrease in running time. To process a given trial, the occurrence processor needs the portfolio graph and the input YELTs of the trial. As we allocate fewer trials to each individual occurrence processor, the cost of loading the portfolio graph into memory will become more and more significant relative to the computation cost necessary to produce the portfolio YELT. Thus, we need to ensure we assign a sufficient number of trials to each occurrence processor to amortize the cost of loading the portfolio graph (loaded only once and shared between all threads of the occurrence processor).

The I/O cost of loading the input YELTs for all trials cannot be amortized by increasing the number of trials evaluated by each occurrence processor because each trial needs its own set of input YELTs. The total size of these YELTs is far greater than the size of the portfolio graph, so mitigating the performance impact of loading input YELTs is important. We do this by assigning more trials than there are cores on a compute node to each occurrence processor. This allows us to start the first batch of trials after loading the portfolio graph and their input YELTs. While this first batch of trials is being evaluated, a dedicated I/O thread of the occurrence processor loads the input YELTs of the next batch of trials into memory. As our experiments in Section 3.4 show, the time it takes to load a trial's input YELTs is slightly lower than the time to evaluate a trial. Thus, the input YELTs for the next batch are available by the time the occurrence processor finishes evaluating the previous batch. With a sufficient number of batches to be evaluated on each occurrence processor, this allows us to hide most of the I/O costs incurred by loading trial YELTs.

3.4 Evaluation

In this section, we evaluate the performance of our location-level analysis platform. We first describe the structure of the portfolio to be used in our experiments in Section 3.4.1. Section 3.4.2 compares the cutwidth-based processing approach against the edge-buffered approach, concluding that the cutwidth-based approach is superior in practice. The remaining experiments focus on choosing optimal parameters for the cutwidth-based approach and evaluating the performance of the whole system for running a complete location-level portfolio analysis using 10,000 trials. In Sections 3.4.3 and 3.4.4, we present a series of single-trial tests to analyze the performance of both the graph optimizer and the occurrence processor depending on the maximum vertex degree used in the degree reduction step of the graph optimizer. These tests were run in isolation on a single workstation with an Intel i7-6700K CPU, 64GB of RAM, and with an SSD as the storage system for input and output. It is reasonable to expect that the optimal parameters determined in these single-trial tests are also optimal in a full-scale run on thousands of trials because trials are evaluated completely independently on separate threads or compute nodes. In Section 3.4.5, we tested the parallel performance of our system by running a 10,000 trial evaluation using 40 c5.18xlarge compute nodes (each with 72 vCPUs and 144GB of memory) on Amazon EC2 [9] and S3 as the distributed storage system [11]. Finally, Section 3.4.6 compares the performance of our system against a commercial system on the market today.

3.4.1 Test Portfolio

Since current commercial systems are unable to perform a full portfolio analysis at the resolution of individual locations in a reasonable amount of time, there do not exist any real-world location-level portfolio data to date that could be used in experiments to evaluate our system. Therefore, we are limited to using synthetic data.

We constructed our test data set from the portfolios of primary insurers and from the portfolio of a reinsurer composed of treaties with primary insurers and insurance contracts for high-value individual properties.[1] This portfolio structure is illustrated in Figure 3.10.

High-value individual properties include bridges or office towers worth hundreds of millions of dollars. Such properties are insured directly by reinsurance companies. The contract insuring each such property is modelled using a subgraph of approximately 50 vertices, with fairly high connectivity near the source and sink of the subgraph. There are 59 such structures in our test portfolio.

Each primary insurer business unit covers 100,000 insured locations and is composed of approximately 400,000 vertices. In this structure, the contract for each location is modelled using a subgraph of 4–5 vertices. Figure 3.11 illustrates this structure (scaled down to only 5 locations in order to fit on the page). The losses from contracts are combined into the insurer's loss YELT using a high-degree merge vertex. Our graph models 700 different primary insurance portfolios, making these structures the bulk of our graph.

The "reinsurer's contractual terms" structure serves as the sink of the graph and takes the losses from the primary insurer business units and high-value properties as inputs. This structure contains approximately 5,000 vertices and models interdependent reinsurance contractual terms for the entire business of a real reinsurance group. The structure includes several large merge vertices, one with in-degree over 1,000 and several with in-degree over 100, making it the most complex component of the graph in terms of connectivity.

Overall, our test graph had approximately 307M vertices and 377M edges. Its structure reflects the flow of risk from individual insured locations via primary insurance contracts to reinsurance treaties and thus should be representative of location-level reinsurance portfolios that we expect to emerge in the real world once systems such as ours make location-level portfolio analysis feasible.

3.4.2 Edge-Buffered vs Cutwidth-Based Approach

The experimental evaluation of our prototype implementation of the edge-buffered approach highlighted its practical deficiencies. Deadlocking was common for graphs that contain filter vertices, as filtering occurrences has the tendency to make the number of occurrences in nearby buffers imbalanced. The edge-buffered approach was not able to process our

[1]Due to confidentiality requirements, the specific companies cannot be identified.

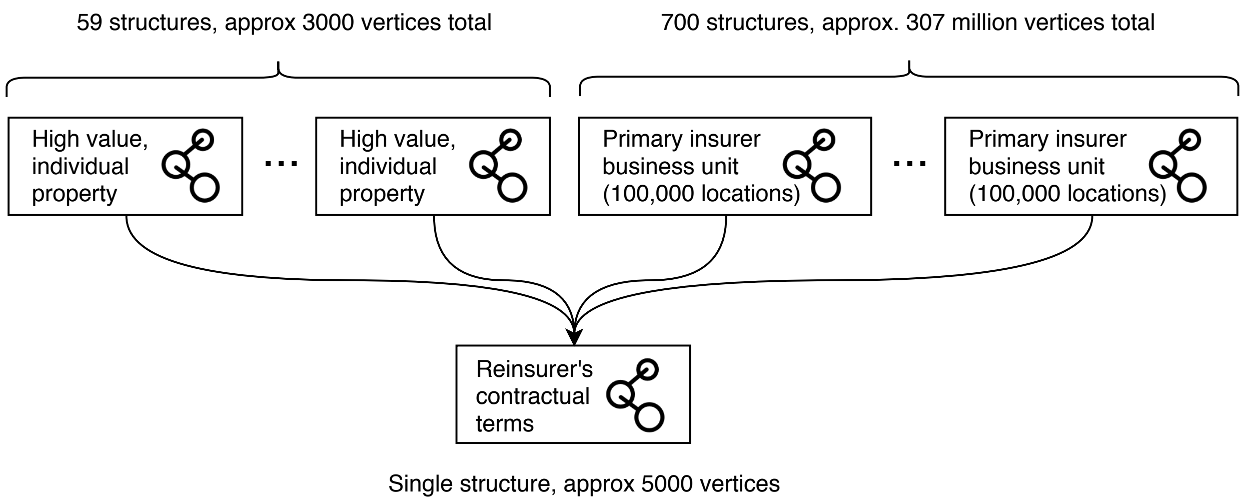
59 structures, approx 3000 vertices total
700 structures, approx. 307 million vertices total
High value, individual property
...
High value, individual property
Primary insurer business unit (100,000 locations)
...
Primary insurer business unit (100,000 locations)
Reinsurer's contractual terms
Single structure, approx 5000 vertices

Figure 3.10: Simplified view of the location-level graph used for evaluation of our system

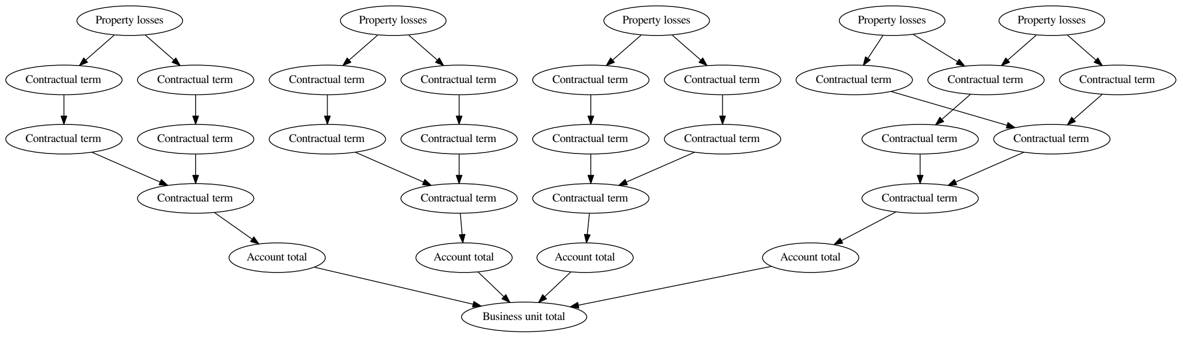

Figure 3.11: A graph representing the portfolio of a primary insurer insuring 5 properties

test portfolio graph with filter vertices present without deadlocking. Deadlocking could be mitigated or even avoided using larger edge buffers, but because a typical graph in our use case has hundreds of millions of edges, the amount of memory we were able to devote to each edge was heavily constrained. This was the case especially during parallel trial processing, where each thread needed its own unshared set of edge buffers. Deadlocks could also possibly be resolved by using sophisticated deadlock detection and resolution strategies. However, such strategies would be highly non-local (i.e., would need to analyze large subgraphs) and would introduce significant performance overhead.

Deadlocking made it impossible to evaluate a single trial on the full portfolio graph using the edge-buffered approach. To obtain a running time comparison between the edge-buffered and cutwidth-based approaches, we evaluated both methods on a reduced input graph obtained by reducing the number of locations in each business unit to 10,000, which resulted in a graph that was about 90% smaller than the full portfolio graph. We also removed some of the filtering structures that were a contributing cause of the deadlocks. We call the resulting graph the *simplified graph* in the following discussion.

It took approximately 6.5 seconds (excluding the time taken to load the graph) to process a single trial of 1.5 million occurrences through the simplified graph using edge buffers able to hold up to 4 occurrences. Using an edge buffer size smaller than 4 occurrences resulted in deadlocking even on the simplified graph. Increasing the edge buffer size to 24 occurrences had no noticeable impact on performance. For edge buffers larger than 24 occurrences long, the occurrence processor began to run out of main memory and began to use swap space, resulting in a sharp increase in running time. Storing a 4-occurrence edge buffer at each of the 37 million edges in the simplified graph required approximately 2.4 gigabytes of memory.

For comparison, using the cutwidth-based approach and the same simplified input graph, it took 16 seconds to optimize the graph and find the ordering, and 6.3 seconds to process the trial. The optimization of the graph needs to be performed only once; all trials use the same optimized graph. This amortizes the 16 seconds over the total number of trials. The ordering resulted in a cutwidth of 300 and, correspondingly, only 10 megabytes of memory being used to store intermediate YELTs during occurrence processing of a single trial.

Since the running times of the two methods were comparable but the cutwidth-based approach both used significantly less memory and did not suffer from deadlocks, we conclude that the cutwidth-based approach is the better choice and is worth the initial cost of finding a low-cutwidth topological ordering of the graph and performing degree reduction.

3.4.3 Graph Optimizer Evaluation

Figure 3.12 illustrates how the choice of the maximum degree d during the degree reduction step affected the cutwidth and size of the optimized graph. On the x-axis, the maximum degree d of the output graph is specified. On the y-axis, we plot the size of the output graph (on the solid line), and the cutwidth of the output graph (on the dotted line). Setting the maximum degree to infinity tells the graph optimizer to essentially skip the degree reduction step. This resulted in a cutwidth of 10,000 and increased the amount of memory required during occurrence processing but left the number of vertices in the graph unchanged. Conversely,

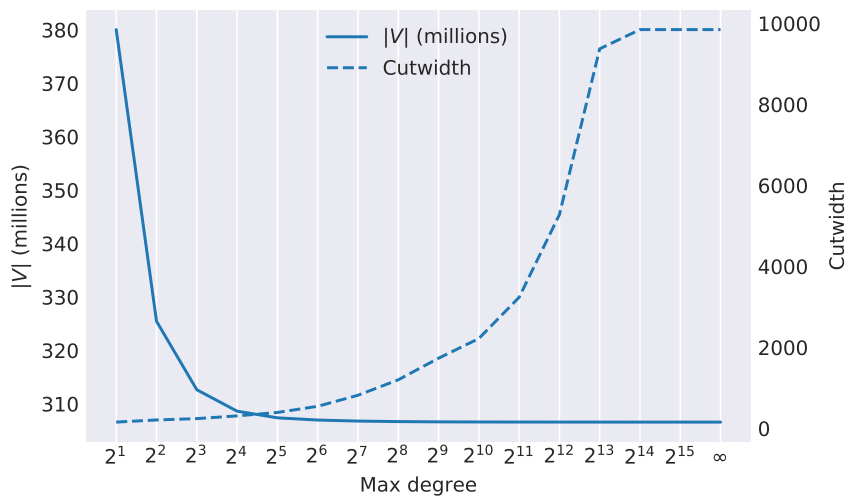

Figure 3.12: Number of vertices vs cutwidth in the optimized graph for different values of the max degree parameter

setting the maximum degree to 2 split all vertices with in or out-degree greater than 2. This resulted in a very low cutwidth of 157 but also in an added 75 million vertices. This is significant because a substantial increase in the number of vertices increases the cost of processing a trial, as discussed in Section 3.4.4. For this particular graph, setting the maximum degree to 16 resulted in a cutwidth of approximately 310, thereby achieving a very small memory footprint while increasing the number of vertices in the graph by only approximately 2%.

In Figure 3.13, we show the breakdown of the running time of the graph optimizer for different values of d. For all but very large maximum degrees, the reduction time dominated the running time of the graph optimizer. Setting the maximum degree to 2 resulted in a significantly larger graph (as shown in Figure 3.12) and in a corresponding noticeable jump in I/O and degree reduction time. For $d \geq 2^{14}$, the reduction time dropped to almost zero. It was at this point that no degree reduction happened at all, and the only contribution of degree reduction to the total running time was scanning the in and out-degrees of the vertices in the graph, only to realize that they were all below d. When this happened, reading the initial graph file, and writing it back out in topological order became the new dominating cost of the graph optimizer. The process of finding the initial DFS ordering was relatively inexpensive and, unsurprisingly, remained constant regardless of the maximum degree bound, as the ordering was computed before degree reduction. I/O time includes the time taken to read the input graph (approximately 30 seconds), and write the output graph (approximately 20

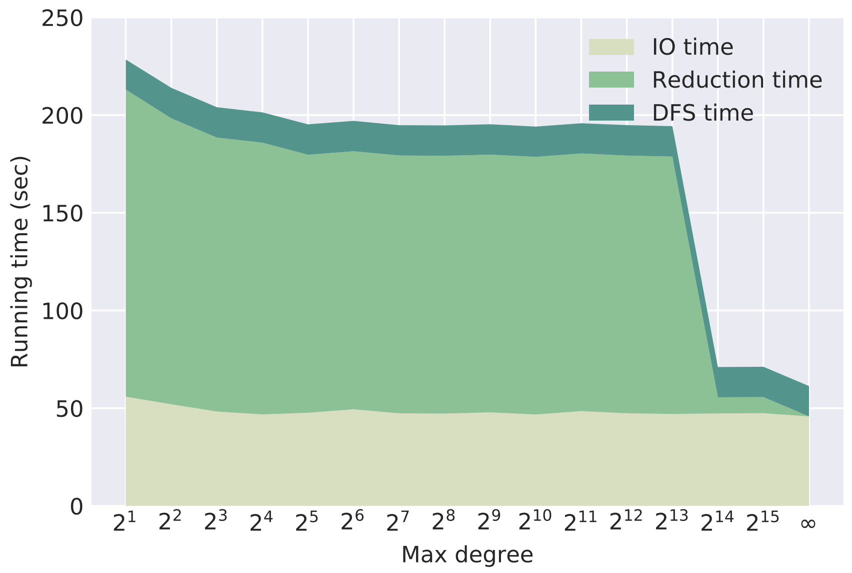

Figure 3.13: Running time for each of the graph optimizer's steps for different settings of maximum degree

seconds).

Because this step operates on the entire graph, and not on a single trial, it only needs to be run once. However, for the same reasons, it cannot be trivially parallelized.

3.4.4 Occurrence Processor Evaluation

Figure 3.14 shows the total running time taken by the occurrence processor to compute the YELT of a single trial. The running time is plotted against the average number of input occurrences each location generated for a single trial, and the maximum degree used during the optimization of the graph. Since individual properties are unlikely to make an insurance claim every year, the average number of occurrences generated per location in practice is typically no greater than 0.2. There were approximately 70 million individual locations in our test portfolio, so 0.2 occurrences per location resulted in 14 million input occurrences.

Running time scaled linearly with the average number of input occurrences per location. As with the graph optimizer, loading the graph into memory had a fixed cost of approximately 25 seconds, varying slightly depending on the number of vertices added during degree reduction. In a multi-trial, multi-threaded environment, the cost of loading the graph is amortized across all trials processed by an individual compute node.

Due to the introduction of several million merge vertices, occurrence processing on a

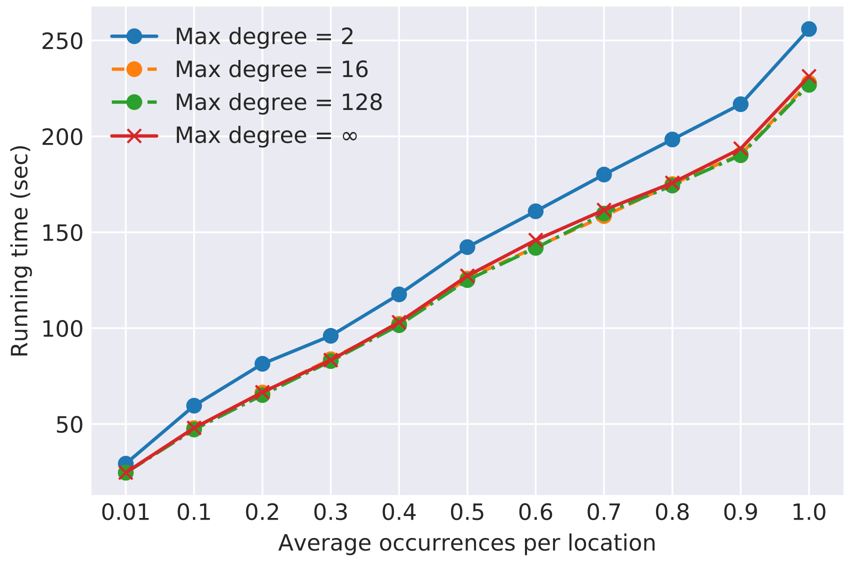

Figure 3.14: Running time for the occurrence processor across different sizes of input occurrences and graphs of varying degree

graph with its maximum degree reduced to 2 incurred a significant performance cost. For a maximum degree of 16 or 128, occurrence processing was slightly faster than on the unreduced graph. This was due to two reasons: First, the number of vertices added for $d = 16$ and $d = 128$ was negligible in comparison to the size of the original graph, so little overhead was added. Second, the original graph had a small number of mergers with high in-degree, while the degree-reduced graph did not. The merging algorithm used in our implementation was not cache-efficient for high-degree mergers, so the degree-reduced graphs achieved better cache locality and thus a slightly better running time.

The unreduced graph had a cutwidth of approximately 10,000 while the degree-reduced graph with $d = 16$ had a cutwidth of 310 and the degree-reduced graph with $d = 128$ had a cutwidth of 820. The amount of memory saved by degree reduction depends on the characteristics of the input occurrences. As mentioned in Section 3.3.4, if none of the occurrences merge, degree reduction provides no benefit to memory usage. In our experiment, occurrences merged so that no YELT contained more than 10,000 occurrences. For the degree-reduced graph with $d = 16$, 15 megabytes were used in total to store intermediate YELTs during occurrence processing of a single trial, the degree-reduced graph with $d = 128$ used 31 megabytes, and the unreduced graph used 112 megabytes. Thus, while degree reduction can result in significant memory savings during occurrence processing (approximately 85% less memory usage for intermediate YELTs in this case), the initial topological ordering found by the graph optimizer was still good enough to compute trials entirely in-memory, even without degree reduction. This can be significant in scenarios where few occurrences combine when merging YELTs.

3.4.5 Evaluation as a Distributed System

In order to evaluate the feasibility of performing, in a reasonable amount of time, a full-scale location-level portfolio analysis consisting of 10,000 trials, we provisioned 40 c5.18xlarge compute nodes from Amazon EC2 [9] to serve as our occurrence processors and submitted a 10,000 trial job using our graph reduced to a maximum degree of 16. We used the fairly large c5.18xlarge nodes for their high network bandwidth and because the high vCPU count (72) allows us to reduce the number of times the graph has to be loaded into memory. We used an average of 14 million input occurrences per trial, an aggressively high estimate of what we would expect from a typical location-level analysis job. Each compute node was issued 250 trials to compute, which was further broken up into 3 to 4 trials for each of the 72 vCPUs available on an c5.18xlarge instance.

We used Amazon's Simple Storage System (S3) for our distributed storage system, as it scales well, is highly available and has high throughput for Amazon compute resources located in the same availability zone. We used Amazon's Simple Queue Service (SQS) for the occurrence processor and graph optimizer queues.

A dedicated I/O thread was run on each compute node to download the graph and input YELTs from S3 in the background while already loaded trials were being processed by occurrence processor threads. The input YELTs totalled 4TB in size. Since each trial requires the graph to be loaded before processing, the occurrence processor threads were able to start processing trials only after the graph and the first batch of trials was loaded by the I/O thread. Once the graph and the input occurrences for the initial batch of trials were loaded, the I/O cost of loading the remaining input occurrence data was hidden, as the I/O thread was able to retrieve input faster than the occurrence processor threads could process them.

Starting from a newly provisioned cluster of occurrence processors with no input data preloaded onto it, the system was able to compute the output portfolio YELTs for 10,000 trials in approximately 17 minutes.

3.4.6 Comparison Against a Commercial System

The substantial licensing fees of commercial risk analytics systems make it infeasible to compare our system against a wide range of them. Due to a working relationship with one of the major vendors,[2] we were given access to a server running their platform.

The vendor's analytics suite offers two separate programs: an insurance client for modeling primary insurance structures and a reinsurance client for modeling reinsurance structures. Using these programs to model a reinsurer's portfolio at location-level requires using the insurance client to model the primary insurance contracts in the reinsurer's portfolio, manually exporting the resulting loss distributions to the reinsurance client, and running the reinsurance client to apply the portfolio's reinsurance treaties to the loss distributions generated by the insurance client.

We used this process to perform a location-level analysis on a real primary insurer's data set of 500,000 locations, representing hurricane risk exposures in a US state. Each location was covered by one primary insurance contract. As the reinsurance structure, we created a simple synthetic contract. A full reinsurance portfolio includes locations from many other states and countries. Thus, this data set represents only a small slice ($\leq 1\%$) of the amount of work required for a typical location-level analysis.

We evaluated the vendor's analytics suite on the vendor's hardware, a virtualized Windows Server 2016 machine running on a Xeon Gold 6154 processor with 16 virtualized cores and 64GB of memory, and another Windows Server machine running Microsoft SQL Server 2017 with 2 virtualized cores and 16GB of memory. With this configuration, it took the vendor's platform approximately 38 minutes to compute the portfolio's losses.

We ran the same experiment on our platform using comparable compute resources: one m5.4xlarge EC2 instance with 16 cores and 64GB of memory. We could not run on the vendor's hardware because the implementation of our platform is Linux-based. With this configuration, our platform took 35 seconds to perform the same analysis (plus an additional 11 seconds to topologically sort the graph and reduce its maximum degree). Due to nuances in the interpretation of some financial contracts, both systems generated different loss distributions in some instances. However, with detailed knowledge of the vendor's interpretation of such contracts, our system is capable of generating matching output.

In addition to being significantly faster, our system is also significantly more flexible. The

[2]Again, confidentiality agreements prevent us from disclosing the name of the vendor.

vendor's system allows only one primary insurance contract per location. The contract itself only supports the three most common terms. The reinsurance client allows users to create portfolios containing multiple contracts of different types, but they are difficult to combine to model arbitrary dependencies between contracts. The system uses a referencing system to direct output from one reinsurance contract to another but only some contracts can be referenced by others and keeping track of the overall structure becomes difficult as more references are added.

On the small data set in this comparison, our system was over 50 times faster than the vendor's system. Therefore, while our system can perform a full-scale location-level analysis in around 30 minutes, we expect the vendor's system to take more than a day. This has a significant impact on the feasibility of location-level analyses in the reinsurance industry. Moreover, we expect that the vendor's system's use of a single SQL server for processing and retrieving data introduces a significant bottleneck that severely hampers its scalability to the size of a full-scale location-level portfolio.

3.5 Conclusion

In this chapter, we presented a scalable, available, consistent and highly performant big data decision support system for processing complex reinsurance portfolios at location-level resolution. Our system is capable of processing several terabytes of data through very large graphs of hundreds of millions of vertices to compute a probability distribution of a reinsurance portfolio's risk. This distribution can be used to help inform reinsurers of the overall risk profile of their portfolio, and which reinsurance contracts they should underwrite.

By employing a flexible graph representation, our system can model arbitrary dependencies between reinsurance contracts. In contrast, many commercial systems on the market impose significant restrictions on the type of portfolio structures they can model. In spite of this greater flexibility, our system is over 50 times faster than at least one commercial system by a major vendor we were able to use for comparison. Moreover, it is unclear whether current commercial systems can scale to the size of a full-size location-level portfolio, an input our system can process in 17 minutes using a scalable cloud-based architecture.

Our system achieves excellent scalability by distributing each analysis over a range of trials. Because reinsurance analyses typically contain 10,000 trials, all of which can be computed independently, our system can, in theory, scale up to 10,000 concurrent threads. Our system is made highly available by making each occurrence processor stateless, and by storing all data necessary to complete any analysis on a fault-tolerant distributed storage system. In the event an occurrence processor fails to process a trial range, the failed task will be pushed back onto the occurrence processor queue after a specified timeout duration. Following this, a healthy occurrence processor will take the rescheduled job from the queue, download the required data from distributed storage, and begin processing the trial range. Data consistency in our system is entirely handled by the distributed storage system. Our implementation uses Amazon S3, which guarantees that the first version of each object uploaded is always consistent. By processing each trial entirely in memory, we achieve excellent single-threaded performance. This, combined with the high scalability of our system, makes our system highly performant.

We note that, while our system is significantly faster than commercial systems on the market today, the running times on the system described in this chapter are still too slow to support interactive workflows. In order to support interactive workflows, we expand our system to cache and reuse intermediate results in the next chapter.

Chapter 4

Efficient Caching for Location-level Analytics

In the previous chapter we introduced a graph-based computational model for reinsurance analytics. In this model, a reinsurance portfolio composed of many interdependent contracts is represented as a graph of transformations, where edges describe the flow of loss data, and vertices describe contract transformations on loss data. This system is highly available, consistent and scalable, but is not yet fast enough to support interactive workflows. In this chapter, we introduce a method of caching a subset of the intermediate results generated during a location-level analysis to support fast computation of incremental analyses, and therefore interactive workflows.

An incremental analysis is an analysis run after completion of an initial analysis and after a series of incremental updates to the graph. Incremental analyses happen when a reinsurance analyst makes small (relative to the hundred-million-vertex scale of the graph) changes to their portfolio graph and recomputes the portfolio's losses to compare how the changes have affected the portfolio's overall loss distribution. Changes are usually localized within a small portion of the graph (e.g., the vertices that comprise an individual treaty or contract). This is because primary insurers insure many properties within a single region, and thus reinsurance treaties typically cover regional clusters of locations. Fast incremental analyses are extremely important to reinsurers, as it allows them to see the impact the terms of a treaty have on their portfolio as they negotiate these terms with a broker over the phone.

We achieve fast incremental analyses by caching (i.e., writing to distributed storage) the loss distributions associated with a carefully chosen subset of edges in the portfolio graph. This allows us to avoid reanalyzing large parts of the portfolio that are unaffected by the changes made elsewhere in the portfolio. Because caching significantly reduces subsequent analysis running times, the time taken to load the portfolio graph becomes a major bottleneck. To address this, and to further improve performance, we split the portfolio graph into multiple "shards" that can be loaded individually. By only loading the shards of the graph needed to process an incremental analysis, we can significantly reduce I/O time.

For 10,000 trials, edge outputs are large enough that caching the results of a large number of edges would use a prohibitive amount of disk space. Thus, the main challenge is to select a small number of edges that ensure that caching their YELTs dramatically reduces the size of the portfolio to be reevaluated after a typical incremental update, while still keeping cache storage requirements low.

Our experiments show that by caching a small number of cache edges allows us to compute the portfolio's loss distribution up to 90 times faster than without caching. This means that an incremental analysis of a location-level reinsurance portfolio can be performed in approximately 10 seconds, allowing reinsurers to see the impact updates to their portfolio have on their loss distribution in real-time as they negotiate contractual terms with brokers over the phone.

The remainder of this chapter is organized as follows: Section 4.1 covers related work on caching intermediate outputs in directed graph processing systems. Section 4.2 describes our caching strategy. Section 4.3 introduces a new graph data structure to support partial loading of the portfolio graph, depending on which cache edges are available. Section 4.4 describes how to support graph updates while maintaining the existing cache edges and data structures. Section 4.5 discusses experimental results that demonstrate the performance of our approach, with a series of comparisons across a variety of use cases. Section 4.6 offers closing comments.

4.1 Related Work

For related work on reinsurance analytics and graph processing, see Sections 2.2 and 3.1. This section covers work related specifically to the problem of caching within graph processing systems.

The Spark data flow and graph processing framework has built-in caching of its intermediate results (called Resilient Distributed Data (RDD) objects), which simply places all RDDs into a Least Recently Used (LRU) cache [99]. Under the LRU system in Spark, each RDD is added to the cache when it is produced. If inserting a new RDD in the cache causes the cache to exceed its configured memory limit, the least recently used RDD is evicted before the new RDD is added. The intention is to avoid computing the same RDD twice within the same job. For example, if an RDD is needed twice at different times in the same Spark job, keeping the RDD cached would prevent that RDD from being recomputed when it is needed the second time later in the job.

LRU caches are conceptually simple, easy to implement, require little configuration and perform well for a wide variety of applications. They are therefore used in a wide variety of systems outside of Spark [71]. For our application, our low-cutwidth vertex ordering described in Section 3.3.3 means that no YELT will need to be recomputed a second time within the same job, which makes LRU caching ineffective. More generally, Spark uses its cache to solve a similar but different problem than ours. Spark's goal is to avoid recomputing intermediate results during a *single* graph evaluation, while our goal is to avoid recomputing intermediate results across *multiple* portfolio analyses, in response to portfolio updates. Thus, our platform would instead benefit from a cache that stores intermediate results between analyses to improve the performance of incremental analyses. A LRU cache for this purpose would perform poorly. At the end of a job, the LRU cache would contain only the outputs from the last vertices in the vertex ordering. These vertices are close to the sink, and their cached edge outputs are therefore the least likely to be valid after a change to the graph is made (as an update to the graph would require the recomputation of all YELTs "downstream" from the update.

Multiple works introduce different variations on the LRU cache within Spark [32, 94, 97, 98]. The Least Reference Count (LRC) cache [98] attempts to reduce the number of RDD recomputations over the default LRU cache implementation by evicting RDDs with the fewest "reference counts" instead of the RDDs with the oldest access time, where reference count is the number of unfinished tasks that require the cached RDD as input. The Least Effective Reference Count (LERC) cache [97] augments the LRC cache by preferring to evict RDDs to be used by succeeding tasks that do not already have all their input RDDs in cache. The Least Cost Strategy (LCS) [32] cache evicts RDDs that take the least time to recover. Another work [94] uses adaptive optimization to determine which RDDs should be cached during processing.

Since these works are focused on caching RDDs within a single analysis for the sake of improving the running time of that single analysis, we expect that applying these caching mechanisms to improve performance of incremental analyses on a large graph would yield results similar to using the LRU cache.

Much of the research on caching in dataflow and graph processing systems is focused on utilizing limited-memory CPU caches to minimize the running time of individual analyses or jobs [46, 47, 52, 53, 64, 68, 79, 101]. This is done by finding vertex orderings that optimizes for locality of reference during graph processing. A vertex ordering optimized for locality of reference can have lower running times compared to other orderings, as they are more likely to have more CPU cache hits, and thus reduce the number of expensive read and write operations to and from main memory. Like the LRU cache, we stress that these works are focused on solving an entirely different problem than ours; they describe how graph processing systems can be optimized to use CPU caches to minimize the impact of memory latency. These works do not cache data in between analyses, nor do they aim to reduce the overall computational effort for subsequent evaluation of modified graphs.

4.2 Caching for Location-Level Reinsurance Analyses

Computing the loss distribution of the portfolio in a location-level graph is time-consuming. By writing the YELTs associated with a subset of edges produced during a from-scratch portfolio analysis to distributed storage, we can significantly reduce the number of YELTs that have to be computed in an incremental analysis by loading the stored YELTs from storage instead of processing the associated vertices and their predecessors. We refer to this process of storing intermediate YELTs to reduce the computational effort of subsequent or incremental analyses as *caching*. Given a graph $G = (V, E)$ and the set of edges with cached YELTs, E_c (henceforth called *cache edges*), an incremental analysis needs to reevaluate only the vertices that can reach the sink of G in $G' = (V, E - E_c)$, provided none of the cache edges' YELTs are affected by the portfolio update that triggered the incremental analysis. The contribution of the vertices that do not have a path to the sink in G' to the portfolio's output is already accounted for in the YELTs associated with the cache edges that separate them from the sink.

This is illustrated in Figure 4.1. Without cache edges, the entire set of vertices must be evaluated, that is, every vertex must transform the YELTs generated by its in-neighbours into an YELT passed to its out-neighbours. When the 4 cached edges are included, the set of vertices that cannot reach the sink in $G' = (V, E - E_c)$ (colored gray) do not have to be evaluated to compute the output of the sink.

Since the YELT generated by any vertex in the portfolio graph depends on the YELTs associated with edges that can reach this vertex, we consider any cache edge reachable from a vertex or edge in G involved in a portfolio update to be *invalidated* by this update—its YELT needs to be recomputed. We call such an edge a *downstream edge* of the update. An incremental analysis needs to recompute the YELTs of all edges that can reach the sink via a path consisting entirely of uncached edges and invalidated cache edges.

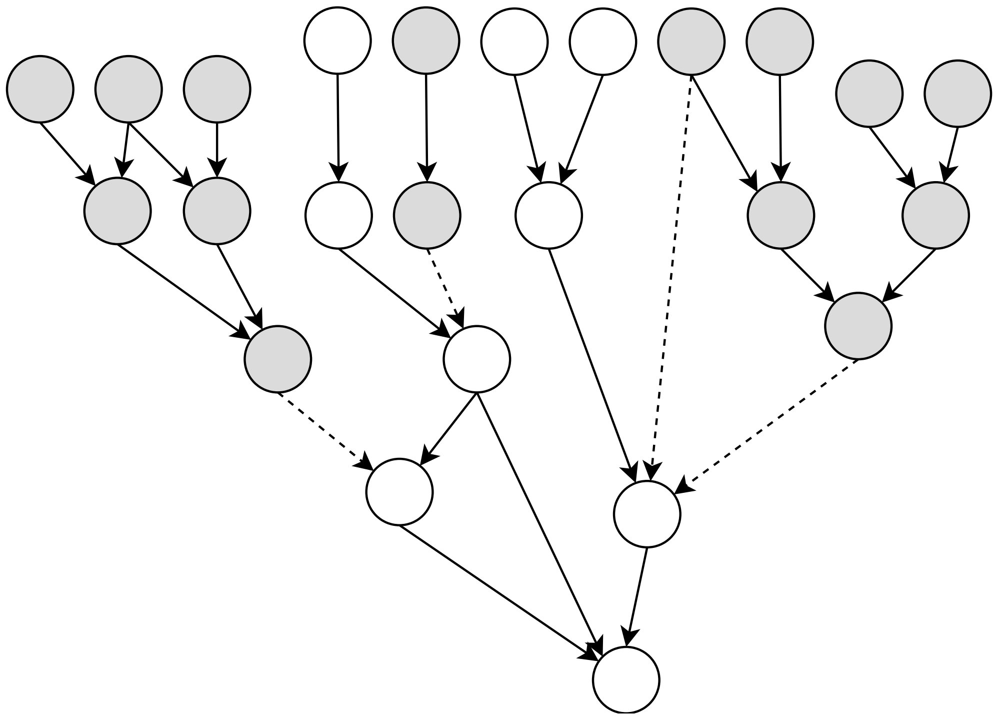

Figure 4.1: A small directed graph, with cache edges shown as dotted lines. The gray vertices do not need to be reevaluated to compute the output of the sink.

4.2.1 Selecting Cache Edges

Selecting which edges to cache has a significant impact on the memory footprint and efficiency of the cache. In the location-level reinsurance analytics use case, graphs are very large and can contain hundreds of millions of edges. The size of each YELT depends on the number of occurrences in the YELT, which itself is dependent on how often occurrences merge during processing of merge vertices. Let us assume a very low estimate of an average of 10 occurrences (160 bytes) per YELT per trial. Since a 10,000 trial simulation is typical for reinsurance analytics, each cache edge requires 1,600 kilobytes. Even with such a small YELT memory footprint, caching even 1% of the edges in a 300 million edge portfolio graph requires 4.8 terabytes of memory.

Conversely, if too few edges are cached, a single user update to the portfolio graph that invalidates one or two cache edges can result in several million vertices being processed in an incremental analysis. Thus, the choice of edges to cache needs to strike a balance between storage cost and the running time of incremental analyses.

To keep storage costs low, we would like to see significant speedup while using less than a terabyte of storage for the edge cache of a single graph. As a result, we need a method of intelligently selecting a configurable number of cache edges from graphs on the scale of hundreds of millions of edges, so that in the event of several incremental updates to the graph, the number of vertices that must be processed to obtain a result for an incremental analysis is small enough such that the incremental analysis can be computed significantly faster than the initial analysis.

Location-level portfolio graphs are generally trees of small 2-edge-connected components (representing primary insurance contracts) and with only very few larger 2-edge-connected components (representing the complex network of reinsurance treaties). Each edge between any two 2-edge-connected components is a bridge; that is, an edge whose removal partitions the graph into two disconnected subgraphs. This implies that a significant fraction of edges in a location-level portfolio graph are bridges.

Our method to select cache edges starts with the bridges of the portfolio graph as the candidate set of potential cache edges. Bridges are promising candidate cache edges because by caching a single such edge, the cost of an incremental analysis is reduced by the number of all "upstream" edges of the chosen edge (that is, all edges reachable from the edge after reversing the directions of the graph) unless these upstream edges are invalidated by the portfolio update that triggered the incremental analysis.

Many of these bridges are close to the sources of the portfolio graph (i.e., locations) and thus separate only a small number of upstream vertices from the rest of the graph. Bridges near the sink of the graph are more likely to partition large numbers of upstream vertices from the sink. However, a cache edge that splits off too many upstream vertices is fragile in the sense that it has a high probability of being invalidated by an update. To balance the promised savings from caching edges (due to many upstream vertices) versus the probability of invalidating many of these edges with an update (also due to many upstream vertices), we select the cache edges so that their removal partitions the graph into subgraphs of roughly equal size. We call these subgraphs *shards* from now on.

Given a target number of n shards, we compute the set of cache edges as follows. We compute the 2-edge-connected components of G (using a depth-first traversal [84]) and replace each component with a single vertex of weight equal to the size of the component. The result is a weighted tree T whose edges are exactly the bridges of G. Figure 4.3 shows the tree T obtained from the graph in Figure 4.2. Let W be the total weight of T, that is, the

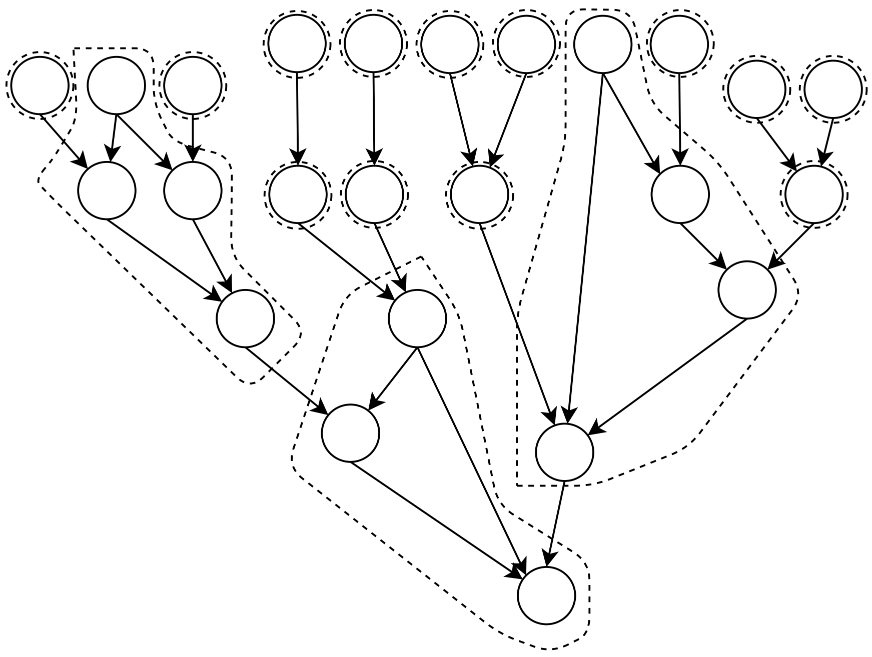

Figure 4.2: The example graph from Figure 4.1, grouped into its 2-edge-connected components by dotted lines. The edges that cross between two components are the bridges.

number of vertices in G, and let $w = W/n$ be the desired shard size. We select cache edges using a postorder traversal of T. Whenever we encounter an edge whose descendant nodes in T have weight at least w, we select the edge as a cache edge and remove all its descendant nodes from T. The removed nodes no longer contribute to the total weight of descendant nodes of any edge considered subsequently. If every node in T has weight at most w and degree at most d, then this approach is guaranteed to produce a set of between n/d and n cache edges that partition G into shards of size between w and dw. For the tree in Figure 4.3, the partition obtained using the edges chosen with parameter $w = 5$ are shown using dotted lines. The corresponding shards are shown in Figure 4.4.

4.3 Efficient Subgraph Loading

While caching reduces the number of YELTs that need to be computed in an incremental analysis, loading the portfolio graph and its input YELTs remains costly. Recall that in Section 3.4, it took approximately 25 seconds to load a large location-level graph into memory. Assuming caching reduces the cost of computing the YELTs in an incremental analysis to a few seconds, loading the portfolio graph thus becomes the bottleneck. We address this by

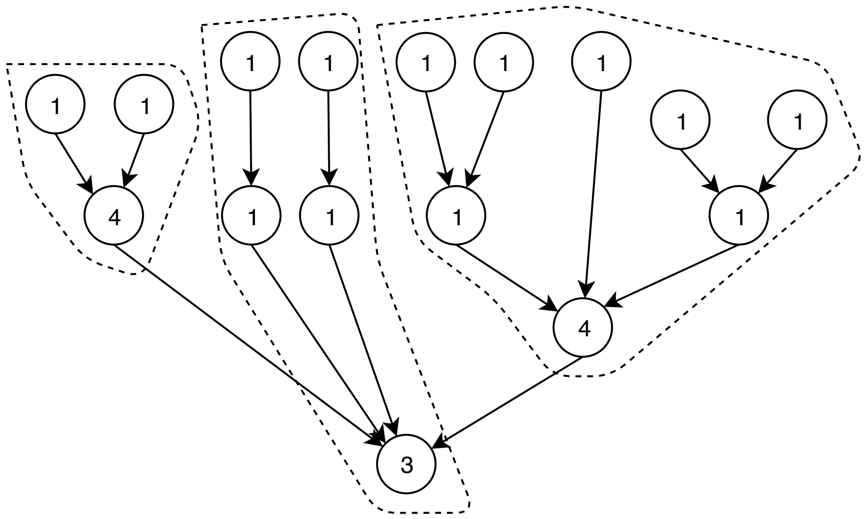

Figure 4.3: The tree T obtained by contracting the 2-edge-connected components of the graph in Figure 4.2. Vertex labels represent vertex weights, that is, the sizes of the 2-edge-connected components represented by the vertices. The dotted lines represent the partitioning of T into subtrees with $w = 5$.

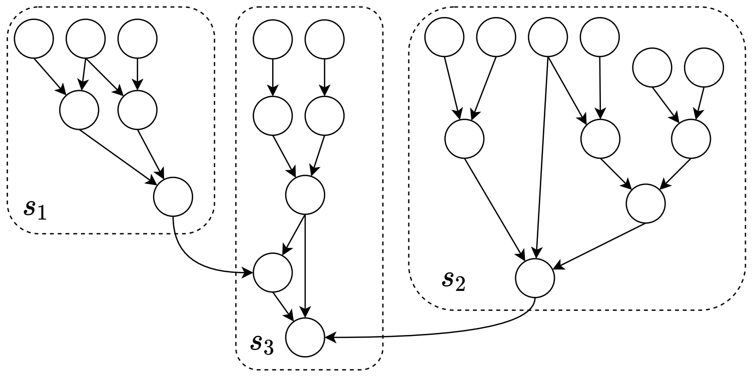

Figure 4.4: The graph from Figure 4.2 broken into 3 shards, s_1 through s_3 corresponding to the subtrees in Figure 4.3. The edges between shards are cache edges.

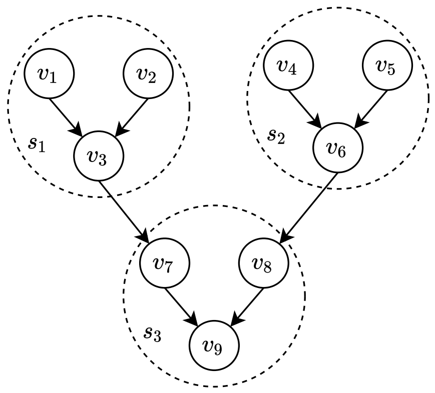

Figure 4.5: A graph with cache edges pointed towards different vertices within the same shard

loading only the shards of the graph that are needed for the incremental analysis. To support this, we modify the in-memory graph representation described in Section 3.3.5 into a *sharded graph representation*, which consists of three parts.

First, we store each shard of G as a separate file in a distributed storage system. Each shard is simply stored as a graph file in the same format described in Section 3.3.5.

Second, we store a file containing the *shard graph*. This graph has one vertex per shard. Its edge set is the set of cache edges connecting these shards. Each vertex in the shard graph stores a reference to the shard it represents, that is, a key identifying the shard's graph file in distributed storage.

To keep track of invalidated cache entries, we store a "version ID" value for each shard in the shard graph. Whenever one of the shards is modified, its corresponding file gets assigned a new version ID. During analysis, whenever the edge output between two shards is written to cache, we also store the version ID of the shard that generated the output. For a given cache edge, we determine if its cache entry has been invalidated by comparing the version ID of the edge's cache data against the version ID of the graph file belonging to the edge's head vertex. If the version IDs do not match, then the cache entry is invalid, and the head vertex must be processed.

Third, since the shard graph only describes which shards are connected, and does not specify which vertices within the shards act as output or input to or from another shard, we store a small *vertex mapping file* to describe, for each cache edge, the vertex in the tail shard that the output from the head shard should be forwarded to.

For example, consider Figure 4.5. Shards s_1 and s_2 both output to s_3, but to different vertices within s_3. Results stored on the cache edge from s_1 to s_3 must be passed to vertex v_7 in s_3. Results on the cache edge from s_2 to s_3 must be passed to vertex v_8 in s_3. These shard-to-shard data flows are not represented in the shard graph at the individual vertex level, and thus the mapping file is used to determine which vertex within s_3 the output from s_1 goes to, and which vertex within s_3 the output from s_2 goes to.

The result is $|E_c|+1$ partitions of the original graph, stored in separate files, and a small shard graph of $|E_c|$ edges and $|E_c|+1$ vertices describing the data dependencies between the shards.

During occurrence processing, a traversal over the shard graph starting from the sink determines whether or not individual shards should be loaded and processed, where only edges with invalidated cache entries are traversed. The vertices in the shard graph that are traversed are therefore the shards that must be loaded and processed; the shards that are not traversed do not need to be loaded as their output can be loaded from cache.

In the context of the platform described in Chapter 3, finding an appropriate set of cache edges, and writing the shards, shard graph and vertex mapping file to storage can all be done as an extension of the graph optimization step.

4.4 Updates

Underwriters may decide to add or remove treaties to or from their portfolio or agree to change the terms of a treaty, after negotiating treaty terms with a broker. Thus, after running incremental analyses to evaluate the impact of the proposed changes on the risk exposure of the portfolio, they may decide to accept the proposed changes, and the portfolio representation needs to be updated accordingly. It is easy to take these changes into account during the next nightly from-scratch portfolio analysis. To allow additional incremental analyses before the next from-scratch analysis, however, we also need to support updates to the graph representation. These updates may not result in the optimal set of cache edges for the updated portfolio, but the updates need to be fast and need to ensure that the updated portfolio representation continues to support fast incremental analyses.

Any portfolio update can be expressed as a sequence of elementary updates of the following types:

- Modification of a vertex's terms (e.g, adjusting the deductible of a vertex)

- Addition of a disconnected vertex
- Deletion of a disconnected vertex
- Edge addition (must keep the graph acyclic)
- Edge deletion

These operations are complete in the sense that any portfolio graph can be constructed and broken down using only these operations.

Thus, we only need to discuss how to support these elementary updates. Any addition, deletion or modification of an edge or vertex in the graph invalidates all cached edges downstream of the change.

Vertex additions and deletions do not invalidate existing cache edges, as a disconnected vertex has no output. Since new vertices are always disconnected, they are not assigned to a shard until they are connected to the graph with an edge addition. When a disconnected vertex is connected to the graph with an edge addition, it joins the shard belonging to its new neighbor. Unlike vertex additions and deletions, vertex modifications operate on connected vertices, and can invalidate cache edges. In the event of a vertex modification, all cache edges accessible by following directed paths from the modified vertex must be invalidated. Similarly, for an added or deleted edge, all cache edges accessible by following directed paths from the head of the added or deleted edge are invalidated. This is true no matter whether the endpoints of the edge are in the same or different shards.

When an edge is added that connects vertices in different shards, the added edge must also be added to the set of cache edges E_c and the shard graph's vertex mapping. This is because the computation of the shard containing the edge's tail vertex cannot be skipped unless a YELT is present in the cache for both the existing cache edge and the newly added edge. If an edge is added between two vertices in shards that do not have an edge directly linking them together in the shard graph, the shard graph must be updated to include an edge between the affected shards.

If the added cache edge creates no cycles in the shard graph, nothing else needs to be done. However, even an edge addition that keeps G acyclic may introduce cycles in the shard graph (see Figure 4.6). In this case, the shard graph must be restructured to eliminate the cycle. This can be done by moving all vertices in the shard containing the tail of the new edge that can reach this tail to a new shard. An example is given in Figure 4.6. An added edge from

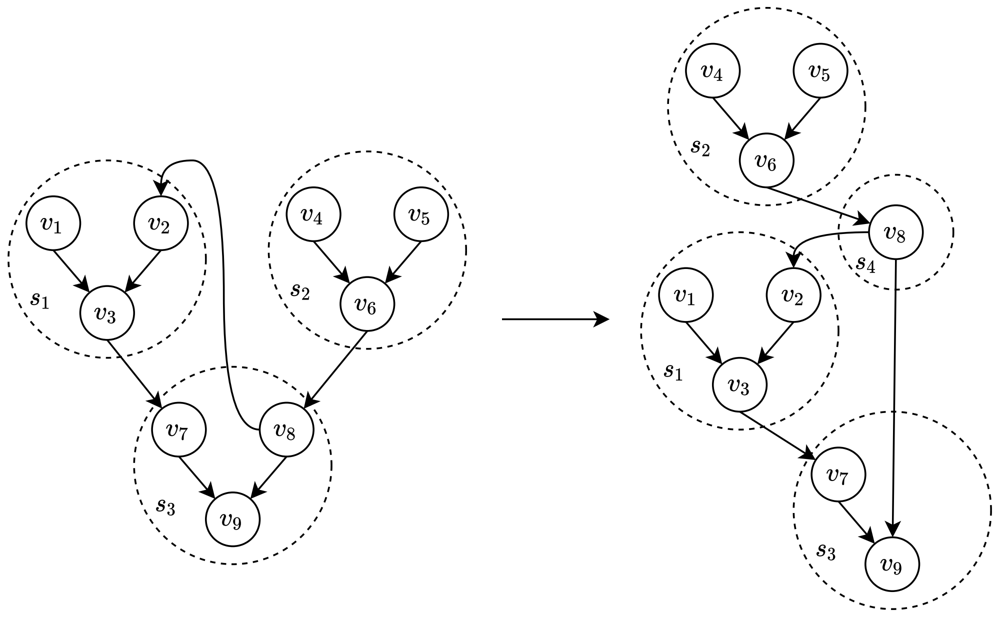

Figure 4.6: The graph from Figure 4.5 with an added edge between the vertices v_2 and v_8 in shards s_3 and s_1. The graph on the right shows the new partitioning of the same graph that eliminates the cycle in the shard graph.

a vertex in s_3 to a vertex in s_1 results in a cycle between these two shards in the shard graph. In the graph on the right, s_3's v_8 vertex is moved into its own shard to remove the cycle from the shard graph.

In addition to the elementary update operations listed above, we also support one non-elementary update operation directly, because simulating it using elementary updates would have the potential to introduce significant imbalances to the shard graph. Our *bulk insert* operation allows the user to add a connected subgraph to the portfolio graph with a single update. All vertices belonging to the bulk-inserted subgraph are placed in the same, newly constructed shard. In doing this, we can avoid several individual edge and vertex addition operations, which would be likely to make one shard significantly larger than others, or, in some cases, introduce a significant number of new shards and cache edges. This is particularly useful when adding a new primary insurer to the portfolio, as this requires the reinsurer to add the primary insurer's entire subgraph to the reinsurer's portfolio graph.

4.4.1 Periodic Repartitioning into Shards

When an edge is added between shards, the number of cache edges increases by 1. Ideally, these kinds of edge additions would trigger a recomputation of cache edges and shards, but for large graphs, this is an expensive operation. Additionally, a complete recomputation would likely use an entirely different set of edges for the cache and would require a full uncached analysis to regenerate. A repartitioning of the graph is therefore best initiated during a period of low user activity, perhaps nightly, and should be followed by a new analysis to populate the cache. As long as edge additions between shards are rare, and repartitioning tasks run regularly, incremental analyses should remain fast, while keeping cache storage costs reasonable.

4.4.2 Realistic Use Cases

Here, we describe a set of common reinsurance use cases for incremental analyses. For each use case, we describe the application, how it modifies the portfolio graph, and the set of update operations it requires.

New Business. In a "new business" incremental analysis, the reinsurer adds a new primary insurer to the reinsurance portfolio, and attaches it to a reinsurance treaty. This is done whenever a reinsurer writes a new business into their portfolio. In terms of the portfolio graph, this use case adds a potentially large (depending on the size of the primary insurer) primary insurance structure to the graph, and attaches it to a vertex representing the beginning of a reinsurance contract.

This use case is performed with our non-elementary *bulk update* operation. This ensures that the primary insurance structure is added to a new shard to facilitate efficient caching for later analyses. Once the new primary insurance subgraph has been added, it is connected to the rest of the portfolio graph with one or more edge addition operations. Each edge added between the new shard and the other shards must be cached. If the primary insurance structure added is much larger or smaller than other shards in the portfolio graph, or if the addition of new edges significantly increases the number of cache edges, the portfolio graph can later be repartitioned offline to find a more balanced set of cache edges.

Pricing. The "pricing" use case occurs when a reinsurance company negotiates the terms of renewal or new business with a primary insurer, typically over the phone. In this use case, the reinsurer updates terms of the reinsurance treaty attached to the primary insurer to see the impact the changes have on the portfolio's overall loss distribution, relative to the premium gained by the renewal or new business. This results in one or more vertex modification operations, which do not affect the structure of the shard graph, but result in all cache edges downstream of the affected vertices to be invalidated.

Timely responses are extremely important for the “pricing” use case. Since multiple reinsurers are competing for the same primary insurance contracts, being early to make a fair offer significantly increases the reinsurer’s chances to get a share of the contract.

Renewal planning. Reinsurers have renewal seasons wherein they price and sign contract renewals with primary insurers. In the months leading up to a renewal season, reinsurers go through a “renewal planning” process, where they adjust their shares in existing contracts and compare the portfolio’s new loss distribution. By doing this, the reinsurer can begin to plan what primary insurer contracts they want to renew and what contracts they want to increase or reduce their share in. Like a “pricing” analysis, a “renewal planning” analysis only modifies the terms of reinsurance structures that control the shares or participation rates of a reinsurance contract. Thus, they result in one more more vertex modifications, which do not alter the structure of the shard graph, but result in cache invalidations downstream.

Timely responses to “renewal planning” incremental analyses are beneficial as they allow the reinsurer to tweak their contract’s shares and get quick feedback on the impact the changes have on the portfolio’s loss distribution. This allows the reinsurer to make multiple adjustments to further fine-tune the shares on each of their contracts.

Renewal. When a primary insurer renews a contract with a reinsurer, the primary insurer typically sends new exposure data containing the up-to-date location loss distributions (YELTs) to the reinsurer. Since reinsurance contracts renew all throughout the year, reinsurers process renewals on a daily basis. In a “renewal” incremental analysis, the reinsurer replaces the old location loss distributions with the newer distributions and recomputes their portfolio loss distribution. In order to change the input location loss distributions, the terms of the vertices that load losses for each affected location must be updated to point to the location of the new files in distributed storage, resulting in several vertex modification operations.

Binder book update. Reinsurers sometimes participate in “binder” contracts. Unlike typical contracts, where loss distributions are only updated upon renewal of the contract, loss distributions in a binder contract update on a regular basis (e.g. monthly). Thus, reinsurers that participate in binder contracts must periodically run “binder book updates”, wherein the loss distributions for a contract are updated. Like the “renewal” use case, this results in a series of modifications to vertices that load input loss distributions from distributed storage.

Retrocession pricing. Retrocession treaties function like insurance for reinsurers. In the event of an extreme catastrophic event or loss to the reinsurer, a retrocession treaty may activate and pay out the reinsurer depending on the terms of the treaty. Reinsurers typically buy retrocession insurance to improve the solvency of their portfolio. Retrocession treaties are priced similarly to primary insurer contracts: the reinsurance and retrocession broker negotiate terms and premiums over the phone until an agreement is made. In the “retrocession pricing” use case, the reinsurer modifies the terms of an existing retrocession treaty, or adds a new retrocession treaty, and compares the impact the changes have on the portfolio’s loss distribution. If the reinsurer only wants to model the result of numerical adjustments to the treaty and not modify the structure, this only results in one or more vertex modifications. If the reinsurer is adding a new retrocession treaty, or is adding new terms to an existing treaty, then new vertices and edges must be added to the graph. If the reinsurer is modifying an existing treaty by removing terms, then the edge and vertex deletion operations are required.

4.5 Evaluation

In this section, we present experimental results that demonstrate the viability of our caching strategy to support incremental location-level portfolio analyses. We start with an evaluation of the number of cache edges obtained for different shard sizes. Intuitively, smaller shards mean faster incremental analyses, but as the number of cache edges increases, the storage cost and the cost of writing cached YELTs to distributed storage during a from-scratch analysis increase. Thus, it is worthwhile to investigate this trade-off. Our first set of running time experiments evaluates for three different shard sizes how the cost of an incremental analysis increases as the number of invalidated cache edges increases. We note that realistic updates of realistic reinsurance portfolios lead to only a small number of invalidated cache edges, so this evaluation of the analysis cost as a function of up to a few hundred invalidated cache edges serves as an extreme stress test of our approach. The second set of running time experiments evaluates the cost of incremental analyses for typical portfolio updates in a range of realistic use cases.

4.5.1 Test Platform and Portfolio

Our evaluation platform uses the system presented in Chapter 3 augmented with the caching strategy proposed in this Chapter. Experiments were run on two separate clusters of compute nodes running Amazon Linux 2, provisioned from Amazon EC2. The first cluster was composed of 350 c5.2xlarge instances (each with 16GB memory and 8 vCPUs), while the second was composed of 40 c5.18xlarge instances (each with 144GB memory and 72 vCPUs).

As explained in Chapter 3, since current commercial systems are unable to perform a full location-level portfolio analysis in a reasonable amount of time, there do not exist any real-world location-level portfolio data to date that could be used in experiments to evaluate our system. Therefore, we used the same synthetic portfolio described in Section 3.4.1, which simulates the structure we anticipate future location-level portfolios to have. The test portfolio had approximately 307M vertices and 377M edges.

4.5.2 Shard Size vs Number of Cache Edges

Figure 4.7 shows the number of cache edges $|E_c|$ as a function of the shard size parameter w. Unsurprisingly, the number of cache edges decreases with increasing shard size, but there are numerous plateaus in the curve, indicating that for our test portfolio, changing the value of w by small increments had little effect on $|E_c|$.

This is caused by vertices in T with many in-neighbors, all of large, but roughly equal subtree weight, say n. For any such vertex v, if $w >= n$, then all of v's in-edges will be selected as cache edges. If $w < n$, then none of v's in-edges will be selected. Thus, at $w = n$, there is a jump in the number of cache edges.

As a result, finding the value w that produces a particular number of cache edges is difficult. In a production system, the step that partitions T into subtrees may need to be run with different values of w, in order to find a partition that generates an appropriate number of cache edges.

4.5.3 Analysis Cost vs Invalidated Edges

To measure the impact of the number of invalidated cache edges on the cost of an incremental portfolio analysis, we ran multiple 10,000-trial incremental analyses on partitions of our portfolio graph into shards with parameter $w = 100{,}000$, $w = 250{,}000$, and $w = 500{,}000$. These

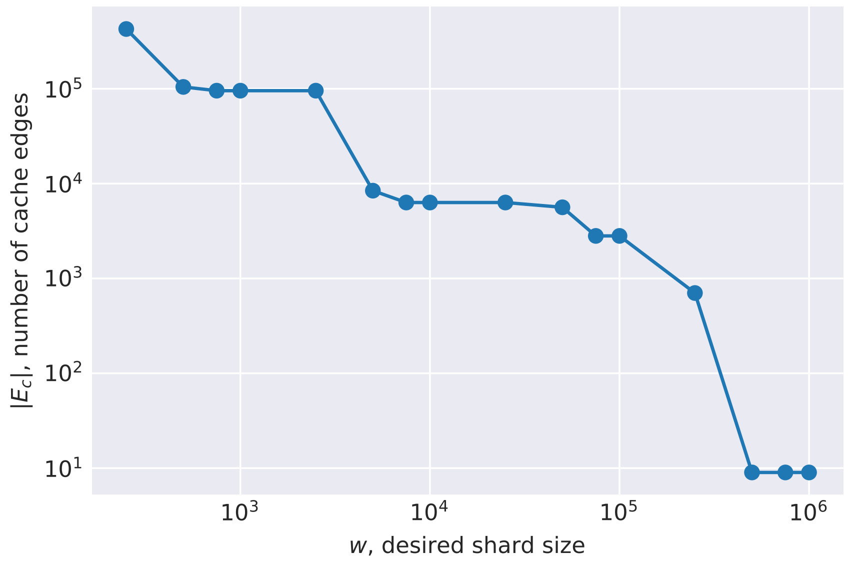

Figure 4.7: Number of cache edges generated for different values of the shard size parameter w.

shard sizes resulted in 2808, 702, and 9 cache edges, respectively. For each partition, we measured the cost of an incremental analysis as a function of the number of invalidated cache edges. For all three values of w, the shard graph had one shard containing the sink of the portfolio graph and all other shards had out-edges to this sink shard and no edges between each other. This is because, while the portfolio graph is extremely large, it has a relatively small depth or distance from any source to the sink and is generally tree-like, making it unlikely to find large enough components that do not contain a sink or source vertex during cache edge selection. As a result, a certain number k of invalidated cache edges translated directly into the evaluation of $k+1$ shards during an incremental analysis.

For $w = 100{,}000$ and $w = 250{,}000$, the sink shard had approximately 8,000 vertices and contained all reinsurance treaties and insurance contracts for high-value properties. The remaining shards were portions of the primary insurance business unit subgraphs covering around 25,000 and 100,000 locations, respectively.

For $w = 500{,}000$, the sink shard became extremely large, containing approximately 250 million vertices. This was the result of two factors: w was larger than the size of any individual primary insurer business unit subgraph, and the reinsurer's contractual terms subgraph was densely connected, containing almost no bridges. Thus, once the partitioning algorithm decided not to form a shard for any individual primary insurer business unit, many of these

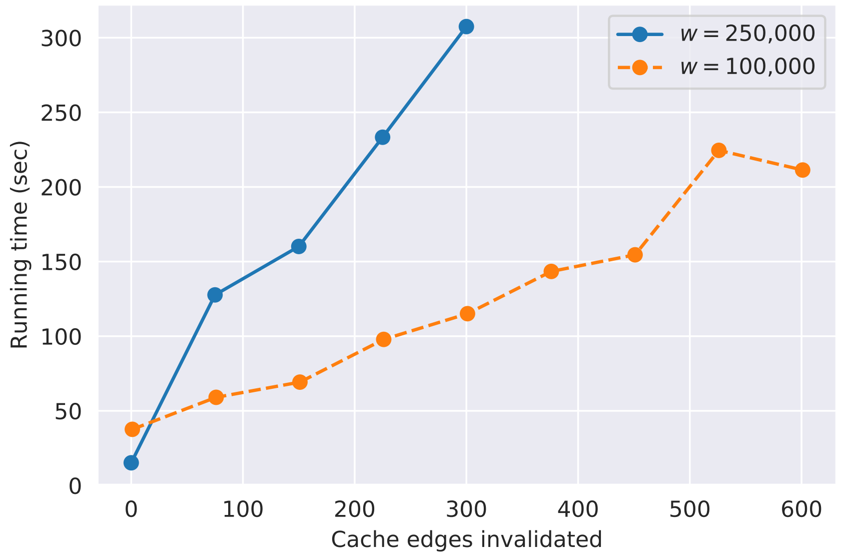

Figure 4.8: Running time of an incremental analysis as a function of the number of invalidated cache edges on a cluster of 350 c5.2xlarge instances. Running times are averaged across 5 runs.

subgraphs merged with the reinsurer's contractual terms subgraph to form a very large shard.

For this experiment, we provisioned 350 c5.2xlarge instances. Each such instance had 16GB of memory and 8 virtual cores, running on a 1st or 2nd-generation Intel Xeon Platinum 8000 series CPU. All input data and cache was stored entirely on Amazon S3. For reasonable shard sizes, these instances have sufficient memory to carry out an incremental analysis across a small number of trials. However, due to the extremely large sink shard for $w =$ 500,000, an incremental analysis with this shard size parameter has to evaluate around 80% of the entire portfolio graph, which would require more memory than which is available on c5.2xlarge instances. We therefore could not run any incremental analyses with the $w =$ 500,000 partition on our c5.2xlarge cluster. On clusters with more memory, the $w = 500{,}000$ partition is expected to perform extremely poorly, for similar reasons. Because the sink shard contains approximately 80% of the entire portfolio graph, each incremental analysis must process, at minimum, 80% of the entire portfolio graph.

Figure 4.8 shows the running time of an incremental analysis with $w = 100{,}000$ and $w =$ 250,000 for up to 600 invalidated cache edges. With a shard size of $w = 100{,}000$, even an incremental analysis with an unrealistically high number of 500 invalidated cache edges took less than 4 minutes. Due to the larger shard size, the running time of an analysis with shard size parameter $w = 250{,}000$ increases faster as the number of invalidated cache edges increases. For more than 300 invalidated cache edges, the larger shard size resulted in the analysis using more memory than available on c5.2xlarge instances. This memory limit was reached only after 600 invalidated cache edges for the smaller shard size achieved with parameter $w = 100{,}000$.

As we've seen in Section 4.4.2 (and explore in more detail in Section 4.5.5), each incremental analysis use case typically updates only one primary insurance, reinsurance or retrocession structure of the portfolio graph. This means that changes for each individual use case are highly localized within a small number of structures within the portfolio. Because of this, a series of changes that results in more than 10–20 invalidated cache edges represents significant changes to the graph, and is atypical for an incremental analysis.

Figure 4.9 shows the results for a smaller range of invalidated cache edges corresponding to typical portfolio updates. In this range, the larger shard size achieved with parameter $w = 250{,}000$ resulted in *faster* running times. Since all cache edges in the partition were in-edges of the sink shard for both $w = 100{,}000$ and $w = 250{,}000$, this means that all cached YELTs need to be read from Amazon S3 no matter how many cache edges are invalidated. The significantly higher number of cache edges for $w = 100{,}000$ led to a significantly higher I/O cost. Figure 4.9 demonstrates that with parameter $w = 250{,}000$, an incremental analysis after virtually any realistic portfolio update takes less than 35 seconds. Any portfolio update that affects only reinsurance treaties and none of the primary insurer contracts, a common use case as discussed in Section 4.5.5, does not invalidate any cache edges. Without invalidated cache edges, an incremental analysis took around 11 seconds in our experiments for $w = 250{,}000$.

For reference, the from-scratch portfolio analysis in Chapter 3 took approximately 1000 seconds to perform without the use of caching. On our c5.2xlarge cluster (which has almost exactly the same total memory, virtual cores, and cost per hour as the cluster used in Chapter 3), the same portfolio with $w = 250{,}000$ and a typical range of 0–10 invalidated cache edges, caching resulted in 40–90 times faster processing.

4.5.4 Cluster Comparison

Due to the c5.2xlarge cluster's inability to complete incremental analyses with more than 300 invalidated cache edges ($w = 250{,}000$), we conducted another set of experiments using a cluster with a much higher per-node memory capacity. The c5.18xlarge cluster consisted of

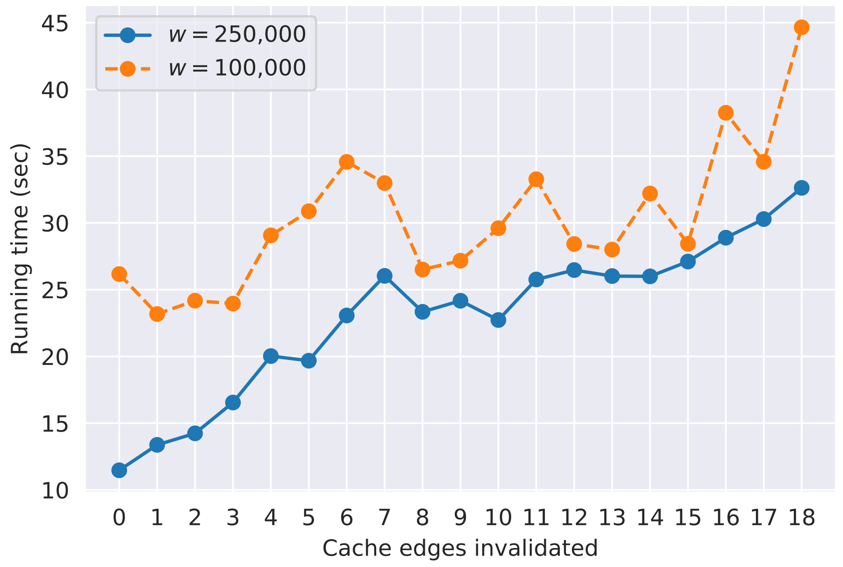

Figure 4.9: Zoomed view of Figure 4.8 for up to 18 invalidated cache edges.

40 c5.18xlarge Amazon EC2 instances. Each c5.18xlarge instance had 144GB of memory and 72 virtual cores, running on the same CPU series as the c5.2xlarge nodes. Both the c5.18xlarge cluster and original c5.2xlarge were roughly equal in terms of memory and compute power. In total, the c5.2xlarge cluster had 5,600GB of memory and 2,800 vCPUs, while the c5.18xlarge cluster had 5,760GB and 2,880 vCPUs. Both clusters had approximately the same hourly rental cost.

In Figure 4.10, we show the incremental analysis running time of the two clusters with $w = 250,000$ from 0 to all 702 invalidated cache edges. As seen in Figure 4.8, the c5.2xlarge cluster runs out of memory while attempting to process any incremental analysis with greater than 300 invalidated cache edges. The c5.18xlarge cluster was able to compute all incremental analyses without running out of memory, despite having approximately equal total memory capacity. This is because each compute node must hold one copy of the portfolio graph in memory at all times during processing. The graph is shared across all worker threads. Since there are only 40 compute nodes in the c5.18xlarge cluster, only 40 copies of the graph need to be held in memory across the entire cluster. In the c5.2xlarge cluster, 350 copies must be held in memory, resulting in limited memory when more cache edges have been invalidated and more shards of the graph need to be loaded.

For the incremental analyses the c5.2xlarge cluster was able to process, the c5.18xlarge cluster had slower running times in general. Since each instance of the c5.18xlarge cluster

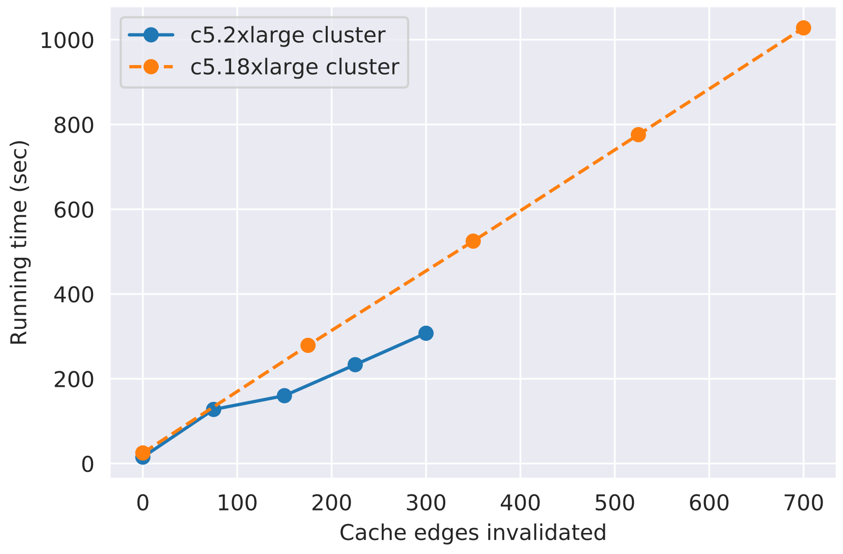

Figure 4.10: Running time of an incremental analysis as a function of the number of invalidated cache edges across different cluster configurations. Running times are averaged across 5 runs.

needed to process a large number of trials to maximize CPU utilization, the 10,000 trials were bundled into a small number of jobs. If certain jobs contain longer running trials, then the entire analysis may wait for the completion of one or two jobs. Similarly, if one instance encounters rate limiting or network interruptions, a single large job can be significantly stalled before eventually completing or being rescheduled by the queue. These problems were largely avoided on the c5.2xlarge cluster, due to its finer distribution of trials.

We note that the difference in performance between the two clusters can be attributed to poor load balancing on the c5.18xlarge cluster. With a more nuanced distribution of trials to each worker (e.g. dynamic load balancing [33]) we expect the c5.18xlarge cluster to have roughly comparable performance to the c5.2xlarge cluster. However, because we believe that the extreme majority of incremental analyses will contain far fewer than 300 invalidated cache edges, we leave this for future work. To cover the unlikely case when an incremental analysis requires recomputing a significant percentage of the portfolio graph, a check can be added that provisions higher-memory resources when the number of invalidated cache edges is large.

Figure 4.11 shows the results for a smaller range of invalidated cache edges corresponding to typical portfolio updates. As with Figure 4.10, the c5.18xlarge cluster performed poorer in comparison, taking on average about 10-15 seconds longer than the c5.2xlarge cluster, with

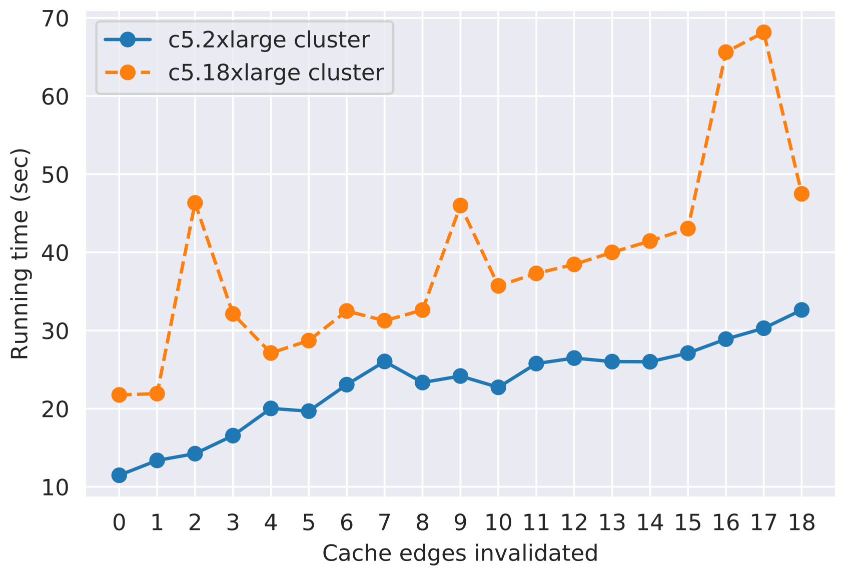

Figure 4.11: Zoomed view of Figure 4.10 for up to 18 invalidated cache edges.

some spikes taking more than 35 seconds longer. The spiking can also be attributed to the coarse distribution of work on the c5.18xlarge cluster.

Since an incremental analysis with more than 10–20 invalidated cache edges would be atypical, we conclude that the c5.2xlarge cluster is generally the better cluster for processing incremental analyses due to its faster running time.

4.5.5 Use Case Evaluation

In this section, we explore the running time of incremental analyses over the set of common reinsurance use cases described in Section 4.4.2. We used the shard size parameter $w =$ 250,000 in these experiments, as it resulted in the best running times for small numbers of invalidated cache edges in the experiments reported above. The use cases can be divided into two categories, of which we evaluate separately, depending on the parts of the portfolio graph affected by the application of the use case.

Changes to reinsurance treaties. The first class of use cases only affects the *reinsurance structures* of the portfolio graph. Portfolio updates of this class include the "pricing", "retrocession pricing" and "renewal planning" use cases. Low running times are extremely desirable for these types of analyses, as they are typically done while negotiating terms over the phone. Since multiple reinsurers are competing for the same pool of client contracts, being early to make a fair offer significantly increases the reinsurer's chances to get a share of the contract. Since all reinsurance structures are part of the sink shard in the $w = 250{,}000$ partition of the portfolio graph, these updates do not invalidate any cache edges and thus, as reported in Section 4.5.3, take around 11 seconds to perform on average.

Changes to primary insurer structures. The second class of use cases covers modifications that affect the structure of the portfolio's *underlying primary insurance contracts*. This covers the "new business", "renewal" and "binder book update" use cases.

In the "new business" use case, the vertices representing the added primary insurer's structure are added to the portfolio as a new shard. The edges connecting the new client to the reinsurance portfolio are added as (initially invalidated) cache edges. The imbalance in the partition this may create can be corrected in the next from-scratch portfolio analysis.

In the "model update" and "binder book update" use cases, the vertices that load the input loss distributions are modified to load the updated loss distributions. For both use cases, all shards containing the new or modified vertices must be reevaluated, as well as their dependent shards.

For both use cases, the running time was the same and depended primarily on the size of the primary insurer shard being added or modified. Figure 4.12 shows the running time for the "new business" use case on our c5.2xlarge cluster as a function of the size of the added primary insurer structure. Typical primary insurers cover up to 100,000 locations, a scenario that led to the incremental analysis taking no more than 16 seconds on our cluster. Some large primary insurers insure 1,000,000–10,000,000 locations. The running time of an incremental analysis naturally increases for such updates. Adding an atypically large new business of 10,000,000 locations is equivalent to increasing the total size of the portfolio graph by about 14% and resulted in an average running time of approximately 230 seconds. This is still 4 times faster than a from-scratch analysis and is acceptable, given that "new business" portfolio updates (especially of this size) usually do not happen in "real time" while negotiating treaty terms over the phone.

4.6 Conclusion

We have demonstrated how to support fast incremental analyses on complex location-level reinsurance portfolios containing hundreds of millions of contracts. Fast incremental analyses

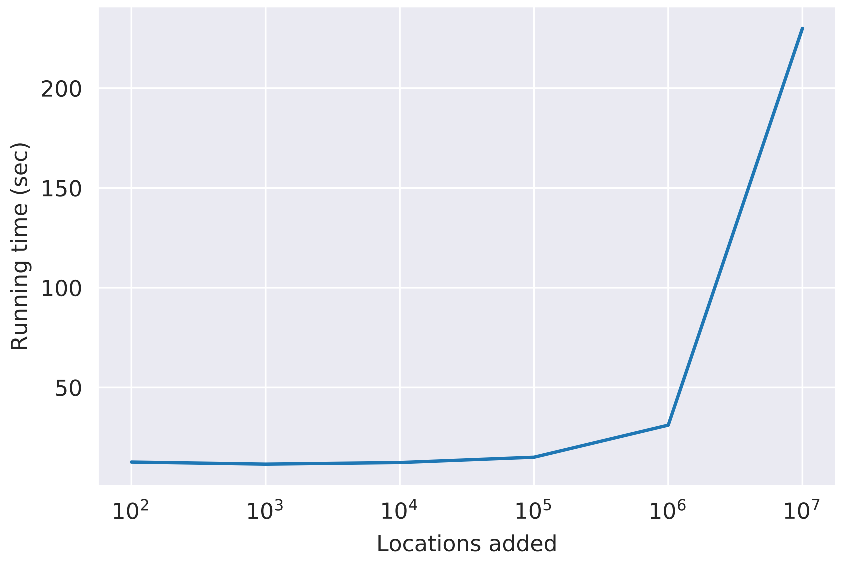

Figure 4.12: 10,000-trial running time as a function of number of locations added for the "new business" use case. Running times are averaged across 5 runs.

are achieved by selecting a small number of cache edges in the graph whose loss distribution data is cached on distributed storage. This allows an incremental analysis to avoid reevaluating the portion of the portfolio that feeds into a cache edge unless a portfolio update invalidates the loss distribution associated with the cache edge. As verified using a number of typical use cases, our cache edge selection strategy is resilient to multiple modifications to the graph, and it exposes a tuning parameter to tune the trade-off between storage cost and reducing the portion of the portfolio to be reevaluated in an incremental analysis.

Reevaluating only part of the portfolio graph in an incremental analysis turned loading the portfolio graph into memory into a bottleneck, which we alleviated using a hierarchical graph representation. At the bottom level is the set of shards into which the graph is partitioned by the cache edges. The dependencies between shards are recorded in a small shard graph. This shard graph allows us to identify the shards required to complete a given incremental analysis, so that shards unnecessary to the current analysis are never loaded.

Using a 2,800-vCPU cloud cluster, we observed incremental analysis running times as low as 11 seconds with only 702 cache edges for several reinsurance analyst use cases. Without caching, the same incremental analysis took over 1000 seconds.

With the introduction of caching, our system from Chapter 3 now supports interactive workflows, making it an *interactive big data decision support system*. Our system is capable of processing several terabytes of input data through a directed graph of hundreds of millions of vertices representing financial transformations. Our system outputs a single portfolio loss distribution, which can be used by reinsurers to help price reinsurance contracts and help determine which contracts they should underwrite and add to their portfolio. As explored through several use cases in this chapter, the introduction of caching allows reinsurers to see the impact adding a specific contract or adjusting the terms of an existing contract has on the loss distribution of their portfolio within approximately 11 – 16 seconds of making the request. This fast response time allows reinsurers to operate the system interactively, making additional adjustments and requests to the system based on the previous responses generated by the system seconds ago.

Chapter 5

Background: Online Analytical Processing and Staleness

In this chapter, we give an introduction to Online Analytical Processing (OLAP) and staleness in distributed data stores. This will serve as the background for Chapter 6, where we will explore staleness in the context of real-time OLAP systems.

5.1 Background

5.1.1 Online analytical processing

Online analytical processing (OLAP) data stores are designed to quickly present generally summarized, broadly scoped information to decision makers (for example, managers and executives) in order to allow them to make informed decisions. A typical OLAP query often involves operating on several thousands or millions of individual data points in the system. Traditional transactional databases, while well suited for smaller queries that operate on one or two dimensions of the data, are ill-suited for executing such large-scale queries in an acceptable amount of time. As such, OLAP data stores, built specifically to efficiently compute large-scale decision support queries, have grown into a billion-dollar industry [23].

In general, OLAP data stores can be viewed on an abstract level as a large, multi-dimensional data cube [90]. A data cube is composed of several dimensions, where each dimension corresponds to a specific property, entity or metric that is deemed important for making OLAP-supported decisions [41]. For example, time, location and product would be three common dimensions for a data cube representing sales by a company. A hierarchy is associated with each dimension, and each record written to an OLAP data store contains a value for each dimension at the finest possible hierarchical level. Each record also contains one or more measure values, which can be combined using aggregation functions (e.g. *sum*, *max*). For example, the data cube for a retail store chain may have dimensions for time and region. A hierarchy for the time dimension may be year, month, day and hour, while the hierarchy for region may be country, province, city and address. Each sale at a store would write a record with the time and location to the OLAP store, along with the sale price as the measure value.

Using a data cube and its associated hierarchies for each dimension, users can summarize or eliminate specific hierarchical levels or dimensions to generate a more compact data cube, called a cuboid. Every cuboid has its own measure values, which can be found by aggregating the measure values of all records within the hierarchies covered by the cuboid. Figure 5.1 illustrates a simple data cube, where the cuboid for 2020 US sales is composed of the sum of four smaller cuboids representing each quarter of the year.

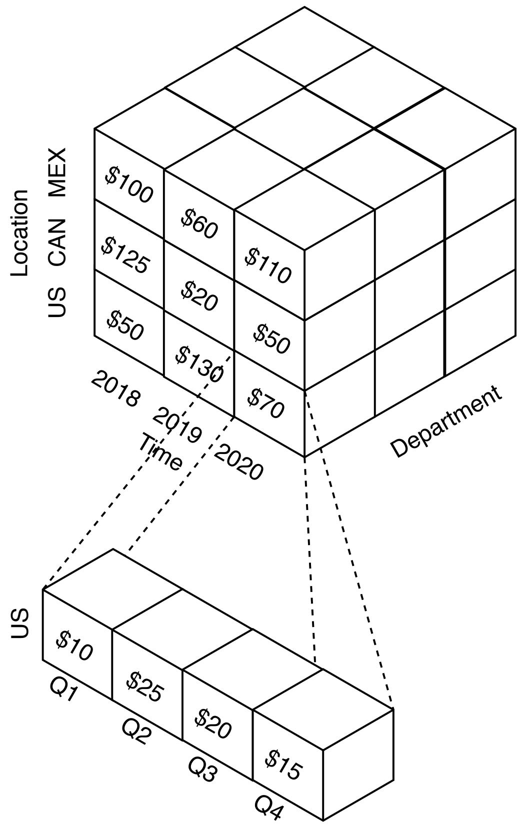

Figure 5.1: A simple data cube demonstrating how a cuboid can “contain” other cuboids.

The set of data cube operators most commonly seen in the literature are roll-up, drill-down, slice and dice [3, 38, 41, 89]. Rolling-up a data cube essentially consolidates levels of a dimension into a single level. As an example, consider a data cube where one dimension represents time. The cube has 365 cuboids in that specific dimension alone, representing revenue (as the measure value) for each day in a specific year. An analyst may not want that level of detail, and decides to roll up the time dimension of the cube to the next level in the hierarchy, the month. A new cube is generated where there are now only 12 values in the time dimension instead of 365. The measure values in each cuboid now become the aggregation of the measures of each of the cuboids in the previous cube, grouped by month. In this case, it makes sense for the aggregation to be the *sum* function, but other aggregation functions may be used.

Drill-down operates in a somewhat opposite manner to roll-up. Instead of summarizing data and reducing the number of cuboids in a cube, drill-down introduces more detail to the cube. While roll-up steps up a level of the dimension hierarchy (say, day to month), drill-down steps down a level in the hierarchy (month to day). Continuing from the previous example, if the analyst decides they would now like to look at daily revenue once again, they perform a drill-down operation on the time dimension, generating a new cube with 365 sets of cuboids, one for each day in the year.

Slicing allows a user to remove specific elements of the hierarchy for a dimension from the cube, resulting in a smaller cube where the specified elements of the dimension have been removed entirely. A dice operation is simply multiple slice operations across multiple dimensions executed at once. An analyst wanting to determine the revenue during the first quarter of the year may do so by slicing all months other than January, February and March.

5.1.2 Distributed Consistency

Quorum-based Replication

To make a data store highly available, data store architects often turn to replication. Replication is the act of storing data on multiple independent nodes, in such a way that if one node becomes inaccessible, there exist several other functioning nodes that contain the same data. Replication can also serve to further scale-out or load-balance a system, as read-only traffic to specific rows of data no longer has to hit the same node; load can be spread out across the different replicas. Replication is also important for data stores which have users in significantly different locations in the world, as reading or writing to a node across the world is likely to have unacceptable levels of latency. Having replicas spread out across the world can help solve this problem as well.

While introducing replication to a system solves many issues important for data accessibility, it also introduces new problems. Now that a system has, say N replicas of its data distributed on different nodes, the data store's existing algorithms for reading and writing need to be modified. Since keeping all replicas up to date is critical, writes need to arrive at each replica in a timely manner. As write operations now must write to multiple replicas, each replica is likely to have a significant period of time where its contents differ from the other replicas (due to factors like varying network latency and workload). This desynchronization of data complicates read operations, as now measures must be taken to ensure the result of the read is up to date. This is best summed up with the CAP theorem [35], which states that no distributed data store can ever simultaneously provide strong consistency, availability and partitioning (or, fault-tolerance) guarantees.

One widely popular method of approaching these problems is to implement a quorum consensus algorithm. Under this scheme, two new variables are introduced to the system, W, the "write quorum" and R, the "read quorum". Under a write operation, all N replicas must be sent the write. After W replica responses indicated the write was committed, the write operation, from the client's perspective, is complete. Note however that in the background, writes for the remaining $N-W$ replicas are in progress. Under a read operation, all N replicas are queried for the same data and, like quorum-based writes, the client waits until it receives R replica responses. Once the R responses have returned, the most up-to-date response is selected, using some means of comparison (in key-value stores, this is usually a simple timestamp storing the time of last write).

Figure 5.2 presents a graphical representation of a read in a quorum system. The client C begins by sending a read request to each of the N replicas, represented here as B_1 through B_3. Once the client receives $R = 2$ replica replies, represented as the dotted lines, the client selects the most up-to-date response, completing the read operation. The remaining $N-R$ responses (1 in this example) are ignored by the client. Figure 5.2 could also be used as an example of a write operation in a quorum system. The write is sent to all N replicas and, after the client receives $W = 2$ responses, the write is considered complete.

Implementing these quorum rules is not enough to guarantee up-to-date reads. Values of N, W and R must be configured in such a way to ensure that at least one of the R responses contains the most up-to-date version of the data being queried. For example, consider the case where `[N=3, W=1, R=1]`. Under these parameters, all writes complete once the data has been committed to one replica, while all reads complete when the client receives its first response from a replica. In other words, a write followed by a read on the same point will only be up-to-date if the first replica to reply to the read is the first replica to reply to the write.

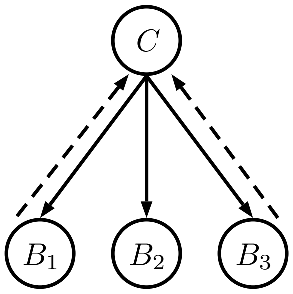

Figure 5.2: An example of a read operation using quorum rules. C is the client, while B_1, B_2 and B_3 are the nodes containing the replicated data relevant to the query. Solid lines represent to read requests, and dotted lines represent responses to the client.

Obviously then, reads are not guaranteed to be up to date in this system.

A well known property of quorum consensus systems is that reads are guaranteed to always be up to date if $W + R > N$. Under this condition, for any read/write pair, there is always at least one more read or write replies than the number of replicas. It then follows that there must be at least one replica that has replied both to the write as well as the read. Therefore, if W, R and N are configured such that $W + R > N$, each read is guaranteed to read the latest write. However, it is also well known that large W, R and N can negatively impact several performance characteristics of a data store.

Types of Consistency

In general, we refer to consistency as a system state of a data store where read or query operations retrieve sufficiently recent results from the data store. However, "consistency" is only a useful term from a general perspective. One survey on the topic [85] compiles several popular definitions of consistency which are applicable mostly to key-value stores. Here we present the definitions relevant to this chapter described in the survey and other popular literature, with some slight revisions so that the definitions may be applicable to data stores where reads operate on multiple writes.

"Strong consistency" guarantees the system offers the best possible consistency at all times. Specifically, under strong consistency, all reads are guaranteed to use the values of the latest relevant writes to the data store. Since this guarantees perfect consistency, the CAP theorem implies that any system that offers strong consistency will have poor availability, partitioning, or both. For example, Cassandra [54] under the $W+R > N$ quorum configuration guarantees strong consistency, but offers poorer availability when compared with less strict configurations.

"Read my writes" guarantees strong consistency, but only for the writes initiated on the same client as the reader.

The "monotonic reads" guarantee requires that, on each client, once a point has been read by a query, any subsequent reads of the same point must read the same or more recent value as the previous read. Essentially, this means that consistency cannot degrade over time.

"Eventual consistency", described in more detail in [92], guarantees that, given a finite amount of time, the system will eventually always read the most recent writes. This is the most lenient guarantee of all, but is typically the least sensitive to partitioning and can provide almost arbitrarily high availability. Additionally, while these aren't covered under the CAP theorem, eventual consistency also often offers the best latency and throughput.

A system that obeys "k-regular consistency semantics" [6] guarantees that, provided the read and write quorums do not overlap, the result from one of the last k completed writes is returned. If there is overlap between the read and write quorums, either a result from the last k completed writes is returned, or the result from a more recent write which has not yet completed on W replicas is returned.

A system that obeys "bounded staleness" has the view that a bounded amount of staleness (that is, the possibility of system inconsistency) in a system is acceptable. k-regular consistency semantics can be viewed as a type of bounded staleness that allows for a bounded amount of staleness with respect to the versioning or ordering of writes.

A final consistency guarantee covered in this thesis, described in [16, 17, 18], is probabilistically bounded staleness (PBS). PBS is similar to bounded staleness, in that they both provide guarantees based on time or version bounds. However, while bounded staleness is strict and guarantees every query obeys its bound, PBS requires each query to meet its bound with a certain probability. PBS is covered in depth in the following section.

5.2 Related Work

5.2.1 Staleness in OLAP

Several papers have been published on the topic of imprecise or ambiguous data in OLAP systems [19, 24, 63]. However, instead of focusing on the uncertainties that arise from lack of synchronization and eventually consistent data, these papers focus on dimensional data and measures which are uncertain by nature. For example, all measures may have a certain amount of error with a known distribution. Another work [19] proposes a modified OLAP model which incorporates this concept of uncertain data and, much like this chapter, presents different metrics of query correctness within this context. However, since the metrics are within the context of inherently uncertain data, the model and corresponding metrics are inapplicable to eventually consistent OLAP systems studied in this chapter.

Another work [75] presents a middleware for distributed OLAP systems which manages replication, insert and query operations through the system to guarantee a certain freshness bound. The paper describes a “freshness index”, which measures how consistent a given replica is at the current point in time from a scale of 0 to 1. The paper also presents a model wherein query-answering OLAP nodes receive batch updates from writable nodes running online transaction processing (OLTP) software. Under this model, the authors compute the freshness index by dividing the time of the last update to an OLAP node by the commit time of the most recent transaction on an OLTP node.

BatchDB [60] is a real-time OLTP/OLAP hybrid system that takes consistency into account by ensuring all OLAP reads operate on a consistent snapshot of the database at a point in time. That is, each read only accesses the writes before a certain timepoint, and no write after that point is read. There is no guarantee how long it may take a write to be visible in the system, and no discussion regarding what relative error or number of missed writes in an OLAP query may look like.

Analyticdb [100] is another real-time OLAP system. It allows the user to send writes in a strongly consistent mode (at the cost of increased latency) and an eventually consistent mode (at the cost of data freshness). Under the eventually consistent mode, all writes are said to be visible after a “bounded delay”, but there is little discussion regarding how this effects the number of missed insertions or the relative error of OLAP queries.

Another work [55] describes an eventually consistent OLAP system where the user can submit a query with an acceptable staleness range, specified as seconds since last update. If a query accesses data from a snapshot which has not been updated within the provided acceptable staleness range, the query is re-routed to another server to get a more recent version of the data (at the cost of increased latency). To evaluate their system’s level of consistency, the authors measured the amount of time it takes for writes to become visible to reads (without using the acceptable staleness range feature). In their experiments, writes were typically visible after 1 millisecond.

5.2.2 Probabilistically bounded staleness in key-value stores

In recent years, there has been increased interest in providing consistency guarantees for quorum systems that operate probabilistically, with several papers published on the subject [2, 8, 16, 17, 18, 59, 62, 95]. In [17], the authors examine eventual consistency from the perspective of quorum based key-value data stores and present a probabilistic metric of staleness for partial quorums. First a closed form method for determining the probability of no intersection between randomly selected read and write quorums is given:

$$p = \left(\frac{\binom{N-W}{R}}{\binom{N}{R}} \right) \tag{5.1}$$

In this equation, the numerator becomes the number of different ways R replicas can be selected from the set of replicas that do not contain the most recent write $(N - W)$, and the denominator becomes the number of ways R replicas can be selected from the total number of replicas N. In other words, the number of quorum selections that yield no intersection between reads and writes, and the number of possible quorum selections in total.

With this in mind, a PBS k-staleness is introduced whose definition is included from [17] below:

Definition 1 (PBS k-staleness)**.** *A quorum system obeys PBS k-staleness consistency if, with probability $1-p$, at least one value in any read quorum has been committed within k versions of the latest committed version when the read begins.*

The value p, while not included in the name of definition, is also a parameter similar to k. For applications where there is little tolerance for stale results, a system that obeys PBS k-staleness for very low values of p would likely be more desirable than a system that only obeys PBS k-staleness for larger values of p. PBS k-staleness essentially describes the probability of a read having k-regular consistency semantics (as described in Section 5.1.2).

Following this, a closed form for determining the upper bound of the value of p for PBS k-staleness is derived. The closed form equation does not take into account time, and instead assumes each write is committed to only W replicas. Because of this, p becomes the probability of the read quorum not intersecting with any of the write quorums, which can be determined using equation 5.1, for k versions:

$$p \leq \left(\frac{\binom{N-W}{R}}{\binom{N}{R}} \right)^k \tag{5.2}$$

The closed form describes an upper bound of p, as it does not account for the fact that other replicas may receive and commit the write before the read is processed.

Additionally, a second type of bounded staleness is presented in the paper. PBS t-visibility imparts time onto the metric, and therefore reflects how long it takes for writes to become eventually consistent. The definition as seen in [17] is presented verbatim:

Definition 2 (PBS t-visibility)**.** *A quorum system obeys PBS t-visibility consistency if, with probability* $1-p$*, any read quorum started at least t units of time after a write commits returns at least one value that is at least as recent as that write.*

In PBS t-visibility, t encodes a lower bound on the range of time between the completion of the latest partial write quorum (W write replies from W replicas) and the time of initiation of the read quorum. Since writes in a quorum system arrive at different replicas at different times, PBS t-visibility is useful for determining the impact of system-specific read and write latencies on consistency. However, because t-visibility strongly reflects the system, and not the quorum configuration as k-staleness does, it is difficult to construct a closed form analysis of t-visibility.

To address this, a method for examining t-visibility in the context of Dynamo-style (that is, distributed, quorum-based, key-value stores) data stores is discussed. In Dynamo-style stores, inconsistency is introduced solely through message reordering. A Dynamo-style system where all reads and writes are serialized and executed in the same order across all replicas would guarantee strong consistency. Therefore, in order to model inconsistency in a Dynamo-style system, message reordering must also be modeled. Using the WARS model, writes and reads are modeled using system-specific time distributions. These distributions are:

- The replica write latency distribution D_w
- The write reply (i.e., acknowledgement) latency distribution D_a
- The replica read latency distribution D_r
- The replica read response latency distribution D_s

For example, a write followed by a read, using WARS, would be modeled as follows. The coordinator begins by sending a write request to N replicas. The distribution D_w is sampled from N times to determine the time taken for each replica to receive the write request and to complete the write locally. The distribution D_a is sampled once for each replica, and summed with each replica's sampled value from D_w to obtain the time at which the coordinator has received the quorum write reply from each replica. Of these, the Wth smallest time is the time at which the coordinator receives W replies from the replicas, satisfying the partial quorum rules for a write. Reads behave similarly, except the distribution D_r is used to sample the time taken for a read request to arrive and complete locally on a replica, and D_s is used to sample the time taken for a read reply to arrive at the coordinator.

As is illustrated in Figure 5.3, t-visibility can be tested using WARS by modeling a single write, waiting t units of time after W write acknowledgements, and modeling a read. If, of the R replicas which responded earliest, at least one completed their read after the write completed, then this specific read is said to contain the most recent write.

This sets the stage for a relatively simple method of determining the value of a system's probability p of PBS k-staleness using event-based simulation. Simply construct a Monte Carlo simulation, where each trial is a WARS-style write, followed by a delay of t units of time, followed by a read. As mentioned above and illustrated in Figure 5.3, this can be easily computed by sampling from D_w, D_a, D_r and D_s and checking if the write and read has been reordered in all of the read replies. If enough trials are simulated, the value of p_{st} is estimated by the ratio of trials which were not consistent over the total number of trials.

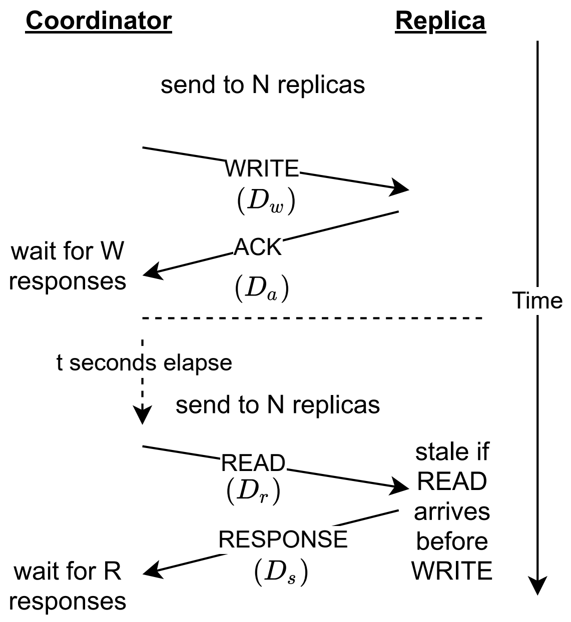

Figure 5.3: The WARS model for Dynamo-style data stores, from [17] with minor adjustments to notation

Chapter 6

Probabilistically Bounded Staleness in Real-time Online Analytical Processing

In this chapter, we present *Aggregate Probabilistically Bounded Staleness* (*A-PBS*). Inspired by the *Probabilistically Bounded Staleness* (PBS) measure [17] for key-value stores, A-PBS measures staleness for *aggregate* queries. [1] While a key-value query only retrieves a single key, an aggregate query in a distributed OLAP system typically aggregates a large set of data items specified by a multi-dimensional bounding box in d-dimensional space. Instead of examining the write/read history of the different copies of a single data item in the case of PBS, the A-PBS measure introduced here depends on the write/read history of the different copies of all data items within a multi-dimensional space, possibly the entire database. This greatly increases the complexity of both measuring and modeling staleness, and clearly distinguishes A-PBS for distributed OLAP systems from the PBS measure for key-value stores.

The A-PBS measure introduced here includes a formal model for describing an OLAP system's data stream and the state of consistency for individual aggregate queries. A-PBS uses either the number of missed inserts or the relative numerical error (incurred by reading stale data) of the query result to quantify staleness. These different cases do not arise for PBS in data stores. We introduce (t, c)-*staleness* for queries that have missed more than c inserts and were issued t time units after the last write, and (t, ϵ)-*staleness* for queries that have a relative numerical error greater than ϵ and were issued t time units after the last write. These measures are then utilized to introduce the following system-wide probabilistic staleness measures: *bounded* (t, c)-*staleness* and *bounded* (t, ϵ)-*staleness*.

Simulation

To complement A-PBS, we also present a generic model and corresponding Monte Carlo simulation of data aggregation in quorum-replicated distributed OLAP systems. Given a list of

[1]The core contributions of this chapter were published in the proceedings of the 21st International Database Engineering and Applications Symposium [21]

system parameters, our model and simulation can be used to estimate staleness for aggregate queries, thereby enabling the exploration of the trade-offs between consistency and latency in quorum-replicated distributed OLAP systems.

Case Study

We used the *VOLAP* [29] quorum-replicated distributed OLAP system for a case study to evaluate our A-PBS measure and Monte Carlo simulation. We observed that the bounded (t, ϵ)-staleness of aggregate queries predicted through our A-PBS measure and Monte Carlo simulation was close to the actually observed staleness of aggregate queries in *VOLAP*.

Our A-PBS analysis revealed that for *VOLAP* a partial quorum with `[N=3, W=0, R=1]` is "good enough" in practice. Even very large aggregate queries that cover the entire database and are issued only 10 milliseconds after the last insert have ≈80% probability to have zero staleness. If staleness occurs for such aggregate queries, the number of missed data items is expected to be low, as only 0.5 inserts are missed on average. This results in only a very small numerical error in the aggregate query result for the *mean* and *sum* aggregation functions, and very close to zero probability of any numerical error for the *max* aggregation function.

6.1 Aggregate Probabilistically Bounded Staleness (A-PBS)

In this section, we present *Aggregate Probabilistically Bounded Staleness (A-PBS)*, a means of analyzing consistency in distributed OLAP systems. Like PBS, A-PBS defines metrics examining the consistency of aggregate queries in terms of missed writes. Unlike PBS, since aggregate queries, especially in OLAP, are conventionally numerical by nature [41], new consistency metrics are introduced that view consistency from the perspective of numerical error.

6.1.1 Data Streams and Queries

While queries in key-value stores essentially retrieve a single value specified by a key from a node, aggregate queries may involve a much larger percentage of the data. For example, a single query in a typical aggregate OLAP system can operate on anything between a single point in the system to all points across all nodes in the system. This is an important distinction which divides key-value staleness analysis from aggregate OLAP-style staleness analysis. Therefore, we begin with a set of definitions describing the stream of incoming insert operations to a

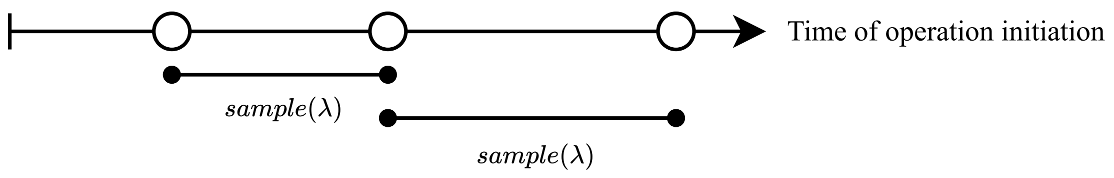

Figure 6.1: A DATA(3, λ, D) insert stream. The white circles represent the points in time at which inserts in the stream are sent from the client. The amount of time between adjacent inserts is determined by sampling from an exponential distribution with parameter λ.

system, henceforth described as the *data stream*, and the *coverage* and *aggregation function* of an aggregate query, both fundamental for further discussion of correctness in aggregate stores.

Definition 3 (Input data stream DATA(n, λ, D))**.** *DATA(n, λ, D) is a stream of n insert operations, where each insert, with measure value sampled from the distribution D, is sent to the system according to a Poisson process with rate λ.*

Definition 4 (Aggregate query Q)**.** *An aggregate query Q is defined by an aggregate function A and a coverage C which describes the probability each insert has of being required in the computation of the aggregate function.*

Figure 6.1 presents a graphical representation of a simple data stream. Since each insert in the stream is sent to the system according to a Poisson process, the amount of time between adjacent inserts in the stream follows an exponential distribution using the same λ parameter.

6.1.2 $(\mathbf{t}, \mathbf{c})$-staleness

We define the notion of a *partially committed* insert (within the context of partial quorum systems) as an insert that has been written to at least W replicas, but less than N replicas. A read operation that depends on one or more partially committed inserts may or may not be consistent, depending on which replicas are accessed by the system during the read.

From this, we define our first metric of staleness in an aggregate setting.

Definition 5 ((t, c)-staleness)**.** *Given an insert data stream DATA(n, λ, D), a query Q, initiated t units of time after each insert in the stream has been partially committed, has (t, c)-staleness if and only if more than c insert operations covered by the query's bounding box were not included in the computation of the aggregate function A.*

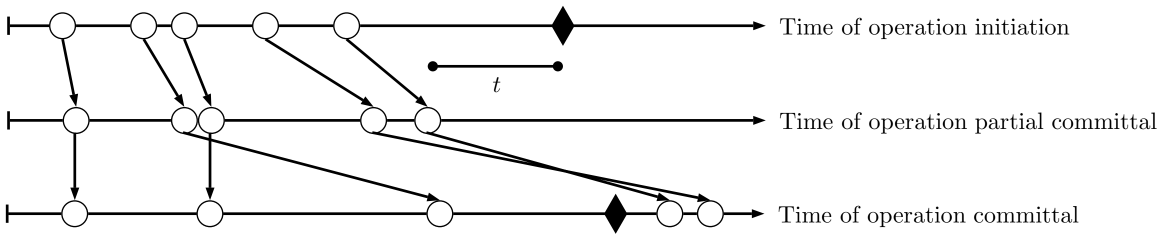
Time of operation initiation
t
Time of operation partial committal
Time of operation committal

Figure 6.2: A query that has $(t, c{=}1)$-staleness. The upper bar represents the time of initiation of a query or insert, the middle bar represents the time at which each insert has been partially committed and the bottom bar represents the time at which the corresponding insert is readable, or the cutoff time at which the query begins to read committed inserts. The last two inserts in the stream and the query are reordered, so more than $c = 1$ inserts are missed.

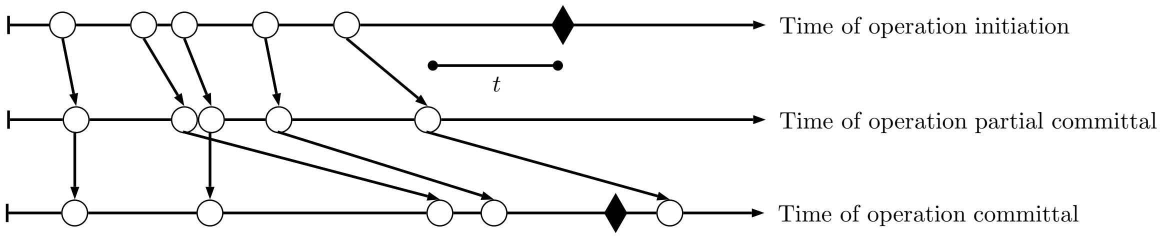
Time of operation initiation
t
Time of operation partial committal
Time of operation committal

Figure 6.3: A query that does not have $(t, c{=}1)$-staleness. Since $c = 1$, the reordering of the last insert in the stream and the query does not impact (t, c)-staleness.

Figures 6.2 and 6.3 demonstrate queries with (t, c)-staleness and without (t, c)-staleness, respectively, and provide a visual representation of (t, c)-staleness and the "slack" parameters t and c. As time proceeds from left to right, insert operations, represented by white circles, are sent to the system according to a Poisson process with parameter λ. Consequently, the distance of time between consecutive inserts obeys an exponential distribution with the same λ parameter. Once each insert has been partially committed (for example, W replica replies have been received in a partial quorum system), t units of time are waited until the query, represented by the black diamond, is initiated.

Since aggregate query freshness depends on each point in a data stream, ensuring that queries do not have (t, c)-staleness is potentially much more difficult to achieve than the key-value PBS equivalent.

6.1.3 Staleness and Error

In key-value PBS, read operations are deemed fresh or stale solely based on the number of writes missed by the read operation. In A-PBS, the number of missed insert operations is not the only means of quantifying query staleness. The numerical error of the aggregation results from a query can also be used as an indicator of a query's staleness, which can be useful in understanding the practical impact staleness has on a query's result and how different the result would be under a perfectly consistent system.

We refer to the (correct) result of a given aggregate query on a perfectly consistent system as the *true aggregate value*, and the (possibly incorrect) result observed from issuing the same query on an eventually consistent system as the *observed aggregate value*. We define the error of a query as the relative error of the true aggregate value and the observed aggregate value, or, more formally:

Definition 6 (Aggregate relative error)**.** *The aggregate relative error of a query Q with an observed aggregate value of o and a true aggregate value of v is* $\frac{|o-v|}{v}$.

With this in mind, we present a definition, much like (t,c)-staleness, to classify the result of a query as being acceptably consistent, with respect to a relative error:

Definition 7 ((t,ϵ)-staleness)**.** *Given an insert data stream DATA(n, λ, D), a query Q with aggregation function A, initiated t units of time after each insert in the stream has been partially committed, has (t,ϵ)-staleness if and only if the query's relative error is greater than ϵ.*

The absence of (t,ϵ)-*staleness* essentially places an upper bound ϵ on the relative error of a query. A query whose relative error is less than or equal to ϵ is said to have an acceptable amount of error, in which case the query is acceptably consistent (similar to c in (t,c)-staleness). Where (t,c)-staleness measures staleness depending on whether or not points are present during the time the aggregation takes place, (t,ϵ)-staleness measures staleness based on the result of the aggregation. Thus, (t,ϵ)-staleness is dependent on the aggregation function A used by the query, as well as the distribution D of measure values in the data stream.

Another important difference is that, unlike for (t,c)-staleness, points missed by a query only impact (t,ϵ)-staleness if the missed point has an impact on the final aggregation. For example, missing a single point will not likely have an impact on (t,ϵ)-staleness when the aggregation function is *max*, as that point would have to be the largest point covered by the query in order to impact the result of the aggregation function. Conversely, missing a single point when the aggregation function is *count* is guaranteed to increase the relative error of the query.

6.1.4 Probabilistic Staleness

So far, we have examined how staleness impacts individual queries. Since we would like to be able to reason about a system's accuracy as a whole, we now introduce staleness metrics on a system-based level.

Definition 8 (Bounded (t, c)-staleness)**.** *A system for aggregate queries on an input data stream DATA(n, λ, D) has bounded (t, c)-staleness with probability p if and only if, with probability p, an aggregate query Q with coverage C does not have (t, c)-staleness.*

Definition 9 (Bounded (t, ϵ)-staleness)**.** *A system for aggregate queries on an input data stream DATA(n, λ, D) has bounded (t, ϵ)-staleness with probability p if and only if, with probability p, an aggregate query Q with coverage C and aggregation function A does not have (t, ϵ)-staleness.*

Using *bounded (t, c)-staleness* and *bounded (t, ϵ)-staleness*, the probability of a system being unacceptably inconsistent (determined by c or ϵ), can be described. For example, given a system constantly ingesting a stream of new points at a rate of 10,000 a second, if we would like to approximate the probability of a query with coverage C returning a result with a relative error of no more than 0.01, we need only to model a stream DATA(n, $\lambda = \frac{1}{10000}$, D), and find the probability p of bounded $(t{=}0, \epsilon{=}0.01)$-staleness for the coverage C and given aggregation function.

6.2 Simulation

In order to evaluate how the different parameters affect staleness within our model, a method for estimating the probability p of bounded (t, c)-staleness and bounded (t, ϵ)-staleness is needed. To accomplish this, we use a Monte Carlo simulation to evaluate repeated trials consisting of an insert stream followed by a query. The result of each trial is whether or not the query has (t, c)-staleness or (t, ϵ)-staleness.

6.2.1 Aggregate Model

The basis of our simulation is a simple model of a distributed quorum-replicated aggregate system with the following properties:

- The set of multi-dimensional point data and their associated measure values is partitioned into m *partitions.*

- Each partition of the data is redundantly stored in N *buckets*. The set of N buckets which replicate a partition is called a *bucket set*.
- Location data used to determine which buckets store which points are held in a structure called an *index*.
- Insertions and queries are produced by *clients*, and are sent to the index to route operations to the relevant buckets.

Insert operations in this model function similar to a typical quorum-replicated key-value store. Inserts are sent from a client to an index, and are then routed to all N buckets within the relevant bucket set. A response is sent to the client from the index once W buckets have reported a successful write.

Like inserts, queries are initiated from a client and sent to an index. An index that receives a query request must route the query to all N buckets for each bucket set relevant to the query. Once at least R bucket set aggregations are received from all relevant bucket sets, the index aggregates the responses from each bucket set and sends the final aggregation to the client.

Figure 6.4 presents a graphical representation of the model.

6.2.2 Simulation Parameters

Each simulated insert represents an insert operation as described in our distributed aggregate model. Using a set of system parameters, which describe the latency timings of insert and query operations of a system, the time at which each simulated insert in the stream is consistent (or readable) at each of N replicas can be determined. The same set of parameters can be used to determine the time a query arrives at each bucket in the system, and the set of R buckets from each bucket set whose responses are the first to arrive at an index. Comparing the insert committal times against the query arrival times can then be used to determine the number of stale inserts.

A summary of the key system parameters is given in Table 6.1. $T(M)$ is a distribution which describes the network latency of sending a message from an index to a bucket or from a bucket to an index. For simplicity, we assume that all query and insertion requests and replies have the same impact on network latency, and thus network latencies for any type of message (insert or query) sent within the system can be drawn from $T(M)$. $T_w(I)$ and $T_w(B)$ are distributions which describe the time taken by an index (I) or bucket (B) to complete the

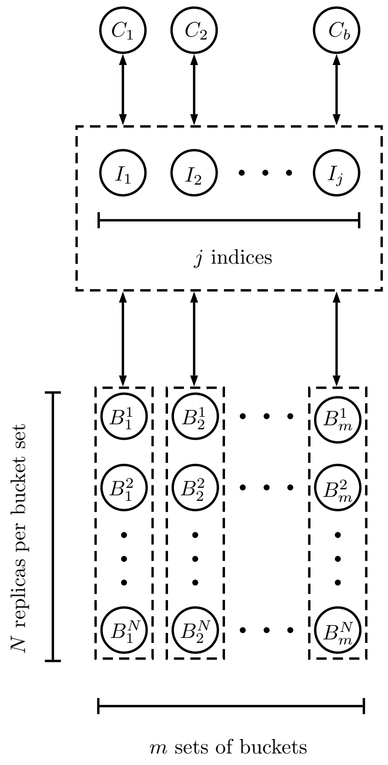

Figure 6.4: Diagram illustrating the node structure of the aggregate model.

local computation required for an insert. Likewise, $T_r(I)$ and $T_r(B)$ describe the time taken by an index or bucket to complete the local computation required for a query.

Name	Description
$T(M)$	Distribution of time taken to send a message from one node to another
$T_w(I)$	Distribution of time taken for local insert work on an index
$T_w(B)$	Distribution of time taken for local insert work on a bucket
$T_r(I)$	Distribution of time taken for local query work on an index
$T_r(B)$	Distribution of time taken for local query work on a bucket

Table 6.1: The set of system parameters

6.2.3 Algorithm

Each trial in our simulation begins by modeling the initiation time of each insert. We refer to the initiation time of an insert as q_i, where i is the number of inserts in the stream so far. We assign the earliest insert in the stream time $q_0 = 0$, and all subsequent inserts $q_i = q_{i-1} + d_i$, where d_i is drawn from the Poisson distribution with parameter λ. The committal time of the insert for each of the N relevant buckets is computed by sampling from $T_w(I)$, $T(M)$ and $T_w(B)$ and adding the insert's initiation time, q_i. The quorum reply time (the time at which the index has received the W write replies) is computed by adding a new sample from $T(M)$ to each bucket replica committal time, and selecting the Wth fastest time. The committal time of each insert, and the insert's quorum reply time are stored for later use.

Once each insert has been simulated, the query simulation begins by determining the time at which all inserts have been partially committed. This is accomplished by taking the max of each insert's quorum reply time from the previous step. The time is then offset by t to get the time of query initiation. This value is added to a random sample of $T_r(I)$ to get the time at which the index has done its local bucket location lookup. Then, for each bucket B in the system, the time of query arrival is recorded by adding a sample of $T(M)$ to the time the index has finished its local work. To determine which buckets are the first R responders, the time of query arrival from each bucket is offset by random samples from $T_r(B)$ and $T(M)$. From this, the time R responses have been received from each bucket set can be observed, and the maximum value is taken to get the time the query has met its read quorum rules for each bucket set.

The query simulation groups all buckets into two sets: those which have responded to the query before R responses have been received from each bucket set, and those which have not. Since those that have not responded in time are not included in the result of the query, they may be safely ignored. The buckets which have responded in time will simulate an aggregation operation on all inserts they contain (within the query's coverage), excluding the points that were committed locally later than the time of query arrival on the current bucket. When evaluating (t,ϵ)-staleness, the aggregation function provided should be used, while when evaluating (t,c)-staleness, the *count* aggregation function should be used. Afterwards, the freshest partial aggregation from each bucket set is determined by selecting the partial aggregation with the greatest *count* value. The freshest partial aggregations from each bucket set are then aggregated together, using the query's specified aggregation function, to compute the observed aggregation value. To compute the true aggregation value, all simulated inserts are aggregated, regardless of committal time. This is the result of the aggregation we would expect to get under a perfectly consistent system. With the true value and the observed value, whether or not the query has (t,ϵ)-staleness can be determined by evaluating the

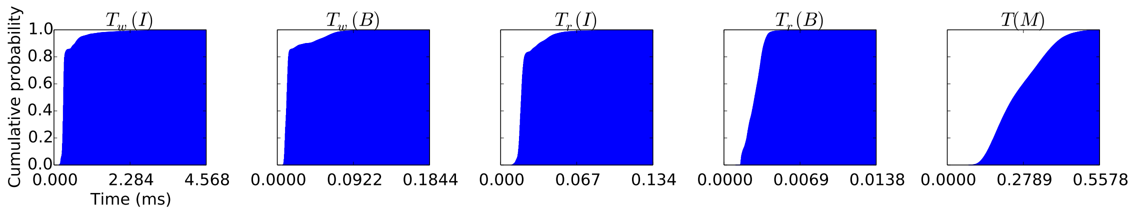

Figure 6.5: Cumulative distribution functions for distributions $T_w(I)$, $T_w(B)$, $T_r(I)$, $T_r(B)$ and $T(M)$ used in Section 6.3. For legibility, only the first 99 percentiles are plotted.

error between the true and observed values. Likewise, (t, c)-staleness can be determined by comparing the observed and true counts with value of c.

6.3 Case Study

In this section, we use the *VOLAP* [29] quorum-replicated distributed OLAP system for a case study to evaluate our A-PBS metrics and Monte Carlo simulation. By using recorded system parameters from *VOLAP* in our Monte Carlo simulation, we compare staleness metrics obtained by simulation against actual staleness metrics observed from an OLAP system.

We obtained the operational latency distributions $T_w(I)$, $T_w(B)$, $T_r(I)$, $T_r(B)$ and the network latency distribution $T(M)$ by sampling the amount of time taken to execute or transmit an insert or query operation in *VOLAP*. The latencies observed for *VOLAP* were from runs on an Amazon EC2 cloud with 8 `c4.xlarge` nodes using the TPC-DS [1] data set with 8 hierarchical dimensions. Read and write latencies were recorded while processing a workload composed of an even mix of queries and inserts. The first 99 percentiles of the distributions used are shown as cumulative distribution functions in Figure 6.5. The ingestion rate was measured to be approximately 20,000 inserts per second with $N = 3$, or $\lambda = \frac{1}{20000}$ seconds between inserts.

To determine the actual probability of bounded (t, c)-staleness on *VOLAP*, we first generate a pool of queries of which we know the approximate coverage and aggregation result. To do this, we submit to the system a stream of 100,000 insert operations, using the TPC-DS [1] data set with 8 hierarchical dimensions as input. A quorum configuration of `[N=1, W=1, R=1]` is used to ensure consistency during the query generation step. After all inserts are complete, the random queries are issued to the system and their results are recorded. Once the pool of queries with known aggregation results and coverage has been generated, we approximate the probability of bounded (t, c)-staleness by repeating several trials of the following. We begin

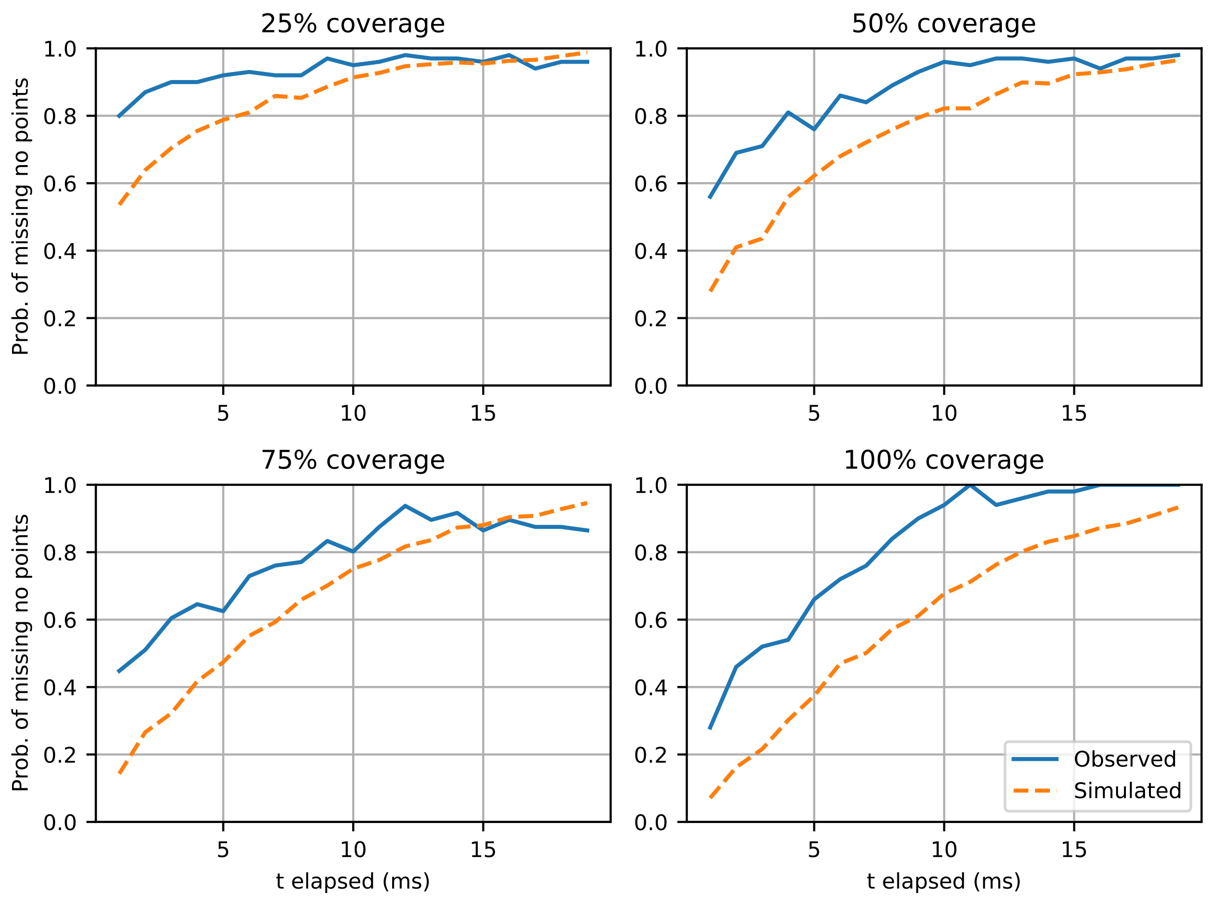

Figure 6.6: Observed and simulated bounded $(t, c=0)$-staleness with `[N=3, W=0, R=1]` and varying query coverage

by clearing all previous inserts from the system and issuing the same stream of inserts (this time with a partial or eventually consistent quorum configuration) used during the query generation step. After waiting t units of time after W quorum responses from each insert have been received, we issue a query from the pool. We compare the possibly incorrect result against the recorded correct result to determine whether the query in this trial was (t, c)-stale or (t, ϵ)-stale.

Figure 6.6 plots the simulated and observed *VOLAP* probabilities of bounded $(t, c=0)$-staleness across increasing t values in the x-axis with varying query coverages. We use the quorum configuration `[N=3, W=0, R=1]` instead of the typical `[N=3, W=1, R=1]`, as setting $W \geq 1$ in our test environment results in the relatively uninteresting case where nearly all queries return correct results. When $W = 0$, queries are initiated t units of time after the last item in the data stream has been sent, without waiting for any partial committal responses. In both the observed and simulated experiments, the probability of bounded staleness is proportional to the coverage of the query. Queries with higher coverage have a lower probability of bounded staleness, as they cover a greater number of possibly unreadable points. For < 100% coverage queries, the simulated probabilities line up reasonably well with the observed probabilities of bounded staleness. For 100% coverage queries, the simulation is somewhat pessimistic compared to the observed probabilities. This is likely because in *VOLAP*, 100% coverage queries are much faster to compute than < 100% queries. The simulation, which uses the same latency distribution regardless of coverage, does not account for that.

In Figure 6.7, we plot the simulated probability and *VOLAP*'s observed probability of bounded $(t,\epsilon{=}0.00001)$-staleness using various aggregation functions and measure distributions. A small amount of relative error (0.00001) is allowed for demonstration purposes, as setting $\epsilon = 0$ is equivalent to bounded $(t,c{=}0)$-staleness for most aggregation functions. We include the *sum* aggregation function, whose rate of change is steady regardless of the number of points (similar to *count*), *max*, as its value is essentially determined by a single point and is therefore highly insensitive to randomly selected missing points, and *mean*, as it is a non-monotonic aggregation function whose rate of change drops as the number of points in the aggregation increases. For the measure distributions, we use the folded normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 1$ to represent a short-tailed distribution of measure values, and an exponential distribution with $\lambda = 1$ to emulate the case where the distribution of measure values is fat-tailed. Both distributions have been selected to yield only non-negative values in order to further contrast *sum*, which is monotonically increasing if D yields only positive values, with *mean*, whose aggregation value can increase or decrease on the inclusion of a single point. 100% coverage queries on a data stream with 100,000 inserts and quorum configuration `[N=3, W=0, R=1]` were used. In both the simulation and the *VOLAP* experiment, the *max* aggregation function has a > 99% probability of the error being bounded by ϵ, illustrating the function's insensitivity to missing points. The *sum* and *mean* aggregation functions have a much lower probability of bounded error, but still have a slightly higher probability than bounded $(t,c{=}0)$-staleness in Figure 6.6 due to the slight amount of slack in ϵ. We note that the *mean* aggregation function has a higher probability of bounded error than *sum*. This is because the amount of relative error incurred by missing a point under *mean* can be offset by missing a point on the side opposite of the true mean. Since we are using positive distributions, a missed point under *sum* cannot be offset by missing a negative

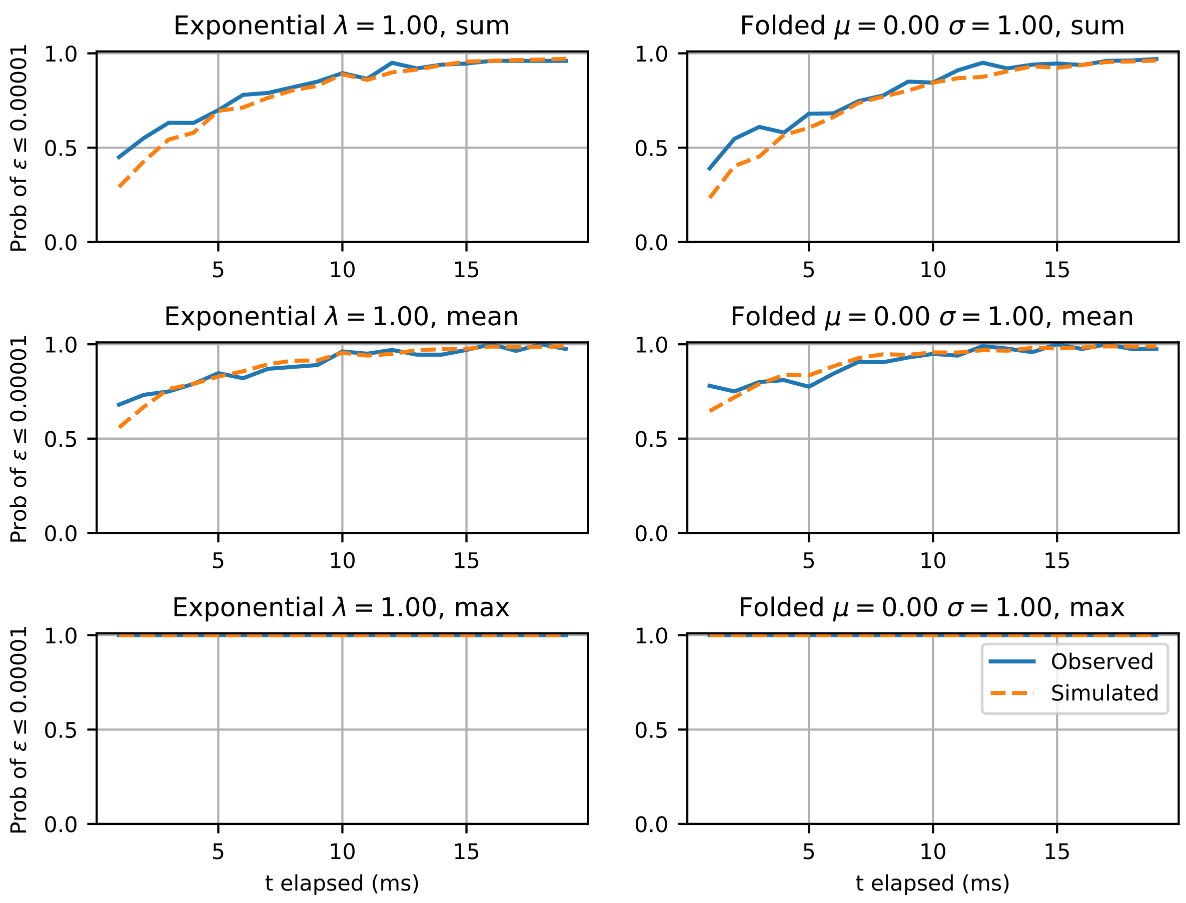

Figure 6.7: Observed and simulated bounded (t,ϵ=0.00001)-staleness with [N=3, W=0, R=1] and varying measure distributions and aggregation functions

point of similar magnitude. A small but noticeable gap in observed and simulated probability of bounded error can be seen for the two measure distributions under *sum* and *mean*. The exponential distribution results in a slightly lower probability with *mean*, as the long, thin tail decreases the likelihood that a pair of missed inserts will offset each other, since the majority of the points lie to the left of the mean. With *sum*, the exponential distribution performs better, as a large part of the total sum is determined by a relatively small number of points with large measure values, which are therefore less likely to be missed compared to the much larger number of points with smaller measure values.

Figure 6.8(a) shows the expected number of missed inserts with varying coverage observed from *VOLAP*. Figure 6.8(b) shows the average relative error of a 100% coverage query on a data stream of 100,000 items with varying measure distributions and aggregation functions, also observed from *VOLAP*. Both figures demonstrate that, in *VOLAP*, when queries are stale, their results are expected to be only one or two points off from the true result.

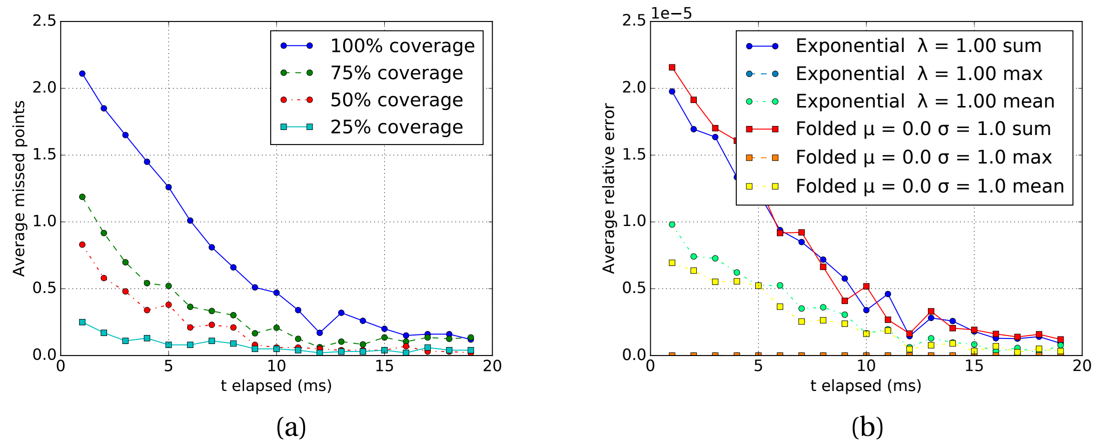

Figure 6.8: Observed average number of missed inserts (a) and relative error (b) with `[N=3, W=0, R=1]`

Figure 6.9 demonstrates the impact a system's read and write speeds have on staleness by plotting the simulated probability of bounded $(t, c{=}0)$-staleness with varying query ($T_r(I)$, $T_r(B)$) and insert ($T_w(I)$, $T_w(B)$) distributions. Under the "fast reads" configuration, an exponential distribution with $\lambda = 1$ (mean of 1 millisecond) is used for the query distributions, and an exponential distribution with $\lambda = 0.5$ (mean of 2 milliseconds) is used for the insert distributions. "Fast writes" uses $\lambda = 1$ for its write distributions and $\lambda = 0.5$ for its read distributions, and "fast reads and writes" uses $\lambda = 1$ for all read and write distributions. In all configurations, the network transmission distribution $T(M)$ is set to $\lambda = 1$. The importance relative query and insert speeds have for bounded (t, c)-staleness is clearly illustrated. When writes are as fast or faster than reads, more inserts are likely to become accessible by a query in the extended amount of time the query takes for processing at the index, leading to high probabilities of bounded (t, c)-staleness. With faster reads, the opposite is true; queries spend less time being processed at the index and thus queries arrive at buckets to aggregate insertions earlier.

6.4 Conclusion

Unlike the previous chapters which focused more on performance, scalability and availability in reinsurance portfolio decision support systems, this chapter focused on consistency in distributed Online Analytical Processing (OLAP) decision support systems, an area where

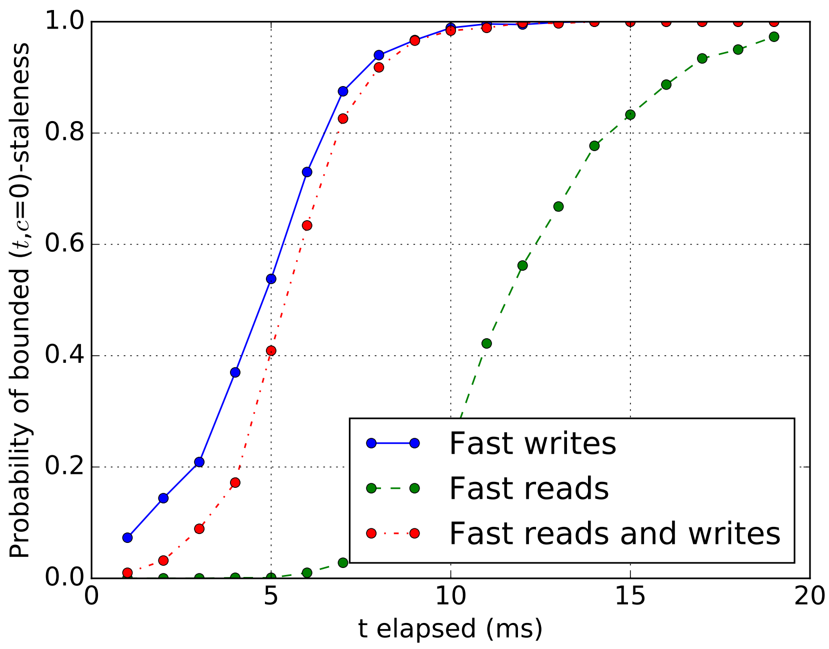

Figure 6.9: Probability of bounded $(t, c=0)$-staleness with varying read and write speeds

consistency is generally harder to measure. Here, we presented *Aggregate Probabilistically Bounded Staleness* (*A-PBS*), a measure for the staleness of *aggregate queries*. Inspired by the *Probabilistically Bounded Staleness* (PBS) measure [17] for key-value stores, A-PBS measures staleness for *aggregate* queries in distributed OLAP decision support systems that aggregate a large set of data items and depend on the write/read history of the different copies of all data items covered by the query.

Our A-PBS measure includes a formal model for describing an OLAP system's data stream and the state of consistency for individual aggregate queries. A-PBS can use either the number of missed inserts or the relative numerical error of the query result to quantify staleness. To complement A-PBS, we have also presented a generic model and corresponding Monte Carlo simulation of data aggregation in quorum-replicated distributed OLAP systems. Given a list of system parameters, our model and simulation can be used to estimate staleness for aggregate queries, thereby enabling the exploration of the trade-offs between consistency and latency in quorum-replicated distributed OLAP systems.

In a case study evaluating our A-PBS measure and Monte Carlo simulation using the *VOLAP* [29] quorum-replicated distributed OLAP system, we observed that the bounded (t, ϵ)-staleness of aggregate queries predicted through our A-PBS measure and Monte Carlo simulation was close to the actually observed staleness of aggregate queries in *VOLAP*.

Our A-PBS analysis also revealed that for *VOLAP* a partial quorum with `[N=3, W=0, R=1]` is "good enough" in practice. Even very large aggregate queries that cover the entire database and are issued only 10 milliseconds after the last insert have ≈80% probability to have zero staleness. If staleness occurs for such aggregate queries, the number of missed data items is expected to be low, as at $t = 10$ milliseconds only 0.5 inserts are missed on average, resulting in only a very small numerical error in the aggregate query result for the *sum* and *mean* aggregation functions, and very close to zero probability of any numerical error for the *max* aggregation function.

Chapter 7

Conclusion

In this thesis, we explored the design and development of interactive big data decision support systems with four key characteristics in mind: *performance*, *scalability*, *availability* and *consistency*. We did this within the context of two applications of interactive big data decision support systems: location-level reinsurance portfolio analytics and real-time OLAP.

Chapters 2, 3 and 4 focused on *performance*, *scalability* and *availability* (and, to a lesser extent, *consistency*) to develop an interactive reinsurance portfolio analytics system. Our system is capable of modelling reinsurance portfolios at a resolution that other systems (both academic and commercial) cannot. Chapters 5 and 6 focused solely on consistency. We described a method for quantifying consistency in eventually consistent OLAP systems. We applied our method to a real OLAP system to quantify the system's level of consistency under different scenarios.

7.1 Reinsurance Analytics

Location-level reinsurance portfolio analytics systems are used by reinsurance underwriters to estimate the probability distribution of the yearly net profit or loss of a given reinsurance portfolio. With this distribution, reinsurers can make well-informed decisions to adjust their portfolio to meet their desired risk profile. For example, if portfolio analysis reveals that there is a 5% chance the reinsurer's portfolio will result in losses large enough to bankrupt the reinsurer, then the reinsurer knows to take action to reduce the risk of their portfolio.

Location-level reinsurance portfolio analytics systems compute a given portfolio by processing terabytes of simulated insurance claims of individual properties through financial transformations specified in the reinsurance portfolio, which in itself is a complex network of hundreds of millions of primary insurance and reinsurance contracts. As discussed in Chapter 2, to the best of our knowledge, there is no research on reinsurance analytics at the location-level, nor are there any commercial systems today capable of modelling a reinsurance portfolio at the location level. This is due to the immense amount of data that must be processed to perform reinsurance at the location level. Instead, the existing research and commercial systems are focused on analytics at the county or region level. This drastically reduces the amount of data to process, but yields less accurate results.

In Chapter 3, we presented a highly scalable, available, consistent and performant location-level reinsurance analytics system. Our system uses a novel portfolio representation, where each contractual term, clause or treaty is described as a vertex in a directed acyclic graph. In this "portfolio graph" representation, edges represent the flow of profits or losses from one contractual term, clause or treaty to another.

We describe how our system achieves excellent scalability by distributing the work across Monte Carlo trials (of which they are typically at least 10,000 in a reinsurance portfolio analysis). Each thread independently computes the portfolio's profit or loss for its assigned trials without any communication or synchronization with other works. This not only results in high scalability, but also high performance, as costly synchronization and communication steps are avoided entirely.

In order to further maximize performance, each worker stores and processes all data in memory. This, however, combined with the fact that each Monte Carlo trial is processed on a single thread, presents a memory problem: if each compute node has t threads and limited memory, how can each worker process t trials in parallel without running out of memory? We address this by optimizing the cutwidth of the graph. For our purposes, the graph's cutwidth essentially describes (for a given topological ordering) the peak number of intermediate results that must be held in memory to compute the portfolio's loss or profit for a single trial. By minimizing the cutwidth, we minimize the memory footprint required to process each trial. We reduce the cutwidth of the portfolio graph using a 2-step process: First, we optimize the graph to reduce the maximum in and out-degree of the graph. High degree vertices are relatively common in reinsurance portfolios, and can significantly increase the cutwidth of a graph. By replacing high degree vertices with semantically equivalent d-ary trees, we can significantly reduce cutwidth without changing the semantics of the graph. Second, we exploit the typical structure of location-level reinsurance portfolios to find an ordering in which vertices should be processed that results in low cutwidths. Our low-cutwidth approach, combined with careful selection of our in-memory graph data structure, allows us to use commodity compute hardware to process trials of a reinsurance portfolio analysis entirely in-memory.

Our system also achieves high availability and consistency by storing all of our input data on a decentralized distributed data storage system (i.e., Amazon S3). This ensures that newly uploaded data is always consistent, and that all data is accessible at all times (barring any significant outages on Amazon's servers). A task queuing system ensures that if any worker fails to produce output after a specified timeout duration, the same task will be rescheduled on a healthy worker to be processed. This ensures that even if some workers experience internal failures, a full result will still be computed.

With similar hardware, our system is over 50 times faster than the the major commercial system we were able to use for comparison. When scaled up to 40 compute nodes, each with 72 cores, our system processed a full-size location-level portfolio, with 4TB of input data and a 307 million vertex graph in only 17 minutes.

In order for our location-level reinsurance analytics system to support interactive workflows, in Chapter 4 we expanded our system to cache and reuse intermediate results. We presented an algorithm that selects a small number of edges from the entire portfolio graph to serve as the set of "cache edges". During processing, each cache edge writes its intermediate output to distributed storage to be reused for later incremental analysis. This allows for later analyses to avoid reevaluating the portion of the portfolio that feeds into a cache edge unless a portfolio update invalidates the cache edge. As verified using a number of typical use cases, our cache edge selection strategy is resilient to multiple modifications to the graph, and it exposes a tuning parameter to tune the trade-off between storage cost and reducing the portion of the portfolio to be reevaluated in an incremental analysis.

Since caching significantly reduces the number of vertices that have to be processed in an incremental analysis, we also introduced a sharded graph representation. The sharded graph representation allows us to only load the sections of the graph required to perform the incremental analysis.

With caching, we observed incremental analysis running times as low as 11 seconds, approximately 90 times faster than without caching. An evaluation across many common reinsurance analytics use cases showed that for the large majority of use cases, incremental analyses only take 11 to 16 seconds to process. This allows reinsurers to quickly determine the impact adding a new contract, or modifying an existing contract's terms, has on their portfolio's bottom line. The fast response time allows reinsurers to operate the system interactively, making additional adjustments and requests to the system based on the previous responses generated by the system seconds ago.

With the addition of caching, the location-level reinsurance analytics system becomes an *interactive* big data decision support system. Our system is capable of processing several terabytes of input data across through a directed graph of hundreds of millions of vertices representing financial transformations, and can output a response in only 11 seconds.

7.1.1 Other Applications

At its core, the reinsurance risk analytics system described in this thesis is a distributed system capable of efficiently processing billions of small records, split up into several thousand Monte Carlo trials, through a directed acyclic graph containing hundreds of millions of vertices that apply transformations to the incoming records. Consequently, this system can be applied to other practical applications not related to reinsurance risk analytics, so long as the application can be reduced to a problem of processing many independent sets of records through a dependency graph of transformations.

For example, consider a large investment portfolio containing many stock options. The investor may use a Monte Carlo simulation to model possible future values for each stock covered by the portfolio. The investor could then construct a directed acyclic graph describing the contracts, fees, and commissions associated with each stock option. The reinsurance risk analytics system described in this thesis could then be applied to process the profits and losses from each stock option across each Monte Carlo trial through the network of financial transformations, quickly generating the estimated profit or loss distribution for the entire portfolio of stock options.

Other applications may include modelling the return and risk of many individual credit loans across a large credit card company, or modelling a government's annual tax revenue under different tax codes described by a directed acyclic graph.

7.2 Aggregate Probabilistically Bounded Staleness

OLAP systems allow users to insert large amounts of numerical measure data within a multidimensional, hierarchical space. Users can query multidimensional subspaces within the dataset to obtain aggregations of the measure data within the subspaces much faster than would be possible on a traditional transactional or key-value database. In the final chapters of this thesis, we explored consistency in the context of Online Analytical Processing (OLAP) interactive big data decision support systems.

Unlike the previous chapters in this thesis, in Chapter 6 we explored consistency in greater detail. We described Aggregate Probabilistically Bounded Staleness (A-PBS), a method of quantifying consistency in OLAP systems. Inspired by the *Probabilistically Bounded Staleness* (PBS) measure [17] for key-value stores, A-PBS measures staleness for *aggregate* queries in distributed OLAP systems. Quantifying consistency for aggregate queries is more challenging than quantifying consistency for simple key-value queries, as the result of an aggregate query depends on multiple data items, rather than just one. This means the consistency of an aggregate query depends on the write history of the different copies of all data items covered by the query.

We first defined a formal model for describing an OLAP system's data stream and the state of consistency for individual aggregate queries. We then defined two measures from which we can quantify the consistency or staleness of an OLAP system on a probabilistic level. The first measure uses the number of data points missed by a query to measure consistency. The second measure instead measures consistency through the relative error of the query.

To complement our model, we presented a generic model and corresponding Monte Carlo simulation of a quorum-replicated distributed OLAP system. Using the simulation and model, along with a set of system parameters, an approximate value of the A-PBS measures of staleness can be computed, allowing users to explore their system's level of consistency under different usage scenarios. We evaluated the accuracy of our model and simulation by using the simulation to approximate the A-PBS measures of a real distributed OLAP system, and comparing them against the actual A-PBS measures observed on the same system. When comparing the A-PBS measures based on relative error, the simulation generated staleness values extremely close to the staleness values actually observed on the OLAP system.

Through our exploration of the A-PBS measures in *VOLAP*, we demonstrated that even though *VOLAP* only formally guarantees eventual consistency, it has a high likelihood of queries being perfectly consistent in practice. Even very large aggregate queries that cover the entire set of data and are issued only 10 milliseconds after the last insert have $\approx$80% probability to be consistent. For queries that are stale, the impact that staleness has on the relative error of the result are extremely likely to be negligible.

7.3 Future Work

We leave the following topics for future work.

Improved scheduling of work to occurrence processors. As noted in Section 4.5.4, we observed that our fine-grained c5.2xlarge cluster outperformed our coarse-grained c5.18xlarge cluster in some experiments. Since both clusters essentially have the same compute capability, this suggests that our current approach for distributing work to occurrence processors is not optimal for coarse-grained clusters with many (e.g. 72) vCPUs per worker. Since coarse-grained clusters have a much higher amount of memory per worker, they are important in computing the memory-intensive initial portfolio analyses that have no cached data from previous analyses. With improved scheduling for coarse-grained clusters, it is likely that the performance of such clusters would come close to matching the performance of roughly equivalent fine-grained clusters.

Online adjustment of cache edges. In Section 4.4, we described how certain types of user submitted portfolio modifications require that additional edges be cached in order to ensure that incremental analyses are still processed in a timely manner. If the user frequently makes such modifications, the number of cache edges can become large. We mention in Section 4.4.1 that this can be addressed by computing a new set of cache edges during periods of low user activity, but this is not ideal for users that need to submit portfolio updates during all hours of the day. A better solution may be to develop a modified version of the cache edge selection algorithm that only considers a small subgraph localized around a single modification. Since the localized cache edge selection algorithm wouldn't need to process the entire portfolio graph, it could be run alongside each graph modification operation, ensuring that the number of cache edges always remains low.

Compilation of portfolio graphs for increased performance. Since each portfolio graph and topological ordering describe a static pipeline of data transformations, compiling this pipeline into machine code to be executed directly on an occurrence processor may significantly reduce the running time of portfolio analyses. This would eliminate the need for the occurrence processor to store an explicit representation of the portfolio graph in memory, and possibly eliminate the computational overhead associated with managing intermediate output reference counts at runtime.

User-facing interfaces for managing location-level graphs. The reinsurance analytics system presented in this thesis allows reinsurance underwriters to quickly compute the risk distribution of extremely large portfolio graphs containing hundreds of millions of vertices. However, it does not provide a user-friendly interface for querying, visualizing or modifying the portfolio. In particular, we expect that a practical visualization of the portfolio will be extremely difficult due to the number of vertices and edges in a typical location-level portfolio graph.

Application of A-PBS simulation to other data synchronization models. While the quorum consensus model is a popular method for synchronizing data in key-value data stores, it is a relatively new idea in distributed OLAP systems. Distributed OLAP systems like Analyticdb [100] and BatchDB [60] instead address the problem of data synchronization by having the "read" (i.e. query processing) nodes periodically retrieve new snapshots of the data from the "write" (i.e. insertion handling) nodes. By adjusting the A-PBS simulation to support consistency models like the ones in Analyticdb and BatchDB, it would become significantly easier to approximate the A-PBS measures of staleness for other OLAP systems that do not synchronize their data using quorum consensus algorithms.

Bibliography

[1] Transaction processing performance council, TPC-DS (decision support) benchmark. http://www.tpc.org. Accessed: 2021-04-26.

[2] Ittai Abraham and Dahlia Malkhi. Probabilistic quorums for dynamic systems. In *Distributed Computing*, volume 18, pages 113–124. Springer, 2005.

[3] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. Modeling multidimensional databases. In *Proceedings of the 13th International Conference on Data Engineering*, pages 232–243. IEEE, 1997.

[4] AIR Worldwide. AIR Worldwide: Catastrophe Modeling and Risk Assessment. https://www.air-worldwide.com/. Accessed: 2021-04-26.

[5] AIR Worldwide. Touchstone Re: Complex Reinsurance Modeling. https://www.air-worldwide.com/software-solutions/Touchstone-Re/. Accessed: 2021-04-26.

[6] Amitanand Aiyer, Lorenzo Alvisi, and Rida A Bazzi. On the availability of non-strict quorum systems. In *Distributed Computing*, pages 48–62. Springer, 2005.

[7] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[8] Ramy E Ali. Consistency analysis of replication-based probabilistic key-value stores. *arXiv preprint arXiv:2002.06098*, 2020.

[9] Amazon Web Services. Amazon EC2. https://aws.amazon.com/ec2/. Accessed: 2021-04-26.

[10] Amazon Web Services. Amazon S3 | Strong Consistency. https://aws.amazon.com/s3/consistency/. Accessed: 2021-04-26.

[11] Amazon Web Services. Cloud Object Storage. https://aws.amazon.com/s3/. Accessed: 2021-04-26.

[12] Analyze Re. Analyze Re. https://analyzere.com/. Accessed: 2021-04-26.

[13] Aon. ReMetrica. https://www.aon.com/reinsurance/analytics-(1)/remetrica.jsp. Accessed: 2021-04-26.

[14] A. Arvind and D. Culler. Tagged token dataflow architecture. Technical report, Laboratory for Computer Science MIT, Cambridge, MA., 1983.

[15] Aman Kumar Bahl, Oliver Baltzer, Andrew Rau-Chaplin, and Blesson Varghese. Parallel simulations for analysing portfolios of catastrophic event risk. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1176–1184. IEEE, 2012.

[16] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[17] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[18] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Quantifying eventual consistency with PBS. *The VLDB Journal*, 23(2):279–302, 2014.

[19] Doug Burdick, Prasad M Deshpande, TS Jayram, Raghu Ramakrishnan, and Shivakumar Vaithyanathan. OLAP over uncertain and imprecise data. *The VLDB Journal*, 16(1):123–144, 2007.

[20] Neil Burke, Oliver Baltzer, and Norbert Zeh. Efficient location-level risk analytics. In *30th Annual International Conference on Computer Science and Software Engineering*, pages 33–42, 2020.

[21] Neil Burke, Frank Dehne, Andrew Rau-Chaplin, and David Robillard. Quantifying eventual consistency for aggregate queries. In *Proceedings of the 21st International Database Engineering & Applications Symposium*, pages 274–282, 2017.

[22] Neil Burke, Andrew Rau-Chaplin, and Blesson Varghese. Computing probable maximum loss in catastrophe reinsurance portfolios on multi-core and many-core architectures. *Concurrency and Computation: Practice and Experience*, 28(3):836–847, 2016.

[23] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[24] Arbee LP Chen, Jui-Shang Chiu, and Frank SC Tseng. Evaluating aggregate operations over imprecise data. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):273–284, 1996.

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[26] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[27] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Review*, 41(6):205–220, 2007.

[28] Frank Dehne, Glenn Hickey, Andrew Rau-Chaplin, and Mark Byrne. Parallel catastrophe modelling on a cell processor. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 24–31. IBM Corp., 2009.

[29] Frank Dehne, David Edward Robillard, Andrew Rau-Chaplin, and Neil Burke. VOLAP: A scalable distributed real-time OLAP system for high-velocity data. *IEEE Transactions on Parallel and Distributed Systems*, 29(1):226–239, 2017.

[30] Jack B Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[31] Jack B Dennis and David P Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, 1974.

[32] Yuanzhen Geng, Xuanhua Shi, Cheng Pei, Hai Jin, and Wenbin Jiang. LCS: An efficient data eviction strategy for Spark. *International Journal of Parallel Programming*, 45(6):1285–1297, 2017.

[33] Einollah Jafarnejad Ghomi, Amir Masoud Rahmani, and Nooruldeen Nasih Qader. Load-balancing algorithms in cloud computing: A survey. *Journal of Network and Computer Applications*, 88:50–71, 2017.

[34] Archontia C Giannopoulou, Michał Pilipczuk, Jean-Florent Raymond, Dimitrios M Thilikos, and Marcin Wrochna. Cutwidth: Obstructions and algorithmic aspects. *Algorithmica*, 81(2):557–588, 2019.

[35] Seth Gilbert and Nancy Lynch. Perspectives on the CAP theorem. *Computer*, 45(2):30–36, 2012.

[36] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[37] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, volume 14, pages 599–613, 2014.

[38] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[39] John R. Gurd, Chris C. Kirkham, and Ian Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[40] Guy Carpenter & Company. MetaRisk. `http://www.guycarp.com/managing-risk/analytics/metarisk.html`. Accessed: 2021-04-26.

[41] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: Concepts and techniques*. Elsevier, 2006.

[42] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proc. of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 77–85. ACM, 2013.

[43] Robert A Iannucci. A dataflow/von neumann hybrid architecture. Technical report, MIT/LCS/TR-418, MIT for ACM Computing Surveys, Cambridge, 1988.

[44] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, pages 59–72. ACM, 2007.

[45] Wesley M Johnston, JR Paul Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM computing surveys (CSUR)*, 36(1):1–34, 2004.

[46] Krishna M Kavi and Ali R Hurson. Design of cache memories for dataflow architecture. *Journal of systems architecture*, 44(9-10):657–674, 1998.

[47] Krishna M Kavi, Ali R Hurson, Phenil Patadia, Elizabeth Abraham, and Ponnarasu Shanmugam. Design of cache memories for multi-threaded dataflow architecture. *ACM SIGARCH Computer Architecture News*, 23(2):253–264, 1995.

[48] Ephraim Korach and Nir Solel. Tree-width, path-width, and cutwidth. *Discrete Applied Mathematics*, 43(1):97–101, 1993.

[49] Mathias Kraus and Stefan Feuerriegel. Decision support from financial disclosures with deep neural networks and transfer learning. *Decision Support Systems*, 104:38–48, 2017.

[50] Matjaž Kukar, Petar Vračar, Domen Košir, Darko Pevec, Zoran Bosnić, et al. Agrodss: A decision support system for agriculture and farming. *Computers and Electronics in Agriculture*, 161:260–271, 2019.

[51] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[52] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. GPOP: A scalable cache-and memory-efficient framework for graph processing over parts. *ACM Transactions on Parallel Computing (TOPC)*, 7(1):1–24, 2020.

[53] Kartik Lakhotia, Shreyas Singapura, Rajgopal Kannan, and Viktor Prasanna. Recall: Reordered cache aware locality based graph processing. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 273–282. IEEE, 2017.

[54] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Review*, 44(2):35–40, 2010.

[55] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. *Proceedings of the VLDB Endowment*, 10(12):1598–1609, 2017.

[56] Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999.

[57] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. GraphLab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, pages 340–349, 2010.

[58] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 295–310, 2015.

[59] Jun Luo, Jean-Pierre Hubaux, and Patrick Th Eugster. PAN: Providing reliable storage in mobile ad hoc networks with probabilistic quorum systems. In *Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc Networking & Computing*, pages 1–12. ACM, 2003.

[60] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 37–50, 2017.

[61] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146. ACM, 2010.

[62] Dahlia Malkhi, Michael Reiter, and Rebecca Wright. Probabilistic quorum systems. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–273. ACM, 1997.

[63] Sally McClean, Bryan Scotney, and Mary Shapcott. Aggregation of imprecise and uncertain information in databases. *IEE Transactions on Knowledge and Data Engineering*, 13(6):902–912, 2001.

[64] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Cache-guided scheduling: Exploiting caches to maximize locality in graph processing. In *1st International Workshop on Architecture for Graph Processing*, 2017.

[65] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[66] Allen Newell. *Unified theories of cognition.* Harvard University Press, 1994.

[67] Rishiyur S Nikhil et al. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on computers*, 39(3):300–318, 1990.

[68] Songjie Niu and Shimin Chen. Optimizing CPU cache performance for Pregel-like graph computation. In *2015 31st IEEE International Conference on Data Engineering Workshops*, pages 149–154. IEEE, 2015.

[69] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[70] Gregory M Papadopoulos and David E Culler. Monsoon: an explicit token-store architecture. *ACM SIGARCH Computer Architecture News*, 18(2SI):82–91, 1990.

[71] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)*, 35(4):374–398, 2003.

[72] Andrew Rau-Chaplin, Blesson Varghese, Duane Wilson, Zhimin Yao, and Norbert Zeh. QuPARA: Query-driven large-scale portfolio aggregate risk analysis on MapReduce. In *IEEE International Conference on Big Data*, pages 703–709. IEEE, 2013.

[73] Reynolds Porter Chamberlain. RPC Tyche. `https://www.rpc-tyche.com/`. Accessed: 2021-04-26.

[74] Risk Management Solutions. Risk Management Solutions, Models, Software & Services. `https://www.rms.com/`. Accessed: 2021-04-26.

[75] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. FAS: A freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 754–765. VLDB Endowment, 2002.

[76] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[77] Yousef Saad. *Iterative methods for sparse linear systems.* Society for Industrial and Applied Mathematics, second edition, 2000.

[78] Oluwarotimi Williams Samuel, Grace Mojisola Asogbon, Arun Kumar Sangaiah, Peng Fang, and Guanglin Li. An integrated decision support system based on ann and fuzzy_ahp for heart failure risk prediction. *Expert Systems with Applications*, 68:163–172, 2017.

[79] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 115–126, 2005.

[80] Jung P Shim, Merrill Warkentin, James F Courtney, Daniel J Power, Ramesh Sharda, and Christer Carlsson. Past, present, and future of decision support technology. *Decision support systems*, 33(2):111–126, 2002.

[81] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 135–146, 2013.

[82] Jurij Silc, Borut Robic, and Theo Ungerer. Asynchrony in parallel computing: From dataflow to multithreading. *Parallel and Distributed Computing Practices*, 1(1):3–30, 1998.

[83] Jaspar Subhlok and Gary Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 62–71, 1996.

[84] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[85] Doug Terry. Replicated data consistency explained through baseball. *Communications of the ACM*, 56(12):82–89, 2013.

[86] Dimitrios M Thilikos, Maria Serna, and Hans L Bodlaender. Cutwidth I: A linear time fixed parameter algorithm. *Journal of Algorithms*, 56(1):1–24, 2005.

[87] TigerRisk Partners. The leading risk-to-capital advisor worldwide. `https://tigerrisk.com/`. Accessed: 2021-04-26.

[88] Ultimate Risk Solutions. Leading Provider of Dynamic Financial Analysis DFA Software. `https://www.ultirisk.com/`. Accessed: 2021-04-26.

[89] Panos Vassiliadis. Modeling multidimensional databases, cubes and cube operations. In *Proceedings of the Tenth International Conference on Scientific and Statistical Database Management*, pages 53–62. IEEE, 1998.

[90] Panos Vassiliadis and Timos Sellis. A survey of logical models for OLAP databases. *SIGMOD Record*, 28(4):64–69, 1999.

[91] Sitalakshmi Venkatraman, Kiran Fahd, Samuel Kaspi, and Ramanathan Venkatraman. SQL versus NoSQL movement with big data analytics. *International Journal of Information Technology and Computer Science*, 8:59–66, 2016.

[92] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[93] Gerd Waloszek and Ulrich Kreichgauer. User-centered evaluation of the responsiveness of applications. In *IFIP Conference on Human-Computer Interaction*, pages 239–242. Springer, 2009.

[94] Zhengyu Yang, Danlin Jia, Stratis Ioannidis, Ningfang Mi, and Bo Sheng. Intermediate data caching optimization for multi-stage and parallel big data frameworks. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 277–284. IEEE, 2018.

[95] Xin Yao and Cho-Li Wang. Probabilistic consistency guarantee in partial quorum-based data store. *IEEE Transactions on Parallel and Distributed Systems*, 31(8):1815–1827, 2020.

[96] Morteza Yazdani, Pascale Zarate, Adama Coulibaly, and Edmundas Kazimieras Zavadskas. A group decision making support system in logistics and supply chain management. *Expert systems with Applications*, 88:376–392, 2017.

[97] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled B Letaief. LERC: Coordinated cache management for data-parallel systems. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[98] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. LRC: Dependency-aware cache management for data analytics clusters. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[99] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[100] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. Analyticdb: Real-time OLAP database system at Alibaba cloud. *Proceedings of the VLDB Endowment*, 12(12):2059–2070, 2019.

[101] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302. IEEE, 2017.

[102] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

[103] Chai Zhengmeng and Jiang Haoxiang. A brief review on decision support systems and it's applications. In *2011 IEEE International Symposium on IT in Medicine and Education*, volume 2, pages 401–405. IEEE, 2011.

[104] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIXATC 15)*, pages 375–386, 2015.