DISS. ETH NO. 20272

Elastic, Reliable, and Robust Storage and Query Processing with Crescando/RB

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
PHILIPP THOMAS UNTERBRUNNER
M.Eng., Cornell University, Ithaca NY, USA
born 16 April 1982
citizen of Austria

accepted on the recommendation of
Prof. Dr. Donald Kossmann (ETH Zurich), examiner
Prof. Dr. Gustavo Alonso (ETH Zurich), co-examiner
Prof. Dr. Peter Druschel (MPI-SWS Saarbrücken), co-examiner
Prof. Dr. Martin Kersten (CWI Amsterdam), co-examiner

2012

Abstract

Recent advances in Utility Computing have given businesses of all sizes the ability to acquire computing resources in the Cloud at the tip of a button. Users of services such as Amazon EC2 or Rackspace can elastically scale their computing resources with growing or shrinking demand. Yet the elasticity of computational resources is of little use if the software itself is not elastic; that is, if the software cannot use the additional resources, or would become unavailable during reconfiguration.

The problem is particularly pronounced at the "bottom" of the software stack, the data tier. At this tier, potentially large amounts of shared application state need to be maintained in a consistent and continuously available way. To make matters worse, workloads of successful web products not only grow in terms of data size and throughput, but also in diversity as new services and functionality are added in response to customer demand. Consequently, the data tier should not only be elastic in terms of data size and processing power, but also be flexible and robust enough to adapt to continuously evolving workloads.

Traditional solutions; i.e., relational database management systems (RDBMS), do not meet these new requirements. Attesting the fact, the recent years have seen a proliferation of specialized "NoSQL" data stores. What individual specimen of this large class of systems have in common is that they radically simplify the programming model in one or more dimensions (query language, data model, consistency model) in order to achieve a higher degree of scalability, availability, and/or performance for specific workloads.

This dissertation presents such a system with a unique set of features: *Crescando/RB*. Crescando/RB implements a push-based, distributed, elastic, and fault-tolerant relational table that is robust to mixed (read-write), unpredictable, evolving workloads. It scales linearly on modern multi-core hardware, scales elastically with the number of machines (to a certain degree), guarantees strong consistency for overlapping multi-key and range queries as well as updates, tolerates permanent failure of any machine in the system, and does not require stable storage (disks). The system is not a purely academic exercise, but is used to solve a large, real-life use case in the context of the Amadeus airline reservation system.

Crescando/RB consists of three components, each of which has been designed and implemented from scratch: Crescando, Rubberband, and E-Cast.

Crescando is a push-based, relational storage engine that uses continuous, parallel, and massively shared scans rather than data indexes. This radical design makes Crescando significantly more robust to unpredictable, evolving workloads, and allows the engine to scale linearly on modern multi-core hardware.

Rubberband is a distribution framework for building elastic, reliable, shared-nothing data stores with strong consistency guarantees. Crescando/RB uses the Rubberband framework to partition and replicate the contents of independent Crescando instances.

Rubberband in turn is intimately based on E-Cast, a dynamic, uniform, causal total order multicast protocol for asynchronous networks with unreliable failure detectors. By taking advantage of E-Cast, Rubberband can guarantee sequential consistency for arbitrarily overlapping multi-key and multi-range operations, and tolerate permanent failure of any machine at any time. Unlike most systems with strong consistency guarantees, Rubberband does not rely on stable storage or perfect failure detectors, so it is easy to deploy in the Cloud. Furthermore, Rubberband is completely wait-free, so data stores built with Rubberband can achieve very high throughput and also remain fully available during reconfiguration (data repartitioning or machine replacement).

Crescando/RB combines the features and advantages of Crescando, Rubberband, and E-Cast into a single, comprehensive system. The experimental results show that Crescando/RB is highly elastic, reliable, and available, and easily meets the stringent performance and data freshness requirements of the Amadeus use case.

In describing the design and implementation of Crescando/RB, this dissertation contributes to our scientific understanding of how to make storage systems robust to unpredictable, evolving workloads, how to take full advantage of the increasing number of CPU cores on modern hardware platforms, and how to elastically and reliably scale to a large, dynamic set of machines without the loss of data consistency or availability.

Zusammenfassung

Jüngste Fortschritte im Bereich des "Utility Computing" haben Unternehmen jeder Größe die Möglichkeit gegeben, auf Knopfdruck Rechenkapazitäten in der "Cloud" anzufordern. Nutzer von Diensten wie etwa Amazon EC2 oder Rackspace können somit ihre Rechenkapazität elastisch dem wachsenden oder schrumpfenden Bedarf anpassen. Jedoch ist diese Elastitzität von Rechenkapazität von geringem Nutzen, sofern die Software selbst nicht elastisch ist, also sofern die Software die zusätzlichen Ressourcen nicht nutzen kann, oder während einer laufenden Rekonfiguration längerfristig nicht verfügbar wäre.

Das Problem ist besonders eklatant auf der untersten Schicht der Softwarearchitektur, der Datenhaltungsschicht. Auf dieser Schicht müssen potenziell große Mengen an gemeinsam genutzten Anwendungsdaten in einer konsistenten und jederzeit verfügbaren Weise verwaltet werden. Erschwerend kommt hinzu, dass die Last erfolgreicher Anwendungen mit der Zeit nicht nur in in puncto Datenvolumen und Anfragendurchsatz steigt, sondern auch in puncto Diversität der Anfragen, da Dienste und Funktionalität auf Basis von Kundenwünschen hinzugefügt werden. Daher sollte die Datenhaltungsschicht nicht nur elastisch hinsichtlich des Datenvolumens und der Rechenkapazität sein, sondern auch flexibel und robust genug sein, um sich an eine fortwährend weiterentwickelnde Anfragenlast anzupassen.

Traditionelle Lösungen, das heißt relationale Datenbankverwaltungssysteme, sind nicht in der Lage diese neuen Anforderungen abzudecken. Diese Tatsache wird untermauert durch das Aufkommen zahlreicher spezialisierter "NoSQL" Systeme im Laufe der letzten Jahre. Die diversen Exemplare dieser breiten Klasse von Systemen eint, dass sie das Programmiermodell in einer oder mehreren Dimensionen (Abfragensprache, Datenmodell, Konsistenzmodell) radikal vereinfachen, um so einen höheren Grad an Skalierbarkeit, Verfügbarkeit oder Performanz hinsichtlich bestimmter Anfragen zu erreichen.

Diese Dissertation präsentiert ein solches System mit einer einzigartigen Kombination von Eigenschaften, *Crescando/RB*. Crescando/RB implementiert eine "push"-basierte, verteilte, elastische und fehlertolerante relationelle Tabelle, die robust gegenüber gemischten (Schreib- und Lesezugriff), unvorhersehbaren und veränderlichen Anfragenlasten ist. Das System skaliert linear auf modernen Mehrkernprozessoren, skaliert elastisch mit der Anzahl der Maschinen (bis zu einem gewissen Grad), garantiert starke Konsistenz bezüglich Multischlüssel- und Bereichsaktualisierungsabfragen, toleriert bleibende Ausfälle beliebiger Maschinen im System, und benötigt keinen Permanentspeicher (Festplatten). Das System ist keine rein akademische Übung, sondern wird eingesetzt, um einen großen, realen Anwendungsfall im Kontext des Amadeus Flugreservierungssystems zu lösen.

Crescando/RB besteht aus drei Komponenten, die alle von Grund auf neu entwickelt wurden: Crescando, Rubberband und E-Cast.

Crescando ist eine "push"-basierte, relationale Datenverwaltungsengine, die kontinuierliche, parallele und massiv gemeinsam genutzte sequentielle Abfragen ("Scans") anstatt Datenindizes verwendet. Dieses radikale Design macht Crescando deutlich robuster gegenüber unvorhersehbaren, veränderlichen Anfragenlasten, und erlaubt Crescando linear auf Mehrkernprozessoren zu skalieren.

Rubberband ist ein verteiltes Softwareframework, welches die Erstellung elastischer, zuverlässiger, "shared-nothing" Datenverwaltungssysteme mit starken Konsistentgarantien ermöglicht. Crescando/RB nutzt das Rubberband Framework um Anwendungsdaten über mehrere unabhängige Instanzen von Crescando zu verteilen und zu replizieren.

Rubberband wiederum ist eng mit E-Cast verknüpft, einem dynamischen, uniformen, kausal-total geordnetem Multicastprotokoll für asynchrone Netzwerke mit unzuverlässigen Fehlerdetektoren. Durch die Verwendung von E-Cast kann Rubberband sequentielle Konsistenz für sich beliebig überlappende Multischlüssel- und Bereichsabfragen sowie Bereichsaktualisierungsabfragen garantieren, und bleibende Ausfälle beliebiger Maschinen zu jedem Zeitpunkt tolerieren. Im Gegensatz zu den meisten Systemen mit starken Konsistenzgarantieren verlässt sich Rubberband nicht auf Permanentspeicher oder perfekte Fehlerdetektoren, weshalb Rubberband sehr einfach in der "Cloud" eingesetzt werden kann. Des Weiteren ist Rubberband vollständig wartefrei. Dadurch können Datenverwaltungssysteme auf Basis von Rubberband sehr hohen Durchsatz erzielen, und sogar während Rekonfigurationen (Datenrepartitionierung oder Maschinenersatz) durchgehend verfügbar bleiben.

Crescando/RB kombiniert die Eigenschaften und Vorteile von Crescando, Rubberband und E-Cast in ein abgeschlossenes, funktionell vollständiges System. Die experimentellen Resultate zeigen, dass Crescando/RB höchst elastisch, verlässlich und verfügbar ist, und mit Leichtigkeit die strengen Performanz- und Datenaktualitätsanforderungen des Amadeus Anwendungsfalles erfüllt.

Anhand der Beschreibung des Designs und der Implementierung von Crescando/RB steuert diese Dissertation dem wissenschaftlichen Verständnis hinsichtlich der Fragen bei, wie ein

Datenverwaltungssystem robust gegenüber unvorhersehbaren, veränderlichen Anfragelasten gemacht werden kann, wie man die laufend zunehmende Anzahl an Prozessorkernen auf modernen Hardwareplattformen vollständig nutzen kann, und wie man ein System elastisch und zuverlässig, ohne den Verlust von Datenkonsistenz oder Verfügbarkeit, über eine grosse, dynamische Menge von Maschinen skaliert.

Acknowledgements

This dissertation would not have been possible without the help of those around me. I first want to thank my advisors Prof. Donald Kossmann and Prof. Gustavo Alonso for their personal engagement and unwavering belief in this work—even though I stubbornly ignored their expert advice more often than not. I feel all the more grateful and honored for their trust and support.

I am indebted to Prof. Johannes Gehrke for introducing me to the research community and teaching me much of what I know about databases. I also thank Prof. Peter Druschel and Prof. Martin Kersten for their insightful feedback and coming to Zurich in person for my examination.

Crescando/RB is not a one-man project. Besides my advisors, who guided and promoted this work on countless occasions, I am sincerely grateful to Georgios Giannikis, Simon Loesing, Janick Bernet, Ralph Steiner, and Khalid Ashmawy, who contributed their time, code, and ideas. I further thank my friends and colleagues Kyumars Sheykh Esmaili, Roozbeh Derakhshan, Ghislain Fourny, Martin Hentschel, Lukas Blunschi, Boris Glavic, Tudor-Ioan Salomie, Nihal Dindar, Tahmineh Sanamrad, Pratanu Roy, Michael Duller and anyone I may have forgotten for taking my mind off when I needed it, making my life as a graduate student a lot less miserable.

Finally, I thank my family for all their love and support. I am eternally grateful to my parents, for giving me the chance to follow my dreams even though they have lead me far away from them. And above all, I must thank my wonderful wife, Yunting, for putting up with me through difficult times, and for being such a strong and loving mother to our little daughter, Elisabeth.

Contents

Chapter 1

Introduction

Utility Computing has made a profound impact on the architecture of the web, and is transforming the IT landscape of large and small businesses alike. Until a few years ago, anyone who wanted to launch a web product (web service, web application, or website) had to buy or lease physical machines and network bandwidth. If the product was unsuccessful and generated less traffic than expected, invested capital would have been wasted. But worse, if the product was a wild success, the machines or network links could crumble under the traffic, quickly driving away potential and existing customers. In short, too much success could easily kill a great business idea [Kra10].

Today, *virtual* machines can be booked in the Cloud with just a few mouse clicks, using services such as Amazon EC2[1] or Rackspace[2]. The difference is not just convenience, but the ability to acquire additional storage capacity (disks, memory), processing capacity (CPUs), and network bandwidth almost instantaneously, at a fine degree of granularity. This allows businesses of all sizes to *elastically* size their computing resources with growing or shrinking demand, transforming fixed costs into (often lower) linear costs.

But there is a catch to this enticing prospect. The elasticity of computational resources is of little use if the software on top is not elastic; that is, if the performance-critical parts of the software stack cannot take advantage of the additional resources, or would need to be taken off-line for an unacceptable period of time. Assuming a typical three-tier architecture (presentation tier, logic tier, data tier [Fow02]), the problem is particularly pronounced at the "bottom" of the stack, the data tier. At the data tier, potentially large amounts of application state need to be maintained in a shared, consistent, reliable, and continuously available manner, making the data tier notoriously difficult to scale, let alone scale elastically.

[1]`aws.amazon.com/ec2`, retrieved Nov. 14, 2011.

[2]`www.rackspace.com`, retrieved Nov. 14, 2011.

To make matters worse, workloads of successful products not only grow in terms of data size and throughput, but also in diversity as new services and functionality are added in response to customer demand. Consequently, the data tier should not only be elastic in terms of data size and processing power, but also be flexible and *robust* enough to rapidly adapt to constantly evolving workloads [CPV11].

Traditional solutions; i.e., relational database management systems (RDBMS), have been struggling to meet these new requirements. RDBMS are complex, feature-rich systems that are not designed to scale horizontally, not designed to tolerate frequent, permanent machine failures, and not designed to automatically adapt to evolving workloads. Attesting the fact that *no size fits all*, the recent years have seen a proliferation of specialized *NoSQL data stores* [Cat10, Sto08]. Individual systems have widely differing features and use cases. But what they all have in common is that they radically simplify the programming model in one or more dimensions (query language, data model, consistency model) in order to enable a higher degree of scalability, availability, and/or performance for specific workloads.

In this dissertation, we present such a specialized system called *Crescando/RB*, which has a unique set of interesting features. Crescando/RB is a push-based, distributed, elastic, and fault-tolerant relational table implementation that is robust to mixed (read-write), unpredictable, evolving workloads. It scales linearly on modern multi-core hardware, scales elastically with the number of machines (to a certain degree), guarantees strong consistency for overlapping multi-key and (multi-)range updates, tolerates permanent failure of any machine in the system, and does not require stable storage (disks). The system is not a purely academic exercise, but is used to solve a large, real-life use case, the Amadeus airline reservation system.

Crescando/RB consists of three largely independent but well-matched components: Crescando, Rubberband, and E-Cast. Crescando is a storage engine that offers predictable performance for unpredictable (evolving) workloads. E-Cast and Rubberband form a distribution framework that allows one to build a NoSQL data store with the aforementioned features, consisting of a large number of independent instances of Crescando (shared-nothing architecture), or in fact any other storage engine.

At the time of writing, the Crescando storage engine is being rolled out for production use at Amadeus, and E-Cast and Rubberband are undergoing evaluation. Apart from its immediate practical impact, this dissertation contributes to, among other things, our scientific understanding of how to make storage systems robust to unpredictable, evolving workloads, how to take full advantage of the increasing number of CPU cores on modern hardware platforms, and how to elastically scale to a large, dynamic set of machines without loss of data consistency or availability.

1.1 Use Case

Amadeus is a world-leading service provider for managing travel-related bookings (flights, hotels, rental cars, etc.). Amadeus' core product is the Global Distribution System (GDS), an electronic marketplace that forms the backbone of the travel industry. The world's largest airline carriers and many thousand travel agencies use the Amadeus GDS to run their businesses and integrate their data.

The main database in the Amadeus GDS is called PNR, for Passenger Name Record. The fact table of this database contains dozens of millions of travel bookings, which are directly accessible through a unique key: the RLOC[3] (record locator). Besides flight and passenger information, a travel booking contains miscellaneous data such as billing information, rental car reservations, and various comment fields for travel and airline agents. In its compressed, binary form, each booking record is a few kilobytes in size. Approximately 2 years worth of data are kept on-line at any given point, which translates into a fact table of several hundred gigabytes in size. At the time of writing, this table sustains a workload of up to one thousand updates and several thousand key-value look-ups per second.

Key-value access is sufficient for most transactional workloads faced by the system. However, it is ill-suited to answer the increasing amount of real-time, decision-support (Operational Business Intelligence) queries that select on attributes other than RLOC, for example: "give the number of first class passengers in a wheelchair, who depart from Tokyo to a destination in the US tomorrow." Queries like this are increasingly common and feature stringent latency constraints, because operational decisions are made based on their results. For example, storm warnings frequently elicit large numbers of such queries, as passengers need to be rebooked according to their destinations, booking classes, and personal needs.

To support such queries, Amadeus maintains a number of materialized relational views on top of the fact table, many of which are updated in real-time through a proprietary message bus. The most important and performance-critical view is a denormalization of flight bookings: one record for every person on some airplane. We will refer to this view as *Ticket* throughout the dissertation. The functional and performance requirements of the Ticket view influenced much of the design of Crescando/RB, and we will re-encounter it throughout the performance evaluation sections of the dissertation. The Ticket schema is described in detail in Appendix A.

The Ticket view is used in a large number of data services: from generating the passenger

[3]An RLOC is a 6-character alphanumeric string. The reader may have already encountered RLOCs on his or her electronic flight itineraries or booking receipts.

list of a single flight to analyzing the customer profile of different airlines and markets (pairs of ⟨source, destination⟩ airports). Since the current Amadeus system has reached a level of complexity where adding views and indexes is no longer feasible let alone economical, a growing number of queries on Ticket do not match the primary index on ⟨flight number, departure date⟩ by the existing solution. As a result, more and more queries have to be answered in batch (off-line), using full-table scans, with very high response time and a dramatic impact on performance for the remainder of the workload. Other queries which do not match the indexes and cannot be processed off-line, are simply not possible.

As a solution to these queries, in this dissertation, we propose a single instance of the Ticket view based on Crescando/RB. In fact, at the time of writing, the Crescando storage engine that is part of Crescando/RB has already been adopted and patented by Amadeus [FMF$^+$11], and a first Ticket view based on Crescando has recently been rolled out to production. Yet Crescando is more than a cost-efficient replacement of existing technology. As will be shown in the dissertation, Crescando allows users to query for *any combination* of Boolean predicates, without regard for data indexes or clustering, while guaranteeing an answer within a few seconds. This unique property allows Amadeus to experiment with new products and product features without investing time and effort in creating and maintaining the otherwise necessary views and indexes.

Encouraged by the performance properties of Crescando, Amadeus seeks to add more and more fields to the Ticket view, and use the system for other, even more challenging use cases, such as ticket pricing or analyzing customer profiles. In these use cases, the data set far exceeds the limitations of a single machine (Crescando is a main-memory engine). In some of these use cases, Crescando is also supposed to act as a highly reliable data store for the primary copy of the data rather than as a view or cache. Consequently, a fault-tolerant (replicated) distribution layer is necessary, which elastically scales to dozens of physical machines, while preserving the consistency and performance properties of the underlying Crescando storage engine. This distribution layer is *Rubberband*, and the combination of Crescando and Rubberband is *Crescando/RB*.

1.2 Solution Overview

As a solution to all of Amadeus' use cases described previously, in this dissertation, we propose a relational data store called *Crescando/RB*. Figure 1.1 shows a high-level visualization of the architecture. Crescando/RB consists of three main components: the Crescando storage engine, Rubberband, and E-Cast. Rubberband is a framework for strongly consistent, partially replicated, elastic data stores, which intimately relies on E-Cast, a reliable, totally ordered multicast protocol.

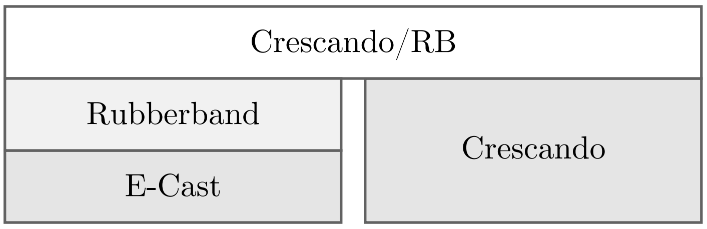


Figure 1.1: *Crescando/RB: Architecture*

1.2.1 Crescando

The Amadeus Ticket use case is a good example for a larger trend which threatens the foothold of traditional database management systems in the enterprise application domain. Database management systems are some of the most complex pieces of software in existence. Highly trained administrators and performance engineers analyze, configure, and tune these systems to match the application requirements. But requirements are not static. Systems must handle diverse, evolving workloads, while their users expect quasi-constant response time [RSQ$^+$08, CPV11, KDGA11].

At the end of the day, there is a limit to what any army of administrators can do, because every additional view or index comes with a cost in update overhead. With Crescando, we propose a rather extreme solution to the dilemma: if data indexes do not scale, then get rid of them.

In short, Crescando is what is called a *scan-only* relational table engine. Crescando may be inferior to traditional database management systems for their sweet spot, but it guarantees good and, more importantly, *predictably good* performance for *unpredictable* workloads. Crescando is based on the following design principles:

Scan-only Query Processing The data is not indexed but accessed exclusively by scans, making Crescando highly robust to query diversity (variation in the selection predicate attributes).

Shared-nothing Architecture The data is horizontally partitioned across the CPU cores of the machine, leading to linear scale-up with the number of cores.

Main-memory Storage Putting all data in main memory takes advantage of the high memory capacity of modern machines and provides high access bandwidth for scans.

Shared Scan Cursors Sharing scan cursors between many queries and updates leads to better access locality and overcomes any memory bandwidth bottleneck.

Query-data Joins Rather than indexing the data, Crescando creates short-lived indexes over the *predicates* of active queries and updates. This can be described as an index-join between the queries and updates on the inner side, and the data on the outer side. Indexing query and update predicates dramatically improves throughput compared to naïve, "nested loops"-style predicate evaluation.

Because of these design choices, Crescando is able to answer any query, regardless of its selection predicates, within a few seconds[4]. Likewise, any update, regardless of its selection predicates, is applied and made visible within a few seconds.

The concrete performance of a Crescando instance mainly depends on the data size, the number of CPU cores, the partitioning of the data and workload across cores, and the distribution of selection predicates in the workload. Comprehensive experimental results are provided in Sections 2.10 and 5.6 of this dissertation. To give a rough idea, a single Crescando instance on a 16-core machine, filled with 16 GB of Ticket data, can easily handle a mixed load of 1,000 realistic queries (as described in Section 1.1) and 1,000 concurrent updates per second, with a query turnaround time and data freshness of roughly 2 seconds. For workloads that inherently require full-table scans, Crescando offers *orders of magnitude* higher throughput than traditional database management systems.

In summary, Crescando is an ideal storage engine for the Amadeus Ticket use case. We further believe that it's robustness to unpredictable workloads make it suitable for many other applications in the area of Operational Business Intelligence, where ad-hoc, analytic queries meet stringent latency and freshness requirements.

1.2.2 E-Cast and Rubberband

Crescando is a powerful storage engine, but is by itself limited to a single physical machine. Rubberband is a distribution framework that we have developed to overcome this limitation. Rubberband is intimately based on E-Cast, a reliable, ordered multicast protocol.

Any Rubberband instantiation (Crescando/RB or otherwise) consists of a set of logical *processes*, communicating exclusively via asynchronous message-passing over an unreliable computer network. When we write "process" here, we mean a self-contained, logical entity or actor from a protocol perspective[5]. A process may consist of multiple threads or operating system processes. Conversely, multiple processes even run inside a single

[4]Obviously, there are cases where this may not be feasible, for example an unconditional "`SELECT *`" query (database dump), where simply transmitting all the query results over the network may take too much time. However, Crescando guarantees that the query processing itself is robust.

[5]Another, synonymous term found in the literature is "node".

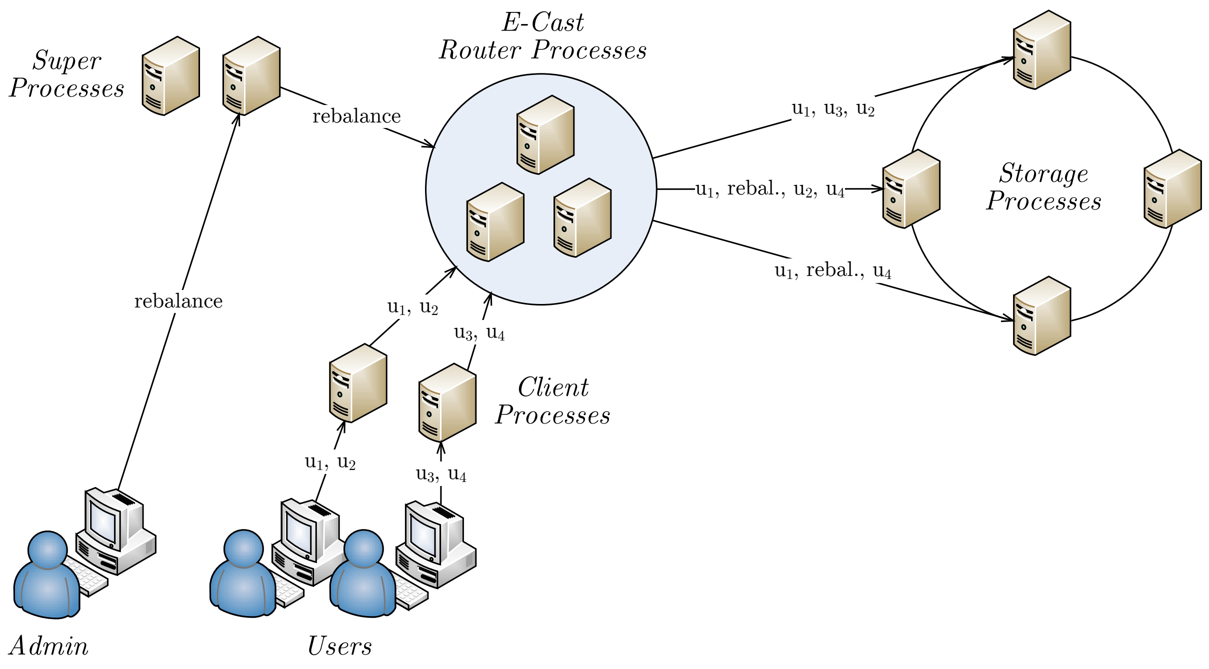


Figure 1.2: *Rubberband: Exemplary Instantiation*

operating system process. Likewise, separate processes may or may not run on separate physical machines.

Figure 1.2 shows an exemplary instantiation. Rubberband defines three types of processes: client, super, and storage processes. Client processes issue read and write requests (queries and updates), and super processes issue reconfiguration requests (explained momentarily). Each storage process runs an instance of some storage engine (Crescando, Memcached, etc.) to hold application data and serve incoming read and write requests.

To tolerate permanent failures of storage processes, Rubberband replicates each data object. Replication means that Rubberband maintains multiple copies (typically 3) of each data object on separate storage process. The data is partitioned in a way such that each storage process holds some overlapping fraction of the complete data set. This is called *partial replication*. The precise mapping of data objects to storage processes is determined by the *configuration* of the system, which may be altered at runtime through reconfiguration requests. One key feature of Rubberband—which sets it apart from all other systems with equivalent consistency guarantees that we are aware of—is that Rubberband can be reconfigured at any point, without compromising data consistency, and without losing system availability.

Client, super, and storage processes are collectively called *application processes*. In general, application processes do not communicate directly with one another[6]. Instead, requests are multicasted through a set of E-Cast *router processes.* These router processes give a number of crucial guarantees regarding message ordering and delivery. Most importantly, the router processes guarantee that any message delivered by any two application processes is delivered in the same order by the two application processes, regardless of any router or application process failures. This guarantee is called *total-order delivery* and allows Rubberband to give strong consistency guarantees for the application data. In essence, the (potentially large) set of storage processes are transparently perceived as a single, highly available, high-performance storage process by the users, even though each actual storage processes holds only a fraction (partition) of the application data, and handles only a fraction of the total workload.

Rubberband intimately relies on the functionality and guarantees of E-Cast. Nonetheless, Rubberband is just one application of E-Cast. As will become clear in Section 3, the multicast protocol is of more general use. For example, a fault-tolerant, distributed lock service based on E-Cast has been developed independently, and the protocol appears to be well-suited for a wider range of applications in systems management and coordination.

In summary, Rubberband provides the following set of features. It

- delivers updates uniformly, in causal total order,
- even arbitrarily overlapping multi-key or range updates (yields sequential consistency of the entire system, as defined by Fekete and Ramamritham [FR10]);
- is completely wait-free (clients can safely submit a high-rate stream of updates without waiting for confirmations);
- elastically scales to large numbers of storage processes;
- allows data stores built on Rubberband to be continuously available, even during reconfiguration (data repartitioning or failed process replacement);
- tolerates permanent, silent failure of any process;
- does not require stable storage (hard disks, flash);
- does not require perfect failure detectors (fail-silent model); and
- is independent of a specific data model or storage engine.

[6]There are exceptions, such as shuffling data during reconfiguration (partial live migration), but we can ignore these at the moment.

A typical instantiation of Rubberband can handle a cumulative throughput of about 20,000 updates per second (and much higher query throughput depending on the isolation level). To put this number into context with other NoSQL data stores with strong consistency guarantees, the 20,000 TPS of Rubberband are close to the *cumulative* write throughput of all applications (over 100) of Google MegaStore [BBC$^+$11]. The highest numbers reported for other systems are 2,500 TPS for Scalaris [SSR08], 2,400 TPS for ecStore [VCO10], 7,000 TPS for CloudTPS [WPC10], 4,000 TPS for Spinnaker [RST11] using regular hard-disks for logging, and up to 20,000 TPS with the log disabled (no means of recovery). This is not an apples-to-apples comparison, because some of these systems have different transaction models. That being said, most of the systems do not tolerate permanent node failures (which is critical in the Cloud), and none of the systems we are aware of is truly elastic; i.e., one cannot repartition the data without loss of availability and/or consistency.

Summing all up, Crescando/RB easily meets the requirements of the Amadeus Ticket use case, and appears suitable for a larger class of challenging use cases in the area of Operational Business Intelligence. Crescando/RB offers high throughput and predictable latency, while being extremely robust to changes in the workload. Its distribution framework, Rubberband, is elastic and continuously available, yet provides stronger consistency guarantees and higher throughput than related systems. In this dissertation, we focus mainly on the result of combining Crescando with Rubberband (i.e., Crescando/RB), but we emphasize that Rubberband is independent of a particular data model or storage engine. Attesting the fact, we have also built a prototype of a system called Memcached/RB, a combination of Memcached and Rubberband.

1.3 Contributions

In describing the design and implementation of Crescando/RB, this dissertation contributes to our scientific understanding of how to make storage systems robust to unpredictable, evolving workloads, how to take full advantage of the increasing number of CPU cores on modern hardware platforms, and how to elastically scale to a large, dynamic set of machines without the loss of data consistency or availability.

The concrete, technical contributions of this dissertation include:

Crescando A scan-only, main-memory, relational table engine which provides predictable performance for unpredictable workloads. Crescando's design and algorithms take full advantage of modern multi-core hardware. Notable algorithms and data structures include a cooperative (shared) scan algorithm called *Clock Scan*, a query-data join algorithm called *Index Union Join*, an efficient logging and recovery scheme for scan-based, in-memory data stores, as well as a variety of index structures that have been optimized for cache locality and micro-parallelism (SIMD instructions).

E-Cast A dynamic, uniform, causal total order multicast protocol for asynchronous networks with unreliable failure detectors. E-Cast is designed to implement highly elastic, large-scale, yet strongly consistent replicated systems in the Cloud. To this end, it combines state-machine replication and group communication into a stateful routing problem. In this dissertation, we provide a rigorous formalization of the routing problem, present its algorithmic solution (E-Cast), prove it correct, and describe its efficient implementation.

Rubberband A framework for building elastic relational data stores with strong consistency guarantees. By taking advantage of E-Cast, data stores built with Rubberband can efficiently support arbitrarily overlapping multi-key and (multi-)range queries and updates, and tolerate permanent failure of any process at any time. Because Rubberband does not rely on stable storage or perfect failure detectors, it is easy to deploy and maintain, even in a public cloud (e.g. Amazon EC2). Rubberband separates the reliable, ordered delivery of operations from their eventual, deterministic execution. Consequently, data stores built with Rubberband can achieve very high throughput and remain available even during elastic reconfiguration.

Crescando/RB A distributed relational data store that combines the features and advantages of Crescando, E-Cast, and Rubberband. Supporting the claims of robustness, elasticity, reliability, and performance made by this dissertation, we present comprehensive experimental results of the entire system and its individual components.

1.4 Outline

The remainder of this dissertation is organized as follows. Chapter 2 introduces the Crescando storage engine. Chapter 3 presents the E-Cast protocol and its implementation. Chapter 4 describes Rubberband and how it relates to E-Cast. Chapter 5 combines Crescando, E-Cast, and Rubberband into Crescando/RB, and Chapter 6 concludes the dissertation. The Appendix contains the schema for the Ticket table used throughput the experiments of this dissertation, as well as proofs of correctness for key theorems and algorithms.

Chapter 2

Crescando

In the last decade, the requirements faced by database management systems have changed significantly. In the fast-paced context of Service Oriented Architectures and Cloud Computing, databases must be able to handle diverse, continuously evolving workloads. As applications are constantly being extended with new functionality, and new data services are being deployed, new types of queries are being added to the workload in an unpredictable way [RSQ$^+$08, CPV11, KDGA11].

A concrete use case, the Amadeus Ticket view, has been discussed in Section 1.1 of this dissertation. Similar use cases are also found in the context of platforms operated by eBay, Amazon, or Salesforce. The latter company, Salesforce, offers Customer Relationship Management (CRM) products in a Software-as-a-Service (SaaS) model. Even though all the data and application logic are hosted by Salesforce, users are able customize their application and even define their own database tables and queries[1]. Providing such a rich service involves highly diverse, evolving workloads; yet, users of the platform will not accept performance regression.

Unfortunately, performance guarantees are difficult to make with traditional database management systems. These systems are designed to achieve *best* performance for every individual query, in isolation, where performance is defined as minimal turn-around time. To this end, they require constant monitoring, maintenance, and tuning by highly trained administrators [SMA$^+$07]. But what users really want in the use cases mentioned, are response time and throughput guarantees for a concurrent, diverse, evolving workload as a whole, without the need for an army of administrators.

As a step toward a solution to this problem, in this chapter, we present a novel relational table engine, Crescando, which offers significantly higher robustness than traditional

[1]See `http://developer.force.com`, retrieved on Nov 25, 2011.

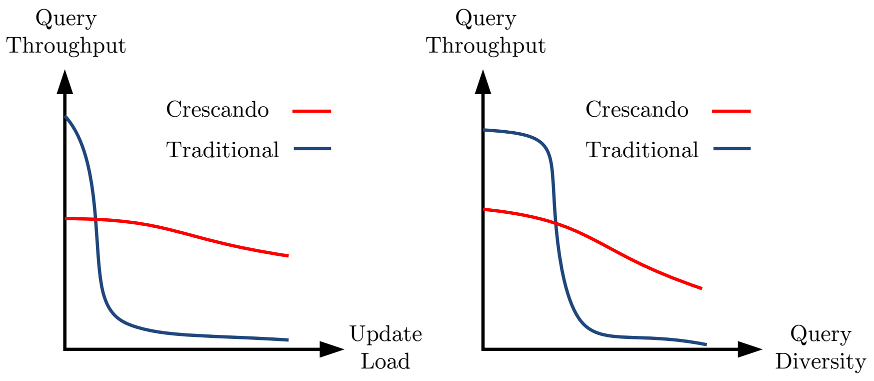


Figure 2.1: *Crescando vs. Traditional Database Management Systems*

database management systems. Consider Figure 2.1. It sketches two charts that compare the behavior of Crescando to that of a traditional database management system. (The performance evaluation at the end of the chapter includes an actual experiment and chart of this kind.)

If the update load is light, then a traditional database management system can support high query throughput by offering just the right indexes and materialized views to support the queries. Unfortunately, the query throughput decreases quickly with an increasing update load, due to lock contention and index maintenance overhead. Likewise, as shown in the second chart of Figure 2.1, the throughput drops sharply with the number of different query types, as soon as some queries require full-table scans. The two effects shown in Figure 2.1 compound, resulting in even lower throughput for workloads with high query diversity *and* concurrent updates.

Crescando is designed for exactly such workloads. Crescando tables may be inferior to traditional solutions for their sweet spot, but they exhibit good and, more importantly, *predictably* good performance for *all* workloads. Crescando achieves this by combining and extending a number of database techniques, some of which had been explored previously in the data warehousing domain [RSQ+08, QRR+08, ZHNB07]:

- Crescando implements a *push-based, scan-only* architecture (i.e., no data indexes), in order to achieve high throughput and predictable performance.
- Crescando uses main-memory storage and data partitioning to scale-up linearly on multi-core machines.
- Crescando employs massively shared scans, in order to overcome the memory-bandwidth bottleneck.

Crescando features a novel shared scan algorithm called *Clock Scan* to achieve both high query and high update throughput with predictable latency. The idea behind the Clock Scan algorithm is to batch incoming queries, and model query/update processing as a *join* between queries and update statements on the one side, and the table on the other side. For main-memory databases, index nested-loop joins are particularly effective, because random access is cheap. But rather than indexing the table, as done in traditional databases, Crescando indexes the *queries*, an idea originally proposed in the context of publish/subscribe systems [CF02, FJL^{+}01]. Crescando introduces join algorithms which support not only queries, but also updates. We refer to the latter as an *update-data join*.

In summary, this chapter makes the following contributions to the state of the art of query processing:

- a novel shared scan algorithm, Clock Scan, which introduces query-data joins and update-data joins into the context of push-based, on-line query processing;
- a new query-data join algorithm, Index Union Join, as well as a related update-data join algorithm, Index Union Update Join;
- an online multi-query optimizer which opportunistically builds predicate indexes for use in Index Union Join and Index Union Update Join;
- a number of cache-efficient index structures which are optimized for modern hardware, including a compact trie (prefix tree) that makes intelligent use of micro-parallelism (SIMD instructions) to accelerate entry lookup;
- a highly effective dictionary compression scheme for strings, including (optional) order-preserving compression and optimizations such as an associative cache implemented fully in software using SIMD instructions;
- an efficient logging and recovery scheme based on a logical redo-log and fuzzy checkpoints, which does not interfere with normal query and update processing;
- a comprehensive experimental evaluation of Crescando, including a performance comparison to a traditional, main-memory relational engine (MySQL HEAP tables); and, finally,
- the (perhaps surprising) insight that predictable performance need not be the result of careful isolation of concurrent activities, but can instead be achieved by rigorous, *deep sharing* of resources.

The remainder of this chapter is organized as follows. Section 2.1 discusses the state of the art and related work. Section 2.2 explains the data and query model. Section 2.3 gives an overview of the architecture and design of Crescando. Section 2.4 explains the Clock Scan algorithm and two baseline scan algorithms, Classic Scan and Elevator Scan. Section 2.5 formalizes and discusses the consistency model of Crescando and Clock Scan. Section 2.6 investigates the idea of query-data joins, and describes the different join algorithms supported by Clock Scan. Section 2.7 discusses multi-query optimization as it applies to Crescando. Section 2.8 describes the local logging and recovery scheme implemented in Crescando. Section 2.9 looks at string compression. Section 2.10 shows the results of an extensive experimental evaluation, and Section 2.11 makes concluding remarks.

2.1 State of the Art

Crescando is designed to provide *predictable performance for unpredictable workloads*. Since the publication of the original Crescando paper [UGAK09], the topic has gained additional momentum in research. Notably, a Dagstuhl seminar titled "Robust Query Processing" has been held in 2010 [GKK$^+$11], bringing together many experts from academia and industry.

In this section, we discuss the traditional approach toward predictable performance, robust optimization, followed by a selection of recent, related work from the data-warehousing and stream-processing domains.

2.1.1 Robust Optimization

In traditional database management systems, a query optimizer is responsible for automatically translating a declarative (SQL) query into a physical execution plan. Unpredictable performance in this context is frequently caused by inaccurate cost estimations leading to erratic decisions by the query optimizer. Ever since the seminal publication on System R [CAB$^+$81], the problem has mainly been approached by increasing the accuracy of cost predictions. These cost predictions are generally based on extensive statistics (data samples, histograms). But as pointed out by Chaudhuri [Cha98], statistics are necessarily incomplete, and for any set of statistics, one can find an execution plan whose cost is vastly underestimated.

In response to this fundamental problem, a number of papers on alleviating the effects of cost *mis*predictions have appeared over the last decade. To name a few examples, Chaudhuri et al. [CLN10] describe cost-based query optimization techniques which compute and optimize not only the expected cost, but also the cost *variance* of parametrized queries (prepared statements). Markl et al. [MRS$^+$04] propose "progressive query optimization", which is to validate cardinality estimates against actual numbers during query execution, in order to abort query processing and re-optimize if strong discrepancies are detected. A related but more intrusive idea are Eddies, mechanisms that dynamically reorder join operators during query execution, in order to improve a sub-optimal execution plan at runtime [AH00]. In contrast to progressive query optimization, Eddies avoid discarding completed work, but require suitable, modified join operators.

2.1.2 Push-based Query Processing

The techniques discussed previously can greatly improve the robustness of query optimization to parametric queries and inaccurate statistics. However, they cannot help in cases where no good query plan *exists*; i.e., in cases where the physical database design (views, indexes, clustering) is simply not suited to answer a given query. This situation arises when applications are extended with new functionality and services, and unanticipated new queries are added to the workload.

A more recent line of research aims to solve this problem by making the architecture of database engines inherently more robust to unanticipated queries. Crescando is an example of this approach. Another example is CJoin by Candea et al. [CPV09, CPV11]. CJoin is a multi-query star-join algorithm for analytic read-only workloads (data warehousing). Like Crescando, CJoin uses deep sharing of query state to increase throughput and predictability.

Crescando and CJoin fall into the larger class of *push-based* (also called *staged*) database engines. Consider Figure 2.2 for an illustration of the difference between a traditional, pull-based execution model, and a push-based execution model. Traditional, *pull-based* engines allocate operators (selection, join, projection, etc.) and resources (threads, memory, I/O) on a per-query basis. Consequently, pull-based engines are easy to modularize and extend, but suffer from bad code and data access locality, since any thread of execution constantly jumps from one operator to another [HA05].

In contrast, push-based engines allocate resources over a network of processing stages (rewrite, optimization, execution) and operators. Queries and intermediate results are streamed (pushed) through this network of stages and operators, connected by buffered queues. Each operator acts as a "mini engine" which can be shared by concurrent queries, creating the opportunity for multi-query optimizations as found in Crescando and CJoin. Because individual operators execute in tight loops and can be bound to specific CPU

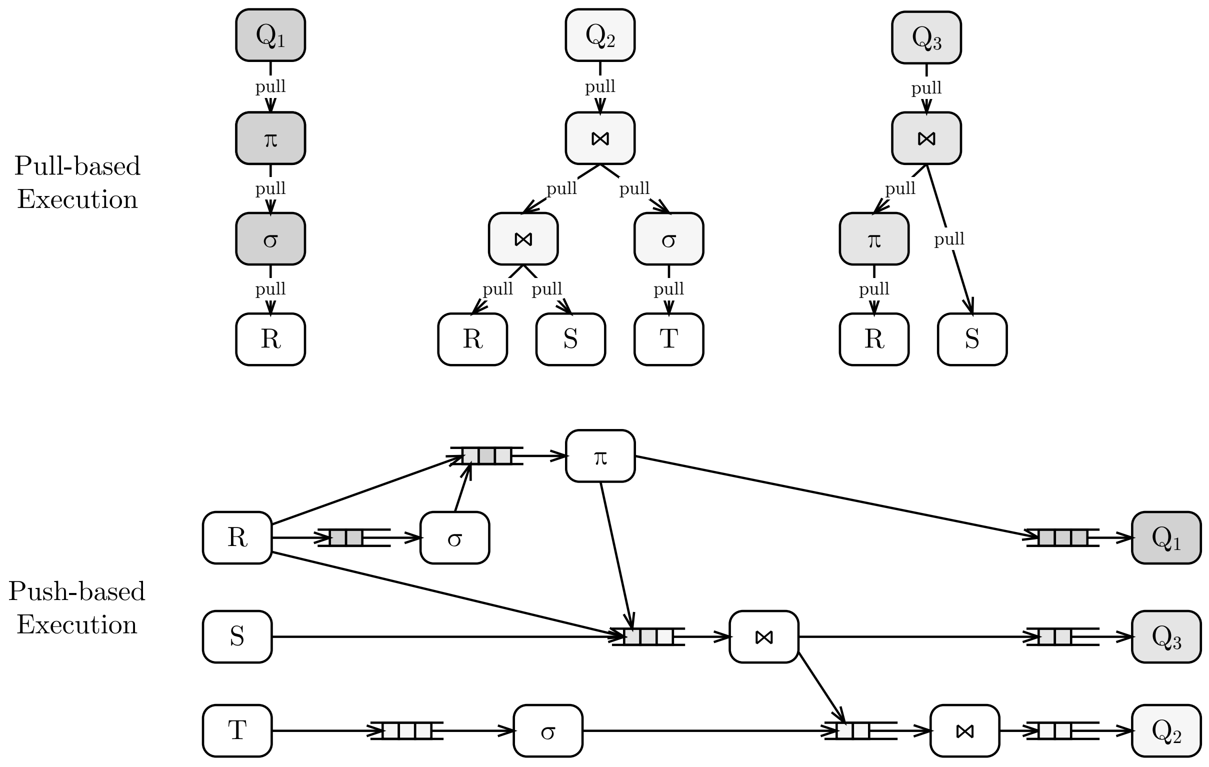


Figure 2.2: *Pull-based vs. Push-based Execution*

cores and memory regions, code and data access locality is generally much better than in pull-based engines [HA05].

Other examples of push-based engines are QPipe [HSA05] and DataPath [ADJ$^+$10]. QPipe and DataPath support full SQL processing, but do not implement the deep state sharing across multiple queries found in Crescando and CJoin. A recent paper by Giannikis [GAK12] extends Crescando and the idea of deep operator-state sharing to all the important relational operators (join, sort, group by). The resulting system is called SharedDB and combines the techniques found in Crescando, CJoin, QPipe, and DataPath into a high-throughput, high-predictability, push-based SQL engine. That being said, the dynamic multi-query optimization and resource allocation problem associated with shared, push-based query processing remains to be solved for general workloads.

2.1.3 Shared Scans

Crescando is a scan-based engine; i.e., data is accessed exclusively through scans, not indexes. To increase throughput, Crescando lets concurrent queries share scan cursors. This idea originates in the data-warehousing domain. Shared scans have been implemented in the context of disk-based database systems such as Red Brick DW [Fer94], DB2 UDB [LBM+07], and MonetDB/X100 [ZHNB07], with the goal of sharing disk bandwidth and maximizing buffer-pool utilization across queries. Driven by the advent of multi-core machines and scans' inherent parallelizability, Raman et al. [RSQ+08] and their system called blink have demonstrated that shared *main-memory* scans are a scalable access path with predictable, low ("constant") latency.

A follow-up paper by Qiao et al. [QRR+08] investigates the optimization problem of sharing a main-memory scan cursor between a subset of pending queries with `GROUP BY` clauses. In blink, `GROUP BY`s are implemented via hashing ("agg-tables"), which implies a certain working set associated with each query, turning this into a bin-packing problem. In contrast to blink, Crescando has been designed for on-line query processing, which means large numbers of non-grouping, selective queries over live, concurrently updated data. The resulting multi-query optimization problem is related to but qualitatively different from the one addressed by blink and disk-based systems. The reason is that the cost of queries in Crescando is dominated by *predicate evaluation*, not aggregation or disk access.

2.1.4 Query-Data Joins

Optimizing the evaluation of single queries by reordering and parallelizing the evaluation of individual predicates has previously been studied by Ross [Ros02, Ros04]. In Crescando, we extend the idea to sets of queries, performing *joins* between the set of active queries or updates, and the set of records in the table. The fitting term "query-data join" originates in the stream-processing literature. Chandrasekaran and Franklin [CF02] were the first to explicitly model sets of query predicates as relations and refer to the interleaved evaluation of query predicates as a join.

A simple example of a query-data join is given in Figure 2.3. Three queries (Q_1, Q_2, Q_3), each with an equality predicate on the *age* attribute of some *Person* relation are transformed into a relation *Query* with attributes *qid* (query identifier) and *age*. For each query, the latter attribute contains the constant to test for equality. The result of evaluating each query against the *Person* relation is then equivalent to the natural join between *Query* and *Person*.

In Crescando, we extend this idea to arbitrary conjunctions of different types of Boolean predicates (equality, range, string prefix). To speed up the join execution, Crescando builds indexes on the query predicates, a technique found in some form in most publish-subscribe and data-streaming systems [CF02, FJL+01, WCY04, LJ03, FSS+10]. For example, one could build a hash index on the *age* attribute of the *Query* relation in Figure 2.3. To

Q_1: `SELECT * FROM Person WHERE age = 31`
Q_2: `SELECT * FROM Person WHERE age = 20`
Q_3: `SELECT * FROM Person WHERE age = 54`

Person

name	age
Amy	24
Bob	36
Cinthia	31
Dave	54
Eve	42
Fred	31
George	59

Query

qid	age
Q1	31
Q2	20
Q3	54

Query ⋈ Person

qid	age	name
Q1	31	Cinthia
Q1	31	Fred
Q3	54	Dave

Figure 2.3: *Query-Data Join Example*

compute the result of the join, it would then be sufficient to probe the hash index once for each *Person*. For large numbers of queries, this *index join* is much more efficient than a naïve *nested-loops join* between *Query* and *Person*.

Crescando uses a special multi-query optimizer to decide at runtime which indexes to build. The baseline optimizer used in Crescando employs a simple cost metric and greedy plan enumeration algorithm inspired by publish-subscribe systems, notably Le Subscribe [FJL+01]. A Master's thesis by Steiner [Ste10] evaluates additional cost metrics and plan enumeration algorithms for Crescando.

What distinguishes Crescando from publish-subscribe and data-streaming systems is that Crescando not only indexes query predicates, but even update and delete predicates. Also, queries in Crescando are short lived (typically, less than 2 seconds) compared to *continuous* queries in publish-subscribe and data-streaming systems. This leads to very different trade-offs in query optimization, and requires novel index data structures.

2.1.5 Shared Nothing

Crescando horizontally partitions the data in order to parallelize query processing on multi-core machines. The different threads of execution communicate exclusively through message queues, and memory access is almost exclusively local to the respective CPU core. In other words, Crescando implements a *shared-nothing* architecture inside a single machine.

Type	Description
`BOOLEAN`	A ternary truth value (true, false, unknown).
`CHAR`	A single ASCII character.
`DATE`	A triple ⟨year, month, day⟩.
`TIME`	A triple ⟨hour, minute, second⟩.
`TIMESTAMP`	A pair of `DATE` and `TIME`.
`INT(M)`	A signed integer number of `M` = 8, 16, 32, or 64 bits.
`UINT(M)`	An unsigned integer number of `M` = 8, 16, 32, or 64 bits.
`FLOAT`	A single-precision floating point number.
`DOUBLE`	A double-precision floating point number.
`VARCHAR(L)`	An ASCII string of up to `L` characters.

Table 2.1: *Crescando Attribute Types*

Architecting single engine instances internally as a shared-nothing system has first been studied on specialized hardware during the era of parallel database systems, most notably in the Gamma Project [DGS$^+$90, DG92] and in Prisma/DB [ABF$^+$92]. In retrospect, this first wave of systems failed to gain acceptance due to the cost of specialized hardware, and the high pace of innovation in (much cheaper) mainstream hardware at that time. The end of single-thread execution speed improvements around the year 2004 [Sut05], and the related advent of parallel off-the-shelf hardware however has lead to a revival of shared-nothing architectures.

Examples of this new wave of systems are H-Store [SMA$^+$07] and DORA [PJHA10]. Since the number of CPU cores and thereby the cost of cache coherence will continue to increase for the foreseeable future [Sut05], one may expect shared-nothing to become the dominant architecture for database engines.

2.2 Data and Query Model

In this section, we give a description of the data and query model of Crescando. We omit a lengthy formalization because Crescando implements just a subset of the widely known relational model [Cod83] in the form embodied by the SQL standard [Int09].

Crescando is a relational table engine; that is, an instance of Crescando contains exactly one table as found in a typical relational database management system. A table is an (unordered) set of records, each of which is a sequence of strongly typed, named attributes, in accordance with some user-defined schema. An example of such a schema for the Amadeus Ticket table (cf. Section 1.1) is given in Appendix A. The type names, semantics, and value ranges in Crescando conform to SQL data types (including `NULL` semantics). A complete list of the attribute types supported by Crescando is given in Table 2.1.

To manipulate and retrieve the records stored in a Crescando table, clients issue *operations*. The four types of operations that Crescando offers are based on the classic CRUD (create, read, update, delete) paradigm popularized by Martin [Mar83]. Alluding to the corresponding SQL expressions, the Crescando operations are called *select*, *insert*, *update*, and *delete*.

In the context of concurrency control, we sometimes refer to select operations as *reads*, and to insert, update, and delete operations collectively as *writes*. In the database literature, reads and writes are also called queries and updates respectively. The terms *query-data join* and *multi-query optimization* used in this dissertation follow the established terminology from the literature.

The following is a semi-formal list of the various operation structures and semantics:

- An insert operation is simply a set of records, which are to be added to the table.
- A select operation s is a pair $s = \langle \sigma, \pi \rangle$, where σ is a set of Boolean predicates, and π is a sequence of attribute names. A select operation $\langle \sigma, \pi \rangle$ retrieves every record in the table, which satisfies every single predicate in σ, projecting out exactly the attributes in π. For example, the select $\langle \{\text{classOfService} = \text{'Y'}, \text{ age} < 18\}, \langle \text{name}, \text{age} \rangle \rangle$ would retrieve the name and age of every under-age person in service class 'Y' (Economy).
- An update operation u is a pair $u = \langle \sigma, \upsilon \rangle$, where σ is a set of Boolean predicates, and υ is a set of pairs $\langle a, v \rangle$, where a is an attribute name, and v is a value of the corresponding type as defined by the schema. An update operation $\langle \sigma, \upsilon \rangle$ affects the set of records which satisfy every single predicate in σ, setting the attributes specified in υ to the corresponding values. For example, the update $\langle \{\text{rloc} = \text{'ABC123'}\}, \{\langle \text{canceled}, \text{true} \rangle\} \rangle$ would cancel all flight tickets associated with the *rloc* "ABC123".
- A delete operation d is just a set of Boolean predicates, which deletes the records matching every single predicate in d from the table. For example, the delete $\{\text{dateOut} < \text{'01-01-2010'}, \text{canceled} = \text{true}\}$ would remove all canceled flight tickets with a departure date older than January 1st, 2010, from the table.

The Boolean predicates used in select, update, and delete operations can be unary or binary. The supported unary predicates are `IS NULL`, `IS NOT NULL`, `ALWAYS TRUE`, and `ALWAYS FALSE`. The semantics of these predicates should be self-explanatory. The `ALWAYS TRUE` and `ALWAYS FALSE` operators are sometimes useful for debugging and query rewrite.

Every binary predicate is a pair $\langle o, c \rangle$, where o is an operator, and c is a constant value of appropriate type. The supported operators are $<$, $\leq$, $=$, $\geq$, $>$, $\neq$, & (binary `AND`), and $\sqsubset$ (begins-with / string prefix). Except for the obvious cases, all data types support all operators[2]. In addition, Crescando allows user-defined predicate functions (UDFs). An important use case for UDFs is partial live migration, described in Section 5.4 of the dissertation.

Optionally, select operations can perform scalar aggregation (aggregation without `GROUP BY`). Formally, an aggregating select s is a pair $s = \langle \sigma, \pi^* \rangle$, where σ is a set of Boolean predicates, and π^* is a sequence of pairs $\langle a, g \rangle$, where a is an attribute name, and g is an aggregation operator. The available aggregation operators are `COUNT`, `MIN`, `MAX`, `SUM`, and `AVG`. The semantics of these operators are analogous to their SQL counterparts.

2.3 Architecture and Design

In a traditional, *pull-based* storage engine architecture, clients issue a single operation (select, insert, update, delete) and wait for the result. The popular ODBC/JDBC API family enforces exactly this pattern. In contrast, the Crescando storage engine follows a strictly *push-based* design (cf. Section 2.1.2), where operations are pushed in (enqueued) on one side, and result tuples are pushed out (dequeued) on the other side. Clients of Crescando are able to (in fact, *encouraged* to) enqueue a large number of concurrent operations. This allows the Crescando engine to pipeline and batch the processing of operations in order to achieve high parallelism, throughput, and predictability.

Figure 2.4 visualizes the push-based, pipelined, parallel architecture of Crescando. The engine horizontally partitions[3] the data across a user-defined number of scan threads. In our implementation, each scan thread is a kernel thread with hard processor affinity, which *continuously* scans a dedicated partition of the data, stored in a local slice of memory called a *segment*. This shared-nothing architecture enables linear scale-up because of the following key properties.

[2]For instance, `VARCHAR` attributes do not support &, whereas they are the only ones to support $\sqsubset$.

[3]Crescando supports round-robin and hash partitioning on single attributes of arbitrary type (integer, string, date, etc.), but any other partitioning scheme could be used in principle.

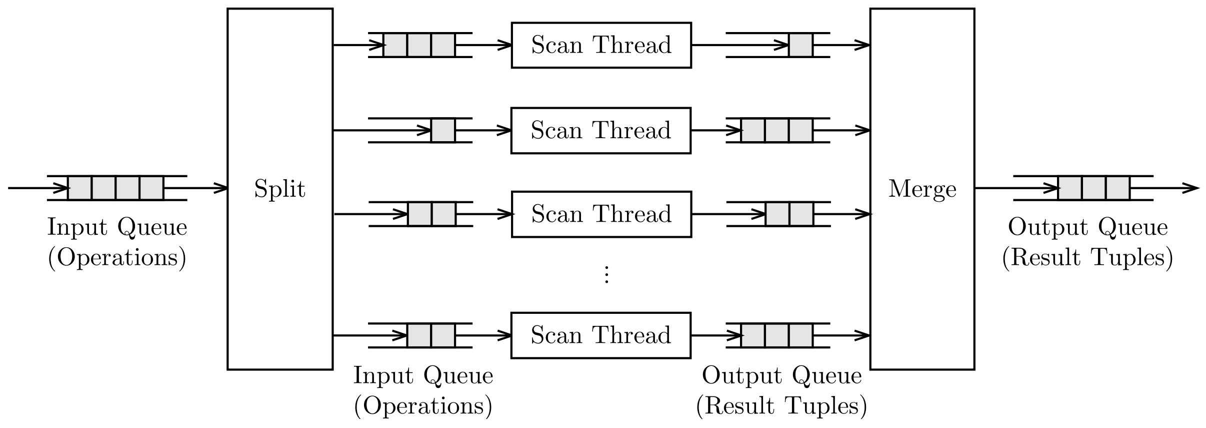


Figure 2.4: *Crescando Architecture*

No Locking Because a scan thread is guaranteed to be the only thread accessing records in its segment, execution can proceed without any locks or latches.

No Cache Contention Distinct processor caches never contain a copy of the same record, so they are implicitly coherent in this performance-critical respect. Records need not be written back to main memory until the scan cursor moves on, even if they are modified.

Minimum Access Distance Binding threads to CPU cores with hard affinity allows Crescando to allocate everything that is local to a thread—call stack, various buffers, predicate index structures (Section 2.6.3), string compression dictionaries (Section 2.9) and so forth—in memory that is physically close to the thread. This is critical on NUMA (non-uniform memory access) architectures such as AMD Opteron or Intel Nehalem. CPUs never access remote memory except for passing operations and operation results.

Crescando is a row-store; i.e., each segment stores records consecutively, as a whole. To be precise, a segment is divided into chunks, which are the equivalent of pages in a traditional, disk-based database. Each chunk consists of a fixed number of slots of fixed size, each of which may or may not contain a record. While slots are fixed size, Crescando does support dictionary compression of variable-sized string attributes. This important feature is described in Section 2.9.

The data layout in Crescando is an instance of the so-called N-ary Storage Model (NSM). We are aware of alternative data layouts which may improve cache locality and performance for certain workloads, namely the Domain Storage Model (DSM, also called column-storage) and Partition Attributes Across (PAX), a hybrid of NSM and DSM [ADHS01]. A

Crescando-related paper by another author investigating the effects of alternative storage layouts is under submission at the time of writing.

All operations (insert, select, update, delete) are processed in three pipelined stages:

Split First, every incoming operation is *split* and put into the input queue of one or more scan threads, in accordance with the chosen data partitioning scheme. Consider a Crescando instance that stores Amadeus Ticket records (cf. Section 1.1), hash partitioned by *rloc*. A select operation such as $\langle\{\text{rloc} = \text{'ABC123'}\}, *\rangle$ is enqueued at exactly one scan thread (namely, the thread where 'ABC123' hashes to), whereas $\langle\{\text{firstName} = \text{'Doe'}\}, *\rangle$ is enqueued at all scan threads. (For convenience, we write "$*$" here as a placeholder for the sequence of all attribute names.) Update and delete operations are split analogously, but an insert operation may be broken-up into multiple insert operations, in case the original operation contained multiple records that hash to distinct segments (scan threads).

Scan Scan threads periodically flush their input queue, thereby *activating* the flushed operations. When and how often a scan thread flushes its input queue depends on the concrete algorithm, and we discuss three such algorithms in Section 2.4. Using the Clock Scan algorithm for instance, a scan thread flushes its input queue at the beginning of every full cycle, and then executes the flushed, active operations in batch. As the scan cursor passes over the data, the stream of scanned records is matched against the batch of active operations, generating an interleaved stream of result tuples. The scan thread promptly puts these result tuples into its local output queue. Once any active operation has completed a full scan, the scan thread puts a corresponding end-of-stream tuple in its output queue and *deactivates* (forgets) the operation.

Merge The local output queues of individual scan threads are periodically flushed and *merged* into a common output queue that is exposed to the client application. Result tuples that belong to regular (non-aggregating) select operations are passed through as-is. Result tuples that belong to aggregating select operations, as well as insert, update, and delete result tuples are buffered until all scan threads have completed the respective operation. At this point the buffered result tuples are merged into a single result tuple for the client. For regular select operations, the end-of-result tuples generated by scan threads allow the merge step to detect when an operation has been deactivated (i.e., completed) by every scan thread. At this point a single end-of-result tuple is put into the common output queue. No result tuple is held back longer than necessary, so client applications see a *single, interleaved stream* of result tuples for all concurrent operations.

The split and merge stages are executed by the client thread(s) calling the enqueue and dequeue functions of the Crescando API. In contrast, the scan stage is executed in parallel, in dedicated threads. Notice that threads communicate exclusively through (highly optimized) queues, and all the heavy lifting (scans, predicate evaluation) is embarrassingly parallel, making this architecture highly scalable. In Section 2.10 we show that a single Crescando instance scales linearly to a large number of CPU cores.

2.3.1 Discussion

At this point, the reader may have concerns with regard to fairness ("cheap" versus "expensive" operations) and resource utilization (busy versus idle threads). After all, a select operation with a single, selective equality predicate on the key attribute may share a scan cursor with a select operation with a large compound predicate on non-key attributes.

For one thing, the fact that Crescando treats every operation the same way can be considered a strong type of fairness. For another thing, Crescando takes advantage of the Law of Large Numbers. The higher the load on the system; i.e., the more operations share a given scan cursor, the more these operations become representative of the workload as a whole, making the throughput and latency of the system highly stable and predictable. The more operations share a given scan cursor, the higher are also the performance benefits of query-data joins. Indeed, the query-data join algorithms used in Crescando render the duration of a scan cycle *logarithmic* in the number of operations sharing the scan cursor (cf. Section 2.10). This significantly lessens any load imbalance between scan threads, and also makes the system highly robust to load spikes.

2.4 Scan Algorithms

Crescando is a scan-only engine. As explained in the previous section, the data and workload are partitioned across a static set of scan threads, which continuously scan the data in their respective, dedicated memory segments. In this section, we introduce a novel, high-throughput scan algorithm called Clock Scan, and compare it to two state-of-the-art algorithms, Classic Scan and Elevator Scan.

2.4.1 Classic Scan

In a straight-forward implementation of the scan stage, each scan thread processes one operation at a time, in an infinite loop. We refer to this naïve algorithm as *Classic Scan.*

Algorithm 1: Classic Scan

```
Data: Segment seg;                                    /* local memory segment */
Data: OpQueue inQueue;                                 /* local input queue */
Data: ResultQueue outQueue;                           /* local output queue */
while true do
    /* activate a single operation in the input queue                       */
    Op op ← inQueue.Get()
    /* execute the active operation against every slot in the segment       */
    foreach Slot $s \in seg$ do
        op.Execute(s, outQueue)
    /* deactivate the operation                                             */
    outQueue.Put(op.EndOfStream())
```

Consider the pseudo-code in Algorithm 1. At the beginning of every scan cycle (while-loop iteration), the operation at the head of the local input queue is dequeued (activated). The data segment is then scanned, one record slot at a time, and the active operation is executed against each slot. The `Execute` function first checks whether the current slot is occupied. If it is not, and the operation is an insert, a record is inserted into the slot ("first fit" policy). If the slot is occupied, and the operation is not an insert, the operation's selection predicates are evaluated. If all predicates are satisfied, the `Execute` function either puts a result tuple in the scan thread's local output queue (select operation), updates the operation's aggregation state (aggregating select operation), modifies the record in the slot (update operation), or marks the slot as free (delete operation). After scanning all records, the scan thread puts an operation-specific end-of-stream result tuple on its local output queue and dequeues (activates) the next operation from the local input queue.

The asymptotic runtime of Classic Scan is obviously $O(n * m)$ for n operations over m slots. What is more interesting is that fact that Classic Scan can be expected to be very inefficient on modern processors, as it makes essentially no use of the cache.

2.4.2 Elevator Scan

A first, significant improvement over Classic Scan is Elevator Scan, given in Algorithm 2. The idea is to have multiple operations *share* a scan cursor. This improves cache locality and allows Elevator Scan to overcome the infamous *memory wall* [WM95, MBK00]. Zukowski et. al. [ZHNB07] and Raman et. al. [RSQ$^+$08] have previously investigated variants of this algorithm for read-only workloads in disk-based and main-memory databases, respectively. In Crescando, we adapt the algorithm for mixed workloads.

Algorithm 2: Elevator Scan

```
Data: Segment $seg$;                                   /* local memory segment */
Data: OpQueue $inQueue$;                                  /* local input queue */
Data: OpQueue $activeQueue$;                             /* local active queue */
Data: ResultQueue $outQueue$;                            /* local output queue */
while true do
    foreach Slot $s \in seg$ do
        /* deactivate all operations that have finished a full scan          */
        while activeQueue.Peek().Finished() do
            outQueue.Put(activeQueue.Get().EndOfStream())
        /* activate all operations in the input queue                        */
        while $\neg$inQueue.IsEmpty() do
            activeQueue.Put(inQueue.Get())
        /* execute all active operations against the slot                    */
        foreach Op $op \in activeQueue$ do
            op.Execute(s, outQueue)
```

Crescando's Elevator Scan maintains a queue of active operations. For each record slot that is scanned, the active queue is iterated over, and the active operations are (implicitly) executed in activation order. This guarantees the same degree of consistency as Classic Scan, even if some operations are writes, see Section 2.5.

At every slot, Elevator Scan activates all pending operations in the input queue, and deactivates all operations that have completed a full scan cycle. Operations can thus hop on the scan at any point, in analogy to a physical elevator. For efficiency reasons, our concrete implementation activates and deactivates operations at a user-defined interval (some fraction of a full cycle), not after every slot.

While Elevator Scan promises significantly better cache locality than Classic Scan, the asymptotic runtime of Elevator Scan is still $O(n * m)$ for n operations over m slots, the same as Classic Scan. Improving the asymptotic runtime requires more drastic changes.

2.4.3 Clock Scan

Even though Elevator Scan greatly improves over the cache locality of Classic Scan, this improvement results at most in some constant factor in runtime. In contrast, Clock Scan performs query/update-data joins over sets of reads and writes to enable *asymptotic* runtime improvements. In this section, we focus on the scan itself. Query/update-data joins are covered in detail in the subsequent section.

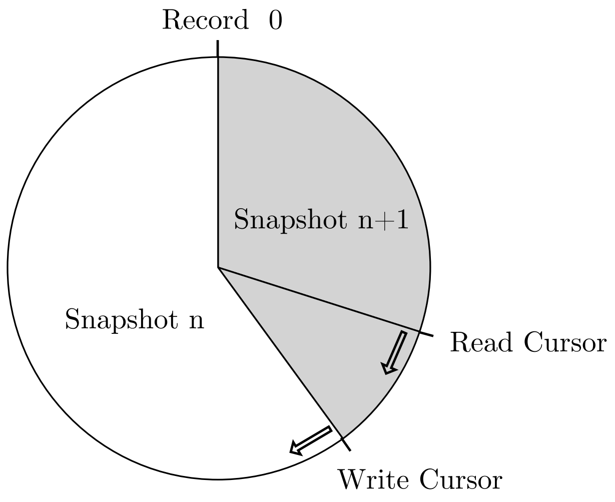


Figure 2.5: *Clock Scan Illustration*

Algorithm 3: Clock Scan

```
Data: Optimizer $opt$;            /* optimizer for query-data and update-data join plans */
Data: Segment $seg$;                                          /* local memory segment */
Data: OpQueue $inReadQueue$, $inWriteQueue$;   /* local input read and write queues */
Data: ResultQueue $outQueue$;                                    /* local output queue */
while $true$ do
    /* activate all writes and strict reads in the input write queue                */
    OpQueue $activeWrites \leftarrow \emptyset$
    while $\neg$ $inWriteQueue$.IsEmpty() do
        activeWrites.Put($inWriteQueue$.Get())
    /* activate all basic reads in the input read queue                             */
    OpSet $activeReads \leftarrow \emptyset$
    while $\neg$ $inReadQueue$.IsEmpty() do
        activeReads.Put($inReadQueue$.Get())
    /* create query and update join plans                                           */
    UpdateJoinPlan $updateJoinPlan \leftarrow$ opt.PlanUpdates($activeWrites$)
    QueryJoinPlan $queryJoinPlan \leftarrow$ opt.PlanQueries($activeReads$)
    /* scan the full segment, chunk-wise                                            */
    foreach Chunk $c \in seg$ do
        updateJoinPlan.Join($c$, $outQueue$);          /* execute update-data join */
        queryJoinPlan.Join($c$, $outQueue$);            /* execute query-data join */
    /* deactivate all active operations                                             */
    foreach Op $op \in (activeReads \cup activeWrites)$ do
        outQueue.Put($op$.EndOfStream())
```

Figure 2.5 shows a high-level illustration of the algorithm idea. Suppose we continuously run two circular scans over the segment: one read scan, one write scan. Let us enforce that the read cursor cannot pass the write cursor and vice versa, i.e., the read cursor is always some delta less than one cycle behind the write cursor. The write cursor executes updates strictly in activation order (as in Elevator Scan). Assume that operations are only activated at the beginning of every full scan cycle, at record 0. It should be easy to see that the read cursor will always see a consistent snapshot, regardless of the *order* in which queries are executed. (A proof is provided in Appendix B.1.) This freedom to reorder and interleave execution allows efficient query-data joins.

At the beginning of each scan cycle, Clock Scan, given in Algorithm 3, flushes the input queues and invokes a special multi-query optimizer to create *join plans* for the active reads and writes. This step is described in detail in Section 2.7. Then, Clock Scan performs the actual, chunk-wise scan of the segment. A chunk is a small partition of a segment analogous to a page in a disk-based database.

Clock Scan merges the two logical cursors into a single physical cursor. This increases cache locality (each record is loaded only once per scan). For each chunk, first the update-data join plan, and then the query-data join plan are executed. One could reverse the order (first query-data join, then update-data join), but this would mean that the effects of any write would be seen by basic reads only in the next scan cycle.

The runtime complexity of Clock Scan is determined by the optimizer and join algorithms. By dividing the memory segment into chunks, Clock Scan essentially performs *blocked* joins, where blocks are intended to fit into the cache. In fact, chunks and the join algorithms described in Section 2.6 are heavily optimized for space, so *thousands* of queries and updates may share a scan cursor without problems.

2.5 Consistency

Crescando does *not* implement classic database transactions with ACID guarantees, because classic transactions do not fit the use case (cf. Section 1.1), and do not fit the push-based processing model. Instead, Crescando gives ordering guarantees regarding the execution of operations in the input stream. We now present a brief formalization of the system model and the consistency guarantees that Crescando provides.

2.5.1 System Model

Let the sequence $S = \langle O, < \rangle$ be the stream of operations created by the client application, where O is the set of operations, and $<$ is the strict total order in which these operations are enqueued (pushed, streamed) into Crescando.

Each operation $o \in O$ is either a read (select) or a write (insert, update, delete). We distinguish *strict* and *basic* reads. The consistency guarantees and execution costs of strict and basic reads differ, as described below.

Crescando stores a table, which is simply a set of records X. For every record $x \in X$ there exists a sequence of operations $S_x = \langle O_x, <_x \rangle$ where $O_x \subseteq O$ is the set of operations that access x, and $<_x$ is the strict total order in which these operations access x. (Note that the model makes no restrictions on $<_x$ with respect to $<$; in particular, it does not define $<_x$ as a sub-relation of $<$.) Let $W_x \subseteq O_x$ be the set of write operations that (over-)write x. The execution of these writes creates a sequence of versions of x.

2.5.2 Guarantees

We separate the consistency guarantees of Crescando into five distinct properties. The properties are defined with respect to the versions of records that are read and written by the operations in the input sequence S. The first four properties are named after analogous types of session consistency in distributed systems literature, as originally defined by Terry et al. [TDP+94]. The fifth property, Consistent Snapshot, is also standard vocabulary from distributed systems literature.

One more note on terminology: whenever we write "preceding" or "successive" as regards operations, we are referring to the input order $<$. Whenever we write "earlier" or "later" as regards versions of some record x, we mean the execution order $<_x$ of writes of x.

Monotonic Reads. *If a* strict *read operation reads a record x, any successive read operation (basic or strict) will read the same or a later version of x. If a* basic *read operation reads a record x, any successive* basic *read operation will read the same or a later version of x.*

Note that when a *basic* read operation reads a record x, it may however read a *later* version of x than some *successive*, *strict* read operation on x.

Monotonic Writes. *A write operation on a record x is executed before any successive write operation on x. In other words, for any pair of writes w, w' on some record x where $w <_x w'$, it holds that $w < w'$.*

Read Your Writes. *The effect of a write operation on a record x will be seen by any successive read operation on x. In other words, for any pair of write w and read r of some record x where $w < r$, it holds that $w <_x r$.*

Write Follows Strict Read. *A write operation on some record x takes place on the same or later version of x than read by any preceding,* strict *read operation. In other words, for any pair of write w and* strict *read r of some record x where $r < w$, it holds that $r <_x w$.*

Note that when a *basic* read operation reads a record x, it may however read a version written by a *successive write* operation on x.

Consistent Snapshot. *A* strict *read operation always sees a* consistent snapshot *of the* entire table*; i.e., the set of records and their respective versions exactly between some pair of writes that immediately succeed each other in the input sequence, as if the writes had been executed strictly in sequence. A* basic *read operation always sees* consistent snapshots *of* individual segments*.*

Proofs of the different guarantees are provided in Appendix B.1.

2.5.3 Discussion

In summary, Crescando guarantees that writes and strict reads are executed exactly in input order. Basic reads are executed in input order with respect to each other, but may read a later version than read or written by some preceding, strict read or write.

The reason for this somewhat curious distinction between basic and strict reads is found in Clock Scan. Clock Scan treats strict reads like writes; that is, basic reads become part of a query-data join plan, while strict reads and writes become part of an update-data join plan. As described in Section 2.4.3, for any record under the scan cursor, Clock Scan first executes the update-data join plan before executing the query-data join plan. Therefore, the result of a basic read r may differ from the result of an equivalent strict read r' if and only if r shares a scan cursor with some successive write w. In this case, r would be executed after w, while r' would be executed before w.

Consider Figure 2.6 for an illustrative example. In the example, there are three operations: a basic read r, a strict read r', and a write w, arriving in sequence $\langle r, r', w\rangle$ in the input queue of some Clock Scan thread, as visualized by the horizontal time axis. The vertical, dashed lines represent the instants at which Clock Scan completes a full cycle and flushes the input queue. Depending on when this happens, different sets of operations share the scan cursors, leading to different execution orders. While there are four possible activation orders as shown, there are only two interesting execution orders (reads commute): $\langle r', r, w\rangle$ and $\langle r', w, r\rangle$. Only the latter deviates from the input order in an observable way, and it happens exactly in the case where the basic read r shares a scan cursor with the successive write w.

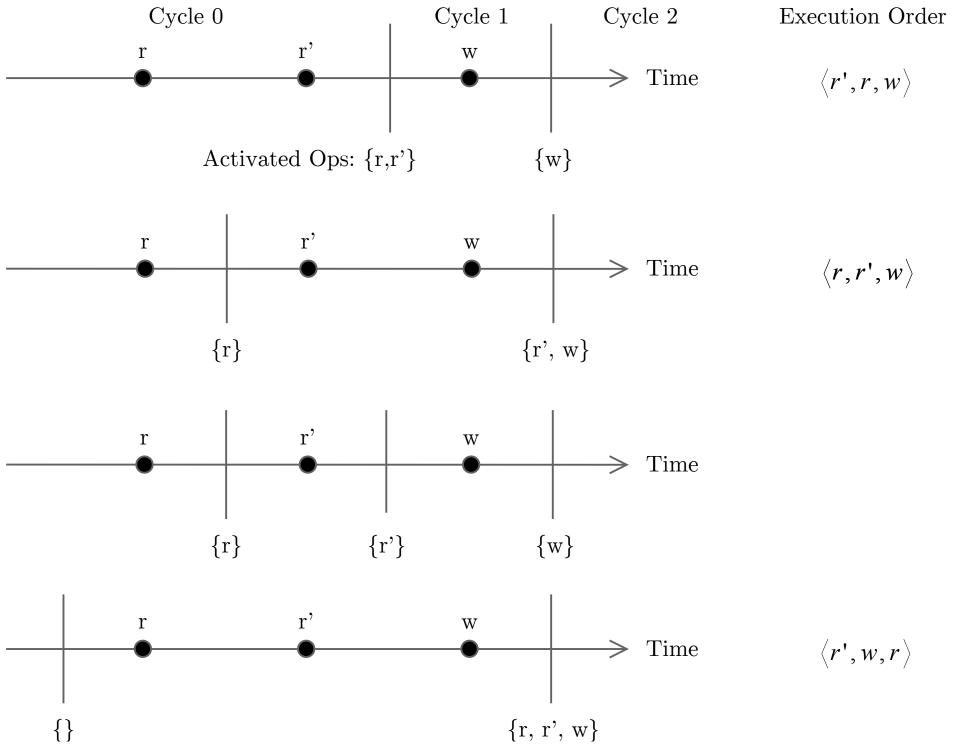


Figure 2.6: *Activation and Execution Order in Clock Scan*

The fact that strict reads see a consistent snapshot of the entire table is a corollary of Monotonic Writes, Read Your Writes, and Write Follows Strict Read (see Appendix B.1). Even for basic reads, any given instance of Clock Scan guarantees that the read sees a consistent snapshot of the scanned segment, as illustrated in Figure 2.5 on page 27. However, the scan cursors of parallel instances of Clock Scan are not synchronized. Hence, basic reads that hit multiple scan threads may see different (but individually consistent) snapshots of the respective segments.

As seen in Section 2.6, update-data joins are more expensive than query-data joins, thus strict reads are more expensive than basic reads in Clock Scan. Consequently, for every individual read, users of Clock Scan are given a choice between maximum consistency or maximum performance. In contrast, Classic Scan and Elevator Scan never reorder operations, so all reads are strict. That being said, even strict reads in Clock Scan are much cheaper than reads in Classic Scan and Elevator Scan, so there is no performance versus consistency trade-off between different scan algorithms.

2.6 Query-Data Joins

The Clock Scan algorithm improves upon Classic Scan and Elevator Scan by interleaving the execution of concurrent queries into so-called query-data joins, a term that originates in the data stream-processing literature [CF02]. We previously discussed the basic idea in Section 2.1.4, where we gave a simple example using queries with just one selection predicate. In this section, we generalize the model to arbitrary conjunctions of Boolean predicates, followed by a detailed description of the different join algorithms supported by Clock Scan.

2.6.1 Model

In Section 2.1.4, we showed that answering a set of queries, each with exactly one equality predicate $a = c$ on some common attribute a, can be modeled as a natural join between the data relation and a query relation with the schema $\langle \text{qid}, a \rangle$, where a just contains the value c for each respective query.

Figure 2.7 illustrates how to extend this model to arbitrary conjunctions of equality predicates. For every attribute of the data relation, a corresponding attribute exists in the query relation. For any given query, when a query has an equality predicate $\langle a = v \rangle$ on a specific attribute a, the value v is stored in the query relation, otherwise `NULL` is stored. Let q_i refer to the i'th attribute of the query relation, and d_i refer to the i'th attribute of the data relation. (This is standard syntax in relational algebra.) The result of executing every individual query is equivalent to a θ-join between the data and the query relation, where the join predicate θ is defined as

$$\theta \overset{\text{def}}{=} \forall i : \text{isNull}(q_{i+1}) \lor q_{i+1} = d_i$$

Extending this model to other comparison operators than equality (greater-than, string prefix, etc.) is straight-forward. One simply creates one attribute in the query relation for every possible combination of query attribute and comparison operator, and adjusts the join condition accordingly.

In practice, it is obviously a bad plan to materialize the proposed query relation and execute the join as-is. For all but very small numbers of attributes and comparison operators the query relation is very sparse; i.e., it contains mostly `NULL`-values. However, this naïve plan can serve as a clean starting point on which to perform *algebraic transformations* into semantically equivalent but more efficient plans.

$$Q_1 = \langle\{\text{rloc} = \text{'ABC123'}\}, *\rangle$$
$$Q_2 = \langle\{\text{rloc} = \text{'XYZ234'}, \text{firstName} = \text{'Amy'}\}, *\rangle$$
$$Q_3 = \langle\{\}, *\rangle$$
$$Q_4 = \langle\{\text{firstName} = \text{'Bob'}, \text{productId} = \text{'LH4711'}\}, *\rangle$$

Ticket

rloc	product	firstName
ABC123	LH4711	Amy
ABC123	LH4711	Bob
ABC123	LH4711	Cinthia
XYZ234	BA007	Amy
XYZ234	LX555	Amy
XYZ234	BA007	Bob

Query

qid	rloc	product	firstName
Q1	ABC123	*NULL*	*NULL*
Q2	XYZ234	*NULL*	Amy
Q3	*NULL*	*NULL*	*NULL*
Q4	*NULL*	LH4711	Bob

Query $\bowtie_\theta$ Ticket

qid	rloc	product	firstName
Q1	ABC123	LH4711	Amy
Q1	ABC123	LH4711	Bob
Q1	ABC123	LH4711	Cinthia
Q2	XYZ234	BA007	Amy
Q3	ABC123	LH4711	Amy
Q3	ABC123	LH4711	Bob
Q3	ABC123	LH4711	Cinthia
Q3	XYZ234	BA007	Amy
Q3	XYZ234	LX555	Amy
Q3	XYZ234	BA007	Bob
Q4	ABC123	LH4711	Bob

Figure 2.7: *Query-Data Join Example for Multiple Predicate Attributes*

For instance, assume we are given a set of queries, each a conjunction of equality predicates, as in Figure 2.7. Let Q = $\langle a_1, a_2, ...a_n\rangle$ be the schema of the query relation Q, where a_1 to a_n are the attributes of the data relation D. Let us vertically partition (project) Q into the following set of relations: QP0 = $\langle\text{qid}\rangle$, QP1 = $\langle\text{qid}, a_1\rangle$, QP2 = $\langle\text{qid}, a_2\rangle$, ... QP$n$ = $\langle\text{qid}, a_n\rangle$. Tuples containing `NULL`-values can be omitted. Consider Figure 2.8 for an example of such a partitioning.

Let us use $=\!\bowtie$ as the symbol for a *left outer join*. It should be easy to see that the following equivalence holds:

$$\text{Q} \bowtie_\theta \text{D} = \text{QP0} \times \text{D} =\!\bowtie \text{QP1} =\!\bowtie \text{QP2}... =\!\bowtie \text{QP}n$$

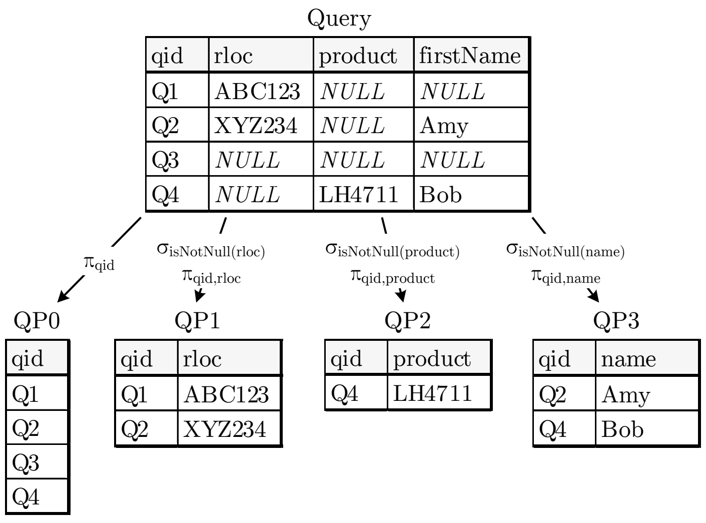


Figure 2.8: *Vertical Partitioning of the Query Relation*

In fact, the schema of QP0 × D contains all the attributes in $QP1$, $QP2$, etc. The left outer joins can thus be replaced by left outer *semi*-joins, denoted by ⋉.

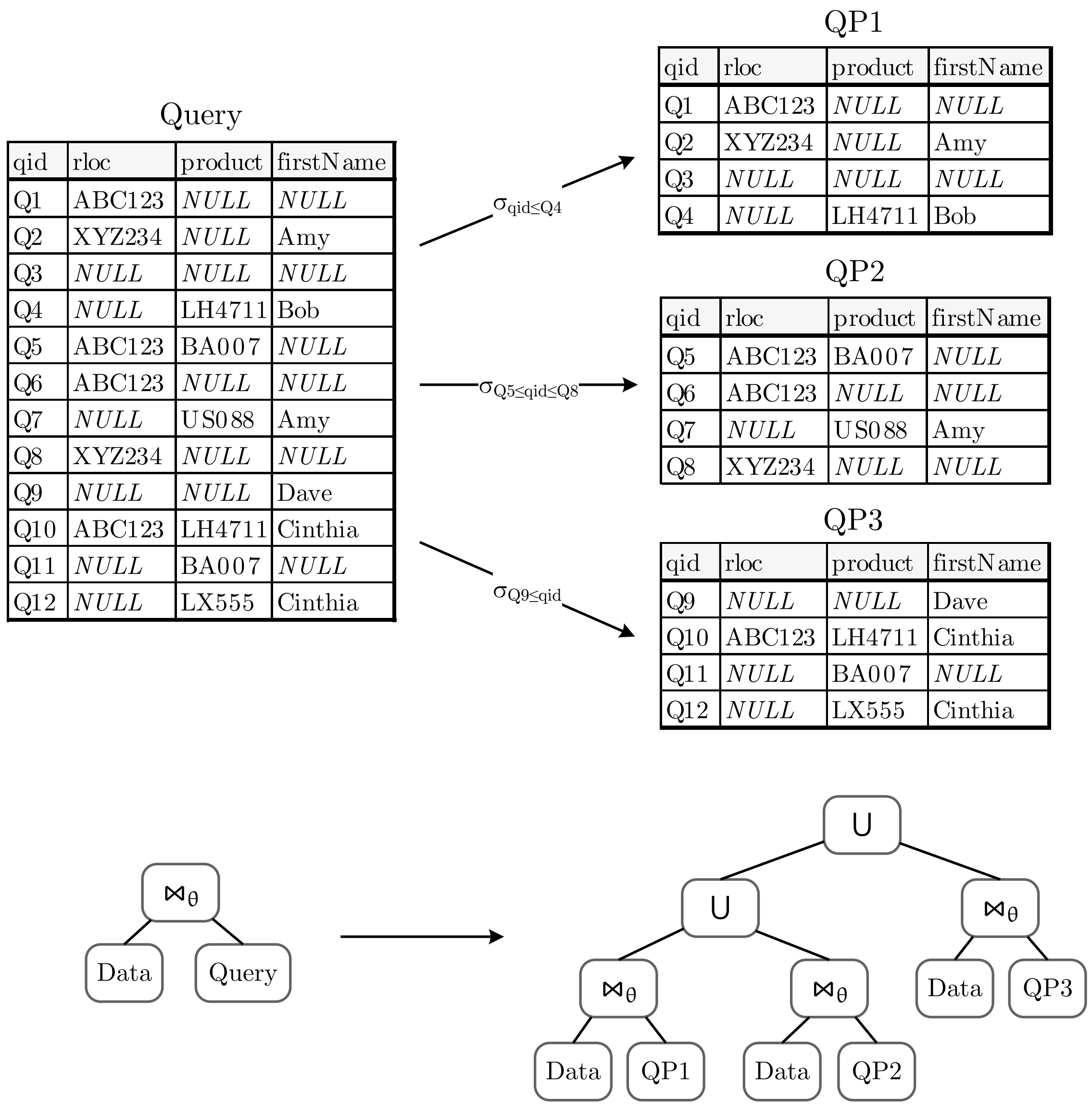
$$\text{Q} \bowtie_\theta \text{D} = \text{QP0} \times \text{D} \ltimes \text{QP1} \ltimes \text{QP2} ... \ltimes \text{QP}n$$

This is now already a quite reasonable execution plan. In words, it says “take every combination of query ID and data tuple (cross-product), filter out all tuples that do not match some tuple in QP1, and repeat the filtering step for the remaining query-partition relations QP2 to QPn.”

At this point, many known optimizations from traditional query processing can be applied. For instance, the intermediate results need not be materialized, they can be *pipelined* (in particular, the cross-product of QP0 and D). The left outer semi-joins can be *reordered* to minimize the number of intermediate results. And most importantly, the query partition relations (QP1 etc.) can be *indexed* to speed up the joins.

2.6.2 Join Algorithms

As explained previously, various algebraic transformations and optimizations can be applied to the basic query-data join model. Each of the resulting “plan templates” can be seen as a specific *query-data join algorithm*. In this section, we present the algorithms that are currently supported by Clock Scan. Having a clean model not only helps in reasoning about the correctness and runtime complexity of these algorithms, it also helps in finding these algorithms in the first place.



Query

qid	rloc	product	firstName
Q1	ABC123	*NULL*	*NULL*
Q2	XYZ234	*NULL*	Amy
Q3	*NULL*	*NULL*	*NULL*
Q4	*NULL*	LH4711	Bob
Q5	ABC123	BA007	*NULL*
Q6	ABC123	*NULL*	*NULL*
Q7	*NULL*	US088	Amy
Q8	XYZ234	*NULL*	*NULL*
Q9	*NULL*	*NULL*	Dave
Q10	ABC123	LH4711	Cinthia
Q11	*NULL*	BA007	*NULL*
Q12	*NULL*	LX555	Cinthia

QP1

qid	rloc	product	firstName
Q1	ABC123	*NULL*	*NULL*
Q2	XYZ234	*NULL*	Amy
Q3	*NULL*	*NULL*	*NULL*
Q4	*NULL*	LH4711	Bob

QP2

qid	rloc	product	firstName
Q5	ABC123	BA007	*NULL*
Q6	ABC123	*NULL*	*NULL*
Q7	*NULL*	US088	Amy
Q8	XYZ234	*NULL*	*NULL*

QP3

qid	rloc	product	firstName
Q9	*NULL*	*NULL*	Dave
Q10	ABC123	LH4711	Cinthia
Q11	*NULL*	BA007	*NULL*
Q12	*NULL*	LX555	Cinthia

Figure 2.9: *Block Nested Loops Join: Query Relation and Plan Transformation*

Block Nested Loops Join

Given a set of queries that share a scan cursor, the most straight-forward way to execute all the queries is this: scan the data relation exactly once; for every record under the scan cursor, iterate over the set of queries; for every query, if all predicates match, generate a result tuple. Interpreting this algorithm as a query-data join, it is the equivalent of a (tuple) nested loops join in traditional query processing.

Function `BlockNestedLoopsJoinPlan.Join(`*Chunk chunk, ResultQueue outQueue*`)`

```
Data: OpSet $operations$
foreach Record $r \in chunk$ do
    foreach Operation $o \in queries$ do
        o.Execute($r$, $outQueue$)
```

Performance problems arise if the query relation is larger than the CPU cache. Because CPUs typically use some variant of LRU (least-recently-used) as a cache replacement policy [Dre07], repeated sequential scan of a large query relation leads to very poor cache-hit ratios. The solution is to partition the data and query relations into small-enough blocks to avoid cache misses. We call this algorithm *block nested loops join* (BNLJ), just like its analogue in traditional query processing.

The horizontal partitioning of the query relation can be interpreted as transforming a single query-data join into multiple query-data joins on smaller query relations, where the results of the individual joins are union'ed at the end (simply by virtue of being put into a common output queue). The transformation of the query relation and query-data join plan is visualized in Figure 2.9.

Recall that Crescando horizontally partitions the table of records into segments, each mapped to a single, dedicated scan thread (cf. Section 2.3). Each segment in turn is horizontally partitioned into chunks. The Clock Scan algorithm (cf. Section 3) executes its query-data join plan one chunk at a time, by calling 4. Thus, chunks are the "blocks" of the data relation. The query relation in turn is partitioned into blocks simply by limiting the number of operations that are activated at every scan cycle.

Let S_{L1} be the size of the L1 cache of the machine Crescando runs on. By default, the chunk size is set to $S_{L1}/3$, and Clock Scan limits the cumulative size of activated operations to $S_{L1}/3$ per scan cycle. (The remaining cache capacity is reserved for other data.) The default settings work well for a wide range of data schema and workloads, but users of Crescando may override them.

Because BNLJ is very simple (few instructions, sequential access patterns), it works well for small numbers of queries (low load). However, the asymptotic runtime complexity of Clock Scan with BNLJ is still $O(n * m)$ for n queries over m records.

Function `IndexUnionJoinPlan.Join(`*Chunk chunk, ResultQueue outQueue*`)`

```
Data: IndexSet indexes;                    /* set of predicate indexes */
Data: ReadSet unindexed;      /* set of unindexed queries (basic reads) */
foreach Record $r \in chunk$ do
    /* probe the indexes for candidate queries                        */
    ReadSet $C \leftarrow unindexed$
    foreach Index $i \in indexes$ do
        $C \leftarrow C \cup$ i.Probe($r$)
    /* execute candidate queries                                       */
    foreach Read $q \in C$ do
        q.Execute($r$, $outQueue$)
```

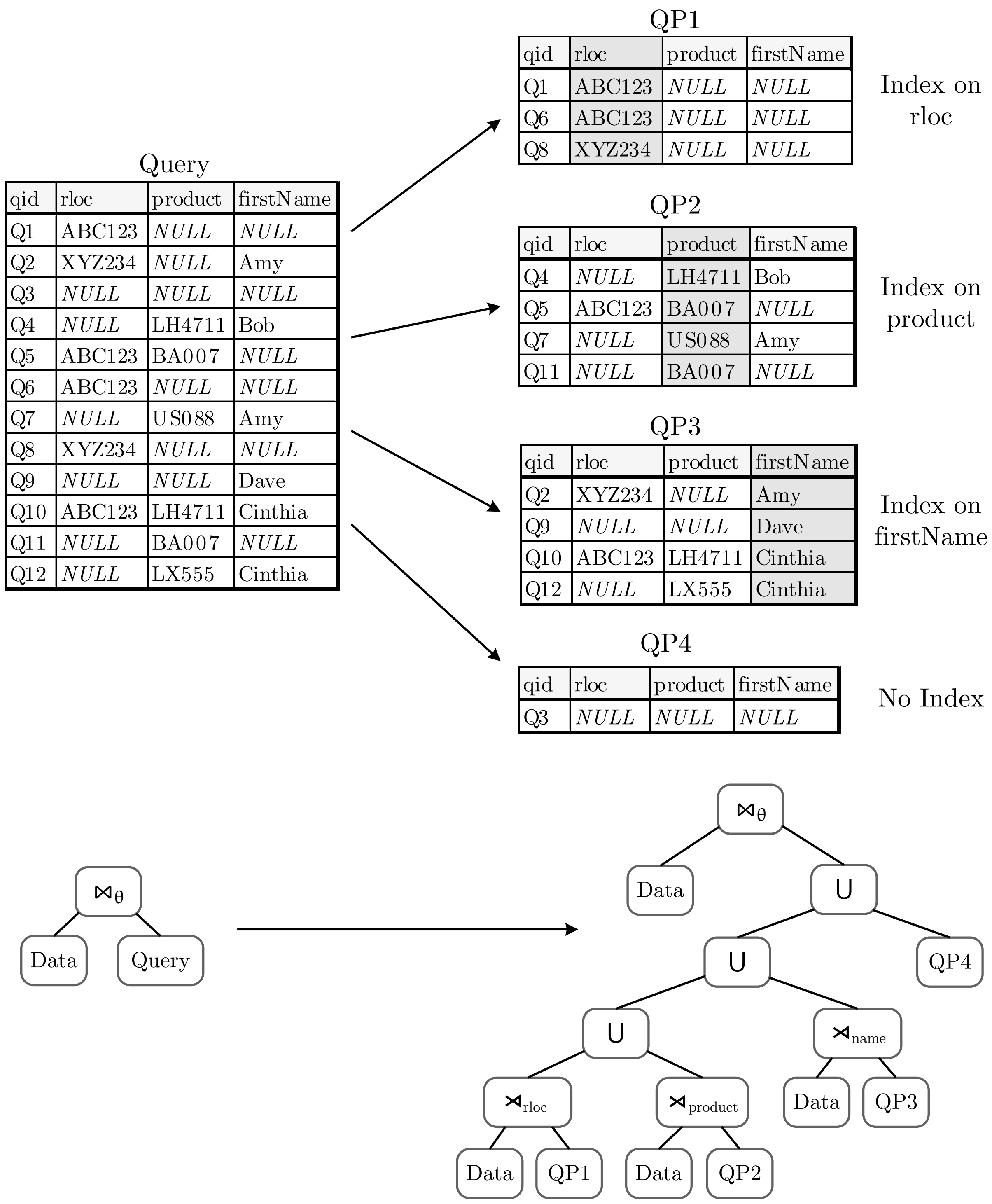
Index Union Join

Assume every select operation in the workload has an equality predicate on the attribute *rloc*. In a traditional database management system, one would probably build a hash index on *rloc* over the data, which is then probed for every single operation. Interpreting the execution of multiple select operations i.e. queries as a query-data join, this approach can be described as a *index nested loop join*, where the query relation is the outer, scanned relation, and the data relation is the inner, indexed relation.

In Clock Scan, many queries share a scan cursor, so the join can be turned inside-out. The data relation becomes the outer, scanned relation, and the query relation becomes the inner, indexed relation. Given a set of queries, each with an equality predicate on *rloc*, one can for example build a hash index on the *rloc* attribute of the query relation (the *rloc* values to check for equality). For every scanned record, the index is then probed, and result tuples are generated for all queries found.

Things become more complicated if we consider arbitrary conjunctions of predicates over multiple attributes, as supported by Crescando. One index is unlikely to cover all or most queries. The solution is to partition the query relation into sets of queries that share some predicate attribute, index the partitions individually, and perform one index nested loops join per partition. Queries that cannot be indexed are handled as a nested loops join. Assuming a proper partitioning (every query is in exactly one partition), the result tuples of the various joins can simple be union'ed. We call this algorithm *Index Union Join*.

The transformation of an exemplary query relation and the transformation of the basic query-data join plan into an Index Union Join is visualized in Figure 2.10. The function, the set of predicate indexes, and the set of unindexed queries together form what is called the *query-data join plan* in Clock Scan (cf. Algorithm 3).



Query

qid	rloc	product	firstName
Q1	ABC123	*NULL*	*NULL*
Q2	XYZ234	*NULL*	Amy
Q3	*NULL*	*NULL*	*NULL*
Q4	*NULL*	LH4711	Bob
Q5	ABC123	BA007	*NULL*
Q6	ABC123	*NULL*	*NULL*
Q7	*NULL*	US088	Amy
Q8	XYZ234	*NULL*	*NULL*
Q9	*NULL*	*NULL*	Dave
Q10	ABC123	LH4711	Cinthia
Q11	*NULL*	BA007	*NULL*
Q12	*NULL*	LX555	Cinthia

QP1 — Index on rloc

qid	rloc	product	firstName
Q1	ABC123	*NULL*	*NULL*
Q6	ABC123	*NULL*	*NULL*
Q8	XYZ234	*NULL*	*NULL*

QP2 — Index on product

qid	rloc	product	firstName
Q4	*NULL*	LH4711	Bob
Q5	ABC123	BA007	*NULL*
Q7	*NULL*	US088	Amy
Q11	*NULL*	BA007	*NULL*

QP3 — Index on firstName

qid	rloc	product	firstName
Q2	XYZ234	*NULL*	Amy
Q9	*NULL*	*NULL*	Dave
Q10	ABC123	LH4711	Cinthia
Q12	*NULL*	LX555	Cinthia

QP4 — No Index

qid	rloc	product	firstName
Q3	*NULL*	*NULL*	*NULL*

Figure 2.10: *Index Union Join: Query Relation and Plan Transformation*

In its outermost loop, Index Union Join iterates over the records of the input chunk. For every record, the algorithm probes every single predicate index, and puts all matching queries into the set of candidate queries C. We call these queries *candidates*, because they each may have additional (non-indexed) predicates that are yet to be evaluated. Finally, all the candidate queries and the set of unindexed queries are iterated over. For each query, the `Execute` function checks whether all the predicates match, in which case it puts a result tuple into the output queue.

The execution plan in Figure 2.10 visualizes what happens in Index Union Join. In algebraic terms, probing an index for candidate queries is a *right semi-join* written as $\rtimes$. Calling the `Execute` function in a loop is a tuple nested loops join, where the join predicate `θ` is encoded in the `Execute` function.

Note that any query can have multiple selection predicates, so there may be many valid partitionings of the query relation. For example, Q5 in Figure 2.10 could just as well have been added to QP1 instead of QP2. Depending on the types of attributes, types of index data structures, and selectivity of individual predicates, one partitioning may be better than another. Quickly finding a good partitioning is a hard optimization problem, which is discussed in Section 2.7.

In contrast to the pseudo-code, the actual implementation of Index Union Join vectorizes the index probing; i.e., it passes the entire chunk to an index' `Probe` function instead of a single record. This improves data and instruction cache locality. Also, the implementation uses a visitor pattern to `Execute` all matching queries inside the `Probe` function, instead of materializing the candidate query set C (dynamic data structures are costly to maintain). These two optimizations can be interpreted as batching and pipelining of intermediate results in the execution plan.

As regards performance, the worst-case (every record matches every index) runtime complexity of Clock Scan with Index Union Join is no better than Classic Scan, Elevator Scan, or Clock Scan with Block Nested Loops Join. However, Index Union Join is a superior choice for any large, reasonably selective set of queries. In this case, runtime is dominated by the *cost of probing the indexes*, which is constant (hash indexes) or logarithmic (trees) in the number of queries sharing a scan cursor. As shown in Section 2.10 the difference in throughput and latency is *orders of magnitude* for realistic workloads.

```
Function IndexUnionUpdateJoinPlan.Join(Chunk chunk, ResultQueue outQueue)
Data: IndexSet indexes;                                   /* predicate indexes */
Data: OpSet unindexed;                                 /* unindexed operations */
Data: InsertQueue insertQueue;                   /* pending insert operations */
foreach Slot $s \in chunk$ do
    Timestamp $t \leftarrow 0$
    while $t < \infty$ do
        if IsOccupied($s$) then
            /* slot is occupied; perform updates and delete operations */
            $t \leftarrow$ PerformNonInserts($s$, $t$, indexes, unindexed, outQueue)
        else if $\neg$insertQueue.IsEmpty() then
            /* slot not occupied, have insert operation; execute it */
            Insert $i \leftarrow$ insertQueue.Get()
            i.Execute($s$)
            $t \leftarrow i.timestamp + 1$
        else
            /* slot not occupied, no pending inserts; go to next slot */
            $t \leftarrow \infty$
```

Index Union Update Join

We now present an extension of Index Union Join that uses predicate indexes also for update and delete operations: Index Union Update Join. Technically, this is not a join, since a join in relational algebra is a side effect-free function. We nevertheless refer to the algorithm presented here as an update-data join, because its implementation and the related optimization problem is sufficiently similar to a query-data join.

The difference to query-data joins is that writes have to be executed in activation order to guarantee consistency (cf. Section 2.5). What makes this hard to do efficiently is the fact that a slot's state may change after each write, thereby changing the set of operations whose predicates match.

Pseudo-code for Index Union Update Join is given above. This is the function that Clock Scan calls for every chunk. The function's persistent state (i.e., the update-data join plan) consists of a queue of pending insert operations, a set of predicate indexes, and a set of unindexed operations (update, delete, and select operations with *strict* consistency level). For simplicity, we assume here that each insert operation consists of exactly one record. The actual implementation does not have this restriction.

```
Function PerformNonInserts(Slot s, Timestamp t, IndexSet indexes, OpSet unindexed, ResultQueue outQueue): Timestamp
  /* probe indexes for candidates */
  OpSet $C \leftarrow unindexed$;
  foreach Index $i \in indexes$ do
      $C \leftarrow C \cup$ i.Probe($s.record$);
  /* find matches; i.e., candidates with timestamp $\geq t$, which match completely */
  OpSet $M \leftarrow \emptyset$;
  foreach Op $o \in C$ do
      if $o.timestamp \geq t \wedge o$.Matches($s.record$) then
          $M \leftarrow M \cup \{o\}$;
  while $M \neq \emptyset$ do
      /* execute match with lowest timestamp */
      Op $o \leftarrow \min_{o.timestamp}\{o \in M\}$;
      o.ExecuteUnconditionally($s$, $outQueue$);
      if IsSelect($o$) then
          /* slot is unchanged; remove op from match set and execute next match */
          $M \leftarrow M \setminus \{o\}$;
      else if IsUpdate($o$) then
          /* slot was updated; repeat function with higher timestamp */
          return PerformNonInserts($s$, $o.timestamp + 1$, $indexes$, $unindexed$);
      else
          /* slot is empty now (operation was a delete); return */
          return $o.timestamp + 1$;
  /* no more matches; return */
  return $\infty$;
```

It aids understanding to first explain the inner function, `PerformNonInserts`. This function is an extension of Index Union Join shown in the previous section. After probing the predicate indexes for candidate update and delete operations, it iterates over all candidates C to find the subset of fully matching operations M.

In Clock Scan, every write and strict read has a timestamp, which is a natural number that corresponds to their activation order by Clock Scan. After computing the set of matching operations M, `PerformNonInserts` looks for the update or delete $o \in M$ with the lowest timestamp greater or equal to t and executes it. (The idea behind t is explained momentarily.) If no such operation exists, the function returns ∞.

If the executed operation o was a (strict) select, the record in the slot has not changed, so execution can directly continue with the match with the next-higher timestamp. If o was a delete, then `PerformNonInserts` returns with $o.timestamp+1$. Otherwise, the function calls itself (tail recursion) with $t = o.timestamp + 1$.

In our optimized implementation, the candidate set C is never materialized. Instead, a visitor pattern is used to evaluate all operation predicates directly in the index `Probe` function, and put matching operations directly into the match set M. Also, the tail-recursive call is just an assignment to t and a `goto` statement.

As regards the timestamp t, note that t is always greater than the timestamp of any operation that has previously been executed against the current record in the slot. This is a critical invariant. `PerformNonInserts` always executes the matching operation with the *lowest timestamp greater or equal* to t. Thus, operations are executed strictly in activation order with respect to any record.

Now, let us look at the main function of Index Union Update Join. For each slot, the function first initializes the variable t to 0. Then it runs the following loop. If the slot is occupied (contains a record), it calls `PerformNonInserts`. This will execute one or more matching operations with timestamps greater or equal to t and return a new, greater timestamp t. If the slot is not occupied (contains no record), the function executes the next pending insert operation i and sets t to $i.timestamp+1$. This ensures that subsequent iterations and calls to `PerformNonInserts` will not execute update and delete operations which precede the insert in activation order. Note that executing an insert against an empty slot may reset t to a lower value. This is okay, because the inserted record is different from any record that was in the slot previously.

Index Union Update Join loops over the slot until t becomes ∞. This happens if and only if the slot is occupied but no matching operations with a high enough timestamp exist, or the slot is empty and no pending insert operations remain. Until t becomes ∞ however, the slot may repeatedly change its state from occupied to unoccupied, as a sequence of insert and delete operations are executed.

A step-by-step example for how Index Union Update Join executes 5 operations against an initially empty slot is given in Figure 2.11. The schema used in the example is $\langle \text{rloc}, \text{firstName} \rangle$. The syntax used to describe an operation is $\langle timestamp, type, details \rangle$, where *details* is defined in the data and query-model in Section 2.2.

As for performance, we note that in the *worst-case* there are only update operations, and each of those n update operations matches every record, every time `PerformNonInserts` is called. The depth of the recursion becomes n, so the runtime complexity for calling `PerformNonInserts` from 6 is $O(n^2)$. Thus, the worst-case runtime complexity for an

$\langle 0, \text{update}, \langle \{\text{rloc} = \text{'ABC123'}\}, \{\langle \text{firstName}, \text{'Bob'} \rangle\} \rangle \rangle$
$\langle 1, \text{insert}, \langle \text{'ABC123'}, \text{'Amy'} \rangle \rangle$
$\langle 2, \text{insert}, \langle \text{'XYZ2345'}, \text{'Bob'} \rangle \rangle$
$\langle 3, \text{update}, \langle \{\text{rloc} = \text{'ABC123'}\}, \{\langle \text{firstName}, \text{'Cinthia'} \rangle\} \rangle \rangle$
$\langle 4, \text{delete}, \langle \{\text{firstName} = \text{'Cinthia'}\}, \{\langle \text{firstName}, \text{'Cinthia'} \rangle\} \rangle \rangle$

t	Slot Contents	Insert Queue	Candidates	Matches	Executed	t_{new}
0		$\langle 1, 2 \rangle$			1	2
2	⟨'ABC123', 'Amy'⟩	$\langle 2 \rangle$	$\{0, 3\}$	$\{3\}$	3	4
3	⟨'ABC123', 'Cinthia'⟩	$\langle 2 \rangle$	$\{0, 3, 4\}$	$\{4\}$	4	5
5		$\langle 2 \rangle$			2	3
3	⟨'XYZ2345', 'Bob'⟩	$\langle \rangle$	$\{\}$	$\{\}$		∞

Figure 2.11: *Index Union Update Join: Step-by-Step Dry Run*

Index Union Update Join of n operations and m records is $O(n^2 * m)$, which is *worse* than the $O(n * m)$ of a straight-forward, sequential execution of all update operations.

In practice, of course, updates are highly selective operations. While there are many use cases for select operations that match large fractions of the table, it makes very little sense to repeatedly overwrite large fractions of a table at a significant rate. Therefore, in practice, the match set M will typically contain 0 or 1 updates. Runtime is thus dominated by the cost of probing the indexes, which is constant or logarithmic in n.

As regards the cost of select operations, Section 2.5 already stated that *strict* reads are more expensive than *basic* reads. The reason is that basic reads are executed by Index Union Join, whereas strict reads require Index Union Update Join. The latter is more expensive, because the match set M in Function 7 has to be constructed and maintained. Also, the execution of strict reads cannot be batched and pipelined as effectively as that of basic reads in Index Union Join.

Other Join Algorithms

At the time of writing, the only query/update-data join algorithms available in Crescando are Block Nested Loops Join, Index Union Join, and Index Union Update Join. During the development however, we experimented with a few other algorithms. These algorithms which did not meet our expectations eventually vanished again. Nevertheless, they serve to further illustrate the generality of the query-data join idea, and may have niche applications in other systems. Two of these algorithms are worth mentioning: Index Intersection Join, and Sort-Merge Join.

Index Intersection Join As explained previously, Index Union Join expects every individual query to be in exactly one predicate index, or in the set of unindexed queries. A natural generalization of the algorithm is to allow each query to be in *multiple* predicate indexes. On first sight, this seems beneficial in cases where a query is not very selective on any individual predicate, but the conjunction of predicates becomes very selective.

Assume a given query q is known to be referenced by, say, 3 predicate indexes. To test whether q matches a given record, it is sufficient to probe every index (there may be more than 3), count the number of occurrences of q, and, if that number is 3, evaluate the remaining, non-indexed predicates of q.

Of course, the predicate indexes need not be probed for every individual query. Instead, *Index Intersection Join* probes the indexes *once* for any given record, collects the number of occurrences of every query found, and later compares these to the number of required occurrences. In essence, counting the number of occurrences requires an *intersection* of the results of probing every individual predicate index. This is reminiscent of query processing in column stores [ABH], and similar techniques can be used (bitmap joins).

In practice, Index Union Join out-performed Index Intersection Join by at least one order of magnitude for all reasonable workloads. The reason is that computing the intersection is too expensive, despite our best efforts at optimization. For queries with at least one highly selective predicate, it is not worth the trouble, because a single predicate index is sufficient for good performance. For all other queries, indexing does not pay off in the first place. That being said, in other systems with very particular workloads and types of indexes, this may not hold and an Index Intersection Join may be a good idea after all.

Sort-Merge Join Indexes are not the only way to improve the asymptotic runtime of joins. An alternative idea is to *sort* both the data and the query relation on some attribute, and then perform a *merge join* between the two relations. This works even for range predicates, in which case one would perform a *sort-merge band join* [DNS91].

We quickly found however that Index Union Join is more effective than Sort-Merge Join for any workload, even for pre-sorted data, due to the low cost of random access (index probing) in cache memory. Some variant of Sort-Merge Join may however be interesting for hardware-based implementations of query-data joins, where maintaining large, complex data structures (indexes) is difficult or ineffective.

$$
\begin{aligned}
Q_1 &= \langle \{\text{sex} = \text{'false'}\}, *\rangle \\
Q_2 &= \langle \{\text{sex} = \text{'true'}\}, *\rangle \\
Q_3 &= \langle \{\text{sex} = \text{'false'}\}, *\rangle \\
Q_4 &= \langle \{\text{sex} = \text{'false'}\}, *\rangle \\
Q_5 &= \langle \{\text{sex} = \text{'true'}\}, *\rangle
\end{aligned}
$$

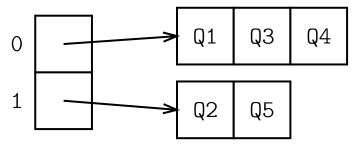


Figure 2.12: *Example Jagged Array*

2.6.3 Predicate Index Structures

In this section, we present the various types of predicate index structures that Clock Scan uses in combination with Index Union Join and Index Union Update Join. The idea of indexing query predicates is not entirely new. As discussed in Section 2.1.4, many existing publish-subscribe and data-streaming systems index query predicates [CF02, FJL$^+$01, WCY04, LJ03, FSS$^+$10]. In these systems, queries are long-lived (*continuous*), and are added and removed from the system on a continuous basis. Consequently, their index structures must be somewhat dynamic. Also, the index structures are typically designed for extremely large numbers of queries (millions), due to use cases in online advertising or information dissemination (e.g., news feeds).

In contrast, queries in Clock Scan have a lifetime of just 1 scan cycle, which means a few seconds at most. Because the indexes are built at the beginning of a scan cycle and are not updated subsequently, the data structures can be *read-only.* Also, Clock Scan is designed to let thousands of queries share a scan cursor, not millions, so cache efficiency often trumps asymptotic complexity. For these reasons, the predicate index structures we present here differ significantly from related work.

Jagged Array

The first, simplest index structure is called a *jagged array.* This data structure is appropriate only for equality predicates over a very small attribute domain. Crescando uses it for equality predicates on `BOOLEAN`, `CHAR`, `INT(8)`, and `UINT(8)` attributes.

A jagged array is an "array of arrays". Every element of the outer array points to the beginning of a (variable-sized) inner array. Each inner array contains a sequence of oper-

$$
\begin{aligned}
Q_1 &= \langle \{\text{firstName} = \text{'Amy'}\}, *\rangle \\
Q_2 &= \langle \{\text{firstName} = \text{'Bob'}\}, *\rangle \\
Q_3 &= \langle \{\text{firstName} = \text{'Cindy'}\}, *\rangle \\
Q_4 &= \langle \{\text{firstName} = \text{'Dave'}\}, *\rangle \\
Q_5 &= \langle \{\text{firstName} = \text{'Eve'}\}, *\rangle \\
Q_6 &= \langle \{\text{firstName} = \text{'Bob'}\}, *\rangle \\
Q_7 &= \langle \{\text{firstName} = \text{'Dave'}\}, *\rangle \\
Q_8 &= \langle \{\text{firstName} = \text{'Bob'}\}, *\rangle
\end{aligned}
$$

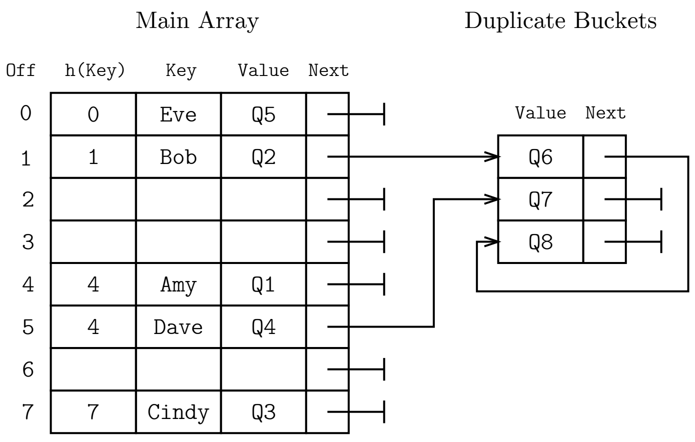


Figure 2.13: *Example Hash Map*

ation IDs. During index lookup, the lookup key (attribute value) is used as a direct offset into the outer array. A concrete example is given in Figure 2.12.

Hash Map

Crescando uses hash maps for equality predicates whenever the attribute domain is too large for the use of a jagged array. To be precise, Crescando uses *closed hashing* with *linear probing* (interval = 1). Duplicates (operations with a selection predicate on the same key) are chained using buckets from a pre-allocated pool of memory. The first element however is put directly into the main array. A concrete example is given in Figure 2.13.

During development, we experimented with many different hashing schemes. Contrary to the findings of Heileman and Luo [HL05] made in 2005, we found that closed hashing with linear probing consistently out-performed double hashing and more complex schemes. While we have conducted no systematic study on the topic, we believe there are plausible reasons for this result. For one thing, the performance gap between CPU cache and DRAM has continued to widen since 2005, shifting the trade-off between sequential access (linear probing) and random access (double hashing etc.) further toward sequential access. For another thing, the set of entries is small and read-only in our case. For a static set of a few hundred to a few thousand entries, it is easy to allocate a sufficiently large array which keeps the number of hash collisions low, and still fit everything into the cache.

We optimized the cost of duplicate traversal by using a pre-allocated, dense pool of overflow buckets. Buckets are very close to each other in memory, so few cache misses are incurred. Also, logical offsets are used rather than plain pointers, significantly reducing the memory footprint of the entire data structure (especially on 64-bit platforms)[4].

That being said, lookup performance does degrade somewhat for large numbers of duplicates. This is a not cause for concern however, because pointer-chasing only occurs when a lookup finds multiple candidate queries, in which case system performance is not dominated by the hash map, but by the rest of the join algorithm and by result creation.

Range Tree

So far, all the examples and data structures we have shown used only equality predicates. But Crescando can also index range predicates. To this end, the system constructs compact range trees, which are equivalent to 1-dimensional R-Trees [Gut84]. Refer to Figure 2.14 for the logical structure of a range tree over an exemplary set of range predicates.

To find the set of operations whose range predicates cover a given lookup key (attribute value), one starts at the root node, and recursively visits all child nodes whose range covers the lookup key. Note that ranges may overlap, so multiple leaf nodes may be visited during a single lookup.

The actual implementation does not use pointers to traverse the tree. Because the data structure is read-only, the entries can be *bulk loaded* into a balanced tree. Nodes are stored compactly, in a pre-allocated, contiguous region of local NUMA memory. Child nodes are addressed through offset calculations. In short, the exemplary tree in Figure 2.14 really looks like Figure 2.15. Also, for illustration purposes, the intermediate and leaf fan-out are 2 in the example. In our implementation, the intermediate and leaf fan-out are 4 or higher, tuned to the specific attribute type.

The implementation ensures that intermediate and leaf nodes are cache aligned. Also, the size of any node is a type-specific power of two (nodes containing `UINT(8)` ranges are smaller than nodes containing `UINT(64)` ranges for instance). Consequently, offset

[4]Because the data structure is read-only, bucket chains could even be coalesced into contiguous arrays after the hash map has been loaded. We reserve this small optimization for future code revisions.

$$\begin{aligned}
Q_1 &= \langle \{\text{age} \geq 30, \text{age} \leq 50\}, *\rangle \\
Q_2 &= \langle \{\text{age} > 70\}, *\rangle \\
Q_3 &= \langle \{\text{age} > 25, \text{age} < 30\}, *\rangle \\
Q_4 &= \langle \{\text{age} \leq 50\}, *\rangle \\
Q_5 &= \langle \{\text{age} \geq 40, \text{age} < 50\}, *\rangle \\
Q_6 &= \langle \{\text{age} < 40\}, *\rangle \\
Q_7 &= \langle \{\text{age} \geq 30, \text{age} \leq 30\}, *\rangle \\
Q_8 &= \langle \{\text{age} \leq 25\}, *\rangle
\end{aligned}$$

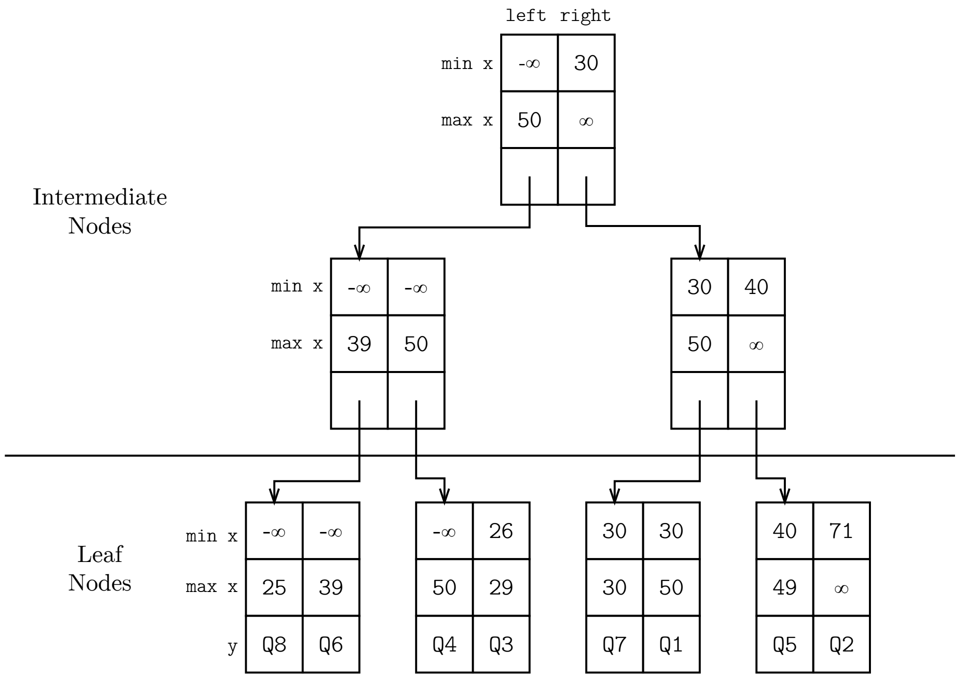
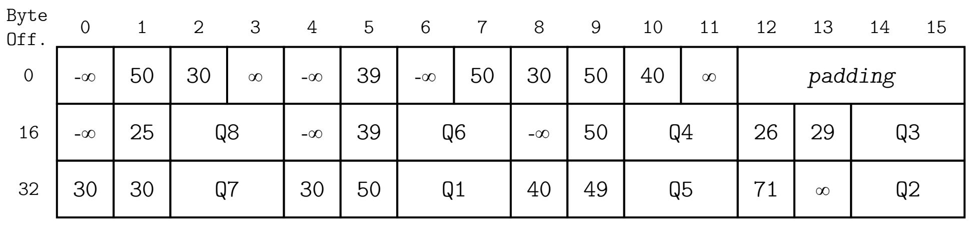


Figure 2.14: *Example Range Tree: Logical Structure*



Byte Off.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	-∞	50	30	∞	-∞	39	-∞	50	30	50	40	∞	*padding*			
16	-∞	25	Q8		-∞	39	Q6		-∞	50	Q4		26	29	Q3	
32	30	30	Q7		30	50	Q1		40	49	Q5		71	∞	Q2	

Figure 2.15: *Example Range Tree: Physical Structure*

$$
\begin{aligned}
Q_1 &= \langle \{\text{firstName} \sqsubset \text{'Alf'}\}, *\rangle \\
Q_2 &= \langle \{\text{firstName} \sqsubset \text{'A'}\}, *\rangle \\
Q_3 &= \langle \{\text{firstName} \sqsubset \text{'Bob'}\}, *\rangle \\
Q_4 &= \langle \{\text{firstName} \sqsubset \text{'Albe'}\}, *\rangle \\
Q_5 &= \langle \{\text{firstName} \sqsubset \text{'Ambe'}\}, *\rangle \\
Q_6 &= \langle \{\text{firstName} \sqsubset \text{'Alf'}\}, *\rangle
\end{aligned}
$$

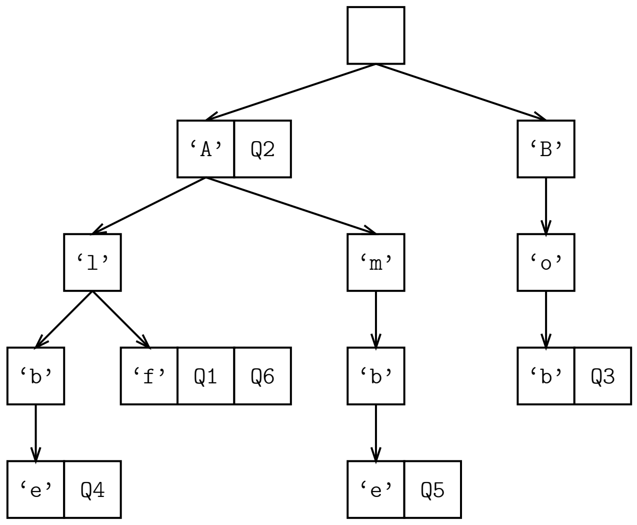


Figure 2.16: *Example Trie*

calculations require just additions and bit-shifting rather than multiplication. Finally, C++ template classes are used to instantiate optimized range tree implementations for different types of attributes (from `UINT(8)` to `TIMESTAMP`), allowing the compiler to heavily optimize offset calculations and inline the type-specific comparison functions.

Trie (Prefix Tree)

Prefix search is a common operation for string attributes. To name an example from the Amadeus Ticket use case, whenever a passenger has forgotten his or her booking number (*rloc*), it may become necessary to search for the booking by passenger name. But because passenger names are often truncated and suffixed in strange ways by airlines, an equality predicate is unlikely to yield results. A prefix search on the first few characters of the passenger's name however is a great way to narrow down the search.

In principle, a prefix predicate can be converted into a pair of range predicates. For example, the predicate $\{\text{firstName} \sqsubset \text{'Doe'}\}$ can be converted into $\{\text{firstName} \geq \text{'Doe'},$

firstName < 'Dof'}. However, the range tree index structure described previously is not designed for variable-length attributes[5]. Crescando provides a specialized index structure for string-prefix predicates: a trie, also called prefix tree.

An example of such a trie is given in Figure 2.16. Traversing this data structure is (conceptually) straight forward. To find the set of queries matching a passenger named, "Alfred" in the example trie, one starts at the root node, proceeds to node 'A' where one finds the matching query Q_2, proceeds to node 'l', and finally to node 'f' where one finds the matching queries Q_1 and Q_6. The result of probing the trie with "Alfred" is thus $\{Q_1, Q_2, Q_6\}$.

Since the original proposal by Fredkin [Fre60], a number of trie variants have been proposed, which improve upon the original in terms of space efficiency and lookup cost by merging leaf nodes and intermediate nodes in various ways [AS07, BHF09a, GJL09, HZW02, Mor68, Ris05]. However, all the trie variants we are aware of have either been designed for large numbers (millions) of dynamic entries, or even larger numbers (billions) of static entries. We implemented some of these data structures and found that they perform poorly, due to their high number of code branches and disregard of modern hardware features. We therefore designed and implemented our own trie variant which is heavily optimized for storing a few thousand entries, and which takes advantage of micro-parallelism (SIMD instructions) on modern processors.

SIMD stands for *single instruction, multiple data* and is a class of parallel computers in Flynn's taxonomy [Fly72]. SIMD computers can perform a single instruction (e.g. multiplication) on a vector of values (typically held in some special register) simultaneously. Such operations appear primarily in stream-oriented computations with few data and control dependencies, such as image or video processing.

In Crescando, we found a way to accelerate *trie traversal* using SIMD instructions. Consider the problem of matching a character of a lookup string against the key characters of the children of some intermediate node. This problem appears in any trie variant (not just Crescando tries), and the optimization we present here should be applicable to other trie implementations.

Assume the trie node in question has a fan-out of 8. Assuming the basic Latin alphabet and ignoring character casing, there are 26 possible characters. Without packed comparison, there are two basic ways to find the child node with matching key character, if one exists. One can allocate an array of 26 pointers (of which 18 will be *null*), and use the lookup character as a direct offset into this pointer array (this is the original proposal by Fredkin [Fre60]). Alternatively, one allocates an array (or linked list) of 8 ⟨keychar, pointer⟩

[5]Range trees do support prefix predicates on *compressed* string attributes however, cf. Section 2.9.

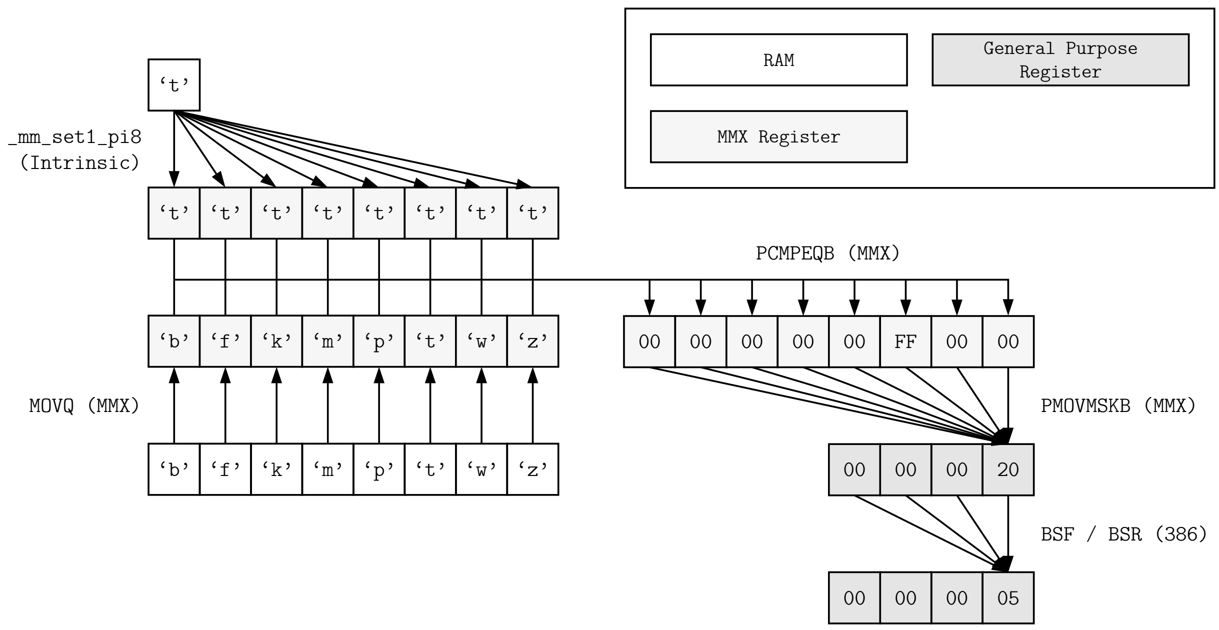


Figure 2.17: *Accelerated Character Matching in Trie Nodes Using SIMD Instructions*

pairs, containing just the key characters and respective child node pointers. The latter approach is more space efficient but requires iteration or binary search over the array.

But using SIMD instructions, one can avoid *both* wasted memory and iteration. The idea is to load the 8 child key characters into a SIMD-enabled register, load the lookup character (8-fold) into another SIMD-enabled register, and perform a *packed comparison* between the two registers. The result of the comparison is a bytemask with at most one byte set, corresponding to the position of the matching characters in the registers. The index of this byte can be computed using "bit-fiddling" instructions, and used as an offset into a separate array of pointers to child nodes.

Figure 2.17 shows the data flow and Assembly instructions for implementing this algorithm on an x86 processor. The example in the figure matches the lookup character 't' against an array of 8 key characters[6]. The necessary MMX instructions have been available since the release of the Intel P55C (Pentium MMX) in 1995. Processors with SSE2 extensions (available since the first Pentium 4 processor in 2001) can compare 16 characters in one step, and the latest processors with Intel Advanced Vector Extensions (AVX) have 256-bit registers and instructions for comparing up to 32 characters in one step [Int11].

Crescando tries feature optimized node implementations for different fan-outs. During bulk loading, the optimal, cache-aligned implementation for each intermediate and leaf

[6]On AMD K10 processors, the `BSF` (bit-scan forward) and `BSR` (bit-scan reverse) instructions are very slow and should be replaced by `LZCNT` (leading zero count) for optimal performance [Gra11].

node is chosen. Intermediate nodes use MMX and SSE2 instructions to match 8 or 16 characters in one step, respectively. AVX instructions are not used at the time of writing, due to the lack of hardware support on our lab machines.

During development, we found that MMX and SSE2 instructions accelerate character matching for fan-outs of 3 to 64. Nodes with a fan-out of 1 are merged with their single child node to minimize pointer chasing. For a fan-out of 2, "iterating" over the 2 key characters is faster. For fan-outs over 64, binary search over the key characters becomes faster than iteration with MMX or SSE2 instructions (matching against 64 characters for example requires 4 iterations with SSE2 instructions).

Like all other types of index structures in Crescando, tries takes advantage of the fact that the set of entries is static. Trie nodes are cache-aligned and stored compactly in a pre-allocated, contiguous region of local NUMA memory. Instead of plain pointers, 16-bit or 32-bit integer offsets are used to address child nodes, which significantly reduces the memory footprint of tries on 64-bit platforms.

2.6.4 Discussion

Crescando is not a stand-alone system, but is intended to form the storage layer of some larger query and transaction processing architecture. In any such architecture, there is a fundamental trade-off between the expressiveness (i.e. potential processing overhead) of operations that are pushed down, and the size of intermediate results that are exchanged between layers.

In the interest of predictability and performance, Crescando restricts select, update, and delete operations to filter on conjunctions of Boolean predicates that are of broad use, and that can also be indexed with great effect. We want to point out however, that with some cost in predictability, complexity, and performance, the model and algorithms presented in this section could be extended to allow more expressive operations and a wider range of use cases.

For example, it is possible to index not only prefix predicates over strings, but arbitrary regular expressions (including SQL `LIKE` predicates). The idea is to merge the deterministic finite automatons (DFAs) that represent individual regular expressions into nondeterministic finite automatons (NFAs) that represent a whole set of regular expressions [DAF+03, DGP+07].

Going even further, Whang et al. [WGMB+09] show how to index not only conjunctions of Boolean predicates (logical `AND`) as supported by Crescando, but also *disjunctions* (logical `OR`). The technique they describe has too much overhead for the use cases considered in this dissertation, but has interesting applications for example in online advertising.

2.7 Multi-Query Optimization

The term *multi-query optimization* traditionally refers to the act of finding common subexpressions among a set of relational queries, for the purpose of sharing intermediate results between those queries [Sel88]. In Clock Scan, the problem is of a different kind. Here, the task is, given a set of queries (or updates), find a query/update-data join plan which minimizes the duration of the next scan cycle. More precisely, the task is to find a set of *predicate indexes* which minimizes the cost of Index Union (Update) Join, or choose Block Nested Loops Join if this is cheaper.

Like traditional query optimization, the problem consists of two parts.

- Estimating the *cost* of an execution plan according to some *metric*.
- *Enumerating* equivalent execution plans for the purpose of finding a (near-)optimal plan with minimum cost.

Cost estimation requires *statistical data* over the records in the table, such as the number of distinct values (i.e., the cardinality) of a given table attribute. In this section, we show a few techniques which Crescando uses to compute attribute cardinalities and histograms as part of regular scans. The techniques and performance results should apply to other systems as well.

Plan enumeration in Crescando is an NP-hard problem, as proven by Steiner [Ste10]. The reason is that even the special case of finding a minimal set of indexes which "cover" all the queries in the query relation is already an instance of the NP-hard minimum set cover problem. Since optimization time counts toward the execution time of the scan cycle, finding *the* optimal solution is out of the question. So instead, Crescando implements a few (highly effective) greedy heuristics.

We first describe the baseline optimizer, of which an earlier version has been published in Unterbrunner et al. [UGAK09]. This optimizer is used in all experiments in Section 2.10 and in the subsequent chapters. Afterwards, we give a brief overview of the extensions proposed by Steiner in his Master's thesis [Ste10].

2.7.1 Baseline Optimizer

The baseline optimizer enumerates plans using the simple greedy algorithm shown in Algorithm 8. First, the algorithm creates a set of all possible pairs $\langle a, t\rangle$, where a is one of the table attributes, and t is one of the predicate types: equality, range, or (string) prefix[7].

[7]We ignore here the fact that not all types of attributes support all types of index. For example, only string attributes support prefix indexes. The implementation enforces these restrictions of course.

Algorithm 8: Baseline Optimizer

```
Data: Gain $thresh$
Input: OpSet $unindexed$; AttributeSet $A$
Output: IndexSet $indexes \leftarrow \emptyset$
/* create set of all possible pairs of attribute and predicate type */
PairSet $C \leftarrow A \times \{\text{equality}, \text{range}, \text{prefix}\}$
/* keep building predicate indexes until gain falls below threshold */
repeat
    Index $idx \leftarrow$ null
    /* first consider building a hash map */
    Pair $\langle a, \text{equality}\rangle \leftarrow max_{\texttt{Gain}(unindexed,\ a,\ equality)}\{\langle a, \text{equality}\rangle \in C\}$
    if Gain($unindexed$, $a$, equality) $>$ $thresh$ then
        $idx \leftarrow$ BuildEqualityIndex($a, unindexed$)
    else
        /* next, consider building a range index */
        Pair $\langle a, \text{range}\rangle \leftarrow max_{\texttt{Gain}(unindexed,a,\text{range})}\{\langle a, \text{range}\rangle \in C\}$
        if Gain($unindexed$, $a$, range) $>$ $thresh$ then
            $idx \leftarrow$ BuildRangeIndex($a, unindexed$)
        else
            /* finally, consider building a prefix index */
            Pair $\langle a, \text{prefix}\rangle \leftarrow max_{\texttt{Gain}(unindexed,a,\text{prefix})}\{\langle a, \text{prefix}\rangle \in C\}$
            if Gain($unindexed$, $a$, prefix) $>$ $thresh$ then
                $idx \leftarrow$ BuildPrefixIndex($a, unindexed$)
    if $idx \neq$ null then
        $indexes \leftarrow indexes \cup idx$
        $unindexed \leftarrow unindexed \setminus \{q \in idx\}$
        $A \leftarrow A \setminus \{a\}$
until $C = \emptyset \vee gain \leq thresh$
```

Initially, no predicate indexes exist and all operations are unindexed. At each iteration of the main loop, the algorithm first attempts to build an equality index (jagged array or hash map) on the attribute with the highest *gain* (to be defined momentarily) for such an index. If the gain is below the threshold, the algorithm considers building a range index (range tree). Finally, the algorithm considers building a prefix index (trie).

If an index has been built, the indexed operations are removed from the set of unindexed operations. Otherwise, no possible type of index has sufficient gain and the algorithm ends.

The gain function is defined as:

$$\texttt{Gain}(O, a, t) := \sum_{o \in O} (1 - \mathit{selectivity}(o, a, t))$$

This definition embodies the following heuristic. Assuming that predicates are highly selective, then the more operations can be indexed, the more operations can be excluded by probing the indexes, and the fewer operations ("candidates") have to be executed by Index Union Join or Index Union Update Join after the index probing phase.

The probability of executing an operation o after probing a predicate index i which contains o is the selectivity of o with respect to the attribute and type of predicates (equality, range, prefix) indexed by i. The *gain* of index i is then simply the sum of the probabilities of *not* evaluating the operations that i contains. The threshold gain thresh marks the point at which executing operations one-by-one becomes cheaper than any index. The baseline optimizer sets thresh to a hard-coded value (we found $\mathit{thresh} = 2.5$ to work well in most cases). The improved optimizer by Steiner [Ste10] discussed in Section 2.7.2 computes optimal values of thresh at runtime.

Predicate selectivities are estimated as follows. Let d_a be the number of distinct values of some attribute a. The selectivity of an equality predicate on a is estimated simply as $1/d_a$. The selectivity of a range predicate of width w on a is estimated as w/d_a. The selectivity of a string prefix predicate of length l is estimated as $1/26^l$. Obviously, these equations yield crude estimates which could be improved with better statistics, see below.

The rationale for first considering an equality index, than a range index, and then a prefix index is the following. A range index or prefix index (with a trivial extension) can also index equality predicates, so the gain of a range index or prefix index on some attribute a is always at least as large as the gain of an equality index on a. However, equality indexes (jagged array, hash map) are known to be more efficient than range indexes (range tree) and prefix indexes (trie). Considering equality indexes first gives the algorithm some bias toward the more efficient index structures.

Cardinality Estimation

In the baseline optimizer, selectivity estimates are based on an estimate of the cardinality (number of unique values) of each attribute in the table. To this end, Crescando employs a simple yet effective technique know as *linear* or *probabilistic counting*, described by Flajolet and Martin [FM85], and by Whang et al. [Wha90]. The probabilistic counting algorithm is implemented as a side-effect of a *statistics select*, which to the scan threads appears to be a regular, unconditional select operation. The idea is to hash all scanned values into a small bitmap and, after the scan has completed, use the number of 1-bits in the bitmap to calculate an estimate of the number of distinct values.

A special statistics manager thread automatically fires such a statistics select whenever a certain fraction of the table has been updated. To this end, in the merge step (cf. Section 2.3), Crescando looks at the number of records modified by every completed write operation. For each record attribute, a separate modification count is kept, to avoid unnecessary statistics selects. After a certain fraction of record fields for some attribute have been updated (default: 10%), the statistics manager is informed. The Crescando API also allows the user to temporarily disable this automatic behavior (useful for bulk-loading), or manually refresh the statistics at any point.

Limitations

The baseline optimizer quickly finds a small set of indexes which cover all or most of the input operations. However, the resulting execution plan may be sub-optimal for two main reasons.

Superficial Metric The metric *selectivity* does not directly correspond to the cost of plan execution. In particular, the metric ignores the cost of probing the indexes. For example, a hash map on string equality predicates is typically much larger and slower than an otherwise equivalent hash map on integer equality predicates. So given the choice between a hash map on strings with gain g and a hash map on integers with slightly lower gain $g - \epsilon$, the baseline optimizer will choose the hash map on strings, even though it would most likely be better to build the hash map on integers. Also, the heuristic that favors equality indexes over range and prefix indexes is a crude hack to be frank. For example, consider the case where 1,000 operations have a range predicate on a, while only 10 operations have an equality predicate on a. Almost certainly, a single range tree on a that covers all 1,010 operations is more efficient than an (equivalent) pair of a range tree over 1,000 operations plus a hash map over the other 10 operations.

Greedy Plan Enumeration Once the decision has been made to build a specific index, the optimizer puts all unindexed operations with a matching predicate into the new index. For example, assume all input operations have an equality predicate on *classOfService*. The optimizer will likely build a single hash map index on *classOfService* which covers all operations. But some of these operations may have another, much more selective equality predicate on, say, *rloc*. The better plan would have been to put these operations into a separate index on *rloc*. But because plan enumeration is strictly greedy, this alternative is never considered.

2.7.2 Improving Plan Quality

The Master's thesis by Steiner [Ste10] improves upon the baseline optimizer in three respects. It generalizes the simple gain metric of the baseline optimizer into a more accurate cost model, improves the accuracy of selectivity estimates, and explores alternative plan enumeration algorithms.

The cost model of Steiner captures the differences between the different index types. At startup, each possible type of index is constructed, loaded with synthetic data of various distributions, and the runtime cost of probing these indexes is measured, normalized, and later used for plan cost estimations.

To improve the accuracy of selectivity estimates, Steiner uses *histogram sketches*, namely CountMin [CM05] and FastAGMS [CG05]. A histogram sketch is a data structure which supports two operations: count an item, and get an estimate of the frequency of some item. Some types of histogram sketches (not considered by Steiner because of their runtime cost) can even give estimates of the cumulative frequency of a range of items, which is obviously useful for estimating the selectivity of range predicates. What distinguishes a histogram sketch from linear counting, is that the estimated frequency is not a crude, single value for the entire item domain, but is a reasonably accurate, piece-wise approximation of the actual frequency distribution.

Finally, Steiner describes alternative plan enumeration algorithms which seek to avoid or repair bad optimizer decisions. One idea is to use *branch and bound.* Instead of picking *one* index with maximum gain at each iteration of the greedy enumeration algorithm, pick the k best indexes and branch the plan enumeration at this point. To avoid exponential overhead with the number of iterations, all but the n best branches can be pruned at certain intervals (bound step). Another idea is to start out with a plan in which *every* operation is part of some index, and then merge the indexes and redistribute operations among indexes until no significant cost reductions can be achieved anymore.

All these improvements and heuristics have been found to have little impact on the performance of Crescando for the current, real-world workload of the Amadeus Ticket use case. The reason for this somewhat surprising result is that the baseline optimizer already finds near-optimal plans for the current Amadeus Ticket workload. However, the improvements increase throughput by up to 100% for synthetic workloads. We refer the reader to the Master's thesis of Steiner [Ste10] for implementation details and comprehensive experimental results.

2.7.3 Generalizations and Future Work

There are a number of possible generalizations of the optimization framework presented here. First, we have only considered single-attribute indexes at this point. But suppose two or more attributes, for example *productId* (flight number) and *dateOut* (departure date), frequently appear together in the selection predicates of operations. Then it may be beneficial to build a multi-attribute index to improve the selectivity of the index and thereby reduce the plan execution cost. However, adding multi-attribute indexes to Crescando would require new or generalized index structures, and would entail a much larger plan space.

Second, one could allow Clock Scan to activate *arbitrary subsets* of the pending basic reads at the beginning of each scan cycle, instead of just flushing the input queue. This would violate the Monotonic Reads guarantee (cf. Section 2.5), since the reordering of basic reads across scan cycles may lead to a basic read seeing older versions of some records than a subsequent basic read. On the other hand, the added flexibility could for example be used to delay select operations that cannot be indexed effectively until there are enough similar operations pending. Such a reordering optimizer would effectively *gamble*, trading increased latency of some operations in the present for the *possibility* of higher throughput and lower latency of other operations in the future. The resulting optimization problem raises interesting questions in terms of metrics, fairness, and workload prediction.

Finally, we have only considered Index Union Join, Index Union Update Join, and Block Nested Loops Join plans here (the latter is created if the optimizer chooses not to build any indexes). If other types of query-data joins were to be considered, the cost metrics and plan enumeration heuristics would have to be generalized accordingly.

2.8 Logging and Recovery

Crescando features a novel, local recovery scheme based on a combination of *logical logging* and *rolling, fuzzy checkpoints*. When logging is activated, Crescando ensures that, for any write operation, the client application receives a result tuple only after the operation has been reliably logged to disk. Thus, at the point where the client application receives a result tuple, it has the guarantee that the write is *durable*; i.e., its effects would still be visible after a crash of Crescando.

Logical or operation logging, as opposed to physical or value logging, logs the actions of the system (in our case, the insert, update, and delete operations), not the effects of those actions (before-images and after-images of records). Pure logical logging is notoriously difficult to implement in disk-based systems, because it requires that the non-volatile system state is *action consistent* under each failure condition. In other words, the non-volatile state excluding the log is such that each action is either completely done or completely undone [GR92].

As it turns out, action consistency is easy to implement in Crescando. Because a consistent snapshot of a segment is inherently action consistent, a rolling, fuzzy checkpoint formed by a set of individually consistent segment snapshots plus a logical log together form an elegant recovery scheme. Key to performance is that Crescando allows one to read a consistent snapshot of a memory segment at low overhead, without blocking the system.

2.8.1 Logical Logging

When logging is activated, Crescando forwards all incoming write operations (insert, update, and delete operations) to a special log manager thread, which communicates with the rest of the system through a pair of input and output queues, exactly like scan threads (cf. Section 2.3). The log manager serializes and writes out all incoming operations into a logical redo-log on disk.

The log file on disk is a fixed-size, circular buffer, consisting of 4 KB blocks. Big operations (inserts) may span multiple blocks, but a block is the atomic unit of writes to disk. (This results in a small degree of internal fragmentation.) Every time a block is written, it receives a monotonically increasing log sequence number, which allows Crescando to determine the beginning and end of the log after a crash. CRC64 checksums guarantee blocks are written atomically regardless of hardware characteristics, and allows Crescando to detect uninitialized or corrupted blocks.

2.8.2 Rolling Fuzzy Checkpoints

The log manager thread periodically (by default every second) flushes the log to disk. As long as flush periods are shorter than the time it takes to scan the data (which is typically the case), and the log is not full, logging has no impact on write latency. As for log size, we experimentally determined that a few dozen megabytes reserved for the log file are sufficient for a multi-gigabyte data set under any sustainable write rate.

To keep the size of the log file bounded, the log manager has to periodically truncate the log (i.e., allow obsolete blocks to be cyclically overwritten). To this end, the log manager maintains a rolling, fuzzy checkpoint on disk. At all times, the checkpoint file contains a set of timestamped, individually consistent snapshots of each memory segment. These snapshots are obtained through unconditional *snapshot selects* issued by the log manager.

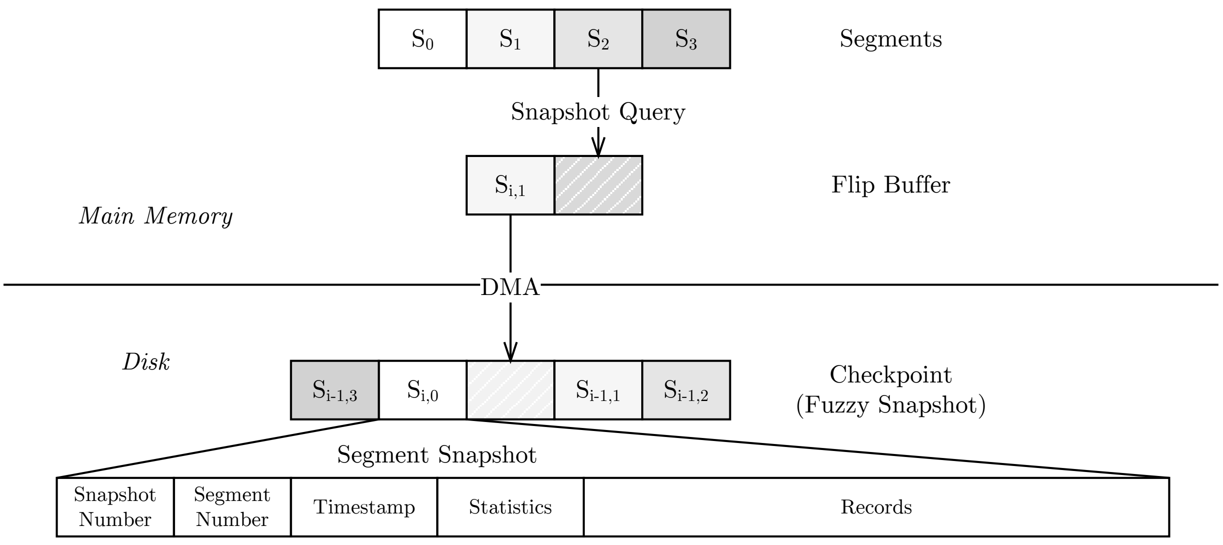


Figure 2.18: *Checkpointing Overview*

Consider Figure 2.18 for an illustration of this checkpointing scheme. By scheduling snapshot selects on segments in a round-robin fashion, $n+1$ segments of disk space are sufficient to always have a checkpoint of n segments of memory. To the scan threads, snapshot selects appear as strict, unconditional select operations. But when executed against a record, a snapshot select does not generate a result tuple. Instead, it copies the record directly to a flip buffer (also called double buffer), which is later used to write the record to disk.

At any given moment, one half of the flip buffer is written to by a snapshot select, while the other half of the flip buffer is read from by the log manager, who writes the contents back to disk. The flip buffer thus ensures that a slow disk does not slow down the scan thread. The log manager writes the checkpoint file in large chunks (1 MB by default), using the `O_DIRECT` flag. This bypasses the operating system's file buffer for minimum CPU and memory overhead [RG10].

2.8.3 Recovery Procedure

After a crash, recovery proceeds in three stages:

1. **Restore Checkpoint** Crescando inspects the headers of all segment snapshots in the checkpoint file. The segment for which there are two snapshots is obviously the one at which the system crashed earlier. Out of the two snapshots of that segment, the one with the higher snapshot number has been written to at the time of the crash, so it is most likely incomplete and is discarded. All the other segment snapshots in the checkpoint file are guaranteed to be complete and are restored to memory.

2. **Read Log** The log file is scanned, checking the log sequence numbers and checksums of scanned blocks to determine the beginning and end of the log. Then, every logged operation whose timestamp is higher than that of any restored segment snapshot is deserialized.

3. **Replay Log** The scan threads are started and the deserialized operations are enqueued, in original input order. For each segment, the operations which are already reflected in the restored snapshot are skipped (easy to do by comparing the segment and operation timestamps). The log is replayed in parallel by all scan threads, using the normal execution logic (Clock Scan, Index Union Join, etc.).

At the end of log replay, all segments are in a consistent state which reflects the effects of all operations ever logged. This is a prefix of the stream of operations submitted by the user, and includes every operation for which the user has ever received a result.

2.8.4 Discussion

Local recovery is mainly intended for single-machine deployments of Crescando. The distributed system presented in this dissertation, Crescando/RB, relies purely on replication for fault-tolerance. In the future however, the local log and checkpoint may be used to optimize live migration (transferring engine state to a remote host, see Sections 4.4.1 and 5.4), and also for disaster tolerance (e.g., general power outage).

The interval at which a complete checkpoint is written to disk is the time it takes to snapshot and write out all records. For example, assuming a single low-cost disk with a bandwidth of 100 MB/sec, and assuming each scan thread scans its segment faster than 100 MB/sec (holds for realistic workloads), a 10 GB data set would be checkpointed every 100 seconds. At any point, the log would therefore contain the write operations of the last 100 seconds.

Log replay is performed in parallel by enqueuing the logged operations to the respective scan threads through the usual interface. The time required for log replay is thus determined by the segment which replays the most operations, which is typically the segment with the oldest snapshot on disk. The set of operations to replay on each segment is a subset of the write operations which arrived during a checkpoint interval. Thus, assuming the write load at the time of failure could be sustained by the system, log replay requires at most one checkpoint interval (100 seconds in the example).

In summary, recovery will take between one and two checkpoint intervals (between 100 and 200 seconds in the example)—one interval for restoring the checkpoint, and up to one interval for log replay, assuming the time to read the log is negligible. The experimental results in Section 2.10.7 match this analysis.

Note that it is possible to trade some log replay time (and thus availability) for average-case performance by writing checkpoints less frequently. The resulting maximum recovery time is easy to derive.

As an optimization, the three recovery steps could be overlapped in time. The idea is to restore the checkpoint starting with the oldest segment snapshot, then immediately start log replay on the restored segment while the next older segment snapshot is read from disk, and so forth. Since there are more log entries to replay for old snapshots, this should decrease the total recovery time compared to strictly sequential execution of the three steps. We have not implemented this optimization at this point, but it would be easy to add.

Also, in the future, the heavy-weight segment snapshots performed by snapshot selects could be replaced by incremental snapshots to better amortize the overhead. Suitable algorithms can be found in the literature, for example in Cao et al. [CVSS$^+$11].

2.9 String Compression

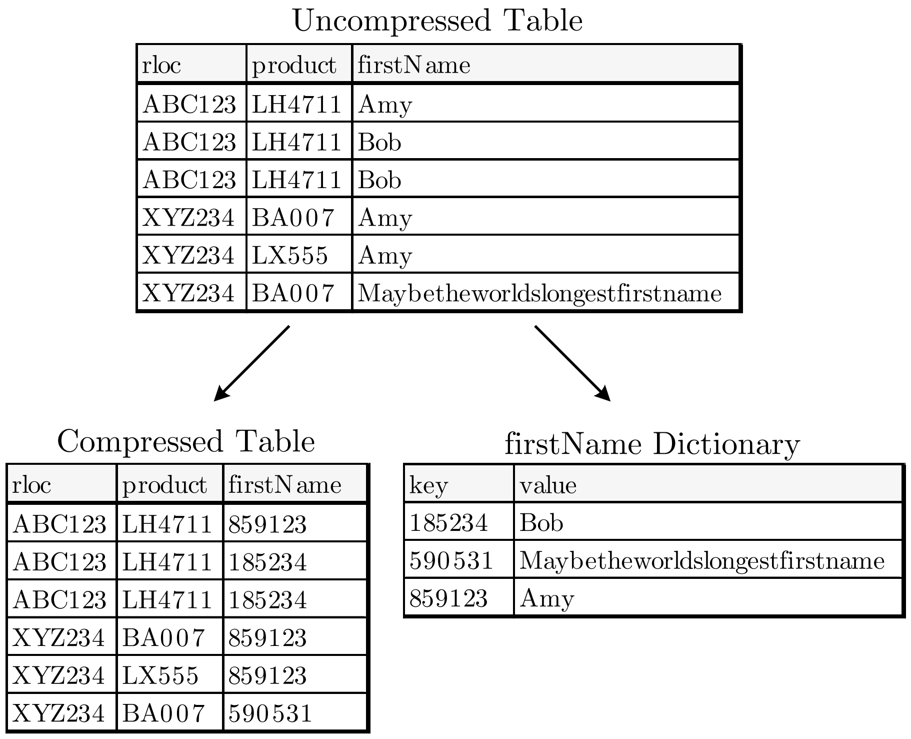
In the original design of Crescando described in Unterbrunner et al. [UGAK09], string attributes were stored as fixed-length array fields. For string attributes with a high maximum length but a low average length, such as a passenger's first name, this resulted in a considerable amount of memory wasted on padding.

In response to this issue, a Master's thesis by Bernet [Ber10] added *dictionary compression* to Crescando. Dictionary compression is based on a bidirectional mapping between string values and integer keys. Every unique string value is mapped to a unique integer key and vice versa. See Figure 2.19 for an illustrative example. By replacing the padded string field with a *significantly smaller* integer field, memory space can be saved. The fewer distinct strings there are relative to the number of records, the higher the space savings become.

2.9.1 Design

Dictionary compression in Crescando follows three design principles:

Coherence Compression adheres to the original design principles of Crescando, in particular the design for predictability and the design for scalability (shared-nothing).



Uncompressed Table

rloc	product	firstName
ABC123	LH4711	Amy
ABC123	LH4711	Bob
ABC123	LH4711	Bob
XYZ234	BA007	Amy
XYZ234	LX555	Amy
XYZ234	BA007	Maybetheworldslongestfirstname

Compressed Table

rloc	product	firstName
ABC123	LH4711	859123
ABC123	LH4711	185234
ABC123	LH4711	185234
XYZ234	BA007	859123
XYZ234	LX555	859123
XYZ234	BA007	590531

firstName Dictionary

key	value
185234	Bob
590531	Maybetheworldslongestfirstname
859123	Amy

Figure 2.19: *Dictionary Compression Example*

Transparency Compression is transparent to the client and to high-level components of the engine, such as the log manager.

No Regression Compression does not negatively impact performance.

In accordance with the first two design principles, dictionary compression is implemented at the level of scan threads. For every compressed string attribute in the schema, each scan thread maintains its own, completely local compression dictionary. Records and selection predicates of operations are transparently compressed and decompressed at the scan thread boundaries. This design leads to redundancies in dictionaries across threads and thus sub-optimal compression ratios in comparison with an alternative, global dictionary. But more importantly, it avoids any shared data structures which would threaten the predictability and scalability of Crescando.

Regarding the *no regression* principle, our experimental results in Section 2.10.8 show that dictionary compression in Crescando not only saves space, it even *accelerates* query processing. To see how, assume a select operation $\langle \text{firstName} = \text{'Bob'}, * \rangle$ needs to be executed, and *firstName* is a compressed string attribute. The naïve solution is to decompress every single record that is scanned, and perform string comparison of "Bob" and the decompressed *firstName* field. A better solution is to *compress the predicate value* “Bob”, and then perform *integer comparison* directly on the compressed *firstName* field of each scanned record. This idea has first been systematically investigated by Ray et al. [RHS95] and has since made its way into various systems.

Using *order-preserving compression*, also range predicates can be compressed and evaluated directly on compressed data [Ant97]. Since prefix predicates can be converted into range predicates (cf. Section 2.6.3 on tries), even prefix predicates can be compressed. The Master’s thesis of Bernet [Ber10] mentions order-preserving compression as future work, but the feature has since been implemented by Bernet and the author of this dissertation.

In summary, all types of predicates over strings available in Crescando can be directly evaluated on compressed data. The experimental results presented in Section 2.10.8 show that predicates on compressed string attributes are in fact as cheap as predicates on plain integer attributes. This is more than a performance bonus; it means dictionary compression made Crescando more *robust* to strings in the schema and workload.

2.9.2 Data Structures

A compression dictionary must support two basic operations: map a string to an integer key (compression), and map an integer key to a string (decompression). The straightforward way to implement these operations is to instantiate two map data structures—one for compression, one for decompression. However, this solution ignores the fact that the two operations are not equally important.

Compression is only used when inserting records or compressing selection predicates. In contrast, decompression is used much more frequently. Decompression becomes necessary whenever a select operation projects a compressed string attribute into a result tuple, or whenever a selection predicate cannot operate on compressed data (this happens when a range predicate is evaluated, but the compression scheme is not order-preserving). Because of this asymmetry, it makes sense to trade some compression performance for space efficiency.

Bidi-map

Bernet [Ber10] introduces a novel type of compression dictionary called *Bidi-map* which trades compression performance for space efficiency. A Bidi-map supports both compression and decompression, but fully indexes the data only in one direction: decompression. In the other direction, the index is more coarse-grained.

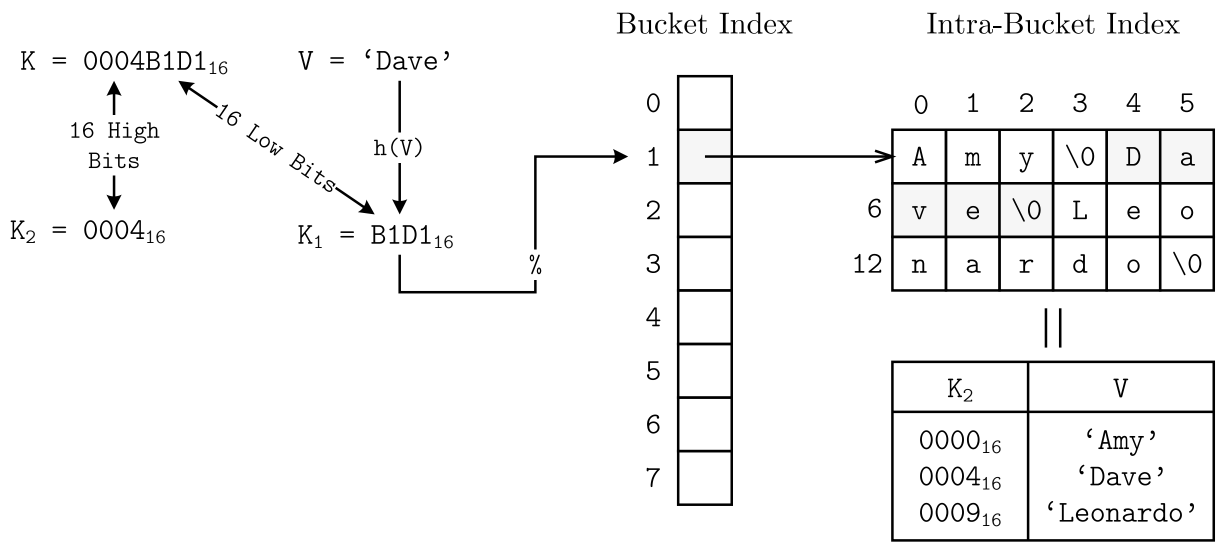


Figure 2.20: *Bidi-map with Direct-Mapping Buckets*

Assume the string value $V =$ 'Dave' maps to the integer key $K = 0004\text{B1D1}_{16}$ (hexadecimal). Decompression works like this. The 16 low bits $K_1 = \text{B1D1}_{16}$ serve as a key into a primary index called *bucket index*. To find the bucket that belongs to K_1, one simply takes the remainder (modulo) of dividing B1D1_{16} by the number of buckets, analogous to open hashing. Each bucket itself is again an index, called *intra-bucket index*. Suppose $K_1 = \text{B1D1}_{16}$ points to the bucket B in the bucket index. Then the 16 high bits $K_2 = 0004_{16}$ point to the value V in the intra-bucket index B. Various bucket implementations with different space and performance characteristics exist.

Conversely, compression works as follows. A hash function h maps the string value V to $K_1 = h(V)$, the 16 low bits of the complete key K. As in decompression, K_1 is used as a key into the bucket index. To find K_2 and complete the key K, *string search* over the bucket contents is required.

Figure 2.20 illustrates a Bidi-map with direct-mapping buckets. These types of buckets use K_2 as a direct offset to the beginning of the string value. Decompression therefore requires constant time $O(1)$.

To find K_2 for a given string that is being compressed, every string in the bucket is compared until a match is found. Thus, compression requires $O(nm)$ time, where n is the bucket size, and m is the length of the lookup string. More efficient algorithms for string search are known [CLRS01], but are not implemented at this point. It is doubtful that there is much to gain, given that buckets typically contain just a few entries.

Bernet [Ber10] proposes a variety of other bucket implementations. Some use K_2 as a direct offset (as described here), others use linear search over an array of ⟨key, pointer-to-value⟩ pairs. Some use reference counting to garbage-collect entries where all instances have been deleted from the table. Others use periodic mark-and-sweep runs, and so forth. These different implementations mark different space and performance trade-offs. For detailed descriptions and performance results, see Bernet [Ber10].

Order-preserving Compression

Bidi-maps are a flexible data structure, but they currently do not support *order-preserving compression* due to the use of hashing to map string values to (parts of) their keys. In the future, order-preserving compression could be added to Bidi-maps by replacing the (non-order preserving) hash function with an order-preserving mapping, for example a compact trie structure as proposed by Binnig et al. [BHF09b].

At the time of writing, order-preserving compression is implemented in a simpler way, through an ordered array of key-value pairs, with gaps left for insertions. To map a given key to its value, binary search over the array is performed. When a new value is being inserted and no gap is found in the array, the dictionary has to be (partially) rebuilt[8].

For example, assume the dictionary contains the entries $\langle 3, \text{'Ben'} \rangle$ and $\langle 4, \text{'Brian'} \rangle$ and we want to add a new entry for 'Bob'. Obviously 'Ben' < 'Bob' < 'Brian' in lexicographical order, so 'Bob' must receive a key between 3 and 4. No such (integer) key exists, so one needs to re-assign some of the keys to make room for 'Bob'. For example, 'Brian' could be mapped to 5 (assuming 5 is free).

Unfortunately, not just the dictionary must be updated, but also all the records that contain 'Brian'. To update these records, a special reorganization operation is injected into the scan. In this example, the reorganization operation selects for 'Brian' and changes the key field in every matching record from 4 to 5. As shown in Section 2.10.8, the overhead of these reorganization operations is tolerable.

Hot Dictionaries and Associative Caching

Most real data sets and workloads have significant skew with respect to the attribute value distributions in records and selection predicates. In the Amadeus Ticket use case for example, there are over 40,000 ticket-issuing agents (the string attribute *office* in the schema), but a tiny fraction of these agents (the airlines) issue over 90% of the tickets [Ber10]. We call the corresponding office entries *hot*, as opposed to other, *cold* entries.

[8] There are, in fact, order-preserving compression schemes which use variable-length encoding for keys to be able to dynamically modify the key domain, thereby avoiding re-assignment of keys to values. This approach comes with (performance) problems of its own however [HLAM06].

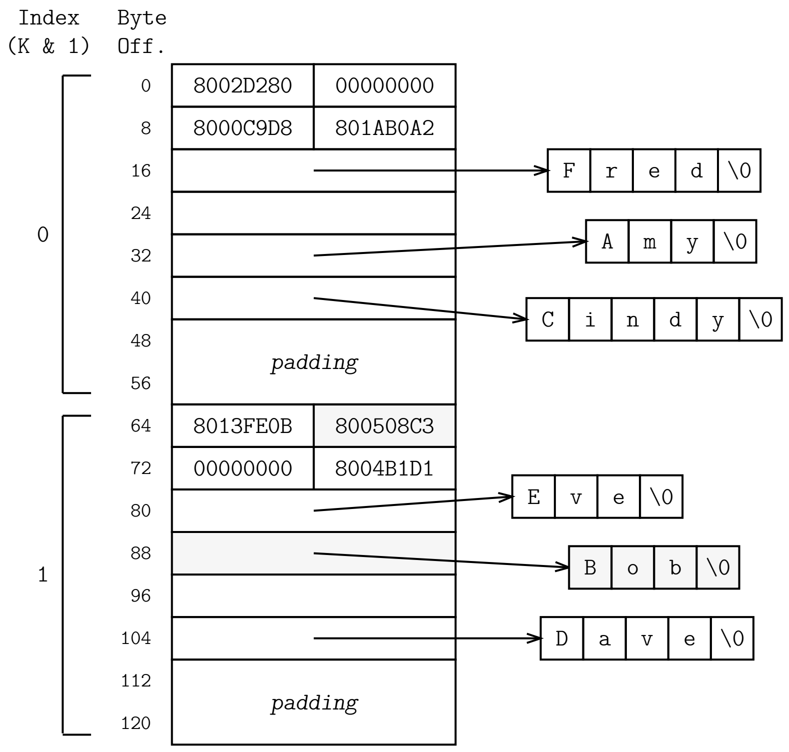


Figure 2.21: *Example Associative Cache (Hot Dictionary)*

Let us distinguish between hot and cold entries by using the highest bit of any 32-bit integer key as a *hint* whether the key refers to a hot or cold entry. That is, the hot key $800508C3_{16}$ maps to the same string as the cold key $000508C3_{16}$.

Crescando stores hot entries in a special, hot dictionary structure that acts as an inclusive cache for the regular, cold dictionary. The cold dictionary is either a Bidi-map or an order-preserving array of key-value pairs, as described previously. For any key that is marked hot, the hot dictionary is searched, otherwise the cold dictionary is searched.

The hot dictionary is implemented as an *associative cache*, in software. Consider Figure 2.21 for an example of this data structure. In the example, the hot dictionary contains 6 entries, split into 2 disjunct sets, also called indexes. Keys with the low-bit set to 0 map to the first index, and keys with the low-bit set to 1 map to the second index. Like associative caches in hardware, the scheme generalizes to larger numbers of indexes by masking additional bits.

Assume we are trying to decompress the (hot) key $800508C3_{16}$. The low-bit is set, so we look into the second index. Just like in the trie implementation described in Section 2.6.3, the 4 key slots are loaded into a SIMD register, and compared to the lookup key with a single SIMD instruction (packed 32-bit comparison). The result is the bit mask 0100_2

(binary). The index of the single 1-bit is computed and used as an offset into the array of pointers, which leads to the null-terminated string 'Bob'.

The pointers point directly to the values stored in the cold dictionary, so no memory space is wasted on duplicate strings. Each index of the hot dictionary is only 48 bytes in size, so it easily fits into a single cache line on modern CPUs. In short, there is *no branching* in the code, and only a *single cache line* is ever loaded to map a key to a pointer. (Following the pointer and accessing the string value may obviously require more cache lines to be loaded.)

The hot dictionary implementation is 4-way associative at the time of writing, because the largest available registers on x86 machines with SSE2 extensions fit 128-bit, that is 4 * 32-bit keys. The latest x86 processors with Advanced Vector Extensions (AVX) feature 256-bit registers [Int11], so 8-way associative hot dictionaries could be implemented once hardware support is more widespread.

But like in associative caching in hardware, not every entry that is hot can be cached. The best one can do given an n-way associative hot dictionary, is to insert the n most frequently used keys of every index. The frequency distribution may change, so Crescando occasionally re-organizes the hot dictionary. Like in the case of optimizer statistics (cf. Section 2.7), re-organization is automatically triggered after a significant fraction of the data has been updated. First the hot dictionary is rebuilt, using the reference counts in the cold dictionary that are also used for garbage collection of unused entries. Then, a special update operation is injected into the scan, which sets the high bits in the keys of compressed records in accordance with the new contents of the hot dictionary.

For realistic workloads, we found that hot dictionaries on top of Bidi-maps increased throughput by 5% to 15% (depending on the hardware) for compression-heavy workloads. This may sound low, but Bidi-maps are already highly optimized themselves. Furthermore, the idea of an associative cache in software that uses SIMD instructions is not limited to dictionary compression in Crescando. Associative caches can be used in any system, either by themselves, or to accelerate any type of dictionary structure. In a different context, the gains achieved with associative caches may be more significant.

2.10 Experimental Evaluation

2.10.1 Platform and Configuration

All experiments reported in this chapter have been conducted on a 16-core machine built from 4 quad-core AMD Opteron 8354 ("Barcelona") processors with 32 GB of DDR2 667

RAM. Each core was clocked at 2.2 GHz and had 64 KB + 64 KB (data + instruction) of exclusive L1 cache, and 512 KB of exclusive L2 cache. The machine was running a 64-bit Linux SMP kernel, version 2.6.27.

Unless otherwise indicated, Crescando was loaded with 15 GB data, round-robin partitioned over 15 scan threads, each running Clock Scan and the baseline multi-query optimizer. Logging and string compression was deactivated, except in the relevant experiments in Sections 2.10.7 and 2.10.8. All data points were averaged over 5 runs of at least 5 minutes each. By *read latency* and *write latency* we refer to the time from an operation being generated by the benchmark driver, to the benchmark driver receiving the complete set of result tuples.

The experiments in this chapter use only *basic* reads, not strict reads, because at the time the experiments were performed, Crescando did not yet support strict reads. We have since learned that strict reads are up to 30% more expensive than basic reads (the exact difference depends on the selectivity, hardware, and concurrent write load). Other than that, Crescando behaves the same. In Section 5.6, we show experimental results for Crescando/RB which include strict reads.

2.10.2 Data and Workload

Amadeus kindly provided us with a 4 GB anonymized sample of live Ticket data, as well as a complete trace of all SQL statements executed against the Ticket table over a time period of 6 months. The data sample was analyzed by Giannikis [Gia09] to extract cardinality distributions of the various attributes, which were then fed into a stochastic data generator. Likewise, the SQL the trace was analyzed and fed into a stochastic workload generator, which would create Crescando operations whose selection predicate attribute and projection attribute distributions closely resembled the live SQL workload.

The Amadeus Ticket data and workload, as well as our data and workload generators are described in detail in the Master's thesis of Giannikis [Gia09].

Amadeus Workload

On average, each select operation in the Amadeus workload has 8.5 distinct predicate attributes and projects out 27 of the 47 attributes. 99.5% of the select operations have equality predicates on both *productId* (flight number) and departure date. Consequently, most select operations are highly selective, matching only a few of the multi-million records. The 0.5% of "odd" select operations had been enough to cause severe problems in the relational database management system used before Crescando became available to Amadeus.

Algorithm 9: Pick Attribute Set

```
Input: N, D, s
Output: P
$Z \leftarrow$ Zipf($s$, $N$);                    /* initialize Z as Zipf distribution */
$V \sim$ B($N$, $D/N$);        /* get random V according to binomial distribution B */
for $v$ = 1 to $V$ do
    $a \sim Z$ ;                                  /* get random a according to Z */
    $p \leftarrow Z[a]$;                      /* get probability p of a according to Z */
    $Z[a] \leftarrow 0$ ;                                    /* remove a from Z */
    $Z \leftarrow Z/(1-p)$ ;                          /* re-normalize remaining Z */
    $P \leftarrow P \cup a$ ;                              /* add a to result set P */
```

The overwhelming majority of update operations have an equality predicate on *rloc* (booking number), so they typically affect just a few records. Insert and delete operations occur at about 1/5th the rate of update operations. Delete operations typically affect a specific *rloc* just like update operations, but there is a small fraction of *bulk deletes* with range predicates on booking and departure dates.

For a more detailed description of the Amadeus workload, please refer to the Master's thesis of Giannikis [Gia09].

Synthetic Workload

As motivated in the introduction, we are not only interested in the performance under the current (index-aware) Amadeus workload, but also in the performance under future, increasingly diverse and unpredictable workloads. In other words, we are interested in workloads where users ask *any query they want*, blissfully ignorant of the data's physical organization.

We created such synthetic workloads as follows. The predicate attribute distribution of queries in the Amadeus workload was replaced by *parametrically skewed* predicate attribute distributions. (The distributions of update, insert, and delete operations are known to be stable over time and were therefore not modified.) To generate a set of selection predicate attributes for use in experiments, Algorithm 9 was invoked.

First, the algorithm decides how many attributes to pick, according to a binomial distribution B defined by the total attributes in the schema, N, and the desired average number of attributes, D. At each of the following iterations of the for-loop, the algorithm picks a random attribute according to some probability distribution Z, which is initially a Zipf

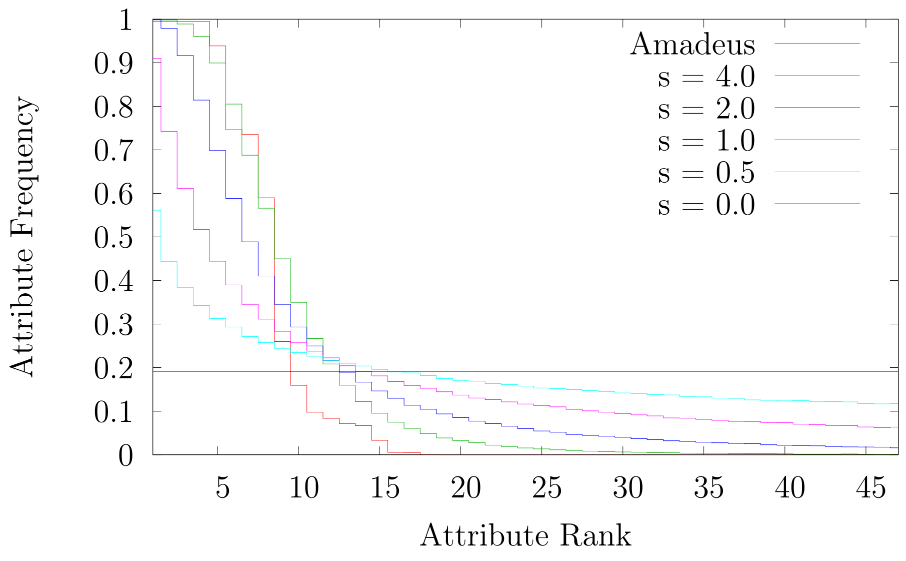


Figure 2.22: *Synthetic Predicate Attribute Distributions: $N = 47$, $D = 9$, Vary Workload Skew (s)*

distribution with a user-defined characteristic exponent s. After an attribute a has been picked, a is removed from Z, and Z is re-normalized.

Repeatedly executing the algorithm with $N = 47$ (number of attributes in the schema), $D = 9$ (desired average number of predicates per query), and different shape parameters s yields the attribute frequency distributions shown in Figure 2.22. Notice that the area under each synthetic distribution is $D = 9$, which means the algorithm picked 9 attributes on average, as requested.

For comparison, the figure also includes the attribute frequency distribution of query predicates in the live Amadeus workload. The initial Zipf distribution of Algorithm 9 retains the frequency ranking of attributes in the Amadeus workload. The synthetic workloads thus represent evolutionary steps, from a relatively close approximation of the Amadeus workload ($s = 4.0$) to a uniform distribution of predicate attributes ($s = 0.0$).

2.10.3 Crescando vs. MySQL

Crescando is designed for predictable performance under unpredictable workloads. In more concrete terms, Crescando is designed to degrade gracefully under mixed (read-write) workloads, workloads for which data indexes are not economical or feasible, and workloads which combine these properties.

To put the performance and behavior of Crescando in this respect into perspective, we compared it to a popular, traditional solution: MySQL 5.1 and its *HEAP* (main memory) table engine. Clearly, this is not an apples-to-apples comparison, because MySQL offers many more features than Crescando. More interesting than absolute numbers is therefore the *shape* of the performance curves. Other existing solutions may reach significantly better performance for specific workloads; e.g., Vertica, MonetDB, or SAP T-Rex for analytic, read-mostly workloads. These systems, however, are not well-suited for the write-heavy workloads found in the use cases we are interested in.

Among main-memory databases, we believe the MySQL HEAP engine is a reasonable representative for comparison, as most main-memory databases are also based on locking. There are a few commercial main-memory databases which feature optimistic concurrency control (IBM solidDB, IBM ObjectGrid). However, optimistic concurrency control does not avoid conflicts, particularly between full-table scans and concurrent updates, leaving little reason to believe the results would differ significantly.

Read-write contention could be reduced by multi-version concurrency control (MVCC) schemes, such as snapshot isolation. Unfortunately, no main-memory based database system features MVCC to the best of our knowledge. We experimented with a commercial disk-based database system which implements snapshot isolation. But even when hand-tuning this system to our needs, and even when the whole database fit into main memory, the commercial system showed an order of magnitude lower throughput than MySQL HEAP for read-only, index-hitting workloads. We found that the disk-based database paid a steep price for needlessly copying data around in main memory and decided that this precluded a meaningful comparison.

We configured MySQL as follows: an index on ⟨*productId*, *dateOut*⟩ (flight number, departure date), plus an index on *rloc*. These are the indexes used by Amadeus for their live system. Logging was disabled. We spawned 16 client threads, of which 15 were pure readers and 1 was a pure writer. This configuration was found to yield maximum throughput on our 16-core machine. Additional indexes or threads did not improve performance.

Throughput under Mixed Load

In the first experiment, we revisit the diagram sketch shown in the introduction to this chapter, Figure 2.1 on page 12. It was claimed that Crescando degrades more gracefully under mixed (read-write) and diverse (hard to index) workloads than traditional solutions.

To verify the claim, we first measured the maximum read throughput of both systems under a variable write load. As Figure 2.23 reveals, MySQL's throughput came close to

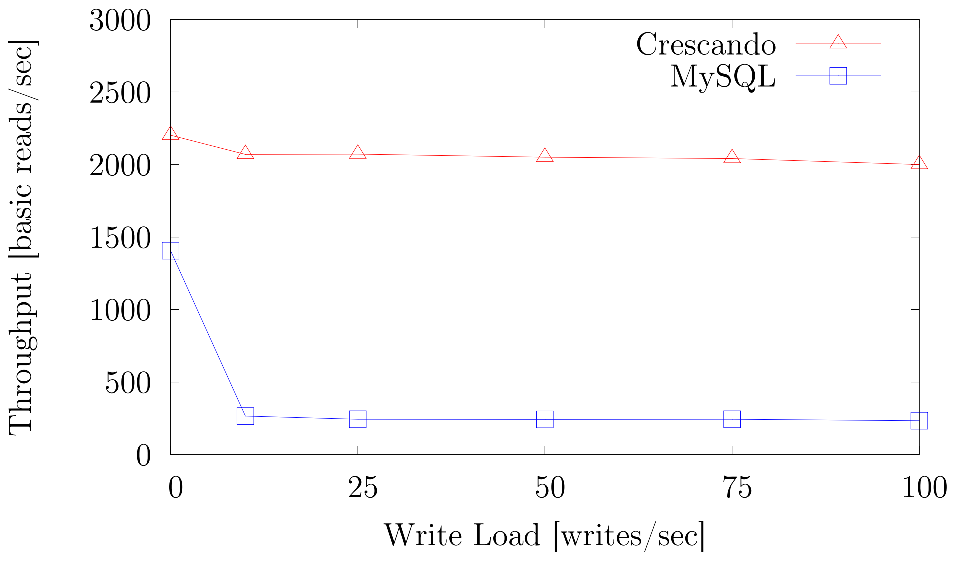


Figure 2.23: *Crescando vs. MySQL Throughput: Amadeus Mixed Load, Vary Writes/Sec*

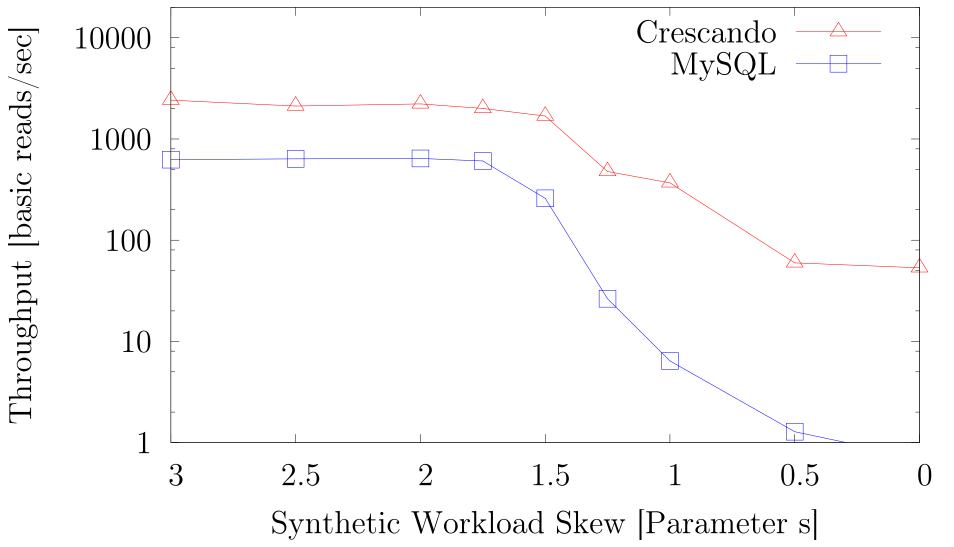


Figure 2.24: *Crescando vs. MySQL Throughput: Synthetic Read-Only Load, Vary Workload Skew (s)*

Crescando only for a read-only workload. As soon as writes were added to the workload mix, throughput in MySQL dropped sharply, presumably due to lock conflicts on the indexes and records. In contrast, there was no performance cliff in Crescando, since there were no locks or lock conflicts to handle. In absolute numbers, Crescando out-performed MySQL by one order of magnitude for mixed Amadeus workloads.

Throughput under Increasing Query Diversity

Next, we measured the maximum throughputs of Crescando and MySQL for synthetic read-only workloads of variable query diversity; i.e., variable predicate attribute skew s. The results are shown in Figure 2.24. For all synthetic workloads, Crescando outperformed MySQL by at least one order of magnitude. The gap widened to over two orders of magnitude for workloads with low skew.

Latency under Mixed Load

We compared the ability of Crescando and MySQL to comply with latency requirements under an increasing write load. First, we measured read and write latency for a mixed Amadeus workload with 100 reads/sec and a variable write rate between 0 and 100 writes/sec. The results are shown in Figure 2.25.

For *read-only* workloads, read latencies of MySQL were superior (1/13/117 msec for the 50th/90th/99th percentile respectively). But with a *single* write in the workload mix, MySQL latency surged to 5,000/13,600/15,800 msec for reads, and 7,800/24,800/31,550 msec for writes. We analyzed the Amadeus workload and found that only one out of 3,500 reads did not hit the indexes in MySQL. The small number of full-table scans caused by these reads sufficed to cause massive read-write contention and operation queuing in MySQL.

Incidentally, the 16 seconds which are the 99th percentile of read latency and half of the 99th percentile of write latency are roughly the time it took MySQL to perform a full-table scan. The decreasing median for increasing write rates is an artifact of "batch processing" of writes due to MySQL's read-write lock policy.

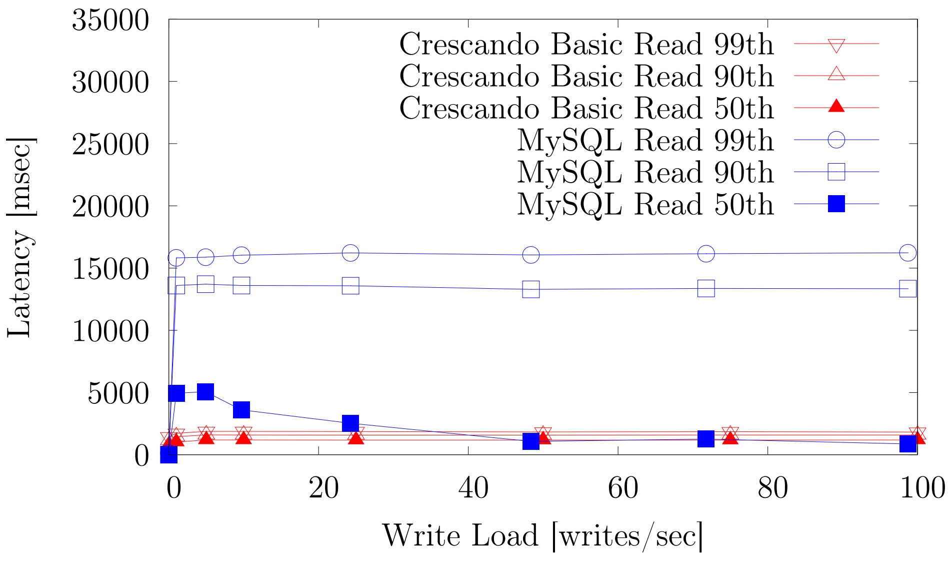
In comparison, the latencies measured for Crescando were quasi constant. Regardless of the write load, read and write latencies stayed below 2,150 msec.

Latency under Increasing Query Diversity

Next, we injected a synthetic read-only load of a constant 100 reads/sec, and varied the predicate attribute skew s between 2.5 and 1.25. Figure 2.26 shows the results.

Latency for MySQL was superior for $s \geq 1.5$ but surged for $s < 1.5$. Note that at $s = 1.25$, over 98% of the queries still hit the indexes in MySQL. Again, in comparison, Crescando's performance degraded much more gracefully.

In summary, throughput of Crescando not only exceeds that of MySQL for the Amadeus workload and all synthetic workloads, throughput in Crescando also degraded much more



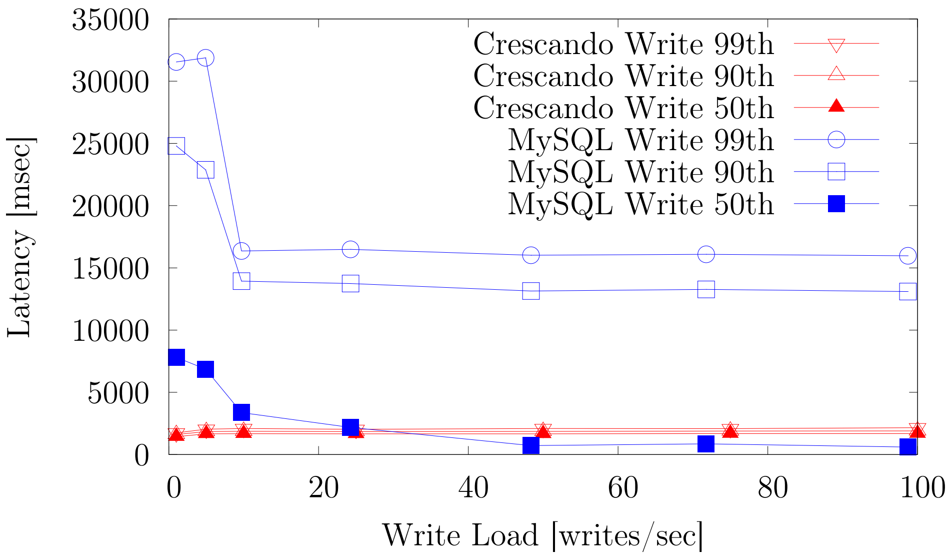


Figure 2.25: *Crescando vs. MySQL Latency: Amadeus Mixed Load, 100 Reads/Sec, Vary Writes/Sec*

gracefully with the write load and with query diversity. Likewise read and write latencies of Crescando are significantly more robust to mixed workloads and query diversity than those of MySQL.

The cross-over points will obviously change when comparing Crescando to other traditional systems, but the message remains. Crescando is significantly more robust to unpredictable, mixed workloads, and promises higher throughput and lower latency when full-table scans cannot be avoided.

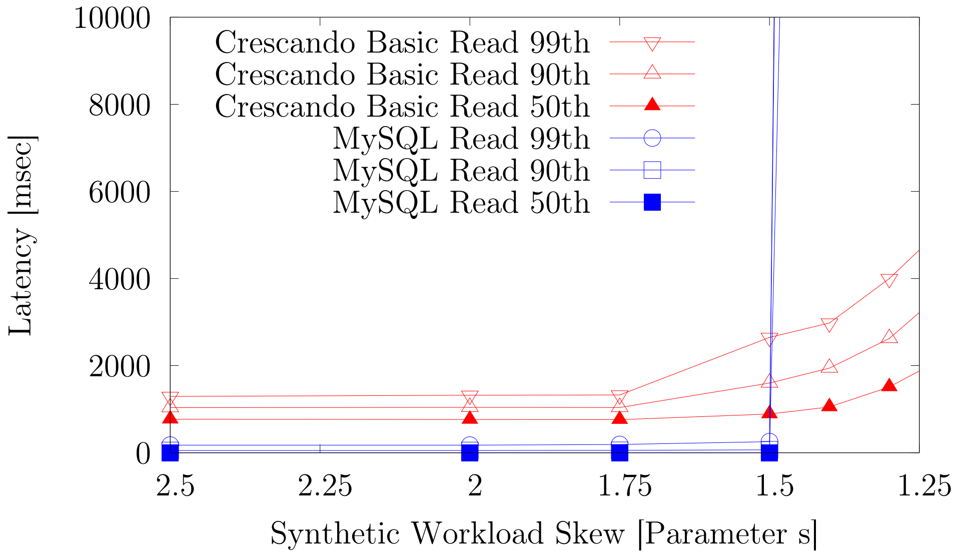


Figure 2.26: *Crescando vs. MySQL Read Latency: Synthetic Read-only Load, 100 Reads/Sec, Vary Workload Skew (s)*

2.10.4 Multi-core Scale-up

Crescando is designed to scale linearly on modern multi-core platforms. To validate this property, we saturated Crescando with a read-only variant of the Amadeus workload. We configured Crescando for *round-robin* partitioning, which means that every scan thread had to process every operation. The data size was fixed to 15 GB, so in the case of only one scan thread, that scan thread had to scan 15 GB, whereas in the case of e.g. 5 scan threads, each scan thread had to scan only 3 GB.

We limited the number of operations to share an Elevator Scan cursor to 32, and to share a Clock Scan cursor to 512. This gave a median query latency of around 1 second for both algorithms, for a fair comparison.

The peak throughputs using Classic, Elevator, and Clock Scan are shown in Figure 2.27. Classic Scan scaled from 0.2 reads/sec for 1 thread to 1.9 reads/sec for 15 threads (9.5x scale-up). In comparison, Elevator Scan scaled from 0.8 reads/sec to 10.5 reads/sec (13.125x scale-up). Elevator Scan scaled better because more computation was done per scanned record, making the algorithm less memory-bound than Classic Scan. The speed-up by a factor of 4 to 5 when comparing Classic Scan to Elevator Scan mirrors the results reported in related work on shared scans in main memory [RSQ+08, QRR+08].

Still, the throughput of Elevator Scan was minuscule in comparison with Clock Scan, which scaled from 42.3 to 558.5 reads/sec (13.2x scale-up). This is over two orders of

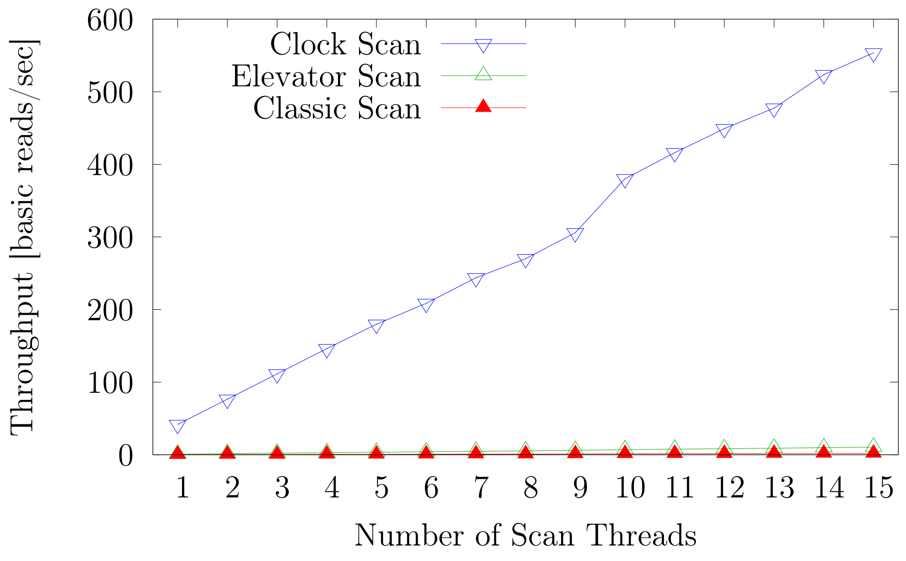


Figure 2.27: *Multi-core Scale-up: Amadeus Read-only Load, Round-robin Partitioning, Vary Scan Threads*

magnitude higher than the throughput of Classic Scan. What is more, the linearity of the curve demonstrates that Clock Scan is *compute bound* on modern NUMA machines. The bump in throughput between 9 and 10 threads is due to the fact that for 10 threads the segments had become small enough to be allocated on the local NUMA node, giving minimum access distance and maximum performance.

The high performance of Clock Scan compared to Elevator Scan is explained by a large fraction of queries having equality predicates on flight number. The multi-query optimizer automatically built an efficient hash map on those predicates.

Crescando is a system designed for good worst-case behavior. In this spirit, we have only shown results for round-robin partitioning here. However, Crescando also supports hash partitioning. Round-robin partitioning is equivalent to hash partitioning where no incoming operation has an equality predicate on the partitioning key, so the results for round-robin partitioning are really the *minimum scale-up* for any choice of hash partitioning attribute, assuming no significant data skew. For perfectly partitionable workloads (every operation has an equality predicate on the partitioning key) scale-up using hash partitioning is *quadratic*.

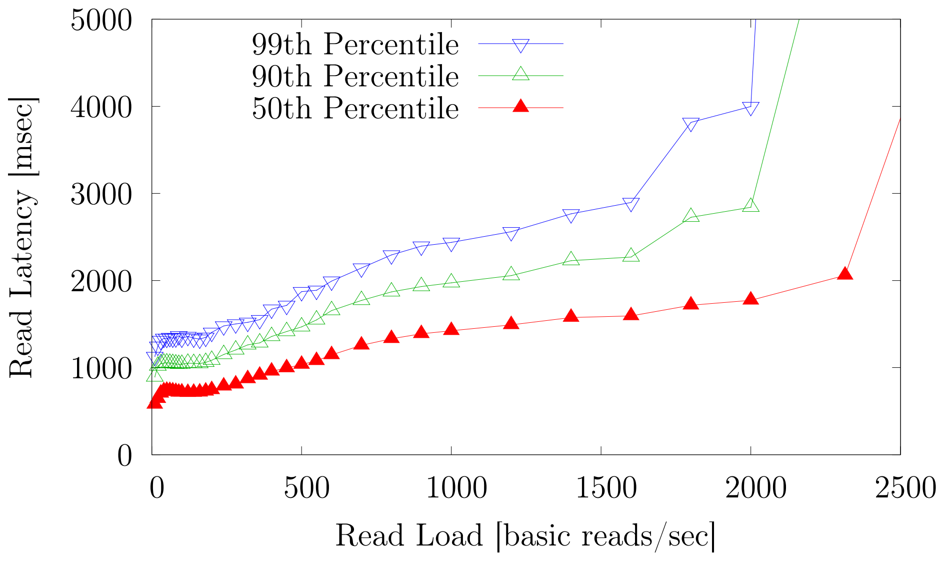


Figure 2.28: *Read Latency: Amadeus Read-only Load, Hash Partitioning, Vary Reads/Sec*

2.10.5 Latency vs. Read Load

In the next experiment, we investigated the robustness of Crescando to bursts in query volume. For this purpose, an increasing load of Amadeus reads was created as we were measuring the 50th, 90th, and 99th percentile of read latency. Figure 2.28 shows the results. Notice that the throughput numbers that Crescando sustains are higher than in the scale-up experiment, because the Crescando configuration was changed to hash partitioning on *productId* (flight number).

As for latency, one can see that it was very much constant up to about 200 reads/sec. We found that this is the point where the working set of Clock Scan (indexes and less-selective queries) exceeded the L1 cache. Between 200 and about 2,000 reads/sec, latency was *logarithmic* in the number of reads, owing to the predicate indexes used by Index Union Join. Beyond 2,000 reads/sec, latency increased sharply. We profiled the cache behavior of the system and found that 2,000 reads/sec was the point where the working set of scan threads began to exceed the L2 cache. At about 2,500 reads/sec, the system reached saturation.

In summary, there is a markedly sub-linear correspondence of load to latency almost up to the point of system saturation. This is a major feature of Crescando, because it means the system is highly robust to load bursts, as well as to load imbalance across (distributed) instances of Crescando and individual scan threads.

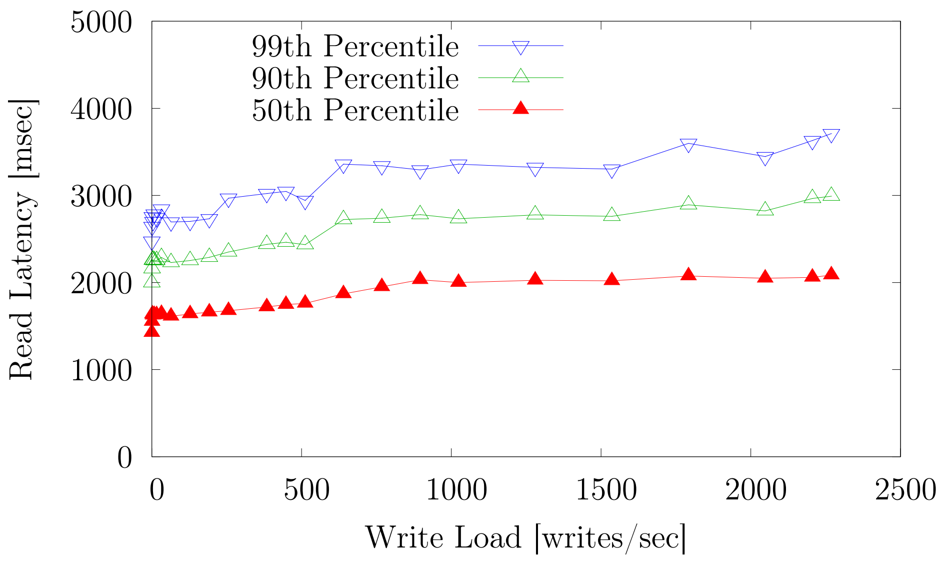


Figure 2.29: *Read Latency: Amadeus Mixed Load, Hash Partitioning, 1,000 Reads/Sec, Vary Writes/Sec*

2.10.6 Latency vs. Write Load

In this experiment, we tested the robustness of Clock Scan to concurrent writes. We pushed a constant 1,000 Amadeus reads per second into the system, and gradually increased the write load to the point where the system became saturated with writes, while measuring the 50th, 90th, and 99th percentile of read latency.

As Figure 2.29 shows, read latency increased by about 35% between 0 and 1,000 writes/sec. Beyond this point, latency hardly increased any further. In the Amadeus workload, most writes have an equality predicate on *rloc*, not on *productId*, so most writes had to be processed by all 15 scan threads. On the other hand, Clock Scan and Index Union Update Join were able to efficiently handle these updates by building a hash map on the *rloc* equality predicates.

In conclusion, Crescando is a very write-friendly system, due to the absence of locking and other means of blocking synchronization. As in the Amadeus use case, the vast majority of writes in real workloads can be expected to be highly selective and feature equality predicates on some key attribute, making Index Union Update Join a highly effective execution strategy.

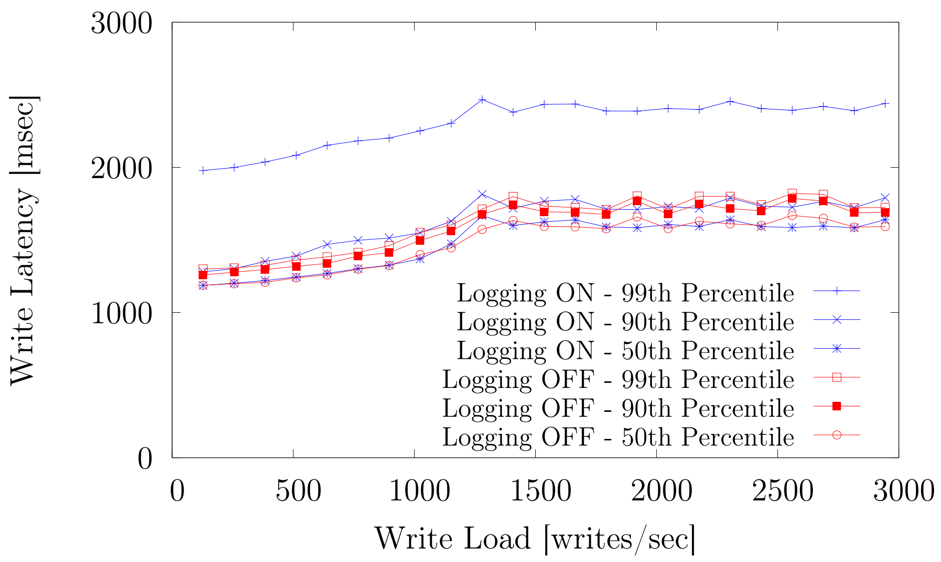


Figure 2.30: *Effect of Logging and Checkpointing on Write Latency: Amadeus Write-only Load, Vary Writes/Sec*

2.10.7 Logging and Recovery

We assessed the quality of Crescando's recovery scheme based on two distinct metrics. One, the latency overhead of logging and checkpointing during normal operation. Two, the time required to resume normal operation after a crash.

Overhead of Logging and Checkpointing

We measured the relative difference in write latency of Crescando under an increasing write load, both with logging enabled and with logging disabled. Write latency is defined as the time from the client enqueuing a write operation to the client receiving the corresponding result tuple. This result tuple is only returned after all potentially affected scan threads have executed the query, and the query has been committed to the log.

Since select operations are not logged, we used a write-only workload to measure the overhead of logging. As Figure 2.30 shows, there was no impact on the 50th and 90th percentile of write latency, only on the 99th percentile. This increase is not because of pending commits. The log manager was configured to flush the log file to disk multiple times per second, so each operation was already committed to the log by the time the last scan thread finished their execution. The "slow" writes had shared a scan cursor with a snapshot select.

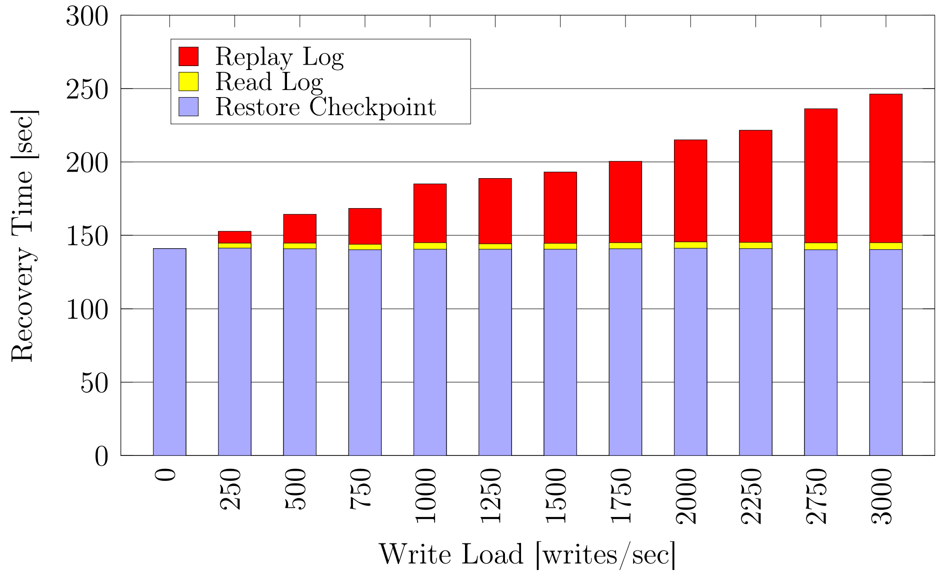


Figure 2.31: *Recovery Time: Amadeus Write-only Load, Vary Writes/Sec*

The 99th percentile increased by about 700 msec with logging enabled, which is approximately the time it took to copy 1 GB of memory from one NUMA node to another on our fully loaded test machine. Writing a snapshot of a segment to disk took about 10 seconds, whereas a scan cycle took less than one second (write latency is almost twice that because each write is executed on multiple segments whose cursors are not in sync). Consequently, more than 90% of the time there is no snapshot select in the system, so only the 99th percentile of writes was affected in the diagram.

In summary, the runtime overhead of logging and checkpointing should be acceptable for most use cases. In the future, the overhead of snapshot selects could be amortized by using incremental snapshot algorithms as found in the literature [CVSS+11].

Recovery Time

In the second recovery-related experiment, we measured the time it took for Crescando to recover and resume normal operation depending on the sustained write load at the time of crash. For each run, we let the test driver generate writes for half an hour before simulating a crash by abruptly rebooting the machine with a hard kernel reset.

We measured the time for each stage of recovery (restore checkpoint, read log, replay log). The results are shown in figure 2.31. The engine could sustain about 3,000 writes/sec in the experiment's configuration, so the results shown cover the entire feasible spectrum.

Attribute	Values	Distinct Values	Net Size (KB)	Padded Size (KB)	Compressed Size (KB)	Compression Ratio
office	3,391,475	36,564	29,775	29,808	13,692	45.93%
name	3,391,475	448,444	24,509	188,783	18,962	10.04%
firstName	3,391,475	397,683	29,808	185,471	19,397	10.46%
sgtVendorValues	3,391,475	27,054	42,824	52,992	13,694	25.84%
		Total:	126,916	457,054	65,745	14.38%

Table 2.2: *Compression Ratios acc. to Bernet [Ber10] for 1 GB of Amadeus Ticket Data*

The numbers match the analysis in the discussion of Section 2.8. Recovery time is dominated by the time it took to read and restore the checkpoint, which is the data volume (15 GB) over the disk bandwidth (110 MB/sec). The log file itself never grew to more than 64 MB. The time to read the log file and deserialize its contents (the write operations) was negligible.

The log replay time was proportional to the write rate and never exceeded the time to restore the checkpoint. In fact, log replay was at least 33% faster than checkpoint restore, because, in contrast to normal operation, scan threads during recovery always have a full input queue and need not update the optimizer statistics until log replay is complete. In summary, recovery time is low and, more importantly, bounded in a predictable manner.

2.10.8 String Compression

In this section we look at the memory savings and performance impact of the dictionary compression option that Crescando offers for string attributes. We have conducted extensive experiments and micro-benchmarks regarding dictionary compression. In this section, we only show the most important results. Additional results and insights will be published in a forthcoming technical report.

Memory Savings

Table 2.2 summarizes the compression ratios Crescando achieves on 1 GB of Ticket data, using dictionary compression for the 4 most suitable string attributes in the schema.

For a given attribute, the *net size* refers to the size of all the string values of the 3.4 million records concatenated together, without null-terminating bytes. This number is a lower-bound on the space requirement for storing the strings without compression. The *padded size* is the space requirement when storing the strings as fixed-length fields (Crescando without compression). The *compressed size* is the cumulative space occupied by the integer key field in the table, plus the space required for a Bidi-map variant called *Bidi2* by Bernet [Ber10]. (Other Bidi-map variants differ in size only by a few percent.)

As can be seen, the memory savings are substantial, even for attributes with a large number of distinct values such as *name* or *firstName*. In total, dictionary compression reduced the memory requirements of 3,391,475 Ticket records by almost 400 MB; i.e., from 1 GB to just over 600 MB.

Latency vs. Read Load

According to results published by Bernet [Ber10], dictionary compression has negligible impact on throughput and latency for the Amadeus workload. Significant effects on performance (positive or negative) could only be observed for very string-heavy workloads. Notably, trie indexes for prefix queries (cf. Section 2.6.3) and order-preserving compression were not yet implemented in Bernet [Ber10], so string compression violated the robustness principle of Crescando when faced with significant amounts of prefix queries.

Since the publication of Bernet [Ber10], proper support for prefix queries as well as optimizations such as hot dictionaries have been added by Bernet and the author of this dissertation. To measure the effects, we created the following two string-heavy, synthetic workloads.

Equality Workload A read-only workload where each select operation has exactly one equality predicate on the attribute *name*. Predicate values are chosen randomly from *name* domain, following the same distribution as the values stored in the table (Amadeus live data).

Prefix Workload Just like the equality workload, but with prefix instead of equality predicates.

As a performance baseline, we created an alternative Ticket schema, where the string attribute *name* was replaced by a 32-bit integer attribute with the exact same value distribution. Likewise, the equality workload and prefix workload have been adapted to perform equality queries and range queries equivalent to the string prefixes in the original prefix workload. In a mathematical sense, the string and integer schemas and workloads are isomorphic.

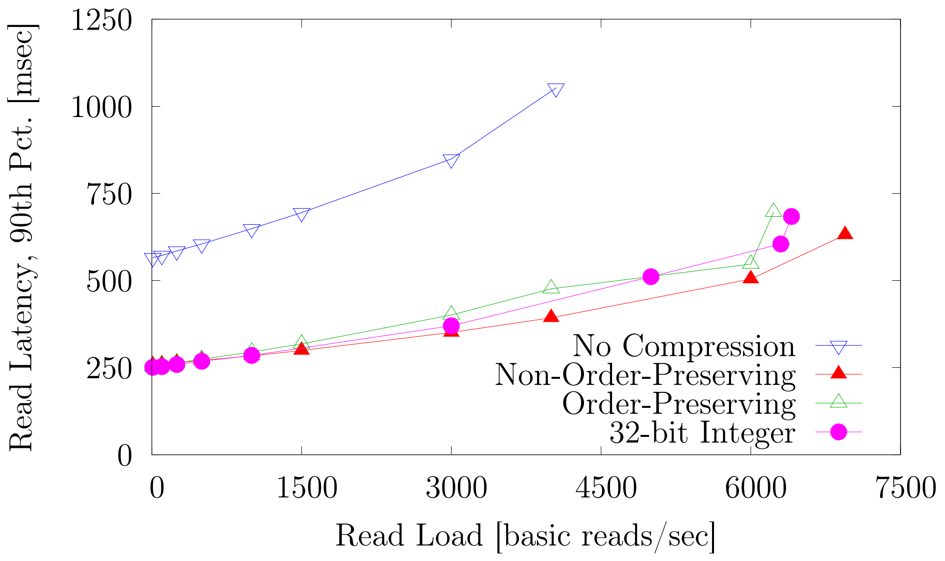


Figure 2.32: *Read Latency: Equality Workload, Vary Reads/Sec, Vary Compression Scheme*

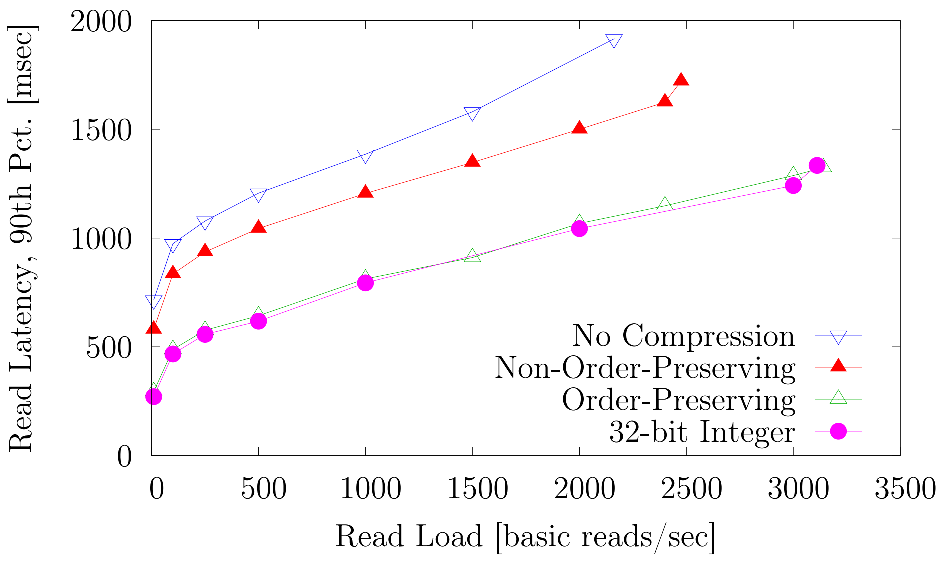


Figure 2.33: *Read Latency: Prefix Workload, Vary Reads/Sec, Vary Compression Scheme*

We loaded a single[9] segment with 1 GB (uncompressed) Amadeus Ticket data and measured how read latency changes for different compression schemes (none, non-order-preserving,

[9]Because dictionary compression is implemented fully inside the scan threads, a multi-threaded experiment is not very interesting.

order-preserving, integer) under an increasing read load op to the point of system saturation. The results for the equality and prefix workloads can be seen in Figures 2.32 and 2.33 respectively.

For equality queries, both non-order-preserving and order-preserving compression roughly halved the latency and doubled the maximum throughput. The reason is that the hash map built by Crescando to index the equality predicates is significantly smaller and faster for integers than for strings. In fact, performance for equality queries over compressed strings is equivalent to the performance of equality queries over integers. (Equality queries are even a bit cheaper than equivalent integer queries due to code compilation artifacts.)

For prefix queries, order-preserving compression allowed the multi-query optimizer to transform the string prefix predicates to integer range predicates, and build range trees over the latter. Consequently, performance for order-preserving compression was the same as for the integer schema and workload.

In contrast, when compression was disabled or non-order-preserving compression was chosen, the best the multi-query optimizer could do was to build tries over the (uncompressed) string values of the prefix predicates. The tries worked very well in fact, but order-preserving compression and range trees are clearly superior for prefix queries.

In the case of non-order-preserving compression, the *name* attribute of each scanned record first had to be decompressed in order to probe the trie index. Owing to the highly optimized dictionary data structures (Bidi-map, Associative Cache), the fact that less data was scanned compared to the uncompressed case *over-compensated* for the cost of decompressing every single record. Note that *name* is a rather large attribute (57 characters), so this curious observation may not hold for other schemas.

In summary, the results show that queries over compressed strings are as cheap as queries over integers. Order-preserving compression appears to pay off for prefix queries, but comes at a price in complexity and update cost (dictionary re-organization) not considered in this experiment.

Latency vs. Write Load

When records are updated, the contents of the compression dictionaries may have to be updated as well. To measure the performance impact of this, we created a fixed prefix workload (as described previously) of 500 reads per second, and added an increasing write load, consisting only of update operations on the *name* attribute. The results are given in Figure 2.34. The numbers show that the overhead of writes, even for very high rates, is negligible.

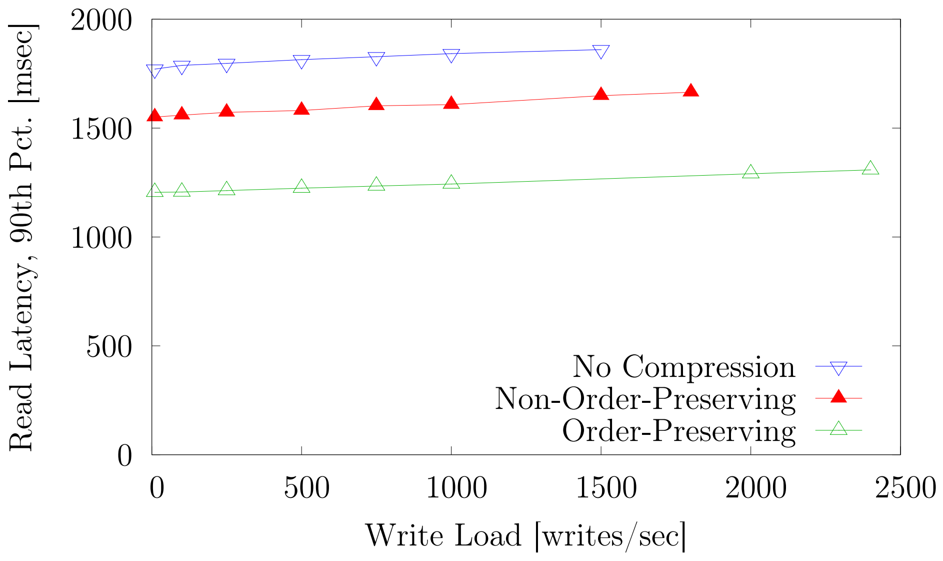


Figure 2.34: *Read Latency: Mixed Load, Prefix Queries Only, 500 Reads/Sec, Vary Writes/Sec, Vary Compression Scheme*

What is not shown in Figure 2.34, is that order-preserving dictionaries occasionally have to be re-organized, as discussed in Section 2.9.2. We measured the cost of re-organization to be less than 500 milliseconds, even when the entire dictionary had to be reconstructed. That is, read and write latency increases by less than 500 milliseconds for those operations which share a scan cursor with a re-organization update.

That being said, the heuristics for dictionary re-organization are very primitive at this point, so unexpected behavior (e.g., repeated re-organization) may occur for certain workloads. Order-preserving compression is a highly experimental feature, and the numbers must be taken with a grain of salt.

2.11 Concluding Remarks

In this chapter, we have described the design, implementation, and performance of Crescando, a push-based, parallel, main-memory, relational table engine that offers predictable performance for unpredictable workloads. Crescando is a complete system, written in about 54,000 lines of C++ (and some Assembly), excluding all tools, benchmarks, and unit tests (another 100,000 lines of C++ and Java). Since September 2011, Crescando is in productive use at Amadeus, solving the Ticket use case described in Section 1.1.

Among the contributions of Crescando to the state of the art of query processing are a novel shared scan algorithm (Clock Scan), a multi-query optimizer for use with Clock Scan, a number of cache-efficient data structures, a novel dictionary compression scheme for strings, and an efficient logging and recovery scheme for scan-based, main-memory engines.

The experimental results demonstrate that Crescando easily meets the throughput and latency requirements of the Amadeus use case. What is more, the system is significantly more robust to mixed workloads and query diversity (future workloads) than traditional relational engines. Dictionary compression for string attributes not only saved memory, but even improved performance and robustness. And, most importantly, the architecture scales linearly with the number of cores, making it ideally suited for modern hardware platforms.

Crescando shows that predictable performance need not be the result of careful isolation of concurrent activities, but can instead be achieved by rigorous, *deep sharing* of resources. Nonetheless, a large space of future generalizations and optimizations remains open. To name just two examples, the multi-query optimization problem can be generalized to accommodate out-of-order operation activation, and the space of query-data join algorithms has hardly been explored.

Chapter 3

E-Cast

Crescando is a complete storage engine that can be used in a stand-alone fashion. However, stand-alone use limits the data size and performance to the capabilities of a single physical machine. To overcome this limitation, we have developed a general distribution framework for elastic, strongly consistent data stores in the Cloud: *Rubberband*. Rubberband is intimately based on a novel multicast protocol called *E-Cast*. In this chapter, we describe the formal model, design, and implementation of E-Cast. Rubberband is the focus of the subsequent chapter.

E-Cast and Rubberband fall into the field of middleware database replication. Middleware database replication solutions use various agreement protocols to ensure that all state-changing messages—hereafter referred to as *updates*—are propagated and applied in the same order at all relevant storage nodes. Most early work follows the classic state-machine approach [Sch90], using full replication (every storage process contains a full copy of the data) for fault-tolerance, implemented via atomic broadcast [DSU04, WS05] of updates over all storage processes in the system [KA00, PGS03, SOMP01].

There is a large body of work that attempts to scale such systems through the use of partial replication (every storage process contains only a subset i.e. partition of the data) and group communication [AT02, FI01, KBB01, SSP10]. Group communication is a combination of communication service (reliable, ordered multicast primitives) and membership service [Bir93, VKCD01]. The membership service establishes consensus on a sequence of system views (configurations), each of which defines one or more process groups. Processes send messages within and (depending on the protocol) across views and groups, and the communication service gives ordering and delivery guarantees for these messages.

Existing group communication toolkits assume that membership changes are infrequent, and it is deemed acceptable that they cause a significant performance hiccup and/or consistency degradation when they take place [BMR10, LMZ10]. But implementing partial replication over large numbers of data objects and storage processes through group communication implies a potentially large number of groups (copies of items will be placed in different subsets of nodes), and also frequent changes to the composition of those groups as the system scales out and in and nodes appear and disappear (due to node failures, load balancing, or simply because the nodes are assigned to other tasks).

To support such challenging scenarios, we propose a novel multicast primitive, *E-Cast*, specifically tailored to provide strong ordering and delivery guarantees across many groups and frequent membership changes. To be precise, E-Cast is a *dynamic, uniform, causal total order multicast protocol*. (We will informally and formally explain what all these terms mean in the course of this chapter.)

E-Cast uses a (small) replicated state machine of router processes to linearize and relay all state-affecting messages exchanged between application processes. The key insight in this respect is that it is possible to determine the destination processes of any message exclusively by looking at the past history of messages, rather than relying on a separate membership service, and without the sender specifying concrete destination processes or process groups. Based only on a *user-defined function* over the sequence of messages delivered so far, the router processes can determine where a message has to be delivered.

As a result, E-Cast is as easy to use and reason about as state-machine replication. But unlike plain state-machine replication, E-Cast gives applications the freedom to choose which message to deliver to which subset of processes (through the user-defined routing function), thus being more efficient and scalable.

In this chapter, we formalize the *stateful routing* problem underlying E-Cast, and show how partial replication with strong consistency guarantees can be expressed as said routing problem. We present the E-Cast protocol and its implementation, and prove that E-Cast is a correct solution of the routing problem. Since E-Cast by itself cannot operate without a concrete routing function, an experimental study of the performance, elasticity, and fault tolerance of E-Cast is deferred to the subsequent chapter on Rubberband.

3.1 Informal Problem Statement

Consider a system consisting of 3 storage processes: A, B, C. The system stores 3 data objects: x, y, z. For purposes of fault tolerance, 2 copies of each data object exist. The storage processes are arranged in a logical ring, where each storage process contains the primary copy of one data object, and a secondary copy of its predecessor's data object. This example of *partial replication*[1] is visualized in Figure 3.1.

[1] To be precise, this is an example of successor replication [KZHL07].

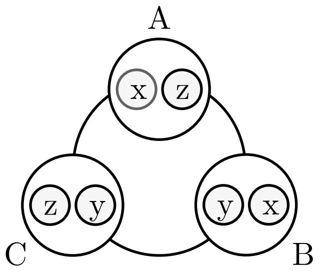


Figure 3.1: *Example of Partial Replication*

Now consider the following 3 multi-key updates: $U_1 = \{x \leftarrow 1,\ y \leftarrow 1\}$, $U_2 = \{y \leftarrow 2,\ z \leftarrow 2\}$, $U_3 = \{z \leftarrow 3,\ x \leftarrow 3\}$. To be *strongly consistent*, the system must behave as if there was a single copy of each data object, and as if the updates were executed strictly in some serial order [FR10]. If each storage process could independently choose the order in which to execute the updates, the processes may disagree on the final values of x, y, and z. To be precise, for every pair of (transitively) conflicting updates, all the affected storage process must agree on an order of execution (serialization order) [FR10].

In the general or worst case (all updates conflict transitively), the only way to guarantee consistency is thus to ensure that *all* storage processes agree on a *single* serialization order. In the spirit of Crescando, we are interested in a solution that performs well in the worst case; i.e., a solution that performs well even under a high number of conflicts.

It is known that distributed agreement on a total order is equivalent to distributed consensus [CT96]. Thus, a system that needs to tolerate process failures and provide strong consistency for arbitrary multi-key updates must, in the general case, establish distributed consensus across *all* storage processes in the system. The E-Cast protocol described in this chapter automatically and efficiently establishes this consensus, and delivers all messages (i.e., updates) in total order (along with other guarantees), regardless of (permanent) process failures.

But total order delivery is not enough. Another problem is how processes learn of exactly those updates that concern them. If the mapping of data objects to processes is *static*, clients can cache the mapping and send updates directly to the affected processes. But if the mapping can change concurrently (for elasticity, load-balancing, or fault tolerance), updates may be delivered to the wrong processes. The resulting problem is called *dynamic partial replication*.

Consider the extended example in Figure 3.2. A new process D joins the ring between A and C. Consequently, the copy of z stored by A moves to D. Suppose this happens concurrently with U_2 and U_3. The system has to ensure that U_2 and U_3 are applied to all

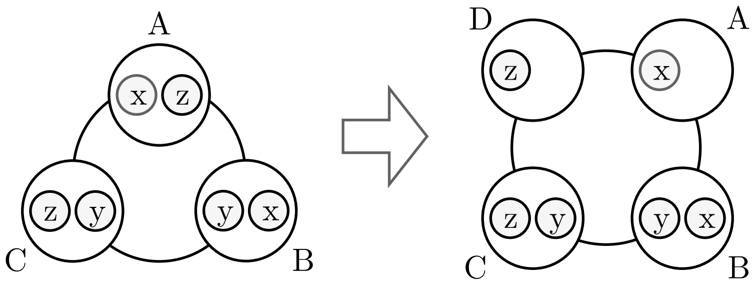


Figure 3.2: *Example of Dynamic Partial Replication*

copies of z, regardless of concurrent repartitioning. Broadcasting all updates is a solution, but obviously scales very poorly. In this chapter we present a better solution.

3.2 State of the Art

Replicated storage systems are typically implemented following the state-machine approach [Sch90], either by directly embedding a consensus algorithm such as Paxos [Lam98], or via reliable, ordered multicast protocols bundled in group communication toolkits (which may in turn use Paxos) [VKCD01, BMR10]. In this section, we give a historical and technical overview over these important techniques as they pertain to the design of E-Cast.

3.2.1 State-Machine Replication

The state-machine approach is an old, simple, and elegant method of implementing a fault-tolerant service. The core idea is as follows. Take any deterministic, event-driven system. Fault-tolerance can be achieved by replicating the (initial) system state over failure-independent machines, and ensuring that every replica sees every state-changing event or command in the same, total order. The idea takes its name from the fact that the replication management protocols are best modeled as deterministic finite state automatons, i.e. state machines, in order to understand how and why they work [Sch90].

The idea goes back to a 1978 seminal paper by Lamport [Lam78], which mostly deals with logical time and the partial ordering of events in a distributed system, but also introduces state-machine replication in an example. In 1984, Lamport elaborated on the idea in another seminal paper [Lam84]. Schneider eventually published a widely cited tutorial of state-machine replication [Sch90], which showed the generality of the approach, and put it in context with different system, network, and failure models in distributed computing.

The state-machine approach is relevant to E-Cast in that the set of router processes that are used to mediate message transfer between application processes form a replicated state machine, where the replicated state is the set of routed messages and their order.

Indeed, E-Cast guarantees that messages between application processes are delivered uniformly (either all destination processes receive a given message, or none), in total order. The E-Cast protocol itself can thus be used to implement a fault-tolerant, replicated state machine at the application level.

3.2.2 Distributed Consensus and Paxos

The difficult part in implementing a replicated state machine is ensuring that every replica sees the same commands in the same, total order, in the face of process and network failures. This problem is strongly related to a fundamental problem in distributed computing called *distributed consensus*. For distributed consensus, a set of processes must agree on a single output value (e.g. accepting or rejecting an input command). Agreement among replicas on a single sequence of commands to be executed reduces to repeated distributed consensus on a command and its predecessor command [Sch90][2].

Fischer, Lynch, and Patterson proved in their seminal paper the impossibility of distributed consensus in asynchronous networks (unbounded message delivery time) with one faulty (spontaneously halting, unreachable, or otherwise unresponsive) process [FLP85]. To be precise, they showed that no deterministic algorithm that solves distributed consensus can be guaranteed to make progress. The consequence is that no real-world[3] replicated state machine can be guaranteed to make progress.

But while it is not possible to formally guarantee progress, it is still possible to solve distributed consensus in practice, and thereby implement replicated state machines. The most well-known protocol for distributed consensus is Paxos [Lam98]. Paxos *eventually* reaches distributed consensus provided that a majority of processes is correct (non-faulty)[4], and *eventually* a single process is able to propose a value (to reach consensus on), and *for a sufficiently long period* no other process makes a conflicting proposal. The latter assumption is key in overcoming the FLP impossibility result.

[2]Conversely, distributed consensus reduces to agreement on a sequence of values, since a trivial solution to distributed consensus given a sequence of values is to choose the first value in the sequence. The two problems are therefore *equally hard* from a theory perspective [HT94, CT96].

[3]Physical networks such as Ethernet cannot guarantee packet delivery due to the possibility of electrical interference, packet collisions, and network congestion.

[4]This requirement is in fact *minimal*. Consensus cannot be reached safely in asynchronous networks without a majority of correct processes [GL02].

Many variants of Paxos exist, such as Fast Paxos [Lam06], Cheap Paxos [LM04], Multi-Paxos [PLL97], and Generalized Paxos [Lam05]. In E-Cast, router processes are replicated using a novel atomic broadcast protocol, which in turn is implemented based on a wait-free variant of Multi-Paxos. We use the term *wait-free* in the sense of asynchronous and pipelining, not in the sense of wait-free parallel algorithms. The FLP impossibility result is overcome by binary exponential backoff [MB76] after conflicting proposals.

3.2.3 Reconfiguring a State Machine

Most state machine implementations are based on Paxos, and like Paxos they assume that the set of replica processes is static [Bur06, CGR07, HKJR10, BBH$^+$11]. Since distributed consensus requires a majority of processes to be live, a long-running system that assumes static membership must rely on processes to eventually *recover* into a previous, well-defined state after they fail. This is called the *fail-recovery* failure model [CGR11], where recovery is typically implemented via *write-ahead-logging* (WAL) to local disks[5].

The assumption that there is a static set of processes that will always be able to recover is problematic in Cloud Computing. Especially in a public cloud, machines may not be under the control of the application, and disks may be of dubious quality. In a cloud environment, it is thus highly desirable to avoid disks for critical data such as recovery logs, and to be able to change the set of replicas dynamically.

What is required is called a *reconfigurable state machine*. The main problem in building a reconfigurable state machine is *pipelining* (overlapping) the sequence of distributed consensus instances in order to reach high throughput, independent of the network latency. Pipelining is particularly difficult when replicas are joining or leaving the system concurrently, which can take a fair amount of time when large amounts of state are to be transferred [AKMS10].

Lamport et al. [LMZ08, LMZ10] describe extensions to Paxos which allow the set of processes to change over a sequence of protocol instances. Other examples of algorithms to the same effect are RAMBO [LS02] and RDS [CGG$^+$09]. While conceptually simple, the implementation details of all these algorithms are extremely tricky [AKMS10]. For example, distributed garbage-collection of obsolete consensus instances is difficult in the presence of arbitrary crash failures. As a result, production-grade implementations of fault-tolerant, reconfigurable state machines are very rare.

[5]Generalizations such as Disk Paxos exist [GL03], which perform WAL to remote disks, but this exacerbates the latency overhead of WAL and extends the membership problem to the disks.

We could not find any open-source implementations of the algorithms mentioned, so we have implemented our own wait-free (asynchronous and pipelining), reconfigurable extension of Multi-Paxos [PLL97] from scratch. This protocol, described in Section 3.5.4, is used to replicate router processes in E-Cast. As a result, E-Cast supports fully dynamic membership of all types of processes, can aggressively pipeline consensus instances, and does not require disks, making it an ideal solution for implementing (generalized) state-machine replication in a cloud environment.

3.2.4 Group Communication

During the time where Lamport formulated state-machine replication, Birman [BJ87] independently developed the *virtual synchrony* model for system replication, also known as *group communication* in the literature.

Group communication is a combination of communication service (more-or-less reliable and ordered multicast primitives) and membership service [VKCD01]. The membership service establishes distributed consensus on a sequence of system views, each of which defines a configuration that consists of one or more process groups (depending on the concrete model and service). Application processes can send messages within and across process groups and views, and the communication service gives ordering and delivery guarantees. For example, a possible ordering guarantee is *FIFO order*, where for every two messages m, m' some process sends, every process that receives the two messages will deliver them to the application in the order they were sent.

State-machine replication can be implemented via *atomic broadcast*, a synonym for *uniform total order broadcast*, over a single group of processes. Atomic broadcast means every message that is delivered by one process is delivered by every non-faulty process (uniformity), and every process delivers messages in the same, total order [DSU04]. This problem is identical to command ordering and uniform execution in state-machine replication.

If the atomic broadcast protocol allows the group of processes to be dynamic, one can use it to implement a reconfigurable state machine. To be precise, reconfiguring a state machine is the same problem as atomic broadcast under the primary partition model of dynamic group communication [VKCD01, Sch06].

The spectrum of group communication is much larger than atomic broadcast however. Other ordering guarantees such as FIFO order (see above) or causal order exist. Some models and protocols assume a single group of processes (broadcast). Other models and protocols generalize this to multiple groups of processes (multicast), which may or may not overlap in various ways. The E-Cast protocol presented in this chapter can be classified as a dynamic, uniform, causal total order multicast protocol.

Vitenberg et al. [VKCD01] provide a comprehensive overview over the available models, Defago et al. give a survey of the many known multicast protocols and their respective guarantees [DSU04], and Birman [Bir10] provides a historical overview of group communication, mentioning many real systems and setting the work in historical context with respect to state-machine replication. "Classic" examples of multicast toolkits are Isis [BR94], Transis [DM96], Horus [vRBM96], and Totem [AMMS$^+$95]. More recent work includes JGroups [DB97], Spread [ADS00], and QuickSilver [OBD08].

Historically, the early work on group communication was more application and performance-oriented, while state-machine replication originated in the theory community. Only recently, the two ideas have been fully acknowledged as two sides of the same coin by the two camps of authors. State-machine replication is being integrated into group communication toolkits to strengthen the formal properties and fault-tolerance of the protocols, while ideas from group communication such as dynamic membership and relaxed ordering guarantees are being integrated into state-machine replication [BMR10].

To see how group communication relates to dynamic partial replication, consider a system which defines one process group per data object (each member process holding a copy). The group communication toolkit ensures that updates of each object are delivered uniformly, in total order, to all members of the respective group. This implicitly preserves consistency. Unfortunately, existing group communication toolkits do not scale beyond a few thousand groups [Bir10], so this straight-forward approach does not work. Creating a 1:N mapping between groups and objects (in order to scale) creates the problem of how to—efficiently and consistently—manage this mapping in case of data repartitioning; i.e., objects moving between groups, or addition and removal of groups.

The E-Cast protocol presented in this chapter has been specifically designed for strong message delivery and ordering guarantees across large numbers of overlapping groups with frequent membership changes. The notion of *group* is slightly misleading here, because E-Cast does not explicitly maintain groups. Instead, E-Cast models multicast as a stateful routing problem. The key idea is to establish a total order over all state-changing messages in the system; i.e., all data updates and configuration updates. Knowing this total order, a dynamic set of router processes forming a reconfigurable state machine use a *user-defined*, plugged-in config module to compute the system configuration at any point in history, and route messages accordingly.

3.2.5 System Management and Coordination

The Rubberband framework presented in the subsequent chapter plugs a specific config module into E-Cast, but Rubberband is just one *application* of E-Cast. Since applications can define arbitrary config modules and routing functions, the E-Cast protocol can be used for any type of distributed system that requires strong consistency guarantees.

For example, E-Cast can serve as a *consensus service* for use in chain replication [vRS04, Fri10]. We revisit this idea in Section 4.5, where we discuss ways to improve the scalability of Rubberband with respect to single-key reads and writes.

Even if the application in question does not require strong consistency at first sight, the ability to replicate critical system state (*hard state*) such as membership, data partitioning, shared locks, and security-related meta-data is extremely useful. See the paper on Yahoo!'s ZooKeeper service by Hunt et al. [HKJR10] for the many use cases of strong consistency in system management and coordination. One such use case, a distributed lock service, has already been implemented with the help of E-Cast (independent of Rubberband).

E-Cast offers the same strong consistency guarantees and *wait-free* protocol properties as ZooKeeper. But in contrast to ZooKeeper, E-Cast does not assume static membership (one cannot add or remove replicas from ZooKeeper at runtime), does not require stable storage (disks), supports partial replication, and gives applications the freedom to choose their own data and query model. Consequently, E-Cast can be used to implement any service that ZooKeeper is used for, and more.

3.3 Formalization

Group communication is a formally sound approach to dynamic partial replication (DPR), but the central *group* abstraction is problematic, as discussed previously. In this section, we present an alternative, novel formalization of DPR, as well as a stateful routing problem (SR), and a reduction of DPR to SR. The formalization of SR naturally leads to its algorithmic solution: E-Cast. The reduction of DPR to SR we present here is then implemented by Rubberband, as described in the subsequent chapter.

3.3.1 Dynamic Partial Replication (DPR)

Recall the informal problem statement for dynamic partial replication in Section 3.1. In the following section, we present a rigorous formalization of the application model and problem statement.

Application Model

The application consists of a set of client processes C, and a set of storage processes S. For each client process $c \in C$, a set of updates U_c, and a strict total order relation $\lhd_c$ on U_c exist. The total order relation $\lhd_c$ is called the write order relation of c.

Every storage process maintains a table. In the systems we construct in this dissertation, a table is some horizontal partition of an application's data set. But in the formal model, tables are opaque atoms.

Let $\mathbb{T}$ denote the universe of all possible tables. For each storage process $s \in S$, an initial table $T_s^0 \in \mathbb{T}$, a set of updates U_s, and a strict total order relation $\blacktriangleleft_s$ on U_s exists. We call $\blacktriangleleft_s$ the execution order relation of s.

Definition 1 (Update). *Let $U \stackrel{\text{def}}{=} \bigcup_{p \in (C \cup S)} (U_p)$ denote the set of all updates. An update $u \in U$ is a function $u \colon \mathbb{T} \to \mathbb{T}$ which, when applied to a table, yields another table.*

Let $W = \left\langle u^0, ..., u^{k-1} \right\rangle$ be some sequence of updates of length k. We write $W(T)$ as a shorthand for the successive application of every update in W on the table $T \in \mathbb{T}$. That is, $W(T) \equiv u^{k-1}(\ldots u^1(u^0(T)))$.

For every storage process $s \in S$, its execution order $\blacktriangleleft_s$ induces a sequence of updates $\langle U_s, \blacktriangleleft_s \rangle = \left\langle u_s^0, u_s^1, ..., u_s^{k-1} \right\rangle$. Executing the sequence of updates on T_s^0 induces a sequence of tables $\langle T_s^0, T_s^1 \ldots, T_s^k \rangle$. We define $\mathbb{T}_s \stackrel{\text{def}}{=} \left\{ T_s^0, T_s^1 \ldots, T_s^k \right\}$ as the set of tables that s holds over its lifetime.

So far, we have defined client processes, storage processes, tables, and updates. This is all very similar to formalizations of state-machine replication [Sch90]. To reason about dynamic partial replication, we additionally need to talk about the effects that a given update may or may not have on a given table. To this end, we define configurations.

Definition 2 (Configuration). *A configuration $\gamma \colon S \times U$ is a predicate over storage process, update pairs. Let Γ denote the universe of configurations. Every update $u \in U$ is also a higher-order function $u \colon \Gamma \to \Gamma$.*

In other words, just like applying an update to a table yields another table, applying an update to a configuration yields another configuration.

Let $W = \left\langle u^0, u^1, ..., u^{k-1} \right\rangle$ be some sequence of updates of length k. We write $W(\gamma)$ as a shorthand for the successive application of every update in W on the configuration $\gamma \in \Gamma$. Furthermore, a function prefix(W, u) is defined, which yields the prefix of W up to but excluding u.

There is exactly one initial configuration $\gamma^0 \in \Gamma$ (the initial system configuration, say a single storage process that contains an empty table). As a syntactic shorthand, we define $T_s^{W,u} \overset{\text{def}}{=} \text{prefix}(W, u)(T_s^0)$ and $\gamma^{W,u} \overset{\text{def}}{=} \text{prefix}(W, u)(\gamma^0)$.

The following is assumed of configurations:

Assumption 1 (Complete Configuration)**.** *If applying some $u \in U$ to $T_s^{W,u} \in \mathbb{T}_s$ yields a different table, then $\gamma^{W,u}(s)$ holds.*

Paraphrasing this, the assumption states that a configuration may not reliably predict whether an update has an effect on a particular table. But, if it predicts that an update will *not* have an effect on a table, then this is invariably true. In E-Cast, this knowledge is exploited to avoid routing and executing updates that are guaranteed to have no effect on particular tables.

As a concrete example, a Rubberband configuration holds a mapping of data objects to storage processes. This mapping conservatively predicts which updates may affect which processes. An update in Rubberband can either change just the table contents of storage processes (data update), or change the mapping itself (configuration update), affecting which processes must deliver and execute which subsequent data update.

Problem Statement

An algorithm is a *correct* solution of **Dynamic Partial Replication** (DPR) iff it executes updates in a way that satisfies the following four conditions.

Integrity. *No storage process $s \in S$ ever holds a table that cannot be reached by applying some sequence of updates W on T_s^0, where $U_W \subseteq \bigcup_{c \in C}(U_c)$. Formally,*

$$\forall s \in S,\ T \in \mathbb{T}_s : \exists U' \subseteq U,\ W \in \text{seq}(U') : T = W(T_s^0)$$

Consistency. *A total order relation over U, call it $<$, exists, which is an extension of the write order of every client process, and the execution order of every storage process. Formally,*

$$\forall u, u' \in U : ((\exists c \in C : u, u' \in U_c \wedge u \lhd_c u') \vee (\exists s \in S : u, u' \in U_s \wedge u \blacktriangleleft_s u')) \implies u < u'$$

The $<$ relation is exactly the single serialization order discussed in Section 3.1. In comparison to the informal problem statement, we additionally require here that the serialization order is compatible with the write order of every client. This type of strong consistency is called *sequential consistency* in the literature [FR10], and is the distributed computing analogue of *linearizability* in parallel computing [HW90].

Compared to simply establishing *some* total execution order (that may not be an extension of the write order of every client), sequential consistency allows clients to reason about the final state of the system after submitting a sequence of updates, *without reading* the system state in-between updates. This is not just a convenience. It allows updates to be *safely pipelined* by the application, which is essential for reaching high system throughput, regardless of the network latency. This is particularly important for our main use case—the Amadeus Ticket view—where most updates originate at the legacy system, and need to be executed at a very high frequency, in submission order.

Atomicity. *If one storage process $s \in S$ executes an update $u \in U$, at some point the table of every correct storage process $s' \in S$ is in a state that can be reached by applying a sequence of updates to $T^0_{s'}$ which includes u. Formally,*

$$\forall s \in S,\ u \in U_s,\ s' \in S : \mathrm{correct}(s') \implies \exists T \in \mathbb{T}_{s'},\ U' \subseteq U_{s'},\ W \in \mathrm{seq}(U') : T = u(W(T^0_{s'}))$$

A process is defined *correct* iff it is not faulty. In this dissertation, we only consider crash failures (no Byzantine failures), but the problem statement is independent of this.

Durability. *For any update $u \in U_c$ of a correct client process $c \in C$, at some point the table of every correct storage process $s \in S$ is in a state that can be reached by applying a sequence of updates to T^0_s which includes u. Formally,*

$$\begin{gathered}\forall c \in C,\ u \in U_c,\ s \in S : \mathrm{correct}(c) \land \mathrm{correct}(s) \implies \\ \exists T \in \mathbb{T}_s,\ U' \subseteq U_s,\ W \in \mathrm{seq}(U') : T = u(W(T^0_s))\end{gathered}$$

To better understand the four conditions, recall the extended example in Section 3.1. To let the storage process D join and hold a copy of data object z, a client process (possibly D itself, D may be both a client and storage process) issues a configuration update u such that any subsequent data update u' on z where $u < u'$ will have an effect on the table of D. Integrity, Consistency, and Atomicity guarantee that the system will not enter a state that violates the application semantics (safety), and Durability guarantees that u and u' cannot simply be ignored (liveness).

Note that none of the correctness conditions make any reference to configurations. This is intentional. A configuration may reveal that a specific update u has no effect on the table held by some storage process s. Executing u by s despite this fact is inefficient, but it is also safe. Further note that the model and problem statement does not include reads (queries). We will discuss reads and different isolation levels only at a semi-formal level, in the context of Rubberband in the next chapter. Given that the model we present here is reasonably clean and compact, it should be possible to formally extend it in the future.

3.3.2 Stateful Routing (SR)

This section models atomic multicast as a stateful routing problem. The subsequent section then reduces dynamic partial replication to stateful routing. Consequently, any algorithm that solves stateful routing, i.e. E-Cast, also solves dynamic partial replication.

Distributed System Model

The system consists of a set of processes P, which are sequences of discrete events. Submitting or delivering a message is an event. Let M be the universe of messages. Define $M^{\Sigma} \subseteq M$ as the set of submitted messages, and $M^{\Delta} \subseteq M$ as the set of delivered messages. Every process $p \in P$ submits a set of messages $M_p^{\Sigma} \subseteq M^{\Sigma}$, and delivers a set of messages $M_p^{\Delta} \subseteq M^{\Delta}$.

We write *submit* and *deliver* here instead of *send* and *receive*. This distinction is intentional. Submitting or delivering a message are events that are observed by the application, while a protocol that implements stateful routing may send and receive many messages that are never seen by the application.

It is assumed that every message $m \in M^{\Sigma}$ is submitted at most once[6]. Let $\mathrm{sub}(m)$ denote the submission event for some message $m \in M^{\Sigma}$, and $\mathrm{Dlv}(m, p)$ denote the set[7] of delivery events for a process $p \in P$ and message $m \in M_p^{\Delta}$.

The causal order relation $e \rightarrow e'$ over events e, e' is defined as the minimal, transitive relation that holds if both e and e' are events of some process p and e precedes e' in the execution of p, or $e = \mathrm{sub}(m)$ and $e' \in \mathrm{Dlv}(m, p)$ for some $p \in P$ and some $m \in M$. The causal order relation $\rightarrow$ is identical to the "happens-before" relation used to define time in an asynchronous distributed system [Lam78].

We call a sequence of messages an *m-seq*. M-seqs are central to both the formal model and the implementation of E-Cast. Let $\mathbb{Q}$ be the universe of m-seqs over the universe of messages M. An m-seq $Q \in \mathbb{Q}$ is a pair $Q = \langle M_Q, \rightarrowtail_Q \rangle$, where $M_Q \subseteq M$ is a set of messages, and $\rightarrowtail_Q$ is a strict total order relation over M_Q. A function $\mathrm{prefix}(Q, m)$ is defined, where Q is an m-seq, and m is a message. If $m \in M_Q$, the function yields the prefix of Q up to but excluding m. If $m \notin M_Q$, then $\mathrm{prefix}(Q, m) = Q$.

[6]This is not a limitation. An application may submit the same string of bits multiple times. This will result in multiple distinct messages. A concrete implementation of stateful routing (E-Cast) can easily use unique message IDs to this effect.

[7]$\mathrm{Dlv}(m, p)$ is a set, since an (incorrect) protocol may deliver a message m multiple times to some process p.

The existence of a function $\mathrm{destset}(Q, m)$ is assumed, which maps every pair of m-seq Q and message m to a finite set of processes $\mathrm{destset}(Q, m) \subseteq P$ called the *destination set* of m according to Q. We define the shorthand $\mathrm{destsetpfx}(Q, m) \overset{\mathrm{def}}{=} \mathrm{destset}(\mathrm{prefix}(Q, m), m)$.

Recall the informal problem statement in Section 3.1. Assume configuration and data updates are messages. The sequence of configuration updates in an m-seq then induces a sequence of configurations, each of which is a mapping of data objects to storage processes. The destset function formalizes this mapping. The prefix function models the fact that the position of an update in the m-seq determines which configuration applies. In the example, the update U_2 then needs to be executed either by A and C, or by C and D, depending on whether or not U_2 precedes the configuration update that lets D join.

Predicates over Message Sequences

To be able to give a compact, formal problem statement for stateful routing, the following predicates over m-seqs are defined.

valid(Q). *Every message in M_Q is submitted, and no message is delivered that is not in M_Q. Formally,*

$$\mathrm{valid}(\langle M_Q, \rightarrowtail_Q \rangle) \overset{\mathrm{def}}{=} M^{\Delta} \subseteq M_Q \subseteq M^{\Sigma}$$

minimal(Q). *Every process that delivers a message m is in the destination set of m according to Q. Formally,*

$$\mathrm{minimal}(\langle M_Q, \rightarrowtail_Q \rangle) \overset{\mathrm{def}}{=} \forall p \in P,\ m \in M_p^{\Delta} : p \in \mathrm{destsetpfx}(Q, m)$$

complete(Q). *For every $m \in M_Q$, every correct process in the destination set of m according to Q delivers m. Formally,*

$$\mathrm{complete}(\langle M_Q, \rightarrowtail_Q \rangle) \overset{\mathrm{def}}{=} \forall m \in M_Q, p \in \mathrm{destsetpfx}(Q, m) : \mathrm{correct}(p) \implies m \in M_p^{\Delta}$$

gap-free(Q). *For every message $m \in M_Q$ submitted by some process p, every message m' submitted previously by p is also in M_Q. Formally,*

$$\text{gap-free}(\langle M_Q, \rightarrowtail_Q \rangle) \overset{\mathrm{def}}{=} \forall p \in P,\ m, m' \in M_p^{\Sigma} : (\mathrm{sub}(m) \rightarrow \mathrm{sub}(m') \wedge m' \in M_Q) \Rightarrow m \in M_Q$$

submit-ordered(Q). *$\rightarrowtail_Q$ is an extension of the causal order relation $\rightarrow$ over the submit events of $M_Q \cap M^{\Sigma}$. Formally,*

$$\text{submit-ordered}(\langle M_Q, \rightarrowtail_Q \rangle) \overset{\mathrm{def}}{=} \forall m, m' \in \left(M_Q \cap M^{\Sigma}\right) : (\mathrm{sub}(m) \rightarrow \mathrm{sub}(m')) \Rightarrow (m \rightarrowtail_Q m')$$

delivery-ordered(Q). *$\rightarrowtail_Q$ is an extension of the causal order relation $\rightarrow$ over the delivery events of $M_Q \cap M^\Delta$. Formally,*

$$\text{delivery-ordered}(\langle M_Q, \rightarrowtail_Q \rangle) \overset{\text{def}}{=} \forall m, m' \in \left(M_Q \cap M^\Delta\right),\ e \in \text{Dlv}(m),\ e' \in \text{Dlv}(m') : \\ (e \rightarrow e') \implies (m \rightarrowtail_Q m')$$

terminated(Q). *Every message m submitted by a correct process p is in M_Q. Formally,*

$$\text{terminated}(\langle M_Q, \rightarrowtail_Q \rangle) \overset{\text{def}}{=} \forall p \in P,\ m \in M_p^\Sigma : \text{correct}(p) \implies m \in M_Q$$

Problem Statement

A protocol is a *correct* solution of **Stateful Routing** iff it transfers messages between processes in a manner that the following condition holds.

Correct Routing. *There is a valid, minimal, complete, gap-free, submit-ordered, delivery-ordered, terminated m-seq Q.*

Correct Routing requires that any message m submitted by a correct process is delivered, in the "right" order, to the processes which the application considers destinations of m, by virtue of the destination function destset. The following (generalizations of) familiar properties from the group communication literature follow from Correct Routing.

Validity. *No delivered message has not been submitted.*

Uniform Causal Order Delivery. *If the submission of $m \in M^\Sigma$ causally precedes the submission of $m' \in M^\Sigma$ then no process $p \in P$ delivers m' before m.*

Uniform Total Order Delivery. *For any pair of messages $m, m' \in M^\Delta$, if some process $p \in P$ delivers both m and m', and delivers m before m', then any process $p' \in P$ that delivers m and m' delivers m before m'.*

Uniform Agreement. *Any message m delivered by any process is delivered by every correct process in the destination set of m, in accordance with some m-seq Q.*

Termination. *Any message m submitted by a correct process is delivered to every correct process in the destination set of m, in accordance with some m-seq Q.*

Note that if $\text{destset}(Q, m) \overset{\text{def}}{=} P$ for any Q and m, then the problem statement is equivalent to uniform atomic broadcast with causal order delivery [DSU04]. Stateful routing is thus a proper generalization of uniform atomic broadcast. As such, it can be used to implement state-machine replication, and all the known impossibility results and lower bounds on protocol complexity apply.

3.3.3 Reduction of DPR to SR

Let every update in dynamic partial replication (DPR) be a message in stateful routing (SR). Let every client process $c \in C$ submit each update $u \in U_c$ exactly once. Then for every client c, the write order $\lhd_c$ corresponds to the submission order over M_c^{Σ}. Let every storage process $s \in S$ execute updates in delivery order. Then the execution order $\blacktriangleleft_s$ corresponds to the delivery order over M_s^{Δ}.

If every update $u \in U$ is delivered to every correct storage process $s \in S$, then Atomicity and Durability hold trivially. However, this "broadcast-based" solution does not scale with the number of storage processes.

But note that if $\gamma^{W,u}(s)$ does not hold, then Atomicity and Durability hold trivially for s with respect to W and u, because the table $T_s^{W,u}$ held by s is already in a state as-if u had been applied. This insight leads to a definition of destset, which avoids broadcast.

Definition 3 (destset)**.** destset$(W, u) \overset{\text{def}}{=} \{s \in S \mid W(\gamma^0)(s)\}$

Corollary 1. *For any sequence of updates W, if $u(T_s^{W,u})$ yields a different table than $T_s^{W,u}$, then s is in the destination set of u according to W.*

This completes the reduction.

The last corollary expresses exactly what is needed to solve dynamic partial replication. Assuming that configurations satisfy the Complete Configuration assumption on page 99, a correct solution of stateful routing guarantees that every update that may affect the table stored by a given process is delivered to that process.

Theorem 1. *Under the reduction, a solution to Stateful Routing is a solution to Dynamic Partial Replication.*

Proof. Validity implies that $\forall s \in S : U_s \subseteq \bigcup_{c \in C}(U_c)$. Integrity follows immediately. The update serialization order $<$ is identical to the event order $\rightarrow$. Consistency follows from Uniform Total Order Delivery and Uniform Causal Order Delivery. By definition of destset, Atomicity follows from Uniform Agreement, and Durability follows from Termination. □

The reduction defines destset based on configurations. Let us call a destset function *perfect* iff it routes updates *exactly* to the set of storage processes whose tables are affected. While appealing in theory, a perfect destset function is impractical given that there may be billions of data objects in the tables. In an implementation, there is a trade-off between the amount of configuration state and the number of updates that are delivered unnecessarily. E-Cast gives applications the ability to implement their own trade-off.

As a concrete example, Rubberband configurations map a *range* of data object keys to each storage process (range partitioning). The destset function of Rubberband routes each data update to exactly those storage processes whose key range overlaps with the key range(s) of the update. This avoids broadcast of updates in most cases, but obviously a specific update may not have an effect on every storage process it is delivered to.

The guarantees made by stateful routing (SR) are in fact a bit stronger than required to solve dynamic partial replication (DPR). For instance, DPR only requires total FIFO order delivery, while SR guarantees total causal order delivery. Also, SR guarantees that m-seqs are gap-free and complete, which together is a stronger uniformity guarantee than required by DPR. DPR "only" requires that the effects of a given update are applied to all or none of the processes' tables. But SR additionally guarantees that for any update u that is applied, every update *previously submitted* by the same client as u is also applied.

These guarantees may not seem interesting from a formal point of view, but they are very useful in practice. For instance, gap-freeness allows applications to pipeline a large number of updates in order to maximize throughput over high latency network links, while still, in the event a client process crashes, have the guarantee that some prefix (and not an arbitrary subset) of the pending updates will be applied. This property is important for our main use case—the Amadeus Ticket view—where a high-rate stream of updates originating at the legacy system must be applied uniformly to guarantee consistency with the legacy system.

3.3.4 Network and Failure Model

The formal problem statements of stateful routing and dynamic partial replication are intentionally free of a specific network or failure model. E-Cast is designed for the Cloud, but in a different environment (e.g., inside a modern multi-core computer), different solutions of the two problems may be possible. The following is a practical network and failure model for the Cloud.

Processes execute asynchronously, and communicate via messages passed over asynchronous, quasi-reliable, loss-less FIFO channels. For quasi-reliable, loss-less FIFO channels, the following guarantees hold [ACT97]:

FIFO Order Delivery For any pair of messages m, m' sent by a process p to another process p'; if p sends m before m', then p' receives m before m'.

Loss-less Delivery For any pair of messages m, m' sent by a process p to another process p'; if p sends m before m' and p' receives m', then p' also receives m.

Quasi-reliability If a correct process p sends a message m to a correct process p', then p' eventually receives m.

Only the last guarantee, quasi-reliability, requires the sender of a message to be *correct*. The other two guarantees must hold regardless of failures. Quasi-reliable, loss-less FIFO channels are easy to implement in practice, using TCP or alternatively UDP with message sequence numbers (to enforce ordered delivery and detect message loss). Thus, we are not making unrealistic assumptions about the network here.

Processes may crash silently, but otherwise behave as specified (no Byzantine failures). We assume processes have access to inaccurate but complete failure detectors. That is, a process may suspect another, correct process of being faulty, but every faulty process is eventually suspected of being faulty by every correct process. Any failure detector that makes purely local decisions based on heartbeat / keep-alive messages and timeouts meets the criteria, so, again, we are not making unrealistic assumptions.

It is known that eventual weak accuracy (i.e., eventually *not* suspecting a correct process of being faulty) is formally required for liveness of uniform atomic broadcast and thus E-Cast [CT96]. The FLP impossibility result [FLP85] implies that such failure detectors are impossible in asynchronous networks. In essence, it is impossible to distinguish between a failed process and a process or network link that is just slow enough to cause a timeout. As is common practice in systems involving distributed consensus, we assume failure detectors are accurate enough to ensure liveness *in practice*.

Processes propagate failure suspicions by submitting messages. By definition, a process p is *faulty* iff p crashes, or a process p' submits a message declaring p faulty. This is a form of process-controlled crash [DSU04]. A single message submitted by a single process can declare a process faulty, but an application may run elaborate monitoring and voting protocols in the background before submitting such a message. Thus, we are not limiting the implementation of failure detectors in any way.

The existence of a user-defined function suspects(m) is assumed, which maps every message m to a set of processes which are declared faulty by m. It is assumed that if a correct process p suspects another process p' of being faulty, then p eventually submits a message m such that $p' \in$ suspects(m).

3.4 Main Protocol

This section is structured as follows. First, the protocol idea behind E-Cast is presented. Then, the main protocol is specified in pseudo-code. Finally, we discuss an important liveness guarantee called Quiescence.

3.4.1 Protocol Idea

E-Cast is specified in terms of two roles: application and router processes. Application processes $A \subseteq P$ submit and deliver messages. Router processes $R \subseteq P$ do not submit or deliver messages; they act as message sequencers and intermediaries between application processes. It is permissible for a single process $p \in P$ to be both an application and a router process.

Assume routers reach distributed consensus on an ever-growing m-seq Q. Based on Q, each router incrementally and deterministically constructs the same sequence of configurations. Based on application-defined logic, these configurations determine the destination sets of the messages in Q, which tell routers where to forward each message in Q.

Let application processes receive forwarded messages in the form $\langle m, t \rangle$, where $m \in M_Q$, and $t \in \mathbb{N}_0$ is the index of the message m in Q. Assume that $M_Q \subseteq M^\Sigma$. If every application process delivers messages in order of t, then Q is valid and delivery-ordered. If the order of t corresponds to the causal order of submission events, then Q is also send-ordered.

Let routers send every message $m \in M_Q$ exactly to the processes in $\mathrm{destsetpfx}(Q, m)$. If every correct process in the destination set of a message $m \in M_Q$ according to Q eventually receives m, and eventually delivers every unique message it receives, then Q is minimal, complete, and terminated. If Q is also gap-free, then Correct Routing holds and the protocol solves stateful routing. Dynamic partial replication is solved by plugging in the right configuration module, as described in the subsequent chapter on Rubberband.

A proper proof of correctness of E-Cast along these lines is provided in Appendix B.2.

3.4.2 Router Replication

The routers establish distributed consensus on a set of messages that are delivered, and a total order of delivery for these messages. To this end, they *propose* and *learn* messages using uniform atomic broadcast with FIFO delivery order. Each router maintains a growing m-seq of learned messages, which determines the messages to deliver and their order.

For a dynamic set of routers, this requires a solution to the reconfigurable state-machine problem [LMZ10] or (the equivalent) uniform atomic broadcast over a single dynamic group [Sch06]. We have implemented an efficient solution discussed in Section 3.5.4.

Algorithm 10: Application Process

```
state variables
    $t_{max} \leftarrow$ -1 ;        /* greatest timestamp of any delivered message */
    $W \leftarrow \langle\rangle$ ;        /* send window (submitted, unstable messages) */
upon submit $m$ do
    append $m$ to $W$
upon receive $\langle$ "stable",id($m$)$\rangle$ do
    remove $m$ from $W$
    acknowledge $m$ to user
upon receive $\langle$ "deliver", $m, t\rangle$ from $r$ do
    if $t > t_{max}$ then
        $t_{max} \leftarrow t$
        deliver $m$ to user
    send $\langle$"confirm",id($m$)$\rangle$ to $r$
periodically
    choose some router $r$
    foreach $m \in W$ do
        send $\langle$"route", $m\rangle$ to $r$
```

3.4.3 Application Process Algorithm

Pseudo-code for application processes is given in Algorithm 10. All the messages that the application submits are sent in submission order. Received messages are delivered to the application in timestamp order. When a message ⟨"stable", m⟩ is received, m was declared stable (i.e., uniformly delivered) by a router and can be acknowledged to the application and removed from the send window. All unstable messages are periodically sent to some router. Any router is safe, though liveness obviously requires that application processes eventually choose a correct (non-faulty) router.

Note that application processes do not have to wait for "stable"-messages between sending a sequence of "route"-messages. Many submitted messages can be pending at the same time (without loss of ordering and delivery guarantees), which is critical for achieving high protocol throughput.

Some obvious optimizations are missing from the pseudo-code for simplicity. Notably, assuming quasi-reliable FIFO channels, an application process a need not re-send any message m to a router r, if a had sent m to r before. So to avoid sending messages multiple times, an application process should only choose a new router if it learns that its current router has failed.

Algorithm 11: Router Process

```
state variables
    $Q \leftarrow \langle\rangle$ ;                          /* learned m-seq */
    $W \leftarrow \langle\rangle$ ;     /* send window (learned, unstable messages) */
    $C \leftarrow \langle\rangle$ ;   /* map of msgs to dest. procs that must confirm */
upon receive ⟨"route", $m$⟩ do
    if $m$ not stable then
        propose ⟨"route", $m$⟩ to all routers
    else
        send ⟨"stable",id($m$)⟩ to submitter($m$)
upon receive ⟨"confirm",id($m$)⟩ from $a$ do
    if $C[m]$ exists then
        $C[m] \leftarrow C[m] \setminus \{a\}$
upon learn ⟨"route", $m$⟩ from some router do
    if $m$ not in $Q$ then
        append $m$ to $Q$ and $W$
upon learn ⟨"stable", id($m$)⟩ from some router do
    remove $m$ from $W$ and remember $m$ as stable
periodically if self is leader then
    foreach $m \in W$ where $C[m]$ does not exist do
        $C[m] \leftarrow$ destsetpfx($Q$,$m$) $\setminus$ suspects($Q$)
        $t \leftarrow$ the index of $m$ in $Q$
        foreach $a \in C[m]$ do
            send ⟨"deliver", $m, t$⟩ to $a$
periodically foreach $m$ where $C[m]$ exists do
    $C[m] \leftarrow C[m] \setminus$ suspects($Q$)
    if $C[m] = \{\}$ then
        propose ⟨"stable",id($m$)⟩ to all routers
        destroy $C[m]$
```

3.4.4 Router Process Algorithm

Pseudo-code for router processes is given in Algorithm 11. Whenever a router *receives* a message that is not known to be stable, it *proposes* it by atomically broadcasting it to all routers, including itself. Whenever a router *learns* a new message m, it appends m to its m-seq Q and send window W. The lead router (see below), periodically sends all messages in its send window to their respective destination sets.

When all application processes in the destination set of a message m have confirmed m or have become suspects, m is proposed stable via atomic broadcast. When a router learns that a message m is stable, it can garbage-collect all state related to m. To later identify m as stable, an implementation may use session and sequence numbers.

For efficiency, only one router (the *leader*) should forward each message to its respective destination set. Given an atomic broadcast protocol between routers, leader election is easy to implement. However, leader election is purely an optimization. Any router may safely consider itself a leader at any time.

Again, note that routers can handle an arbitrary number of pending (unstable) messages. The implementation we present in Section 3.5 is fully pipelined.

Also, the pseudo-code omits many obvious optimizations for simplicity. For instance, a router need not re-propose a message m which is already in Q when m is received. Further, to avoid application processes having to repeat "route"-messages in order to receive a "stable"-message, a router process can safely send a message $\langle \text{"stable"}, \mathrm{id}(m)\rangle$ to the application process that submitted m, as soon as the router process learns $\langle \text{"stable"}, \mathrm{id}(m)\rangle$. For efficiency, an implementation should let a designated router process do this (e.g., the leader). Other optimizations found in our implementation are described in Section 3.5.7.

3.4.5 Quiescence

In addition to Correct Routing, E-Cast satisfies the following, useful liveness property.

Quiescence. *For any message $m \in M_a^\Sigma$ submitted by a correct application process $a \in A$, eventually the application process is delivered a message $\langle \text{"stable"}, \mathrm{id}(m)\rangle$.*

Quiescence ensures that stable messages are eventually acknowledged to the application, and, implicitly, that all state related to stable messages is eventually garbage collected. Acknowledgements are also useful for user feedback (durability) and flow control (limiting the number of unstable messages). A proof of both Correct Routing and Quiescence is provided in Appendix B.2.

3.5 Implementation

In this section, we give an overview of our reference implementation of E-Cast. The E-Cast protocol has been implemented in roughly 10,000 lines of Erlang [Arm10], a functional, concurrency-oriented language. The Erlang process and distribution model closely matches

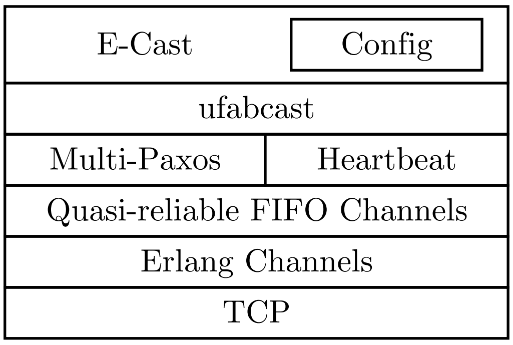


Figure 3.3: *E-Cast Protocol Stack*

our formal model, so Erlang was a natural choice of language. The codebase of the implementation is substantially larger than the pseudo-code in the previous section might suggest, because the implementation has to deal with many real-world issues such as failure detection, distributed garbage collection, and congestion control.

3.5.1 Protocol Stack

Figure 3.3 visualizes the protocol stack of our E-Cast implementation. Proceeding bottom-up in the stack, TCP is abstracted away by the Erlang process and channel model. Erlang processes communicate through asynchronous FIFO channels. Within a single virtual machine instance, Erlang guarantees that the channels are loss-less, quasi-reliable FIFO channels. But across virtual machine instances, only FIFO delivery is guaranteed. Temporary link failures can cause messages to be lost. Where necessary, we added sequence numbers to messages to detect and recover from message loss.

Using the resulting loss-less, quasi-reliable FIFO channels, we have implemented a wait-free, reconfigurable variant of Multi-Paxos, which is used for uniform FIFO atomic broadcast (ufabcast), described in more detail in Section 3.5.4. Ufabcast in turn is used to replicate router processes; more precisely, ufabcast is used to replicate a common m-seq between router processes, see Algorithm 11. To detect router failures, and for garbage collection of stable Paxos instances, ufabcast uses a simple heartbeat protocol, discussed in Section 3.5.5. In order to tell router processes where to forward application messages, applications must plug a user-defined `Config` module into the routers. The nature of this module is described in Section 3.5.3.

Router membership is implemented entirely on the level of ufabcast. In order to join or part, routers atomically broadcast special join or part commands to each other which are not seen by the application. This mechanism is explained in more detail in Section 3.5.4.

Application membership on the other hand is determined by the sequence of application messages sent through E-Cast, as linearized by the router processes (through ufabcast). In other words, router and application membership are independent of one another. There is no circular dependency, which is essential for the modularity of the protocol stack, and for the usability of E-Cast.

Regarding the choice of Erlang as an implementation language, we found that Erlang is a good fit for the formal model, but is not very efficient when it comes to number crunching and maintaining complex data structures. Achieving high protocol throughput thus required considerable engineering effort. In accordance with good Erlang design practices [Arm10], each layer of the E-Cast protocol stack is implemented by at least one separate Erlang process. The layers communicate with each other exclusively by message passing over Erlang channels, forming a process pipeline of considerable depth.

This design minimizes the amount of state maintained by each Erlang process (and state is expensive in Erlang), and entails a high degree of parallelism. As the experimental results in Section 4.6 show, this parallelism allows E-Cast to compete performance-wise with protocols implemented in potentially more efficient languages such as C or Java.

3.5.2 Leader Election

Under normal operation, application processes communicate only with the so-called "lead" router. This lead router is not elected explicitly. Instead, the *oldest* router in the system, in join order, is defined to be the leader. The join order of routers is thus also called the *succession order*.

Whenever a router joins or parts, the (possibly new) lead router sends out the latest, timestamped succession order to all application processes. Thus, all processes in the system agree on who the lead router is, except during the brief periods where the router membership changes and not all processes have yet been informed. Even then, disagreements on leadership do not violate safety of the protocol, they only affect performance (cf. binary exponential backoff in Section 3.5.5).

3.5.3 Config Modules and Message Routing

E-Cast applications are required to implement an abstract data type called `Config`, which corresponds to the *config* structure of the formal application model (Section 3.3.1).

A `Config` object has the following interface: apply a message (yielding the next `Config`), get the destination set of a message, get the suspect set of a message, and get the current set of application processes. These functions are sufficient for arbitrary membership and routing logic, from atomic broadcast to dynamic partial replication.

Every router holds an `MSeq` object, replicated via ufabcast. An `MSeq` object is a sequence of `Config` objects, created by successively *applying* messages learned through ufabcast. When a new router joins, the existing routers send the new router their (identical) `MSeq` objects. The `MSeq` object implements all the functions used in Algorithm 11, in particular the destset and suspects functions.

To compute the destination set of a message m, the `MSeq` object finds the corresponding `Config`, and calls the `Config`'s destination function, passing m as an argument. The suspect set of a message is computed in an analogous manner.

The implementation of `MSeq` objects is part of E-Cast. It mainly involves indexing (by message ID) and garbage-collecting the sequence of `Config` objects created by successive application of messages. Only the `Config` module must be provided by users of E-Cast.

In summary, the `Config` module represents a piece of user-defined business logic, plugged into the reconfigurable, replicated state machine formed by the router processes. The `Config` module itself defines an automaton, where applying a message transitions the automaton into the next state, and the destination and suspect sets are outputs of the automaton. The fact that Erlang is a non-destructive (variables can be assigned only once), garbage-collected language makes it easy to implement `Config` modules correctly.

In Section 4.3 of the next chapter on Rubberband we describe a concrete `Config` module which routes messages in a manner that solves dynamic partial replication by relying on the ordering and delivery guarantees of E-Cast.

3.5.4 Uniform FIFO Atomic Broadcast

As mentioned repeatedly, E-Cast routers propose and learn messages through a uniform FIFO atomic broadcast (ufabcast) protocol. This protocol itself is implemented as a reconfigurable, replicated state machine based on a wait-free variant of Multi-Paxos, where every proposed message is a command to the state machine.

Wait-free Multi-Paxos

When a router proposes a message, the ufabcast protocol assigns to the message a monotonically increasing, local sequence number, and creates a corresponding state-machine command. That command is then proposed through a novel, wait-free variant of Multi-Paxos. We use the term *wait-free* in the sense of asynchronous and pipelining, not in the sense of wait-free parallel algorithms.

The original Multi-Paxos protocol is due to Prisco et al. [PLL97]. Our improved implementation of this protocol relies on the ordering and delivery guarantees of the quasi-reliable, loss-less FIFO channels between processes to avoid waiting for acknowledgements between the sending of successive phase 2 ("accept") messages.

When Multi-Paxos accepts a command, the command is passed on to the upper protocol layer for execution. The upper layer reorders the messages inside executed commands according to the proposer's sequence number (to re-establish FIFO order). If, after reordering, there are no missing messages by the given proposer (as evident from the sequence numbers), the messages are forwarded to the local router process, to be learned and applied to the local `MSeq` object.

Because Multi-Paxos guarantees that every message is learned uniformly, in some total order, and the upper protocol layer reorders the message sequence delivered by Multi-Paxos in a deterministic fashion that establishes FIFO order, every message learned by a router process is learned uniformly, in total FIFO order.

Reconfiguration and Fault Tolerance

When a router detects the failure of another router, or receives a join request from a new router, it proposes a special "join" or "part" command. When such a command is executed, the ufabcast state machine enters a new *view* (epoch) with updated membership, discarding any non-executed commands of the old view. Discarded commands are promptly re-proposed in the new view, unless they are redundant "join" or "part" commands that have become obsolete. This is an implementation of the *stop-sign method* for reconfiguring a state machine, formally described by Lamport et al. [LMZ10].

Every router executes the same commands in the same order. Thus, all routers hold the same, replicated `MSeq` object at the point where a given "join" command is executed. Whenever a router executes a "join" command, it sends the joining router its current `MSeq` object[8]. The new router waits until it receives an `MSeq` object. Other messages which the new router learns before it receives an `MSeq` object are not discarded, but safely buffered. After the new router has received its `MSeq` object, it applies the buffered messages and is henceforth in the same, replicated state as all the other routers.

When a router crashes or becomes unresponsive, another router will eventually detect the failure and propose a "part" command. At the point where a router learns a "part" command regarding some other router, it can garbage collect all state pertaining to that

[8]It is sufficient for *one* router to send its current `MSeq` object (all routers hold the same `MSeq` object). For extra liveness, our implementation has *all* routers send their latest `MSeq` object.

router, such as obsolete consensus instances and messages from executed commands that have not been forwarded/learned (because of preceding, missing messages in FIFO order).

To summarize, in contrast to other atomic broadcast protocols such as Zab [JRS11], E-Cast does not rely on process recovery from stable storage (fail-recovery model). Any router that is ever suspected of failure can be parted and replaced without threatening safety of the protocol (we discuss liveness below). Consequently, E-Cast can tolerate *permanent*, *silent* failure of routers (fail-silent model).

As regards liveness, the protocol can make progress as long as a *majority* of routers in the latest view is live. (This degree of liveness is indeed optimal for any system that requires distributed consensus in an asynchronous network, as proven by Gilbert and Lynch [GL02].) So, for example, if there are 5 routers in the latest view, then at least 3 are necessary for distributed consensus, which means at most 2 routers may crash before the remaining routers are able to reach consensus on a new view. As we show in Section 4.6.9, router fail-over requires only a few seconds in our implementation.

3.5.5 Garbage Collection and Failure Detection

To route an infinite sequence of messages with finite memory, all state related to stable messages must eventually be garbage collected, along with related state such as `Config` objects. The fact that Erlang is a garbage-collected language is of little help with this problem, because Erlang does not know which messages are stable, and what related meta-data is obsolete. The necessary *distributed* garbage collection logic has to be implemented explicitly at all levels of the protocol stack, in a fault-tolerant, safe, and live manner.

At the level of the main protocol, garbage collection is based on "stable"-messages, which are atomically broadcasted by the lead router when it learns that a specific message has been confirmed by all unsuspected destination processes. Whenever some router learns a "stable"-message of the form $\langle$"stable"$, m\rangle$, it informs its `MSeq` object that m is stable. The `MSeq` object garbage-collects the stable prefix of the message history (m-seq) it contains, including the induced, obsolete `Config` objects.

Figure 3.4 shows the logical structure of an `MSeq` object, and how garbage collection operates. The connection lines between messages and `Config` objects indicate which `Config` applies to which message; i.e., which config is used when computing the destination set or suspect set of a given message. The gray-shaded circles and rectangles depict stable messages and stable `Config` objects, respectively. The stable prefix of the m-seq will never have to be routed again, so these messages and the induced `Config` objects can safely be garbage collected.

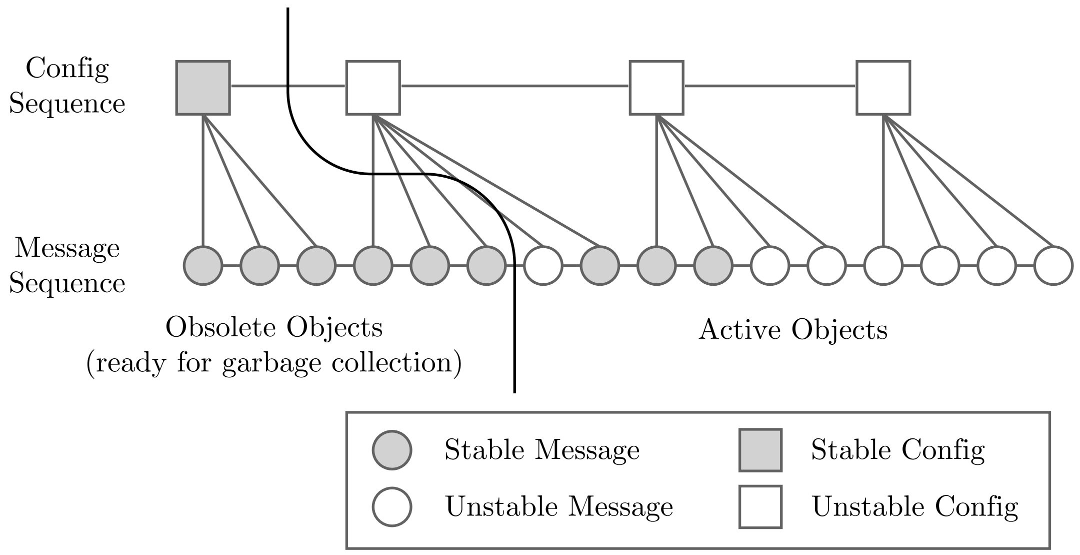


Figure 3.4: *Garbage Collection inside* `MSeq` *Objects*

Garbage collection of state-machine commands and Multi-Paxos instances is implemented via periodic (default: 250 msec) heartbeat messages between routers. A heartbeat message contains the sender's ID, as well as the Paxos instance number of the latest state-machine command that has been executed by the sender. All state related to commands numbered lower-or-equal the lowest, latest executed command of any non-faulty router can safely be garbage collected.

The heartbeat protocol is also used for push-style failure detection. When a router r does not receive heartbeat messages from another router r' for a configurable period of time (default: 3 seconds), r proposes a "part" command regarding router r'. It is not possible for 2 routers to "part" each other following a link failure, because one of the two commands is guaranteed to be executed first, and subsequent commands by the parted router are ignored.

This simple failure detection scheme works well in a LAN environment, but there is obviously room for improvement, such as making the failure timeout adapt to the network quality and topology [SPU11, HDYK04, HTC05]. That being said, the experimental results in Section 4.6 show that E-Cast allows router processes to join or leave in a matter of seconds, even under full load. A live router that suddenly finds itself parted can quickly rejoin. Thus, occasional false positives caused by inaccurate failure detectors are not a serious problem.

3.5.6 Flow and Congestion Control

The E-Cast protocol implementation is heavily pipelined, both across and within application and router processes. At any given moment, thousands of messages may be unstable, and thousands of concurrent consensus instances may be active. To avoid thrashing and maximize throughput, a number of flow control and congestion control measures had to be implemented.

Our extended Multi-Paxos implementation uses *binary exponential backoff* in case of conflicts between concurrent proposals, analogous to the classic Ethernet multi-access scheme [MB76]. This breaks symmetry in case of multiple proposers and thus prevents live-locks during the (brief) periods when router membership changes and multiple routers are proposing commands, thereby overcoming the FLP impossibility result [FLP85].

Furthermore, the number of active (not-yet-garbage-collected) consensus instances is limited (default: 32,768), using information from the heartbeat protocol described previously. As a result, a slow router cannot fall behind too far in the execution of commands. This avoids thrashing on the maintenance overhead of active consensus instances, and also bounds the amount of protocol state that has to be transferred in case of a router crash. (After the crash of a router, the remaining routers must exchange state in order to guarantee uniform learning of commands proposed by the crashed router.) Note that "part" commands override the limit, they are always immediately broadcasted. Otherwise the protocol could dead-lock in case a router crashed and the limit of active consensus instances had already been reached.

Even with a high-throughput consensus protocol in place, a fast application process (or a large number of slow application processes) can still overwhelm the lead router process with "route"-messages. E-Cast holds meta-data for each unstable (in-transit) message, so letting the number of unstable messages grow without bounds could quickly lead to congestion, thrashing, and out-of-memory errors. To avoid congestion, we use a combination of sliding-window flow control and source throttling with explicit congestion notifications.

Our implementation of the application process algorithm (page 108) limits the number of pending (unacknowledged) messages to a multiplicatively growing and shrinking congestion window. The window is initially very small (default: 1 message). Periodically (default: 500 msec), the window size is doubled, up to a user-defined maximum (default: 1,024 messages). Conversely, when a congestion notification is received, the window size is halved. This scheme is an instance of multiplicative-increase/multiplicative-decrease (MIMD) flow control [CJ89], which we have chosen because it adapts very quickly to the available network and processing resources.

Each router process maintains statistics concerning the number of unstable messages for each application process. Periodically (default: 250 msec, half the growth period of application process congestion windows), the router looks at the statistics. When the total number of unstable messages exceeds a user-defined congestion threshold t (default: 1,024 messages), the router divides the threshold value t by the number of application processes n. Each application process a whose individual number of unstable messages m_a exceeds its fair quota t/n is sent a congestion notification.

As a concrete example, assume the congestion threshold is set to $t = 1,000$. There are $n = 4$ clients, where $m_1 = 100$, $m_2 = 200$, $m_3 = 400$, and $m_4 = 800$. The total number of unstable messages is $100 + 200 + 400 + 800 = 1,500$ which exceeds the threshold. The fair quota $t/n = 1,000/4 = 250$ is calculated, and clients 3 and 4 who exceed the quota are sent congestion notifications. As a consequence, they will shrink their congestion windows, which will implicitly throttle their send rate.

Because throttling is based on explicit congestion notifications, this scheme is not suitable for very high latency links (the feedback delay may become too high, leading to oscillation). However, our experimental results in Section 4.6 indicate that the scheme is very fair and stable in LAN and data center environments, for a wide range in the number of clients (1 to 65,536), even when the network is saturated.

3.5.7 Optimizations

In this section, we describe a number of important optimizations that have been added to our implementation of E-Cast. We also discuss future optimizations that have been identified but not yet implemented.

Delayed Acknowledgements

Unlike Algorithm 11 suggests, routers need not atomically broadcast “stable”-messages, because the order in which these messages are received is irrelevant. Instead, any scheme that guarantees that every router eventually learns of every stable message will guarantee Quiescence (Section 3.4.5) and thereby eventual garbage collection. So instead of atomic broadcast, a *gossip protocol* (specifically, a *dissemination protocol* [DGH$^+$87]) could be used to spread this information, as proposed by Guo et al. [GHvR$^+$97].

That being said, introducing additional protocols increases system complexity considerably and is a source of functional and performance bugs. The simpler solution we have chosen is to buffer “stable”-messages up to a configurable period of time (default: 10 msec). If a “route”-message is proposed before the timeout, the buffered “stable”-messages are piggybacked onto the “route”-message. We call this idea *delayed acknowledgements*, because of its conceptual similarity to the corresponding optimization found in TCP.

Message Clumping

Another optimization we have implemented is *message clumping*. Routers clump proposed (logical) messages together into bigger (physical) messages, in order to reduce the number of atomic broadcast instances per (logical) message in the m-seq. To do this automatically, proposed messages are not atomically broadcasted immediately, but are buffered up to a configurable period of time (default: 10 msec). If enough messages have been proposed (default: 3), or if the timeout is hit (whichever happens first), the buffer is flushed and the flushed messages are clumped and atomically broadcasted together.

As we show in Section 4.6, delayed acknowledgements and message clumping together roughly double the maximum throughput of E-Cast for small messages (where E-Cast is compute bound on the atomic broadcast protocol). On the downside, the two optimizations increase latency under low throughput. In the future, an improved implementation of E-Cast could adaptively enable and disable the two optimizations depending on the load. This would not affect the actual protocol and should therefore be easy to add.

Reads and Unsafe Messages

We have not really discussed *reads* in this chapter; i.e., messages that are known not to change the system state. Obviously, reads can be sent through E-Cast just like any other message. To this end, the user-defined destination function of `Config` objects can be made to distinguish between read and write messages, in order to implement read-one-write-all (ROWA) semantics or even quorum reads. Rubberband, discussed in the next chapter, implements ROWA replication with sequential consistency (as defined by Fekete and Ramamritham [FR10]) in exactly this manner.

However, certain (expensive) guarantees of E-Cast, such as uniform delivery, are not necessary for reads, because nothing serious happens if a read is lost. Depending on the application, even ordering guarantees may be relaxed for reads. Frankly, we have not properly explored the design space at this point, but one simple protocol extension has already been implemented. Application processes can request *unsafe* delivery when submitting a message. When a router receives such an unsafe message, it does not atomically broadcast it to the other routers. It also does not apply the message to the `MSeq` object (thus, unsafe messages cannot change the system configuration). Instead, the router just computes the destination set of the message, and forwards the message to each destination process, along with the timestamp of the latest regular (safe) message that had been forwarded to the respective destination process.

Assuming the application processes are submitting all their messages (safe or unsafe) to the lead router, the fact that the lead router communicates with all destination processes through quasi-reliable FIFO channels means that the timestamps of safe messages and subsequent unsafe messages match. Only if the application processes receive safe messages from another router—which should only happen in case the router leadership changes, see Section 3.5.2—will the timestamps not match. The application sees the timestamps of delivered messages, so it can simply drop unsafe messages where detected message reordering is a consistency problem. In the next chapter, we discuss how Rubberband uses unsafe messages to implement efficient *snapshot reads*.

Future Optimizations

The strict modularity of the protocol stack allows improvements to be made in a localized, incremental manner. For example, the existing failure detector is based on fixed timeouts, which may lead to false positives on a flaky network. The failure detector can be improved or completely replaced without changes to the rest of the protocol stack. Similarly, the flow and congestion control scheme is not optimized for slow, wide area networks, and could be adapted for use cases that require it.

More importantly, the current implementation of E-Cast does not distribute the load evenly among router processes. Most of the work is done by the lead router, which consequently forms a throughput bottleneck. To better scale with the number of router processes, the atomic broadcast protocol could be replaced with a (somewhat) scalable atomic broadcast protocol such as Ring-Paxos [MPSP10], without changes to the main E-Cast protocol.

Likewise, the main protocol could distribute the message-forwarding load among routers, by assigning message-forwarding responsibility to routers in a round-robin fashion. Application processes would then receive projections of the original message sequence from distinct routers, from which they would have to reassemble the original sequence. All that is required for each message is the timestamp of the immediate predecessor message in the original sequence. This information is available to every router (since there is distributed consensus on the entire m-seq and the destination set of each message) and could easily be forwarded to the application processes. However, as long as the atomic broadcast between routers is the main bottleneck, this optimization is unlikely to have a large effect, so we have not yet implemented it.

3.6 Concluding Remarks

In this chapter, we have presented a dynamic, uniform, causal total order multicast protocol for asynchronous networks: E-Cast. In contrast to other multicast protocols, E-Cast does not require users to explicitly create and maintain process groups. Instead, message routing is based exclusively on a user-defined function over the past history of messages. We have formalized this idea, have shown how it enables dynamic partial replication with strong consistency, and have presented its efficient algorithmic solution: E-Cast.

The implementation of E-Cast we have described deals with all the difficult details that arise in real-world protocol design, such as leader election, distributed garbage collection, failure detection, and congestion control. The router processes in E-Cast form one of the very few implementations of a fully reconfigurable state machine.

The entire protocol stack, including the reconfigurable state machine implementation, is highly modular and extensible, leaving room for future generalizations, optimizations, and combinations with other protocols.

Chapter 4

Rubberband

The popularity of the Cloud as a platform for application deployment, and the extreme requirements of some applications in terms of data size, transaction throughput, and availability have sparked a proliferation of specialized storage systems. Earlier examples of cloud storage systems, most notably Amazon Dynamo [DHJ$^+$07], sacrifice data consistency in favor of performance and availability. Newer systems such as MegaStore [BBC$^+$11], Spinnaker [RST11], or Scalaris [SSR08] have shown that strong consistency—if implemented well—does not preclude high performance.

The widespread deployment of such strongly consistent systems is currently hampered by three factors. First, most systems rely on stable storage, infallible clients, or infallible coordinator processes for fault tolerance and recovery. Second, many systems require application developers to manually (and aggressively) split their data and workload into units of consistency. And third, all the existing systems we are aware of are not truly *elastic*. While there are systems that can be scaled to large numbers of storage processes, it is difficult or impossible to dynamically change the assignment of data objects to storage processes and units of consistency without loss of consistency or system availability.

In this chapter, we present a data store framework that solves these three problems and also offers very high performance: Rubberband. At the core of Rubberband is the E-Cast protocol described in the previous chapter. In Section 3.3.3, we have presented a formal reduction of dynamic partial replication to stateful routing, the problem solved by the E-Cast protocol. Rubberband is a concrete implementation of this reduction on top of E-Cast, written in about 9,000 additional lines of Erlang.

Using Rubberband, any plain storage engine can be made fault-tolerant and elastic with moderate effort. We have combined Rubberband both with Crescando, the scan-based relational table engine described in Chapter 2; and with Memcached [Fit04], a simple key-value store. Both Crescando and Memcached are main-memory engines, and adding consistency, elasticity, and fault tolerance at the distribution layer allows one to use these systems as an efficient “Tier 0” store rather than a cache. The combination of Rubberband and Crescando, together called *Crescando/RB*, solves all the Amadeus use cases described in Section 1.1 and is discussed in additional detail in Chapter 5.

In summary, Rubberband

- delivers writes uniformly, in causal total order,
- even arbitrarily overlapping multi-key or range writes (yields sequential consistency of the entire system, as defined by Fekete and Ramamritham [FR10]);
- is completely wait-free (clients can safely submit a high-rate stream of writes without waiting for confirmations);
- elastically scales to large numbers of storage processes;
- allows data stores built on Rubberband to be continuously available, even during reconfiguration (data repartitioning or failed process replacement);
- tolerates permanent, silent failure of any process;
- does not require stable storage;
- does not require perfect failure detectors; and
- is independent of a specific data model or storage engine.

Full support for multi-key and range writes means users do not need to manually split their data and workload into units of consistency. Dynamic partial replication enables elastic scale-out, and the non-dependence on stable storage and perfect failure detectors significantly improves practical adoptability in the Cloud.

The remainder of this chapter is organized as follows. Section4.1 discusses the state of the art and related work. Section 4.2 gives an overview of the design of Rubberband. Section 4.3 shows how Rubberband is tied together with the E-Cast protocol. Section 4.4 explains how Rubberband solves the important problem of system reconfiguration for elasticity and fault tolerance. Section 4.5 discusses the consistency guarantees available in Rubberband, and how the system can be extended to better scale with the workload. Section 4.6 shows the results of an extensive experimental evaluation of E-Cast and Rubberband. Section 4.7 concludes the chapter.

4.1 State of the Art

We are obviously not the first to build a data store (framework) that uses partial replication. In this section, we present the most influential work in this context, and discuss how it pertains to Rubberband.

4.1.1 Distributed Hash Tables

Distributed hash tables (DHTs) are a class of distributed systems which implement a dynamic key-value mapping, similar to a regular hash table (hence the name). DHTs are assumed to be highly decentralized, fault-tolerant, and scalable in order to meet the definition [BKK+03]. Such a system is useful for a variety of purposes, such as domain name resolution, peer-to-peer file sharing, or web caching, to name a few.

Arguably the most influential DHTs are CAN [RFH+01], Chord [SMK+01], Pastry [RD01], and Tapestry [ZHS+04]. These systems differ primarily in how the key-value mapping is partitioned across processes, and how key-to-value lookup is performed.

To partition the key domain over a dynamic set of processes, most DHTs use a variant of *consistent hashing* [KLL+97]. For example, Chord [SMK+01] arranges all processes in a logical ring, where each process is assigned a single, unique key (the process *identifier* or ID). A process is made responsible for all entries whose keys fall in the range between its own ID, and the ID of the immediate successor process on the logical ring of processes (in order of ID). In Chord, key-to-value lookup requires $O(log(n))$ hops along the ring on average, where n is the number of processes on the ring.

The Rubberband framework we present in this chapter uses the same consistent hashing scheme as Chord. But in contrast to Chord, Rubberband does not route lookup requests along the ring of processes. Instead, E-Cast is used to route messages.

4.1.2 NoSQL Data Stores with Eventual Consistency

The emergence of large-scale, cloud-based services has created a need for highly scalable, fault-tolerant data stores. Traditional relational database management systems (RDBMS) are generally not designed to scale horizontally or tolerate frequent, permanent machine failures. There are exceptions, such as Oracle RAC [ATJK09] or MySQL Cluster [RT04], but performance and manageability of these systems are generally perceived as poor [Cat10], due to the excessive complexity and feature set of distributed

RDBMS. So in response to these problems, a number of novel systems have been developed, which radically simplify the programming model (query language, data model, consistency model) in order to enable a higher degree of parallelism, fault tolerance, and performance. The Crescando/RB system presented in this dissertation is an example of such a system.

This class of systems is often referred to as “NoSQL” data stores, which is a play of words on the fact that these systems do not aim to support the same functionality—SQL—as traditional RDBMS [Cat10]. The simplest type of NoSQL data store, a *key-value store*, offers essentially the same API as a distributed hash table: read a value for a given key, write a value for a given key, and delete a given key.

Most NoSQL data stores abandon strong consistency (ACID, synchronous replication) in favor of weak or *eventual consistency* (BASE, asynchronous replication), in order to achieve higher scalability, availability, and/or performance. The research project that pioneered the idea of an eventually consistent data store was Bayou [TTP+95]. Well-known, modern examples of eventually consistent data stores are Amazon Dynamo [DHJ+07], Amazon SimpleDB [Hab10], Apache Cassandra [LM10], and Yahoo! PNUTs [CRS+08].

There appears to be no universally agreed-upon definition of eventual consistency, but the least common denominator of the definitions found in the literature essentially states that given a sufficiently long period of time over which no writes are submitted, an eventually consistent system ensures that eventually all writes propagate through the system, which eventually reaches an *internally consistent* state [Vog09, FR10]. This is equivalent to the idea of *anti-entropy* and *internal convergence* of gossip protocols (epidemic protocols) as originally proposed by Demers et al. [DGH+87]. The latter paper is a precursor to Bayou.

Under *no* definition of eventual consistency are there *absolute guarantees* in terms of some total order in which the effects of concurrent, conflicting writes become visible[1]. Concrete systems with eventual consistency offer at best *statistical guarantees* of data consistency with respect to concurrent writes.

4.1.3 NoSQL Data Stores with Strong Consistency

In some use cases, such as the Amadeus Ticket table which contains flight reservation data, eventual consistency is not good enough. While users may be able to live without SQL and other conveniences of traditional database management systems, occasionally

[1]Establishing such a total order amounts to solving distributed consensus as discussed in Section 3.2.2. Solving distributed consensus marks the difference between strong consistency and weak consistency models such as eventual consistency [FR10].

forgetting the fact that a plane ticket has been paid and confirmed is not an acceptable system "feature". For these types of use cases, a number of NoSQL data stores with strong consistency guarantees have been proposed in the recent years.

One common approach found in the literature is extending a DHT or key-value store with agreement protocols, particularly Paxos [Lam98]. Examples of this approach are Spinnaker [RST11], ElasTraS [DAEA09], G-Store [DAEA10], and Google MegaStore [BBC$^+$11].

Spinnaker [RST11] and ElasTraS [DAEA09] guarantee strong consistency for *single-key* reads and writes. G-Store [DAEA10] is a key-value store framework that provides strong consistency for dynamically created but *non-overlapping key groups*. MegaStore [BBC$^+$11] is an extension of BigTable [CDG$^+$08] which offers an equivalent abstraction to G-Store's key groups called *entity groups*.

While all four systems scale well and provide "Paxos-strength" consistency guarantees, the big restriction is that key groups and entity groups must not overlap. So neither Spinnaker, nor ElasTraS, nor G-Store, nor MegaStore allow the example updates given in the informal problem statement in Section 3.1. With the notable exception of ElasTraS, it is also not clear how to repartition or re-group the data in these systems without taking them off-line or breaking consistency. Finally, all four systems require stable storage as part of their network protocols (they use write-ahead-logging to recover from process failures), making it hard to deploy the systems in a public cloud.

In contrast to Spinnaker, ElasTraS, G-Store, and MegaStore, Rubberband supports online repartitioning without loss of consistency, does not require manual partitioning of the data and workload into units of consistency (key or entity groups), and does not rely on stable storage.

Strongly consistent NoSQL data stores with support for *arbitrarily overlapping* multi-key or range operations are rare, but they exist. We are aware of 4 such systems: ecStore [VCO10], CloudTPS [WPC10], Scalaris [SSR08], and P-Store [SSP10]. In contrast to Rubberband, these four systems support read-modify-write transactions (like traditional databases), rather than atomic CRUD operations (like Crescando, cf. Section 2.2). On the other hand, they all suffer from specific shortcomings that make them inapplicable to our use case.

ecStore [VCO10] offers strong consistency for arbitrary, overlapping range queries, just like Rubberband. But while Rubberband relies on read-one-write-all (ROWA) replication, ecStore uses a combination of primary-backup replication and quorum-based voting. For a replication factor of 3 for instance, ecStore always reads and writes exactly 2 replicas (= a quorum) of a data item. In the case of writes, 1 written replica has to be the primary copy. Consistency follows from the fact that the quorums of every pair of operations overlap [OV99]. This rather complicated scheme has certain advantages in terms of write availability (one backup replica is allowed to be unreachable). The big downside is that quorum reads require *multi-version duplicate elimination.* Indeed, the results of Vo et al. [VCO10] show that ecStore performs poorly for range queries.

More importantly, ecStore uses Two-phase Commit (2PC) rather than a Paxos-based protocol, so the system may block indefinitely in case of permanent process failures [BHG87]. Finally, ecStore is not wait-free (clients cannot submit more than one transaction at a time without losing ordering guarantees), and requires disks for write-ahead-logging of 2PC.

CloudTPS [WPC10] is based on SimpleDB [Hab10] or, alternatively, BigTable [CDG$^+$08]. The system is interesting because it extends existing, "battle-proven" data stores with strong consistency guarantees. But like ecStore, CloudTPS uses 2PC, so it cannot tolerate permanent process failures. Also, it is not wait-free, and Zhou et al. [WPC10] do not claim CloudTPS is elastic, even tough the underlying data stores SimpleDB and BigTable are.

In contrast to the systems mentioned previously, Scalaris [SSR08] uses an extension of Paxos Commit [SRHS10] rather than 2PC. Consequently, Scalaris can tolerate permanent process failures like Rubberband. However, Scalaris is not wait-free; in fact, it uses distributed locking. Given that locking leads to severe performance problems under the workloads we consider in this dissertation (cf. Section 2.10.3 on Crescando versus MySQL), there is no reason to assume that *distributed* locking would improve matters. It is further not clear how to change the data partitioning in Scalaris without losing both read and write availability of the system for a significant amount of time.

ecStore, CloudTPS, and Scalaris tightly integrate 2PC or Paxos Commit into their concurrency control and replication protocols. In contrast to this, P-Store [SSP10] follows the middleware-based replication approach of Rubberband. That is, like Rubberband, P-Store is based on a self-contained atomic multicast protocol. But rather than multicasting individual operations, P-Store uses the atomic multicast protocol to reach agreement on *transaction commit order.* Agreement on transaction commit order is sufficient to implement 1-copy serializability, as proven by Raz [Raz94]. However, P-Store is not elastic, and like ecStore, CloudTPS, and Scalaris, it is not wait-free.

That being said, P-Store is exemplary in its theoretic soundness and architectural modularity, and its atomic commit protocol could be combined with E-Cast and Rubberband to yield an elastic system that supports general database transactions. To make the system wait-free with only minimal restrictions to the transaction model (namely determinism), ideas from Thomson and Abadi [TA10] could be applied.

In summary, there are now a number of NoSQL data stores that offer strong consistency guarantees. Some of them feature a richer transaction model than Rubberband. However, Rubberband is the only system that is both wait-free and also supports range reads and range writes, and techniques are known to generalize the transaction model of Rubberband without sacrificing its unique features. Furthermore, Rubberband is the only data store framework that is elastic, strongly consistent, and continuously available at the same time. Because Rubberband is not tied to a specific data or query model, it makes these features available any storage engine, including Crescando. And because Rubberband does not require stable storage, it is very easy to deploy in the Cloud.

Despite or because of these improvements over prior art, the experimental results in Sections 4.6 and 5.6 show that Rubberband is highly competitive in terms of throughput and latency compared to the systems mentioned here.

4.2 Design Overview

In this section, we give an overview of the various process and message types that Rubberband defines. We explain how Rubberband partitions and replicates the application data for scalability and fault-tolerance, and we describe how Rubberband detects and reaches agreement on process failures.

4.2.1 Process Types

Rubberband defines and implements the following three types of processes, each of which is also an E-Cast application process.

Client Processes submit *read* and *write messages* on behalf of the application. Depending on the type of message and the chosen isolation level (cf. Section 4.5), an individual read or write may be transmitted through E-Cast, or directly sent to one or more storage processes. An application can spawn and use any number of client processes. A typical design would be to use one client process per user session, but this mapping is up to the application.

Storage Processes each store a partition of application data, as explained in the subsequent section. Delivered read and write messages are forwarded to the user-defined storage engine (for example Crescando) in an order that preserves consistency. Conversely, result tuples emitted by the storage engine are reliably streamed back to the respective client process, which in turn delivers them to the application.

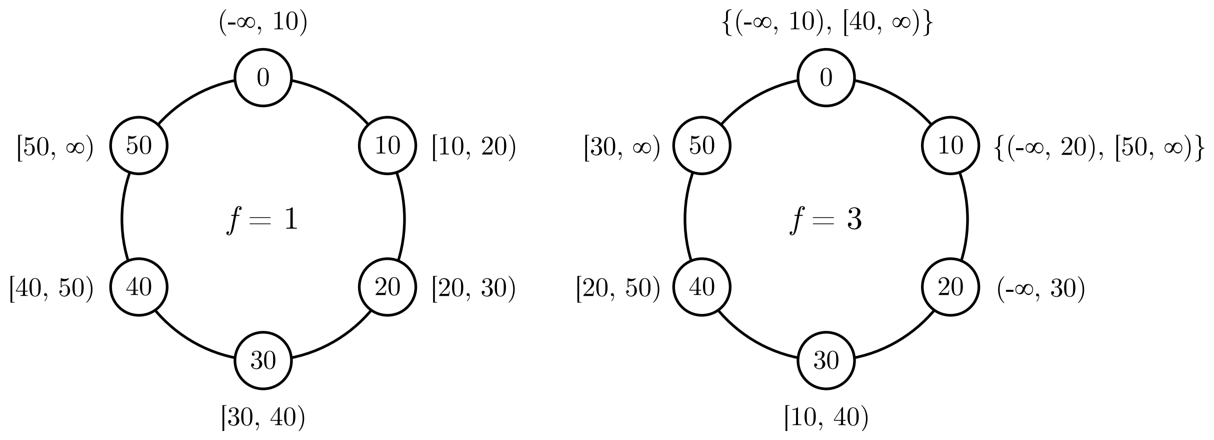


Figure 4.1: *Example of Successor Replication with f = 1 and f = 3*

Super Processes are able to submit *configuration messages* on behalf of the application. Configuration messages may update the mapping of keys to storage processes, explained in detail in Section 4.4. In order to be able to create configuration messages, super processes maintain a replica of the latest system configuration (set of processes and mapping of keyspace partitions to storage processes.) In addition, super processes are responsible for failure detection, see Section 4.2.3.

Recall Figure 1.2 on page 7 for an exemplary instantiation of Rubberband that shows all three types of processes.

4.2.2 Data Partitioning and Replication

Rubberband dynamically partitions the data over the storage processes along a single, user-defined key domain, using the consistent hashing scheme popularized by the Chord DHT [SMK+01]. In Rubberband, the mapping of keys to storage processes can be changed at any point, without loss of consistency, and with minimal impact on availability, as explained in Section 4.4.

Like Chord, Rubberband arranges storage processes in a logical ring. Each storage process is assigned a unique key, called the process identifier (ID), which determines the process' position in the ring. If a process p is assigned the ID k and its (in key-order) successor p' is assigned the ID k', then p holds the primary replica of all data objects with a key greater-than-or-equal k, and less-than k'. We call the range of keys a process p holds the *partition* of p.

In order to tolerate permanent failures of storage processes, Rubberband uses a simple replication scheme called *successor replication* [KZHL07]. Successor replication is widely used in NoSQL data stores, including Amazon Dynamo [DHJ+07], Apache Cassandra [LM10], and Scalaris [SSR08]. In successor replication, a replication factor of f means that the $f-1$ successors of a storage process p will hold a backup of every object that p holds the primary replica of. Consequently, the partitions of successive storage processes overlap (assuming $f > 1$), and it requires the correlated failure of f successive storage processes for data loss to occur.

An example of successor replication with replication factors $f = 1$ (no replication) and $f = 3$ is shown in Figure 4.1. Note the wrap-around of the keyspace at process 0. Also note the open ranges denoted by ∞ and $-\infty$. Successor replication in Rubberband does not require the keyspace to be finite. Our implementation allows *any* term in the Erlang programming language (numbers, strings, binaries, tuples) to be used as a key. The fact that Erlang defines a total order relation[2] over all possible terms becomes handy here.

We decided to use successor replication in Rubberband because of the scheme's inherent efficiency with respect to range reads and range writes. Rubberband uses *read-one-write-all* (ROWA) replication. To answer a range read, it is thus sufficient to forward the read to any set of storage processes whose union of partitions fully covers the requested range[3]. Conversely, range writes must be forwarded to any storage process whose partition overlaps with the written range. This scheme generalizes to multi-range reads and multi-range writes in a straight-forward manner, which Rubberband implements as well.

In contrast to alternative schemes such as symmetric replication [KZHL07], successor replication does not require a hash function. That being said, Rubberband does support optional, user-defined hash functions. The resulting hash partitioning has advantages over range partitioning with respect to load balancing, but range reads and range writes obviously become less efficient.

4.2.3 Failure Detection

E-Cast detects and masks failures of router processes automatically, as explained in Section 3.5.5. Failure detection of E-Cast application processes (here, Rubberband client, storage, or super processes) is the responsibility of application processes themselves however. As per the E-Cast network and failure model (Section 3.3.4), E-Cast merely assumes that for any failed application process p, eventually a message m is submitted such that $p \in \text{suspects}(m)$.

[2]`http://www.erlang.org/doc/reference_manual/expressions.html`, retrieved January 14, 2012.

[3]If the partitions of storage processes overlap, duplicate elimination of result tuples becomes necessary. Rubberband does this automatically by applying additional range filters on result tuples where necessary.

At the time of writing, we have implemented the following, simple failure detection scheme that meets the formal assumptions. The oldest super process in the system monitors all other super, client, and storage processes using the standard Erlang `monitor` function. The oldest super process is in turn monitored by every other super process. So assuming at least 2 super processes, every application process is being monitored by at least one super process.

When a super process p is informed of the (possible) failure of a monitored process p', p asks every other super process to check their connection to p'. If at least *half* of the super processes (including p) vote that p' has failed, then p submits a message m to E-Cast, such that $p' \in \text{suspects}(m)$. Any E-Cast router process that learns such a message m considers p' failed, and consequently will not send nor expect any more messages from p' (see Algorithm 11, page 109). If, however, not enough super processes vote that p' has failed, then p resumes monitoring of p'.

Note that requiring half of the super processes to agree that some process p' has failed does not constitute a majority quorum as normally required for distributed consensus. This is intentional. Distributed consensus is achieved simply by the fact that a message declaring p' as a suspect is learned by the E-Cast routers. Any single process is allowed to submit such a message as far as E-Cast is concerned. The point of requiring additional votes is simply to avoid the situation where a super process that temporarily loses connectivity single-handedly declares the rest of the system faulty.

As of Erlang R15B, `monitor` of remote processes is implemented by periodic tick messages exchanged between virtual machines, called *nodes* in Erlang terminology. Termination of a remote process is usually detected almost instantly (Erlang nodes automatically signal each other failures of monitored processes), but in case of silent failure (e.g. a broken network cable) failure detection requires between 45 to 75 seconds[4] under default virtual machine settings.

Clearly, the failure detection scheme we have described can be significantly improved in terms of scalability, accuracy, and tardiness. For example, storage processes could monitor their neighbors using frequent heartbeat messages with adaptive timeouts, rather than centralizing all responsibility at the super processes. Also, processes may respond to heartbeat messages but cease to do any actual work. There is a wide range of literature on adaptive, scalable failure detection that can be applied here [SPU11, HDYK04, HTC05]. Because E-Cast leaves the application (Rubberband) fully in control of failure detection, changes are easy to make. This is an important feature of the protocol. For the time being, the simple failure detection scheme described here is sufficient for our purposes.

[4]`http://www.erlang.org/doc/man/kernel_app.html`, retrieved January 15, 2012.

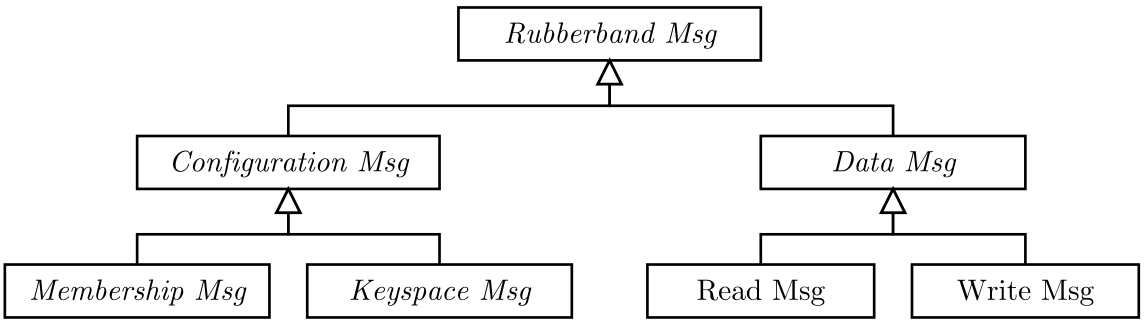


Figure 4.2: *Rubberband Message Type Hierarchy*

4.3 Message Routing with E-Cast

Rubberband defines a variety of message types. A UML diagram of the type hierarchy is given in Figure 4.2. The two main types of messages are *data messages* and *configuration messages*. Every data message is either a *read message* or a *write message*.

A configuration message is either a *membership message* or a *keyspace message*. A membership message changes the set of client, storage, or super processes that are considered part of the system, while keyspace messages change the mapping of keyspace identifiers (data partitions) to storage processes. The different types of membership and keyspace messages and how repartitioning is implemented are explained in Section 4.4.

With the exception of certain data messages (cf. Section 4.5), all messages are transmitted through E-Cast. For message routing, each router process maintains a local `MSeq` object. Whenever a router process learns a new message, it appends the message to its local `MSeq` object, so conceptually an `MSeq` represents the ever-growing history of routed messages.

An `MSeq` object is implemented as a garbage-collected sequence of `Config` objects (cf. Sections 3.5.3 and 3.5.5), created by successively *applying* the appended messages to the latest `Config`. The `MSeq` module is fully implemented by E-Cast, but the initial `Config` object to be used by an `MSeq` is plugged-in by the application (here, Rubberband) during system bootstrap.

In the case of Rubberband, a `Config` object consists of a set of client process references, super process references, and a mapping of keyspace identifiers to storage process references. A valid `Config` module must define a few callback functions for use by the E-Cast `M-Seq`. These functions are `destset`, `apply`, and `suspects`. The Rubberband `Config` module and the implementation of these callback functions are sketched in Algorithm 12.

The `destset` function computes the destination set of data messages in read-one-write-all (ROWA) manner, as explained in Section 4.2.2. The destination set of a configuration

Algorithm 12: Rubberband Config Module Sketch

```
function apply(m, c) begin
    if m is a configuration message and sender of m is an unsuspected process then
        update (side-effect free) the config c and return the new config
    else
        return the original config c
end
function destset(m, c) begin
    if sender of m is an unsuspected process then
        if m is a data message then
            p ← the set of key ranges (partition) read or written by m
            if m is a read message then
                (deterministically) find and return a set of unparted, unsuspected storage
                processes in c whose union of partitions fully covers p
            else m is a write message
                find and return the set of all unparted, unsuspected storage processes in c
                whose partitions overlap with p
        else m is a configuration message
            return the union of the set of all super processes in c, and the set of all
            unsuspected client and storage processes in c that are affected by m
    else
        return {}
end
function suspects(m) begin
    if sender of m is an unsuspected process then
        return the set of all processes suspected by m
    else
        return {}
end
```

message is simply the set of all affected processes, which always includes the set of super processes, since these must maintain a replica of the current system configuration.

Note that the `destset` function must be deterministic to ensure agreement on the destination set of a message across multiple router processes. In the case of read messages, there are typically many possible, correct destination sets due to multiple replicas being available for each data object. To balance the load over the available replicas, each read message has a *replica offset* field, which is set by the client process during message creation. Conceptually, the ring of storage processes is rotated by the chosen offset before lookup, but the implementation details are quite complex because one has to account for the possibility of unavailable (parted or suspected) storage processes on the ring.

The `apply` function works as follows. Whenever a data message is applied to a `Config` object, it yields the original `Config` object (since reads and writes do not change the system configuration). But when a configuration message is applied to a `Config` object, it yields a new `Config` object which reflects the effect of the message and may change the routing of subsequently applied messages. For example, when a *suspect message* m is applied such that suspects$(m) = \{p\}$ where p is some storage process in the keyspace, then Rubberband will stop routing subsequent read messages to p and instead use the neighbors of p on the keyspace, which hold replicas of the data partition assigned to p.

The `suspects` function should be self-explanatory at this point. More interesting is that messages sent by suspected processes have no effect. This is to ensure that suspected processes cannot corrupt the system state. Our concrete implementation of the `Config` module does not remember which processes are suspected. Instead, it is sufficient to know which processes are *not* suspected; i.e., which client, storage, and super processes are currently part of the system. This property may seem trivial, but it is important in order to ensure that all state related to suspected processes can eventually be garbage collected. Garbage collection is an extremely tricky detail in reconfigurable state-machine implementations [AKMS10], so we want to point out that E-Cast hides all of this complexity from the application code, without restricting the types of systems that can be built.

In summary, the `Config` module described here implements exactly the formal reduction of dynamic partial replication to stateful routing described in Section 3.3.3 of the previous chapter. The `destset` function in Algorithm 12 clearly meets the formal definition on page 104, which (informally speaking) states that the destination set of any message m must contain every process whose state is affected by m. Assuming the storage processes handle every delivered message in delivery order, it follows that Rubberband is a correct solution of dynamic partial replication, at least as far as the formal model is concerned.

4.4 Reconfiguration

One of the main features of Rubberband is its ability to change the system configuration (membership and data partitioning) at runtime, without loss of consistency, and with minimal impact on availability. As explained in the previous section, system reconfiguration is initiated by the submission of configuration messages to E-Cast. There are two types of configuration messages: membership messages and keyspace messages. Membership messages change the set of processes that are considered part of the system, while keyspace messages change the mapping of keyspace identifiers to storage processes. We will now discuss membership and keyspace messages in more detail.

4.4.1 Process Lifecycle and Partial Live Migration

There are three types of membership messages and four types of keyspace messages. A membership message is either a *join message*, a *part message*, or a *suspect message*. These messages relate to the lifecycle of every Rubberband process (client, storage, or super process) in a straight-forward manner.

A join message is submitted exactly once by each process, right after the process spawns, and before submitting any other message. The point at which a join message appears in the `MSeq` object replicated by the E-Cast routers marks the point from which on the process is considered part of the system. When a process gracefully leaves the system, it submits a *part message*, and when a process is suspected of failure, a corresponding *suspect message* is submitted by some other process, as explained in Section 4.2.3.

When a storage process joins the system, it is not immediately assigned an identifier (ID) in the keyspace. Instead, the process enters *standby* mode, awaiting further instructions. These instructions may be delivered to the process at a later point, in the form of a keyspace message, created by a super process in response to a user request. The following four types of keyspace messages exist:

Expand Message Expands the keyspace by assigning an ID to a *standby* storage process, which upon delivery of the message becomes *active*.

Contract Message Contracts the keyspace by removing an (*active*) storage process from it. If the storage process was already suspected by a previous suspect message, then all state related to the process can now be garbage collected. Otherwise, the storage process will eventually deliver the contract message, at which point it returns into *standby* mode.

Replace Message Replaces an *active* storage process with a *standby* storage process, assigning the same ID to the replacement process which the replaced process had previously. If the replaced storage process was already suspected by a previous suspect message, then all state related to the replaced process can now be garbage collected. Otherwise, the replaced storage process will return to *standby* mode.

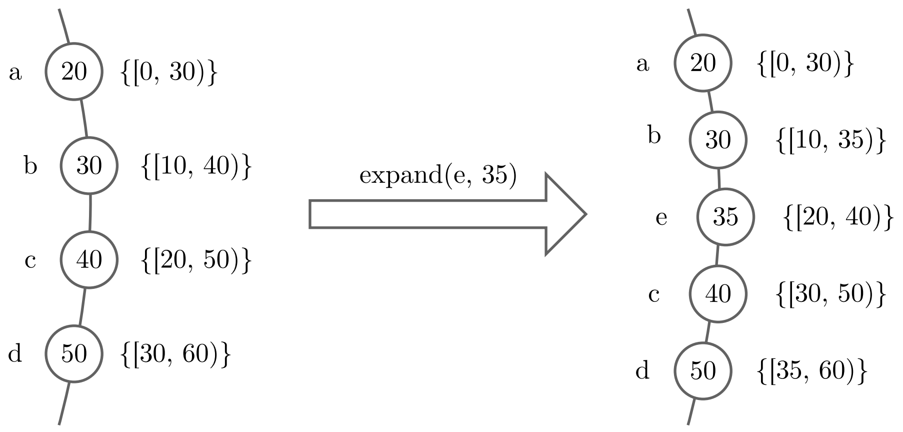


Figure 4.3: *Reconfiguration Example: Expand*

Rebalance Message Assigns a different ID to an *active* storage process, thereby rebalancing the data distribution in the keyspace. For implementation reasons, a rebalance message is not allowed to change the order of storage processes in the keyspace.

Note that expand and contract messages alone would be sufficient to transform one mapping of IDs to processes to any other mapping of IDs to processes. Replace and rebalance messages have been added in order to lower the number of reconfiguration steps and consequently the amount of data shuffling required for these frequent transformations. One may consider adding even more types of keyspace messages, but we believe the given types of messages are a good trade-off between system complexity and efficiency.

Keyspace messages contain all the information that storage processes need in order to shuffle the data in accordance with the requested keyspace transformation. To be precise, every keyspace message contains a list of *instructions*, each of which tells a specific storage process to copy, receive, or delete some partition of data. These instruction lists are created by the respective super process that submits a configuration message. The reason the instruction lists are explicitly created rather than being implicit in the keyspace transformations is that for a given keyspace transformation there can be multiple correct instruction lists, and choosing the "best" of these is a non-trivial optimization problem, discussed in detail in Section 4.4.2.

Consider Figure 4.3 for an example with replication factor $f = 3$. On the left side of the figure, a part of a larger (irrelevant) keyspace is shown, containing the 4 storage processes a, b, c, and d, which have been assigned the identifiers 20, 30, 40, and 50, respectively. The data partition assigned to each storage process is shown right next to the respective storage process. The user has requested to expand the keyspace, assigning the identifier 35 to storage process e.

The right side of the figure shows the keyspace after the transformation. Note that the data partitions of individual processes change in accordance with the rules of successor replication. Storage process e now holds the partition $\{[20, 40)\}$. The partitions of b, c, and d have shrunk. Consequently, e needs to receive copies of all data objects in the partition $\{[20, 40)\}$, and b, c, and d need to delete part of their data. A possible list of instructions to that effect would look like this:

$$\begin{gathered}\langle \text{copy}, b, e, \{[20, 40)\}\rangle \\ \langle \text{delete}, b, \{[35, 40)\}\rangle \\ \langle \text{delete}, c, \{[20, 30)\}\rangle \\ \langle \text{delete}, d, \{[30, 35)\}\rangle\end{gathered}$$

When a storage process receives a keyspace message, it filters out and transforms the instructions which concern itself, and passes them on to the storage engine (e.g. Crescando) for execution. In the example, storage process b would find that

$$\begin{gathered}\langle \text{copy}, b, e, \{[20, 40)\}\rangle \\ \langle \text{delete}, b, \{[35, 40)\}\rangle\end{gathered}$$

concern itself, which it would pass on to the storage engine as

$$\begin{gathered}\langle \text{send}, e, \{[20, 40)\}\rangle \\ \langle \text{delete}, \{[35, 40)\}\rangle\end{gathered}$$

while storage process e would tell its storage engine to

$$\langle \text{receive}, b, \{[20, 40)\}\rangle$$

The storage engines can shuffle the data (possibly gigabytes) in the background, through direct connections (not E-Cast or Rubberband). We call this act of shuffling data *partial live migration*, in accordance with the established terminology in the literature. In Section 5.4, we give an overview of how partial live migration is implemented in Crescando.

Consistency and Asynchronous Reconfiguration

Data consistency in the face of partial live migration relies on the fact that every storage engine executes its instructions atomically, in the same total order with respect to the read and write messages that are delivered before and after. We emphasize that it is *not* necessary to stop the system during reconfiguration. Indeed, E-Cast and Rubberband *never wait*. Messages (reads, writes, reconfigurations) can be submitted at any point, and will be promptly delivered. An arbitrary long sequence of configuration messages can safely be submitted before even the first reconfiguration has been completed. Storage engines simply have to handle delivered messages in a way that is *equivalent* to handling them serially, in order to maintain sequential consistency of the whole system.

There is one small caveat however. Since the instruction list is created by the super process submitting a keyspace message, correctness relies on the fact that the configuration does not change concurrently, between the point where the super process submits a keyspace message, to the point where the keyspace message is delivered. Our implementation solves the problem by timestamping every keyspace message with the configuration version at the point where the message is created. The `Config` object (see previous section) then tests whether the configuration has already changed at the point where the keyspace message is learned. In the (rare) case where this happens, the keyspace message is ignored (does not alter the configuration, empty destination set). The super process can detect this because only an ignored message is acknowledged by E-Cast without having been delivered first.

The fact that Rubberband is wait-free even during reconfiguration (messages keep being delivered) enables *asynchronous reconfiguration*. Storage engines can interleave data shuffling with the execution of read and write operations insofar as this remains equivalent to serial execution. For more details, see Section 5.4 and the Master's thesis by Ashmawy [Ash11], which explain how partial live migration is implemented in Crescando.

Availability and Failure Detection

Rubberband `Config` objects keep track of storage processes which are busy with reconfigurations (known by inspection of the instruction lists). Processes that are busy continue to receive write messages as usual, but only receive read messages if there are no alternative, non-busy replicas of the requested data. When the busy storage process is ready to process read messages again, it submits a *ready message*. When all storage processes that are involved in a reconfiguration (i.e., all storage processes in the destination set of the respective configuration message) have submitted a ready message, the reconfiguration is considered complete.

This fairly straight-forward scheme allows data stores built on Rubberband to stay *continuously available*, even during reconfiguration, assuming a sufficient replication factor. The reason is that Rubberband separates the problem of uniform agreement on a total order of read and writes (done by E-Cast) from the act of executing the read and writes (done later by the storage engines). The time when a write is finally executed by every affected storage process is irrelevant for consistency and durability. E-Cast guarantees that every (acknowledged) write is eventually, uniformly executed. Each individual storage process is allowed to execute writes as soon as they are delivered. Thus, if there is a single, non-busy, unsuspected storage process in every replication group, these storage process can immediately execute any write, and the effects of any write are immediately available for reading. Keeping track of which storage processes are busy and which are not ensures that most reads are immediately processed. Consequently, a data store built with Rubberband is *fully available as long as there is a single non-busy, unsuspected storage process in every replication group.*

The information about complete and incomplete reconfigurations is also important for failure detection and liveness. Whenever a storage process is suspected which is involved in an incomplete reconfiguration, all the other storage processes involved in the reconfiguration are added to the destination set of the suspect message. At the point where the suspect message is delivered to a storage process, that storage process can abort any pending reconfiguration instructions pertaining to the suspected remote process. Without this information, storage processes would have to implement their own failure detection to avoid blocking indefinitely. We refer to Ashmawy [Ash11] for further details.

4.4.2 Partial Live Migration as an Optimization Problem

As pointed out previously, every keyspace message contains a list of instructions for the affected storage processes. But depending on the replication factor, many different instruction lists are possible. Given a specific metric such as load balance, system availability, or probability of data loss, one instruction list may be "better" than another. The super process which creates the keyspace message thus faces an interesting, multi-dimensional optimization problem.

Let us revisit the reconfiguration example discussed previously (Figure 4.3 on page 137). The original instruction list told storage process b to copy the partition $\{[20, 40)\}$ to storage process e. An alternative solution would be to have storage process c copy $\{[20, 40)\}$ to e. Yet another solution would be to let b copy $\{[20, 30)\}$, and to let c copy $\{[30, 40)\}$. Since storage process d holds the partition $\{[30, 60)\}$, it too could participate, and so forth.

This optimization problem appears in some form in any partially replicated system that deals with permanent process failures. While there is a wide range of literature on load balancing in distributed hash tables (DHTs), we are not aware of any work on NoSQL data stores, which considers the problem of shuffling a large amount of data atomically as an optimization problem by itself. We believe this is due to two main facts. First, eventually consistent systems do not need to *atomically* shuffle data. An approach that incrementally shuffles and compares data in the background—cf. anti-entropy [DGH+87, DHJ[+]07, LM10]—is more natural and useful in this context. Second, systems that use hash partitioning typically also use *virtual servers* [SMK[+]01]—a single physical node hosts many storage processes (virtual servers), widely spread over the keyspace. Consequently, the cost of data shuffling when a physical node joins or fails is spread over the system.

The problem of atomically shuffling large partitions of data thus manifests only in the context of *elastic range partitioning with strong consistency*, where anti-entropy is not applicable, and virtual servers would negate the advantages of range partitioning. Besides Rubberband, only ecStore [VCO10] and Scalaris [SSR08] currently support elastic range partitioning with strong consistency. Neither of these systems seems to recognize partial live migration as an optimization problem in its own right at this point.

Optimizing for Fault-Tolerance

Rubberband is designed to be wait-free and continuously available, even during reconfiguration, as explained in the previous section. Storage processes that are not instructed to participate in partial live migration proceed with executing read operations, write operations, and even other reconfigurations. The downside of this approach is that there is no safety net in case partial live migration fails. If the sender of a data partition crashes before the transmission is complete, the receiver is left in an inconsistent state, and all other originally available replicas may already have moved on.

In principle, this problem can be solved by stopping the system during reconfiguration, or by giving storage processes the ability to *time travel*; i.e., the ability to read an older snapshot of the data, for example by rolling back the log. But stopping the system is in conflict with our design principles, and time travel is an expensive feature that is not supported by most storage engines (including Crescando and Memcached).

So instead, we propose to mitigate the problem by performing partial live migration in a way that minimizes the likelihood of data loss. That is, we model partial live migration as an optimization problem, where the optimization metric is the probability of data loss.

Consider Figure 4.4 for an example. This is the same example used previously in Section 4.4.1, except for the fact that storage process d has failed now (visualized by coloring the node gray). The replication factor is still $f = 3$.

In the case where storage process c is chosen to copy all of partition $\{[20, 30)\}$ to the new storage process e (Solution 1), the failure of c during the transmission will leave c, d, and e in a failed state. As explained previously, for a replication factor of f, the failure of f successive storage processes causes data loss. In the example, b will already have deleted $\{[35, 40)\}$ to reflect the new data partitioning, so the data objects in $\{[35, 40)\}$

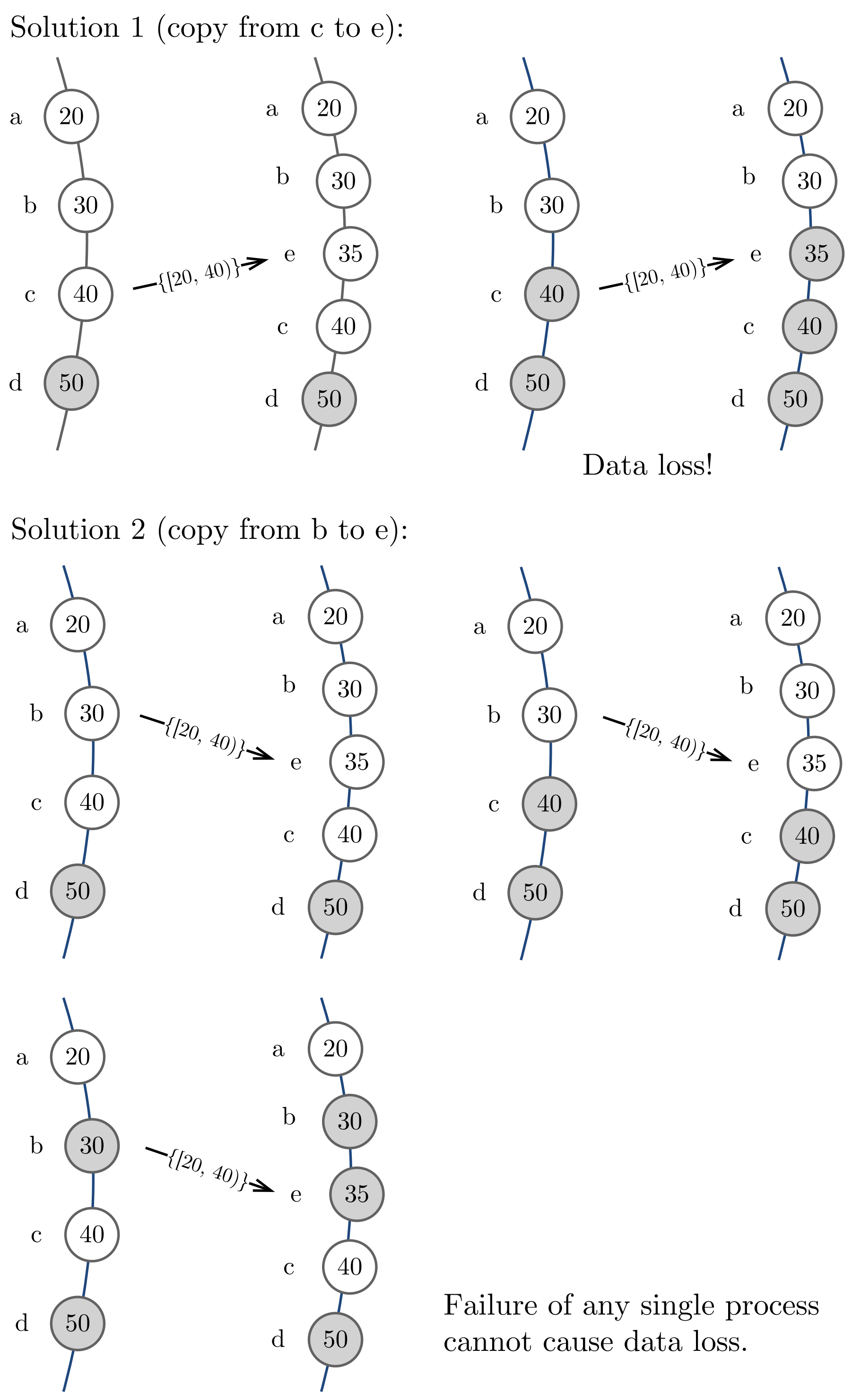

Solution 1 (copy from c to e):
a 20
b 30
c 40
d 50
{[20, 40)}
e 35
Data loss!
Solution 2 (copy from b to e):
{[20, 40)}
Failure of any single process cannot cause data loss.

Figure 4.4: *Reconfiguration Example: Expand - Revisited*

Algorithm 13: Reconfiguration Optimizer Framework

```
$k_{old} \leftarrow$ the keyspace before transformation
$k_{new} \leftarrow$ the keyspace after transformation
$Dest \leftarrow$ every storage process $s \in S$ where partition($s$, $k_{new}$) $\supset$ partition($s$, $k_{old}$)
foreach $s \in Dest$ do
    $Src[s] \leftarrow$ every minimal set of non-suspected processes $S' \subseteq S$ in $k_{old}$ where
    $\bigcup_{s' \in S'}$(partition($s'$, $k_{old}$)) $\supseteq$ (partition($s$, $k_{new}$) - partition($s$, $k_{old}$))
$Sol \leftarrow \Pi_{s \in Src} Src[s]$
return $\min_{\texttt{rank}(sol)} \{sol \in Sol\}$
```

will be permanently lost. In contrast, if storage process b is chosen to copy the partition $\{[20, 30)\}$ to e (Solution 2), then the failure of any single process cannot cause data loss. For any group of 3 successive storage processes, at least one storage process is live.

The example given is relatively simple, but things get complicated quickly for higher replication factors and particularly for contract messages, where multiple storage processes need to receive data from their neighbors. There can be chains of dependencies between storage processes, so the failure of a single storage process can cause a whole chain of processes to fail transitively.

Finding the optimal solution that minimizes the *actual* probability of data loss would require accurate estimates of transmission durations and the probability of individual and correlated failures of storage processes. Obtaining this information is a research topic by itself. So instead, we consider the simplified problem of minimizing the probability of data loss while ignoring transmission durations and correlated failures (except the modeled dependencies), and treating all storage processes as equally likely to fail.

Algorithm 13 shows the reconfiguration optimizer framework in high-level pseudo-code. This algorithm is executed whenever a super process creates a configuration message and needs to build an instruction list. First, the algorithm computes the set of storage processes that need to receive data from their neighbors. These processes are called *destinations.* In our example, this would be just storage process e, but in the case of a contract message, it may be multiple processes. Then, for each destination process, the algorithm computes every minimum set of storage processes whose union of partitions covers the missing partition. These sets of processes are called *source sets.*

Next, the algorithm computes the Cartesian product over all source sets of all destinations (indicated by Π in Algorithm 13, in slight abuse of mathematical syntax), giving the set of all possible, minimal ways to shuffle data between storage processes that transforms the keyspace from k_{old} to k_{new}. Each of those we call a *solution* which can be transformed into an equivalent instruction list for the configuration message. Finally, the solutions are ranked, and the best solution is returned.

This may seem like a lot of computation, but for practical replication factors ($f \leq 5$), the number of solutions is manageable. In our original example (no suspected/failed processes), there is only one destination (e) and three solutions:

$$\{\{e, \{b\}\}\}$$
$$\{\{e, \{c\}\}\}$$
$$\{\{e, \{a, d\}\}\}$$

These solutions read as "e receives data from b", "e receives data from c", and "e receives data from a and d".

We do not consider non-minimal solutions such as $\{\{e, \{b, c\}\}\}$. These may be advantageous from a performance and load-balance perspective, but ignoring transmission times, the probability of data loss is always higher when more storage processes than necessary are sending data. In addition, more processes are busy with reconfiguration, lowering system availability. Another positive effect of ignoring non-minimal solutions is that it prunes the solution space significantly, thus lowering optimization time.

Solutions are ranked as follows. First, one creates a directed graph for each solution, where every storage process in the solution becomes a graph node, and every pair of ⟨source, destination⟩ processes is connected by a directed edge from source to destination. For a concrete example, consider Figure 4.5. This is the inverse of the previous example. The storage process e is removed again; i.e., the keyspace is being contracted. The figure shows 4 alternative solutions (more exist, but this is irrelevant for the explanation). These 4 solutions are transformed into the 4 corresponding graphs shown in Figure Figure 4.6.

Given one of these graphs, one can now efficiently compute (graph flooding) the set of process failures that would result from any single process failure. Every possible single process failure is simulated, and for each failure case the number of failed processes in every replication group of the new keyspace is computed. Obviously, one only needs to consider replication groups which are affected by the reconfiguration.

Since the replication factor is $f = 3$, the affected replication groups in the example are $g_1 = \{a, b, c\}$ and $g_2 = \{b, c, d\}$. As a matter of fact, there would be 3 more replication groups affected, but the predecessor of a and the 2 successors of d are ignored in this example for simplicity. In general, these groups cannot be ignored, because there may be failed processes in the groups, which could affect the optimality of a solution, as described next.

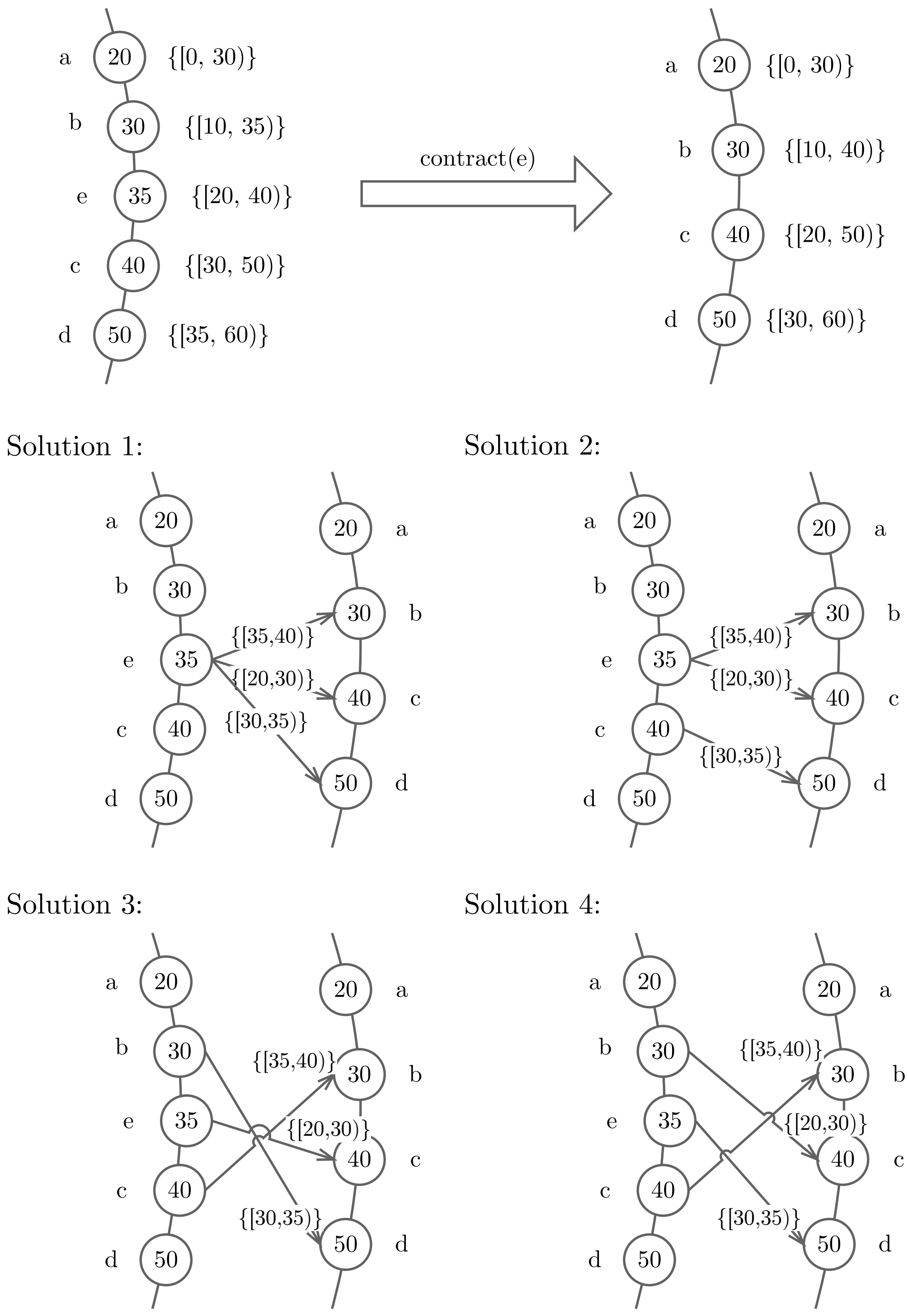


Figure 4.5: *Reconfiguration Example: Contract*

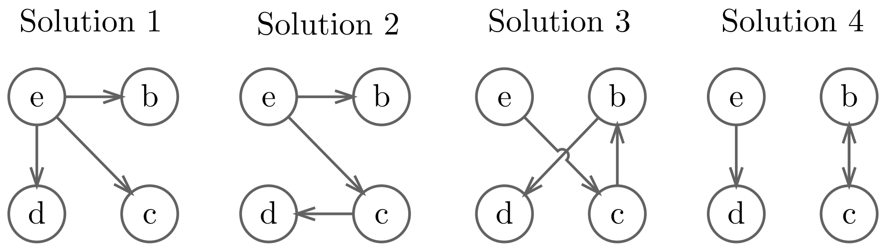


Figure 4.6: *Reconfiguration Example: Contract - Solution Graphs*

Assume process b fails in Solution 1. There are no outgoing edges for b, so nothing else would happen. In replication groups g_1 and g_2 would be 1 failed process (b). This fact is encoded as a sequence of integers in descending order (does not apply here): $\langle 1, 1\rangle$. The test whether one failure case is *worse* than another, one only has to compute this sequence, and perform a *lexicographical comparison* between the sequences. That is, the sequence $\langle 3, 2, 1\rangle$ would be worse than $\langle 3, 1, 1\rangle$ and also worse than $\langle 2, 2, 2\rangle$, since $\langle 3, 2, 1\rangle > \langle 3, 1, 1\rangle > \langle 2, 2, 2\rangle$. The heuristic here is that losing $k + 1$ replicas in one replication group is typically more likely to lead to data loss and/or system unavailability than losing k replicas in multiple replication groups.

Obviously, the worst case for Solutions 1, 2, and 3 is the failure of e. In this case, processes b, c, and d would fail transitively. Thus, in replication group g_1 there would be 2 failed processes, and in replication group g_3 there would be 3 failed processes. This fact is encoded as the integer sequence $\langle 3, 2\rangle$. For Solution 4, the worst case is failure of b or c. In either case, there would be 2 failed processes in both g_1 and g_2. This is encoded as $\langle 2, 2\rangle$.

As an optimization of the graph flooding, the optimizer marks processes in the graph as *dominated* when they are failed *transitively* during the simulation; i.e., the dominated process fails as a result of one of its source processes failing. The optimizer skips simulations that start with such a dominated processes failing, since it is guaranteed at this point that one has already simulated a case in which a superset of processes fail. Suppose the optimizer simulates the failure of e in Solution 3. The simulation of failure of processes b, c, and d can be skipped afterwards. (Note that it is incorrect to simply skip all simulations that start with a process that has incoming edges; i.e., a process that has sources; because there may be cycles in the graph, as for Solution 4.)

Once the optimizer has computed the worst case for each solution, it computes the “best worst case”. First, the solutions are ranked by their worst-case integer sequences. If there is a tie, the solution with fewer sender processes is ranked higher, since it gives better system availability (fewer processes busy with reconfiguration). Remaining ties are resolved by coin toss.

In the example, the highest-ranked solution is Solution 4, since $\langle 2, 2\rangle < \langle 3, 2\rangle$. Solution 4 is chosen by the optimizer and subsequently transformed into an instruction list to be put into the configuration message.

In conclusion, the optimizer may not choose the solution that minimizes the *actual* probability of data loss. However, the optimizer finds the best solution according to a reasonable ranking function, which avoids solutions that are clearly bad. That being said, we have barely scratched the surface of the partial live migration optimization problem. Other heuristics and ranking functions may give better results, and we have so far ignored other additional, important dimensions such as load balance and reconfiguration time. For the time being, the optimizer presented here fulfills its duty, and we hope it serves as a baseline and inspiration for future research.

4.4.3 Bootstrapping the System

In this section, we explain how to bootstrap the system, which also gives us the opportunity to recapitulate how the entire protocol stack plays together.

To create a new instance of Rubberband, one first has to start an E-Cast router process. This first router process is *told* (by the user) that it is the first, and is also given an initial `Config` object. In the case of Rubberband, the initial configuration is an empty set of super processes, an empty set of client processes, and an empty keyspace i.e. mapping of identifiers to storage processes.

Similarly, the first super process is *told* that it is the first, and is given a set of router process references. After contacting one of the routers, the super process (like any other application process) is subsequently informed of any changes in router leadership.

Super processes form a reconfigurable state machine whose state changes whenever a configuration message is delivered. The replicated state is simply the latest system configuration, which is required for creating configuration messages (cf. previous section). At least one super process must join before any client or storage process joins, to ensure that the configuration held by the super processes corresponds to configuration held by the latest `Config` object inside the router processes. Once there are at least one router process and one super processes in the system, other processes can join in any order.

When a storage process starts, it submits a join message to E-Cast and enters the pool of *standby* processes. When the user (or some automatic load balancer) later decides there is a need for another storage process in the keyspace, it picks an identifier (key) and a standby storage process, and instructs one of the super processes to expand the keyspace. In response, the super process creates an expand message (using the optimizer described in the previous section), and submits the message to E-Cast. The message is routed to all super processes and affected storage processes, updating the routers' `Config` objects on the fly. Likewise, super processes update their copy of the mapping once the message is delivered to them. Any subsequently submitted data messages (reads and writes) will take the newly added storage process into account for their routing.

4.5 Consistency and Scalability

In this section, we discuss the consistency guarantees that Rubberband derives from E-Cast, show how lowering the isolation level of reads can increase system throughput, and explain how Rubberband can be extended to scale linearly with single-key operations.

4.5.1 E-Cast and Sequential Consistency

E-Cast guarantees that messages are delivered uniformly, in causal total order. And as explained in Section 4.3, the `Config` module that Rubberband plugs into E-Cast ensures that every process affected by a given message will also be in the destination set of the message. Assuming that storage engines execute delivered messages in delivery order, it then follows that a data store built with Rubberband is *sequentially consistent.*

Sequential consistency means that reads and writes are performed in a global total order, which is compatible with the read and write order of every client process [FR10]. In other words, users can (almost) forget about the fact that they are dealing with a distributed system. The price being paid for this level of consistency and fault tolerance is a limit on system throughput. The E-Cast protocol forms a so-called *consensus bottleneck.*

We argue that this limitation is fundamental in nature and not an oversight in the system architecture. No fault tolerant, distributed system that wants to guarantee sequential consistency for arbitrarily overlapping operations (reads and writes) can scale linearly. As stated in the informal problem statement in Section 3.1, sequential consistency for arbitrarily overlapping operations requires, *in the general case*, distributed consensus on a global, total order of all operations. In more formal terms, *sequential consistency does not compose.* That is, two histories (orders of events or operations in a system) that individually satisfy sequential consistency cannot, in the general case, be composed into a single, compatible history that satisfies sequential consistency [FR10].

To see why, consider an exemplary system that consists of two storage processes, a and b, which both execute two concurrent, conflicting writes $w_1 = \{x \leftarrow 1,\ y \leftarrow 1\}$ and $w_2 = \{x \leftarrow 2,\ y \leftarrow 2\}$. Suppose a executes w_1 before w_2, and b executes w_2 before w_1. After executing the two writes, a holds $\{x = 2,\ y = 2\}$ and b holds $\{x = 1,\ y = 1\}$.

Individually, both a and b are sequentially consistent, since they both execute the writes serially. But if a subsequent read r reads from both a and b, say object x from a and object y from b, then r sees $\{x = 2,\ y = 1\}$, which is an impossible state for any strictly sequential execution of w_1 and w_2. Distributed consensus on the read and write order is necessary. By its very nature, this implies a serialization bottleneck, so unlimited, linear scale-out is fundamentally impossible.

That being said, a *limited* degree of scale-out is still possible. Looking closer at sequential consistency, it turns out that distributed consensus on a global, total order is not always necessary. Specifically, the system "only" needs to reach distributed consensus on an execution order for every pair of conflicting operations. What is required, formally, is a *partial order relation* which establishes an order between every pair of conflicting operations.

Looking at related systems, ecStore [VCO10], CloudTPS [WPC10], and Scalaris [SSR08] use synchronous, atomic commit protocols. Whenever a transaction is being committed, the responsible transaction manager checks whether all distributed locks are still held (or whether composable conflict relations hold for all replicas), accordingly decides whether to commit or abort the transaction, and then uniformly propagates the decision to all affected replicas. The fact that no two conflicting transactions can commit concurrently establishes the required partial order.

While it allows general ACID transactions (sequences of read and write operations), there is one important problem with this approach. Atomic commit is a synchronous, multi-phase operation, and consistency requires that conflicting transactions commit serially. Consequently, system throughput becomes limited by network latency. Processes spend a lot of time just waiting for messages to arrive. Indeed, the experimental results of ecStore, CloudTPS, and Scalaris show that implicit ordering of transactions only scales as long as the fraction of write and multi-key transactions (anything that requires synchronization) in the workload is low. Because processes spend a lot of time waiting for commit votes, the throughput numbers reported for these systems are in fact much lower than those of Crescando/RB (Crescando with Rubberband) for practical numbers of machines (a few dozen). Section 5.6 compares our results to the numbers reported for ecStore, CloudTPS, and Scalaris.

The design of P-Store [SSP10] on the other hand is much closer to that of Rubberband, in that P-Store is also based on a self-contained atomic multicast protocol. The multicast protocol used by P-Store is *genuine* in that it does not order messages whose destinations (process groups) do not overlap. Indeed, the protocol scales well for single-group messages, but throughput degrades dramatically for multi-group messages. One can assume this is due to the fact that P-Store must then reach consensus on message order across groups, which quickly becomes expensive in a protocol that is designed to treat this as an exceptional case.

From this perspective, Rubberband can be said to follow a more conservative approach with respect to consistency and performance. The E-Cast protocol makes no attempt to detect conflicts and construct a minimal partial order. Instead, consistency is derived from the fact that all operations—conflicting or not—are brought into a total order which is a linearization of the partial order required for consistency.

This *design for the worst case* may waste resources in cases where the overwhelming majority of operations do not conflict and would not need to be ordered. But it is consistent with the design of the Crescando storage engine in that users are given strong, clear, and predictable performance guarantees. The experimental results in Section 5.6.5 indeed show that the combination of Crescando and Rubberband into Crescando/RB yields a distributed system that is highly robust to global transactions.

All that being said, we believe it is possible to significantly increase throughput for workloads that predominantly consist of single-key operations, without incurring a serious loss in performance for multi-key operations. We have not implemented the idea at the time of writing, but we explain the necessary protocol extensions in detail in Section 4.5.3.

4.5.2 Read Isolation Levels

Rubberband client processes can perform reads at different isolation levels, giving users the ability to trade read consistency for performance and scalability if desired. We have currently implemented the following three levels of isolation, which users can choose from for every individual read.

Sequential Read Every sequential read of a client process c sees a globally consistent snapshot of the application data, which is more recent than any snapshot seen by any preceding sequential read of c (Monotonic Read), and reflects the effects of any preceding write by c (Read Your Write), but not the effects of any successive write by c (Write Follows Read). In other words, Rubberband guarantees *sequential consistency* [FR10] with respect to sequential reads, hence the name.

Snapshot Read Snapshot reads are brought into total order with respect to writes. Consequently, snapshot reads are guaranteed to see a globally consistent snapshot of the data. However, no guarantees concerning the execution order of snapshot reads with respect to submission order are given. (Monotonic Read, Read Your Write, and Write Follows Read are *not* guaranteed.)

Basic Read Basic reads scale linearly with the number of storage processes (no protocol bottleneck). However, *multi-key* basic reads may see partial effects of concurrent writes, for reasons explained momentarily.

Snapshot reads are transmitted as *unsafe* messages in E-Cast, previously explained in Section 3.5.7. This means that every snapshot read is sent to (what the client believes is) the lead router, which injects it into the sequence of writes it forwards, but *without* atomically broadcasting the read to other routers. For every storage process s which a router r forwards the snapshot read to, r attaches the timestamp of the latest write it previously forwarded to s. When s receives the read, it checks whether the timestamp of the read indeed matches the timestamp of the latest write it executed. In this case, it executes the read, otherwise it rejects it. It follows that there is a total order between all writes (in fact, all messages sent through E-Cast) and all successful (uniformly executed) snapshot reads. Therefore, snapshot reads are guaranteed to see a globally consistent snapshot of the data.

In contrast to sequential reads and snapshot reads, every basic read is sent directly to the affected storage processes, so there is no E-Cast protocol bottleneck. The client process attaches to every basic read the configuration version it expects the destination storage processes to be in. When a storage process s receives a basic read, it checks whether the configuration version matches (i.e., there has been no concurrent reconfiguration). In this case, it executes the read, otherwise it rejects it. It follows that there is a total order between all configuration messages (always sent through E-Cast) and successful basic reads, so at least *configuration consistency* is guaranteed. A multi-key basic read that is executed by multiple storage processes may see partial effects of concurrent writes however, since individual storage processes may have executed different prefixes of the global write sequence at the point where they receive a specific basic read.

To be able to send basic reads directly to storage processes, every client process needs to maintain an up-to-date copy of the keyspace. At the time of writing, this is implemented by client processes being delivered configuration messages through E-Cast, just like the super processes. For extremely large numbers of clients (thousands) and frequent reconfigurations, this is not the most efficient solution. Fortunately, consistency is not violated if some clients hold an out-dated copy of the keyspace, because storage processes will simply reject those clients' reads. So in future versions of Rubberband, clients could easily be made to receive configuration messages directly from super processes, or even through a separate gossip protocol.

In summary, all three read isolation levels are implemented purely in Rubberband and E-Cast and are thus available for any storage engine. Note that the three isolation levels

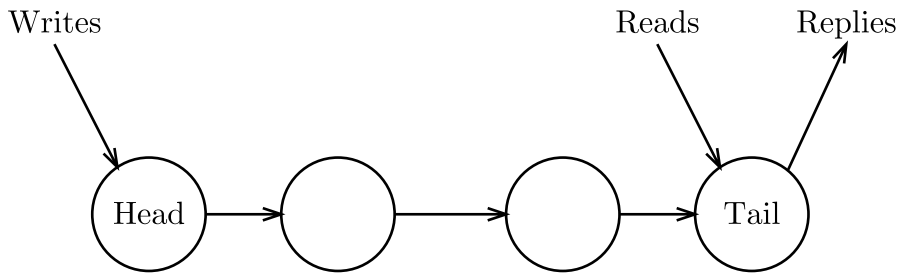


Figure 4.7: *Chain Replication*

by no means exhaust the design space. They have been chosen because they represent interesting trade-offs between consistency and performance, and because they can be found in similar form in many systems, so users should find them familiar and useful.

4.5.3 Scaling with the Workload

For reasons given in Section 4.5.1, no sequentially consistent system can scale linearly with respect to multi-key operations. But it is still possible to scale linearly with respect to *single-key* operations (assuming negligible workload skew), for the simple reason that single-key operations on separate keys do not conflict and consequently need not be ordered. We now propose an extension to Rubberband to this effect.

Chain Replication

Assume for the sake of argument that *all* operations in the system are single-key, and also assume for the moment that the system configuration is completely static. To implement sequential consistency, it is then sufficient to execute reads and writes of individual data objects uniformly, in total FIFO order. In other words, each data object stored in the system can be modeled as an isolated, replicated state machine (+ FIFO order). The NoSQL data stores Spinnaker [RST11], G-Store [DAEA10], and MegaStore [BBC^{+}11] implement exactly this idea, when they propose to run a separate instance of (some generalization of) the Paxos protocol [Lam98] for every key or non-overlapping group of keys.

We argue that a separate instance of Paxos for every key or group of keys is overkill. Instead, we propose to generalize a much simpler protocol called *chain replication*, originally published by van Renesse and Schneider [vRS04].

Chain replication works as follows. Let all storage processes be arranged in a chain (hence the name), as visualized in Figure 4.7. Let clients send their writes to the head of the chain, and their reads to the tail of the chain. Whenever a process receives a write, it executes the write and forwards it along the chain. Whenever a process receives a read, it executes and answers the read. Now, *assuming the existence of a consensus service for agreement on membership (chain structure)*, it becomes very easy to guarantee sequential consistency for reads and writes. The reason is that all operations are executed in a total order by the tail of the chain, and there is a very limited number of failure scenarios. (See van Renesse and Schneider [vRS04] for the full protocol and proofs.) The insight here is that it is not necessary to run an instance of Paxos or some other expensive protocol for every single read or write, but only in case of changes to the membership[5].

Van Renesse and Schneider [vRS04] suggest using multiple, overlapping instances of chain replication to implement replication groups in a distributed hash table. Following up on this idea, we propose to use chain replication for replication groups in Rubberband. That is, single-key read and write messages are sent directly to the storage processes, which order and disseminate them via chain replication. Consequently, throughput for single-key operations can be scaled linearly with the number of storage processes (assuming a constant replication factor). The rest of the system remains as-is. In particular, the elasticity features of Rubberband are not affected, since chain replication can naturally handle dynamic membership.

Mixing Single-Key and Multi-Key Operations

We previously explained that sequential consistency does not compose, in that histories that individually are sequentially consistent cannot, in the general case, be composed into a single, compatible history that is sequentially consistent. Agreement on the order of conflicting operations is necessary to ensure that only composable histories manifest.

In the extension to Rubberband we propose, all multi-key operations are sent and ordered through E-Cast, while all single-key transactions are sent directly to the storage processes, to be ordered through chain replication. Figure 4.8 gives an overview of such an extended Rubberband instance with 6 storage processes and a replication factor $f = 3$. Each storage process is the head of some replication group (chain) and the tail of another. The diagram shows where client processes send single-key and multi-key operations that are directed at the first replication group.

[5]This insight expands to other protocols, not just chain replication. Indeed, many atomic broadcast protocols (e.g. Zab [JRS11] used by Yahoo! ZooKeeper [HKJR10]) use Paxos only to elect a leader, and then let the leader order regular messages. However, these protocols are typically much more complex than chain replication.

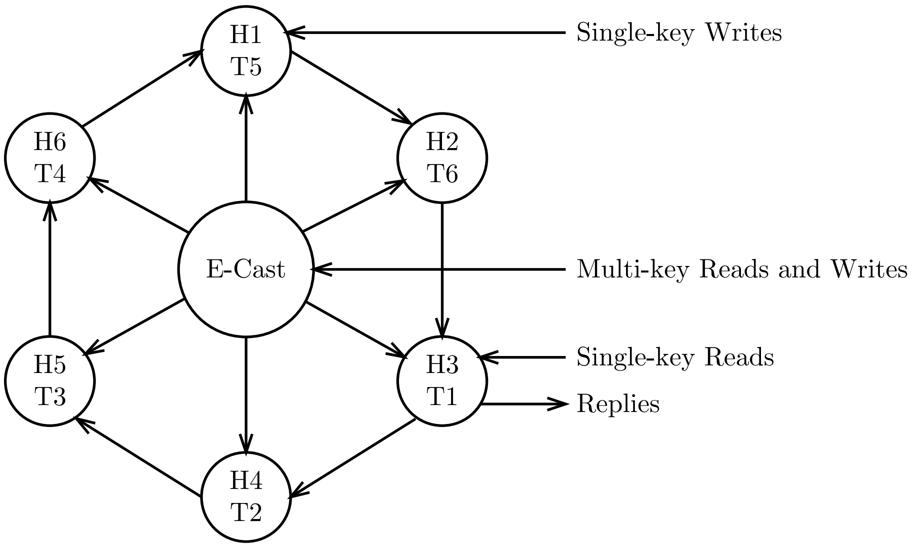


Figure 4.8: *Rubberband with Chain Replication*

While E-Cast and chain replication individually guarantee a total order of conflicting operations, the composition of the two does not, at least not automatically. Consider the following example. Let there be 2 groups (chains) of storage processes. The first group replicates data object x, and the second group replicates data object y. The following 4 writes are concurrently submitted:

$$\begin{aligned} w_1 &= \{x \leftarrow 1\} \\ w_2 &= \{x \leftarrow 2\} \\ w_3 &= \{x \leftarrow 3,\ y \leftarrow 3\} \\ w_4 &= \{x \leftarrow 4,\ y \leftarrow 4\} \end{aligned}$$

Assume E-Cast decides that w_3 precedes w_4 in the history of multi-key writes, and chain replication decides that w_1 precedes w_2 in the history of single-key writes to x.

Let the first replication group consist of processes s_1 and s_2, where s_1 is the head of the chain. It is entirely possible that s_1 delivers the writes in the order $\langle w_3, w_1, w_2, w_4\rangle$, and s_2 delivers the writes in the order $\langle w_3, w_4, w_1, w_2\rangle$. Each history preserves the total order of single-key writes, as well as the total order of multi-key writes. But the histories do not compose, because w_3 and w_4 conflict with w_1 and w_2 and the processes disagree on the order. Consequently, storage process s_1 will finish in state $\{x \leftarrow 4, y \leftarrow 4\}$, while s_2 will finish in state $\{x \leftarrow 2, y \leftarrow 4\}$.

As a solution, have chain replication not just forward each single-key operation, but also information about which of the multi-key operations (if any) immediately precedes the forwarded single-key operation in the execution order chosen by the head of the chain.

Let us write $\langle o, o' \rangle$ to represent a message that contains the single-key operation o', as well as the information that o is the multi-key operation that immediately precedes o'. In the example, storage process s_1 would then send the following sequence of messages to s_2 via chain replication:

$$\langle w_3, w_1 \rangle$$
$$\langle w_3, w_2 \rangle$$

In practice, this partial ordering information can be efficiently encoded with timestamps. E-Cast already assigns a unique timestamp to every delivered message, so these timestamps can be used as placeholders for the multi-key operations. Assuming for example that w_3 has the timestamp 47, storage process s_1 would send the messages $\langle 47, w_1 \rangle$, $\langle 47, w_2 \rangle$.

Define $<$ as the minimal transitive relation such that $o < o'$ holds iff

- o and o' are multi-key operations where E-Cast serializes o before o', or
- o and o' are single-key operations on the same key, and chain replication serializes o before o', or
- o is a multi-key operation, o' is a single-key operation, and chain replication delivers some message $\langle o, o' \rangle$.

Theorem 2. *For any pair of delivered, conflicting operations o, o', either $o < o'$ or $o' < o$ (but not both).*

Proof. E-Cast delivers multi-key operations in strict total order that is compatible with $<$. Likewise, chain replication delivers operations on the same key in strict total order that is compatible with $<$. What remains to be shown then, is that for any mixed pair of single-key and multi-key operation o, o', either $o < o'$ or $o' < o$.

We assume the theorem is wrong and proof by contradiction. That is, we assume there is either a pair of conflicting operations o, o' where no relation holds (Case 1), or both $o < o'$ and $o' < o$ (Case 2). Without restriction of generality, assume o is a multi-key operation, and o' is a single-key operation.

Case 1: No relation holds, so no message $\langle o, o' \rangle$ was delivered by chain replication. Likewise, no message $\langle o'', o' \rangle$ where $o < o''$ or $o'' < o$ could have been delivered, since $<$ is a transitive relation and then $o < o'$ or $o' < o$. But then no message containing o' could have been delivered at all, contradicting the assumption.

Case 2: Both $o < o'$ and $o' < o$ hold. Since $o < o'$ holds, it must be that chain replication delivered a message $\langle o'', o' \rangle$ where $o'' = o$ or $o'' < o$. Also, $o' < o$ holds, so chain replication must have delivered a message $\langle o''', o' \rangle$ where $o < o'''$. Thus, it must hold that $o'' < o'''$. But chain replication guarantees to deliver exactly one message with o', so it must be that $o'' = o'''$. This contradicts the fact that E-Cast delivers multi-key operations in strict total order.

Both cases lead to a contradiction. This completes the proof. □

We have just proven that agreement on which multi-key operation immediately precedes any given single-key operation yields a partial order relation $<$, which defines an order between every pair of conflicting operations. To maintain consistency in the system it is therefore sufficient to guarantee that every storage process executes delivered operations in an order that is compatible with $<$.

Guaranteeing Liveness

If every multi-key operation was broadcasted and executed by every storage processes, it would be sufficient to delay the execution of every single-key operation o until the storage process in question executes the multi-key operation that has been agreed to immediately precede o. However, Rubberband does not broadcast multi-key operations, it *multicasts* them. Not every storage process sees every multi-key operation. To guarantee liveness of the proposed protocol, additional measures are thus necessary.

Assume that whenever the head of a replication chain delivers a single-key operation, it immediately forwards it down the chain. To every message passed down a replication chain, let the sender process attach the timestamp of the latest multi-key operation it has been delivered by E-Cast. The following protocol invariant then follows directly from the *update propagation invariant* of chain replication [vRS04].

Invariant. *At the point where a storage process s learns that a process up a replication chain has been delivered the multi-key operation o, it is guaranteed to have seen every single-key operation of that replication chain that is meant to be executed before o.*

Another important invariant follows immediately from the uniform total order guarantee (page 103) made by E-Cast.

Invariant. *At the point where a storage process s is delivered a multi-key operation o by E-Cast, it is guaranteed to have been delivered every multi-key operation $o' < o$ of which s is in the destination set.*

Define $lb(s)$ as the lowest-ordered multi-key operation that has either been delivered to a storage process s, or that has been delivered to any replication chain that s is a member of.

We call $lb(s)$ the *lower-bound* multi-key operation of s. Due to the two invariants above, it is safe for a storage process s to execute any pending (delivered but not executed) single-key operation ordered before or immediately after $lb(s)$. Provided there are no lower-ordered single-key operations, it is also safe for s to execute $lb(s)$. (Obviously, s must also respect the order relation $<$ if there are multiple operations that are safe to execute.)

To ensure *liveness*, it must be guaranteed that for every storage process s, every multi-key operation delivered to s eventually becomes $lb(s)$, and for every single-key operation o delivered to s, eventually $o < lb(s)$, or $lb(s) < o$ and $lb(s)$ is the immediate predecessor of o as determined by chain replication.

The following two scenarios threaten liveness.

- No more single-key operations arrive at a storage process s, so s does not inform processes down the chain about the multi-key operations it has been delivered. Consequently, the processes down the replication chain cannot advance their lower bound, and cannot execute pending operations.
- No more multi-key operations arrive at a storage process s, so s cannot advance $lb(s)$. Any single-key operation o where $lb(s) < o$ and $lb(s)$ is not the immediate predecessor of o can never be executed.

To overcome the problem, we propose to send periodic null-messages, driving the protocol forward in the absence of incoming operations. This idea goes back to the classic Chandy-Misra-Bryant (CMB) algorithm for distributed parallel simulation [Bry77, CM81]. For liveness, it is sufficient that i) storage processes periodically send null-messages containing information about their latest, delivered multi-key operation down the chain; and ii) that null-messages are periodically broadcasted through E-Cast. It should be easy to see that this guarantees that for any storage process s, $lb(s)$ will eventually advance past any pending operation. Alternatively, a gossip protocol (epidemic protocol) can be used to disseminate the information [DGH+87].

Summary and Conclusion

We have presented an extension of Rubberband that allows client processes to send single-key reads and writes directly to the affected storage processes, without losing uniform, total order delivery guarantees. The proposal combines E-Cast with chain replication [vRS04]. It uses partial-ordering information attached to chain-replicated messages to efficiently enforce an execution order between any pair of conflicting operations.

Because chain replication does not require global coordination in the absence of reconfiguration, the resulting system should scale (almost) linearly with the single-key workload as storage processes are added. The price to be paid is some additional read and write latency, since storage processes must delay the execution of delivered operations to guarantee safety.

4.6 Experimental Evaluation

In this section, we present the results of an experimental evaluation of E-Cast and Rubberband on top of a dummy engine. Instances of this engine contain no data and immediately acknowledge all incoming operations, allowing us to assess the throughput and latency properties of E-Cast. Full-scale, end-to-end experiments of Rubberband on top of Crescando and Memcached are presented in Section 5.6 at the end of the next chapter.

As a baseline for comparison, we have also implemented an inelastic variant of Rubberband that uses Spread [ADS00] instead of E-Cast. Spread is an open-source[6], commercially supported multicast toolkit known for very high performance. Note that Spread does not offer the same level of consistency guarantees as E-Cast (discussed later), and the Spread-based variant of Rubberband does not support dynamic reconfiguration.

4.6.1 Platform

All experiments were run on a cluster of 30 dual Intel Xeon L5520 2.26 GHz (i.e., 8-core) machines with 24 GB RAM, connected through 1 Gbit Ethernet. The machines were running Linux, kernel version 2.6.32. Rubberband/E-Cast were running on Erlang/OTP[7] version R14B02. Both Rubberband/E-Cast and OTP were compiled to native code, using the HiPE compiler [SPC+03] shipped with Erlang/OTP. The Spread benchmark clients were written in Java, running on Sun JDK 64-Bit version 1.6.0-22.

4.6.2 Workload and Configuration

Unless otherwise indicated, experiments were run with 1 client, 16 storage, 1 super, and 3 router processes. Storage process identifiers were uniformly distributed over the key domain, 32-bit unsigned integers. Each read or write was directed at a single key, selected randomly according to a uniform distribution over the key domain. Thus, for a replication

[6] `http://www.spread.org`, retrieved January 20, 2012.

[7] OTP is the collection of Erlang standard libraries and tools.

factor f, each read was delivered to 1 storage process, each write was delivered to f storage processes, and the load was evenly balanced across the storage processes. Each message carried a payload of 256 byte dummy data. Each data point was averaged over 5 runs of at least 3 minutes each. Error bars show the standard deviation.

An important feature of Rubberband is its support for multi-key and multi-range operations. But from the perspective of E-Cast, there is no difference between delivering an operation to many processes because it affects many keys, or because the replication factor is high. We therefore restricted the experiments to single-key reads and writes, but using different replication factors.

Whenever we write *transaction* in this section, we mean the period from the point where the client process submits a read or write message to E-Cast, to the point where the client process receives an acknowledgement from E-Cast that the request has been reliably delivered to all destination storage processes. Each client process was submitting read and write messages asynchronously, while limiting the number of pending transactions to 1,024. This limit was chosen high enough for a single client to saturate E-Cast and Spread.

4.6.3 Write Throughput vs. Number of Storage Processes

In the first experiment, we investigated how write throughput in Rubberband changes with an increasing number of storage processes. The generalization discussed in Section 4.5.3 (E-Cast + chain replication) has not been implemented at the time of writing, so Rubberband sent all write messages through E-Cast in this experiment. The results thus show how E-Cast behaves with respect to an increasing number of (overlapping groups of) application processes receiving messages.

According to Figure 4.9, Rubberband sustained over 25,000 write transactions per second (TPS) for up to 2,048 storage processes with replication factor $f = 1$; 20,000 TPS with $f = 3$; and 15,000 TPS with $f = 5$. For 16,384 storage processes and $f = 3$, Rubberband could still sustain about 13,000 TPS.

One may argue that there is little point in supporting thousands of processes if the write throughput is limited to, say, 20,000 TPS. This may be true for some applications, but it overlooks that i) storage processes may be *virtual* (many storage processes per node/-machine); ii) single-key writes could be performed outside E-Cast, as discussed in Section 4.5.3; and iii) there are other use cases of E-Cast such as distributed simulations or online games, which require thousands of overlapping process groups, but not necessarily higher throughput, see Ostrowski et al. [OBDA08].

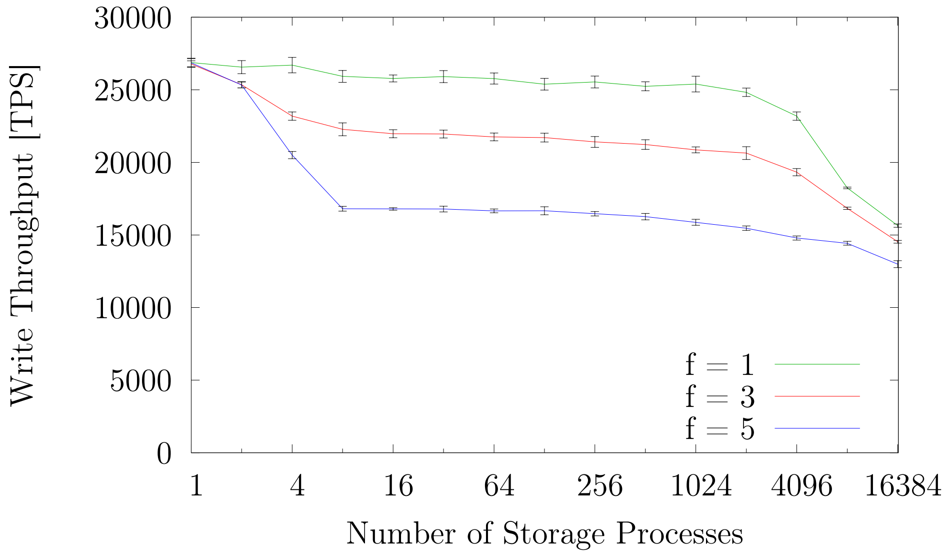


Figure 4.9: *Write Throughput: 256 B Payload, 3 Router, 1 Client, Vary Storage*

20,000 TPS is in fact high enough to solve many large-scale use cases. Schütt et al. [SSR08] report that the database back-end of Wikipedia sustains just 2,000 TPS. Google reports 3 billion write transactions per day for MegaStore [BBC+11], which is about 35,000 TPS on average. This totals over 100 applications and *many* entity groups (units of consistency), while Rubberband and E-Cast implement a *single* unit of consistency. And, most importantly, 20,000 TPS is high enough to solve the Amadeus Ticket use case, which requires only about 1,000 TPS.

We repeated this experiment for sequential reads rather than writes. Irrespective of the replication factor, sequential read throughput was identical to write throughput for $f = 1$, which is not surprising given the fact that Rubberband makes E-Cast forward read and write messages in read-one-write-all (ROWA) manner. We thus show only results for write throughput here. An experiment that investigates read throughput at different isolation levels is presented in Section 4.6.7.

E-Cast vs. Spread

To put the performance of E-Cast into perspective, we also implemented an inelastic variant of Rubberband based on Spread [ADS00]. We configured Spread so that each individual storage process and each replication group on the ring formed a multicast group. This allowed us to perform read-one-write-all (ROWA) message routing as in E-Cast. We used Spread version 4.1, configured for (mostly-)uniform causal total order delivery (the

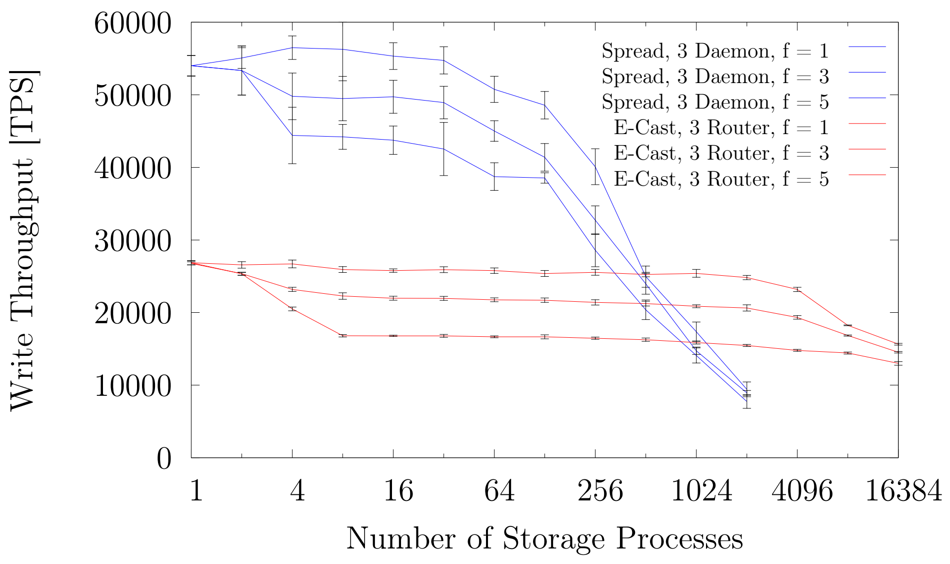


Figure 4.10: *E-Cast vs. Spread, Write Throughput: 256 B Payload, 3 Router/Daemon, 1 Client, Vary Storage*

strongest available guarantees in Spread) over unicast network links. We then repeated the previous experiment.

This was not an apples-to-apples comparison. E-Cast guarantees uniform delivery of messages in *all* cases, whereas Spread does not guarantee uniform delivery in case of router (called *daemons* in Spread) failure and membership changes [Sta02]. In addition, the mapping of key ranges to process groups in Spread cannot be changed easily[8]. On the other hand, Spread offers a larger choice of ordering and delivery guarantees than E-Cast, and can be configured to perform well on wide-area networks. Finally, E-Cast is implemented in Erlang, a high-level, functional programming language, while Spread is implemented in C. Despite these important differences, the comparison with Spread sets the performance of E-Cast into perspective with the state of the art.

Figure 4.10 shows that for 3 daemons, 16 storage processes, and a replication factor $f = 3$, Spread could sustain 50,000 write transactions per second. However, for more than 1,024 storage processes, E-Cast became faster than Spread. When adding more than 2,048 storage processes, Spread became extremely unstable and repeatedly crashed before the benchmark could complete.

As Figure 4.11 shows, adding daemons allowed Spread to scale to higher numbers of

[8]Consider what happens if a client process sends a write message while the assignment of key ranges to process groups changes concurrently.

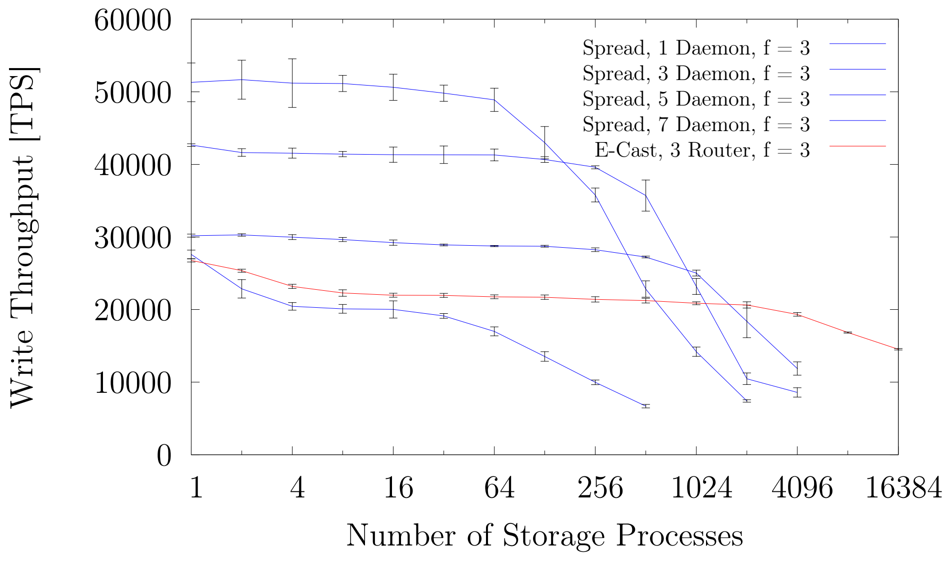


Figure 4.11: *E-Cast vs. Spread, Write Throughput: 256 B Payload, 3 Router, 1 Client, 16 Storage, Vary Daemon*

storage processes, but at the cost of significantly lower throughput for low numbers of storage processes. The reason is that Spread partitions the load among daemons, which appears to work best for 3 daemons in our experiment. Still, regardless of the number of daemons, E-Cast out-performed Spread for high numbers of storage processes. Also note that even with 7 Spread daemons, the benchmark failed to complete for more than 4,096 storage processes.

In summary, at its sweet spot, Spread achieved roughly twice the throughput of E-Cast. However, Spread did not provide the same consistency guarantees as E-Cast (notably, uniform delivery in case of router/daemon failure), and did not scale as well with the number of storage processes. Most importantly, in the Spread-based implementation, clients were holding a static mapping of keyspace partitions to replication groups. Making this mapping fully *dynamic*—the main contribution of E-Cast and Rubberband—would require additional messages and/or protocol rounds with Spread, at the expense of actual throughput.

4.6.4 Write Throughput vs. Number of Client Processes

Next, we measured throughput for a varying number of client processes. This tested both the ability of E-Cast to efficiently preserve causal order, as well as the protocol's ability to handle congestion.

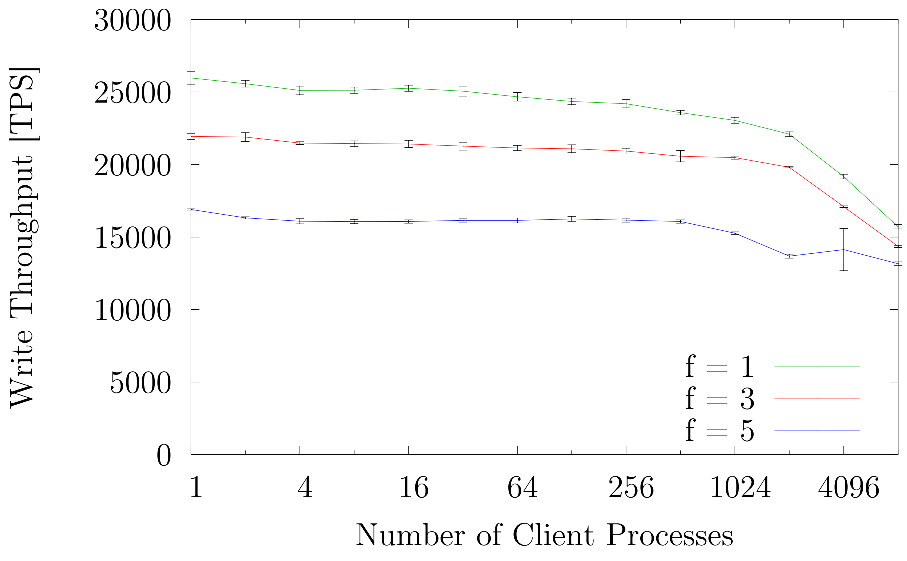


Figure 4.12: *Write Throughput: 256 B Payload, 3 Router, 16 Storage, Vary Client*

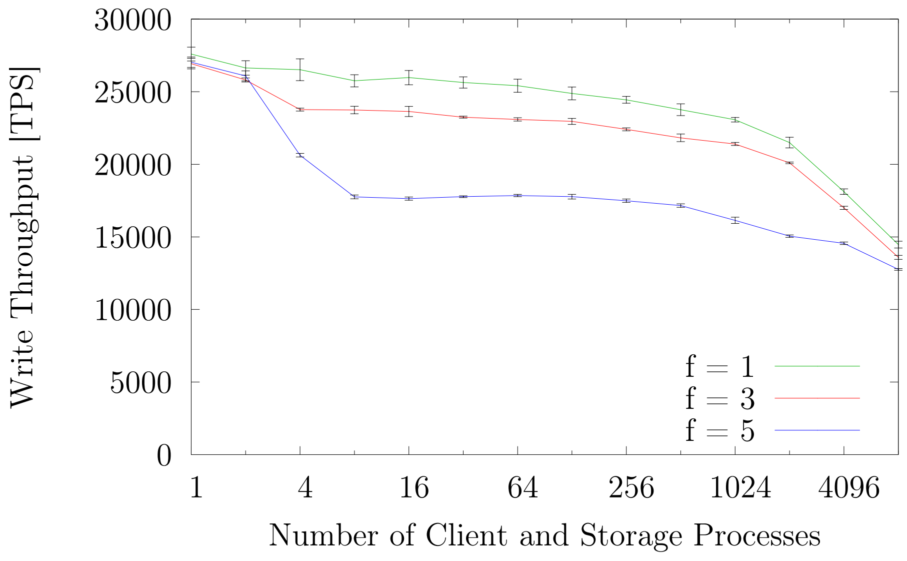


Figure 4.13: *Write Throughput: 256 B Payload, 3 Router, Vary Client and Storage*

Figure 4.12 shows that Rubberband's throughput stayed almost constant up to around 1,024 clients. Traces of individual clients (not shown) revealed that all client processes achieved roughly the same, steady throughput, demonstrating the effectiveness of E-Cast's congestion control scheme (cf. Section 3.5.6). For over 1,024 clients, throughput began to degrade logarithmically, similar to increasing the number of storage processes.

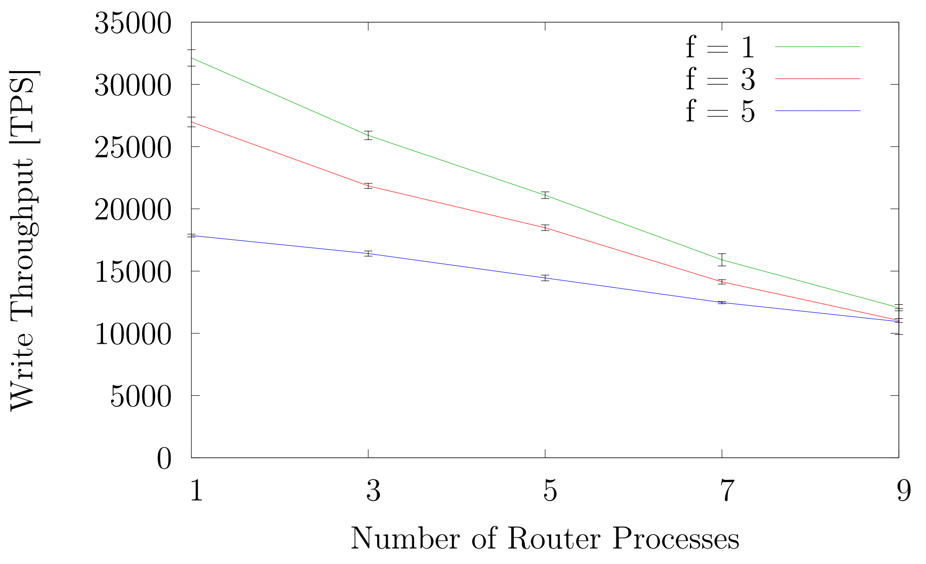


Figure 4.14: *Write Throughput: 256 B Payload, 1 Client, 16 Storage, Vary Router*

According to Figure 4.13, throughput remained stable even when scaling *both* the number of storage and client processes at the same time. In other words, E-Cast can handle very large numbers of sender and receiver processes at the same time.

We attempted to repeat these experiments with Spread, but the Spread daemons quickly crashed whenever more than 32 clients were flooding the system with messages. We suspect that Spread lacks the necessary congestion control features.

4.6.5 Write Throughput vs. Number of Router Processes

In the next experiment, we measured how write throughput of Rubberband changed as we scaled the number of router processes. As Figure 4.14 shows, throughput decreased as router processes were added. This is not surprising, since router processes in E-Cast form a reconfigurable, replicated state machine. No work is partitioned between routers, though this could change if the atomic broadcast protocol were replaced with a somewhat more scalable protocol such as Ring-Paxos [MPSP10]. That being said, the drop in throughput from 3 to 5 routers was only around 10%, so the additional fault tolerance may well be worth the cost in performance for some applications.

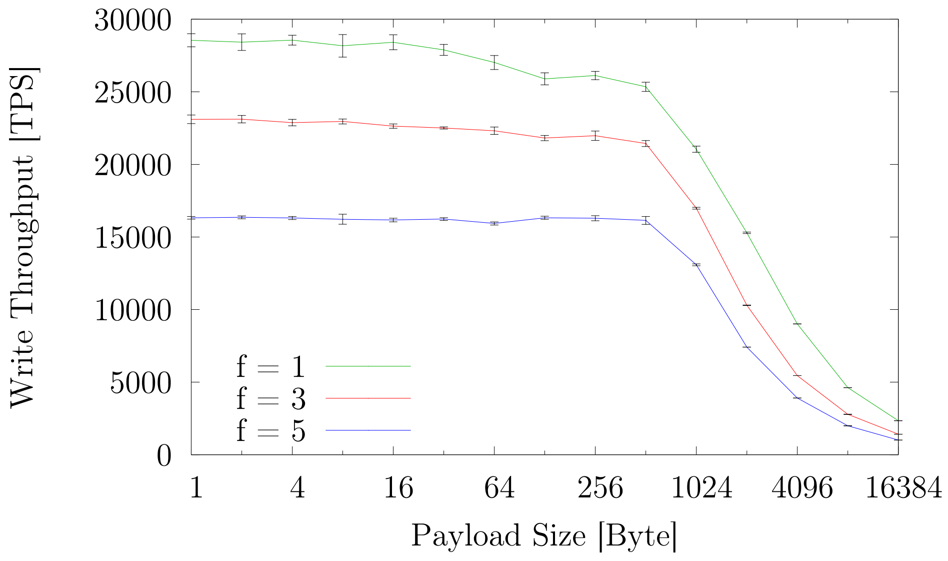


Figure 4.15: *Write Throughput: 3 Router, 1 Client, 16 Storage, Vary Payload*

4.6.6 Write Throughput vs. Payload Size

Figure 4.15 shows how the write throughput changed as we varied the (dummy) payload per message. The curves and additional kernel traces we conducted (not shown) indicate that E-Cast was compute-bound up to a payload size of roughly 1 KB. For larger payloads, the 1 Gbit Ethernet interface of the lead router became the bottleneck. In the current implementation, each write message payload must be sent $f + r - 1$ times[9] by the lead router, where f is the replication factor on the ring, and r is the number of routers. So for $f = 3$ and $r = 3$ for example, each write message is sent 5 times by the lead router. The numbers in Figure 4.15 match the analysis.

Rubberband is clearly not optimized for transmitting very large messages. Applications that require extremely high bandwidth, such as distributed file systems, should consider sending large payloads out-of-band, using E-Cast only for critical meta-data and concurrency control. For most applications, in particular NoSQL data stores, the available bandwidth appears sufficient as-is however, especially since (potentially large) message replies typically need not go through E-Cast. For a concrete example, see Section 5.1 on how Crescando/RB returns read results directly to client processes.

[9]This could be optimized, see Section 3.5.7.

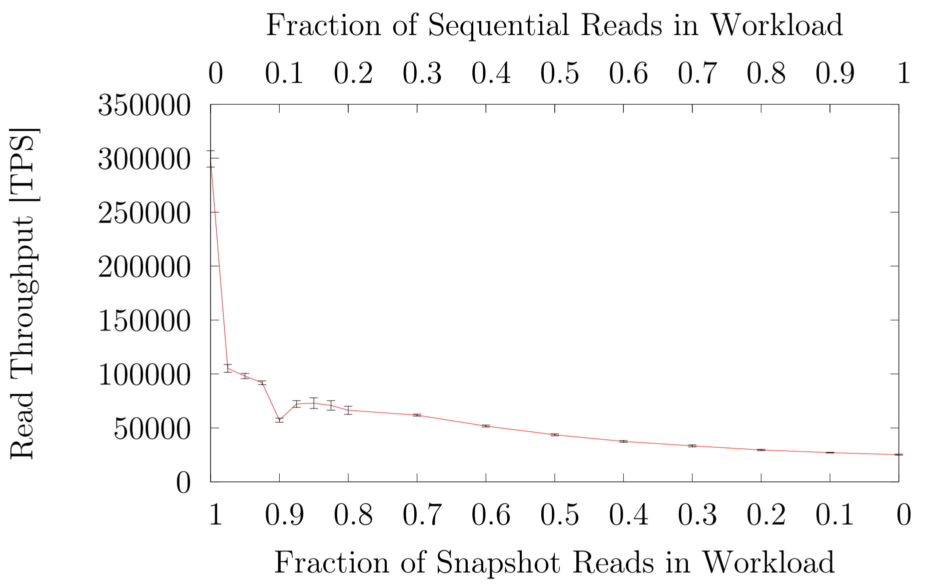


Figure 4.16: *Read Throughput: f = 3, 256 B Payload, 3 Router, 1 Client, 16 Storage, Vary Ratio of Sequential Reads to Snapshot Reads*

4.6.7 Read Throughput vs. Isolation Level

Rubberband supports 3 levels of isolation for reads: sequential, snapshot, basic reads. Sequential reads are sent through E-Cast just like writes. Snapshot reads are not atomically broadcasted between routers, but are directly timestamped by the lead router and forwarded to the destination processes. And basic reads are not even sent to the router processes, but directly to the storage processes. Further details are given in Section 4.5.2.

In the absence of a concrete storage engine, there is little point in measuring basic read throughput. Basic reads completely bypass the E-Cast protocol. In this experiment, we therefore investigated only how read throughput changes in accordance with the ratio of sequential reads to snapshot reads in the workload. Figure 4.16 shows the results.

For a pure snapshot read workload, the system reached a throughput of around 300,000 TPS. Conversely, for a pure sequential read workload, throughput was roughly 25,000 TPS. Mixing the two types of reads did not cause any significant ill-effects. Only at 90% snapshot reads and 10% sequential reads do the results show a curious dip in throughput. We investigated the anomaly and found that this was just an artifact of sub-optimal process scheduling by the Erlang runtime. One router process consists of about 10 Erlang processes. For snapshot reads, a different Erlang process was the (compute-bound) bottleneck than for sequential reads. At a 9:1 ratio, both of these processes were under full load, at which point the Erlang runtime failed to optimally assign CPU cores to processes.

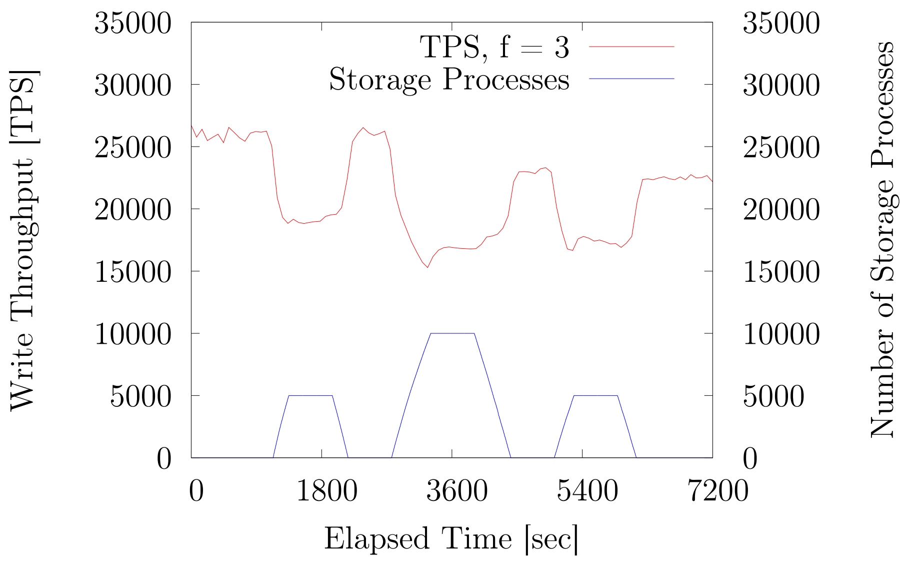


Figure 4.17: *Write Throughput: 256 B Payload, 3 Router, 1 Client, Elastic Storage*

In summary, sequential reads are over 10x as expensive as snapshot reads in the given configuration, at least as far as E-Cast and Rubberband are concerned. For complete systems that include a real storage engine and result shipping for reads, the difference is somewhat lower, see Sections 5.6.4 and 5.6.5 on Crescando/RB. Even so, applications can clearly gain significant throughput benefits from surrendering some read consistency.

4.6.8 Elasticity

In the next experiment, we dynamically scaled Rubberband from 1 to 10,000 storage processes and back, and measured how write throughput for $f = 3$ changed over time. Figure 4.17 plots the resulting trace.

The small dip in throughput before each plateau phase shows the performance overhead of reconfiguration in E-Cast. Evidently, it was negligible. By design, E-Cast and Rubberband do not stop transmitting messages during reconfiguration. The small overhead that was measured can be entirely attributed to the cost of computing and maintaining the sequence of Rubberband `Config` objects at the router processes, as described in Section 4.3.

The throughput at the end of the experiment was 10% lower than at beginning of the experiment, even though E-Cast had garbage-collected all state associated with stable messages and parted storage processes. We suspect that the Erlang runtime itself had accumulated some garbage state. Given that E-Cast rapidly recovers from router failures, an easy fix would be to restart router processes should a performance problem arise.

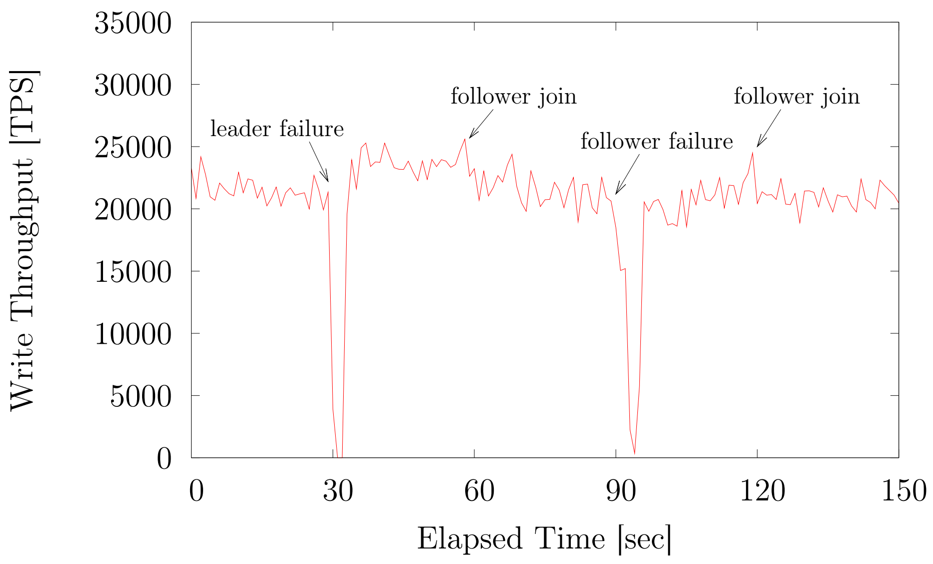


Figure 4.18: *Write Throughput: f = 3, 256 B Payload, 3 Router, 1 Client, 16 Storage, Fail Router*

In summary, the trace shows that E-Cast and Rubberband can elastically scale to large numbers of storage processes. The system never stopped transmitting messages, which is critical for asynchronous reconfiguration and continuous availability. Throughput remained stable, even during periods of massive system reconfiguration. The numbers here must be taken with a grain of salt of course, since a real application may have to transfer large amounts of data during reconfiguration. In Section 5.6, we show the results of various elasticity experiments with Rubberband on Crescando, where multiple gigabytes of data had to be transferred during each reconfiguration.

4.6.9 Tolerance to Router Failures

E-Cast routers form a reconfigurable, replicated state machine. An obvious question is thus, how quickly E-Cast can mask and compensate router failures. The answer is given by Figure 4.18, which shows a throughput trace for various failure scenarios. As in most experiments, we measured write throughput for a replication factor $f = 3$.

The trace starts with 3 router processes. After 30 seconds, the lead router was crashed. By design, all application processes talk to the lead router, so throughput immediately dropped to zero. Since there was uniform agreement on the sequence of unstable messages, and every router at any point was ready to send those messages, there was no state to be transferred after the router crash. The heartbeat protocol used by routers (cf. Section 3.5.5) was configured to declare routers faulty after 3 seconds. Accordingly, the trace shows that E-Cast resumed normal operation after 3 seconds. Throughput after the failure was slightly higher than before. This is because after the failure there were only 2 routers left, and E-Cast is faster for lower numbers of routers, as shown previously.

60 seconds into the experiment, the crashed router was replaced by a new router and throughput went back to the original number. Note that the protocol did not pause long enough (1 second sampling interval) for a throughput dip to appear in the trace where the new router joined.

90 seconds into the experiment, a follower router (i.e., not the leader) was crashed. Shortly after, E-Cast stopped transmitting messages. This is because E-Cast ensures that no router falls behind too far in processing messages, see Section 3.5.6 on flow control. After 3 seconds however, the routers reached agreement on the new membership, the failed router was removed from the system, and the remaining routers were allowed to resume normal operation. Again, throughput increased afterwards due to only 2 routers remaining.

120 seconds into the experiment, a new router joined again, and the system quickly stabilized at the original throughput.

In summary, the trace shows that E-Cast rapidly recovers from router failures, even at peak load. Together with the fact that E-Cast has higher throughput in "degraded" mode than under normal operation, this means that the protocol will not become overloaded as a result of router failures.

4.6.10 Latency vs. Write Throughput

In the final experiment of this chapter, we measured how transaction duration (latency) changed under varying load. One client would create a specific background load, while another client would send a single message and measure how long it would take to receive an acknowledgement. For each data point, the experiment was repeated 1,000 times.

Figure 4.19 shows the results. For readability, error bars are shown only for a replication factor of 3. In the upper chart, delayed acknowledgements and message clumping were disabled. In the lower chart, the two optimizations were enabled. The acknowledgement delay was set to 10 msec (the default), so routers waited up to 10 msec to piggyback a "stable"-message onto a "route"-message (cf. Section 3.5.7). Also, E-Cast routers would wait for up to 10 msec to clump 3 messages together before atomically broadcasting them.

As the charts show, the two optimizations roughly doubled the throughput. On the other hand, the optimizations caused increased latency under low load. An improved implementation of E-Cast could adaptively enable or disable throughput optimizations depending on the load.

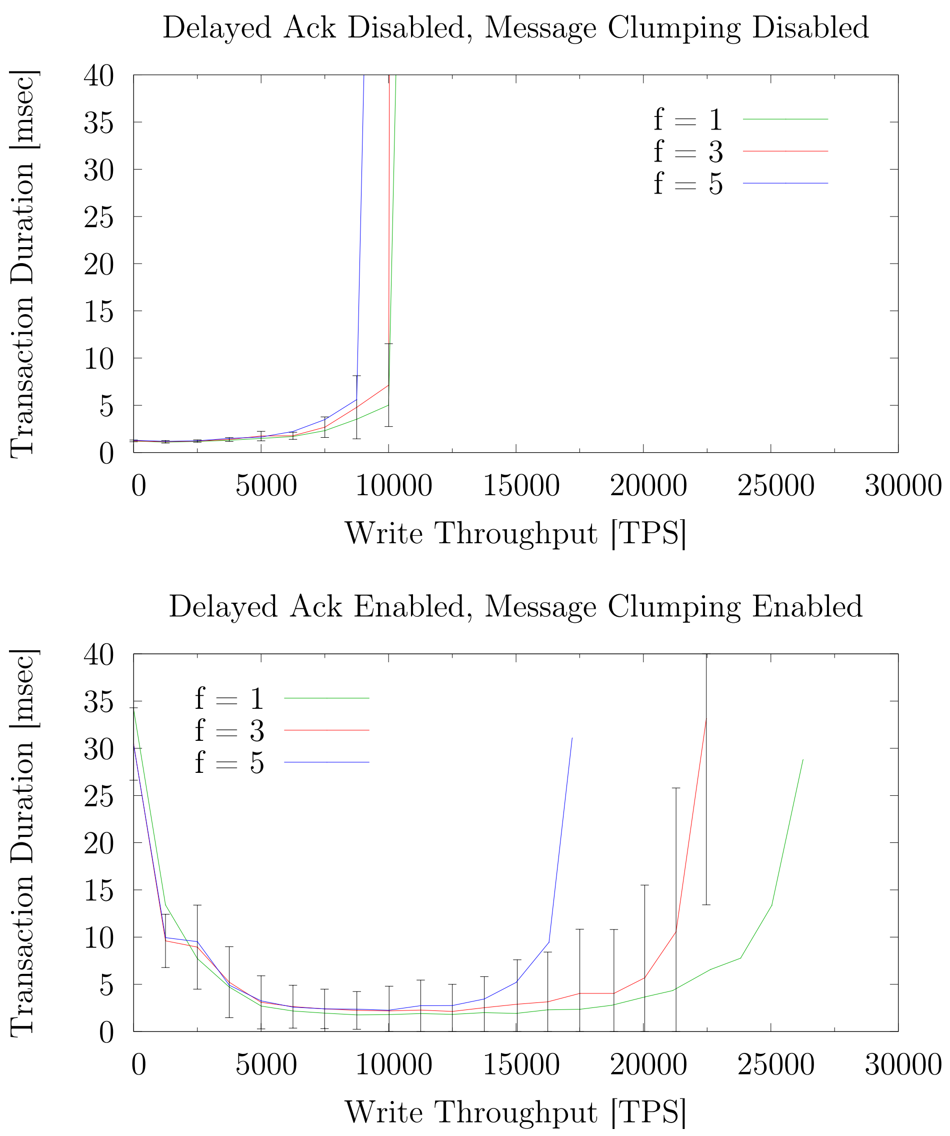


Figure 4.19: *Transaction Duration: 256 B Payload, 1 Background Client, 1 Latency Client, 16 Storage, Vary Load*

In absolute numbers, we measured a latency of roughly 2 msec at 10,000 TPS, with optimizations enabled. At peak throughput, the latency was 30 msec or higher, but it stayed below 10 msec for up to 95% of peak load. Unlike most protocols with similarly strong ordering and delivery guarantees, for example Zab [JRS11], E-Cast does not require write-ahead-logging on stable storage (disks, flash). Consequently, latency is dramatically lower for E-Cast.

In the case of Crescando/RB, the latency overhead of Rubberband and E-Cast can safely be ignored, since the Crescando storage engine is designed to answer queries in a matter of *seconds*, not milliseconds. All the more important however is the fact that Rubberband separates the reliable, ordered delivery of messages (by E-Cast), from their eventual, deterministic execution (by Crescando or some other storage engine).

4.7 Concluding Remarks

In this chapter, we have described the design and implementation of Rubberband, a NoSQL data store framework built on top of the E-Cast protocol. Rubberband is fully functional and implemented in about 9,000 lines of Erlang, in addition to the 10,000 lines of the E-Cast protocol. Rubberband delivers arbitrarily overlapping multi-key and multi-range reads and writes uniformly, in causal total order. This guarantees *sequential consistency.* Because Rubberband is based on E-Cast, it also tolerates permanent, silent failure of any process, does not require stable storage, and does not require perfect failure detectors. This makes Rubberband ideally suited for deployment in the cloud.

What is more, Rubberband is completely wait-free in the sense of being asynchronous and pipelined. A given client can have many read and write transactions pending at a time, without loss of consistency guarantees. This property holds even during elastic reconfiguration, which allows NoSQL data stores built with Rubberband to remain available during reconfiguration. We found that wait-free reconfiguration leads to an interesting optimization problem, particularly with respect to fault tolerance, and we have presented an efficient solution of said problem.

The experimental results show that Rubberband and E-Cast can handle about 20,000 write transactions per second in typical system configurations. This is enough for many large-scale, practical use cases, including the Amadeus Ticket use case. For future use cases, where throughput may be insufficient, we have presented a possible extension of Rubberband. The extension combines E-Cast with chain replication in order to scale linearly with the number of storage processes as regards single-key operations in the workload.

Chapter 5

Crescando/RB

In this chapter, we come full circle and combine the Crescando storage engine presented in Chapter 2 with the E-Cast/Rubberband middleware presented in Chapters 3 and 4. The result is a fully functional, relational NoSQL data store for the Cloud called *Crescando/RB*.

Crescando/RB combines the positive features of Crescando, E-Cast, and Rubberband. Owing to the Crescando storage engine, Crescando/RB is robust to mixed workloads and query diversity. And owing to E-Cast and Rubberband, Crescando/RB can be elastically and transparently scaled out to a large cluster of machines, providing increased throughput, storage space, and fault tolerance over a stand-alone instance of Crescando. Furthermore, Crescando/RB is entirely wait-free. The system design and API encourages users to enqueue a stream of operations and reconfiguration requests. Crescando/RB can thus be characterized as a push-based, distributed, fault-tolerant, parallel storage engine, which takes the idea of pipelining and resource sharing to the extreme.

This chapter is structured as follows. Section 5.1 presents the multi-tier architecture of Crescando/RB and provides an outline of the system's API toward user applications. Section 5.2 explains the consistency guarantees provided by Crescando/RB, and how they relate to Rubberband and E-Cast. Section 5.3 describes the different data partitioning schemes available in Crescando/RB, and how they are being implemented on top of Rubberband and E-Cast. Section 5.4 provides an overview of how partial live migration is implemented in the Crescando storage engine. Section 5.5 shows the graphical administrative console of Crescando/RB, as well as a graphical demo client. Finally, Section 5.6 presents the results of a comprehensive experimental evaluation of the complete system, and Section 5.7 makes concluding remarks.

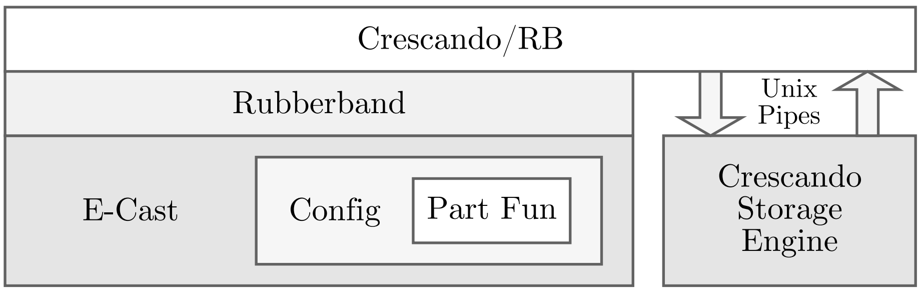


Figure 5.1: *Crescando/RB Multi-Tier Architecture*

5.1 Architecture

The complete multi-tier software architecture of Crescando/RB is visualized in Figure 5.1. Each Crescando/RB storage process holds an instance of the Crescando engine, which runs in a separate operating system (Linux) process, and communicates with the Erlang code through a pair of Unix pipes. All the cross-language wiring code is wrapped up in a self-contained library (not shown), which allows Crescando to be used in Erlang applications other than Crescando/RB. Obviously, client and super processes do not run storage engine instances, but other than that the architecture is the same for all types of processes.

As mentioned before, Crescando/RB and its individual components are all *wait-free*. The system design encourages users to enqueue a (high-rate) stream of operations and reconfiguration requests. Crescando/RB ensures that operations and requests are handled in causal total order, providing the illusion that the entire system is implemented by a single, highly available, sequential process.

Under the hood, of course, a Crescando/RB instance consists of many independent client, storage, super, and router processes. Each of those *logical* processes in turn consists of a number of physical processes. An E-Cast router process for instance consists of about 10 Erlang processes, each Crescando/RB client process spawns a separate (short-lived) Erlang process to collect results for each individual read operation, and every instance of the Crescando storage engine runs a number of scan threads. In other words, Crescando/RB forms a deep, parallel pipeline of processes through which operations, requests, and results are streamed. Crescando/RB can thus be summarized as a push-based, distributed, fault-tolerant, parallel storage engine.

Figure 5.2 gives an overview of the various processes that are involved in streaming operations and results between user code and Crescando engine instances. The diagram shows only how sequential reads and writes are streamed. Snapshot and basic reads skip some of the layers. To be precise, snapshot reads are not atomically broadcasted to other router processes, and basic reads are sent directly from client processes to storage processes at

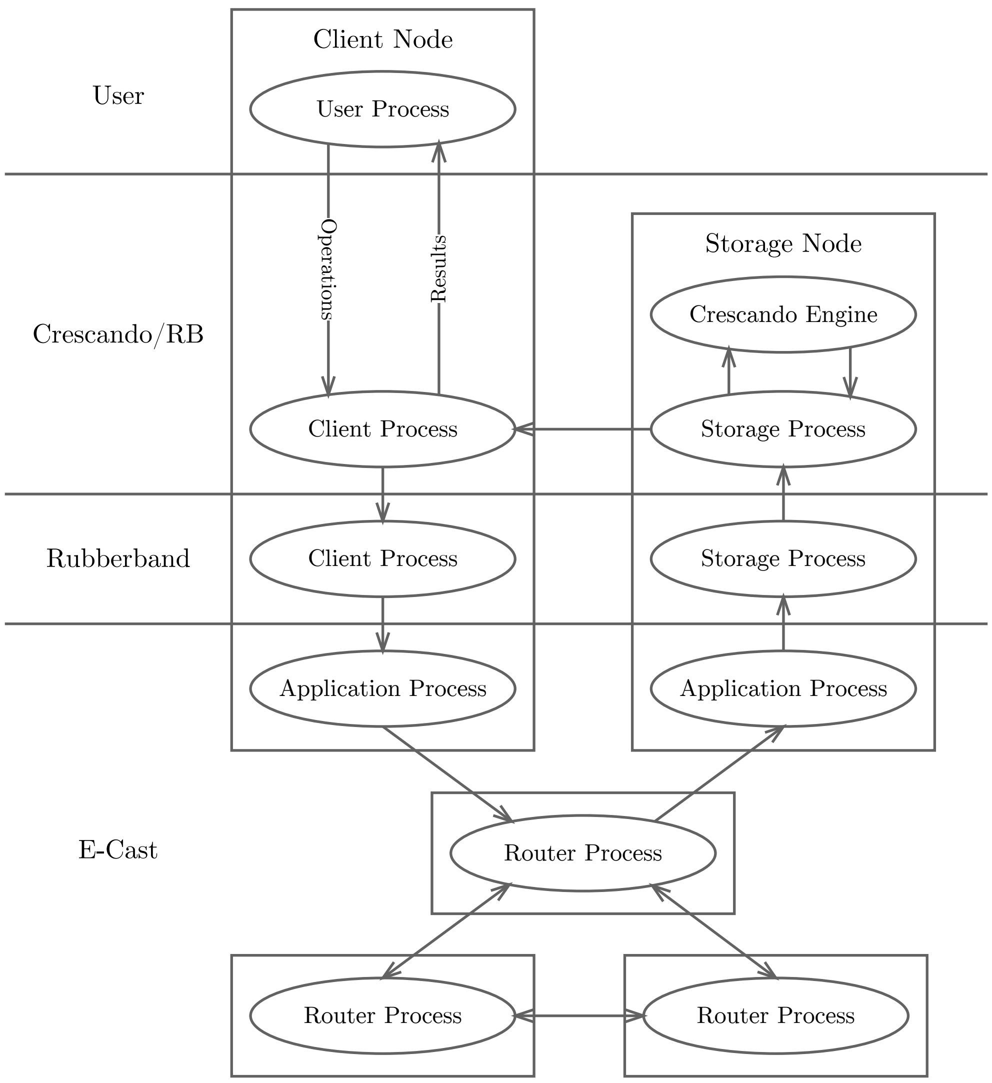


Figure 5.2: *Operation and Result Streaming in Crescando/RB*

the Rubberband layer. Also note, that the diagram shows only logical processes. The physical process pipeline is much deeper and highly parallel.

5.1.1 Interfaces

Toward the user, Crescando/RB provides an API that is similar to a stand-alone Crescando instance. Client processes allow users to enqueue insert, update, delete, and select operations. Result tuples are delivered back asynchronously, through Erlang process-to-process messages. Likewise, super processes allow users to make reconfiguration requests, and the results of these requests (success / failure) are eventually delivered back asynchronously.

The interface of client, storage, and super processes is written in Erlang, but can easily be accessed from other programming languages, through the Erlang distribution protocol[1]. Erlang/OTP includes libraries for C (`erl_interface`[2]) and Java (`jinterface`[3]) that allow C and Java programs respectively to exchange messages with Erlang processes in a convenient manner. The Crescando/RB administrative console and demo client presented in Section 5.5 for example are implemented in pure Java and make use of `jinterface`. Equivalent libraries for other programming languages are available from third parties.

The interface between storage processes and the storage engine is also kept simple, in order to make it easy to combine different storage engines with Rubberband. Rubberband delivers messages from a small set of well-defined types: data messages, keyspace messages, and suspect messages. Data messages contain read and write requests (select, insert, update, and delete operations), keyspace messages contain a list of simple instructions for partial live migration (cf. Section 4.4.1), and suspect messages inform the storage engine that one of its neighbors on the keyspace may have failed. The engine-specific wiring code between storage process and storage engine is responsible for passing these messages through to the storage engine (in original order), and for shipping the results of read and write requests to the respective client processes, see Figure 5.2.

In the case of Crescando/RB, this wiring code uses Unix pipes (called `ports` in Erlang) to exchange messages between Erlang code and the Crescando storage engine, which is implemented in C++. Alternatively, the Crescando storage engine could have been linked directly into the Erlang runtime (through so-called linked-in drivers or native-implemented-functions), but we found that Unix pipes offer the right trade-off between safety, convenience, and performance for our purposes.

Storage and super processes also feature a number of hooks for monitoring the system, accessible through the Erlang distribution protocol. Without these hooks, the system would be of limited practical use. The parameters that user applications can monitor include process status (active, standby, suspected/failed), keyspace (mapping of storage processes to identifiers), storage process load, and storage process space usage. The administrative console presented in Section 5.5 aggregates and displays much of this information.

[1]`http://www.erlang.org/doc/apps/erts/erl_dist_protocol.html`, retrieved January 25, 2012.
[2]`http://www.erlang.org/doc/apps/erl_interface/index.html`, retrieved January 25, 2012.
[3]`http://www.erlang.org/doc/apps/jinterface/index.html`, retrieved January 25, 2012.

5.2 Consistency and Isolation Levels

Crescando/RB offers the same consistency guarantees and read isolation levels as the underlying Rubberband framework, see Section 4.5. In this section, we revisit the consistency guarantees provided by Rubberband, and briefly explain how they relate to the consistency guarantees offered by Crescando storage engine instances.

Sequential Read By way of the `Config` module plugged into E-Cast (cf. Section 4.3), and by taking advantage of the formal properties of E-Cast (cf. Section 3.3), Rubberband ensures that every storage process that is affected by a data or configuration message is uniformly delivered that message. The interface code between Rubberband storage processes and their respective Crescando engine instances guarantees that data messages and repartitioning instructions are forwarded to the Crescando engine in delivery order. The interface code maps sequential reads in Rubberband to strict reads in Crescando. As explained in Section 2.5, the Crescando engine guarantees that writes and strict reads are executed strictly in input order. Consequently, Crescando/RB executes writes and strict reads in causal total order, which means Crescando/RB guarantees *sequential consistency* for writes and sequential reads, regardless of any concurrent reconfiguration.

Snapshot Read As explained in Section 4.5.2, snapshot reads are delivered in total order with respect to other snapshot reads, sequential reads, and writes. Since sequential consistency requires also FIFO order delivery, not just total order delivery, sequential consistency is not guaranteed. Still, Crescando/RB guarantees total order execution of writes, sequential reads, and snapshot reads. This is often referred to as *strong consistency* in the literature [FR10].

Basic Read Rubberband does not guarantee to deliver basic reads in any order with respect to other read and write operations. Since there are no global ordering guarantees, little would be gained from mapping basic reads in Rubberband to strict reads in Crescando. So, for optimal performance, basic reads in Rubberband are mapped to basic reads in Crescando.

5.3 Data Partitioning and Message Routing

Rubberband plugs a `Config` module into E-Cast, which tells E-Cast where to route messages (cf. Section 4.3). Rubberband itself requires users to plug a `Part Fun` (a function closure is called `fun` in Erlang) into the `Config` module. The `Part Fun` is expected to map any given read or write message to a partition of the key domain; i.e., the set of key ranges read or written by the given message. By calling this function, Rubberband can compute the destination set of read and write messages without even knowing the format of these messages. The Crescando/RB API defines a higher-order function that creates a `Part Fun` for use with Rubberband, given as input a table schema, a named attribute in that schema, and one of the partitioning schemes described below.

Users can freely choose any attribute in the table schema to be used for partitioning, including strings and dates. All Crescando data types are supported, see table 2.1 on page 19. Rubberband by itself performs untyped, dynamic range partitioning, using only Erlang term comparison, see Section 4.2.2. Crescando/RB however offers three different partitioning schemes, by providing three alternative implementations of a `Part Fun`.

Range Partitioning The `Part Fun` extracts and returns the key attributes of records in insert operations as-is. Consequently, the data is range partitioned, in accordance with the rules of successor replication laid out in Section 4.3. Range predicates of select, update, and delete operations are also extracted and returned as-is. It follows that read messages (select operations) and write messages (insert, update, delete operations) are all routed to a minimal set of storage processes. Write messages whose operations lack a predicate on the key attribute are broadcasted to all storage processes. Read messages whose select operations lack a predicate on the key attribute are multicasted to a minimal set of storage processes whose partitions cover the entire key domain. For n storage processes and a replication factor f, this means $\lceil n/f \rceil$ storage processes.

Hash Partitioning The `Part Fun` hashes the key attributes of records in insert operations to a 32-bit unsigned integer. Consequently, the data is hash partitioned (consistent hashing [KLL+97]) by Rubberband. Equality predicates of select, update, and delete operations are extracted and hashed by the `Part Fun`, so "point reads" and "point writes" are forwarded to 1 or f storage processes respectively, for a replication factor f. Range predicates however cannot be hashed effectively[4]. Update, delete, and select operations with range predicates on the key attribute are thus treated as-if they had no predicate on the key attribute, at least for purposes of message routing. Compared to range partitioning, hash partitioning makes it easier

[4]In the case of very small ranges over a discrete key domain, hashing the individual points of the range may be worth the effort. But in most cases, a range predicate will hash to a large enough set of keys that affects most or all active storage processes. Crescando/RB therefore does not attempt to hash range predicates.

to balance the load over the storage processes, but is obviously a poor choice in terms of performance if operations are expected to have range predicates.

Random Partitioning E-Cast requires the `destset` function of `Config` modules to be deterministic. To simulate random partitioning, Crescando/RB performs hash partitioning of the data, but ignores the predicates of select, update, and delete operations. Consequently, all update and delete operations are broadcasted to all storage processes, and select operations are multicasted to $\lceil n/f \rceil$ storage processes, where f is the replication factor and n is the number of active storage processes. Obviously, hash partitioning is always a better choice than random partitioning in practice, but random partitioning can be useful as a performance baseline.

5.4 Partial Live Migration

Crescando/RB implements all the reconfiguration functionality of Rubberband, described in Section 4.4. When a user (system administrator or some automatic load balancing tool) decides to change the current data placement; i.e., the mapping of storage processes to identifiers on the keyspace; they request an expand, contract, replace, or rebalance transformation from one of the super processes. In response to such a request, the super process creates and submits a configuration message, which contains an (ordered) list of instructions about which storage process must copy or delete which partition of its data. Storage processes forward these instructions to the storage engines, which are expected to execute the instructions in input order.

This whole scheme is called *partial live migration.* In this section, we provide an overview of partial live migration in Crescando/RB by way of a concrete example. The topic is covered in full detail by the Master's thesis of Ashmawy [Ash11].

Assume a client process submits the following sequence of sequential read and write messages: $\langle w_1, w_2, r_1, w_3, r_2 \rangle$. Concurrently, a super process submits a configuration message c_1. E-Cast may serialize these messages in the order $\langle w_1, w_2, r_1, c_1, w_3, r_2 \rangle$.

Assume that c_1 performs the same keyspace transformation as in the example of Section 4.4.1 on page 136. The transformation is visualized again in Figure 5.3. The transformation requires that storage process e receives copies of all data objects in the partition $\{[20, 40)\}$, and that storage processes b, c, and d delete part of their data. A possible list of instructions with this effect looks as follows.

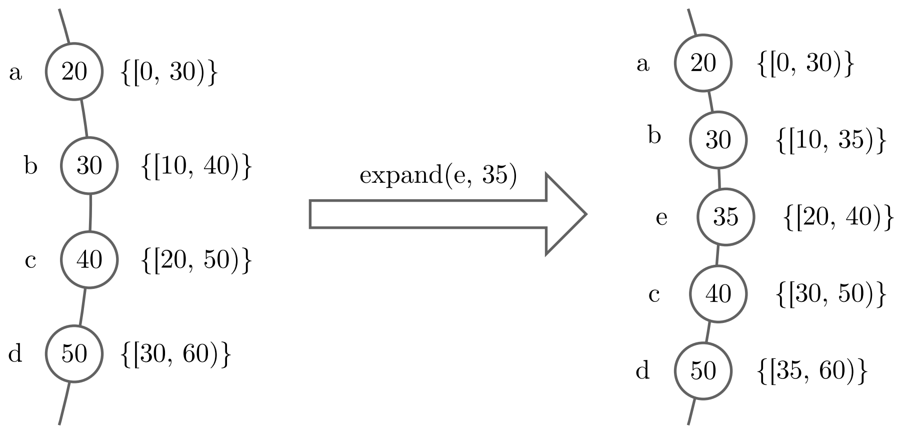


Figure 5.3: *Keyspace Transformation Example*

$$\begin{gathered}\langle \text{copy}, c, e, \{[20, 40)\}\rangle \\ \langle \text{delete}, b, \{[35, 40)\}\rangle \\ \langle \text{delete}, c, \{[20, 30)\}\rangle \\ \langle \text{delete}, d, \{[30, 35)\}\rangle\end{gathered}$$

According to this list of instructions, storage process c shall copy the entire partition $\{[20, 40)\}$ to storage process e. (There are in fact multiple alternative solutions, as discussed in Section 4.4.1, but this is not relevant here.)

Assume the different read and write messages affect the following ranges of keys:

$$\begin{aligned} w_1 :&\ \ [35, 35] \\ w_2 :&\ \ (-\infty, \infty) \\ r_1 :&\ \ [35, 35] \\ w_3 :&\ \ [35, 35] \\ r_2 :&\ \ (-\infty, \infty) \end{aligned}$$

In accordance with the message forwarding rules of read-one-write-all replication (cf. Section 4.2.2) and the data placement before and after c_1, E-Cast forwards each message to the following sets of storage processes:

$$\begin{aligned} w_1 :&\ \ \{b, c, d\} \\ w_2 :&\ \ \{a, b, c, d\} \\ r_1 :&\ \ \{b\} \\ c_1 :&\ \ \{b, c, d, e\} \\ w_3 :&\ \ \{e, c, d\} \\ r_2 :&\ \ \{a, c\} \end{aligned}$$

Let us look closer at storage processes c and e. Storage process c delivers the message sequence $\langle w_1, w_2, c_1, w_3, r_2 \rangle$, while storage process e delivers the message sequence $\langle c_1, w_3 \rangle$.

Storage process c transforms the delivered sequence of messages into to the following sequence of operations, to be executed by its storage engine:

$$\begin{aligned}
&\langle \text{write}, \mathit{op}(w_1) \rangle \\
&\langle \text{write}, \mathit{op}(w_2) \rangle \\
&\langle \text{send}, \langle \mathit{id}(c_1), 0 \rangle, e, \{[20, 40)\} \rangle \\
&\langle \text{delete}, \{[20, 30)\} \rangle \\
&\langle \text{write}, \mathit{op}(w_3) \rangle \\
&\langle \text{strict read}, \mathit{op}(r_2) \rangle
\end{aligned}$$

Likewise, storage process e forwards this sequence of instructions and operations to its storage engine:

$$\begin{aligned}
&\langle \text{receive}, \langle \mathit{id}(c_1), 0 \rangle \rangle \\
&\langle \text{write}, \mathit{op}(w_3) \rangle
\end{aligned}$$

We write $\mathit{op}(m)$ to describe the fact that the storage process does not forward entire data messages, but just the payload of the messages; i.e., the Crescando operation (insert, update, delete, select) inside the message. Likewise, $\mathit{id}(m)$ denotes a unique ID associated with message m.

When storage process e executes the instruction $\langle \text{receive}, \langle \mathit{id}(c_1), 0 \rangle \rangle$, it listens for incoming TCP connections until storage process c connects with a matching message ID and sequence number. Then, storage process e performs a strict read of the key range $[20, 40)$, streams the resulting records to storage process c (through the direct TCP connection), and storage process c performs a bulk-insert of what it receives.

The use of message IDs and sequence numbers requires explanation. While unlikely in practice, it is possible to construct examples that lead to a *deadlock* if processes were allowed to execute pending send and receive instructions in any order (consistency problems notwithstanding). The reason are possible cycles of send-receive dependencies between storage processes. Deadlock can be prevented by executing pending instructions in the same order (no circular wait) by all storage processes. To enforce this order, each instruction is tagged with the configuration message ID (our implementation just uses the timestamp assigned by E-Cast), and a monotonically increasing sequence number within the scope of the configuration message. In the example, the send and receive instructions are tagged with $\mathit{id}(c_1)$ and the sequence number 0.

At the time of writing, send, receive, and delete instructions are all mapped to regular select, insert, update, and delete operations of Crescando. For instance, to send all records that fall into a given range of keys, our implementation enqueues a strict select operation with a range predicate, and all the results that the local engine returns are forwarded to the remote engine, where they are bulk-inserted. (Optional) hash partitioning requires range predicates over hashed record attributes. These predicates are pushed down into Crescando in the form of UDFs (user-defined functions), see Section 2.2.

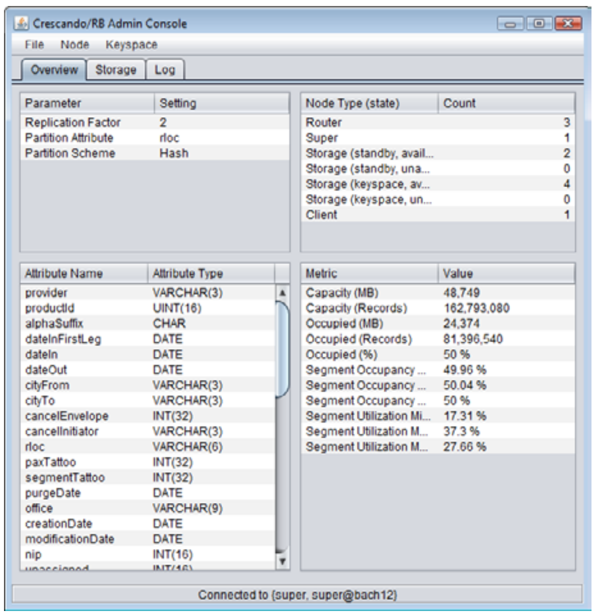
The complete, black-box implementation of partial live migration is described in detail in the Master's thesis of Ashmawy [Ash11]. In the future, it is certainly possible to modify the Crescando storage engine (white-box solution) for even better performance and availability during partial live migration. For example, it would be possible to take advantage of the local logging and snapshot functionality of Crescando, described in Section 2.8. Another option is to make consistent snapshots of Crescando instances using the `fork` system call, as proposed by Kemper and Neumann in the HyPer database system [KN11]. We refer to Ashmawy [Ash11] for further details and additional possibilities.

While the current implementation treats Crescando as a black box, the code is considerably more complex than the description here suggests. For one thing, the implementation must deal with process and link failures. For another thing, the implementation is able to overlap the execution of multiple repartitioning instructions and the preceding and succeeding read and write operations. For example, two *send* instructions can be executed at the same time, since the select operations that they map to do not conflict.

Overlapped execution improves performance and availability because Crescando is a push-based storage engine that is optimized for executing multiple operations in parallel. Things are complicated by the presence of basic reads, which are allowed to be re-ordered with respect to normal write operations, but should not be re-ordered with respect to receive, and delete instructions (this would violate configuration consistency). The resulting performance and consistency issues are discussed in detail by Ashmawy [Ash11].

5.5 Administrative Console and Demo Client

For purposes of demonstration and experimentation, we have implemented an administrative console and a demo client for Crescando/RB. Both applications feature graphical user interfaces, are written in pure Java, and communicate with Crescando/RB through the Erlang distribution protocol. The administrative console and demo client show that it is quite easy to use Crescando/RB from other programming languages.



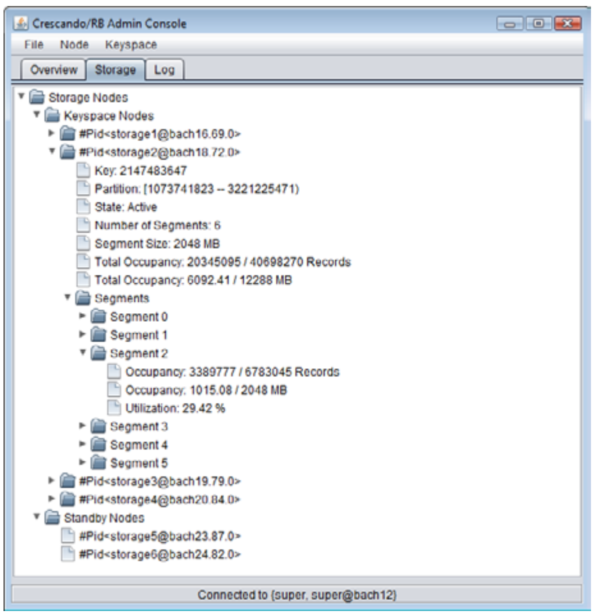
Parameter	Setting
Replication Factor	2
Partition Attribute	rloc
Partition Scheme	Hash

Node Type (state)	Count
Router	3
Super	1
Storage (standby, avail...	2
Storage (standby, una...	0
Storage (keyspace, av...	4
Storage (keyspace, un...	0
Client	1

Attribute Name	Attribute Type
provider	VARCHAR(3)
productId	UINT(16)
alphaSuffix	CHAR
dateInFirstLeg	DATE
dateIn	DATE
dateOut	DATE
cityFrom	VARCHAR(3)
cityTo	VARCHAR(3)
cancelEnvelope	INT(32)
cancelInitiator	VARCHAR(3)
rloc	VARCHAR(6)
paxTattoo	INT(32)
segmentTattoo	INT(32)
purgeDate	DATE
office	VARCHAR(9)
creationDate	DATE
modificationDate	DATE
nip	INT(16)
unassigned	INT(16)

Metric	Value
Capacity (MB)	48,749
Capacity (Records)	162,793,080
Occupied (MB)	24,374
Occupied (Records)	81,396,540
Occupied (%)	50 %
Segment Occupancy ...	49.96 %
Segment Occupancy ...	50.04 %
Segment Occupancy ...	50 %
Segment Utilization Mi...	17.31 %
Segment Utilization M...	37.3 %
Segment Utilization M...	27.66 %

Connected to {super, super@bach12}



```
▼ Storage Nodes
  ▼ Keyspace Nodes
    ▶ #Pid<storage1@bach16.69.0>
    ▼ #Pid<storage2@bach18.72.0>
        Key: 2147483647
        Partition: [1073741823 -- 3221225471)
        State: Active
        Number of Segments: 6
        Segment Size: 2048 MB
        Total Occupancy: 20345095 / 40698270 Records
        Total Occupancy: 6092.41 / 12288 MB
      ▼ Segments
        ▶ Segment 0
        ▶ Segment 1
        ▼ Segment 2
            Occupancy: 3389777 / 6783045 Records
            Occupancy: 1015.08 / 2048 MB
            Utilization: 29.42 %
        ▶ Segment 3
        ▶ Segment 4
        ▶ Segment 5
    ▶ #Pid<storage3@bach19.79.0>
    ▶ #Pid<storage4@bach20.84.0>
  ▼ Standby Nodes
      #Pid<storage5@bach23.87.0>
      #Pid<storage6@bach24.82.0>
```

Connected to {super, super@bach12}

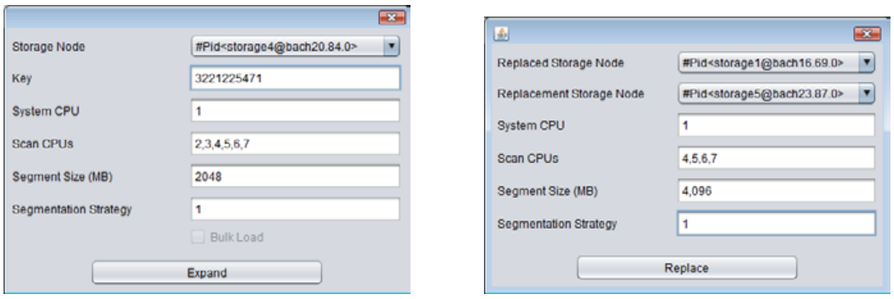
Figure 5.4: *Administrative Console: Overview and Storage Panel*

5.5.1 Administrative Console

Super and storage processes allow user processes to register as event listeners. Super processes inform their event listeners whenever the keyspace or membership changes. Likewise, storage processes periodically inform their event listeners of the current space usage and utilization. Users provide the hostname of a machine where a super process is running, and the console then registers as an event listener at the super process and (automatically) at every live storage process.

The incoming events are aggregated and visualized as shown by the two screenshots in Figure 5.4. Besides static configuration such as replication factor and table schema, the overview panel of the administrative console (left screenshot) also shows the number of processes of each type, and aggregate figures on space usage and system utilization. The storage panel (right screenshot) gives detailed information on each storage process, such as the data partition assigned to the storage process, and the space usage and CPU utilization of every individual memory segment / scan thread.

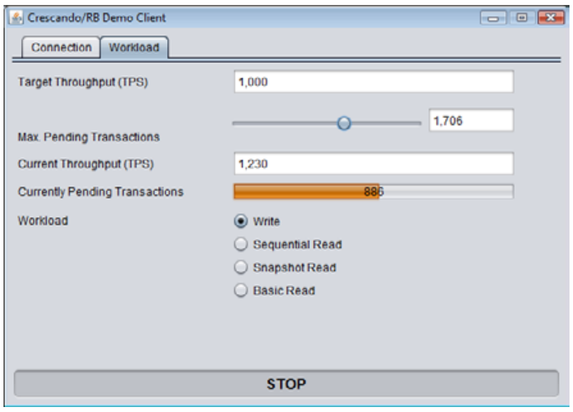
Besides giving human users a convenient way to monitor the system, the administrative console is also able to issue keyspace transformation requests to super processes. Figure 5.5, shows screenshots of expand and replace dialogs, which can be used to issue expand and replace requests, respectively. Corresponding dialogs exist for contract and rebalance requests.



Storage Node	#Pid<storage4@bach20.84.0>
Key	3221225471
System CPU	1
Scan CPUs	2,3,4,5,6,7
Segment Size (MB)	2048
Segmentation Strategy	1
	Bulk Load

Replaced Storage Node	#Pid<storage1@bach16.69.0>
Replacement Storage Node	#Pid<storage5@bach23.87.0>
System CPU	1
Scan CPUs	4,5,6,7
Segment Size (MB)	4,096
Segmentation Strategy	1

Figure 5.5: *Administrative Console: Expand and Replace Dialogs*



Target Throughput (TPS)	1,000
Max. Pending Transactions	1,706
Current Throughput (TPS)	1,230
Currently Pending Transactions	886
Workload	[✓ Write] Sequential Read Snapshot Read Basic Read

Figure 5.6: *Demo Client: Workload Panel*

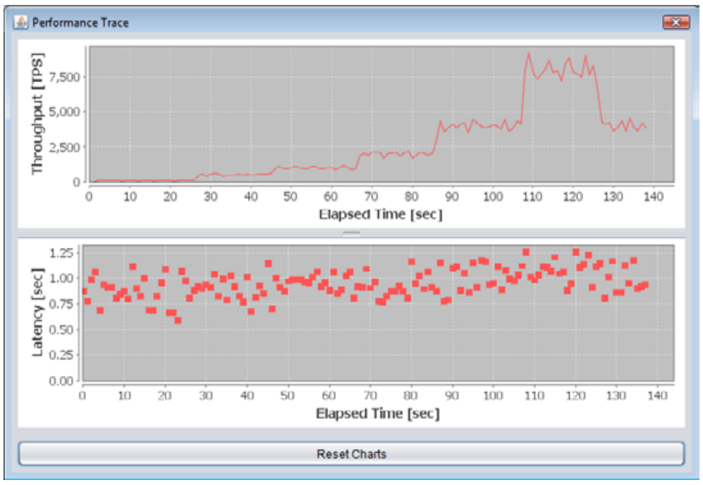


Figure 5.7: *Demo Client: Throughput and Latency Charts*

5.5.2 Demo Client

The demo client is a fairly simple Java application, which generates and sends read and write requests to a user-specified client process. The client process can run on the same machine as the demo client, but it does not have to (it makes no difference to the Erlang distribution protocol). Users can choose a target throughput (transaction output rate), an upper limit on the number of pending transactions, and a workload type (writes, sequential reads, snapshot reads, basic reads). The demo client then generates single-key operations of the desired type, using a leaky bucket algorithm for rate control [Tan02]. The graphical user interface for shaping the workload is shown in Figure 5.6.

The demo client measures the current throughput and latency (transaction duration) at a 1 second sampling interval. The measurements are plotted in continuously updated charts, shown in Figure 5.7. When combined with the administrative console, the demo client is a great way of experimenting interactively with the asynchronous reconfiguration and fault tolerance features of Crescando/RB.

5.6 Experimental Evaluation

In the final experiments section of this dissertation, we assess the scalability, elasticity, and fault tolerance of the complete system, Crescando/RB. We have also put Rubberband on top of Memcached [Fit04], a simple key-value store. This system, called Memcached/RB, is just a prototype and lacks the partial live migration code necessary for dynamic repartitioning. Nonetheless, it gives performance insights and is evidence that Rubberband is independent of any specific storage engine.

5.6.1 Platform

Like the experiments of the previous chapter, all experiments presented here were run on a cluster of 30 dual Intel Xeon L5520 2.26 GHz (i.e., 8-core) machines with 24 GB RAM, connected through 1 Gbit Ethernet. The machines were running Linux, kernel version 2.6.32. Crescando/RB was running on Erlang/OTP version R14B02, compiled to native code.

5.6.2 Data and Workload

All experiments used the Amadeus Ticket schema and realistic attribute-value distributions, as discussed in Section 2.10 and in more detail in the Master's thesis of Giannikis [Gia09]. All experiments were run with 3 router processes. The data was hash partitioned by *rloc* (booking number) in Rubberband, and round-robin partitioned across scan threads of Crescando engine instances. Unless otherwise indicated, each read message contained a select operation with a single equality predicate on *rloc*. Likewise, each write message contained an update operation with an equality predicate on *rloc*.

5.6.3 Write Scalability

In the first set of experiments, we measured how write throughput of Crescando/RB and Memcached/RB behaved with respect to an increasing number of storage processes. Each storage process was bulk-loaded with 2 GB of Amadeus Ticket data. In the case of Crescando/RB, each Crescando engine instance was running a single scan thread.

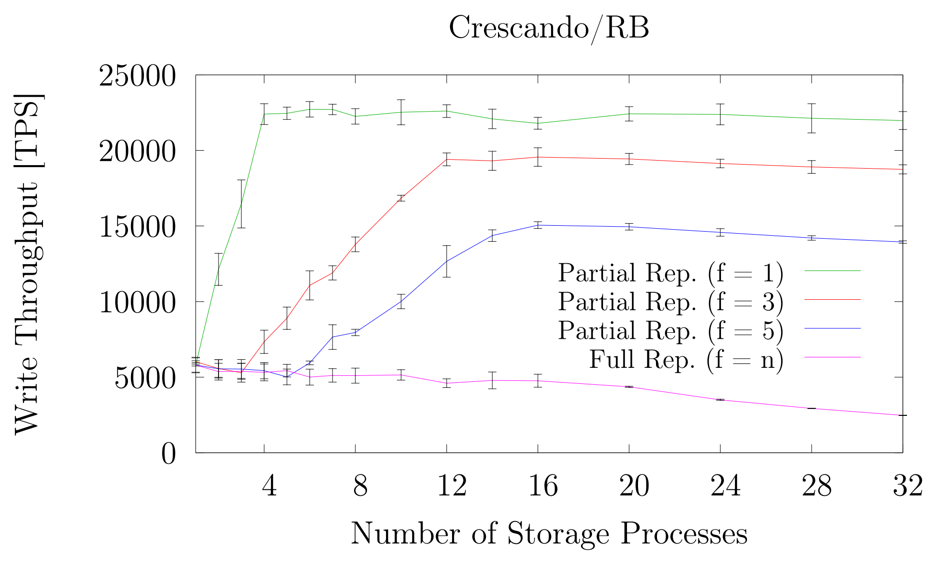
We varied the number of storage processes between 1 and 32, and repeated the experiment for replication factors $f = 1$, 3, 5, and n, where n was the number of storage processes. In the latter case, every storage process had to execute every single write, and E-Cast had to broadcast every corresponding write message.

Figure 5.8 shows the results. Both Crescando/RB and Memcached behaved quite similarly. A single Crescando/RB storage process could handle about 5,500 write transactions per second (TPS), while a single Memcached/RB storage process could handle about 7,000 TPS. Memcached could probably do better, but the open-source library[5] we used to access Memcached from Erlang was not very efficient.

For all cases where $n \leq f$, the workload was fully replicated across all storage processes, so write throughput was limited to 5,500 or 7,000 TPS respectively. But as the number of storage processes began to exceed f, Rubberband started to partition the workload over the storage process. Since every storage process contained a fixed amount of data, but had to handle a linearly decreasing amount of transactions, throughput increased linearly up to the point where the E-Cast protocol became a bottleneck.

In the case of $f = 3$, throughput reached about 20,000 TPS for 12 respectively 8 storage processes, while in the case of $f = 5$, throughput still reached about 15,000 TPS for 16 respectively 12 storage processes. Comparing this to the results for Rubberband on top of

[5]`https://github.com/joewilliams/merle`, retrieved January 26, 2012.



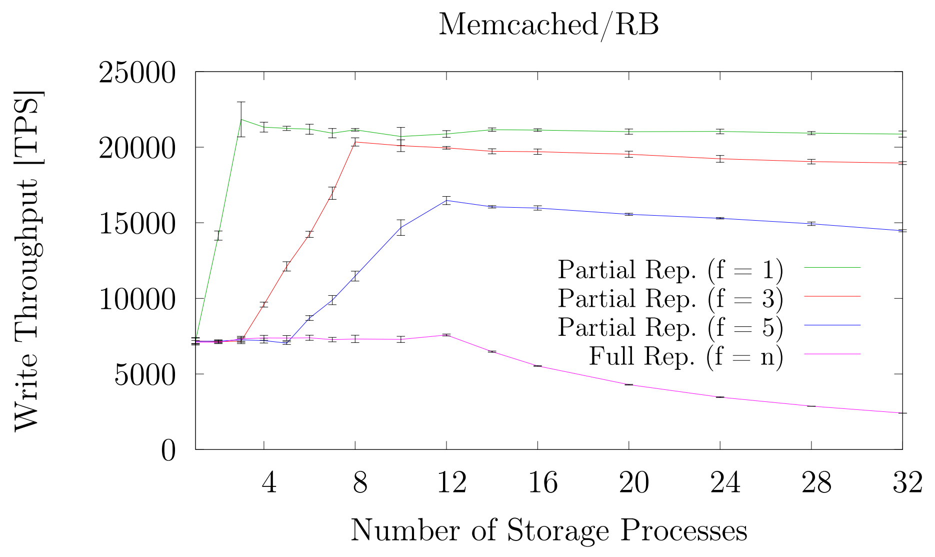


Figure 5.8: *Write Throughput: 3 Router, 1 Client, Vary Storage, Vary Rep. Factor f*

a dummy engine (null-store) presented in Section 4.6.3 on page 159, the throughput limit of Crescando/RB and Memcached/RB is 10% to 20% lower than for the dummy engine.

According to performance traces we collected, the additional overhead is due to 2 facts. First, the `Part Fun` called for message routing (cf. Section 5.3) is more expensive for Crescando/RB and Memcached/RB than for the null-store, because messages have a more complex structure. And second, storage engines do real work here, putting load on the machines. This load leads to more variance in response time, which increases the number of unstable messages maintained by the E-Cast routers, which puts extra load on the (compute-bound) routers and consequently decreases throughput.

The full-replication case is equivalent to the case where a write operation does not have a predicate on the partitioning attribute (every storage process must execute the write), so it shows the throughput of Crescando/RB and Memcached/RB for *global write transactions* under any replication factor. Global write throughput was 5,000 or 7,000 TPS respectively between 1 and 12 storage processes. Past 12 storage processes, E-Cast increasingly became a bottleneck. For 32 storage processes, global write throughput was still around 2,500 TPS. In realistic workloads (including the Amadeus Ticket use case), the vast majority of writes feature an equality predicate on a key attribute, so throughput seems more than sufficient.

Comparing the absolute numbers to other NoSQL data stores with strong consistency guarantees, the highest throughput number reported by Schütt et al. [SSR08] for Scalaris is 2,500 TPS for 16 storage processes. Vo et al. [VCO10] report that ecStore can handle 2,400 TPS with 72 storage processes. CloudTPS achieves 7,000 TPS with 40 storage processes on HBase, and 3,200 TPS with 20 storage processes on Amazon EC2 (HighCPU Medium instances) [WPC10]. Rao et al. [RST11] report up to 4,000 TPS for Spinnaker, using regular hard-disks for logging, and up to 20,000 TPS with the log disabled (no means of recovery). These numbers are all for mixed workloads (read and write) where most or all transactions consisted of one single-key read or write.

In terms of use cases, the Amadeus Ticket table currently sustains less than 1,000 writes per second, see Section 1.1. Schütt et al. [SSR08] state that Wikipedia handles about 2,000 TPS on its back-end database, and Baker et al. [BBC$^+$11] report that MegaStore handles on average 35,000 write transactions per second. The latter is the cumulative write load of 100 distinct applications and many more units of consistency, while Crescando/RB provides a *single* unit of consistency.

In summary, write throughput does not scale beyond about one dozen storage processes at this point (though we explain how to improve scalability in Section 4.5.3), but the numbers we report are already much higher than those of related systems. What is more, the throughput *per storage process* is at least one order of magnitude higher than that of related systems, owing to the fact that Crescando/RB is a completely *wait-free, push-based* system. Processes never stop and wait for messages when they have work to do. The sustained write throughput is sufficient for the Amadeus Ticket use case, and many (if not all) use cases mentioned in the related work.

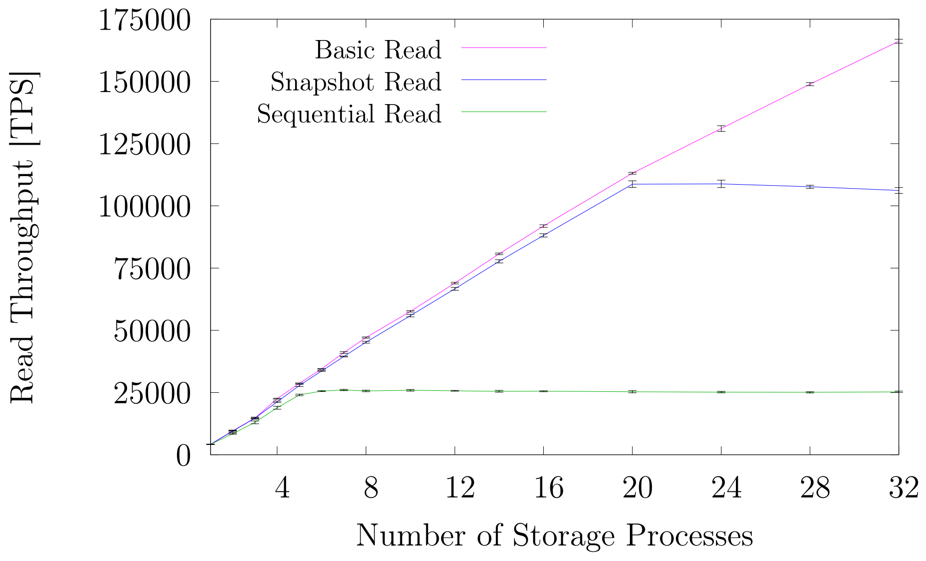


Figure 5.9: *Read Throughput: f = 3, 3 Router, 16 Client, Vary Storage, Vary Isolation*

5.6.4 Read Scalability

Crescando/RB supports three levels of read isolation—sequential read, snapshot read, and basic read—as described in Section 5.2. We repeated the previous scalability experiment for a replication factor $f = 3$, measuring read throughput (rather than write throughput) for each of the three isolation levels. The results are shown in Figure 5.9.

Again, each storage process was able to handle about 5,500 read transactions per second (TPS). Since Crescando/RB uses read-one-write-all (ROWA) replication, each read transaction was handled by exactly one storage process. Consequently, throughput increased linearly with the number of storage processes, up to the point where a protocol bottleneck was hit. In the case of sequential reads, the limit was 25,000 TPS, while the system could sustain up to 110,000 TPS of snapshot reads. Because client processes send basic reads directly to the storage processes, no protocol bottleneck was encountered in this case.

While applications can gain a lot of throughput from using snapshot reads rather than sequential reads, the relative difference (4x) is not as high as for Rubberband on top of the dummy engine (10x), see Section 4.6.7. As with writes, the difference in throughput (compared to Rubberband on top of the null-engine) is due to the more expensive `Part Fun` and the fact that the machines of storage processes are under high load here. The effects just happen to be greater under higher throughput; i.e., for snapshot reads as opposed to writes or sequential reads.

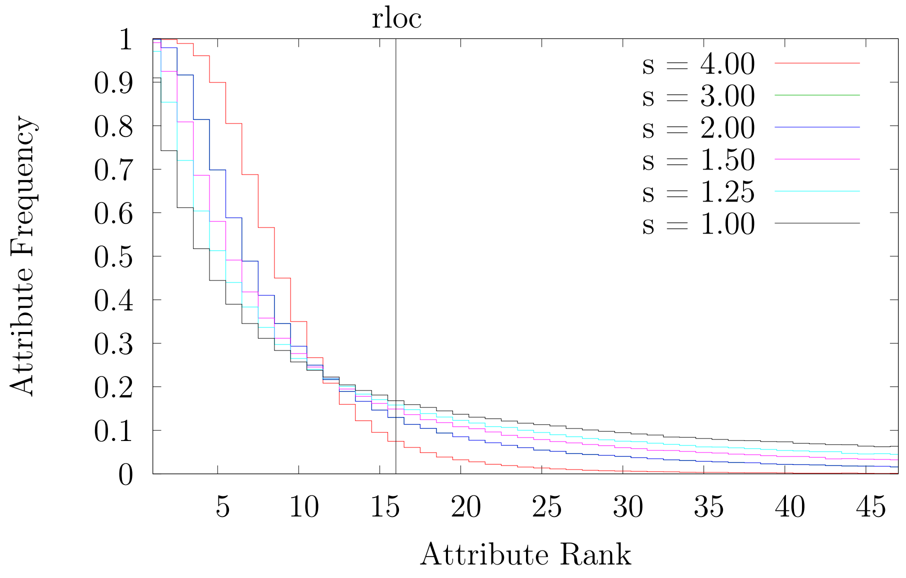


Figure 5.10: *Synthetic Predicate Attribute Distributions: N = 47, D = 9, Vary Workload Skew (s)*

5.6.5 Robustness to Query Diversity

In the Crescando/RB experiments presented so far, we have only used single-key reads and single-key writes (equality predicate on *rloc*). But the Crescando storage engine is designed to be robust to diverse, unpredictable workloads, not just single-key operations. To see to what degree the robustness to query diversity of Crescando carries over to Crescando/RB, we repeated the previous read scalability experiments, but using synthetic workloads with parametric predicate attribute skew, as originally presented in Section 2.10.2.

The skew parameter s was varied between 4.0 (which is similar to the actual Amadeus Ticket workload), and 1.0. Figure 5.10 shows the resulting predicate attribute frequency distributions. Note that the *rloc* attribute, by which we partition data across storage processes, is only ranked 16. Thus, between 80% and 92% of the select operations (depending on s) did not have a predicate on *rloc*, and consequently became *global read transactions*. That is, they each had to be processed by $\lceil n/f \rceil$ storage processes (cf. Section 5.3), where n is the total number of storage processes and f is the replication factor.

The results for all combinations of isolation level, workload skew, and numbers of storage processes are given by Figure 5.11. As in the previous scalability experiments, each storage process was running a single Crescando scan thread over 2 GB of Amadeus Ticket data. Crescando/RB performs read-one-write-all (ROWA) replication, so for $n \leq 3$ storage processes, this was a scale-up experiment (constant data set, increasing computing power).

And indeed, for each of the 3 isolation levels, one can see that throughput increases linearly up to $n = f = 3$ storage processes.

For $n > 3$, the data size increased linearly with the number of storage processes, turning this into a scale-out experiment (increasing data set, increasing computing power). Only the 8% to 20% of the workload with an equality predicate on *rloc* (cf. Figure 5.10) were processed by a single storage processes. The remaining 92% to 80% of the workload had to be processed by $\lceil n/f \rceil$ storage processes, making it impossible for throughput to increase much beyond 3 storage processes. This is not a shortcoming in the system design but an intrinsic property of the types of workloads we consider.

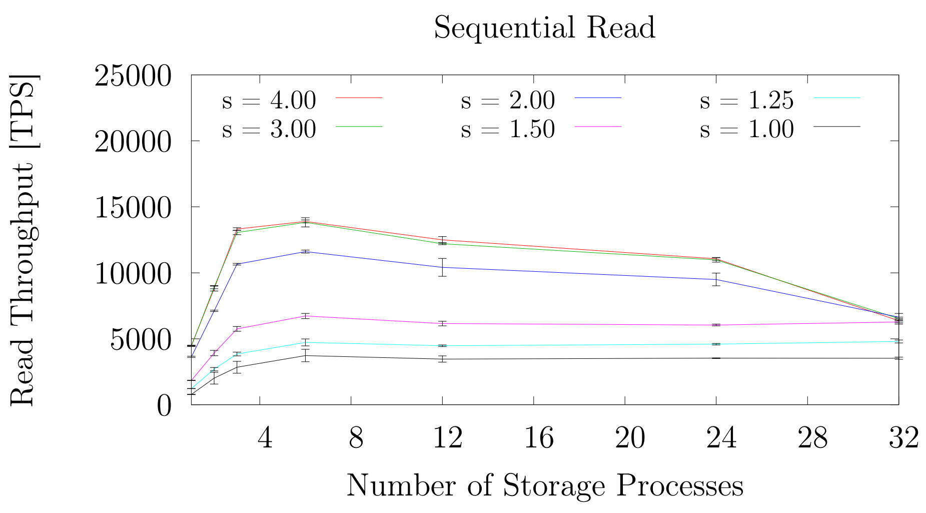
For sequential and snapshot reads where $s \geq 2$ and more than 24 storage processes, E-Cast became a throughput bottleneck. For 32 storage processes, E-Cast was able to route 6,500 sequential reads and 10,000 snapshot reads, respectively. Throughput for these combinations of workload and large system size could be improved by partitioning the responsibility for message forwarding across the router processes, as discussed in Section 3.5.7. Alternatively, users should consider basic reads, which face no bottleneck. Since many of the "odd" queries in real-world workloads can be expected to be analytic in nature, basic reads may often be good enough.

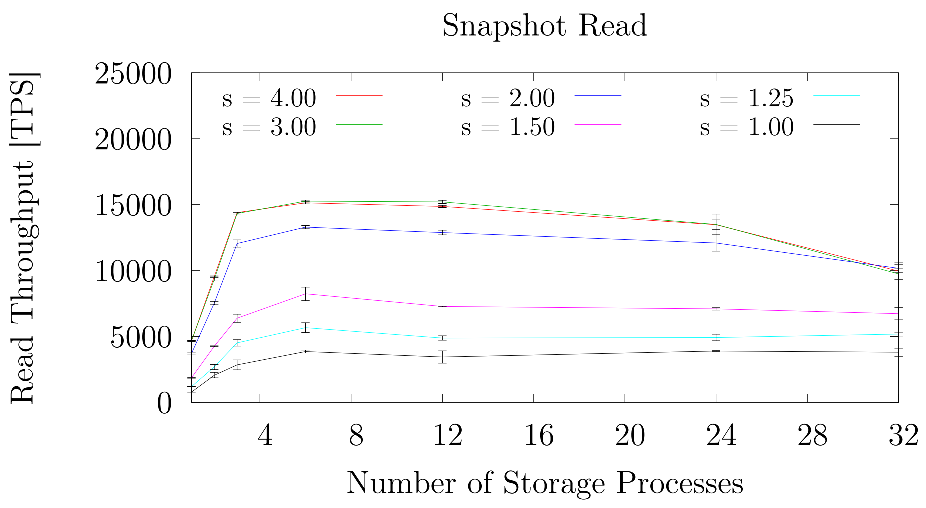
All that being said, 6,500 *global* sequential reads or 10,000 global snapshot reads over 32 storage processes is a very good result. 6,500 TPS is still higher than the throughput of most related work for *single-key* transactions at this system size, see Section 5.6.3. And because Crescando is a storage engine that scales linearly with the number of cores on a machine (cf. Section 2.10.4), it is sufficient to run just 1 storage process per machine. Thus, 32 storage processes can store hundreds of gigabytes of data in practice, which is easily enough for all our use cases.

In summary, Crescando/RB retains much of the robustness to unpredictable, evolving workloads of the Crescando engine, while scaling to dozens of storage processes (machines). Global reads with strong consistency guarantees are expensive for high numbers of storage processes, but these operations are likely to be rare in real workloads.

5.6.6 Elasticity

One of the main features of Crescando/RB is its ability to scale elastically; i.e., the ability to change the mapping of keyspace identifiers to storage processes (and thus the data placement) without loss of consistency or availability. To put this ability to the test, we loaded a single Crescando/RB storage process with 16 GB of data. We then added another storage process every 5 minutes, always expanding the keyspace, up to a total of 16 processes, each running on a separate machine. The replication factor was set to $f = 3$.





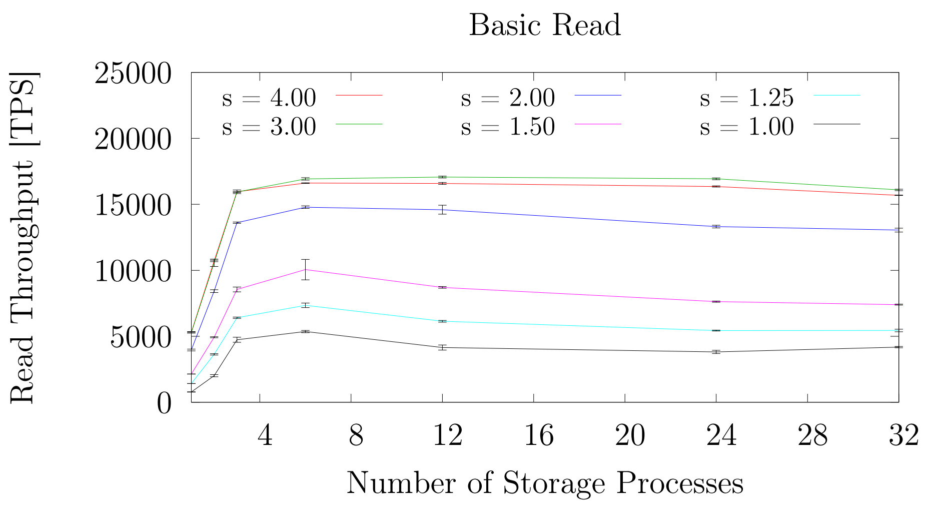


Figure 5.11: *Read Throughput: f = 3, 3 Router, 16 Client, Vary Storage, Vary Workload Skew (s), Vary Isolation Level*

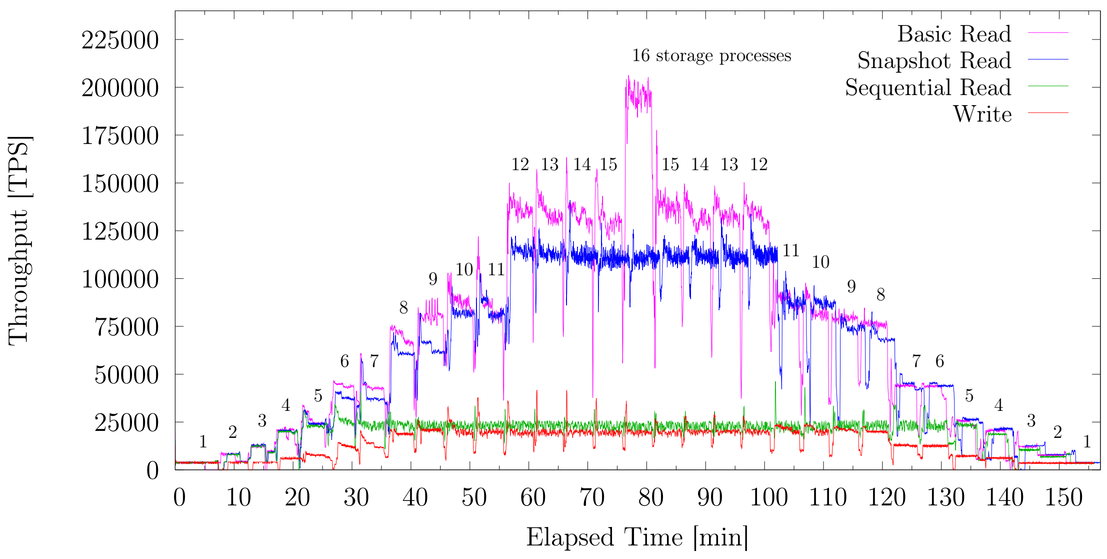


Figure 5.12: *Throughput: f = 3, 3 Router, 16 Client, Elastic Storage, Vary Workload Type*

(Thus, for 16 storage processes, each storage process was holding roughly 3 GB data.) We then removed a storage process every 5 minutes, always contracting the keyspace, until the system was scaled back to just 1 storage process.

Throughput

The experiment was repeated 4 times while measuring throughput at a 1 second sampling interval; one iteration using a pure write workload, and one more iteration for each read isolation level. We used 16 client processes spread over 4 machines to make sure that the system was saturated. Figure 5.12 shows the resulting throughput traces.

Throughput scaled elastically as storage processes were added/removed. For each type of workload or isolation level, the corresponding trace also shows the same throughput limits discussed in the previous experiments. But in contrast to the previous read and write scalability experiments, each Crescando instance was given all 8 cores of the respective machine to run its scan threads on, so the throughput for 1 storage process was similar to the previous experiments, despite the larger data volume (16 GB as opposed to 2 GB).

The step shape of the traces is simply an artifact of (the lack of) load balancing. To keep the experiment simple, we had chosen not to rebalance the keyspace after each process addition or removal. Consequently, throughput was limited by the process with the largest data partition. Only in case of 1, 2, 4, 8, and 16 processes was the load perfectly balanced.

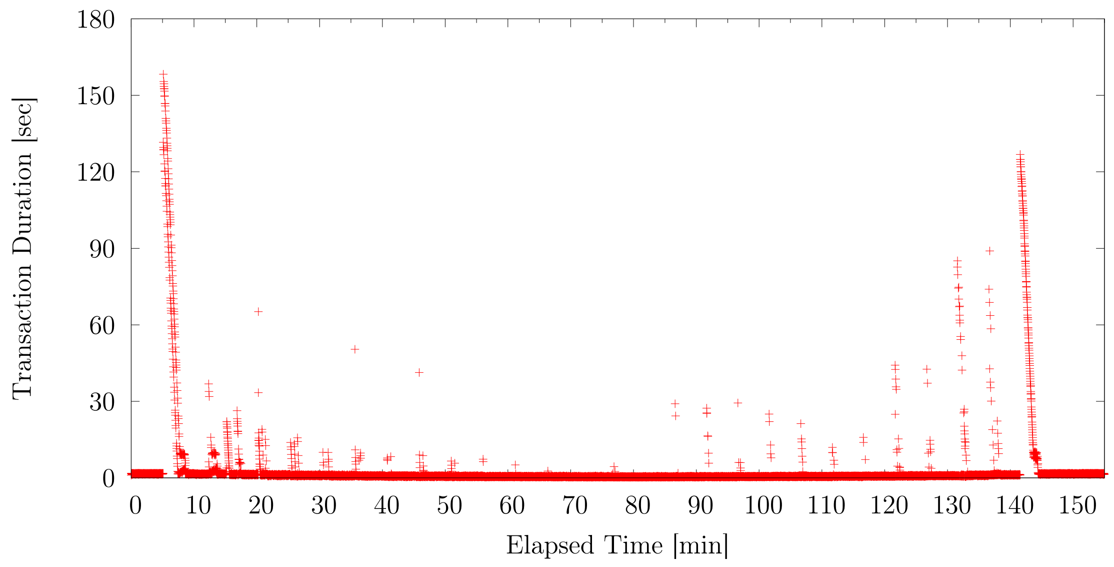
In all 4 traces, one can see a brief dip in throughput every 5 minutes, at the point where the keyspace was expanded or contracted. The longest dips appear where the system was scaled from 1 to 2 storage processes. At this point, the first storage process had to copy all 16 GB of data to the second storage processes. Since the machines were connected through a 1 Gbit interface, copying 16 GB required roughly 150 seconds, during which incoming reads and writes were queued or had to share scan cursors with the reconfiguration operations (cf. Ashmawy [Ash11] for details). As more processes were added, less data had to be shuffled each time, and the dips in throughput became less pronounced.

Latency

We repeated the experiment a few more times, measuring the latency (transaction duration) for a fixed background workload 2,000 write TPS. Figures 5.13 and 5.14 show latency scatter-plots for snapshot reads and writes respectively (sampling interval 1 second). In each figure, the upper chart shows the range between 0 and 180 seconds of latency, while the lower chart zooms in on the range between 0 and 4 seconds of latency. The data underlying each pair of charts is identical. We performed the experiment also for sequential reads and basic reads, but the results did not differ from those of snapshot reads. We therefore omit those charts.

The results show that the vast majority of transactions complete within less than 2 seconds for 1 storage process, down to less than 0.75 seconds for 16 storage processes (due to less load per process). Only during keyspace expansion and contraction were transactions running for longer periods of time, because some of the storage processes were busy with partial live migration (cf. Section 5.4). The height of the latency spikes in the write latency chart shows approximately the time it took to shuffle data between storage processes. As may be expected, the highest latency spike is 160 seconds when the system is scaled from 1 to 2 storage processes, since then all 16 GB of data had to be copied, which required approximately 160 seconds over the 1 Gbit network link. In the case of snapshot reads (or any other read), operations were very rarely delayed when there were already more than 2 storage processes in the keyspace. The reason is that there was always one replica that was not busy ($f = 3$), and Rubberband automatically forwards reads to non-busy replicas whenever possible, see Section 4.4.1.

In this experiment, we considered a write transaction complete only after every affected storage process had executed the write. Consequently, a significant fraction of the workload was delayed during each reconfiguration, giving the impression of system unavailability. But as explained in Section 4.4.1, it is quite irrelevant when the last storage process executes a given write. At the point where E-Cast acknowledges that a write message is stable (which requires just a few milliseconds), E-Cast and Rubberband *guarantee that*



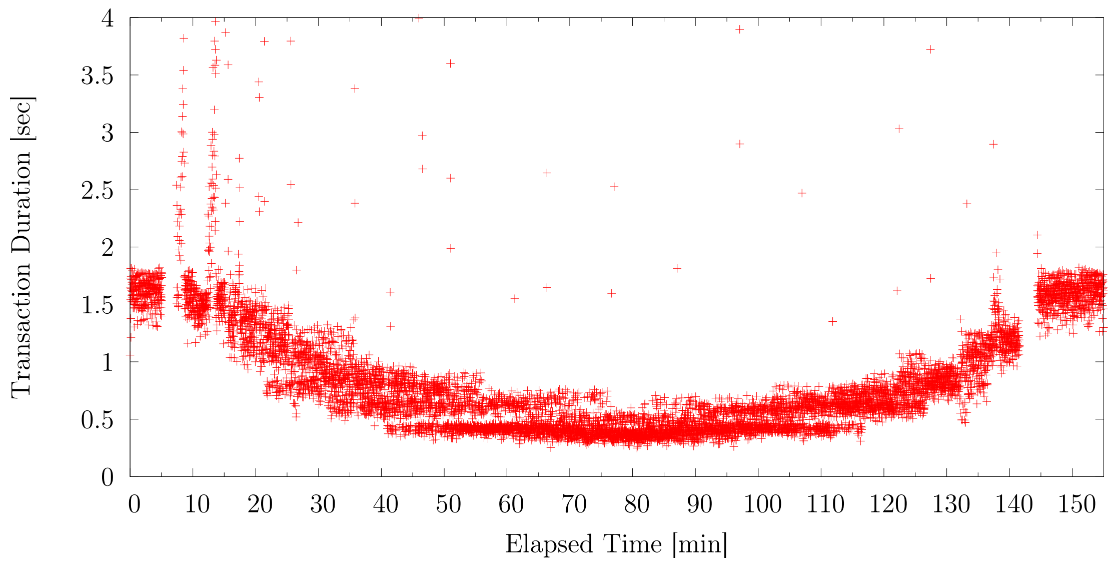
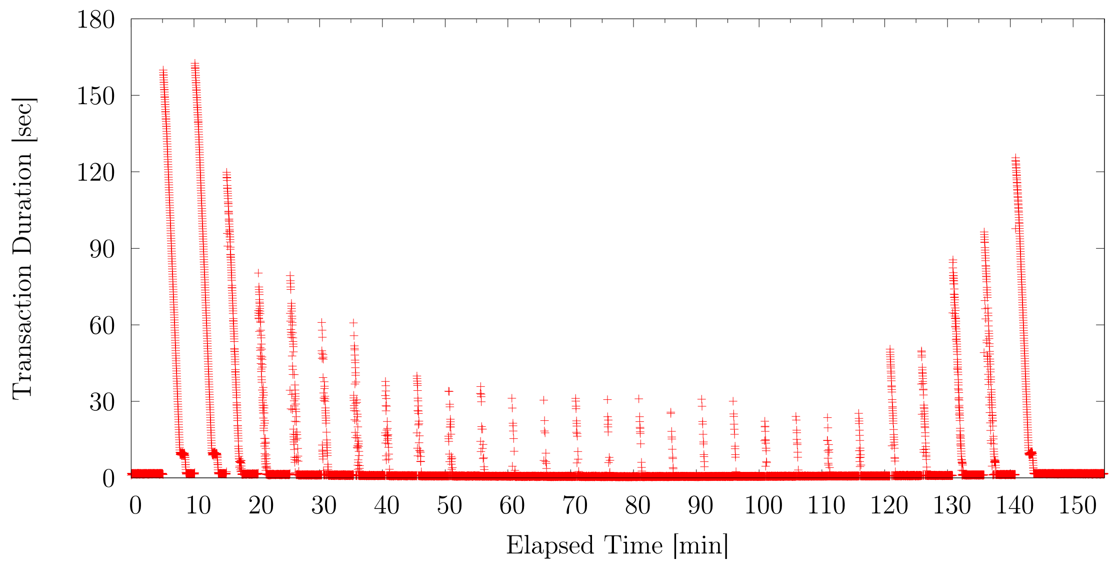


Figure 5.13: *Snapshot Read Latency: f = 3, 2,000 Background Write TPS, 3 Router, 1 Background Client, 1 Latency Client, Elastic Storage*

every write is eventually, uniformly executed. What is more, the effects of a given write can be seen much sooner than when the last storage process executes the write. It is sufficient that the *non-busy* storage processes execute the write, since Rubberband forwards subsequent reads to these non-busy processes. It follows that Crescando/RB (and any other system built on top of Rubberband) is able to remain *fully available* during reconfiguration.



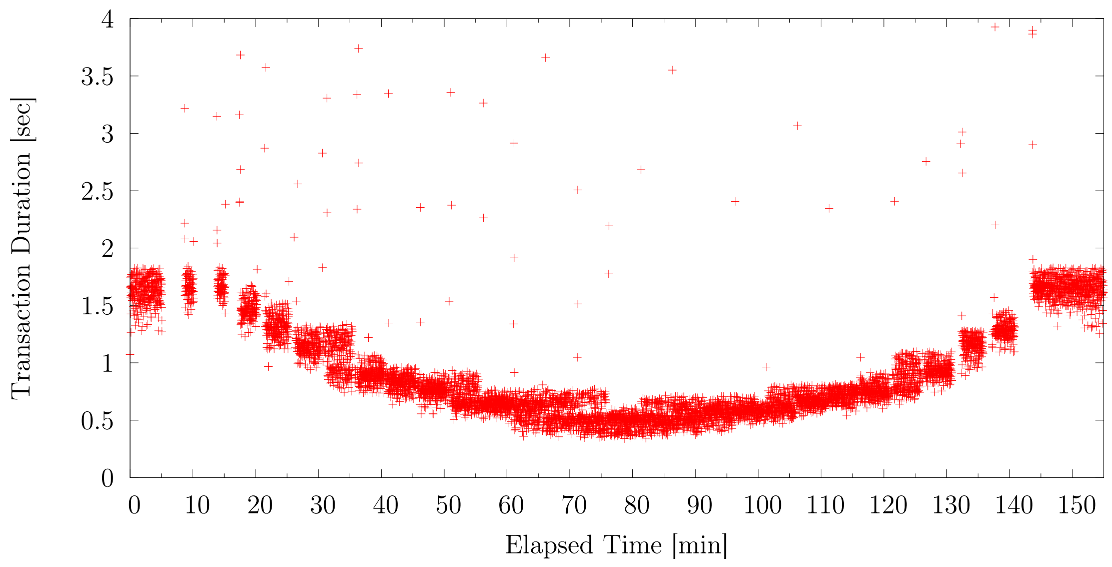


Figure 5.14: *Write Latency: $f = 3$, 2,000 Background Write TPS, 3 Router, 1 Background Client, 1 Latency Client, Elastic Storage*

In summary, Crescando engines shuffle data at a rate that fully saturates a 1 Gbit network link during partial live migration. Crescando/RB remains available during this period, assuming a sufficiently high replication factor. And after each reconfiguration, the system quickly stabilizes again, both in terms of throughput and in terms of latency.

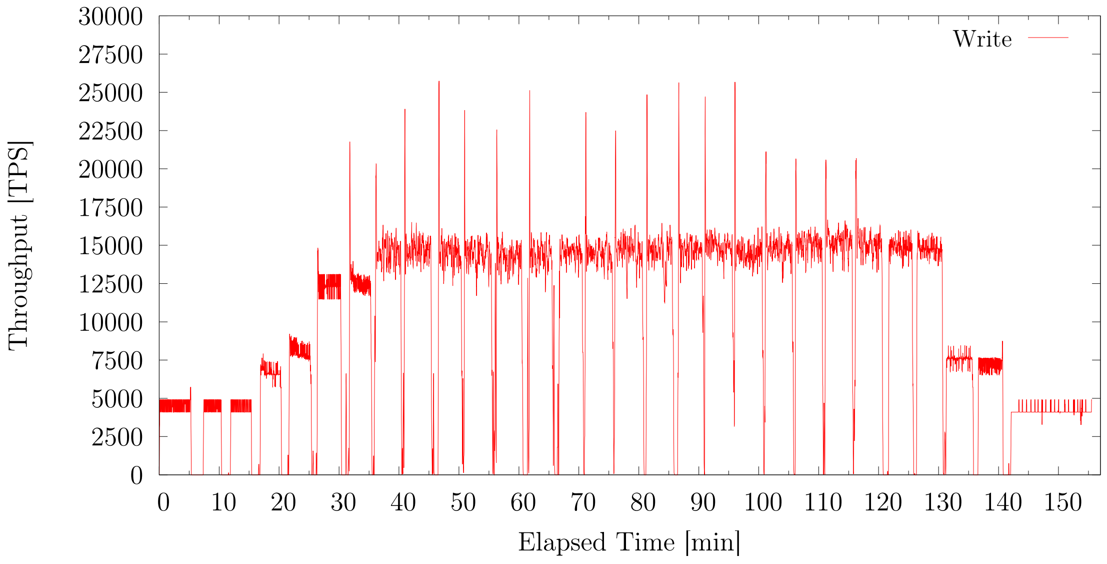


Figure 5.15: *Write Throughput on EC2: $f = 3$, 3 Router, 16 Client, Elastic Storage*

Elasticity on Amazon EC2

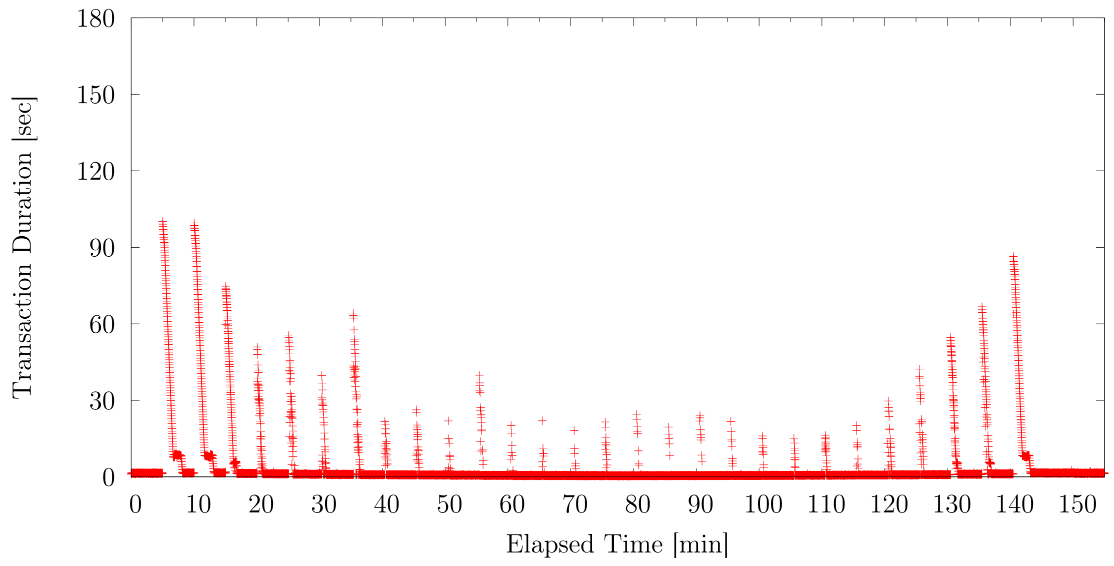
To demonstrate that Crescando/RB can be deployed in a public cloud, we repeated some of the previous elasticity experiments on Amazon EC2 (Elastic Compute Cloud)[6]. EC2 offers pay-as-you-go computational resources, by booking virtual machine instances in one of Amazon's data centers located around the globe.

We booked 20 instances of type *Cluster Compute Quadruple Extra Large* (`cc1.4xlarge`). Each of these instances had 23 GB of memory, 2 Intel Xeon X5570 2.93 GHz, quad-core CPUs with hyper-threading, and a 10 Gbit Ethernet interface. 1 EC2 instance was running the client processes. 3 EC2 instances were each running one router process. And the remaining 16 EC2 instances were each running one storage process. For budgetary reasons, we only repeated the write throughput and latency measurements.

The resulting charts are given in Figures 5.15 (Throughput) and 5.16 (Latency) respectively. The throughput trace does not differ much from that produced by our lab machines (Figure 5.12). Despite the higher clock rate of the EC2 machines (2.93 GHz as opposed to 2.26 GHz), throughput per storage process was identical to our lab machines, at slightly over 4,000 TPS. This can most likely be attributed to the overhead of virtualization of the EC2 instances.

Peak throughput on EC2 was lower however, at 15,000 TPS as opposed to 20,000 TPS. Again, some of that may be due to virtualization, but part of it can also be attributed to

[6]`http://aws.amazon.com/ec2`, retrieved January 29, 2012.



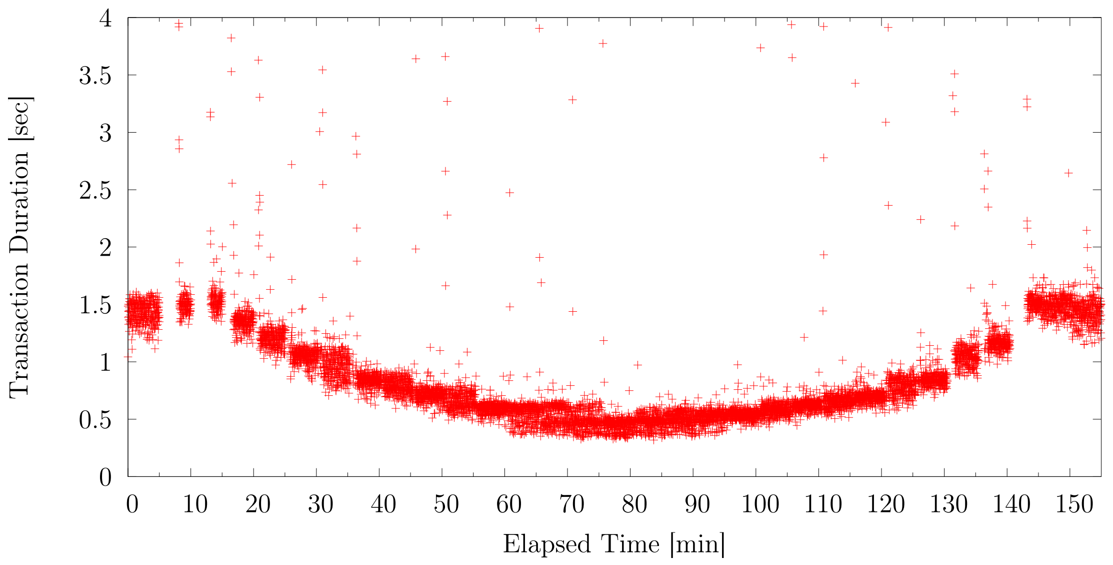


Figure 5.16: *Write Latency on EC2: f = 3, 2,000 Background Write TPS, 3 Router, 1 Background Client, 1 Latency Client, Elastic Storage*

the increased network latency and variance in network latency, which resulted in a larger number of unstable messages in E-Cast, which in turn put more load on the (compute-bound) router processes. All in all however, E-Cast appears to work well in a public cloud.

The latency scatter-plot of EC2 (Figure 5.16) is also very similar to that produced by our private lab machines (Figure 5.14). Yet, two differences meet the eye on closer inspection.

First, the EC2 plot does show some more variance in transaction duration during the stable periods between system reconfigurations. The variance is most visible at the end of the experiment, around minute 150. Relative to the total latency, which is dominated by the Crescando storage engine, the variance is negligible though.

Second, the EC2 instances had 10 Gbit network interfaces. This lowered the duration of partial live migration, as evident from the lower latency spikes at reconfiguration events. According to the plot, copying all 16 GB of data to a newly launched storage process required slightly over 100 seconds, which means Crescando engines were shuffling data at a rate of around 160 MB/sec.

In summary, Crescando/RB works well in a public cloud. Increased network latency and variance in this environment does have a measurable effect on performance, but not seriously so. Partial live migration profits from a 10 Gbit network, but there is room for improvement. The Master's thesis by Ashmawy [Ash11] presents some ideas for future work in this direction.

5.6.7 Fault Tolerance

In the final set of experiments, we investigated how Crescando/RB behaves in case of storage process failures. To this end, we started 6 storage processes and bulk-loaded them with 12 GB of data in total. As usual, the replication factor was set to $f = 3$, so each storage process was holding about 6 GB of data.

Throughput

In the first experiment, we measured how read throughput (snapshot reads) changed under single-process and double-process failures. The trace in Figure 5.17 shows that the system originally sustains just over 40,000 TPS. After 10 minutes, one of the storage processes was killed. Some partition of the keyspace was now covered by just 2 rather than 3 replicas. Consequently, the load was not evenly balanced anymore and throughput dropped to just under 30,000 TPS.

15 minutes into the experiment, the failed storage process was replaced. While the replacement process received data from its neighbors, throughput dropped some more, just as in the case of keyspace expansion in the previous elasticity experiments. However, the system remained available, and after just 1 minute (required to copy 6 GB of data) throughput jumped back to the original 40,000 TPS.

Note that the throughput was temporarily a bit higher after reconfigurations. This is because after a reconfiguration, a number of pending operations had already queued up at

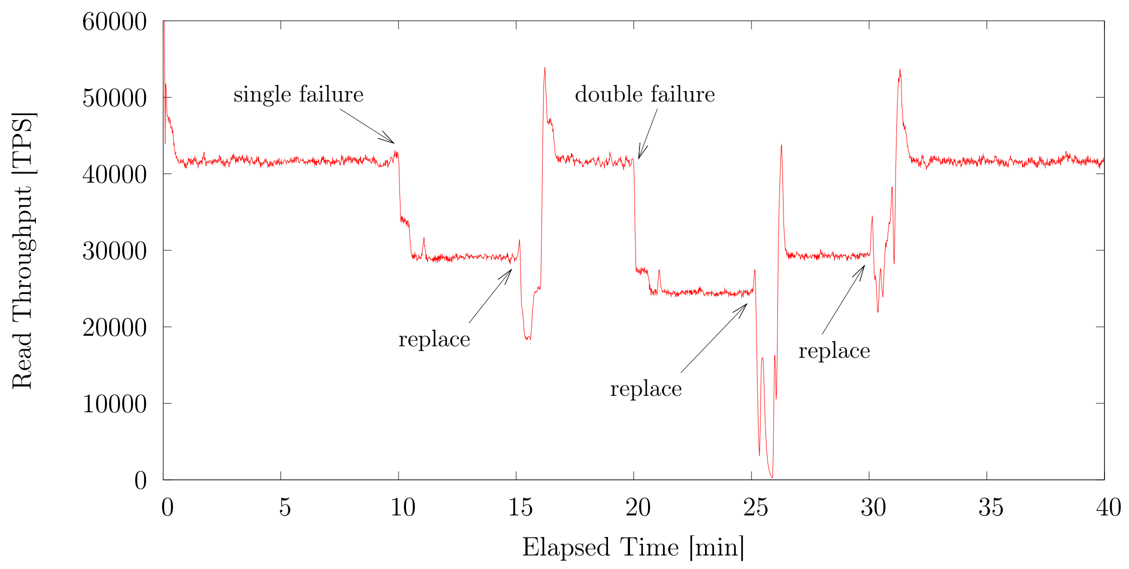


Figure 5.17: *Snapshot Read Throughput: f = 3, 3 Router, 4 Client, 6 Storage, Fail Storage*

the previously busy processes. The storage engines of these processes could then operate at maximum efficiency, allowing the storage processes to catch up with the rest of the system, and throughput temporarily increased as the back-log of pending operations was worked off.

20 minutes into the experiment, 2 storage processes were killed at the same time. Throughput dropped to about 25,000 TPS. Perhaps coming as a surprise, the throughput was not much lower than in the case of a single process failure (30,000 TPS). The reason is that throughput is determined by the slowest (or most-loaded) storage process, and Rubberband was able to balance the load quite well over the remaining processes.

After 25 minutes, one of the failed storage processes was replaced. At that point, all storage processes of one of the replication groups had either failed or were busy with partial live migration. As a result, throughput briefly dropped to zero. After 1 minute, one of the failed storage processed had been replaced, and throughput increased back to 30,000 TPS. Finally, after 30 minutes, the second failed storage processes was replaced, and throughput returned to the original 40,000 TPS.

Latency

The experiment was repeated for a fixed background workload of 8,000 Write TPS, while snapshot read latency was being measured at a 1 second sampling interval. The resulting

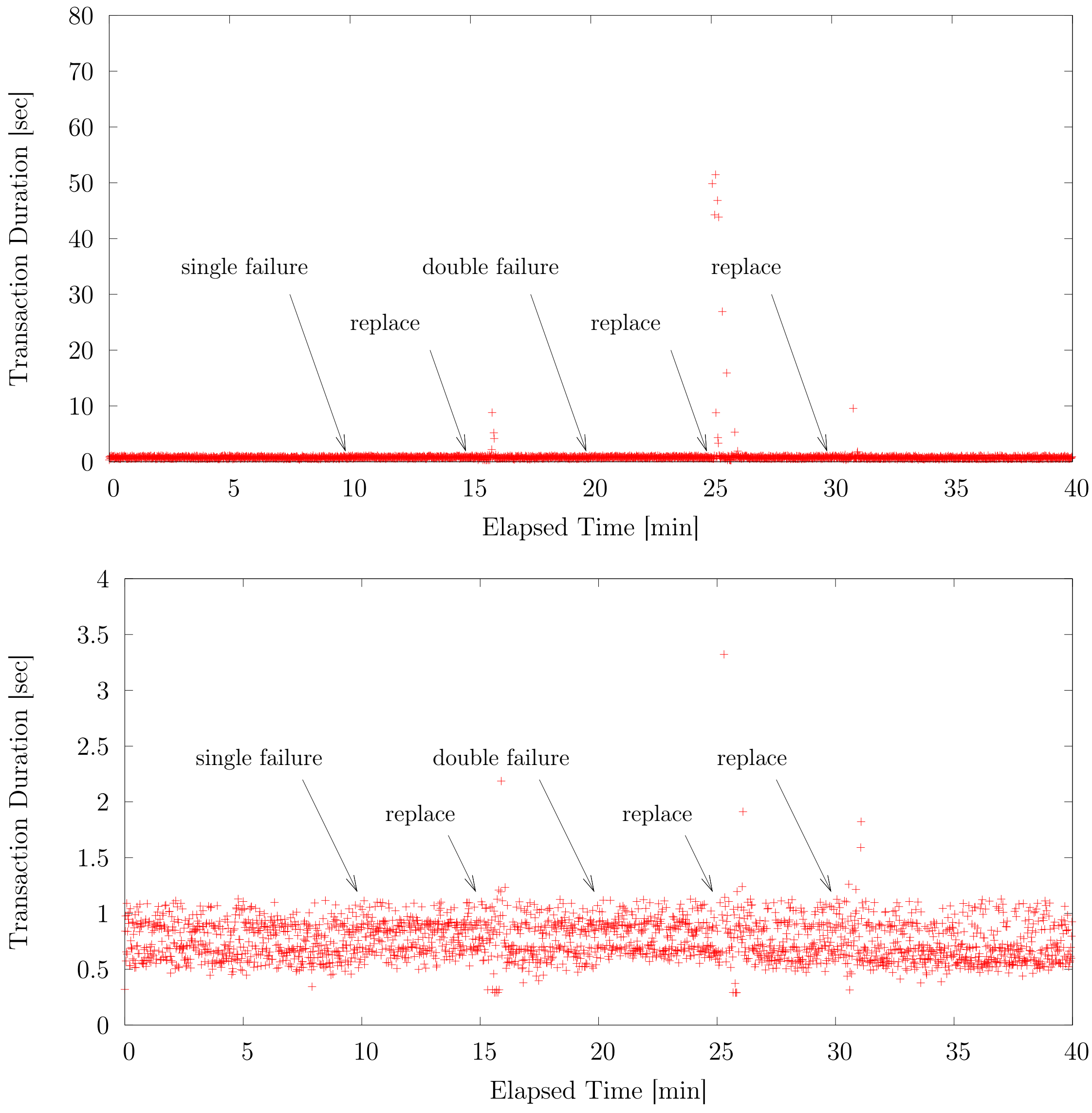


Figure 5.18: *Snapshot Read Latency: f = 3, 8,000 Background Write TPS, 3 Router, 4 Background Client, 1 Latency Client, 6 Storage, Fail Storage*

scatter-plots are shown in Figure 5.18. As before, both charts show the same data. The lower chart merely zooms in to the range between 0 and 4 seconds of latency.

The vast majority of reads was completed within just over 1 second. Process failures had no visible effect on latency. The reason is that the write load per process is not affected by process failures, as long as the keyspace is not also contracted.

The larger latency spike at minutes 25 to 26 shows that part of the keyspace was not available while data was being shuffled. The smaller latency spikes at minutes 16 and 31 are caused by the fact that the newly joined storage process was not immediately able to process operations when it declared itself ready (cf. Section 4.4.1), but rather had to refresh its optimizer statistics first (cf. Section 2.7.1). This is in fact a small performance bug that could be fixed by declaring storage processes ready only after they have refreshed all their statistics.

In summary, system throughput degrades gracefully when storage processes fail. The system remains available, even during process replacement, assuming a sufficient replication factor. After process replacement, the system quickly stabilizes again, both in terms of throughput, and in terms of latency.

5.7 Concluding Remarks

In this chapter, we have combined the Crescando storage engine and the Rubberband distribution framework into a push-based, elastic, fault-tolerant relational table implementation with sequential consistency guarantees: Crescando/RB. The Crescando storage engine has been implemented in roughly 54,000 lines of C++ (and some Assembly). E-Cast and Rubberband have been implemented in roughly 19,000 lines of Erlang. Composing these components into a single, usable system—Crescando/RB—required another 8,500 lines of C++ and 4000 lines of Erlang code. The resulting system combines the robustness and high throughput of the Crescando storage engine with the elasticity and fault tolerance of Rubberband.

In contrast to related work, Crescando/RB separates the reliable, ordered delivery of messages (by E-Cast), from their eventual, deterministic execution (by Crescando). This separation enables (very high) throughput independent of the network latency, and, most importantly, allows the system to remain fully available during elastic reconfiguration.

Chapter 6

Conclusions

To fully leverage the benefits of cloud computing, systems must be elastic and tolerate permanent node failures. Traditional solutions for shared storage—relational database management systems—have been struggling to meet these new requirements. Consequently, the recent years have seen a proliferation of specialized NoSQL data stores for the Cloud.

Motivated by a large-scale industry use case, the Amadeus airline ticket reservation system, we have presented such an elastic, fault-tolerant data store called *Crescando/RB*. The system consists of three main components: Crescando, Rubberband, and E-Cast.

Crescando is a push-based, relational storage engine that uses continuous, parallel, and massively shared scans rather than data indexes. As we have shown, this radical design makes Crescando significantly more robust to unpredictable, evolving workloads, and allows the engine to scale linearly on modern multi-core hardware. An important insight in this regard is that deep sharing of resources rather than careful isolation of concurrent requests realizes scalability and predictable performance.

Crescando is of practical value by itself, but as main-memory engine it is limited to the capabilities of a single machine. In order to scale elastically to large data sets, and in order to achieve higher availability than any single machine can provide, Crescando/RB partitions and replicates the contents of independent Crescando instances (shared-nothing architecture) using a novel distribution framework called Rubberband. Rubberband is intimately based on E-Cast, a dynamic, uniform, causal total order multicast protocol for asynchronous networks with unreliable failure detectors.

By taking advantage of E-Cast, Rubberband can guarantee sequential consistency for arbitrarily overlapping multi-key and (multi-)range operations, and tolerate permanent failure of any process at any time. Unlike most related systems with strong consistency guarantees, Rubberband does not rely on stable storage or perfect failure detectors, so it is easy to deploy in the Cloud. Furthermore, Rubberband decouples the reliable, ordered delivery of operations (reads and writes, but also reconfiguration requests) from their eventual, deterministic execution. As a result, data stores built with Rubberband can achieve very high throughput and also remain fully available during reconfiguration.

Crescando/RB combines the features and advantages of Crescando, Rubberband, and E-Cast into a single, comprehensive system. The many processes that are involved in query processing in Crescando/RB form a deep, parallel pipeline, taking the idea of push-based query processing and resource sharing to the extreme. Our experimental results show that this design makes Crescando/RB highly elastic and available, and allows it to easily meet the performance requirements of the Amadeus use case. Indeed, the throughput numbers presented in this dissertation are significantly higher than those reported for related systems with comparable consistency guarantees.

In describing the design and implementation of Crescando/RB, this dissertation contributes to our scientific understanding of how to make storage systems robust to unpredictable, evolving workloads, how to take full advantage of the increasing number of CPU cores on modern hardware platforms, and how to elastically scale to a large, dynamic set of machines without loss of data consistency or availability.

The technical contributions of the Crescando storage engine include a cooperative (shared) scan algorithm called Clock Scan, a query-data join algorithm called Index Union Join, and a variety of novel index structures that have been optimized for modern hardware. Among the technical contributions of E-Cast is a formalization of dynamic partial replication as a stateful routing problem (which E-Cast solves), the E-Cast protocol itself, and a high-performance, reconfigurable, replicated state machine implementation with pluggable behavior. Finally, Rubberband and Crescando/RB show how a strongly consistent system can be reconfigured elastically, without loss of consistency or availability. In this context, we have also discovered and solved an interesting new optimization problem with respect to fault tolerance during partial live migration.

The implementation of Crescando/RB consists of roughly 62,500 lines of C++, and 23,000 lines of Erlang. These numbers do not include a wide range of tools, unit tests, and benchmark drivers written in various languages. Because Crescando/RB is a complete, practical system, it touches many deep technical issues, from main-memory index structures to protocols for solving distributed consensus. For some of these topics, we feel that we have barely touched the surface. For example, multi-query optimization in Crescando may benefit from a more general, formal model of query-data joins. E-Cast can be generalized, optimized, recombined, and applied to many other use cases in systems management and coordination. And we have presented a possible extension to Rubberband which would allow the throughput for single-key reads and writes to increase linearly with the number of storage processes in the system.

We hope that, in the future, others will continue the research where we had to move on in the interest of building a complete system, and we hope that Crescando/RB will inspire many more exciting systems and services in the Cloud.

Appendix A

Amadeus Ticket Schema

```
CREATE TABLE Itinerary (
      provider                 VARCHAR(3),
      productId                UINT(16),
      alphaSuffix              CHAR,
      dateInFirstLeg           DATE,
      dateIn                   DATE,
      dateOut                  DATE,
      cityFrom                 VARCHAR(3),
      cityTo                   VARCHAR(3),
      cancelEnvelope           INT(32),
      cancelInitiator          VARCHAR(3),
      rloc                     VARCHAR(6),
      paxTattoo                INT(32),
      segmentTattoo            INT(32),
      purgeDate                DATE,
      office                   VARCHAR(9),
      creationDate             DATE,
      modificationDate         DATE,
      nip                      INT(16),
      unassigned               INT(16),
      pnrQualifier             UINT(64),
      paxQualifier             UINT(64),
      sgtQualifier             UINT(64),
      name                     VARCHAR(57),
      firstname                VARCHAR(56),
      sex                      BOOLEAN,
      cabin                    CHAR,
      classOfService           CHAR,
      bookingStatus            VARCHAR(2),
      codeShareType            VARCHAR(2),
      bookingDate              DATE,
      subclass                 UINT(8),
      posCrs                   VARCHAR(3),
      posCountry               VARCHAR(2),
      cancelFlag               CHAR,
      mariage                  VARCHAR(3),
      yieldValue               INT(32),
      rvIndicator              CHAR,
      rvValue                  INT(32),
      cnxNumber                INT(8),
      did                      VARCHAR(16),
      iid                      VARCHAR(16),
      indexingVersion          INT(16),
      sgtVendorFormat          VARCHAR(2),
      sgtVendorValues          VARCHAR(15),
      inboundCnxTime           INT(32),
      inboundSgtTattoo         INT(32),
      outboundCnxTime          INT(32),
      outboundSgtTattoo        INT(32)
);
```

Appendix B

Proofs

B.1 Crescando

Lemma 1. *For any input sequence of operations, if all scan threads run either Classic Scan or Elevator Scan, then Monotonic Reads, Monotonic Writes, Read Your Writes, and Write Follows Strict Read hold.*

Proof. Classic Scan and Elevator Scan execute operations strictly in input order against any particular record slot. Records do not move between slots. Thus, for any pair of operations o, o' where o precedes o' in the input sequence, for any record x, o is executed against x before o'.

The four consistency guarantees each require that some projection of the input sequence is executed in input order with respect to any record x. Since the *entire* input sequence is executed in input order, all the guarantees hold. □

Theorem 3. *For any input sequence of operations, Monotonic Reads are guaranteed.*

Proof. If all scan threads run either Classic Scan or Elevator Scan, Monotonic Reads follow immediately from Lemma 1. What remains to be shown is that Monotonic Reads hold also for Clock Scan.

Without restriction of generality, let r, r' be a pair of read operations in the input sequence, where r precedes r', and both operations read some record x. Two cases must be considered. If r is a strict read operation (Case 1), or both r and r' are basic read operations (Case 2), than it must be shown that r' reads the same or a later version of x than r. (If no such pair of operations exists, Monotonic Reads hold trivially.)

Clock Scan activates operations in input order, so r precedes r' in activation order. If r is executed in an earlier scan cycle than r', then Monotonic Reads hold immediately. So assume that r and r' are activated in the same scan cycle.

Case 1: r is a strict read operation, so it is executed as part of the update-data join plan. If r' is also a strict read operation, than it will be executed after r, because both BNLJ and Index Union Update Join (IUUJ) execute operations in activation order. If r' is a basic read operation, than it will be executed as part of the query-data join plan. Because Clock Scan executes first the update-data join plan, and then the query-data join plan for each scanned segment, r is executed before r'. In both sub-cases, r is executed before r'. Once inserted, records do not move around in memory, so clearly, for any record x read by r and r', r' sees the same or a later version than r.

Case 2: Both r and r' are basic read operations, so they are both executed as part of the same query-data join plan. Reads obviously do not update the records, so both r and r' see the same version of any record x.

In both cases, r' sees the same or a later version of any record x than r. □

Theorem 4. *For any input sequence of operations, Monotonic Writes are guaranteed.*

Proof. If all scan threads run either Classic Scan or Elevator Scan, Monotonic Writes follow immediately from Lemma 1. What remains to be shown is that Monotonic Writes hold also for Clock Scan.

Without restriction of generality, let w, w' be a pair of write operations in the input sequence, where w precedes w', and both operations write some record x. (If no such pair of operations exists, Monotonic Writes hold trivially.) Clock Scan activates operations in input order. If w is executed in an earlier scan cycle than w', then Monotonic Writes hold immediately. So assume that w and w' are activated in the same scan cycle. Both BNLJ and IUUJ execute write operations in activation order. It follows that w is executed before w' against x. □

Theorem 5. *For any input sequence of operations, Read Your Writes is guaranteed.*

Proof. If all scan threads run either Classic Scan or Elevator Scan, Read Your Writes follows immediately from Lemma 1. What remains to be shown is that Read Your Writes holds also for Clock Scan.

Without restriction of generality, let w, r be a pair of write and read operations in the input sequence, where w precedes r, and both operations access some record x. (If no such pair of operations exists, Read Your Writes holds trivially.) Clock Scan activates operations in input order. If w is executed in an earlier scan cycle than r, then Read Your Writes holds immediately. So assume that w and r are activated in the same scan cycle. r is either a strict read (Case 1), or a basic read (Case 2).

Case 1: r is a strict read, so both w and r are executed as part of the same update-data join plan. Index Union Update Join executes operations in activation order. It follows that w is executed before r against x.

Case 2: r is a basic read, so w is executed as part of the update-data join plan, and r is executed as part of the query-data join plan. Because Clock Scan executes first the update-data join plan, and then the query-data join plan for each scanned segment, w is executed before r against x. □

Theorem 6. *For any input sequence of operations, Write Follows Strict Read is guaranteed.*

Proof. If all scan threads run either Classic Scan or Elevator Scan, Write Follows Strict Read follows immediately from Lemma 1. What remains to be shown is that Write Follows Strict Read holds also for Clock Scan.

Without restriction of generality, let r, w be a pair of strict read and write operations in the input sequence, where r precedes w, and both operations access some record x. (If no such pair of operations exists, Write Follows Strict Read holds trivially.) Clock Scan activates operations in input order. If r is executed in an earlier scan cycle than w, then Write Follows Strict Read holds immediately. So assume that r and w are activated in the same scan cycle. r is a strict read, so both r and w are executed as part of the same update-data join plan. Both BNLJ and IUUJ execute operations in activation order. It follows that r is executed before w against x. □

Theorem 7. *For any input sequence of operations, Consistent Snapshots are guaranteed.*

Proof. From Monotonic Writes, Read Your Writes, and Write Follows Strict Read it follows that for any record x, for any pair of operations r, w, where r is a strict read, and w is the last write operation that writes x which precedes r in the input sequence, r sees the version written by w. It follows immediately that strict reads see consistent snapshots of the entire table (any set of records).

Parallel instances of Clock Scan independently reorder basic reads with respect to writes during activation. Thus, basic reads do not see a consistent snapshot of the entire table. However, Clock Scan always executes its update-data join plan before its query-data join plan. Records do not move between record slots. Thus, for any record x, all the basic read operations in the query-data join plan see the state of x after all writes in the update-data join plan have been applied. □

B.2 E-Cast

B.2.1 Correctness

Theorem 8. *E-Cast is a correct solution of Stateful Routing.*

Router processes propose and learn messages via uniform atomic broadcast with FIFO delivery. It is easy to see that, by definition of uniform atomic broadcast [DSU04], all router processes maintain agreement on an ever-growing m-seq Q.

To prove that E-Cast solves Stateful Routing, we need to prove that Correct Routing holds. That is, we prove that the m-seq Q that the routers agree on is valid, minimal, complete, gap-free, submit-ordered, delivery-ordered, and terminated.

Lemma 2. *Q is valid.*

Proof. To appear in Q, a message m must be learned by some router r. To be learned by r, m must previously be proposed by some router r'. For m to be proposed by r', r' must previously receive a message $\langle$"route"$, m\rangle$. Assuming no Byzantine failures, an application process a must previously send the message $\langle$"route"$, m\rangle$ to r', which in turn implies that m must previously be submitted. □

Lemma 3. *Q is minimal.*

Proof. No application process ever delivers a message m for which it did not receive a $\langle$"deliver"$, m, t\rangle$ message. No router process ever sends a message $\langle$"deliver"$, m, t\rangle$ to any application process that is not in $\mathrm{destsetpfx}(Q, m)$. Minimality of Q follows immediately. □

Lemma 4. *Q is submit-ordered.*

Proof. It is to show that for any pair of messages $m, m' \in M_Q$ where $\mathrm{sub}(m) \rightarrow \mathrm{sub}(m')$, it holds that $m \rightarrowtail_Q m'$. We first proof that this holds if $m, m' \in M_a^{\Sigma}$ for some application process a (Case 1, FIFO order delivery). We then proof the general case (Case 2, causal order delivery).

Case 1: Assume $m, m' \in M_a^{\Sigma}$ for some application process a. For the message m' to be learned by any router process, it must first have been proposed by some router process, call it r. To be proposed by r, r must have received a message $\langle$"route"$, m'\rangle$ from a. At that point, either m was already learned or not. If m was already learned, then m must be in Q at that point, so clearly $m \rightarrowtail_Q m'$. If m was not already learned, then m can also not have been acknowledged. As obvious from the application process algorithm, and since processes communicate through quasi-reliable FIFO channels, r must have received $\langle$"route"$, m\rangle$ before it received $\langle$"route"$, m'\rangle$. Therefore, r must have proposed m before m'. Since the atomic broadcast protocol guarantees FIFO order delivery, every router must learn m before m'. It follows that $m \rightarrowtail_Q m'$.

Case 2: Assume m and m' were submitted by different application processes. Without restriction of generality, say m was submitted by application process a, and m' was submitted by application process a'. For sub$(m) \to$ sub(m') to hold, there must be a message m'' submitted by a, such that sub$(m) \to$ sub(m'') or $m = m''$, and m'' is delivered to at least one application process. (Otherwise there could be no chain of causality between sub(m) and sub(m').) But to be delivered to any application process, m'' must first be learned by some router. Therefore, m'' must precede m' in Q; i.e., $m'' \rightarrowtail_Q m'$. If $m = m''$, then obviously $m \rightarrowtail_Q m'$. If $m \neq m''$, then Case 1 applies and $m \rightarrowtail_Q m''$. By transitivity of $\rightarrowtail_Q$, it holds that $m \rightarrowtail_Q m'$. □

Lemma 5. *Q is delivery-ordered.*

Proof. It is to show that for any pair of messages $m, m' \in M_Q$, where the delivery of m, call the event e, precedes the delivery of m', call the event e'; i.e., $e \to e'$, it holds that $m \rightarrowtail_Q m'$. We first proof that this holds if $m, m' \in M_a^\Delta$ for some application process a (Case 1). We then proof the general case (Case 2).

Case 1: The timestamp t of any message $\langle$"deliver"$, m, t\rangle$ sent by any router process, is the index (position) of m in Q. As obvious from the application process algorithm, application processes deliver messages strictly in timestamp order. It follows that for any application process a, for any pair of messages $m, m' \in M_a^\Delta$ where a delivers m before m', it holds that $m \rightarrowtail_Q m'$.

Case 2: For $e \to e'$ to hold, there must be a message m'' submitted by a, such that $e \to$ sub$(m'') \to e'$. By analogous argument to the proof of submit-orderedness, it must hold that $m \rightarrowtail_Q m''$, and $m'' \rightarrowtail_Q m'$. By transitivity of $\rightarrowtail_Q$, it holds that $m \rightarrowtail_Q m'$. □

Lemma 6. *Q is gap-free.*

Proof. Choose any pair of messages m, m' submitted by an application process a, where sub$(m) \to$ sub(m'), and m' is in M_Q. It is to show that m is also in M_Q. We assume that m is not in M_Q and proof by contradiction.

Application processes send "route"-messages strictly in submission order. Since m is not in M_Q, m cannot be acknowledged at any point, so for any router r to which a sends a message $\langle$"route", m'$\rangle$, a first sends a message $\langle$"route", m$\rangle$.

Processes communicate through quasi-reliable FIFO channels. Therefore, any router that receives a message ⟨“route”, m’⟩ must first receive a message ⟨“route”, m⟩. Since m is not in M_Q and thus obviously not stable, any router process that receives and proposes m' must also, previously, receive and propose m. Since the atomic broadcast algorithm preserves FIFO order, for m' to be learned, m must be learned first. Since m' is in M_Q, m must be in M_Q as well. Contradiction, hence proof. □

Lemma 7. *Q is complete.*

Proof. We proof by contradiction. Assume Q is *not* complete; i.e., there is some message $m \in M_Q$ that is not delivered by some correct application process $a \in \text{destsetpfx}(Q, m)$.

Assuming at least one correct router process that eventually considers itself a leader, every message $m \in M_Q$ is eventually sent to and received by every correct application process, including a. To be precise, a must receive a message $\langle\text{“deliver”}, m, t\rangle$. Consider the first time a receives $\langle\text{“deliver”}, m, t\rangle$. For a not to deliver m, a must have previously received a message $\langle\text{“deliver”}, m', t'\rangle$ where $t' > t$.

Router processes send “deliver”-messages in order of Q, that is timestamp order. Since a had not received and confirmed m yet, and a is a correct process, the router process that sent $\langle\text{“deliver”}, m', t'\rangle$ must previously have sent $\langle\text{“deliver”}, m, t\rangle$. Processes communicate through quasi-reliable FIFO channels, so a cannot have received $\langle\text{“deliver”}, m', t'\rangle$ before $\langle\text{“deliver”}, m, t\rangle$. Contradiction, hence proof. □

Lemma 8. *Q is terminated.*

Proof. Assuming every correct application process eventually sends its submitted messages to a correct router process that eventually considers itself a leader, every submitted message is eventually learned by every correct router process. Q is terminated by definition. □

Proof of Theorem 8. Theorem 8 follows by definition from Lemmas 2 through 8. □

B.2.2 Quiescence

Theorem 9. *For any possible execution of E-Cast, Quiescence holds.*

Let Q be the ever-growing m-seq that the router processes agree on. We prove the following Lemma.

Lemma 9. *Every message $m \in M_Q$ is eventually considered stable by every correct router process.*

Proof. We assume there is at least one correct router process that eventually considers itself a leader. For any message $m \in M_Q$, eventually one such router process will send ⟨"deliver", m, t⟩ messages (with some irrelevant timestamp t) to all processes in $D = \text{destsetpfx}(Q, m)$. Let us refer to this router process as r.

Eventually all correct application processes will respond with $\langle\text{“confirm”}, \text{id}(m)\rangle$ to r. Eventually r receives all the $\langle\text{“confirm”}, \text{id}(m)\rangle$ messages sent by correct application processes. It is assumed that the failure detectors used by application processes are complete. Therefore, any faulty process $a' \in D$ which does not confirm m is eventually suspected of failure, i.e. a correct application process submits a message $m' \in M_Q$ such that $a' \in \text{suspects}(m')$.

At the point where r has received $\langle\text{“confirm”}, \text{id}(m)\rangle$ from all correct application processes in D, and has learned of the failure of all faulty application processes in D, r proposes $\langle\text{“stable”}, \text{id}(m)\rangle$ to all router processes. Eventually, all correct router processes learn this message and consider m stable. □

Proof of Theorem 9. Lemma 9 guarantees that every message m in M_Q is eventually considered stable by every correct router process. As obvious from the router process algorithm, from the point on where a router process considers a message m stable, it responds to $\langle\text{“route”}, m\rangle$ messages with a message $\langle\text{“stable”}, \text{id}(m)\rangle$. Since the application process a that submitted m is correct, it eventually re-sends the message $\langle\text{“route”}, m\rangle$ to a correct router process and eventually receives a message $\langle\text{“stable”}, \text{id}(m)\rangle$ in response. □

List of Tables

List of Figures

List of Algorithms

Bibliography

[ABF$^+$92] P. M. G. Apers, C. A. V. D. Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, and A. N. Wilschut, "Prisma/db: A parallel, main memory relational dbms," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 4, no. 6, 1992.

[ABH] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented database systems," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 2.

[ACT97] M. K. Aguilera, W. Chen, and S. Toueg, "Heartbeat: A timeout-free failure detector for quiescent reliable communication," in *Proceedings of the 11th International Workshop on Distributed Algorithms*, ser. WDAG '97, 1997.

[ADHS01] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB '01, 2001.

[ADJ$^+$10] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez, "The datapath system: a data-centric analytic processing engine for large data warehouses," in *Proceedings of the 2010 International Conference on Management of Data*, ser. SIGMOD '10, 2010.

[ADS00] Y. Amir, C. Danilov, and J. Stanton, "A low latency, loss tolerant architecture and protocol for wide area group communication," in *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, ser. DSN 2000, 2000.

[AH00] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in *Proceedings of the 2000 International Conference on Management of Data*, ser. SIGMOD '00, 2000.

[AKMS10] M. K. Aguilera, I. Keidar, D. Malkhi, and E. Shraer, "Reconfiguring replicated atomic storage: A tutorial," *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, no. 102, 2010.

[AMMS+95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, "The totem single-ring ordering and membership protocol," *ACM Transactions on Computer Systems (TOCS)*, vol. 13, no. 4, 1995.

[Ant97] G. Antoshenkov, "Dictionary-based order-preserving string compression," *The VLDB Journal*, vol. 6, no. 1, 1997.

[Arm10] J. Armstrong, "Erlang," *Communications of the ACM (CACM)*, vol. 53, no. 9, 2010.

[AS07] N. Askitis and R. Sinha, "Hat-trie: A cache-conscious trie-based data structure for strings," in *In Proceedings of the 30th Australasian Conference on Computer Science*, ser. ACSC '07, 2007.

[Ash11] K. Ashmawy, "Partial live migration in scan-based database systems," Master's thesis, ETH Zurich, 2011.

[AT02] Y. Amir and C. Tutu, "From total order to database replication," in *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ser. ICDCS '02, 2002.

[ATJK09] M. Ault, M. Tumma, B. Jones, and S. Karam, *Oracle 11g Grid & Real Application Clusters: Oracle 11g Grid Computing with RAC.* Rampant TechPress, 2009.

[BBC+11] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. León, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, ser. CIDR '11, 2011.

[BBH+11] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos replicated state machines as the basis of a high-performance data store," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI '11, 2011.

[Ber10] J. Bernet, "Dictionary compression for a scan-based, main-memory database system," Master's thesis, ETH Zurich, 2010.

[BHF09a] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *Proceedings of the 35th International Conference on Management of Data*, ser. SIGMOD '09, 2009.

[BHF09b] ——, "Dictionary-based order-preserving string compression for main memory column stores," in *Proceedings of the 35th International Conference on Management of Data*, ser. SIGMOD '09, 2009.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., 1987.

[Bir93] K. P. Birman, "The process group approach to reliable distributed computing," *Communications of the ACM (CACM)*, vol. 36, no. 12, 1993.

[Bir10] K. Birman, "A history of the virtual synchrony replication model," in *Replication: Theory and Practice.* Springer, 2010.

[BJ87] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *Proceedings of the eleventh ACM Symposium on Operating Systems Principles*, ser. SOSP '87, 1987.

[BKK$^+$03] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in p2p systems," *Communications of the ACM (CACM)*, vol. 46, no. 2, 2003.

[BMR10] K. Birman, D. Malkhi, and R. V. Renesse, "Virtually synchronous methodology for dynamic service replication," Microsoft Research, Tech. Rep. MSR-TR-2010-151, 2010.

[BR94] K. P. Birman and R. V. Renesse, *Reliable Distributed Computing with the ISIS Toolkit.* IEEE Computer Society Press, 1994.

[Bry77] R. E. Bryant, "Simulation of packet communication architecture computer systems," Massachusetts Institute of Technology, Tech. Rep., 1977.

[Bur06] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '06, 2006.

[CAB$^+$81] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost, "A history and evaluation of system r," *Communications of the ACM (CACM)*, vol. 24, no. 10, 1981.

[Cat10] R. Cattell, "Scalable sql and nosql data stores," *SIGMOD Record*, vol. 39, no. 4, 2010.

[CDG$^+$08] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, 2008.

[CF02] S. Chandrasekaran and M. J. Franklin, "Streaming queries over streaming data," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02, 2002.

[CG05] G. Cormode and M. Garofalakis, "Sketching streams through the net: distributed approximate query tracking," in *Proceedings of the 31st International conference on Very Large Data Bases*, ser. VLDB '05, 2005.

[CGG$^+$09] G. Chockler, S. Gilbert, V. Gramoli, P. M. Musial, and A. A. Shvartsman, "Reconfigurable distributed storage for dynamic networks," *Journal of Parallel and Distributed Computing*, vol. 69, no. 1, 2009.

[CGR07] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '07, 2007.

[CGR11] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming.* Springer, 2011.

[Cha98] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of the 17th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '98, 1998.

[CJ89] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN Systems*, vol. 17, no. 1, 1989.

[CLN10] S. Chaudhuri, H. Lee, and V. R. Narasayya, "Variance aware optimization of parameterized queries," in *Proceedings of the 2010 International Conference on Management of Data*, ser. SIGMOD '10, 2010.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[CM81] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM (CACM)*, vol. 24, no. 4, 1981.

[CM05] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, 2005.

[Cod83] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM (CACM)*, vol. 26, no. 1, 1983.

[CPV09] G. Candea, N. Polyzotis, and R. Vingralek, "A scalable, predictable join operator for highly concurrent data warehouses," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, 2009.

[CPV11] ——, "Predictable performance and high query concurrency for data analytics," *The VLDB Journal*, vol. 20, no. 2, 2011.

[CRS$^+$08] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, no. 2, 2008.

[CT96] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, 1996.

[CVSS$^+$11] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White, "Fast checkpoint recovery algorithms for frequently consistent applications," in *Proceedings of the 2011 International Conference on Management of Data*, ser. SIGMOD '11, 2011.

[DAEA09] S. Das, D. Agrawal, and A. El Abbadi, "Elastras: An elastic transactional data store in the cloud," in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, ser. HotCloud'09, 2009.

[DAEA10] ——, “G-store: a scalable data store for transactional multi key access in the cloud,” in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC ’10, 2010.

[DAF$^+$03] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer, “Path sharing and predicate evaluation for high-performance xml filtering,” *ACM Transactions on Database Systems (TODS)*, vol. 28, no. 4, 2003.

[DB97] B. B. Dept and B. Ban, “Adding group communication to java in a non-intrusive way using the ensemble toolkit,” Cornell University, Tech. Rep., 1997.

[DG92] D. DeWitt and J. Gray, “Parallel database systems: The future of high performance database systems,” *Communications of the ACM (CACM)*, vol. 35, no. 6, 1992.

[DGH$^+$87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, “Epidemic algorithms for replicated database maintenance,” in *Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’87, 1987.

[DGP$^+$07] A. Demers, J. Gehrke, B. Pandal, M. Riedewald, V. Sharma, and W. White, “Cayuga: A general purpose event monitoring system,” in *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*, ser. CIDR ’07, 2007.

[DGS$^+$90] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen, “The gamma database machine project,” *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 2, no. 1, 1990.

[DHJ$^+$07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s highly available key-value store,” in *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP ’07, 2007.

[DM96] D. Dolev and D. Malki, “The transis approach to high availability cluster communication,” *Communications of the ACM (CACM)*, vol. 39, no. 4, 1996.

[DNS91] D. J. DeWitt, J. F. Naughton, and D. A. Schneider, “An evaluation of non-equijoin algorithms,” in *Proceedings of the 17th International Conference on Very Large Data Bases*, ser. VLDB ’91, 1991.

[Dre07] U. Drepper, “What every programmer should know about memory,” 2007. [Online]. Available: http://people.redhat.com/drepper/cpumemory.pdf

[DSU04] X. Défago, A. Schiper, and P. Urbán, “Total order broadcast and multicast algorithms: Taxonomy and survey,” *ACM Computing Surveys*, vol. 36, no. 4, 2004.

[Fer94] P. M. Fernandez, “Red brick warehouse: a read-mostly rdbms for open smp platforms,” in *Proceedings of the 1994 International Conference on Management of Data*, ser. SIGMOD ’94, 1994.

[FI01] U. Fritzke Jr. and P. Ingels, “Transactions on partially replicated data based on reliable and atomic multicasts,” in *Proceedings of the 21st International Conference on Distributed Computing Systems*, ser. ICDCS ’01, 2001.

[Fit04] B. Fitzpatrick, “Distributed caching with memcached,” *Linux Journal*, no. 124, 2004.

[FJL$^+$01] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, “Filtering algorithms and implementation for very fast publish/subscribe systems,” in *Proceedings of the 2001 International Conference on Management of Data*, ser. SIGMOD ’01, 2001.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM (JACM)*, vol. 32, no. 2, 1985.

[Fly72] M. J. Flynn, “Some computer organizations and their effectiveness,” *IEEE Transactions on Computers*, vol. 21, no. 9, 1972.

[FM85] P. Flajolet and G. N. Martin, “Probabilistic counting algorithms for data base applications,” *Journal of Computer and System Sciences*, vol. 31, no. 2, 1985.

[FMF$^+$11] D. Fauser, J. Meyer, C. Florimond, D. Kossmann, G. Alonso, G. Giannikis, and P. Unterbrunner, “Continuous full scan data store table and distributed data store featuring predictable answer time for unpredictable workload,” WO Patent Application 2011/023 652 A2, 2011.

[Fow02] M. Fowler, *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional, 2002.

[FR10] A. Fekete and K. Ramamritham, "Consistency models for replicated data," in *Replication: Theory and Practice.* Springer, 2010.

[Fre60] E. Fredkin, "Trie memory," *Communications of the ACM (CACM)*, vol. 3, no. 9, 1960.

[Fri10] S. L. Fritchie, "Chain replication in theory and in practice," in *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, ser. Erlang '10, 2010.

[FSS$^+$10] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitski, E. Vee, S. Venkatesan, and J. Zien, "Efficiently evaluating complex boolean expressions," in *Proceedings of the 2010 International Conference on Management of Data*, ser. SIGMOD '10, 2010.

[GAK12] G. Giannikis, G. Alonso, and D. Kossmann, "Shareddb: Killing one thousand queries with one stone," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 5, no. 6, 2012.

[GHvR$^+$97] K. Guo, M. Hayden, R. van Renesse, W. Vogels, and K. P. Birman, "Gsgc: An efficient gossip-style garbage collection scheme for scalable reliable multicast," Tech. Rep., 1997.

[Gia09] G. Giannikis, "Deadalus: A distributed crescando system," Master's thesis, ETH Zurich, 2009.

[GJL09] U. Germann, E. Joanis, and S. Larkin, "Tightly packed tries: How to fit large models into memory, and make them load fast, too," in *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, ser. SETQA-NLP '09, 2009.

[GKK$^+$11] G. Graefe, A. C. König, H. A. Kuno, V. Markl, and K.-U. Sattler, "Robust query processing," in *Dagstuhl Seminar Proceedings*, no. 10381, 2011.

[GL02] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, 2002.

[GL03] E. Gafni and L. Lamport, "Disk paxos," *Distributed Computing*, vol. 16, no. 1, 2003.

[GR92] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1992.

[Gra11] T. Granlund, "Instruction latencies and throughput for amd and intel x86 processors," January 2011. [Online]. Available: http://gmplib.org/~tege/x86-timing.pdf

[Gut84] A. Guttman, "R-trees: a dynamic index structure for spatial searching," *SIGMOD Record*, vol. 14, no. 2, 1984.

[HA05] S. Harizopoulos and A. Ailamaki, "Stageddb: Designing database servers for modern hardware," in *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2005.

[Hab10] M. Habeeb, *A Developer's Guide to Amazon SimpleDB.* Addison-Wesley Professional, 2010.

[HDYK04] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The φ accrual failure detector," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, ser. SRDS '04, 2004.

[HKJR10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Annual Technical Conference*, ser. USENIXATC '10, 2010.

[HL05] G. L. Heileman and W. Luo, "How caching affects hashing," in *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics*, ser. ALENEX/ANALCO '05, 2005.

[HLAM06] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden, "Performance tradeoffs in read-optimized databases," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, ser. VLDB '06, 2006.

[HSA05] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki, "Qpipe: a simultaneously pipelined relational query engine," in *Proceedings of the 2005 International Conference on Management of Data*, ser. SIGMOD '05, 2005.

[HT94] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Cornell University, Tech. Rep., 1994.

[HTC05] Y. Horita, K. Taura, and T. Chikayama, “A scalable and efficient self-organizing failure detector for grid applications,” in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, ser. GRID ’05, 2005.

[HW90] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, 1990.

[HZW02] S. Heinz, J. Zobel, and H. E. Williams, “Burst tries: a fast, efficient data structure for string keys,” *ACM Transactions on Information Systems (TOIS)*, vol. 20, no. 2, 2002.

[Int09] *SQL*, International Organization for Standardization (ISO) Std. ISO/IEC 9075(1-4,9-11,13,14):2008, Rev. 3, 2009.

[Int11] Intel Corporation, “Intel advanced vector extensions programming reference,” June 2011. [Online]. Available: http://software.intel.com/file/36945

[JRS11] F. Junqueira, B. Reed, and M. Serafini, “Zab: High-performance broadcast for primary-backup systems,” in *Proceedings of the 2011 International Conference on Dependable Systems and Networks*, ser. DSN 2011, 2011.

[KA00] B. Kemme and G. Alonso, “Don’t be lazy, be consistent: Postgres-r, a new way to implement database replication,” in *Proceedings of the 26th International Conference on Very Large Data Bases*, ser. VLDB ’00, 2000.

[KBB01] B. Kemme, A. Bartoli, and O. Babaoglu, “Online reconfiguration in replicated databases based on group communication,” in *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, ser. DSN 2001, 2001.

[KDGA11] V. Kantere, D. Dash, G. Gratsias, and A. Ailamaki, “Predicting cost amortization for query services,” in *Proceedings of the 2011 International Conference on Management of Data*, ser. SIGMOD ’11, 2011.

[KLL$^+$97] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web,” in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC ’97, 1997.

[KN11] A. Kemper and T. Neumann, “Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots,” in *Proceedings of the 27th IEEE International Conference on Data Engineering*, ser. ICDE ’11, 2011.

[Kra10] T. Kraska, “Building database applications in the cloud,” Ph.D. dissertation, ETH Zurich, 2010.

[KZHL07] S. Ktari, M. Zoubert, A. Hecker, and H. Labiod, “Performance evaluation of replication strategies in dhts under churn,” in *Proceedings of the 6th International Conference on Mobile and Ubiquitous Multimedia*, ser. MUM ’07, 2007.

[Lam78] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM (CACM)*, vol. 21, no. 7, 1978.

[Lam84] ——, “Using time instead of timeout for fault-tolerant distributed systems,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 6, no. 2, 1984.

[Lam98] ——, “The part-time parliament,” *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, 1998.

[Lam05] ——, “Generalized consensus and paxos,” Microsoft Research, Tech. Rep. MSR-TR-2005-33, 2005.

[Lam06] ——, “Fast paxos,” *Distributed Computing*, vol. 19, no. 2, 2006.

[LBM+07] C. Lang, B. Bhattacharjee, T. Malkemus, S. Padmanabhan, and K. Wong, “Increasing buffer-locality for multiple relational table scans through grouping and throttling,” in *Proceedings of the 23rd IEEE International Conference on Data Engineering*, ser. ICDE ’07, 2007.

[LJ03] H. K. Y. Leung and H.-A. Jacobsen, “Efficient matching for state-persistent publish/subscribe systems,” in *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON ’03, 2003.

[LM04] L. Lamport and M. Massa, “Cheap paxos,” in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, ser. DSN 2004, 2004.

[LM10] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Operating Systems Review*, vol. 44, no. 2, 2010.

[LMZ08] L. Lamport, D. Malkhi, and L. Zhou, "Stoppable paxos," Microsoft Research, Tech. Rep., 2008.

[LMZ10] ——, "Reconfiguring a state machine," *SIGACT News*, vol. 41, no. 1, 2010.

[LS02] N. Lynch and A. A. Shvartsman, "Rambo: A reconfigurable atomic memory service for dynamic networks," in *Proceedings of the 16th International Conference on Distributed Computing*, ser. DISC '02, 2002.

[Mar83] J. Martin, *Managing the Data Base Environment.* Prentice Hall, 1983.

[MB76] R. M. Metcalfe and D. R. Boggs, "Ethernet: distributed packet switching for local computer networks," *Communications of the ACM (CACM)*, vol. 19, no. 7, 1976.

[MBK00] S. Manegold, P. A. Boncz, and M. L. Kersten, "Optimizing database architecture for the new bottleneck: Memory access," *The VLDB Journal*, vol. 9, no. 3, 2000.

[Mor68] D. R. Morrison, "Patricia—practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM (JACM)*, vol. 15, no. 4, 1968.

[MPSP10] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring paxos: A high-throughput atomic broadcast protocol," in *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN 2010, 2010.

[MRS+04] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, "Robust query processing through progressive optimization," in *Proceedings of the 2004 International Conference on Management of Data*, ser. SIGMOD '04, 2004.

[OBD08] K. Ostrowski, K. Birman, and D. Dolev, "Quicksilver scalable multicast (qsm)," in *Proceedings of the 2008 Seventh IEEE International Symposium on Network Computing and Applications*, ser. NCA '08, 2008.

[OBDA08] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn, "Programming with live distributed objects," in *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ser. ECOOP '08, 2008.

[OV99] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, 2nd ed. Prentice Hall, 1999.

[PGS03] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed Parallel Databases*, vol. 14, no. 1, 2003.

[PJHA10] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki, "Data-oriented transaction execution," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, 2010.

[PLL97] R. D. Prisco, B. W. Lampson, and N. A. Lynch, "Revisiting the paxos algorithm," in *Proceedings of the 11th International Workshop on Distributed Algorithms*, ser. WDAG '97, 1997.

[QRR+08] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman, "Main-memory scan sharing for multi-core cpus," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 1, no. 1, 2008.

[Raz94] Y. Raz, "Serializability by commitment ordering," *Information Processing Letters*, vol. 51, no. 5, 1994.

[RD01] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, ser. Middleware '01, 2001.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01, 2001.

[RG10] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10, 2010.

[RHS95] G. Ray, J. R. Haritsa, and S. Seshadri, "Database compression: A performance enhancement tool," in *Proceedings of the 7th International Conference on Management of Data*, ser. COMAD '95, 1995.

[Ris05] S. Ristov, "Lz trie and dictionary compression," *Software—Practice & Experience*, vol. 35, no. 5, 2005.

[Ros02] K. A. Ross, "Conjunctive selection conditions in main memory," in *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '02, 2002.

[Ros04] ——, "Selection conditions in main memory," *ACM Transactions on Database Systems (TODS)*, vol. 29, no. 1, 2004.

[RSQ+08] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle, "Constant-time query processing," in *Proceedings of the 24th IEEE International Conference on Data Engineering*, ser. ICDE '08, 2008.

[RST11] J. Rao, E. J. Shekita, and S. Tata, "Using paxos to build a scalable, consistent, and highly available datastore," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 4, no. 4, 2011.

[RT04] M. Ronström and L. Thalmann, "Mysql cluster architecture overview: High availability features of mysql cluster," A MySQL Technical White Paper, MySQL AB, 2004.

[Sch90] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, 1990.

[Sch06] A. Schiper, "Dynamic group communication," *Distributed Computing*, vol. 18, no. 5, 2006.

[Sel88] T. K. Sellis, "Multiple-query optimization," *ACM Transactions on Database Systems (TODS)*, vol. 13, no. 1, 1988.

[SMA+07] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (it's time for a complete rewrite)," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07, 2007.

[SMK+01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01, 2001.

[SOMP01] A. Sousa, R. Oliveira, F. Moura, and F. Pedone, "Partial replication in the database state machine," in *2001 IEEE International Symposium on Network Computing and Applications*, ser. NCA '01, 2001.

[SPC$^+$03] K. Sagonas, M. Pettersson, R. Carlsson, P. Gustafsson, and T. Lindahl, "All you wanted to know about the hipe compiler: (but might have been afraid to ask)," in *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, ser. ERLANG '03, 2003.

[SPU11] B. Satzger, A. Pietzowski, and T. Ungerer, "Autonomous and scalable failure detection in distributed systems," *International Journal of Autonomous Adaptive Communication Systems*, vol. 4, no. 1, 2011.

[SRHS10] F. Schintke, A. Reinefeld, S. Haridi, and T. Schütt, "Enhanced paxos commit for transactions on dhts," in *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10, 2010.

[SSP10] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems*, ser. SRDS '10, 2010.

[SSR08] T. Schütt, F. Schintke, and A. Reinefeld, "Scalaris: Reliable transactional p2p key/value store," in *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, ser. ERLANG '08, 2008.

[Sta02] J. R. Stanton, "A users guide to spread version 0.11," 2002. [Online]. Available: http://www.spread.org/docs/guide/users_guide.pdf

[Ste10] R. Steiner, "Multi-query optimization in a scan-based relational main-memory table," Master's thesis, ETH Zurich, 2010.

[Sto08] M. Stonebraker, "One size fits all: an idea whose time has come and gone," *Communications of the ACM (CACM)*, vol. 51, no. 12, 2008.

[Sut05] H. Sutter, "The free lunch is over — a fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, 2005.

[TA10] A. Thomson and D. J. Abadi, "The case for determinism in database systems," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, 2010.

[Tan02] A. S. Tanenbaum, *Computer Networks*, 4th ed. Prentice Hall, 2002.

[TDP$^+$94] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch, "Session guarantees for weakly consistent replicated data," in *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, ser. PDIS '94, 1994.

[TTP$^+$95] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95, 1995.

[UGAK09] P. Unterbrunner, G. Giannikis, G. Alonso, and D. Kossmann, "Predictable performance for unpredictable workloads," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, 2009.

[VCO10] H. T. Vo, C. Chen, and B. C. Ooi, "Towards elastic transactional cloud storage with range query support," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, 2010.

[VKCD01] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev, "Group communication specifications: A comprehensive study," *ACM Computing Surveys*, vol. 33, no. 4, 2001.

[Vog09] W. Vogels, "Eventually consistent," *Communications of the ACM (CACM)*, vol. 52, no. 1, 2009.

[vRBM96] R. van Renesse, K. P. Birman, and S. Maffeis, "Horus: a flexible group communication system," *Communications of the ACM (CACM)*, vol. 39, no. 4, 1996.

[vRS04] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '04, 2004.

[WCY04] K.-L. Wu, S.-K. Chen, and P. S. Yu, "Interval query indexing for efficient stream processing," in *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*, ser. CIKM '04, 2004.

[WGMB$^+$09] S. E. Whang, H. Garcia-Molina, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, and R. Yerneni, "Indexing boolean expressions," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, 2009.

[Wha90] K.-Y. Whang et. al., "A linear-time probabilistic counting algorithm for database applications," *ACM Transactions on Database Systems (TODS)*, vol. 15, no. 2, 1990.

[WM95] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, 1995.

[WPC10] Z. Wei, G. Pierre, and C.-H. Chi, "Cloudtps: Scalable transactions for web applications in the cloud," Vrije Universiteit, Tech. Rep. IR-CS-053, 2010.

[WS05] M. Wiesmann and A. Schiper, "Comparison of database replication techniques based on total order broadcast," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 17, no. 4, 2005.

[ZHNB07] M. Zukowski, S. Héman, N. Nes, and P. Boncz, "Cooperative scans: Dynamic bandwidth sharing in a dbms," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07, 2007.

[ZHS$^+$04] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, 2004.

Curriculum Vitæ: Philipp Thomas Unterbrunner

Research Interests

Main Memory Data Management
Reliable Distributed Systems
Distributed Concurrency Control
Search and Optimization

Education

2007–2012	Doctoral Candidate in Computer Science ETH Zurich, Switzerland
2006–2007	Master of Engineering in Computer Science Cornell University, Ithaca NY, USA
2001–2005	Diplom-Ingenieur (FH) in Software Engineering FH Oberösterreich, Hagenberg im Mühlkreis, Austria

Professional Experience

2007–2012	University Assistant and Doctoral Candidate ETH Zurich, Switzerland
2009	Intern, Software Development Engineer Microsoft, SQL Server Core, Redmond WA, USA
2005–2006	Software Engineer Philotech GmbH, Unterhaching, Germany
2004–2005	Software Engineer EADS Military Aircraft, Ottobrunn, Germany

Publications

P. Unterbrunner, G. Alonso, and D. Kossman, "E-Cast: Elastic Multicast," ETH Zurich, Tech. Rep. TR-719, 2011.

D. Fauser, J. Meyer, C. Florimond, D. Kossmann, G. Alonso, G. Giannikis, and P. Unterbrunner, "Continuous Full Scan Data Store Table and Distributed Data Store Featuring Predictable Answer Time for Unpredictable Workload," WO Patent Application 2011/023652 A2, 2011.

G. Giannikis, P. Unterbrunner, J. Meyer, G. Alonso, D. Fauser, and D. Kossman, "Crescando" (Demo Paper), *Proceedings of the 2010 International Conference on Management of Data*, ser. SIGMOD '10, 2010.

P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann, "Predictable Performance for Unpredictable Workloads," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, 2009.

N. Gupta, A. Demers, J. Gehrke, P. Unterbrunner, and W. White, "Scalability for Virtual Worlds", *Proceedings of the 25th IEEE International Conference on Data Engineering*, ser. ICDE '09, 2009.

P. Unterbrunner, "Toward Intelligent Agents for Real-Time Avionic Systems," Diploma thesis, FH Oberösterreich, 2005.

Affiliations

- Member of the Association for Computing Machinery (ACM) and the ACM Special Interest Group on Management of Data (SIGMOD)
- Member of the Institute of Electrical and Electronics Engineers (IEEE) and the IEEE Computer Society
- Member of the Gesellschaft für Informatik e.V. (GI) and the GI Fachbereich Datenbanken und Informationssysteme (DBIS)