Master’s Thesis Nr. 30
Systems Group, Department of Computer Science, ETH Zurich
in collaboration with
University of Stuttgart and Amadeus IT Group SA

Partial live migration in scan-based database systems

by
Khalid Ashmawy

Supervised by
Prof. Paul J. Kuhn
Prof. Donald Kossmann
Prof. Gustavo Alonso
Philipp Unterbrunner

Zurich, 17 December 2011
Declaration

Hereby I certify that I have realized this work on my own, and that all sources that I have used or consulted are duly noted herein.

Furthermore, I certify that I know and I accept that I have no right to exploit the results of my Master Thesis by any means without the written permission of the Institute of Communication Networks and Computer Engineering (IKR).

Khalid Ashmawy
17 December 2011
Abstract

This master thesis aims to add partial live migration features in Crescando[17]. Crescando is a scalable, distributed relational table implementation based on parallel, collaborative scans in memory. These features developed in this thesis provide the building block for implementing elastic scalability and high availability in Crescando. Elastic scalability refers to adding or removing storage nodes to a cluster without downtime. High availability refers to avoiding unplanned outages by eliminating single points of failure. One of the methodologies for providing high availability is fault tolerance by replication.[10] Both, elastic scalability and high availability, require an efficient method to copy or move data across storage nodes, which this master thesis provides.

The problem is tackled in a black-box approach. Crescendo external user interface is used to solve the problem, rather than altering its implementation. Crescando’s simple operations (Select, Insert, Delete) are used as the elementary units to provide the functionality of copying and moving data across nodes. One of the challenges of building such a system is migrating the contents of a relational table with minimal impact on the whole system availability and performance. Optimizations are incorporated to achieve efficient data transfer such that data transfer rate saturates a gigabit Ethernet interface. The system interrupt duration is minimized to the period required for data transfer. Moreover, certain consistency guarantees must be provided by the solution. Our solution guarantees linearizability[9], a well-known strong consistency guarantee.

The migration system developed in this thesis is employed by a higher level layer known as Rubberband[16]. Rubberband implements a well-known replication scheme, known as successor-list replication[14]. Rubberband instructs appropriate nodes in a dynamic set of nodes to shuffle data using the migration system developed in this thesis. They are instructed to shuffle data in order to maintain successor-list replication scheme as storage nodes join and part the system.
Preface

This thesis is part of the Crescando research project developed by Philipp Unterbrunner at the Systems Group, Department of Computer Science, ETH Zürich. The thesis is part of INFO TECH master program (Stuttgart University) in Computer Science. The thesis duration was 6 months, from 1st of May 2011 to 1st November 2011.

Acknowledgments

First most I would like to express my sincere gratitude to Philipp Unterbrunner, my supervisor, for providing me with the opportunity to work on Crescando in a challenging project and for providing me with encouragement and guidance during the thesis. I would like to thank Prof. Paul J. Kühn, Prof. Donald Kossman and Prof. Gustavo Alonso for their support. Thank you for making this opportunity possible. I would like to express my deepest appreciation to my parents for always believing in me and motivating me all along the way. Through the years, I came along life-long friends, I greatly value their friendship, and I would like to take this opportunity to thank them.
Contents

Declaration i

Preface v

1 Introduction 1
  1.1 Problem Statement ............................................ 2
  1.2 Contribution .................................................. 3
  1.3 Thesis outline ................................................ 3

2 Crescando/RB 5
  2.1 Architecture .................................................. 5
  2.2 Crescando ...................................................... 6
    2.2.1 Overview .................................................. 6
    2.2.2 Operations ............................................... 9
    2.2.3 Scan Cycle ............................................... 10
    2.2.4 Basic performance numbers .............................. 11
  2.3 E-Cast ....................................................... 12
  2.4 Rubberband .................................................... 13

3 Architecture and Design 17
  3.1 Design space and related work ............................... 17
  3.2 Methodology Used ............................................. 20
  3.3 Architecture .................................................. 21
  3.4 Design ......................................................... 22
    3.4.1 Overview ................................................ 22
    3.4.2 Request Queue .......................................... 23
    3.4.3 Remote copying operations .............................. 24
    3.4.4 Inconsistent Select ................................... 27
  3.5 Correctness ................................................... 29
    3.5.1 Formal Model ............................................ 29
    3.5.2 Correctness properties ................................ 30

4 Implementation Details 31
  4.1 Interface ...................................................... 31
  4.2 Overview ...................................................... 33
  4.3 Implementation ............................................... 34
  4.4 Remote Transfer Protocol ................................. 39
  4.5 Optimizations ............................................... 41
4.5.1 Pipelining operations ........................................ 41
4.5.2 Send, receive and engine buffer .......................... 42

5 Performance Evaluation ........................................ 43
  5.1 Test Environment ............................................. 43
  5.2 Primitive Operations ........................................ 43
    5.2.1 BulkLoad and Insert .................................... 45
    5.2.2 Select .................................................. 48
    5.2.3 Refreshing Statistics .................................... 51
  5.3 End-to-End .................................................. 52
    5.3.1 Remote Copy ............................................ 52
    5.3.2 Crescando/RB ........................................... 58

6 Conclusions ..................................................... 61
  6.1 Future Work ................................................ 61

A Appendix ......................................................... 63
  A.1 Amadeus Itinerary Schema .................................. 63
Chapter 1

Introduction

Cloud computing continues to rise in popularity, accompanied by a paradigm shift to a model offering much more flexibility than past computing methods. Along with the rising number and popularity of cloud platforms (such as Microsoft Azure[5], Amazon EC2[1], Google App Engine[3] and that provided by Facebook), vast amounts of applications are being deployed on the cloud.[2] The workload experienced by those applications is unpredictable. Elastic scalability is one of the key features that are critical for those platforms. Elastic scalability refers to offering the ability for nodes to join and leave dynamically within a running cluster without downtime. Elastic scalability is a vital component to cloud computing. Moreover dependable and reliable computing have become a necessity in cloud computing, where the demand for high availability is continuously increasing. That is, the service provided to the end-users should tolerate node failures without service interruption. One of the methodologies for providing high availability is fault tolerance by replication, which refers to storing multiple copies of data objects at more than one location.[10]

Challenges lie in implementing a system with the mentioned properties. One of the key challenges is ensuring that consistency guarantees are met by each storage node of the system, and that the set of storage replicas in the system are consistent with respect to each other. A well-known strong consistency guarantee is linearizability[9], where the system behaves as if only one copy of each data object exists. Linearizability is the consistency model guaranteed by each storage node in this thesis. A system fulfilling such strong consistency guarantees while providing elastic scalability and high availability is a difficult and challenging task. This thesis develops partial live migration at the storage engine level, which is integrated in a cloud storage framework to deliver a system with the mentioned properties, elastic scalability and high availability.

The storage engine used in this thesis is Crescando. Crescando is a scalable, distributed relational table implementation. Crescando has been designed to perform predictably under unpredictable workloads.[17] This means that Crescando provides latency and freshness guarantees for all workloads. Crescando achieves this by operating on data in main memory in parallel, collaborative scans. Processing platforms which require heavy write workload mixed with
read workload benefit from these latency and freshness guarantees. Generally if the workload is unknown or changing, Crescando is quite powerful in being able to meet customer-based service-level agreement regarding latency and freshness.

Partial live migration developed in this thesis is employed by E-Cast and Rubberband. E-Cast and Rubberband [16] is a generic framework for elastic, fault-tolerant, strongly consistent data stores for the cloud. E-Cast and Rubberband is typically used with one or more Crescando storage engine. E-Cast is a casual-total-order multicast protocol that solves stateful routing. Rubberband is an application of E-Cast, where Rubberband uses E-Cast to implement replication (successor-list replication[14]). The replication scheme used in the system is ROWA (Read-One-Write-All) where a read can be processed by a single replica, but a write have to be processed by every replica to ensure that all replicas are consistent with respect to each other. E-Cast, Rubberband and Crescando form Crescando/RB system, an elastic, scalable, fault tolerant cloud storage which ensures linearizable consistency. E-Cast and Rubberband delegate partial live migration to the system developed in this thesis in order to maintain successor-list replication scheme in Crescando/RB.

This thesis develops a partial live migration system on top of Crescando storage engine. This system allows nodes to copy or move specified part of their data to another node. Rubberband and E-Cast act as a higher level layer instructing nodes within Crescando/RB system to shuffle their data to ensure the correct partitioning and replication in accordance with the latest configuration. The logic used by Rubberband in instructing nodes to shuffle data around in order to maintain compliance with successor-list replication scheme is out of the scope of this thesis and is discussed in E-Cast and Rubberband paper[16].

1.1 Problem Statement

This master thesis aims to implement a system on top of Crescando storage engine which add partial live migration features to Crescando.

The requirements of the system are as follows:

1. **Consistency**: The live migration implementation must preserve linearizability. This is discussed in more details in section 3.5.

2. **Migration duration**: The migration duration should be such that the transfer rate saturates a gigabit Ethernet interface. This is discussed in more details in section 5.3.1.

3. **Impact of migration on the system performance**: Only the nodes involved in the migration shall be impacted. The impact duration should only be for the period of migration. This is discussed in more details in section 5.3.1.
1.2 Contribution

This thesis presents the design and implementation of a partial live migration system. This system is built on top of Crescando. It is implemented in a black-box approach. It is implemented without changing the storage engine implementation, where the external user interface of Crescando is used. This system allows data to be shuffled as requested between dynamic set of storage nodes (running Crescando) joining and leaving the cluster. The data transfer rate upon which nodes copy or move data saturates a gigabit Ethernet interface. The live migration implementation preserves linearizability. Only the nodes involved in the migration are impacted, for the period upon which migration takes place. This presented system serves as the building block for implementing elastic scalability and high availability in Crescando/RB[16].

1.3 Thesis outline

The document is organized as follows: Chapter 2 presents Crescando/RB which includes Crescando storage engine, E-Cast and Rubberband. Chapter 3 presents the design space and related work of partial live migration systems, followed by architecture and design of the system developed in this thesis and finally a formal proof of correctness of that system. Implementation details are elaborated in Chapter 4. Chapter 5 investigates Crescando primitive operations performance using micro-benchmarks and the whole system performance using benchmarks and elaborates how the developed system impacts Crescando/RB throughput. Chapter 6 concludes the thesis. The Appendix provides the Amadeus Itinerary schema, a schema used heavily in the experiments.
4 Introduction
Chapter 2

Crescando/RB

2.1 Architecture

Crescando/RB is an elastic, scalable, fault tolerant cloud storage system. It consists of two orthogonal layers: the storage engine (Crescando) and the distribution layer. The distribution layer itself consists of Rubberband and E-Cast. Crescando is a scan-based storage engine. E-Cast is a casual-total-order multicast protocol that solves stateful routing. Rubberband is an application of E-Cast, where Rubberband implements successor-list replication using E-Cast.[16]

The protocol stack used by Crescando/RB is shown in figure 2.1. There are 5 layers in the shown protocol stack. These are all within the application layer of Crescando/RB system. Underlying layers such as Transport, Internet, Link and Physical are not shown. Messages in Crescando/RB system go through the presented protocol stack. However, when migration occurs between two nodes, the data transfer occurs only on the Partial Live Migration layer directly between the two nodes.

![Figure 2.1: Crescando/RB Protocol Stack](image-url)
Crescando/RB system contains four types of processes. Storage processes (Crescando), router processes, super processes and client processes. These are illustrated in figure 2.2. Client processes submits read and write messages which the router processes forwards to the appropriate storage processes while super processes actively maintain the mapping of keys to storage processes.

This master thesis develops a partial live migration system on top of Crescando storage engine. The functionality developed in this thesis is employed by Rubberband to shuffle data among a dynamic set of nodes to ensure the correct partitioning and replication of the data. Rubberband maintains successor-list replication using partial live migration system.

### 2.2 Crescando

#### 2.2.1 Overview

Crescando is the storage engine used in Crescando/RB. Crescando is a scalable, distributed relational table implementation. Crescando is based on a scan-only architecture (no indexes) and it uses the main-memory storage and data partitioning to scale-up linearly on multi-core machines. The rationale for avoiding data indexing is that data indexing lead to complex systems that are expensive to maintain and does not guarantee predictable performance for all workload types. Traditional databases can support high query throughput for light update load, however as soon as the update load increase, the performance drops quickly. Data is partitioned in Crescando into memory segments, such that each
2.2 Crescando

Operations

![Diagram of Crescando Memory segments]

Memory Segment 1, Core 1  Memory Segment 2, Core 2  Memory Segment 3, Core 3  Memory Segment 4, Core 4

Figure 2.3: Crescando Memory segments

A memory segment is assigned to one dedicated processor core. Operations are queued for each memory segment and handled by the dedicated processor. In figure 2.3, the data is partitioned into 4 memory segments: segment 1, segment 2, segment 3 and segment 4. Each core is responsible for a memory segment. Operations for a specific memory segment are handled by the core responsible for that memory segment. These design decisions discussed here allow Crescando to provide predictable performance for unpredictable workloads.[17]

Crescando should provide latency and freshness guarantees for all workloads. Described formally, the following are the performance requirements of Crescando:

- **Query Latency** where any query must be answered in two seconds,
- **Data Freshness** where updates must be visible within two seconds,
- **Query Diversity** dictating that the system must support any query regardless of its Select predicates,
- **Update Load** where average update load is 1 update/GB*sec and peak load is 20 updates/GB*sec for up to 30 seconds and **Scalability** where the system must scale linearly with the read workload by adding machines and individual cores.[17]

Crescando employs collaborative (shared) scans known as Clock Scan. A scan cycle in Clock Scan is illustrated in figure 2.4. The Clock Scan algorithm uses two scan cursors, one cursor for write operations and another cursor for read operations. The write cursor always leads the read cursor but cannot pass the read cursor. Operations are batched together and categorized into either queries or updates. Updates are executed on the data under the scan cursor. The read cursor always sees a consistent snapshot of the data due to the fact that the write cursor executes Updates strictly in arrival order. That design allows Clock Scan to achieve both high query and update throughput with predictable latency.[17]

Data stored in each Crescando storage engine is horizontally partitioned into fixed-size memory segments. Each segment is divided into chunks. Each chunk is further divided into slots, which is the smallest unit size which can be either empty or occupied by a record. Each memory segment has its own scan thread responsible for executing operations on its memory segment and reporting the results. For optimization purposes, each scan thread has hard processor affinity. Memory segments are allocated in a place in the main memory with minimal distance to its corresponding scan thread to minimize access cost. As illustrated in figure 2.5, each scan thread has an input queue which is a queue of the
Algorithm 1: Classic Scan Thread

Data: Segment seg
Data: Output queue iq / / input query and update queue
Data: Result queues aq

The asymptotic runtime of Classic Scan is $n^2 \log m$ for a set of $n$ queries and $m$ updates. The runtime is a function of the number of records accessed and the number of updates, but does not depend on the number of operations that are satisfied.

4. SCAN ALGORITHMS

4.2 Elevator Scan

A first improvement over Classic Scan is Elevator Scan. Even though Elevator Scan greatly improves upon the algorithm only activates operations at record 0, regardless of consistency even if some operations are writes.

Figure 2.4: Crescando Scan Cycle, (taken from [17])

Figure 2.5: Input and Output Queue for each Crescando Storage Engine (taken from [17])
2.2 Crescando

Operations that are to be executed on the corresponding memory segment and an output queue which contains the results of the operations. A controller manages the input and output queue, which places the operations that are to be performed by each scan thread in its corresponding input queue. Periodically the results generated are flushed to the controller’s result queue.[17]

Crescando storage engine supports recovery using write-ahead logging and checkpointing. Crescando uses write-ahead logging, such that Updates are appended to the log prior to being reported as a completed operation. A snapshot query is used to generate a checkpoint file. The snapshot query is periodically issued by a background thread, and it is an un-predicated query (matches everything). Recovery after a crash proceeds for each segment in parallel, where the snapshot segment is loaded from disk then the log is replayed such that unfinished operations can be executed.[17] Write-ahead logging and checkpoint technique is the only manner for recovery. Upon failure, the system has to be taken offline until the storage node recovers and the dataset is loaded into main memory, yielding the service unavailable for that duration of time.

Crescando keeps a small set of statistics that can be efficiently maintained by scanning the data. This includes the number of records, the number of null values for each attribute and the number of distinct values for each attribute. Linear and probabilistic counting is used to implement this, which occurs as a side-effect of a periodical statistics query. This is implemented using hash map. In this manner Crescando storage engine gather statistics about data-distribution and null-values.

2.2.2 Operations

Crescando operations are: Select, AggregateSelect, Insert, Delete and Update. Operations in Crescando can be classified into either read operations or write operations. Read operations are Select and AggregateSelect, while write operations are Insert, Delete and Update. Conjunctions can be expressed in the same query. An example of a query with conjunction is: Select all where \( x > 10 \) and \( x < 200 \). Disjunction cannot be expressed in the same query. Disjunctions have to broken down into more than one query. An example of a disjunction broken into two queries is: Select all where \( x < 10 \) or Select all where \( y < 10 \). Select, AggregateSelect, Insert, Delete and Update operations are enqueued to the engine using the following calls: csd_storage_select_enqueue, csd_storage_aggregate_select_enqueue, csd_storage_insert_enqueue, csd_storage_delete_enqueue and csd_storage_update_enqueue. These operations are executed by the engine in their arrival order. Each operation is given a unique identifier. Dequeued results can later be associated with their respective operation through the identifier. Results generated as a result of executing these operations can be retrieved by csd_storage_results_dequeue. A call to csd_storage_results_free must follow calling csd_storage_results_dequeue to free the resources used by the dequeued results. A single end-of-stream result is eventually returned for each operation, which indicates that no further results for the respective operation will be returned.

Data can be placed into Crescando storage engine either by bulk loading it or
by inserting it. Inserting data into Crescando engine is carried out by issuing \texttt{csd\_storage\_insert\_enqueue} as previously explained. Bulk loading data into Crescando engine takes place at a higher rate than inserting data into Crescando engine. This is mainly why it is preferred to bulk load large amount of data as opposed to inserting it. To bulk load data, \texttt{csd\_storage\_bulk\_load} is used. Crescando offers other ways to bulk load data, either from a file or through shared memory, however these are not discussed here. Later, in section 3.4 we show when \texttt{csd\_storage\_bulk\_load} and \texttt{csd\_storage\_insert\_enqueue} are used to remote copy data from one node to another.

Data is stored in Crescando in tables. Each table has a specified schema. Every table has scan threads associated with it. Scan threads of a specific table can be controlled using \texttt{csd\_storage\_table\_start\_scan\_threads} and \texttt{csd\_storage\_table\_stop\_scan\_threads}. The former call starts the scan threads for a specified table and the latter call stops the scan threads for a specified table. The following operations: Select, AggregateSelect, Insert, Delete and Update can only be enqueued to a certain table once it’s scan threads have been started using \texttt{csd\_storage\_table\_start\_scan\_threads}. Data can only be bulk loaded to a certain table prior to starting its scan threads. However once the table’s scan threads have been started, the data can only be placed within the engine by inserting it through \texttt{csd\_storage\_insert\_enqueue}.

Crescando statistics can be refreshed at any point of time, which forces Crescando to issue internally a statistical query. Refreshing Crescando’s statistics is carried out through a call to \texttt{csd\_storage\_optimizer\_statistics\_refresh}. Refreshing statistics can be enabled or disabled. Disabling them prevent periodical query from being issued. Refreshing statistics is enabled using \texttt{csd\_storage\_optimizer\_statistics\_enable} and disabled using \texttt{csd\_storage\_optimizer\_statistics\_disable}. In section 3.4.3 we highlight when Crescando’s statistics are disabled, and when they are enabled back again. The latency which refreshing statistics requires, depends on the size of data changed and the number of attributes per record as illustrated in the last section of this chapter.

### 2.2.3 Scan Cycle

As mentioned, Crescando partition its data horizontally into memory segments. Every memory segment has scan threads iterating over it in scan cycles. In these scan cycles, operations discussed in the previous section are fetched from the memory’s segment input queue, executed against the data in the memory segment. The results of these executions are placed in the memory’s segment output queue. Each scan cycle is composed of the following steps:

**Activation** Read operations are copied from the input queue into a read queue used internally by the scan thread, while write operations are copied from the input queue into a write queue used internally by the scan thread.

**Optimizing** Query plans are created at this stage. The system index the predicates of the query as explained in Crescando’s paper\cite{17} for optimization reasons.

**Execution** All data are scanned, slot by slot in each chunk where operations
2.2 Crescando

are applied. The execution depends on the operation type:

**Insert** A first fit policy is adopted here, where a record is inserted into the first empty slot encountered.

**Update** If a record matches the operation's predicate list, then the record's is updated according to the update operation.

**Delete** If a record matches the operation's predicate list, then that slot is set as empty.

**Select** If a record matches the operation's predicate list, then the record is included in the result.

**Deactivation** Operations are deactivated, and a flag is added to the result queue to indicate that end-of-results has been reached.

The scan cycle is where the core operations of Crescando storage engine are performed; thereby this phase is very performance critical.

### 2.2.4 Basic performance numbers

Basic performance numbers of Crescando's operations are presented here as they influenced the design of the partial live migration system. These operations are the operations responsible for retrieving data from storage engine and placing data to storage engine. They are the core operations of remote copying data between two storage nodes. The basic performance numbers presented here are extracted from the experiments presented in Chapter 5.

In these experiments, the rate of Select, BulkLoad and Insert is the metric measured along with varying number of cores, record size and number of attributes. These experiments will investigate these operations scalability along with adding cores. Moreover they will provide insight regarding the relation between these operations' rate and the schema used. In another experiment, the latency required to refresh statistics is analyzed while changing size of data added and number of attributes per record. This will provide more understanding of the effect of refreshing statistics on remote copying.

For a realistic schema (Amadeus Itinerary schema presented in the Appendix), BulkLoad and Insert do not scale up with increasing number of cores. BulkLoad has a rate around 375Mbyte/s while Insert has a rate around 295Mbyte/s. Select has a rate of 125Mbyte/s and scales up to 350Mbyte/s. Remote copying uses Select to retrieve the data from the sender node and bulkLoad (or Insert) to place the data in the receiver node. Using one core, remote copying performs at a rate of 125Mbyte/s. Remote copying scales up to 315Mbyte/s with bulk loading on the receiver end (and 287Mbyte/s with inserting on the receiver end). With bulk loading on the receiver end, 1 Gbyte of itinerary records takes 8.2 seconds to be remote copied between two nodes using one core on each node and 3.2 seconds using 3 cores on each node. Using 3 cores on each node, remote copying can copy up to 18 Gbytes of itinerary records in less than one minute.

The schema used has a significant effect on the operations rate. This ef-
fect has been investigated by changing the number of attributes and record sizes to understand how BulkLoad, Insert and Select performance change accordingly. Bulk load peak performance reached in the experiments was over 500Mbyte/s. Insert and Select peak performance was over 350Mbyte/s. The lowest rate reached by BulkLoad, Insert and Select is 125Mbyte/s, 100Mbyte/s and 22Mbyte/s respectively.

Refreshing the statistics has a latency that has a linear relationship to the data size changed. The data size changed refers to the amount of data placed in the engine or removed from the engine. The number of attributes per record in the data exhibits a similar relationship with the latency needed to refresh statistics. Using the itinerary records, around 9 seconds is needed to refresh the statistics after 1GB have been bulk loaded or inserted on the receiver end using remote copying. However the latency required to refresh statistics in real situation is lower. Due to the fact that refreshing statistics is implemented using hash maps, the distribution of data plays an important role. The data used in the experiments has a random distribution, thereby lookups in the hash map trigger more cache misses, resulting in higher latency.

2.3 E-Cast

The distribution layer of Crescando/RB consists of E-Cast and Rubberband. E-Cast and Rubberband are used to support elasticity in Crescando/RB system. E-Cast is a causal-total-order multicast protocol[16] E-Cast has been developed to provide strong delivery guarantees across many groups. E-Cast supports frequent membership changes. The key feature to E-Cast is the ability to determine the destination process of a message exclusively by analyzing the past history of messages in the system rather than relying on a membership protocol.[15] This is particularly challenging due to the dynamic nature of data objects mapping to processes, as these mapping change over time. The whole Crescando/RB system must provide linearizable consistency guarantees; the system must behave as if there was a single copy of each data object and all Updates are executed in some strict serial order[9]. E-Cast ensures that a distributed agreement takes place between the nodes involved. This is known as consensus and is implemented in Crescando/RB using E-Cast protocol.

E-Cast ensures that messages related to certain data objects reach all the processes holding that data object such that the processes receive the messages in their order sent, even if data repartitioning occurs. Processes can join or leave the pool of processes, data can be repartitioned however E-Cast will guarantee delivery in the right order mentioned.

Three types of E-Cast application processes exist: client, storage and super processes. Client submits read and write messages which the router forwards to the appropriate storage processes while super processes actively maintain the mapping of keys to storage processes.

Crescando/RB has different read types. These read types are differentiated according to the isolation level guaranteed by the read. Linearizable read ensures
linearizability, while snapshot read is guaranteed to see a consistent snapshot between two writes in the global sequence of Updates. On the other hand, basic read is a read which offers no formal guarantees and is a best effort operation. E-Cast protocol routes both linearizable read and snapshot read. Basic read however is forwarded to its corresponding Crescando storage node right away. This is the reason why basic read can scale higher than linearizable read and snapshot read. Likely, write operations go through E-Cast and thereby are as expensive as linearizable reads.

2.4 Rubberband

As previously mentioned, Rubberband exists in the distribution layer of Crescando/RB. Rubberband implements a well-known replication scheme known as successor-list replication\cite{14}. Successor-list replication involves modeling our storage processes as a ring, such that each storage process is assigned a unique key within the ring. A process $p$ with key $k$ holds the primary replica of all objects whose key is greater than or equal to $k$ and less than $\dot{k}$ where $\dot{k}$ is the key of $p$ successor. A replication factor $f$ convey that $f$ replicas exist for each data object. $f - 1$ successors of process $p$ hold a back-up for every object process $p$ carries a primary copy of.$^1$

In figure 2.6 an example of a successor-list replication is illustrated, where all the data keys fall in the range $[0, 4000]$. Four nodes exist in the ring. The first, second, third and fourth node have the keys 0, 1000, 2000 and 3000 respectively.
The system shows a successor-list replication scheme with a replication factor $f$ of 3. Four objects resides in the system, $A$, $B$, $C$ and $D$ with keys 10, 1500, 2500 and 3001 respectively. The first node carries the primary replica of $A$ since its key 10 falls between 0, the first node’s key and 1000, the second node’s key. Similarly the second node, third node and fourth node carry $B$, $C$ and $D$ respectively. Due to the existence of replication factor of 3, $A$ is also carried by the second and third node while $B$ is carried by the third and fourth node, $C$ is carried by the fourth and first node and $D$ is carried by the first and second node. Each object is replicated three times in the system, once as a primary replica and two times as a secondary replica. This is an example of successor-list replication implemented in Crescando/RB. This logic is implemented by Rubberband. Rubberband adjusts the system as processes are added or removed in order to maintain compliance with successor-list replication scheme. To achieve that, data is copied or transferred between nodes using the migration system developed in this thesis.

Rubberband is an application of E-Cast. Rubberband uses E-Cast to route messages to their storage location destination. E-Cast and Rubberband framework is used with Crescando storage engine; however Rubberband is independent of the storage engine implementation and can be deployed with any storage engine. This portability is achieved by having a function called Range Fun, which is deployed in a module in E-Cast such that this function uses Rubberband to perform its functionality. Range Fun extracts the destination key for each write message.[16]

Expand, Contract, Replace and Rebalance are Rubberband configuration messages. Expand expands the ring of processes by adding one process to the ring with a unique key. Contract contracts the ring of processes by removing one process from the ring. Replace replaces a storage process in the ring with an idle process. Rebalance updates a key that is assigned to one storage process in the ring. Each configuration message contains a list of instructions, informing the storage processes the operations they need to perform.[16] As these operations are executed by the storage engine, partial live migration system is employed to shuffle data around. Rubberband ensures that the logic of these instructions shuffle data around such that the system is in compliance with successor-list replication scheme.

In Crescando/RB, router processes forward requests to the appropriate storage processes while super processes actively maintain the mapping of keys to storage processes. Bootstrapping the system proceed by bringing up the first router and the first super process. The first super process is given the first router ID. When a storage process starts, it E-Casts a join message and enters the pool of idle processes. A part message is E-Casted to leave the pool of idle processes. Subsequently super processes maintain the successor-list replication scheme in the system through sending the messages described above: Expand, Contract, Replace and Rebalance. Figure 2.7 displays a message which is routed to their appropriate storage processes through the router processes in Crescando/RB system. The super processes in the figure maintain the object-storage processes mapping.
2.4 Rubberband

Figure 2.7: Rubberband Message Routing, taken from [16]
Chapter 3

Architecture and Design

Partial live migration system developed in this thesis lies in the system storage layer of Crescando/RB. The migration system is built on top of Crescando. The design of the system follows a black-box approach. The system is designed using Crescando external interface, without changing the storage engine implementation. The performance of Crescando in retrieving data from the engine and placing data in the engine is one of the factors for this design decision. Select is used to retrieve data from the engine and BulkLoad (or Insert) is used to place data into the engine. In Crescando/RB system, the distribution layer (E-Cast and Rubberband) instructs nodes to shuffle data around to ensure the correct partitioning and replication with the latest configuration of the system. This logic is carried out by Rubberband. This is the manner in which partial live migration system is invoked in Crescando/RB system.

3.1 Design space and related work

Two main strategies exist for implementing live migration functionality in Crescando: White-box approach and Black-box approach. The former refers to modifying the internals of the Crescando engine to provide the functionality, while the latter employs the external interface of the Crescando engine to carry out migration. Each approach possesses its own benefits and limitations. The difference lies mainly in data extraction and placement. However transferring data from one node to another is common to both approaches.

"White-box" refers to changing or using the internals of the engine as opposed to the external user interface to implement the functionality of data extraction and placement. While this might be optimal in certain cases, it has its own drawbacks in other cases. A possible idea in this approach is to develop a system to extract data from the engine and place data into the engine. While extracting and placing data might be implemented more efficiently in this approach, this approach will interfere with the functionality of the engine. This methodology must use the data in a mutually exclusive manner along with the engine. Synchronization will complicate both the storage engine and the migra-
A hybrid approach exists, which is based on the idea used by HYPER[4]. The running Crescando process is forked leaving two instances of Crescando in memory. The original instance which handles requests is left as-is, and continues to serve requests. This exploits hardware replication mechanisms, where the hardware copy original Crescando’s data on update. The data cease to be a shared resource in this case. This approach can be used on the sender side, where extracting data can then proceed efficiently by reading from memory. Moreover in order to implement partial live migration, a system must be built that selectively extracts data from memory. In this approach, challenges lie in ensuring a consistent copy of Crescando takes place, due to the fact that Crescando is multi-threaded and fork has issues with that. The additional memory required by this approach depends on the update load (as data is copied on update). In worst case, the system must have memory capacity big enough to duplicate the original Crescando instance.

On demand migration is a migration technique outlined in [7] where a new processing node added to the pool serving a certain workload immediately have transactions routed to it and fetches data from the old node as needed. The main advantage of this approach is that it minimizes service interruption; however transactions that update data which have not yet migrated incur expensive procedure of first fetching all the data that is affected by the update from the remote source node. Expensive synchronization is needed between the source and the destination node moreover failure recovery is complicated in this model. In addition to that, this technique does not support partial migration. This technique does not directly apply to Crescando/RB, as it assumes that no replicas of data objects exists, however the core idea to transfer data from one node to another applies. In heavy update situations, this approach does not offer advantages. The main difference between this approach and the migration system developed in this thesis is that the migration system designed in this thesis supports partial live migration and that data is transferred to the new node prior to operating on it. These design decisions used in the thesis contribute to a much simpler failure recovery model as opposed to on demand migration recovery model.

The "Iterative copy" technique has been proposed as a method for live database
3.1 Design space and related work

migration for elasticity in a multitenant database for cloud platforms in [8]. The proposed technique consists of the following phases: pre-migration phase, migration phase and post-migration phase. In pre-migration phase the ownership of the data to be migrated belongs to the old node, while in post-migration phase the ownership of the data belongs to the new node. The migration phase itself consists of the following phases: begin migration phase, iterative copy phase and atomic handover phase. In begin migration phase, a snapshot of the database is taken and transferred to the new node, during which the old node serves transactions for the data to be migrated normally while keeping a log of the updated data. In an iterative copy phase, as the old node has served transactions for the data being migrated, the new node state of the data will lag behind that of the old node, thereby the changes are transferred to the new node iteratively, where in iteration $i$ the log of changes since iteration $i - 1$ are transferred to the new storage node (while the old node continues to serve transactions for that data normally). This phase terminates when the amount of state transferred in subsequent iterations converges. In the next phase, atomic handover, the ownership is transferred to the new node, after which post-migration phase is reached. The illustrated technique does indeed offer a very interesting way to minimize service interruption even if more data is transferred between the nodes as a result of migration. A problem in applying this technique directly to Crescando/RB system, is that the system assumes that migration is fully completed prior to executing subsequent operations operating on that data, which are then routed to the new node. If this behavior is changed, then the technique can be applied to Crescando/RB system. This technique as opposed to the one developed in the thesis is particularly efficient when update workload is light. However in a heavy update scenario this technique does not offer a major advantage, as the state changes will be dense between iteration and its subsequent iteration requiring a large state transfer. This will result in delaying the migration process. Moreover this technique, as opposed to the one developed in this thesis, does not support partial migration.

The problem of migration has also been studied in virtualization literature. A technique similar to iterative copy is proposed in [6], where the memory of one virtual machine is migrated to another in three phases: push phase, stop-and-copy phase and pull phase. During the first phase, the source virtual machine continues to run while certain pages are pushed across the network to the destination. Any modified page must be re-sent for consistency reasons. Stop-and-copy phase stops the source virtual machine and copy pages across to the destination virtual machine, after which the destination virtual machine can be started. Finally, the pull phase takes place, such that if a page has not been copied yet, the page is faulted and pulled across the network from the source virtual machine. An improvement have been proposed to that technique in [11] where a log is created at the source node, transferred over the network to the destination node and then replayed at the destination node. These share the same idea with iterative copy technique, where in databases, these two techniques are analogous to the scenario where the source node holds ownership of the data while a snapshot of its data is transferred, followed by iterating over any changes that occured during the transfer at the source node and transferring it to the destination node (up to a certain limit, that is when changes converge from one iteration to the next) such that both contain the same data
and service interruption is minimized. While this thesis is focused on partial live migration for elastic scalability purposes, a whole survey on live reorganization of databases is provided in the following survey [13].

3.2 Methodology Used

Black-box approach is used in this thesis to implement partial live migration system on top of Crescando. The partial live migration system will use Crescando operations (Select, BulkLoad and Insert) to extract data from the engine and place data into the engine. These operations are efficient enough to implement such a system and one can benefit from separation of concerns.

Select is used to extract data from Crescando while either BulkLoad or Insert is used to place data to Crescando. BulkLoad is used when migration is the first process for a node; otherwise Insert must be used to place data. This is due to the fact that BulkLoad can only be used prior to starting scan threads, and scan threads have to be started prior to executing other operations. To migrate data from one node to another, Select is issued on the sender node where data is to be copied from. This data is then transferred over the network to the receiver node. BulkLoad or Insert is used on the receiver node to place the data.

Select is used to retrieve data from the engine. Disjunctions are expressed as one or more Select queries. An example of that is: Select all where $x < 10$ or Select all where $y > 2000$. In each Select query, data is selected according to one of the following:

- **Attributes value**: Select can select data according to the attributes value. Any record whose attribute value fall within a user specified range will be included in the result set. This range has an inclusive start value and an exclusive end value. An example is Select all where $x \geq 10$ and $x < 1000$. This query will select all records whose $x$ attribute fall in the range $[10, 1000]$.

- **Attributes hashed value**: Select can select data according to the attributes hashed value. This is possible as built-in predicates can be given by the user to select. Select then evaluates this predicate against each record and includes the record in the result set if the predicate evaluates to true for that record. To select data according to their hashed value, the predicate would feed the record into a hash function and evaluates to true if the record hash value is within a user specified hash range. An example is Select all where $hash(x) \geq 10$ and $x < 1000$. This query will select all records whose $hash(x)$ value fall in the range $[10, 1000]$.

As the external interface and its capabilities are used to empower and build the partial live migration system, they also set their limitations. The migration rate between one node and another is bottlenecked by the lowest of the following: rate of retrieving data from Crescando storage engine, transferring data over the network from the sender node to the receiver node, rate of placing data in Crescando storage engine. Select is used to retrieve data from the engine, while BulkLoad (or Insert) is used to place data into the engine. Thereby these
operations are very significant for the partial live migration system. These operations performance has been studied in the Chapter 5. Future changes in their performance will affect the migration system accordingly.

### 3.3 Architecture

Partial live migration system is used to shuffle data around, as instructed by Rubberband in Crescando/RB system, to maintain the system in compliance with successor-list replication. As nodes join or leave the system, partial live migration takes place in order to achieve this compliance. Migration within Crescando/RB is illustrated in figure 3.1. Rubberband is responsible for the logic of instructing the nodes to carry certain operations (RemoteSend, RemoteReceive) in order to invoke remote copying between them. The super node in this figure using Rubberband and E-Cast instructs one node to invoke RemoteSend and another node to invoke RemoteReceive. The two nodes then perform migration. This takes place on the Partial Live Migration layer in Crescando/RB protocol stack as illustrated in figure 2.1. Nodes communicate directly with each other in this migration process.

Partial live migration system consists of four primitive operations: RemoteSend, RemoteReceive, PartitionDelete and RemoteFail. These four operations give the functionality of copying or moving a user specified portion of data from one Crescando node to another. These operations are enqueued to the system in a manner similar to other operations (Select, BulkLoad and Insert).
3.4 Design

3.4.1 Overview

As previously mentioned, the functionality of copying or moving a user specified portion of data from one Crescando node to another is provided using four primitive operations: RemoteSend, RemoteReceive, PartitionDelete and RemoteFail. RemoteSend instructs the node executing it to act as a sender node and specifies the other end (receiver node) to send the data to. RemoteReceive instructs the node executing it to act as a receiver node and specifies the other end (sender node) to receive the data from. Remote copying takes place between the sender node and the receiver node when RemoteSend is executed on a node and RemoteReceive is executed on its corresponding receiver node. Moreover every RemoteSend is given an identification number, such that it will send data to the receiver node only if the receiver node is executing a RemoteReceive with the same identification number. In this manner, every remote copy transaction is labeled with a certain identification number and if multiple remote copies are instructed between two nodes they can be differentiated. As an example illustrated in figure 3.2, remote copying occurs from node x to node y, if node x is instructed to RemoteSend to node y with identification A and node y is instructed to RemoteReceive from node x with identification A.

The basic procedure of remote copying from one node to another has been illustrated. The data copied in this procedure is specified by the user. This data is expressed as a disjunction of range predicates on the partitioning key. This can either be a range predicate on attributes value or range predicate on attributes hash value. As an example the user might instruct RemoteSend to copy all records that match: 10 <= x < 1000 where x is an attribute. Moreover, the user can divide the data based on the attributes hash values. As an example, a RemoteSend might instruct the node to send to the receiver node all records whose x hash value match: 10 <= hash(x) <1000 where hash(x) returns x hash value.

In remote copying process, data is copied from one node to another. In order to move data from one node to another as opposed to copying it, data is copied at first as explained using remote copying and then deleted from the sender node. A primitive operation, PartitionDelete, has been created to delete a specified portion of data from the sender node. The data is specified in the same manner.
3.4 Design

as it is specified with RemoteSend. As an example, to move data from node x to node y, one proceed as the in the example above, node x is instructed to RemoteSend to node y with identification A and node y is instructed to RemoteReceive from node x with identification A. Following that, node x is instructed to PartitionDelete the same portion of data it was instructed to RemoteSend. This will copy the data first, and then delete it from the sender node.

RemoteFail is an operation that can be called by the user at any point of time to abort RemoteSend or RemoteReceive which is currently taking place or which is enqueued to be executed later. RemoteFail is created to deal with hosts failing. When a host fails, all the nodes that have RemoteSends or RemoteReceives operations dealing with this host will abort these operations upon receiving RemoteFail with the host information. Crescando/RB system upon detecting that a host failed will issue RemoteFail to all nodes interacting with that host.

3.4.2 Request Queue

Each Crescando node has a queue of requests which it executes in arrival order. These requests can be one of the following operations: Select, AggregateSelect, Insert, Update, Delete, BulkLoad, RemoteSend, RemoteReceive and PartitionDelete. RemoteSend is enqueued to the sender node while RemoteReceive is enqueued to the receiver node. Each RemoteSend must correspond to a certain RemoteReceive to perform remote copying. A remote copy transaction is a unique transaction between two endpoints identified by a unique number. This is the same number given to the sender node by RemoteSend and to the receiver node by RemoteReceive. In this manner, the system is able to associate a certain RemoteSend with a certain RemoteReceive even if multiple remote copy transactions exist between the two nodes.

The replication system buffers all incoming input operations to the input queue. The system applies transformation for every input operation in the input queue to create an output operation. In this transformation process, some operations are transformed to one or more different operations, while others are copied as is. These are placed in the output queue in the same order of the input queue (arrival order). The output queue contains only simple operations which Crescando understands: Select, AggregateSelect, Insert, Update, Delete, and BulkLoad. The operations that are transformed to one or more different operations are the following:

- RemoteSend: RemoteSend is transformed into one or more Select operations. If disjunctions are involved, more than one Select query are required to select the data desired. When the data selected is returned back by the engine, it is sent over the network to the receiver node.

- RemoteReceive: RemoteReceive is transformed into one or more BulkLoad (or Insert) operations. BulkLoad is used if no prior operations have been enqueued (and scan threads are not running) otherwise Insert is used. BulkLoad has a slightly higher insert rate than Insert. This is used to place the data receiver over the network from the sender node to the
storage engine.

- **PartitionDelete**: It is transformed into one or more Delete operations. If disjunctions are involved, more than one Delete operations are required to delete the data.

The other operations (Select, AggregateSelect, Insert, Update, Delete, and BulkLoad) are copied as is. RemoteFail have been omitted as RemoteFail is not enqueued in the same manner, however it is executed right away when it is enqueued to abort all RemoteSends and all RemoteReceive interacting with a certain host. All output operations are enqueued to Crescando engine in the same order of input operations (arrival order). The partial live migration system controls when each output operation is enqueued to Crescando engine. Some operations are enqueued in batches, while others are enqueued one at a time. Some operations require blocking subsequent operations until they run to completion or until they are enqueued to the engine. The following is how the partial live migration system handles the remote copying operations:

- **RemoteSend**: RemoteSend operation needs only to block until it is enqueued to the engine before any subsequent write operations are handled.

- **RemoteReceive**: RemoteReceive operation needs to block until it is fully completed by the engine before any subsequent operations are handled.

- **PartitionDelete**: PartitionDelete operation needs to block until it is enqueued to the engine before any subsequent operations are handled.

These all agree with the correctness properties presented in section 3.5.2.

### 3.4.3 Remote copying operations

Partial live migration functionality is provided using the following operations:

- **RemoteSend**: instructs sender node to send data to receiver node.

- **RemoteReceive**: instructs receiver node to receive data from sender node.

- **PartitionDelete**: instructs the node to delete a specified portion of data.

- **RemoteFail**: instructs the node to abort all RemoteSends and all RemoteReceives that interact with a certain host.

The steps involved in RemoteSend operation will be illustrated here. As presented in the state diagram in figure 3.3, the following are the steps of RemoteSend operation:

1. **Establish connection**: The sender node attempts to establish connection to the receiver node.

2. **Satisfy agreement**: If connection is successfully established, the sender node will try to satisfy the agreement property. The agreement property refers to matching the RemoteSend identification number on the sender node with the same identification number on the receiver node executing RemoteReceive.
Connection Established
Agreement Satisfied
Data queried
Data sent successfully
Remote Send Success
Remote Send Fail
Sending / Receiving Failed
Remote Fail
Wait for receiver acknowledgement

Figure 3.3: RemoteSend State Machine
3. Query data: Once the agreement property is satisfied, the data portion to be migrated will be queried by issuing one or more Select queries.

4. Send data: The result of the queries issued in the previous step will be sent over the network to the receiver node.

5. Wait for receiver acknowledgment: When all data has been successfully transferred to the receiver node, the sender node waits for an acknowledgment from the receiver node. Once the acknowledgment has been successfully received, the RemoteSend operation is finished successfully.

RemoteSend operation can finish unsuccessfully. If RemoteFail operation is issued at any point of time with the host RemoteSend is interacting with (receiver node), then RemoteSend operation finishes unsuccessfully. Moreover if any transfer error occurs during "Send Data" or "Wait for receiver acknowledgment" occurs, then RemoteFail finishes unsuccessfully. In "Establish connection" step, RemoteSend will attempt to establish connection to the receiver node. If connection cannot be established, RemoteSend will continue on retrying until either a connection is established successfully or RemoteFail operation is issued to abort the RemoteSend operation. In "Satisfy agreement" step, if RemoteSend fails to satisfy the agreement property (no RemoteReceive operation exists on the receiver node with the same identification number), then RemoteSend will block the sender node until the agreement property is satisfied or RemoteFail operation is issued to abort the RemoteSend operation.
RemoteReceive operation is very similar in its states to RemoteSend operation. The following are the steps of RemoteReceive operation as illustrated in the state diagram in figure 3.4:

1. Establish Connection: The receiver node attempts to establish connection to the sender node.

2. Satisfy agreement: If connection is successfully established, the receiver node will try to satisfy the agreement property.

3. Receive data: Once the agreement property is satisfied, the receiver node will receive data over the network from the sender node.

4. Bulk load / insert data: As data is received, the receiver node will place the data in its storage engine by either bulk loading it or inserting it.

5. Send Acknowledgment: Once all the data have been received and placed in the storage engine, the receiver node will send an acknowledgment to the sender node. If this acknowledgment is successfully sent to the sender node, the RemoteReceive operation is finished successfully.

Similar to RemoteSend operation, RemoteReceive operation can finish unsuccessfully if RemoteFail operation is issued at any point of time with the host RemoteReceive is interacting with (sender node) or if any transfer error occurs during "Receive data" or "Send Acknowledgment". If RemoteReceive operation finishes unsuccessfully, the receiver node will delete all the contents of its table it is remote copying to. In similar fashion to RemoteSend operation, RemoteReceive have similar behavior in "Establish connection" step and "Satisfy agreement step" if they were not fulfilled. Retrials will be made in the former and blockage will occur in the latter. As previously mentioned, Crescando has keeps a small set of statistics, which is maintained through issuing a periodical statistical query. Basic performance numbers including the latency required to refresh Crescando statistics has been presented in section 2.2.4. To avoid any side effects, Crescando statistics are disabled as soon as RemoteReceive operation is called by calling csd_storage_optimizer_statistics_disable and then enabled after the operation runs to completion using a call to csd_storage_optimizer_statistics_enable.

PartitionDelete operation consists of issuing one or more Delete operations. As soon as end-of-stream result is returned by the storage engine for all these issued Delete operations, PartitionDelete is finished successfully. If any of the Delete operation fails, then PartitionDelete is finished unsuccessfully. If any of the Delete operation fails, then PartitionDelete is finished unsuccessfully.

RemoteFail has no states. The operation immediately aborts all RemoteSend and RemoteReceive operations that interact with a certain host.

### 3.4.4 Inconsistent Select

Inconsistent Selects in Crescando has no formal guarantees. While executing an inconsistent Select, Crescando storage engine does not guarantee that a write operation enqueued will be executed following that inconsistent read. In this case it does not guarantee that write follows that read. Due to that stated
fact, the partial live migration system disallows enqueuing any remote copying operations during which the engine is executing any inconsistent select, to avoid having the result of any remote copy operation visible to the inconsistent read operation. This agrees with the correctness properties presented in section 3.5.2.
3.5 Correctness

In this section we present a formal model of the partial live migration system, followed by that we present the correctness properties that need to hold to have a correct partial live migration system. A formal proof leaves no area to different interpretations and removes any ambiguities. This formal proof is the essence of proving the correctness of a migration system.

The partial live migration system developed in this thesis has a couple of optimizations implemented for efficiency purposes. This section can be used to proof that the system developed in this thesis along with these optimizations, are correct.

3.5.1 Formal Model

Users enqueue operations to Crescando/RB system. These input operations are transformed by the partial live migration system developed in this thesis to output operations. These output operations are then enqueued to Crescando engine. Here we show the set of input and output operations to the partial live migration system. The set of input operations to the system is \( \text{Op}_i \) where \( \text{Op}_i \subseteq A \cup B \) while the set of output operations of the system is \( \text{Op}_o \) where \( \text{Op}_o \subseteq B \) such that:

\[
\begin{align*}
A &= \text{RemoteSends} \cup \text{RemoteReceives} \cup \text{PartitionDeletes}, \\
B &= \text{Inserts} \cup \text{Updates} \cup \text{Deletes} \cup \text{R}_i \cup \text{R}_c, \\
\text{R}_i &= \text{InconsistentSelects} \cup \text{InconsistentAggregateSelects} \\
\text{R}_c &= \text{ConsistentSelects} \cup \text{ConsistentAggregateSelects}
\end{align*}
\]

The sets of input and output operations may overlap.

An input event occurs as the partial live migration system receives an input operation. Upon receiving input operations, the system transforms these input operations to output operations. Producing an output operation is an output event.

There exists a function \( f \) such that \( f \) maps input events to input operations and output events to output operations; \( f(e_i) \Rightarrow o_i \) where \( e_i \in E_i \cap o_i \in \text{Op}_i \) and \( f(e_o) \Rightarrow o_o \) where \( e_o \in E_o \cap o_o \in \text{Op}_o \). There exists a function \( h \) such that \( h \) maps input operations to input events and output operations to output events; \( h(o_i) \Rightarrow e_i \) where \( o_i \in \text{Op}_i \cap e_i \in E_i \) and \( h(o_o) \Rightarrow e_o \) where \( o_o \in \text{Op}_o \cap e_o \in E_o \). A function \( g \) exists such that \( g \) maps input operations to output operations; \( g(o_i) \Rightarrow o_o \) where \( o_i \in \text{Op}_i \cap o_o \in \text{Op}_o \).

The sequence of input events is defined as \( \mathbb{I} = (E_i, \prec_i) \) where \( \prec_i \) is a strict total order relation over the input events, \( E_i \). The sequence of output events is defined as \( \mathbb{O} = (E_o, \prec_o) \) where \( \prec_o \) is a strict total order relation over the output events, \( E_o \). \( \mathbb{I} \) and \( \mathbb{O} \) are disjoint; i.e. \( \mathbb{I} \cap \mathbb{O} = \emptyset \). Given the assumption that input events are transformed to output events in a single thread, the sequence of all events is defined as \( \mathbb{E} = (E_i \cup E_o, \prec_e) \) where \( \prec_e \) is an extension of \( \prec_i \) and \( \prec_o \) such that:

\[
\forall e, e' \in E_i : e \prec_i e' \implies e \prec_e e'
\]
∀_e,e_o ∈ E_o : \dot{e} ≺_o e_o = \Rightarrow \dot{e} ≺ e_o

### 3.5.2 Correctness properties

A system is a correct solution of a partial live migration if and only if for any sequence of input events, the correctness properties mentioned here hold.

As previously mentioned, the partial live migration system transforms input operations to output operations. A system that ignores transforming an input event to an output event is an incorrect system. Partial live migration system must produce an output event for every input event:

\[ \forall e_i ∈ E_i \forall o_o ∈ g(f(i)) \exists e_o ∈ E_o : f(e_o) = o_o \]

Input events can involve either a write operation or a read operation. If a certain order exists between input events involving write operations, then the partial live migration system must produce output events for these input events with the same order. A system that produces output events with a different order is an incorrect system. The order between input events involving write operations must be the identical to the order of the output events produced from these input events:

\[ \forall \dot{i},\ddot{i} ∈ E_i : \dot{i} \neq \ddot{i} \cap \dot{i} ≺ \ddot{i} = \Rightarrow \forall o_\dot{i} ∈ f(\dot{i}) \forall o_\ddot{i} ∈ f(\ddot{i}) \forall e_o ∈ g(o_\dot{i}) \forall e_o ∈ h(o_\dot{i}) \dot{e} ≺_o e_o \]

Inconsistent Selects in Crescando has no formal guarantees. While executing an inconsistent Select, Crescando storage engine does not guarantee that a write operation enqueued will be executed following the read. Due to that stated fact, all operations from set A (remote copying operations) must not be enqueued to the engine during inconsistent Select execution to avoid having its result visible to an inconsistent read operation. No event from set A is allowed to be enqueued into the engine from the point an inconsistent Select was enqueued into the engine until end-of-stream result for that Select has been received from the engine:

\[ h(g(f(r ∈ Opi))) : r ∈ R_i = \langle result_{begin}, result_{end} \rangle \] where \( result_{begin}, result_{end} ∈ E_o \cap result_{begin} ≺_o result_{end} \]

\[ \forall r ∈ Op_i \forall a ∈ Opi : r ∈ R_i \cap a ∈ A \Rightarrow \not\exists_{e_i ∈ h(a)} result_{begin} ≺_o e_i ≺_o result_{end} \]

The previous three properties are the correctness properties of a partial live migration system built on top of Crescando storage engine. The system must produce an output event for every input event. The order between input events involving write operations must be the identical to the order of the output events produced from these input events. The system must not allow any event from set A to be enqueued to the storage engine when it is executing an inconsistent Select operation.
Chapter 4

Implementation Details

This chapter elaborates on the implementation details of the partial live migration system. The partial live migration system has been built on top of Crescando. The partial live migration has been developed using black-box approach, using the external interface of Crescando without changing its internal implementation. In Crescando/RB, Rubberband instructs the nodes within the system to carry out migration as needed using this system developed here. When migration takes place, nodes communicate directly with each others.

4.1 Interface

The partial live migration system has been implemented in C++ and integrated on top of Crescando. The interface however is written in Erlang. Operations are issued through Erlang and reported back to Erlang in the form of callback functions. Partial live migration system itself was fully written in C++ for efficiency purposes. A similar system written in Erlang will be much slower as will be shown in chapter 5.

Partial live migration system consists of four primitive operations. These operations are:

- RemoteSend
- RemoteReceive
- PartitionDelete
- RemoteFail

These operations operate on data within specified table. The first three operations are operations that are enqueued similar to other operations (Select, Insert, etc...) while the last operation is an operation that is executed immediately. The approach used to implement these operations is a black-box approach, where these operations have been implemented using the external interface of Crescando engine without changing its internal implementation.
RemoteSend operation is used to instruct a sender node to send certain data to a receiver node. The RemoteSend operation expects the following:

- **Table**: Table which RemoteSend will extract the data from.
- **List of predicates**: Each item in the predicate list represents a Select query. Each item in the predicate list contains the following:
  - **Attribute ID**
  - **Predicate operator**: can be either a built-in operator or a user-defined function.
  - **Range**: range is defined by an inclusive low bound and exclusive high bound.

The table is queried for each predicate such that the predicate operator on the specified attribute ID has a result that is between the low bound inclusive and high bound exclusive. The data can be selected according to its value using a built-in operator. Moreover, the data can also be selected according to its hash value using a user-defined function. Each record is fed to the user-defined function as an input, and if the function returns true, then the record will be included in the result set, otherwise the record is dismissed from the result set. This allows the functionality of choosing data according to their hash value, where the user-defined function would feed the record into a hash function, and return true if it is within the desired specified range. Low bound can be open, as well as the high bound. The former queries for items less than high bound exclusively and the latter queries for items more than low bound inclusively. If both were open, then an un-predicated query is issued (one that matches everything).

- **Endpoint**: The endpoint consists of a host or an IP address and a port number. This endpoint represents the receiver node, where the data will be sent to.

- **Identification number**: An identifier is associated with each RemoteSend operation. Remote copying will only take place if the receiver node is executing a RemoteReceive operation with the same identification number.

RemoteSend operation ends with a status of either success or failure, which is reported to the caller.

RemoteReceive operation is used to instruct a receiver node to receive data from the sender node. RemoteReceive operation expects the following:

- **Endpoint**: The sender node where data will be received from.

- **Identification number**: An identifier is associated with each RemoteReceive operation. Remote copying will only take place if the sender node is executing a RemoteSend operation with the same identification number.

RemoteReceive operation ends with a status of either success or failure, which is reported to the caller.

PartitionDelete is an operation that instructs a node to delete a specified portion
of data. PartitionDelete expects a list of predicates. This list of predicates has a format similar to the one in RemoteSend operation. Typically the same list of predicates is given to RemoteSend operation then later to PartitionDelete operation to move data instead of copy it. PartitionDelete operation ends with a status of either success or failure, which is reported to the caller.

RemoteFail is used to abort all RemoteSends and RemoteReceives interacting with a certain host. RemoteFail as opposed to RemoteSend, RemoteReceive and PartitionDelete is invoked immediately when issued. RemoteFail expects an endpoint. This endpoint contains the same elements as the one in RemoteSend operation. RemoteFail will abort all operations that involve that endpoint. This includes RemoteSends to that endpoint or RemoteReceives from that endpoint, regardless of whether this operation is currently taking place or is enqueued to take place. This operation is used to deal with hosts failing, such that when a host fails RemoteFail is issued to all other nodes that interact with that host in order to abort all RemoteSends and RemoteReceives interacting with that host. All operations (RemoteSends and RemoteReceives) which have been aborted as a result of RemoteFail will report failure to the caller.

4.2 Overview

RemoteSend is implemented by issuing one or more Select. The number of Select issued is equal to the number of items in the list of predicates supplied to the RemoteSend. This follows since in a single Select issued to Crescando, disjunctions are not allowed within the same query. Every item in the list of predicates represent a disjunction, thereby the number of Select queries are equal to the number of items in the list of predicates. If there are no items in the list of predicates, then an un-predicated query is issued. Prior to enqueueing these Selects to Crescando engine, a connection has to be established with the receiver node and the agreement property has to be satisfied. Following that data is sent to the receiver node. The protocol used in communication with the receiver node is a protocol created in this thesis, called Remote Transfer Protocol. This protocol will be elaborated later in this section 4.4.

RemoteReceive is implemented by issuing one or more BulkLoad or Insert operations. BulkLoad is typically faster than Insert and can only be used if the scan threads of Crescando have not yet been started. Scan threads must be started prior to executing any operation other than BulkLoad (Select, AggregateSelect, Update, Insert, and Delete). Typically when a node is just started, BulkLoad can be used, however prior to executing any other operation, scan threads must be started, and thereby Insert usage in RemoteReceive becomes mandatory. RemoteReceive involves establishing a connection with the sender node, satisfying identification number agreement and receiving data sent from the sender node and placing them within the storage engine. The protocol used in communication with the sender node is Remote Transfer Protocol as well.

PartitionDelete is implemented by issuing one or more Delete operations for the predicates supplied. The combination of RemoteSend and PartitionDelete primitive can be used to move data between nodes as opposed to copying it.
RemoteFail is implemented by going over all RemoteSends and RemoteReceives that interact with the endpoint supplied to RemoteFail and aborting them.

4.3 Implementation

Partial live migration system developed consists of three parts. As illustrated in figure 4.1 these are labeled part A, part B and part C. Part A, the table buffer, is responsible for receiving table commands and managing when each operation is enqueued to Crescando engine. Part B, is responsible for dequeuing results from Crescando engine and handling them. Part C, the copy server, is a server deployed for receiving data from other Crescando processes. Each part is handled by a different thread. All these threads have hard affinity to the core supplied by the user to Crescando as system thread. Two attributes are supplied by the user to Crescando in terms of cores, scan threads and system thread. The former specifies cores to be used by scan threads, while the latter specifies a core to be used for other operations. In this manner, Crescando can isolate between the functionality of one part and another, resulting in a better performance.

Part A is mainly responsible for receiving commands that are enqueued to the engine to be executed, which can be one of the following: Select, Aggregate-Select, Insert, Update, Delete, Remotesend, RemoteReceive, PartitionDelete. Upon receiving any of those commands, they are queued to the table buffer. The table buffer is a queue where operations resides. These input operations are transformed to output operations, which are then enqueued to Crescando engine. Remotesend, RemoteReceive and PartitionDelete operations are transformed to Select, BulkLoad (or Insert) and Delete. When the engine receives a RemoteFail with an endpoint, RemoteSend and RemoteReceive entries in the
table buffer that match that endpoint are removed from the table buffer. This part also controls when operations are enqueued to Crescando engine. Some operations are blocked before being enqueued to the engine, which depends on the type of the operation being executed and its subsequent operations. On the other hand, some other operations can be enqueued to the engine in batches. To illustrate how table buffer handles operations in terms of blockage, operations in Crescando can be categorized into either Read or Write. This is illustrated in table 4.1.

- **RemoteSend** (and other Read operations): blocks only the first *subsequent write* operation and all operations following that write operation until all operations preceding this write operation are *enqueued* to the engine. RemoteSend operation is reported as enqueued to the engine, when all of its consisting Select queries are enqueued to the engine. All operations until that first subsequent write operations can be enqueued to the engine right away. Enqueuing them in batch does not break the correctness as illustrated in section 3.5. This follows directly from the fact that commutative property holds for read operations.

- **RemoteReceive**: blocks *all subsequent* operations until it is *completed*. This follows from the fact that RemoteReceive consists of one or more write operations.

- **PartitionDelete**: blocks *all subsequent* operations until it is *enqueued* to the engine. PartitionDelete is reported as enqueued to the engine, when all of its consisting Delete operations are enqueued to the engine.

As mentioned previously, InconsistentSelect and InconsistentAggregateSelect lack formal guarantees. For these read types, the engine does not ensure the visibility of a write operation enqueued during their execution. For that reason, Part A blocks all remote copying operations (RemoteSend, RemoteReceive and PartitionDelete) from being enqueued to the engine if the engine is executing any inconsistent Select query.

In figure 4.2 the functionality of part A is demonstrated for the following operations queue: Select, AggregateSelect, RemoteSend, RemoteReceive, Delete and PartitionDelete. These are handled in the following manner:
Figure 4.2: Table Buffer Example 1
1. Select, AggregateSelect, RemoteSend: These operations are enqueued at once to Crescando engine. No blockage is necessary between these operations, since they are all read operations. The next subsequent write operation (RemoteReceive) and all its subsequent operations are blocked until all operations in this step are enqueued to the engine.

2. RemoteReceive: RemoteReceive is enqueued to the engine. RemoteReceive blocks all subsequent operations it is fully completed.

3. Delete: Delete is enqueued to the engine. Delete blocks all subsequent operations until it is fully completed.

4. PartitionDelete: PartitionDelete is enqueued to the engine.

As previously remote copying operations are not allowed to be enqueued to the engine during inconsistent queries execution. Figure 4.3 shows an example with the following operations queue: InconsistentSelect, PartitionDelete and RemoteReceive.

1. InconsistentSelect: InconsistentSelect is enqueued to the engine. All remote copying operations (RemoteSends, RemoteReceives and PartitionDeletes) are blocked until InconsistentSelect finishes completely. After this operation fully completes, the next operation is handled.

2. PartitionDelete: PartitionDelete is enqueued to the engine. After this operation is enqueued to the engine, the next operation is handled.

3. RemoteReceive: RemoteReceive is enqueued to the engine.

The second part of the partial live migration system is Part B. Part B is respon-
sible for dequeuing results from the engine. Figure 4.4 clarifies how the system handles these results. These are results that are generated from the engine due to operations execution. These results fall into two categories:

- Results of operations requested explicitly by the user: These are returned to the user
- Results of a remote copying operation: These are categorized as follows
  - Results of Select (RemoteSend): These are written to a buffer called write buffer. This buffer holds the data that will be sent to the receiver node.
  - Results of Insert (RemoteReceive): These are used to confirm RemoteReceive operation.
  - Results of delete (PartitionDelete): These are used to confirm PartitionDelete operation.

The third part of the partial live migration system is Part C. Part C is a server which is used for receiving data from other Crescando processes during remote copying. The server handles data using synchronous IO multiplexing by using select function call. The server has a state machine for each connection, to identify the state the connection is in. These states are demonstrated in figure 4.5. The states are as follows:

1. Initiated state: Initially each connection is in initiated state. Upon connection establishment with the other end, the state proceeds to connected state.

2. Connected state: The agreement property is satisfied when the other end is executing the corresponding remote operation with the same identification number (identification will be elaborated in Remote Transfer Protocol in the next section), the connection is transferred to the next state.

3. Identified state: When the connection is in Identified state, the data can
4.4 Remote Transfer Protocol

Remote Transfer Protocol is a protocol developed in this thesis to transfer data between two nodes during remote copying procedure. This protocol is used between two nodes, a node executing RemoteSend with a certain identification number while the other is executing RemoteReceive with the same identification number. These nodes communicate with each other directly using Remote Transfer Protocol.

Remote Transfer Protocol proceeds as follows:

1. IdMsg: An IdMsg is sent from the \textit{sender node} to the \textit{receiver node}. The IdMsg contains an identification number, which is the identification number of the RemoteSend operation being executed at the sender node.

2. StartMsg: A StartMsg is sent from the \textit{receiver node} to the \textit{sender node} only if the receiver node is executing a RemoteReceive operation with the same identification number contained in the previous message. The sender node upon receiving the StartMsg from the receiver node will issue the Select queries and send their results to the receiver node.

be transferred. When all the data has been transferred successfully, the state proceeds to the next state.

4. Success state: The connection has finished successfully when it has reached the Success state.

If transfer error occurs or a RemoteFail operation aborts either this RemoteSend on the sender node or RemoteReceive on the receiver node, then the state becomes Fail state.

Figure 4.5: Part C, Connection States
3. RecordsMsg: RecordsMsgs are sent from the sender node to the receiver node containing the data to be remote copied. RecordsMsg has an end-of-stream indicator, which indicates if it is the last RecordsMsg for the stream. The receiver node places this data in its storage engine upon receiving RecordsMsgs from the sender node.

4. ComitMsg: ComitMsg is sent from the receiver node to the sender node when receiver node receives the last RecordMsg (indicated using end-of-stream indicator). When the ComitMsg is received successfully by the sender node, both operations RemoteSend and RemoteReceive finishes successfully.

If transfer error occurs during, RemoteSend and RemoteReceive fail. If RemoteFail operation is issued, it will abort the remote copying operation. If the remote copying operation fails, the receiver node will empty its table, and then it can proceed normally. The sender node can always proceed normally right away if remote copying fails.

In figure 4.6 Remote Transfer Protocol takes place between two nodes. The RemoteSend node is requested to transfer 330 records. The communication proceeds as follows:

1. IdMsg: IdMsg with ID 123 is sent from the sender node to the receiver node.

2. StartMsg: StartMsg is sent from the receiver node to the sender node. The sender node issues the Select queries.

3. RecordsMsg: Four RecordsMsgs are sent from the sender node to the receiver node given the assumption that each RecordsMsg can only hold 100 records and that the Select queries issued at the sender node generated 330 records. The last RecordsMsg has end-of-stream indicator turned on. The receiver node places the records contained within the RecordsMsg in
4.5 Optimizations

Numerous optimizations have been implemented in the partial live migration system developed in this thesis. As previously mentioned, Part A transforms input operations to output operations and enqueues those output operations to Crescando. An optimization has been implemented, which aims to pipeline as much output operations as possible to the engine without breaking any of the correctness properties discussed in section 3.5. This optimization is elaborated in the next section. In addition to that, the migration system utilizes a number of buffers in order to increase the efficiency of the remote copying process as will be elaborated.

4.5.1 Pipelining operations

As previously mentioned, Part A transforms input operations to output operations and enqueues those output operations to Crescando. RemoteReceive requires all subsequent operations to be blocked until it is fully completed and PartitionDelete requires all subsequent operations to be blocked until it is enqueued to the engine. The migration system cannot pipeline operations in this case as it will break the correctness properties. On the other hand, RemoteSend and all subsequent read operations (not interleaved with any write operation) can be enqueued to the engine immediately without breaking the correctness properties. These operations are enqueued by the migration system to Crescando as a batch. In this manner, these operations are pipelined within Crescando. This optimization aims to improve the system throughput and reduce the impact on the whole system as will be elaborated in chapter 5.
4.5.2 Send, receive and engine buffer

The migration system utilizes a number of buffers in order to increase the efficiency of the remote copying process. This process is particularly performance critical due to the huge amount of data that can be transferred in the process. The remote copying process involves issuing a write system call to write the data to the socket, and on the other end issuing a read system call to read data from the socket. System calls are expensive as compared to function calls. For optimization reasons, the system aims to decrease these expensive system calls, thereby these buffers have been created:

- Write buffer: Used on the sender node.
- Read buffer: Used on the receiver node.
- Engine buffer: Used on both nodes.

The write and read buffer are used in the data transmission process. The engine buffer is used in engine results handling process. As each node can act as a sender node or as a receiver node, each node has these three buffers.

Data is written by the sender node to the write buffer. The write buffer flushes the data when it is full, which has the effect of sending the data to receiver node by issuing a write system call. The write buffer can be flushed before the buffer is full. This occurs if the write buffer is yet not full and the last record to be sent is reached (indicated using end-of-stream indicator).

The receiver node uses the read buffer. The read buffer is processed when it is either full or the end-of-stream has been received. When the read buffer is processed, it is decoded into a list of RecordsMsg. Each RecordsMsg contains the number of records in the message and the messages themselves. These records are placed within the receiver node storage engine as the read buffer is being processed. For optimization reasons, size \( S \) is given to the both the write and read buffer, where a buffer with size \( S \) can hold \( N \) records.

The engine buffer is where data dequeued from the engine are placed. The size of the engine buffer is set such that it can hold \( N \) records. In this sense all \( N \) records placed in the engine buffer as a result of a Select (from RemoteSend) can be written directly to the write buffer. In this case, if all records in the engine buffer are resulting from Select (from RemoteSend), then they can fit perfectly in the write buffer. As the write buffer becomes full, it will be flushed sending data to the receiver node. Data received will fit perfectly in the receiver’s node read buffer filling it. As the read buffer becomes full, the data in the buffer will be processed. If the data to be remote copied is large enough, every iteration retrieving results from the engine will fill all the engine buffer with records to be remote copied. These records will then fill all the write buffer, which will be flushed and sent to the receiver end to fill the read buffer. Continuing in this manner adds more efficiency in retrieving records from the engine and transferring it to the receiver node. Figure 4.7 displays how the write buffer, read buffer and engine buffer are used together.
Chapter 5

Performance Evaluation

This chapter evaluates the performance of Crescando primitive operations (Select, BulkLoad and Insert) in section 5.2 using micro-benchmarks. Analyzing these operations performance is crucial to the migration system, due to the fact that the migration system is built using these operations. Following that, the system End-to-End performance is evaluated in section 5.3 using benchmarks. The micro-benchmarks used to assess Select, BulkLoad and Insert can be used to explain the benchmarks results.

5.1 Test Environment

All benchmarks ran on dual Intel Xeon L5520 2.26Ghz (8-cores with Hyper-Threading) machine. The machine has 24GB RAM and is running Linux kernel version 2.6.32.

5.2 Primitive Operations

The micro-benchmarks evaluating Crescando primitive operations consists of three experiments for each of the following operations: BulkLoad, Insert and Select and two experiments for analyzing refreshing statistics in Crescando.

The first three experiments analyze the rate of BulkLoad, Insert and Select operation in Megabyte per second (rate of bulk loading, rate of inserting and rate of selecting) with varying a specific dimension. The first experiment investigates if the operation scales up (experience an increase in its rate) along with increasing the number of cores used. Amadeus Itinerary schema is used in the first experiment. This schema has a size of 305 bytes.

The second experiment investigates how the record size affects the operation rate. In this experiment, 6 schemas have been used. All of these schemas have only one string attribute. The string attribute size is altered resulting in
schemas having the following sizes:

1. 50 bytes
2. 250 bytes
3. 350 bytes
4. 400 bytes
5. 2000 bytes
6. 3000 bytes

The third experiment explores how the number of attributes and the type of attributes (string attributes or integer attributes) in each record impact the operation rate. In this experiment, 8 schemas have been used. All these schemas have a size of 32 bytes. These schemas can be categorized into two groups. The first group uses only integer attributes and the second group uses only string attributes, otherwise they are identical. The schemas in each group differ in the number of attributes and the size of each attribute. There are 4 schemas in each group, and they are as follows:

1. 4 attributes each of 8 bytes size
2. 8 attributes each of 4 bytes size
3. 16 attributes each of 2 bytes size
4. 32 attributes each of 1 byte size

Refreshing statistics is analyzed in the micro-benchmarks using two experiments. In these experiments the latency required to refresh statistics is measured along with varying a specific dimension. The first dimension varied is the size of data bulk loaded prior to refreshing statistics, while the second dimension varied is the number of attributes in data bulk loaded prior to refreshing statistics. Amadeus Itinerary schema is used to analyze the first dimension. The schemas used to analyze the second dimension are the same 8 schemas used in the third experiment illustrated above.
5.2 Primitive Operations

5.2.1 BulkLoad and Insert

The first experiment has been applied to BulkLoad and Insert as illustrated in figure 5.1 to analyze if they scale up with adding more cores. Amadeus Itinerary schema has been used in this experiment. BulkLoad does not scale up, this can be explained from the fact that BulkLoad code is single threaded. BulkLoad rate fluctuates between 374 Mbyte/s and 380 Mbyte/s. On the other hand, Insert experiences a slight drop in rate when more cores are added. This is due to a bottleneck in the code responsible for splitting the data across many cores. Insert rate varies between 300 Mbyte/s and 260 Mbyte/s. BulkLoad and Insert, both, do not scale up with adding more cores; however both operate at a rate higher than 128 Mbyte/s, thereby potentially saturating 1 Gigabit Ethernet interface.
The second experiment has been applied to BulkLoad and Insert as shown in figure 5.2. In this experiment the effect of the record size on the rate of bulk loading and the rate of inserting is studied. Both, BulkLoad and Insert have a peak rate for record of sizes around 500 bytes. The code allocating memory to records and placing them within the engine seems to be optimal for record sizes around 500 bytes. This peak rate is 500 Mbyte/s for BulkLoad and 350 Mbyte/s. In this experiment, BulkLoad reaches a minimum rate of 250 Mbyte/s and Insert reaches a minimum rate of 200 Mbyte/s. Generally throughout the experiment, BulkLoad has a rate higher than Insert. This experiment shows how the record size of the schema used impact the BulkLoad or Insert rate.
In figure 5.3 the third experiment has been applied to BulkLoad and Insert. This experiment assesses the effect of number of attributes and their type on BulkLoad and Insert rate. From the experiment it is evident that the number of attributes is inversely proportional to the rate of bulk loading and inserting. Moreover, it is apparent that operations have a much higher rate when integer attributes are used as opposed to string attributes. Similar to the second experiment, BulkLoad has a higher rate than Insert rate. BulkLoad has a rate between 400 Mbyte/s and 300 Mbyte/s for integer attributes, and a rate between 200 Mbyte/s and 125Mbyte/s for string attributes. Insert rate as previously mentioned is lower than that of BulkLoad. Insert has a rate around 250Mbyte/s for integer attributes and a rate between 150 Mbyte/s and 100 Mbyte/s for string attributes.

BulkLoad and Insert both do not scale up with adding more cores. The schema used plays a significant role in BulkLoad and Insert rate. They both have a peak performance for record sizes around 500 bytes, and their rate have an inversely proportional relation with the number of attributes. BulkLoad and Insert rate are higher for integer attributes as opposed to string attributes.
5.2.2 Select

The first experiment has been applied to Select to assess if Select scales up with adding cores. As shown in figure 5.4, Select operation scales up with adding cores up to a certain limit. In the figure it can be seen that after the 3rd core, the Select rate fluctuates without any significant increase or decrease in rate. Select rate rises from 125 Mbyte/s using one core to 250 Mbyte/s using two cores and then to 335 Mbyte/s using three cores. Afterwards, Select rate drops slightly to have a rate of 265 Mbyte/s with four cores and a rate of 288 Mbyte/s using five cores.
The effect of the record size in Select rate has been analyzed in the second experiment shown in figure 5.5. Select behavior is identical to BulkLoad and Insert behavior in the second experiment. Select has a peak rate for record sizes around 500 bytes. This peak rate is around 350 Mbyte/s.
The third experiment shows the effect of the number of attributes and their type on Select rate in figure 5.6. Select behavior is identical to BulkLoad and Insert regarding the number of attributes, where Select rate has an inversely proportional relationship to the number of attributes used. On the other hand, Select behavior is different than BulkLoad and Insert behavior regarding the attributes type, the type of attributes does not affect Select rate. Select rate varies between 60 Mbyte/s and 20 Mbyte/s in this experiment. Select rate is 20 Mbyte/s when a schema consisting of 32 attributes, each having a size of 1 byte is used. This schema is considered to be far from realistic schemas used in real scenarios.

Select scales up with adding cores. The schema used affects Select rate significantly. Select has a peak performance of 350 Mbyte/s for record of sizes around 500 bytes. Select rate is inversely proportional to the number of attributes used in each record. However, the type of attributes does not affect Select operation rate.
5.2 Primitive Operations

5.2.3 Refreshing Statistics

In these experiments the latency required to refresh statistics is measured along with varying a specific dimension. The first dimension varied is the size of data bulk loaded prior to refreshing statistics, while the second dimension varied is the number of attributes per record in data bulk loaded prior to refreshing statistics.

Two experiments have been carried out to analyze the latency required to refresh statistics following bulk loading data. In the first experiment, the latency required to refresh statistics is analyzed after bulk loading the following amount of data: 256 Mbyte, 512 Mbyte, 768 Mbyte and 1024 Mbyte. This is illustrated in figure 5.7. It can be concluded from this experiment that the latency required to refresh statistics following bulk loading a certain amount of data is directly proportional to the size of data bulk loaded. Around 9.5 seconds is needed to refresh statistics following bulk loading 1 Gbyte of data. Refreshing statistics occurs at a rate around 105 Mbyte/s in this case. The rate of refreshing statistics is usually higher than 105 Mbyte/s. This is due to the fact that in those experiments the attributes values are randomly chosen, thereby having a random distribution. Considering the fact that refreshing statistics is implemented using hash maps, random distribution results in more cache misses during look-ups as they are scattered with more diversity.
In the second experiment, the latency required to refresh statistics is analyzed while the number of attributes and their type are changed. This is illustrated in figure 5.8. The type of attributes affects the latency minimally, where integer attributes require a slightly lower latency as opposed to string attributes. The latency with integer attributes varies between 5 seconds and 38 seconds while the latency required with string attributes varies between 8 seconds and 42 seconds. There is a directly proportional relationship between the number of attributes and the latency required to refresh statistics, where the latency grows up by around 33 more seconds when the number of attributes is increased from 4 to 32. However as mentioned the latency required to refresh statistics in the experiments carried out here are usually higher than than the latency required to refresh statistics in real scenarios with real data.

5.3 End-to-End

5.3.1 Remote Copy

The first three experiments used with Crescendo primitive operations are applied to remote copying. The first experiment investigates if remote copying scales up (experiences an increase in its rate) along with increasing the number of cores used. Amadeus Itinerary schema is used in the first experiment. The second experiment investigates how the record size affect remote copying rate and the third experiment explores how the number of attributes and the
5.3 End-to-End

The type of attributes (string attributes or integer attributes) in each record affect remote copying rate. For each experiment, remote copying is used once with bulk loading on the receiver node and another with inserting on the receiver node to assess the difference between the performance of remote copying with bulk loading and inserting. In these experiments, the sender and the receiver node are different cores on the same machine and the network interface used is loopback interface.

Following these three experiments, the response time and turnaround time for a read operation are analyzed once following a RemoteSend operation and another following a RemoteReceive operation and are compared to the case when no operation precedes the read operation. This is repeated for a write operation, however only the turnaround time is measured (as write operations have no response time). This shows the impact of partial live migration system on the overall system. Finally an experiment is carried out to assess how the buffer size used (write, receive and engine buffer) affects remote copying rate. Amadeus Itinerary schema is used in both experiments.

The number of cores dimension has been analyzed along with the rate of remote copying in figure 5.9. In this experiment the Amadeus Itinerary schema is used. The number of cores in this figure refers to the number of cores assigned to scan threads. Each node (receiver node and sender node) has one different core assigned to system threads. An interesting conclusion can be drawn from the figure, as viewed earlier in figure 5.4 Select seems to have the lowest rate as compared to BulkLoad and Insert, thereby one can postulate that it will form the bottleneck in remote copying, which is confirmed in this figure. As also learned
Performance Evaluation

from the first experiment of Select that Select scales up with adding more cores up to 3 cores. As shown in the figure, remote copying operates at a rate of 125Mbyte/s (with bulk loading or inserting) when one core is used. In this case, 1 Gbyte of data would require 8.1 seconds to be completely transferred. Adding one more cores pushes the remote copy rate much higher, almost doubling its speed to 240Mbyte/s requiring only 4.2s to transfer 1 Gbyte data. In agreement with the first Select experiment, adding yet another core pushes the remote copy rate higher to around 315Mbyte/s with bulk loading and 287Mbyte/s with inserting requiring only 3.2s to transfer 1 Gbyte of data with bulk loading and 3.5s to transfer 1 Gbyte of data with inserting. Adding more cores after this point only causes fluctuations in the rate between 250Mbyte/s and 300Mbyte/s. As confirmed in previous experiments, bulk loading has a higher rate than inserting which is evident in this experiment where remote copying using bulk loading has a higher rate than remote copying using inserting. This is evident as soon as Select no longer becomes the bottleneck. However the difference in the remote copying rate with bulk loading and inserting is minimal. Using three cores for scan threads, remote copy is capable of replicating 17 Gbyte of data in less than one minute assuming an average rate of 300Mbyte/s.

Figure 5.10 shows the results of the second experiment with remote copy. This experiment illustrates how the record size affects remote copying rate. Resembling Select, Bulkload and Insert behavior with record sizes, it seems remote copy has a peak rate for record sizes around 500 bytes. This peak rate is 320 Mbyte/s with inserting on the receiver node and 350 Mbyte/s with bulk loading on the receiver node. The lowest rate reached for remote copy is 90 Mbyte/s.
5.3 End-to-End

Table 5.1: Response and turnaround time of a read operation

<table>
<thead>
<tr>
<th>Preceding Operation</th>
<th>Response Time (seconds)</th>
<th>Turnaround Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0.0172 ± ε0.0008</td>
<td>2.8 ± ε0</td>
</tr>
<tr>
<td>RemoteSend</td>
<td>0.0172 ± ε0.0008</td>
<td>2.78 ± ε0.04</td>
</tr>
<tr>
<td>RemoteReceive</td>
<td>3.58 ± ε0.1</td>
<td>6.3 ± ε0.1</td>
</tr>
</tbody>
</table>

for record sizes around 50 bytes.

Figure 5.11: Remote Copy Rate vs. Number of attributes

In the third experiment the number of attributes and their types are varied to see their effect on the remote copy rate. As apparent in figure 5.11. Type of attributes does not affect the experiment. Likewise, the method used to place data on the receiver node (BulkLoad or Insert) does not contribute to differences in the results. This experiment agrees with previous experiments, an inversely proportional relationship exists between the number of attributes and the rate of the operation. In the worst case where the number of attributes is highest the rate is around 22 Mbyte/s and in the best case where the number of attributes is lowest the rate is around 60Mbyte/s (this agrees with the third experiment applied to Select as shown in figure 5.6).

The response and turnaround time of a read operation are evaluated following RemoteSend and RemoteReceive and are compared to the case when no operation precedes it. This is shown in table 5.1. Amadeus Itinerary schema is used in this experiment. The read operation is a Select operation that reads 1 Gbyte of data. The RemoteSend operation sends 1 Gbyte of data and the RemoteRe-
Table 5.2: turnaround time of a write operation

<table>
<thead>
<tr>
<th>Preceding Operation</th>
<th>Turnaround Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0.4 ± ε0</td>
</tr>
<tr>
<td>RemoteSend</td>
<td>3.8 ± ε0.1</td>
</tr>
<tr>
<td>RemoteReceive</td>
<td>4.1 ± ε0.1</td>
</tr>
</tbody>
</table>

receive operation receives 1 Gbyte of data. In this setup, 3 cores are assigned to the scan threads and one core is assigned to system thread on each node. Read operation following RemoteSend operation resembles read operation with no operation preceding it in response and turnaround time. This follows from the optimization elaborated in 4.5.1, where all read operations not interleaved with write operation are enqueued in batches to the engine and are pipelined by engine. In addition to that, the existence of a large enough buffer allows processing data rapidly. However, read operation following RemoteReceive has a response time of 3.58 seconds and a turnaround time of 6.3 seconds. These operations cannot be pipelined by the engine similar to read operations as it will break correctness properties elaborated in section 3.5.2. The response time and turnaround time of RemoteReceive depends mainly on the amount of data being remote copied and the rate upon which this data is being remote copied. The impact duration in the worst case scenario lasts only for the period of migration.

The same experiment is repeated for a write operation; however only the turnaround time is measured as write operations have no turnaround time. The write operation used is an Update operation. Amadeus Itinerary schema is used in this experiment. In this experiment the Update operation updates one record. RemoteSend operation sends 1 Gbyte of data and RemoteReceive operation receives 1 Gbyte of data. In this setup, 3 cores are assigned to scan threads and one core is assigned to system thread on each node. The experiment results are in table 5.2. In this experiment, write operation has a turnaround time of 3.8 seconds following a RemoteSend operation and a turnaround time of 4.1 seconds following a RemoteReceive operation as opposed to 0.4 seconds with no operation preceding it. Write operation has a slightly less turnaround time following RemoteSend operation as opposed to following RemoteReceive operation. When the write operation follows RemoteSend, the write operation can be enqueued to the engine when all Select upon which the RemoteSend consists of are enqueued to the engine. On the other hand, when the write operation follows RemoteReceive, the write operation can be enqueued to the engine when all Inserts upon which the RemoteReceive consists of are completely finished by the engine. This might be the cause of the slight difference in the turnaround time between a write operation following a RemoteSend operation and a write operation following a RemoteReceive operation. Both turnaround times depend mainly on the amount of data being remote copied and the rate upon which it is being remote copied. The impact duration in the worst case scenario lasts only for the period of migration.
Figure 5.12 shows an experiment which assess how the buffer size used (read and write buffer) affects remote copying rate. In this experiment, the engine buffer size is also adjusted accordingly, such that if the read and write buffer can hold $N$ records, then the engine buffer is adjusted such that it can also hold $N$ records. Amadeus Itinerary schema is used in this experiment. This experiment shows how important is the size of the buffer used in remote copying. Small buffers would result in a bottleneck during remote copying. As illustrated in the figure, buffer sizes from 1 Kbyte to 100 Kbyte form a bottleneck in remote copying. A steep increase is experienced in transfer rate upon increasing buffer size from 1 Kbyte to 100 Kbyte. Following that steep increase, a lower increase in transfer rate is experienced upon increasing buffer size from 100 Kbyte to 1000 Kbyte. Following that, saturation is reached, where the buffer size ceases to be the transfer bottleneck. The current implementation uses a buffer size of 2 Mbyte for read, write and engine buffer.

Remote copying rate scales up with adding more cores (up to a certain limit, 3 cores). The schema used affects remote copying rate significantly. Remote copying performs at its peak rate when the record size is around 500 bytes. Moreover remote copying rate varies inversely with the number of attributes per record. The type of attributes however does not affect remote copying rate according to the experiments carried out. Experiments carried out with realistic schemas saturate Ethernet gigabit interface. Moreover, the impact of the partial live migration system in worst case scenario only lasts for the period of migration. The buffer size used (read, engine and write buffer) affects remote copying rate significantly. A buffer size of 2 Mbyte is used in the system developed in this thesis, which, according to the experiments carried out, does
not bottleneck the remote copying rate.

Partial live migration system was implemented in C++ for efficiency purposes. A similar system in Erlang would have a much slower performance. In such a system, Select has a bottleneck of 50 Mbyte per second. 1 Gbyte of itinerary records takes 3.4 seconds to be transferred using the system developed in this thesis when 3 cores are assigned to scan threads as opposed to 20.4 in a similar system developed in Erlang.

5.3.2 Crescando/RB

![Figure 5.13: Crescando/RB throughput](image)

Figure 5.13 shows an experiment carried out with Crescando/RB system. In this experiment, the system starts with one node containing 16 Gbyte of data, and then one node is added at a time until the system has 16 nodes, and then one node is removed at a time until the original setup is reached (one node with 16 Gbyte of data). While adding or removing nodes from the system, Rubberband instructs one or more nodes to transfer data around in order to maintain the successor-list replication scheme used. When the system has 16 nodes of data, data is repartitioned such that each node carries 1 Gbyte of data. The schema used in Crescando/RB system is the Amadeus Itinerary schema. Write and read (with isolation levels: linearizable, snapshot and minimal) throughput are measured during this process.

In this figure, whenever a node is added or removed from the system, the throughput of write and read operations suddenly change. The throughput falls under average for a short duration of time then rises above average for a short duration of time then saturates to average once again. This is the effect of remote copying on the system. The duration of this process is longer if more data is being remote copied, which is the case in the beginning and the end of
the experiment where half of the 16 Gbyte has to be remote copied as opposed to during the middle of the experiment where only small part of the data has to be remote copied. The throughput falls under average as requests cannot be handled due to remote copying, then rises above average as these previous requests fill the queues and are then suddenly executed by the engine after remote copying finishes, then the system saturates again.

Minimal read scales linearly with the added machines until it reaches a throughput around 200,000 minimal reads per second. Linearizable read and write however, due to the strong consistency guarantees they provide, rise a little in the beginning and then saturates to around 25,000 reads per second as they eventually become bottlenecked by the communication needed to provide the strong consistency guarantees. Snapshot read fall in between the two spectrums having a throughput of 120,000 reads per second.

Elastic scalability has been illustrated in this figure, where nodes can be added and removed from within a running cluster scaling up and down as needed in a flexible manner. As more nodes are added to the system, the impact of adding and removing nodes on current transactions is decreased, due to the fact that more nodes are available to respond to queries and smaller portion of data has to be moved around during remote copying.
Chapter 6

Conclusions

Development and implementation of partial live migration system on top of Crescando storage engine has been accomplished in this thesis. Crescando/RB system utilizes this developed functionality establishing elastic, fault tolerant, strongly consistent storage in the cloud.

The system developed in this thesis has been implemented in a black-box approach, where Crescando external user interface has been used without changing its internal implementation to retrieve data from the engine and place data to the engine. The partial live migration system has been implemented in C++ for efficiency purposes. Numerous optimizations have been integrated in the design and the implementation. These optimizations include having a read buffer, write buffer and engine buffer of appropriate sizes to contribute in a more efficient data transfer. Moreover the partial live migration system aims to pipeline operations in Crescando engine as much as possible. To understand the performance of the system, Crescando’s performance in retrieving and placing data have been studied at first using micro-benchmarks, following that, the system performance has been evaluated End-to-End and finally an experiment that shows elastic scalability has been carried out on Crescando/RB.

The partial live migration system developed preserves linearizability of queries and updates to the system. The migration duration is such that the transfer rate saturates a gigabit Ethernet interface for most schemas used. Only the nodes involved in the migration are impacted and only for the duration of the migration. In some cases (read operation following a RemoteSend for example) the nodes are not impacted at all. Having stated these properties in the developed system, the developed system satisfies all the requirements mentioned in section 1.1.

6.1 Future Work

Future work can be categorized into two main aspects, either in retrieving data from Crescando engine and placing data to Crescando engine or in transfer-
ring data between nodes on the network. Techniques like white-box and hybrid approach represent future work in the former aspect, while RDMA and compression both address improvements for the latter aspect.

One of the approaches that can provide improvements in the first aspect, is the white-box approach. In white-box approach, retrieving data from Crescando engine and placing data to Crescando engine can be implemented using an external system specifically customized for that purpose. Synchronization should take place between Crescando and the developed system. Developing such a system will add more complexity to Crescando engine and if poorly implemented can degrade both the engine and the developed system performance. However if properly designed and implemented, such a system can offer higher rate in retrieving data from the engine and placing data in the engine. The fact that the developed system offers partial live migration not only full live migration adds more complexity to that approach.

Hybrid approach is an approach mentioned in HYPER[4], which can offer a huge advantage in terms of retrieving data from the engine and placing data in the engine. In this approach, Crescando process is forked leaving two instances of Crescando in memory. The original instance which handles requests is left as-is and continues to serve requests. The other instance is used to retrieve the portion of data needed. This approach can be used on the sender side to extract data from the engine. However challenges exist to ensure that a consistent copy of Crescando takes place, due to the fact that Crescando is multi-threaded and fork has issues with that. The additional memory required by this approach depends on the update load (copy on update). In worst case, the system must have memory capacity big enough to duplicate the original Crescando instance.

RDMA (Remote Direct Memory Address) exploits zero-copy networking where message transfer is faster and latency is minimal. RDMA represents a potential idea in the second aspect, where transferring data between two nodes in the network can be improved. In RDMA, the CPU no longer takes part in the process (context switching does not affect the process as well) on the grounds that the network adapter takes the complete responsibility of transferring data directly to or from application memory.[12]

One of the features that can contribute to faster transfer rate between nodes is compression, where the data is compressed in a highly efficient manner prior to sending it over the network and decompressed in a highly efficient manner when received on the receiver node. Compression and decompression, both, have to be efficient enough such that it does not form a bottleneck in the remote copying rate. As a result of compression, less data is transferred over the network yielding less latency for the migration process, provided that retrieving data from the engine and placing data to the engine is not the bottleneck in the migration process.

All the ideas mentioned in this section represent potential ideas that might contribute in a higher throughput for the migration process.
Appendix A

Appendix

A.1 Amadeus Itinerary Schema

/* The Amadeus Itinerary Table */
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) ;
Bibliography

