COPYRIGHT AND USE OF THIS THESIS

This thesis must be used in accordance with the provisions of the Copyright Act 1968.

Reproduction of material protected by copyright may be an infringement of copyright and copyright owners may be entitled to take legal action against persons who infringe their copyright.

Section 51 (2) of the Copyright Act permits an authorized officer of a university library or archives to provide a copy (by communication or otherwise) of an unpublished thesis kept in the library or archives, to a person who satisfies the authorized officer that he or she requires the reproduction for the purposes of research or study.

The Copyright Act grants the creator of a work a number of moral rights, specifically the right of attribution, the right against false attribution and the right of integrity.

You may infringe the author’s moral rights if you:

- fail to acknowledge the author of this thesis if you quote sections from the work
- attribute this thesis to another author
- subject this thesis to derogatory treatment which may prejudice the author’s reputation

For further information contact the University’s Copyright Service.

sydney.edu.au/copyright
CHERRY GARCIA:
TRANSACTIONS ACROSS
HETEROGENEOUS DATA STORES

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy in the School of Information Technologies at The University of Sydney

Akon Samir Dey
October 2015
Abstract

In recent years, cloud or utility computing has revolutionised the way software, hardware and network infrastructure is provisioned and deployed into production. A key component of these vast, diverse and heterogeneous systems is the persistence layer provided by a variety of data store and database services, broadly categorised into what is referred to as NoSQL (Not only SQL) databases or data stores. These come in many flavours from simple key-value stores and column stores to database services with support for SQL-like interfaces.

These systems are primarily designed to operate at internet-scale with high scalability and fault-tolerance in mind. As a result, they typically sacrifice consistency guarantees and often support only single-item consistent operations or no transactions at all. While these consistency limitations are fine for a wide class of applications, there are a few or sometimes only parts of larger applications that need ACID transactional guarantees in order to function correctly.

To address this, we define a data store client API, we call REST+T (REST with Transactions), an extension of HTTP that supports transactions on one store. Then, we use this to define a client-coordinated transaction commitment protocol and library, called Cherry Garcia, to enable easy applications development across diverse, heterogeneous data stores that each support single-item transactions. We extend the well-known YCSB benchmark, to present YCSB+T, to enable us to group multiple data store operations into ACID transactions and evaluate properties such as throughput. YCSB+T also provides the ability to detect and quantify data store anomalies that result from the execution of the workload. Finally, we describe our prototype implementations of REST+T in a system called Tora, and our client-coordinated transaction library, also called Cherry Garcia, that supports transactions across Windows Azure Storage (WAS), Google Cloud Storage (GCS) and Tora. We evaluate these using both YCSB+T and micro-benchmarks.
Acknowledgements

This thesis has enabled me to discover my strengths and overcome my weaknesses. It has taught me to appreciate the friendship and support of family, friends and colleagues who have enabled, empowered and encouraged me in this endeavour.

I would like to express my deepest gratitude to my supervisor, Prof. Alan Fekete, for his support and guidance. His kindness, resourcefulness and willingness to help in my research work and outside of it, made this dissertation possible.

My heartfelt thanks go to my co-supervisor, Assoc. Prof. Uwe Röhm, for his guidance, support, encouragement and ideas on presenting my work. He has complemented Alan as a guiding force behind this thesis.

I thank members of the Database Research Group, Michael Cahill, Jack Galilee, Hyungsoo Jung, Meena Rajani, Ying Zhou, Vincent Gramoli and Paul Greenfield for their feedback and support. I also grateful to Arunmoezhi Ramachandran, Raghunath Nambiar, Sherif Sakr, David Bermbach, Jörn Kuhlemkamp for their collaboration. Many thanks to Lynne Hutton-Williams and Robyn Kemmis for their friendship, kindness and timely help.

I am eternally grateful to my parents, Bharati and Samir Dey, for their unconditional love, a wonderful childhood, and for instilling in me the curiosity to learn. I thank my brother, Kautuk Dey, for his encouragement and support during this work.

Most of all, I would like to thank my wife, Aditi, for her love and understanding during this very challenging time of our life. I could not have done this without you.
Contents

Abstract iii

Acknowledgements v

List of Figures xvi

List of Algorithms xvii

1 Introduction 1

1.1 Problem definition ......................................................... 2
1.2 Contributions ................................................................ 2
1.3 Relevant publications ......................................................... 3
1.4 Organisation of the Thesis .................................................... 4

2 Background 5

2.1 Motivating example .......................................................... 5
2.2 Database Management Systems ........................................... 6
  2.2.1 Transactions in Database Management Systems ................. 7
  2.2.2 Properties of Transactions ............................................ 8
  2.2.3 Transaction Programming Interface ................................ 9
  2.2.4 Requirements of Transactional Systems ............................ 9
2.3 Distributed Concurrency Control ........................................... 10
2.4 Serializability ................................................................ 10
  2.4.1 Serializability in Distributed Databases ............................ 12
  2.4.2 Serializability in Heterogeneous Federation ....................... 13
  2.4.3 Weak Isolation .......................................................... 14
2.5 Distributed Transaction Recovery ......................................... 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.1</td>
<td>REST+T data header metadata</td>
<td>72</td>
</tr>
<tr>
<td>4.4.2</td>
<td>REST+T methods</td>
<td>75</td>
</tr>
<tr>
<td>4.5</td>
<td>Limiting REST+T to HTTP verbs</td>
<td>77</td>
</tr>
<tr>
<td>4.6</td>
<td>A Case Study - Tora: a transaction-aware NoSQL data store</td>
<td>78</td>
</tr>
<tr>
<td>4.6.1</td>
<td>REST+T interface</td>
<td>79</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Storage layer</td>
<td>80</td>
</tr>
<tr>
<td>4.7</td>
<td>Chapter Summary</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>Cherry Garcia - The Protocol</td>
<td>81</td>
</tr>
<tr>
<td>5.1</td>
<td>Challenges</td>
<td>82</td>
</tr>
<tr>
<td>5.2</td>
<td>Intuition</td>
<td>84</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Overview of Two-Phase Commit</td>
<td>84</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Distributed Two-Phase Commit</td>
<td>86</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Write-Ahead Log (WAL) Protocol</td>
<td>87</td>
</tr>
<tr>
<td>5.3</td>
<td>An example application</td>
<td>88</td>
</tr>
<tr>
<td>5.4</td>
<td>Assumptions on the platforms</td>
<td>90</td>
</tr>
<tr>
<td>5.5</td>
<td>Protocol overview</td>
<td>90</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Phase 1</td>
<td>91</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Phase 2</td>
<td>92</td>
</tr>
<tr>
<td>5.6</td>
<td>Time</td>
<td>93</td>
</tr>
<tr>
<td>5.7</td>
<td>Data item records</td>
<td>94</td>
</tr>
<tr>
<td>5.8</td>
<td>Data store abstraction</td>
<td>95</td>
</tr>
<tr>
<td>5.9</td>
<td>Transaction abstraction</td>
<td>96</td>
</tr>
<tr>
<td>5.10</td>
<td>Cherry Garcia Protocol</td>
<td>97</td>
</tr>
<tr>
<td>5.10.1</td>
<td>Start transaction</td>
<td>97</td>
</tr>
<tr>
<td>5.10.2</td>
<td>Transactional read</td>
<td>97</td>
</tr>
<tr>
<td>5.10.3</td>
<td>Transactional write</td>
<td>98</td>
</tr>
<tr>
<td>5.10.4</td>
<td>Transaction commit</td>
<td>100</td>
</tr>
<tr>
<td>5.10.5</td>
<td>Transaction abort</td>
<td>102</td>
</tr>
<tr>
<td>5.10.6</td>
<td>Transaction recovery</td>
<td>103</td>
</tr>
<tr>
<td>5.10.7</td>
<td>Performance Analysis of the Protocol</td>
<td>103</td>
</tr>
<tr>
<td>5.11</td>
<td>Deadlock detection and avoidance</td>
<td>104</td>
</tr>
<tr>
<td>5.11.1</td>
<td>First preparer wins</td>
<td>104</td>
</tr>
<tr>
<td>5.11.2</td>
<td>Ensuring at least one winner</td>
<td>105</td>
</tr>
</tbody>
</table>
7.4.2 Workload properties .................................. 143
7.4.3 Defining new workloads .............................. 144
7.5 An example execution of YCSB+T ...................... 145
7.6 Chapter Summary ...................................... 147

8 Evaluations .............................................. 149
  8.1 Evaluating REST+T and Tora ......................... 150
    8.1.1 Micro-benchmark for performance and correctness of REST+T 150
    8.1.2 Performance and Correctness of REST+T .................. 154
  8.2 Evaluating Cherry Garcia ............................. 156
    8.2.1 Benchmarking framework .......................... 158
    8.2.2 YCSB+T transactional NoSQL benchmark .......... 158
    8.2.3 Micro-benchmark to test Cherry Garcia scalability .. 159
    8.2.4 Experiment setup ................................ 160
    8.2.5 Request rate limiting in WAS and GCS ............ 163
    8.2.6 Performance across Heterogeneous Data Stores ...... 164
    8.2.7 Scalability without Contention across Heterogeneous Stores 167
    8.2.8 Scalability of a Single Client Host and Single Server 172
    8.2.9 Impact of network on latency and throughput ....... 175
    8.2.10 Scalability with Multiple Client Hosts and Server .... 177
    8.2.11 Scale-out Test with 8 YCSB+T Clients and 4 Tora Servers 179
    8.2.12 Effect of Key Access Distribution on Abort Rates 180
    8.2.13 Impact of Contention on Abort Rates .............. 181
    8.2.14 Impact of Read-Write Ratio ..................... 182
    8.2.15 Impact of Number of Records Per Transaction .... 188
    8.2.16 Evaluating the Parallel Commit Optimisation .... 191
    8.2.17 Evaluating the One-Phase Commit Optimisation ... 198
    8.2.18 Impact of Network Latency ........................ 200
    8.2.19 Impact of Network Time Skew ..................... 203
    8.2.20 Summary of Cherry Garcia Evaluations ............ 207
  8.3 Evaluating YCSB+T as a Benchmark .................... 208
    8.3.1 Evaluating Performance and Scalability .......... 209
    8.3.2 Performance impact of Transactions .............. 209
    8.3.3 Evaluating Correctness .......................... 209
9 Conclusions and Future Work

9.1 Conclusions

9.1.1 REST+T

9.1.2 Cherry Garcia

9.1.3 YCSB+T

9.2 Limitations and Future Work

9.2.1 Time

9.2.2 REST+T

9.2.3 Cherry Garcia

9.2.4 Evaluating other approaches

9.2.5 YCSB+T

Bibliography

Index
List of Figures

2.1 Statechart for two-phase commit protocol . . . . . . . . . . . . . . . 21
2.2 Statechart for two-phase commit protocol with termination and
restart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.3 Choices under CAP Theorem . . . . . . . . . . . . . . . . . . . . . 34

4.1 Two applications successfully using HTTP conditional PUT on two
records using a If-Match header . . . . . . . . . . . . . . . . . . . 66
4.2 Two applications displaying anomalous behaviour as a result of using
HTTP with conditional PUT . . . . . . . . . . . . . . . . . . . . 68
4.3 REST+T object state transition diagram . . . . . . . . . . . . . . . 70
4.4 Two applications using REST+T to update two records . . . . . . . 71
4.5 Aborts when using REST+T to update two records . . . . . . . . . 72
4.6 Two applications displaying correct behaviour using REST+T . . . 73

5.1 Cherry Garcia data store object state transition diagram . . . . . . 85
5.2 Cherry Garcia architecture . . . . . . . . . . . . . . . . . . . . . . 87
5.3 Abstraction as a distributed write-ahead-log (WAL) . . . . . . . . . 89
5.4 The timeline describing 3 transactions running on 2 client hosts to
access records in 2 data stores using a third data store as a CDS . . . 93
5.5 Inconsistency resulting from a lack of single-item transactions . . . 111
5.6 Inconsistency resulting from a lack of test-and-set operations (like
If-Match) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

6.1 UML class diagram of the Cherry Garcia library . . . . . . . . . . . 120
6.2 Architecture of the Tora server and REST+T application . . . . . . 126
7.1 YCSB+T architecture . . . . . . . . . . . . . . . . . . . . . . . . . 136
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Correctness behaviour of HTTP using If-Match vs. REST+T</td>
<td>156</td>
</tr>
<tr>
<td>8.2 Throughput and aborts of HTTP using If-Match vs. REST+T</td>
<td>157</td>
</tr>
<tr>
<td>8.3 Request latency with HTTP using If-Match vs. REST+T</td>
<td>157</td>
</tr>
<tr>
<td>8.4 Throughout for 1 to 24 YCSB+T threads to access WAS, GCS and Tora stores with and without transactions</td>
<td>165</td>
</tr>
<tr>
<td>8.5 Anomaly scores for 1 to 24 YCSB+T threads to access WAS, GCS and Tora stores with and without transactions</td>
<td>165</td>
</tr>
<tr>
<td>8.6 Throughput of 1 and 2 record transactions with 1 to 20 micro-benchmark threads to access GCS</td>
<td>168</td>
</tr>
<tr>
<td>8.7 Throughput of 2, 3, 4 and 5 record transactions with 1 to 20 micro-benchmark threads to access GCS</td>
<td>169</td>
</tr>
<tr>
<td>8.8 Throughput of 1 and 2 record per transactions with 1 to 20 micro-benchmark threads to access WAS</td>
<td>170</td>
</tr>
<tr>
<td>8.9 Throughput of 2, 3, 4 and 5 records per transactions with 1 to 20 micro-benchmark threads to access WAS</td>
<td>170</td>
</tr>
<tr>
<td>8.10 1, 2 and 3 record transactions with 1 to 20 micro-benchmark threads to access WAS, GCS and Tora</td>
<td>171</td>
</tr>
<tr>
<td>8.11 2, 3, 4 and 5 record transactions with 1 to 20 micro-benchmark threads to access WAS, GCS and Tora</td>
<td>171</td>
</tr>
<tr>
<td>8.12 Throughput and latency of 1 YCSB+T client against a 1-node Tora cluster</td>
<td>173</td>
</tr>
<tr>
<td>8.13 Transaction throughput of using private vs. public IP Tora server address on AWS</td>
<td>176</td>
</tr>
<tr>
<td>8.14 Transaction commit latency of using private vs. public IP Tora server address on AWS</td>
<td>177</td>
</tr>
<tr>
<td>8.15 Throughput of 4 YCSB+T client hosts against a 4-node Tora cluster</td>
<td>178</td>
</tr>
<tr>
<td>8.16 Throughput of YCSB+T with 16 threads per client host and a 4-node Tora cluster</td>
<td>179</td>
</tr>
<tr>
<td>8.17 Aborts measured varying the key generator with 16 YCSB+T client threads against a 1-node Tora cluster</td>
<td>183</td>
</tr>
<tr>
<td>8.18 Throughput measured varying the key generator with 16 YCSB+T client threads against a 1-node Tora cluster</td>
<td>183</td>
</tr>
</tbody>
</table>
8.19 Throughput measured varying the key generator with 16 YCSB+T client threads against a 1-node Tora cluster, X-axis expanded to focus on throughput difference.

8.20 Latency measured varying the key generator with 16 YCSB+T client threads against a 1-node Tora cluster.

8.21 Aborts measured varying theta with 16 YCSB+T client threads against a 1-node Tora cluster.

8.22 Measured throughput varying theta with 16 YCSB+T client threads against a 1-node Tora cluster.

8.23 Measured throughput varying theta with 16 YCSB+T client threads against a 1-node Tora cluster with expanded y-axis to show throughput difference.

8.24 Measured number of read failures due to an invalid version, varying theta with 16 YCSB+T client threads against a 1-node Tora cluster.

8.25 Throughput of YCSB+T with 16 threads connecting to a 1 Tora node.

8.26 Impact of 1 vs 2 records per transaction observed on throughput with WAS.

8.27 Impact of 2, 3, 4 and 5 records per transaction observed on throughput with WAS.

8.28 Impact of transactions with 1 record per transaction observed with and without transactions on throughput with WAS.

8.29 Impact of transactions with 2 records per transaction observed with and without transactions on throughput with WAS.

8.30 Impact of transactions with 5 records per transaction observed with and without transactions on throughput with WAS.

8.31 Ratio of throughput without and with transactions with WAS.

8.32 Impact of transactions with 1 record per transaction observed with and without transactions on throughput with Tora.

8.33 Impact of transactions with 5 records per transaction observed with and without transactions on throughput with Tora.

8.34 Ratio of throughput without to with transactions with Tora.

8.35 Ratio of throughput of one less record without to one more record with transactions with Tora.
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.36</td>
<td>Performance comparison of serial vs. parallel commit phase with Tora with 2 records per transaction</td>
</tr>
<tr>
<td>8.37</td>
<td>Performance comparison of serial vs. parallel commit phase with Tora with 3 records per transaction</td>
</tr>
<tr>
<td>8.38</td>
<td>Performance comparison of serial vs. parallel commit phase with Tora with 5 records per transaction</td>
</tr>
<tr>
<td>8.39</td>
<td>Analysis of serial vs. parallel commit phase with Tora with 2 records per transaction</td>
</tr>
<tr>
<td>8.40</td>
<td>The effect of 1-phase optimization observed with WAS</td>
</tr>
<tr>
<td>8.41</td>
<td>YCSB+T client read and ICMP ping latencies observed across different AWS EC2 regions with Tora server in US East</td>
</tr>
<tr>
<td>8.42</td>
<td>Throughput from 16 YCSB+T threads across different AWS regions with Tora server in US East</td>
</tr>
<tr>
<td>8.43</td>
<td>Aborts per second from 16 YCSB+T threads across different AWS EC2 regions with Tora server in US East</td>
</tr>
<tr>
<td>8.44</td>
<td>Impact of latency on transaction throughput across different AWS EC2 regions with Tora server in US East</td>
</tr>
<tr>
<td>8.45</td>
<td>Correlation between transaction abort rate and transaction throughput across different AWS EC2 regions with Tora server in US East</td>
</tr>
<tr>
<td>8.46</td>
<td>Percentage of aborts vs. induced time skew in milliseconds with 16 YCSB+T client threads and 1 Tora server</td>
</tr>
<tr>
<td>8.47</td>
<td>Percentage of read failures vs. induced time skew in milliseconds with 16 YCSB+T client threads and 1 Tora server</td>
</tr>
<tr>
<td>8.48</td>
<td>Analysis of percentage of read failures and aborts vs. induced time skew in milliseconds with 16 YCSB+T client threads and 1 Tora server</td>
</tr>
<tr>
<td>8.49</td>
<td>Transaction throughput vs. induced time skew in milliseconds with 16 YCSB+T client threads and 1 Tora server</td>
</tr>
<tr>
<td>8.50</td>
<td>Observed commit latency vs. induced time skew in milliseconds with 16 YCSB+T client threads and 1 Tora server</td>
</tr>
</tbody>
</table>
# List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start transaction</td>
<td>98</td>
</tr>
<tr>
<td>2</td>
<td>Read a record</td>
<td>99</td>
</tr>
<tr>
<td>3</td>
<td>Write a record</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>Commit transaction</td>
<td>101</td>
</tr>
<tr>
<td>5</td>
<td>Abort transaction</td>
<td>102</td>
</tr>
<tr>
<td>6</td>
<td>Recover transaction status</td>
<td>103</td>
</tr>
<tr>
<td>7</td>
<td>Serializable read a record</td>
<td>114</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Increasingly, desktop and mobile application developers are choosing cloud infrastructure to take advantage of their proven high-availability and scalability characteristics. The data management infrastructures available in the cloud (e.g., Google Cloud Storage) are simple to setup and to access, and they require little or no system administration. They scale in and out seamlessly and are highly available and fault-tolerant.

In spite of this, cloud-based distributed data stores have some limitations (prompting the term NoSQL). Queries may be restricted to access via the primary key, without joins or associative access to other fields. Also, the services often provide limited transaction support: perhaps transactions are limited to a single item, or to an entity group of collocated data; some systems (e.g., Cassandra) offer only underlying mechanisms such as read-latest-value and test-and-set. The limited query capability can be worked around by coding the query processing in the application. However, the lack of general ACID transaction guarantees can be a severe hindrance to application development, demanding skills to reason about concurrency and fault-recovery issues that are “beyond the developers’ pay grade”.

There are some known ways to support multi-item transactions. If every application must manage transactional access to the various data stores, this is complex, prone to programmer error, and likely to have incorrect behaviour as the application evolves. If one asks each data store to provide the capabilities (for example, through a standard such as X/Open or WS-Transaction), this limits the application to a subset of potential stores, and it may compromise the existing features
such as availability and scalability. Another approach is to use middleware to
coordinate transactional access to the data store. This approach is more suitable
for situations where the applications are deployed in a controlled environment.

1.1 Problem definition

We explored the cloud data store and NoSQL database space and found that there
are numerous approaches to making data available as a service. However, there is
one question that stands out. Is there a better way to implement data stores API
in order to provide a better, more usable system that supports transactions?

We studied numerous systems that implement multi-item transactions across
homogeneous data stores using different approaches. We found that these sys-
tems support different programming models and interfaces. We wanted to know
whether it is possible to enable applications to efficiently access multiple, heteroge-
neous data store instances, each with a potentially different API, in a transactional
manner. If so, it is necessary to know the minimum required features and charac-
teristics of the underlying data store that will enable such a system to be built.

As we began to build the system and analysed its expected characteristics,
it was evident that we needed to determine how we could reliably evaluate the
scalability, performance and correctness behaviour of this system. It was also
important to know what are the scalability, reliability, throughput and failure
characteristics of such a system and how to measure them.

1.2 Contributions

Our approach is to place the transaction support in a client-side library. This
library offers an API that lets the application developer start a transaction, access
data items within it, and then commit or abort the transaction. Our transactions
have Snapshot Isolation (though it is easy to adapt the protocol for Serializable
transactions) as we will see in Section 5.16. The library is constructed by imple-
menting a transaction coordination class along with a set of data store abstraction
classes. Each data store abstraction keeps state about the transaction that access
it (such as timestamps, write-set, and commit status) as well as cached copies
of data items; it then utilises the specific capabilities of the data store, for example, it might install new versions of the items at the transaction completion, using test-and-set to prevent lost updates. This approach leverages the engineering achievements of existing data stores towards fast access, clustering, fault-tolerance and scalability, while giving the application developer a simple coding model with multi-item ACID transactions.

We implement our protocol as a library, that we call Cherry Garcia\(^1\) with an easy-to-use API that defines a client coordinated transaction management protocol with a pluggable data store abstraction layer enabling it to handle transactions across more than one heterogeneous data store.

In this thesis, we make the following contributions:

- We define an extension of the HTTP protocol, we call REST+T (REST with Transactions) \([39]\), that can be used to provide an API that supports transactional access to web-service endpoints and data stores in a scalable manner.
- We define a client-coordinated transaction protocol, we call Cherry Garcia \([38,40]\), to enable efficient multi-item transactions across heterogeneous key-value stores by distributed applications.
- We describe the implementation of this protocol as a library in Java. We code data store abstractions for Windows Azure Storage (WAS), Google Cloud Storage (GCS) and Tora (with an optimised RESTful interface to a high-throughput key-value store).
- We evaluate the scalability, performance, validity and reliability characteristics of our implementation with the help of micro-benchmark and the YCSB+T \([37]\) benchmark, an extension of YCSB \([29]\), suitable for evaluating web-scale transactional systems.

### 1.3 Relevant publications

1. Akon Dey, Alan Fekete, and Uwe Röhm. *Scalable Distributed Transactions across Heterogeneous Stores*. In IEEE 31st International Conference on Data Engineering (ICDE 2015), Seoul, South Korea, April 13–17,

\(^1\)Cherry Garcia is a name of a Ben & Jerry’s ice-cream flavour with heterogeneous aspects of chocolate and fruit.


4. Akon Dey, Alan Fekete, and Uwe Röhm. **Scalable Transactions Across Heterogeneous NoSQL Key-value Data Stores.**, In PhD Workshop, within Proceedings of VLDB Endowment, August 2013.

1.4 **Organisation of the Thesis**

This thesis is organised as follows. Chapter 2 describes a motivating example using modern cloud infrastructure to introduce concepts and set the stage for discussions in the remaining part of the document. This is followed by Chapter 3 where different approaches to solving these problems are presented along with related work. We then introduce the REST+T, an extension of HTTP that provides transactional data access semantics in Chapter 4. Chapter 5 introduces Cherry Garcia, a client-coordinated transaction API in the form of a library and a protocol that enables applications to access heterogeneous data stores using full ACID transaction semantics with support in the store for only single-item transactions. An implementation of REST+T, we call Tora, and the Cherry Garcia library are described in detail in Chapter 6. The YCSB+T benchmarking framework and its relation to YCSB is described in Chapter 7. This is followed by Chapter 8 which discusses the experiments we ran to evaluate the various performance, scalability and failure characteristics of REST+T and Cherry Garcia using YCSB+T and a combination of micro-benchmarks. Finally, Chapter 9 discusses the conclusions of the thesis and presents possible future areas of research.
Chapter 2

Background

In this chapter we present the background information used in the rest of this document, create a context for discussions to come, and set a stage for the motivations behind this work. We begin with a motivating example and use it to introduce the concepts and terminology used in the remaining chapters. This is followed with an overview of database systems and then elaborate the variety of research and commercial NoSQL database systems. We describe the various characteristics and provide an insight into the motivations behind them. After this, we discuss the various transaction models developed over the years and discuss their applicability to the modern cloud-based application development space. Later, in Chapter 3 we will look more closely at the research literature. Here instead we concentrate on the key ideas.

2.1 Motivating example

Let us consider a typical social network application called Fakebook. Fakebook allows users to post messages, share pictures and video clips, in addition to searching and making new friends. In order to enable the social network to scale to hundreds of millions (or even billions) of users, the creators of Fakebook use a distributed key-value store to store the information about each user. This information consists of user details like the person’s name, date of birth, geographic location, email address, additional personal information and most importantly a list of friends who are also Fakebook members.

When a user logs into the site, the user is presented with a list of messages and
updates to their page from friends and other sources along with their own posts for others to see. When a user logs in, the backend application server loads all relevant information needed to render the home page for the user by retrieving it from the key-value data store. The key-value data store is a type of database that falls under the category of databases called NoSQL databases (covered later in Section 2.7). These are scalable, distributed systems which typically provide data access using the primary key and have limited or no support for transactions.

Users typically navigate to their home page and then click on various links and which lead to new pages. As they navigate the pages they may load messages, pictures and video clips posted by others. Each picture, post or clip is stored in another store more suited for larger binary objects called a BLOB stores.

When a user reaches a friend’s page, they may want to befriend them on Fakebook by making a friend request. This creates a friend entry for the friend in the users friend list and a reciprocal entry for the new friend’s user record in user distributed key-value store. On the face of it, this is a simple operation; read the two user records from the distributed key-value store and append each user to the other user’s friend list and save the records.

In reality, this ought to be a transaction involving two records stored in a distributed key-value store. If the first record operation succeeds and the second fails the first must also be undone. Essentially, either both records must be updated or neither should.

The distributed key-value store can scale linearly as a result of its limited transaction capability. However, the limitation makes multi-item operations described above difficult to code without a framework that supports transactions.

2.2 Database Management Systems

In the literature, a database is described as follows [44]:

A database is a collection of related data. By data, we mean known facts that can be recorded and that have implicit meaning.

while a database management system is described as below [44]:

A database management system (DBMS) is a general-purpose
2.2. DATABASE MANAGEMENT SYSTEMS

A collection of software that enables users to create and maintain a database.

DBMS software enables users to create, manipulate, and retrieve data using an easy to use interface without exposing the storage and retrieval details of the low-level data storage medium.

The advantage of using a DBMS are:

- **Data Independence:** The DBMS provides an abstract view of the data by hiding the details of data representation and storage.

- **Efficient Data Access:** The DBMS utilises indexes and other storage optimisations to enable the data to be accessed efficiently. This is particularly useful when the data is stored on external storage devices.

- **Data Integrity and Security:** The DBMS can be used to enforce integrity constraints, visibility, and control access to the data.

- **Data Administration:** Centralised administration of a database by professionals can make the task of organising and fine-tuning the storage and retrieval of data efficient.

- **Reduced Application Development Time:** The high-level interface with support for commonly used functionality helps to reduce the application development and deployment time.

- **Concurrent Access and Crash Recovery:** The DBMS enables multiple users to access the data concurrently as though they were accessing the data one user at a time. The simplifies the task of application development by hiding the complexity of concurrent data access.

2.2.1 Transactions in Database Management Systems

A transaction is an abstraction provided by the DBMS that consists of a set of operations that are executed as a group and simplifies the task of concurrent access to the database. The operations can be both read and write operation on either individual data items or other database entities which can include a physical page or an entire table. A transaction consisting of only read operations is called a
query, while a transaction comprised of operations that modify data is called an update.

Applications that access the database start by making a request to begin a transaction, followed by performing a series of read and/or write operations on data, and finally requesting to either commit the changes or abort (also called rollback) the transaction. Two transaction, $T_1$ and $T_2$ are said to be concurrent if there is an overlap between the interval $[\text{begin-time}(T_1), \text{commit-time}(T_1)]$ and $[\text{begin-time}(T_2), \text{commit-time}(T_2)]$.

2.2.2 Properties of Transactions

The transaction can be described in terms of what are called “ACID” properties:

- **Atomicity**: All or none of the changes as a result of a transaction must be evident in the database. It should not be possible to have only some of the changes and not the others. This is often called the “all or nothing” property.

- **Consistency**: Changes to the database as a result of a transaction must transform it from one consistent state to another. This means that data integrity constraints and rules must not be violated by the changes. We note that this is a responsibility of the application coder, rather than the platform.

- **Isolation**: Each transaction execution should be unaware of the changes being concurrently made by other transactions running on the system.

- **Durability**: Once a transaction is committed, the changes are guaranteed to persist regardless of process or system failure. These changes must be visible to subsequent transactions.

Database management systems provide transactional access to data using different techniques. These techniques have an inevitable overhead, limiting the overall performance and throughput of the system.
2.2.3 Transaction Programming Interface

So that the ACID properties of transactions can be maintained, the application program must demarcate the boundaries of the transaction and the expected outcome – success or abnormal termination. For this, the system must provide the following three calls:

- **Begin**: to mark the beginning of the transaction.
- **Commit**: to mark the successful end of the transaction.
- **Rollback**: to mark the unsuccessful termination of the transaction and a request to abort it.

During its execution, the application program defines the transaction boundaries using these calls to group the set of operations that constitute the transaction. These transactions may be invoked in sequences, loops or even executed in parallel by different threads of control. The transaction concept can be applied to many modern application scenarios that involve access to data in databases and data stores.

2.2.4 Requirements of Transactional Systems

The primary requirement of a transactional system is that it provide ACID guarantees for operations designated to the set defined by transactions. In order to achieve this, it must at least provide the following components:

- a form of concurrency control that guarantees the isolation properties of both committed and aborted transactions.
- a transaction recovery mechanism that ensures that atomicity and durability of transactions is maintained.

In order for such a system to be usable in a real-world scenario, the system must ensure that the above guarantees are provided while maintaining good performance. Performance is measured using the following metrics:

- **high-throughput**, defined as the rate of successful transactions per unit time.
- **shorter response times**, defined as the time the transaction was issued and time it was perceived to be completed.
2.3 Distributed Concurrency Control

Modern enterprises are typically based upon a distributed architecture made up of multiple applications and data servers. The servers can be homogeneous or heterogeneous in terms of products, interfaces, protocols and they can either be largely autonomous or dependent on other servers. This has become an increasingly diverse landscape with the advent of cloud computing and the proliferation of open-source technologies.

When data is distributed across multiple servers, it can be across homogeneous systems that may comprise multiple instances of similar systems, or it may be on a set of completely independent, heterogeneous systems. The latter is becoming the increasingly common scenario. These federated systems are called multidatabase systems in the literature.

Implementing transactional semantics in these systems has many challenges. These challenges ultimately boil down to two primary issues. First, to ensure the atomicity of transactions so that sequential operations are able to see “all or nothing” of preceding transactions. Second, to make it possible for concurrently running operations to be isolated from each other’s influence.

In addition to this, applications must be able to rely on the durability of the persistent store regardless of whether it is a traditional database or modern data store. This is ensured by building robust failure recovery and fault-tolerance into the system.

2.4 Serializability

Concurrently running transactions can lead to the same data being read and/or written by different threads of execution. This results in three of the following types of conflicts:

Lost updates This problem occurs as a result of two concurrently running transactions that perform a read operation on the same data item, then update the value and write it back. The second write prevails while the first write operation is overwritten or “lost”.

Inconsistent reads This problems is visible when one transaction reads multiple
values and writes only part of its intended updates to those values. The same values are then read by another transaction which get an inconsistent image of the values due to the partial update.

**Dirty-reads** If an update to an value is made following which it is rolled back as a result of intention or failure, then any reader who reads the updated value before it is rolled back gets a dirty value.

A transaction schedule is said to be *serializable* if its outcome (the resulting database state) is equal to the outcome of its transactions as though they were executed serially without overlapping in time. In reality, transactions may execute concurrently (they overlap and their operations interleave). Transaction serializability is the most well known correctness criterion for execution of concurrent transactions.

Each transaction must be correct by itself (because it meets certain integrity conditions). A schedule that consists of an execution of more than one transactions is *correct* if each transaction still meets its integrity conditions. If transactions do not overlap in time they cannot interfere with each other. They are said to be “serial” because there is complete isolation between them. The order of transaction execution is valid as long as no dependency exists between them. Thus, a correct schedule is defined as one that consists of an execution that is equivalent to any serial execution of the transaction.

Serializability is achieved in database systems using a combination of the following techniques [72]:

**Pessimistic Concurrency Control (locking):** Operations must first lock data records to be operated upon using a lock table before proceeding. If a conflicting operation is in progress, the attempt to acquire a lock will block until the lock is released.

**Logging:** The description of the operations to be carried out by each operation are logged by each transaction using log records. These log records are used to both, undo aborted transactions, as well as recover after a crash to bring the database to a consistent state before further processing can start.

**Optimistic Concurrency Control:** Operations continue without blocking or conflict verification until transaction commitment, following which conflict-
indicating conditions are checked and transactions are aborted if patterns of conflict emerge.

**Managing multiple versions of data items:** The system maintains multiple versions of the data so that blocking can be avoided but read operations may read potentially obsolete data.

Different DBMS implementations use a combination of these techniques in different ways and making trade-offs. This determines how each system performs under different workloads.

### 2.4.1 Serializability in Distributed Databases

The challenges of ensuring serializability in distributed databases or multi-database systems is a lot more complicated than single monolithic DBMSs. To address this the following techniques are used as described in Weikum and Vossen, Chapter 18 [139]:

**Distributed Two Phase Locking (D2PL)** This involves the acquisition of locks for all object to be operated upon at each site followed by the release of locks respectively. There are primarily two variations of how locks are managed. The first is Primary Site 2PL in which one of the sites is chosen as the location where they are managed. The other is Distributed 2PL (D2PL) which depends on all sites having knowledge of every other site in the system. This is done in various ways. One of which is to propagate locking operation information to every site so that every site is aware of what locks are held across the system. The other is to manage locks locally and only propagate lock information when needed (typically when it begins to release the locks). This makes deadlock detection in distributed databases significantly more difficult than single site deadlock management as described below.

**Distributed Timestamp Ordering (DTO)** Every operation that arrives is assigned a timestamp. Operations are performed in the order of timestamps by the local scheduler at each site. Timestamp assignment is tricky because each scheduler is aware of only the local operations and not the global context of the operations from the point of view of the global transaction.
Timestamp generation can be performed centrally across the system or in a distributed manner at each individual site.

**Distributed Serialization Graph Testing (DSGT)** SGT involves maintaining a conflict graph for operations belong all the transactions to determine wait-for cycles. Doing this on single node is considerably simpler than implementing it to work in a distributed manner. This is because the task of finding cycles are much easier on a single site than doing across multiple sites.

**Optimistic Protocols** These techniques use an optimistic approach which assumes that conflicts between transactions are uncommon. This involves three phases of operation: the read phase in which objects are read and writes are executed on a copy visible to only the transaction in question, followed by the validation phase in which the transaction operations are validated to prevent conflicts, and finally the write phase in which the changes are persisted at each database site.

**Distributed Deadlock Detection** The task of detecting deadlocks where two conflicting transactions acquire locks on records the other has already locked is done by analysing the dependencies in a wait-for graph by looking for cycles. This can be done centrally or in a distributed manner using techniques like *edge chasing* which involves sending a *probe* message to the blocking transaction which in turn forwards it to all transactions it is waiting for. A cycle is detected when a transaction receives a probe with its own identifier in it. An alternate approach is called *path pushing*, in which, entire wait-for paths are passed between transactions. Each site can compose its own understanding of the wait-for graph and eventually detect the cycles in it.

### 2.4.2 Serializability in Heterogeneous Federation

Serializability across heterogeneous systems can be achieved using techniques that use local guarantees to implement global serializability. It is particularly challenging to construct a globally serializable schedule using local guarantees because local sites may cause indirect conflict with schedules on other nodes.
The ticket method \cite{51,52} is an explicit measure to ensure global serializability across multiple heterogeneous servers. It is light-weight and allows interoperability across a wide variety of database systems. Each database maintains a special data item called a ticket which is accessed only by global transactions. The ticket is essentially a timestamp. Each transaction must either read the ticket or issue a take-a-ticket operation. The ticket is read, incremented, and written back. The value of ticket is used to determine the serialization order of the transaction at each site.

The ticket read or issue operation can happen in an implicit manner, explicit manner, or a mix of both. An explicit ticket-based approach involves each transaction reading a ticket or issuing a take-a-ticket operation while in the implicit approach in which ticket ordering graphs are not maintained at each site.

2.4.3 Weak Isolation

Serializable isolation implemented with Strict Two-Phase Locking (S2PL) is widely considered too slow for high-performance transaction processing applications because shared locks must be held during the entire duration of the transaction. This results in many lock conflicts and high rates of deadlocks. Gray et al. \cite{59} proposed to provide application developers with a choice of isolation levels weaker than fully serializable. A weaker isolation level is implemented by early release of some of the locks acquired by S2PL, or by avoiding taking some locks altogether. The paper describes specific locking algorithms that provide each level; named “Degree 0”, “Degree 1”, and so on. Later, in the ANSI SQL standard, Degrees 1 and 2 were called Read Uncommitted and Read Committed, respectively.

When serializable executions are not guaranteed the interleaving of operations in concurrent transactions can cause incorrect results. These are called anomalies. Anomalies are not be possible when the transactions executed serially. Weak isolation levels give application developers a mechanism that trades performance for correctness. Hence, performing updates at any isolation level other than serializable, requires careful analysis to avoid data corruption. While measuring the effect on throughput may not be difficult, it is difficult to measure how much the choice of a weak isolation level will have on correctness.
To verify the correctness of a system composed of many different types of transactions when using weak isolation, involves complex analysis of the interactions between all pairs of transactions.

### 2.5 Distributed Transaction Recovery

Transaction recovery in distributed systems is faced with difficulties similar to those of concurrency control in distributed systems. This is particularly true for atomicity, the all-or-nothing semantics of transactions, as it must extend to update operations across multiple servers. This requires updates on all servers to be committed, or all updates on all of them to be undone.

This complexity primarily arises due to partial failure scenarios such as the failure of one server while others continue to function normally. In addition to this, correctness depends on the reliable message exchange between the participating servers. Therefore, message loss resulting from network component failures make it difficult to determine the fate of the transaction.

It is impossible for a receiving server to determine the cause a message not being received. It could happen because of a genuine server crash, network failure, or just a slow server or network. The conservative assumption is that the sending server has crashed. Unfortunately, it is also impossible to determine at what point the server crashed; it could have failed right before it could complete the decisive part of its side of the commit, while the other servers could have started actions to commit its part of the transaction. This ends up in an unacceptable state where one server has committed its part of the transaction while the other will have to undo its updates during restart. These types of scenarios lead to inconsistent data.

The solution is to implement a special handshake protocol between the two involved servers from a family of distributed commit protocols, the most widely used instance of which is the two-phase commit (2PC) protocol. The protocol builds upon each servers’ local capabilities for logging and transactional crash recovery to which a couple of message rounds are added to establish a “contract” between the servers to ensure that either all servers commit the transaction, or all of them undo it.

There is an inevitable implication within the protocol that there are certain
situations under which a failed server must communicate to other servers during its restart to discover the system-wide decision about the termination status of one or more in-doubt (also called uncertain) transactions.

The elegance of the protocol lies in the fact that it makes very few assumptions about how the different servers implement their local recovery, as long as there is in an understanding of the notion of winner versus loser transactions. This why it relatively easy to implement a distributed commit protocol in a federated system.

Federated systems are multi-tiered systems in which a transaction, often implicitly, manipulates data along entire hierarchies of servers and spans a tree of work units that are all subject to the distributed commit protocol. Such situations are appropriately handled by a hierarchical version of the two-phase commit protocol. We will discuss this generalisation of the flat two-phase commit protocol to trees in Section 2.5.2.

There are a number of opportunities for optimisations of both the flat and the hierarchical variants of two-phase commit protocol. These optimisations mostly aim to reduce the number of communications and/or logging overhead of the protocol. We discuss the most important of these optimisations in Section 2.5.3.

2.5.1 The Basic Two-Phase Commit Algorithm

This section presents the basic two-phase commit algorithm in three subsections: we begin with the actual protocol that specifies the message exchange and logging actions, cover the necessary steps when failures occur, opportunities for and limitation of independently recovering individual servers after a failure.

2PC Protocol

The protocol known as (basic) two-phase commit, or 2PC, serves to ensure the atomicity a distributed transaction. The critical point within such a transaction that has made updates on more than one data server is when the application program requests the commit of these updates. Each server will be responsible for the crash resilience of its local updates. The participating servers are participants, sometimes denoted in the literature as “agents”, while the servers on which the participants run are usually referred to as resource managers to emphasise that not only traditional data servers can participate in a transaction but also more
general servers, such a queue manager. On failure, these participants must later undo the local updates resulting from the transaction.

The latter may lose the log entries that would be necessary to redo their part of the transaction, which is feasible because log buffers do not need to be forced before commit. The committed participants, on the other hand will release the transactions local locks (or perform the analogous steps in a non-blocking type of concurrency control protocol) to enable the new transactions to see the updates and even modify the same data. This implies that, in full generality, it is not possible to undo the updates of the successfully committed participants at a later point when one or more other participants have failed. The invocation of high-level compensation steps would be feasible only if no subsequent transaction perform a semantically conflicting operation, otherwise the reproducibility of the resulting schedule can not be ensured which would violate the isolation property of the transactional ACID contract.

This is solved by introducing a transaction coordinator that mediates between the application program’s commit request and the various participants. The coordinator is a process that runs on either the client or any server. The “shared storage” maintained by the coordinator is a stable log that holds log entries about the commit status of transactions.

By initiating two rounds of message exchanges with the participants (hence the name, two-phase commit), the coordinator ensures a unanimous outcome of the transaction – either all participants perform local commits or all perform local rollbacks.

In the first round, the voting or preparation phase, the coordinator conducts a vote or poll, by asking each participant whether it is ready to commit the transaction. This message is usually called a prepare, request-to-vote or request-to-prepare message. A participant replies “yes” when all log entries are on the stable log that would be needed to redo the local updates; it may have to force its log buffer before replying. If a participant has crashed recently or is not able to commit local updated of the transaction for some other reason, it replies “no”. When a “yes” vote is received by the coordinator from all the participants, it knows that the transaction can be safely committed and announces this by sending a “commit” message to every participant. This is the second of the two message rounds is called the decision phase. If even one participant replies “no”, the
coordinator declares an “abort”. The participants acknowledge the coordinator’s message regarding the transaction’s fate with an “ack” message in both cases.

For this, the commit protocol needs to be made resistant with regard to certain failure classes. The failures to consider are as follows:

**Message losses**: A message does not arrive at the destination process because of a network failure (for example a router failure or software failure on a gateway).

**Message duplication**: Some network component may end up duplicating a message, perhaps attempting to recover from a transient failure causing the same message to arrives multiple times at the destination (possibly interleaved with other messages and hence difficult to detect).

**Transient process failures**: One or more of the involved processes, participants or coordinators exhibit a “soft crash” and need to be restarted without damage to any data already on secondary storage.

These failures fall under the category of *omission failures*, in contrast to *commission failures* which include messages with maliciously manipulated content (for example, saying “yes” when the participant actually sent a “no”). Here, we consider omission failures and disregard commission failures. Handling manipulated messages would lead us into a broader class of *distributed consensus protocols* known as *Byzantine agreement*.

The two-phase commit protocol must cope with message and process omission failures. In particular, it is possible that a participant replies “yes” to the coordinator’s poll then crashes. The participant can no longer simply perform local crash recovery as if there were no distributed transaction. Its ability to redo the transaction’s local update does not mean that it is the correct way to recover. The coordinator may decide that the transaction needs to be rolled back because another participant voted “no”, while this may not be known at the time when the failed participant restarts (it is definitely not known to the restored participant itself). To solve this, the restored participant must check back with the coordinator before it can decide to treat the transaction as a winner. Therefore, the participant must learn that the transaction needs to be globally rolled back from
the coordinator. Thus, each participant that votes “yes” must actually be prepared to go either way – redo the transaction’s local updates or undo the updates. Further more, during restart, the participant needs to have a means of detecting that it has already replied to the coordinators poll before the crash and needs to contact the coordinator (or wait until the coordinator resend the decision once again). This is done by writing a prepared log entry to the participant’s stable log before sending a “yes” reply to the coordinator.

Two-phase commit is called a blocking protocol because a participant becomes dependent on the coordinator by having to communicate with it once it is in the Prepared state. The participant may become blocked for an indefinite period by waiting for the decision from the coordinator. A transaction’s global fate prevents the release of any locks that are held on behalf of the prepared transactions local parts (and there may be multiple prepared transactions at the same time). This is a potentially critical aspect as a blocked participant does not have knowledge of fate of the global transaction. However, this issue is not nearly as critical as it may sound with carefully operated, highly reliable servers and network connections, and distributed transactions that do not span more than a few (for example, less than five) servers. The evidence indicates that the practical viability of two-phase commit in many industrial-strength information systems, including internet-based applications.

The coordinator needs to write additional log entries to track the progress in the protocol in addition to the participant’s prepare log entries and the final commit or rollback log entries. First, the coordinator must write a begin log entry before it starts its polling message round. Additionally, the coordinator must write an end log entry at the end of the second phase once it has received “ask” messages from all participants, however, this log entry does not need to be forced immediately. In order to create the end log entry, the coordinator must have a list of participants from which it anticipates an “ack” message that resilient to coordinator failures. The coordinator also includes a list of participants in the begin log entry.

Here, “force-write” requires a log entry to be created in the log buffer and then forced to disk. At some later point in time, the coordinator should discard log entries about distributed transactions that have been terminated in the past, failing which, the coordinator log would grow indefinitely. Additionally, log truncation
is necessary for fast restart, especially given that the coordinator log is embedded in a server’s log file. Thus, as soon as the end log entry is on the stable log, the coordinator can forget the transaction and garbage collect its specific log entries. This is why the end log entry is also called a “forgotten” or “done” log entry in the literature.

Log entries are used to remember the last relevant state of a process. When a process restarts after failure, it resumes its part of the protocol in the last known local state. There is no way for the participant to know whether it has already sent the reply before the crash or whether it crashed between writing the prepared log entry and sending the reply. This uncertainty is inherent and cannot be eliminated by additional log entries because it is not possible to combine disk write and message send into an atomic operation. Similarly, when the coordinator fails and is restarted in the Begin state, it simply reinitiates the poll by sending prepare messages to all participants, possibly creating duplicate messages to some or even all participants.

These considerations can be systematically cast into a finite state automaton that specifies the behaviour of a participant or coordinator. Figure 2.1 (from Waikum and Vossen [139], Chapter 19) gives a full picture of the states though which the various processes proceed and the messages that are sent and/or expected to be received in a given state. The figure represents a statechart specification – a formalism used to describe long-lived business workflows. It is, in essence, a finite state automaton (more specifically, a set of communicating automata whose cross-product forms the entire statechart) with logical conditions attached to each state transition.

In Figure 2.1, the ovals represent the states of a process; each corresponding to a specific log entry. The process is in a particular state if and only if the stable log contains the corresponding log entry. The coordinator’s Forgotten state corresponds to the presence of the end log entry while the other states are self-explanatory. Initial states are represented by incoming edges with a small point as origin; final states have no outgoing edges. Transitions are labeled with event-condition-action rules (ECA) in the form “event [condition]/action,” where each component of the triple is optional and committed when insignificant. A condition is fired if the specified event occurs and the condition is true; the state transition then executes the specified action, the current state is left, and the new state is
Figure 2.1: Statechart for two-phase commit protocol
entered.

In this scenario, events are message receipts and the actions are message sends. For instance, the transition labeled *Prepare1*/yes1 from the first participant’s Initial state to the Prepared1 state specifies that the transition occurs upon the receipt of a prepare message and then sends the message yes1 as the transition’s action. Both the origin and the destination of each message is determined by the protocol. The statechart in Figure 2.1 depicts three orthogonal components, one for each process participating in the protocol, that execute in parallel, synchronised by the exchange of messages determined by their state transitions.

The two participants have identical behaviour to be viewed as two instantiations of the same statechart. We distinguish their states and messages by the number suffix to avoid notational ambiguities.

**Restart and Termination Protocol**

The protocol of Figure 2.1 is robust with respect to failures. However, it does not guarantee that all processes make active progress toward a global commit or rollback after a failure because messages can get lost without any of the three involved processes failing even during normal operation. A remedy could be to resend the messages and keep doing so until the recipient responds. However, such repetition of message sends should be driven by timeouts. These considerations leads us to two extensions of the basic two-phase commit protocol:

- How a restarted protocol should proceed is specified by a *restart protocol*.
- How a process should behave upon a timeout while it is waiting for some message is specified by a *termination protocol*.

As all participants follow the same protocol, we have four cases as:

**The coordinator restart protocol:** The continuations of the coordinator’s protocol after the failure of a coordinator.

**The coordinator termination protocol:** The behaviour of the coordinator upon timeout, for instance, caused when it suspects a failure of some participant or a network problem.
2.5. DISTRIBUTED TRANSACTION RECOVERY

The **participant restart protocol**: The continuation of participant’s protocol after a failure of the participant.

The **participant termination protocol**: The behaviour of the participant upon timeout, for instance, when it detects a coordinator failure and is unable to communicate with it.

A process cannot distinguish between a genuine communication failure with a partner and when there is “merely” a network problem. The ability to distinguish between these two cases would require additional networking services and there is no foolproof way of accomplishing this. The termination protocols handle both cases uniformly due to this.

The statechart of Figure 2.1 describes two kinds of additional transitions for the precise specification of the restart and termination behaviour:

**F transitions** are triggered during restart after a process failure. Once the process’s last state is determined from the log entries on the process’s stable log, the transition is made without any further preconditions.

**T transitions** are triggered upon timeout and are also made without further preconditions.

Figure 2.2 (from Waikum and Vossen [139], Chapter 19) shows the complete 2PC statechart with T and F transitions from all relevant states. The restart and termination transitions are crucial for progress towards a global commit or rollback. The various failure cases are:

**Coordinator restart**: When the coordinator fails in the Initial state, it automatically reinitiates the entire protocol (which does not require a F transition).

**Coordinator termination**: When the coordinator observes a timeout, because one or more participants have not replied (in time), it resends messages according to its current state.

**Participant restart**: When a participant fails in its Initial state, it has not yet given up its decision autonomy and can choose to unilaterally abort its part of the transaction.
Figure 2.2: Statechart for two-phase commit protocol with termination and restart
2.5. DISTRIBUTED TRANSACTION RECOVERY

Participant termination: The participant behaviour upon timeout is essentially the same as during restart.

Theorem 2.5.1 (Theorem 19.1 in Chapter 19 of Weikum and Vossen [139]) guarantees a safety property in that a certain unacceptable outcome can be ruled out. A safety property generally states that “nothing bad will ever happen”. In addition, we need a termination guarantee which is a liveness properties which is essentially that “something good will eventually happen”. This states that the system of processes will eventually reach either the global state (Committed, ... , Committed) or (Aborted, ... , Aborted) after a finite number of state transitions (including T and F transitions).

Theorem 2.5.1 The 2PC protocol guarantees the atomicity of distributed transactions, in that it ensures that if one process reaches a final state (i.e. its local Committed or Aborted state), then either all processes (i.e. all participants and the coordinator) are in their Committed or all of them are in their Aborted states.

This liveness property is captured in Theorem 2.5.2 (Theorem 19.2 in Chapter 19 of Weikum and Vossen [139]) under the assumption that there are a finite number of failures provided that all processes run for a sufficiently long duration.

Theorem 2.5.2 For a finite number of failures (process failures with subsequent restarts or message losses), the 2PC protocol will reach a final global state (with either all participants committed or all participants aborted) with a finite number of state transitions.

Independent Recovery

The primary disadvantage of the 2PC is that it is a potentially blocking protocol. To overcome this, it is modified so that it never blocks regardless of the type of failure. This is done by constructing a protocol that ensures independent recovery for each process. This allows a failed and restarted process to convert local transactions into a local final state without having to communicate with any other process. This is achieved by assuming that at most one process failure has occurred during the execution of the commit protocol, and no message losses have
occurred other than the ones that have resulted from the process failures. This assumption, listed below in Theorem 2.5.3 (Theorem 19.3 in Chapter 19 of Weikum and Vossen [139]), is called the single-failure assumption.

**Theorem 2.5.3** Under the single-failure assumption, independent recovery can be guaranteed by an appropriately designed distributed commit protocol.

The Forgotten coordinator state is ignored and it is assumed that Committed and Aborted are the actual final states in order to discuss the difficulty of guaranteeing independent recovery. The problem is that a participant can be in its Prepared state while another participant is already in a final state, either Committed or Aborted. This prevents independent recovery because the prepared participant is unable to determine whether another participant is committed or aborted solely on the basis of its own local state.

**Theorem 2.5.4** There exists no distributed commit protocol that can guarantee independent process recovery in the presence of multiple failures.

The multiple-failure scenario covered by Theorem 2.5.4 (Theorem 19.3 in Chapter 19 of Weikum and Vossen [139]), is not unrealistic: double failures sometimes occur within short time windows of vulnerability, even though it is very infrequent. Therefore, it must be accepted that distributed commit and independent process recovery cannot be reconciled for fundamental reasons. So, the potential for blocking under certain conditions is an inherent property of every conceivable commit protocol, and the search for a protocol that eliminates this limitation of 2PC is futile.

### 2.5.2 The Transaction Tree Two-Phase Commit Algorithm

The choice of the process that takes on the role of the transaction coordinator is a difficult choice because it depends on the following factors:

**Transaction initiator:** This signifies the process that initiates the transaction and could be the client, the application server, or the database server itself. It is the process that issues the commit or rollback requests.
Reliability and speed of participants: This depends on the number of participants, their location, the characteristics of their network connection, and the speed at which it responds to the various rounds of messages needed for transaction commitment to be possible.

Communication topology and protocol: The network protocol the transaction initiator uses to communicate with the participants during the execution of the transaction and to what extent and by what means the participants communicate among each other.

The transaction initiator is chosen as the coordinator in the simplest case. This makes perfect sense if the initiator is a reliable, fast and well connected application server. However, these considerations reveal three important observations:

- The processes involved dynamically form a tree with the initiator as the root during the execution of the transaction. Each edge in this process tree corresponds to a dynamically established communication link over which a process submits a request to another participant and receives the corresponding reply. An edge is created whenever a new participant is added to the transaction. A link can be reused for subsequent requests between two processes once it is established.

- The process tree can be flattened by choosing a coordinator for the execution of the commit protocol by having the coordinator talk directly to each process in the tree. Flattening is feasible in most cases because all processes can piggy back their network addresses and those of their callees within the reply message that they send to their caller; enabling the initiator to gather all necessary network addresses at commit time.

- Flattening is a special case of restructuring the communication tree to optimise the execution of the commit protocol that occurs during transaction execution. The commit communication tree reuses existing communication links and simply "rotates" the tree around the newly chosen coordinator.

The message flow and writing of log entries in the hierarchical commit protocol follow from the two roles of an intermediate node, participant with regard to its caller and coordinator for its subtree. The process first sends prepare messages to
its children. This message then cascades down the tree until it reaches all the leaf nodes. A reply wave then moves up the tree. When a non-root, non-leaf process receives yes votes from all its children, it finally force-writes its prepared log entry and sends a yes vote to its parent. Subsequently, during the second phase of the protocol, a commit message wave moves down the tree, and finally ack messages are collected on the way up.

The correctness of hierarchical 2PC follows from fact that it is simply a finite number of instantiations of the basic 2PC protocol. All it additionally needed is to ensure that all intermediate nodes behave consistently: if they decide, as a coordinator for their subtree, that their subtree should commit the transaction they must vote “yes” to their parent, and consequently the same must hold for the abort case. Thus, this invariant is guaranteed.

2.5.3 Optimised Algorithms for Distributed Commit

Different kinds of execution costs and potential bottlenecks are incurred by the two-phase commit protocol leading to a number of possible optimisations. These extensions address the following performance dimensions:

1. Minimising runtime overhead and maximising the sustained transaction throughput with the given computer and network resources by reducing the required number of message and the required number of log writes, particularly forced log writes.

2. Minimising the local lock contention and response times by shortening the “critical path” from the start of the commit protocol to the point when local locks can be released.

3. Increasing throughput by lowering the probability of blocking or striving for independent recovery under as many circumstances as possible.

Defining the presumed behaviour of a process (i.e., its default reaction) in the absence of more explicit information can lower the number of messages and force log writes. This can speed up the commit protocol and contribute to reducing the time before locks are released. A similar effect can be achieved by parallelising the message wave down and up the process tree. This can also be achieved by
eliminating of subtrees early from the protocol message rounds when these subtrees can be inferred to be irrelevant.

**Presumed-Abort and Presumed-Commit Protocols**

The flat basic 2PC protocol with $n$ participants and a coordinator, has a total execution cost of $4n$ messages and $2n + 2$ forced log writes. The number of forced log writes is a result of two log entries by each participant, for the prepared and the commit or rollback log entry, and two forced log entries by the coordinator for the begin and the commit or rollback entry.

It is necessary to force the coordinator’s begin log entry so that the coordinator can remember the transaction and determine its termination in case the coordinator fails before the commit or rollback entry is written. The number of forced log entries can be reduced $2n + 1$ without explicit optimisations in many cases.

The message and forced logging costs of 2PC can be reduced by making specific presumptions about a process’s behaviour when a certain piece of information is missing. This leads to two different variations of the 2PC, known as the presumed-abort (PA) and presumed-commit (PC) protocols. The default behaviour of the presumed-abort (PA) variant is to rollback a transaction when there is no information about the global decision and also no way of reconstructing this information. The presumed-commit (PC), on the other hand, presumes that the transaction should be committed in this kind of situation. These protocols must be designed cautiously to prevent inconsistent behaviour among the participating processes. The basic 2PC does not make any presumptions along these lines. Therefore, it is also called the presumed-nothing (PN) protocol.

The candidates for such relaxation are:

- To eliminate of the coordinator’s begin log entry.
- To avoid forcing the participants’ commit or rollback entries since this information can be obtained from the coordinator who must have stable commit or abort log entry.
- To eliminate the participants’ ack messages in the second phase of the commit protocol.
 Presumed-abort (PA) protocol: It is possible to drop all of the above information without risking globally inconsistent decisions for loser transactions. If no begin log entry is written, the participants’ rollback log entry is not forced, and that no acknowledgements are sent during the second phase; without acknowledgements, the coordinator would either have to keep its logs entries about a transaction forever, adversely interfering with logs truncation, or we would have to assume that it can discard its log entries about a transaction regardless of the presence or absence of ack messages. These optimisations work only for loser transactions; the presumed-abort protocol cannot afford missing all three of the above mentioned pieces of information for winner transactions.

 Presumed-commit (PC) protocol: Another optimisation is to use force-written commit logs without ack messages. Therefore, once the coordinator’s commit log entry is garbage-collected, later inquiries by participants is answered according to the commit presumption. In order to ensure that this third state of the coordinator is not confused with the first stage, the coordinator must have a stable log entry that captures its being in the first phase of the protocol. The begin log entry of the basic 2PC is used for this purpose by forcing this entry to the stable log for correctness. The characteristic features of the actual presumed-commit (PC) protocol are: forced begin and commit log entries at the coordinator, no ack messages for winner transactions but explicit acknowledgements for losers. Therefore, the participants do not need to force the commit log entry to their local log which is a significant saving to logging cost.

The presumed-abort variant of the 2PC protocol for \( n \) participants saves \( n \) messages (the ack messages) and \( n + 2 \) forced log writes for aborted transactions (including the coordinator’s being log entry) while there are no savings for committed transactions. On the other hand, the presumed-commit protocol saves \( n \) messages and \( n \) forced log writes (for the participants’ commit log entries), and there are no savings for the case of aborted transactions. The presumed-abort protocol has slightly higher savings in network and disk resource consumption but it optimises the much more infrequent case.

 Presumed-any (PA) protocol: It may be the case that in a heterogeneous federation of servers, a given server can handle only one out of the three protocols, PA, PC, and PN, and these capabilities may vary across servers. The problem is
how to reconcile PA, PC, and PN within one transaction when a distributed trans-
action involves servers with different capabilities. The solution, referred to in the
literature as the presumed-any protocol, is to ensure that the coordinator includes
sufficient information in the form of forced log entries to cope with both PA and PC
participants; no special steps are necessary for PN as this option essentially boils
down to either PA or PC depending on the given transaction’s outcome. Thus, a
presumed-any coordinator force-writes begin, commit and rollback log entries and
it expects acknowledgements from PA participants for committed transactions and
for PC participants for aborted transactions.

The intermediate nodes in a process tree need to force-write a begin log en-
try under the presumed-commit protocol even if their entire subtree is read-only,
and this cost does not arise with the presumed-abort protocol. This is why the
presumed-abort is considered the protocol of choice and has been selected as the
basis for the industry standard hierarchical presumed-abort 2PC, X/Open XA.
However, the presumed-commit alternative is certainly the more attractive ap-
proach for the flattened version of the 2PC.

**Read-Only Subtree Optimisation**

The coordinator in a process tree does not have a priori knowledge of all partici-
pants of the transaction. For instance, some participants may not have performed
any updates on their local data; these may include both intermediate nodes in
the tree as well as leaf nodes. However, at the time the coordinator initiates the
commit protocol, it is often impossible to know these read-only participants. Elim-
inating read-only participants from the protocol’s message and logging efforts as
early as possible is an important optimisation. A third kind of vote called “read-
only” is introduced in addition to the two options “yes” and “no” to make this
possible.

A participant that has voted “read-only” in the first phase does not receive the
coordinator’s decision and can be eliminated from the second message round. Once
participant receives the prepare message from the coordinator and has replied with
read-only vote, it can immediately start releasing its local locks without writing
any log entries.

There are two complications with this seemingly straight-forward optimisation.
First, intermediate nodes in the process tree can vote read-only only if none of their descendants in the entire subtree has performed any updates. Second, this optimisation needs to be integrated with other optimisations, particularly the ones based on presumption. Further analysis shows that the presumed-commit protocol needs to have forced begin log entries for the top-level coordinator and all subcoordinators.

**Coordinator Transfer**

It may be desirable to choose a coordinator that differs from the transaction initiator for reliability or communication efficiency reasons. This amounts to choosing a new root and “rotating” the tree around this new root node for a process tree. The new tree has exactly the same edges as the old one but the parent-child relationship is reversed for all edges along the path from the new root to the old one.

The choice of the coordinator and the corresponding conceptual rotation of the tree can be dynamically carried out when the initiator has issued the commit (or rollback) request. The commit protocol is executed as usual once this is done. This is one of the very earliest 2PC optimisations for linear communication where the process tree forms a single, non-branching path. We could derive a linear communication structure even if the tree has a more general shape at the expense of possibly having to establish new bilateral sessions. However, the case that the tree is already a linear path with a small number of processes more than two is very frequent in practice. The optimisation is known as the *linear 2PC*.

The benefit of this 2PC variant is gained from choosing the last participant in the chain, the leaf of the linearised tree, as the coordinator. This is also known as the *last-agent optimisation*.

**Reduced Blocking**

We have seen that 2PC is prone to blocking. This also holds true for the optimised versions discussed here. Theorem 2.5.4 shows that the impossibility of independent recovery is an inherent property of the distributed commit problem. So the best we can do is to reduce the probability of running into blocking situations as much as possible. This is done by defining a commit protocol that eliminates blocking in
certain situations like single process failures to make sure that it behaves correctly under all possible circumstances (even though it is susceptible to blocking).

The essence of such a protocol is to avoid global states whose set of possible successor states contains both local commit and local rollback states. The \textit{three-phase commit (3PC) protocol} is a full-fledged protocol based on the idea. In 3PC, the coordinator first collects votes and causes participants to become prepared then it disseminates the outcome of the voting phase to all participants. After this it waits until it is certain that all participants know the result. The coordinator finally ask the participants to commit to rollback only after this newly introduced message is successfully sent to all participants.

The 2PC protocol has been considered with mixed feelings in practice because of its susceptibility to blocking. So much so that many applications have been designed to avoid distributed transactions altogether. It is true that 2PC does not scale up to transactions that span hundreds of servers, but for a small number of participants on carefully administered and highly reliable servers, the benefit of achieving distributed data consistency in an easy-to-program way largely out-weighs the lower probability of creating performance problems. So much so that in heterogeneous federations with autonomously operated servers, the standardised form of distributed commit supported by all important commercial systems, known as XA, is an excellent option for building applications with decentralised, interrelated data like e-commerce and more advanced e-services on the Internet.

\section{Distributed systems}

Distributed systems are difficult to build because failures can occur at many points and each type of failure has to be dealt with appropriately. The design and architecture of the system needs to assume partial system failure as a norm and reliably work around these problems.

\textbf{Brewer’s CAP Theorem}

In an invited talk at PODC 2000 \cite{Brewer2000}, Eric Brewer, made the following conjecture:

\textbf{Theorem 2.6.1 (CAP theorem)} It is impossible for a web service to provide the following three guarantees:
CHAPTER 2. BACKGROUND

Figure 2.3: Choices under CAP Theorem

- **Consistency**
- **Availability**
- **Partition-tolerance**

All three of these properties are desirable and expected from real-world web services.

A formal proof of Brewer’s CAP theorem was presented by Gilbert and Lynch [56]. Figure 2.3 describes the three choices available in a distributed system out of which any two are achievable.

Large-scale distributed key-value store implementations choose **Availability** and **Partition-tolerance** and sacrifice **Consistency** in their implementation. However, traditional database systems that implement ACID transactions choose **Consistency** over **Availability**. We are faced with the same choice in order to implement ACID transactions in NoSQL data stores which choose **Availability** over **Consistency**.

**Eventual Consistency**

Eventual consistency [136] is a consistency model which depends on the notion that given sufficient time without any updates to the system, all updates will
eventually propagate to all replicas resulting in a consistent system. A stricter interpretation of this states that every update to the system must reach each replica within a uniformly bounded period of time.

**Timeline Consistency**

In a timeline consistency model [28], the global order of updates to an object are applied in exactly the same order on all replicas. Replicas may be out of synchronisation with the primary copy but will eventually reach a consistent state by application of the updates in the order in which they were applied to the original object.

### 2.7 NoSQL and NewSQL database systems

Traditionally application state needed to be stored in a persistent storage system in the form of a database. As the popularity of Relational Database Management Systems (RDBMS) grew they assumed the role of the de facto database technology for persisting application state. RDBMS technologies were originally designed for Online Transaction Processing (OLTP) applications typically found in enterprises. They were suitable for certain type of access patterns where the schema is well understood and was mostly flat (relations). In addition, these systems came with strong transaction guarantees governing the behaviour in scenarios when there were catastrophic failures.

With the internet age came application that needed to store data but the access patterns and data models were very different, dynamic, and ever changing. Many applications need little transaction guarantees and have simple access patterns, usually via some primary key (like the user name). The de facto choice of persistence naturally pointed towards traditional RDBMS technology. However, this soon proved to be untenable causing system developers and architects to go back to the drawing boards.

These applications gave birth to a new breed to highly scalable, fault-tolerant and dynamically load balanced data stores that came to be known under the broad term NoSQL databases (short for Not Only SQL databases) like MongoDB, Cassandra and SimpleDB. These systems gave up some key properties of traditional
RDBMS systems like full ACID compliant transaction support and strict database schema. The data query capabilities provided by these systems are also limited, often to only primary key searches. Indexes and searches on attribute values of data items is virtually non-existent or limited at best.

A common incarnation of these systems is called a distributed key-value store. It is characterised by a simple data model which consists of an opaque data record that is addressable by its primary key. Additional metadata can be stored along with each data record in the form of metadata header information or explicit metadata fields.

Another relatively new class of systems called NewSQL systems that are distributed and scalable but support the relational model and have SQL as their primary interface. Examples of this type of systems include H-Store, Google Spanner, VoltDB, Clustrix and SAP HANA.

Different distributed key-value data stores or NoSQL data stores exhibit a mix of different performance, scalability, availability characteristics and architectures. While these systems may differ in architecture and implementation they strive to achieve the following characteristics. This is discussed in further detail in the original YCSB paper [29].

**Scale-out** Distributed NoSQL databases are able to support the large data sizes and high request rates because they are able to spread the request load, and the data to be stored, across a large number of commodity servers each hosting a part of the data. A successful scale-out mechanism is able to effectively spread the data and client requests across these machine without exposing any bottlenecks.

**Elasticity** Elasticity enables a system to add capacity by adding new servers and spreading the load effectively while the system is still running. It complements the scale-out capabilities of a system.

**High availability** Commodity hardware is prone to failure making the availability of the system an important requirement. This is particularly important when these systems host data belonging to multiple tenants.

It is impossible to provide all the desirable features into one system and each one makes different design and architecture choices. For instance it is impossible
to simultaneously achieve consistency, availability and partition tolerance [24]. Therefore, these systems have to make the following trade-offs in order to exhibit the characteristics mentioned above:

**Read versus write performance** Higher read performance is achievable with good random I/O throughput while higher write performance can be achieved using append-only log-structured storage systems.

**Latency versus durability** Persisting data to disk achieves durability but significantly increases write latency. Not syncing writes to the disk reduces latency and improves throughput but reduces the durability guarantees.

**Synchronous versus asynchronous replication** Replicating data improves performance, system availability and avoids data loss. This can be done either synchronously or asynchronously. Synchronous replication increases write and update latency while asynchronous replication reduces latency but also reduces consistency guarantees caused by stale data.

**Data partitioning** Data can be stored in a row-oriented or column-oriented storage structure. Row-oriented storage structures are suitable for applications that need to access a large proportion of the fields in each data record during a transaction while column storage structures are suited to applications that use a very small number of the total fields in each record and are suitable for situations when the records have a very large number of fields.

We are interested in systems that choose consistency over availability and provide a transactional interface to the application. In particular, we would like to evaluate systems that support multi-item transactions.

As systems have matured, some of the features of traditional SQL databases (RDBMS) have crept into these modern, scalable systems; albeit in a restricted form, without sacrificing the scalability and fault-tolerance characteristics. These systems are called NewSQL databases or data stores. These systems support some form of query language similar to SQL (or a subset of it) and a data model that is not restricted to flat relations. There is some support for indexes on data attributes which the query engine is able to take advantage of.
2.8 Transactions in NoSQL database systems

In some systems, transactions are guaranteed on single-item operations only and query capability is limited. Data is looked up using the primary key and more complex queries are supported in only a few systems. Others support only single item transactions. The design of these systems focus on scalability, performance and consistency of operations on single data items. They use Paxos [82] or other consensus algorithms [73, 102] to ensure consistency while others use replication protocol optimisations to achieve greater performance to support native multi-item transactions.

However, most systems leave multi-item transactions to be handled by the application. This is susceptible to programmer error and the resulting implementation is often completely wrong.

One way to address this is to implement a relational database engine to provide the query capabilities and transaction support with the raw data stored in a distributed key-value store [21]. This is suitable for applications that require a complete SQL interface with full transaction support. The performance and transaction throughput of the system is limited only by the underlying data store and queue implementation.

The relative simplicity of the data store API makes application development simple and robust. These applications most often use a write-once-read-many (WORM) data access pattern and function well under eventual consistency guarantees. However, there are increasing demands for applications that are built to run on the same data that require better consistency guarantees across multiple records.

One way to solve this is to implement transactional capabilities within the data store itself. The data store manages the storage as well as transaction management. These systems support transactions across multiple keys with a distributed, homogeneous key-value store and focus on providing a better, more capable distributed data store and optimise the transaction coordination across it.

Another approach uses middleware for caching and coordinating transactional data access. Each leaf record is associated with the root record using a foreign key. The foreign key is used to cluster related records and partition data across storage nodes and any transactions across entity groups use two-phase commit
(2PC).

The middleware approach works well when the application is hosted in the cloud and there is a known and controlled set of data stores used by the application. They perform well in this situations and provides one programming interface to the application simplifying the data store access. However, these systems require to be setup and maintained separately increasing the management overhead.

This is not suitable for all use cases, particularly where individual application instances need the hands-off, low maintenance features characteristic of key-value stores and each may have different access privileges to individual data stores.

Another way of implementing multi-key transaction support for distributed key-value stores is to incorporate the transaction coordinator into the client. We know of two implementations that use this approach. It depends on a central fault-tolerant timestamp service called a timestamp oracle (TO) to generate timestamps to help coordinate transactions and a locking protocol to implement isolation. The locking protocol relies on a read-evaluate-write operation on records to check for a lock field associated with each record. However, it does not take advantage of test-and-set operations available in most key value stores making this technique unsuitable for client applications spread across relatively high-latency WANs. No deadlock detection or avoidance is implemented further limiting its use over these types of networks.

2.9 Evaluating database systems

Computer systems of all types are evaluated using benchmarks that measure features at a particular level of abstraction. Database benchmarks deal with data management access, with a large collection of items and operations that access and modify those items (perhaps as get/put, or as SELECT/UPDATE operations). An aim of these benchmarks is to satisfy the criteria of a successful benchmark prescribed by Gray [60]. They are: relevance to an application domain, portability to allow benchmarking of different systems, scalability to support benchmarking large systems, and simplicity to ensure that the results are easy to understand. A useful benchmark must endeavour to exhibit these properties even when applied to new classes of systems.
2.9.1 TPC

Transaction Processing Performance Council (TPC) is a non-profit organisation setup in 1988 under the leadership of Omri Serlin. Its main intention is to define transaction processing and database benchmarks, and disseminate objective, verifiable performance data to the industry. The benchmarks are used to evaluate the performance of computer systems; the results of which are made public through their web site (http://www.tpc.org). Its charter is: the creation of good benchmarks; and, the creation of good processes for reviewing and monitoring these benchmarks. The belief is that a good benchmark is the basic foundation of civilised and fair competition.

During the late 1980’s, the online transaction processing (OLTP) space was hot and the stakes were high and competitors in the industry were aggressively trying to claim that they had the best OLTP system. This is when the contenders used the TP1 benchmark, originally developed at IBM, to measure the performance of their system. This benchmark measured the performance of ATM transactions performed in batch mode without network or user interaction components similar to the TPC-B benchmark. This caused performance numbers to be inflated and there was no way to supervise or control the benchmark process leading to a lack of credible results and confusion among the different stake holders.

In the mean time, in April 1, 1985 issue of Datamation, Jim Gray along with 24 others collaborators from both academia and industry, anonymously published an article titled, “A Measure of Transaction Processing Power.” [19] It outlined a test for on-line transaction processing system called “DebitCredit.”

Unlike previous benchmarks, it specified a true system-level benchmark that included the network and user interaction components of the workload. It also outlined key features of the benchmarking process that were incorporated into the TPC process including; the publishing of total system cost along with the performance rating; test specification in terms of high-level functional requirements; well defined benchmark workload scale-up rules; and, constraints on the transaction response time.

Prior to the formation of the TPC, vendors used TP1 and DebitCredit, and often, their own interpretation of these benchmarks, to muddy the waters with unreliable performance numbers which finally came to an end with eight companies
forming the Transaction Processing Performance Council (TPC) in August, 1988.

2.9.2 Benchmarks

Traditional data management platforms are measured with industry standard OLTP benchmarks like TPC-C [133], TPC-E [134], and TPC-W [132]; these have focused on emulating end-user application scenarios to evaluate the performance (especially the throughput, and throughput relative to system cost) of the underlying DBMS and application server stack. These benchmarks run a workload with queries and updates that are performed in the context of transactions, and the integrity of the data is supposed to be verified during the process of the execution of the benchmark. If the data is corrupted, the benchmark measurement is rejected entirely.

The TPC-E benchmark simulates a brokerage firm with customers who generate transactions that include trades, account inquiries, and market research. In turn, the brokerage firm interacts with financial markets to execute orders on behalf of the customers after which relevant account information in the database is updated. This benchmark evaluates the scalability of the database by varying the number of customers defined for the brokerage firm to emulate the workloads found in businesses of different sizes. It defines a mix of transactions that must be maintained. It measures the throughput in transactions per second (tps), in particular, this refers to the number of Trade-Result transactions sustained over a period of time by the server.

The TPC-C benchmark is an online transaction processing (OLTP) benchmark. It is a more complex benchmark compared to previous OLTP benchmarks such as TPC-A due to its richer variety of transaction types, complex database schema and overall structure of execution. It incorporates a mix of five different types and complexity of concurrent transactions that are either executed in an on-line manner or queued for deferred execution. The database consists of nine different tables, each with a different range of record and population sizes. The primary metric measured is the throughput in transactions per minute (tpmC).

It simulates a computing environment, complete with a population of users executing transactions against a database, in which the principal activities (transactions) are centred around an order-entry environment. Transactions include the
entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. The benchmark portrays the activity of a wholesale supplier, however, it represents any industry that must manage, sell, or distribute a product or service and is not limited to any particular business segment.

TPC-W benchmark simulates a controlled internet commerce environment that simulates the activities of a business oriented transactional web server. The workload exercises a wide variety of system components found in such environments. These are characterised by multiple online browser sessions; database access and update for dynamic page generation; consistent web objects; the execution of multiple concurrent transaction types of varying complexity; on-line transaction execution modes; many database tables with a wide variety of sizes, attributes, and relationships; integrity checks for ACID properties of transactions; and varying contention on data access and update. Performance is measured in the number of web interactions processed per second.

The activities of a retail store is simulated by multiple web interactions, in which the response for each interaction is subject to a time constraint. The benchmark simulates three different application profiles by varying the ratio of browse to buy operations. These are primarily shopping (web interactions per second (WIPS)), browsing (WIPSb) and web-based ordering (WIPSo). The measure of performance are the WIPS rate, its associated price per WIPS ($/WIPS), and the date of availability of the priced system configuration.

### 2.9.3 Evaluation metrics

The primary task of evaluating a database system is to measure its performance across various metrics. The workload is designed to target particular system features and measure its performance using different metrics described here.

**Performance throughput:** The most common evaluation metric is performance throughput of the system. This can be in terms of the number of operations or transactions completed per unit time. This is often measured in transactions per second (tps) or operations per second (ops).

**Execution time:** In some cases, the system is evaluated based on how long the
system takes to perform the set of operations defined by the benchmark workload. The TeraByte Sort or TeraSort\footnote{http://sortbenchmark.org/} benchmark is a good example of this. It measures the time taken to sort one terabyte of data.

**Running cost:** Often a larger, more resource-hungry system outperforms systems that consume less resources. The running cost of the workload is an important measure. It helps the user to determine the long-term costs associate with running the database system in production.

**Energy consumed:** The energy consumed by the database system affects the total running cost of the system both due to higher electricity as well as cooling equipment purchase, maintenance and running costs. The JouleSort \cite{116} benchmark is an example of a benchmark that measures the number of records sorted with one unit of energy measured in joules.

**Performance under constraints:** Sometimes, the user has an existing system that has certain memory, storage, processing or service level agreements to adhere to while performing its tasks. These constraints are known ahead of time and the database system must be able to support the desired set of operations defined by the workload within the set constraints. For instance, the MinuteSort benchmark, a derivative of AlphaSort \cite{101}, evaluates the number of records sorted within a minute of execution by the system.

**Correctness measure:** Most benchmarks assume that the system behaves correctly during the execution of the workload. For instance, in the case of database benchmarks, it may assume that no anomalies are introduced during the course of running it. However, this may not be true. A measure of how many anomalies are introduced as a result of a particular workload on a database is important to verify its correctness behaviour that may be critical to the use case.

### 2.9.4 Properties of a good benchmark

Benchmarking complex, multi-tiered systems like databases is repetitive, time-consuming and exhausting \cite{117}. A good benchmark needs to ensure that the
results of the benchmarking process is not confusing or misleading so that it can aid in the prevention of making the wrong architecture, design or database system choices. The lack of a good benchmark can make benchmarking new application domains or newly developed database systems, like NoSQL systems, even more painful.

The following are characteristics of the benchmark that enable it to be suitable for testing the performance and usability of a wide variety of database systems while also making it possible to simulate a large spectrum of application use cases [14]:

**Repeatability:** The workload and operations executed during the benchmark should be repeatable. This allows identical workloads to be executed against different database systems in various configurations in order to compare them. This ensures the ability to replay exactly the same sequence of operation across multiple benchmark executions.

**Applicability Across a Wide Application Domain:** It should be possible to emulate a wide variety of application use cases using predefined workloads to enable application designers and architects to objectively pick a suitable database system that meets their needs. Additionally, it must be possible to extend existing workloads so that application-specific operations or scenarios can be incorporated into the workload in order to be able to perform a more realistic evaluation. This includes cases of increasing and decreasing load and varying periodicities of workload intensity like sudden spikes and troughs.

**Provide Suitable Abstractions:** A widely usable benchmark does not make any assumptions about the specific capabilities of the database under test and is unaware of the specific database implementation. For instance, it may not know whether the database supports transactions with ACID guarantees. The benchmarking framework must provide suitable abstractions so that this is possible.

**Easy to use:** It should be easy to configure, run, use, and extend the benchmark. The output resulting from the execution of the benchmark must be easy to interpret so that it can be used to make objective decisions.
Capable of simulating Distributed Application: It should be easy to define a workload simulating a distributed application. The distribution and coordination of the workload should be handled correctly and efficiently. The results of the execution of the benchmark should be gathered across all the benchmark workload instances in a correct and efficient manner as though they were running on a single machine.

Minimal Impact on Results: The benchmarking infrastructure must be sufficiently light-weight so that it is neither a performance bottleneck nor does it adversely skews measurements. This is possible only if the benchmark itself is scaleable so that any change in the implementation of the benchmark does not adversely effect measurement results.

Fine-Grained Measurements: Measurements should be collected and stored in a suitably fine-grained manner so that they be easily sliced and diced for further analysis.

Support Different Deployment Topologies: It should be possible to simulate scenarios in which database applications as well as the database itself are deployed in different deployment topologies that may include geo-distributed deployments. Applications are increasingly making use of more than one database technology simultaneously. It should be possible to benchmark these types of heterogeneous setups.

Support Micro-Analysis: Sometime it is necessary to evaluate certain specific features of the database; for example its index structures. It is common practice to use micro-benchmarks to perform these evaluations. A good benchmark must allow the user to define specific patterns of operation that can be used to define micro-benchmarks that analyse specific features of the database system in isolation.

2.10 Chapter Summary

As the application space changes and Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), Infrastructure-as-a-Service (IaaS) and other service-oriented architectures evolve, it is becoming increasingly evident that, higher throughput,
lower latency, greater scalability and the need for fault-tolerance has often come at the cost of features like consistency, correctness and transactional guarantees provided by traditional data stores and databases systems. This has enabled a plethora of successful application to function and grow to unprecedented scale. However, we have seen that the need to integrate with heterogeneous data persistence technologies and web-services while maintaining traditional transactional semantics is a real need. The current approach is to use traditional RDBMSs, to deal with applications that need transactions in a partitioned or shared manner in order to achieve the scalability objectives. However, this approach does not always result in the desirable characteristics and lacks the same ease of deployment and development.

This exposes a gap in database technology where there is a need for the scale-out and fault-tolerant features of modern NoSQL systems and the robust transactional semantics of traditional database systems. There have been numerous efforts to address this. We discuss many of these systems in Chapter 3. Later, we describe our proposal to deal with this in the form of a transactional data access extension of HTTP, we call REST+T, and a client-coordinated transaction protocol and library, we call Cherry Garcia, in Chapters 4 and 5 respectively.
Chapter 3

Related Work

Databases have been a topic of research and development for many years. Over the years the technology has evolved and been applied to different domains. However, many of the underlying principles that apply to these class of systems has remained the same. In this chapter, we explore the related work on the subject of transaction processing and database management systems.

3.1 Cloud and web services

In this age of online shopping, the web-application is the critical component that ensures that the seller charges for the correct items bought and the buyer receives what was promised. Behind the web-server, most systems are backed by some form of a transactional database or data store the enables the seller to make the guarantees the buyer seeks. Without this, the seller would soon lose credibility causing his business to fail.

HTTP [50] has emerged as the protocol of choice for making information available over the internet; primarily in the form of HTML documents, this has gradually evolved into dynamic, application and system state information. For example, Wi-Fi network modems seen in nearly every household worldwide provide an HTTP interface to a web-based management console. This is implemented by a bare-bones HTTP server that enables any browser to manipulate settings on the router using the web-based UI.

REST [51] has made it possible to implement services that enable applications to use HTTP as a means of communication over the network. This is done by
allowing the application to communicate with the service using Remote Procedure Calls (RPC) implemented using HTTP methods. Leaving each REST endpoint to maintain its own state is the essence of the simplicity of the architecture making it highly scalable and robust. Competing technologies like SOAP, on the other hand, have been less successful due to the inherent session oriented approach.

Transactional updates over HTTP have so far been implemented using protocols like WebDAV [57] and its extensions like CalDAV [34]. These systems use a pessimistic locking protocol that depends on an application session in the form of leases. In addition, due to the pessimistic locking protocol, there is an implicit underlying expectation that the entire collection of documents (or objects) being accessed reside in one homogeneous system. This design is monolithic in nature and not suitable for scale-out applications.

### 3.2 Distributed Concurrency Control

Parallel and distributed databases have been a topic of research and development for many years. The textbook by Özsu and Valduriez [103] covers the topic in detail. Various distributed concurrency control algorithms have been developed over the years. Bernstein et al. [15] and Pamadimitriou [104], cover the details of the various techniques developed. The applications of distributed concurrency control and deadlock handling in the operating systems domain are discussed in Silberschatz et al. [124] and Tanenbaum [130].

A comprehensive overview of the subject of concurrency control in multi-database and distributed database systems is provided by Breitbart et al. [22]. Gray [62] pioneered distributed locking and defined its fundamental principles. Alsberg and Day [8] originally described primary site two-phase locking (2PL). Subsequently, Menasce et al. [93] and Traiger et al. [131] discussed the subject of locking in distributed systems further. Bernstein et al. [17] introduced serialization graph testing (SGT) and timestamp ordering (TO) and described techniques that exploited knowledge from analysing potential conflicts among predeclared transactions. Boksenbaum et al. [20] described sophisticated timestamp-based transaction certification techniques.

The work in this thesis is based on the theorems originally stated in Breitbart and Silberschatz [23] listed under Theorem 5.16.1 in Chapter 5, a detailed proof
3.3 DISTRIBUTED TRANSACTION RECOVERY

for which was later given by Mehrotra et al. [92] in 2000.

Georgakopoulos et al. [54, 55] proposed the ticket method to force conflicts between global transactions locally. The paper also prescribes correctness arguments for mixing explicit and implicit tickets in the same federated system. Schenkel et al. [120] extended this work, applying it to federated systems with weaker serializability guarantees such as snapshot isolation (SI). Other efforts [119] have continued to investigate this approach extending it to techniques using graph-based concurrency control and object model transaction management in multi-database systems [135].

3.3 Distributed Transaction Recovery

Gray [62], Lindsay et al. [41], and Lampson and Sturgis [84] independently described the two-phase commit (2PC) protocol. A brief historical account of the evolution of 2PC is described in Gray and Reuter [61]. This includes remarks about the use of 2PC-style protocols in industrial systems that predate the earliest scientific publications on the topic. Skeen and Stonebraker [127] presented a comprehensive and rigorous state-transition model. They have also proven the impossibility of distributed commit with independent local recovery. Harley [66, 67] developed the statechart formalism of the protocol used in Section 2.5.1.

Distributed commit protocols have been covered extensively in textbooks by Bernstein et al. [15], Gray and Reuter [61], Bernstein and Newcomer [16], and Özsu and Valduriez [103]. System implementations with distributed transactions have been described in detail by Eppinger et al. [46], Haskin et al. [68], and Nett et al. [100]. The books by Mullender [98], Lynch [90], and Coulouris [31] are good sources on broader topic of distributed consensus algorithms, reliable messaging, and failure resilience of distributed systems in general. A broader background on networks and communication protocols can be found in textbooks by Kurose and Ross [78] or Walrand and Varaiya [138].

Mohan and Lindsay [97] first explicitly addressed distributed commit for transaction trees. See Mohan et al. [96] for additional work on the topic. They were also responsible for the development of the presumed-abort and presumed-commit protocols and the read-only optimisations for process trees described in Sections 2.5.2 and 2.5.3. Lampson and Lomet [83] and Al-Houmaily et al. [6] have done more
recent work on improving the presumed-commit protocol. The use of a coordinator and centralised log sharing by participants for reduced logging cost has been studied by Stamos and Cristian [128].

In order to minimise communication costs and to shorten the critical path until locks can be released, algorithms for coordinator transfer have been developed by Segall and Wolfson [121], Wolfson [141], and Raz [115]. The last agent optimisation was introduced by pioneering work by Gray [62].

Skeen [125] developed a three-phase extension for independent recovery in a limited setting of single failures. 3PC is also covered by the textbook by Bernstein et al. [125]. A survey of the entire family of non-blocking commit protocols is covered by Babouglou and Toueg [10]. Performance issues related to cooperative termination have been covered in Daschamp [42].

Excellent surveys of the plethora of optimisations for distributed commit protocols have been provided by Samaras et al. [118] and Chrysanthis et al. [27]. The latter has also provided quantitative results about the performance of different protocol variants; additional performance studies were done by Gupta et al. [64] and Liu et al. [87]. The reconciliation of different 2PC variants in a federation of heterogeneous servers, especially of the presumed-outcome family, has been studied by Al-Houmaily and Chrysanthis [5]. They coined the term “presumed any” for the protocol sketch described at the end of Section 2.5.3. Wolski and Vei-jalainen [142] as well as Muth and Rakow [99] have investigated ways to emulate 2PC on top of servers that do not by themselves support the participant protocol of 2PC.

### 3.4 Transaction Models

Wrapping operations on data within transactions comes at a significant performance cost. Over the years, industry and researchers alike, have sought techniques to reduce this cost and improve the throughput of database systems to allow applications, both web-based and others, to serve the customer better.

Often many of these operations can span many minutes, or even hours before being completed. Any transaction would have to hold onto locks on individual records for the period with a significant impact on performance and concurrency.
To handle this, a notion of Long Lived Transactions (LLTs) that could be broken up into Sagas were introduced \[53\]. This allowed each individual component of the Saga to be interleaved with other short running transactions to improve concurrency and throughput.

In particular, web-based applications can be modelled as workflows and has been shown to be better handled by Workflow Management Systems (WFMS) than traditional transaction models \[7\]. These systems can be better modelled using Sagas and Flexible transactions \[43\]. Further, Transactional Intent built into an application framework \[52\] has been used to describe and implement an apology-oriented computing framework \[70\] and eventual consistency \[136\] based on ACID 2.0 \[71\] properties. Numerous other approaches to handling web-based application failure have also been described \[63\].

Eventual consistency and other weaker consistency models were introduced in the context of replicated databases by Sheth et. al. \[123\]. A good taxonomy of the correctness criteria can be found in Ramamritham et al. \[113\].

### 3.5 Transactions over HTTP

The REST interface to the graph database, Neo4j \[4\], allows the client to maintain a session with the server to perform transactional updates. Explicit API calls to start, update, commit and abort transactions is used by the client. This approach does not support heterogeneity of data stores and is suitable for accessing a single data store instance only.

Numerous large distributed key-value stores implement transactions on certain sets of items. For instance, Google Megastore \[11\] provides the ability to group multiple objects into static entity groups in order to commit them using a single call. A similar concept is employed by G-Store \[35\] which allows individual records to be included in an entity group. Entities can be migrated from one group to another using a migration protocol.

JEST \[109\], a REST interface to OpenJPA \[^4\] uses a notion of fetch-groups to allow the application to define object closures in a traditional database that can be modified by the application and then updated in a single PUT or POST operation.

[^4]: \url{http://openjpa.apache.org/}
Large-scale, cloud-based, distributed key-value storage services like Windows Azure Storage (WAS) \cite{25} and Google Cloud Storage (GCS) \cite{3} support single item consistent updates and single item read-on-write consistency. This is enabled using conditional operations using If-None-Match request headers that use the object ETag for version comparison. WAS returns an ETag that uniquely identifies the record version. GCS, on the other hand, returns a generation number which is essentially a nanosecond granularity timestamp of the object.

Recent work by Pardon and Pautasso \cite{105} reiterate the limitations of HTTP and the current REST frameworks towards support for transactions. They describe a RESTful implementation of a system that uses a Test-Cancel/Confirm (TCC) pattern along with a coordinator service to provide transactional semantics to web service endpoints. However, their approach uses a central transaction coordinator service to perform the transaction commit using 2PC. The central transaction coordinator service must be highly available and fault tolerant in order for this approach to be usable in typical large scale applications. In the event this is not the case, the transaction coordinator becomes the bottleneck similar to the middleware approach in which the middleware becomes the bottleneck.

3.6 Transaction support in NoSQL and Cloud Storage Systems

In recent years there have been numerous implementations of distributed key-value stores, each exhibiting a different mix of performance, scalability, availability characteristics and alternate architectures. These include Amazon Dynamo \cite{36}, Google BigTable \cite{26}, Yahoo! PNUTS \cite{28}, HBase, and Cassandra \cite{80}, Amazon SimpleDB, Amazon S3. These systems use commodity hardware and exhibit scalability and high-availability but provided lower consistency guarantees, often limited to only eventual consistency \cite{136}. Typically only single item transactions are guaranteed and query capability is limited. Data is accessed using the primary key and data scan support is limited to only a few systems like PNUTS and SimpleDB.

More recently, there have been developments like Spinnaker \cite{114}, Windows Azure Storage \cite{25}, Google Cloud Storage that provide single item consistency
guarantees. The system design focuses on scalability, performance and single
key consistency. Spinnaker uses a Paxos-based protocol to ensure consistency
while COPS \cite{88} and Granola \cite{32} use replication protocol optimisations to achieve
greater performance while supporting native multi-item transactions.

Despite these advances, the bulk of the systems leave multi-item transactional
data access to the client application. This is prone to programmer error and the
results are often completely incorrect.

In order to address these issues, some systems have implemented a relational
database engine to provide the query capabilities and transaction support with
the raw data stored in a distributed key-value store \cite{21}. This is suitable for
applications that require an RDBMS for persistence with the advantage that it
provides a complete SQL interface with full transaction support. The performance
and transaction throughput of the system is limited only by the underlying queue
implementation.

Most applications built using key value stores work well because of the rel-
ative simplicity of the programming interface to the data store. Many of these
applications use a write-once-read-many (WORM) data access pattern to the key
value store and function well under the eventual consistency setting. However,
there are applications that are built to run on the same data that require greater
consistency across multiple keys.

The first approach to address this issue is to implement transactional capabil-
ities within the data store itself. The data store manages the storage as well as
transaction management. The Spanner \cite{30} system from Google is a distributed
data store that supports transactions.

The COPS \cite{88} and Granola \cite{32} systems implement the distributed key-value
store with a custom API to enable applications to transactional access the data
store. Similarly, HyperDex Warp \cite{47} is a high-performance distributed key-value
store that provides a client library that supports \textit{linearizable} transactions. The
client library simplifies the access to the data items on behalf of the application
using an API provided by the data store which maintains multiple versions of
each data item. These systems support transactions across multiple keys with
a distributed, homogeneous key-value store. The focus of these systems is to
build a better, more capable distributed data store and optimise its transaction
coordination.


3.7 Middleware coordinated transactions

Another way is to use middleware to provide caching and transactional data access. Google Megastore [11] is a transactional key-value store built on top of BigTable. Records are collocated in a tree structure, called entity groups, where each leaf record is associated with the root record using a foreign key. The foreign key is used to cluster related records and partition data across storage nodes and transactions across records spanning clusters is done using 2PC.

In G-Store [35], related data items are grouped into key groups that are cached locally on a single node. Transactions are only allowed within key groups and keys are allowed to migrate from one key group to another using a group migration protocol. Greater transaction throughput is achieved because data items are cached on the local node.

Deuteronomy [85] unbundles the data storage and transaction manager into two separate entities. It defines a protocol, which is an optimisation of their earlier work [89], to perform transactional data access using the transaction component (TC) and the data component (DC). This system allows multiple hybrid data stores to be used.

The CloudTPS [144] design uses data store access through a transaction manager split across multiple nodes into local transaction managers (LTM). LTM failures are handled using transaction logs replication across LTMs.

The middleware approach works well when the application is hosted in the cloud and there is a known and controlled set of data stores used by the application. They perform well in this situations and provides one programming interface to the application simplifying the data store access. However, these systems require to be setup and maintained separately.

This is not suitable in our use case where individual application instances need the hands-off, low maintenance features of key value stores and each may have different access privileges to individual data stores.

3.8 Client coordinated transactions

An alternate way of implementing multi-key transaction support for distributed key-value stores is to incorporate the transaction coordinator into the client. We
know of two implementations that use this approach. Percolator \cite{108} implements multi-key transactions with snapshot isolation semantics \cite{12}. It depends on a central fault-tolerant timestamp service called a timestamp oracle (TO) to generate timestamps to help coordinate transactions and a locking protocol to implement isolation. The locking protocol relies on a *read-evaluate-write* operation on records to check for a lock field associated with each record. It does not take advantage of test-and-set operations available in most key value stores making this technique unsuitable for client applications spread across relatively high-latency WANs. No deadlock detection or avoidance is implemented further limiting its use over these types of networks.

ReTSO \cite{75} relies on a transaction status oracle (TSO) that monitors the commit of all transactions to implement a lock-free commit algorithm resulting in high transaction throughput. It utilises a high-reliability distributed write-ahead log (WAL) system called BookKeeper to implement the TSO providing snapshot isolation semantics. Timestamps are generated by a central timestamp oracle. The need to have a TSO and a TO for transaction commitment is a bottleneck over a long-haul network. This prevents this approach to be effective in a WAN layout.

Omid \cite{58} is a tool for lock-free transactional support in large data stores such as HBase. It uses a centralised scheme and implements snapshot isolation, to guarantee that all read operations of a transaction are performed on a consistent snapshot of the data. In contrast to a lock-based approach, in which unreleased, distributed locks held by a failed or slow client may block other clients, its lock-free commit algorithm does not suffer from this problem. Additionally, it lightly replicates a read-only copy of the transaction metadata into the clients where they can locally service a large part of queries.

Due to this, it does not require the modification of either the source code of the data store nor the tables’ schema while the overhead on the data servers is negligible. Their experiments indicate that the implementation on a dual-core machine can be used to service up to a thousand client machines. It scales up to 124K write transactions per second while the increase in latency is just 10 ms. This, the authors claim, is well above the transaction rates supported by existing large data stores, making it suitable for application to these systems without becoming a bottleneck.
This approach uses a central Transaction Status Oracle (SO) similar to the ReTSO. It keeps track of all the objects that have been committed. As such, it is suitable for a controlled environment where all the clients are able to discover the TSO and use it to order transaction commits.

3.8.1 Comparison to our approach

Our approach in this thesis is similar in many ways to Percolator and ReTSO. We use the transaction start time to obtain the transaction read set. We also use the transaction commit timestamp to tag all the records that belong to the transaction write set. Unlike these systems ours does not depend on any centralised timestamp oracle or logging infrastructure. We utilise the underlying key-value store and its features to provide transaction across multiple records. There is no need to install or maintain additional infrastructure. Our approach enables transactions to span across hybrid data stores that can be deployed in different regions and does not rely upon a central timestamp manager. In the current version, we rely on the local clock to keep time but it is compatible with approaches like TrueTime [30] and HLC [77]. We use a simple ordered locking protocol to ensure deadlock detection and recovery without the need of a central lock manager.

3.9 Time in Distributed Systems

During the 1970’s the advent of networks enabled computers to be connected to each other over network giving rise to the first distributed computing systems. Many applications depend on the ordering of events in order to function properly. This gave rise to the requirement for a reliable way to determine timestamps and ordering across these federated systems.

This involves synchronisation of clocks across the nodes in the distributed system as discussed by Liskov [86]. The most well known approach was proposed by Lamport [81]. It uses the partial ordering of events across a distributed system using logical timestamps also called Lamport Clocks. A distributed algorithm is used to synchronise the system of logical clocks so that it can be used to totally order events. This approach was then used to define bounds on the skew across clocks in a distributed system.
The Time Protocol RFC868 [111] and the Daytime Protocol RFC867 [110] are examples of early approaches to synchronising clocks across distributed systems. These were superseded by the Simple Network Time Protocol (SNTP) [95] and the Network Time Protocol (NTP) [94]. These are based on techniques such as Cristian’s algorithm, Berkeley algorithm, and Marzullo’s algorithm. For instance, NTP achieves time synchronisation using a hierarchy of NTP servers to provide time. It removes outliers using a statistical approach and then determines the clock adjustment to be made on the node.

Cristian’s algorithm [33] uses a probabilistic approach to determine the local time using the remote server time and the network round trip time (RTT). This is done using the formula $T_{local} = T_{server} + RTT/2$, where $T_{local}$ is the resulting local server time, $T_{server}$ is the time at the server, and $RTT$ is the measured round trip time. The Berkeley algorithm [65] makes use of a master node that is chosen using a leader election process followed by a time synchronisation phase that involves determining the average clock time across all the slaves using Cristian’s algorithm which is followed by pushing out a positive or negative adjustment to each node depending on its detected level of time skew. Marzullo and Owicki [91] proposed an algorithm to efficiently determine reliable sources of time from a number of sources in order to more reliably calculate time in a distributed system.

Google Spanner uses TrueTime to determine time across the network. It depends on high-fidelity atomic clocks and GPS clocks as a source of time and a variation of Marzullo’s algorithm [91] to calculate the error margins. Google Percolator [108] on the other hand, uses timestamp oracle (TO), a central timestamp issuing entity as the source of time. Every application that needs a timestamp sends a request to the TO which responds with a timestamp.

In practice, the TrueTime implementation at Google is able to achieve a sustained error bound of 6 milliseconds. This is not sufficient for large distributed applications spanning many machines over heterogeneous network with widely varying latencies. Hardware Logical Clocks [77] combines the concept of Logical Clocks from [81] and applies it to hardware clocks by ensuring monotonic hardware clock behaviour. This is being used as the transaction ordering in an open-source implementation of Spanner called CockroachDB [79]. It addresses some of the limitations of TrueTime and is suitable for deployment without specialised hardware.

It is clear that there is a lot of interest and recent activity in the area of time
management in large distributed systems. This is an even more interesting problem
due to the growing popularity of virtual machine and container technology that
are increasingly being deployed using cloud infrastructure as a service.

\subsection*{3.10 Database benchmarks}

A large body of work has been done by the database community, both in the
industry and academia, in the area of evaluating database systems and their per-
formance and other characteristics. For instance, a survey and taxonomy for
approaches to measuring and monitoring consistency is given by Bermbach and
Kuhlenkamp \cite{13}.

These benchmarks are designed to measure the performance of traditional
OLTP databases systems. The assumption is that the system under test (SUT)
is a traditional relational database system which fully supports transactions elim-
inating the need to measure consistency. In addition, they do not really test for
elasticity, scale-out characteristics and fault-tolerance. These features are tested
by newer cloud benchmarks.

\subsection*{3.10.1 Cloud services benchmarks}

Binnig et al \cite{18} made the case that traditional database benchmarks (like the TPC
suite of benchmarks) are not sufficient for analysing modern cloud services. They
proposed a new benchmark that evaluates the characteristics of these systems
such as scalability, cost, pay-per-use and fault-tolerance. Further, they proposed
that the benchmark should focus on the complete cloud services stack instead of
relying on micro-benchmarks to evaluate individual tiers like application servers,
message systems and the storage layer. In contrast to the TPC-W benchmark,
they proposed that the benchmark for the cloud should define additional Web 2.0
like interactions and emulate modern technologies such as AJAX which change
modern web application access patterns.

Since cloud services are typically designed to scale linearly, they proposed that
scalability should be measured in number of web interactions (WI) per unit time
(seconds) in a given response time interval (WI per second i.e. WIPS). Like the
TPC-W benchmark, they propose a measure of cost of operation in dollars per
WIPS ($/WIPS). Another important property identified was the ability of cloud systems to adapt to peak loads. To measure this, they propose a way to maintain a sustained load and introduce periods of increased WIPS to simulate spikes. Lastly, they recommend that the performance and behaviour of the system under test (SUT) be evaluated in the presence of failures.

Later, Kossman et al. [76] used a TPC-W workload to evaluate a number of cloud-based database systems for OLTP applications. The focus of the benchmarking is to measure the end-to-end performance and cost of running enterprise web applications with OLTP workloads on alternative cloud services. They found that the various alternative services available at the time varied significantly in both cost and performance. In addition, they reported significant scalability issues particularly in situations where the system is overload. They also found that the different vendors had different business models that target different kinds of applications: Google seems to favour small applications with light workloads while Azure was the most affordable service targeted towards medium to large applications. They were able to confirm that these systems lacked support to upload large data volumes. For instance, it was found that it was difficult to upload a 1 TB or more of raw data through standard APIs provided by the vendor. However, the study failed to conclusively determine what is the right database architecture for cloud computing. This can be attributed to the immaturity of the technology and market place at the time.

3.10.2 YCSB

During the same time, a new benchmark, called YCSB, was developed by Cooper et al. [29] at Yahoo! Research to evaluate web-scale data management systems, especially the available-despite-failures key-value stores in the NoSQL category. The focus of this benchmark is raw performance and scalability; correctness is not measured or validated as part of the benchmark, and the operations do not fall within transactions (since these systems may not support transactions nor guarantee data consistency). YCSB is actually a flexible framework within which the workload and the measurements can be extended.

Both YCSB and the benchmark proposed by Bennig et al. [18] are focussed on evaluating cloud services as opposed to the SPECweb2009 [129] benchmark
which is designed to evaluate web servers. This benchmark is designed to evaluate web servers that implement banking, e-commerce and support applications and not modern cloud services. The evaluation is done to measure the performance and power utilisation for SSL and non-SSL connections for both small and large downloads.

Neither of the above cloud and web server benchmarks provide the ability to define explicit transactions to group operations. Further, they do not explicitly evaluate the correctness of the system under test (SUT); neither do they measure nor quantify anomalies introduced as a result of running the workload. However, the number of erroneous operations and the rate of these occurring are captures and measured.

An extension of YCSB called YCSB++ [106] is designed to evaluate non-transactional access to distributed key-value stores. It adds functionality to enable bulk loading of data into HBase and Accumulo in the form of an extended API. It also allows operations like B-Tree splits to be performed on the database indexes to simulate application usage. Other useful features include the ability to launch YCSB clients from multiple nodes and to coordinate readers and writers to simulate complex read-after-write application scenarios.

Important work from UCSB [45,107] has extended YCSB for evaluating novel systems that support transactions. These measurements are performed by modifying the DB client and encapsulating each database operation in a transaction within the methods of this class. This is suitable for measuring the performance of the system but it does not provide an ability to measure the new tiers (the transactional overhead of individual database operations, nor is the consistency measured) that we propose.

While not building on YCSB, there have been other researchers that have offered benchmarks for consistency properties in data platforms. In the context of a traditional (centralised) database, Shasha and Bonnet [122] measured the number of read operations that do not return the correct latest data.

For clouds, Wada et al. [137] measured the probability of returning stale values, as a function of how much time had elapsed between the latest write and the read.

Our approach here, of measuring the extent to which the data has deviated from a consistent state, follows that used by Fekete et al. [48] for a centralised database.
A different approach to measure consistency is found in Zellag and Kemme [143] where the execution trace is captured, and the non-serializable executions are detected by looking for cycles in the dependency graph.

3.11 Chapter Summary

In this chapter we surveyed the existing body of research and related work in the area of transactions in distributed database system. We began by looking at Cloud and Web Services. We then gave an overview of related work in Distributed Concurrency Control and Distributed Transaction Recovery.

Later, we looked at the various Transaction Models proposed by various groups of researchers and looked at Transactions over HTTP in detail. This section on Transaction support in NoSQL and Cloud Storage Systems explored systems related to our work. In particular, alternate architectural approaches were discussed in the section on Middleware Coordinated Transactions. The section on Client-coordinated Transaction discussed approaches employing similar techniques as ours. This is followed by a discussion of the complexities of determining Time in Distributed Systems.

Finally, we discussed the related work in the area of database benchmarking in detail. The discussion later focussed on Cloud Services Benchmarks after which the Yahoo! Cloud Serving Benchmark (YCSB) which we extend in Chapter 7 was discussed.
Chapter 4

REST+T: Scalable transactions over HTTP

Modern network technology has revolutionised the way even simple household devices are used to a point where everything is connected to some network or another. Rapidly changing hardware and link layer protocols have ensured that connectivity is getting cheaper, faster and more reliable by the day. The ability for higher level transport and application layer protocols to remain the same while continuing to work with this rapidly advancing technology stack is the primary reason for the rapid growth of applications that rely on the internet in order to function efficiently.

This has brought a significant change in the mindset of architects, engineers and system developers, to the point that the Software-as-a-Service (SaaS) approach to making programs available for use is rapidly gaining popularity. The key to the success and popularity of SaaS is HTTP (and HTTPS) and complementary protocols like WebSockets \[49\] along with JavaScript and AJAX. The simplicity and ubiquitous use of HTTP has made it the corner stone of service-oriented applications that support HTTP-based API. In addition, the ready availability of a wide variety of robust application server technologies along with ready-made and well understood frameworks for security, authentication, authorisation and web-UI development have made it an even more compelling choice. This is further aided by the more robust and rich programming capabilities available within the web-browser and client side technologies like HTML5, CSS3, JSON and JavaScript and frameworks like Twitter’s Bootstrap and Google’s AngularJS.
While HTTP (and HTTPS) have gained popularity and wide-spread use, it has primarily happened in the context of making data available. The ability to deal with single endpoints or resource made available as URIs (either URLs or URNs) is widely and easily understood. Interacting with each endpoint as a single isolated resource is typical, while interacting with multiple endpoints in the context of an application transaction, is becoming more common.

While most applications do not need to coordinate application calls with more than one endpoint at a time, however, there are situations when there is such a need. At this point, the way to deal with this is to use application-specific approaches to handle multiple updates. Failure scenarios are dealt with in an ad-hoc manner, on a case by case basis.

Traditional approaches to handling transactions are targeted towards enterprise application development where it is a controlled environment. Administrators manage access to applications and deployment of software. In this scenario, techniques developed to use SOAP-based technologies like the WS-* suite of protocols is suitable. These include the Web Services Atomic Transactions and related protocols. However, the assumptions around a controlled enterprise setting is rapidly changing. The advent of micro-service oriented application development and the proliferation of open-source technologies and NoSQL systems is beginning to challenge this assumption of control over the choice of technology within the enterprise.

Overall, dealing with multiple resources in a consistent transactional manner has been ignored and largely left to the application developer. As data persistence services like databases and data stores like Amazon’s Simple Storage Service (S3), Microsoft’s Windows Azure Storage (WAS) and Google’s Google Cloud Storage (GCS) become increasingly popular, using HTTP as a way to store application state in a data store that is essentially a Software-as-a-Service with an HTTP API is becoming increasingly frequent.

As the question of reliable data storage is being addressed by these systems, it is becoming clear that the ability to manipulate and maintain application state in these data stores in a reliable, consistent and transactional manner is becoming increasingly important.

In this chapter, we look at an extension of HTTP, we call REST+T (for REST with Transactions) \[39\], that natively supports transactional access to generic
web-service endpoints, resources and data stores in particular. We begin with a brief overview of the space and the challenges we had to overcome, followed by a description of the protocol. After this, we discuss each extended HTTP method in detail and discuss application scenarios where they should be used. This is followed by an simple example application to illustrate its use in real-life scenario.

4.1 State and state change

Traditionally, applications have used databases to store application data state. For instances, a banking application may maintain the state of each bank account in a database in the form of its current balance and the last ten transactions.

Similarly, modern applications use REST endpoints to store application state. As the application goes about performing its designated tasks, it interacts with the REST endpoints to update the state with appropriate requests. While many operations may involve interacting with a single endpoint, there are situations when more than one endpoint is updated, via multiple individual state changes, as part of a single application-defined operation. A simple example of this is the deduction of a sum from one bank account and addition of the same amount to another bank account, as part of a funds transfer from one account to the other which is exposed via an HTTP web-service API.

Data items stored in key-value stores are examples of endpoints that keep application state. For this reason, throughout the rest of this chapter, we use the term data item interchangeably with the term endpoint.

It is crucial for many applications to ensure that the changes to state across multiple data items occur in a transactional manner. This is particularly challenging when there are multiple concurrent applications attempting to make simultaneous updates to the same set of data items. Transactions are a suitable programming abstraction that enables application developers to ensure that the applications do not step on each others toes while executing concurrently.
4.2 Challenges

HTTP has made it incredibly easy to develop new applications as a result of the wide-spread use of web-services to expose software APIs. This service-oriented approach has significant advantages. Security, authentication, authorisation, integration, portability, scalability, load-balancing and fault-tolerance is well understood and successful deployment architecture and implementation patterns have been developed over the years having stood the test of time. In addition to this, the proliferation of REST as an approach to building APIs has made it easy to develop new applications through “mash-ups” of multiple services. An example of this may be a photo sharing application that uses a photo storage service like Flickr\footnote{\url{http://www.flickr.com}} and a social media application like Facebook\footnote{\url{http://www.facebook.com}} making it possible to share photos with friends.

This is a very powerful capability, both Flickr and Facebook are highly scalable and reliable systems that provide APIs that applications residing anywhere on the internet can access regardless of the software or hardware platform. There are literally thousands of different types of software exposing APIs that have been developed. This heterogeneous set of ever-growing ubiquitous APIs have given rise to a plethora of mash-ups in the form of web and mobile applications.

As the variety of applications grow with time, we are beginning to see the
need for these APIs to increasingly display stateful, dependable behaviour. For instance, Amazon Web Services provides the ability to store large blocks of data in the form of BLOBs (Binary Large OBjects) in their Simple Storage Service (S3). S3 is known to be an eventually consistent system but it is widely used as a way to store data using File System abstractions for Big Data applications like Hadoop Distributed File System (HDFS). This is done by using the REST API underneath a Java library that exposes this File System interface.

Owing to the limited guarantees, a result of its eventually consistent behaviour, the file system interface provided is not able to provide robust semantics that the application can depend on. For instance, it is common to find that large writes to a file are not fully visible by other reader applications. A recent developed file system called Elastic MapReduce File System (EMRFS) uses DynamoDB to store file system metadata in order to guarantee traditional HDFS semantics.

An example of how HTTP with conditional PUT/POST operations to achieve “transactional” behaviour is illustrated by the example in Figure 4.1. Transaction $t_1$ reads (using GET) two records A and B. It then increments A and decrements B. It then writes (using PUT) record A using a condition that the record A has not been modified since it was last read by $t_1$ using a If-Match HTTP header with the ETag received with response to the original GET operation. This is then done with the other record B as well. Transaction $t_1$ succeeds. However, transaction $t_2$ attempts to write record A with a condition that it has not been modified since it was read. This fails because $t_1$ has already successfully written to it.

Compensatory operations are used when part of a transaction is successful and the remaining operations are unsuccessful. This is essentially an attempt to “undo” the successful operations. However, this approach has limitations and does not always work and can lead to anomalous behaviour.

Figure 4.2 illustrates one such situation. Application 1 starts transaction $t_1$ to read records A and B each with value 10. In about the same time, Application 2 starts transaction $t_2$ and reads records B and C each with value 10. Transaction $t_1$ modifies records A and B and $t_2$ modifies records B and C. Once this is done, $t_2$ conditionally writes the modified values of records B and C. At this point $t_1$ conditionally writes record A with its new value 11. Soon, Application 2 starts another transaction, $t_3$, which reads records A and C. This is when $t_1$ attempts to write record B but the precondition fails. Transaction $t_3$ is now ready to write
the modified values of records A and C and does so successfully. Transaction $t_1$ in the mean time, realises that $t_1$ is partially successful and attempts to undo the operation to write record A. Since, transaction $t_3$ has already overwritten record A with value 12, the undo operation for $t_1$ fails leaving the data in an inconsistent state.

It is expected that the number of these types of applications of web services will continue to grow steadily. They will require better correctness guarantees and more robust behaviour without making sacrifices to the scalability and scale-out characteristics they heavily depend on. In addition, fault-tolerance and load-balancing will continue to be needed. It is clear that any middleware or coordinated approach to providing better guarantees or transactional semantics will not scale. The reasons for this is that managing access to these systems in a scalable and efficient manner is non-trivial, error prone and requires the transaction coordinator to have a priori knowledge of all the applications that will access the data.
The common approach by applications to handle the lack of transaction guarantees is to use operation retries. On failure, compensatory operations are used to undo successful operations that are part of the same transaction. This approach is hard to program against. A programming abstraction that provides transactional access is needed. In order to do this successfully and easily, the API supported by the endpoint must natively support transactions.

### 4.3 REST with Transaction support

The REST way of implementing services provides the ability to keep state at the endpoint and not at the client. This makes it possible for multiple application instances to simultaneously operate on the data item without compromising consistency. This is typically done using a test-and-set operation employing the use of the `If-Match` HTTP request header and the record ETag as the header value.

Database systems have used the 2-phase commit protocol (2PC) effectively in production systems for many years now. We use this time-tested approach as a basis of our extensions to REST over HTTP and optimise it to remove some of its drawbacks.

In essence, each data item is treated as though it were a single record database. A client-coordinated transaction commit process is performed across multiple data items. In order to enable this, we extend the HTTP API with five new methods: PREV, PREPARE, COMMIT, ABORT and RECOVER.

We are able to support non-blocking, optimistic transactions guaranteeing snapshot isolation semantics using these extended HTTP methods, additional HTTP headers and test-and-set operations.

### 4.4 The REST+T Proposal

In this section, we define the REST+T protocol and the extensions to the set of standard HTTP protocol methods. Figure 4.3 describes the state transitions that occur during the process of committing a transaction on each data items as part of a multi-item transaction. We use the same event-condition-action rules (ECA) described earlier in Chapter 3 Section 2.5.1 State transitions are labeled in the
form “event [condition]/action,” where each component of the triple is optional and committed when insignificant. As described earlier, initial states (labeled “Initial”) are indicated by incoming edges with a small point as origin; final states have no outgoing edges (the DELETED state).

In a typical REST+T deployment, requests from the client would be received by the server in the form of REST+T requests. In this statechart diagram, the implicit condition for each state transition is that all the conditions specified by any conditional request headers (such as If-Match) be met in order for the request to succeed. The use of the conditional operation headers is a way to guarantee single-item transactions. This method must be used in order for REST+T to guarantee transactional behaviour is a situation where multiple items are manipulated within a single transaction. On failure to meet the specified condition the appropriate error (typically, error code 412 — Precondition failed) is returned leaving the record in its current state.

Earlier in this chapter, Figure 4.1 illustrated the use of HTTP and conditional
updates using the `If-Match` header to successfully perform transactional updates to multiple items. In the example illustrated in Figure 4.2 we observe that anomalous behaviour is possible even with the use of conditional updates using techniques described above.

In Figure 4.4 we use the same example as Figure 4.1 to show the method exchange with the endpoint to perform transaction updates to records A and B. Transaction \( t_1 \) performs a PREPARE on both records A and B which succeed before committing the transaction by calling COMMIT on each record. In the mean time, the call to PREPARE record A by \( t_2 \) fails resulting in its failure.

Transaction \( t_1 \) exchanges two extra method calls in order to properly commit the transaction across records A and B. However, transaction \( t_2 \) is able to detect the failure on the call to PREPARE record A.

In Figure 4.5 transactions \( t_1 \) and \( t_2 \) both read records A and B. However, transaction \( t_1 \) calls PREPARE on record A while transaction \( t_2 \) calls PREPARE on record B. Both these calls succeed causing transaction \( t_1 \) to proceed to call PREPARE on record B and \( t_2 \) to call PREPARE on record A. Subsequently both these operations fail causing both transactions \( t_1 \) and \( t_2 \) to fail. The applications 1 and 2 must call the ABORT method on records A and B respectively in order to rollback the changes.
We will see how the Cherry Garcia protocol reduces the occurrence of these type of transaction failure scenarios by ordering PREPARE method calls in a deterministic fashion in Chapter 5 Section 5.11.

The nature of anomalies introduced by the interaction of Application 1 and Application 2 illustrated in Figure 4.2 is avoided when using REST+T. Figure 4.6 show that transaction $t_2$ successfully calls COMMIT on records B and C. This prevents the call by transaction $t_1$ from calling PREPARE on record B. However, transaction $t_3$ is not able to PREPARE record A because it is already in the PREPARED state due to transaction $t_1$. When transaction $t_1$ attempts to PREPARE record B it fails because it is already in the COMMITTED state due to transaction $t_2$. This causes Application 1 to call ABORT on record A leaving records A, B and C in a consistent state.

4.4.1 REST+T data header metadata

In addition to the standard HTTP headers passed between the client and the server with each REST+T request and response respectively, we define additional header in order to maintain sufficient transaction state of the data item along with the data. These additional headers help the client communicate the transaction state of each record it operates on as part of the application-defined transaction.
4.4. THE REST+T PROPOSAL

Figure 4.6: Two applications displaying correct behaviour using REST+T
**ETag:** Every data item has a version tag associated with it. This is unique for each version of the object and can be implemented as a timestamp or a unique hash code for the object. This tag can be used to perform test-and-set operations on the data store using the If-Match HTTP request header.

**Transaction-Id:** This specifies a transaction identifier, typically a URI, that can be used to identify the application defined transaction that updated the object version. The URI structure and content is application defined and is not mandated by the REST+T protocol. The intention behind having this header field is to enable other concurrent and subsequent application transactions to be able to identify the transaction that modified the data record and determine its final outcome.

**Transaction-State:** This specifies the state of the data item as defined by the state diagram in Figure 4.3. It can be either PREPARED, COMMITTED, or DELETED. An object is readable when it is in the COMMITTED state. An object in the PREPARED state must be recovered and then rolled back or rolled forward and committed. Records marked DELETED must be garbage collected.

**Valid-Time-Start:** This header field specifies the timestamp from when the version of the data item is readable by a client. This is set by the PREPARE method.

**Valid-Time-End:** This is the timestamp after which the version of the data item is not readable by any client. This is set by the DELETE method which is then followed by the COMMIT method. This value of this header is set to zero (signifying an invalid end time) for currently valid versions of the object in the COMMITTED state.

**Lease-Time:** This header specifies the timeout after the timestamp specified by the Valid-Time-Start header when the data item must be recovered by the client to be either rolled back to the previous COMMITTED state or rolled forward and set to a COMMITTED state from the current PREPARED state.
4.4.2 REST+T methods

REST+T extends the standard HTTP methods in order to enable the application to change the state of the data items and to obtain the latest version from the data store. It must be noted that REST+T does not alter the behaviour of standard HTTP GET, HEAD, PUT, POST, PATCH and DELETE when used in HTTP mode, i.e., when using standard HTTP headers.

The following is a list of these methods and a detailed description of each:

**GET:** The GET method returns the data item along with header information in the form of an ETag the uniquely identifies the version of the data item along with the standard HTTP headers. The actual contents of the data can be application or service specific and REST+T does not impose any restrictions on it. For all practical purposes, this method is identical to the standard HTTP GET method.

In order to indicate whether the object returned is a valid version, the `Transaction-State` header is returned along with the response to indicate whether the object has been successfully committed to the data store. If the `Transaction-State` header is set to COMMITTED the object is consistently written. If set to PREPARED, it is not properly committed and needs to be recovered and either rolled back or forward depending on whether the transaction was successfully committed or not.

**PUT/POST:** The PUT and POST methods behave exactly as defined by the HTTP protocol. However, to maintain data consistency, an application should not, in general, use these methods to update data; instead the PREPARE method must be used as described below. In the case where a transaction updates only a single item, the PUT or POST methods can be used, writing back with a conditional update header to ensure that the changes are done only if the item’s ETag is unchanged. This is equivalent to the one-phase commit optimisation in 2PC.

**PREPARE:** When there are more than one items to be updated, each item is first written to its respective data store using the PREPARE method and a conditional update header. If successful, the data item is moved to the
PREPARED state and subsequent PUT, POST or PREPARE operations are permitted on the object.

The application instance that prepared the data item is identified by the Transaction-Id HTTP header. This transaction identifier is in the form of URI that identifies the transaction.

To ensure that an object does not remain in the PREPARED state indefinitely, the PREPARE method takes a Lease-Timeout header. This ensures that the upper bound on the time an object remains in the PREPARED state.

GET operations on an object that is in the PREPARED state returns the last written version of the object when the lease time has not expired. If the lease time expires, the prepared version of the item is returned with the Transaction-State set to PREPARED. This is used by the client to recover the transaction based on the state of the transaction identified by the Transaction-Id header.

In addition, the PREPARE operation sets the object Valid-Time-Start header to indicate the timestamp from when the transaction can be considered to be committed. A Valid-Time-End header is used to indicate the time when the data item is no longer valid. If the object is already in the PREPARED state, the operation fails with the HTTP error code 412 (Precondition failed).

**COMMIT:** Once all the objects have been prepared to their respective data stores, their changes can be made permanent using the COMMIT method on each object. Only previously prepared objects can be committed using this method.

**ABORT:** If a client is not able to prepare all items involved in the transaction, it must call the ABORT method on the already prepared items in order to rollback the changes. This makes the item available for updates to other clients. A version tag and a conditional update header is used for this operation to ensure that ABORT operation is performed on the version of the object intended by the client.

**RECOVER:** If a PREPARE operation fails with error code 412 because the
object is already in the PREPARED state, the client may choose to inspect
the version of the object in the PREPARED state to see if it needs to be
recovered using a RECOVER request.

This is used to perform lazy transaction recovery in the situation when a
client dies without completing the COMMIT phase. The URI set in the
Transaction-Id header is used to discover the fate of the transaction. If
the URI points to a valid endpoint with the Transaction-State header set
to COMMITTED, the transaction must be rolled forward by calling the
COMMIT method on the item, else if it does not exist or if the Transaction-
State header is set to DELETED then the ABORT operation must be issued.

**PREV:** If a GET operation returns an object whose Valid-Time-Start is later
than the start time of the application transaction, it should read a previous
version of the object. The PREV method is used to do this. A conditional
HTTP header is used using the object version tag to fetch a valid version of
the data item with respect to the version previously read.

**DELETE:** A DELETE call removes a data item from the data store. Unlike the
typical implementation of DELETE in standard HTTP, this operation per-
forms a logical delete by setting the Valid-Time-End to the transaction com-
mitment time and setting the item Transaction-State header to DELETED.
Deleted items are garbage collected after the timeout set using the Delete-
Timeout header expires.

### 4.5 Limiting REST+T to HTTP verbs

In the previous sections we described REST+T and the extensions to the standard
set of HTTP verbs with PREV, PREPARE, COMMIT, ABORT and RECOVER.
This extends standard HTTP verbs with five additional methods to implement.
This can be a severe hindrance for existing web browsers, application servers,
HTTP client libraries and other security and related software that depend on this
existing pattern.

However, the additional methods are explicitly defined as new methods for
better illustration of the protocol extension and a clearer understanding. All five
methods, can use request query parameters or path extensions to achieve the same thing. For instance, instead of the PREPARE method the system could implement a PUT method with a query parameter called method and assign it a value PREPARE as follows:

```
PUT /data/item?method=PREPARE HTTP/1.1
Host: tora
```

This will enable existing HTTP client libraries such as the Apache HttpClient[^3] to function seamlessly with the REST+T API without requiring extension.

Another approach to providing this is to use a custom HTTP headers to specify the method. The HTTP request for the PREPARE would look like the following:

```
PUT /data/item HTTP/1.1
Host: tora
Method: PREPARE
```

We chose to go with the former approach because the method calls would be explicit and would help us explain the extensions more easily. However, this requires us to implement additional handlers in the standard HTTP server as well as define additional method types in the Apache HttpClient implementation as described in Chapter 6 Sections 6.6 and 6.7, respectively.

Implementing REST+T as HTTP header extensions or via query parameter passing would have been easier to implement in existing HTTP servers and would need no change at all to the Apache HttpClient code. However, this approach would be harder to understand and debug.

### 4.6 A Case Study - Tora: a transaction-aware NoSQL data store

In this section we illustrate the REST+T approach by describing the implementation of Tora[^4], a high-performance distributed key-value store that provides a REST+T interface. Tora has three primary components, the REST+T interface, [^3]: http://hc.apache.org/httpclient-3.x/

[^4]: Tora is the word for tiger in the Japanese language.
the transactional data storage system and the distribution and fault-tolerance layer.

Here, we focus on the REST+T interface. The transactional data store is an existing system called WiredTiger described in further detail in Section 4.6.2 later in the chapter. The distribution and fault-tolerance of the storage system is beyond the scope of this thesis.

We have developed a high-performance key-value data store that will have native support for 2-phased updates and deletes without compromising the scalability and availability characteristics of typical distributed NoSQL data stores. We are of the opinion that these extensions to the traditional API with the operations; PREV, PREPARE, COMMIT, ABORT, and RECOVER in addition the standard GET, PUT, and DELETE methods will enable the client-coordinated transaction commitment protocol to work more efficiently by reducing the number of API calls to the data store while continuing to support traditional non-transactional access using only the basic set.

Tora is a key-value store that implements this API to store blobs addressable using a key. It is written in C++ using the Boost ASIO library (used to achieve asynchronous I/O operations) to implement the socket and HTTP interface and uses the transactional capabilities of WiredTiger, a high-performance embedded key-value store, to persist data in a transactional manner.

We take a more in-depth look at the implementation of Tora in Chapter 6 Section 6.6.

### 4.6.1 REST+T interface

The REST+T implementation requires just 2407 lines of C++ code in 16 source files using the Boost library. This included all the REST+T request and response handling and network I/O. The current version does not support any security measures to perform authentication and authorisation of request. However, with a bit of code restructuring, it should be fairly easy to plugin any third-party mechanism. Chapter 6 Section 6.6 describes this in further detail.

The implementation obtains the system time at microsecond granularity using the `gettimeofday(2)` function. The obtained timestamp is used to generate the ETag and Last-Modified response headers.
4.6.2 Storage layer

We use WiredTiger \[140\] to store the data items on disk. WiredTiger is a high-performant key-value store providing transactional access to data stored on disk with support for snapshot isolation semantics. It uses a log-structured storage layout taking advantage of lock-free data structures to enable high transaction rates even with higher degrees of concurrency.

4.7 Chapter Summary

In this chapter we looked at the REST+T extensions to HTTP that enable web and service-oriented applications to be written reliably against web-service and other endpoints. We described the challenges faced with current HTTP based approaches to provide transaction guarantees and propose a set of additional HTTP methods that enable an HTTP interface to be used for better transaction guarantees. In Chapter 6 we describe the implementation of Tora in detail and show how easy it is to extend the Apache HttpClient library to support these additional methods. Finally, the REST+T API is evaluated against standard HTTP with conditional writes using a simple micro-benchmark in Chapter 8.
Chapter 5

Cherry Garcia - The Protocol

In this chapter, we describe the design of Cherry Garcia, a client-coordinated transaction processing protocol, that enables application defined transactions involving multiple data items that may reside in separate, possibly heterogeneous, data store instances. Applications can use a library implementing the Cherry Garcia protocol to access one or more data items stored in one or more heterogeneous data stores with transactional semantics.

The library implementing the protocol exposes an API that abstracts the data store instances as a class called Datastore. The applications access data items in the data stores using this interface via a transaction coordinator abstraction, a class called Transaction.

Each data record is addressable using its key, and its value can be accessed using an object of a class called Record. For simplicity, we assume that keys identifying data items are strings. However, in practice the key can be extended to support other simple or composite types.

We begin this chapter by describing the challenges in providing transactional access to multiple data items that reside across distributed data stores in Section 5.1. Next, in Section 5.2 we describe the intuition behind our proposed approach. This is followed by Section 5.3 in which we describe a typical user application that performs operations on multiple data items. In Section 5.4 through Section 5.10, we define the Cherry Garcia protocol, define the prerequisites, and describe it in detail. Later, Section 5.11 describes techniques used to detect and avoid

\footnote{Cherry Garcia is a name of a Ben & Jerry’s ice-cream flavour with heterogeneous aspects of chocolate and fruit.}
deadlocks between concurrently transactions. Next, optimisations to the protocol are discussed in Section 5.12. While Section 5.13 covers different failure scenarios possible and how the protocol handles them. In Section 5.14 we provide a sketch of a proof of correctness of our algorithm. We discuss approaches to extending Cherry Garcia to implement fully serializable transactions in Section 5.15 and list the algorithm for one of them. Finally, in Section 5.16 we discuss the correctness of this proposal.

5.1 Challenges

As described in Section 2.7 in Chapter 2, modern distributed NoSQL data stores are designed with scalability and high availability in mind. The distributed architecture enables the data items to be spread across the storage nodes in the cluster using some form of data item key to node mapping. In order for such an architecture to perform well for item lookups, there must be little to no coordination across the actual nodes that store the data. In addition to this, the assignment of data items to nodes may change over time depending on various factors including; the number of nodes in the cluster, the data placement and load balancing algorithm, and actual number of records stored in the cluster.

Support for transactions across multiple data items requires coordination of more than one node in the cluster for every transaction. CloudTPS \[144\] for instance, implements a key migration protocol in order to ensure that the Local Transaction Manager (LTM) can locally coordinate transactions across multiple records in a key-group on the same node. This does not scale as the number of transactions increase and the number of nodes in the storage cluster grows as a result of growing data.

The sources of this scalability bottleneck are:

- Storage space: the need to keep extra transaction state for each data record
- Network communication: the messaging overhead of transaction protocol coordination

In order to avoid these performance issues, these systems typically provide lower transactional guarantees on data items stored in it. For instance, Amazon’s
Simple Storage Service (S3) provides only eventually consistency; essentially, there is no guarantees that when a record is written to the data store, its latest version will subsequently be read by the same or another application, particularly when the system is being actively updated. Eventually, when the system settles down, the value will be propagated to all replicated storage nodes ensuring that all readers will see the same value.

A slightly, higher level of transactional support is called Timeline Consistency \cite{28} where the system guarantees that at any point, a reader will not get an older version of the data it read in a previous read operation.

While these are weak guarantees, in reality, they often work quite well. This is particularly true for write-one-read-many (WORM) applications like web-content delivery and infrequently updated data.

However, in recent years, new systems have begun providing higher transactional guarantees on single data item updates for the data stored in them. Two examples of commercially available systems with single data item transactional access are Google Cloud Storage (GCS) and Windows Azure Storage (WAS). Other research prototypes and open source systems with similar capabilities have also been developed.

There are various approaches to implementing transactions across multiple data items. Broadly, these can be classified into three categories.

The first implements transaction support in the distributed data store infrastructure itself. It is more suitable for homogeneous systems and makes it possible to implement performance optimisations otherwise harder to implement in heterogeneous systems.

The second involves implementing the transaction coordination in the middleware between the application and the data store. This can cause the middleware to be the performance and scalability bottleneck depending on the middleware and it architecture. This is suitable for access to heterogeneous data stores even though the inclusion and exclusion of data stores can have significant procedural overhead.

The last technique involves coordinating transactions in the client application. We use this technique to implement transactional access to multiple data items in heterogeneous data stores.
The techniques described here are discussed in closer detail in Chapter 2 Section 2.8 and Chapter 3 Section 3.6. In the remaining part of this chapter we describe our solution in further detail.

5.2 Intuition

As we have seen in the previous section, each data item is individually updated in a coordinated manner from one consistent state to another. To support multi-item transactions, all we need to do is to make sure that either all the items are updated or none of them are in one atomic step. This requires coordinating the updates across all the data items involved in the transaction to be performed simultaneously. This is difficult to manage considering failure cases.

In order to achieve this, we consider each data item as a single item database and then coordinate the transactions using the client as the transaction manager. This idea is borrowed from the 2-phase commit distributed transactions commitment protocol described in detail in Chapter 2 Section 2.5. Figure 5.1 depicts the different possible states of the transaction and each data record in each data store. We use the same event-condition-action rules (ECA) used in Section 2.5.1.

5.2.1 Overview of Two-Phase Commit

In a centrally managed system, the responsibility of transaction coordination is left to a single transaction manager (TM). The TM achieves this by executing the commitment logic in response to the application’s request to \texttt{commit()} the transaction. So that this can happen, the first task is to determine all the participating databases (called resource managers (RM)) involved in the transaction. This is followed by requesting each RM to vote towards the final outcome of the transaction. After this, each RM is informed about the final fate of the transaction resulting from this vote.

The TM uses the following transaction commitment process when the application issues a transaction commit request to it:

\textbf{Prepare} Each resource manager (RM) (or database) is asked to vote whether the transaction can be committed or not.
Figure 5.1: Cherry Garcia data store object state transition diagram
Decide If all participating RMs vote yes, the commit log record is written by the coordinator to durable storage.

Commit Inform all participating RMs that the transaction has committed (or not).

Complete After receiving an acknowledgement from all participating RMs write a commit completion record to indicate the end of phase 2.

The prepare phase is referred to as phase 1 and the commit phase is also called phase 2, hence the name two-phase commit protocol (2PC). If even one of the participants responds to the prepare request in the negative, the transaction is aborted. A detailed description of 2PC and related protocols is discussed in Section 2.5 while related work is discussed in Section 3.3. In particular, our protocol adopts some of the techniques from the coordinator log and implicit yes variants of 2PC.

5.2.2 Distributed Two-Phase Commit

Two-phase commit is widely used to coordinate work across independent resource managers. Normally transactions are committed locally on each node in the network to access only local resources. However, there are occasions when data may be distributed across different nodes. In such situations, the transaction involves more than one transaction manager.

In this situation, one transaction manager is chosen to be the root transaction manager which is issued the original transaction \texttt{begin()} request. As the transaction involves more data residing on different nodes, additional transaction managers get involved to form a tree of transaction managers (see Section 2.5.2 for details). The root transaction manager takes on the role of the \texttt{coordinator} which is responsible for transaction commitment and the remaining transaction managers are \texttt{participants}.

When the application issues a transaction commit request to the root TM, it follows the transaction commitment process listed below:

Local prepare Ask the local resource manager to prepare for commit.
Prepare Request each participating transaction manager to vote whether the transaction can be committed by their local resource manager or not.

Decide If all participating TMs vote yes, the commit log record containing all the list of all participating transaction managers is written to durable storage.

Commit Inform all participant TMs that the transaction has committed (or not).

Complete After receiving an acknowledgement from all participating TMs write a commit completion record to indicate the end of phase 2.

The Cherry Garcia architecture as shown in Figure 5.2 uses the same approach to achieve transactional behaviour across records stored in different data stores. Each record is treated as though it were an independent resource and the coordination is performed by the TM that resides in the client.

5.2.3 Write-Ahead Log (WAL) Protocol

Recall that in database systems, physiological logging is used to ensure durability and resilience during crash recovery. The Write-Ahead Log (WAL) protocol requires that the page log record is written to durable storage before the updated page in order to ensure that the transaction can be recovered on failure.
1. Each page in memory has a log sequence number (LSN) field identifying the log record for the update operation that made the most recent update to the page.

2. Every update operation must maintain the LSN field.

3. Before a page can be copied to durable storage, the copy process must first ask the log manager to copy all log records up to and including the current page’s LSN to persistent storage.

4. Once the log records are persisted, the in-memory version of the page can be written over the persistent version.

5. The page must be fixed in memory during this process of writing and copying to ensure consistency of the operations performed.

Cherry Garcia uses the same principles as WAL to achieve transactional behaviour. Figure 5.3 describes how Cherry Garcia makes use an abstraction of a distributed WAL. Each record contains both the current (committed) state of the data record, the previous version of the record (previous image) as well as the status of the last prepare or commit request. Each record header contains metadata that denotes which transaction was responsible for the last state change. This is analogous to the LSN concept used in a traditional database WAL. The final state of the transaction is recorded by writing the Transaction Status Record (TSR) into the coordinating data store (CDS). This performs the task of the final commit record in the traditional WAL approach.

The Cherry Garcia protocol, that we present in Section 5.10 adheres to the following ordering: an update operation is recorded in a store with status PREPARED; then the TSR is written in CDS; and then the update operation in the store gets status COMMITTED. The TSR can be garbage collected once it is no longer needed for any store, because all stores have stably recorded the commit.

5.3 An example application

Listing 5.1 is an example of an application that uses the API to read two data records, one ("saving") resides in an instance of Google Cloud Storage, abstracted
### 5.3. AN EXAMPLE APPLICATION

by the Datastore \( gds \), while “checking” is stored in Windows Azure Storage represented as Datastore \( wds \). The example also uses a third store (explained later) that acts as the Coordinating Data Store (CDS).

```java
1 public void UserTransaction() {
    Datastore cds = Datastore.create("credentials.xml");
    Datastore gds = Datastore.create("goog_creds.xml");
    Datastore wds = Datastore.create("msft_creds.xml");
    Transaction tx = new Transaction(cds);
    try {
        tx.start();
        Record saving = tx.read(gds, "saving");
        Record checking = tx.read(wds, "checking");
        int s = saving.get("amount");
        int c = checking.get("amount");
        saving.set("amount", s - 5);
        checking.set("amount", c + 5);
        tx.write(gds, "saving", saving);
        tx.write(wds, "checking", checking);
        tx.commit();
    } catch (Exception e) {
        tx.abort();
    }
}
```

Listing 5.1: Example code that uses the CG API to accesses two data stores
5.4 Assumptions on the platforms

In order to support multi-item transactions, the design requires that each data store provide the following capabilities (found widely even among NoSQL stores):

- the option when reading for single-item with strong consistency (a read may require the platform to return the latest version of the data item)
- atomic conditional update and delete on single items, similar to Test-and-Set (a change may be requested so the item is altered only in case that it has a particular value or attribute); this allows optimistic concurrency control with checks that certify the absence of conflicting updates
- ability to include user-defined metadata along with the content of a data item; used to tag each version with the information about the creating transaction, and also to include a previous version within the item

In addition to this, each client application performing a transaction must have write-access to some storage that can be made globally readable to others. This is used to keep the status information for the transaction. For simplicity of presentation, we describe a single separate store called the Coordinating Data Store (CDS) where all Transaction Status Records (TSR) are placed; however, in practice we expect the transactions to spread this information among diverse stores, including perhaps some where the data is stored. We also require a reliable, approximately synchronised, source of timestamps at the clients. In our implementation this comes from local clocks at the clients. More on these aspects is explained later.

5.5 Protocol overview

In essence, the protocol calls for each data item to maintain the last committed and perhaps also the currently active version, for the data and relevant metadata. Each version is tagged with metadata pertaining to the transaction that created it. This includes the transaction commit time and transaction identifier that created it, pointing to a globally visible Transaction Status Record (TSR) using a Universal Resource Identifier (URI). The TSR is used by the client to determine which version of the data item to use when reading it, and so that transaction commit can happen just by updating (in one step) the TSR. The transaction identifier,
stored in the form of a URI, allows any client regardless of its location to inspect the TSR in order to determine the transaction commitment state. Using the status of the TSR, any failure can be either rolled forward to the later version, or rolled back to the previous version. The test-and-set capability on each item is used to determine a consistent winner when multiple transactions attempt concurrent activity on a conflicting set of items. A global order is put on the records, through a consistent hash of the record identifiers, and used when updating in order to prevent deadlocks. This approach is optimised to permit parallel processing of the commit activity.

The system is implemented as a runtime that is linked to the client application. The architecture of the library is described in Figure 5.2. It provides programming abstractions for transaction coordination and for the client-side representation and local cache for each data store, through Transaction and Datastore classes.

In more detail, the transaction start time, $T_{\text{start}}$, is set at the start of the transaction and remembered in the Transaction object. It is used to select the correct version of the records read from the data stores. The transaction commit timestamp, $T_{\text{commit}}$, tags all the records that belong to the transaction write set.

As the transaction executes, data items are read from the appropriate data stores and each is cached in-memory within the Datastore abstraction provided by the library. The items modified during the course of the transaction are kept in the Datastore object’s cache until the transaction tries to commit. The Transaction object itself keeps a list of its write-set (the keys of all modified items).

Once the application has performed all the updates to the data items, the changes are committed to their respective data stores. The transaction commit operation is performed in two phases of processing.

### 5.5.1 Phase 1

The current timestamp is obtained using the TrueTime API to set the transaction commit time, $T_{\text{commit}}$. Then every dirty record in the client’s record cache is stamped with the commit timestamp metadata. Each record is marked with a PREPARED flag and the TSR URI after which they are pushed to their respective data stores using a test-and-set operations to ensure that no conflicting transaction has written a newer version since the current transaction read the data items.
This is different to the Time Stamp Ordering approach described in Chapter 2 Section 2.4.1, which relies solely on the timestamp of the operation to determine the order in which they must be applied. It is not possible to compare the object versions across different client application instances in a distributed manner. However, this technique achieves essentially the same effect by comparing object version tags in the test-and-set operation.

5.5.2 Phase 2

If all the records have been successfully written in PREPARE state, then a Transaction Status Record (TSR) is written (using a test-and-set operation) to the Coordinating Data Store (CDS); this indicates that the transaction is truly committed. If the transaction client fails after this point, the transactions effects will be recovered by roll forward of the records.

Once the TSR is created, the records in the write set are marked with a COMMITTED tag and once again pushed to their respective data stores. This operation can be done in parallel to improve performance. The TSR is then lazily deleted once all records have been committed in order to forget the transaction.

This approach does not use any centrally managed infrastructure to perform transaction commits. The underlying key value store and its inherent features are used to implement transaction across multiple records that may reside on different individual stores. As there is no need to install or maintain any central infrastructure making it suitable for use across heterogeneous data stores that can span multiple data centres across geographical regions.

To illustrate an execution timeline, Fig 5.4 depicts two application clients, $C_1$ and $C_2$, as they access two records $r_1$ and $r_2$, stored in WAS and GCS respectively. In this example, a CDS is different from the WAS and GCS instances and hosted in a third data store instance. Transaction $t_1$ reads both records and processes them before issuing the transaction commit. The commit occurs in two phases, during the prepare phase both records are written in global order to their respective data stores and marked with a PREPARED state. The TSR is then written to the CDS to indicate that the prepare phase is successful and this is the point of commit. The following phase is then performed in parallel.

The application on $C_2$ starts transaction $t_2$ and reads the two objects. In
Figure 5.4: The timeline describing 3 transactions running on 2 client hosts to access records in 2 data stores using a third data store as a CDS

the prepare phase, the first record write fails because $t_1$ has already written a new version of the object. The record is recovered but since the transaction $t_1$ is still active because its lease time has not expired, $t_2$ aborts. When the client $C_2$ starts another transaction $t_3$ which reads the same two records which are in the PREPARED state, the previous version is used. When $t_3$ attempts to prepare the record $r_1$, the operation fails because the version read and the version in the data store do not match. At this point, the record may be read in order to be recovered but since in this case the transaction $t_1$ is still active, the recovery operation is aborted and subsequently $t_3$ is also aborted.

In the rest of this chapter we go deeper into detail on the components of the library and the algorithms.

5.6 Time

Timestamps for transaction start and commit are acquired from a reliable source of time which is abstracted to match the TrueTime API (as in Spanner [30]). The API method `now()` returns a numeric value $t$ along with an error margin $\epsilon$ associated with it. Thus the “time” is seen as an interval, $t \pm \epsilon$. Two timestamps are compared using the `before()` and `after()` methods; these return boolean values.
A timestamp $t \pm \epsilon$ is said to happen before another timestamp $t' \pm \epsilon$ if $t + \epsilon < t' - \epsilon$. This is represented as $t < t'$. Similarly, $t \pm \epsilon$ is said to happen after another timestamp $t' \pm \epsilon$ if $t - \epsilon > t' + \epsilon$. This is represented as $t > t'$.

However, if there is an overlap in the intervals $t \pm \epsilon$ and $t' \pm \epsilon$, i.e. $t - \epsilon \leq t' - \epsilon \leq t + \epsilon$ or $t - \epsilon \leq t' + \epsilon \leq t + \epsilon$, then the order in which the two events, $t$ and $t'$ occur, cannot be determined. In this situation, the methods before() and after() will return false. The CG protocol is written to abort a transaction whenever we are unable to determine the order but need to know it.

Our design works with any distributed source of time that can be abstracted with this API (i.e. it estimates the time and provides a bound on the uncertainty in that estimate). In our implementation, we use local clocks at the clients, and an uncertainty of 1 ms, rather than the atomic clock-based techniques of Spanner. With transaction clients around a data centre, epsilon would need to be higher (say 10-15ms) due to NTP synchronisation error margins. The epsilon would be higher still across WANs due the higher network latency observed. Higher values of $\epsilon$ leads to more frequent aborts when comparing timestamps that overlap.

### 5.7 Data item records

The data record has application data whose structure is transparent to the protocol, and a record header with metadata organised as a collection of field names and their values. A detailed description of each field is listed below. It is important to note that the previous version would not be explicitly stored again as metadata, if the particular data store allows native access to the history of versions.

**valid time start** ($T_{valid\_start}$): The timestamp after which the version of the record is considered committed if in the COMMITTED state.

**valid time end** ($T_{valid\_end}$): The timestamp after which the record is considered invalid. This is used to indicate that a record is deleted.

**lease time** ($T_{lease\_time}$): The transaction lease time to complete the transaction commit. The record state recovery is executed if the lease time expires and the record is in the PREPARED state.
transaction identifier ($TxID$): Signifies the URI of the transaction that last updated the record. The transaction URI can be examined to determine the fate of the transaction.

transaction state ($TxState$): The last update state of the record, whether PREPARED or COMMITTED.

last update time ($T_{last\_update}$): The last update timestamp on the data store server.

version tag (ETag): A data store generated version tag for the version of the record. This tag can be used to perform conditional read and write operations.

previous version (Prev): For a record in the PREPARED state, this has the previous version of the record (with metadata fields corresponding to each above, and also the application data).

## 5.8 Data store abstraction

The Datastore class provides a standard API that enables the Transaction class to coordinate data access from one data store, including record-by-record processing in the phases of the transaction’s commit operation (these methods will be implemented using specific functionality of the particular store, such as its conditional-write capability). To do this, the Datastore object keeps client-side state pertaining to each data record accessed by the application belonging to the context of the Transaction object. This may include information of data store credentials, record caches with multiple versions, and connection pools. The following is a description of the methods implemented by the Datastore class.

start(): start a transaction

read(key): read a record with the specified key from the data store, either from the cache or using the store’s consistent-read.

write(key, record): write a record with the specified key that belongs to the data store. The new value is recorded in cache, but not yet passed to the store.
**prev(key, record)**: return the previous version of the record associated with the specified key from the data store.

**delete(key)**: delete a record identified by the key (this is remembered in the cache, and will only happen in the data store after commit).

**prepare(key, record)**: if the version of the record matches the version on the data store or if the record is a new record, push the contents of the record and its previous version to the data store and mark it with the `PREPARED` state.

**commit(key, record)**: update record identified by the key to the `COMMITTED` state in the data store.

**abort(key, record)**: restore the record identified by the key to the previous `COMMITTED` state if it is in the `PREPARED` state.

**recover(key)**: recover the record identified by the key if it not in the `COMMITTED` state.

### 5.9 Transaction abstraction

The following is a detailed description of the methods supported by the Transaction class.

**start()**: start the transaction

**read(ds, key)**: read a consistent version of the record specified key from the data store and cache it in the transaction cache.

**write(ds, key, record)**: write the record to the transaction cache to be persisted in the data store specified by ds using transaction commitment protocol.

**delete(ds, key)**: mark the record identified by the specified key as deleted in the transaction cache to be deleted from the data store at commit time.

**commit()**: commit all the changes to the data stores using the commitment protocol.
abort(): abort the transaction

recover(txid): return the status of the transaction. The returned state can be used to rollforward or abort the transaction.

The transaction context is available to the application in the form of an object of the Transaction class. It has a record cache for items that have been read from and/or are to be written to their originating data stores. The records in the cache are addressed using a combination of the record key and a data store identifier.

In addition, the TSR must be persisted so that other clients accessing the data can refer to it. The TSR is a special type of globally accessible data record whose identifying key is its transaction identifier that is a Universal Resource Identifier (URI) with a Universally Unique Identifier (UUID) of the form \( \text{http://(host): (port)/uuid} \) as the key persisted in a data store called the Coordinating Data Store (CDS). The CDS uses the same technology as the individual data stores and is expected exhibit the same reliability and scalability characteristics.

5.10 Cherry Garcia Protocol

In this section we discuss the protocol methods in detail and list the respective algorithms. We then analyse the performance in terms of the number of message exchange rounds, the number of messages and the number of forced log writes needed by the protocol and compare it with two well known commit protocols, the basic 2PC and its presumes-abort variant (a widely used optimisation).

5.10.1 Start transaction

The transaction is started by creating a unique transaction identifier using a UUID generator and setting the transaction start time \( T_{\text{start}} \) to the current time using the TrueTime API.

5.10.2 Transactional read

If the supplied key is already in the transaction’s cache, that version is used, to enforce that the transaction sees any of its previous writes to the record. Otherwise, the record is read from the data store using the supplied key. The data store
Algorithm 1 Start transaction

1: function START
2:    T.identifier ← UUID()
3:    T.start ← now()
4:    T.state ← STARTED
5: end function

populates the record header and contents that can be used by the transaction code. If the record is in PREPARED state, then we try to determine the status of the writer of the latest version, and bring the record up-to-date. The details are explained below as Transaction recovery.

If we cannot determine the status of the writer of a PREPARED current version (because the lease time has not expired and there is no TSR yet), then the read attempt fails, and the reading transaction will itself abort. This is a pessimistic approach.

An alternate approach would be to return the last committed version of the record. However, the current transaction would be aborted if the previously incomplete transaction is committed successfully. If the incomplete transaction is aborted, then the transaction would succeed. This is an optimistic approach. The degree of success of each approach depends on various factors including the number of concurrent transactions and the number of records involved in each transaction.

Once we have a committed state for the record, we find the latest version which is valid for the current reader transaction’s start time, $T_{start}$. This may be the current version, or the previous version; if we cannot find a version valid for the reader’s snapshot, the read fails.

When a valid version of the record is read from the data store, it is put into the transaction record cache and also returned to the caller.

5.10.3 Transactional write

Transactional write operation is simple. The record value associated with the key is written to the Transaction object cache. In Algorithm 1, line 2 ensures that the earlier version of the record is marked as the previous version if it already exists in the cache and no write operation of the same record is performed during the course of the current transaction. The data record is only written to the actual
Algorithm 2 Read a record

1: function READ(datastore, key)
2:   if $\exists T.cache(datastore, key)$ then
3:     return $T.cache(datastore, key)$
4:   end if
5:   record $\leftarrow$ datastore.read(key)
6:   if record.state $\neq$ COMMITTED then
7:     checktime $\leftarrow$ now()
8:     tx_record $\leftarrow$ coord.datastore.read(record.TxID)
9:     if $\exists$ tx_record then
10:        if tx_record.state = COMMITTED then
11:           datastore.commit(key, record)
12:        else
13:           datastore.abort(key, record)
14:        end if
15:     else
16:        if checktime $\neq$ record.T lease_time then
17:           throw Exception(“Read fails”)
18:        end if
19:        datastore.abort(key, record)
20:       go to 5
21:   end if
22: end if
23: if $T._{start} \neq$ record.T valid_start then
24:   record $\leftarrow$ datastore.prev(key)
25: if $T._{start} \neq$ record.T valid_start then
26:   throw Exception(“Read fails”)
27: end if
28: end if
29: $T.cache.put$(datastore, key, record)
30: return record
31: end function
data store at the time of executing the transaction commit.

Algorithm 3 Write a record

1: function WRITE(datastore, key, record)
2:     if record.TxID ≠ T.TxID then
3:         record.dirty ← true
4:         record.prev ← T.cache.get(datastore, key)
5:         record.TxID ← T.TxID
6:         T.cache.put(datastore, key, record)
7:     end if
8: end function

5.10.4 Transaction commit

The transaction commit is performed in two phases.

The Prepare phase: The record cache is inspected and all dirty objects are inserted into the write-set. Each record in the write-set is marked with the transaction status record URI, the transaction commit time, and the transaction state is set to PREPARED then conditionally written to the respective data store in a fixed total order. This is done by performing the operation in the order of the hash values of the identifying keys of the records. The reason for this is discussed in more detail in Section 5.11.1 later in this chapter. The operation is performed using the Datastore.prepare() method which utilises a conditional write to the data store using the record version tag (ETag or equivalent mechanism). The prepare phase is considered to be successful if all dirty records are successfully prepared. Should one wish to provide Serializable isolation, one needs to also prevent read-write conflicts, by additionally checking that each unmodified but accessed item is unchanged, between its initial access and the end of the transaction. This is further discussed in Section 5.15.

The Commit phase: The TSR is written to the coordinating data store to indicate that all the records have been successfully prepared. The records are then committed by calling the data store commit() method for all records in parallel. The record commit method marks the record with the COMMITTED state. The operation is performed using the Datastore.commit()
Algorithm 4 Commit transaction
1: function COMMIT
2: $T$.commit_time $\leftarrow$ now() $\triangleright$ phase 1: prepare
3: $T$.lease_time $\leftarrow$ now() + commit_timeout
4: for $(\text{datastore, key, record}) \in \text{ordered(cache)}$ do
5: if record.isDirty() then
6: \hspace{1em} record.$T$.valid_start $\leftarrow$ $T$.commit_time
7: \hspace{1em} record.$T$.lease_time $\leftarrow$ $T$.lease_time
8: \hspace{1em} status = datastore.prepare(key, record)
9: if status = ERROR then
10: \hspace{2em} recov_rec $\leftarrow$ datastore.recover(key)
11: \hspace{2em} if recov_rec then needs recovery
12: \hspace{3em} if $\exists$ coord datastore.recover(recov_rec.$TxID$) then
13: \hspace{4em} datastore.commit(key, recov_rec)
14: \hspace{2em} else
15: \hspace{3em} datastore.abort(key, recov_rec)
16: \hspace{2em} end if
17: \hspace{1em} end if
18: prev_rec $\leftarrow$ datastore.read(key)
19: if $\exists$ prev_rec then
20: \hspace{1em} datastore.write(key, record)
21: \hspace{1em} end if
22: status = datastore.prepare(key, record)
23: if status = ERROR then
24: \hspace{1em} abort()
25: \hspace{1em} return ERROR
26: \hspace{1em} end if
27: \hspace{1em} end if
28: end for
29: $T$.state $\leftarrow$ COMMITTED $\triangleright$ phase 2: commit
30: coord.datastore.write($T.TxID$, $T$.record)
31: for all $(\text{datastore, key, record}) \in \text{cache.keys()}$ do
32: \hspace{1em} datastore.commit(key, record)
33: end for
34: return SUCCESS
35: end function
method which also utilises a conditional write to the data store using the record version tag (ETag or equivalent mechanism). This ensures that the one-phase commit optimisation (described in Section 5.12) does not violate transactional behaviour. Once the records are committed the transaction status record is deleted asynchronously from the coordinating data store.

### 5.10.5 Transaction abort

If the transaction commit operation has not been initiated the abort operation is trivial. The record cache is cleared and the transaction is marked as aborted.

#### Algorithm 5 Abort transaction

```plaintext
1: function ABORT
2: T.state ← ABORTED
3: coord_datastore.write(T.TxID, T.record)
4: for all (datastore, key, record)incache.keys() do
5:    if record.state = PREPARED then
6:       datastore.abort(key, record)
7:    end if
8: end for
9: return SUCCESS
10: end function
```

If some of the updated records have had the Datastore.prepare() method executed, then transaction can be aborted if the TSR has not been written to the transaction coordinating data store. In this case, the rollback is performed by issuing an abort on all prepared records. In a data store that does not support multi-version records, the rollback operation is performed by overwriting the record with the application data and metadata that are found in the metadata field `Prev`. In this situation, we rollback to the previous state of the current version, but we have lost the former contents of the `Prev` field itself; this will never be needed, since the version we are restoring was itself COMMITTED.

Once the transaction status record has been written to the coordinating data store the transaction cannot be aborted.
5.10.6 Transaction recovery

If an application fails during the commit process, individual data items may need to be recovered; this will happen lazily as relevant records are accessed by other transactions. When a data item is read, its transaction state is inspected. If it is in the COMMITTED state, recovery is not necessary. However, if it is in the PREPARED state it is either rolled forward or rolled back, depending on the transaction status. The writer transaction’s URI is used to inspect the state of the Transaction Status Record. We may have to roll the data record forward (that is, marking the record header with the COMMITTED state) if the writer did commit, as determined by the TSR. If the writer aborted, which may either be known from the TSR or because the lease time has expired with no TSR, then we rollback the record, as described above for Transaction abort.

Algorithm 6 Recover transaction status

```
1: function recover(TxID)
2: tx_record ← coord_datastore.read(TxID)
3: if ∃ tx_record ∧ tx_record.state = COMMITTED then
4:    return true
5: end if
6: return false
7: end function
```

5.10.7 Performance Analysis of the Protocol

As we have seen in Section 2.5, 2PC protocols have traditionally been analysed in terms of number of rounds, message and forced log records. In order to do this, we consider every request to PREPARE, COMMIT, ABORT or RECOVER to be one message and the response to the request to be a separate message. The protocol consists of two rounds. The prepare phase consists of one message per dirty data record and a corresponding response resulting in $2n$ messages. Writing the TSR to the coordinating data store and getting its response involves 2 messages concluding phase 1. The parallel commit phase consists of requests which is another $n$ COMMIT messages. The responses can be ignored as they are performed asynchronously.

Table 5.1 compares Cherry Garcia with basic 2PC and one of its two widely
implemented variants, *presumed-abort (PA)*. All three protocols use two rounds of messages. Cherry Garcia has an overhead of having to write the TSR at the end of phase 1. This adds two messages to phase 1. However, phase 2 only requires the requests to be sent to the data stores. The response to the COMMIT request can be ignored. Any failure will be recovered lazily by a subsequent reader. The number of forced log writes is \( n + 1 \), one for each data record and one for the TSR.

### 5.11 Deadlock detection and avoidance

Cherry Garcia uses a two-pronged approach to prevent deadlocks occurring due to the concurrent execution of conflicting transactions. The first is to detect deadlocks and the second involves avoiding deadlocks.

#### 5.11.1 First preparer wins

Each client calls the *PREPARE* method in the order of a hash values of the keys. The hash function used can be any uniformly distributed hash function suitable for the distribution of the keys. For the purpose of this discussion, we use the default Java `java.lang.String.hashCode()` implementation\(^2\) available with JDK 6 that uses \( s[0] \ast 31^{(n-1)} + s[1] \ast 31^{(n-2)} + \ldots + s[n-1] \) to generate the hash value.

As the client calls the conditional *PREPARE* method with an *ETag*, any failures indicating that the record is already in the PREPARED or COMMITTED state indicates that another concurrent transaction has previously successfully completed or is in the process of doing so. This is treated as a potential deadlock causing the client to rollback all records prepared so far.

---

\(^2\) [http://docs.oracle.com/javase/6/docs/api/java/lang/String.html#hashCode()](http://docs.oracle.com/javase/6/docs/api/java/lang/String.html#hashCode())
5.12. OPTIMISATIONS

5.11.2 Ensuring at least one winner

Concurrent conflicting transactions prepare the records in the same order based on the hash values of the record keys. Only one of the transactions will succeed in performing a conditional write operation to the data store. The other transaction aborts after rolling back its prepared records. There is only one global ordering of calling the \textit{PREPARE} method because every client make the calls the data records in its write-set in the same global order of the hash values of the key of each record. The global order ensure that there is no cyclic conflict.

While the arguments above show that our protocol is safe, that is, it will not allow conflicting transactions to commit together, we also want it to be live, that is, forward progress should occur: we want to be sure that among the conflicting transactions, one will be able to commit. This is not quite achieved, as our protocol will abort both transactions if they have obtained timestamps that overlap (as intervals, considering the epsilon from clock skew).

5.12 Optimisations

In this section, define the performance improvements to the Cherry Garcia protocol to enable it to perform well in a distributed, heterogeneous environment.

5.12.1 One-phase transaction commit optimisation

If there is just one data item in the write-set of the transaction, the prepare-phase of the commitment protocol can be avoided and the new version can be written directly to the data store with status COMMITTED. To ensure correct behaviour, the commit operation is performed using the same test-and-set technique to prevent a conflicting write operation from overwriting the changes.

The TSR is not written as it will never be consulted, saving one record write operation. This is similar to the \textit{last agent optimisation} described in Chapter 2 Section 2.5.3.
5.12.2 Parallel commit phase

The PREPARE method is called in the order of the hash values of the keys to ensure global ordering. This can be slow for a large write-set, particularly in the case of a slow network connection. After the prepare phase, the TSR is written following which the COMMIT method can be called in parallel on all records. This can reduce transaction commit latency significantly for large write-sets.

5.12.3 Read-only transaction optimisation

If there are no updates performed by a transaction. On commit, the transaction is simply marked completed at the client. This approach is based on the read-only subtree optimisation discussed in Chapter 2 Section 2.5.3.

5.13 Failure scenarios

There are primarily three types of failures that can occur in this system. These are:

**Data store failures:** Modern cloud data store are resilient systems. They are designed for fault-tolerance and high-availability. In the event of failure, it is likely that more than just the data store is inaccessible. However, if this happens the client recovers the transaction and rolls forward or rolls back the transaction based on the commit resolution.

**Network failures:** The network is the most fragile element of the cloud ecosystem. Failure of individual components like data stores and applications can be mitigated using fault-tolerance techniques but network failure is difficult to deal with. This is why the transaction lease is important. It enables lazy transaction recovery to function after the transaction lease has expired.

**Client failures:** Individual client application fail frequently. Every transaction commit is associated with the lease time out to ensure that a client application failure does not prevent progress of other clients. If the lease expires, the transaction is recovered based on the state of the TSR and individual records are lazily recovered.
5.13.1 Performance implications of retries

Lazy transaction recovery and retried operations have a negative impact on transaction latency and throughput. This is particularly true when the applications use low bandwidth (mobile) or high latency (wide area) networks to access the data store.

The failure of any operation, such as a failed prepare() operation resulting in a call to recover(), can result in the exchange of multiple additional messages between the client and data store. This, in turn, increases the time taken to commit the transaction.

This is a drawback of our design as a result of the transaction coordinator residing in the client while transaction state is maintained in the data store. It requires the transaction coordinator to obtain transaction state by querying the data store in failure cases, resulting in additional message exchanges. This can be a problem in low bandwidth and high latency networks.

5.14 Correctness

We now discuss the justification for our claim that the Cherry Garcia protocol provides correct ACID transactions.

The requirement of a transaction to be “Atomic” means that the outcome is either that all changes are installed in the database to be seen by other transactions, or else none of the changes are installed. That is, when the transaction commits, all its changes are installed, and when the transaction aborts, all its changes are going to be undone. For Cherry Garcia, the single event that marks the change of status from undecided to committed is line 30 of Algorithm 4, namely the writing of the TSR with state=COMMITTED (to be more precise, the crucial point is when the TSR is persistent in the coordinating data store). The event that transitions the transaction status to aborted is more complicated: it can occur if a TSR is written with status ABORTED, or else if the transaction’s lease time has expired without a COMMITTED status TSR being written. Once a transaction has written the TSR with state=COMMITTED, then the various data items the transaction wrote will be gradually shifted into having record state=COMMITTED, either by the coordination protocol in Algorithm 4 (line 32), or else, by other transactions
that try to read the item, and detect the committed status of the writer from the TSR (line 8 of Algorithm 2). Once a transaction has aborted, the versions it produced will be gradually removed, and the previous committed state of each item will be restored. This happens from the coordinator itself (line 6 of Algorithm 5), or when a reader detects the abort, either from the TSR or from there being no TSR after the expiry of the lease-time.

The “Consistency” aspect of ACID is not provided by the data management, but rather, by the application code. There is an expectation that each transaction must be written so that, when run by itself, it takes the data from a state that is consistent (that is, where all the applications constraints are valid) to another state where all constraints hold.

The “Isolation” that Cherry Garcia offers is Snapshot Isolation (SI). While this is not as strong as serializability, it is widely used (including as the strongest isolation offered by the very successful commercial platform Oracle DB), and thus it seems to satisfy the needs of many enterprises. There are two essential properties we need to have, for a system to provide SI. The “snapshot read” property says that the version of some item x that a transaction T sees when T reads x, is the version that was produced by the transaction that committed most recently before T started, among all the transactions that committed changes to x. The “First Committer Wins” property says that it is not allowed for two concurrent transactions to both commit changes to an item x. We now discuss each property in turn.

**Lemma 5.14.1** Let transaction T successfully read record x. The result returned by T is that of the version of x with the greatest timestamp, among all transactions that produce versions of x and that also commit before the start of T.

**Proof** The snapshot reading property is ensured by the use of the timestamp by transaction T. The timestamp is set to be $T_{\text{start}}$, the clock time when T begins. By the property of clocks, this is greater than the commit timestamp used for versions created by transactions that commit before T starts, and greater than the commit timestamp in versions created by transactions that commit after T starts (we neglect here the slight skew in timestamps between sources; this may mean that T does not see some transaction that committed in real time just momentarily before T starts, but this is usual in distributed system work;
in essence, we use the order of timestamps as the order that defines whether one transaction is treated as committed before or after \( T \) starts). The Cherry Garcia protocol ensures that the reads of \( T \) see a version \( v \) of \( x \) written by \( T' \) with the either of the following characteristics (i) \( v \) is the current version of \( x \), \( v \) has committed, and \( v.T_{\text{valid, start}} < T_{\text{start}} \) or else (ii) the current version of \( x \) committed with \( T_{\text{start}} > x.T_{\text{valid, start}} \), and \( v \) is the previous version of \( x \), with status COMMITTED and \( x.prev.T_{\text{valid, start}} < T_{\text{start}} \). The code ensures this by lines 6 through 21 of Algorithm [2] which make sure that a read occurs only when the data item has been rolled forward or rolled back; if the status is unclear (because there is no TSR and the lease is still running) then the read attempt fails. Once this certainty of status is achieved, the test in line 23 being false gives the case (i), while the test in line 23 being true and the test in line 25 being false gives case (ii).

There is a degree of uncertainty with respect of timestamp ordering. This leads to four cases when comparing timestamps \( T_{\text{start}} \) and \( T_{\text{valid, start}} \). The first two case are when either \( T_{\text{start}} \) happens before or after \( T_{\text{valid, start}} \). Time management systems like TrueTime handle these situations correctly. There is then the case when the two timestamps overlap and it is uncertain which happened before the other and the other case when they are certainly of the same value. The uncertainty around the latter two cases are taken care of by the lines 23 and 25 because only the certain \( T_{\text{start happens before}} T_{\text{valid, start}} \) case is successful.

The prevention of concurrent transactions updating an item is guaranteed by the fact that each writer produces a new version, and the following lemma shows that multiple versions of an item cannot originate from concurrent transactions.

**Lemma 5.14.2** Let transactions \( T \) and \( U \) each successfully commit, producing versions of record \( x \). Then \( T \) and \( U \) are not concurrent: either \( T \) commits before \( U \) starts, or else \( U \) commits before \( T \) starts.

**Proof** We will use proof by contradiction: suppose that \( T \) and \( U \) are concurrent, that is \( T_{\text{start}} < U.commit.time \) and also \( U_{\text{start}} < T.commit.time \), and that \( T \) produces version \( v \) of \( x \), and \( U \) produces version \( w \) of \( x \). Without loss of generality, we can let \( T \) be the transaction that is the first to execute the prepare operation of the record \( x \) in the data store that manages \( x \). At this time, the version \( v \) of \( x \)
was placed in the data store. When producing the version $w$ of $x$ for $U$, a version was found to become $w.prev$, this could have been the version in $U$’s cache, if $x$ had already been read by $U$, or it could have been a version fetched at the time of producing $w$, but in either case, the code ensures that $w.prev$ has status that has been determined and that $w.prev$ was produced by a transaction that committed before $U_{\text{start}}$ (using the argument of Lemma 5.14.1); in particular, $w.prev$ is not $v$. Thus when $U$ executes the prepare on record $x$, to try and install its version $w$, the test-and-set check will fail. This contradicts the hypothesis that $U$ successfully produced its version $w$.

“Durability” for Cherry Garcia is derived from the durability in the various data stores; the techniques used vary between stores, but typically, each item is kept in several replicas so that it will survive a crash of any one physical machine. Since Cherry Garcia writes the versions and TSRs into the data stores, each version will remain in the system (and it will keep its COMMITTED transaction status) until it is explicitly replaced by a later version. While the TSR itself will be eventually deleted, this does not happen until the TSR will no longer be consulted (because every item written has its record_state persistently changed to COMMITTED, and the TSR is not consulted unless the record_state is PREPARED).

5.14.1 Effect of Violations of Data Store Assumptions

The fundamental assumptions of Cherry Garcia must be met by each data store in order to guarantee transactional behaviour. Here we discuss the implication of violating each assumption in detail.

No single-item transactions

In an environment without single-item transactions, the client is not guaranteed to receive the latest version of the data record from each data store. This prevents the prepare operation (and subsequent commit, if the prepare succeeds) from being applied to the correct version of the data record. This could cause a version of the data record previously updated by another concurrently executing transaction to be overwritten causing an inconsistency as a result of an inconsistent read.
The sequence diagram in Figure 5.5 illustrates one such scenario which causes a lost update resulting in an inconsistency. Transaction $t_1$ reads records A and B with values 10. Then, transaction $t_1$ adds 1 to A and subtracts 1 from B. After transaction $t_1$ has committed, transaction $t_2$ reads the same records A and B. However, due to the lack of single-item transactions, it reads value 10 wrongly for A and 9 correctly for B. Transaction $t_2$ then adds 1 to A and subtracts 1 from B. Transaction $t_2$ introduces an inconsistency as a result of the lost update by writing the value 11 instead of 10 due to the lack of single-item transactions.

No test-and-set operations

Once the client has read a version of a data record its version information in the form of a timestamp or version identifier such as an ETag is recorded. In order to make sure that the client updates the data record only if there is no other conflicting transaction, a conditional operation such as a HTTP PUT (or REST+T PREPARE) is used to prepare the data record for subsequent commitment. Without a reliable test-and-set (or equivalent) capability, the operation can overwrite...
the data update from a concurrent transaction. This can result in lost updates and/or partial updates.

The sequence diagram in Figure 5.6 illustrates one such scenario which causes lost updates. Transaction $t_1$ and $t_2$ both read records A and B both with a value of 10. Transaction $t_1$ adds 1 to A and subtracts 1 from B. On the other hand, transaction $t_2$ subtracts 1 from A and adds 1 to B. Without a reliable test-and-set operation, transaction $t_2$ overwrites the value of 11 written to record A by $t_1$ and $t_2$ overwrites the value 11 with value 9. This results in partial updates from both transactions $t_1$ and $t_2$ resulting in an inconsistent state.

No time ordering guarantee

When a client reads a version of the data record in the COMMITTED state, it must decide whether the returned version of the record is the correct version to be used by the application based on the transaction start time. To do this reliably, it should be possible to compare the timestamp on the data record against the transaction start time. Reading any version committed after the start of the transaction will cause the client to read a version that is newer than what should be read. In another situation, the client may read an older version of the data record instead of the current committed version as a result of erroneous timestamp
Comparison.

In Figure 5.5, transaction $t_2$ can wrongly read a previous version of record A as a result of a wrong timestamp comparison. This can happen if the clock on the data store server hosting record A is ahead of the client.

5.15 Implementing serializability

The approach described so far in this chapter is able to achieve Snapshot Isolation semantics. While these are common in practice (being available in several widely-used single-site platforms such as Oracle DB, SAP etc), there is a possibility that data integrity can be violated by concurrent activity when transactions run with Snapshot isolation. By doing more work, we can achieve the full semantics of Serializable Isolation for transactions. Essentially, this can be achieved by committing a new version of all records that the transaction accesses; where the transaction requests just to read the record, but not modify it, our algorithm described here will ensure that there is also a new version (with the same value as before, but with an increased valid timestamp). The pseudo-code for this is in Algorithm 7; for Serializable mode, lines 30 to 34 show that the new version of the record is created and the record is marked dirty after inserting it into the cache. We note that there is no need to create a new version if the record is already in the transaction’s cache, because the version will have been made on the earlier access that brought the record in from the data store for the first time. Our algorithm ensures that all records accessed, both those from the read-set and those where the transaction made changes, are participants in the CG commitment protocol, whose validation check will prevent a concurrent transaction accessing the object. This ensures serializable execution, as we prove in Section 5.16 below.

This approach is a brute-force method of achieving serializable transaction execution. Its performance will be slow when there are large fractions of records where the transaction reads but does not intend to modify. However, it is very easy to implement, uses the same record format as the original CG algorithm, and it is portable across WAS, GCS and Tora. A more sophisticated approach would be to keep separate a timestamp piece of metadata with the latest version of each record, to keep track of the highest timestamp of any reader transaction, as well as the valid-start timestamp from the usual CG structure (which captures the
Algorithm 7 Serializable read a record

1: function \textsc{read}(datastore, key)
2: \textbf{if} \ \exists \ T.cache(datastore, key) \textbf{then}
3: \hspace{1em} return \ T.cache(datastore, key)
4: \textbf{end if}
5: \hspace{1em} record ← datastore.read(key)
6: \hspace{1em} \textbf{if} record.state \neq \text{COMMITTED} \textbf{then}
7: \hspace{2em} checktime ← now()
8: \hspace{2em} tx\_record ← \text{coord}.datastore.read(record.TxID)
9: \hspace{2em} \textbf{if} \ \exists tx\_record \textbf{then}
10: \hspace{3em} \textbf{if} tx\_record.state = \text{COMMITTED} \textbf{then}
11: \hspace{4em} datastore.commit(key, record)
12: \hspace{3em} \textbf{else}
13: \hspace{4em} datastore.abort(key, record)
14: \hspace{3em} \textbf{end if}
15: \hspace{2em} \textbf{else}
16: \hspace{3em} \textbf{if} checktime \neq record.lease\_time \textbf{then}
17: \hspace{4em} throw Exception(“Read fails”)
18: \hspace{3em} \textbf{end if}
19: \hspace{2em} datastore.abort(key, record)
20: \hspace{1em} \textbf{go to 5}
21: \hspace{1em} \textbf{end if}
22: \hspace{1em} \textbf{end if}
23: \hspace{1em} \textbf{if} T.start \neq record.valid\_start \textbf{then}
24: \hspace{2em} record ← datastore.prev(key)
25: \hspace{2em} \textbf{if} T.start \neq record.valid\_start \textbf{then}
26: \hspace{3em} throw Exception(“Read fails”)
27: \hspace{2em} \textbf{end if}
28: \hspace{1em} \textbf{end if}
29: \hspace{1em} T.cache.put(datastore, key, record)
30: \hspace{1em} \textbf{if} T.isolation = \text{SERIALIZABLE} \textbf{then}
31: \hspace{2em} record.dirty ← true
32: \hspace{2em} record.prev ← T.cache.get(datastore, key)
33: \hspace{2em} record.TxID ← T.TxID
34: \hspace{1em} \textbf{end if}
35: \hspace{1em} return record
36: \textbf{end function}
maximum writer). This would allow us to code the standard rules of Optimistic Concurrency Control, and so prevent concurrent conflicting accesses while allowing concurrent readers.

5.16 Correctness for Serializable transactions

The “Atomicity”, “Durability” and “Consistency” results from Section 5.14 directly apply to Algorithm 7, which executes all the steps of the usual CG protocol, but also does some extra version creations. Therefore, all we need to show here is that the “Isolation” provided with this algorithm is Serializable.

In order to evaluate the isolation characteristics of the Algorithm 7 we will construct the equivalent serial execution, based on placing the transactions into the serial execution according to an appropriate total order, one that relates all the transactions. For a system that uses Algorithm 7 along with the usual Cherry Garcia commit processing, the total order that will serialise transactions is the order defined by the commit timestamps. That is, we define $T < T'$ by $T\.commit\.time < T'\.commit\.time$.

As an aid to the argument, we will use the ideas of distributed transaction management, whereby one can consider a global distributed execution in terms of the sub-transactions at each site, and we apply that in a model where each data record is treated as a separate site. That is, we will consider the activity of the various transactions on a single record at a time. We here quote Theorem 18.1 in Chapter 18 of Weikum and Vossen [139] which states the following fundamental result, that global serializability follows from local (conflict) serializability when there is a common total order on the transactions that can serialise each site separately:

**Theorem 5.16.1** Let $s$ be a global history with local histories $s_1, \ldots, s_n$ involving a set $T$ of transactions such that each $s_i, 1 \leq i \leq n$, is conflict serializable. The following holds:

$s$ is globally conflict serializable iff there exists a total order “$<$” on $T$ that is consistent with the local serialization order of the transactions, i.e.,
(\forall t, t' \in T, t \neq t') \implies
(\forall s_i, 1 \leq i \leq n, t, t' \in \text{trans}(s_i))(\exists s'_i, s'_i\text{serial}, s_i \approx_c s'_i) \ t <_{s'_i} t' \)

Now we prove serializability by applying the above theorem, together with the following lemma, that indicates that for any given record \(r\), the local activity of the system on \(r\) is equivalent to executing those same activities (excluding those from aborted transactions) in the serial order defined by commit timestamps.

**Lemma 5.16.2** When executing Algorithm 7 along with the other protocols according to Cherry Garcia, the return value when transaction \(T\) reads record \(r\) is the value of the version created by the transaction \(U\) such that \(U\) has the greatest commit timestamp among all transactions that wrote \(r\) and committed before \(T\).

**Proof** We build on the arguments from Section 5.14, with the added information that Algorithm 7 ensures that a version is created for every record accessed in a transaction, and that for transactions that do not modify \(r\), the version they produce has the same value as the previous version of \(r\). Lemma 5.14.1 tells us that the return value for \(T\) will be that of the version produced by the transaction with highest timestamp that produced a version of \(r\) and committed before \(T\) started; because each transaction either is a writer of \(r\), or else it keeps the same value in the version it produces as was there before, this is the same as the value from the highest timestamp among writers of \(r\) that commit before \(T\) starts. Now we appeal to Lemma 5.14.2 to see that no transaction that wrote \(r\) is concurrent with \(T\); that is, the highest timestamp among writers of \(r\) that commit before \(T\) starts, is exactly the highest timestamp among writers of \(r\) that commit before \(T\) commits. This is exactly what the lemma here requires.

### 5.17 Chapter Summary

Database technology has evolved significantly over the years, however, the basic principles, particularly those related to correctness and transaction behaviour, remain the same. We use the principles developed over many years of research and development to define Cherry Garcia, a protocol that enables scalable transactions across heterogeneous data stores. We began by describing the challenges faced in the modern enterprise related to application development with the rapid and
prolific growth of web-services and other endpoints. Then, the intuition and basis of our approach was described using the analogy of a deconstructed write-ahead log (WAL). Later, we provided a detailed description of the commitment protocol and associated state management. This was followed by a description of the Cherry Garcia client library, whose implementation in Java we will describe in further detail in Chapter 6. Chapter 8 will be dedicated to the evaluation of Cherry Garcia to understand its characteristics under various environmental conditions. This is done using both micro-benchmarks as well as the YCSB+T benchmark.
Chapter 6

Implementation

In this chapter, we begin by describing the implementation of Cherry Garcia, the Java library, and its various components. Next, we describe the Datastore abstractions for Windows Azure Storage (WAS), Google Cloud Storage (GCS) and Tora, a high-performance key-value store with a REST+T (extended HTTP) interface built using a WiredTiger storage engine. Later, we provide an overview of the implementation of Tora and the extension to the Apache HttpClient library to support REST+T methods.

6.1 Implementation Overview

The Cherry Garcia library is implemented in Java using the standard Java classes available with JDK 1.6 and the Apache HTTP client\footnote{https://hc.apache.org/} library. The UML diagram of the Cherry Garcia library in Figure 6.1 describes the relationship between the Transaction, Datastore, Record and other associated classes.

In Cherry Garcia, timestamps are obtained using a pluggable implementation of the TrueTime API. The default implementation of this references the clock on the local host. This makes it possible to switch to an alternate time system in the future by changing just runtime settings.

A Transaction Status Record (TSR) is needed by every update transaction to capture transaction state. In order to prevent this from becoming a serialisation bottleneck, we can distribute the load exerted upon the Coordinating Data Store
(CDS) to process TSR requests across all the participating data stores. Since a given transaction needs a particular data store to maintain its TSR. Different transactions can store their TSRs in different data stores so that a single data does not become a bottleneck. The transaction identifier used in this scheme is a Universal Resource Identifier (URI) specifying the location of the TSR so that it can be accessed by any client reading the record.

The Datastore implementation (Google, Azure, and Tora) performs version management of data records based on the characteristics of each data store. Two versions of each record, the currently committed/prepared version and the previously committed version, are needed. If the data store does not natively offer version support (like Tora does), the Datastore class encodes both versions within a single BLOB. Garbage Collection is not a concern here since older versions are overwritten with each new version created.
6.2 Datastore abstraction

The Datastore interface (Listing 6.1) is used to access all data store. Since each type of data store provides a different API, this interface is used to hide the details from the Transaction class.

Similarly, the Record class is used to hide the structure of the application data record from the implementation of the library. It is considered to be a BLOB by the rest of the library. This makes it possible to be easily extend it to implement richer data types. The record access keys are always of type String. This can also be extended to support richer key types.

Every method implemented in the library can either succeed or fail. On failure, a DatastoreException that describes the nature of the failure is thrown. The cause of the failure is usually specific to the implementation, deployment and data store.

```java
public abstract class Datastore {
    public abstract void open() throws DatastoreException;
    public abstract void close() throws DatastoreException;
    public abstract RecordHeader head(String key) throws DatastoreException;
    public abstract Record read(String key) throws DatastoreException;
    public abstract Record prev(String key, Record record) throws DatastoreException;
    public abstract Record write(String key, Record value) throws DatastoreException;
    public abstract void delete(String key) throws DatastoreException;
    public abstract void prepare(String key, Record record) throws DatastoreException;
    public abstract void commit(String key, Record record) throws DatastoreException;
    public abstract void abort(String key, Record record) throws DatastoreException;
    public abstract Record recover(String key) throws DatastoreException;
}
```

Listing 6.1: The Datastore interface

6.3 Datastore abstraction for WAS

The Datastore abstraction for Windows Azure Storage (WAS) is implemented using the Apache HttpClient library version 4.3 (or later). An earlier implementation using the “official” WAS Java client was found to be much slower than using the REST API directly though the Apache HTTP client library. In addition, the WAS Java Client implements its own data record caching making it harder to handle data record life cycle management.

Therefore, we used the Apache HTTP client and commons library to implement the Datastore abstraction class called AzureDatastore. It uses the WAS REST API to read (GET), write (PUT) and delete (DELETE) data records.
For the sake of implementation simplicity, each record is stored along with its previous version as a serialised Java object. The record headers are stored as user-defined metadata headers supported by WAS. Each protocol-defined record header is prefixed with the string \textit{x-msft-meta-} and passed as a HTTP request header to enable it to be stored along in the data store.

The previous version of each object is stored at the end of the object data and is pointed to by an additional header field called \textit{x-msft-meta-previous\_offset}. In the current version of the code, the previous version of the object is serialised using Java serialization. This is clearly not optimal but easy to implement. We plan to use a more lightweight and portable serialisation format like Google Protocol Buffers or Twitter’s Thrift in the future.

The \texttt{prepare()}, \texttt{commit()}, \texttt{abort()} and \texttt{delete()} methods use the conditional write operations supported by the WAS REST API described earlier. We take advantage of the conditional write capabilities supported using the \textit{If-Match} and \textit{If-Unmodified-Since} HTTP request headers supported by the REST API.

The \texttt{prepare()} method for WAS is implemented by performing a conditional PUT of a new version of the record along with a copy of the previous version to the data store with the \textit{transaction state} header (\textit{x-msft-meta-transaction\_state}) set to the string “PREPARED”. This is done by setting the value of the \textit{If-Match} HTTP header to the \textit{ETag} of the previously read version of the record. This ensures that the record is written only if no other client has written a new version of the record. The PREPARED header setting ensures that any reader must either recover the record or use the older version. However, if the object is new, the \textit{If-Unmodified-Since} request header with a value set to “Thu, 01 Jan 1970 00:00:00 GMT” is used. This ensures that no object version has been written before the one being written to prevent lost updates and inconsistent writes.

The \texttt{commit()} operation is implemented using a PUT operation on the record using the \textit{If-Match} header with the \textit{ETag} returned during the \texttt{prepare()} or \texttt{recover()} operation. This ensures that the version prepared by the client was the one that is issued the commit request. The \textit{transaction state} set to the string “COMMITTED” along with the \textit{transaction commit time} and other headers respectively. Once committed, the record can be read and used by the data store client without needing recovery.

A prepared operation on a record is aborted using the \texttt{abort()} method which
is implemented by writing the previous version of the record read from the data store. In order to ensure that the version read is the one being aborted, the If-Match header is used using the ETag of the version of the record previously read.

Similarly, the delete operation is implemented using the delete() function that uses the HTTP DELETE method with the If-Match header set to ensure that only the version read by the client is deleted.

6.4 Datastore abstraction for GCS

Google implements its own Java HTTP client that is similar to the Apache HttpClient library to access Google Cloud Storage and other Google Cloud services. The API is subtly different from the Apache version enabling the access control to be handled quite elegantly by the library. This is the reason why we use it to access GCS.

In order to enable consistent writes and allow conditional operations for our implementation, we use the object versioning provided with the GCS 2.0 XML REST API. With versioning set, every object is assigned a generation number returned with the x-goog-generation response header when a record is read. This is essentially a high-resolution (nanosecond precision) timestamp of the record.

We use this generation number as a unique record version identifier along with the x-goog-if-generation-match request header to perform conditional updates on data items. After an object is read, its x-goog-generation header is captured and treated like an ETag in the case of WAS and passed as the value of the x-goog-if-generation-match request header to implement conditional PUT, POST and DELETE operations.

As with WAS, the prepare(), commit(), abort() and delete() methods are implemented using these conditional HTTP operations supported by the GCS REST API to enable the algorithms listed in Chapter 5 to work.

Like WAS, the prepare() method for GCS is implemented by performing a conditional PUT of a new version of the record along with a copy of the previous version to the data store with the transaction state header set to the string “PREPARED”. This is done by setting the value of the x-goog-if-generation-match HTTP header to the value returned in the x-goog-generation header when the
record was previously read. This ensures that the record is written only if no other client has written a new version of the record. The PREPARED header setting ensures that any reader must either recover the record or use the older version. If the object is new, the \textit{x-goog-if-generation-match} header is set to the value 0. This ensures that no object version is written before the current operation completes to prevent lost updates and inconsistent writes.

In GCS, the \textit{commit()} operation is implemented using a PUT operation on the record using the \textit{x-goog-if-generation-match} header with the value returned during the \textit{prepare()} or \textit{recover()} operation to ensure that the version prepared by the client is the one the transactions that issued the commit requested. The \textit{transaction state} is set to the string “COMMITTED” along with the \textit{transaction commit time} and other headers respectively. Once committed, the record can be read and used by the data store client without needing recovery.

A prepared operation on a record is aborted using the \textit{abort()} method which is implemented by writing the previous version of the record read from the data store. In order to ensure that the version read is the one being aborted, the \textit{x-goog-if-generation-match} header is used similar to the \textit{prepare()} and \textit{commit()} methods.

Similarly, the \textit{delete()} method uses the HTTP DELETE method with the \textit{x-goog-if-generation-match} header appropriately set to ensure that only the version read by the client is deleted.

So that transaction recovery can be performed, both the current and previous versions of the object are stored together in the data store. The \textit{x-goog-meta-prefix} is used for record headers defined by Cherry Garcia and the offset to the previous version of the record is stored as a serialized Java object.

\section{Datastore abstraction for Tora}

The extended transaction support provided by Tora (as discussed in Chapter 4), makes it much easier to build the Datastore abstraction to it. The standard GET, PUT, DELETE HTTP methods are used for read(), write() and delete() methods for the Datastore class implementation respectively. The prepare() method uses the extended HTTP PREPARE method provided by Tora. Similarly, the commit(), abort() and recover() methods in the Datastore class use the COMMIT,
6.6 TORA: A REST+T IMPLEMENTATION

ABORT and RECOVER REST+T methods respectively.

In order to perform consistent updates to the data records test-and-set operations are performed using the If-Match standard HTTP header.

Unlike the other two Datasource implementations the Tora Datasource client implementation class does not maintain the current and previous version of all records read. This is made possible due to the availability of the PREV REST+T method which supports the use of If-Match headers to ensure that the previous version of the record returned is the one required by a client.

To improve throughput and reduce the latency impact of new TCP/IP connection establishment, we use HttpClient connection pools to manage persistent connections to the Tora servers.

6.6 Tora: a REST+T implementation

Tora is a REST+T server written in C++ using the Boost Asynchronous IO library2 to handle requests. It is a heavily modified version based off on a Boost example HTTP server implementation. The architecture diagram in Figure 6.2 describes the high-level architecture of the Tora server and REST+T client application. The underlying storage is an instance of WiredTiger [140].

The main loop handles client requests and responds to them asynchronously. The data is stored in a WiredTiger data store instance. Each key and data record has the data stored as a character array. The metadata consists of the TxID as character arrays and ETag, TxStatus, TxCommitTime, TxLeaseTime, TxValidTime, TxEndTime as 64-bit long integer values. The void initialize(const std::string& root) method of the class request_handler, listed in Listing 6.2 performs the task of initialising the data store and defining the schema for each record stored.

The void request_handler::finalize() method performs the task of closing the connection to the WiredTiger store. This is only called at the time of exiting the Tora server.

The WT_SESSION *request_handler::open_session() method opens a new data store session with the WiredTiger store.

2http://www.boost.org/doc/libs/1_59_0/doc/html/boost_asio.html
Figure 6.2: Architecture of the Tora server and REST+T application
6.6. TORA: A REST+T IMPLEMENTATION

static const char∗ RFC1123_DATE_FORMAT = "%a, %d %b %Y %H:%M:%S %Z";

request_handler::request_handler(const std::string& root)
  : root_(root)
  {
    /* empty */
  }

WT_CONNECTION *request_handler::wt_conn = NULL;

void request_handler::initialize(const std::string& root)
{
    WT_SESSION *session;
    WT_CURSOR *cursor;
    int ret;

    if ((ret = wiredtiger_open(root.c_str(), NULL, "create,session_max=1000", &wt_conn)) != 0)
        fprintf(stderr, "Error connecting to %s: %s\n", root.c_str(), wiredtiger_strerror(ret));
    ret = wt_conn->open_session(wt_conn, NULL, NULL, &sess);

    // A key maps to a set of transaction metadata and the value to be stored
    // key -> ETag, TxStatus, TxCommitTime, TxLeaseTime, TxValidTime, TxEndTime, TxID, Value
    ret = session->create(sess, "table:access", "key_format=S, value_format=QQQQQQSS");
}

void request_handler::finalize()
{
    wt_conn->close(wt_conn, NULL);
}

WT_SESSION *request_handler::open_session()
{
    WT_SESSION *sess = NULL;
    wt_conn->open_session(wt_conn, NULL, NULL, &sess);
    return sess;
}

Listing 6.2: Initialisation code for the REST+T service handler

6.6.1 Request-response loop

The first method invoked to handle the incoming request is request_handler
class method void handle_request(const request& req, reply& rep).
It subsequently calls the appropriate request_handler method depending on
the type of the request. This is called request routing in the literature.

void request_handler::handle_request(const request& req, reply& rep)
{
    if (req.path.empty() || req.path[0] != '/'
        || req.path.find("..") != std::string::npos) {
        rep = reply::stock_reply(reply::bad_request);
        return;
    }

    if (req.path[req.path.size() - 1] == '/') {
        rep = reply::stock_reply(reply::bad_request);
        return;
    }

    // Request path must be absolute and not contain ". . . . . . .
    // If path ends in slash (i.e., is a directory) then add index.html".
    if (req.path[req.path.size() - 1] == '/') {
        rep = reply::stock_reply(reply::bad_request);
        return;
    }
CHAPTER 6. IMPLEMENTATION

Listing 6.3: The main REST+T service handler routine

Listing 6.4 shows the implementation of the handler for the GET method. WiredTiger stores the data records under the key which equals the path passed as part of the REST+T GET request. The record along with the additional metadata like the ETag and TxID are retrieved and returned along with the data value. The GET method must check to see if the current record is in the PREPARED state and return the PREPARED record if its lease has already timed out so that it can be recovered. This is done by looking up a record with the key composed of the concatenation of the record key path and /../prep. For instance, a key /data/savings will have a PREPARED record with key /data/savings/../prep. There can never be a conflict of this name with a regular record because HTTP (and therefore, REST+T) normalises the path that consists a relative path like ../ or ./ ensuring that no legitimate request for such a path will ever be passed to the handler.

If the PREPARED record exists, it is returned. If not, the current record and its metadata is returned. The WT_SESSION is used to create a transaction following which the WT_CURSOR is used to search for the record within the context of a transaction. This transaction is created using a call to WT_SESSION method begin_transaction(sess, "isolation=snapshot") and committed or
6.6. TORA: A REST+T IMPLEMENTATION

rolled back using the commit_transaction() and rollback_transaction() method calls respectively.

```c
void request_handler::handle_get(const request& req, reply& rep)
{
    // Fetch the record from the database
    WT_SESSION* sess = req.sess;
    WT_CURSOR* cursor = req.cursor;
    const char* key = req.path.c_str();
    std::string prep_key = req.path + "/.././.pre";
    const char* prep_key = prep_key.c_str();
    uint64_t timestamp = 0, status = 0;
    uint64_t commit_ts = 0, lease_ts = 0, valid_ts = 0, end_ts = 0;
    const char* txid;
    const char* value;
    int ret;
    char date[32];
    struct tm tm;
    struct timeval t;
    uint64_t ts = 0;
    sess->begin_transaction(sess, "isolation=snapshot" );
    strftime(date, sizeof(date), RFC1123_DATE_FORMAT, localtime( 
        ((gettimeofday(&t, NULL), &t.tv_sec), &tm) ));
    ts = (1000000LL * t.tv_sec) + t.tv_usec;
    cursor->set_key(cursor, prep_key, str);
    if ((ret = cursor->search(cursor)) == 0) {
        if (cursor->get_value(cursor, &timestamp, &status, &commit_ts, &lease_ts, &valid_ts, &end_ts, &txid, &value) == 0) {
            sess->commit_transaction(sess, NULL);
            rep = reply::content_reply(reply::ok, value, strlen(value), date, ts, 
                timestamp, status, txid, commit_ts, lease_ts, valid_ts, end_ts);
            return;
        }
    }
    cursor->set_key(cursor, key);
    if ((ret = cursor->search(cursor)) == 0) {
        // key -> timestamp/ETag, TxStatus, TxCommitTime, TxLeaseTime, TxValidTime, TxID, Value
        if (cursor->get_value(cursor, &timestamp, &status, &commit_ts, &lease_ts, &valid_ts, &end_ts, &txid, &value)) {
            sess->rollback_transaction(session, NULL);
            rep = reply::stock_reply(reply::internal_server_error, date, ts);
        } else {
            sess->commit_transaction(sess, NULL);
            rep = reply::content_reply(reply::ok, value, strlen(value), date, ts, 
                timestamp, status, txid, commit_ts, lease_ts, valid_ts, end_ts);
        }
    } else {
        sess->rollback_transaction(sess, NULL);
        rep = reply::stock_reply(reply::not_found, date, ts);
    }
    return;
}
```

Listing 6.4: The main REST+T service handler routine
CHAPTER 6. IMPLEMENTATION

6.7 Apache HttpClient REST+T Extension

The REST+T extensions to the Apache HTTP client library are fairly easy to implement. For instance the code listed in Figure 6.5 is all that is needed to implement the extension to the HttpEntityEnclosingRequestBase class in order to work with any existing application using the Apache HttpClient Java library. The key to the functionality of this class is the getMethod() method that returns the value of the static string METHOD_NAME which is set to PREPARE.

The rest of the library uses this to construct the REST+T request to the server.

```
@NotThreadSafe
public class HttpPrepare extends HttpEntityEnclosingRequestBase {
    public final static String METHOD_NAME = "PREPARE";

    public HttpPrepare() {
        super();
    }

    public HttpPrepare(final URI uri) {
        super();
        setURI(uri);
    }

    @Override
    public String getMethod() {
        return METHOD_NAME;
    }
}
```

Listing 6.5: Implementation of the REST+T API extension of Apache HttpClient for the PREPARE method

6.8 Chapter Summary

We began this chapter by describing the implementation of the Cherry Garcia protocol in a Java library of the same name. We described the details of the Datastore implementation of the Windows Azure Storage (WAS), Google Cloud Storage (GCS) and our own REST+T data store called Tora. We highlighted the subtle differences between these systems and showed that the Datastore abstraction is able to easily hide the somewhat complex interaction with the individual data stores in an efficient manner. The example code in Section 5.3 shows how
easy it is to write transactional applications with data stores that do not provide native multi-item transactions.

Subsequently, we described the implementation of Tora in the C++ programming language using the Boost library and its Asynchronous IO classes to provide a REST+T interface to the WiredTiger storage engine. The use of the WiredTiger storage engine API within the Tora code were also described in detail using code snippets.

Later in this thesis we will discuss how we use the Java implementation of Cherry Garcia for our performance evaluations in Chapter 8. The evaluations also cover the Tora data store implementation to evaluate the correctness and performance characteristics of the REST+T API extensions to standard HTTP.
Chapter 7

The YCSB+T Benchmark

Database system benchmarks like TPC-C and TPC-E focus on emulating database applications to compare different DBMS implementations. These benchmarks use carefully constructed queries, executed within the context of transactions, to exercise specific RDBMS features and measure the throughput achieved. Cloud services benchmark frameworks, like YCSB, on the other hand, are designed to evaluate the performance of distributed NoSQL key-value stores. Early examples of these systems did not support transactions, and so the benchmarks use single operations that are not inside transactions. Recent implementations of web-scale distributed NoSQL systems, like Spanner and Percolator, offer transaction features to cater to the needs of this new breed of web-scale applications. This inability to wrap multiple operations into a single transaction has exposed a gap in the available benchmarks.

In this chapter, we begin by identifying the issues that need to be addressed when evaluating transaction support in NoSQL databases. This is followed by a description of the YCSB+T framework, an extension of YCSB, that wraps database operations within transactions. This framework includes a validation stage to detect and quantify database anomalies resulting from the workload and a way to gather metrics to measure transactional overhead. In this context we describe a workload, called the Closed Economy Workload (CEW), which can be run within the YCSB+T framework for this purpose. Lastly, we discuss execution details of the framework and associated workloads and share our experience with using YCSB+T to evaluate some NoSQL systems in Chapter 8.
CHAPTER 7. THE YCSB+T BENCHMARK

7.1 Background

It is evident that cloud or utility computing [69] is rapidly becoming the deployment architecture of choice for application across large established enterprises and small startups and businesses alike. New data management technologies, broadly classified under the NoSQL database category have come to the forefront. These include Amazon S3 [1], Google BigTable [26], Yahoo! PNUTS [28] and many others. These systems leverage the distributed deployment platforms to support large-scale storage and throughput while tolerating node failures. As a trade-off, they typically offer applications with less in query expressivity (for example, they may not perform joins) and less transactional and consistency guarantees.

Early NoSQL systems, such as Amazon’s AWS Simple Storage Service (S3) [1] or Amazon’s internal-use Dynamo [28], provide little, if any, support for transactions and information consistency. Others allow read operations to return somewhat stale data, and typically only eventual consistency [136] or time-line consistency [28] is offered respectively. Recent commercially available offerings like Windows Azure Storage (WAS) [25] and Google Cloud Storage (GCS) [2] allow the transaction-like grouping of multiple operations but restrict this to involve either a single item or a collection of items that are collocated. Another approach to better support for application logic lies in richer operations such as test-and-set or conditional put.

As we have seen in Section 3.6, several designs have recently been proposed to overcome these limitations of NoSQL databases. This includes Google Percolator [108], G-Store [35], CloudTPS [144], Deuteronomy [85], Megastore [11] and Google Spanner [30].

The question is, how do we evaluate these designs? Traditional data management platforms were evaluated using industry standard benchmarks like TPC-C [133] and TPC-E [134]; these focused primarily on emulating end-user application scenarios to evaluate the performance (especially the throughput, and throughput relative to system cost) of the underlying DBMS and application server stack.

These benchmarks typically run a workload with queries and updates that are performed in the context of transactions, and the integrity of the data is supposed to be verified during the process of the execution of the benchmark. If the data is
corrupted, the benchmark measurement is rejected entirely.

In the case of web-scale data management, especially the “available-despite-failures” key-value stores in the NoSQL category, a new benchmarking framework YCSB \cite{29} has become accepted. The focus of this benchmark is raw performance and scalability; correctness is not measured or validated as part of the benchmark, and the operations do not fall within transaction boundaries (largely because the target systems do not support traditional notions of transactions nor guarantee data consistency). YCSB is actually a flexible framework within which the workload and the measurements can be extended.

Here we introduce a benchmarking framework (called YCSB+T) that retains the flexibility of YCSB by allowing the user to implement the \texttt{DB} Java interface to their database or data store; allows for additional operations apart from the standard read, write, update, delete and scan; enables the user to define workloads in terms of these operations; and most importantly, allows these operations to be wrapped into transactions. Further, it includes a validation stage that enables consistency checks to be performed on the database or data store after the workload had completed in order to detect and quantify transaction anomalies.

This approach is intended to fill the gap between traditional TPC-C-style benchmarks that are designed with transactional RDBMSs in mind and the non-transactional HTTP/web-service benchmarks which have no ability to define transactions.

In this chapter we:

- describe our extension of the YCSB benchmark that we call YCSB+T, with support for transactional operation and workload validation to detect and quantify anomalies resulting from the workload.

- describe a simple benchmark workload, we call the Closed Economy Workload (CEW), built using YCSB+T that we can use to evaluate the performance and correctness of systems.

- Share our experience in using the benchmark to evaluate some NoSQL data stores.
7.1.1 Brief overview of YCSB

The Yahoo! Cloud Services Benchmark (YCSB) is an extensible benchmarking framework that is widely used to measure the performance and scalability of large-scale distributed NoSQL systems like Yahoo! P-NUTS [28] as well as traditional database management systems like MySQL. Figure 7.1 illustrates the architecture of YCSB+T benchmarking framework. The light coloured rectangles represent YCSB components and the dark coloured rectangles represent additions and enhancements that are part of YCSB+T.

When the YCSB+T client is launched, the workload executor instantiates and initialises the workload then loads the data into the database to be tested or executes the workload on the database using the DB client abstraction. The CoreWorkload workload, defined by default with the YCSB framework, defines different mixes of the simple database operations such as read, update, delete and scan operations on the database. The doTransactionRead() operation reads a single record while the doTransactionScan() operation is used to fetch more than one record identified by a range of keys. Data records are inserted using the doTransactionInsert() method and doTransactionUpdate() is issued to perform record updates in the database. The doTransactionReadModifyWrite() reads a
7.2. BENCHMARK TIERS

The sequence of operations on records in the database are defined by settings specified in the workload parameter file. The YCSB framework provides six predefined workloads called workloads A, B, C, D, and E that can be used to generate a variety of read-intensive, update-intensive and scan-intensive workloads on the database respectively. Some of the important settings available in the framework are discussed in Section 7.4.2.

The framework includes a DB client abstractions for various NoSQL and other data stores and databases including Cassandra, HBase, MongoDB and MySQL.

The YCSB+T client is started with a specified number of client threads. Each thread instantiates a workload to perform the specified database operations by interacting with the DB client abstraction. This behaviour can be altered using command-line parameters.

### 7.2 Benchmark tiers

YCSB contains two benchmark tiers for evaluating the performance and scalability of the cloud serving system. In addition, the original proposal [29] discusses two additional tiers to address availability and replication which is discussed and implemented by the YCSB++ [106] benchmark.

We propose two additional tiers for evaluating the transactional overhead on throughput of the data store, and a measure of the transactional consistency of the database as a result of concurrent execution of the benchmark workload.

Tier 1 and 2 are described in detail in the YCSB paper [29] but tiers 3 and 4 are discussed as future work. These are implemented by YCSB++ [106] framework. In the remaining part of this section we provide the details of two additional tiers introduced by YCSB+T to address transactional overhead and transactional consistency.

#### 7.2.1 Tier 5 - Transactional overhead

The Transactional Overhead tier of the benchmark focuses on measuring the overhead of the individual database operations (exposed via the DB client class) when
they are executed in a transactional context. The latency of database Create, Read, Update and Delete (CRUD) operations and scan operations are measured in both transactional and non-transactional modes. In addition, the latency for DB start(), abort() and commit() method calls are collected in both modes.

The Transactional Overhead tier of the YCSB+T framework aims to determine the overhead of transactions and identify the impact of each database operation that executes within a transactional context in comparison to its execution outside a transactional context.

This tier of the benchmarking framework requires the workload executor to capture the metrics associated with the database transactional operations start(), commit() and abort() and the workload methods to capture the latencies measured for the CRUD and scan database operations.

The overhead of transactional access to the data store is measured by calculating the difference in the latency of each database CRUD and scan operation along with the start, commit and abort operations within a non-transactional context in comparison to the latency observed by those performed within a transactional context.

7.2.2 Tier 6 - Consistency

The primary reason to execute database operations within the context of a transaction is to ensure that the ACID transaction properties and consistency in particular are preserved. For instance, the isolation level used to execute transactions has a significant impact on the performance of database operations.

The Consistency tier of the YCSB+T framework is designed to detect consistency anomalies in the data introduced during the execution of the workload and quantify the amount of anomalies detected.

A database validation phase is added to the workload executor in order to achieve this. The database validation process is implemented as a method of the workload class. It involves going over all the records in the database and applying an application-defined consistency check to the content of the database in order to determine the level of consistency of its contents. An application-specific anomaly score is calculated to quantify the degree of inconsistency detected in the data as a result of running the workload. The expectation is that a score of zero
indicates that the data is entirely consistent (as a result of a serialisable execution),
while a value other than zero indicates a degree of inconsistency quantified as an
application-specific numeric value.

**Application-specific Anomaly Score**

An anomaly can happen as a result of various reasons such as write skew, foreign
key violations, or lost updates. The impact of these violations on consistency can
effect different applications in a different way.

**Definition 7.2.1** *The application-specific anomaly score is defined as a function
of the state of the database that numerically quantifies the effect of the anomalies
on the application.*

For instance, in a banking application, it may happen that the difference of one
dollar may requires the daily bank reconciliation process to be run all over again
costing thousands of dollars. Therefore, a difference of one dollar would result in a
number in the thousands and a difference of a larger sum would result in a larger
application-specific anomaly score.

### 7.3 Benchmark details

In this section we describe the architecture of YCSB+T and provide a detailed
descriptions of the enhancements made to support transactional access to data
stores along with the validation stage added to the workload.

#### 7.3.1 Architecture

As seen in Figure 7.1, the workload executor consists of additional functionality to
perform validation. This is achieved by adding method to validate the consistency
of the data store. It is invoked after the workload executor has loaded data into
the database before executing the workload or has completed the execution of the
workload on the database. The `validate()` method, by default, is a no-op and is
only implemented by workloads that perform database validation.
The client thread is able to wrap database operations performed within a transaction by calling `DB.start()` method before calling the workload `doTransaction()` or `doInsert()` method. If the call to this method succeeds, the transaction is committed by calling the `DB.commit()` method else the `DB.abort()` method is called.

The `DB start()`, `commit()` and `abort()` methods are empty, no-op methods by default. This enable the framework to be backward compatible with existing code written to run with YCSB.

### 7.3.2 Closed Economy Workload

In order to evaluate the transactional properties of a database, it is necessary to device a way to perform concurrent operations on it so that potential inconsistencies in the form of anomalies can be introduced. The Closed Economy Workload (CEW) is designed to achieve this goal. The fundamental approach is to have multiple concurrent transactions that perform conflicting operations on an overlapping set of database records to induce anomalies.

This is achieved with a simplified simulation of a closed economy, one in which money does not enter or exit the system during the evaluation period. The economy consist of a collection of predetermined number of accounts and a total cash in the system that is initially distributed evenly across all the accounts. Anomalies are induced using concurrent operations on the individual account records using multiple running threads to update them simultaneously.

This is implemented in a class called `ClosedEconomyWorkload` which extends the `Workload` class. It is a simplified model of a closed economy where everyone has a bank account which has an initial balance of $1000.

#### Load phase

The `doInsert()` method is overloaded to use the `CounterGenerator` to generate keys starting from 0 (or a value specified by the `insertstart` workload parameter) to the maximum key value specified by the `insertcount` workload parameter. Each key identifies an account number that is assigned an initial account balance set to an equal portion of the total amount determined by the `total_cash` workload parameter.
The load phase is invoked by passing the `-load` option to the YCSB+T client. It must be noted that loading can be done using a single thread only. Parallel loads using multiple concurrently running client threads is not fully supported.

**Transaction phase**

The transaction phase is implemented in the `doTransaction()` method in the Workload class. Its primary task is to pick and call the database operations `insert()`, `read()`, `update()`, `delete()` or `scan()` based on their probabilities specified in the workload configuration file. The operations perform the following tasks:

- **doTransactionInsert()** creates a new account with an initial balance or the captured balance from `doTransactionDelete()` operation described below.
- **doTransactionRead()** reads a set of account balances determined by the key generator.
- **doTransactionScan()** scans the database given the start key and the number of records and fetches them from the database.
- **doTransactionUpdate()** reads a record and add $1 from the balance captured from delete operations to it or subtracts $1 from its balance and adds it to the balance captured and writes it back.
- **doTransactionDelete()** reads an account record, add the amount to the captured balance (capture used in `doTransactionInsert()`) and then deletes the record.
- **doTransactionReadModifyWrite()** reads two records, subtracts $1 from the one of the two and adds $1 to the other before writing them both back.

The transaction phase is invoked by passing the `-t` option to the YCSB+T client. The number of parallel client threads can be controlled at launch time using the `−thread < num_threads >` command line option.

**Validation phase**

In the case of CEW, each transaction must maintain an invariant value for the sum of all account values. Thus, an inconsistency is shown as a difference between
the sum of account values at the start and end of the execution. The validation phase is implemented by overloading the validate() method from the Workload class. This method iterates over all the records in the database adding up the account balance and finally validating the total against the total stored after the load stage determined from the value of total_cash workload parameter. Since one would expect more anomalies to be introduced as more operations are performed, we present the metric for inconsistency as the amount of change in total balance, divided by the number of operations that were executed. Specifically, the following is a formal definition of the simple anomaly score:

\[ \gamma = \frac{|S_{\text{initial}} - S_{\text{final}}|}{n} \]

where,

- \( \gamma \) - simple anomaly score
- \( S_{\text{initial}} \) - initial sum of all the accounts
- \( S_{\text{final}} \) - final sum of all the accounts
- \( n \) - the total number of operations executed

Cleanup phase

Another YCSB+T extension is the cleanup phase added to enable the database to be returned to its original state after the records inserted in the load phase and transaction execution phases have been completed. This has been very useful while using WAS and GCS in particular because of the inherent inability to delete buckets and containers in a reliable and efficient manner.

This phase is implemented by invoking the doDelete() method for all the inserted keys to clear up the data store and return it to a pristine condition. It is a useful option when dealing with a large number of keys and deleting them is a tedious manual process when using a web browser.

The cleanup phase is executed by specifying the -clear option to the YCSB+T client. This can be done using a single thread only. Clearing records in parallel using multiple concurrently running threads is not fully supported.
7.4 Running the YCSB+T benchmark

Here we discuss the various steps to setup and run the YCSB+T benchmark.

7.4.1 Setup the data stores

The first step before the benchmark can be run is to make sure the data store access credentials are all setup correctly. The exact details of how this is done depends on the specific data store.

Once the data store credentials are setup, data records can be loaded into the data store. This is done by running YCSB+T with the `-load` option. Listing 7.1 is an example of how this is done.

```
1 $ java com.yahoo.ycsb.Client -db com.yahoo.ycsb.db.RawHttpDB \
   -P workloads/closed_economy_workload -load
```

Listing 7.1: YCSB+T client load data store command

7.4.2 Workload properties

The workload properties file consists of many tuneable parameters. The most commonly altered ones are listed below:

**Ratio of data store operations** The ratio of operations on the records in the data store can be controlled using the workload properties file. This is done by varying the `readproportion`, `updateproportion`, `insertproportion`, `readmodifywriteproportion`, and `scanproportion` parameters to vary the proportion of read, update, insert, read-modify-write, and scan operations. Each of these values must be a value between 0 and 1 indicating the probability of the operation to occur. The sum of all these values must all add up to 1.

**Key distribution** The distribution used to select the keys of the records to operate on is done by setting the `requestdistribution` parameter. The valid values for this parameter can be `zipfian`, `hotspot`, `uniform`, `skewed`, or `exponential` corresponding to the ZipfianGenerator, HotspotGenerator, UniformGenerator, SkewedGenerator and ExponentialGenerator key generator classes respectively.
The \textit{theta} parameter to the Zipfian generator can be controlled by setting the \textit{zipfianrequestdistributiontheta} parameter which takes a value between 0.1 to 0.99 in order to vary the contention from less to more respectively. The default value of the parameter is 0.99.

**Operation count** The number of data store operations to be performed during the transaction execution phase of the benchmark is controlled by the \textit{operationcount} parameter.

**Record count** The number of records to create in the load phase and the number to operate on during the transaction phase is controlled by the \textit{recordcount} parameter.

**Histogram of results** The default behaviour of YCSB (and YCSB+T) is to capture the latency distribution of all the operations on the data store in the form of a histogram. The number of histogram buckets to capture are controlled by the \textit{histogram.buckets} parameter. It is important to note that there is small cost of maintaining the histogram on the performance of the benchmark. Therefore, it is useful to know that it can be set to 0 in order to collect metrics in only one bucket.

### 7.4.3 Defining new workloads

The YCSB class called Workload can be extended to implemented any new functionality. The ClosedEconomyWorkload is an extension of the Workload class. It implements a rudimentary data validation stage by implementing the \texttt{public boolean validate(DB db)} Workload method. The listing below in Listing 7.2.

```java
// Perform validation of the database db after the workload has executed.
// @return false if the workload left the database in an inconsistent state, true if it is consistent.
// @throws WorkloadException
public boolean validate(DB db) throws WorkloadException {
    HashSet<String> fields = new HashSet<String>();
    fields.add("field0");
    System.out.println("Validating data");
    HashMap<String, ByteIterator> values = new HashMap<String, ByteIterator>();
    int counted_sum = 0;
    for (int i = 0; i < recordcount; i++) {
        String keyname = buildKeyName(validation_keysequence.nextInt());
        try {
            db.start();
            db.read(table, keyname, fields, values);
            db.commit();
        } catch (DBException e) {
            System.out.println("Failed to read data: ", e.getMessage());
        }
        db.verify();
        System.out.println("Verifying data");
        db.commit();
    }
    for (String field : fields) {
        if (values.containsKey(field)) {
            ByteIterator iter = values.get(field);
            while (iter.hasNext()) {
                System.out.println(field + " = " + iter.next());
            }
        } else {
            System.out.println("Field " + field + " not found");
        }
    }
    for (String key : keys) {
        if (values.containsKey(key)) {
            ByteIterator iter = values.get(key);
            while (iter.hasNext()) {
                System.out.println(key + " = " + iter.next());
            }
        } else {
            System.out.println("Key " + key + " not found");
        }
    }
    return true;
}
```
7.5. AN EXAMPLE EXECUTION OF YCSB+T


```java
    throw new WorkloadException(e);
  }
  counted_sum += Integer.parseInt(values.get("field0").toString());
}

if (counted_sum != totalcash) {
    System.out.println("Validation failed");
    System.out.println("[TOTAL CASH], " + totalcash);
    System.out.println("[COUNTED CASH], " + counted_sum);
    int count = actualopcount.intValue();
    System.out.println("[ACTUAL OPERATIONS], " + count);
    System.out.println("[ANOMALY SCORE], " + Math.abs((totalcash - counted_sum)/(1.0 * count))
    return false;
  } else {
      return true;
  }
```

7.5 An example execution of YCSB+T

The following is a demonstration of an actual execution of the YCSB+T client where the key-value store server and the client run on the same machine in order to reduce network latency and maximise throughput. The following is an example of the command line used to execute the YCSB+T client with 16 threads as described in Listing 7.3. Note that the number of histogram buckets specified in this particular execution is 0.

```
$ java com.yahoo.ycsb.Client -db com.yahoo.ycsb.db.RawHttpDB \
-P workloads/closed_economy_workload -threads 16 -t
```

Listing 7.3: YCSB+T client command line

The contents of an example of the `workload/closed_economy_workload` properties file is shown in Listing 7.4.

```
recordcount=10000
operationcount=1000000
workload=com.yahoo.ycsb.workloads.ClosedEconomyWorkload
totalcash=10000000
readproportion=0.9
readmodifywriteproportion=0.1
requestdistribution=zipfian
fieldcount=1
fieldlength=100
writeallfields=true
```
Listing 7.4: A CEW workload properties file

YCSB+T Client 0.1
Command line: -db com.yahoo.ycsb.db.RawHttpDB -P workloads/closed_economy_workload -threads 16 -t
Loading workload...
Starting test.
Validation failed
[TOTAL CASH], 1000000
[COUNTED CASH], 999971
[ACTUAL OPERATIONS], 1000000
[ANOMALY SCORE], 2.9E-5
Database validation failed
[OVERALL], RunTime(ms), 124619.0
[OVERALL], Throughput(ops/sec), 8024.458549659362
[UPDATE], Operations, 200206
[UPDATE], AverageLatency(us), 1536.4616944547117
[UPDATE], MinLatency(us), 1202
[UPDATE], MaxLatency(us), 80946
[UPDATE], Return=0, 200206
[UPDATE], >0, 200206
[COMMIT], Operations, 1000000
[COMMIT], AverageLatency(us), 0.083521
[COMMIT], MinLatency(us), 0
[COMMIT], MaxLatency(us), 795
[COMMIT], Return=0, 1000000
[COMMIT], >0, 1000000
[START], Operations, 1000000
[START], AverageLatency(us), 0.081408
[START], MinLatency(us), 0
[START], MaxLatency(us), 658
[START], Return=0, 1000000
[START], >0, 1000000
[READ-MODIFY-WRITE], Operations, 100103
[READ-MODIFY-WRITE], AverageLatency(us), 6.128447698870164
[READ-MODIFY-WRITE], MinLatency(us), 5
[READ-MODIFY-WRITE], MaxLatency(us), 187
[READ-MODIFY-WRITE], >0, 100103
7.6  CHAPTER SUMMARY

In this chapter looked at YCSB+T, an extension of the Yahoo! Cloud Services Benchmark (YCSB), with the ability to wrap multiple database operations into single a transaction along with a database validation stage that executes after completing the workload.

We also described the Closed Economy Workload (CEW) that can be used to evaluate the performance of a data store using a mixture of read and read-modify-write operations to simulate an application scenario. The workload consists of a validation phase that evaluates the consistency of the data and measures any anomalies that are detected.

This benchmark is suitable to performance test systems that provide transactions in cloud-based NoSQL systems as well as traditional databases. We have used it extensively for the experimental evaluations performed in Chapter 8.

Listing 7.5: Output of running the CEW workload

We evaluate the Cherry Garcia library and the Tora REST+T implementation using the YCSB+T benchmark in Chapter 8 Section 8.2.6 to Section 8.2.13.
We found that the YCSB+T framework can be used to do an apples-to-apples comparison between competing data stores and enables the application developer to define a workload that simulates the application closely. The workload validation stage is particularly useful for validating the consistency guarantees of the system and measuring the number and type of anomalies introduced.

In addition, the framework is completely backward compatible with YCSB enabling existing benchmark code to run without any modification enabling existing users to easily migrate their benchmarks.

In the future we will develop additional workloads that will target specific anomalies that are observed at various transaction isolation levels\(^\text{[12]}\) and develop measures to quantify these. These will be executed against our client coordinated transaction library and distributed key-value store as well as publicly available cloud services like Google Cloud Storage (GCS) and Windows Azure Storage (WAS).

We have released the source code for these workloads and the enhancements made to the YCSB+T framework under the Apache 2 license\(^1\). We are also currently exploring the possibility of incorporating the changes into the main YCSB source tree so that the greater community can benefit. In the future we will explore ways to integrate with YCSB++ to take advantage of its distributed client execution, coordination and monitoring capabilities that will be useful for running web-scale simulations against web-scale transactional NoSQL key-value stores.

\(^1\)https://github.com/akon-dey/YCSB
Chapter 8

Evaluations

In this chapter, we evaluate the various proposals from earlier chapters of this thesis. Our methodology is experimental: we take a prototype implementation of a proposed protocol, subject it to a workload, and measure characteristics such as throughput, abort rate, or the amount of inconsistencies in the data. Some experiments aim to understand how the protocol acts as features of the workload or environment vary. Other experiments compare the proposed protocol to a baseline.

We begin with evaluating the REST+T protocol for access to a single store. We use Tora as a reference implementation. One focus of this evaluation is to compare it with a commonly used naïve approach which uses application-level optimistic concurrency control based on modification time. The primary goal of this experiment is establish whether the REST+T framework is suitable for application development.

Later in the chapter, we evaluate the Cherry Garcia protocol using its Java implementation. First, we explore its characteristics and suitability towards applications that access heterogeneous data stores. Due to the restrictions with the public clouds available to us, the remaining experiments are done with multiple instances of one store implementation on dedicated hardware. With this we determine the protocols scalability characteristics under various application and deployment scenarios. We then measure the effectiveness of the protocol optimisations, and we determine the impact of environmental parameters like time skew and network latency.

Finally, we look at our uses of the YCSB+T benchmark in order to reflect
on its effectiveness as a benchmark for evaluating the performance, throughput and transactional behaviour of NoSQL databases and data stores. In particular, we look at its ability to measure anomalies introduced as a result of data access contention.

8.1 Evaluating REST+T and Tora

In this section, we begin by defining a micro-benchmark to determine the correctness characteristics of REST+T under very high contention and compare it with other techniques like conditional writes over HTTP. We then use it to evaluate REST+T using Tora and discuss the results of the evaluation.

8.1.1 Micro-benchmark for performance and correctness of REST+T

The purpose of this micro-benchmark is to determine the performance and correctness behaviour of REST+T. We do this by accessing the server with very high degree of concurrency and evaluate it in comparison with conditional writes over HTTP. We call this micro-benchmark “2 Account Transfer” (2AT).

To do this, we use a simple application implemented to read two records that simulate two bank accounts. Initially, each account is assigned a fixed balance of $100,000. The application consists of two concurrently running threads. Each thread randomly selects one of the two records and subtracts an amount of $10 from its balance and then reads the other record and adds the previously subtracted amount of $10 to it. Both records are then written back to the data store in the same order in which they were originally read. If the behaviour of the application is correct, the values stored of the two account records always add up to $200,000, the original sum of both accounts. However, if there are any erroneous read, update or undo operations during the course of running the application, the sum of the balances in the two account records diverges from $200,000.

This benchmark is run in two modes in order to compare REST+T with a commonly used naïve technique to access the same Tora data store. It must be noted that we use the same Datastore abstraction to handle the low-level HTTP and REST+T interfaces to the data store. The Datastore implementation does
In the first case, we use standard HTTP to read and update the records. All read operations use the HTTP GET method while updates are performed with the HTTP PUT method with the If-Match header set with the ETag value returned during the preceding GET operation. If the second of the write operations fails due to an ETag header mismatch, then the first write is undone using a compensatory write. This is common practice (often called application-level optimistic concurrency control). As we will see, it does not guarantee consistent data, although some concurrent races are prevented. The code listing in Listing 8.1 shows the run() method for each 2AT application thread. The code uses the HTTPDatastore class to hide the low-level details involved in data store access over HTTP. However, it does not perform any form of data or connection caching. The ETag handling is abstracted using the Datastore class for ease of use.

In the second mode, the identical operations are performed by the application using the transaction methods of REST+T to communicate with the Tora server using 2-phased writes with the If-Match header set to the ETag value during the GET operation. The details of the usage of the REST+T API is hidden using the REST_TDatastore class.

The code listing in Figure 8.2 illustrates the application code using REST+T API. Section 8.1.2 describes the experiment which is followed by a detailed analysis of the result and an explanation.

```java
1 public void run() {
2     try {
3         this.creds = new HTTPCredentials(new File("conf/HTTPCredentials.props"));
4         this.ds = new HTTPDatastore(this.creds);
5         this.ds.open();
6     } catch (Exception e) {
7         e.printStackTrace();
8         return;
9     }
10     String A = "A";
11     String B = "B";
12     String keyA = A;
13     String keyB = B;
14     long start = System.currentTimeMillis();
15     int tx_count = 0;
16     for (int i = 0; i < this.loops; i++) {
17         try {
18             if (random.nextInt() % 2 == 0) {
19                 // Code for optimistic transactions
20             }
21         }
22     }
23 }
```
keyA = A;
keyB = B;
} else {
keyA = B;
keyB = A;
}
Record recordA = ds.read(keyA);
Record recordB = ds.read(keyB);
int a_balance = Integer.parseInt(new String(recordA.getBody()));
int b_balance = Integer.parseInt(new String(recordB.getBody()));
a_balance += 10;
b_balance -= 10;
recordA.setBody(Integer.toString(a_balance).getBytes());
recordB.setBody(Integer.toString(b_balance).getBytes());
try {
    ds.write(keyA, recordA);
    try {
        ds.write(keyB, recordB);
    } catch (DatastoreException e) {
        recordA.setBody(Integer.toString(a_balance - 10).getBytes());
        ds.write(keyA, recordA);
        throw e;
    }
} catch (DatastoreException e) {
    this.aborts++;
}
} catch (DatastoreException e) {
    tx_count++;
} catch (DatastoreException e) {
    e.printStackTrace();
}
}
long end = System.currentTimeMillis();
try {
    this.ds.close();
} catch (DatastoreException e) {
    e.printStackTrace();
}
System.out.println("Thread: " + this.tid);
System.out.println("Run-time: " + (end - start));
System.out.println("Avg tx/millisecond: " + (tx_count*1.0)/(end - start));
System.out.println("Number of transaction aborts: " + this.aborts);

Listing 8.1: The main loop for each thread executed by the 2 Account Transfer micro-benchmark using HTTP

public void run() {
    try {
        this.creds = new REST_TCredentials(new File("conf/REST_TCredentials.props"));
        this.ds = new REST_TDatastore(this.creds);
        this.ds.open();
    }
8.1. EVALUATING REST+T AND TORA

```java
} catch (Exception e) {
    e.printStackTrace();
    return;
}
String A = "A";
String B = "B";
String keyA = A;
String keyB = B;
TransactionId id = new TransactionId();
long start = System.currentTimeMillis();
int tx_count = 0;
for (int i = 0; i < this.loops; i++) {
    try {
        if (random.nextInt() % 2 == 0) {
            keyA = A;
            keyB = B;
        } else {
            keyA = B;
            keyB = A;
        }
        long startTime = System.currentTimeMillis();
        Record recordA = ds.read(keyA);
        if (recordA.getCommitTime() > startTime || recordA.isPrepared())
            continue;
        Record recordB = ds.read(keyB);
        if (recordB.getCommitTime() > startTime || recordB.isPrepared())
            continue;
        int a_balance = Integer.parseInt(new String(recordA.getBody()));
        int b_balance = Integer.parseInt(new String(recordB.getBody()));
        a_balance += 10;
        b_balance -= 10;
        recordA.setBody(Integer.toString(a_balance).getBytes());
        recordA.setTxID(id);
        recordB.setBody(Integer.toString(b_balance).getBytes());
        recordB.setTxID(id);
        try {
            long commitTime = System.currentTimeMillis();
            recordA.setCommitTime(commitTime);
            recordB.setCommitTime(commitTime);
            recordA.setLeaseTime(commitTime+1000);
            recordB.setLeaseTime(commitTime+1000);
            recordA.setValidTimeStart(commitTime);
            recordB.setValidTimeStart(commitTime);
            ds.prepare(keyA, recordA);
            try {
                ds.prepare(keyB, recordB);
                ds.commit(keyA, recordA);
                ds.commit(keyB, recordB);
            } catch (DatastoreException e) {
                ds.abort(keyA, recordA);
                throw e;
            }
        } catch (DatastoreException e) {
            ds.abort(keyA, recordA);
            throw e;
        }
    }
```
CHAPTER 8. EVALUATIONS

Listing 8.2: The main loop for each thread executed by the 2 Account Transfer micro-benchmark using REST+T

8.1.2 Performance and Correctness of REST+T

Aim: In this experiment we measure the throughput and evaluate correctness behaviour of the REST+T protocol in comparison to the existing approach that uses HTTP with conditional PUT operations by setting the If-Match request header and undoing failed bases by resetting the prior value. The aim is to measure the effect of concurrent access on the performance and correctness behaviour of the two approaches in a high-contention read-write situation.

Method: In this experiment we run the 2 Account Transfer (2AT) micro-benchmark application described in Section 8.1.1 with two threads and a Tora server. Both the application and the Tora server are executed on a single machine. For this evaluation we use a Apple MacBook Air running Mac OS 10.9.2 with 8GB 1600 MHz DDR3 RAM, 128GB SSD and a dual-core 1.8GHz Intel Core i5 processor.

The benchmark is executed 5 times. In each run, there are 10,000 read-modify-write operations executed by each application thread. We measure the number of successfully completed operations versus the number of failed write operations or
8.1. EVALUATING REST+T AND TORA

aborted transaction prepare operations achieved per second. After it completes, we measure the throughput in terms of the completed operations per unit time and calculate the number of anomalies detected as a result of the execution of the application. If no inconsistencies are introduced, the net sum of both records is 200,000. We calculate the number of anomalies by counting the difference between the sum of the amounts in both the accounts and the expected sum of 200,000. Each error creates a difference of 0, 10 or 20 in the sum. We use the difference to determine the number of errors that were permitted.

**Observations:** The graph in Figure 8.1 shows the number of errors permitted as a results of running the micro-benchmark application. The error bars reflect a 95% confidence interval (assuming a normal distribution and taken over 5 runs).

The application using HTTP was unable to detect all types of erroneous PUT operations despite using the conditional PUT operations with the *If-Match* header that uses the *ETag* returned during the preceding GET operation. As a result of this, the number of errors detected after the application was an average value of 2627.60 (based on the difference in sum of both accounts records of $26276) for the HTTP application. In contrast, the resulting difference was 0 for every run of the application when using REST+T indicating no anomaly.

The graph in Figure 8.2 plots the number of operations deemed successful (where both records were successfully written) and the number of failed transactions, REST+T detects more conflicts and operations fail with 189.02 aborts per second versus 51.26 successful commits per second. On the other hand, the HTTP application detects 36.29 failure scenarios and aborts while 266.40 transaction per second are completed.

Figure 8.3 displays the difference in the request latencies between the REST+T and conditional HTTP calls. It is evident that there isn’t any noticeable difference in the observed latency of requests between HTTP and REST+T operations.

The number of aborts using HTTP averaged 36.29 aborted transactions per second versus an average of 189.02 per second using REST+T. The successful transaction using HTTP were 266.40 per second against 51.25 for REST+T. The total number transactions run against the Tora servers were 302.69 per second for HTTP versus 240.28 per second for REST+T highlighting the overhead of the REST+T protocol of approximately 25%, which is a small cost compared to the
potential damage from introduced inconsistencies.

Explanation: As a result of the high degree of contention between the two threads, in the case of REST+T, the PREPARE method fails when there is a conflicting PREPARE call made by the other thread. If this is the second of the two PREPARE operations, the already prepared record is then aborted by calling the REST+T ABORT method. Only if both PREPARE calls succeed does the COMMIT method get called for both records. This is the reason for the much higher proportion of aborts observed when REST+T is used.

The two-phased write required by REST+T requires two times the number of extended-HTTP method calls per record compared to standard HTTP PUT. This accounts for the lower transaction throughput of REST+T compared to standard HTTP. We can conclude that REST+T is suitable for transactional access to data stores under very high contention albeit at a low success rate (25%) and correctness guarantees (0 inconsistencies).

8.2 Evaluating Cherry Garcia

Cherry Garcia provides the ability to access data stores in heterogeneous data stores and enables read and update operations to be performed with transaction
Figure 8.2: Throughput and aborts of HTTP using If-Match vs. REST+T

Figure 8.3: Request latency with HTTP using If-Match vs. REST+T
semantics. The next few sections focus on determining the performance and correctness characteristics of this protocol.

We begin by describing the various benchmarking frameworks we will use to perform our evaluations. This is followed by the description and results of a series of experiments to explore the characteristics of Cherry Garcia under different application scenarios. Each experiment is discussed in detail followed by a detailed explanation of the results followed by a description of the insights gained from it.

In order to effectively evaluate Cherry Garcia, we use the YCSB+T benchmarking framework (described in Chapter 7) for the evaluations in this section. We leverage both the original capabilities of YCSB to measure performance and throughput as well as YCSB+T’s ability to enable multiple data store operations to be grouped into transactions and its ability to detect and measure anomalies introduced as a result of these operations. The setup used to run the experiments is described in detail followed by a series of experiments whose results are analysed and explained.

### 8.2.1 Benchmarking framework

We use various tools to evaluate our implementation of the Cherry Garcia protocol in order to measure and quantify different aspects in our experiments. In our experiments, we use both the YCSB+T benchmark framework from Chapter 7 and also some micro-benchmarks. These are described below in further detail.

### 8.2.2 YCSB+T transactional NoSQL benchmark

The primary purpose YCSB+T is to measure the scalability of a data store API and determine its correctness behaviour under different workloads; YCSB+T can detect any resulting anomalies and measure their number. The benchmark settings are altered in order to vary the degree of concurrency and contention. We can compare the protocol and system under test with other techniques. In particular, we will compare Cherry Garcia with a system that lacks cross-store transactions.

In most of our experiments described in this chapter, we use the YCSB+T workload called the Closed Economy Workload (CEW) described in Chapter 7, Section 7.3.2 for our evaluations. We determine the Simple Anomaly Score (SAS), defined in Section 7.3.2, that is a measure based on the ratio of the difference
between the observed value of the total of all accounts and the expected total of all accounts. Thus, this metric reflects the rate at which cross-item data consistency is compromised due to the lack of ACID transaction guarantees.

### 8.2.3 Micro-benchmark to test Cherry Garcia scalability

The primary purpose of the micro-benchmark (which we will call ”uncontended bank transfer” (UBT)) is to test for scalability of Cherry Garcia without contention and without the processing overhead of a sophisticated benchmarking framework.

We define this simple micro-benchmark that initialises the data store with records, each with a number as its value. Each transaction reads two records and writes a modified value back to each of the records that are read. This is done across multiple threads that do not access common data records. The focus of this benchmark is to measure throughput and the overhead of transactions compared to non-transactional data store access. It is not designed to be used to study the impact of concurrent access or check for data consistency.

The benchmark primarily consists of multiple threads that run in parallel. Each thread creates the number of records according to the transaction batch size in the data store and follows with a loop which consists of starting a transaction, reading the records in the batch, writing them back, and finally committing the transaction. The code listing in Figure 8.3 describes the main routine executed by each thread.

```java
public void run() {
    try {
        this.credential = new AzureCredentials(
            new File("conf/AzureCredentials.properties"));
    } catch (Exception e) {
        e.printStackTrace();
    return;
    }

    this.ds = new AzureDatastore(credential);
    this.tx = new Transaction(ds);

    // populate the data store with the records
    for (int j = 0; j < this.count; j++) {
        try {
            tx.begin();
            String key = "data-record-" + this.tid + "-" + j;
            Record record = new Record("record " + j + "created by thread ");
```
CHAPTER 8. EVALUATIONS

Listing 8.3: The run() method for each thread executed by the uncontended bank transfer micro-benchmark

8.2.4 Experiment setup

The Amazon Elastic Compute Cloud (EC2) [74] infrastructure is used to execute the YCSB+T client that uses the Cherry Garcia client library to access data records in the data stores.

For these experiments, unless otherwise stated, the YCSB+T client is run on m1-xlarge boxes with 8 cores and 15.1 GB RAM and the default 10GB EBS volume. The Tora servers also run on m1-xlarge boxes with 8 cores and 15.1 GB RAM and the default 10GB EBS volume in the same placement region. No special network placement or network settings are used in order to simulate typical application deployment scenarios in a production cloud. The machine uses a AWS native Amazon Machine Image (AMI) based on RedHat/CentOS Linux running
Java JDK 7.

We expect Cherry Garcia to be used in applications which involve operation that are largely single item transactions. However, there are occasions when these applications need transactions that involve a small number of records greater than one where transactional behaviour is essential. Additionally, there is a need to access multiple data stores each of which exposes a different data store specific data access API. The application itself could reside on an end-user machine, like a laptop or mobile phone, or in a data centre. It is expected that as technology and the application usage evolves, the choice of data store will change. As a result, the typical application developer must be able to quickly and reliably switch data store technologies without having to rewrite large parts of the application.

We evaluate if for the following criteria keeping the above scenario in mind:

**Testing for heterogeneity:** Each data store is accessed through the `Datastore` class implementation specific to that data store abstracting the data store specific access protocol and data and metadata storage infrastructure. In these experiments, we use the data store abstraction for Windows Azure Storage (WAS), Google Cloud Storage (GCS) and Tora to make it possible to access them from the YCSB+T benchmark client.

Windows Azure Storage (WAS) is a distributed key-value store which exposes a RESTful HTTP API suitable for storing BLOBs. It provides single item transactions using conditional PUT, POST and DELETE operations. Google Cloud Storage (GCS) is similar to WAS but uses a different authentication mechanism and different HTTP headers. Tora, described in Section 6.6, is a data store that uses WiredTiger as a storage engine and exposes the REST+T API described in Chapter 4.

**Evaluating scalability:** The number of YCSB+T client threads are increased in order to increase the transaction load for the purposes of evaluating the scalability characteristics. Since, each YCSB+T client thread executes within its own context, each individual thread simulates a separate instance of the client application.

**Varying data access contention:** The YCSB+T client uses the `requestdistribution` workload parameter to control the generation of the keys by the key
CHAPTER 8. EVALUATIONS

generator. The possible distributions are zipfian, for a Zipfian distribution for picking the keys; latest, which favours recently selected keys over others; hotspot, which selects keys from a range defined as the hotspot at a higher priority over the remaining keys; and uniform, for using the uniform distribution of keys.

We use the Zipfian distribution by default unless otherwise stated. The Zipfian distribution generator is tuned using the \texttt{zipfianrequestdistributiontheta} which takes a value form 0.01 to 0.99 to increase or decrease the contention further. We use the default theta value of 0.98 in our tests unless otherwise stated.

Another way to change contention is by varying the proportion of read, write, update and scan operations. This is done by specifying a value between 0 to 1.0 for the \texttt{readproportion}, \texttt{insertproportion}, \texttt{updateproportion} and \texttt{scanproportion} properties in the workload configuration file. We use a ratio to 90:10 of read to update (read-modify-write) operations unless otherwise stated.

\textbf{Varying network latency:} The typical network latency in a data centre is somewhere between 1 millisecond and 10 milliseconds. Across data centres in the same metro region it is within 20 milliseconds. In contrast, the latency across the east and west coast of the United States is anywhere between 80 and 100 milliseconds. We place the YCSB+T client in the \textit{US East (Virginia)} region unless otherwise stated. Similarly, we use the US East region for Windows Azure Storage (WAS) for all our experiments involving it while Google Cloud Storage (GCS) does not provide a way to specify the location of the primary storage service.

The lease time is most significant when it is disproportionately small or large compared to the latency. During these experiments the \textit{lease time} is set to 1 second regardless of the network latency. The impact of latency given a constant lease time on the transaction abort rate is discussed in Section 8.2.18.

\textbf{Testing for data correctness:} In our experiments we use the Closed Economy Workload (CEW) to measure the number of anomalies detected and calculate the Simple Anomaly Score (SAS) to determine correctness of the YCSB+T
8.2. EVALUATING CHERRY GARCIA

8.2.5 Request rate limiting in WAS and GCS

During the process of running our benchmarks, we observed that WAS uses a form of request rate limiting. We used a single Windows Azure user account for all our tests. We found that HTTP GET and PUT requests to the service return HTTP error code 503 (*Resource temporarily unavailable*) when the request rates exceed the set limit. In an attempt to overcome this per connection limit, we tried to increase the number of concurrent connections to WAS. However, this did not allow the total rate across the multiple connections to increase.

In addition to rate limiting, some form of traffic shaping is also performed. Response times of GET and PUT requests slow down just before the requests begin to fail with error code 503.

We tried to use more than one container to store the data in an attempt to circumvent the limits imposed by the rate limiter. To ensure identical behaviour, these containers were placed in the same region. However, this did not change the behaviour of the rate-limiting system. No noticeable difference in the limits were observed in the case of multiple containers. This suggests that these limits are imposed on a per account basis. The system also appears to take into account the source IP address from where the requests originated.

We attempted to improve throughput by ensuring that the WAS container was only locally replicated without geo-replication enabled. This did not have a noticeable impact on the permitted request rate. However, we observed a negative impact of increased network latency on the throughput due to the increase in distance of the client from the container.

Like WAS, GCS implements a flavour of request rate limiting that seems to work at two levels. Exceeding the limits causes client access token renewals to fail with HTTP error code 403 (*Forbidden*), where as exceeding service level rate limits results in an HTTP error code 503 (*Resource temporarily unavailable*).

Rate limiting has had a significant impact on our experiments with heterogeneous stores. It prevented us from attempting to scale beyond 16 threads for most micro-benchmark tests due to the unpredictability of the rate limiting mechanism. In a way, this can be considered to be a testament to the design of the rate limiting
systems built into WAS and GCS. They did an excellent job of allowing us to use the system within the allowed limits. On the other hand, ensuring that we were not permitted to abuse it. Due of this, most of our subsequent experiments use only instances of Tora to explore the scalability of our design.

8.2.6 Performance across Heterogeneous Data Stores

Aim: This experiment is designed to evaluate the ability of Cherry Garcia to enable applications to access multiple data stores, measure the performance of transactions spanning heterogeneous data store instances, and determine the scalability characteristics in a heterogeneous setting.

Here we try to evaluate the effectiveness of Cherry Garcia to enable an application to access multiple data stores simultaneously in order to measure any performance overhead.

Method: The experiment is setup with one YCSB+T client running on a single m1-xlarge EC2 host located in the US East region. It accesses data in four data stores simultaneously. The data store instances used for this experiment are a Windows Azure Storage (WAS) container located in US East, a Google Cloud Storage (GCS) account, and two instances of Tora servers running on two m1-xlarge EC2 nodes in the same region as the client.

When a record is written, it is assigned to a data store based on the hash value of its key. This ensures an even spread of the records across each of the four data store instances. In this experiment the client access the data stores in both transactional and non-transactional modes. In the transactional mode, the DB abstraction internally uses the Transaction abstraction to access the Datastore inside the context of a transaction. When not using transactions, the Datastore abstraction is used directly to read and write to the data store outside the context of a transaction.

Observations: The number of client threads were varied from 1, 2, 4, 8, 12, 16, 20 and 24 in both modes to access all four data stores. The table 8.1 lists the observed distribution of requests to each data store instance. The records are accessed using a Zipfian distribution pattern with a read to read-modify-write ratio of 90:10.
8.2. EVALUATING CHERRY GARCIA

Figure 8.4: Throughput for 1 to 24 YCSB+T threads to access WAS, GCS and Tora stores with and without transactions

Figure 8.5: Anomaly scores for 1 to 24 YCSB+T threads to access WAS, GCS and Tora stores with and without transactions
During our initial execution of the experiment, we realised that at higher request rates made to Windows Azure Storage and Google Cloud Storage, we had encountered rate limiting. These rate limits were imposed based on the number of concurrent connections per container and the total number of concurrent requests made to the service. We found that once the rate limiter kicked in, the resulting sustainable request rate was unpredictable to an account or container. This skewed the initial results of this experiment.

In order to prevent request rate limiting from taking effect, we capped the total number of transactions to 400 per second across all YCSB+T client threads using the \(-\text{target 400}\) YCSB+T client command-line argument. This enabled us to overcome the unpredictability of transaction throughput as a result of the imposition of request rate limits during the course of the experiments.

The graphs in Figure 8.4 describes the throughput achieved with 95 percentile confidence intervals observed during 5 runs of the experiment by varying the number of YCSB+T client threads. Figure 8.5 shows the corresponding average Simple Anomaly Scores (SAS) observed with 95 percentile confidence intervals for this experiment. The trend line shows a second degree polynomial growth in the SAS as the number of concurrent threads increases due to the increase in data contention.

Table 8.1 lists the percentage of update (read-modify-write) operation combinations involving two records in each data store. There are two Tora servers (TS1 and TS2) along with one WAS and one GCS container. The bias of the Zipfian distribution key generator is responsible for the skew in the spread of operations across the data stores in favour of TS1 and TS2.

In both non-transactional and transaction access modes, we observe that the throughput increased linearly until 16 threads after which the throughput levels out and does not continue to increase any further. Non-transactional data store

<table>
<thead>
<tr>
<th>Threads</th>
<th>GCS-GCS</th>
<th>GCS-TS1</th>
<th>WAS-GCS</th>
<th>GCS-TS2</th>
<th>WAS-TS1</th>
<th>WAS-TS2</th>
<th>TS1-TS1</th>
<th>TS2-TS2</th>
<th>WAS-WAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.29%</td>
<td>13.36%</td>
<td>12.56%</td>
<td>14.27%</td>
<td>12.27%</td>
<td>11.28%</td>
<td>12.21%</td>
<td>11.30%</td>
<td>5.25%</td>
</tr>
<tr>
<td>2</td>
<td>7.37%</td>
<td>13.70%</td>
<td>12.72%</td>
<td>14.42%</td>
<td>12.40%</td>
<td>11.40%</td>
<td>12.34%</td>
<td>11.42%</td>
<td>5.31%</td>
</tr>
<tr>
<td>4</td>
<td>7.44%</td>
<td>13.82%</td>
<td>12.83%</td>
<td>14.55%</td>
<td>12.51%</td>
<td>11.50%</td>
<td>12.45%</td>
<td>11.52%</td>
<td>5.35%</td>
</tr>
<tr>
<td>8</td>
<td>7.75%</td>
<td>14.30%</td>
<td>13.37%</td>
<td>15.16%</td>
<td>13.03%</td>
<td>11.98%</td>
<td>12.97%</td>
<td>12.00%</td>
<td>5.58%</td>
</tr>
<tr>
<td>16</td>
<td>8.18%</td>
<td>15.20%</td>
<td>14.11%</td>
<td>16.00%</td>
<td>13.76%</td>
<td>12.61%</td>
<td>13.70%</td>
<td>12.67%</td>
<td>5.89%</td>
</tr>
<tr>
<td>20</td>
<td>8.00%</td>
<td>14.88%</td>
<td>13.81%</td>
<td>15.66%</td>
<td>13.46%</td>
<td>12.37%</td>
<td>13.40%</td>
<td>12.40%</td>
<td>5.76%</td>
</tr>
<tr>
<td>24</td>
<td>7.98%</td>
<td>14.84%</td>
<td>13.77%</td>
<td>15.62%</td>
<td>13.43%</td>
<td>12.34%</td>
<td>13.36%</td>
<td>12.36%</td>
<td>5.75%</td>
</tr>
</tbody>
</table>

Table 8.1: Mix of cross data store access combinations
access performs only marginally better in spite of the protocol overhead of transactional behaviour. The aggregate Simple Anomaly Score (SAS) we observed during transactional experiments is zero in all configurations. This shows that the implementation of the protocol ensures data consistency for the application. On the other hand, non-transactional access introduces inconsistencies in the data. The frequency of this increases as the number of concurrent threads goes up.

**Explanation:** The higher throughput for non-transactional access from a given number of concurrent threads is due to transaction coordination overhead of Cherry Garcia. The net throughput after 16 threads levels off for both transactional and non-transactional modes because of the throttling at 400 transactions per second. We will observe a similar levelling-off due to system bottlenecks in performance tests conducted in experiments in Section 8.2.7.

The anomalies detected during this execution exhibits a second order polynomial growth pattern in relation to the number of client threads. This is attributed to the higher data contention as the number of threads increases.

The uneven spread of requests across the different data stores is an artefact of the Zipfian key generator and the way the data store is selected for each record. The data store is picked using the formula $d_{key} = \text{hash}(key) \% n$, where $key$ is the identifying key for a data record, $d_{key}$ is the data store assigned to the key $key$, and $n$ is the total number of data stores. The highly skewed probability of the key being selected across the key space due to the Zipfian distribution causes some data stores to be picked more frequently than the others (see Table 8.1 for details).

### 8.2.7 Scalability without Contention across Heterogeneous Stores

**Aim:** This experiment is designed to measure the scalability and performance in a controlled environment with heterogeneous data stores to overcome the unpredictability of measuring performance resulting from request rate limiting that is activated by high request rates on WAS and GCS data stores.

Many internet applications exhibit the “embarrassingly parallel” scale-out property due to their key-based access pattern from the client. Examples of these are Social Network applications in which content accessed from the browser can easily
be partitioned and scaled-out. Here we look at the scalability characteristics when simultaneously accessing heterogeneous data stores.

**Method:** First, we ran the client to access data only in GCS. The uncontended bank transfer micro-benchmark application was executed on a c1-xlarge Amazon EC2 host in the US East region. We varied the number of records accessed per transaction from 1 to 5 with 1 to 20 client threads in transactional access mode.

Then we repeated the same experiment with access to data only in WAS. As before, the micro-benchmark application was executed on a c1-xlarge Amazon EC2 host in the US East region. We increased the number of records accessed per transaction from 1 to 5 with 1 to 20 client threads in transactional access mode.

**Observations:** The graph in Figure 8.6 shows that with both 1 and 2 records per transaction, the transaction throughput increases as the number of client threads increases. Similarly, the graph in Figure 8.7 illustrates that the throughput growth
as a result of concurrent transactions is linear. However, the transaction throughput diminishes as the number of records read and modified per transaction increases. The graph in Figure 8.8 and 8.9 show the same trend with WAS.

We observe that the application performs much better with WAS than with GCS. An average throughput of 73.05 transactions per second is achieved with GCS at 20 client threads and 1 record per transaction. However, the average throughput with 5 records per transaction at 20 threads is just 6.97 transactions per second.

In contrast, an average throughput of 733.03 transactions per second is achieved with WAS at 20 client threads and 1 record per transaction. However, the average throughput with 5 records per transaction at 20 threads is just 94.29 transactions per second.

**Explanation:** As we increase the number of threads, the throughput of the transactions increases near-linearly. We were able to achieve a peak throughput of 354.819 transactions per second with 20 threads and 1 record per transaction. As the number of records per transaction is increased the throughput declines. This
Figure 8.8: Throughput of 1 and 2 record per transactions with 1 to 20 micro-benchmark threads to access WAS

Figure 8.9: Throughput of 2, 3, 4 and 5 records per transactions with 1 to 20 micro-benchmark threads to access WAS
8.2. EVALUATING CHERRY GARCIA

Figure 8.10: 1, 2 and 3 record transactions with 1 to 20 micro-benchmark threads to access WAS, GCS and Tora

Figure 8.11: 2, 3, 4 and 5 record transactions with 1 to 20 micro-benchmark threads to access WAS, GCS and Tora
### Table 8.2: Observed connection latency with WAS and GCS from Amazon US East region

<table>
<thead>
<tr>
<th>Data store</th>
<th>average</th>
<th>min</th>
<th>max</th>
<th>stddev</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAS</td>
<td>13.426 ms</td>
<td>14.36 ms</td>
<td>12.305 ms</td>
<td>0.77248341 ms</td>
</tr>
<tr>
<td>GCS</td>
<td>168.526 ms</td>
<td>139.28 ms</td>
<td>236.53 ms</td>
<td>39.6724595 ms</td>
</tr>
</tbody>
</table>

is due to the overhead of additional network RPC calls as a result of the Cherry Garcia protocol.

We were not able to get reliable numbers beyond 20 concurrent threads. This is due to the unpredictable behaviour of the request rate limiting infrastructure that is part of the WAS and GCS resulting in unreliable results. The variation in the observed throughput with single record transactions as the number of concurrent threads are increased in Figure 8.10 displays signs of decay in performance as a result of rate-limiting.

The latency characteristics measured during the tests are listed in Table 8.2. The Tora servers were local with an average latency of 2.045 milliseconds. In addition to the rate limiting, the latency plays a negative role in reducing the observed transaction throughput. This shows that, even without contention, a close negative correlation between the network latency and achievable transaction throughput. The impact of network latency is explored in further depth in Section 8.2.18.

### 8.2.8 Scalability of a Single Client Host and Single Server

**Aim:** This experiment explores the scalability characteristics of Cherry Garcia to access a single data store using just one client.

The intention is to emulate internet applications that exhibit “embarrassingly parallel” scale-out property due to their key-based access pattern from the client.

**Method:** In order to further evaluate the scalability and performance of Cherry Garcia, we now abandon the use of WAS and GCS in favour of just using Tora. This is done to escape the limitation imposed on the maximum attainable transaction throughput as a result of request rate-limiting performed by both WAS and GCS. Using Tora makes it possible for the application to maximise the transaction throughput without the data store becoming the bottleneck.
Figure 8.12: Throughput and latency of 1 YCSB+T client against a 1-node Tora cluster

For this experiment, we ran the YCSB+T client and the Tora server on one c1-xlarge EC2 node each in the same AWS region. The number of YCSB+T client threads is increased from 1 through to 64. This way we were able vary the workload without changing the available resources.

```
8 vmsstat 1
2 procs memory swap io system cpu
 r b swpd free buff cache si so bi bo in cs us sy id wa st
4 0 0 14422140 44252 752752 0 0 0 0 56 56 0 0 100 0 0
0 0 0 14422140 44252 752756 0 0 0 0 42 39 0 0 100 0 0
6 0 0 14422148 44252 752772 0 0 0 0 52 58 0 0 100 0 0
0 0 0 14422148 44260 752788 0 0 0 0 38 40 0 0 100 0 0
8 0 0 14422024 44260 752788 0 0 0 0 46 49 0 0 100 0 0
10 0 0 14422024 44260 752832 0 0 0 0 44 50 0 0 100 0 0
1 0 0 14392936 44260 760224 0 0 0 0 2099 2254 3 0 97 0 0
12 16 0 14285260 44268 766308 0 0 0 84 18493 19514 37 2 61 0 0
20 0 0 14236816 44268 766324 0 0 0 0 56463 56982 71 7 22 0 0
14 10 0 14212408 44268 766344 0 0 0 0 72747 73309 70 11 18 0 0
4 0 0 14211804 44268 766360 0 0 0 0 121130 131207 33 8 58 0 1
8 13 0 14215948 44268 766376 0 0 0 0 121130 131207 33 8 58 0 1
16 13 0 14215948 44268 766376 0 0 0 0 141656 126503 17 8 74 0 1
7 0 0 14215900 44276 766396 0 0 0 204 136099 124859 17 6 76 0 1
18 3 0 14214928 44276 766408 0 0 0 0 134687 124018 16 5 78 0 1
0 0 0 14220952 44276 766428 0 0 0 0 111069 100585 9 5 85 0 1
20 0 0 14221712 44276 766444 0 0 0 0 3122 3694 0 0 100 0 0
0 0 0 14222036 44276 766448 0 0 0 0 2544 3189 0 0 100 0 0
22 0 0 14221892 44284 766456 0 0 0 156 2614 3254 0 0 100 0 0
1 0 0 14221892 44284 766472 0 0 0 0 2587 3236 0 0 99 0 0
24 0 0 14221796 44284 766492 0 0 0 0 2568 3225 0 0 100 0 0
```
Listing 8.4: vmstat 1 output on the YCSB+T client host while running YCSB+T
8.2. EVALUATING CHERRY GARCIA

Listing 8.5: vmstat 1 output on the Tora server while running accessed from one YCSB+T host

Observations: Figure 8.12 shows that transaction throughput increased linearly until 16 threads during which the average latency for each request remained within the 500µs mark. As the number of threads are increased beyond 16, the latency begins to increase until it reaches 4.5ms at 64 threads. This increases latency pointing to a performance bottleneck somewhere in the system.

We logged the outputs of netstat, iostat, vmstat, mpstat and sar which are part of the standard UNIX/Linux system monitoring tools. The output of vmstat listed in Figure 8.4 indicates that the System parameters in and cs, which correspond to the number of interrupts (including clock interrupts) and context switches respectively, are both very high during certain times when the benchmark is executed as seen between lines 15 and 19. The vmstat results from the Tora server host shown in Figure 8.5 does not reveal a bottleneck.

Explanation: On close observation of the netstat output, it appears as though the network stack is under stress because the socket send buffers are constantly full. The output of vm_stats and iostat did not indicate any significant CPU and memory utilisation bottleneck and neither the disk is overly utilised at both the YCSB+T client host and Tora server.

As the number of parallel YCSB+T threads running within the same JVM increases, the thrashing caused by the context switches between threads also has an impact on the achievable net transaction rate by each process. In addition to the context switches between threads, there is an impact of garbage collection on the performance of the JVM.

8.2.9 Impact of network on latency and throughput

Aim: In this experiment we explore the effect of the network on scalability characteristics of Cherry Garcia when one client application with 16 threads accesses a single data store.

The modern enterprise is comprised of a complex web of (potentially thousands
of) machines that are interconnected in complex network topologies. Here we try to understand the effect of the network on the Cherry Garcia protocol and establish some baseline performance numbers.

Method: In order to understand the impact of network on transaction throughput more deeply, we fixed the number of client threads at 16. Next, we set the client so that it runs for a period accessing the Tora server through its private IP access followed by a period when its public IP address is used. For this experiment, we setup one YCSB+T client host and one Tora server both on c2-xlarge EC2 nodes in the same AWS region. Note that these experiments were run at a later period of time using different hardware (c1-xlarge vs c2-xlarge) due to the unavailability of the original type.

Observations: The graph in Figures 8.13 shows the throughput from a YCSB+T client with 16 threads accessing the private and public IP address interfaces of the Tora server. Correspondingly, the graph in Figure 8.14 shows the difference in latency when switching from a private to a public network interface. Explanation:
8.2. EVALUATING CHERRY GARCIA

We observe an increase in average latency when the data store is accessed using its private IP address versus its public IP address is 75.77 microseconds per commit operation. At the same time, the resulting transaction throughput decreases by 534.98 transactions per second. The increase in the network latency and lower throughput with the public IP address can be attributed to the overhead of network packet routing in AWS. In the rest of our experiments, we only use the public IP address so that we are able to simulate an end-user scenario where heterogeneous stores reside in diverse, public data stores like GCS and WAS.

8.2.10 Scalability with Multiple Client Hosts and Server

**Aim:** In this experiment, we attempt to scale the workload further by increasing the number of client hosts and the number of Tora servers.

This experiment is designed to test the scalability of the CG protocol and measure its throughput under scale-out conditions. This is done by increasing the number of client machines. This is designed to simulate, at a smaller scale, a typical application deployment pattern in a large enterprise.
CHAPTER 8. EVALUATIONS

Throughput (transactions/second)

YCSB+T Client Threads

0 2000 4000 6000 8000 10000 12000 14000 16000
1 32 60 91 121 152 182 213 244

Figure 8.15: Throughput of 4 YCSB+T client hosts against a 4-node Tora cluster

Method: The experiment is performed using a setup similar to that used in the experiment in Section 8.2.8 using 4 application hosts and a cluster of 4 Tora servers in EC2. We vary the number of threads on each client host from 1 through to 64 threads. This results in the overall application thread count range of 4 to 256.

The key space is distributed among YCSB+T client running on each host so that each has its own partition of keys whose records are stored across all the Tora servers. This avoids issues related to time synchronisation across the different hosts which can be off by up to 100 milliseconds even when NTP is running. The impact of time skew on the performance of the protocol is evaluated in Section 8.2.19.

Observations: The graph in Figure 8.15 shows that the performance on each host scales linearly until 16 threads (an aggregate of 64 threads across 4 client hosts) and then flattens out. We observed that the socket send buffers on the servers were full suggesting a network bottleneck at the client.

Explanation: The behaviour seen across four client hosts fits the same pattern we observed in Section 8.2.8 when only one client and one server were used. This suggests that the scalability is bottle-necked at the client.
8.2. EVALUATING CHERRY GARCIA

8.2.11 Scale-out Test with 8 YCSB+T Clients and 4 Tora Servers

**Aim:** We run this experiment to further attempt to test the scalability of the CG protocol and measure its throughput under scale-out conditions. This is done by fixing the number of threads per application host and varying the workload on the Tora servers by increasing the number hosts where the client application executes.

**Method:** We ran YCSB+T with a mix of 90:10 read to read-modify-write operations in a Zipfian data access pattern with theta set to 0.99 in order to explore the scale-out capability of Cherry Garcia. We used 1 to 8 client application hosts each running YCSB+T client 16 threads. All tests use the same 4-node Tora cluster. We collected the throughput numbers from 5 runs spread across these clients and aggregated the throughput.

**Observations:** The graph in Figure 8.16 shows that the peak client performance (when there are 16 threads on each YCSB+T client) scales linearly as more clients are used. With 8 client hosts, we reach an average peak throughput of 23288 transactions/second.
Explanation: If we limit the number of parallel threads running on a single YCSB+T client host to 16, the bottleneck is not encountered. This allows the commitment algorithm to scale-out linearly to 8 clients. We did not test with a greater number of clients due to the cost of hardware.

8.2.12 Effect of Key Access Distribution on Abort Rates

Aim: This experiment is aimed at determining the effect of contention on transaction abort rate. The YCSB+T framework, like YCSB, provides the ability to define the sequence of keys selected to be accessed in the data store. This is done by setting the type of distribution to be used by the key generator. Each key generator can be used to simulate different types of application usage patterns. YCSB+T provides a choice of four key generators to pick from. These are:

- **uniform** - keys are picked using a uniform distribution. That is the probability of a key being generated is equal across the entire key space.

- **hotspot** - keys are picked with a bias where a fraction of the key space, defined by the hotspotdatafraction (default 0.2), is picked more frequently than other keys. The parameter hotspotopnfraction (default 0.8) specifies the fraction of operations where a key from the hotspot is picked.

- **zipfian** - keys are picked using a Zipfian distribution where the probability of a key being generated across the entire key space.

- **latest** - keys are picked in a way such that the latest key has a higher probability of being picked over the others in the key space.

Method: In this experiment, we maintain the number of YCSB+T threads at 16 and ran the benchmark using one Tora server with a client and server host in AWS using c3-3xlarge configuration. Then we vary the distribution used by the key generator by setting the requestdistribution workload property to uniform, hotspot, zipfian and latest respectively.

Observations: We observe that the throughput across distributions does not change noticeably as shown in Figure 8.18. Figure 8.20 shows a slight difference.
8.2. EVALUATING CHERRY GARCIA

in latency. Figure [8.19] provides a clearer view of the differences in throughput. However, the differences in the percentage of aborted transactions as seen in Figure [8.17] clearly shows that the contention is much higher with the latest and zipfian settings. An aborted transaction rate of 18.934% is observed with latest as opposed to 3.942%, 0.552% and 0.186% for zipfian, hotspot and uniform distributions respectively.

Explanation: The latest distribution has a very high contention on the most recently picked key. This causes significant read-write and write-write contention on that record. This is the reason for the high rate of aborted transactions when the latest distribution is used. The Zipfian key generator has a lower probability of picking the same key and hence has a lower probability of contention over a key space of 100,00 records. This is the reason for the comparatively lower number of aborts in the tests using the Zipfian key generator. The uniform and hotspot key generators have much lower inter-thread contention due to the large key space of 100,00 records and relative large hotspot of 20% records which amounts to just 2,000 records.

After considering this behaviour and the targeted use cases for Cherry Garcia, we determined that the Zipfian distribution would be best suited for our tests. It has reasonably high contention to expose concurrency problems while also simulating applications where there are a few records that are being constantly accessed and modified while others are relatively untouched. This is typical of large-scale internet applications which exhibit a long-tail.

8.2.13 Impact of Contention on Abort Rates

Aim: In light of the previous experiment, we pick the Zipfian key generator so that we can increase the read and write contention with a small number of transactions over a large key space of 10,000 records. We vary the contention by changing the value of theta from 0.10, 0.30, 0.50, 0.70, 0.90 through to 0.99.

Method: The experiment is setup to run on two Amazon EC2 m3.2xlarge servers, one running as a YCSB+T client and the other as a Tora server. The number of YCSB+T client threads is set to 16 with a constant read to read-write ratio of 50:50 for 1 million transactions.
Observation: The graph in Figure 8.21 shows that the aborts increase as the contention increases, however, the number of aborts are infrequent even with extreme contention. All the plots include error margins with 95 percentile confidence intervals. With theta set to 0.99 the average aborts detected is 33,810 per million transactions over 5 runs. This decreases to 5898.4, 1911.6, 1862.2, 1888.6 to 1885.4 for theta values of 0.90, 0.70, 0.50, 0.30 and 0.10 respectively.

The graph in Figure 8.22 depicts the corresponding average transaction throughput (both aborts and commits) for the same experiment. The peak transaction throughput is higher when theta is set to 0.99 with an average of 1900.198 transactions per second which drops about 2.45% as it approaches a theta value of 0.1. The transactions throughput of 1870.71, 1863.72, 1855.93, 1854.48 and 1853.59 transactions completed per second is observed for theta values of 0.90, 0.70, 0.50, 0.30 and 0.10 respectively.

The graph in Figure 8.24 shows the number of reads with 95 percentile error margins across theta values from 0.1, 0.3, 0.5, 0.7, 0.9 to 0.99. It was observed that the number of read failures, as a result of concurrent updates, is not affected by varying the contention. We will explore this aspect further in Section 8.2.14 and Section 8.2.19 later in this chapter.

Explanation: This increase in the transaction completion rate as a result of the higher number of aborts can be attributed to the shorter running time for aborted transactions. In the context of the YCSB+T workload, failed operations can be a result of an abort due to a failure of the client to read a valid version of a record due to conflicting an update or due to the failure to commit a read-modify-write operation as a result of commit contention.

As the theta increases the probability is skew of the Zipfian distribution in cases. This causes the contention on a few, “popular”, records to increase. This results in more read-write and write-write contention on these records resulting in higher aborts.

8.2.14 Impact of Read-Write Ratio

Aim: In this experiment we measure the transaction throughput while varying the ratio of read operations to read-modify-write operations. This is used to simulate
8.2. EVALUATING CHERRY GARCIA

Figure 8.17: Aborts measured varying the key generator with 16 YCSB+T client threads against a 1-node Tora cluster

Figure 8.18: Throughput measured varying the key generator with 16 YCSB+T client threads against a 1-node Tora cluster
Figure 8.19: Throughput measured varying the key generator with 16 YCSB+T client threads against a 1-node Tora cluster, X-axis expanded to focus on throughput difference.

Figure 8.20: Latency measured varying the key generator with 16 YCSB+T client threads against a 1-node Tora cluster.
8.2. EVALUATING CHERRY GARCIA

Figure 8.21: Aborts measured varying theta with 16 YCSB+T client threads against a 1-node Tora cluster

Figure 8.22: Measured throughput varying theta with 16 YCSB+T client threads against a 1-node Tora cluster
Figure 8.23: Measured throughput varying theta with 16 YCSB+T client threads against a 1-node Tora cluster with expanded y-axis to show throughput difference

Figure 8.24: Measured number of read failures due to an invalid version, varying theta with 16 YCSB+T client threads against a 1-node Tora cluster
### 8.2. EVALUATING CHERRY GARCIA

#### Figure 8.25: Throughput of YCSB+T with 16 threads connecting to a 1 Tora node

different class of applications that have different ratios of read and read-modify-write operations.

**Method:** We ran YCSB+T on 1 client host that is a c1-xlarge EC2 instance in AWS. Similarly, the Tora server used in this experiment is a c1-xlarge EC2 instance situated in the same availability zone. The number of YCSB+T threads used are 1, 2, 4, 8, or 16 threads and the read to read-modify-write ratio is from 90:10, 80:20, to 50:50. In this experiment, the 10,000 records are accessed in a Zipfian distribution pattern.

**Observation:** The graph in Figure 8.25 describes the results of the tests performed. The number of transactions scales linearly up to 16 client threads (this gives approximately 2912 distributed transactions per second with a 90:10 mix of read and read-modify-write transactions respectively). The peak performance with an 80:20 transaction mix and 16 threads is 2281 transactions per second, while a 50:50 mix resulted in a maximum of 1352 transactions per second. The graphs clearly shows that the transactional throughput scales linearly up to 16 threads on a single client host regardless of the mix of data stores being accessed.
Explanation: The proportion of write operations increases are we go from 90:10, 80:20 to 50:50 read to read-modify-write operations. The Closed Economy Workload (CEW) defined a read transaction to read one record while the read-modify-write transaction reads two records, modifies them and writes them back. Therefore, the read transaction involves one read RPC while the read-modify-write transaction involves two read RPCs and the transaction commitment RPCs for the two modified records. The lower achieved throughput from a higher proportion of read-modify-write to read operations is the reason for the lower throughput as the proportion of read-modify-write operations is increases in the transaction mix.

8.2.15 Impact of Number of Records Per Transaction

Aim: This experiment is designed to measure the impact of the number of records modified within a transaction on the transaction throughput. In Section 8.2.14 we observed that the proportion of write operations had a negative impact on the transaction throughput. This experiment will help us understand the usability of Cherry Garcia for applications where the number of records modified in a transaction is large.

Method: We use the UBT micro-benchmark by varying the number of records in each transaction from 1 through to 5 and measure the transaction throughput. This is done with different number of concurrent transactions starting at 1 all the way to 20. The benchmark is setup to run in the AWS EC2 data centre in N. Virginia (US East) and uses a Windows Azure Storage (WAS) container hosted in the US East region.

In order to overcome the unreliable latency experienced in accessing WAS resulting from its rate-limiting, traffic shaping and network latency, we also run the same benchmark to access a Tora server located in the EC2 US East region.

Observations: The graph in Figure 8.26 shows that the average number of transactions for 1 record per transaction on WAS is 40.30 transactions per second which increases linearly to 733.03 transactions per second for 20 concurrent threads. With 2 records transaction, the increase is linear from 11.68 transactions per second for 1 thread to 209.71 transactions per second for 20. There is clearly an
Figure 8.26: Impact of 1 vs 2 records per transaction observed on throughput with WAS

Figure 8.27: Impact of 2, 3, 4 and 5 records per transaction observed on throughput with WAS
Figure 8.28: Impact of transactions with 1 record per transaction observed with and without transactions on throughput with WAS overhead as the number of records involved in the transaction increases from 1 to 2.

The graph in Figure 8.27 plots the transaction throughput for 2, 3, 4 and 5 records per transaction. The overhead is a factor of roughly 2.5 from 1 to 2 records. The obtained throughput is roughly inversely proportional to the number of records involved in the transaction.

The graph in Figure 8.28 shows that there is very little effect of transactions when only one record is involved in the transaction as a result of the 1-phase commit optimisation. In fact, the net throughput with transactions appears to be better than without transaction overhead which is unexpected.

This unexpected observation can be attributed to impact of the change in latency on the transaction throughput. We performed these tests over a period of a few hours. It was observed that the read and write operation latency of the WAS service varied over time. It must be noted that despite the appearance that the non-transactional access has a lower throughput compared to the transactional access, it is within the 95 percentile error margins of the transactional throughput for the entire experiment.

The impact of network latency is an important one. We will discuss the effect
of latency on transaction throughput in further detail in Section 8.2.9.

The graphs in Figure 8.29 and Figure 8.30 clearly show that there is an increasing overhead of transactions as the number of records involved in the transaction increases.

The graph in Figure 8.31 plots the ratio of the transaction throughput when running without transactions to with transactions for different number of records involved in the transaction. The observed ratio for 1 record in the transaction is close to 1. However, this ratio is decreases to just under 1.6 for 2 records per transaction. It drops to between 1.3 and 1.4 for 3 records in a transaction to a bit over 1.2 for 4 records per transaction. With 5 records per transaction it is about 1.2. This shows that there is an increasing overhead of transactions as the number of records involved increases beyond one record. This overhead is most pronounced with two record transaction and becomes less pronounced as the number of records involved increases.

In order to determine the equivalent throughput in the two modes, we plot the ratio of average throughput for 3, 4 and 5 records without transactions versus 2, 3 and 4 records with transactions in the graph in Figure 8.35. Interestingly, the result of the average throughput with one more record without transaction to running transactionally with one less record is consistently close to 1.

**Explanation:** It is clear that there is an overhead of transactions due to the additional writes needed. The number of RPCs during the transaction commit process is given by $2n + 1$, where $n$ is the number of records. This is because each record is first prepared (accounting for the first $n$), then the Transaction Status Record (TSR) is written (accounting for the single additional write) which is then followed by the commit for each record (accounting for the next $n$). However, the resultant latency is lower than $2n + 1$ owing to the parallel commit optimisation (explored in Section 8.2.16). The resultant latency for each transaction is given by $n + n/p(n) + 1$, where $n$ is the number of records and $p(n)$ is the function that defines the permitted number of parallel write operations possible during the commit phase.

### 8.2.16 Evaluating the Parallel Commit Optimisation

**Aim:** As the number of records modified by the transaction increases, there is a
Figure 8.29: Impact of transactions with 2 records per transaction observed with and without transactions on throughput with WAS

Figure 8.30: Impact of transactions with 5 records per transaction observed with and without transactions on throughput with WAS
8.2. EVALUATING CHERRY GARCIA

Figure 8.31: Ratio of throughput without and with transactions with WAS

Figure 8.32: Impact of transactions with 1 record per transaction observed with and without transactions on throughput with Tora
Figure 8.33: Impact of transactions with 5 records per transaction observed with and without transactions on throughput with Tora

Figure 8.34: Ratio of throughput without to with transactions with Tora
8.2. EVALUATING CHERRY GARCIA

Figure 8.35: Ratio of throughput of one less record without to one more record with transactions with Tora

negative impact on the net throughput. We propose a parallel commit optimisation in Section 5.12.2 to address this. This experiment is designed to evaluate the impact of the parallel commit optimisation on the transaction throughput.

Method: We measure the performance of the micro-benchmark in two parallel commit and serial commit phases with 2, 3, 4 and 5 records per transaction against a 1-node Tora instance. In the experiment, the application and the Tora server are deployed on two c1-xlarge EC2 instances in the same EC2 availability zone.

Observations: The graphs in Figure 8.36, Figure 8.37 and Figure 8.38 clearly show that until about 16 threads, where we hit the performance limit, transaction throughput improves with a parallel commit phase. This is consistently true for 2, 3 and 5 records per transaction. After 16 threads, a system bottleneck is encountered following which the implementation with a serial commit phase performs better than the one with the parallel commit phase. The initial improvement in performance is a result of an increase in throughput due to a shorter second phase. Once the throughput bottleneck is encountered, the overhead of the additional threads used to perform the parallel commit has a negative effect on the throughput.
Figure 8.36: Performance comparison of serial vs. parallel commit phase with Tora with 2 records per transaction

Figure 8.37: Performance comparison of serial vs. parallel commit phase with Tora with 3 records per transaction
8.2. EVALUATING CHERRY GARCIA

Figure 8.38: Performance comparison of serial vs. parallel commit phase with Tora with 5 records per transaction

Figure 8.39: Analysis of serial vs. parallel commit phase with Tora with 2 records per transaction
Explanation: The use of the parallel commits improves commit latency, thereby, increasing transaction throughput until the point where the network throughput and JVM resource contention causes this to level out. The levelling out of the throughput earlier in parallel commit mode below that achieved using a serial commit mode indicates that the bottleneck is hit with the parallel mode before it is hits in the serial mode. Therefore, with a relatively few number of records and a few parallel commitment threads, the parallel commit phase is useful in improving transaction throughput.

8.2.17 Evaluating the One-Phase Commit Optimisation

Aim: The number of records involved in a transaction is often one. This is typical in many web-scale applications. It is desirable that the throughput and latency achieved when using the CG protocol when only one record is involved in the transaction be minimal. In this experiment, we measure the impact of the one-phase commit optimisation for the case where there is only one record involved in the transaction.

Method: We measure the performance of the micro-benchmark that accessed a Windows Azure Storage (WAS) container in both transaction and non-transactional modes. We then compare the performance of accessing one record using standard HTTP without transactions for single record versus transactions with one single record.

Observations: For the graph in Figure 8.40, the single record updates in both transactional and non-transactional mode perform similarly. The difference seen in the performance is within the 95 percentile error margins over 5 runs. We see a similar result with Tora in the graph in Figure 8.32.

However, with two records per transaction there is a clear overhead of transactional access. Transaction throughput observed is proportional to the number of records involved in the transaction. The performance increased linearly until 16 threads, after which, no throughput increase is observed.

Explanation: The one-phase commit optimisation ensures that there is no additional RPC performed when there is only one record that is modified in the transaction. The is the same as a normal single record HTTP PUT. However,
Figure 8.40: The effect of 1-phase optimization observed with WAS
Figure 8.41: YCSB+T client read and ICMP ping latencies observed across different AWS EC2 regions with Tora server in US East

there is a slight overhead with transactions due to record caching. This can be seen in Figure 8.34.

8.2.18 Impact of Network Latency

Aim: Modern applications are typically deployed in a diverse, heterogeneous environments and often access the data store via mobile networks and remote locations. The network latency in these situations can be quite large to begin with and can vary significantly over time. In this experiment we explore the impact of network latency as a result of the placement of the client application and the data store server.

Method: In order to compare the different deployment scenarios, we ran the Tora server on a c2-xlarge EC2 instance in the US East (N. Virginia) AWS EC2 region and connect to it from the c2-xlarge EC2 nodes running YCSB+T with 16 client threads from hosts in the US East (N. Virginia), US West (Oregon), Asia Pacific (Tokyo), Asia Pacific (Singapore) and Asia Pacific (Sydney) regions.
8.2. EVALUATING CHERRY GARCIA

Figure 8.42: Throughput from 16 YCSB+T threads across different AWS EC2 regions with Tora server in US East

Figure 8.43: Aborts per second from 16 YCSB+T threads across different AWS EC2 regions with Tora server in US East
Figure 8.44: Impact of latency on transaction throughput across different AWS EC2 regions with Tora server in US East

Figure 8.45: Correlation between transaction abort rate and transaction throughput across different AWS EC2 regions with Tora server in US East
8.2. EVALUATING CHERRY GARCIA

Observations: The average observed latency on commit operations for the various regions is plotted on the graph in Figure 8.41. The mean ICMP ping latency to N. Virginia was found to be 401.6 $\mu$s from within N. Virginia while it was 74820 $\mu$s, 155000 $\mu$s, 239600 $\mu$s and 243000 $\mu$s for Oregon, Tokyo, Singapore and Sydney respectively. The respective latencies observed with read operations were 1625 $\mu$s, 72127 $\mu$s, 159890 $\mu$s, 232405 $\mu$s and 241638 $\mu$s correspondingly.

The effect of this difference in latency results in the net attainable throughput of 6063.213877 transaction per second when the client resides in N. Virginia to 125.58, 58.20, 40.14 and 38.99 transactions per second for Oregon, Tokyo, Singapore and Sydney respectively as shown in Figure 8.42.

The graph in Figure 8.43 plots the observed number of aborted transactions per millisecond. With the application running in N. Virginia, the average number of transaction aborts per second of 24.01 when 16 YCSB+T threads are running. At the same time, the rate of aborts are 0.65/sec for Oregon, 0.28/sec for Tokyo, 0.21/sec for Singapore and 0.21/sec for Sydney respectively.

From the graph in Figure 8.44, it is clear that there is an inverse relationship between transaction throughput and network latency. Similarly, the graph in Figure 8.45 shows a strong correlation between number of transaction aborts and network latency.

Explanation: The higher the network latency the longer it takes for a transaction to commit due to the time taken for the messages to be exchanged between the data stores and the application running the CG library. If we keep the number of threads constant, the transaction throughput is directly impacted by the latency when there is no bottleneck. This is because the amount of time taken to commit each transaction is increased, there by, reducing the number of transactions completed per unit time. As the throughput decreases, the contention is reduced. This causes the aborts to drop as a result of lower contention even though the number of concurrent threads is the same.

8.2.19 Impact of Network Time Skew

Aim: In a practical deployment scenario the application instances are likely to be deployed widely. The obvious consequence of this is that the individual hosts
executing the application will be widespread resulting in potentially significant difference in the actual time on each. We explore the impact of time skew as a result of the limitations of network time synchronisation algorithms in this experiment.

**Method:** We begin with introducing a time skew window from $0\text{ms}$ to simulate no time skew. This is followed by a time skew of to $5\text{ms}$, $10\text{ms}$, $20\text{ms}$, $50\text{ms}$ and $100\text{ms}$. Now, we measure the impact of the time skew on the percentage of transaction aborts, the percentage of read failures due to inability to find a valid version of the record, the transaction throughput, and finally, the observed commit latency.

The experiment is setup so that 100,000 transactions are executed with a YCSB+T client running 16 threads to access a Tora server in the same AWS data centre in N. Virginia. We vary the time skew across executions but keep it constant within each execution of the micro-benchmark.

**Observation:** The graph in Figure 8.46 shows the percentage of aborts starts at 0.41% for a skew of $0\text{ms}$ to 0.47%, 1.42%, 3.19%, 6.49% and 9.63% for $5\text{ms}$, $10\text{ms}$, $20\text{ms}$, $50\text{ms}$ and $100\text{ms}$ respectively. Clearly, the skew has a significant negative
8.2. EVALUATING CHERRY GARCIA

The graph in Figure 8.47 plots the percentage read operation failures against the time skew. It shows that for the time skew varying from 0ms, 5ms, 10ms, 20ms, 50ms through to 100ms the percentage read operations that fail increase from 0.00016%, 0.05849%, 0.89102%, 2.42985%, 5.27119% to 7.98391% respectively.

There is a clear correlation between the time skew and the percentage failures in both transaction aborts as well as read operation failures. The graph in Figure 8.48 clearly illustrates this.

The graphs in Figure 8.49 and Figure 8.50 show that there is not a significant impact on the transaction throughput or commit latency. The slight decline in the number of completed transactions as the skew is increased to 100ms is a result of the aborted transactions and the need to read older versions of objects that is a direct result of the valid version check failures resulting from the time skew.

**Explanation:** Time skew increases the error window ($\epsilon$) for time stamps. This causes the order of events occurring within the span of the error window to be
Figure 8.48: Analysis of percentage of read failures and aborts vs. induced time skew in milliseconds with 16 YCSB+T client threads and 1 Tora server

Figure 8.49: Transaction throughput vs. induced time skew in milliseconds with 16 YCSB+T client threads and 1 Tora server
Figure 8.50: Observed commit latency vs. induced time skew in milliseconds with 16 YCSB+T client threads and 1 Tora server

indistinguishable from each other. This has a negative effect on transaction commits. This is because a transaction, $x_1$, starting after a committed transaction, $x_0$, whose start timestamp is within the error window of the commit time of the previous transaction, fails because it appears to conflict as a result of the failure of the happens-after test. As the time skew increases, the error margin in the timestamps grows, causing the probability of transaction aborts to increase accordingly. The increase in the time skew also causes read failures to increase due to the inability to find a valid version of records because of the timestamp overlap caused by the increased skew. However, an increase in time skew does not have a negative impact on the latency of transactional data access operations or completed (committed and aborted) transactions.

8.2.20 Summary of Cherry Garcia Evaluations

Cherry Garcia (CG) is expected to be deployed in the modern enterprise that provides services over the Internet. The typical deployed system will consist of a bank of beefy application servers and another bank of beefy data store servers
supporting data stores of different types. The application will be exposed to very high request rates from web-based clients and access the data stores with high transaction rates. Each transaction typically involves one record. Some times there are two records but on a few occasions there are three or more.

Another scenario may have a mobile application client that is spread across a wide geographical region. The application stores state in the central key-value store and accesses it using CG. The usage patter is similar to the previous paragraph but the network is significantly less reliable while the network latencies and time skew is notably higher.

We find that CG will perform well when the deployment scenario involves low latency network connections. As the latency increases the abort rates increase due to the higher probability of contention among competing application threads or instances. The protocol enables the net transaction throughput to scale near linearly when contention is low. This is a desirable characteristic of internet scale systems.

However, CG is not suitable for applications where the transactions routinely involve large number of records. The amount of time needed to perform phase 1 of the commit algorithm increases linearly with respect to the number of records. This is clearly not a desirable property when a large number of records are involved. Further optimisations to the algorithm to explore parallelising the prepare phase and grouping operations on records that belong to the same data store can be explored to address this. These optimisations will make the library less generic and will apply to fewer number of data stores.

8.3 Evaluating YCSB+T as a Benchmark

The focus of this section is to determine the effectiveness of the YCSB+T framework to produce useful information when running against the data store under test. In particular, we focus on examining whether the existing YCSB metrics remain useful for performance and scalability, and also whether the transaction overhead and consistency tiers give valuable insight to the extent of anomalies introduced and the overhead of transactions.
8.3. EVALUATING YCSB+T AS A BENCHMARK

8.3.1 Evaluating Performance and Scalability

YCSB+T retains the capability of YCSB to perform performance and scalability tests on the data store under test. The experiments conducted in Sections 8.2.8 to 8.2.11, for instance, use YCSB+T to measure the throughput and scalability of the Cherry Garcia library to access Tora servers.

It retains the metrics gathering capability of YCSB and adds metrics to capture transaction related operational metrics as well. This along with the ability to gather performance and scalability metrics and plot histograms can enable the user to observe the system under test closely and better optimise it to suit the application.

8.3.2 Performance impact of Transactions

YCSB+T can be used to evaluate the performance impact of transactions and its overhead. It can also be used to measure the impact of changing certain parameters and observe the effect on various performance and scalability metrics simultaneously. An example of this is described in Sections 8.2.13 to vary the contention by changing the theta value used by the Zipfian key generator in order to change the order in which the records in the data store are accessed. The benchmark can then be executed with an appropriate workload to understand the impact of the change. Similarly, in Section 8.2.8 we measure the impact of transactions on the throughput to determine the overhead of transactions across varying number of threads.

8.3.3 Evaluating Correctness

YCSB+T extends YCSB by adding a validation stage to the workload. After the performance test has been run, the workload can implement a validation stage specific to the workload. This can be used to evaluate whether the workload has introduced any anomalies to the data stores. We have use this feature widely with the Closed Economy Workload (CEW) to experimentally show that Cherry Garcia can perform transactions across heterogeneous data stores in a scalable manner while ensuring ACID transaction semantics.
8.4 Chapter Summary

In this chapter we evaluated REST+T and Cherry Garcia using YCSB+T and other micro-benchmarks targeted towards particular aspects of these systems. We began by describing the micro-benchmarks and the use of YCSB+T. Then we ran an experiment using a micro-benchmark to evaluate the correctness behaviour of REST+T as an interface to a data store or a web-services endpoint. Then we use YCSB+T and micro-benchmarks to evaluate the support for heterogeneity, scalability and characteristics under different concurrency and network latency situations. We then explored the properties of YCSB+T as a benchmark for web-scale data stored and databases.

REST+T makes it easier to write code and deal with failure scenarios in a more reasonable manner. In our experimentation using a very simplified application scenario with just two records involving two application threads we were able to show that a naïve approach using standard HTTP with conditional writes is significantly harder to use and the possibility of failures is much higher. The latency was as high as 13.138% in our simple, high-concurrency scenario involving just two concurrent application threads. However, at the same time this correctness guarantee provided by REST+T comes at cost of 20.62% lower transaction throughput.

In our experience with developing micro-benchmarks using HTTP, REST+T and Cherry Garcia, we find that the Record, Datastore and Transaction abstraction make it significantly easier to write applications and deal with possible failure scenarios. In addition to this, the data store abstraction provided by the API enables easy migration of application from one data store to another or to incorporate the use of an additional data store much easier.

Experiments were conducted to explore the correctness and performance characteristics of Cherry Garcia using Windows Azure Storage (WAS), Google Cloud Storage (GCS) and our own implementation of a REST+T compliant data store called Tora. Our experience has been that it is possible to easily write applications that ensure ACID transaction compliance using this API. In addition, it is also easier to reason about the correctness of the behaviour of the application in a reliable manner making the application development task much simpler.

The evaluations show that network latency has a significant negative impact on
transaction throughput, overall application performance, transaction abort rates and consistent read failures. The transaction throughput is not impacted by the concurrency level illustrating the protocols scale-out characteristics. However, the transaction goodput is certainly significantly effected.

The two protocol optimisation, one-phase commit optimisation and the parallel commit phase, help to keep the performance overhead of transaction coordination under acceptable limits without compromising the scalability and scale-out characteristics of the algorithm.

Lastly, we looked at YCSB+T as a benchmark and explored its suitability as a general-purpose mechanism to evaluate the performance of web-scale data stores as well as traditional relational databases with SQL interfaces. The ability of YCSB+T to enable operations to be grouped into transaction and subsequently perform validation has proven to be very useful in ensuring that Cherry Garcia was behaving in a correct manner. We have effectively used the Closed Economy Workload (CEW) during our development process to ensure the correctness of the implementation of the coordination as well as the data store abstraction layers.
Chapter 9

Conclusions and Future Work

We began this thesis by identifying a problem: database technology has evolved recently into scale-out, fault-tolerant NoSQL data stores that handle scalability and high-throughput but provide limited transaction capabilities. In particular, many applications need the scalability and fault-tolerant characteristics in most of the use cases. However, a few situations need traditional transaction guarantees on multiple data items. In recent developments, this has been achieved for systems with multiple but homogeneous stores by implementing a middleware that performs the transaction coordination or by implementing it in the data store itself. A third approach has been to implement the transaction coordination in the client while keeping transaction state in the data store.

Our proposal, Cherry Garcia, deals with providing transactions over heterogeneous stores. As detailed in this thesis, we use a client-based method to guarantee snapshot isolation semantics to enable applications to perform operations on multiple data items that may reside on different, heterogeneous data stores in a scalable and reliable manner.

While searching for a solution to this problem we realised that many data stores provided APIs that are insufficient for use in transactional applications. This resulted in the development of the REST+T extension of HTTP and its implementation called Tora.

Once we had solutions, we needed a way to evaluate them. The existing database and cloud serving benchmarks like YCSB were not suitable. This prompted us to develop YCSB+T, an extension of YCSB, with additional features to perform transactions and determine the degree of correctness.
9.1 Conclusions

We began by describing the problem space, presented the background and described the related work in Chapters 1, 2 and 3 respectively. This was followed by a proposal for a better data store API in the form of a HTTP extension called REST+T in Chapter 4. In Chapter 5 we described Cherry Garcia, a protocol and its implementation as a library, suitable for coordinating transactions across heterogeneous data stores supporting different APIs. Chapter 6 covered the implementation of the various systems developed. In order to evaluate CG and REST+T we developed the YCSB+T benchmark described in Chapter 7. This was followed by a thorough evaluation of these systems in Chapter 8. Here we summarise our conclusions from each chapter.

9.1.1 REST+T

Chapter 4 described a transactional extension of HTTP, we call REST+T, that allows scalable transactional access to multiple items in a single data store. Later, we described the implementation details of Tora, a data store that implements the REST+T protocol using WiredTiger as a storage engine.

From our evaluation in Chapter 8 we found that REST+T is linearly scalable and reliable. We performed tests to compare it with the commonly used test-and-set approach using If-Match HTTP header and compensatory transactions. We were able to conclude that REST+T makes it easier to write code and deal with failure scenarios in a more reasonable manner.

In our experiments in Section 8.1 we were able to show that a naïve approach using standard HTTP with conditional writes is significantly harder to use and the possibility of failures is much higher even in very simple applications involving only two records accessed by just two application threads. The latency impact of using REST+T was 13.138% in our simple, high-concurrency scenario involving two concurrent application threads. The correctness guarantee provided by REST+T came at the reasonable cost of a 20.62% lower transaction throughput.

We observed a negative impact of the number of records on the transaction throughput. The network latency and timestamp error margins also have
9.1. CONCLUSIONS

a negative impact on the throughput. This is evident in the evaluations in Sections 8.2.9, 8.2.18 and 8.2.19 respectively.

9.1.2 Cherry Garcia

In Chapter 5, we described a client-coordinated transaction commitment protocol and a Java library implementation called Cherry Garcia that uses a two-phased approach to transaction commits that span multiple data items stored in multiple heterogeneous data stores, provided each displays some (widely provided) properties and functionalities.

Later in Chapter 6, we described implementation details of how the algorithms from the previous chapter were implemented for Windows Azure Storage, Google Cloud Storage and our own implementation of a REST+T implementation called Tora.

We used a combination of micro-benchmarks and the YCSB+T benchmark to evaluate it in Chapter 8. We found that the Cherry Garcia protocol scaled near linearly as the number of clients increased and the number of servers were proportionally increased. This shows a nice scale-out behaviour suitable for modern cloud-based and mobile data stores applications. Further, we were able to explore its correctness characteristics. We also found that network latency has a negative impact on the transaction throughput and the error bounds on the timestamps. Both these factors negatively affected the rate of transaction commits in high contention scenarios.

When the number of records in the write-set of a transaction is really large we expect the transaction prepare phase to take significant time that is proportional to the number of records involved. This will need to be mitigated using some tree-based key ordering scheme. We did not evaluate these scenarios but the impact of the number of records measured in Section 8.2.15 gives us a good indication of what to expect.

9.1.3 YCSB+T

We described an extension of the YCSB benchmark (called YCSB+T) in Chapter 7. YCSB+T enables operations in a workload to be grouped into transactions. The outcome of each transaction, the commit and abort rates, and latencies are
all recorded. A data validation phase is added at the end of the benchmark execution to inspect the data in the data stores to determine if any anomalies were introduced as a result of running the workload. We defined the Closed Economy Workload (CEW) that simulates an economic system that consists of a fixed sum of money and does not interact with any external source of money. For CEW, we define the Simple Anomaly Score (SAS) that defines a measure of how many anomalies were introduced as a result of running the benchmark workload.

In Chapter 8, we used YCSB+T to evaluate Cherry Garcia. We found that it was easy to use and suitable for most of our use cases. Overall, our experience is that it meets many of the guidelines for a good benchmark set forth in Section 2.9.4. As a result of our success with using it, we are currently in the process of making it part of the standard YCSB benchmark.

9.2 Limitations and Future Work

Even though we have made significant progress described in the previous section, there are some limitations that open up possibilities for future work. These are discussed here.

9.2.1 Time

Spanner [30] and other relevant developments in the areas of research and practice have paved the way for open-source implementations like CockroachDB [79] to be developed. CockroachDB implements a version of Hardware Logical Clocks to keep time and to order transactions. It is clear that there is a need for this, as the reliance on NTP [94] and the understanding of its limitations are becoming more widespread. This has lead to extensive research in the area of algorithms and systems to determine and maintain time in distributed systems. We expect a lot of activity in this area and look forward to contributing to it.

9.2.2 REST+T

Our evaluation of REST+T has shown that it has potential to be a suitable way of implementing transactional data store and web-service endpoints. In the near
future, we will present this in the form of an RFC and attempt to refine it further into a IETF\cite{http://www.ietf.org} standard so that it can gain further acceptance across academia and industry.

\textbf{9.2.3 Cherry Garcia}

The Cherry Garcia implementation and our evaluation of it has shown that this approach is both easy to use and performs reasonably well with desirable characteristics suitable for many class of applications. The formal proof of correctness of the algorithm presented in Chapter \ref{chap:cherry} Section \ref{sec:correctness} make this more compelling.

It will be even more useful if the variety of data stores and NoSQL systems it can support is increased by enriching the API. In the near future we will apply this approach to HBase, Accumulo, Cassandra and other NoSQL stores. A key task will be to generalise the technique to make it suitable for accessing different stores to achieve protocol abstraction. Another important aspect would be to leverage upcoming technologies to reliably determine time in distributed systems.

\textbf{9.2.4 Evaluating other approaches}

Cherry Garcia can be used to access data across heterogeneous data stores. However, it is not limited to use in homogeneous settings. Other approaches like the middleware approach have been used in these scenarios. We will evaluate Cherry Garcia against other approaches to further characterise its strengths and weaknesses.

\textbf{9.2.5 YCSB+T}

We are presently working on additional workloads that will target specific anomalies that are observed at various transaction isolation levels \cite{http://www.ietf.org} and develop measures to quantify these. We will run these against our client coordinated transaction library and distributed key-value store as well as publicly available cloud services such as Google Cloud Storage (GCS) and Windows Azure Storage (WAS).

We intend to release these workloads along with the enhancements made to the
YCSB+T framework to the community as open source. We are also currently exploring the possibility of incorporating the change into the main YCSB source tree so that the greater community can benefit. We will also explore ways to integrate with YCSB++ for its distributed client execution along with its coordination and monitoring capabilities that are useful for running large-scale simulations against web-scale transactional NoSQL key-value stores.
Bibliography


[99] Peter Muth, Thomas C Rakow, Gerhard Weikum, Peter Brössler, and Christof Hasse. Semantic concurrency control in object-oriented database


Index

2AT
   2 Account Transfer, 148
2PC, 39

ACID, 1, 8
application-specific anomaly score, 136
availability, 34
benchmark metrics, 42
benchmark properties, 44
Brewer’s CAP theorem, 33

CalDAV, 48
CAP theorem, 33
CEW
   Closed Economy Workload, 138
Cherry Garcia, 3, 4
Client coordinated transactions, 54
Cloud services benchmarks, 58
consensus algorithm, 38
consistency, 34
data item, 65
database, 6
Database benchmarks, 58
database management system, 6
elasticity, 36
endpoint, 65
eventual consistency, 34

GCS
   Google Cloud Storage, 51
HLC
   Hardware Logical Clocks, 57
HTTP, 47
IaaS, 45
Lamport Clocks, 56
linearizable, 53
LLT, 50
logical clocks, 56, 57
Middleware coordinated transactions, 54
NewSQL, 35
NoSQL, 1, 2, 35
NTP, 57
OLTP, 35
online transaction processing
   OLTP, 40
PaaS, 45
Paxos, 38
RDBMS, 35
REST, 48, 65
REST+T, 3, 4
RPC, 48
SaaS, 45
Sagas, 50
scale-out, 36
SGT
  serializability graph testing, 48
SI
  snapshot isolation, 49
timeline consistency, 35
TO
  timestamp ordering, 48
TPC
  TPC-C, 41
  TPC-E, 41
  TPC-W, 42
  Transaction Processing Performance Council, 40
  transaction, 7
  TrueTime, 57
  WAS
    Windows Azure Storage, 51
WebDAV, 48
WFMS, 51
WORM, 38, 53
YCSB, 3, 56, 59
YCSB++, 60
YCSB+T, 3, 4