M.Sc. Coursework Audit Form

Vanderbilt University
Computer Science

Student: Austin Hunt Advisor: Dana Zhang

- **For Registration:** *Please submit this form (signed by advisor only) to the program coordinator by the last day of registration. This is required to verify that your registration supports progression toward satisfying the course distribution requirements.*
- **For Graduation:** *Prior to submitting the Intent to Graduate Form, you must submit this form (signed by advisor only) to the program coordinator. The program coordinator will submit it to the DGS for you.*

List four CS 6000+ courses for at least 12 credit hours. If you have taken more than four such courses, choose four to list here, and list the rest in the section below. Do not include non-CS courses in this section.
Exclusions: CS 7999 / CS 8390

	Course Number	Grade	Course Status *(if applicable)*
1.	CS 6315	A	
2.	CS 6381	A+	
3.	CS 6388	A+	
4.	CS 6387	A+	

List four additional courses approved by the DGS and advisor carrying graduate credit in CS or another department for at least 12 credit hours. At most, one course (3 credits) may be CS 8390.
Exclusions: CS 7999

	Course Number	Grade	Course Status *(if applicable)*
1.	CS 5262	A+	
2.	CS 5278	A+	
3.	CS 5279	A	
4.	CS 5283	A+	

If you intend to graduate under the non-thesis option, list two additional CS courses carrying graduate credit for at least 6 credit hours.
Exclusions: CS 5250 / CS 5270 / CS 5281 / CS 7999 / CS 8390

	Course Number	Grade	Course Status *(if applicable)*
1.	CS 5287	A+	
2.	CS 8395	A+	

I have taken CS 3250 (Algorithms), CS 3270 (Programming Languages), and CS 3281 (Operating Systems) or equivalents:

- [x] Yes
- [ ] No

M.Sc. Degree Type:

- [ ] Thesis
- [x] Non-thesis

GPA: 4.000

Total Credit Hours: 30 / 30
(Including in progress)

Advisor signature _____ Date _____

DGS signature _____ Date _____

Austin J. Hunt

Vanderbilt University School of Engineering

MS in Computer Science Portfolio

Part Time Student

Entered March 2021 | Graduating December 2022

Principal Source of Financial Support: Full-Time Developer Position at College of Charleston

Advised by Dr. Dana Zhang

Courses and Grades			
Course Number	Course Name	Instructor	Grade
2021 Summer			
CS-5278-50	Principles of Software Engineering	Yu Sun	A+
CS-6381-50	Distributed Systems Principles	Aniruddha S Gokhale	A+
2021 Fall			
CS-5283-50	Computer Networks	Taylor T Johnson	A+
CS-5287-50	Principles of Cloud Computing	Aniruddha S Gokhale	A+
2022 Spring			
CS-5279-50	Software Engineering Project	Yu Sun	A
CS-6315-50	Automated Verification	Taylor T Johnson	A
2022 Summer			
CS-5262-50	Foundations of Machine Learning	Charreau Sieanna Bell, Jesse Spencer-Smith	A+
CS-6388-50	Model-Integrated Computing	Janos Sztipanovits, Tamas Kecskes	A+
2022 Fall			
CS-6387-50	Topics in Software Engineering: Security	Sam Hays, Christopher J White	A+
CS-8395-50	Special Topics: Quantum Computing	Charles Easttom	A+

Professional Goals & Interests

While I struggle to set specific career goals in stone on a landscape as kaleidoscopic and shifting as computer science, I know independently of that shifting that I want to solve problems and help people — ideally in a creative and collaborative environment.

Having spent a number of years working in network infrastructure with an encouraging team at College of Charleston that was always ready to explain and improve upon how things worked, I do have a special interest in computer networking — especially as it intersects with automation. I remember the thrill of getting to write my first script for automating firmware upgrades of Tripp Lite UPSes, and then later the excitement of learning how to use Ansible to manage bulk configuration changes across a network, and then later the hands-on fun of using that skill to configure a cluster of RHEL Icinga 2 satellite servers for a new network monitoring and alerting system. Watching what amounts to a series of gears click into place and function together to serve a collective purpose is a beautiful thing. That's why I carry a steadfast interest in understanding, designing, and building systems of moving parts, as was probably evident to Dr. Gokhale in my work for his Distributed Systems course, or perhaps to my parents in my childhood obsession with setting up and subsequently toppling mile-long chains of Dominoes, or crafting weird Rube Goldberg machines. I aim to follow this interest in all of my professional pursuits moving forward, with the understanding that distributed systems can manifest in many forms.

Moreover, anyone who has worked with me longer than a month knows I'm an automation fanatic, and that will certainly be a defining factor in my professional trajectory. This enthusiasm was kickstarted when I was an undergraduate at College of Charleston and Dr. Jim Bowring tasked my classmates and I with building our own automated testing frameworks from scratch for existing open source software projects on GitHub. Over the course of this MS program, I got to dive into many automation-centric projects, some collaborative and some individual, that allowed me to really work on and improve my skills as an automator in the context of a deeper toolkit including tools like Ansible, Google Cloud Platform (GCP), Amazon Web Services (AWS), Vagrant by HashiCorp, Apache Kafka, and distributed datastores like Couchbase. Also, a significant part of my day job involves discussing and identifying opportunities for automation with my teammates and implementing solutions with scripting, API development, and other codeless tools like Power Automate, e.g., automating mass data synchronization from on-premise network shares to SharePoint Online site collections, integrating our ITSM software with Cascade CMS to automate provisioning of accounts for new web content managers, using GitHub Actions and Docker to automate the testing and deployment of a series of in-house web applications, and more. I find this kind of work, and more generally the DevOps domain, highly satisfying, and I aim to continuously integrate it, if you will, into my career as a technical professional.

Last but especially not least, my love for full-stack web development is a major gravitational force in my life (pulling me primarily toward port 8000 on localhost, as Django is my favorite framework); the origin of this force was a research project started with Dr. Ayman Hajja at College of Charleston in the fall of 2017. Our project, which we titled PolyPy, was a pedagogical web application specifically focused on Python education; it allowed instructors to use a set of novel templating frameworks to generate many (*Poly*) quasi-random Python (*Py*) coding challenges, assign those challenges to their students, and view detailed reports to gain insight into both shared and individual strengths and weaknesses related to both reading and writing Python code. From this project (our work on which is published in the 2019 IEEE Frontiers in Education Conference proceedings), I learned that full-stack web development provided the perfect blend of art and problem solving; it was a colorful concoction of coding

and creativity, and I was obsessed. The PolyPy project lasted only for two years but it ultimately ignited my interest in developing for the web — an interest which is still steadily aflame and which will certainly help to illuminate my future career path.

The past year and a half of active engagement in Vanderbilt University's challenge-laden MS program, combined with my full-time, fused IT role spanning networking, DevOps, and full-stack web development, provides me with a unique perspective that I know I can wield confidently moving forward. While I cannot quite concretely picture myself in a specific position, company, or location five years from now, I am searching for opportunities to make positive, real-world impacts in collaborative and creative ways with a motivated team.

Academic Achievements During MS Program

Since starting the MS program in the summer of 2021, I have also held a full-time development job and have maintained a 4.0 GPA while taking two classes each semester. This isn't necessarily the most exciting or sparkling achievement, but nonetheless it is certainly a product of continuous hard work both academically and professionally. I learned early as a high school student that the objective with school is to learn and explore, not to get good grades — good grades tend to be the natural byproduct of excitement and interest, sort of like how wave function collapse in quantum mechanics is the natural byproduct of environmental interaction (I just learned this in Dr. Easttom's Quantum Computing course). I say this to emphasize that my GPA has less to do with an obsession with my record than a real appetite for growing professionally. With this appetite — and a supplemental unwillingness to put my name on things that I'm not proud of — I've made and I continue to make a consistent effort to produce high-quality and often over-the-top work both as a student and as an employee.

Attending remotely with a full-time job, I unfortunately did not participate in any research projects with faculty, nor in any CS-related campus clubs or activities, but I feel my lack of involvement in those areas has been counterbalanced by frequent eye-opening deep dives into unfamiliar CS topics like deep learning, formal verification with symbolic model checking, the design of metamodels with WebGME to craft domain-specific modeling languages (DSMLs), and much more. I've spent many a morning, afternoon, and night reading, writing, and coding as part of this program in pursuit of the idea conveyed by an admittedly overused Einstein quote: **"If you can't explain it simply then you don't understand it well enough."** This program has really pushed me to deconstruct things I do not understand into smaller and smaller components until I can explain the full idea, and thus has undoubtedly improved my ability to both speak and write simply about complex technical topics. Classes like Distributed Systems and Principles of Cloud Computing (both taught by Dr. Aniruddha Gokhale) posed challenges not only of **designing and building** things like distributed publish-subscribe systems and cloud-based data processing pipelines, but also of clearly demonstrating how those systems work. Just a few months ago, a topic like quantum computing seemed beyond cognitive reach and more like an abstract science fiction subject, but I have now written a full five-page paper analyzing a quantum-resistant, lattice-based digital signature scheme called CRYSTALS-Dilithium and have implemented Shor's prime factorization algorithm with Qiskit-based quantum programming as part of Dr. Easttom's Quantum Computing course. Similarly, prior to this program, I didn't have any experience with machine learning, and certainly could not have explained things like backpropagation, overfitting, or feature engineering, but as part of CS-5262, I have now worked hands-on with low-level training, validation and testing of machine learning models for things like employee churn prediction and image classification in Google Colab and have also acquired a bigger-

picture grasp on the machine learning workflow and the determination of a given model's business value (or lack thereof).

Theoretical understanding is certainly critical to growth and success in the computer science domain, and theory has been a major part of this program's curriculum. On that same note, though, learning how to use new tools is also an important part of growth as a developer — one cannot keep up with a shifting landscape without a similarly shifting skillset. Each course that I have taken as a Vanderbilt student has not only introduced me to new tools but has required me to use them in solving equally new kinds of problems. With these applied introductions, I've gotten to work in a practical way with tools like nuXmv for symbolic model checking (which I leveraged for an Icinga 2 case study to tie it into my day job), ZeroMQ for asynchronous message processing (which I used with Guoliang Ding to create distributed publish-subscribe systems), the Spring Framework for building Java-based enterprise applications (which I used to build Cyberbull for simulating stock market investments in a local, risk-free environment), or the lightweight Express web framework for Node.js (which Tucker Hawkinson, Alexandra Falkner and I used as a benchmark to evaluate the performance of our own aptly-named Lickety Split framework).

As part of this program, I have also started writing and publishing computer science articles on platforms like DEV and Medium; it has been my goal for a year or so to start actively engaging with and contributing to the developer community in this form to strengthen my public developer profile. Professor Hays's course on security in software engineering, with its plethora of open-ended homework questions on key cybersecurity topics, has given me the exact push I needed to establish a consistent technical blog, which now contains a series of articles under the umbrella of software and information security, such as A Security Showdown in the Clouds: Comparing the Security Philosophies of GCP and AWS, Breaking Down Modern Trust: Digital Signatures and Their Impact on Business, and Movin' On Up: An Analysis of The Privilege Escalation Vulnerability CVE-2022-26923; each one draws both from my experience as an IT employee in higher education and from material we are covering in CS-6387.

Curriculum Vitae (CV)

Academic Degrees

BS Computer Science

College of Charleston | Charleston, SC | Aug 2015 - May 2019

As a College of Charleston student, I was a member of the Honors College, I worked on research with Dr. Ayman Hajja at the lively intersection of pedagogy and web development, I got to present that research on campus to multiple audiences of varying sizes and technical backgrounds, I competed with a fellow CS major in the 2019 Booz Allen Hamilton Hackathon, I competed in the 2019 National Cyber League cybersecurity competition, I developed a strong foundation as a computational thinker through a challenging CS curriculum, and I ultimately graduated summa cum laude with a cumulative GPA of 3.95/4.0 and a major GPA of 3.93/4.0. I also worked part-time as a Student Network Engineer for the College's IT department from the fall of 2017 until graduation.

MS Computer Science (In Progress)

Vanderbilt University | Nashville, TN (Remote from Greenville, SC) | May 2021 - Dec 2022

As a remote Vanderbilt student working full-time as a developer, I gained significant hands-on development experience spanning the domains of automation, cybersecurity, distributed systems management, cloud computing, computer networking, machine learning, quantum computing, modeling (and metamodeling), and general software development. This hands-on experience was compliemented strongly with rich theoretical exploration and a curricular emphasis on communicating and collaborating with others around technical projects.

Professional Employment

Digital Communications Developer

College of Charleston | Charleston, SC | Feb 2022 - Present

I am one of a two-member web team (now *Digital Communications*) for College of Charleston with a broad range of duties spanning: securely configuring and maintaining on-prem and cloud Linux servers (Ubuntu, RHEL); managing Apache and Nginx web server configurations, including SSL cert renewals, VirtualHost modifications, error troubleshooting, etc.; scripting and programming in JS, Python, Bash, and PowerShell; developing, maintaining, documenting, and testing Flask APIs for system integrations as well as Django web apps to meet business needs at a low cost; automating builds and CI/CD processes with Docker/Docker Compose, GitHub Enterprise, GitHub Actions, and Webhooks; collaborating with our network security team and leveraging CLI network analysis tools to troubleshoot and resolve network issues and optimize network security of web projects; containerizing legacy apps to improve manageability and speed up development time; continuing to maintain and scale Icinga 2 for network monitoring and alerting (inherited from previous position); troubleshooting and resolving technical problems of varying complexity for functional users (tickets); using Git heavily for version control of software and web projects; supporting IAM modernization with Azure AD management and scripting against on-prem AD; helping to manage multiple web tools including multiple content management systems, the Twilio platform for SMS messaging, a Microsoft Teams-integrated

Live Chat solution, and more; rolling out a new SharePoint Online intranet and assisting departments with migrating away from on-premise infrastructure into the cloud; managing the technical end of a new CRM system for student success management; automating data migrations and synchronizations from on-premise resources to the cloud for functional departments (Python & Powershell)

Webmaster

College of Charleston | Charleston, SC | Dec 2019 - Feb 2022

This position included the same responsibilities as above. The new Digital Communications Developer title primarily addressed a discrepancy between the Webmaster position description and pay and actual duties which expanded over time due to increasing campus needs, new projects, and limited IT personnel.

Temp Employee (Web Team)

College of Charleston | Charleston, SC | May 2019 - Dec 2019

I was offered this position upon graduating by the same supervisor who hired me for the Student Network Engineer position. Responsibilities included: implementing and scaling a new Icinga 2 environment for network monitoring and alerting; managing a distributed hierarchy of RHEL servers; using Ansible for distributed configuration deployments; working closely with our network infrastructure and network security teams to enable and troubleshoot zoned-based NCPA/SNMP/SSH monitoring traffic across new and legacy infrastructure

Student Network Engineer (Part-Time During Undergrad)

College of Charleston | Charleston, SC | Oct 2017 - May 2019

This position introduced me to the world of network infrastructure management. Responsibilities included: using ZTerm to manage access switch software; installing, replacing, and configuring access switches and uninterruptible power supplies (UPSes); working with DHCP configuration for IP address assignment to new infrastructure; configuring UPS network interface cards; automating mass UPS firmware upgrades with Python; managing and organizing inventory (cables, switches, and other assets); helping to train new student network engineers; working in a wide variety of environments across campus to manage physical infrastructure

Achievements, Honors & Awards

Context for Cistern Standard Awards: In March of 2022, I worked with our Human Resources department to gather requirements for and set up on their behalf an automated flow (with Power Automate) that would allow College employees to recognize other employees for exhibiting one or more of the institution's Core Values; if approved, an optionally anonymous recognition triggers automated generation of a recognition certificate which gets sent to the person being recognized.

Cistern Standard - Recognized for Innovation

College of Charleston | Charleston, SC | Oct 2022

Anonymous said: *"Austin is an asset to the College! He should be recognized for our core value of innovation for the way he continuously problem solves with the college's many Cascade CMS users*

who have varying degrees of comfortability in a system that has many constraints. Cascade requires a lot of creative solutions and Austin is full of them and is super helpful."


COLLEGE of CHARLESTON
1770

Certificate of Achievement

Recognized by Anonymous

Awarded to

Austin Hunt

Thank you for embodying CofC's Core Value of Innovation and for following The Cistern Standard.

INNOVATION

Office of Human Resources

2022-10-17

Cistern Standard - Recognized for Innovation

College of Charleston | Charleston, SC | Sept 2022

Ashley Pagnotta said: *"Because of some changes Google made to the pricing of their education offerings, we had to transition from having unlimited Google Drive storage to a fairly small, limited amount. I had a LOT of stuff in my Google Drive, both teaching and research related, and I tried a few different ways to move it over to OneDrive with very little success. After posting about my frustrations on Yammer, Austin jumped in and let me know he (possibly in conjunction with other colleagues?) had built a migration tool that we could use. He got me all set up, and of course my ridiculously large amount of data broke the tool multiple times, but he kept working at it until he was able to get everything migrated from Google Drive to OneDrive. I'm sure it was a huge pain on his end, but I really appreciated it!"*

Vanderbilt University Tuition Scholarship

Vanderbilt University | Nashville, TN | 2021 - 2022

Vanderbilt University offered a $30K tuition scholarship for those admitted to enroll in the May 2021 cohort of the online MS in computer science program.

National Cyber League Individual Game, Rank 192/3607

College of Charleston | Charleston, SC | Apr 2019

Below is a screenshot of my individual performance report from the 2019 National Cyber League competition, in which I placed 192/3607; this competition covers ethical hacking techniques related to cryptography, steganography, port scanning, network traffic analysis, log analysis, password cracking, and web application exploitation.

Best Computer Science Research Poster (Departmental Award)

College of Charleston | Charleston, SC | Apr 2019

Outstanding Student Award, CS Department

College of Charleston | Charleston, SC | May 2019

Speaker at College of Charleston Research Speaker Series

College of Charleston | Charleston, SC | Jan 2018

Linda Robertson Scholarship

Linda and Stephen Robertson | Charleston, SC | 2017 - 2019

South Carolina Palmetto Fellows Enhancement

South Carolina Commission on Higher Education | Charleston, SC | 2016 - 2019

South Carolina Palmetto Fellows Scholarship

South Carolina Commission on Higher Education | Charleston, SC | 2015 - 2019

Computer Science Leading Edge Scholarship

College of Charleston CS Department | Charleston, SC | 2015 - 2019

College of Charleston Academic Merit Scholarship

College of Charleston Financial Aid | Charleston, SC | 2015 - 2019

Certifications

Docker Certified Associate (DCA)

Mirantis | Jan 2021

Certified Associate in Python Programming (PCAP)

OpenEDG Python Institute | May 2020

Certified Entry Level Python Programmer (PCEP)

OpenEDG Python Institute | May 2020

Service

Construction Volunteer

East Cooper Habitat for Humanity | Oct 2015 - Apr 2016

Worked with a group of fellow first year students from College of Charleston on the construction of a house — from the foundation to the roof — under fantastic guidance from the ECH staff.

Fundraising with Portraiture

Austin Hunt Portraiture | Jun 2020

Started a fundraising campaign in June 2020 to donate 100% of portrait drawing commission payments to Black Lives Matter fundraisers chosen by the commissioners.

Publications

A Novel E-Learning Platform for Building and Publishing Student-Driven Personalized Lessons

IEEE Frontiers in Education 2020 | Uppsala, Sweden (Remote) | Oct 2020

PolyPy: A Web-Platform for Generating Quasi- Random Python Code and Gaining Insights on Student Learning

IEEE Frontiers in Education 2019 | Cincinnati, OH | Oct 2019

CS Skills & Expertise

Development Languages & Libraries

Python Java Bash PowerShell PHP HTML5 JS TS

CSS3 Sass Jinja2 EJS Apache Velocity GraphQL YAML

Markdown JSON nuXmv React jQuery

Operating Systems for Development:

Ubuntu CentOS RHEL Windows MacOS

Web

Django Express Spring Node.js Flask React Twilio

Apache Web Server Nginx Web Server Heroku Cloudflare

DNS Management SSL certificate configuration and renewal

DevOps & Infrastructure

Git GitHub GitHub Actions Travis CI Docker Docker Compose

Kubernetes SSH & associated security practices Ansible

Google Cloud Platform Grafana Icinga 2 Cron Splunk

Nagios Cross Platform Agent (NCPA) WebGME (modeling & metamodeling)

AWS IAM AWS Serverless Application Model (SAM) AWS CloudFormation

AWS EC2 AWS Cloudwatch AWS Lambda AWS S3 AWS API Gateway

Webhooks Linode (IaaS) Azure Active Directory OAuth configuration

Microsoft Graph Power Automate

Collaboration & Remote Work

SharePoint Online Microsoft Teams Zoom Webex Wrike

Basecamp Slack Keybase Azure DevOps TeamDynamix ITSM

ServiceNow VPN

Knowledge and Mastery of Computer Science Concepts

Substantial Contribution to a Significant Software Artifact

`Cyberbull Investment Simulator - Brokerage Account Management Meets Design Patterns`

This is an application called Cyberbull Investment Simulator that I built with the Spring Framework as a final project for Vanderbilt University CS-5278 (Principles of Software Engineering). The **open-ended problem statement** was as follows: develop a large-scale software project that solves a problem in your life and leverage established software architecture design patterns in its design.

This project is a reflection of my personal interest in the stock market and is an exploration of many Gang of Four design patterns in the context of stocks and brokerage account management. It provides an environment for a user to look at stock market data and act on that data in a simulated brokerage account that's completely free of financial risk. The application makes heavy use of **object-oriented** programming techniques for **structure and organization**, as well as **functional** programming techniques to draw **clear connections** between the **domain intent** (brokerage account management) and the **computation**. It also largely depends on the this Java library for interacting with the Yahoo Finance API, plus this free News API for collecting current news filtered by business names. **Note: the underlying stock database for this application is generated using this CSV file which only includes the S&P 500.**

How Does it Employ Object-Oriented Design Patterns?

The Command Pattern

The command pattern — described in the link above as **a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request** — is used heavily for maintaining records of activity, where "activity" is actually a command invoker that executes and stores concrete implementations of different operations.

At the Account Level

An AccountActivity Commander is used to keep a record of account-level activity, namely transfers, where a TransferOperation is a concrete AccountOperation that gets executed and stored by the Account Activity Commander. This commander is useful for displaying a history of transfer operations on a dashboard.

```
package edu.vanderbilt.cs.cyberbull.core.activity.account;
import edu.vanderbilt.cs.cyberbull.core.activity.account.operations.AccountOperation;
import java.util.ArrayList;
import java.util.List;
public class AccountActivity {
  private final List operations = new ArrayList<>();
  public List getOperations(){
    return this.operations;
  }
  public void executeOperation(AccountOperation operation,
                               boolean saveOperation){
    if (saveOperation) {
        operations.add(operation);
    }
    operation.execute();
  }
}
```

At the Portfolio Level

A PortfolioActivity Commander is used to keep a record of portfolio activity, represented as PortfolioOperations. A PortfolioOperation is most generally an OrderOperation, which is subclassed by MarketOrderOperation and LimitOrderOperation; these operation classes respectively represent a Market Order of a stock and a Limit Order of a stock — the difference between these order types is documented here. Moreover, since a portfolio consists of a set of open positions (a position just represents some investment in some number of shares of a stock), and a position can be updated over time by buying or selling more of that position's stock, Cyberbull also stores the operations affecting a given position in an operation list for that position. This means we can view a history of orders at the portfolio level and at the position level (within that portfolio) on an account dashboard.

```
package edu.vanderbilt.cs.cyberbull.core.activity.portfolio;
import edu.vanderbilt.cs.cyberbull.core.exceptions.InsufficientFundsException;
import edu.vanderbilt.cs.cyberbull.core.activity.portfolio.operations.PortfolioOperation;
import java.util.ArrayList;
import java.util.List;
```

```
public class PortfolioActivity {
  private final List operations = new ArrayList<>();
  public List getOperations(){
    return this.operations;
  }
  public void executeOperation(PortfolioOperation operation,
                               boolean saveOperation)
              throws InsufficientFundsException {
    if (saveOperation) {
      operations.add(operation);
    }
    operation.execute();
  }
}
```

The Strategy Pattern

One important thing to note is that while a bank account simply has a balance associated with it, a brokerage account has both a balance and a core position. A core position represents a balance of uninvested cash just sitting in the account waiting to be invested or transferred somewhere. The *balance* is the sum of the investment values in the account's portfolios and the core position. Now, for handling transfers, we are only interested in the transferable balance of an account, which for a brokerage account is just the core position. So, to find transferable balance of an account that could be one of two different types, Cyberbull uses the strategy pattern. The BalanceFinderContext class serves as the context which takes in an account and routes the "get the transferable balance" logic to the appropriate strategy (there's a bank account balance finder strategy which just returns the balance if it's a bank account, and a brokerage account strategy which returns the core position if it's a brokerage account). This way, the client doesn't need to know what type an account is, or how to handle that type.

```
package edu.vanderbilt.cs.cyberbull.core.account.balancefinder;
import edu.vanderbilt.cs.cyberbull.core.account.Account;
import edu.vanderbilt.cs.cyberbull.core.account.BankAccount;
import edu.vanderbilt.cs.cyberbull.core.account.BrokerageAccount;

public class BalanceFinderContext {
  private BalanceFinderStrategy strategy;
  private final Account account;
  public BalanceFinderContext(Account account){
    this.account = account;
    if (account instanceof BrokerageAccount){
      this.strategy = new BrokerageAccountBalanceFinder();
    } else if (account instanceof BankAccount){
      this.strategy = new BankAccountBalanceFinder();
    }
  }
  public double executeStrategy(){
    return this.strategy.execute(this.account);
  }
}
```

The Factory Method Pattern

With the factory method, you provide an interface for creating objects in a superclass, but you use subclasses to change the type of object that gets created.

Positions

When managing a stock portfolio, you can open different types of positions depending basically on what you are hoping to profit off of. In a nutshell, if you want to profit off of a business losing money, you can open a short position, AKA you can "short" the company's stock. Alternatively, and most commonly, you may want to profit off of a company's growth buy buying shares and selling them at a later date at a higher price; this is called a "long" position. You can check out Investopedia for more information about the finance side of things. Now, for creating positions within the app, there is a PositionFactory interface with a createPosition() method, and an IPositionFactory superclass that implements that interface to create a Position superclass. Below that, there is a ShortPosition class and a LongPosition class, representing the different subclasses, or types, of market positions. Running parallel, there is a ShortPositionFactory and a LongPositionFactory responsible for creating those respective types of Position objects. Below are code snippets for the PositionFactory interface, the LongPositionFactory implementation of that interface, and LongPosition class.

```
package edu.vanderbilt.cs.cyberbull.core.position;
import edu.vanderbilt.cs.cyberbull.core.Stock;

public interface PositionFactory {
  public Position createPosition(Stock stock, double quantity);
}
....
....
package edu.vanderbilt.cs.cyberbull.core.position;
import edu.vanderbilt.cs.cyberbull.core.Stock;

public class LongPositionFactory implements PositionFactory{
  @Override
  public Position createPosition(Stock stock, double quantity) {
    return new LongPosition(stock, quantity);
  }
}
....
....
package edu.vanderbilt.cs.cyberbull.core.position;
import edu.vanderbilt.cs.cyberbull.core.Stock;
import edu.vanderbilt.cs.cyberbull.core.activity.portfolio.operations.OrderOperation;

public class LongPosition extends Position{
  public LongPosition(Stock stock, double quantity,
                      OrderOperation orderOperation) {
    super(stock, quantity, orderOperation);
  }
  public LongPosition(Stock stock, double quantity) {
    super(stock, quantity);
```

```
  }
}
```

Random Attributes

The factory pattern is also used to generate random phrases (for titles and descriptions) for the different account types. Since this is a simulator, it allows for you to randomly generate new accounts without having to manually enter account numbers, routing numbers, titles, and descriptions. Currently, there is a RandomPhraseFactory implemented by RandomBrokerageAccountTitleFactory and RandomBankAccountTitleFactory, which each handle the random generation of phrases appropriate to their respective account types. For example, a brokerage account title may be something like "investments for retirement" but that same random choice would not make sense for a bank account, which may be titled something like "joint savings account" or "credit union".

The Bridge Pattern

The Bridge pattern (for separating abstractions from implementations) is used pretty heavily since the goal with such a large project is to divide the complexity of the underlying code base from the client, or the controller. The Dashboard and DashboardService serve largely as this bridge of communication between the simple front end and complex back end by persistently storing backend implementations as attributes (i.e. *dependency injection*) which get called upon to handle front end requests. The dashboard service is the most persistent object across the controllers, allowing state to be shared and maintained.

```
package edu.vanderbilt.cs.cyberbull.core;

import com.kwabenaberko.newsapilib.models.Article;
import com.opencsv.exceptions.CsvException;
import edu.vanderbilt.cs.cyberbull.core.account.Account;
import edu.vanderbilt.cs.cyberbull.core.account.AccountManager;
import edu.vanderbilt.cs.cyberbull.core.activity.Operation;
import edu.vanderbilt.cs.cyberbull.core.db.StockDB;
import edu.vanderbilt.cs.cyberbull.core.exceptions.InsufficientFundsException;
import edu.vanderbilt.cs.cyberbull.core.news.NewsManager;
import edu.vanderbilt.cs.cyberbull.core.stock_history.DailyHistoryVisitor;
import edu.vanderbilt.cs.cyberbull.core.stock_history.MonthlyHistoryVisitor;
import edu.vanderbilt.cs.cyberbull.core.stock_history.WeeklyHistoryVisitor;
import yahoofinance.histquotes.HistoricalQuote;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

public class Dashboard {
  private final AccountManager accountManager;
  private final NewsManager newsManager;
  private final StockDB stockDB;

  public Dashboard() throws IOException, CsvException {
```

```
  accountManager = new AccountManager();
  newsManager = new NewsManager();
  stockDB = new StockDB();
}
public List getAllStocks(){
  List allStocks = new ArrayList<>();
  getBrokerageAccountList().forEach(ba ->
              ba.getPortfolio()
              .getPositions()
              .forEach(p ->
                  allStocks.add(p.getStock())));
  return allStocks;
}
public boolean addBankAccount(String title,
                              String description,
                              String routingNumber,
                              String accountNumber){
  return accountManager.addBankAccount(
                                     title,
                                     description,
                                     routingNumber,
                                     accountNumber);
}
public boolean removeBankAccount(String accountNumber){
  return accountManager.removeBankAccount(accountNumber);
}
public boolean addBrokerageAccount(String title,
                                   String description){
  return accountManager.addBrokerageAccount(
                                     title,
                                     description);
}
public boolean removeBrokerageAccount(String accountNumber){
  return accountManager.removeBrokerageAccount(accountNumber);
}
public void clearBrokerageAccountList(){
  accountManager.clearBrokerageAccountList();
}
public void clearBankAccountList(){
  accountManager.clearBankAccountList();
}
public boolean addRandomBrokerageAccount(){
  return accountManager.addRandomBrokerageAccount();
}
public boolean addRandomBankAccount(){
  return accountManager.addRandomBankAccount();
}
public List getBrokerageAccountList(){
  return accountManager.getBrokerageAccountList();
}
public List getBankAccountList(){
  return accountManager.getBankAccountList();
}
public void transfer(String from, String to, double amount)
```

```
        throws InsufficientFundsException, NullPointerException  {
    accountManager.transfer(from,to,amount);
  }
  public void transfer(Account fromAccount, Account toAccount,
                       double amount) throws
                       InsufficientFundsException,
                       NullPointerException  {
    accountManager.transfer(fromAccount,toAccount,amount);
  }
  public Account getBankAccount(String accountNumber){
    return accountManager.getBankAccount(accountNumber);
  }
  public Account getBrokerageAccount(String accountNumber){
      return accountManager.getBrokerageAccount(accountNumber);
  }
  public void updateBankAccountBalance(String accountNumber,
                                       double newBalance){
    accountManager.updateBankAccountBalance(
        accountNumber, newBalance);
  }
  public List getAllActivity(){
    return accountManager.getAllActivity();
  }
  public List getSP500(){
    return stockDB.getSP500();
  }
  public Optional getStock(String symbol){
    return stockDB.getStock(symbol);
  }
  public List getDailyStockHistory(Stock stock){
    DailyHistoryVisitor visitor = new DailyHistoryVisitor();
    return visitor.visit(stock);
  }
  public List getWeeklyStockHistory(Stock stock){
    WeeklyHistoryVisitor visitor = new WeeklyHistoryVisitor();
      return visitor.visit(stock);
  }
  public List getMonthlyStockHistory(Stock stock){
    MonthlyHistoryVisitor visitor = new MonthlyHistoryVisitor();
    return visitor.visit(stock);
  }
  public List<Article> getBusinessNews(String businessName){
    return newsManager.getBusinessNews(businessName);
  }
}
```

The Visitor Pattern

The visitor pattern, as mentioned in the link above, "lets you separate algorithms from the objects on which they operate". In this project, the visitor pattern is used to easily provide multiple methods of accessing price history of a stock. Specifically, the visitor (one of the concrete implementations of the StockHistoryVisitor interface) generates the price history of a given stock at an interval determined by the type of visitor, e.g. daily, monthly, or weekly.

That is, different types of visitors "visit" the historical data to generate a differently-intervaled lists of historical price points. The code below demonstrates the MonthlyHistoryVisitor implementation.

```
package edu.vanderbilt.cs.cyberbull.core.stock_history;
import edu.vanderbilt.cs.cyberbull.core.Stock;
import yahoofinance.YahooFinance;
import yahoofinance.histquotes.HistoricalQuote;
import yahoofinance.histquotes.Interval;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
public class MonthlyHistoryVisitor implements StockHistoryVisitor {
  public List visit(Stock stock){
    List history = new ArrayList<>();
    try {
      yahoofinance.Stock yfinanceStock = YahooFinance.get(
                                          stock.getSymbol());
      try{
        history.addAll(yfinanceStock.getHistory(Interval.MONTHLY));
      } catch (Exception e){
        System.out.println("Exception when getting MONTHLY history");
      }
    } catch (IOException e){
      System.out.println(e.getMessage());
    }
    return history;
  }
  public static void main(String[] args){
    Stock tesla = new Stock("TSLA");
    MonthlyHistoryVisitor visitor = new MonthlyHistoryVisitor();
    visitor.visit(tesla);
  }
}
```

The Proxy Pattern

As discussed in the link above, a proxy controls access to some original object, allowing you to perform something either before or after the request reaches that original object. In the context of this project, the proxy pattern comes to life with the Stock class. This is because Cyberbull is largely dependent on YahooFinance. It pulls YahooFinance data at runtime specific to chosen stocks, e.g. opens, closes, volumes, general price information, etc. The proxy pattern is seen in how the internal Stock class is used to interact with and process information from the Yahoo Finance API. It's also seen in the use of the NewsFinder which utilizes the SDK for this free news API) to send customized requests to filter out news and handle custom formatting of the responses from that API.

The Chain of Responsibility Pattern

This is pretty typical of web applications where requests get routed from a front end down through a chain of backend modules.

The Iterator Pattern

The iterator pattern is used to iterate over historical quotes of a stock and build a stringified JSON object. Also, the `iterator()` method of the CSVReader class within the Stock Database class (StockDB) is used to handle the iterative parsing of a static CSV file of the S&P 500 stocks. This is the class that forms the underlying primary database of stocks for the application. *Note: it currently only includes the S&P 500 Stocks but can be extended.*

```
package edu.vanderbilt.cs.cyberbull.core.db;
import com.opencsv.CSVReader;
import com.opencsv.exceptions.CsvException;
import edu.vanderbilt.cs.cyberbull.core.Stock;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class StockDB {
  private final ArrayList sp500;
  public StockDB() throws IOException, CsvException {
    sp500 = new ArrayList<>();
    parseCSV();
  }
  private void parseCSV() throws IOException, CsvException {
    CSVReader reader = new CSVReader(
        new FileReader("src/main/resources/static/csv/sp500.csv"));
    Iterator iter = reader.iterator();
    iter.next(); // header
    while (iter.hasNext()){
      String[] record = iter.next();
      String symbol = record[0];
      String businessName = record[1];
      String industry = record[2];
      Stock stock = new Stock(symbol); // symbol
      stock.setBusinessName(businessName);
      stock.setIndustry(industry);
      sp500.add(stock);
    }
  }
  public List getSP500(){
    return sp500;
  }
  public Optional getStock(String symbol){
    return sp500.stream().filter(
        stock->stock.getSymbol().equals(symbol)
    ).findFirst();
  }
}
```

Functional Programming

This project makes very heavy use of functional programming with streams, filters, tons of lambda expressions, and other items in this category. To provide some key examples:

Example 1: Checking for Existing Open Positions Before Trade

When making a trade (i.e. buying or selling a stock), Cyberbull uses functional programming to quickly filter and find out if there is already an existing open position for the stock being bought (or sold):

```
Optional positionOptional = account
    .getPortfolio()
    .getPositions()
    .stream()
    .filter(position -> position
        .getStock()
        .getSymbol()
        .equals(symbol)
    ).findFirst();
```

Example 2: Getting the Total Balance of a Brokerage Account

To get the balance of a brokerage account, as mentioned above, we need the sum of all of the total values of the current open positions plus the core position (uninvested, sitting money) of the account.

```
@Override
public double getBalance() {
    return this.corePosition + getPortfolio().getPositions()
                               .stream()
                               .map(Position::getCurrentValue)
                               .reduce(0.0, Double::sum);
}
```

Example 3: Getting a Stock from the Database

To get a specific stock by a symbol from the Stock Database, Cyberbull streams the list of SP500 stocks and filter with a predicate asking if the current stock's symbol matches the input.

```
public Optional getStock(String symbol){
    return sp500.stream().filter(
            stock -> stock.getSymbol().equals(symbol)
    ).findFirst();
```

Example 4: Quickly Building a Stock History JSON Map

To quickly build a list of JSON objects representing key values of historical quotes of a given stock, Cyberbull streams the output of the visitor (there's a visitor for monthly history, daily history, and weekly history) and adds the output of `buildQuoteJson(quote)` to a history list for each historical quote object.

```
public ArrayList getMonthlyHistory(){
    ArrayList history = new ArrayList<>();
    MonthlyHistoryVisitor monthlyHistoryVisitor = new MonthlyHistoryVisitor();
    monthlyHistoryVisitor.visit(this).stream().forEachOrdered(
            quote->history.add(buildQuoteJson(quote))
    );
    return history;
}
```

Example 5: Getting the Total Value of a Watch List

To get the total value of a given watch list (where a watch list is a collection of stocks that you are interested in but perhaps not invested in yet), we stream the stocks in the watchlist and use a method reference to the getCurrentPrice method of Stock, then use reduce to get the sum of the doubles.

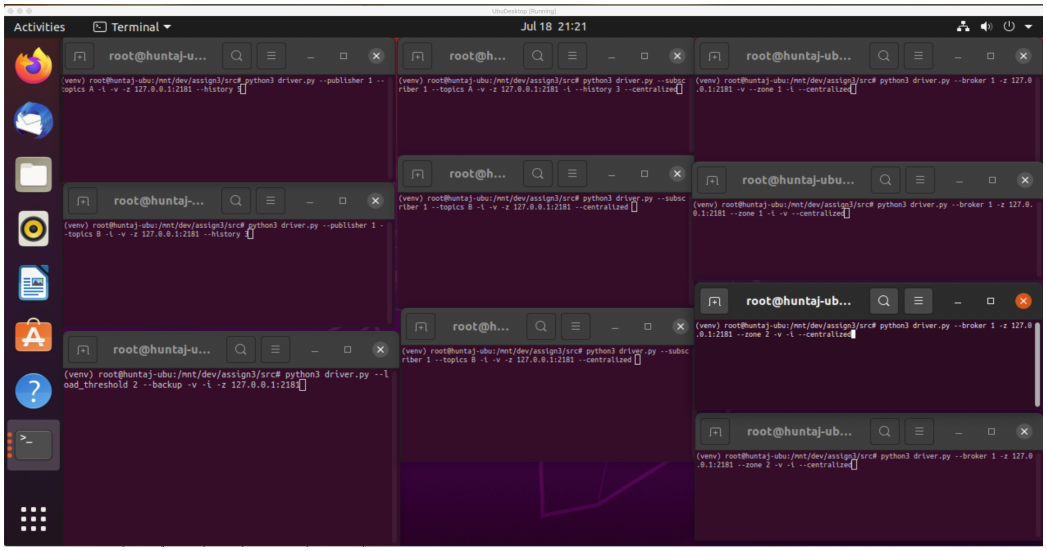
```
public double getTotalValue(){
    return this.stocks.stream()
            .map(Stock::getCurrentPrice)
            .reduce(0.0, Double::sum);
}
```

Example 6: Listing All Stocks You're Invested in Across All Accounts

Here's a fun one. After you have built a series of brokerage accounts and have invested in stocks across different portfolios, the app does the following in order to collect a list of all of the stocks you've invested in across your accounts.

```
List allStocks = new ArrayList<>();
dashboardService.getBrokerageAccounts().forEach(ba ->
            ba.getPortfolio()
                .getPositions()
                .forEach(p ->
                    allStocks.add(p.getStock())));
```

Application of CS Concepts in Solving an Advanced Problem



A Python Framework for Load-Balanced, Fault-Tolerant Publish/Subscribe Distributed Systems Built With ZeroMQ, an asynchronous messaging library and Apache ZooKeeper, a distributed coordination service

This project, built in collaboration with Guoliang Ding for CS-6381 (Distributed Systems) is the 3rd iteration of a Python framework for creating distributed publish/subscribe systems on virtualized networks. It builds on this project, the second iteration, which built on this project, the first iteration. It offers a framework for spinning up a publish/subscribe system either on a single host or on a virtualized network with a tool like Mininet. It offers two main models of message dissemination, namely **centralized dissemination** in which a message broker forwards all messages from publisher to subscriber and decouples/anonymizes their communication, and **decentralized dissemination** in which a publisher and subscriber speak directly with each other after a broker matches them with each other.

It offers integrated performance / latency analysis by allowing you to configure subscribers to write out latency data (between publishers and subscribers) to a provided filename, which provide insight about how long it takes for messages with specific topics from specific publishers to reach the subscriber (this is done by including the publish time in the message that gets sent).

Problem Statement

The original problem statement merely required that we define the base logic for creating publishers, subscribers, and brokers with some quality of service guarantees using ZooKeeper. However, as mentioned earlier in this portfolio, I love automating things, and building a framework to dynamically spin up these entities in a parameterized way made the most sense to me.

Architecture Overview

The project is structured as a library of classes including:

ZooKeeperClient. This class is a superclass inherited by the others providing basic ZooKeeper client functionality, like clearing znodes, listening for state changes, opening or closing a

connection with a ZooKeeper server, starting or stopping a ZooKeeper session, and creating, modifying, or getting the value of a znode.

BackupPool. This is the first object that should be created when spinning up a publish-subscribe system. This class represents a backup pool of dormant brokers that automatically promote themselves to leaders of new zones when the system's load threshold (defined with a `--load_threshold` parameter to the system) is exceeded by current system load read from a ZooKeeper znode (which is shared by all brokers for current load status updates via a watch mechanism). Backup pool brokers are not initially assigned a zone.

Broker. This class is used to create a message broker to serve as middleware between publishers and subscribers (with optional anonymization depending on whether the system is configured for centralized or decentralized message dissemination, represented by a boolean variable `centralized`).

Publisher. This class represents a single publisher in a Publish/Subscribe distributed system. A publisher does not need to know who is consuming the information; it simply publishes information independently of the consumer. If the publisher has no connected subscribers, it will drop all messsages it produces; note that a "subscriber" can be a Subscriber instance or a Broker instance.

Subscriber. This class represents a single subscriber in a Publish/Subscribe distributed system. A subscriber is indifferent to who is disseminating the information, as long as it knows their address(es). Subscriber can subscribe to specific topics and will listen for relevant updates about those topics across all publisher connections. If there are many publishers with relevant updates for a given topic T, the subscriber will only receive messages produced by the publisher who owns that topic (i.e. the one who started publishing about T first).

These classes are instantiated by the framework's driver, which is the main gateway for the framework. A user who wants to spin up a publish/subscribe system executes the driver while passing it arguments that define things like which entity to create (from the above options), how many of that entity to create, the address of the host running the ZooKeeper service, the file path to which consumed messages should be written (if spinning up a subscriber), the value to use for the system load threshold (beyond which new brokers are provisioned automatically), the zone number (to which the entity being spun up should be assigned), the topics to either publish or subscribe to (if spinning up a publisher or subscriber), and more.

Example Code Snippets - Provisioning Subscribers, Publishers, Brokers, & BackupPools

The below code snippet is executed by the driver if you pass the "`--subscriber N`" argument, where N is the number of subscribers you want to generate.

```
subscribers = create_subscribers(
    # how many to create
    count=args.subscriber,
    # optionally write consumed messages to a file
    filename=args.filename if args.filename else None,
    # address of the broker (create broker first); (default=127.0.0.1)
```

```
    broker_address=args.broker_address,
    # centralized or decentralized dissemination
    centralized=args.centralized,
    # topics to subscribe to
    topics=args.topics,
    # whether to run indefinitely
    indefinite=args.indefinite,
    # how many "rounds" if not indefinite, (default=15)
    max_event_count=args.max_event_count if args.max_event_count else 15,
    # ip address of host running ZooKeeper, structured as list
    ZooKeeper_hosts=args.ZooKeeper_hosts,
    # whether to log verbosely
    verbose=args.verbose,
    # the sliding window length to request from publishers
    requested=args.history
)
```

Similarly, the below code snippet is executed by the driver if you pass the "`--publisher N`" argument, where N is the number of publishers you want to generate.

```
publishers = create_publishers(
    # how many to create
    count=args.publisher,
    # address of message broker (create broker first); (default=127.0.0.1)
    broker_address=args.broker_address,
    # topics to publish
    topics=args.topics,
    # how long to sleep before each publish
    sleep_period=args.sleep if args.sleep else 1,
    # port on which to publish messages (used by subscriber or broker)
    bind_port=args.bind_port if args.bind_port else 5556,
    # whether to publish indefinitely
    indefinite=args.indefinite,
    # how many rounds if not indefinite, (default=15)
    max_event_count=args.max_event_count if args.max_event_count else 15,
    # ip address of host running zookeepr, structured as list
    ZooKeeper_hosts=args.ZooKeeper_hosts,
    # whether to log verbosely
    verbose=args.verbose,
    # the sliding window length offered to subscribers.
    offered=args.history,
)
### NOTE: publisher is indifferent to dissemination strategy
### (centralized boolean is not passed as argument)
```

The following code snippet gets executed by the driver if you pass the "`--broker 1`" argument. You can only create one broker at a time with the driver so you must use `1` as the argument.

```
create_brokers(
    # whether centralized or decentralized dissemination
    centralized=args.centralized,
    # on which port to receive publisher registration (default=5555)
    pub_reg_port=args.pub_reg_port,
    # on which port to receive subscriber registration (default=5556)
    sub_reg_port=args.sub_reg_port,
    # whether to run indefinitely
    indefinite=args.indefinite,
    # if not indefinite, how many messages to broker (default=15)
    max_event_count=args.max_event_count if args.max_event_count else 15,
    # number of seconds after which to autokill broker
    # (for testing leader election with multiple brokers / fault tolerance)
    # default = None
    autokill=autokill,
    # ip address of host running ZooKeeper, structured as list
    ZooKeeper_hosts=args.ZooKeeper_hosts,
    # whether to log verbosely
    verbose=args.verbose,
    # whether to use broker as primary replica. If false,
    # runs in background as backup, promotable by load balancer to primary.
    primary=args.primary,
    # zone to which broker gets assigned (for fault tolerance in that
    # zone)
    zone=args.zone
)
```

Lastly, the driver executes the following code snippet if you pass the argument "`--backup`". This will create a BackupPool process that auto-creates new zones for load balancing when system load meets the `--load_threshold` value (the default load threshold is 3, where the formula for current system load is `(publishers + subscribers) / (zones)`).

```
backup_pool = BackupPool(
    # ip address of host running ZooKeeper, formatted as list
    ZooKeeper_hosts=args.ZooKeeper_hosts,
    # whether system is using centralized or decentralized dissemination
    centralized=args.centralized,
    # whether to run indefinitely
    indefinite=args.indefinite,
    # if not indefinite, max number of rounds to listen for state change of
    # primary
    max_event_count=args.max_event_count if args.max_event_count else 15,
    # whether to log verbosely
    verbose=args.verbose,
    # the load threshold for the system
    threshold=args.load_threshold,
)
```

```
backup_pool.watch_system_load() # implementation below
backup_pool.wait_for_trigger() # implementation below
```

The following methods (used in the above snippet) are snippets pulled from the BackupPool class. In short, the BackupPool watches the system load stored in a shared state znode, and auto-provisions new zones (with new brokers) if that system load ever goes beyond the load threshold for the system.

```
def wait_for_trigger(self):
        while True:
            time.sleep(0.5)
....
def watch_system_load(self):
@self.zk.DataWatch(self.current_load_znode)
def dump_data_change (data, stat, event):
    if not event:
        pass
    elif event.type == 'CHANGED':
        # Load has been updated by a broker
        self.debug('System load has changed!')
        if self.need_more_capacity():
            self.debug('Need more capacity!')
            self.spin_up_new_broker() # implementation below
....

def spin_up_new_broker(self):
    self.debug('Spinning up a new broker!')
    new_zone = self.get_new_zone_number()
    broker = Broker(
        centralized=self.centralized,
        indefinite=self.indefinite,
        max_event_count=self.max_event_count,
        ZooKeeper_hosts=self.ZooKeeper_hosts_arg,
        verbose=self.verbose,
        zone=new_zone
    )
    # Create a new zone managed by this new primary broker
    broker.connect_zk()
    broker.start_session()
    broker.setup_fault_tolerance_znode()
    broker.setup_shared_state_znode()
    broker.setup_load_balancing_znode()
    broker.watch_shared_state_publishers()
    broker.watch_shared_state_subscribers()
    self.debug(f'Running election for new broker {id(broker)}')
    broker.zk_run_election()
```

Example Publish Subscribe System

You can find a visual demonstration of an example publish subscribe system created with this framework on my YouTube channel here. To give a written overview, these are the steps taken (for creating a publish subscribe system that uses the centralized dissemination strategy for anonymization of publishers and subscribers via message brokers):

1. Ensure a ZooKeeper service is running on some host that is accessible from the current environment. In the case of this example, ZooKeeper is running on localhost port 2181 (127.0.0.1:2181).

2. Create a backup pool process first that will auto-provision new broker zones when a load threshold is met (in this case, 2 clients per zone where client is a publisher or subscriber). This will create a ZooKeeper znode `/shared_state/current_load` with value 0 since no publishers or subscribers are running yet.

```
python3 driver.py --load_threshold 2 --backup --verbose \
    --indefinite --ZooKeeper_hosts 127.0.0.1:2181
```

3. Create broker 1 that will use the centralized dissemination strategy and assign it to zone 2. This broker will immediately become the primary, or leader, broker for zone 2 since that zone doesn't exist yet.

```
python3 driver.py --broker 1 --ZooKeeper_hosts 127.0.0.1:2181 \
    --zone 2 --verbose --indefinite --centralized
```

   This broker, once created, sets a new value on the `/shared_state/current_load` znode (still 0.0 since there are no publishers or subscribers yet). When that happens, the BackupPool detects the change but does not do anything since system load is still below threshold 2. In fact, we've added capacity, not load, by adding a zone. Capacity of the system is represented by the zones (denominator); load is represented by publishers or subscribers. Broker 1 now begins an event loop (listening for new registrations or messages).

4. Create a subscriber to topic A (Subscriber 1) configured for centralized dissemination. Use `--history 3` to configure a requested history, or sliding window size, of 3 messages. Publishers whose offered value is smaller will not satisfy this request.

```
python3 driver.py --subscriber 1 --topics A --history 1 --ZooKeeper_hosts 127.0.0.1:2181 \
    --verbose --indefinite --centralized
```

   Once created, Subscriber 1 is randomly assigned to an existing zone. Zone 2 is the only zone currently. The subscriber registers with the primary broker for that zone. It sends a registration message basically identifying itself (its address, its id, the topics to which it subscribes, and its

requested history size). When Broker 1 registers Subscriber 1, it updates the `/shared_state/current_load` znode which notifies the BackupPool. We now have one client and one zone, so the load is 1.0. Broker 1 also added a znode with Subscriber 1's identifying info as a child of the shared state `/shared_state/subscribers` znode, which all brokers watch along with a similar `/shared_state/publishers` znode to keep matchmaking data synchronized.

5. Now we begin to achieve fault tolerance. Create Broker 2, and assign it to zone 2 with the centralized dissemination strategy.

```
python3 driver.py --broker 1 --ZooKeeper_hosts 127.0.0.1:2181 \
    --zone 2 --verbose --indefinite --centralized
```

   The broker enters an election for zone 2 to become a leader of that zone when the current primary dies. That primary is running indefinitely without autokill turned on so we would need to CTRL+C to kill it in order to test the re-election.

   As mentioned previously, all brokers watch a collection of shared state ZooKeeper znodes to keep their matchmaking data synchronized; when this new broker (as when any new broker) gets created, it acknowledges and saves information about the existing clients; that is, it stores the identifying information about publishers and subscribers in local dictionaries to help with matchmaking newly registered clients with pre-existing ones. Also, current load is still 1.0 since we only have 1 client and 1 zone (zone 2 has two brokers).

6. Create a subscriber to topic B (Subscriber 2) configured for centralized dissemination.

```
python3 driver.py --subscriber 1 --topics B --indefinite --verbose \
    --ZooKeeper_hosts 127.0.0.1:2181 --centralized
```

   Once created, Subscriber 2 is randomly assigned to an existing zone. Zone 2 is the only zone currently. The subscriber registers with the primary broker for that zone (Broker 1). It sends a registration message identifying itself (its address, its id, the topics to which it subscribes, and its requested history size). When Broker 1 registers Subscriber 2, it updates the `/shared_state/current_load` znode which notifies the BackupPool. We now have two clients and one zone, so the load is 2.0. Broker 1 also added a znode with Subscriber 2's identifying info as a child of the shared state `/shared_state/subscribers` znode, which Broker 2 acknowledges and saves locally to keep its own matchmaking data synchronized. Note that neither subscriber is actually consuming anything since no one is publishing messages yet.

7. Now create a new centralized dissemination broker (Broker 3) and assign it to zone 1. This will decrease system load back to 1.0 (2 clients / 2 zones = 1.0).

```
python3 driver.py --broker 1 --ZooKeeper_hosts 127.0.0.1:2181 \
    --verbose --zone 1 --indefinite --centralized
```

Since zone 1 did not exist before this broker, Broker 3 is now the primary for zone 1. Broker 3 has updated the value of the `/shared_state/current_load` znode to 1.0.

8. Now create a publisher of topic A. This will be Publisher 1.

```
python3 driver.py --publisher 1 --topics A --indefinite \
    --verbose --ZooKeeper_hosts 127.0.0.1:2181
```

The publisher gets randomly assigned to Zone 1 (of the two Zones), and thus registers with the primary of Zone 1 (Broker 3), passing its identifying information during registration. Broker 3 subscribes to Publisher 1 and updates the `/shared_state/publishers` znode with the information of this new publisher, and the other brokers (1 and 2) in the system (who are watching that znode) update their local information to include that publisher's information. The system load is now 1.5 (2 subs + 1 pub / 2 zones = 3 / 2 = 1.5).

The publisher begins publishing messages for topic A, but nothing else happens in the system. This is because the one subscriber to topic A in the system is requesting a history (sliding window size) of 3, but the publisher is only offering a size of 1 by default since no argument was passed for that setting. Thus, it cannot support the request of the topic A subscriber. In other words, this publisher's messages are currently not going any farther than Broker 3 in Zone 1 (they're being ignored). To address this, we can kill Publisher 1 with CTRL+C, and restart it with a history argument value of 5. When it's killed, it unregisters from the system and the shared state system load value drops back to 1.0.

```
python3 driver.py --publisher 1 --topics A --indefinite \
    --verbose --ZooKeeper_hosts 127.0.0.1:2181 --history 5
```

Now, Publisher 1 re-registers (once again with Broker 3 who is primary for Zone 1), and the shared state system load value re-increases back to 1.5. Broker 3 also realizes that the topic A which Publisher 1 is publishing has a subscriber: Subscriber 1. So, it starts to relay the messages it is consuming from Publisher 1 over to Subscriber 1. The subscriber sees the publisher as the broker, because the broker as the middleware is anonymizing the publishing and subscription.

9. Now create a new publisher (Publisher 2) for topic B with an offered history size of 3.

```
python3 driver.py --publisher 1 --topics B --indefinite \
        --verbose --ZooKeeper_hosts 127.0.0.1:2181 --history 3
```

Publisher 2 gets randomly assigned to Zone 2 whose primary is still Broker 1. Publisher 2 registers with Broker 1 as normal, and Broker 1 updates the `/shared_state/publishers` znode with Publisher 2's information (which is acknowledged and saved by the other Brokers in the system). Broker 1 also updates the system load znode to 2.0 (2 pubs + 2 subs / 2 zones = 4 / 2 = 2.0). We see now that Subscriber 2, subscribed to topic B, begins consuming Publisher 2's topic B messages because Subscriber 2 is requesting only a history size of 1 (default), and Broker 1 has "made their match" based on their topic and offer/request relationship.

10. Now let's test the topic ownership feature. Kill off Publisher 2. The publisher will unregister and the system load will drop to 1.5 again. Now re-create it but change its topic to A instead of B such that we have two publishers publishing messages of topic A.

```
python3 driver.py --publisher 1 --topics A --indefinite \
    --verbose --ZooKeeper_hosts 127.0.0.1:2181 --history 3
```

Publisher 2 will once again register; this time it registers with Broker 3 who is primary of Zone 1. Broker 3 subscribes to Publisher 2 and updates the shared state values (load is once again 2.0) which is acknowledged and saved by the other brokers. However, Publisher 2 knows by reading a ZooKeeper znode that another publisher already owns topic A (it's locked), and thus it does not publish any messages about the topic. Instead, it logs "I do not have priority for A"; it will do this until that lock is released or it is killed. Knowing this, let's now kill and re-register Publisher 2 as a publisher of topic B again.

```
python3 driver.py --publisher 1 --topics B --indefinite \
    --verbose --ZooKeeper_hosts 127.0.0.1:2181 --history 3
```

It registers with Broker 3, primary of Zone 1, and system load comes back to 2.0 (2 subs, 2 pubs, 2 zones).

11. Now let's test automatic provisioning of new zones and brokers triggered by system overload. Adding one additional client (a pub or sub) should push the system over the threshold of 2.

```
python3 driver.py --subscriber 1 --topics B --indefinite \
        --verbose --ZooKeeper_hosts 127.0.0.1:2181 --centralized
```

A new subscriber (Subscriber 3) is now spun up. It registers with Broker 1, primary of Zone 2. Broker 1 updates the shared state with the Subscriber 3's information, and updates the shared system load state value to 2.5 (3 subs + 2 pubs / 2 zones = 2.5). The BackupPool reads this change, compares it to the system load threshold (it's higher!), and then automatically provisions a new zone (Zone 3) with a new primary broker (Broker 4); this reduces the system load back down to 1.667 (3 subs + 2 pubs / 3 zones = 5 / 3 = 1.667), which it saves to the `/shared_state/current_load`. The logs from the new broker are printed to standard output in the same window as the BackupPool since that is the process that spawned the new broker. Now, any new client (pub or sub) will have one additional possible zone to be randomly assigned to for broker communication (with options of zone 1, zone 2, and zone 3). Note that when this BackupPool auto-provisioned the new broker (Broker 4), it read all of the existing brokers' shared state, pulled in all of their collective matchmaking data, and handed it to the new broker to ensure it was synchronized as a new member of the system.

How does this project extend on iteration 2?

The previous project (iteration 2) extended the first by adding in Apache ZooKeeper for distributed coordination. Specifically, it used **kazoo**, a Python library for ZooKeeper, to handle **multi-broker** pub/sub with **warm passive** replication between brokers. The ZooKeeper usage was completely transparent, meaning if you used one broker, the project functions exactly the same as the first (which did not use ZooKeeper and was only functional with one broker). If you use multiple brokers in iteration 2, and one broker fails, ZooKeeper enables all publishers and subscribers to continue functioning as though nothing happened by simply electing the next available broker as the leader.

This project focuses on adding Quality of Service properties to the framework, and extends on iteration 2 in **three primary ways**:

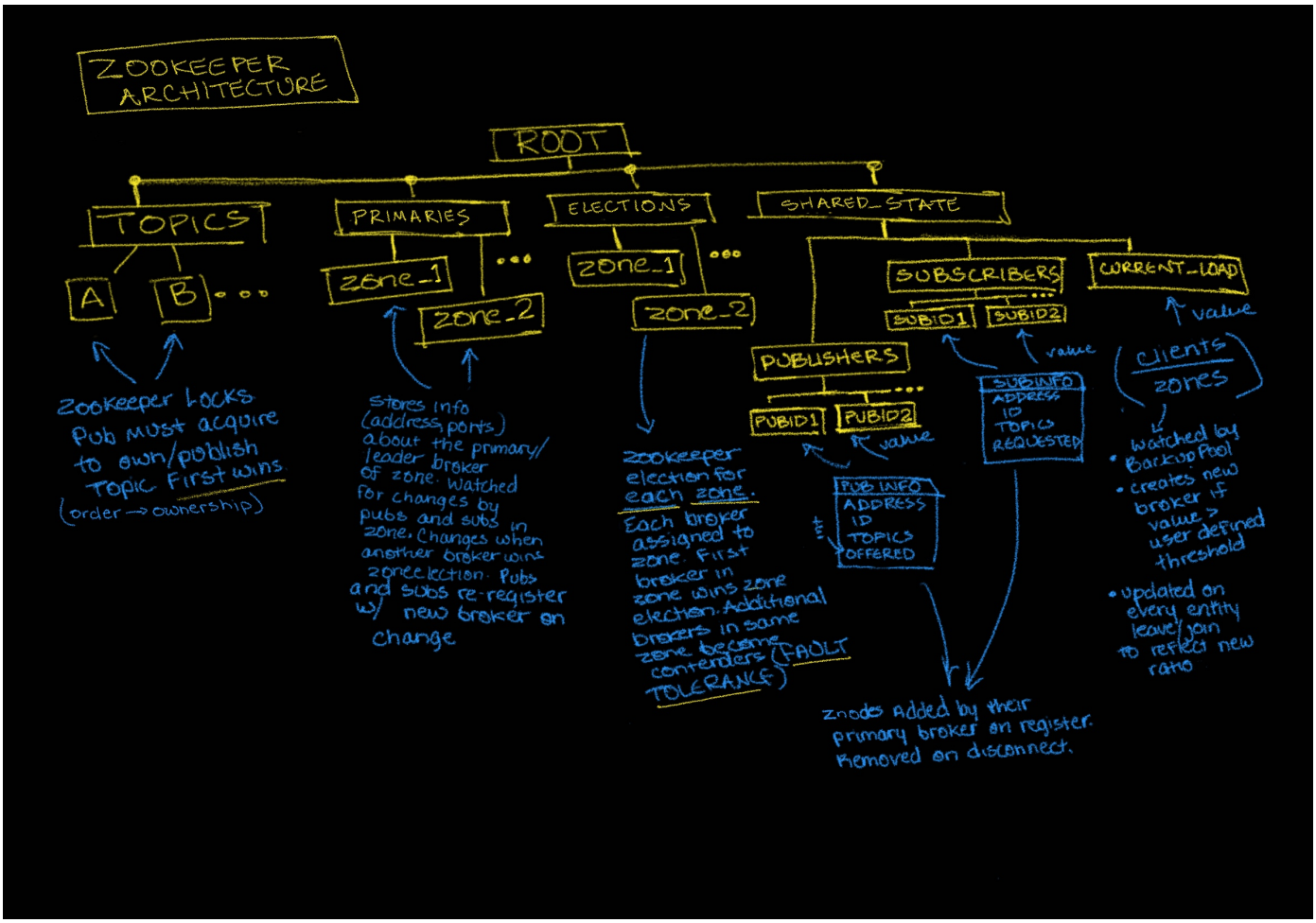
1. **Load balancing** in addition to fault tolerance. In the second iteration, we used ZooKeeper for fault tolerance using the primary-backup scheme without caring about the load on the primary broker. Now, we introduce load balancing such that there can be multiple primary broker replicas that each handle requests from clients (a client is a publisher or a subscriber). Each message broker that gets created gets assigned to a **zone**. The first broker assigned to a zone becomes its primary broker (or "leader"), then any additional brokers assigned to that zone become "contenders" for leadership in that zone, so each zone has one primary leader broker and a set of backup brokers that step in if the current primary fails (thus, each zone has its own **fault tolerance**). Load is balanced across zones by **randomly** assigning clients (pubs and subs) to one of the available zones at the time of creation. Ex: if there are 3 zones, each with a primary broker as its leader, and I create a publisher, that publisher gets randomly assigned to one of those 3 zones. A **BackupPool** process keeps track of the current system load (defined as **(number of publishers + number of subscribers) / (number of zones)**), AKA **clients per zone** as it relates to a user-defined threshold. When the load exceeds the user-defined threshold, a new broker gets created and added to a brand new zone for more distributed load balancing, and new clients include the new zone in the random zone assignment decision.

2. **Ownership strength.** In the first iterations, if we had more than one publisher publishing the same topic T, then all the events from all of those publishers would get relayed to the subscribers subscribed to topic T. Now, we introduce a measure of topic ownership for publishers, so that only messages about topic T from the publisher with the highest ownership strength over that topic T are relayed to the interested subscribers. The first publisher to publish about a topic Becomes the owner for that topic By acquiring an Apache

ZooKeeper lock representing that topic. Any other publisher that tries to publish about that topic will not be able to until that lock is released, which happens when the first publisher process ends, intentionally or not.

3. **History.** In the first iterations, when a publisher of topic T and a subscriber to topic T registered with a broker, that publisher and that subscriber were matched based on that topic T, and the subscriber would receive all messages about topic T, one at a time, as the publisher published those messages. Now, we introduce a new requirement that the last N messages published about a topic T be preserved in a sliding window fashion, and a subscriber to T can only receive messages from a publisher of T if the subscriber requests some number of messages less than or equal to the size of that sliding window offered by the publisher. In other words, the publisher's "offered" historical window of messages MUST be greater than or equal to the subscriber's "requested" number of historical messages. If a publisher maintains the last 10 samples of topic T while a subscriber wants the last 20 samples of topic T, then this subscriber cannot get the data even though the topic of interest is common, because the "offered vs requested" dominance relationship is not satisfied.

How is ZooKeeper used?



Leader Election

In iteration 2, there was only a single ZooKeeper broker election observed by the brokers to achieve fault tolerance. This used an `/electionpath` znode with ZooKeeper. The first broker to get created won the election immediately and became the leader, and any subsequent brokers would become contenders for the leadership by entering the `/electionpath`

election and waiting for the current leader to fail (those savages!). Now, with the load balancing model, we have multiple zones that **each have their own broker elections**. So instead of elections being managed with a single `/electionpath` znode, elections are now managed per-zone, with `/elections/zone_ZONENUMBER`. Within a given zone, the leader election happens the same way as before. When you create a broker, you assign it to a zone. If it's the first broker assigned to that zone, the zone-related znodes are initialized (including that election znode) and that broker becomes the zone's primary, or leader. If you assign a broker to a zone that already has a primary, it will enter the election for that zone and wait for the zone's primary to fail as a contender in the zone's leader election. The contenders are the backups, so this provides fault tolerance for each zone in the system, and thus for the system as a whole.

Shared State

We also now use ZooKeeper to maintain shared state across all of the brokers, particularly for the synchronization of matchmaking data. Why did we do this? Well, assume we hadn't. Now, imagine a publisher of topic A with an offered history of 5 gets created and is randomly assigned to zone 3, which means it registers with zone 3's primary broker. Then, a subscriber to topic A with a requested history of 2 gets created and is randomly assigned to zone 1. Regardless of centralized vs decentralized dissemination, without shared state across the brokers in the different zones, the subscriber would not know about the publisher even though it publishes its topic of interest and satisfies the offered vs requested dominance relationship. So, here's how the shared state works:

1. Each broker watches a `/shared_state/publishers` znode with `@ChildrenWatch` to monitor for additions or removals of publishers by any of the other brokers, regardless of zone. When the children change, the watch triggers and the broker updates its internal matchmaking data to reflect the global shared state.

2. Each broker watches a `shared_state/subscribers` znode with `@ChildrenWatch` to monitor for additions or removals of subscribers by any of the other brokers, regardless of zone. When the children change, the watch triggers and the broker updates its internal matchmaking data to reflect the global shared state.

3. Once a publisher gets randomly assigned to a zone on creation, it registers with that zone's primary broker, at which point the broker adds a znode named with the publisher's ID as a child of the `/shared_state/publishers` znode, with JSON containing the publisher's ID (str), Topics (list), Offered (int), and Address (str)

4. Once a subscriber gets randomly assigned to a zone on creation, it registers with that zone's primary broker, at which point the broker adds a znode named with the subscriber's ID as a child of the `/shared_state/subscribers` znode, with JSON containing the subscriber's ID (str), Topics (list), Offered (int), and Address (str)

5. When a publisher disconnects, it tells the broker before disconnecting, and the broker removes its respective znode from the children of the `/shared_state/publishers` znode

6. When a subscriber disconnects, it tells the broker before disconnecting, and the broker removes its respective znode from the children of the `/shared_state/subscribers` znode

This allows the brokers in the various load-balanced zones to know about a global state across all of the zones so that cross-zone matchmaking between publishers and subscribers can still work.

Load Monitoring and Auto-Broker Provisioning

In order to achieve dynamic scaling according to system load, we use another ZooKeeper znode called `/shared_state/current_load`. Here's how it is used:

1. We have a BackupPool process that watches this znode for data changes with @DataWatch. When it changes, if the value of the znode (the current system load) is greater than a user defined threshold passed to the the Backup Pool process, then provision, configure, and start a new broker as **the primary of a brand new zone**.

2. Whenever a new publisher or a subscriber registers OR disconnects with any broker in any zone, update `/shared_state/current_load` to reflect the current value of the following formula: (num_publishers + num_subscribers) / (num_zones). Generally speaking, registration means load increase, disconnection means load decrease.

Watch Event

In iteration 2, each publisher and subscriber set a watch on the znode `/broker` which would store the information about the current leader, or primary, broker, which most recently won the `/electionpath` election. Now, as we have made the elections zone-specific with `/elections/zone_ZONENUMBER` znodes, we have also made this primary broker info storage zone specific, since each zone has its own primary, or leader. So instead of writing its information to /broker when a broker becomes a zone leader, it writes its information to `/primaries/zone_ZONENUMBER` when it becomes a zone leader. In return, the publishers and subscribers, after getting randomly assigned to a zone on creation, begin watching their respective `/primaries/zone_ZONENUMBER` to obtain the most updated information about their current broker leader. If it changes, that means the previous primary/leader broker has died or has been manually terminated and they register with the next broker contender who wins the zone's election.

Communication Skills in Computer Science

A Multi-Phase Tail Latency Analysis of Couchbase Server Using Automation and the Yahoo! Cloud Serving Benchmark

Austin Hunt

Department of Computer Science

Vanderbilt University

Nashville, TN, USA

austin.j.hunt@vanderbilt.edu

Guoliang Ding

Department of Computer Science

Vanderbilt University

Nashville, TN, USA

guoliang.ding@vanderbilt.edu

The following paper was authored in collaboration with Guoliang Ding for Dr. Aniruddha Gokhale's CS-5287 Principles of Cloud Computing course. It extends on a separate project — also focused on Couchbase evaluation — that we worked on for his CS-6381 Distributed Systems course. It provides a detailed analysis of:

- The architecture of Couchbase, a distributed NoSQL cloud database,
- The architecture of our automated testing framework (written in Python) for evaluating Couchbase as a distributed datastore, and
- The test results obtained from executing that framework against a live Couchbase cluster, with focus on relationships between operation latencies (for read, insert, update, and delete) and parameters like durability configuration, cluster size, bucket size, and operation type.

The underlying software project on which the paper is based uses tools like Ansible, Vagrant, AWS EC2, the Couchbase CLI, the Yahoo! Cloud Serving Benchmark (YCSB) project, the Python 3 Couchbase SDK, and of course Couchbase to automatically and iteratively provision, evaluate, and deprovision clusters of various configurations while collecting data that highlights how configuration changes affect cluster performance.

Variables of Interest

We test and analyze Couchbase along a variety of metrics in an auto-provisioned cloud infrastructure environment using AWS EC2; we analyze metrics like:

- Cluster size impact on latency of read, write, insert, and delete operations, specifically with a homogeneous architecture (same services on every node) to understand how scaling cluster size affects cluster performance
- Multidimensional service scaling impact on read, write, and delete operations, to understand how scaling out a service X by one additional node affects cluster performance. Couchbase

offers multi-dimensional scaling of services, meaning that you can scale specific services independently based on what your application needs are. For example, if you are dealing with massive amounts of data, you may want to independently scale out your data service more so than the other services. These are the main services within a Couchbase cluster.

- Using Yahoo! Cloud Serving Benchmark (YCSB), how does database size (for a given cluster size N) affect tail latency of different operation types (read, update, and insert)? We vary database size via variations in 1) document count, 2) document field count (fields per document), and 3) document field size (in bytes).
- Using Yahoo! Cloud Serving Benchmark (YCSB), how does a change in request distribution impact tail latency of different operation types (read, update, and insert)?
- Using Yahoo! Cloud Serving Benchmark (YCSB), how does the ratio between read, update, and insert requests affect overall tail latency of database operations?

Automation

The framework is architected such that all it needs is an AWS Access Key ID and AWS Secret Access Key set as environment variables. It can be executed with Vagrant using the command `vagrant up --provision`. The provisioner for the Vagrant box is an Ansible Playbook that handles the automatic provisioning and configuration of a set of Ubuntu 20.04 AWS EC2 instances and then kicks off the test framework via a central, parameterized driver file. The framework itself then uses the provisioned EC2 instances to automatically and iteratively create and tear down various Couchbase cluster architectures via custom Python classes DataManager and ClusterManager. Python is used for not only the data management (via DataManager) via the provided Python 3 SDK, but also the management of the cluster architecture (via ClusterManager) in order to eliminate the need for interacting manually with the Couchbase Web GUI. Note: since much of the cluster architecture management could not be done using the default Python 3 SDK, we created methods that "wrapped" the Couchbase CLI (which could manage cluster architecture) to make this a **fully Pythonic project.**

The framework writes out latency data during test execution, which is used afterward to generate various plots that reveal relationships between the various variables we tune (e.g. cluster size vs. operation latency and bucket size vs. operation latency). If you want plots when the framework is done running tests, just pass -plt to the driver.

Contributions

Guoliang and I divided the work for the paper up such that he covered the analysis of Couchbase's architecture, and I covered the analysis of the automation framework, including the results from its execution. On the software side, I worked on building the Python automation framework and setting it up to link with Ansible, Vagrant, and AWS EC2, and Guoliang worked with the integration of the Yahoo! Cloud Serving Benchmark data workloads into that framework.

A Multi-Phase Tail Latency Analysis of Couchbase Server Using Automation and the Yahoo! Cloud Serving Benchmark

Austin Hunt
Department of Computer Science
Vanderbilt University
Nashville, TN, USA
austin.j.hunt@vanderbilt.edu

Guoliang Ding
Department of Computer Science
Vanderbilt University
Nashville, TN, USA
guoliang.ding@vanderbilt.edu

***Abstract*—As the production, storage, and analysis of data become increasingly integral to the success of modern businesses, and as data becomes increasingly kaleidoscopic due to the influx of new data sources, NoSQL (Not Only Structured Query Language) database systems are becoming a popular choice for data management because of their ability to quickly and scalably handle large volumes of structured, semi-structured, and unstructured data in a way that is friendly to developers, and more broadly, to change. One of the solutions in this arena is Couchbase Server, an open source database software package for building and managing distributed, document-oriented, multi-model NoSQL databases optimized for modern interactive applications. In this paper, we provide a detailed, two-phase tail latency analysis of Couchbase Server Community Edition v7.0 using a custom automated testing framework built with the Couchbase CLI and Couchbase Python 3 SDK in combination with the Yahoo! Cloud Serving Benchmark (YCSB), a robust and open source framework for analyzing the performance of different "key-value" and "cloud" serving stores like Couchbase. We analyze and report on relationships between tail latencies and dataset sizes, request distributions, ratios between various operation types (read/insert/update), and cluster architecture (homogeneous versus heterogeneous service layouts).**

***Index Terms*—Couchbase, NoSQL, YCSB, Tail Latencies, AWS, Python**

I. Introduction

Traditional relational database management systems (RDBMS), still lingering from their introduction in the 1970s era of mainframes and back-office business applications, are struggling to keep up with the requirements of the current digital era in which data grows continuously in both volume and complexity. Designed and engineered to run on a single server where the bigger, the better, an RDBMS is generally scaled vertically by adding more processors, memory, and storage, but both physics and nonlinear price increases often make vertical scaling impractical, sometimes even impossible. In addition, the strict consistency requirements of RDBMS make it difficult to scale them horizontally, and inflexible RDBMS data models, despite their provision of safety and consistency with data management, make changing the data model a difficult process, which is problematic in a world of continuous change. In the digital era, the ever-changing structure of data that needs to be stored challenges the idea of pre-planned, fixed schemas when building solutions that meet long term running requirements. Consequently, the huge amount of data generated from the internet, cell phones, social media, and the growing Internet of Things (IoT) demands a more modern, flexible, and scalable solution than is offered by traditional data stores.

A. A New Wave of Data Stores

Document database systems, which store information as documents in formats like JSON, XML, and BSON, have been developed to address the limitation of relational database systems and provide the flexibility, performance, and scalability required by the modern Internet and mobile applications we are now accustomed to using. From the mid-2000s to 2020, a steady rise in the adoption of document-oriented database technology by small startups, IT shops, and established Fortune 500 companies, is a consequence of its capacity to simplify data management by making it a flexible part of continuous application development [Mon21].

B. Couchbase

Couchbase, an open-source, document-oriented, multi-model, distributed NoSQL database, lives among the new wave of modern document database systems belonging to the NoSQL family. By offering strong consistency guarantees, various options for simple horizontal scaling, and impressively high performance with latencies on the order of microseconds, Couchbase currently stands as one of the most widely used NoSQL solutions and is used by companies like Wells Fargo, Tesco, eBay, and PayPal. These companies use Couchbase to guarantee high service availability while managing massively high-throughput processes like 1) real-time fraud monitoring for over 50 million daily transactions (Wells Fargo), 2) catalog scaling and inventory management for millions of products (Tesco), 3) processing millions of user analytics updates every minute (PayPal), and supporting about 1.3 billion live e-commerce listings worldwide at any given moment (eBay). [Cou21c].

C. The Yahoo! Cloud Serving Benchmark (YCSB)

This innovative tsunami of new database technologies carries with it the need for testing – more specifically, the need for a method of comparing those technologies in a standardized, "apples-to-apples" kind of way. After all, there is not much advantage to a new wave of technologies if there's no clear way of deciding which one to use for a given application. In 2010, the Yahoo! Cloud Serving Benchmark (YCSB) was introduced with the stated goal of meeting that need [Coo+10]. This project wraps standardized performance comparisons of many of these new generation cloud data serving systems into a simple framework that allows you to configure and execute specific workloads against specific target databases. The YCSB framework's extensible design supports easy definition of custom workloads and easy extension to new systems [Coo+10]. Currently, the framework supports targeted performance testing against nearly 50 data serving systems, including but not limited to MongoDB, AWS DynamoDB, Cassandra, Redis, CouchDB, Google BigTable, and most importantly for this project, Couchbase.

D. Why Tail Latency?

For this research, we focus much of our analysis and reporting on the relationship between tail latency, or *high-percentile latency*, and various configurations of Couchbase (to be described in more detail). Tail latencies, while rare by definition, stand to have a greater impact on business relationships with customers in realistic application environments than averages and can greatly influence the user experience, thus it is important to understand those tail latencies so they can be reduced or avoided.

II. Overview of Couchbase

In this section, we'd like to introduce some key concepts underlying Couchbase Server and provide a foundation for the performance evaluation in section IV.

A. Couchbase Server

Couchbase Server refers to the actual database software package. The package comes as two different versions: the paid Couchbase Server Enterprise version and the free Couchbase Server Community Edition available for download and evaluation from the Couchbase website. In this evaluation, Couchbase Server Community Edition 7.0.0 is used. Couchbase also provides a CLI and a Python 3 software development kit (SDK) which we make heavy use of for the automation of testing.

B. Node

A Couchbase node is a physical or virtual machine that hosts a single instance of Couchbase Server [Cou21b]. A node can only have one single instance of Couchbase Sever running on it, and Couchbase keeps node management simple by providing only a single node type, which greatly helps with the installation, configuration, management and troubleshooting of the cluster as a whole. In this evaluation, we will use AWS EC2 instances to host the Couchbase Server and serve as the nodes. More specifically, we use a set of five t2.xlarge instances, each of which come with 2 virtual CPUs, 4 gigabytes of RAM, 8 gigabytes of disk space, and low-to-moderate network performance.

C. Cluster

A cluster contains one or more nodes, which can be added or removed from a cluster on the fly without impacting the performance of the cluster. Much of this evaluation revolves around the automatic adjustment of cluster configuration to measure its relationship with latencies of data operations.

D. Bucket

A bucket can be roughly compared to a database in traditional RDBMS terms, but instead of housing a set of tables with columns and rows, a bucket simply houses a collection of documents, specifically in the JSON format. There are three types of buckets in Couchbase [Cou21b]:

- Standard, or "Couchbase", buckets, which are the default and store data both in memory and persistently.
- Ephemeral buckets, which are designed for usage by applications that do not require data persistence
- Memcached buckets, which are now deprecated – these were originally designed for use alongside other database platforms, even RDBMS platforms, to serve a caching function

In this evaluation, the Standard Couchbase buckets are used and the performance difference of bucket types is currently beyond the scope.

E. Services

A service in Couchbase is an isolated set of processes dedicated to a particular task or set of tasks [Cou21b]. Couchbase Server breaks its functionality into a set of seven core services, only 4 of which were relevant to and are actively used in this evaluation:

- The **data** service is used to store, set, and retrieve data items using their keys (the hashes of which map to specific vBuckets).
- The **query** service is the engine responsible for parsing N1QL queries, where N1QL is a SQL-like query language designed and optimized for Couchbase.
- The **index** service handles the creation of indexes for use by the query service; indexes can be created to improve query performance. This evaluation creates a default index for each bucket which is used when testing query operation latency.
- The **search** service is responsible for creating the indexes specifically for Full Text Search (FTS) and handles language-aware searching. The Full Text Search is one of the specific operation types whose performance is tested in this evaluation.

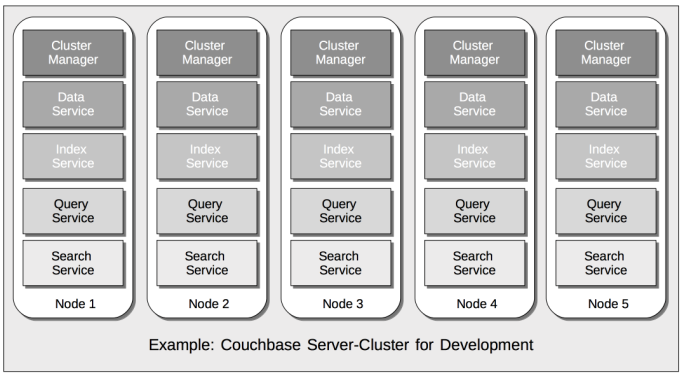
By design, these services can be deployed, maintained, and provisioned independently of one another (for the most part) using a framework that Couchbase documentation aliases

"multi-dimensional scaling". There are only a few exceptional requirements, namely 1) you must have the data service running on at least one node, and 2) certain services are interdependent and if running on a node, require that their counterpart services also be running on that same node (e.g., query and index). A node, depending on its underlying hardware, may be configured to have one or all services running on it; this can be decided upon by the development team for the client application, which allows for optimal use of resources based on the known requirements of the application. Here, it is important to note that we are using the Community Edition of Couchbase for our evaluation, and this edition comes with some tighter constraints on service layouts that are outlined in more detail in subsection IV-C.

F. Architecture and Scaling

Because of the logical separation of services and the ability to assign services to different hosts, a Couchbase cluster can be architected in various ways – realistically many ways, actually, considering the combinatorics behind multidimensional scaling possibilities as cluster size increases. For development purposes, the simplest cluster architecture is a homogeneous one in which all nodes are running the same service quotas, as seen in Figure 1.

Fig. 1: Development Cluster Homogeneous Service Architecture [The picture is from [Cou21b]]



Corresponding to this cluster architecture is the homogeneous scaling methodology in which horizontal scaling simply entails adding one or more additional nodes that run the exact same set of services already running on each of the existing nodes. Figure 2 illustrates this scaling methodology clearly.

A more advanced cluster architecture that is more appropriate to production environments is the heterogeneous service architecture where different nodes will have different services running such that each node carries unique responsibilities. For example, in certain cases, performance may be optimized by dedicating several high-resource nodes to more critical services like the data service or index service. Figure 3 represents a multidimensional service layout in a Couchbase cluster.

Fig. 2: Homogeneous Scaling [The picture is from [YN19]]

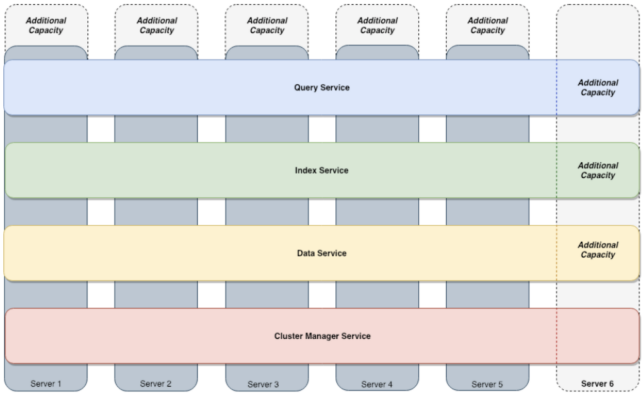
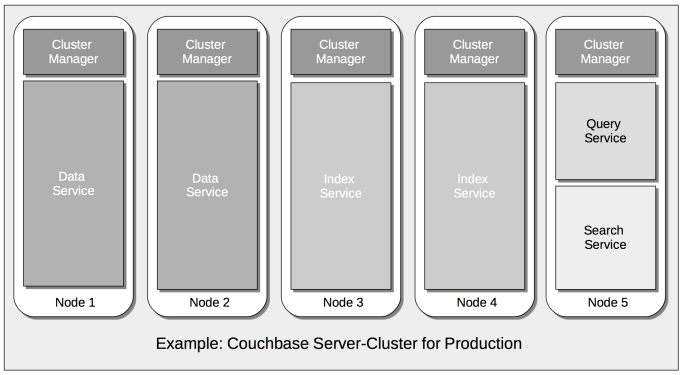


Fig. 3: Production Cluster Heterogeneous Service Architecture [The picture is from [Cou21b]]

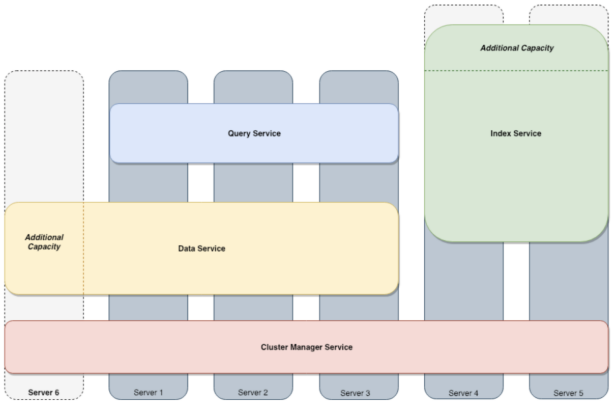


Corresponding to this architecture is the multi-dimensional scaling methodology, in which the addition of a new node to the cluster involves the intentional, calculated decision about specific services to be run on that new node for optimized application performance. This scaling approach is highlighted by Figure 4. Subsection IV-C is dedicated to an analysis of the impact of multidimensional service scaling on tail latencies of different operations.

G. CLI, SDK and Data Access

Couchbase provides a diverse arrangement of command-line interface (CLI) tools to handle a bulk of the management and monitoring of cluster infrastructure, from the cluster configuration itself down to the nodes and vBuckets. Couchbase also provides a multitude of language-specific software development kits (SDKs) that enable developers to easily integrate application code and logic with a Couchbase datastore. Both the CLI [Cou21a] and Python SDK [Cou21d] are leveraged in this evaluation to develop an automated testing framework. Details are provided in section III.

Fig. 4: Multi-Dimensional Scaling [The picture is from [YN19]]



III. The Automated Testing Framework

The evaluation of Couchbase Server Community Edition 7.0.0 that we present in this paper is driven entirely by an automated testing framework built with Python 3.8. The tests on which we report are run by the framework against a Couchbase Server cluster running on AWS EC2 infrastructure – specifically five Ubuntu 20.04 "t2.xlarge" instances with public addresses enabled. The framework uses the combined power of the Couchbase Server Python 3 software development kit (SDK) and the Couchbase command line interface (CLI) to automatically manage, respectively, both 1) data operations and 2) cluster configuration. These two types of management are handled, respectively, with a Data Manager class and a Cluster Manager class, each written with Python. The Couchbase Server Python 3 SDK (version 3.1.3) allows the framework to perform data operations rapidly using a high-performance C library called 'libcouchbase' which communicates to the cluster over Couch- base's binary protocols. The Couchbase Server command line interface (CLI) is provided to manage and monitor clusters, servers, vBuckets, XDCR (cross datacenter replication), and so on; while the CLI was not designed specifically for use with Python, we choose to use Python to trigger the shell CLI commands such that all data and configuration operations are wrapped and abstracted in the Python framework.

A. Execution Overview

While Couchbase Server does offer a genuinely easy-to-use web interface for managing data and cluster configuration, we wanted to eliminate the need for manual interaction in this performance evaluation. The framework was built such that the only prerequisite to running it is providing your AWS access key ID and secret access key in an environment file, and then "provisioning" (with one command) a Vagrant virtual machine with an Ansible[HM17] master playbook.

The following outlines the primary steps executed when the framework is triggered to run:

1) A local virtual machine is spun up with Vagrant, and is provisioned with an Ansible[HM17] master playbook
2) The master playbook automatically provisions five Ubuntu 20.04 "t2.xlarge" EC2 instances with public addresses enabled (in addition to private VPC addresses) and installs the Couchbase Server software on each of the instances.
3) After the EC2 instances are provisioned, the playbook kicks off a framework driver script
4) Depending on arguments passed to the driver, the driver uses some combination of the DataManager, the Cluster-Manager, and the YCSB CLI to automatically configure various clusters and run workloads against them (as it does this, it also writes latency data for each operation to an output file)
5) After the workloads are finished running, the driver uses the output data files to generate plots, many of which are included in this paper

This framework's modular architecture will enable us to extend it to other testing conditions with minimal effort, some of which are discussed in section V.

IV. Evaluation

In this evaluation, we are interested in evaluating the impact of database size (controlled by record counts, field sizes, field counts), operation type ratios (e.g., read/insert/update ratios), request distributions, and cluster architecture (homogeneous versus heterogeneous service layouts) on the tail latency of key data operations on a Couchbase cluster of five nodes. The testing is separated into the following two phases:

- In phase one, we use a fixed cluster size of five nodes with a homogeneous service architecture. With these variables holding as constants, we use YCSB to analyze the impact (on tail latency, specifically) of other variables that are easy to tune via YCSB such as record (document) counts, field sizes, field counts (document sizes), request distributions (explained in subsection IV-A2), and ratios between different operation types (i.e., read vs. write).
- In phase two, we again use a fixed cluster size of five nodes, but this time with focus on heterogeneous service architectures, with service layout and operation type acting as independent variables such that we can understand how the tail latency of each operation type changes as certain services are scaled horizontally (within the constraints of Couchbase Community Edition 7.0).

A. Phase One Test Plan: Using the YCSB Framework

As briefly mentioned in subsection I-C, the YCSB framework supports a wide range of workloads, which can either be defined ahead of time in static configuration files or at runtime by passing arguments to the YCSB executable, which provides a lot of flexibility through various parameters. In this project, the following parameters are picked to investigate their impact on the tail latency.

1) Database Size: The YCSB framework is generally separated into three steps on the client side [Coo+10]:

- The Load phase, where the data are loaded into the database system
- The Run phase, where workloads are issued to the database system
- The Report phase, where the performance statistics are printed to the screen or output to a designated location

During the load phase, the size of the documents and the number of documents can be controlled to determine the overall size of the database. By default, each document has 10 fields and each field is 100 bytes long, making each document 1KB. In this evaluation, we tune the values of the following sizing parameters to gauge their impact on the tail latency of different operations:

- Record count; this is the number of documents inserted into the database. We test values of 1K, 10K, and 100K for this parameter.
- Field count; this is the number of fields, or keys, in each document. We test values of 10 and 500 for this parameter.
- Field length; this is the size, in bytes, of each field in each document. We test values of 10 and 100 for this parameter.

Table I depicts the overall database sizes resulting from the various combinations of the above values. Note that for the YCSB tests, we use a constant memory quota of 256MB for our test bucket. This means that Couchbase will need to optimize performance by ejecting data from memory to disk based on its least recently used algorithm for four of the database sizes in the table (500MB, 100MB, 500MB, and 5GB) since they exceed the 256MB memory quota.

Database Sizes			
Record Count	Field Count	Field Length	Size
1000	10	10B	100KB
1000	10	100B	1MB
1000	500	10B	5MB
1000	500	100B	50MB
10000	10	10B	1MB
10000	10	100B	10MB
10000	500	10B	50MB
10000	500	100B	500MB
100000	10	10B	10MB
100000	10	100B	100MB
100000	500	10B	500MB
100000	500	100B	5GB

TABLE I: Database sizes as the products of record count, field count, and field length

2) Request Distribution: During the run phase, the YCSB framework randomly chooses from the documents that were previously loaded during the load phase to conduct read or write operations [Coo+10]. This decision is controlled by random distributions and YCSB has the following built-in distributions that will be tested:

- Uniform, where all records in the database are equally likely to be chosen.
- Zipfian, where some records are extremely like to be chosen (also known as the head of the distribution) and the probability rapidly decreases from these records (also known as the tail of the distribution).
- Hotspot, where some percentage of the data items are accessed by some percentage of the operations

3) Read/Write Proportions: During the run phase, the YCSB framework must decide which operation to perform when, as constrained by user-defined arguments for "proportions" of the different operations. In this test, we define nine different operation proportions and analyze them in conjunction with the other discussed variables in order to study how, or if, ratios between operation types impacts overall tail latency of a Couchbase system.

- Read-heavy
  - 100% read
  - 90% read, 10% insert
  - 90% read, 10% update
- Write heavy
  - 100% insert
  - 100% update
  - 10% read, 90% insert
  - 10% read, 90% update
- Equal read-write
  - 50% read, 50% insert
  - 50% read, 50% update

4) The Test Algorithm: The test automation is structured as a nested loop, where each level of the loop represents one of the parameters.

The following pseudocode outlines, at a high level, the testing approach for this phase.

Algorithm 1 YCSB Test Automation Driver

```
createHomogeneousCluster(nodeCount=5)
opProps = [list of r/u/i proportions]
requestDists = [uniform, zipfian, hotspot]
recordCounts = [1K, 10K, 100K]
fieldCounts = [10, 500]
fieldLengths = [10, 100]
for each rc ∈ recordCounts do
    for each fc ∈ fieldCounts do
        for each fl ∈ fieldLengths do
            for each rd ∈ requestDists do
                for each prop ∈ opProps do
                    bucket = createBucket(bucketSize)
                    ycsb(rc, fc, fl, rd, prop, bucket)
                end for
            end for
        end for
    end for
end for
generatePlots() // build plots with written data
```

B. Phase One Test Results

In this section we provide the results obtained from the execution of the testing framework in phase two, organized by the various relationships we were intending to analyze.

1) Field Count vs. Tail Latency: As seen in Figure 5, the number of fields in each document in the Couchbase database ultimately did not really impact the tail latency of the insert operation - the same is true for the update operation. The reason for this is that when testing with YCSB, we used a constant value of False for both the *persistTo* and *replicateTo* parameters, such that YCSB would not enforce any strict data durability requirements during the write operations. That is, since we were not requiring each write operation to replicate or persist data across the 5-node cluster, our writes were fast regardless of the number of fields. However, as seen in Figure 6, there does appear to be a positive correlation between field count and read latency. This indeed makes sense simply because larger documents imply greater usage of memory, which in turn implies greater likelihood of ejecting data to disk. Couchbase's impressive performance largely comes from its automatic management of a "caching layer", which comes down to keeping as much data in memory as possible with minimal disk interactions, where the memory quota can tuned at the bucket level; we tuned our test bucket with a memory quota of 256MB. We can see from Table I that the largest database size for a field count of 10 is 100MB, which is less than 50% of our bucket memory quota, which means there should have been no need for ejection. However, for a field count of 500, the largest overall database size is 5GB, which far exceeds our memory quota for the bucket, guaranteeing ejection of some data to disk. Thus, many of the read operations on documents with a field count of 500 had to go all the way to disk to get data that wasn't available in memory, resulting in higher tail latencies.

Fig. 5: Field count impact on insert tail latency in Couchbase

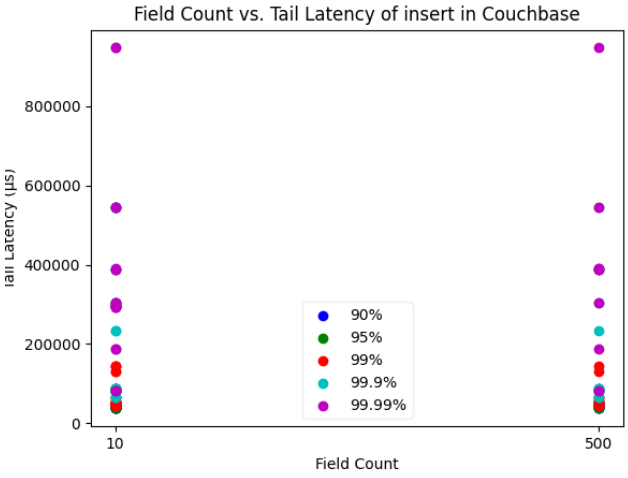
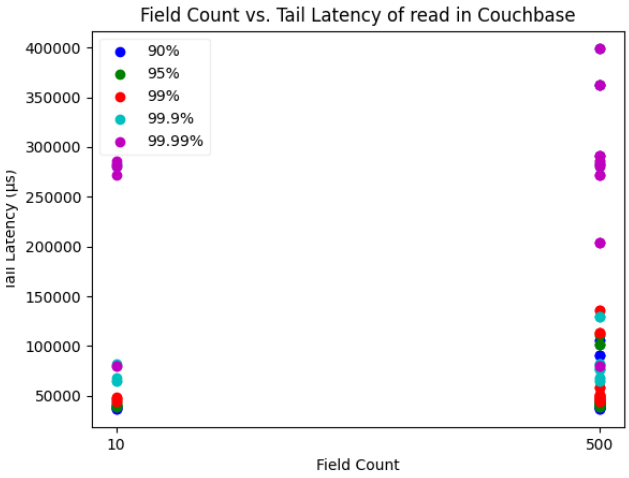
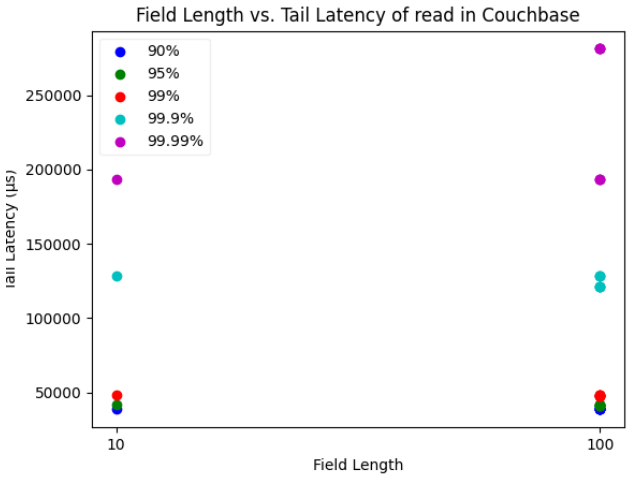


Fig. 6: Field count impact on read tail latency in Couchbase



2) Field Length vs. Tail Latency: As was the case in subsection IV-B1, field length did not impact the tail latencies of the write operations because without durability requirements of replication and persistence, the speed of write operations are largely indifferent to document size variations. As seen in Figure 7, the read operation tail latency was heavily impacted by the field length. Of course, field length is yet another way of changing the document size, and thus the overall database size, so it is guaranteed to have a positive correlation with the likelihood of ejecting data to disk due to memory quota consumption, which implies a positive correlation with read latency.

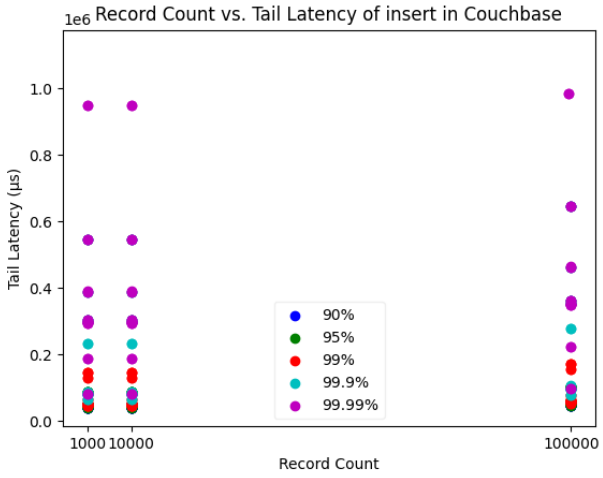
Fig. 7: Field length (bytes) impact on read tail latency in Couchbase



3) Record Count vs. Tail Latency: Similar to the case in subsections IV-B1 and IV-B2, the tail latency of the

write operations (update and insert) were not significantly impacted by modifying the record (document) count, for the same reason: changing the database size does not change the durability requirements enforced in write operations. This lack of correlation is depicted in Figures 8 and 9. The tail latency of the read operation, however, was significantly positively correlated with the increasing record count. As the number of documents in the database grows, so does the usage of the available memory quota, which increases the likelihood of ejecting data to disk; because of this, a greater proportion of read operations must go all the way to disk to read data since it is no longer in memory. The natural result is higher tail latencies for the read operation with a larger number of stored documents.

Fig. 8: Document/Record count impact on insert tail latency in Couchbase



4) Request Distribution vs. Tail Latency: While figures 11, 12, and 13 hint that there's not a significant relationship between request distribution and tail latency regardless of operation type, it is worth noting that YCSB's "zipfian" request distribution does seem to offer around 40% lower latency for the update operation specifically. With the zipfian request distribution, some data items (called a "hotspot") have a greater probability to be targeted by operations than other items. We can make sense of the lower 99.99th percentile for zipfian if we consider that a random "hotspot" could, by chance, align with the "active set" – that is, the set of data items currently in the Couchbase Cluster memory as opposed to disk. In other words, zipfian could randomly choose the ideal set of data items to operate on such that few or none of them have been ejected to disk.

5) Operation Proportion vs. Tail Latency: This particular test yielded results that are both interesting and perhaps the least insightful. As shown in Figures 14 and 15 in which the X axis labels are formatted as *read ratio - update ratio - insert ratio*, the average 95th and 99th percentile latency across all

Fig. 9: Document/Record count impact on update tail latency in Couchbase

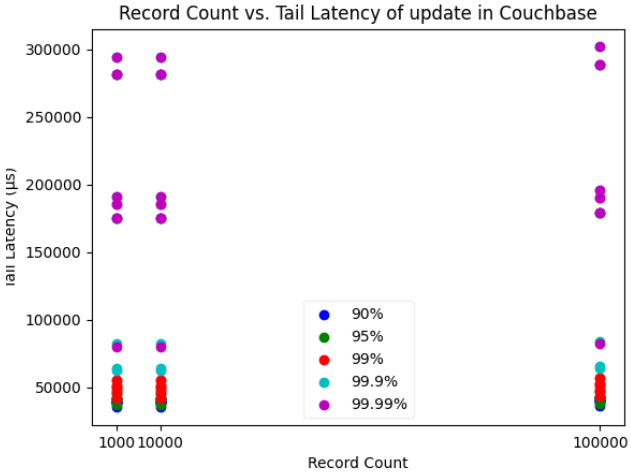
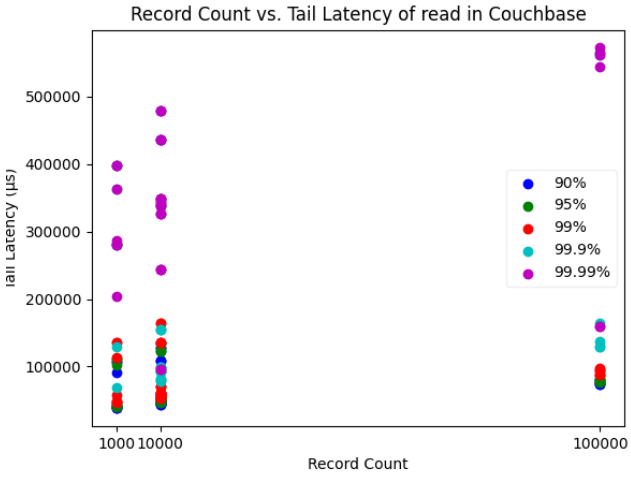


Fig. 10: Document/Record count impact on read tail latency in Couchbase



operations appear disconnected from whether a workload is read-intensive or write intensive. First, the local minima appear at 90% read-10% insert, 10% read-90% insert, and 100% read; thus, it cannot be concluded from this data that either a read or write-intensive workload provides lower overall tail latencies for all operations. Moreover, the local maxima appear at 100% update, 50% read-50% insert, 50% read-50% update, and 90% read-10% insert; thus it cannot be concluded from the maxima that either a read or write-intensive workload leads to higher overall tail latencies for all operations.

C. Phase Two Test Plan: Heterogeneous Service Scaling

In this section we provide our phase two approach taken to test the effect of multidimensional scaling on the tail latency

Fig. 11: Request distribution impact on insert tail latency in Couchbase

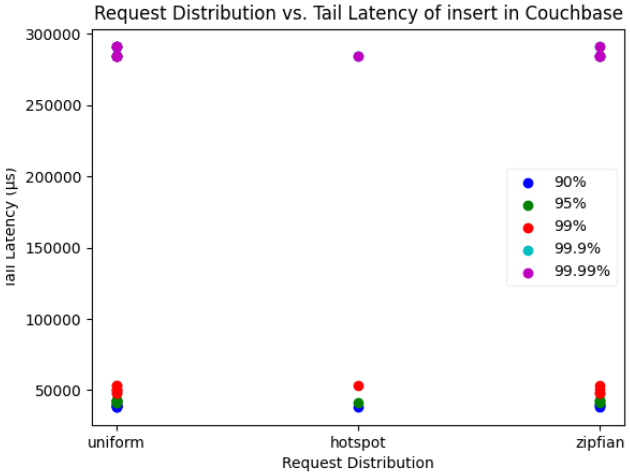
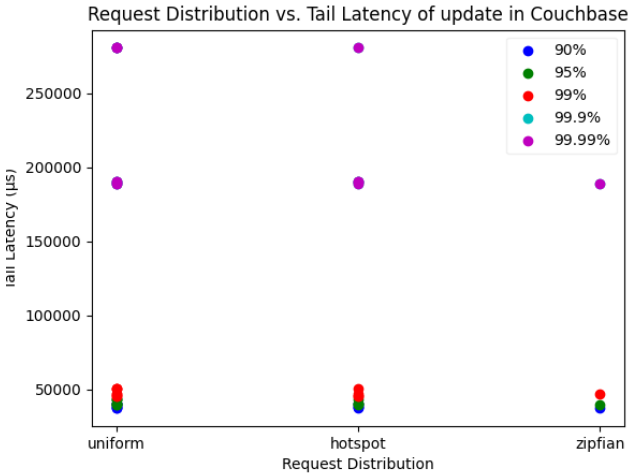


Fig. 12: Request distribution impact on update tail latency in Couchbase

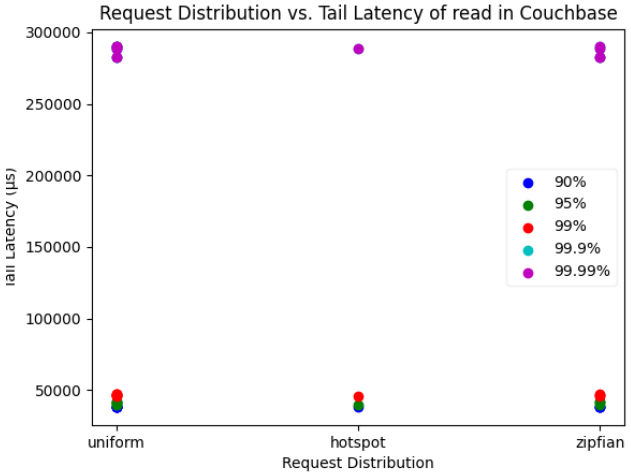


of various operations in Couchbase. Unfortunately, we were constrained to a limited set of test cases by the Community edition of Couchbase, which requires that every node in a given cluster runs one of three possible combinations of services, listed below:

- data
- query, index, data
- FTS, query, index, data

Based on those requirements, the first thing to note is that every node must run the data service. This means that the data service could not be independently scaled from one to many nodes. Also, since the query service cannot run independently, in order to test the impact of scaling the query service, we

Fig. 13: Request distribution impact on read tail latency in Couchbase



needed to scale not just the query service from one to many nodes, but the query, index, and data service grouped together from one to many nodes, with the remaining nodes for each iteration only running the data service. This same constraint applies to the index service. As a result, the query and index services could not be tested independently and thus the results of the query service scaling tests are the same results for the index service scaling test. Lastly, since the FTS (full text search) service can not run independently, in order to test the scaling of FTS, we needed to scale not just the FTS service from one to many nodes, but the FTS, query, index, and data services grouped together, with the remaining nodes for each iteration only running query, index, and data. This in essence means that all nodes are running query, index, and data for all iterations, but only the FTS service is scaling horizontally even though it is tied to the other services. In short, we are only able to test the impact of scaling FTS, query, and index, where the testing for query and index is essentially duplicated.

Our driving test algorithm for testing the impact of multidimensional scaling on the tail latency of each operation is shown in Algorithm 2. Unlike the tests in the previous phase, this driving algorithm is only one flat loop rather than a nested one.

D. Phase Two Test Results

1) FTS Scaling vs. Operation Latency: As seen in Figures 16, 17, and 18 scaling the FTS service beyond one node to two significantly reduced the top percentile latencies for the insert, update, and FTS operations. However, there appears to be a slight spike (local maximum) in tail latency for the update, delete, and FTS operations when FTS is running on three of the five nodes. Since three happens to be the first number allowing for the existence of a quorum, this could be related

Fig. 14: Operation Proportion vs Average 95th Percentile Latency Across All Operations

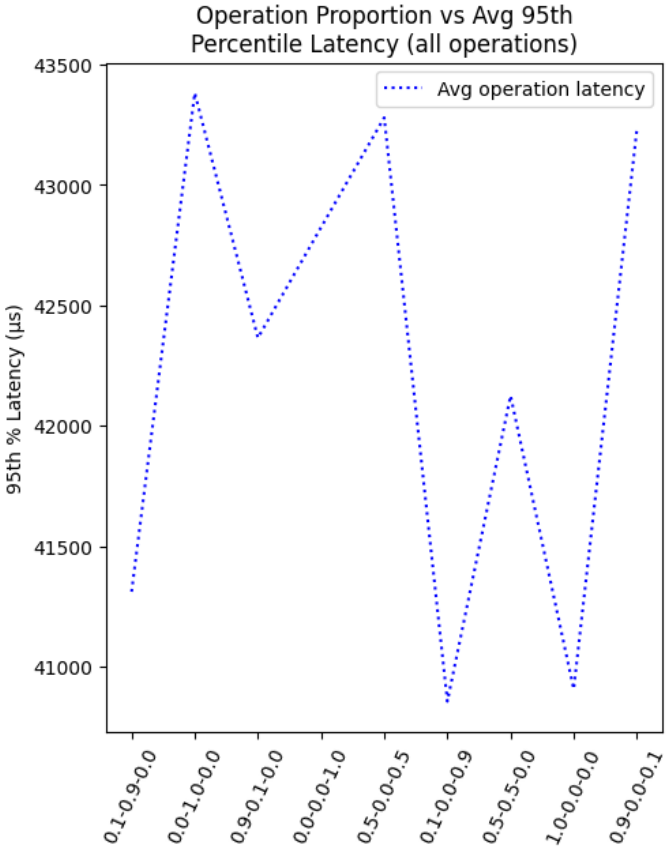
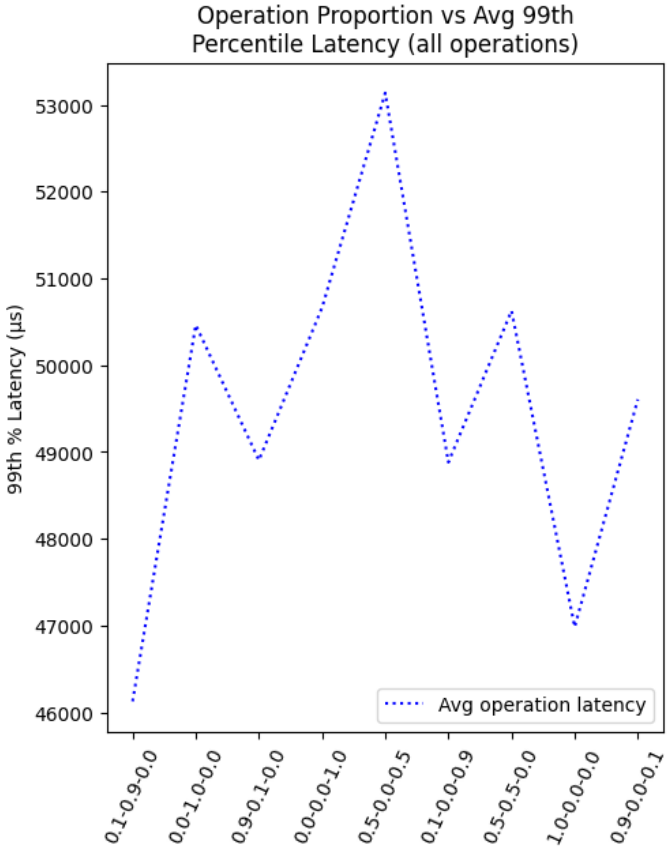


Fig. 15: Operation Proportion vs Average 99th Percentile Latency Across All Operations



to the way Couchbase handles cluster metadata management using the Chronicle consensus algorithm.

2) Query & Index Scaling vs. Tail Latency: As mentioned in subsection IV-C, the impact of scaling the query and index services could not be tested independently due to their inter-dependence in Couchbase Server. So, this subsection should be interpreted as a discussion of the impact of scaling both the query and index service together. In Figures 23 and 25, we see a similar maximum in top-percentile latencies for the update and N1QL query operations when the query and index services are running on three of the five nodes. For the FTS operation in Figure 21, we see a consistent, gradual decline in tail latency as the query and index services scale horizontally, and we see an even sharper decline in the tail latency of the delete operation, which essentially bottoms out once the query and index services are running on two nodes. It appears from Figure 22 that there is the the least amount of correlation between the scaling of query and index and the tail latency of the insert operation. When one considers that the primary service responsible for the insertion of new documents by key into the database is the data service and not the query or index service, this lack of correlation can be expected. Moreover, the consistent decline in the FTS tail latency with the scaling of the query and index service aligns with the documentation that states that the query service is responsible for performing scan operations on relevant search indexes in order to serve FTS queries; adding more of what's serving the queries should indeed have a positive impact on query latency.

V. Unresolved Problems & Future Work

While we were able to capture a number of key metrics in this evaluation of Couchbase, we have a couple of items we were hoping to include in the evaluation but were unable to due to time constraints. We outline them below.

1) Replication Count Impact On Operation Latency: In this evaluation, the number of replications is not treated as a variable. It is expected as data replication requirements change, the tail latency of write operations would change as well since all mutations would need to be streamed to the replicas. Coupled with different durability settings (i.e., Couchbase's *majority*, *majorityAndPersistToActive*, and *persistToMajority* durability configurations), the impact could be even more substantial. For example, for a cluster of a given size, how does changing the number of required replications from one to three affect the tail latency of the insert operation for each of the three main durability configurations? With more replication

Algorithm 2 Service Scaling Test Driver

1: $CLUSTERSIZE$ = 5
2: $DURABILITY$ = 'medium'
3: $BUCKETSIZE$ = 200 // *num docs*
4: $bucket$ = createBucket($BUCKETSIZE$)
5: $serviceLayouts$ = getServiceLayouts()
6: **for each** $slayout \in serviceLayouts$ **do**
setupHeterogeneousCluster($slayout$)
7: flushBucket($bucket$) // clean slate
8: runInserts($bucket$,...) // *write latencies*
9: runN1QLSelects($bucket$,...) // *write latencies*
10: runFTS($bucket$,...) // *write latencies*
11: runUpdates($bucket$,...) // *write latencies*
12: runDeletes($bucket$,...) // *write latencies*
13: flushBucket($buckete$,...) // clean slate
14: **end for**
15: $generatePlots()$ // *build plots with written data*

Fig. 16: Effect of Scaling FTS Service on FTS Operation

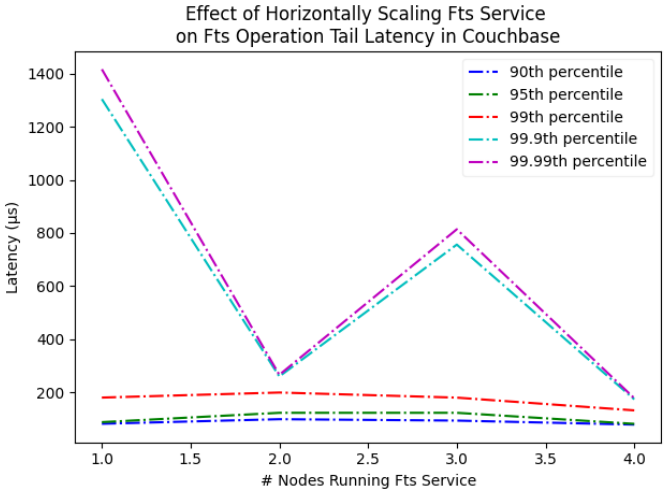


Fig. 17: Effect of Scaling FTS Service on Insert Operation

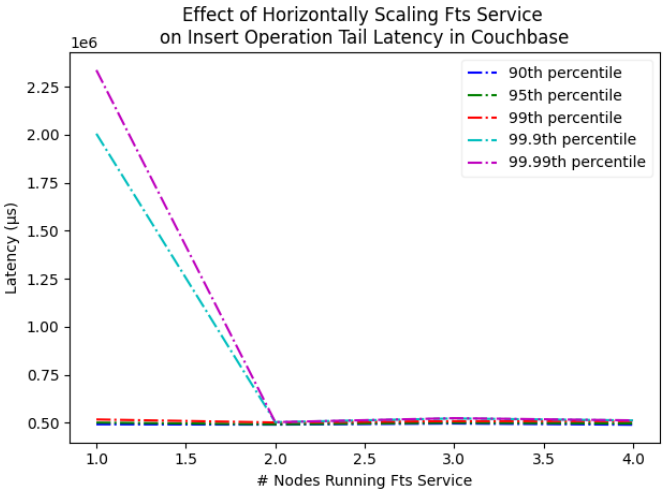
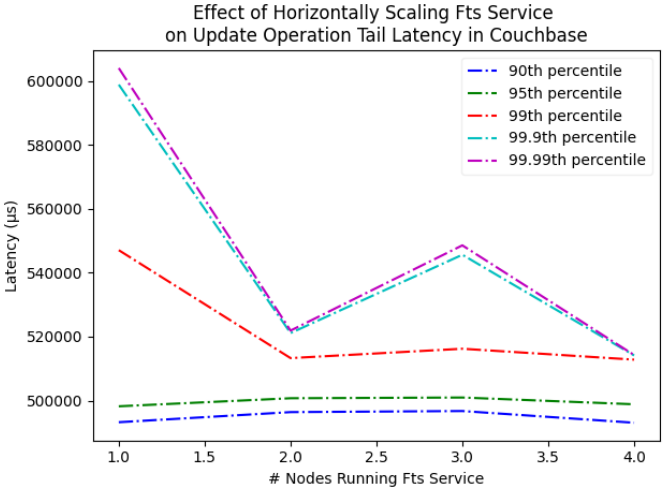


Fig. 18: Effect of Scaling FTS Service on Update Operation



and stronger durability settings, the cluster is more resilient in case of failures, but is it worth the expected increase in write tail latencies that could impact the customer experience?

2) Node Failover Impact on Operation Latency: Another item we did not address in the evaluation is the impact of node failover on the latency of the different operation types. More specifically, we wanted to generate a line graph (for each operation type) of the operation latencies (for, say, 100 operations) over time, during which a node was "gracefully" failed over (one of the two main node failover types offered by Couchbase Server, the other being "hard") while those operations were executing. The expected result would be a spike in the latency at the time of node failover. However, a key consideration with a test like this is that due to the vBucket partitioning within Couchbase as discussed in section II, an operation will only be affected if the node being failed over holds the active data on which the that operation is executing. That is, if a data operation is executed on some key K, the hash of that key points to a vBucket which belongs to some node N based on the cluster map, and the latency of that operation should only be affected if N is the one being intentionally failed over. Since our framework already includes both a Cluster Manager component and a Data Manager component as described in section III, it currently offers the capacity to run this kind of test, but it has not yet been implemented.

3) Work Load Condition of the Server: In this evaluation, the EC2 instances run only the Couchbase Server software. However, in a real-world application, cluster nodes may run a number of applications, which may introduce a series of background activities that may execute irregularly and cause resource contentions on the server, resulting in performance degradation for one or all applications involved. Therefore, we

Fig. 19: Effect of Scaling FTS Service on Delete Operation

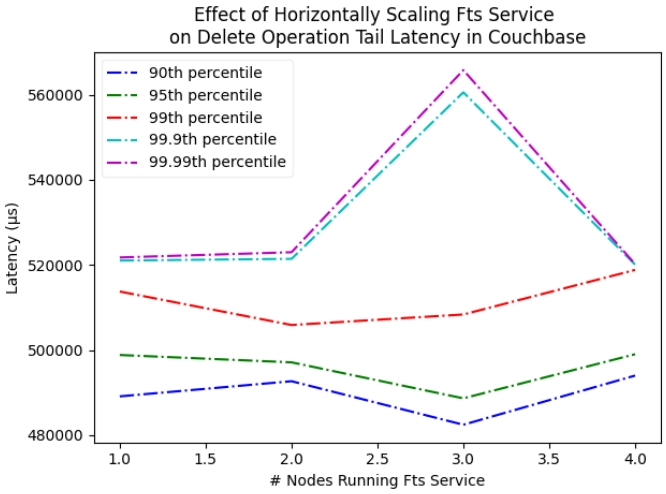


Fig. 20: Effect of Scaling FTS Service on N1QL Query Operation

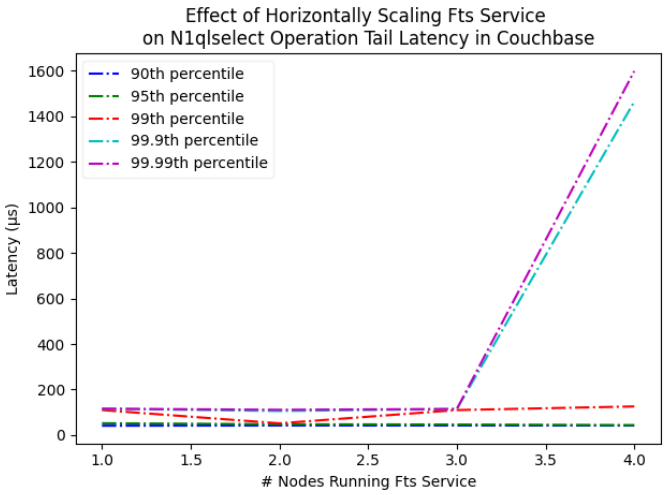


Fig. 21: Effect of Scaling Query & Index Services on FTS Operation

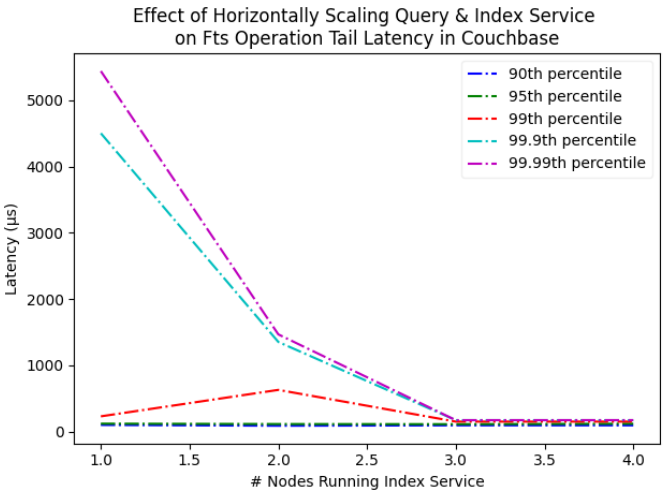
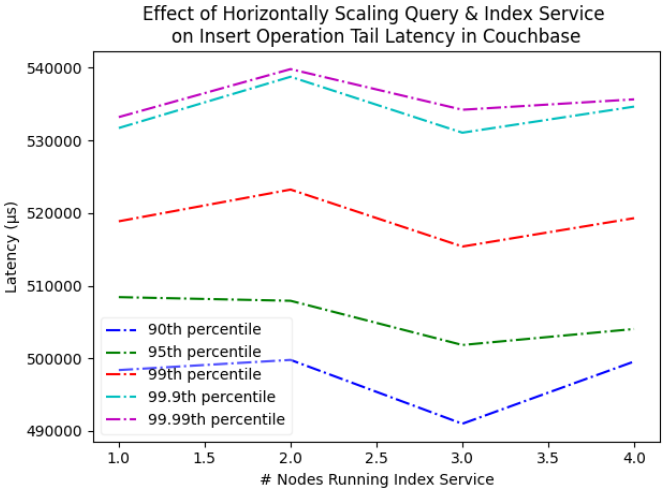


Fig. 22: Effect of Scaling Query & Index Services on Insert Operation



have planned but not yet implemented an approach for testing the effect of daemon processes on Couchbase operations. The plan is to spawn up a certain docker container that will consume a certain percentage of available CPU while making all other variables like cluster size, database size, and service layout constant, and analyze the daemon process's impact on the tail latency of different operation types.

4) Instance Type: In this evaluation, we are hosting our Couchbase cluster on a set of virtual machines provisioned with AWS EC2, a service that provides many configuration options around networking, volume storage, resource allocation via selection of instance types, and more. In this analysis we use the "t2.xlarge" instance type, as it meets the hardware requirements specified by Couchbase documentation. However, we remain interested in plotting the relationship between EC2 instance types and tail latencies of Couchbase data operations; we know horizontal scalability is a big advantage of Couchbase, but how does it respond to vertical scaling? Of course, the general expectation is that the more powerful the instance type, the smaller the tail latency (not to mention the guaranteed EC2 price change). We would also expect the existence of an inflection point at which vertical scaling no longer results in a significant reduction in tail latency. Also, that expected inflection point is likely different for different workloads. For each real-world application, there must exist some optimal set of instance types for a Couchbase cluster. If you can plot cost increase and tail latency decrease along a

Fig. 23: Effect of Scaling Query & Index Services on Update Operation

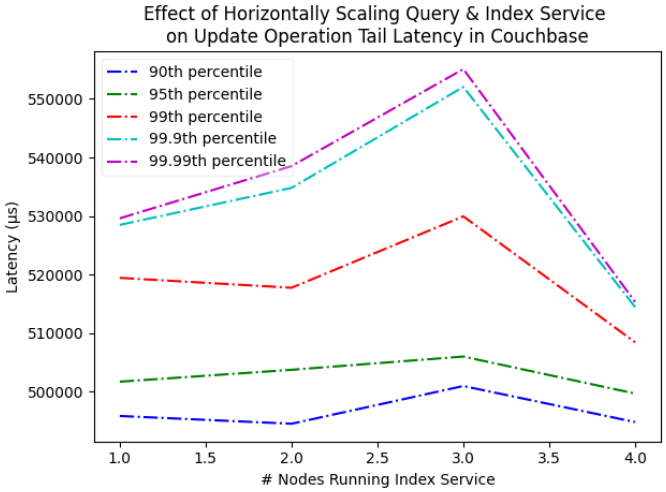
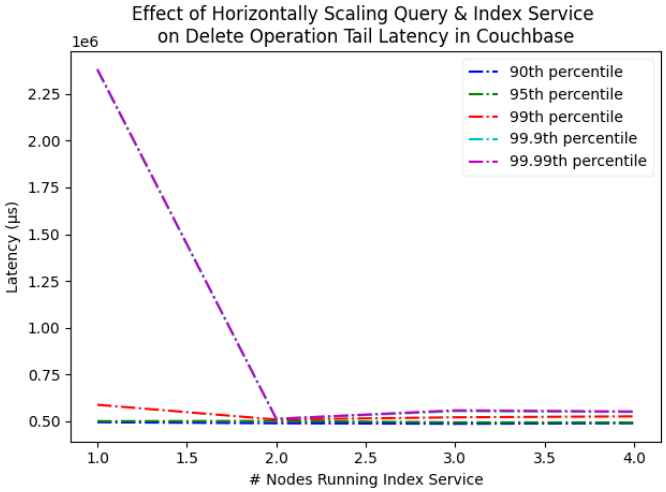


Fig. 24: Effect of Scaling Query & Index Services on Delete Operation

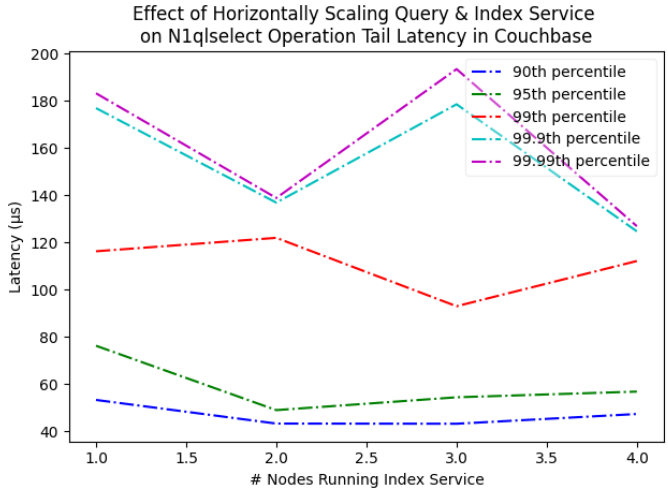


change in instance type configurations for a cluster, then you can identify the point at which the tail latency's negative slope is no longer steeper (if it ever is, that is) than the cost increase's positive slope; that is the inflection point where vertical scaling is not worth the money.

VI. Conclusion

Our goal with this study was to take Couchbase Server – an open-source, document-oriented, multi-model, distributed NoSQL database riding the wave of next-gen database systems – and gain an understanding of its multitude of configurable properties and how they interact with each other, and more importantly, how they combine to impact the tail latencies of key operations such as reads, insertions, updates, and deletions.

Fig. 25: Effect of Scaling Query & Index Services on N1QL Query Operation



We approached this goal with a strong focus on automation such that we could fine-tune and gauge a number of different settings without much overhead. This project, which makes heavy use of orchestration tools like Ansible and Vagrant in conjunction with cloud infrastructure via AWS EC2, combines experiences and perspectives from the rich arenas of both distributed systems and cloud computing, and rests on the shoulders of other open source giants who contributed frameworks like YCSB that allowed entire phases of this evaluation to be executed.

References

[Coo+10] Brian F Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154.

[HM17] Lorin Hochstein and Rene Moser. *Ansible: Up and Running: Automating configuration management and deployment the easy way*. " O'Reilly Media, Inc.", 2017.

[YN19] Artsiom Yudovin and Yauheniya Novikova. *Technical NoSQL Comparison Report:Couchbase Server v6.0, MongoDB v4.0, and Cassandra v6.7 (DataStax)*. 2019.

[Cou21a] Couchbase. *Couchbase CLI Reference*. https://docs.couchbase.com/server/current/cli/cli-intro.html. 2021.

[Cou21b] Couchbase. *Couchbase Documentation*. https://docs.couchbase.com/server/current/introduction/intro.html. 2021.

[Cou21c] Couchbase. *Couchbase Featured Customers*. https://www.couchbase.com/customers. 2021.

[Cou21d] Couchbase. *Couchbase Python SDK Reference*. https://docs.couchbase.com/python-sdk/current/hello-world/start-using-sdk.html. 2021.

[Mon21] MongoDB. *When to Use a NoSQL Database*. https://www.mongodb.com/nosql-explained/when-to-use-nosql. 2021.

Conduct Independent Inquiry in Computer Science

A Contextual Analysis of CRYSTALS-Dilithium, a Quantum-Resistant, Lattice-Based Digital Signature Scheme

Austin Hunt

Department of Computer Science

Vanderbilt University

Nashville, TN, USA

austin.j.hunt@vanderbilt.edu

I wrote the following paper for the CS-8395 Special Topics: Quantum Computing course taught by Dr. Easttom. The challenge was provide an analysis of a chosen quantum-resistant cryptographic algorithm, and while doing some initial investigation I came across a July 2022 announcement from the National Institute of Standards and Technology (NIST) about the first group of winning quantum-resistant cryptographic algorithms from its six-year competition; CRYSTALS-Dilithium was one of those. Before writing, I knew nothing about quantum-resistant cryptography or the role of lattice-based mathematics in that domain. However, I had recently written an article about digital signatures and their impact on modern business security, so CRYSTALS-Dilithium — a digital signature scheme — struck me as a worthwhile investigation. **It's easier to learn about new things if you can connect them to things you've already learned about.** With that, I should note that when writing this paper, I made a concerted effort to provide as much background and context around new topics before referencing or introducing them. For example, before diving too deeply into a discussion of lattice-based cryptography and its value in quantum resistance, I first provide an explanation of lattices in relation to vector spaces. Also, before discussing the Closest Vector Problem (CVP), I make an effort to explain the Shortest Vector Problem (SVP), which is generalized by CVP. There are other instances of this deconstruction and contextualization in the paper. I mentioned earlier in this portfolio that I have made a habit of deconstructing things I don't understand until I can meaningfully reassemble the parts I do understand. I took that approach with this paper, which entailed a significant amount of research on my part in the forms of reading other papers and articles and watching quite a few YouTube videos, including this 2018 Conference on Cryptographic Hardware and Embedded Systems presentation from Gregor Seiler (with IBM Research) on the design of CRYSTALS-Dilithium.

This research project did not involve the development of software or any other artifacts apart from the following paper.

A Contextual Analysis of CRYSTALS-Dilithium, a Quantum-Resistant, Lattice-Based Digital Signature Scheme

Austin Hunt
School of Engineering
Vanderbilt University
Greenville, SC
austin.j.hunt@vanderbilt.edu

***Abstract*—This paper presents an analysis of CRYSTALS-Dilithium, a novel algorithm for generating digital signatures that can withstand future cyberattacks from quantum computers. As announced in July 2022, CRYSTALS-Dilithium was one of the first four winners in the post-quantum cryptography (PQC) standardization project managed by The National Institute of Standards and Technology (NIST). NIST started this competition to call upon cryptographers across the globe for the creation and vetting of quantum-resistant encryption methods and standards in preparation for an inevitable future of quantum-computing-based assaults. As a member of the Cryptographic Suite for Algebraic Lattices (CRYSTALS), Dilithium is a lattice-based algorithm providing not only security based on the hardness of lattice problems, but also competitive public key compression and multiple efficient implementations.**

This paper includes an overview of the Dilithium scheme for quantum-resistant digital signatures and aggregates contextual information around deeper topics supporting its principal design aspects.

***Index Terms*—CRYSTALS, Dilithium, quantum computing, quantum-resistant cryptography, encryption, digital signatures**

I. Introduction

Maintaining trust in the digital world requires a certain guarantee that messages being sent and received are not being tampered with during transmission. Through the use of public-key cryptography, digital signatures provide that guarantee: before transmission, messages are hashed to produce message digests which are then encrypted with the sender's private key to produce the message's digital signature. The digital signature, appended to the message before transmission, allows the receiver to confirm that the message is coming from the expected sender; by hashing the message into a message digest using the same hash function as the sender, and by decrypting the appended digital signature with the sender's public key to obtain the *signed* message digest, the receiver can simply compare the two message digests to ensure they are the same. If they are, the integrity and authenticity of the message is verified.

Since today's digital signatures are built on public-key cryptography (PKC) as described, they are vulnerable to the upcoming wave of quantum computing since the prime factorization on which PKC's security relies is specific to the context of classical computing; factoring large numbers and thus breaking PKC will be a relatively easy and quick task for quantum computers, as shown by Jiang et al. in their work with quantum annealing for prime factorization [1].

Guided by this inevitability, The National Institute of Standards and Technology (NIST) initiated a competitive program in 2016 centered on the development of new public-key cryptographic algorithms resistant to quantum computing attacks [2], which comes down to replacing our currently standardized reliance on prime factorization for security. The candidate algorithms submitted to this competition are either general encryption algorithms for protecting data confidentiality during transmission or digital signature algorithms for protecting data integrity.

In July 2022, NIST announced the selection of four winning candidate algorithms for quantum-resistant public-key cryptography, one of which was the CRYSTALS-Dilithium digital signature algorithm [3]. This algorithm, recommended as the primary choice among two others (FALCON [4] and SPHINCS+ [5]), provides digital signature security based not on prime factorization problems, but the hardness of lattice problems, which have become quite popular in the defensive push for quantum-resistant cryptography as explored by Nathan Manohar in their 2016 Harvard research [6].

II. The CRYSTALS-Dilithium Scheme

This section will outline various important aspects of the CRYSTALS-Dilithium Scheme introduced by Ducas et. al in 2018 [7], and it will additionally provide contextual information for deeper topics on which this scheme rests, such as the hardness of the closest vector problem underlying lattice-based cryptography, the intentional omission of discrete Gaussian sampling by Dilithium's developers, the use of the Learning with Errors problem, and the foundational Fiat-Shamir with aborts technique for lattice-based signature schemes on which Dilithium's design is based.

A. The Hardness of Lattice-Based Cryptography

1) Lattices: To understand lattice-based cryptography - that is, cryptography involving and ultimately depending on lattices - it is important to first understand the concept of a lattice, which relies on some linear algebra. Ultimately, a lattice $\mathcal{L}$ can be thought of as an infinite, evenly spaced out grid of points extending into infinity in all directions, where that grid can be any number of dimensions N. If we consider the points as vectors (i.e., quantities with both a distance and a direction from the origin of the grid) and the grid itself as a vector space, then each vector in that infinite N-dimensional vector space, by definition, can be expressed as a linear combination of N basis vectors, meaning the full lattice can be described using those N basis vectors. Put more simply, every point in an N-dimensional lattice represents a weighted sum of N basis vectors. For example, a 3-dimensional lattice can be described with the basis vectors

$$\hat{\boldsymbol{\imath}} = \begin{bmatrix}1\\0\\0\end{bmatrix}, \hat{\boldsymbol{\jmath}} = \begin{bmatrix}0\\1\\0\end{bmatrix}, \hat{\boldsymbol{k}} = \begin{bmatrix}0\\0\\1\end{bmatrix}$$

since every 3-dimensional vector v can be expressed as a weighted sum of those vectors, i.e.

$$v \in \mathbb{R}^3 \implies v = a * \hat{\boldsymbol{\imath}} + b * \hat{\boldsymbol{\jmath}} + c * \hat{\boldsymbol{k}}$$

It is key to remember that a lattice can be described by an infinite number of basis vectors.

2) Shortest Vector Problem: The fundamental quantity of interest with lattice-based cryptography, as MIT Professor Vinod Vaikuntanathan explained quite well in his 2015 talk on the mathematics of lattices [8], is the shortest non-zero vector in a lattice. That is, given an arbitrary basis, the goal with this famous Shortest Vector Problem is to find the shortest non-zero vector (of length λ_1) in the lattice described by that basis. An extension to this problem is simply to find an α-approximation of that shortest vector, or more specifically, a vector that is at most $\alpha * \lambda_1$ in length, λ_1 being the shortest length. Extending further upon this idea is the Shortest Independent Vectors Problem, which involves finding the shortest n linearly independent[1] vectors in a lattice such that you obtain the first shortest length λ_1 up to the nth shortest length λ_n.

3) Closest Vector Problem: Contrary to the previously provided 3-dimensional example, lattices used in lattice-based cryptography can be hundreds or even more than a thousand dimensions, as explained by Chris Peikert, a researcher at University of Michigan College of Engineering [9]; this implies the use of hundreds or more than a thousand basis vectors to describe these lattices. In the simplest form, this kind of cryptography relies on the hardness of finding, in a such a high-dimensional lattice, two points (or vectors) that are relatively close to each other. The Closest Vector Problem is a generalization of the Shortest Vector Problem described as follows: given a basis B for a lattice $\mathcal{L}$ and a random vector v not necessarily in $\mathcal{L}$, find the vector (linear combination of the basis) in $\mathcal{L}$ that is closest to v in Euclidean distance.

[1]Vectors are linearly independent if they cannot be written as linear combinations of each other.

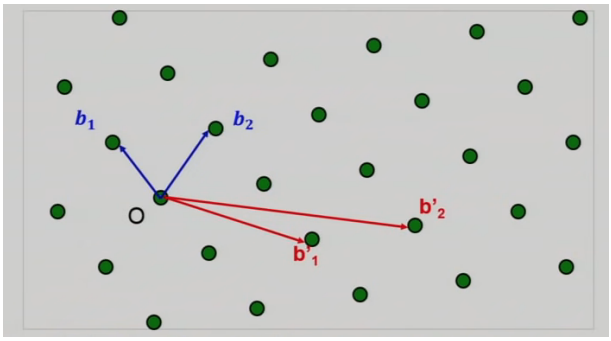


Fig. 1: Good Basis vs. Bad Basis

4) Hardness: In 1998, Miklós Ajtai proved that the Shortest Vector Problem (SVP) is NP-hard [10], and since Goldreich et al. proved the following year that the hardness for the Shortest Vector Problem implies equivalent hardness for the Closest Vector Problem [11], both of these fundamental problems on which lattice-based cryptography is based offer protection from both classical and quantum attacks. The best known algorithms for solving these lattice problems run in exponential time, even using approximation, and even using quantum computing, which is why public-key cryptosystems built on top of such problems, e.g., CRYSTALS-Dilithium, are less vulnerable than classical PKC.

5) Good Bases versus Bad Bases: It is important to note here that with lattices, which can be described with infinitely many bases, there is a notion of good bases and bad bases. In short, the difference comes down to the length, or Euclidean distance, of the basis vectors. A good basis for a lattice is comprised of short vectors, and a bad basis is comprised of long vectors, as visualized very simplistically in Figure 1, in which the blue basis is good and the red basis is bad. This is important because that lattice problems are easier to solve when the given basis is good and are harder to solve when the given basis is bad [8].

B. Short Integer Solution (SIS) and Learning with Errors (LWE) Problems

The Short Integer Solution (SIS) problem is equivalent to the Shortest Vector Problem on a given lattice, but is expressed a bit differently. SIS is posed as follows:

Given a matrix $A \in \mathbb{Z}_q^{n\times m}$, find a vector $r \in \mathbb{Z}^m$ such that

$$Ar = 0 \text{ (over } \mathbb{Z}_q^n\text{) and } ||r_2|| \leq \beta$$

Essentially, find a short (that is, with norm bounded by β) vector r of dimension m that when transformed by a matrix A produces 0, where A is in $\mathbb{Z}_q^{n\times m}$, meaning A is a matrix whose columns are in the set of all n-dimensional integer vectors modulo q. SIS is an average case problem, meaning any randomly selected instance of this problem will be hard

to solve, which is a necessary requirement for quantum-robust cryptography problems.

Building on top of SIS, the Learning with Errors (LWE) problem, introduced by Oded Regev in 2005 [12], is another quantum-robust lattice problem centered on finding a secret vector in a lattice from information clouded with intentional noise, or error. Similar to SIS, LWE has parameters n and q, with the addition of an error distribution parameter for adding noise that makes finding the secret more difficult. The LWE problem takes multiple forms. The LWE search problem is posed as follows:

Given a fixed n and q, and an error distribution,

find the secret $s \in \mathbb{Z}_q^n$ given many 'noisy inner products':

$$a_1 \leftarrow \mathbb{Z}_q^n, b_1 \approxeq \langle s, a_1 \rangle \bmod q$$

$$a_2 \leftarrow \mathbb{Z}_q^n, b_2 \approxeq \langle s, a_2 \rangle \bmod q$$

$$...$$

$$\Longrightarrow$$

$$a_1 \leftarrow \mathbb{Z}_q^n, b_1 = \langle s, a_1 \rangle + e_1 \in \mathbb{Z}_q$$

$$a_2 \leftarrow \mathbb{Z}_q^n, b_2 = \langle s, a_2 \rangle + e_2 \in \mathbb{Z}_q$$

$$...$$

where e_i for each noisy sample i is generally sampled from a discrete Gaussian distribution. Extending on the LWE search problem, the LWE decision problem poses the challenge of distinguishing (a_i, b_i) pairs generated as described above (with partial corruption via error vector sampling) from truly random and uniform (a_i, b_i) pairs. The LWE problem is another problem leveraged by CRYSTALS-Dilithium [13], and solving it has been proven to be at least as hard as solving approximate lattice problems in the worst case [12].

1) Discrete Gaussian Sampling: As discussed by János Folláth in their 2014 exploration of Gaussian sampling in lattice-based cryptography [14], many cryptographic algorithms involving lattices have been designed to require sampling from discrete Gaussian distributions as described in subsection II-B. According to Regev et al. [15, p. 2], "a discrete Gaussian distribution over some fixed lattice $\mathcal{L}$, denoted as $D_{\mathcal{L},s}$ for some parameter $s > 0$, is a distribution in which each lattice point is sampled with probability proportional to the probability density function of a continuous Gaussian distribution of width s evaluated at that point". These samplings according to literature are used because they provide a way to approximate uniform error vectors without knowing the lattice structure ahead of time. The developers of CRYSTALS-Dilithium, as confirmed by Gregor Seiler [13], intentionally avoided the use of discrete Gaussian sampling in their scheme because it is both difficult to implement correctly and it is even more difficult to achieve constant time execution. This omission contributed to one of their principal design considerations: the scheme needed to be easy to implement securely.

2) Ring-LWE and Module-LWE: One of the principal design goals held by the Dilithium developers was modularity, and a significant factor in achieving that goal was the decision to use Module-LWE rather than plain or Ring-LWE. [13]. To understand this reasoning, we need to understand each of these types of Learning With Errors problems. First, a ring in abstract algebra is just a set R that is closed under the two operations of addition and multiplication (meaning for any $x \in R, y \in R$, $x + y \in R$ and $x * y \in R$). Put even more simply by Dr. Chuck Easttom in his book on Quantum Computing Fundamentals [16, p. 80], "A ring is essentially just an abelian group[2] that has a second operation [of multiplication]". Albrecht et al. explains in their 2017 paper comparing Large Modulus Ring-LWE with Module-LWE [17] that the Ring-LWE problem is basically a form of LWE in which the n-dimensional vectors are replaced by polynomials with degree less than n. With this problem, you start by choosing "a ring R of dimension n, a modulus q and an error distribution χ over a related space of dimension n denoted $K_{\mathbb{R}}$". [17, p. 2]. Then, similar to how a_i is sampled from Z_q^n with plain LWE as shown in subsection II-B, you now instead sample a_i from $\frac{R}{qR}$, ending up with a similar sampling that looks like

$$a_i \leftarrow \frac{R}{qR}, b_i = \frac{1}{q} * a * s + e \bmod R^v$$

$$...$$

where R^v is the dual of ring R. A more precise definition is offered by Albrecht et al. in Section 2.3 of their paper [17, p. 9]. Albrecht et al. offers additional insight on Module-LWE (MLWE), showing that it was actually introduced to address shortcomings in both plain and Ring-LWE by "interpolating between the two" [17, p. 4]. He defines the MWLE informally as basically the same problem as RLWE but with the single elements a (sampled from $\frac{R}{qR}$) and s (the secret) from ring R replaced with module elements (vectors) over the same ring R. From Richard E. Borcherds' online course on Ring Theory [18], one ascertains that that modules in mathematics are defined essentially the same way as vector spaces [3], thus module elements are just vectors. On that note, according to Gregor Seiler's presentation [13], the key advantage of Module-LWE is using a whole vector over the ring R such that security of the digital signature scheme can easily be tuned by adjusting the length of that vector without ever having to change R itself. Since the same ring R is used for all security levels, he says the arithmetic for Dilithium only needs to be optimized one time.

C. Small Public Key and Signature

Another principal design goal shared by the Dilithium developers was the need for a small total size of the digital

[2]An abelian group is a group whose operation, e.g., addition, is commutative

[3]As briefly hinted at in subsection II-A, a vector space is a set of vectors closed under the operations of addition and scalar multiplication; another way of viewing it is as the set of all linear combinations of the basis vectors describing the vector space.

signature combined with the public key. Multiple design choices contributed to achieving this goal, as outlined in the following subsections. Ultimately, CRYSTALS-Dilithium offered the second-smallest combined size (after FALCON [4]) among the NIST PQC candidates.

1) Fiat-Shamir with Aborts: One of the team members behind Dilithium - Gregor Seiler, an IBM researcher - in his presentation on the scheme at the 2018 Conference on Cryptographic Hardware and Embedded Systems (CHES) [13], pointed out that the scheme design is based on a technique called Fiat-Shamir with Aborts. This technique, invented by Vadim Lyubashevsky in 2009 [19], opened the door to lattice-based signature schemes that could produce digital signatures on the order of about 50,000 bits compared to prior schemes that were producing signatures on the order of millions of bits in length; considering digital signatures need to be transferred over the network in addition to whatever messages are being signed to verify their integrity, this implied a significant advantage over previous techniques. Also, the security of Lyubashevsky's technique was based on the hardness of the shortest vector problem as described in subsection II-A.

2) Compression: CRYSTALS-Dilithium also leveraged developments from a series of papers in 2012 [20] and 2014 [21] to implement digital signature compression which helped them achieve signature sizes over 50% smaller by essentially only sending about half of the signature data over the network. On top of this, Dilithium introduced compression of the public key as well, achieving about a 60% smaller public key than previous schemes with only a slight increase in the size of the digital signature (by 100 bytes). Keeping in mind the Fiat-Shamir with Aborts advantage of reducing the signature size from millions of bits to 50,000 bits, a 100 byte increase is very minor.

III. Implementation

The CRYSTALS-Dilithium project is housed on GitHub [22] in two forms: one is the reference implementation written in plain C and the other implementation is optimized to run with the AVX2 instruction set.[4]. This section provides a simplified look at the Dilithium algorithm and includes performance data shared by Seiler in 2018 [13] for the reference version of the implementation.

A. The Algorithm

The Dilithium scheme is split into 3 main components: key generation, signing, and verification.

1) Key Generation:

$$A \leftarrow R^{5\times 4}$$

$$s_1 \leftarrow S_5^4, s_2 \leftarrow S_5^5$$

$$t = As_1 + s_2$$

$$pk = (A, t), sk = (A, t, s_1, s_2)$$

For key generation, a matrix A is obtained via random sampling. Then, two short vectors s_1 and s_2 are also sampled. Then, A, s_1, and s_2 are used to calculate t, an LWE vector. A and t then comprise the public key, while A, t, s_1 and s_2 comprise the secret, or private, key.

2) Signing:

$$y \leftarrow S_\gamma^4$$

$$w = Ay$$

$$c = \text{H}(\text{High}(w),\ M) \in B_{60}$$

$$z = y + cs1$$

$$||z||_\infty > \gamma - \beta \vee ||\text{Low}(w - cs_2)||_\infty > \gamma - \beta \implies \text{restart}$$

$$sig = (z, c)$$

The signing portion of the scheme uses a Fiat-Shamir transform. First, a short vector y is chosen, then Ay is stored in w. Then, the high bits of w are put into a hash function with a message M (signature compression comes from using only the high bits of w) and the digest (hash output) becomes the challenge polynomial c. Then, the sum $y + cs_1$ is stored in z. At this point, there is the important step of rejection sampling where basically the scheme is checking if either z or $w - cs_2$ reveals any secret information. If it does, the signing process is restarted. Otherwise, the digital signature is created using z and c.

3) Verification:

$$c' = \text{H}(\text{High}(\underbrace{Az - ct}_{w - cs_2}), M)$$

$$||z||_\infty \leq \gamma - \beta \wedge c' = c \implies \text{accept}$$

This verification process run by the message receiver first assigns another challenge polynomial c' using what equates to the high bits of w. It then runs yet another rejection sampling step to check if that new challenge polynomial c' equals the previous one c that was included in the received digital signature. This check corresponds to the classical digital signature verification step of comparing a message digest produced by hashing the received message against the message digest produced from decrypting the digital signature with the sender's public key. In both cases, a match indicates integrity and authenticity of the received message.

B. Overview

The main two operations important to the performance of the Dilithium scheme are polynomial multiplication in a fixed ring R (because it doesn't need to change thanks to MLWE as previously discussed) and the extensive parallel use of the SHAKE XOF (extendable output function)[5]. Seiler emphasizes that their implementation runs in constant time [13].

[4]Advanced Vector Extensions 2 (AVX2) is an extension to the Intel x86 instruction set that is able to handle single instruction multiple data (SIMD) instructions over 256-bit vectors. [23]

[5]An extendable output function (XOF) is just a cryptographic hash function that takes a bit string as input and can output a random string of bits of any desired length n.

C. Performance Data

Table I displays performance data from tests run with the reference implementation on the Intel Skylake i7-6600U processor. At a glance, one can see that multiplication and the use of the SHAKE extendable output function make up most of the time for the signing portion of the scheme by a significant margin. Table II displays similar performance data from tests run with the AVX2-implementation on the same processor. With this implementation, one quickly notices that each of the column totals dropped significantly, with the total median cycles for the signing portion of the scheme dropping from approximately 2.2 million to a competitive 635,000 on average.

IV. Conclusion

The CRYSTALS-Dilithium digital signature scheme submitted to and selected as one of a few winners of the NIST PQC competition exhibits many interesting characteristics, some of which are fundamental and shared by many lattice-based cryptography projects, some of which are unique and offer a competitive edge against other digital signature schemes in the quantum-resistant digital signatures arena. The scheme, like others, utilizes decades of lattice-based mathematics and number theory to provide robust protection against the upcoming surge of quantum-based attacks, and simultaneously offers implementation benefits like high performance, impressive efficiency, and strong public key and signature compression that will make deploying it practically beneficial.

References

[1] S. Jiang, K. A. Britt, A. J. McCaskey, T. S. Humble, and S. Kais, "Quantum annealing for prime factorization," *Scientific reports*, vol. 8, no. 1, pp. 1–9, 2018.

[2] L. Chen, D. Moody, and Y.-K. Liu, "Post-quantum cryptography," https://csrc.nist.gov/projects/post-quantum-cryptography, 2022, accessed: 2022-09-24.

[3] C. Boutin, "Nist announces first four quantum-resistant cryptographic algorithms," 2022.

[4] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhange, "Falcon: Fast-fourier lattice-based compact signatures over ntru," https://falcon-sign.info/, 2017, accessed: 2022-09-24.

[5] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The sphincs+ signature framework," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 2129–2146, accessed: 2022-09-24.

[6] N. Manohar, "Hardness of lattice problems for use in cryptography," Ph.D. dissertation, 2016.

[7] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-dilithium: A lattice-based digital signature scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 238–268, 2018.

[8] Vinod Vaikuntanathan, "The mathematics of lattices i," https://www.youtube.com/watch?v=LlPXfy6bKIY, 2015, accessed: 2022-09-24.

[9] Chris Peikert, "Lattice cryptography: A new unbreakable code," https://www.youtube.com/watch?v=2IyotuA8eJc, 2019, accessed: 2022-09-24.

[10] M. Ajtai, "The shortest vector problem in l2 is np-hard for randomized reductions (extended abstract)," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, ser. STOC '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 10–19. [Online]. Available: https://doi.org/10.1145/276698.276705

[11] O. Goldreich, D. Micciancio, S. Safra, and J.-P. Seifert, "Approximating shortest lattice vectors is not harder than approximating closest lattice vectors," *Information Processing Letters*, vol. 71, no. 2, pp. 55–61, 1999.

[12] O. Regev, "The learning with errors problem," *Invited survey in CCC*, vol. 7, no. 30, p. 11, 2010, accessed: 2022-09-24.

[13] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-dilithium: A lattice based digital signature scheme," 2018.

[14] J. Folláth, "Gaussian sampling in lattice based cryptography," *Tatra Mountains Mathematical Publications*, vol. 60, no. 1, pp. 1–23, 2014, accessed: 2022-09-24.

[15] D. Aggarwal and O. Regev, "A note on discrete gaussian combinations of lattice vectors," *arXiv preprint arXiv:1308.2405*, 2013, accessed: 2022-09-24.

[16] C. Easttom, *Quantum computing fundamentals*. Addison-Wesley Professional, 2021.

[17] M. R. Albrecht and A. Deo, "Large modulus ring-lwe $\geq$ module-lwe," *Cryptology ePrint Archive*, 2017.

[18] Richard E Borcherds, "Rings and modules 1 introduction," https://www.youtube.com/watch?v=4WmIDodIgaclist=PL8yHsr3EFj52XDLrmvrFDgw 2021, accessed: 2022-09-24.

[19] V. Lyubashevsky, "Fiat-shamir with aborts: Applications to lattice and factoring-based signatures," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2009, pp. 598–616, accessed: 2022-09-24.

[20] T. Güneysu, V. Lyubashevsky, and T. Pöppelmann, "Practical lattice-based cryptography: A signature scheme for embedded systems," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012, pp. 530–547.

[21] S. Bai and S. D. Galbraith, "An improved compression technique for signatures based on learning with errors," in *Cryptographers' Track at the RSA Conference*. Springer, 2014, pp. 28–47.

[22] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Dilithium," https://github.com/pq-crystals/dilithium, 2018.

[23] Intel, "Intrinsics for intel advanced vector extensions 2 (intel avx2)," https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx2.html, 2022, accessed: 2022-09-24.

	Key Generation	Signing	Signing (average)	Verification
Multiplication	89,591	987,666	1,280,053	143,924
SHAKE	178,487	314,570	377,068	161,079
Modular Reduction	11,944	120,793	163,017	10,626
Rounding	6,586	108,412	137,324	11,821
Rejection Sampling	60,740	76,893	94,607	28,082
Addition	8,008	58,696	79,498	10,723
Packing	7,114	17,183	18,856	8,883
Total	381,178	1,778,148	2,260,429	396,043

TABLE I: Median cycles of 5000 executions on Intel Skylake i7-6600U processor for reference implementation

Median cycles of 5000 executions on Intel Skylake i7-6600U processor				
	Key Generation	Signing	Signing (average)	Verification
Multiplication	15,974	155,721	201,347	25,471
SHAKE	96,779	170,232	205,847	90,921
Modular Reduction	1,034	7,902	10,541	708
Rounding	728	7,541	9,904	2,479
Rejection Sampling	62,272	67,193	81,278	27,737
Addition	8,028	46,755	62,453	8,659
Packing	6,997	16,200	17,526	8,712
Total	199,306	510,298	635,019	174,951

TABLE II: Median cycles of 5000 executions on Intel Skylake i7-6600U processor for AVX2-optimized implementation