SOFTWARE ARCHITECTURES AND PATTERNS FOR PERSISTENCE IN HETEROGENEOUS DATA-INTENSIVE SYSTEMS

by

AARON SCHRAM

B.S., University of Colorado at Boulder, 2006

M.S., University of Colorado at Boulder, 2011

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirement for the degree of

Doctor of Philosophy

Department of Computer Science

2015

This thesis entitled:
Software Architectures and Patterns for Persistence in
Heterogeneous Data-Intensive Systems
written by Aaron Schram
has been approved for the Department of Computer Science

_____
Professor Kenneth M. Anderson, Chair

_____
Professor Richard Han

_____
Professor James Martin

_____
Professor Nenad Medvidovic

_____
Professor Judith Stafford

Date_____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

ABSTRACT

Schram, Aaron (Ph.D., Computer Science)

Software Architectures and Patterns for Persistence in Heterogeneous Data-Intensive Systems

Thesis directed by Associate Professor Kenneth M. Anderson

Software engineers are faced with a variety of difficult choices when selecting appropriate technologies on which to base a software system. As the typical software user has become accustomed to systems being "on-demand" and "always-available," the software engineer is more concerned than ever before about the issues of system scalability, availability, and durability. In the absence of expertise in distributed systems, architectural decisions become complex, slowing feature development and introducing error. Software engineering is in need of robust patterns and tools that increase the accessibility of specialized technologies developed for the completion of specialized tasks. This dissertation describes my existing work related to the challenges of domain modeling and data-access in large-scale, heterogeneous data-intensive systems and extends this work to include novel architectures for utilizing multiple large-scale data stores effectively. This research focuses on increasing the accessibility and flexibility of these data stores, which typically afford scalability, availability, and durability at the cost of added complexity for the application developer. The resulting architecture and associated implementation alleviates common challenges faced by small and medium software enterprises during the development of heterogeneous data-intensive software applications.

DEDICATION

To

my parents and grandparents

who I will always admire for their

struggles, sacrifices, and hard work

ACKNOWLEDGEMENTS

I would like to thank my colleagues in the various software engineering organizations that I have been a part of throughout my career for the valuable experience and lessons taken from each. Specifically I would like to thank Nate Sammons, William Butler, and Burke Webster for their continued support and friendship through this and many other shared experiences. This dissertation would not have been possible without the positive and long lasting relationship with my advisor, Dr. Kenneth M. Anderson, which began so many years ago when I was an optimistic and overly curious undergraduate. Thank you for guiding me on a journey only you saw I needed to take. Finally, I would like to thank my incredibly supportive network of friends and colleagues for, among many other things, ensuring I enjoy the moment.

CONTENTS

TABLES

FIGURES

CHAPTER 1: Introduction

Big Data is a rapid increase in public awareness that data is a valuable resource for discovering useful and sometimes potentially harmful knowledge.
- Stephen Few [1]

This dissertation concerns itself with research issues related to the task of designing and implementing architectures capable of leveraging a variety of purpose-built frameworks and systems. While significant progress has been made in computer science research recently—specifically in the areas of large-scale computing and data processing—software engineering researchers have struggled to provide adequate and widely adopted abstractions for the construction of heterogeneous systems that are rapidly becoming commonplace in small and medium sized enterprises. Minimalizing the complexities encountered when adopting purpose-built frameworks aids in the greater transition towards a heterogeneous style of application architecture based upon numerous specialized technologies.

This introduction details the conditions that influenced the creation of today's heterogeneous approach to application development by exploring the need for further research in the design and application of patterns for large-scale systems of systems. We present an overview of past advances in enterprise patterns and technologies that have made contributions to the success of this research domain and discuss remaining challenges.

In the late 1990s, software companies began to experience massive consumer adoption of their technologies. Desktop and laptop computers quickly became ubiquitous in homes, schools, and offices. The Internet was also experiencing adoption at a staggering rate with 50% of adults (18+) going online by the year 2000 [2]. The rapidly growing base of users operated under new software paradigms and expectations compared to the expert users of the previous decades—expecting software packages to be accessible via a web browser. The browser greatly simplified the adoption of new software in a variety of ways for its users. The user no longer needed to install software from physical media (floppy or compact disk) thus, avoiding many complex failure cases associated with user-initiated software installations on widely varying hardware. Additionally, the user need not worry about backing up important information stored by the program: if the user's system experienced a hardware failure, the data was still safe with the application provider. Applications could be accessed that were running on machines with much more advanced hardware than the user owned, increasing performance while lowering cost for the user. Finally, the software was always up-to-date, providing the user with the latest in features and bug fixes without waiting for the next version to be shipped.

As more and more users adopted Internet based applications, software companies providing these applications experienced an incredible amount of growth. The growth of these businesses forced software engineering teams to innovate rapidly. Some of these efforts were conducted in well-funded research and

development organizations, such as government labs and large corporations (e.g., IBM), while others took place within the stereotypical software "startup" (e.g., Google, Yahoo!). Software applications of this nature became know as "Software as a Service" (SaaS) products. The name refers to the fact that software is delivered to the user as an "on-demand" service without the need for physical installation media. While this drastically decreased production costs for the business, it shifted many critical responsibilities from the user to the vendor, specifically the software engineering team. This also drastically reduced the software release cycle, which allowed vendors to push updates for features and bug fixes more frequently.

A typical SaaS application is hosted in a data center and operated by the same organization that develops the application. Current examples include Google Gmail [3], Facebook [4], and other applications of similar nature where users access the application via a remote client (e.g., web browser, mobile client) and the hosted application manages all aspects of the user's data. This style of application deployment motivated and continues to motivate software engineering teams to adopt multi-tenant architectures.

A multi-tenant architecture uses the same software and storage systems to service requests from many different clients and store data from many different users. A user from Company X will have their data stored in the same manner and possibly on the same physical system as a user from Company Y. This is an important architectural feature. Although it allows software engineers at the application level to quickly and easily interact with data, regardless of the

customer, it implies co-locating customers of extremely high and low use. A quality of service problem, attributed to a customer of extremely high use, affects the experience of other users.

As user- and machine-generated data volumes grew rapidly the bottlenecks of existing storage systems were quickly exposed. It was common for applications to be scaled "vertically", meaning the application (deployed on a single host) would be relocated to more capable hardware, providing improved processing, storage, and memory. This was an expensive and often error-prone process. If scaling vertically was not practical for an organization, the other option afforded to the organization was to performance tune the application to run more efficiently on existing hardware. Although this avoided the costs associated with purchasing new hardware, scaling via this method was often a stopgap and as costly as new hardware—if not more so—in additional engineering time. As more and more systems reached the limits of what could be accomplished by scaling vertically or performance tuning, software teams strove for a new approach.

Running a SaaS application on a single host introduces a variety of problems that can have disastrous consequences for any business, small or large. A system failure at the application or storage level can affect the user experience or result in data loss—both of which are devastating problems when selling an "on-demand" and "always available" service. Researchers and application developers were motivated to create new systems which were "horizontally" scalable, available, and durable. Focusing on these attributes allowed the SaaS vendor to offer a high

quality and cost-effective product to users while containing the costs of hosting, storage, and bandwidth.

Modern software systems are expected to be always available and, therefore, require redundancy throughout the application and the ability to quickly adapt to large increases in usage. For most applications this implies a distributed approach to system architecture. Increasing system capacity through distribution is described as scaling horizontally and is routinely accomplished by partitioning data and processing capacity across multiple hosts. Successfully implementing these strategies is extremely complex and requires expertise in distributed systems; however, the benefits for a software organization are numerous. Users expect on-demand products and providing them is now mandatory for even the smallest software-based businesses, which requires these software businesses to engineer their products to handle unavoidable situations, like hardware failures and extreme usage, without incurring downtime.

To keep up with changing demands, software businesses must architect systems that can scale efficiently without causing an interruption in service for their customers. As a business experiences success, there is incredible pressure to keep up with the largely unpredictable demand for their products or services. This makes it necessary for systems to be able to scale effectively with little to no notice. In traditional vertically-scalable systems, the software company orders better hardware and software from a vendor and then slowly integrates the new capabilities into their existing product. This approach impacts existing and new

customers, as the software team is required to shift their focus from implementing new features and customer requests to integrating new hardware and software. In some cases, the updates require new design and logic for the persistence tier of the application, which makes the persistence aspects of the application cumbersome to scale. Many applications are forced to implement complex "sharding" algorithms where the data is partitioned to accommodate future growth. This causes organizations to buy more equipment than they need and utilize it inefficiently unless the sharding algorithm is properly optimized. Undesired consequences of improper data sharding are numerous but often occur due to pinning a client to a single host that fails or becomes unresponsive due to high system load.

SaaS systems today make use of a variety of horizontally-scalable data storage solutions. Many encourage adopters to take advantage of inexpensive, off-the-shelf hardware that can be easily added to existing systems. In these systems, determination of how to partition an ever-growing amount of data is left up to the data storage system which transparently recognizes newly added hardware (i.e., servers) as additional resources, avoiding the need for expensive and customized hardware solutions. Through this strategy, increasing the capabilities of a software system to match business needs can be done incrementally and without incurring downtime, which allows a software or operations team to scale an application without diverting time away from feature development. With the advent of Infrastructure as a Service (IaaS) or "Cloud Computing", vendors like Amazon Web

Services enable organizations to scale systems up or down as needed, paying for only the processing and storage they use.

As the acquisition and maintenance of hardware continues to be commoditized by the Cloud, software engineers are provided an ever-increasing set of technologies on which to build their applications. These technologies are produced and distributed by experts in distributed systems, and, although these technologies are able to accommodate enormous growth in data, they are also extremely complex. Data storage, as a field of study, is where the most innovative approaches to scalability have been developed. Data is generated with staggering velocity and variation, which makes it extremely difficult to persist into rigid schemas. Applications making use of rigid specification languages such as XML, SQL, and others have struggled to adapt to ever-changing data structures that are generated by a diverse set of users and systems while gaining, modifying, and losing attributes with every iteration.

What was needed was the ability to easily accept data from a variety of sources without enforcing strict validation and binding logic. It became difficult to use contract-driven transport and storage interfaces. The popularization of REST [5] and JSON [6], [7] made communication between systems straightforward and flexible, through the use of standardized conventions, built upon HTTP and nested data structures. Additionally, key-value data structures became extremely prevalent throughout the persistence tier of the application. Many popular large-scale, open source storage systems use key-value pairs to achieve easily replicable

and flexible schema-less data storage (e.g., Cassandra, Hadoop). These systems use a unique key to identify an object of interest and store an entire object graph representation (i.e., embedded object graph) as a value. This approach not only allows an application to store an object in its current structure, regardless of the defined storage schema, but it also allows for objects to be split and/or replicated transparently among a cluster of machines, thus improving data availability and application scalability. Avoiding costly, error-prone schema updates allows an organization to adapt quickly to new challenges.

Software engineering, as a discipline, is increasingly at the center of the majority of engineering research, within and outside of computer science. Even beyond schools of engineering, it is common for research groups to incorporate a variety of available and adaptable software tools to interpret the growing amount of data being generated and captured in nearly every discipline. The most straightforward and commonly used systems are now capable of activities that would have been out of reach to all but the most well funded projects just a decade ago. It has never been easier to develop and deploy a SaaS application. The growing acceptance of Cloud based systems and web frameworks (e.g., Ruby on Rails) has redefined the skillsets required to be a successful application developer.

Understanding the motivating factors that led to the development and adoption of SaaS architectures is critical to the study of software engineering for heterogeneous systems. With complex system attributes such as scalability, availability, and durability required to meet user expectations, research is needed

to understand and extract accessible patterns. Doing so will allow software engineers to benefit from these technologies without requiring expertise in distributed systems, which can overwhelm even the most experienced application development teams. Organizations looking to manage the complexities associated with these technologies are subdividing team members into specialized groups of Software Configuration Management (SCM) engineers who manage the deployment and maintenance of these systems. These teams refer to themselves as developer operations (i.e., DevOps). Furthering software engineering research in this domain, we work to minimize the complexities inherent in developing distributed systems, offering an alternative to segmenting the engineering team in an effort to limit the amount of valuable engineering hours required to build and deploy heterogeneous systems—returning full focus to application development.

Indeed, in today's SaaS architectures scalability, availability, and durability are not "nice to haves" but instead, are basic requirements upon which the foundation of an application is built. *Better patterns and architectures in this area of research will allow software engineers to return focus to abstractions specific to their problem domain, without being overwhelmed by the complexities associated with the implementation of specific distributed systems.* In the next chapter, we will discuss, in detail, the classes of data storage technologies available to the software engineer and the complexities associated with the evolution from homogeneous to heterogeneous SaaS architectures.

CHAPTER 2: Large-Scale Data Stores

In this chapter, we present an introduction to large-scale heterogeneous systems and the specialized frameworks used to construct them. We focus on software as a service (SaaS) architectures as SaaS is the primary architecture used by production web applications today. We will also discuss, in detail, examples of the different types of specific tasks required by large-scale systems and the frameworks commonly used to accomplish each. The conclusion of this chapter will outline the challenges faced by an organization as they begin to design and implement a system of this nature. Given a thorough review of these specialized technologies, we will be well suited to recommend how each technology can be made more accessible to the software engineering community.

The primary roles fulfilled by software applications involve capturing, understanding, and presenting data. As data sets become too large to fit into memory or on local disks, applications need to contend with "Big Data" [8]. Because data continues to grow exponentially, the reasonable mitigation strategy is to acquire more resources to store and process it. This comes in the form of cheap and expendable computer servers or nodes which can be acquired for minimal cost and in large numbers (i.e. Moore's Law [9]). Networking these nodes into a cluster allows systems to be scaled horizontally with incremental and predictable cost to the organization.

Distributed systems researchers have provided software engineers with a diverse set of tools and techniques that can be used to efficiently utilize clusters of servers. The focus of tools in this research area is largely concentrated on improving data storage and access algorithms at low levels of abstraction and is primarily presented at conferences such as VLDB [10]. It must be noted that this research is incredibly important; without it, the software industry would not be where it is today. *However, an unfortunate side effect of a focus on low-level system attributes is that the abstractions provided to users (i.e., application developers) by these technologies require an in-depth knowledge of the underlying distributed systems to correctly and effectively use the provided Application Programming Interfaces (APIs).* This forces the application developer to deal with immense amounts of complexity when adopting new data storage technologies. To successfully adopt these technologies, an application developer must not only envision the current and future needs of the end user (i.e., the system user), but also understand the constraints which the architect of the system was under—appreciating the perspective, values, and trade-offs made by the system architect is key to effective utilization.

To better understand the current state of these frameworks, it is helpful to organize the most widely adopted technologies and present an overview of each.

2.1 Data Store Technology Overview

Capturing data is core to every software application and, as such, will be the focus of this section and a critical aspect of this dissertation. As the rate at which

data is captured continues to increase in volume and velocity, the need for specialized data storage systems is drastically increasing. Software engineers must choose from an ever-growing list of technologies developed to capture and sift through data. These technologies fall into one or more of the following classes shown in Table 2-1 and will be described throughout the remainder of this chapter.

Class	Technology	Attributes
Relational	Oracle, MySQL	Highly-structured data model, complex scalability, optimized for OLTP
Columnar	Bigtable, Cassandra	Semi-structured data model, linearly scalable, storage optimized for data reads
Key-value	GFS, Hadoop, Dynamo	Hash-based data model, linearly scalable, efficient reads and writes
Document	Lucene, MongoDB	Easily adoptable data model, used for information retrieval, optimized for flexible ad-hoc data access
Network	Pregal, Neo4j, Giraph, Titan	Relationship data model, small objects only, optimized for network traversals

Table 2-1. Popular classes of large-scale data stores.

2.1.1 Relational

Typically, an application begins by persisting data in a relational database management system (RDBMS). This type of solution stores data in a series of tables, each of which contains rows and columns. Data is grouped together on disk as continuous rows in a random access structure or an order generally matching the access pattern (e.g., B-tree). Each row is a datum, which is required to conform to

the schema associated with the table, thus, having a set of attributes represented by columns. Tables are linked via foreign keys that allow for traversal from one row to another. Object graphs are stored in a reference based and normalized representation. This design allows for space efficient utilization of storage resources (i.e., no data duplication due to normalization), but it requires random access to the storage layer when data references are resolved. This can make read operations on large data sets compute intensive, especially if the data is too large for memory or a solid-state drive (SSD), forcing data retrieval from a hard disk drive (HDD).

Much research has been done on making relational systems more read efficient, including denormalizing the schema as in a Star Schema [11]. Even with this approach, row based systems have been shown to underperform other designs when it comes to online analytical processing tasks [12]. However, even with their limitations, relational data stores provide end users with an extremely accessible data model that affords straightforward persistence and ad-hoc querying. To scale relational technologies to accommodate large amounts of data, data is partitioned in predetermined and application specific shards which makes persistence and retrieval across shards difficult if not carefully implemented. Availability is accomplished by replication of data to a "hot" backup. In the case of a system failure, all requests are switched to the replicas. Flexibility is largely given up to provide the user with data validation and normalization. It is worth noting that systems of this nature have been widely used for decades and scaled to extreme use. Harizopoulos showed that typical online transaction processing (OLTP) systems

spend their time divided almost evenly between logging, locking, latching, and buffer management [13]. However, removing any of these features from the OLTP system has been proven to drastically increase its performance [14], making RDBMS a viable technology choice for organizations willing to customize, tune, and maintain them. Typical vendors for this class of storage system are Oracle, IBM, SAP and Microsoft. Open Source alternatives are MySQL [15] and PostgreSQL [16] among others.

2.1.2 Columnar

Columnar data stores provide an easily distributed storage system while still providing users a simple data model. The data model typically provides structure in the form of Rows and Columns, essentially exposing a multidimensional map or tree in which to store data. A wide-column columnar data model first implemented by Google Bigtable [17] uses rows or "row keys" to index into the first (i.e., outer) map. The second (i.e., inner) map contains columns and column values which do not need to adhere to any schema, providing flexibility to the user and grouping columns together on disk. Other columnar technologies, such as Google Dremel [18], allow the enforcement of a flexible schema and differ widely in their implementation, yet still group columns together for persistence on disk, which enables extremely efficient data access.

Regardless of the system, domain modeling for columnar storage systems is non-trivial and is a research area investigated by the author [19]. Further work is needed to better provide tools and processes for adopting columnar systems in place

of existing relational-based applications. Distributed columnar stores, such as Cassandra [20], distribute data across a cluster of machines by row. This operation is often transparent to the user and done in a way that enables trivial expansion of the cluster as needed. Cassandra achieves high levels of availability by replicating rows and columns on many machines throughout the cluster. If a node is lost due to a failure, the data is available to be read from an active replica. It is common to set a replication factor when deploying such a system, which determines the number of replicas that must be maintained by the system. Often replication is handled without intervention from the user making columnar stores compelling technologies for organizations in need of "out of the box" scalability and availability.

Columnar systems have been deployed at Google [17] and Facebook [20] for many years but recently have experienced tremendous growth among small to medium sized enterprises. Google Dremel, released as Google BigQuery [21], has gained popularity for its analytics capabilities and has a popular open source implementation, Apache Parquet [22]. HBase [23] and Cassandra are open source implementations of wide-column stores which have both sparked an assortment of vendors seeking to ease complex transitions to columnar stores from relational data stores—the most prominent being DataStax [24], which has focused primarily on making Cassandra accessible to the enterprise.

2.1.3 Key-Value

Key-value technologies are typically implemented as distributed hashes. In contrast to columnar (i.e., wide-column) systems, key-value stores do not provide

the user with a data model abstraction on top of a simple map. Operations such as *get()* and *put()* are supported across these systems, with each providing enhanced API functionality depending on anticipated use. There are a variety of in-memory key-value systems including Redis [25], MemcacheD [26], and EhCache [27]. These systems distribute a hash of values across a cluster using memory as storage for quick access. Hashes can typically be configured to be persistent through a machine power cycle. If the data is too large to be stored in memory, a more appropriate, file system based technology is typically chosen. Systems such as Google File System [28], Hadoop [29], and Dynamo [30] provide the user with a persistent distributed hash. Google File System and Hadoop in particular expose a file system based API, similar to POSIX [28], which allows consumers to invoke create, read, update, and delete (CRUD) operations on files and folders, addressing data via paths. In the case of GFS and Hadoop, the file data split and placement is determined by a single Master node. All file access requests from clients are handled by the Master node that sends back location information for each split that the client can then use to request data directly from individual cluster nodes. Master nodes are often replicated in near real time to allow for "hot" failovers, should the Master node become unavailable.

Dynamo does not provide a file system abstraction, instead implementing a very simple *get()-put()* interface and focusing on scalability, availability, and durability. Unlike GFS and Hadoop, Dynamo does not rely on a single Master node to partition its data, relying on *consistent hashing* to partition data, which allows

for easily expanding a database across a cluster of machines. Dynamo allows any node within the cluster to be addressed directly to read or write data, routing clients to the correct host within one network hop. In this way, Dynamo strives to be "always-writable" or available, even while experiencing multiple node failures.

All of the described disk based key-value systems support out-of-the-box replication that is largely transparent to the user, providing a high degree of availability. By completely eliminating the ability to impose a data model, key-value (distributed file system) technologies provide the foundation for other large-scale data formats and stores, such as Apache Parquet and HBase, respectively. Unfortunately, the lack of a flexible domain model forces software engineers to store complex objects and relationships in unfamiliar and convoluted ways, often denormalizing object graphs and requiring the persistence of data in a way that anticipates how it will be accessed. Although this challenge is not unique to key-value stores, the lack of even the most basic support for a data model makes adoption of these technologies difficult for enterprises struggling to scale.

2.1.4 Document

Document data stores provide the user with the ability to model objects as documents. Document data stores often expose the binding of *fields* to primary types such as strings, numbers, and dates. This paradigm is familiar to anyone who has worked with an object oriented language and especially those who have previous experience with object relational mapping technologies (ORMs). While the type of document that each vendor supports differs (XML, JSON, BSON, etc.), they

typically require each document to provide a unique identifier when stored. They differ from relational databases by allowing flexibility at the field level of each document. In a relational database each row in a table must have the same number of columns as all the other rows present in the table. Document stores give the user the flexibility to persist documents (i.e., rows) in a collection that differs in the number and type of fields (i.e., columns) associated with each document.

In addition to providing a relaxed schema, document stores are often associated with information retrieval tasks. The APIs exposed by document stores are rich with query support. Lucene [31], a search engine technology, is one of the most popular open source implementations of a document store. It provides full text search, attribute filtering, and faceting over any set of documents stored in its index. It also provides out of the box support for text stemming, tokenization, stop word removal, and result scoring. The flexibility of Lucene has led to the development of two popular distributed search engines, Solr [32] and Elasticsearch [33]. While they differ in implementation, they both provide a highly available and scalable storage solution without relinquishing a familiar domain model and ad-hoc query support like other large-scale data stores. However, these features do come at a cost. Document data stores are not designed to handle very large (e.g., gigabytes) individual files like distributed file systems are and, as such, are not optimized for reading and appending large amounts of data from and to disk. This often makes them a poor choice for the underlying systems of batch processing frameworks such as MapReduce [34]. In practice, they can be problematic to scale

for large amounts of data and difficult to make durable. MongoDB [35] in particular has had significant scaling problems documented by the teams struggling to adopt this promising technology [36].

2.1.5 Network

Network based data storage technologies persist an interconnected graph of vertices and edges. A graph is an extremely expressive way to capture objects and relationships and allows for an easy domain model transition from a relational database because most of the available vendor solutions allow the addition of an arbitrary number of properties to any vertex or edge, otherwise referred to as a property graph. Network stores are most commonly used when the number of edges between nodes becomes unmanageable in any other storage technology or when the application benefits greatly from graph algorithms and theory (e.g., shortest path between two nodes). Many organizations use a network to model friend and follower relationships, allowing quick access to large numbers of friends or followers and making the extremely common "Do you know?" feature trivial to implement. However, network stores are typically poor at handling large files or vertices and edges that have a large number of properties associated with them. Scalability and fault-tolerance of these systems vary widely between implementations. Systems like Google's Pregel [37] are extremely scalable (e.g., thousands of commodity machines) and fault-tolerant. An open source implementation of Pregel is available as Apache Giraph [38]. The open source graph database Titan [39] layers a graph API on top of the columnar store Cassandra. Titan cleverly maps networks onto

Cassandra's data model, which allows Titan to take full advantage of Cassandra's high availability and scalability attributes. However, the most popular network store available to the enterprise is Neo4j [40], which can be made highly available through master-slave replication and scalable by the addition of more read-capable replicants.

All of the described technologies provide a wide array of desirable features for applications. Unfortunately, the research surrounding these technologies focuses primarily on storing and retrieving arbitrary *data*. While this is an important focus, it is limited from the perspective of the software engineer who has become accustomed to working with *objects and relationships*. Seldom does the software engineer decompose problems in terms of just *data*. A general lack of empathy for the software engineering perspective has resulted in numerous challenges that must be overcome in order for an application developer to properly benefit from this valuable set of technologies.

CHAPTER 3: Towards N Data Store Architectures

Although there has been an explosion of new technologies available to the software engineer in the last decade, significant challenges still remain. The distributed systems researchers who developed the systems described in the previous chapter were focused on issues concerning *scalability, availability, and durability*. While these issues are of extreme importance, they are not the *only* issues of concern to a software engineer tasked with making technology choices for their application.

3.1 Challenges for Software Engineering

Software engineers, as end users of large-scale technologies, often lack expertise in distributed systems. These large-scale technologies introduce *scalability, availability, and durability at the cost of complexity* for the end user, the software engineer. Although these system attributes are extremely attractive for any application developer, the complexity of adopting them may make "scaling up" or performance tuning more viable for an organization that lacks distributed system expertise. This is unfortunate, as "scaling out" has been shown time and again to better serve an organization in terms of cost and performance.

Software engineering organizations are required to make technology decisions frequently and one of the primary attributes these organizations evaluate when looking to adopt new technologies is *accessibility*. Many of the technologies

detailed in the previous chapter *lack a vision for how application developers will use them*. Recognizing what influences an application developer to adopt a technology is essential to creating an accessible data store.

Understanding features that motivate an application developer to adopt a particular technology is an emerging area of software engineering research. Empirical evidence is not available for all technology choices but much work has been done on programming language adoption. Meyerovich et al. found that open source libraries, existing code, and experience, largely determine language choice while intrinsic features such as performance, reliability, and simple semantics do not [41]. The features found to matter least in language adoption are broadly the focus of large-scale data stores. While we cannot empirically claim that the same features influence data store adoption as language adoption, we do contend that both are motivated by accessibility, which has not been the focus of large-scale data stores thus far. Features such as scalability, availability, and durability are as essential to data store technologies as performance and reliability are to programming languages, but they may not be of primary importance with regards to user adoption.

Hanenberg et al. [42]–[45] provide another example of accessibility motivating technology choices. Hanenberg and his colleagues focus on providing empirical evidence for claims on both sides of the "static vs. dynamic" language debate. Their work shows that there are clearly defined areas where it does and does not matter what type of language one uses to accomplish a development task.

Their work suggests that there are certain activities where static type systems have distinct advantages. The results show that static, in comparison to dynamic, type systems improve system maintainability [42], provide implicit design decision documentation [43], and improve the usability of unfamiliar APIs (even with the aid of a modern IDE) [44]. In spite of empirical evidence, dynamic languages are increasingly popular due to their accessibility. Proponents argue that type conversion is time-consuming and errors due to type inference are simple to correct despite research showing this to be invalid for systems over ten lines of code [45].

Regardless of the advantages promised by large-scale data stores, they must be carefully weighed against the complexities they will introduce. The advances in scalability, availability, and durability attack aspects of what Fred Brooks describes as the *accidental difficulties* of software engineering [46]. Although these attributes contribute to reducing some difficulties, they create a great deal of accidental complexities that distract the software engineer from focusing on *essential difficulties*. In his article, *No Silver Bullet*, Brooks identifies three "breakthroughs" in accidental difficulties that large-scale systems reintroduce during adoption.

Brooks notes advances in high-level languages will decrease complexities encountered during software development. Today, it is common for a large-scale data store to ship with its own domain specific language (DSL) or data access paradigm, which results in additional complexity for the software engineer as now they must gain an in-depth understanding of additional abstractions for each specific system they wish to introduce into their architecture. Examples include

SQL, CQL (Cassandra), MapReduce (Hadoop), Gremlin (Titan), and Cypher (Neo4J). Adopting an additional system results in intensive effort for the engineer, which deters them from choosing the best system for the task at hand even when the system offers a much-desired feature set.

The second accidental difficulty Brooks considers solved is time-sharing. Time-sharing, he states, "preserves immediacy". In a large-scale distributed system, resource coordination happens rapidly at varying levels of abstraction. However, at the layer closest to the software engineer, the processing paradigm for large amounts of data is batch processing. The most widely used large-scale system for data processing is currently Hadoop, which is an open source implementation of Google's MapReduce. MapReduce, and as a result Hadoop, is a distributed system that allows the user to harness the power of an arbitrary number of computers without having to understand the underlying complexities. Utilizing MapReduce affords an organization an incredible amount of data processing capabilities that has helped propel the widespread adoption of Hadoop. What many organizations do not comprehend is the trade-off in productivity they are making because Hadoop exposes its processing capabilities via a batch processing interface, which by default does not ensure proper time-sharing of the system by leaving submitted jobs in a queue until cluster resources are made available for processing. For this reason, even Google, the creators of MapReduce, have largely abandoned the framework, opting for more real time technologies [47] that facilitate more iterative development. Large-scale data stores often make it difficult for software engineers

to preserve in their minds the context of the problem they are solving; while batch jobs or long running queries are being scheduled and run over hours, days and—in extreme cases—months, the software engineer will likely move to other tasks only to be interrupted at a later point with job failures or incorrect results.

The third accidental difficulty cited by Brooks as solved is that of the unified programming environment. We have yet to see a body of work on this subject, but one can imagine a need to again solve this difficulty for the class of applications addressing "Big Data" challenges. A unified programming environment for large-scale systems will need to be designed from inception to work with heterogeneous systems that leverage a diverse array of languages and access patterns. Without research in this area, software engineers are forced to context switch frequently and gain in-depth understandings of the complete system. This results in immense levels of complexity for the software engineer, which, in turn, limits the quality and velocity of application development in comparison to homogeneous systems.

The software engineering researchers of the 1980s and 1990s worked diligently to reduce the accidental complexities involved in software development. They pioneered new languages, patterns, processes, and programming environments that allowed the software engineer to focus on the essential difficulties of software. As applications have moved from single install, desktop environments, to complex data-intensive infrastructures deployed across thousands of computers, research is needed to afford the same support of the past to the software engineers of the present. Only through the development of new tools and

techniques can the field of software engineering improve the accessibility of large-scale, distributed, heterogeneous systems of systems.

3.2 Opportunities for Software Engineering

This dissertation examines open issues in the area of software architectures for large-scale, heterogeneous data-intensive systems. Software engineering research is largely deficient in the study of emerging trends related to the development of highly-distributed SaaS applications composed of specialized data storage technologies, instead of "one-size fits all" systems. The lack of widely adopted and well-proven architectures and patterns in this area leads software organizations to make costly investments in both financial and human capital before pursuing technologies that will benefit themselves and their customers.

In the previous chapter, we provided an overview of existing technologies related to large-scale data storage, detailing the minimum knowledge required by a user for non-expert level utilization. These systems, without modification, typically require expert-level knowledge in distributed systems and a deep understanding of the data modeling capabilities of each. Data models afforded by these technologies even go so far as to force the denormalization of object graphs, which causes data duplication and the elimination of object references. This technique, among others, is foreign to software organizations that have previously relied on only relational data storage technologies and requires well-documented abstractions to increase system accessibility. Without research into the complexities encountered during the adoption of large-scale data stores, software engineers will continue to give up

accessibility and flexibility in exchange for scalability, availability, and durability. *The idea that software engineers can be afforded all of these highly-desirable features simultaneously serves as motivation for this work.*

By leveraging an understanding of available technologies, this dissertation will work to abstract away complexities inherent in the adoption of one or many large-scale data stores. Design patterns and architectures have been proven to aid in software engineering tasks by allowing the engineering team to efficiently communicate complex subjects without lengthy and distracting discourse [48], [49]. While there is a significant amount of peer-reviewed research available for small, object-oriented systems [50], much work is needed before a team of software engineers can effectively communicate about the issues and concerns that present themselves while developing with a variety of disparate data stores.

This dissertation develops and documents novel architectures and patterns to be applied by organizations that wish to adopt specialized technologies into homogeneous or existing heterogeneous environments. The use of these abstractions, alone or in combination with others will greatly decrease the risk required for an organization to adopt highly specialized frameworks and serve to increase the accessibility of these frameworks for all involved.

CHAPTER 4: Related Work

Much research has been accomplished by the distributed systems community to provide application developers with numerous, extremely capable technologies. Other computer science disciplines are now diligently working to contribute. Although the resources were available for the software engineering community to devote attention to, their focus was elsewhere. Software engineering has improved the accessibility of software development immensely in the past. Unfortunately, many of these advances are not directly applicable to large-scale heterogeneous systems without modification or extension. As detailed in previous chapters, the lack of architectural patterns, unified programming languages, and development environments for distributed, heterogeneous systems forces the software engineer to cope with immense accidental complexities.

Homogeneous "one-size fits all" systems afforded a set of constraints that made eliminating numerous accidental difficulties reasonable. The first desktop environments and object oriented programming languages were proposed decades ago and continue to be improved by the software engineering community to this day. With many common problems solved, the focus of the community has evolved beyond object oriented design patterns (e.g., the transformation of the OOPSLA conference into the SPLASH conference). The rapid growth in data has pushed database designers to innovate independent of the software engineering community. Michael Stonebraker first noted the move from traditional database management

systems by describing the death of the "one-size fits all" data store [51], [52], foreseeing that traditional relational systems—in use for over 25 years—would be replaced by a variety of specialized technologies. Stonebraker has since been involved in the creation of many well-known large-scale data stores including C-Store [14] and H-Store [53], which later became commercialized as Vertica [54] and VoltDB [55]. It was the distributed systems community that anticipated and responded to the seemingly ever-increasing amount of data software engineers are faced with today.

Abstracting away the complexities of working with multiple data stores is an emerging area of interest for both academia and industry. Scott Leberknight was the first from industry to attach a name to this challenge. In a blog post later popularized by Martin Fowler [56], [57], Leberknight describes *polyglot persistence*; he says, "Polyglot Persistence, like polyglot programming, is all about choosing the right persistence option for the task at hand" [58]. *This description captures the motivation for this dissertation.* Software engineers should be encouraged to use the appropriate technology for data storage just as they do with languages. However, similar to how introducing an excess amount of programming languages into an application stack complicates application development, so does the introduction of many data stores. In polyglot programming, researchers are developing systems to allow application developers to easily program in one to many languages within the same file with limited complexity [59]. Similar efforts—albeit in spirit rather than implementation—are required to address the complexities of

polyglot persistence. Although some efforts are detailed below, they largely result from work within the distributed systems community and focus on attributes other than those of accessibility and flexibility.

One approach to polyglot persistence simplification is through the use of a universal REST-based API. A REST-based approach to persistence across one or many database-as-a-service systems (cloud-based databases) was first proposed by Haselmann et al [60]. They acknowledge the growing variety of DBaaS vendors and APIs and stress the need for a common access standard based on REST. The goals of their proposed API are to be flexible, exchangeable, and comparable. They describe ways to attack these desired attributes by focusing on entity handling, schema definition, and querying. The API allows for the persistence and retrieval of any entity through reserved URIs such as "/schema" and "/queries" where the developer would register a series of schemas and queries to be enforced and used by the application. Attributes of stored entities can be retrieved through the use of XPath expressions under the reserved URI "/xpath". The paper does not provide an implementation of the API and, as stated by the authors, "...is particularly meant as a basis for further discussion of the general notion of a universal DaaS-API". Aside from the lack of API implementation, the work does not discuss some very complex issues. In particular, developing a strategy for the decomposition of an object graph for storage in relational and non-relational systems is non-trivial and critical to successful adoption of the proposed API. Additionally, the focus on schema development and enforcement as a primary aspect of the API is a poor choice for the

purposes of this dissertation: many large-scale data stores lack schemas and are able to provide high degrees of flexibility because of it. Additionally, the notion of ad-hoc querying and query languages beyond those specified in the paper is largely ignored, forcing the user to register queries with the "/queries" endpoint before being able to execute them and only supporting SQL as a query language.

Extending upon the universal REST-based API proposal of Haselmann et al. is the ORESTES system by Gessert et al. which provides a practical implementation of a unified, REST-based storage and access API for four popular NoSQL data stores (Versant, db4o, Redis, and MongoDB) [61]. While a primary motivation for the system is improving network latencies for applications accessing multiple DBaaS systems, the ORESTES middleware also provides a REST-based implementation for accomplishing polyglot persistence. The resulting API appears to expose an interface similar to Haselmann et al. but with a variety of additions, including caching and support for transactions. In a subsequent paper, they detail their efforts towards a unified REST API for DBaaS and briefly describe the notion of a Polyglot Persistence Mediator, a system component that "routes data to different storage backends based on declarative SLAs." The prototype implementation described in the paper is capable of routing entities to either the MongoDB and/or Redis data stores. Of interest is the authors' choice to allow the user to specify field-level boundaries on latency, availability, and replication factors in a declarative way. Issues concerning latency are certainly important when working with a variety of Cloud providers and allowing thresholds to be set in a

declarative way makes these otherwise non-trivial guarantees largely transparent to the user. While ORESTES is a significant advancement over the work proposed by Haselmann et al., it reflects some critical design choices that make it difficult to extend for the purposes of this dissertation. First, increasing the accessibility of polyglot persistence for the software engineer is not the primary motivation for the system. As such, the system designers developed an independent system that must be deployed and maintained in addition to the application and data storage systems, thereby adding to the complexities already present when adopting multiple data stores and adding additional overhead in the case of cache misses during entity retrieval. The authors' implementation of a unified REST-based API for cloud data stores ignores the need for an abstraction of the querying languages of each data store, leaving it up to the user to understand the intricacies of each. Similar to the schema constraints imposed by Haselmann et al., the ORESTES system enforces schemas on schema-less data stores, while going even further to provide transactions on top of systems that lack transaction support. While their prototype looks promising, again, the primary system motivator is SLA fulfillment, not end user accessibility of polyglot persistence.

Polyglot persistence not only introduces complexities associated with data persistence but also with data access (e.g., retrieval and querying). As discussed by Gessert et al., one approach to making the retrieval of data from a variety of systems more accessible is to automatically route access requests to the "best" store, based on some set of criteria. One system developed by Lim et al. focuses

specifically on the implications of polyglot persistence in relation to query support [62]. Their system, Cyclops, is a concrete implementation of an approach they refer to as DBMS+ that focuses on providing unified query support for Esper, Storm, and Hadoop. The choice of these systems is interesting because of the mix of stream processing and batch oriented interfaces. When the user submits a query to the DBMS+ system, it is routed to the best system for the task. The researchers accomplished automatic routing by developing a query language that allows their system at run-time to choose between available query processing frameworks. As with ORESTES, the user is able to declaratively register execution requirements with the system. Doing so allows the user to easily describe what performance they require without implementing complex system logic. The authors' also take an "imperial" approach to implementing their query API, which they believe gives them an advantage over a federated approach by providing "full control of what gets executed and how". A federated approach, in contrast to an imperial approach (i.e., direct execution engine access), makes use of the storage system-specific query language, possibly introducing errors during query translation. An imperial approach to polyglot persistence—if imperial access were provided by all vendors—is extremely interesting and, as shown by Lim et al., could greatly enhance accessibility to application developers while decreasing the effort needed to access many data stores simultaneously. A limitation of the imperial approach is that offering an expert user extension points into an underlying system becomes difficult as a result of bypassing the system specific query language.

While the work discussed here is an important first step toward providing abstractions on top of the inherent complexities of working with heterogeneous systems at scale, the work contributed by this dissertation offers a different approach. *We are not primarily motivated by issues of performance or the creation of a new language or system, but by the need for expressive patterns that eliminate the tight coupling of an application to its data stores.* As far as we are aware, we are among the first members of the software engineering community to not only acknowledge the lack of accessibility provided by these technologies, but to actively work towards rectifying it. To do so, we leverage previous work in the area of enterprise architectures and patterns for one-size-fits-all systems, and work to adapt them to polyglot persistence systems. The contributed work differs from others described in this chapter by focusing on limiting the accidental complexities of adopting these technologies, not only the practical matters associated with system performance. We believe this approach results in greater accessibility of these technologies, allowing software engineers to focus on the essential difficulties of software development.

CHAPTER 5: Evolution of the Persistence Tier

The development of new persistence architectures for aiding in the utilization of specialized persistence technologies is needed to encourage software engineers to focus on the non-functional attributes of most importance to their application. A software application typically fulfills the functions of capturing (i.e., collecting and storing), understanding (i.e., analyzing), and presenting (i.e., reporting) data (see Figure 5-1). To satisfy each function, application technologies must be chosen carefully. Technologies provided by distributed systems researchers are designed and evaluated with scalability, availability, and durability as primary drivers, leaving the application developer to choose from a set of complex technologies to fulfill each function of the application.

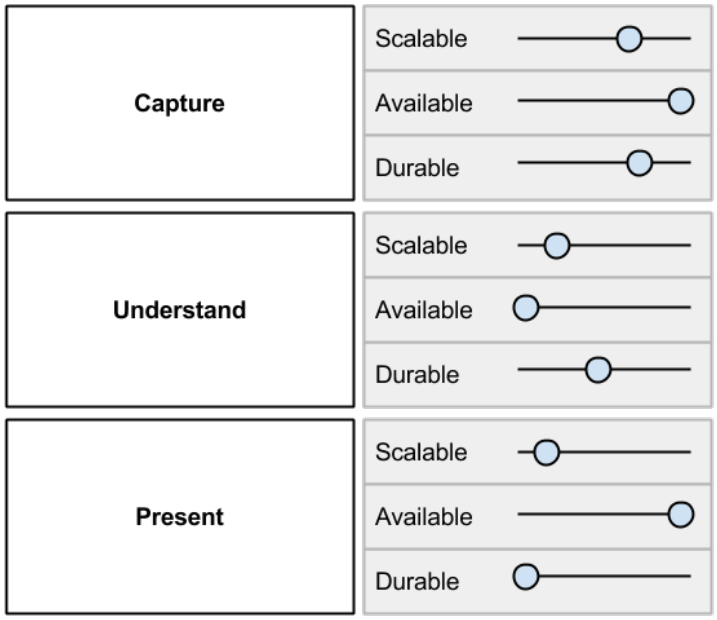

Figure 5-1. Application functions with associated technology trade-offs.

In addition to scalability, availability, and durability, we propose that software engineers make technology choices based on *accessibility* and *flexibility*. As discussed in Chapter 3, software engineers often choose technologies based on these two attributes. This behavior follows as a practicing software engineer is less a scientist than engineer, primarily concerned with task completion rather than pure scientific pursuit.

The movement away from one-size-fits-all data stores and towards specialized data stores has allowed applications to adapt to significant increases in data and usage. However, adopting a heterogeneous architecture can be overwhelmingly complex for an application developer. Each specialized system provides the end user with a different set of data storage and access APIs. In addition to mastering a variety of interfaces, the application developer must also understand how to modify the data model for acceptance by each system, which requires a deep understanding of the underlying distributed system. To successfully use many of these systems, the software engineer must move past the tools of object oriented analysis and design, and force themselves into the realm of the system designer whose focus is on performance and storage of the data, not necessarily providing accessible APIs.

This dissertation details the development of an architectural pattern and associated framework that provides accessible and flexible polyglot persistence. This work extends the abstract concept of polyglot persistence, which encourages the use of many specialized data stores, by providing an expressive and extensible

implementation. Pairing polyglot persistence with extensions to enterprise architectural patterns allows for the encapsulation of logic related to persisting and retrieving objects and relationships from one to many data stores while also decoupling the application from data store specific implementation details. Pairing in this manner enables the adoption and abandonment of data stores without the need for the entire development organization to gain expertise in the intricacies of each data store utilized throughout the architecture.

5.1 Traditional Enterprise Architecture

A traditional enterprise architecture will isolate application code (e.g., controllers, validators, etc.) from business logic via a service tier [63]. The service tier, in turn, leverages a Repository pattern to manage persistence details. Repositories encapsulate all knowledge related to persisting and retrieving objects and relationships to and from data stores. Repositories typically accept and return data models, isolating the application from the details of the underlying persistence framework (e.g., JPA, JDBC, ODBC, ActiveRecord, etc.). The Repository pattern, shown via sequence diagram, is given in Figure 5-2.

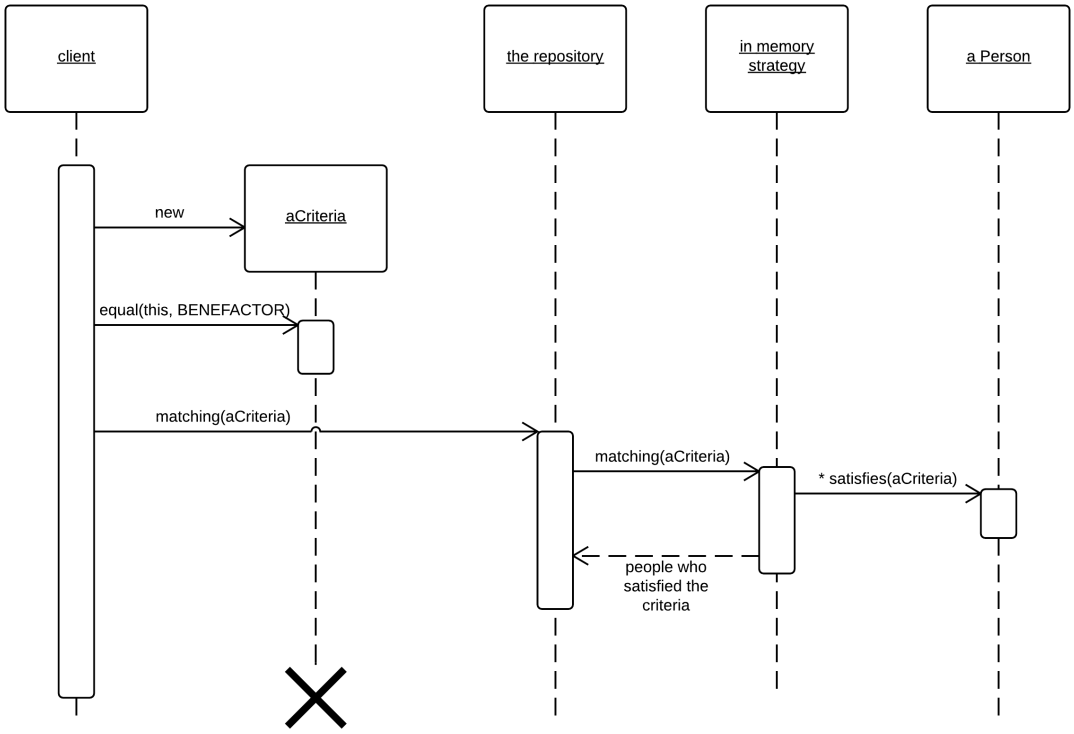

Figure 5-2. Adapted from the Repository pattern as described by Fowler

In Fowler's representation [64], [65], the client interacts with the Repository to persist and query the underlying storage mechanism which could range from an in-memory collection to a database or distributed data store. The Repository exposes interfaces to the client that accept and return data models, masking the details of data mapping and allowing the replacement or extension of persistence technologies without forcing changes through dependent code paths. It is common practice to implement a Repository per domain object. An example of a homogeneous enterprise architecture, utilizing a service and persistence tier (i.e., a three-tier or multi-tier architecture), is shown in Figure 5-3.

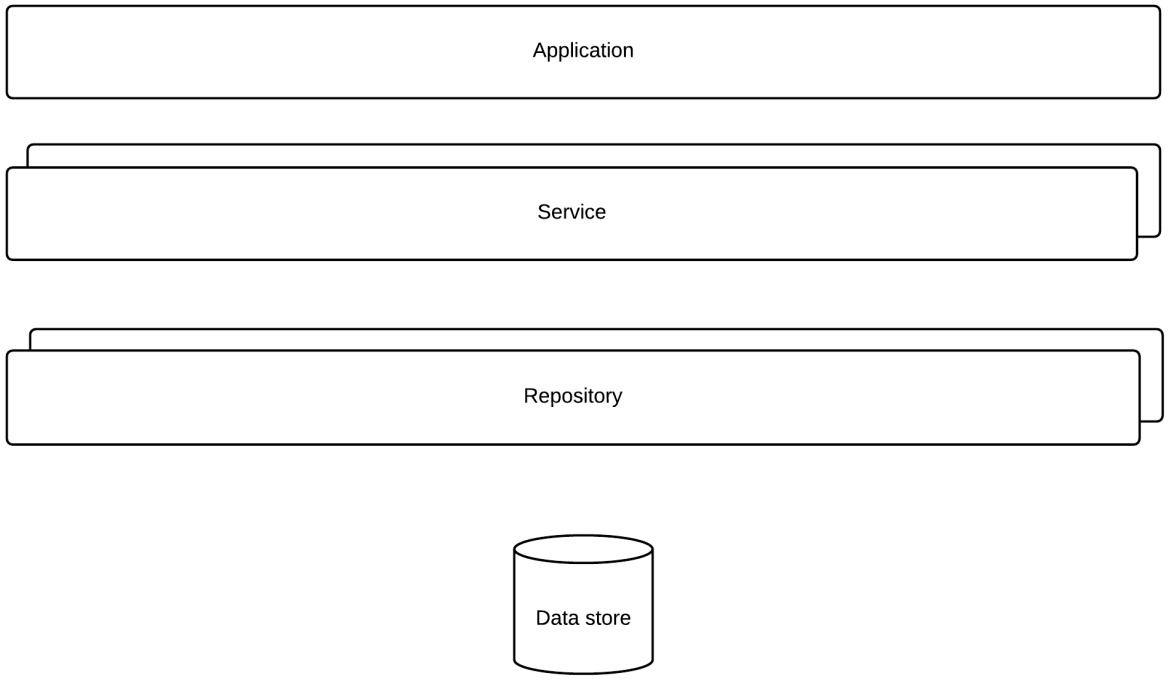

Figure 5-3. A traditional homogeneous enterprise architecture.

In the above architecture, each Repository only interfaces with one persistence technology such as a relational database. This is an accepted initial architecture for SaaS applications as it provides the application with isolation of concerns at each level of the software stack, and allows the developer to pass a domain model from tier to tier similar to the Model-View-Controller [64] concept, also widely used throughout SaaS applications.

5.2 Traditional Multi-Data Store Approach

As an application matures and requires the use of a specialized storage system, the persistence or service tier must be augmented to interface with multiple persistence technologies. Adding additional responsibilities to Repositories or

Services achieves varied results. This approach primarily violates the single responsibility principle [66]. As detailed previously, the service tier is designed to handle business logic and the Repository is primarily concerned with the details of persistence. This approach to polyglot persistence is provided for reference in Figure 5-4.

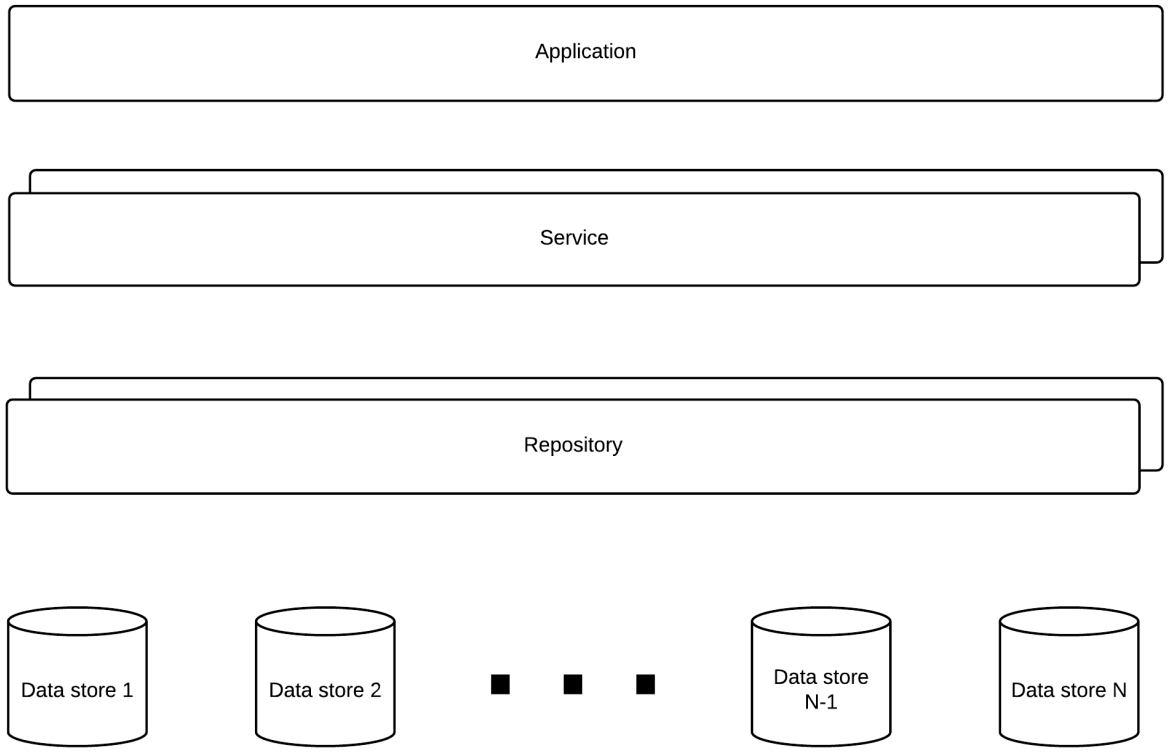

Figure 5-4. Ad-hoc polyglot persistence.

While this approach can lead to successful polyglot persistence, as experienced by Project EPIC [19], [63], the onus is on the software engineering team to understand the nuances of each underlying data store and associated persistence framework. Although the above architecture isolates the application from the complexities of persistence among multiple data stores, it tightly couples the

Repositories or Services to more than one persistence framework, which makes working with multiple data stores brittle and difficult to adopt or abandon. Additionally, this approach has the undesired side effect of putting a great deal of intelligence into a single Service or Repository, depending on where the developer chooses to embed this behavior. Each Service or Repository will be responsible for understanding how to convert data models, interact with multiple stores, and reconcile differences among data stores. While this approach may be maintainable for a small number of data stores, it does little to mitigate the complexities associated with implementing a large number of data stores. Combining all of these responsibilities into otherwise straightforward components is a clear violation of the single responsibility principle.

5.3 Traditional Persistence Architecture Design Constraints

It is the goal of this dissertation to enable the adoption of any number of stores without vast increases in complexity for the software engineer. The number of specialized systems available to application developers is likely to continue to increase, not decrease, and the software engineer must be afforded the tools to mitigate the inherent complexities of adopting such an architecture. To accomplish this, the enterprise architecture pattern must be enhanced to take advantage of polyglot persistence. As the isolation of data mapping through persistence frameworks already occurs within the persistence tier, this tier provides a reasonable point of extension. What is required of this extension is the ability for the persistence tier to continue to accept and return a data model regardless of the

underlying storage system. Doing so will allow applications to adopt polyglot persistence without incurring the cost of the redevelopment of their existing Services, thereby increasing accessibility and providing the application with the flexibility of multiple, possibly redundant, data stores. Designing, implementing, and maintaining such an architecture is non-trivial and fraught with challenges.

Satisfying the need for the persistence tier to accept and return a data model to its client—required to provide accessibility—is not straightforward when implementing polyglot persistence. When using one data store, maintaining this contract between the client and the Repository is trivial. There is one strategy for data model decomposition and reconstitution provided by the application developer or the underlying persistence framework. Extending this contract across multiple persistence frameworks becomes complex as each data store exposes its own strategy for data mapping. Numerous systems enforce schema and data normalization (e.g., relational data stores) while others encourage denormalization (e.g., columnar, key-value). A reasonable abstraction of how each data store requires its data model to be formatted for effective storage is shown in Table 5-1.

Class	Data Model	Backing Collection	Interface
Relational	Table	Array[row][column]	add(), remove()
Columnar	Row	Map.Entry<[row key], Map<[column name], [value]>>	put(), get(), del()
Key-value	Hash	Map<[key], [value]>	put(), get(), del()
Document	Document	Map<[field name], [value]>	put(), get(), del()
Network	Vertex, Edge	Map<[property name], [value]>, Map<[from id], [to id]>	put(), get(), del()

Table 5-1. Collection based representations of data store data models.

Without proper isolation, these data mapping strategies are implemented in an ad-hoc manner, blending separations of concern and limiting knowledge reuse. To decrease the complexities associated with data model transformations, it is helpful to abstract these details into reusable components, decoupling them from the Repository. To accomplish this abstraction, each component must understand the type of data store being utilized when performing domain model decomposition. While relational data stores may be more straightforward due to technologies such as object relational mappers, similar technologies are not provided by all data stores. Understanding the analogous implementation of a primary key or query criteria in a columnar, key-value, document, or network data store is difficult and dependent on use case. It is common to use complex object identifiers in non-

relational stores that can communicate a great deal more information than a simple incremental counter. In Project EPIC (Empowering the Public with Information in Crisis), the composite key used to store data in Cassandra is a combination of a number of factors chosen by its application developer including a social media search term, the current date, and a fragment of the MD5-hash of the content being stored, all delimited by colons (e.g. "flood:2015106:a"). In addition to providing a great deal of information about the content being stored, the Project EPIC key, through the hash fragment, also determines how data is distributed among the cluster for well-balanced storage. By slightly modifying an otherwise static row key with a dynamic hash fragment, hot spots are avoided because a slight, but predictable, variation in row key forces Cassandra to place the key throughout the cluster instead of on a single node. Creating an abstraction capable of adapting to such a use case is indeed difficult.

The next chapter introduces and elaborates on the Diamond architectural style for polyglot persistence. This architectural style [67] has been developed to represent how the persistence tier of a modern enterprise application should work to reduce the amount of work required to adopt an unknown number of data stores. The Diamond architecture provides a set of abstractions that require the application developer to construct the persistence tier of their application in an extensible way, allowing the adoption or abandonment of data stores while also isolating any necessary code modifications from the rest of the application.

CHAPTER 6: The Diamond Architecture

This chapter introduces the Diamond architecture for polyglot persistence. *The Diamond architecture is a set of abstractions developed by the author that, when utilized in combination, greatly reduce the level of effort required to extend the persistence tier of an application to interface with an unknown or varying number of data stores.* The Diamond architecture is designed to not only augment a traditional persistence architecture but to further serve as an archetypal style during application inception.

6.1 Overview

Traditional enterprise applications rely on persistence tiers that interface with a single data store type. It is the responsibility of that data store to serve as the one-size-fits-all solution for the persistence needs of the application. Although the data store itself may provide mechanisms for scalability, availability, and durability, the persistence tier is often architected to interact with one and only one data store. *The Diamond architecture recognizes the need to design for an unknown or varying number of data stores and emphasizes the concepts of accessibility and flexibility for data store utilization.* Designing the persistence tier to interface with 1 to N data stores not only allows the application to easily adopt new and highly desirable data store technologies, but also provides the application with the ability to incorporate scalability, flexibility and durability within the persistence tier.

The Diamond architecture encourages the reuse of the existing traditional (i.e., three-tier architecture) architectural components, honoring contracts already established between the application, service, and persistence tiers. A traditional enterprise persistence architecture can be adapted to a Diamond architectural style by, at a high level, leveraging two novel abstractions within the persistence tier. Transitioning to a polyglot persistence architecture through the Diamond architecture is done additively, in an effort to limit changes to existing code paths.

This dissertation describes the design and implementation of a polyglot persistence architecture in its entirety. As described in the previous section, this is a non-trivial task, but one that has the potential to make a significant contribution in the area of software engineering for large-scale, heterogeneous data-intensive systems. The rest of this chapter describes the design of such an architecture.

6.2 Approach

We are beginning with the popular enterprise application architecture, or three-tier architecture, to form our initial design constraints [68]. In this approach, an application is composed of an application tier, a service tier, and a persistence tier. Often the persistence tier is implemented as a Repository pattern that is responsible for interacting with an external data store (e.g., a relational database). Data is passed between tiers using a rich data model that can be easily mutated along the way. Communication between the application and the data store is done through a persistence technology (e.g., JDBC, ORM, etc.). Applying these constraints to our design forces the architecture towards acceptance of a rich data

model while additionally working within the constraints of existing method contracts provided by Services. The ideal design would work within these constraints, requiring very few architectural changes. The primary design constraint is that of a single point of entry and egress into the Service (e.g., the application tier expects a traditional interface for persistence, regardless of the number of underlying data stores). A successful design will allow for this constraint but also easily adapt to an unknown number of data stores.

We begin with a naive approach to polyglot persistence provided by the traditional enterprise architecture. To adapt to multiple data stores through this approach, each data store is communicated with directly through complex Services or Repositories, as shown in Figure 5-4 of the previous chapter. A slightly less naive approach is to develop one Repository component per data model destined to be stored in a particular data store.

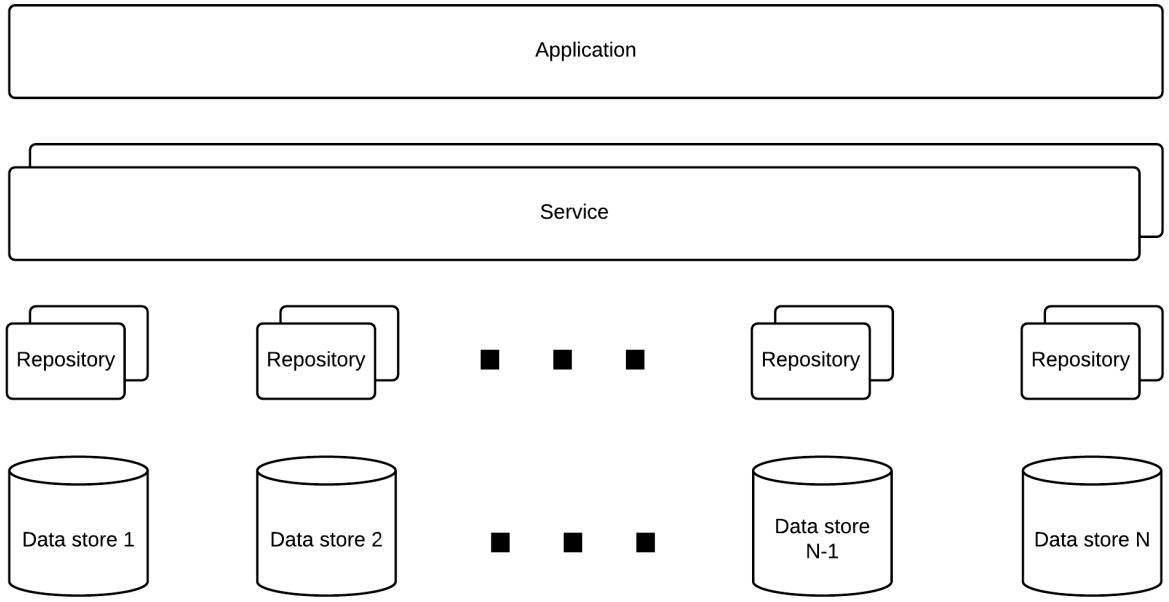

Figure 6-1. Repository per domain object per data store architecture.

Implementation of a more informed architectural approach presents three distinct challenges. First, the Repository pattern, as described by Fowler, ensures isolation of persistence from the client but requires an increase in intelligence to properly route data between different data stores. Abstracting this intelligence in a way that maintains existing contracts between the persistence and service tiers, while preserving the desire to keep data store-specific persistence logic decoupled and straightforward, is difficult. Second, data model decomposition and reconstitution is required to persist to and read from multiple data stores with varying data models. Third, invoking multiple Repositories incurs the cost of needing to handle multiple results, which can be complex. An ideal architecture for polyglot persistence would provide reasonable solutions for each of these challenges while also allowing extension in areas where the end user may have unanticipated

needs. Each of these challenges, as addressed by the Diamond architecture, will be discussed, in detail, throughout the remainder of this chapter.

6.3 Diamond Architectural Components

6.3.1 Command

The Command pattern is documented by Gamma et al. [50] as a behavioral design pattern. Due to its broad applicability, the Diamond architectural style has been influenced directly by the Command pattern. The Command pattern is shown in Figure 6-2.

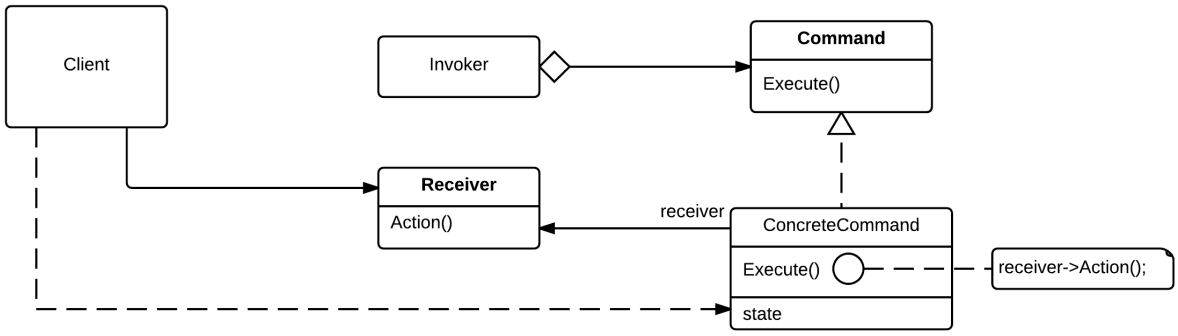

Figure 6-2. Command pattern [50].

The Command pattern is primarily used to isolate behavior needed by a class for future execution. A Command, as defined by Gamma et al., is responsible for implementing a single method, *execute()*, which has no return value and, when invoked, performs a discrete unit of work. The interface is provided in Figure 6-3.

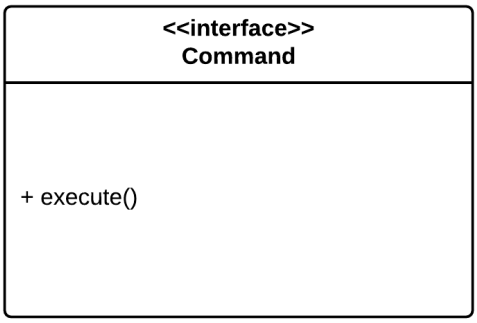

Figure 6-3. Conventional Command interface.

In this way the Command pattern creates a loosely coupled processing element that holds state. The Diamond architecture makes use of this pattern to encapsulate typical persistence scenarios such as CRUD operations in concrete Command implementations. Using Commands in this manner forces loosely coupled and highly reusable persistence logic throughout the application. For the purposes of polyglot persistence it is necessary to enhance the signature of the Command interface. The Diamond architecture recommends the Command interface shown in Figure 6-4.

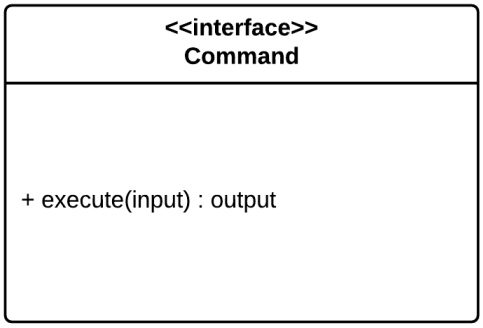

Figure 6-4. Diamond Command interface.

The Command pattern recommended by the Diamond architecture affords the ability to accept an arbitrary input and return an arbitrary output. Although not required, this small change is recommended in an effort to avoid forcing the Command to hold the state of objects (input and output) that will likely change with each invocation. Adding input and output to the interface allows the Command invoker to pass context sensitive input to the Command and have the Command respond with context sensitive output. Enabling this behavior at the method level, instead of during object construction, eliminates the need to construct a new object as input changes. In this way, many Commands with distinct behavior can be allocated only once, referenced later, and invoked as needed.

6.3.2 Runner

The Runner architectural component is responsible for invoking an arbitrary set of Commands. The Runner accepts a collection of Commands and then invokes them based on its defined behavior. The Runner is meant to encapsulate the manner in which Commands are executed. This responsibility keeps the Runner ignorant of the capabilities of the Commands and leaves it to focus on how to execute an arbitrary series of Commands. The Runner interface is straightforward and requires that the implementer accept a collection of Commands and input to pass to each Command during its execution. Running multiple Commands will produce multiple results; therefore, the implementer is additionally required to return the total set of results occurring from Command execution. The Runner interface is shown in Figure 6-5.

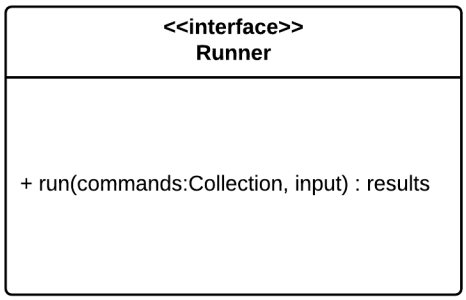

Figure 6-5. The Runner interface.

In practice, it would be typical to provide, at a minimum, the ability to run Commands consecutively and concurrently. Other interesting Runners are described in Chapter 7. *The Runner fulfills the role of the Invoker as described by Gamma et al.* Encapsulating the details of invoking a Command enables code reuse in complex scenarios such as the concurrent execution of Commands.

6.3.3 Data Model Converter

The data models provided by each data store vary greatly among data store implementations. To effectively persist and retrieve data from each store, data models passed from the application or service tiers require modification. Data models must be *forward converted* in preparation for persistence by a data store. During forward conversion, the original data model may lose fidelity via attribute removal or be transformed in other ways to adapt to effective storage by each store. *Backward conversion* is the forward conversion process in reverse, where an attempt is made to return a data model to the client that may have been pieced together from multiple underlying data stores. Isolating this responsibility from

the Repository allows the Repository to remain focused on delegation and persistence details. The Converter interface is implemented by the user, likely through callbacks, which decouples the rest of the architecture from domain model conversion. Ideally this conversion logic is reused throughout the application where required. Issues of data model versioning and migration will need to be addressed during implementation. As discussed by Stonebraker, in the future, all but the most complex data transformations might be handled without the need for a programmer through reuse of common use cases or integrations with external tools [69]. Sensible default Converters will provide the ability to convert a reference based data structure to an embedded structure and vice versa.

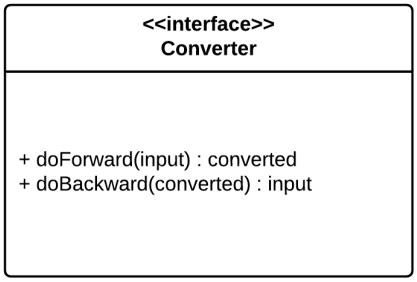

Figure 6-6. The Google Guava Converter interface.

For the sake of reuse, the Converter interface shown in Figure 6-6 is that of the Google Guava framework [70]. This interface has been found by the author to be well designed and applicable to many usage scenarios, such as persistence in the case of the Diamond architecture. Another Converter that provides the functionality to forward and backward convert data could certainly be used, but for the purposes of this dissertation, we will use the interface provided by the Google

Guava framework. To justify our use of the Converter, it is helpful to describe an example. Given a data model where a User owns many Documents, we will illustrate the conversion process in Figure 6-7.

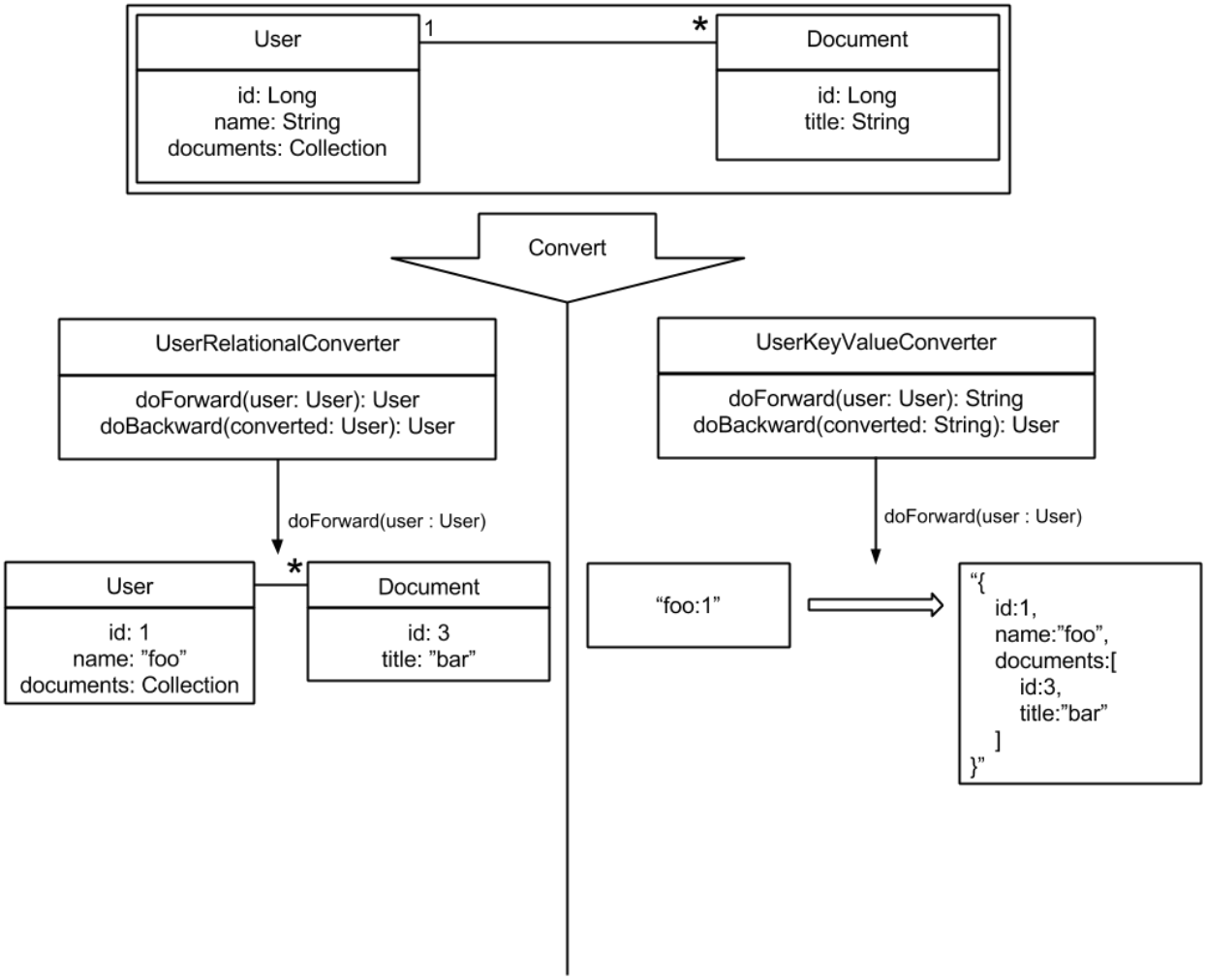

Figure 6-7. An example data model conversion.

In the example above (Figure 6-7), the User data model is passed to two Converters that convert the data model for storage in relational and key-value data stores. The UserRelationalConverter performs no operation in this example, assuming the underlying persistence framework will accept the User data model as is (e.g., via the use of an object-relational mapper). The UserKeyValueConverter

implementation converts the data model from a reference-based structure to an embedded document model representation that can easily be persisted to a key-value data store. To accomplish this conversion, the UserKeyValueConverter creates a composite key from the name and ID of the User and serializes the full data model to JSON to store as a value. Although the actual behavior of each converter will differ by data model and use case, this component attempts to isolate these details from the rest of the architecture. For this example, the backward conversion behavior would perform the analogous operations in reverse. The UserRelationalConverter would pass the returned object graph from the object-relational mapper back to the service tier that requested it. The UserKeyValueConverter would perform the forward conversion process in reverse, using the key to create a User object with the appropriate ID and name and the information in the embedded documents attributes to recreate a collection containing a Document with the correct ID and title. The resulting object graph would match the one that was originally handed to the Converter for the forward conversion process. The important lesson here is that these types of conversions can be codified to handle a significant number of real-world data models and that straightforward extension points can be added to default Converters to allow software engineers to handle additional object models that require customization to be converted accurately.

6.3.4 Reconciler

Invoking more than one Command will produce more than one result or results set. A core principle of the Diamond architecture is allowing an application to adopt additional data stores without forcing contract changes further up in the application stack (e.g., the service and application tiers). To achieve this, the persistence tier must "fan-out" using a Runner to interact with multiple data stores and "fan-in" before returning a result to the greater application stack. To accomplish this fan-in behavior, the architecture provides for a Reconciler component that is responsible for this behavior. A Reconciler is required to make decisions about how to map the multiple results produced by invoking a number of Commands into a single result. The interface for a Reconciler is shown in Figure 6-8. The Reconciler *reconcile()* method accepts a list of results and is required to return a single result.

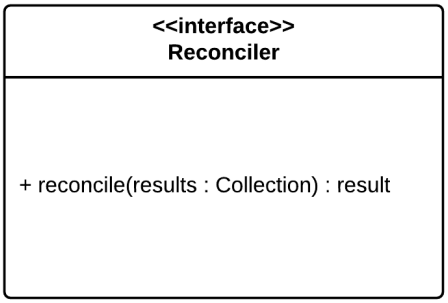

Figure 6-8. The Reconciler interface.

In practice, a Reconciler will have a variety of context sensitive behavior. It may be favorable to apply an order of precedence when multiple results have been returned by multiple data stores (e.g., take from relational first, columnar second,

key-value third, etc.). It is also possible to perform a merge of all the results, forming a composite result that is a combination of attributes stored across disparate data stores. These are just a few of the possible use cases for such a construct. Although similar to the Reduce described in MapReduce, the Reconciler is meant to force the transformation of many results to a single result, not apply an arbitrary transform to the results.

6.3.5 Pipeline

In the context of the Diamond architecture, a Pipeline is a construct that has a single point of entry and egress, much like a Command or Service, and is typically responsible for maintaining a collection of Commands. *The Pipeline is, as described by Gamma et al., the Client for the Command pattern.* In the Diamond architecture, this is largely an implementation detail rather than a constraint. Because it is common for Commands to not hold state in our architecture, they may be created anywhere in the application and added to a Pipeline at any time during the application lifecycle. A Pipeline is responsible for implementing a single method, *run()*, which begins execution of the processing elements it contains. However, it will also likely provide the ability to add Commands to its state. Figure 6-9 provides the high-level Pipeline interface.

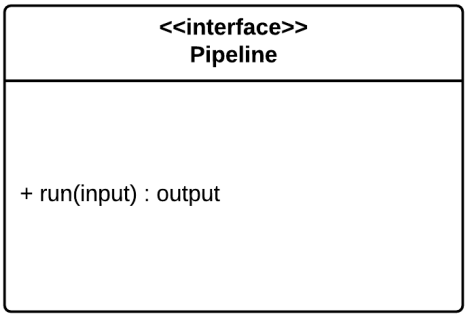

Figure 6-9. The Pipeline interface.

The reader will note that the definition for a Pipeline and a Command are nearly identical. This is intentional. The Pipeline is a container of Commands that similarly accepts input and returns output. In practice, this constraint is used to force the Commands present in the Pipeline to accept and return the same output as the Pipeline.

Adding onto the Pipeline construct, we define the DiamondPipeline. A DiamondPipeline is a Pipeline that requires a Runner and a Reconciler, which provides the "fan-out" and "fan-in" behavior that lends the Diamond architecture its name. The conceptual DiamondPipeline architecture is depicted in Figure 6-10.

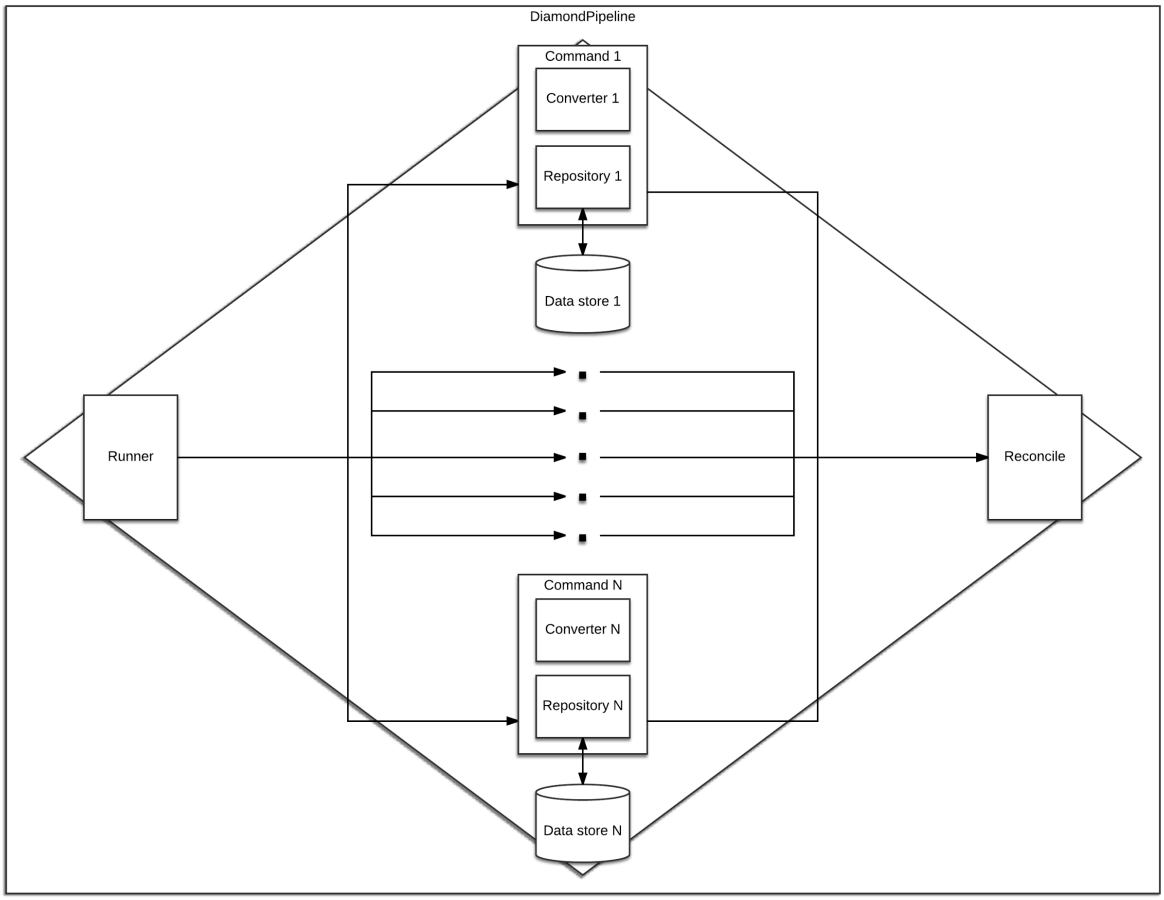

Figure 6-10. The DiamondPipeline.

As can be seen in Figure 6-10, the DiamondPipeline is composed of a Runner, a collection of Commands, and a Reconciler. Once defined, the DiamondPipeline can be used throughout the application, encapsulating all the necessary behavior to interact with any number of data stores. Although a Pipeline can be executed directly by a Service in the service tier, it may still be helpful to provide a Repository that contains many Pipelines.

6.3.6 RepositoryFacade (Optional)

To extend the Repository pattern to accommodate polyglot persistence, the existing Repository in use by a Service must take on additional responsibilities. Ideally, the polyglot Repository adheres externally to the responsibilities imposed by Fowler but internally performs similarly to a Facade pattern [50] thus, wrapping the complexities associated with delegating persistence tasks. The client (i.e., service tier) invokes methods on the RepositoryFacade, which, in turn, delegates to the Pipeline that interacts with traditional Repositories through Commands developed for each underlying class of data store as in Figure 6-11.

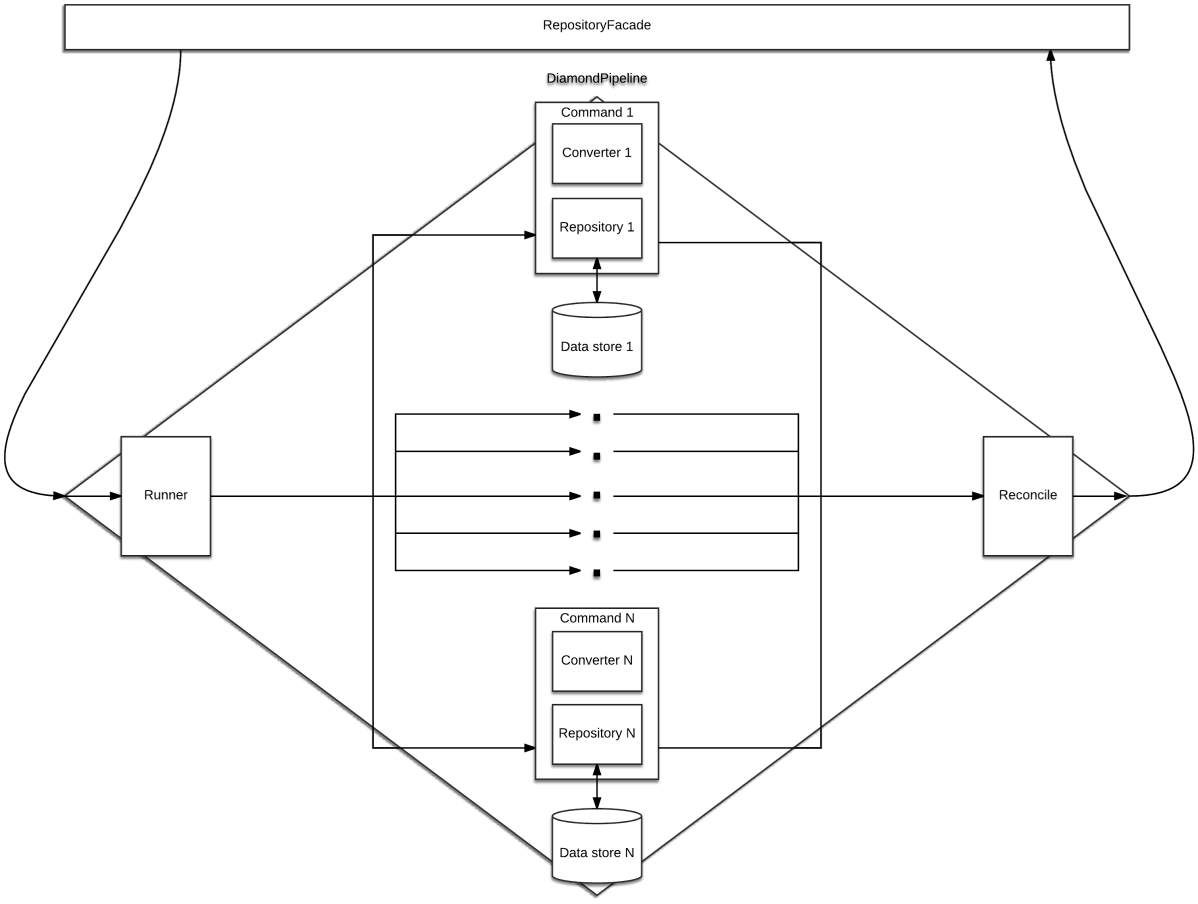

Figure 6-11. DiamondPipeline with RepositoryFacade.

This implies the creation of a Repository per data store class (e.g. relational, columnar, key-value, document, and network). Each sub-Repository would be configured with the specific persistence framework for each data store in use. In practice, the configuration of these Repositories will likely occur via an inversion of control framework (a.k.a. dependency injection), alleviating the need to instantiate and configure Repositories, Commands, and Pipelines before each use.

6.3.7 Final Architecture

The design of an architecture for polyglot persistence involves the extension and creation of non-obvious, architectural patterns within the persistence tier. The motivation behind the Diamond architecture is the desire to maintain a simple, data model-based persistence contract between the service tier and the greater application. Doing so allows business logic to be isolated within the service tier and persistence details to be confined to the Repository. To the greatest extent possible, the service tier is not to be concerned with the details of persistence, for a single data store or multiple data stores. Through the use of Converters, the data model of the application can be converted for use by a variety of stores without complicating the delegation responsibilities of the RepositoryFacade. The final Diamond architecture is provided in Figure 6-12.

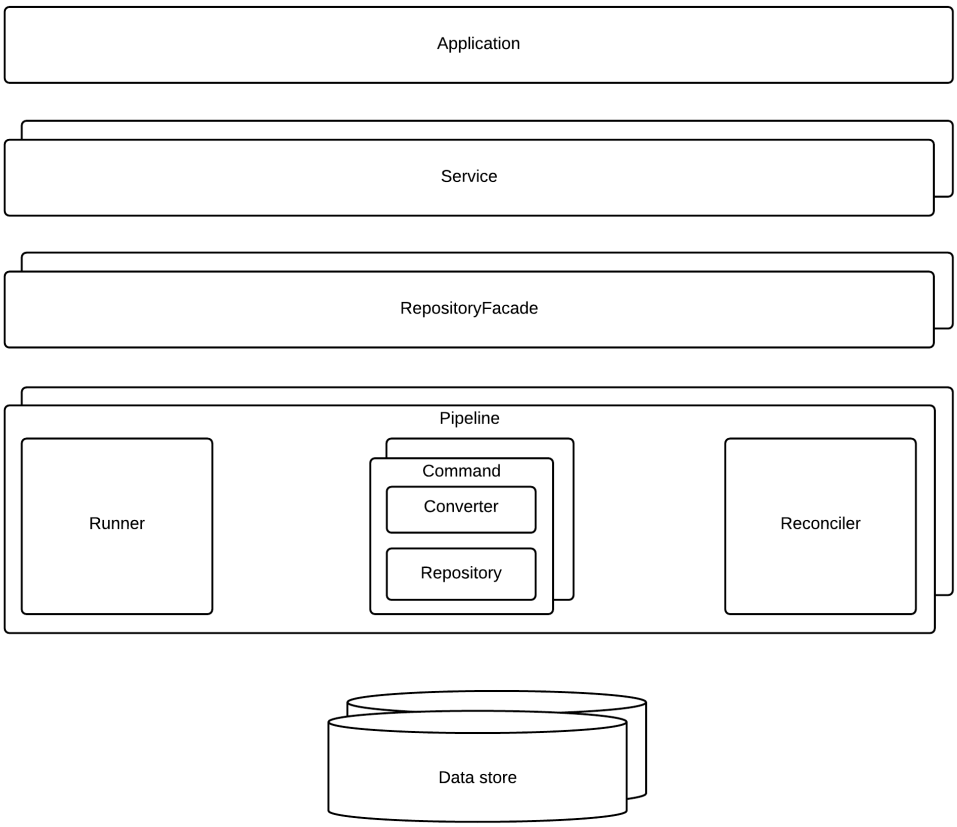

Figure 6-12. The Diamond architecture.

To provide a clear understanding of the common communications between the architectural components, a sequence diagram describing the interactions of each component is shown in Figure 6-13.

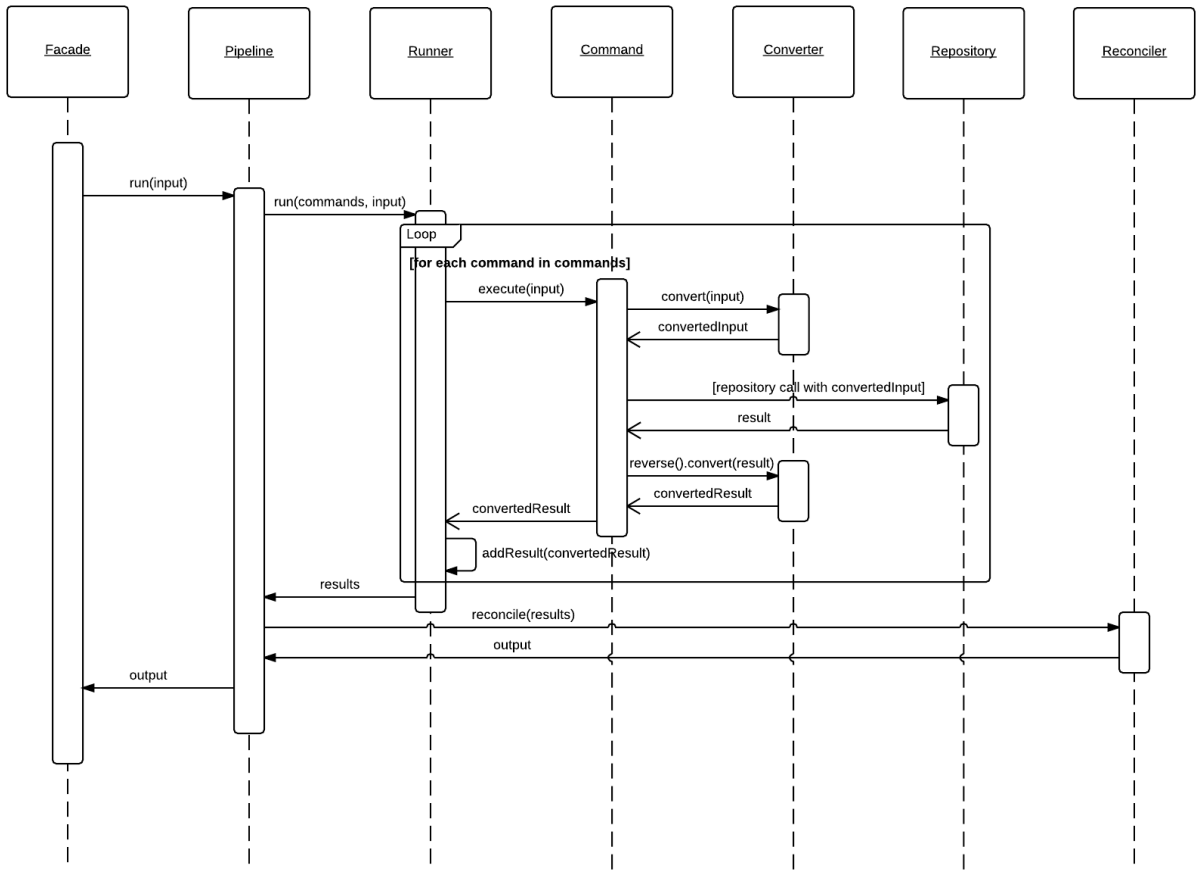

Figure 6-13. Diamond architecture sequence diagram.

The architecture is designed to increase both the accessibility and flexibility of large-scale data stores. To achieve accessibility, the architecture focuses on maintaining existing concepts of enterprise architectures and preserving the desire of application developers to work with rich data models. Flexibility is accomplished by allowing the application developer to easily adopt or abandon data stores. These attributes are of the utmost importance to application developers as it is highly likely they will be confronted with an increasing, not decreasing, amount of specialized technology choices in the future.

CHAPTER 7: Implementation: The Machinist Framework

The Diamond architecture is intended to be easily implementable in a variety of languages. For the purposes of this dissertation the reference implementation was developed in the Java programming language [71]. Java was largely chosen due to its widespread adoption among enterprise software organizations as well as the increasing number of Java-based data stores. Regardless of language preference, the overall Diamond architecture is designed to afford accessible polyglot persistence for the implementer.

7.1 The Machinist Framework

The Java-based implementation of the Diamond architecture described by this dissertation is produced as the Machinist framework, the name originating from a machinist being a person who uses machine tools to make or modify parts of a greater system. This name is meant to enforce the belief that software architectures are moving away from one-size-fits-all data stores and towards data stores (i.e., tools) built specifically for specialized tasks (e.g., information retrieval, graphs, analytics, real-time streaming, etc.). The remainder of this chapter focuses on describing the Machinist framework as a concrete implementation of the Diamond architecture. The full class diagram for this framework appears in Appendix B.

7.1.1 Command

The Machinist framework offers implementations of Commands required by typical CRUD use cases. These Commands encapsulate common behaviors necessary when persisting and retrieving data from a data store. The implementation of these commonly occurring usage scenarios will be discussed, in detail, through the remainder of this section.

7.1.1.1 Persist

Persistence logic in an enterprise application is isolated from the rest of the application through use of the Repository pattern. The Diamond architecture recommends that existing and newly developed Repositories be wrapped by a Command which provides a callback for the execution of Repositories after data model conversion. The workflow of a persist Command is shown below:

1. Forward convert input for persistence
2. Pass converted input to Repository through a callback for persistence
3. Backward convert Repository result and return the converted result

Although the persist Command does not hold onto input or output as state, it does require a Converter and a Repository callback during initialization. The Converter is used to convert domain objects forwards into a form suitable for use by the callback and wrapped Repository for persisting. The class diagram for the SaveCommand is shown in Figure 7-1.

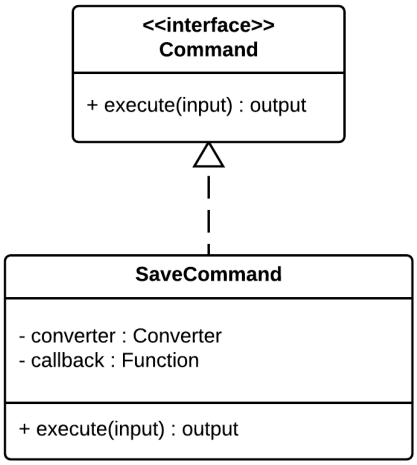

Figure 7-1. The SaveCommand class diagram.

An example implementation of the SaveCommand provided by the Machinist framework is shown below.

```
public class SaveCommand<I, M> implements Command<I, I> {
    private Converter<I, M> converter;
    private Function<M,M> repositoryCallback;

    public SaveCommand(Converter<I, M> converter, Function<M,M> repositoryCallback) {
        this.converter = converter;
        this.repositoryCallback = repositoryCallback;
    }

    @SuppressWarnings("unchecked")
    @Override
    public I execute(I input) {
        M converted = converter.convert(input);

        M result = repositoryCallback.apply(converted);

        return converter.reverse().convert(result);
    }
}
```

Figure 7-2. The Machinist framework SaveCommand implementation.

The code given is straightforward and simply applies the workflow outlined previously in this section. For implementation purposes, Machinist relies on Google

Guava for Converter and Function implementations. This is not required but leveraging Guava-provided implementations will allow those who have already implemented data model Converters or useful Functions to utilize their existing implementations. The code shown in Figure 7-2 enables the enforcement of constraints through the use of Java generics. The application developer is required to provide two generic types when instantiating the SaveCommand, one for input (*I*), the other for the converted or intermediate (*M*) value. The goal of this enforcement is to align the Converter output with the input required by the Repository. In this manner, a data model can be adapted to any Repository signature through conversion and allow the conversion to enforce type safety through generics. Although not shown separately, the update workflow is typically the same or similar to the persist workflow.

7.1.1.2 Read

Reading from a data store presents a variety of challenges that are often associated with querying. The Machinist framework makes no effort to provide the user with a single, unified query interface. Instead, the framework relies on each Repository to understand how to interact with its underlying data store. Through a Repository callback, any method exposed on the Repository can be accessed by the Command that holds on to the callback. If needed, custom Converters that understand how to create complex, data store specific, queries using the data model provided to the Converter can be implemented. When retrieving objects from a data store, it is common to only need a forward conversion to generate a key or complex

query. For this reason, Commands responsible for retrieving data from data stores receive an additional argument during construction, which is responsible for providing the Command with a way to construct keys or queries for data retrieval. The enumerated workflow for read Commands is listed below.

1. Generate a lookup key or complex query from input
2. Pass the generated lookup or query to Repository through a callback
3. Backward convert Repository result and return the converted result

The Machinist implementation of this workflow is similar to that of the SaveCommand code shown in Figure 7-2. The additional constructor argument provided to the FindCommand is a Guava Function that is responsible for generating a lookup key or complex query from the provided input. The class diagram and implementation code is provided in Figure 7-3 and Figure 7-4 below.

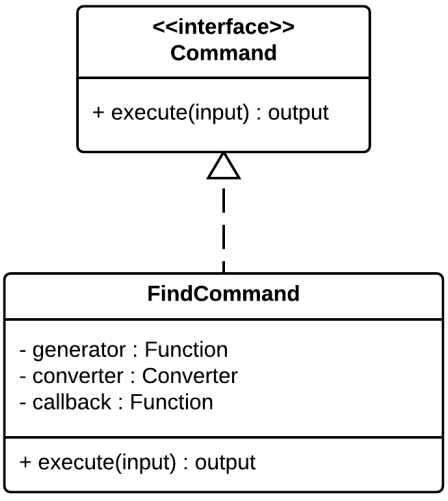

Figure 7-3. The FindCommand class diagram.

```
public class FindCommand<I, M, O, ID> implements Command<I, O> {
    private Function<I, ID> keyGenerator;
    private Converter<O, M> valueConverter;
    private Function<ID, M> repositoryCallback;

    public FindCommand(Function<I, ID> keyGenerator, Converter<O, M> valueConverter, Function<ID, M> repositoryCallback) {
        this.keyGenerator = keyGenerator;
        this.valueConverter = valueConverter;
        this.repositoryCallback = repositoryCallback;
    }

    @Override
    public O execute(I input) {
        ID key = keyGenerator.apply(input);
        M result = repositoryCallback.apply(key);

        return valueConverter.reverse().convert(result);
    }
}
```

Figure 7-4. The Machinist framework FindCommand implementation.

The Machinist implementation of the read workflow again uses the Google Guava Converter and Function interfaces to provide key generation, data model conversion, and a Repository callback. Java generics are used to provide enforcement of data types. The FindCommand enforces key or query (*ID*), input (*I*), intermediate (*M*), and output (*O*) types. An example of a callback for a FindCommand is shown on the third page of Appendix A; this callback shows how the search functionality for the example application is implemented for the relational data store. Examples of other callback functions appear throughout the code samples of this dissertation. For example, the solid outline-highlighted sections of code in Figures 9-5 and 9-7, are instances of commands being configured with callbacks.

7.1.1.3 Delete

The recommended delete workflow is a simplified read workflow. Delete implementations require the ability to generate a lookup key or complex query

criteria to locate candidate results, but they do not require data model conversion, as they do not return results. The delete workflow is as follows:

1. Generate a lookup key or query from input
2. Pass generated lookup to Repository through a callback

The Machinist implemented delete workflow uses two functions, a lookup key or query criteria generator and a Repository callback. It provides generic type enforcement for input (*I*) and keys (*ID*). The class diagram and implementation is shown in Figure 7-5 and Figure 7-6.

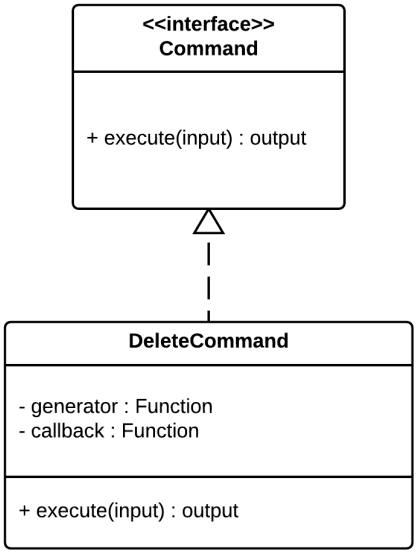

Figure 7-5. The DeleteCommand class diagram.

```
public class DeleteCommand<I, ID> implements Command<I,Void> {
    private Function<I, ID> keyGenerator;
    private Function<ID, Void> repositoryCallback;

    public DeleteCommand(Function<I, ID> keyGenerator, Function<ID, Void> repositoryCallback) {
        this.keyGenerator = keyGenerator;
        this.repositoryCallback = repositoryCallback;
    }

    @Override
    public Void execute(I input) {
        ID key = keyGenerator.apply(input);
        return repositoryCallback.apply(key);
    }
}
```

Figure 7-6. The Machinist framework DeleteCommand implementation.

7.1.1.4 Many or All

Applications often require the persistence or retrieval of more than one object at a time. This allows the application to take advantage of batch persist and read operations provided by data stores. To persist more than one entity at a time, the Machinist SaveAllCommand implements a modified persist workflow.

1. For each input: forward convert input for persistence
2. Persist batch of converted entities using Repository through callback
3. For each result: backward convert Repository result
4. Return converted results

To retrieve multiple objects, the read workflow is also slightly modified. The following workflow is appropriate for retrieving multiple results from a data store.

1. Generate lookup key or complex query from input
2. Pass generated lookup to Repository through callback
3. For each result: backward convert Repository result
4. Return converted results

The Machinist implementations for each workflow are given in Figure 7-7 and Figure 7-8 for reference.

```
public class SaveAllCommand<I, M> implements Command<Iterable<I>, Iterable<I>> {
    private Converter<I, M> converter;
    private Function<Iterable<M>, Iterable<M>> repositoryCallback;

    public SaveAllCommand(Converter<I, M> converter, Function<Iterable<M>, Iterable<M>> repositoryCallback) {
        this.converter = converter;
        this.repositoryCallback = repositoryCallback;
    }

    @Override
    public Iterable<I> execute(Iterable<I> inputs) {

        Collection<M> entities = new ArrayList<>();
        for (I input : inputs) {
            entities.add(converter.convert(input));
        }

        Iterable<M> rawResults = repositoryCallback.apply(entities);

        Collection<I> results = new ArrayList<>();
        if (rawResults != null) {
            for (M result : rawResults) {
                results.add(converter.reverse().convert(result));
            }
        }

        return results;
    }
}
```

Figure 7-7. The Machinist framework SaveAllCommand implementation.

```
public class FindAllCommand<I, M, O, ID> implements Command<I, Iterable<O>> {
    private Function<I, ID> keyGenerator;
    private Converter<O, M> valueConverter;
    private Function<ID, Iterable<M>> repositoryCallback;

    public FindAllCommand(Function<I, ID> keyGenerator, Converter<O, M> valueConverter, Function<ID, Iterable<M>> repositoryCallback) {
        this.keyGenerator = keyGenerator;
        this.valueConverter = valueConverter;
        this.repositoryCallback = repositoryCallback;
    }

    @Override
    public Iterable<O> execute(I input) {
        ID key = keyGenerator.apply(input);
        Iterable<M> rawResults = repositoryCallback.apply(key);

        Collection<O> results = new ArrayList<>();
        if (rawResults != null) {
            for (M result : rawResults) {
                results.add(valueConverter.reverse().convert(result));
            }
        }

        return results;
    }
}
```

Figure 7-8. The Machinist framework FindAllCommand implementation.

The implementations make use of the same constructs (e.g., Functions, Converters, and Repository callbacks) as the previous Commands, but they enforce slightly different generic types, requiring java.lang.Iterables to handle multiple inputs and outputs.

7.1.2 Runner

To provide flexible methods of invoking commands, the Machinist framework implements three useful Runners. Runners provided by the Machinist framework are capable of executing Commands consecutively or concurrently.

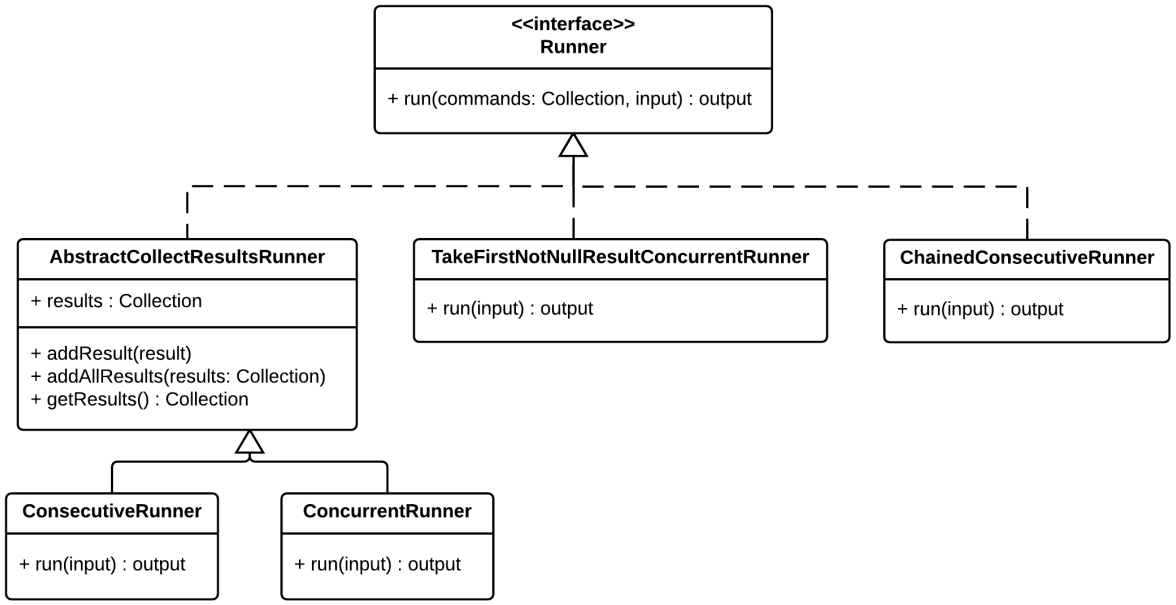

Figure 7-9. Machinist framework Runner implementations.

The concurrent implementations provide blocking or non-blocking behavior. If blocking, the results of each Command will be collected by the Runner and returned once all Commands have executed (i.e., the ConcurrentRunner behavior). If a non-blocking Runner is used, such as the TakeFirstNotNullResultConcurrentRunner, the first available result from any Command will be immediately returned, canceling any other Commands.

The default behavior for consecutive Runners is to block until the Command has completed execution. Machinist provides two consecutive Runners. The ConsecutiveRunner executes Commands serially, collecting all returned results as it goes. The ChainedConsecutiveRunner also executes Commands serially but uses the output of the previous Command as the input for the next Command, eventually

returning the result of the last Command executed. This behavior is helpful when executing trees of Commands.

7.1.3 Reconciler

Reconcilers are often highly application dependent, and, because of this, Machinist provides only one basic but useful implementation. The default Reconciler returns the first non-null result from a set of results. This basic behavior provides a reasonable "fan-in" behavior for the framework. Many applications will need to implement more complex implementations (e.g., sort results via a Comparator, query an additional data source for metadata, etc.). The class diagram for the FirstNotNullResultReconciler is shown in Figure 7-10.

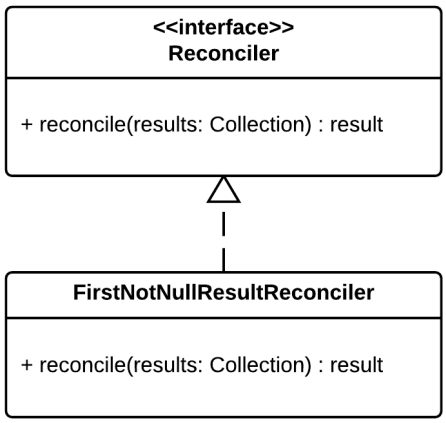

Figure 7-10. The FirstNotNullResultReconciler class diagram.

7.1.4 Pipeline

Machinist provides a variety of Pipeline implementations that are intended to be applicable to a wide variety of production usage scenarios. Although the high-level interface is quite simplistic, the class structure provided by the Machinist

framework offers the user a large degree of expressivity and flexibility through composition. In this section, we will provide an overview of each Machinist Pipeline and their anticipated usage scenarios.

Machinist extends the top-level Pipeline interface, creating the CommandPipeline interface, which adds the ability to hold a collection of Commands to the top-level Pipeline. The internal structure for this storage is a java.util.Collection as implemented by AbstractCommandPipeline. Machinist Pipelines that extend the AbstractCommandPipeline use a java.util.ArrayList as the concrete implementation, but any java.util.Collection can be used. ArrayList was chosen as the default to preserve execution order and allow the user to duplicate Commands if necessary. Figure 7-11 shows the class diagram for the AbstractCommandPipeline.

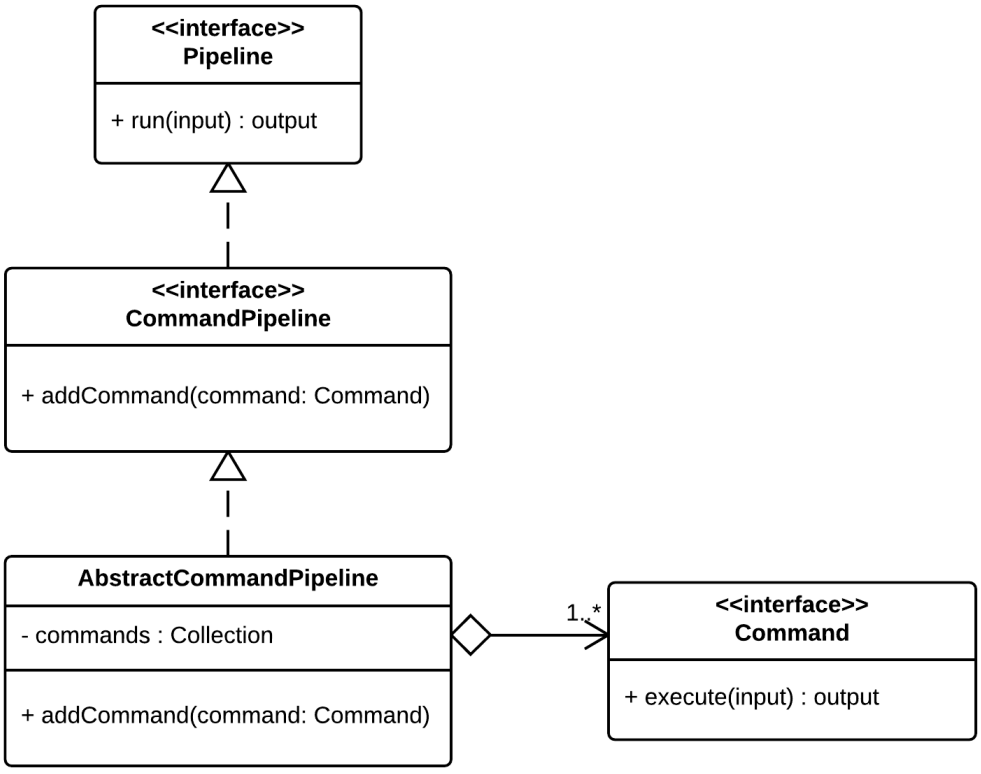

Figure 7-11. The Machinist Pipeline class structure.

Pipelines that extend the AbstractCommandPipeline are not required to conform to the Diamond architecture, as the CommandPipeline does not require the use of a Runner and Reconciler. The CommandPipeline interface is available to users of the framework that wish to use the Pipeline and Command concepts outside of the persistence tier. Omitting the use of Runners and Reconcilers in the persistence tier is possible but not recommended.

The recommended point of extension for users of the Machinist framework is the DiamondPipeline. This Pipeline requires a Runner and a Reconciler upon initialization, which enforce a "fan-out" and "fan-in" behavior with each run of the Pipeline. This behavior is responsible for the Pipeline's characteristic shape as

shown in Figure 6-10. The DiamondPipeline class is not abstract and implements the *run()* method of Pipeline which simply delegates to the Runner and Reconciler given at initialization.

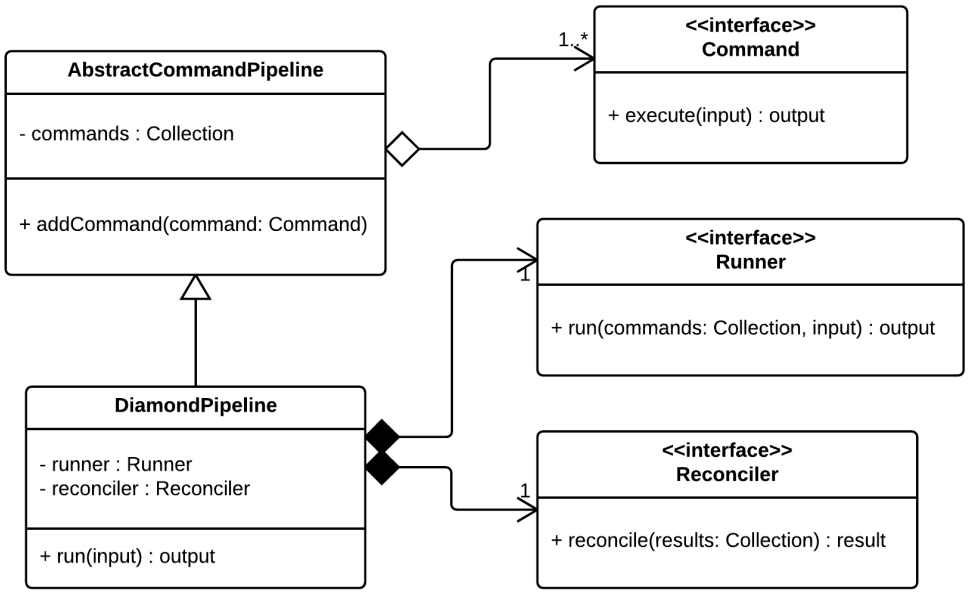

Figure 7-12. The Machinist framework DiamondPipeline class structure.

Pipelines that extend the DiamondPipeline are straightforward to develop. Extension is typically accomplished through composition and is achieved by initializing a Pipeline with a custom Runner or Reconciler, which is exactly how the Pipelines of the Machinist framework are implemented. There are a variety of Pipelines provided by the framework to satisfy commonly encountered production usage scenarios. Each will be briefly detailed in the following sub-sections.

7.1.4.1 Consecutive Pipelines

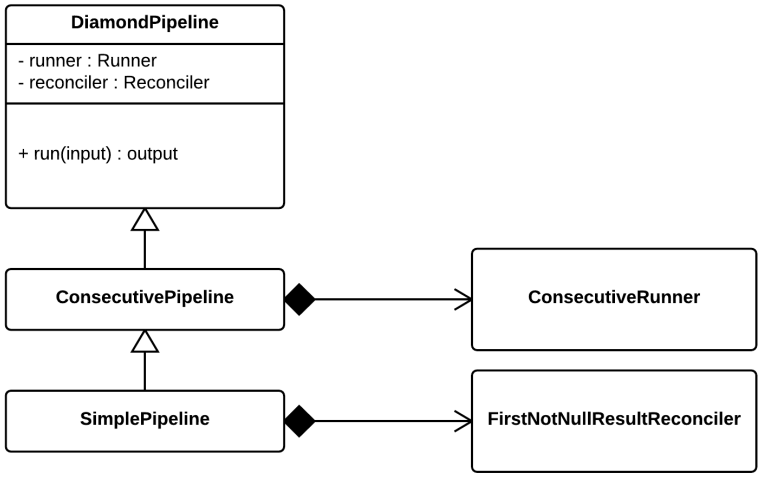

Figure 7-13. The ConsecutivePipeline and SimplePipeline class diagram.

The consecutive Pipelines run Commands in consecutive order using the ConsecutiveRunner. Input is passed to each Command and the Runner collects the results. The ConsecutivePipeline can be instantiated directly but requires that a Reconciler be provided through the constructor. The SimplePipeline is an extension of the ConsecutivePipeline. It utilizes the ConsecutiveRunner but defaults the Reconciler to the FirstNotNullResultReconciler. The SimplePipeline is the most basic Pipeline available to an adopter of the Machinist framework.

7.1.4.2 Concurrent Pipeline

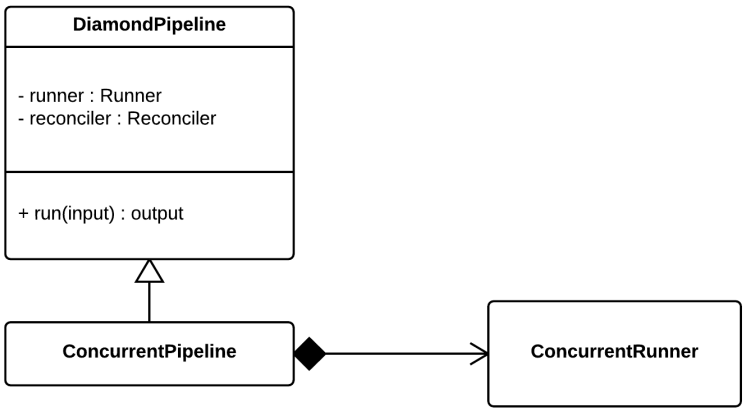

Figure 7-14. The ConcurrentPipeline class diagram.

The ConcurrentPipeline invokes Commands simultaneously using the ConcurrentRunner. By using the ConcurrentRunner, each Command is executed concurrently, but the Runner will block until all Commands have returned. This behavior is implemented using the Java concurrency API (e.g., java.util.concurrent.Executors, Callable, etc.). The results of the concurrently executed Commands are collected by the Runner and passed to the Reconciler by the Pipeline, which is user defined for this Pipeline.

7.1.4.3 Flash Pipeline

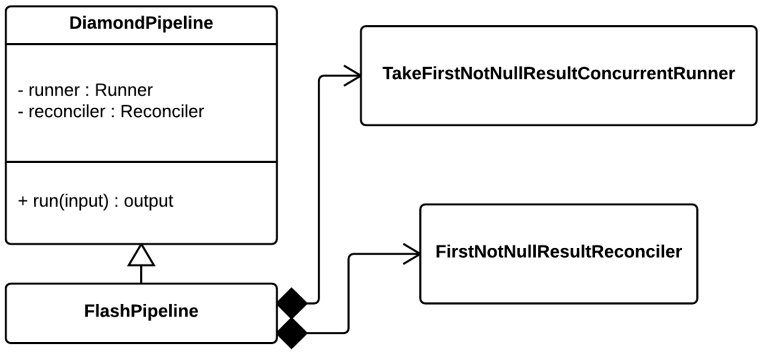

Figure 7-15. The FlashPipeline class diagram.

The FlashPipeline was developed to retrieve results via the fastest available method from any available data store. A typical usage scenario is to concurrently query a number of data stores that hold similar information and return the first result available. This behavior, similar to the ConcurrentPipeline, is implemented using the Java concurrency framework with the primary difference being the FlashPipeline returns the output of a Command as soon as one is available, without awaiting the completion of additional Commands. For this reason, the user is not required to provide a Reconciler, as there will be only a single fastest result or results set. The Reconciler used by the FlashPipeline serves as an additional null check but is largely a no-op.

7.1.4.4 Tiered Pipelines

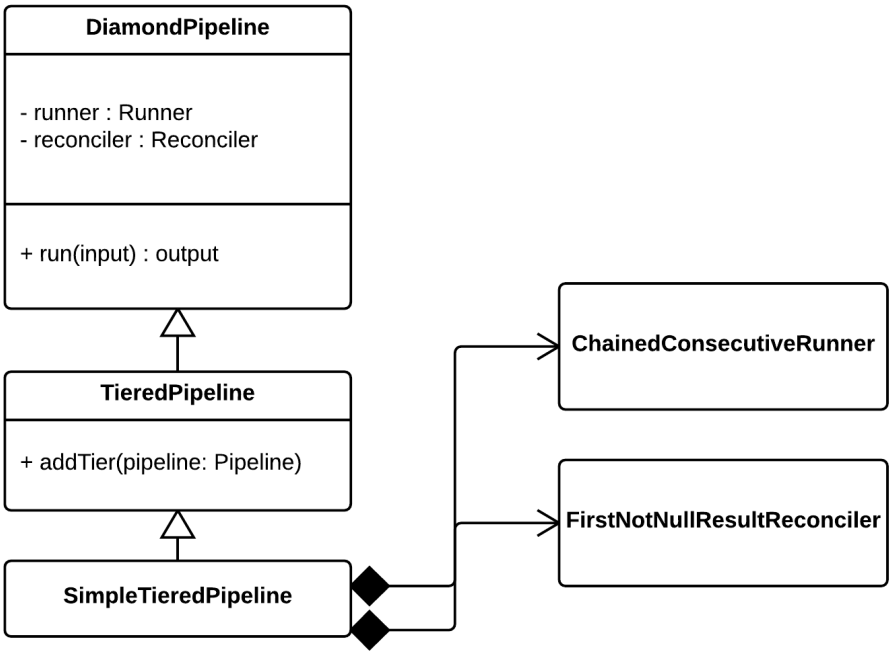

Figure 7-16. The TieredPipeline and SimpleTieredPipeline class structure.

When persisting to multiple data stores, the need to break persistence operations into tiers occurs frequently. For instance, persisting to a relational database before any other data stores will assign a uniquely generated primary key to the data model. Although this is not the only way to define a primary key for data model lookup, many systems already persist to a relational database, and doing so before persisting to additional data stores ensures that at least one copy of the data model has been persisted successfully with the added benefit of assigning a primary key. Non-traditional classes of data stores, such as key-value and columnar, frequently make use of composite keys for entity lookup. Already having

an identifier present on the data model allows it to be combined with other attributes to form such a composite.

The TieredPipeline is a DiamondPipeline with an ancillary method that allows the addition of a *tier*. This allows the user to create an arbitrary number of tiered Commands. In the Machinist framework each tier can either be a single Command or a Pipeline. If a Pipeline is used, the Pipeline is able to use an arbitrary Runner and Reconciler, independent of the parent TieredPipeline. This is an interesting feature of the TieredPipeline as it allows the user to configure execution behavior per tier (e.g., execute the first tier consecutively, the second concurrently, etc.). In this way the user may want to ensure that data is written to critical data stores first and then concurrently, possibly asynchronously, write to additional stores. The SimpleTieredPipeline provides defaults for the Runner and Reconciler of the TieredPipeline. The default Reconciler, similar to the SimplePipeline, is the FirstNotNullResultReconciler. The default Runner is the ChainedConsecutiveRunner, which uses the output of the previous tier as input for the next tier.

7.1.5 Decorators

The Machinist implementation was developed to be expressive and extensible, specifically focusing on the composability of its components. One way the framework supports these attributes is by providing a series of Decorators for each architectural component.

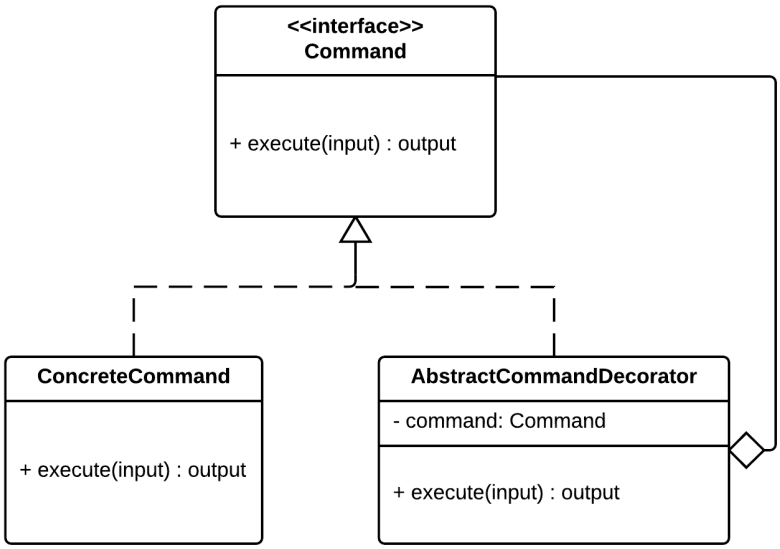

Figure 7-17. The Decorator pattern.

The most common component to decorate is the Command interface. Decorating a Command is straightforward and allows for the modification of behavior before and after the execution of the decorated Command. Machinist offers two useful Command decorators: a SourceCommandDecorator and a TimeCommandDecorator.

7.1.5.1 Source Decorator

In applications utilizing polyglot persistence, it is helpful to understand the specific data store the data has been retrieved from. This information can be wielded in a variety of ways such as aiding in debugging and providing the Reconciler with source metadata to make increasingly intelligent reconciliation decisions. Providing source information also allows the application to reconstitute a

result or results set that has been split apart and persisted across multiple data stores (e.g., attribute one in data store one, attribute two in data store two, etc.).

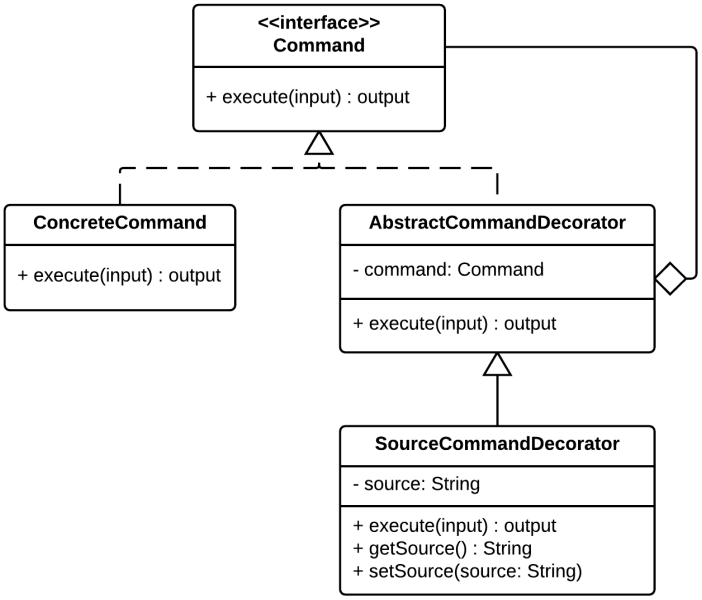

Figure 7-18. The SourceCommandDecorator class diagram.

The Machinist framework provides an extensible abstract class that accepts a Command during initialization. Extending this class—as the SourceCommandDecorator does—allows the application developer to easily make new Decorators for any Command. In the case of the SourceCommandDecorator, the source is defined during construction of the Decorator. The Decorator then executes the decorated Command, and the source is set on the result of the Command execution before the Decorator returns.

7.1.5.2 Time Decorator

Gathering empirical data on the performance of each data store in use by an application is not always apparent when utilizing multiple data stores. Capturing

this information through the Machinist framework is straightforward. Similar to attaching a source to a Command result, a Decorator can be used to instrument Command execution with a timer.

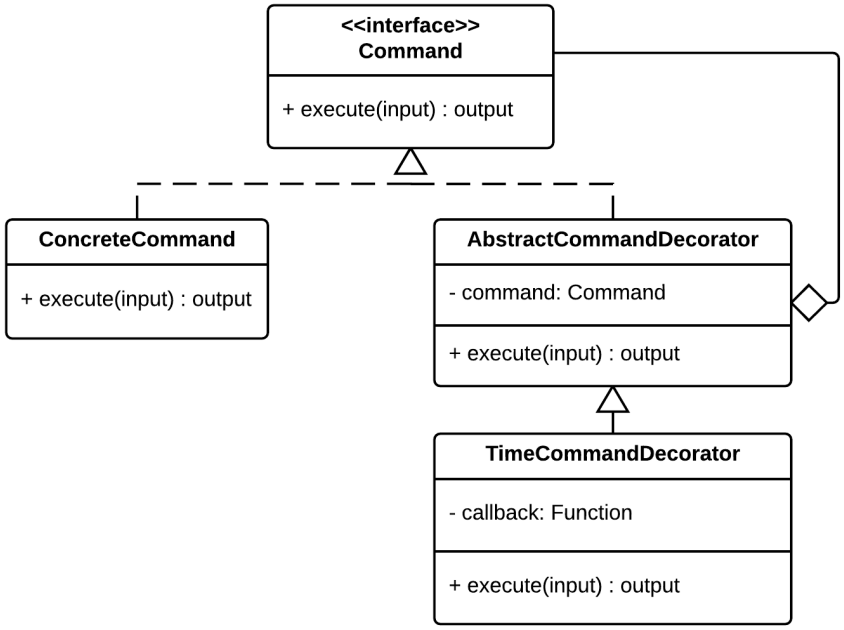

Figure 7-19. The TimeCommandDecorator class diagram.

The TimeCommandDecorator accepts a callback that will be passed the elapsed runtime, in nanoseconds, of the decorated Command as a method argument. Using this information it is possible to determine the runtime of each Command, which is helpful for debugging, performance tuning, and generally gaining a better understanding of how each data store in an application is functioning. An interesting use of this data is to empirically show which data stores perform better for similar or identical tasks. Because the framework offers a callback, any number of behaviors can be implemented including logging timings or asynchronously persisting timings to additional data stores for future analysis.

CHAPTER 8: MBox: A Multi-Data Store Inbox

To fully demonstrate the expressivity of the Diamond architecture and its implementation—the Machinist framework—it is helpful to look at a concrete, multi-data store application. MBox is a web-based email reader designed to take advantage of multiple data stores. Implemented in Java, MBox serves as a functioning example of the Diamond architecture, depending directly on the Machinist framework. In this chapter we will provide an overview of the MBox application, showcasing many useful features of the architecture and framework.

8.1 Overview

MBox is a read-only IMAP client developed using core Java and the Spring Framework (i.e., Spring Boot, Spring Data, and Spring MVC [72]. The user interface has been developed with React [73], a JavaScript framework developed by Facebook, and Bootstrap [74], a CSS framework developed by Twitter. A screenshot of the application is provided in Figure 8-1 below.

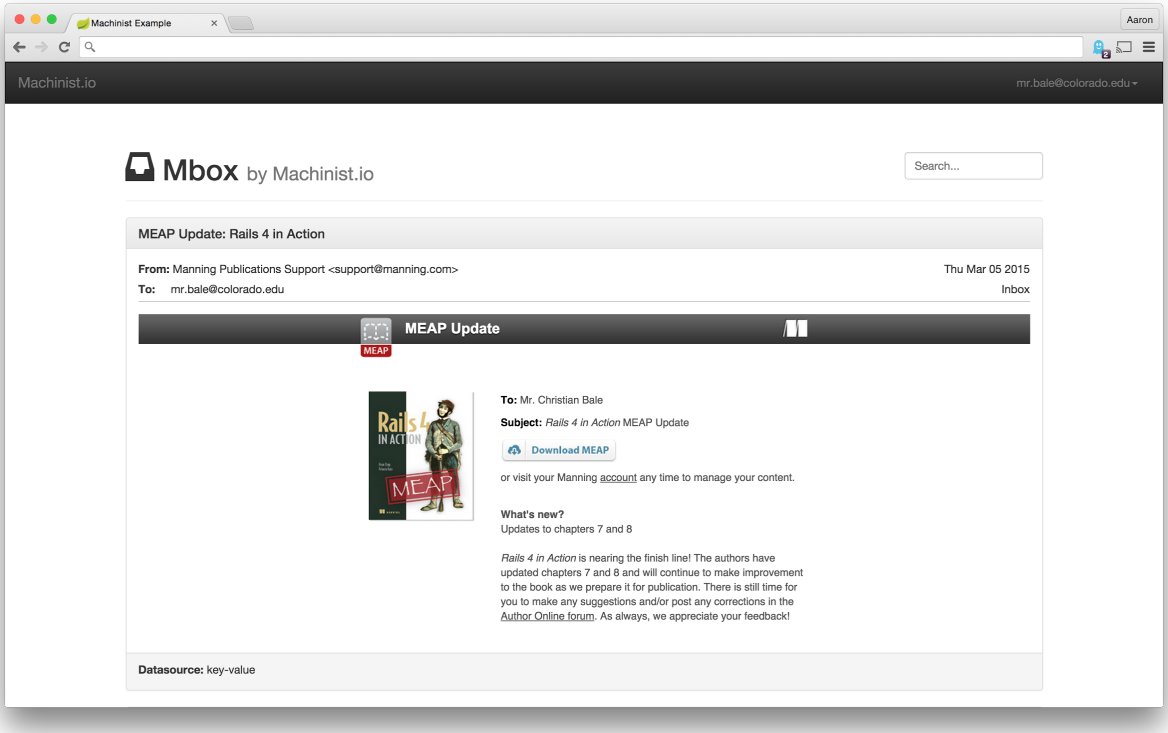

Figure 8-1. The MBox application.

The MBox application is modest, as can be seen in Figure 8-1, but also designed to exercise a variety of complex usage scenarios. It is worth noting the attributes of each email displayed by the user interface. The following attributes are required to be available to the user interface to render properly:

- Title - the title of the email
- From address - the email address or addresses of the sender
- To address - the email address or addresses of recipients
- Date received - the date-and time that the email was received
- Content - the text or html content of the email
- Datasource - the datasource that the email was retrieved from

The application also provides search functionality over all fetched emails via the search input in the upper right of the layout. Searches are executed as the characters are typed to give the user immediate feedback. The goal of the MBox application is to provide a non-trivial example of the Diamond architecture through the use of the Machinist framework. We feel that the feature set contributed covers many commonly encountered usage scenarios for heterogeneous data-intensive systems.

8.2 Design and Architecture

The MBox architecture follows an enterprise architecture pattern consisting of an application, service, and persistence tier. The persistence tier, which had focused on a traditional Repository pattern, has been enhanced to make use of the Diamond architecture, through the Machinist framework. The MBox application was designed to use four independent classes of data stores: relational, key-value, columnar, and document. The high-level architecture of the MBox application is shown in Figure 8-2.

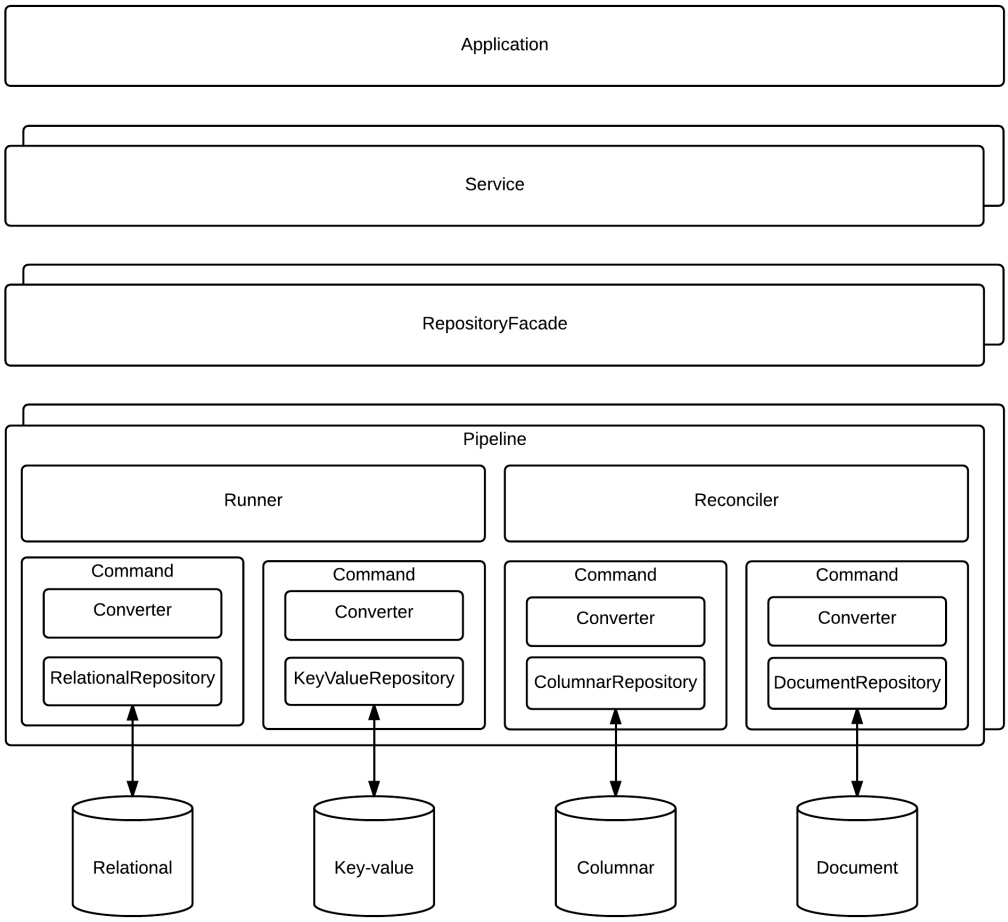

Figure 8-2. The MBox architecture utilizing four disparate data stores.

The above architecture implements the full Diamond architecture including the RepositoryFacade. To utilize multiple data stores, a series of Pipelines were developed for fulfilling persistence operations (e.g., save, find, find all, search, etc.). The application architecture follows the pattern shown in Figure 6-12, creating a Service and RepositoryFacade per domain object.

8.2.1 Data Model

The domain objects available in the MBox application are User and Email. The class diagram for the data model is provided via UML in Figure 8-3.

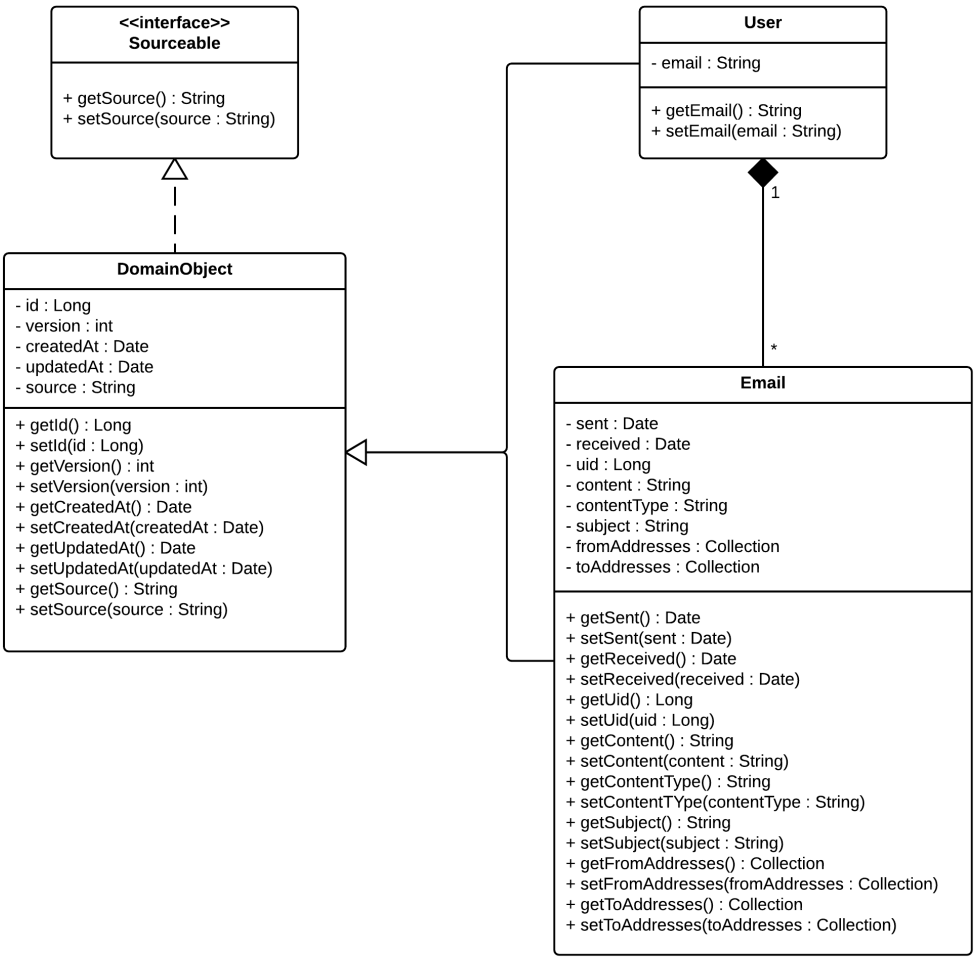

Figure 8-3. The MBox data model.

The common parent of the data model is the DomainObject, which is designed to hold attributes commonly needed for persistence, regardless of data store. Each object in the data model extends this common parent and adds its own attributes. The User class is intended to be simplistic and only holds the email address associated with an IMAP account to enable the fetching of messages from a hosted

email server. The Email model mimics common attributes provided by email message APIs, specifically MBox uses the javax.mail API.

8.2.2 Persist

The Machinist framework strives to leverage the existing persistence strategy of the application. The persistence strategy of the MBox application is implemented through use of the Repository pattern. In this pattern there will be a Repository per data store per domain object. The advantages of this approach are numerous, but specifically of note is the composability of this strategy. Architecting persistence in this way allows for the application to use one or many Repositories through composition as needed, either by interacting with multiple data stores, multiple data models, or both. The Diamond architecture allows for the accomplishment of all of these actions, but, if required, the application can choose to communicate directly with any data store that is needed through a Repository, bypassing the architecture (e.g., Pipelines and Commands). This approach results in a high degree of flexibility, allowing the architecture to adapt to unanticipated use cases if necessary. Three typical persistence and retrieval scenarios are implemented in the MBox application for the Email domain object. Each scenario leverages multiple data stores and independent aspects of the Machinist framework. We will describe, in detail, the following scenarios: saving Emails, finding all Emails for a given User, and searching persisted Emails.

For reference, the interfaces for the EmailService, EmailRepositoryFacade, and underlying Repositories are provided in Figure 8-4 and Figure 8-5. Of note is

the heterogeneity of the underlying Repository method definitions. Coping with this level of heterogeneity is one of the strengths of the Diamond Architecture. Converters are used by Commands to adapt the domain model to any signature required by the underlying Repository.

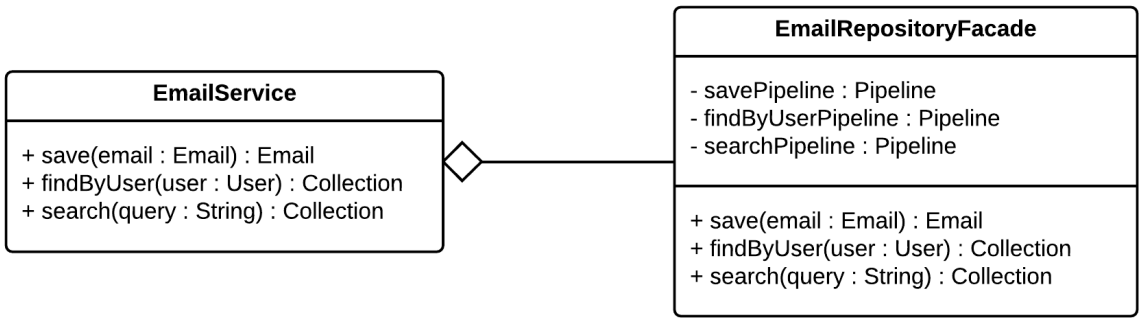

Figure 8-4. The EmailService and EmailRepositoryFacade definitions.

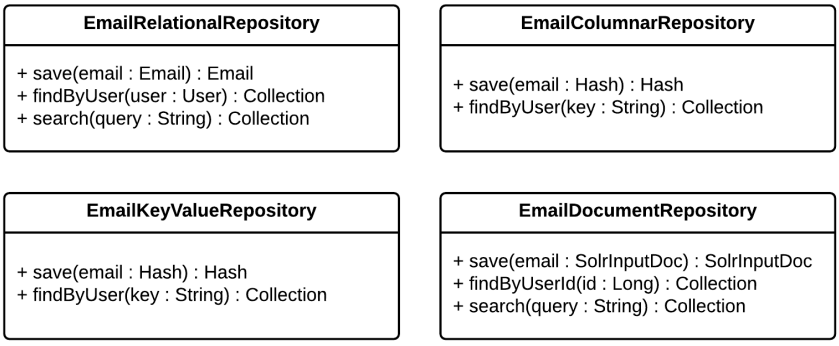

Figure 8-5. The set of Email Repositories with diverse method definitions.

8.2.3 Save

An MBox Email is persisted to all four available data stores of the application. To accomplish this using the Machinist framework, a Pipeline containing a series of SaveCommands is created. Each SaveCommand implements

the persist workflow as described in Chapter 7 and requires the application developer to implement a data model Converter and Repository callback. A SaveCommand is constructed per data store. The conceptual architecture with associated concrete Machinist implementations is given in Figure 8-6 below.

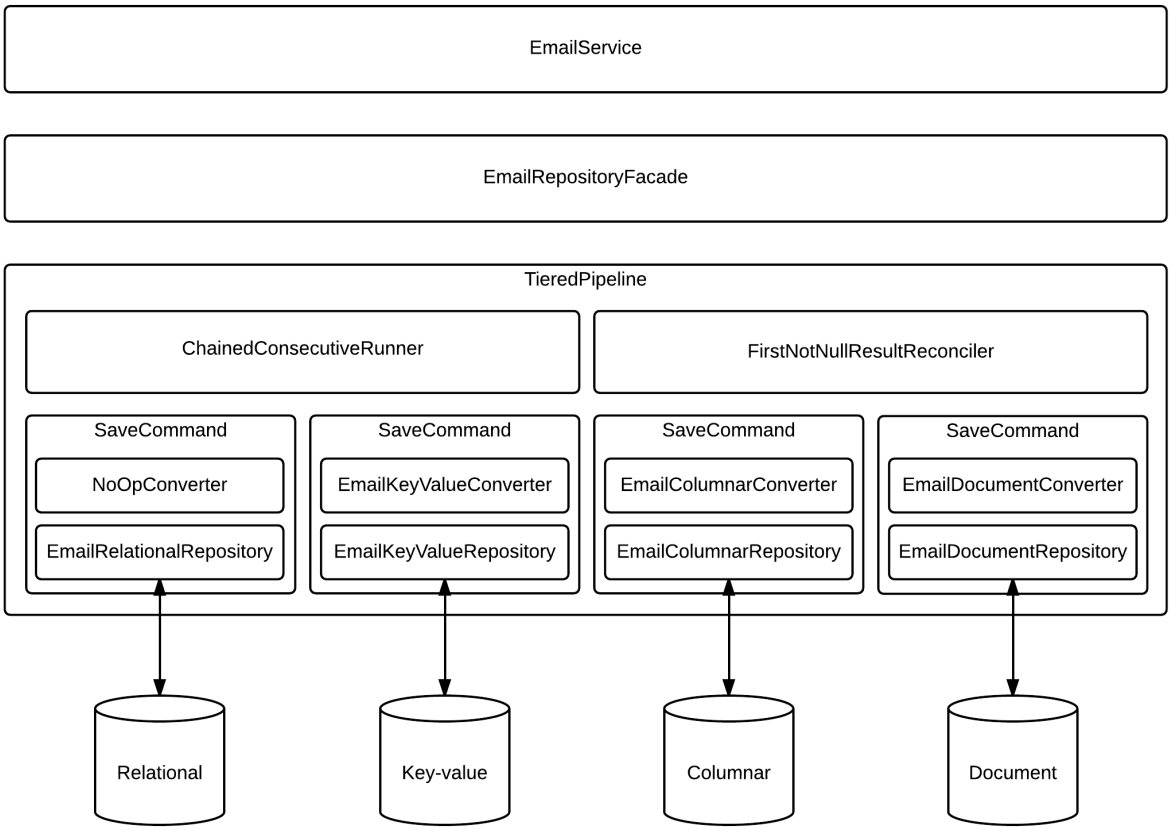

Figure 8-6. The save Email architecture.

As shown above, the individual Repositories are implemented on a per data store basis and are responsible for encapsulating all behavior necessary for CRUD operations. The Repository callback wraps direct method calls to the Repository to provide one level of indirection, which enables the execution of arbitrary behavior, not only calls to the Repository. The relational and document Repositories utilize

the Spring Data Framework [75] which affords full object relational mapping capabilities for the data model. The key-value Repository is implemented as an in-memory hash, and the columnar Repository is implemented using the DataStax Java Driver for Apache Cassandra [76]. All Repositories implement a save method with varying method signatures as previously shown in Figure 8-5. To adapt to the heterogeneous method signatures of these Repositories, Converters must be implemented. For the Email save Pipeline, one framework provided Converter and three custom Converters are needed.

The relational and document Repositories accept rich data models for persistence. Spring Data affords object relational mapping capabilities for these Repositories. No conversion is necessary for the relational Repository; however, the SaveCommand constructor requires a Converter so the NoOpConverter is used. This Converter passes input through without modification, which is a common use case for object relational mapper-based Repositories that internally understand how to properly deconstruct and reconstruct a data model for persistence.

The key-value and columnar Repositories each utilize similarly developed Converters. Generally, in key-value and columnar data stores, a complex and normalized object graph must be modified significantly to enable effective storage. Because these data stores do not provide traditional table structures and data reference capabilities, it is common to de-normalize the data model and embed referenced structures. This is the approach taken by the MBox application for key-value and columnar persistence.

```
public class EmailKeyValueConverter extends FieldBasedCompositeKeyConverter<Email, Map<String, String>> {
    public EmailKeyValueConverter() {
        super(new EmailKeyValueKeyGenerator());
    }

    @Override
    protected Map<String, String> doForward(String key, Email email) {
        return singletonMap(key, new Gson().toJson(email));
    }

    @Override
    protected Email doBackward(Map<String, String> stringStringMap) {
        Assert.isTrue(stringStringMap.size() == 1, "Size of Key-value must be 1!");

        String value = stringStringMap.values().toArray(new String[stringStringMap.size()])[0];

        return new Gson().fromJson(value, Email.class);
    }
}
```

Figure 8-7. The EmailKeyValueConverter implementation of the MBox application.

To effectively store an Email in the key-value and columnar data stores, the Converter is responsible for embedding the associated User during conversion. The result of the conversion is a hash with a composite key and JSON representation of the embedded object graph as its value. The composite key is a string generated from attributes present on the Email (e.g., ID, UID, User ID, User email address, etc.) joined by a colon. The JSON representation is generated by the Google Gson library [77] and stored as a string.

Although the document Repository provides object relational mapping capabilities through Spring Data, a Converter is still needed to effectively store the data model. The Converter used for the document Repository is necessary to enhance the data model before persistence. This is done through the addition of attributes to the data model. For information retrieval tasks, content is often duplicated and analyzed (e.g., splitting, stemming, n-gram generation, etc.) differently to fulfill a variety of use cases. In the MBox application, Email content

is typically a complex MIME object such as a HTML document. Although a search index could be built from HTML, it would likely result in poor search results. Parsing the content of the Email and storing it separately results in a dramatic increase in relevant search results.

```
public class EmailDocumentConverter extends Converter<Email, SolrInputDocument> {
    private DocumentObjectBinder binder;

    public EmailDocumentConverter() {
        this.binder = new DocumentObjectBinder();
    }

    @Override
    protected SolrInputDocument doForward(Email email) {
        SolrInputDocument inputDocument = binder.toSolrInputDocument(email);

        String content = email.getContent();
        if(StringUtils.isNotEmpty(content)) {
            inputDocument.addField("plain_content_en", Jsoup.parse(content).text());
        }

        Long userId = email.getUser().getId();
        if(userId != null) {
            inputDocument.addField("user_id_l", userId);
        }

        return inputDocument;
    }

    @Override
    protected Email doBackward(SolrInputDocument inputDocument) {
        return binder.getBean(Email.class, ClientUtils.toSolrDocument(inputDocument));
    }
}
```

Figure 8-8. The EmailDocumentConverter implementation of the MBox application.

The EmailDocumentConverter adds two additional fields to the Email data model to achieve effective persistence. The first field is used to store the parsed Email content which is generated by the jsoup library [78]. As previously mentioned, the parsed content is used for search specific tasks. The second additional field is the ID of the User associated with the Email. This is necessary to

filter search results (i.e., Emails) by User. The addition of this field allows the application to store all Emails, regardless of User, in a single index while only returning results belonging to the current User. This behavior is accomplished via a filter query.

To execute the four individual SaveCommands, which make use of the previously described Repositories and Converters, a Pipeline must be created. As described in the previous chapter, there are a variety of implementations provided by the framework. For the purposes of the MBox application, a TieredPipeline is used. This is the optimal Pipeline for this usage scenario because we would like to ensure the Email is persisted to at least one data store (i.e., relational) before attempting to persist to additional data stores. Because a relational database will also assign a unique identifier (i.e., primary key) upon a successful save, we are able to use this unique identifier where required in other data stores (e.g., composite keys, document identifiers, etc.).

```
TieredPipeline<Email, Email> emailSavePipeline = new SimpleTieredPipeline<>();
// tier one is save to relational database
emailSavePipeline.addCommand(relationalCommand);
// tier two is a concurrently executed save to key-value and document
DiamondPipeline<Email, Email> tierTwo = new ConcurrentPipeline<>(new FirstNotNullResultReconciler<>());
tierTwo.addCommand(keyValueCommand);
tierTwo.addCommand(documentCommand);
tierTwo.addCommand(columnarCommand);
// add tier two
emailSavePipeline.addTier(tierTwo);
```

Figure 8-9. The configuration of the MBox Email save Pipeline.

By using a SimpleTieredPipeline, the Pipeline will be defaulted to make use of a ChainedConsecutivePipeline and a FirstNotNullResultReconciler. The first tier is simply the Command used to save Emails to the relational data store. The

second tier is a ConcurrentPipeline that will concurrently persist to the key-value, document, and columnar data stores. Of note is the type enforcement, through Java generics, of the Pipeline inputs and outputs: the Pipeline is defined to accept an Email as input and requires that an Email be provided as output. This ensures that the Pipeline enforces a similar method signature to the Facade or Service method that is executing it. In the case of the MBox application, the application tier will initiate the call sequence by invoking the save method on the Service. The Service will in turn invoke the save method on the Facade which will then delegate to the configured Pipeline and result in the persistence of the Email messages.

8.2.4 Find By User

In many applications that allow for the creation of a user account, it is common to fetch all entities associated with the user account; the MBox application is no different. When a user authenticates to the application, Emails are fetched from the server and persisted in the manner described in the previous section. Each Email is associated with a User before persistence, which provides a data model capable of retrieving all Emails belonging to a given User. The MBox application is concerned with providing the optimal performance when retrieving data, so our choice of storage strategies and Pipelines reflects this consideration. The high-level architecture used to find all Emails belonging to a given User is shown in Figure 8-10 below.

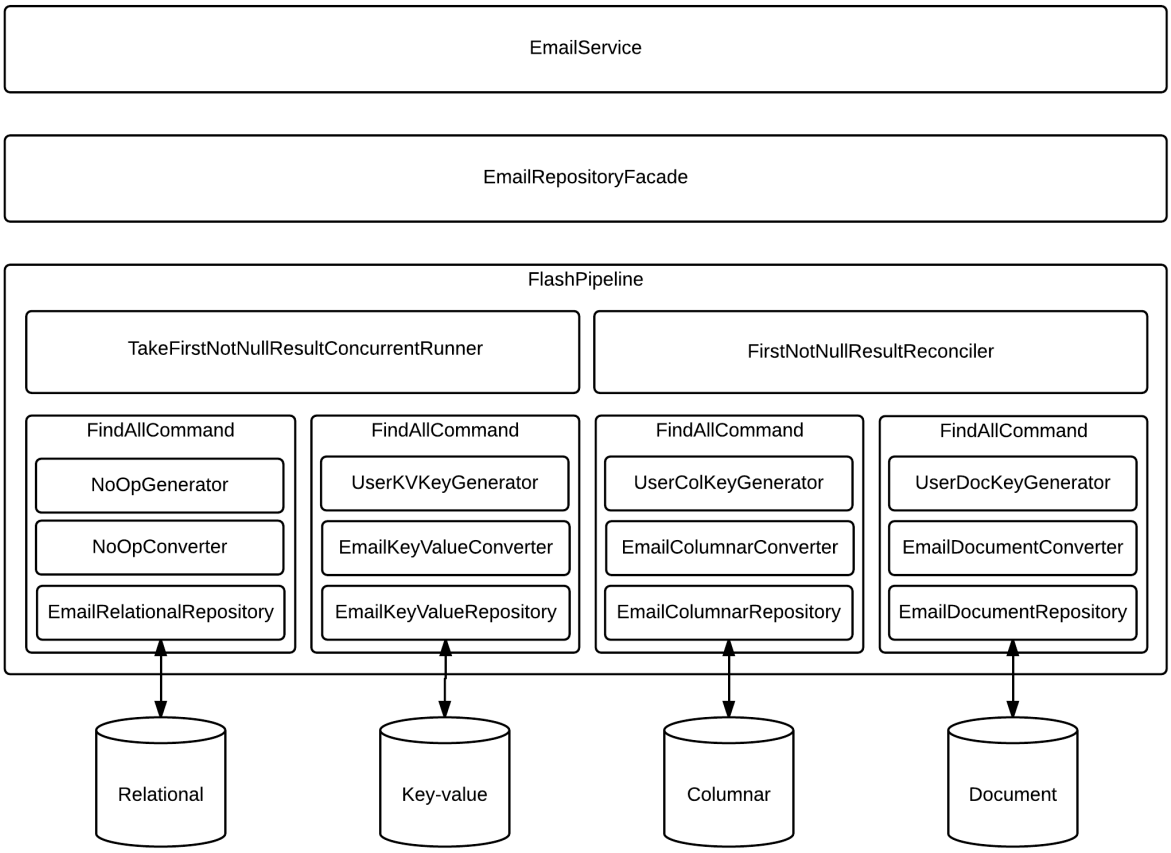

Figure 8-10. The find all Emails for User architecture.

The find architecture is similar to the save architecture with a few notable differences. The architecture makes use of the framework provided FindAllCommand, which is responsible for implementing the read many or all workflow described in Chapter 7. In the find workflow, a Function is needed to produce a key suitable for data store specific lookup. The result of this operation could be as simple as calling a method that returns the object's primary key (i.e., ID) or as involved as constructing a complex query criteria that accounts for advanced operations such as paging and filtering. The key generators developed for use in the find all Emails belonging to a User scenario are responsible for accepting

a User and generating a simple data store specific lookup key. Because the relational Repository accepts a User in its find method, no key generator is necessary. The key-value and column key generators create composite keys from a variety of User attributes (i.e., ID, email address) by joining the attributes together with colons to create the final lookup key. The document key generator simply returns the primary key (i.e., ID) of the given User, which will be used to filter the eventual results set by User. The Converters remain the same as for the save case. This is one of the advantages of encapsulating all behavior pertaining to data model conversion in discrete components. Having this behavior spread throughout a Repository or set of Repositories would negatively impact reuse. In fact, many of the Converters developed by the application make use of existing key generators through composition to generate composite keys during forward conversion in preparation for persistence.

The second notable difference between the save and find architectures is the Pipeline used. For retrieving results in the most efficient manner available, the framework provides the FlashPipeline. This Pipeline concurrently executes its Commands and awaits the first non-null result or results set returned. Once the result has been received, all Commands still executing are cancelled. The FlashPipeline is ideal for MBox because the application is not opinionated about where a result originates from and the application makes a conscious choice to duplicate all information required by the user interface in each data store. The resulting query operation always retrieves results from the fastest available data

store, even if the fastest store changes from query to query. This desirable behavior is easy to achieve using the Diamond architecture. The code to accomplish this behavior is shown in Figure 8-11.

```
DiamondPipeline<User, Iterable<Email>> emailFindByUserPipeline = new FlashPipeline<>();
emailFindByUserPipeline.addCommand(relationalCommand);
emailFindByUserPipeline.addCommand(keyValueCommand);
emailFindByUserPipeline.addCommand(documentCommand);
emailFindByUserPipeline.addCommand(columnarCommand);
```

Figure 8-11. The configuration of the MBox find Emails by User Pipeline.

8.2.5 Search

Search has come to be an expected feature of many applications. MBox implements comprehensive search across multiple data stores, again using the Diamond architectural style. However, not all data stores lend themselves to being suitable for search tasks. Document is a class of data stores that typically excel with regard to information retrieval tasks. To enable full-text search over a user's Emails, the MBox application utilizes the capabilities of the document and relational data stores. In a production application, a relational data store is unlikely to perform as well as a document store for search tasks. Using both classes of stores in this application is intended to show the flexibility of the architecture, and additionally demonstrate a usage scenario where the quality of results are able to degrade slightly to provide durability and optimal performance. The high-level search architecture is shown in Figure 8-12 below.

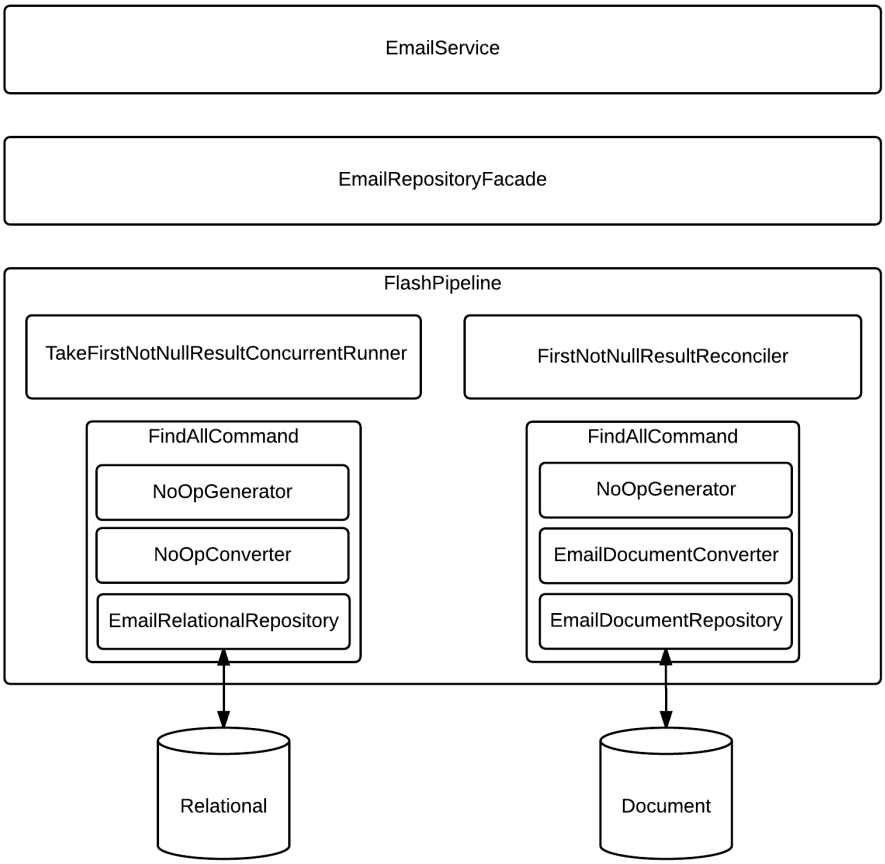

Figure 8-12. The search Emails for User architecture.

As the diagram indicates, the search architecture only utilizes two of the four available data stores. The EmailService and EmailRepositoryFacade both provide search methods that take a query string as a method argument. This query is passed to the Pipeline as input, enforced by Java generics as a string, and requires no conversion before being given to each Repository.

```
DiamondPipeline<String, Iterable<Email>> searchPipeline = new FlashPipeline<>();
searchPipeline.addCommand(relationalCommand);
searchPipeline.addCommand(emailCommand);
```

Figure 8-13. The configuration of the MBox search Pipeline.

To enable proper search functionality, the application is again utilizing a FlashPipeline to optimize the persistence tier for the best performance possible among the available data stores. If the application would have instead optimized for improved precision or recall of search results, a ConcurrentPipeline, which blocks until all Commands have executed and returned, with a custom Reconciler might have been used. To increase precision, a custom reconciliation strategy should be implemented to apply an order of precedence to the results set; for instance, the Reconciler could be implemented to prefer results from the more search capable document store. Increasing recall could be accomplished by creating a reconciliation strategy that merges all results from both data stores.

8.2.6 Decorators

The framework-provided Decorators are used throughout the MBox application, most notably to attach a data source to a result. The data source attribute is displayed in the footer of each Email presented by the user interface, as seen in Figure 8-1. For debugging purposes, the TimeCommandDecorator is also used frequently to compare and contrast the performance of each data store on a task-by-task basis. Having empirical timings of such operations is easily achieved using the Machinist framework. The code required to decorate the Commands used by MBox is provided below.

```
// LOG timing
relationalCommand = new TimeCommandDecorator<>(relationalCommand, new LogTimingCallback(DATA_SOURCE_RELATIONAL));
keyValueCommand = new TimeCommandDecorator<>(keyValueCommand, new LogTimingCallback(DATA_SOURCE_KEY_VALUE));
documentCommand = new TimeCommandDecorator<>(documentCommand, new LogTimingCallback(DATA_SOURCE_DOCUMENT));
columnarCommand = new TimeCommandDecorator<>(columnarCommand, new LogTimingCallback(DATA_SOURCE_COLUMNAR));

// Add a source field to the results
relationalCommand = new SourceAllCommandDecorator<>(relationalCommand, DATA_SOURCE_RELATIONAL);
keyValueCommand = new SourceAllCommandDecorator<>(keyValueCommand, DATA_SOURCE_KEY_VALUE);
documentCommand = new SourceAllCommandDecorator<>(documentCommand, DATA_SOURCE_DOCUMENT);
columnarCommand = new SourceAllCommandDecorator<>(columnarCommand, DATA_SOURCE_COLUMNAR);
```

Figure 8-14. The MBox application Command Decorators.

Use of the Decorators is straightforward and easily extended. The callback required by the TimeCommandDecorator, for our purposes, simply prints the provided timings to standard out. A more advanced implementation could keep track of timings for other scenarios such as determining what configured Command would be most efficient to execute based on historical runtimes, avoiding the need to execute all configured Commands when there is a high likelihood the same Command will always return first.

CHAPTER 9: Evaluation

In this chapter we present the evaluation of our architecture using a variety of common scenarios encountered by heterogeneous data-intensive applications. The developed framework, Machinist, and associated application, MBox, are used to perform the evaluation.

9.1 Overview

The ideal architecture for persistence in heterogeneous data-intensive systems will result in accessible and flexible polyglot persistence. In addition, the implementation of an associated framework will not hinder attributes afforded by the underlying, large-scale data storage systems. These systems are designed to offer scalability, availability, and durability with a consistent focus on performance. The ideal architecture must minimize the overall effort required to adopt large-scale data stores without conceding the desirable attributes of these technologies.

By specifically leveraging the author's professional background as the co-founder and Chief Technology Officer of a large-scale data-intensive software startup, an independent software consultant, a software engineer at a publicly traded company, and an academic research assistant, we have the necessary experience to discuss the constraints present in highly demanding, large-scale software environments. Each existing system the author has contributed to leverages state-of-the art tools from many computer science disciplines outside of

software engineering, including distributed systems, programming languages, machine learning, natural language processing, and human-centered computing.

9.2 Approach

The approach taken for the evaluation of the Diamond architecture and associated implementation is that of a combined analytical and quantitative nature. We will enumerate a variety of scenarios that commonly occur within heterogeneous data-intensive systems. For each scenario, we will be able to compare and contrast our approach, through the use of our architecture and framework, to a more traditional approach that does not utilize our architecture. We will use the functionality of the MBox application to constrain each evaluation scenario, putting each usage scenario in the context of an existing application's architecture to cater to a familiar problem domain and data model.

9.3 Assumptions

To analytically evaluate the Diamond architecture against more traditional approaches to polyglot persistence, we must begin by describing our initial assumptions of both architectures. Our focus will be on the architectural changes that must occur in the architectures to fulfill each scenario. For the sake of this evaluation, we will assume that the data stores in use are already available to the application developer (e.g., data store deployment, schema definition, etc.). Both architectures will be assumed to be operating on the same data model, that of the MBox application which is composed of a User and a set of Email messages.

The initial starting point for each approach (three-tier vs. Diamond) will be a full architecture implementation that makes use of a relational data store. The traditional enterprise application architecture will consist of application, service, and persistence tiers. The service and persistence tiers make use of Repository implementations that provide save and find operations. The Diamond approach will consist of application, Service, RepositoryFacade, Pipeline, and Sub-Repository components for a relational data store. It is also assumed that the Diamond approach will have existing implementations for save and find operations against this data store. The initial conceptual architectures are provided in Figure 9-1.

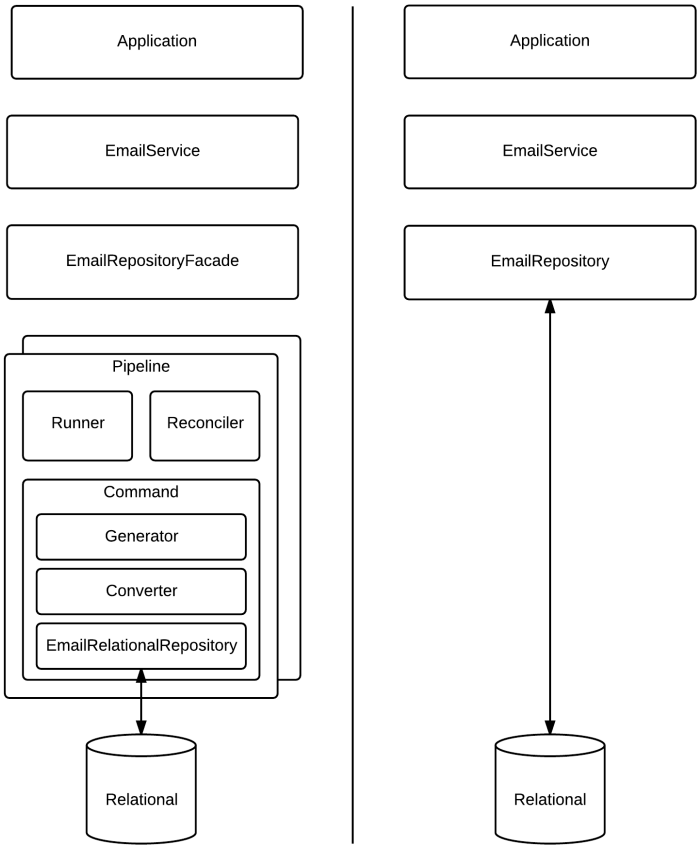

Figure 9-1. The initial evaluation systems using a Diamond architecture (left) and a traditional three-tier architecture (right).

As can be seen above, for a single data store there is some amount of additional work required by the Diamond architecture. The Diamond architecture requires a Pipeline for CRUD-based methods present in the service tier (e.g., save, find, find all, search, etc.). Each of these Pipelines requires a Command that in turn requires a Function (i.e., key or query generator for finds), Converter, and Repository. The addition of the Pipeline and its components, as well as the RepositoryFacade, is additional work in comparison to the traditional architecture. However, this effort is similar to the work of developing a system in anticipation of the system being run using multiple threads or on multiple servers. Up front effort is required even if the software is initially running in a single thread or on a single server, but that effort results in measurable savings when the system is later scaled. The implementation code for both architectures is given in Appendix A.

9.4 Evaluation Scenarios

The remainder of this chapter will discuss a number of evaluation scenarios, which contrast the advantages and disadvantages of the approaches contributed by the architectures. As a result, we contribute a thorough evaluation of our work.

9.4.1 Additional Data Store

As an application transitions from a homogeneous to a heterogeneous architectural style, it will require the ability to add an additional data store to its existing architecture. The need for an additional data store can arise from a variety of requirements including specialization, such as information retrieval tasks,

performance, or redundancy, among others. Using the two initial architectures, we will discuss the steps required to add an additional data store to both architectures.

In a traditional enterprise architecture, there are multiple possible points of extension for the addition of a data store, the first being the existing Repository. The existing Repository already isolates the details of persistence for a single data store from the service tier, making the Repository a natural point of extension. The second reasonable point of extension is the existing service tier. Selecting the service tier as the point of extension is also reasonable as it allows the Repository to continue to focus on encapsulating the persistence details of a single store. Because the MBox application is using Spring Data for its relational Repository, extending existing Services (i.e., EmailService) requires the least amount of effort. A Spring Data Repository abstracts and obfuscates many of the implementation details associated with relational persistence, making it difficult to adapt for polyglot persistence. It is far easier to create additional Repositories than enhance the existing Spring Data Repository. Applying this reasonable strategy, the following steps are required to add an additional data store to our traditional architecture.

1. Create an additional Repository for the new data store
2. Inject the new Repository into existing Services

A new Repository must be created to encapsulate the persistence details of the store and then added to the existing service tier for later use.

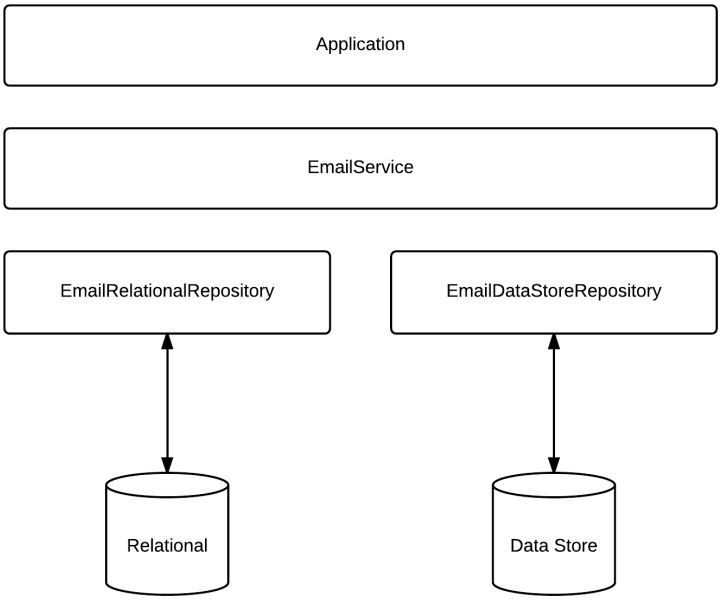

Figure 9-2. Three-tier architecture with two data stores.

The steps required to prepare the Diamond architecture for an additional data store are very similar to the steps required to prepare a traditional, three-tier, architecture. It is helpful to point out that we will compare the details of actual persistence scenarios (i.e., making use of the additional Repository) later in this chapter. For now, we wish to focus on the amount of work required to prepare the architectures for an additional data source without explicitly invoking the additional data source (e.g., save, find, find all, etc.). The steps for additional data store inclusion required by the Diamond architecture are listed below.

1. Create an additional Repository for the new data store
2. Ensure the new Repository is available for use by existing Pipelines

As with the traditional approach, only two high-level steps are required by the Diamond architecture, and they are similar in effort. Allowing the Repositories to be discovered by the service tier or existing Pipelines can be done by directly instantiating the Repositories. However, in an enterprise software organization, this is likely to occur through an inversion of control framework, such as the Spring Framework, which provides access to the Repositories through an application context (i.e., a global context of instantiated classes). Regardless, the Diamond architecture requires less modification than the traditional architecture since the service tier is already configured to invoke Pipeline Commands via the RepositoryFacade (requiring no Service refactoring). The new data source will automatically be invoked correctly as soon as the Pipelines are updated to include the new Commands that read and write to the new data store through the new Repository. The prepared two data store Diamond architecture is shown in Figure 9-3 below for reference.

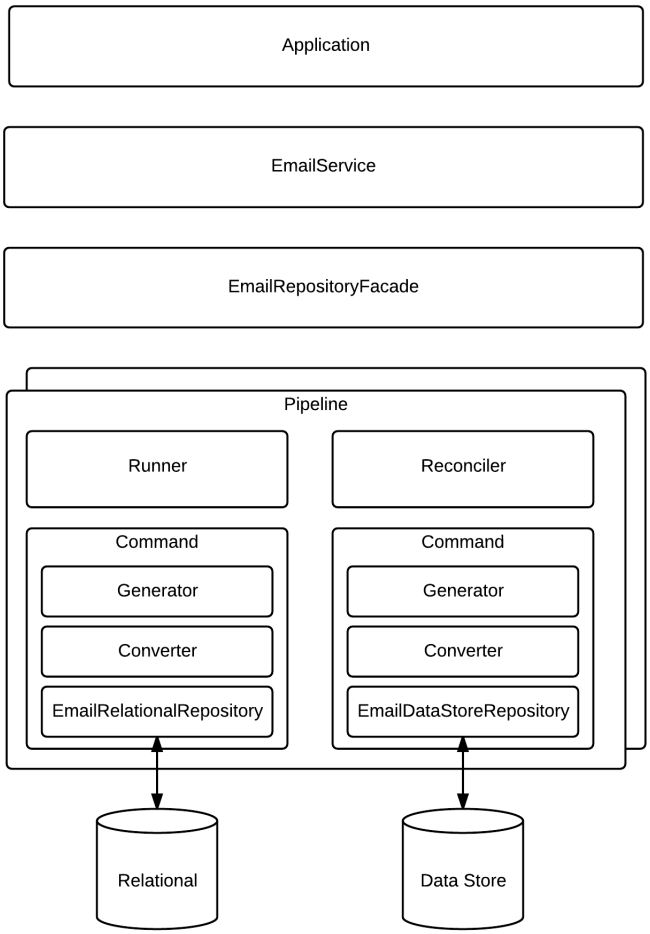

Figure 9-3. Diamond architecture with two data stores.

9.4.2 Persist Data to Additional Store

When adding an additional data store, the first step is to enable the persistence of data to the new store. In a traditional three-tier architecture, as described in the previous scenario and shown in Figure 9-2, an existing Service will be required to take on additional responsibilities. The following steps are required by the traditional architecture to enable persistence to a newly available data store.

1. Create save method on new data store Repository
2. Add data model conversion code to the existing Service save method
3. Invoke each Repository
4. Collect results of each Repository invocation
5. Implement the reverse data model conversion per Repository result
6. Add additional logic to Service to reconcile multiple Repository results

The steps listed above would be added to the save method of the EmailService in the traditional architecture. Doing so requires the refactoring of the save method to adapt to multiple data stores, converting, invoking, and reconciling with different behavior for each store within a single method. The implementation of the three-tier architecture EmailService save method is below.

```
public Email save(Email email) {

    // Save Email to Relational store
    Email relationalResult = emailRelationalRepository.save(email);
    ////////////////////////////////////////////////////////////////////////////////
    // Convert Email for Document store: converting result so we get ID from Relational
    DocumentObjectBinder binder = new DocumentObjectBinder();
    SolrInputDocument inputDocument = binder.toSolrInputDocument(relationalResult);

    String content = email.getContent();
    if(StringUtils.isNotEmpty(content)) {
        inputDocument.addField("plain_content_en", Jsoup.parse(content).text());
    }

    Long userId = email.getUser().getId();
    if(userId != null) {
        inputDocument.addField("user_id_l", userId);
    }
    ////////////////////////////////////////////////////////////////////////////////
    // Save Email to Document store
    SolrInputDocument documentResult = emailDocumentRepository.saveDocument(inputDocument);
    ////////////////////////////////////////////////////////////////////////////////
    // Backwards convert results
    Email result1 = binder.getBean(Email.class, ClientUtils.toSolrDocument(documentResult));
    Email result2 = relationalResult;
    ////////////////////////////////////////////////////////////////////////////////
    // reconcile
    return result2 != null ? result2 : result1;
}
```

Figure 9-4. Traditional architecture EmailService save method for two data stores.

Figure 9-4 explicitly shows the steps required to achieve polyglot persistence in a traditional architecture across a relational and document data store. Each line style represents a different behavior required to enable polyglot persistence. The solid outline sections show interactions with the underlying traditional, per data store, Repositories. The cross-hash (i.e., x) annotation highlights code associated with data model conversion, both forwards and backwards. Finally, the dashed annotation shows the code necessary to provide reconciliation of the multiple results returned by each Repository. Of note is that adapting each of the highlighted sections of code to accommodate an increasing number of data stores is non-linear.

Additional data stores will require additional code across all highlighted sections. This additional code may need to be intermingled throughout the existing method to provide the necessary functionality, as is shown by persisting to the relational data store first to assign a primary key for use later in the method (i.e., solid outline sections). Additionally, conversion and reconciliation code (i.e., cross-hash and dashed sections respectively) is arbitrarily complex and will result in effort-intensive refactorings as the number of data stores increase.

The Diamond architecture provides constructs for many of these behaviors. The steps required by the Diamond architecture are listed below.

1. Create additional save method on data store Repository
2. Implement a new data model Converter for the data store
3. Add a new SaveCommand to the existing save Pipeline

The Machinist framework allows the user to utilize full implementations of common constructs encountered during the development of heterogeneous persistence tiers. Leveraging these components during data persistence results in clear isolation of concerns. The code for adding an additional data store to the save Pipeline using the Machinist framework is given below.

```
public Pipeline<Email, Email> emailSavePipeline() {
    // Relational with no conversion
    Command<Email, Email> relationalCommand = new SaveCommand<>(new NoOpConverter<>(), emailRelationalRepository::save);
    // Document
    Command<Email, Email> documentCommand = new SaveCommand<>(new EmailDocumentConverter(), emailDocumentRepository::saveDocument);

    DiamondPipeline<Email, Email> emailSavePipeline = new SimplePipeline<>();
    emailSavePipeline.addCommand(relationalCommand);
    emailSavePipeline.addCommand(documentCommand);

    return emailSavePipeline;
}
```

Figure 9-5. Diamond architecture Email save Pipeline for two data stores.

Similar to Figure 9-4, Figure 9-5 explicitly shows the code required to achieve polyglot persistence through utilization of the Machinist framework. In contrast to the traditional approach, the Diamond architecture scales linearly through the use of its architectural components. Adding an additional data store to an application is accomplished by adding an additional Command to an existing Pipeline (i.e., enforcing a linear relationship between Commands and data stores). All conversion, Repository, and reconciliation behavior (i.e., cross-hash, solid, and dashed highlights respectively) is encapsulated throughout the architecture, enforcing the single responsibility principle and minimizing non-linear growth as additional persistence technologies are adopted. This characteristic gives the Diamond architecture a significant advantage when adapting an application to a large number of data stores.

9.4.3 Retrieve Data from Additional Store

After the successful persistence of data to a newly available store, the need for data retrieval becomes apparent. In a traditional architecture, the service tier requires modification to take full advantage of the new store. Below, we list the

steps required to enable an existing service tier to retrieve data from an additional data store.

1. Add additional find method to data store Repository
2. Add code to generate key from data model
3. Invoke each Repository
4. Collect results of each Repository invocation
5. Implement data model conversion per Repository result
6. Add additional logic to reconcile multiple Repository results

Again, the existing find Service method must be refactored to accommodate an additional data store. The refactored method is given below. Although in our implementation we are leveraging Spring Data to enable data model conversions automatically, this is not the case with other data stores and would result in a more complex refactoring.

```
public Iterable<Email> findByUser(User user) {
    // Get lookup key
    String documentID = user.getId().toString();

    // Invoke Repositories
    Iterable<Email> relationalResult = emailRelationalRepository.findByUser(user);
    Iterable<Email> documentResult = emailDocumentRepository.findByUserId(documentID);

    // No convert necessary due to Spring Data usage

    // reconcile
    return relationalResult != null && !Iterables.isEmpty(relationalResult) ? relationalResult : documentResult;
}
```

Figure 9-6. Traditional architecture Service find method for two data stores.

At this point, there is code duplication between the save and find data model conversion, and there is likely to be duplication between the invocation, collection, and reconciliation strategies of each method. The Diamond architecture requires the following steps for the identical scenario.

1. Create additional find method on data store Repository
2. Implement a data model key generator Function for the data store
3. Add a new FindCommand using the key generator and existing Converter to the existing find Email Pipeline

The Diamond approach again leads to clear isolation of concerns and allows the user to leverage the existing data model Converter from the persistence scenario. The code to add an additional data store to the find Pipeline is given below.

```
public Pipeline<User, Iterable<Email>> emailFindByUserPipeline() {
    // Find all emails for a user
    Command<User, Iterable<Email>> relationalCommand = new FindAllCommand<>(new NoOpConverter<>(), new NoOpConverter<>(),emailRelationalRepository::findByUser);
    Command<User, Iterable<Email>> documentCommand = new FindAllCommand<>(new EmailDocumentKeyGenerator(), new NoOpConverter<>(),emailDocumentRepository::findByUserId);

    DiamondPipeline<User, Iterable<Email>> emailFindByUserPipeline = new SimplePipeline<>();
    emailFindByUserPipeline.addCommand(relationalCommand);
    emailFindByUserPipeline.addCommand(documentCommand);

    return emailFindByUserPipeline;
}
```

Figure 9-7. Diamond architecture find Pipeline for two data stores.

9.4.4 Split Data Model Across Data Stores

Given a complex data model, it can be beneficial to store different aspects of the data model in a disparate number of data stores. One common example of this scenario is when a data model holds on to a large amount of content (e.g., HTML

content body, encoded media data such as a large photo or video). Storing the metadata for the data model in a different store than the content can lead to performance advantages. Splitting the data model in the traditional architecture again requires the modification of the existing Service methods to accommodate the following steps.

1. Service save method must be modified to select only needed attributes off of data model before invoking each Repository
2. Service find method must be modified to query each store for a given attribute
3. Service save method must be modified to reconstruct data model from results
4. Service find method must be modified to reconstruct data model from results

In the traditional architecture, any method that interacts with the data store through the use of the data model (likely most methods, excluding calls like count queries) will require modification if the data model must be split after the initial persistence tier architecture is implemented. The Diamond architecture requires the steps listed below.

1. Modify existing Converters to not pass through all attributes to conversion result
2. Add reconstitution logic to existing Reconciler

The advantage to the Diamond architectural approach is, by modifying the existing Converters to only pass through certain attributes on forward conversion (e.g., through predicates or other means), the split is accomplished for all usages of the Converters. The same is true of the reconstitution behavior in the Reconciler. A user of the framework is able to leverage the isolation of concerns provided by these architectural components. Although here we are evaluating a Service with only a few methods, updating a larger traditional-architecture-based system to support this type of change would quickly become overwhelming and error prone.

9.4.5 Per Data Store Data Model Attributes

Data stores vary widely in their methods of data storage. To effectively utilize a purpose-built data store, the data model is often required to adapt to the needs of the data store which, in some cases, requires the addition of data model attributes on a per data store basis. The MBox application, described in Chapter 8, was required to add additional attributes to its data model for the document store to provide full text search across a user's Emails. A traditional architecture would require the refactoring of the Service's save method to the workflow shown below.

1. Existing Service save method must be modified to add additional attributes to data model before Repository invocation
2. Existing save method must be modified to recognize attribute or disregard attribute when backwards converting repository result
3. Existing find method must be modified to recognize attribute or disregard attribute when backwards converting Repository result

The work required per data store to modify the traditional architecture to support additional data model attributes is similar to the split data model scenario. Any location in the Service that provides the data model to a Repository will need to add additional attributes before doing so. This implies modifying the Service anywhere that attributes could be used including all save and find methods. The Diamond architecture requires similar modifications but provides existing constructs to do so, thus isolating the Service from any changes.

1. Modify existing Converter to add additional attributes

Again, utilizing the Converter construct allows the developer to make a change in a single place without the need to refactor an existing Service method. This type of isolation allows for changes to be made throughout the persistence tier without affecting the rest of the application.

9.4.6 Concurrent Persistence

An efficient application will not want to wait to serially execute each persistence operation on every Repository. Significant performance improvements in polyglot persistence can be achieved by concurrently persisting across multiple data stores. Modifying the traditional architecture requires the enhancement of many aspects of the existing save method. The existing conversion and reconciliation code could be used unmodified, but the code responsible for invoking each Repository must be refactored.

1. Create a thread pool
2. Wrap the relational Repository call in a thread
3. Wrap the additional data store Repository call in a thread
4. Execute each threaded Repository call concurrently
5. Block until all Repository threads have returned results

Again, in the simple case, a single instance of this refactoring is not an abundant amount of work, but within a larger architecture, it could quickly become daunting. The Machinist approach is much simpler.

1. Switch the existing save Pipeline with the ConcurrentPipeline

As requirements change, the method in which Commands are executed can be easily changed using the Diamond architecture, allowing the developer to experiment with different styles of Command invocation to determine which style best suits their application's needs.

9.4.7 Tiered Persistence

Consecutive or concurrent persistence is adequate for many environments, but for some applications it is necessary to persist in tiers. The canonical use case for this type of behavior in a polyglot persistence environment is the need to ensure that an action has occurred before executing additional actions. In the context of persistence, an application may want to ensure that a data model is successfully written to a relational store before being persisted to additional stores. This provides a simplistic form of data integrity by forcing the data to be placed successfully in one data store before being written, possibly asynchronously and concurrently, to additional stores. In the traditional architecture, the Repository invocation behavior of the Service's save method must be modified to achieve this.

1. Group Repository calls and associated conversion code in arbitrary order
2. Write method of execution per group of Repository call groups (concurrent, consecutive, etc.)
3. At the successful execution of each group
    a. Convert results
    b. Reconcile to one result
4. If not the last group, pass result to next group and execute

The code required to accomplish this in a traditional architecture is largely dependent on how many data stores the architecture is utilizing. As we are focused here on the work required to change the invocation methods of the Repositories, using many data stores will result in large amounts of code that suffers from the drawbacks described in the previous persistence scenarios such as lack of code reuse and major refactorings. Using the Pipeline construct from the Diamond architecture requires minimal steps.

1. Switch existing Pipeline implementation for TieredPipeline
2. Add each Command or set of Commands to a tier as needed

The Machinist framework provides a TieredPipeline implementation that can be swapped for any existing Pipeline. Creating the TieredPipeline requires the developer to perform the additional work of grouping commands into sub-Pipelines

and adding them as tiers to the parent Pipeline. However, the extra effort expended on creating sub-Pipelines provides the application developer with a flexible approach to tiered persistence which has many uses including defining transactional boundaries and ensuring data integrity while simultaneously being conscious of performance.

9.4.8 Retrieve Fastest Available

Retrieving a result from all available data stores is important when the data model has been split across the data stores. However, in situations where the data has been duplicated instead of split, it is not necessary to wait for each store to return the same result. When the data model has been duplicated across many available data stores, the application would like to take the fastest result without concern for where it was retrieved. Accomplishing this in a traditional architecture would require the refactoring of the invocation of the available Repositories.

1. Create thread pool
2. Wrap each Repository call in a thread
3. Execute each threaded Repository call concurrently
4. Take the result of the first thread to return
5. Cancel all outstanding thread executions

The Machinist framework provides a Pipeline for this exact purpose; thus, there is again only one change required for the Diamond architecture.

1. Switch existing Pipeline for FlashPipeline implementation

The FlashPipeline implementation handles the details of non-blocking concurrent execution for the application developer. Changing an existing consecutive or concurrent Pipeline usage to a FlashPipeline involves changing a single line of code. Doing so ensures that, even if the data store that performs the fastest from request to request varies, application code remains unchanged.

9.4.9 Enable or Disable Data Store

As a business experiences growth, the software products they produce adapt incrementally. It is often helpful for application developers to test multiple deployments of similar data stores or the same data store with varying configurations, resulting in the need to quickly disable or re-enable the code paths responsible for utilizing a data store. In both architectures, the application developer has two relatively basic options; they can simply comment out the code path, or they can surround all code paths that are using the data store with a conditional check of a property that indicates if the data store should be used at the present time.

In the traditional architecture, commenting or removing all code used to interact with a data store (i.e., converting, Repository invocation, result reconciliation) would be error prone and time consuming. The better approach would be to wrap each of the functions with a conditional statement to facilitate the

enablement or disablement of a store through a property. This style of programming is not ideal and results in errors and a tremendous amount of work for the developer.

The Diamond architecture lends itself to such a usage scenario. Because Commands contain the requisite knowledge for interfacing with a specific data store, the developer can simply remove them from inclusion by the Pipeline by commenting out or removing the Commands. Additionally, because the Pipeline controls which Commands are given to the Runner for execution, a Pipeline can be decorated or extended to only pass Commands associated with enabled data stores to the Runner.

9.4.10 Results Reconciliation

In any application where multiple data stores are being utilized, the application is being constantly presented with results from each data store. Determining which results to return to the rest of the application can be a complex task. As additional data stores are added, the result reconciliation logic must be adapted in all locations where results are reconciled. Changing the reconciliation logic in a traditional architecture requires:

1. The modification of each method that handles results from multiple data stores: at a minimum, the find and save methods in our example

The Diamond architecture provides the Reconciler construct to encapsulate this behavior. Common behaviors, such as taking the first non-null result from a set of similar results, can be easily implemented by using the provided FirstNotNullResultReconciler. For this example scenario, the Diamond architecture requires a single change.

1. Add additional behavior to the existing Reconciler

This modification would change the reconciliation behavior of every Pipeline making use of the Reconciler.

9.5 Quantitative Features of the Evaluation

To evaluate the overall effort required by architectures for data store adoption it is helpful to provide a quantitative comparison between the two. To enable this comparison we again make use of the MBox application. The final application provides a full implementation of the EmailService interface for each architectural approach. The three-tier architecture provides an EmailService implementation that contains all the behavior required to save Emails, find Emails by a given User, and search Emails. The Diamond architecture implementation configures the necessary Pipelines for the same functionality outside of the EmailService and ensures the Pipelines are available to the RepositoryFacade, which in turn is invoked by the Diamond EmailService. Both implementations

utilize four disparate classes of data stores: relational, document, key-value, and columnar. The line of code counts for each approach is provided in Table 9-1.

Traditional		**Diamond**	
Component	**Lines of Code**	**Component**	**Lines of Code**
save(email:Email)	37	SavePipeline	34
save(emails:Collection)	71	SaveAllPipeline	16
findByUser(user:User)	40	FindByUserPipeline	16
search(query:String)	9	SearchPipeline	9

Table 9-1. Line of code counts for traditional (left) vs. Diamond (right) EmailService implementations utilizing four data stores.

The line of code counts for the traditional approach was derived from the ThreeTierEmailService implementation. The derived counts of the Diamond architecture were taken from the custom classes required to configure the necessary Pipelines and provide conversion of the MBox data model (i.e., EmailDocumentConverter, EmailKeyValueConverter, EmailColumnarConverter, EmailKeyValueKeyGenerator, EmailDocumentKeyGenerator). The total line of code counts for each approach is provided in Table 9-2 below.

Traditional	Diamond
157 LOC	75 LOC
52% line of code reduction	

Table 9-2. Total line of code disparity between the traditional and Diamond architectural approaches.

As is shown by the line of code comparison the Diamond architectural approach to polyglot persistence results in significant code savings for the persistence tier developer. When considered in combination with the analytical evaluation conducted previously in this chapter the Diamond architecture provides an architecture that minimalizes the complexities and overall effort associated with adoption of numerous specialized data storage technologies.

9.6 Evidence of Usefulness for Data Store Experimentation

When adopting new data stores a developer will often experiment with a variety of specialized technologies. Using the Diamond architecture the application developer can not only more easily add numerous data stores to their existing application but can also easily experiment with the available data stores once they have been integrated. Experimentation leads to a better understanding of how data stores compare to one another in terms of performance, durability, and flexibility as well as an overall understanding of how the greater application benefits from the utilization of multiple data stores. The MBox application greatly benefitted from this type of exploration, primarily through the utilization of the

TimeCommandDecorator, which provided empirical timings of data store operations. An example log output of a typical user workflow is given in Figure 9-8.

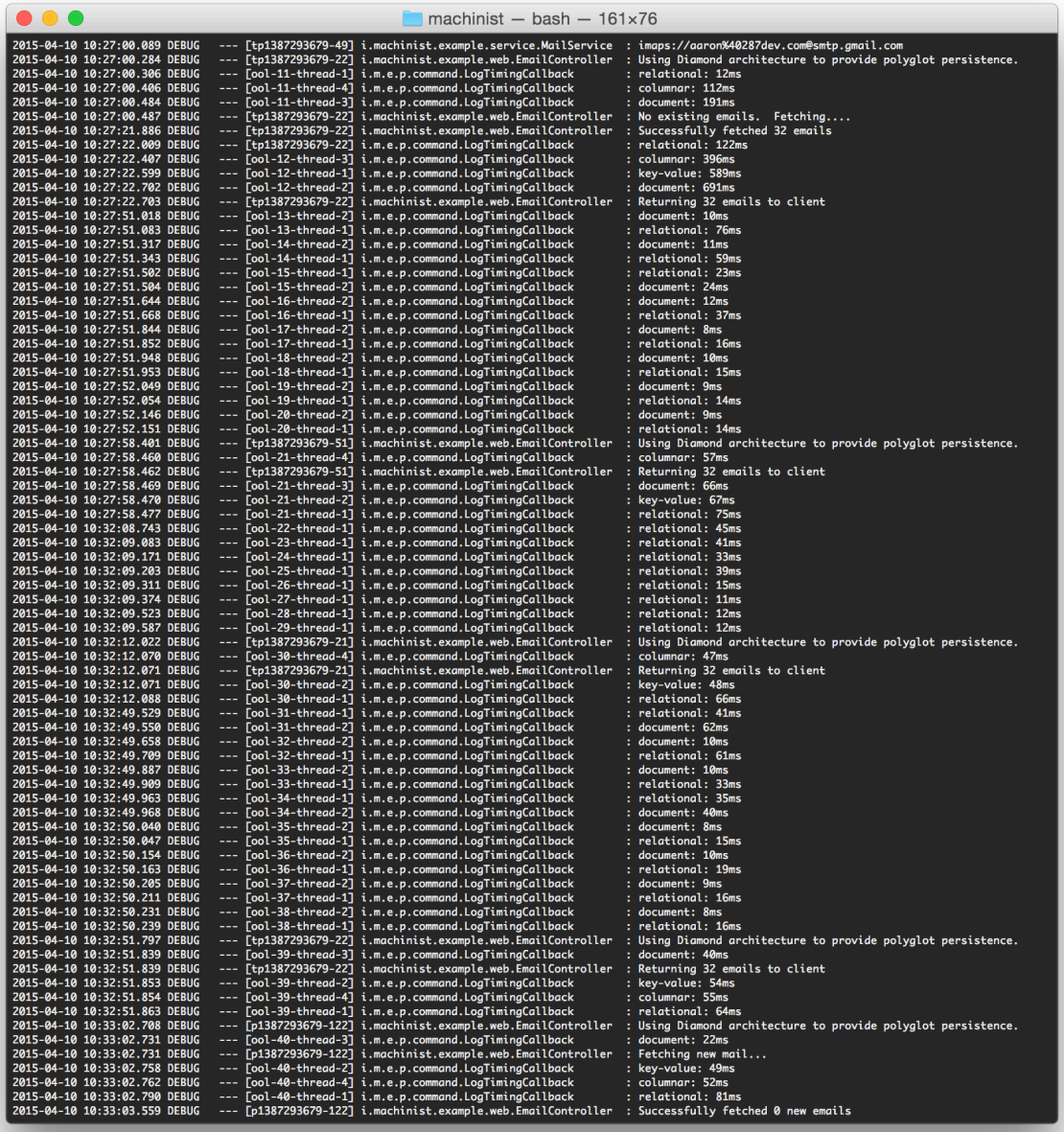

Figure 9-8. Log output of MBox application.

The log output shows the empirical timings, in milliseconds, of the four disparate data stores utilized by the MBox application (i.e., relational, document,

key-value, and columnar). The log output shows the user login, Email fetch from IMAP, Email persistence across all four data stores, and Email search. Beginning at *2015-04-10 10:27:58.477* the MBox application temporarily shutdown the document store to demonstrate the durability of the Diamond approach. During this time period the user continued to issue searches against the system that were fulfilled solely by the relational store. At *2015-04-10 10:32:49.550* the document store is restored to full functionality and begins to again service requests. Once the cache of the document store is again primed (*2015-04-10 10:32:51.839*) it can be seen that it returns all Emails for a User over 10 milliseconds quicker then the other three data stores. While these performance results are indeed interesting they additionally serve to highlight how valuable data store experimentation can be to understanding what scenarios and conditions result in the effective utilization of a data store within the persistence tier and what system attributes propagate throughout the greater application.

9.7 Results

The results of our evaluation show the many advantages the Diamond architecture has over the traditional approach. Although, the Diamond architecture does require some amount of upfront work to implement, it is similar in scope to developing for multi-threaded or multi-server environments. The results show that, through many common polyglot persistence scenarios, the implementation of the traditional three-tier architecture requires up to 50% more work than the implementation of the Diamond architecture utilizing the Machinist framework.

Additionally, the Diamond architecture limits greater than linear growth in persistence code by providing a one to one mapping between Commands and data stores. This strategy allows the developer to better prepare the persistence tier of their application for increasing numbers of data stores. The major lessons learned from our evaluation are:

1. Isolating all persistence responsibilities within the Service tier is a poor approach that
    a. Results in too much responsibility for a component that is intended to focus solely on business logic.
    b. Ruins testability of the Service.
2. Isolating the persistence workflows within Commands allows for reuse by limiting behavior duplication in Services.
3. Conversion code (e.g., data model conversion, selective attributes, additional attributes) is difficult to keep isolated in a Service method. Using a Converter encapsulates these responsibilities and ensures single points of code change.
4. Methods of Repository invocation are diverse and subject to change. Changing the Pipeline implementation is far easier than refactoring a Service method.

5. Diamond architecture Service and Facade methods are simplistic, often only invoking the run method of a Pipeline. This facilitates testability and maintains the focus of the Service on business logic
6. Creating new Functions (i.e., key generators and callbacks), Converters, Runners, Reconcilers, and Pipelines is easily accomplished through composition and Decorators. The same is not possible through Service extensions.

The level of effort required to provide persistence without the Diamond architecture and Machinist framework for two data stores has been demonstrated to be substantial. Adding additional stores using the traditional architecture would also be substantial and would quickly become unmanageable. Maintaining a Service that must coordinate for *N* data stores is incredibly complex; however, the Diamond architecture was designed specifically to fulfill this task. The level of effort to add an additional store is the same as described in our evaluation scenarios using the Machinist framework. Through this design, we have provided an architecture capable of adapting to varying numbers and types of data stores in a given system over time. Furthermore, the types of small, isolated changes required by the Machinist framework to work with a new data store significantly reduces the costs typically associated with task. This reduction, in turn, significantly increases the accessibility of working with multiple data stores. Since the cost to entry has been significantly reduced, developers can feel more confident that "the leap" to

experiment with or adopt a new data store is one that is manageable and safe as opposed to error-prone and dangerous. Finally, the architecture developed provides the scalability, availability, and durability required to meet the expectations of current users of large-scale, heterogeneous data-intensive systems.

CHAPTER 10: Future Work

The movement away from one-size-fits-all systems and towards systems composed of specialized data storage technologies will grow and result in future research concerning architectural patterns for large-scale heterogeneous systems. A future direction for work in this research area is data model decomposition and reconstitution. Currently, the application developer is responsible for designing how to decompose the data model for effective storage by each data store. The application developer is also responsible for implementing the details of data model reconstitution after retrieval from multiple stores. These complexities could be isolated from non-expert users or generated automatically based on best practices or other information inferred from the code base. As an example, a successful implementation would allow the application developer to declaratively specify what data store to persist the data model to without needing to also determine how best to store it. Such a system would drastically increase accessibility of these technologies by limiting the accidental complexities associated with data model conversion. Research is needed to understand how to ensure that data model Converters understand data model versioning and the underlying data store migrations that support them. An automated data model Converter generator could greatly aid these tasks.

An additional research direction would focus on the creation of a federated or imperial query API for data stores. It is overwhelming to learn the details of all of

the available data store query technologies because they vary drastically from high-level languages (e.g., SQL, CQL, Cypher, Gremlin) to extensive programming paradigms (e.g., MapReduce). A query framework that could provide an extensible abstraction for underlying query technologies would revolutionize accessibility for data store technologies. This research could further be extended to enable the automatic detection and execution of a query on the best system for the task. Enabling this type of query intelligence would minimize the expertise required to implement the individual Repositories leveraged by the Diamond architecture.

Utilizing multiple data stores results in a variety of data integrity issues for the application developer. Although these issues exist in most modern software companies today, increasing the number of places that data is stored is likely to intensify these complexities. Providing future direction on how to deal with these issues in highly heterogeneous environments would have a major impact on this style of application architecture. Ideally, the architectural constructs provided by this dissertation could serve as the foundation for this work. A series of Pipelines and Commands could be developed to help the persistence architect understand where data was placed and, in the most ideal scenario, where data *should* be placed. This type of tool support will provide incredible value to organizations adopting heterogeneous architectures. The Commands and Pipelines provided through the Machinist framework are useful but basic in comparison to their full potential.

A final research area is the challenge of data integration at scale. Integrating data from disparate sources is difficult and at the core of the promises

made by “Big Data”. Stonebraker suggests the following techniques for data integration at scale: human involvement, transformations, entity consolidation, cleaning, and wrappers [69]. This is likely another area where one-size does not fit all, and data integration technologies will need to utilize many techniques to achieve success. The Data Tamer System [79] which works to automate the data integration process, is one such example. Future work in architectural patterns will need to provide for data integration systems that process large amounts of disparate data sources and produce canonical forms of data for storage. Leveraging the work of the Data Tamer System, intelligent and even run-time configured Converters, Commands, and Pipelines could be built to help adapt integration strategies to multi-data store environments. Integrating these technologies with heterogeneous architectures is important because the canonical representation of an object is likely to be formed through an iterative process as new knowledge is discovered over time. Aspects of the architecture that interact directly with the data model may need to be notified of changes made by the data integration system. For example, an intelligent data model decomposition strategy may need to incorporate knowledge of a new attribute or type added by the data integration system. Research in this area could result in systems capable of storing and integrating vast and disparate data sources without the need for time-consuming development tasks.

CHAPTER 11: Conclusions

The contribution of this dissertation is the implementation of an architecture for the enablement of polyglot persistence. The Diamond architecture provides a series of constructs that allow a traditional enterprise architecture (i.e., three-tier architecture) to adopt an unknown or varying number of data stores. In this way, we contribute to the field of software engineering the design, implementation, and evaluation of an architecture useful for polyglot persistence. The resulting software enables application developers to adopt and/or abandon large-scale data stores without overwhelming complexity. Minimizing complexities in polyglot persistence makes this style of architecture accessible to more software engineers which, in turn, will aid the transition to a multitude of systems with a heterogeneous style of architecture that are not built solely upon one-size-fits-all technologies. The experience gained through the design and implementation of this polyglot persistence framework will enable future recommendations on how to best develop tools and techniques for this new architectural era.

REFERENCES

[1] S. Few, “Visual Business Intelligence – If Big Data Is Anything at All, This Is It,” *Perceptual Edge*, 18-Aug-2014. [Online]. Available: http://www.perceptualedge.com/blog/?p=1955. [Accessed: 14-Mar-2015].

[2] K. Zickuhr and A. Smith, “Digital differences,” Pew Internet, Washington, DC, Apr. 2012.

[3] “Gmail - Free Storage and Email from Google.” [Online]. Available: https://www.gmail.com/intl/en/mail/help/about.html. [Accessed: 14-Mar-2015].

[4] “Welcome to Facebook - Log In, Sign Up or Learn More,” *Facebook*. [Online]. Available: https://www.facebook.com/. [Accessed: 14-Mar-2015].

[5] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” University of California, Irvine, 2000.

[6] D. Crockford, “JSON,” *JSON*, 2002. [Online]. Available: http://www.json.org/. [Accessed: 20-Mar-2015].

[7] E. T. Bray, “RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format,” Mar-2014. [Online]. Available: http://tools.ietf.org/html/rfc7159. [Accessed: 14-Mar-2015].

[8] M. Cox and D. Ellsworth, “Application-controlled demand paging for out-of-core visualization.,” in *Proceedings of the 8th conference on Visualization*, Phoenix, AZ, USA, 1997, p. 235–ff.

[9] G. E. Moore, “Cramming More Components Onto Integrated Circuits,” *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, Jan. 1998.

[10] “VLDB Endowment Inc.” [Online]. Available: http://www.vldb.org/. [Accessed: 16-Mar-2015].

[11] “Star schema - Wikipedia, the free encyclopedia.” [Online]. Available: http://en.wikipedia.org/wiki/Star_schema. [Accessed: 16-Mar-2015].

[12] D. J. Abadi, S. R. Madden, and N. Hachem, “Column-stores vs. row-stores: how different are they really?,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 967–980.

[13] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, “OLTP through the looking glass, and what we found there,” 2008, p. 981.

[14] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, and others, “C-store: a column-oriented DBMS,” in *Proceedings of the 31st international conference on Very large data bases*, 2005, pp. 553–564.

[15] “MySQL :: The world’s most popular open source database.” [Online]. Available: http://www.mysql.com/. [Accessed: 14-Mar-2015].

[16] “PostgreSQL: The world’s most advanced open source database.” [Online]. Available: http://www.postgresql.org/. [Accessed: 14-Mar-2015].

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[18] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, “Dremel: interactive analysis of web-scale datasets,” *Proceedings of the VLDB Endowment*, vol. 3, no. 1–2, pp. 330–339, 2010.

[19] A. Schram and K. M. Anderson, “MySQL to NoSQL: data modeling challenges in supporting scalability,” in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, 2012, pp. 191–202.

[20] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[21] “Google BigQuery - Fully Managed Big Data Analytics Service — Google Cloud Platform.” [Online]. Available: https://cloud.google.com/bigquery/. [Accessed: 14-Mar-2015].

[22] “Apache Parquet.” [Online]. Available: http://parquet.incubator.apache.org/. [Accessed: 14-Mar-2015].

[23] “HBase – Apache HBase™ Home.” [Online]. Available: http://hbase.apache.org/. [Accessed: 14-Mar-2015].

[24] “DataStax - NoSQL Cassandra Database | Fastest, Most Scalable.” [Online]. Available: http://www.datastax.com/. [Accessed: 14-Mar-2015].

[25] “Redis.” [Online]. Available: http://redis.io. [Accessed: 18-Mar-2015].

[26] “memcached - a distributed memory object caching system.” [Online]. Available: http://memcached.org/. [Accessed: 14-Mar-2015].

[27] “Ehcache | Performance at Any Scale.” [Online]. Available: http://ehcache.org/. [Accessed: 14-Mar-2015].

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” in *ACM SIGOPS operating systems review*, 2003, vol. 37, pp. 29–43.

[29] “Welcome to Apache™ Hadoop®!,” *Apache Hadoop*, 26-Feb-2015. [Online]. Available: http://hadoop.apache.org/. [Accessed: 14-Mar-2015].

[30] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” in *ACM SIGOPS Operating Systems Review*, 2007, vol. 41, pp. 205–220.

[31] “Apache Lucene - Welcome to Apache Lucene.” [Online]. Available: http://lucene.apache.org/. [Accessed: 14-Mar-2015].

[32] “Apache Solr -.” [Online]. Available: http://lucene.apache.org/solr/. [Accessed: 14-Mar-2015].

[33] “Elasticsearch: RESTful, Distributed Search & Analytics | Elastic.” [Online]. Available: https://www.elastic.co/products/elasticsearch. [Accessed: 14-Mar-2015].

[34] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, p. 107, Jan. 2008.

[35] “MongoDB.” [Online]. Available: http://www.mongodb.org/. [Accessed: 14-Mar-2015].

[36] E. Bentley, “MongoDB mocked after posting “100GB Scaling Checklist” - JAXenter,” *JAXenter*, 03-Oct-2013. [Online]. Available: http://jaxenter.com/mongodb-mocked-after-posting-100gb-scaling-checklist-106822.html. [Accessed: 14-Mar-2015].

[37] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[38] “Giraph - Welcome To Apache Giraph!” [Online]. Available: http://giraph.apache.org/. [Accessed: 14-Mar-2015].

[39] “Titan: Distributed Graph Database,” *Titan: Distributed Graph Database*. [Online]. Available: http://thinkaurelius.github.io/titan/. [Accessed: 14-Mar-2015].

[40] “Neo4j, the World’s Leading Graph Database.” [Online]. Available: http://neo4j.com/. [Accessed: 14-Mar-2015].

[41] L. A. Meyerovich and A. S. Rabkin, “Empirical analysis of programming language adoption,” 2013, pp. 1–18.

[42] S. Kleinschmager, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik, “Do static type systems improve the maintainability of software systems? An empirical study,” in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, 2012, pp. 153–162.

[43] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik, “An empirical study of the influence of static type systems on the usability of undocumented software,” *ACM SIGPLAN Notices*, vol. 47, no. 10, p. 683, Nov. 2012.

[44] P. Petersen, S. Hanenberg, and R. Robbes, “An empirical comparison of static and dynamic type systems on API usage in the presence of an IDE: Java vs. groovy with eclipse,” 2014, pp. 212–222.

[45] S. Hanenberg, “An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time,” in *ACM Sigplan Notices*, 2010, vol. 45, pp. 22–35.

[46] Brooks, “No Silver Bullet Essence and Accidents of Software Engineering,” *Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987.

[47] Y. Sverdlik, “Google Replaces MapReduce With New Hyper-Scale Cloud Analytics System,” *Data Center Knowledge*, 25-Jun-2014. [Online]. Available: http://www.datacenterknowledge.com/archives/2014/06/25/google-dumps-mapreduce-favor-new-hyper-scale-analytics-system/. [Accessed: 14-Mar-2015].

[48] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy, “Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance,” *Software Engineering, IEEE Transactions on*, vol. 28, no. 6, pp. 595–606, 2002.

[49] B. Unger and W. F. Tichy, “Do Design Patterns Improve Communication? An Experiment with Pair Design,” in *Proceedings of International Workshop Empirical Studies of Software Maintenance*, 2000.

[50] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissedes, *Design patterns: elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1994.

[51] M. Stonebraker and U. Cetintemel, "' One size fits all': an idea whose time has come and gone," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, 2005, pp. 2–11.

[52] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era:(it's time for a complete rewrite)," in *Proceedings of the 33rd international conference on Very large data bases*, 2007, pp. 1150–1160.

[53] R. Kallman, Y. Zhang, J. Hugg, D. J. Abadi, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, and M. Stonebraker, "H-store: a high-performance, distributed main memory transaction processing system," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, Aug. 2008.

[54] "Real-Time Analytics Platform | Big Data Analytics | MPP Data Warehouse." [Online]. Available: http://www.vertica.com/. [Accessed: 14-Mar-2015].

[55] "In-Memory Database, NewSQL & Real-Time Analytics | VoltDB." [Online]. Available: http://voltdb.com/. [Accessed: 14-Mar-2015].

[56] M. Fowler, "PolyglotPersistence," *Martin Fowler*, 16-Nov-2011. [Online]. Available: http://martinfowler.com/bliki/PolyglotPersistence.html. [Accessed: 19-Mar-2015].

[57] P. J. Sadalage, *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Upper Saddle River, NJ: Addison-Wesley, 2013.

[58] S. Leberknight, "Polyglot Persistence," *Altamira*, 15-Oct-2008. [Online]. Available: https://www.altamiracorp.com/blog/employee-posts/polyglot-persistence. [Accessed: 14-Mar-2015].

[59] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich, "Safely composable type-specific languages," in *ECOOP 2014–Object-Oriented Programming*, Springer, 2014, pp. 105–130.

[60] T. Haselmann, G. Thies, and G. Vossen, "Looking into a REST-based universal API for Database-as-a-Service systems," in *Commerce and Enterprise Computing (CEC), 2010 IEEE 12th Conference on*, 2010, pp. 17–24.

[61] F. Gessert, S. Friedrich, W. Wingerath, M. Schaarschmidt, and N. Ritter, "Towards a Scalable and Unified REST API for Cloud Data Stores," in *Jahrestagung der Gesellschaft für Informatik*, Stuttgart, Germany, 2014, pp. 723–734.

[62] H. Lim, Y. Han, and S. Babu, "How to Fit when No One Size Fits.," in *CIDR*, 2013, vol. 4, p. 35.

[63] K. M. Anderson and A. Schram, “Design and implementation of a data analytics infrastructure in support of crisis informatics research: NIER track,” 2011, p. 844.

[64] M. Fowler, *Patterns of enterprise application architecture*. Boston: Addison-Wesley, 2003.

[65] E. Hieatt and R. Mee, “P of EAA: Repository,” *Martin Fowler*. [Online]. Available: http://martinfowler.com/eaaCatalog/repository.html. [Accessed: 14-Mar-2015].

[66] R. C. Martin, *Agile software development: principles, patterns, and practices*. Upper Saddle River, N.J: Prentice Hall, 2003.

[67] D. Garlan and M. Shaw, “An Introduction to Software Architecture,” in *Advances in Software Engineering and Knowledge Engineering*, vol. I, V. Ambriola and Tortora, Eds. New Jersey: World Scientific Publishing Company, 1993.

[68] “Multitier architecture,” *Wikipedia, the free encyclopedia*. 01-Mar-2015.

[69] M. Stonebraker, “What Does ‘Big Data’ Mean (Part 4)? | blog@CACM | Communications of the ACM,” *Communications of the ACM*, 14-Mar-2013. [Online]. Available: http://cacm.acm.org/blogs/blog-cacm/162095-what-does-big-data-mean-part-4/fulltext. [Accessed: 14-Mar-2015].

[70] “google/guava · GitHub.” [Online]. Available: https://github.com/google/guava. [Accessed: 14-Mar-2015].

[71] “Java Software | Oracle.” [Online]. Available: https://www.oracle.com/java/index.html. [Accessed: 14-Mar-2015].

[72] “Spring Framework,” 2015. [Online]. Available: http://projects.spring.io/spring-framework/. [Accessed: 14-Mar-2015].

[73] “A JavaScript library for building user interfaces | React.” [Online]. Available: http://facebook.github.io/react/. [Accessed: 14-Mar-2015].

[74] “Bootstrap · The world’s most popular mobile-first and responsive front-end framework.” [Online]. Available: http://getbootstrap.com/. [Accessed: 14-Mar-2015].

[75] “Spring Data.” [Online]. Available: http://projects.spring.io/spring-data/. [Accessed: 16-Mar-2015].

[76] “datastax/java-driver · GitHub.” [Online]. Available: https://github.com/datastax/java-driver. [Accessed: 14-Mar-2015].

[77] “google-gson - A Java library to convert JSON to Java objects and vice-versa - Google Project Hosting.” [Online]. Available: https://code.google.com/p/google-gson/. [Accessed: 16-Mar-2015].

[78] “jsoup Java HTML Parser, with best of DOM, CSS, and jquery.” [Online]. Available: http://jsoup.org/. [Accessed: 16-Mar-2015].

[79] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu, “Data Curation at Scale: The Data Tamer System.,” in *CIDR*, 2013.

APPENDIX A – Mbox Persistence Implementations

```
public class ThreeTierEmailService implements EmailService {
    private EmailRelationalRepository emailRelationalRepository;

    public ThreeTierEmailService(EmailRelationalRepository emailRelationalRepository) {
        this.emailRelationalRepository = emailRelationalRepository;
    }

    @Override
    public Email save(Email email) {
        return emailRelationalRepository.save(email);
    }

    @Override
    public Iterable<Email> save(Iterable<Email> emails) {
        return emailRelationalRepository.save(emails);
    }

    @Override
    public Iterable<Email> findByUser(User user) {
        return emailRelationalRepository.findByUser(user);
    }

    @Override
    public Iterable<Email> search(String q) {
        return emailRelationalRepository.findBySubjectContainingIgnoreCaseOrContentContainingIgnoreCase(q,q);
    }
}
```

Three-tier architecture, single data store, EmailService implementation for MBox application.

```
public class EmailRepositoryFacade {
    Pipeline<Email, Email> savePipeline;
    Pipeline<User, Iterable<Email>> findByUserPipeline;
    Pipeline<String, Iterable<Email>> searchPipeline;
    Pipeline<Iterable<Email>, Iterable<Email>> saveAllPipeline;

    @Autowired
    public EmailRepositoryFacade(Pipeline<Email, Email> savePipeline,
                                 Pipeline<User, Iterable<Email>> findByUserPipeline,
                                 Pipeline<String, Iterable<Email>> searchPipeline,
                                 Pipeline<Iterable<Email>, Iterable<Email>> saveAllPipeline) {
        this.savePipeline = savePipeline;
        this.findByUserPipeline = findByUserPipeline;
        this.searchPipeline = searchPipeline;
        this.saveAllPipeline = saveAllPipeline;
    }

    public Email save(Email email) {
        return savePipeline.run(email);
    }

    public Iterable<Email> save(Iterable<Email> emails) {
        return saveAllPipeline.run(emails);
    }

    public Iterable<Email> findByUser(User user) {
        return findByUserPipeline.run(user);
    }

    public Iterable<Email> search(String query) {
        return searchPipeline.run(query);
    }
}
```

Diamond architecture RepositoryFacade implementation used by MBox.

```
@Bean(name = "emailSavePipeline")
public Pipeline<Email, Email> emailSavePipeline() {
    // Relational with no conversion
    Command<Email, Email> relationalCommand = new SaveCommand<>(new NoOpConverter<>(), emailRelationalRepository::save);

    DiamondPipeline<Email, Email> emailSavePipeline = new SimplePipeline<>();
    emailSavePipeline.addCommand(relationalCommand);

    return emailSavePipeline;
}

@Bean(name = "emailSaveAllPipeline")
public Pipeline<Iterable<Email>, Iterable<Email>> emailSaveAllPipeline() {
    // Relational with no conversion
    Command<Iterable<Email>, Iterable<Email>> relationalCommand = new SaveAllCommand<>(new NoOpConverter<>(), emailRelationalRepository::save);

    DiamondPipeline<Iterable<Email>, Iterable<Email>> emailSavePipeline = new SimplePipeline<>();
    emailSavePipeline.addCommand(relationalCommand);

    return emailSavePipeline;
}

@Bean(name = "emailFindByUserPipeline")
public Pipeline<User, Iterable<Email>> emailFindByUserPipeline() {
    // Find all emails for a user
    Command<User, Iterable<Email>> relationalCommand = new FindAllCommand<>(new NoOpConverter<>(), new NoOpConverter<>(), emailRelationalRepository::findByUser);

    DiamondPipeline<User, Iterable<Email>> emailFindByUserPipeline = new SimplePipeline<>();
    emailFindByUserPipeline.addCommand(relationalCommand);

    return emailFindByUserPipeline;
}

@Bean(name = "emailSearchPipeline")
public Pipeline<String, Iterable<Email>> emailSearchPipeline() {
    Command<String, Iterable<Email>> relationalCommand = new FindAllCommand<>(new NoOpKeyGenerator<>(), new NoOpConverter<>(), new Function<String, Iterable<Email>>() {
        @Nullable
        @Override
        public Iterable<Email> apply(String input) {
            return emailRelationalRepository.findBySubjectContainingIgnoreCaseOrContentContainingIgnoreCase(input, input);
        }
    });

    DiamondPipeline<String, Iterable<Email>> searchPipeline = new SimplePipeline<>();
    searchPipeline.addCommand(relationalCommand);

    return searchPipeline;
}
```

The Pipeline implementation for a dual data-store MBox application.

APPENDIX B - Machinist Class Diagram