MySQL to NoSQL
Data Modeling Challenges in Supporting Scalability

Aaron Schram
Department of Computer Science
University of Colorado
Boulder, Colorado, USA
aaron.schram@colorado.edu

Kenneth M. Anderson
Department of Computer Science
University of Colorado
Boulder, Colorado, USA
kena@cs.colorado.edu

Abstract

Software systems today seldom reside as isolated systems confined to generating and consuming their own data. Collecting, integrating and storing large amounts of data from disparate sources has become a need for many software engineers, as well as for scientists in research settings. This paper presents the lessons learned when transitioning a large-scale data collection infrastructure from a relational database to a hybrid persistence architecture that makes use of both relational and NoSQL technologies. Our examples are drawn from the software infrastructure we built to collect, store, and analyze vast numbers of status updates from the Twitter micro-blogging service in support of a large interdisciplinary group performing research in the area of crisis informatics. We present both the software architecture and data modeling challenges that we encountered during the transition as well as the benefits we gained having migrated to the hybrid persistence architecture.

Categories and Subject Descriptors D.2.11 [**Software Engineering**]: Software Architectures; D.2.13 [**Software Engineering**]: Reusable Software.

General Terms Design, Reliability.

Keywords *crisis informatics; NoSQL; data modeling; scalability; software architecture; software infrastructure*

1. Introduction

Collecting, integrating and storing large amounts of information is quickly becoming a necessity among software engineers in industry, as well as by scientists in research settings. Crisis informatics [8] is one research area in which this need has never been greater. Crisis informatics studies how information and communication technology are used in emergency and hazard response. An emerging branch of this discipline investigates how members of the public make use of social media and other forms of computer-mediated communication to aid one another during times of mass emergency [7]. The analysis of this type of data relies heavily on a robust and scalable data collection infrastructure. The ephemeral nature of the data (e.g. Twitter status updates) requires collection to be done in real-time and with uncompromising reliability. Since Fall 2009, we have been engaged in the design and development of this type of data collection infrastructure via our work in Project EPIC (Empowering the Public with Information in Crisis) [7].

We have designed and developed this infrastructure in an iterative fashion, implementing it initially using a standard, three-tier web architecture. We then expanded the system to expose layers of services that could be leveraged by other research groups—as well as within our own team—to isolate the complexities of interacting with our data model and permit reuse of our collection tools. We also exposed a layer of web services to allow geographically distributed mobile clients better access to our services and data. While these are modern software engineering practices, we were eventually faced with the challenge of scaling our persistence tier due to the enormous amounts of data produced by even a single mass emergency event [1], leading to the need for us to pursue additional techniques and technology.

We report here on the current state and future direction of the data collection infrastructure we are developing to support research in crisis informatics. Software engineering has begun to play an even more critical role than we could have initially imagined in this domain. The data collected and stored by our system are vital to the research of our colleagues in such areas as natural language processing (NLP), systems and security, internet policy, and human centered computing (HCC). For these groups to extract

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPLASH'12 October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1563-0/12/10…$15.00.

representative samples and to make statistically significant research claims they rely almost entirely on the accurate and timely collection of social media data by our system.

This dependence puts constraints of scalability and robustness at the forefront of system design, often receiving priority over simple feature requests involving our analysis tools. These constraints put a great deal of pressure on our software engineering research team because the work they do on the design and development of the infrastructure is often concealed from the rest of the research group. Having accurate and complete data sets are often thought of as a given by research colleagues, who have become accustomed to working with publicly available data sets such as the "Brown Corpus" [6]. Our goal is to meet this expectation and provide them with near real-time access to data sets of similar quality. In designing a system to support these highly demanding needs we are tasked with providing a system that is fault-tolerant in a number of areas involving highly distributed and scalable systems—characteristics not typically required when developing software prototypes in service of software engineering research.

2. Background

Project EPIC's data collection infrastructure is composed of multiple services that can be leveraged individually or in composition. These services abstract away the inherent complexities present in collecting, storing, and analyzing status messages from the Twitter micro-blogging platform. Each service exposes a well-defined set of interfaces that can be utilized as-is or extended to meet the varying needs of a particular client. Project EPIC shares these services across multiple web and command line applications running on separate machines and in separate Java virtual machines allowing each application to create, modify, and query entities in a consistent way on a shared data store [1].

Like many web architectures, our first choice for a storage solution was a traditional relational data store. For our purposes we chose the open source relational database management system, MySQL.[1] It offered the best set of features for our project's needs and is widely used amongst some of the biggest web companies in existence, including Facebook and Flickr. We also heavily relied upon the popular object relational mapping (ORM) framework, Hibernate, which is known to integrate well with MySQL. The focus at the time of the initial system design was on quickly providing our colleagues with a set of features that would be immediately useful in collecting the datasets they needed to conduct their own research. Scalability and reliability were considered but, as with many initial projects, it was difficult to anticipate the scope of our storage needs.

Although it is well known that traditional relational databases can be made to scale through techniques such as the *sharding* of data amongst a set of machines or the acquisition of often-expensive hardware, we report here on our investigation into a class of technologies referred to as NoSQL. The requirements placed on our infrastructure by our research colleagues are those where current NoSQL technologies excel, such as high availability and scalability. In this paper, we report on the challenges facing our software engineering group as we transition from a MySQL-only persistence architecture to a hybrid model which makes use of both MySQL and NoSQL.

3. NoSQL

NoSQL is a term used to describe a broad class of technologies that provide an alternative approach to data storage compared with traditional relation database management systems. Often these technologies provide the user with low-cost solutions to the problems of high availability and scalability at the loss of a flexible query language, making ad hoc access of the data generally more difficult. The term NoSQL was initially meant to make this clear by standing for "*no SQL*" but the term has more recently been updated to mean "*not only SQL*" as very few production systems eliminate relational databases altogether.

The current offerings of these technologies are heavily influenced by Google's Bigtable [2] and Amazon's Dynamo [4] systems. Current NoSQL systems include: HBase, MongoDB, Riak, Voldemort, Cassandra, Memcached, Tokyo Cabinet, Redis, and CouchDB. Each has its own specialties and they are often differentiated by how they can scale to handle extremely large datasets.

In contrast to a relational database, a NoSQL datastore attempts to group similar data together on disk to limit the number of seeks required to manipulate data to improve access times. The data models provided by NoSQL systems often force the user to structure their data in easily distributable multidimensional maps (across a cluster of machines). Access to these key/value maps is provided by APIs that expose traditional, easily understood map operations (i.e. get, put, contains, and remove). This approach has the added benefit of allowing the segments of the data to be read and processed in parallel using a MapReduce [3] framework, such as Hadoop. Project EPIC has chosen to migrate the data collection aspect of its software architecture to the Apache open source project Cassandra [5]. This decision was driven in part due to the growing popularity and high development activity on the Cassandra project as well as the availability of a newly released product from DataStax, which bundles together Cassandra, Hadoop, and Hive. Hive offers a SQL-like syntax for analyzing data stored in Hadoop; it is an attempt to provide a query lan-

[1] Note: All of the specific technologies mentioned in this paper are either well known or easily found via an Internet search.

guage to those projects that prefer the query functionality provided by relational storage technologies over the Bigtable-influenced, non-relational APIs of NoSQL.

3.1 Cassandra

Cassandra was originally a project developed at the popular social media site, Facebook, to serve data to hundreds of millions of users during peak usage levels [5]. Specifically, it was designed to fulfill the storage requirements of the Inbox Search feature, which allowed users to quickly search all the contents of their message inbox. It was released to the open source community in July 2008, later transitioning into an Apache Incubator project in March 2009. It is now in commercial use at a variety of companies, such as Digg, Reddit, and Twitter.

Cassandra is a mix of techniques taken from Bigtable and Dynamo, essentially running the Bigtable data model [2] on top of the Dynamo fully-distributed architecture [4]. It possesses the most favorable traits of both its predecessors resulting in a fault tolerant, decentralized system with rich data modeling capabilities. It is fully distributed and even allows for replication to take place between data centers. Cassandra was an ideal fit for the research goals of Project EPIC because it directly attacks the complex problems of data replication, scalability, and 100% uptime while allowing our existing data models to be represented (albeit not without significant work in making the transition—see Section 6).

4. System Architecture

Project EPIC's existing architecture (see Fig. 1) is a production-ready system that includes multiple web and command line applications. In its two years of deployment it has collected over 2B disaster-related status messages covering numerous mass emergency events that occurred in 2010-2012 while maintaining 99% uptime. This reliability has been achieved through careful design of the infrastructure's service tier, which is responsible for abstracting the inherent complexities involved in data collection, persistence, and aggregation from disparate sources [1].

The service tier relies on the persistence tier to handle all interactions with the data store. Initially these interactions were limited to MySQL, taking responsibility for isolating and abstracting away the complexities of managing database create, read, update, and delete (CRUD) operations, transactions, and queries. These interactions are accomplished through the use of a callback that wraps logic defined in the service tier that will eventually be executed in the persistence tier, often within the context of a database transaction. The configuration of the persistence tier is done through the use of property files, allowing clients to specify, at run-time, such settings as database hosts, ports, and names. Indeed, it would even be possible for a particular client to configure services to communicate with completely separate data stores. This can be easily accomplished because all of Project EPIC services are *wired* at run-time through the use of the Spring dependency injection framework. This allows the service consumer to configure each service independently and, in some cases, even swap out configuration settings during program execution. The use of the Spring framework also enables a high degree of modularity, increasing testability through mock objects frameworks (e.g. JMock), and allowing our team to develop the infrastructure incrementally.

A strong advantage of using Project EPIC's service tier is the ability to work with a rich set of domain objects. These domain objects model many of the commonly encountered artifacts of crisis informatics research, providing getter and setter methods for each available property. An example domain object is the Tweet object, which exposes such attributes as the user that generated the tweet, the time the tweet was created, and the text of the tweet. If used in conjunction with the appropriate service, all fields are automatically populated for the consumer of the service regardless of the source of the tweet. Currently a tweet may be retrieved directly from Twitter over HTTP, or from any relational database, or a Lucene index. The client requires no specialized knowledge for how to retrieve a tweet matching their needs from each potential data source, which otherwise would require the knowledge of many different APIs; instead the client is simply returned a fully populated object graph that fulfills the constraints of the query issued by the client.

The flexibility of this architecture allows for the addition of a number of different persistent storage solutions without changing the client software. As long as the contract provided by the service to the client remains valid, how the persistence tier chooses to store the information should be irrelevant to the client. The Project EPIC architecture then allows for the addition of a high availability storage solution, in this case Cassandra, to be introduced incrementally into the system without breaking current clients. This transition would not have been possible if our clients had been interacting directly with our data stores rather than the abstract interfaces of our service and persistence tiers.

5. Data Model

One of the key design changes that must be accomplished for a successful transition from relational technology to the use of NoSQL techniques and technology is the transformation of the system's existing data model. Indeed, this can be a challenging task. Software engineering students are taught to model the world as objects that interact with one another via messages and to think of relationships between objects in terms of one-to-one, one-to-many, many-to-one, and many-to-many. A single object is seldom valuable without well-defined relationships with other objects. This style of design is well suited for transferring an object

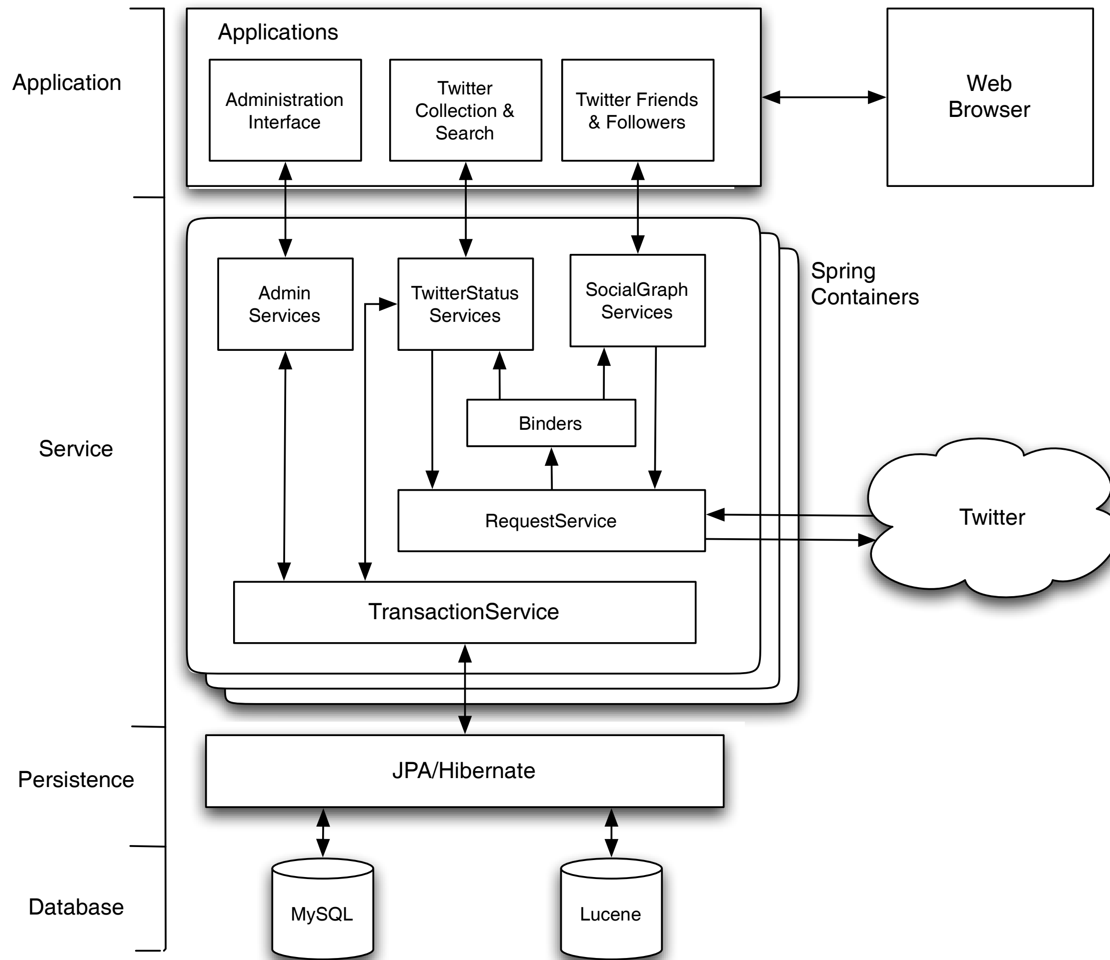

Figure 1. The Project EPIC software architecture before the addition of NoSQL technologies [1]. As a testament to its flexible, abstract design, the architecture remains largely the same after the transition. Existing services can continue to access data initially persisted in MySQL and Lucene. In the persistence layer, new infrastructure is added to manage access to a Casandra cluster (see Fig. 7). New services can then be created to access the new persistence infrastructure.

model into a relational database. The relational style allows a software engineer to model objects as database tables and the relationships between objects as primary and foreign keys that link the tables together. These relationships can then be exploited by issuing queries via SQL.

Today there is a direct, well-traveled path for software engineers to take requirements and develop complex models that are easily represented in traditional relational data stores. Requirements can be read and easily translated into UML, which is used to model complex relations and actions. Current UML tools can even generate the source code for classes from UML diagrams. That code can then be annotated using frameworks like Hibernate to automatically generate the necessary database tables to persist objects without requiring the developer to have any knowledge of the underlying relational database. The tasks required to take an arbitrary data model from a set of requirements to a fully functioning persistence tier have been abstracted fantastically well. These are very valuable tools to have available to modern day software engineers as it allows a software engineer to focus on the application being built not on the complex details required to create and manage a database. It is now standard practice to rely on ORM frameworks for handling all interactions with the database, enabling the client to interact only with objects and their relationships. Indeed, Project EPIC's persistence and service tiers and the models they share are based on these same *best practices*.

Project EPIC's data model is a rich set of plain old Java objects (POJOs). POJOs are often referred to as Java beans, implying that they conform to the convention that each object exposes a set of properties that will have similarly

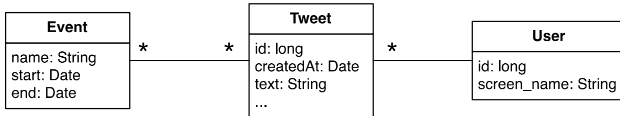

Figure 2. Project EPIC's simplified object model for collecting data from Twitter during a disaster event.

named getter and setter methods also available. As an example a Person object with a *name* property will, by convention, expose two methods: *getName()* and *setName()*. This allows other Java frameworks to make assumptions about how to interact with this object. Using run-time reflection it becomes possible to access or define the value of any object property easily using the Java bean convention. The Java Persistence API (JPA), which defines a specification for automatically persisting Java objects, makes use of these POJOs. All of Project EPIC's domain objects are marked with Java 5.0 annotations that allow them to be persisted using the JPA with little effort on behalf of the development team. This technology allowed our team to move from whiteboard to a fully functioning persistence system in a short amount of time simply by following JPA best practices. However—shortly after initial deployment—it became clear to the team that although we were able to develop and deploy a system quickly, we were ignorant of details that cause performance bottlenecks in these systems.

JPA provides the developer with a simple set of annotations to define how to persist object properties and object relationships. Objects are often automatically discovered via the *@Entity* annotation and database tables, column names, and column types are often created via reflection of the object's properties. Relationships between objects can be persisted through the use of the *@OneToOne, @OneToMany, @ManyToOne,* and *@ManyToMany* annotations. These annotations are extremely powerful as they enable a developer to model a variety of complex object relationships; incredibly, these annotations are able to automatically generate complex primary keys, foreign keys, and join tables that are needed to model these relationships in the underlying database. Although a great and necessary asset, these annotations can produce a variety of performance problems forcing the relational database into inefficient operations to support an arbitrary model being utilized in a separate tier of the application.

For example, in Fig. 2, we present a simplified version of the object model we use when collecting data from Twitter during a disaster event: *Users generate tweets; tweets can be associated with one or more Events.* Collecting data from Twitter during a mass emergency event is the primary focus of the Project EPIC data collection infrastructure. Collection must be done in real-time with little to no errors during the designated event window. Tweets collected during an event remain associated with that event. As an example, during March of 2011, Project EPIC was monitoring several events including the Japan earthquake; the continuing conflicts of the "Arab Spring"; the Christchurch, New Zealand earthquake; and a local wildfire near Boulder, Colorado. Due to the constraints of the Twitter Streaming API, our infrastructure returned tweets that match any of up to 400 distinct search terms. The infrastructure must analyze each tweet returned by that API and associate it with its corresponding event(s). Once data collection for an event ends, the corresponding tweets can be exported to a variety of formats or moved from production to other machines to free up storage space for new events.

There is a many-to-many relationship between Event and Tweet and there is a one-to-many relationship between User and Tweet. If we annotated the Java objects that represent Event, Tweet and User with the @ManyToMany and the @OneToMany annotations, an object-relational manager such as Hibernate would automatically produce the join tables and foreign keys in a relational data store that would allow, e.g., a developer to traverse from an Event object to a collection containing all of that event's Tweet objects. The problem is that Hibernate will create that collection whether it contains 100 Tweets or 75 million (the size of our Haiti dataset). In the latter case, the client program will sit blocked as Hibernate pulls back the information needed to instantiate 75 million instances of the Tweet class and will eventually crash as the system runs out of memory.

There are other issues that can occur with the use of ORM technologies at scale but this example illustrates the essential problem: ORM frameworks can not scale to truly large datasets as the relationships between objects will cause the framework to pull information into memory unnecessarily. For instance, the simple act of collecting a new tweet from Twitter and adding it to an existing event can cause an object-relational manger to pull into memory all of the tweets associated with an event if the system is not engineered to guard against such automatic behavior. This automatic behavior is incredibly useful; it unfortunately just does not scale to large datasets.

Modeling relationships is the most difficult aspect of using ORM frameworks. As we have just seen, application-level logical abstractions often do not translate into efficient storage representations. This makes it very difficult to build a highly available and scalable application using these technologies, especially when following what are widely held as good software engineering principles such as object-oriented design heuristics and data normalization. What we have found is that in order to enable scalability many of these software engineering best practices must be employed *outside of the persistence tier*. In the world of high scalability, data is often replicated and distributed amongst hundreds or thousands of machines. *Normalized, relational data have no place here*, and this in turn leads to design choices that deal with the complexities associated

Row Key 1	Column Name 1	•••	Column Name N
	Value	•••	Value
•••			
Row Key N	Column Name 1	•••	Column Name N
	Value	•••	Value

Figure 3. Casandra's Column Family: Each row maps to a potentially different set of columns.

with highly scalable and distributed systems that do not mesh well with the classical software engineering view of the world. But, these tradeoffs typically enable true and easy horizontal scalability, something that is difficult to achieve with a traditional relational database system.

6. Making the Transition

The data collection aspect of Project EPIC's software infrastructure was identified as the first piece of the architecture to be transitioned to NoSQL technology. Now that we have billions of tweets to store and analyze, the benefits of making this transition are many. Data replication, horizontally-scalable storage, and high availability are characteristics that are difficult to achieve with our initial design based on relational databases but are straightforward to achieve with NoSQL. Additionally, our team was not made up of professional system administrators yet more and more of our daily tasks were moving from software engineering research to monitoring, maintaining, and scaling our existing storage solution. Moving our events and tweets into a NoSQL data store fulfills many of our storage needs and promises to reduce the maintenance tasks that were taking us away from our research. In addition, the transition presented several software engineering and data modeling challenges with implications for software architecture and the design of scalable software systems.

To make the transition, we needed to ensure that the existing services would not break simply because our data moved from a relational database to a NoSQL platform. The current service tier allows clients to retrieve all tweets collected during an event. An example would be a client asking for all tweets associated with the March 2011 Japan earthquake. The service tier also allows for the retrieval of a set of tweets by user regardless of event association. Finally, our research colleagues require the ability to retrieve tweets by specifying a date range, since it is often necessary to focus on a subset of the tweets collected during an event. We will address each one of these concerns individually and how these requests require a specific approach to object/data modeling within a NoSQL data store.

6.1 Events and Tweets

As mentioned above, Project EPIC has decided to adopt Cassandra, a NoSQL technology developed and maintained by the Apache Software Foundation. Cassandra directly addresses the need to have our data replicated across multiple machines to enable high availability; in addition, it provides a flexible mechanism for modeling the objects within our application domain. Cassandra makes use of a data model similar to that described in the work on Google's Bigtable [2]. In particular, Cassandra provides modeling concepts similar to Bigtable's *rows*, *columns*, and *column families*. Cassandra also exposes another type known as a *super column*; however super columns have been shown to impose a 10-15% performance penalty on reads and writes, so we have decided not to use them as our current object modeling tasks do not require their use.

The first step in transitioning our existing relational model is denormalizing our data. In a relational model, a many-to-many relationship would require a join table, tying events to tweets, allowing a single tweet to be associated with many events. Modeling the same type of relationship in Cassandra can be done in two ways. First, it is possible to maintain a join table representation as a column family (in essence maintaining the data in a relational form). Another option, more suited to our purposes, is to simply store the representation of the tweet in multiple places. Although this duplication may seem like a poor choice as it goes against the practices of normalizing (i.e. not duplicating) data in the relational style, in NoSQL the assumption is that "storage is cheap" and one should not shy away from storing duplicate copies of an artifact when necessary.

Cassandra makes use of *column families* to store its data (see Fig. 3). A column family consists of *rows* that point to many *columns*. Each *column* has a *column name* and a *column value*. This structure is essentially a hash table of hash tables. Note: there is no requirement that each row store the same columns. One row can have columns x, y, and z storing a string, an integer and a date while the next row has columns a, b, and c storing an integer, a float and a string.

We will be using the column family data structure to model the relationship between Events and Tweets. For our purposes we will create one column family called *Events*. This column family will use event data as its *row key* and tweet data for its columns. A *row key* in Cassandra is a unique key that allows the client to index into the column family and retrieve columns. Row keys can be of any number of types including—but not limited to—strings, dates, and numbers. With respect to Fig. 2, we would like to retrieve tweets based on event names. It would also be possible to generate unique event ids and maintain the mappings between the event names and the unique ids somewhere else. Indeed, this is something that we do in our production system but for the purposes of illustration we will simply use a unique event name as our row key. Cassandra can then use these keys to distribute and/or replicate our data across a cluster of machines.

Event Name 1	Tweet Id 1	•••	Tweet Id N
	JSON	•••	JSON
•••			
Event Name N	Tweet Id 1	•••	Tweet Id N
	JSON	•••	JSON

Figure 4. Events Column Family.

The next step in converting our existing relational model will be storing the tweets themselves. Each tweet is given a unique numeric identifier from Twitter. This unique id will be used as the column name for our event columns. This decision merits some explanation. Cassandra is a schema-less data store, which means that it enforces no requirements that the rows contain similar columns, as mentioned above. One example where this would occur would be in storing a set of users and their attributes. The unique username would serve as the row key and the associated columns would store the attributes allowing the retrieval of attribute values by attribute name. As such, a client could ask for a user's date of birth by specifying the row key, perhaps *jsmith*, and the attribute of interest, perhaps *date_of_birth*. This would return to the client the user's date of birth. Our storage of events and tweets will not work like this since tweets are delivered with a large set of metadata that changes over time (as Twitter evolves the services and information it provides to its developers). We take advantage of Cassandra's lack of schema enforcement to store a new tweet in every column of an event row, using the tweet's unique id as the column name. Thus, each row in our Events column family will contain a different number of columns based on the number of tweets collected for that event (which can number in the tens of millions or more). The value for each column will be the raw JSON object that Twitter delivered to us at the time of collection (see Fig. 4). A column value can be of any number of types. In this example JSON can simply be stored as a string or it may be advantageous to store the JSON as a series of bytes in a compressed format, requiring compression and expansion for all writes and reads. These operations could be isolated within the service or persistence tiers and may limit the amount of time spent on IO resulting in increased performance.

These data modeling choices allow us to map our existing object model into Cassandra while providing us with the most flexibility for analyzing the data at a later point in time. By storing the full JSON object, no information about the tweet is lost. However, since each tweet contains a complete copy of its user's metadata, this approach results in a large amount of data duplication since rather than storing the information about a user only once (as we would in a relational model) we now are storing the entire user object for each tweet that a user contributed to the event.

Despite this duplication, this approach to storing the data is in fact better than our current relational structure because, as mentioned above, the attributes of a tweet returned to us by Twitter are often subject to change without notice. With the relational model, you are either forced to store the tweet as a BLOB or CLOB (thus losing the very power that the relational approach was trying to provide) or you must spend time updating your schema to store the new metadata and then migrate your entire data store to the new schema. With NoSQL, the promise of relational functionality was never offered in the first place but in exchange it offers a data store that provides horizontal scalability ("need more data, just add another machine to your cluster") and high availability through replication across the cluster. However, this technique is not without its faults. Storing the raw JSON data of the tweet as the column value limits Cassandra's ability to provide useful results to queries against the column's values. These concerns are addressed in the next section.

6.2 Tweets and Attributes

In some situations, our collaborators on Project EPIC may need to run queries against the attributes of the tweets that our data collection infrastructure has captured during an event, across multiple events, or even independent of an event altogether. A common case—and one our infrastructure supported before this migration—is retrieving all tweets that have been collected for a given Twitter user. If a user then wanted to further limit the results to tweets only generated during a specific event, they could easily filter the results by date or some other attribute.

To support this functionality in Cassandra, our Events column family is not enough. It only implements traversal of our original many-to-many relationship from events to tweets; details concerning attributes—such as which user created a particular tweet—are hidden away in the JSON object stored as a column value in that column family, which is an opaque data type from Cassandra's point of view. In order to go the other direction—from tweets to events—or to search tweets directly we will need to define a new column family in Cassandra.

There are multiple ways to model this column family for this particular use case. One option would be to follow the event example set forth in the previous section using the screen_name of the user as the row key in place of the event name. Although this would work, it is not as versatile as we might like. It limits us to searching only on the screen_name and not via other attributes such as the text of the tweet, the day it was created, or its latitude and longitude (if it was a geocoded tweet).

To enable this type of search, we make use of the *secondary indexing* feature provided by Cassandra. Secondary indexes allow the client to execute very simple queries against column values that can be indexed by Cassandra.

Tweet Id 1	createdAt	text	screen_name	•••
	Date	String	String	•••
•••				
Tweet Id N	createdAt	text	screen_name	•••
	Date	String	String	•••

Figure 5. Tweets Column Family.

This feature makes it worth our while to model each tweet as a row (using its unique tweet id as the row key) in which each column corresponds to an attribute of the tweet that we care about. We can then ask Cassandra to create a secondary index on any of the columns that we know will be used to locate tweets independent of an event; for this use case, we will index the *screen_name* column of each row. The resulting column family can be seen in Fig. 5.

Using this representation a tweet and its attributes can now be retrieved via a lookup on the tweet's unique id or by looking up all tweets that match a given value for a column that carries a secondary index. Indexing the screen_name attribute will allow a client to request all tweets for a given user. As an example all the tweets for a user with a known screen_name *jsmith* could be retrieved by asking for all tweets where *"screen_name = 'jsmith'"*. In addition, in order to implement the other direction of our many-to-many relationship between Events and Tweets, we could simply add one (or more) columns that store the event names (or event ids) of the events each tweet is associated with. Then, it would not matter what query brought us to a particular tweet, since pulling the event information from the appropriate column(s) will allow us to access information about that tweet's events from other column families. However, performance may suffer slightly in this case because the read operations of the disk may not be sequential like they will be for the Events column family.

6.3 Time Slicing Events

An important aspect of data collected by Project EPIC is the matter of temporality: When was a tweet collected? When did an event start? How active was a particular user on this day? Many of our research colleagues base their analysis techniques on time windows and timelines. As such our data model must support the ability to partition data by time. The fidelity of these time windows can range from one day to the full duration of an event. To support that functionality using NoSQL technology, we will store the data in the smallest increment required, one day, and "roll-up" to a desired duration. Doing so will enable a high degree of flexibility for use by our existing services.

To support time slicing of tweets across events we will need to segment our data by the day the tweet was collected from Twitter. To accomplish this we will need a mapping of days to tweets collected on those days. This could be done via the addition of a new column family. That approach, however, would require the service tier to join together the data from the new column family and the Events column family, which is not ideal for our purposes. Until now we have omitted discussing a common "gotcha" in NoSQL data modeling. Indeed, it is a problem present even in our previous discussion of the Events column family.

A single event in the Events column family may contain an extremely large number of tweets (in the 10s of millions). This results in a single row key with an extremely large number of columns. Generally this is considered a bad practice. When Cassandra attempts to replicate keys and their associated data (columns) around a cluster of machines all of the key's data is replicated as a unit. This can result in long delays or timeouts when adding additional nodes to the cluster. Our Events column family, as shown in Fig. 4, could encounter scalability problems during events of long duration. Adding the requirement of supporting the partitioning of the data by time actually enables and ensures linear scalability. By decreasing the number of columns stored with each key the amount of data that must be moved with each key when it is replicated across the cluster is also decreased, resulting in faster key replication. In fact, data reads may also be more efficient because the client may now specify the exact data they are interested in receiving instead of requesting all the data available. To enable this new approach, we must slightly modify our Events column family via the use of a *composite row key*.

Our initial row key for our Events column family was simply a unique event name. This key mapped a single event to all the tweets associated with the event. Since the row key for our Events column family is simply a string, we will make a small modification to the string to support the time partitioning requirements. The new row key will be a composite string starting with the unique event name and ending with the day the tweets in this row were collected; the two elements of a composite key are separated—by convention—with a colon (":").

Given a date range and an event name, we can now construct the required composite keys in our service tier to retrieve the desired set of tweets. This change can be seen in Fig. 6. This simple change now allows for the full range of time slicing operations required by our research colleagues and also greatly enhances scalability by preventing the rows of the Events column family from becoming too large to efficiently replicate across a cluster. In addition, this scheme is easily extended to handle time windows of finer granularity by splitting existing rows into smaller rows and extending the key to also include a timestamp.

7. After the Transition

The previous section discussed various data modeling challenges that software engineers will encounter when their scalability needs force a transition away from relational technology and towards NoSQL technology. However, this transition is not one of completely replacing the former with the latter—hence the "not only SQL" expansion for the NoSQL term—but rather adding NoSQL technologies into an existing software infrastructure providing it with a hybrid persistence architecture.

After the addition, of NoSQL technologies to Project EPIC's data collection infrastructure, its software architecture now takes the form shown in Fig. 7. The existing persistence-related components—our transaction service, Hibernate, MySQL, and Lucene—are all still present and all services and applications that previously made use of them still function with their previous levels of scalability and reliability [1]. Now, however, an additional API called Hector exists within the persistence tier and is now available to any service within our service tier. Hector is a Java wrapper for Cassandra's native API, which makes use of the Apache project Thrift. Thrift handles interactions with the services of a Cassandra cluster and Hector provides access to Thrift via a Java API.

Project EPIC's software architecture was well-suited for a transition from a relational-only persistence architecture to a hybrid persistence architecture due to its use of the Spring dependency injection framework. In Fig. 1, all of the services shown in the service tier have abstract interfaces that get implemented by particular concrete classes. All interactions between services occur via the abstract interfaces and rely on Spring to plug-in concrete implementations at run-time to achieve desired functionality.

So, while it was not made explicit in Fig. 1, it is *not* the TwitterStatusService that talks to the TransactionService but the MySQLTwitterStatusService that talks to the ProjectEPICTransactionService at run-time. These latter two classes are concrete implementations of the previous two abstract interfaces. MySQLTwitterStatusService knows how to make use of the transaction service and Hibernate to store and access tweets in MySQL. Meanwhile the Twitter Collection & Search application that lives within the application tier knows only about the abstract TwitterStatusService interface and knows nothing about MySQL and has no dependence on it.

Event Name 1: Day W	Tweet Id 1	•••	Tweet Id N
	JSON	•••	JSON
•••			
Event Name 1: Day X	Tweet Id 1	•••	Tweet Id N
	JSON	•••	JSON
•••			
Event Name N: Day Y	Tweet Id 1	•••	Tweet Id N
	JSON	•••	JSON
•••			
Event Name N: Day Z	Tweet Id 1	•••	Tweet Id N
	JSON	•••	JSON

Figure 6. Events Column Family partitioned by day.

As a result of this carefully designed software architecture, the transition to Cassandra within the infrastructure is easily accommodated. We simply needed to create a concrete implementation of the TwitterStatusService called CassandraTwitterStatusService that encapsulates the knowledge of how to create the appropriate column families in Cassandra (via Hector) to store events, tweets and twitter users after they have been retrieved from Twitter by the RequestService (also an abstract interface with multiple concrete implementations) shown in Fig. 1. Since CassandraTwitterStatusService is hidden behind the abstract TwitterStatusService interface, the existing Twitter Collection & Search application runs *unmodified* on top of this new implementation and now has the ability to collect and search over significantly larger datasets than before. The difficult part in making this transition was the data modeling challenges discussed in the previous section; the actual transition due to the abstract and flexible nature of our software architecture was straightforward.

Of course, in making this transition there exists the need to create and configure a Cassandra cluster, and that is a non-trivial task. However, developers who are in contexts that have acquired datasets sufficient in size to require the use of NoSQL technologies are often in settings that have access to the system administration expertise and resources needed to acquire the hardware, configure the cluster and install the relevant software. While our software infrastructure can be run on a single machine, the advantages of NoSQL technologies cannot be truly realized until they are running on a sizeable cluster of machines.

For instance, for popular queries, Twitter can deliver 50-60 tweets per second, twenty-four hours per day, via its Streaming API. That translates to ~5M tweets per day. With our old infrastructure, that rate would cause our memory-based queues to fill with 1000s of unprocessed tweets as Hibernate Search struggled to keep pace. Now, on our cluster, Cassandra's ability to provide sub-millisecond inserts allows us to process 50-60 tweets per second with no need to store tweets in a queue waiting for our persistence mechanism to update its records. We are now confident that we can handle the 100+ tweets per second rate (~8.6M tweets per day) that we experienced while collecting data during the 2011 Japan Earthquake.

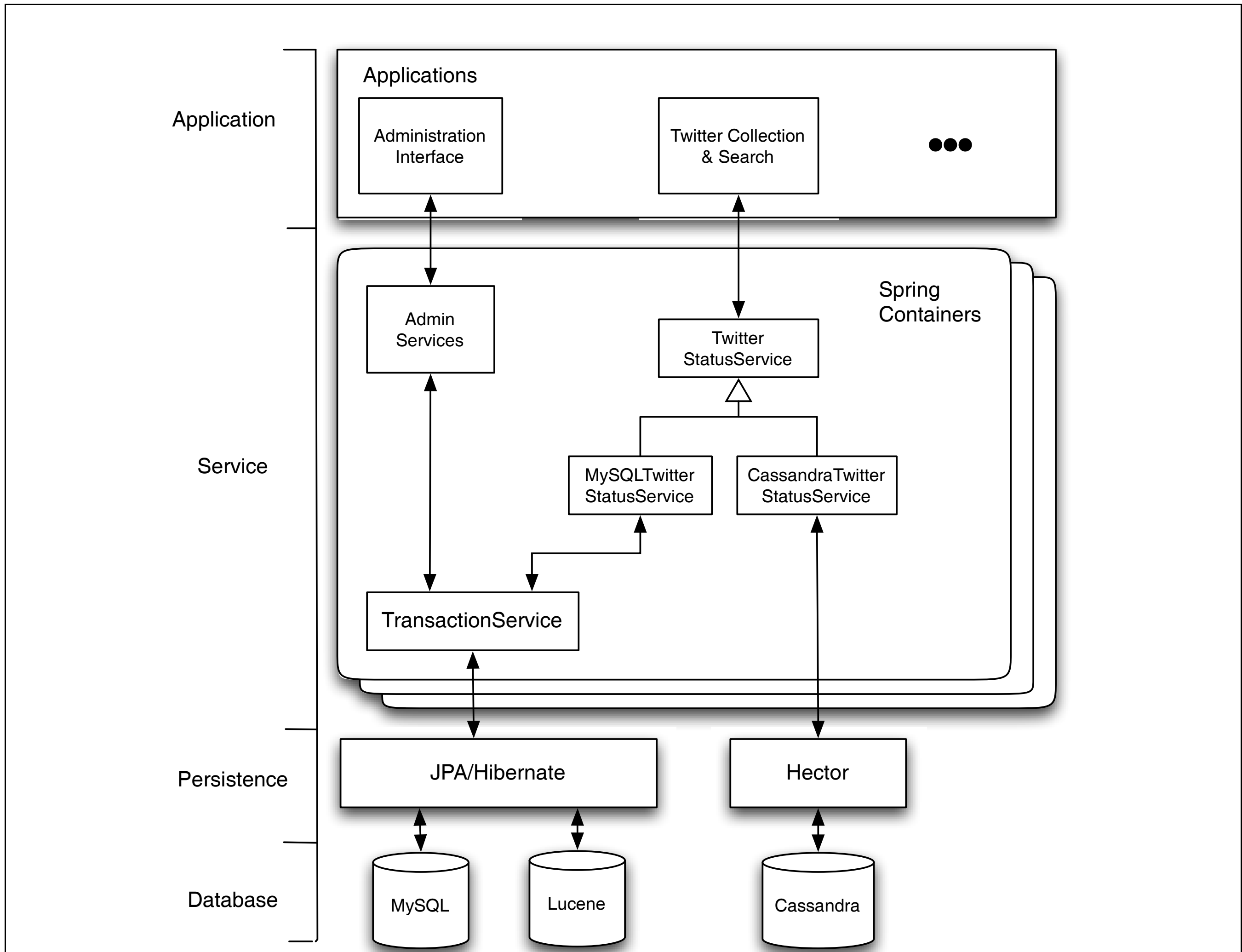

Figure 7. The architecture of the Project EPIC software infrastructure after the addition of Cassandra, a NoSQL data store. This diagram elides details present in Fig. 1 to focus on two important aspects. The first is that none of the services that previously depended on our initial persistence layer have to change. They can continue to store and access data in MySQL and Lucene. The second is that those services that require the scalability and availability guarantees of NoSQL can add an additional implementation of their service interface that stores and accesses data in Cassandra. Due to our use of Spring, we can now flexibly plug-in the service implementations that will meet a wide range of scalability constraints.

Indeed, during the days leading up to and including the first week of the 2012 London Olympics (14 days), our new infrastructure collected 40 million tweets (98.2GB) on a 4 node Cassandra cluster, collecting on 712 user accounts and keywords. At one point during that collection, our system received a burst of tweets that caused our in-memory queue to expand to 40,623 tweets; after the spike, our system cleared that queue in less than two minutes by processing the tweets at a rate of 491 tweets per second. We are quite pleased with the improvements our Cassandra-based system is providing during the collection of truly large-scale “mass convergence” events.

It is important to note that due to the design of our software architecture and our use of Spring, we have the ability to deploy the Project EPIC software infrastructure in a wide variety of configurations: from a single researcher storing Twitter data in JSON files (we have a service not shown in Fig. 1 and Fig. 7 that can persist tweets to a single file) to a research group running the infrastructure on a single powerful server (as Project EPIC did for its first two years) to an even larger research group running a hybrid persistence architecture on a large cluster of machines (as Project EPIC does today). The use of Spring by our software infrastructure allows each of these configurations to be realized in a straightforward manner via the editing of a few configura-

tion files. It is important to note the value of this flexibility as it is sometimes overlooked. The nature of the work done by Project EPIC is inherently multi-disciplinary involving a wide variety of individuals and technical skillsets. Providing an infrastructure capable of enabling any individual involved in the research activities to collect and analyze relevant data is a significant accomplishment.

Having performed this work to add Cassandra to the persistence tier of our software infrastructure, we gain significant options for advancing the research goals of Project EPIC, especially with respect to real-time analysis. Up until this point even providing simple statistics such as the number of data points available in our data sets had proven time consuming and troublesome.

In particular, the move to Cassandra now allows us to develop services that will analyze and index our datasets in parallel. For instance, there is an implementation of Lucene that is built on top of Cassandra. As we collect additional datasets, we will be able to use that variant of Lucene to make sure our index stays up-to-date even as the total number of tweets moves into the billions. Lucene also enables a variety of information retrieval techniques to be applied to our data at scale. Using Lucene to generate term vector representations of our data has proven valuable in applying similarity metrics to tweets, which allows for easy identification of retweets and exploration of the search space.

Finally, there are analysis tools that are being developed for Cassandra that have now become available for use in advancing Project EPIC's research goals. DataStax, for instance, has developed a product that combines Hive, Hadoop and Cassandra such that Hadoop operates directly against Cassandra, effectively mimicking Hadoop's native file system HDFS. Hadoop's MapReduce operations can now be applied directly against data stored in Cassandra column families enabling the use of other Hadoop compatible frameworks to help us scale.

We will be leveraging the Apache project Mahout to apply distributed machine learning and data mining algorithms to our large datasets, enabling a variety of clustering and classification capabilities. The Apache project Hive enables an SQL-like language called QL that can interact directly with data stored in Hadoop. By using DataStax, the data stored in Cassandra can be accessed by Hadoop and, transitively, by Hive giving back some of the advantages of working with a structured query language lost by transitioning to NoSQL. Hive allows our research team to view and store our data in new and interesting ways without requiring even the knowledge of our service tier, translating every query into a set of MapReduce jobs that are executed in parallel across the Cassandra cluster. This can drastically improve the execution time of complex queries, the results of which can be stored and made available to applications through our service tier.

These features will allow us to provide an experimental browsing interface into the data being stored in our Cassandra cluster by our service tier. The set of services created to support this interface can then be used by Project EPIC researchers to understand the types of information that can be extracted from our datasets; this, in turn, would aid the design and implementation of new services and applications that would be directly useful to the research of our NLP and HCC collaborators on Project EPIC.

8. Conclusions

Software engineers are increasingly encountering development situations in which it is straightforward to collect large amounts of data. While NoSQL technologies provide a means for scaling beyond the capabilities of relational databases, they bring a wealth of data modeling challenges that make it difficult for developers to understand how best to migrate data previously stored using a relational schema to the schema-less world of NoSQL. In addition, NoSQL platforms are not meant to replace relational databases, placing pressure on software engineers to create software infrastructures that adopt hybrid persistence architectures that contain both types of technologies. Without the right software architectural approach, these hybrid architectures are difficult to achieve such that the resulting infrastructure is maintainable and straightforward to evolve. In this paper, we present the approach that Project EPIC has adopted to meet the significant data modeling challenges that occur when migrating from a relational approach to the NoSQL approach as well as the software architecture challenges of producing a flexible and extensible software infrastructure. Our data collection infrastructure can now scale in a straightforward manner to handle the increasingly large sets of data that we collect and analyze during times of crisis in support of our crisis informatics research agenda.

Acknowledgments

This material is based upon work sponsored by the NSF under Grant IIS-0910586.

References

[1] Anderson K. M. & Schram A. Design and Implementation of a Data Analytics Infrastructure in Support of Crisis Informatics Research: NIER Track. In *33rd International Conference on Software Engineering,* pp. 844–847. May 2011.

[2] Chang F., Dean J., Ghemawat S., Hsieh W. C., Wallach D. A., Burrows M., Chandra T., Fikes A. & Gruber R. E. Bigtable: A Distributed Storage System for Structured Data. In *7th Symposium on Operating Systems Design and Implementation,* pp. 205–218. Nov. 2006.

[3] Dean, J. & Ghemawat, S. Mapreduce: Simplified Data Processing on Large Clusters. *Communications of the Association of Computing Machinery,* 51(1):107-113. Jan. 2008.

[4] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. & Vogels, W. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS Operating Systems Review,* 41(6):205-220. Oct. 2007.

[5] Lakshman, A. & Malik, P. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review,* 44(2):35-40. Apr. 2010.

[6] Malmkjaer, K,The Linguistics Encyclopedia. 2nd Edition. Routledge, 688 pages, 2004.

[7] Palen L., Anderson K. M., Mark G., Martin J., Sicker D., Palmer M. & Grunwald D. A Vision for Technology-Mediated Support for Public Participation & Assistance in Mass Emergencies & Disasters. In *Proceedings of the 2010 ACM-BCS Visions of Computer Science Conference,* pp. 8:1—8:12. Edinburgh, United Kingdom, 2010.

[8] Palen L. & Liu S. B. Citizen Communications in Crisis: Anticipating a Future of Ict-Supported Participation. In *ACM Conference on Human Factors in Computing Systems,* pp. 727–736. San Jose, CA, USA, Apr. 2007.