2015 IEEE/ACM 1st International Workshop on Big Data Software Engineering

Embrace the Challenges: Software Engineering in a Big Data World

Kenneth M. Anderson
Department of Computer Science
University of Colorado,
Boulder, CO 80309–0430
Email: ken.anderson@colorado.edu

***Abstract*—The design and development of data-intensive software systems—systems that generate, collect, store, process, analyze, query, and visualize large sets of data—is fraught with significant challenges both technical and social. Project EPIC has been designing and developing data-intensive systems in support of crisis informatics research since Fall 2009. Our experience working on Project EPIC has provided insight into these challenges. In this paper, we share our experience working in this design space and describe the choices we made in tackling these challenges and their attendant trade-offs. We highlight the lack of developer support tools for data-intensive systems, the importance of multidisciplinary teams, the use of highly-iterative life cycles, the need for deep understanding of the frameworks and technologies used in data intensive systems, how simple operations transform into significant challenges at scale, and the paramount significance of data modeling in producing systems that are scalable, robust, and efficient.**

I. Introduction

The field of software engineering (SE) faces new challenges in a time when it has never been easier to generate and/or collect large volumes of data. These challenges span a wide range of issues both technical and social, each with its own set of complicated trade-offs. Combined, these challenges represent a difficult landscape for software engineers—and SE researchers—to navigate. What problems represent research challenges and which are merely the result of complex choices with unclear or difficult to predict consequences?

Project EPIC [1] is a research project that investigates how people make use of social media during times of crisis [2]; it is an area that, as a result, requires the collection of large amounts of social media data (consisting mainly of Twitter data), and the design, development, and deployment of scalable and robust software infrastructure. Since Fall 2009, we have been tackling design challenges related to big data and data-intensive software systems. In that time, we have created two production data-intensive systems—EPIC Collect and EPIC Analyze—that are used on a daily basis by our research collaborators to collect and analyze data sets consisting of tens of millions to hundreds of millions of tweets.

As a result, we have firsthand experience with the SE challenges that exist when working on big data [3] and data-intensive software systems. In this paper, we briefly describe our work in this area and then identify the challenges that we consider most significant in this space and provide insights into the problems that are encountered and the trade-offs that can be made when tackling them. We also indicate the choices we made and why we made them to aid SE researchers and practitioners dealing with similar issues. Our lessons learned touch on the lack of developer support tools for data-intensive systems, the importance of multidisciplinary teams, the use of highly-iterative life cycles, the need for understanding the frameworks used in data intensive systems, the way in which simple operations transform into significant challenges at scale, and the need to get data modeling right to produce systems that are scalable, robust, and efficient.

II. Project EPIC

The SE researchers working on Project EPIC had to solve two issues intrinsic to work in big data: how to effectively collect and store ever-growing data sets and how to analyze that data efficiently. A key theme emerged quickly in our work on these two issues—*embrace failure*—each time we applied a technique and found that it did not scale or it failed to work reliably, we would learn something valuable that would then guide our subsequent design efforts.

Our attempts to store significant amounts of Twitter data moved from simple systems managing flat files to *n*-tier web applications using relational technologies [4] to our eventual solution of making use of NoSQL technology [5] on a cluster of machines [6]. Along the way we had to wrestle with data modeling approaches that a) first attempted to filter the data we stored to b) then migrating to storing the data in normalized form to c) abandoning normalization and storing unmodified copies (and, in the case of user objects, duplicate copies) of the data Twitter provides for use later in analysis.

With each switch in persistence technology we would gain an appreciation for the importance of data modeling which, if done poorly, limits the ability to use the data for analysis downstream. Indeed, getting data modeling wrong in data-intensive systems is painful since it is easy to generate hundreds of gigabytes of data in the "wrong" format (in that the selected structure does not allow answers to important questions to be computed efficiently) and then face the unenviable task of reformatting that data into new forms and/or migrating the data into a new persistence technology.

With respect to analysis, Project EPIC had several goals to achieve: 1) the partial automation of the manual analysis process that our analysts already use to study smaller data

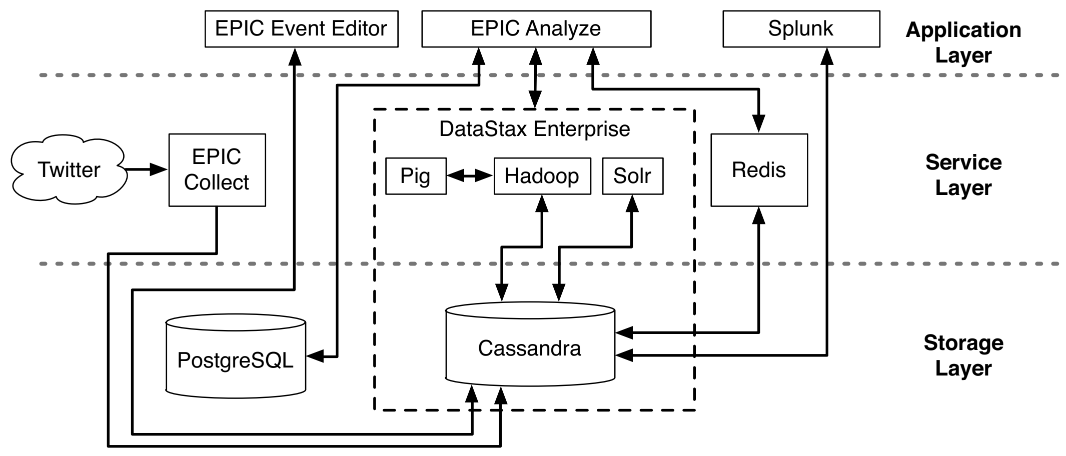

Fig. 1. The Software Architecture of EPIC Collect and EPIC Analyze [7]

sets, 2) the efficient generation of metrics and statistics that our analysts need for each data set, and 3) the ability to support exploratory analysis of large data sets in an efficient manner [8]. With respect to the latter, our analysts want to be able to ask questions of the data and get back answers in seconds (not minutes) independent of the size of the data set (from tens of thousands of tweet to tens of millions) so as not to interrupt their analytical "flow" as they work with the data.

Creating an analysis environment that met these goals involved a significant engineering effort but also a research exploration of the trade-offs involved with selecting one class of technology over another and then determining the software architectural style that would allow these disparate technologies to be integrated into a single operational system [7].

The software architecture encompassing both systems is shown in Fig. 1. EPIC Collect consists of the EPIC Event Editor that allows analysts to indicate the events and keywords they are interested in and a middleware component that connects to the Twitter Streaming API and stores the tweets that flow in from that service into a large in-memory queue. Multiple workers process that queue in parallel, computing various statistics, logging each tweet, and then handing the tweets to a Cassandra cluster for long-term storage.[1]

Our Cassandra cluster then acts as the foundation of EPIC Analyze, our analysis environment. We make use of Pig and Hadoop to calculate metrics that require processing every tweet in a data set; Solr to enable advanced search capabilities over those tweets; Redis to cache the results of certain queries and to aid in the pagination of our data sets; and Postgres to store shared annotations (comments and tags) that our analysts make on tweets as they analyze our data sets. All of these technologies are orchestrated by the EPIC Analyze web application (implemented in Ruby on Rails) to provide our analysts with a wide range of advanced browsing, filtering, searching, and analytical capabilities. This architecture has multiple extension points; this has allowed us to integrate third-party analysis tools (e.g. Splunk) into the system allowing our analysts to pick and choose among a variety of analytical options.

[1]All of the technologies mentioned in this paper (e.g. Cassandra, Redis, etc.) have websites, documentation, tutorials, etc. that can be found via Internet search; we have thus abbreviated explanations here in the interest of space.

Both of these systems have been described in detail in the papers cited above. For this paper, we will simply make note of the extreme heterogeneity of this software infrastructure. We make use of six generic software frameworks (Postgres, Pig, Hadoop, Solr, Cassandra, and Redis) each with their own concepts and APIs, three major custom-built components, a third-party data analysis tool, and a variety of "glue code" not shown in the diagram. Furthermore, the architecture shown in Fig. 1 is a logical representation that does not reveal the intricate nature of the deployment of this architecture onto a cluster of machines—four for Cassandra, one each for Redis and Postgres, four for Hadoop and Solr, one for EPIC Collect, and one for EPIC Analyze—all located within a professionally-maintained data center at the University of Colorado. Both the architecture and the operational aspects of these systems place significant demands on the team of SE researchers that have worked on Project EPIC since Fall 2009.

We now turn to the lessons learned about the design challenges of data-intensive software systems as we worked to design, develop, and deploy EPIC Collect and EPIC Analyze.

III. Design Challenges of Data-Intensive Software Systems

In this section, we identify and discuss challenges that are associated with the design and development of data-intensive software systems. Where appropriate, we identify choices that can be made to address these challenges and what trade-offs need to be considered when making those choices.

A. *Lack of Developer Support*

The first challenge that is encountered by software engineers moving into the design space of data-intensive systems is the paucity of tools designed to help developers work with the highly heterogeneous software architectures that these systems require. In contrast, consider the tools that a developer has when developing a single desktop or mobile application. Modern development environments such as XCode or Eclipse, provide advanced text editors, code completion, automated refactoring, advanced application frameworks, static analysis, UI development tools, relational database development tools, integrated debuggers, and the like.

In contrast, the developer of a data-intensive system loses nearly all of that support and has to cobble together general tools (editors, command-line build systems, standalone compilers, etc.) to produce an environment that is somewhat productive. There are no advanced modeling tools for Hadoop or integrated Hadoop debuggers; there are no tools that automatically configure Redis or MongoDB or Cassandra with the data models that a system requires and certainly no support for migration when those data models evolve over time. Furthermore, when it comes time to deploy the system on a cluster (of possibly hundreds) of machines, there are complicated system administration tasks to learn alongside obscure provisioning, virtualization, and configuration management frameworks to make use of to transition a system from design and development to production operation. Developers in this space are essentially back to square one, similar to desktop application developers back in the 1980s. Project EPIC's experience validates this situation: building data-intensive systems is currently a laborious, error-prone process. We are focusing our efforts in this space on a general framework that significantly reduces the cost of building systems that work with multiple data stores.

While this is an unfortunate and challenging situation, it represents a major research opportunity for software engineering. What modeling frameworks do developers of data-intensive systems need? What debugging tools would be most helpful? How do software engineers model and select the deployment options their system needs and can the actual deployment be automated or significantly reduced in complexity? The distributed systems community has developed a wide range of tools and frameworks that can be used to design and build these systems but with very little support for the software engineer. Now, the SE research community must work to make these tools and frameworks accessible to a wider range of software developers with appropriate support for modeling, design, and debugging.

B. *Need for Multidisciplinary Teams*

A second challenge in working on the design of data-intensive software systems is the necessity of a multidisciplinary team that provides expertise on a diverse set of skills and topics. Data intensive systems are typically "full-stack" applications that require well-designed *user interfaces* with the ability to display, browse, sort, filter, query, and analyze information, *middleware components* that respond to requests from the UI as well as perform analytics, collect data, migrate data, clean data, etc. to *persistence technologies* that store data often on clusters of machines. The management of those clusters may require an entire array of software components to handle *provisioning, virtualization, and configuration*. In addition, these systems have to handle an iterative *data life cycle* that includes a) developing questions to be answered, b) curating the potential data sources, c) collecting data from these sources, d) cleaning the collected data, e) storing it, f) processing/analyzing the data, and then g) displaying and visualizing the data in response to queries. The results of those queries provide new information that may lead to new questions that trigger a new iteration of the data life cycle. Even if the results do not trigger new questions, these systems must still support the ongoing collection of data from the identified data sources leading inevitably to questions of long-term storage and the development of data management and data retention policies that then must be implemented.

To properly handle the design and development of these systems and the demands of the data life cycle, a team developing data intensive systems requires, in general, people skilled in software engineering, distributed systems, system administration, data analysis (statistics and machine learning, graph theory, etc.), information retrieval, natural language processing, data persistence, information visualization, and user interface design and development and, specific to an individual project, application domain experts who can identify the types of data that can be collected or generated and the important questions that can be asked of the domain. It can be quite challenging to assemble such a team; Project EPIC was able to do so given its origin as a large NSF project supported by a multidisciplinary team with experience performing interdisciplinary research. In addition, Project EPIC was situated at a research university where it was possible to consult with experts outside the team as needed to cover any gaps in the team's knowledge and skills.

C. *Iterative Life Cycles and a Commitment to the Domain*

A third challenge in working with big data is the need for highly-iterative life cycles with a development team that is committed to understanding the application domain of the system and the needs and culture of its end users. This is, of course, true of all software systems (if one wants them to be used [9]) but these concerns are especially stressed in these situations. Understanding of the domain is essential because one needs to make sure the system will answer the right questions. Getting the questions wrong will lead to the collection of the wrong data (and lots of it) which will eventually have to be migrated, used to answer different questions, or deleted—all tasks that will take time and will prevent the team from moving the project's focus to the right questions and the right data. Understanding the needs of the system users is also essential in determining the right set of technologies to incorporate into the system design. Will the users be asking the same set of questions of the data each time or will their questions evolve over time? Do they need interactive access to the data or will batch-generated reports be sufficient? Do the reports present information on the entire data set or on windows of time within the data set? Are textual reports of statistics and metrics enough or is visualization required? Is 24/7 data collection required? Are the data sources steady or bursty? Are there multiple data sources or just one? Do metrics need to be calculated as data streams in, or can they be generated after the data is stored?

The answers to these questions all impact the technology decisions that must be made and how the data that is collected/generated is stored and processed. Technologies for batch processing data make fundamentally different assumptions than technologies for processing streaming data. These technologies can be combined into a single system (using, for instance, the Lambda Architecture [10]) but at the cost of significant complexity for the software engineers and operational complexity for system administrators. The only way to mitigate that cost is for the developers and users to work closely together, to be willing to iteratively prototype solutions, to test those prototypes with users, and to be willing to undertake the pain of having to migrate large amounts of data to new solutions if initial technology choices prove to be inadequate with respect to changing requirements.

We have documented our own experience working in this way on a crisis informatics system designed to reunite pets with their families after disasters [11]. In that work, we describe the range of user-centered design techniques we used in the development of this data-intensive system and the modified agile life cycle we employed to ensure the system would meet the needs of its user community. Agile life cycles and user-centered design techniques provide a basis for meeting this challenge but additional SE research is needed to further explore this space: how do teams deal with the consequences of making a wrong design choice; what tools can be provided to minimize the impact of storing large data sets in the "wrong" format; what new roles are needed on the development team to ensure that steady progress is made when designing and developing data-intensive systems?

D. Matching Frameworks with Requirements

A fourth challenge in designing data-intensive systems is the need to properly match the characteristics of the distributed systems frameworks available for use in developing big data systems with the requirements of the system-under-development. We did not select Cassandra as our primary persistence framework because it was "popular." We selected it because it perfectly matched the needs of EPIC Collect [6]. We were in a situation where our previous choice of using a relational database to store Twitter data had reached its limits.

Firstly, Twitter at the time (early 2011) was rapidly evolving the types of metadata that would ship with each tweet. Each time the metadata changed, we had to make a decision as to whether we wanted to store one of the new attributes. If so, we had to alter the schema of our database, migrate our existing set of data to the new schema, and update our software to properly store the new attributes when saving tweets. This type of change rapidly grew into a maintenance nightmare that threatened our ability to collect data 24/7—a requirement imposed by our collaborators whose research depended on getting "complete" data sets on the mass emergency events they selected for study.

Secondly, the size of our data sets were moving into the territory where relational databases demand "vertical scaling" techniques (essentially buying an expensive server with more memory and disk space) or shifting to using sharding techniques that require the developers to rewrite their software to store data in multiple relational databases partitioned by some sort of application-defined key. This aspect was again a potential maintenance nightmare: now we would have to manage multiple relational databases with very little tool support and we had the potential of getting our partitioning strategy wrong and ending up with unbalanced data sets (and thus unbalanced load) across the multiple machines.

Finally, with relational databases it is difficult to ensure that data is reliably saved; one can regularly run database "dumps" and then ensure that the extracted data is backed-up on a separate machine but this adds to the operational complexity of the environment and we were (at the time) a team of two SE researchers trying to perform our own research while providing a production software system that collects data 24/7.

Cassandra helped us solve each of these problems. The fact that Cassandra does not impose a schema allowed us to simply store the JSON objects delivered by Twitter unmodified. Changes in the metadata no longer had an impact during collection; it might have an impact during analysis (e.g. when a computation relies on a particular attribute being present) but at the time we were focused solely on getting to a reliable situation with respect to data collection and had decided that we would deal with analysis-related problems at a later stage. We could have adopted a similar strategy with our relational database but once one starts storing BLOBs in a relational database one should examine why a relational approach is being used in the first place.

Cassandra replaces the need for vertical scaling and sharding with the more palatable option of horizontal scaling. Indeed, Cassandra's design is inspired by Google's BigTable [12] and Amazon's Dynamo [13]; it was designed for horizontal scaling from the start. With horizontal scaling, if a system is running out of disk space, simply add another server to the cluster. If a system needs more processing power, add another server. When Cassandra runs on a cluster, it automatically partitions the data across the cluster, performs reads and writes in parallel, and is able to scale to extremely large sets of data. The software engineer no longer has to worry about handling sharding-related issues at the application layer. Instead, data can be persisted to Cassandra and those details are handled automatically. Issues related to vertical scaling also go away since horizontal scaling provides significant disk space and compute power by clustering cheaper, commodity hardware.

Finally, Cassandra nearly eliminates the need for software engineers to worry about data loss. That is because it can be configured with a replication factor and it will then ensure that each row of data sent to it is stored multiple times on different servers in the cluster. Replication occurs when the row is being written for the first time so if the write completes, developers know that the data has also been replicated. Replication may also occur when the size of the cluster changes: the addition of a new node may allow distinct copies to live on distinct nodes while the removal of a node may require creating new copies of the rows stored on that server to be created on yet another server to ensure that the replication factor is maintained. If the primary node for a row goes down, Cassandra will automatically return the row from one of its replicas. All of this is handled automatically after the replication factor is set for the first time removing the need for developers to handle replication at the application level.

We therefore selected Cassandra as our persistence solution since it gave us the *scalability*, *flexibility*, and *reliability* that solved (indeed eliminated) very real problems that we were struggling with at the time. We reviewed a wide range of NoSQL solutions before selecting Cassandra. MongoDB is a capable, indeed popular, document database but back in 2011 it was not reliable when operating in a cluster configuration and our data sets were too big to be stored on a single machine. HBase did not offer the performance that we needed with respect to our reads and writes. Solr/Lucene is a document-based system that focus on providing advanced search capabilities and should not be used as a general purpose, long-term storage solution. These are just examples of the systems we reviewed alongside Cassandra and each one failed to meet our needs *with respect to data collection*.

The key lesson here is that members of a development team must invest the time to deeply understand the wide range of software frameworks that are available to include in the design of a data-intensive software system whether they are considering a candidate persistence technology, a distributed computation framework, a search technology, a caching technology, etc. The technology that is selected for inclusion should closely match the requirements of the larger system and solve real problems that the system would encounter without its use. This is a generally-applicable statement to system design but it is especially true in the big data design space. Generic frameworks that are misapplied will not scale or will not function at a level of reliability needed to prevent problems during production operation.

This challenge represents another research opportunity for the SE community. The characteristics of these systems should be documented from the perspective of a software architect: when should each framework be used; what frameworks work well together and what sort of scalability, flexibility, and reliability guarantees can be achieved; can abstraction layers be created on top of these frameworks to make them more accessible to a wider range of software engineers?

E. Easy Becomes Hard at Scale

One surprising challenge in the design and development of data-intensive software systems is the way in which functionality that is straightforward to implement for desktop and mobile applications—such as sorting, maintaining data model consistency, or displaying data—transform into significant engineering challenges, often requiring the use of unfamiliar algorithms and/or approaches. For instance, at Project EPIC, we regularly deal with files of Twitter data exported from Cassandra that are tens of gigabytes in size. Students working with large data for the first time will often complain that they are unable to open these files to see what is in them. When questioned, it becomes clear that the students were trying to load the entire file into memory. Such difficulties become learning experiences about how to work with files of this side, handling them in chunks, or importing the data in some form into another database for processing. This type of transformation is needed for other operations as well.

When displaying large data sets, one can choose to provide a "big picture view" via descriptive stats or graphs but one can never display the entire data set at once. Firstly, the data set is too big to bring into main memory, and secondly, the time it would take to transfer, e.g., millions of tweets across a network connection for display in a web application is too long for human operators. Instead, developers must "paginate" the data set and display just fifty tweets (or some similar number of tweets) per "page" of a data set instead. Adding the mechanisms to handle pagination over a data set is an unfamiliar operation for many software engineers. In desktop and mobile applications, many developers are used to being able to load all of their data into memory at once and then display it in various ways across the views of their application. EPIC Analyze instead keeps an index of tweet references for a data set in Redis and uses array manipulations to quickly calculate the tweet references that appear in a particular page. It then sends those tweet references to Cassandra to retrieve the tweets which are then displayed in the browser. This display process occurs in under a second (even for data sets containing millions of tweets) but is way more complex than what would be required for a desktop application. Yet this type of architectural complexity is required to achieve sub-second response on large data sets.

Likewise the task of sorting data transforms from a straightforward operation to a significant engineering challenge. With EPIC Analyze, we have data sets consisting of millions of tweets. These tweets are by default sorted by tweet id. When analysts view Twitter data sorted in this way, it becomes natural to want to view the data sorted by other attributes, e.g. user screen names or favorite counts. Unfortunately, it is not possible to sort this data on an alternative attribute at interactive speeds. One would need to read all of the tweets into memory, sort them on the new attribute, and write them back to Cassandra in the newly sorted order and one would need to do that for each sort dimension. Out of curiosity, we tested this approach for sorting on a small data set of $\sim$180K tweets. It took seven minutes to read all of these tweets into main memory for sorting and this is one of Project EPIC's smaller data sets. This approach simply would not work on data sets consisting of millions of tweets.

Instead, we developed a new incremental sorting method [14] that sorts our data sets as a batch process along all desired sort dimensions; it is incremental in that it can handle the case where a data set is under active collection and is able to sort new tweets that are added to the data set each day without having to re-sort previously sorted tweets. Our method takes about five minutes to process the 180K tweet data set and once it is done it can display a page of sorted tweets along any dimension in under one second. While we eventually reached the state that allowed our analysts to sort our data sets along any dimension they desired, it took a significant amount of design and engineering to offer a feature that is otherwise standard across "small data" applications.

Finally, in many small-scale applications, it is important to maintain data model consistency. For instance, if one were to model EPIC Collect's data model in a relational database, one would create tables for events, keywords, tweets, and users. Events are associated with a start date and an end date and a set of keywords. Tweets are collected that contain those keywords; they are stored in the tweets table and are associated with a particular event. Each tweet was generated by a particular user and the user's information is stored in the users table. An event has many tweets; a user has many tweets; each tweet belongs to a single user and at least one event. With a relational model, third normal form is a desired goal and so one would strive to maintain a single entry per unique user in the user table; each time a tweet is collected from a previously-seen user, the collection software must read the metadata associated with the user and see if it has changed. If it has, it must update the information in the user table overwriting the information that was stored there previously (object-relational mappers typically do not support keeping track of previous values of the fields in a database row). If an event is deleted, then most relational systems would delete all of its associated tweets and keywords. If a user no longer has any tweets, it too might be deleted. Likewise deleting a user would delete its tweets and might cascade to delete events and keywords.

All of this is done in an attempt to enforce data consistency such that any data in the database is up-to-date and that out-of-date or inconsistent data is never shown to an end user. *All of these concerns are obsolete in data-intensive software systems that make use of NoSQL technologies.* Rather than attempt to maintain only a single copy of a user, each tweet is stored with its embedded user object; this means that within our data store, we have one copy of the user object per tweet that user generated. There is no attempt to keep track of the "latest version of a user;" instead one re-designs questions to care only about "the state of the user at the time this tweet was generated." Furthermore, there is no attempt to avoid wasted disk space due to duplication because the underlying premise of NoSQL is that "disk space is cheap" and that if one needs more space one will add additional servers to the cluster.

This approach to data storage is a complete reversal of how many software engineers were trained to think and it can be difficult to set aside time-honored approaches such as third normal form and relational technology and embrace the new approaches that are needed to scale to truly large data sets. The key lesson associated with this challenge is that developers must be willing to abandon a wide range of techniques that will no longer work when dealing with large data sets and must be open to iteratively combining general purpose frameworks in various ways until a "straightforward" operation—such as sorting—becomes feasible at scale.

F. Data Modeling

As mentioned in Section I, data modeling is paramount to the success of data-intensive software systems. A system must be collecting the types of data needed to answer the questions of its users and storing that data in such a way that those questions can be answered in an efficient manner. Of course, it is impossible to anticipate every question that a user might have for a given application domain. In traditional information systems, this problem is addressed by general-purpose query systems. These systems can answer arbitrary questions over structured data in an efficient manner; performance penalties are encountered only when questions cannot take advantage of indexes created for questions that were known from the start.

In big data and data-intensive systems, asking questions on data not structured to answer that question in an efficient manner often causes major problems: either the existing data has to be reformatted (at great cost) or a computation is devised to produce the answer but often with a considerable performance penalty. Often, it is easiest to just update the collection software to restructure new data in the format that will allow both existing and new questions to be answered in an efficient manner and simply acknowledge that the new questions cannot be applied to the old data. One can observe this happening with new features of large social media sites like Twitter and Facebook. New features often only apply to data that was generated after the feature was introduced and is not available on previously created data.

A major difficulty with data modeling in big data systems is the way in which seemingly minor choices with respect to the way data is stored and referenced in NoSQL systems can have major impacts on overall system performance. As documented in [6] and [7], we had to apply significant effort to the design of the row keys we used to store tweets in Cassandra. A row key is used to retrieve rows from a column family (similar to tables in relational database technology except that no schema is enforced); each row has a set of columns; and each column consists of a key and a value. As there is no schema, each row can have a different number of columns and column names can be completely different from row to row.

With respect to performance, row keys play a critical role. Firstly, servers in a Cassandra cluster will split (i.e. partition) the space of possible keys among themselves. All keys that fall within a partition are stored on the same server. Secondly, row keys determine how many columns can be associated with a given row; this is important because Cassandra replicates rows. If a row is "narrow" then it will consume less disk space and be easier to replicate. If a row is "wide" it can be quite large and impose performance penalties when replicated, especially when a node is added or removed from the cluster and the key space must be re-partitioned. Thirdly, row keys determine what can be queried. If one stores row keys that make use of application domain information such as dates and times, an application can make time-based queries by formulating the appropriate row keys correctly. However, if row keys do not contain application domain information and instead are simply unique integers, then one loses the ability to query particular rows individually and must instead fall back to iterating over all the rows in a column family (a much slower operation).

When storing tweets, one might consider using a tweet's unique id as the row key and then store each of its metadata key-value pairs as columns in the resulting row. However, this would create rows that are too narrow (only one tweet per row) and would not make best use of Cassandra's ability to handle rows with lots of columns; plus it would be impossible to query for a given tweet individually unless one maintained an index of all previously stored tweet ids. Furthermore, since tweet ids are monotonically increasing, one faces the possibility of all tweets streaming in for a given time period being mapped to a single node in a cluster, thus overworking it and under-utilizing the rest of the cluster.

Thus, one must carefully design row keys such that a) they generates rows that are not too wide to facilitate replication, b) contain application domain information in them so they are easily queried, and c) easily partitioned so that their associated rows are evenly distributed across a cluster. With EPIC Collect, we use row keys that consist of three parts: a keyword, a julian date, and a hexadecimal digit. An example is "flood:2015032:a" which represents one sixteenth of all tweets that contain the keyword "flood" collected on Feb. 1, 2015.

What this scheme does is that it ensures that our rows maintain a reasonable width, are easily queried, and partition the load of reading/writing rows across all nodes in a cluster. With this design, collecting 1M tweets on a given day for a given keyword generates 16 different row keys each containing only 62.5K tweets. If one needs to review "hurricane" tweets collected on January 1st of 2014, then one asks Cassandra for row keys: "hurricane:2014001:0" to "hurricane:2014001:f". The hexadecimal digit ensures that rows for a given keyword and julian date are evenly distributed across the cluster since there is more of a chance that each of the sixteen possible row keys fall into different partitions on the cluster.

Unfortunately, none of these issues were clear when we first started using Cassandra to store our Twitter data. It required a lot of work to understand why we were experiencing performance problems, linking those problems back to our row keys, and then developing a proper design for our row key to ensure good performance. The lesson of this particular challenge then is to not underestimate the difficulty in getting the data models of data intensive systems to properly take advantage of the persistence technologies being used and therefore be in a position to offer the best performance and reliability that each technology can provide. There are future SE research opportunities here to devise ways in which row keys might be automatically generated for a particular class of persistence technology given some description of the data model of a big data software system, as well as finding ways to reduce the complexities of working with more than one data storage technology at a time.

IV. Conclusions

In this paper, we identify six challenges related to the design of data-intensive software systems. We provide insight into these challenges by sharing our experiences working to design and develop two data-intensive systems for Project EPIC. We hope that the identified issues demonstrate the complexities that are involved in working in this design space and that our approach to these challenges can aid other software engineers and SE researchers solve problems when working on their own data-intensive software systems. We also identified a variety of research opportunities for the SE community to pursue to make it easier to design data-intensive systems in the future.

Acknowledgment

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

References

[1] L. Palen, J. Martin, K. M. Anderson, and D. Sicker, "Widescale computer-mediated communication in crisis response: Roles, trust & accuracy in the social distribution of information," 2009, http://www.nsf.gov/awardsearch/showAward.do?AwardNumber=0910586.

[2] L. Palen, K. M. Anderson, G. Mark, J. Martin, D. Sicker, M. Palmer, and D. Grunwald, "A vision for technology-mediated support for public participation & assistance in mass emergencies & disasters," in *ACM-BCS Visions of Computer Science*, April 2010, Article 8. 12 pages.

[3] M. Cox and D. Ellsworth, "Application-controlled demand paging for out-of-core visualization," in *Proceedings of the 8th Conference on Visualization*. IEEE Computer Society Press, 1997, pp. 235–ff. [Online]. Available: http://dl.acm.org/citation.cfm?id=266989.267068

[4] K. M. Anderson and A. Schram, "Design and implementation of a data analytics infrastructure in support of crisis informatics research (nier track)," in *33rd Int. Conf. on Software Engineering*, 2011, pp. 844–847.

[5] P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2012.

[6] A. Schram and K. M. Anderson, "MySQL to NoSQL: Data modeling challenges in supporting scalability," in *ACM Conf. on Systems, Programming, Languages and Applications: Software for Humanity*, October 2012, pp. 191–202.

[7] K. M. Anderson, A. A. Aydin, M. Barrenechea, A. Cardenas, M. Hakeem, and S. Jambi, "Design challenges/solutions for environments supporting the analysis of social media data in crisis informatics research," in *48th Hawaii International Conference on System Sciences*. IEEE, January 2015, pp. 163–172.

[8] K. M. Anderson, A. Schram, A. Alzabarah, and L. Palen, "Architectural implications of social media analytics in support of crisis informatics research," *IEEE Bulletin of the Technical Committee on Data Engineering*, vol. 36, no. 3, pp. 13–20, September 2013.

[9] W. Orlikowski, "Learning from notes: Organizational issues in groupware implementation," in *1992 ACM Conference on Computer-Supported Cooperative Work*, 1992, pp. 362–369.

[10] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications, 2015.

[11] M. Barrenechea, K. M. Anderson, L. Palen, and J. White, "Engineering crowdwork for disaster events: The human-centered development of a lost-and-found tasking environment," in *48th Hawaii International Conference on System Sciences*. IEEE, January 2015, pp. 182–191.

[12] 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," in *7th Symposium on Operating System Design and Implementation*, 2006, pp. 205–218.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazons highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[14] A. A. Aydin and K. M. Anderson, "Incremental sorting for large dynamic data sets," in *1st IEEE International Conference On Big Data Computing Service And Applications*, March 2015, in press.