Cloud Documents Storage Correctness

Aspen Olmsted
Department of Computer Science, College of Charleston
Charleston, SC 29401
e-mail: olmsteda@cofc.edu

Abstract— **In this paper, we investigate the problem of providing correctness guarantees when representing transaction data in semi-structured documents in cloud-based systems. We compare traditional relational database correctness guarantees including normalization and domain constraints with our correctness guarantees for document-oriented databases. In this research, we specifically focus on transactional data that would have traditionally been stored in a relational database system. We ensure that our new guarantees improve the data quality while not reducing the availability of the systems.**

Keywords-web services; distributed database; modeling

I. INTRODUCTION

In this work, we investigate the problem of representing transactional data in a platform as a service (PAAS) cloud-based document storage system. Document-oriented storage systems are excellent in providing availability to client applications. Unfortunately, they sacrifice consistency and durability to achieve this availability. The CAP theory [1] [2] states that distributed database designers can achieve at most two of the properties: consistency (C), availability (A), and partition tolerance (P).

In traditional relational databases, database normalization is used to ensure that redundant data is not stored in the system. Redundant data can lead to update anomalies if the developer is not careful to update every instance of a fact when modifying data. Normalization is also performed to ensure unrelated facts are not stored in the same tuples resulting in deletion anomalies.

Relational databases also provide data correctness guarantees through the use of constraints. Constraints can take the form of domain constraints where the value of an attribute is limited using either the specific attributes data type, check constraints or referential integrity. Out of the box, document-oriented databases allow each document to have its own structure. The designer can write validation code, but that code cannot check other records stored in the system.

There are two major document-oriented database systems in production today. They are named CouchDB [3] and MongoDB [4]. Both systems store schema-less semi-structured data with the goal of providing high availability and redundancy. International Business Machines (IBM) offers a cloud service based on CouchDB named Coudant [5].

Our goals in this research are to allow the developer the high availability provided by these cloud-based document-oriented data storage systems and also have a higher level of correctness guarantees. In this work, we provide normalization algorithms and domain checks for both data types and referential guarantees.

The organization of the paper is as follows. Section II describes the related work and the limitations of current methods. In Section III, we give a motivating example where our normalization and correctness algorithms will be helpful. Section IV describes standards used for semi-structured data validation. Section V explores breaking our model into a directed graph and how to partition that graph for normalization. Section VI contains information on how we can add semantics to the data model to help in our partitioning algorithm. Section VII describes how we generate the validation function to ensure the correctness of documents on creation and modification. We conclude and discuss future work in Section VIII.

II. RELATED WORK

Constraint specification and enforcement have been a part of relational database model research since Codd [6] originally wrote the specification. Recently work on the auto-generation of SQL code to enforce these constraints from the UML model has been done by Heidenreich, et al. [7] and Demuth, et al. [8]. In both these works, the focus is on the generation of the SQL code for relational databases for the invariants. Document-oriented databases require additional work to ensure the constraints can be guaranteed while not decreasing the availability of the service.

Research in the distributed database community has been conducted for decades on finding a balance between availability and consistency. Recent research can be grouped into one of three goals: 1.) to increase the availability with strict replication, 2.) to increase consistency with lazy replication, and 3.) to use a hybrid approach to increase the availability. Document-oriented databases were developed to allow the implementer to have a high availability while sacrificing immediate consistency. We can group the consistency and availability research into four groups.

1) Increasing Availability of Strict Replication: Several methods have been developed to ensure mutual consistency in replicated databases. The aim of these methods is eventually to provide one-copy serializability (1SR). Transactions on traditional replicated databases are based on reading any copy and writing (updating) all copies of data items. Based on the time of the update propagation, two main approaches have been proposed. Approaches that update all replicas before the transaction can commit are called eager update propagation protocols; approaches that allow the propagation of the update after the transaction is committed are called lazy update propagation. While eager update propagation guarantees mutual consistency among the replicas, this approach is not scalable. Lazy update

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-607-1

propagation is efficient, but it may result in a violation of mutual consistency. During the last decade, several methods have been proposed to ensure mutual consistency in the presence of lazy update propagation (see *[9]* for an overview.) More recently, Snapshot Isolation (SI) *[10] [11]* has been proposed to provide concurrency control in replicated databases. The aim of this approach is to provide global one-copy serializability using SI at each replica. The advantage is that SI provides scalability and is supported by most database management systems.

2) Increasing Consistency in Lazy Replication: Breitbart and Korth *[12]* and Daudjee, et al. *[13]* propose frameworks for master-slave, lazy-replication updates that provide consistency guarantees. These approaches are based on requiring all writes to be performed on the master replica. Updates are propagated to the other sites after the updating transaction is committed. Their framework provides a distributed serializable schedule where the ordering of updates is not guaranteed.

The approach proposed by Daudjee et al. provides multi-version serializability where different versions of data can be returned for requests that read data during the period that replication has not completed.

3) Hybrid Approach: Jajodia and Mutchler *[14]* and Long, et al. *[15]* both define forms of hybrid replication that reduce the requirement that all replicas participate in eager update propagation. The proposed methods aim to increase availability in the presence of network isolation or hardware failures. Both approaches have limited scalability because they require a majority of replicas to participate in eager update propagation. Most recently, Irun-Briz et al. *[16]* proposed a hybrid replication protocol that can be configured to behave as eager or lazy update propagation protocol. The authors provide empirical data and show that their protocol provides scalability and reduces communication cost over other hybrid update protocols. In addition to academic research, several database management systems have been developed that support some form of replicated data management. For example, Lakshman and Malik *[17]* describe a hybrid system, called Cassandra, which was built by Facebook to handle their inbox search. Cassandra allows a configuration parameter that controls the number of nodes that must be updated synchronously. The Cassandra system can be configured, so nodes chosen for synchronous inclusion cross data center boundaries to increase durability and availability.

4) Buddy System: In our previous work *[18]-[20]*, we provide an architecture and algorithms that address three problems: the risk of losing committed transactional data in case of a site failure, contention caused by a high volume of concurrent transactions consuming limited items, and contention caused by a high volume of read requests. We called this system the Buddy System because it used pairs of clusters to update all transactions synchronously. The pairs of buddies can change for each request allowing increased availability by fully utilizing all server resources available. Consistency is increased over lazy-replication because all transactional elements are updated in the same cluster allowing for transaction time referential integrity and atomicity.

An intelligent dispatcher was placed, in front of all clusters, to support the above components. The dispatcher operated at the OSI Network level 7. The high OSI level allowed the dispatcher to use application specific data for transaction distribution and buddy selection. The dispatcher receives the requests from clients and distributes them to the WS clusters. Each WS cluster contains a load balancer, a single database, and replicated services. The load balancer receives the service requests from the dispatcher and distributes them among the service-replicas. Within a WS cluster, each service shares the same database. Database updates among the clusters are propagated using lazy-replication propagation.

After receiving a transaction, the dispatcher picks the two clusters to form the buddy pair. The dispatcher selects the pair of clusters based on versioning history. If a version is in progress and the request is modifying the data, then the dispatcher chooses set containing the same pair currently executing the other modify transactions. Otherwise, the set contains any pair with the last completed version. The primary buddy receives the transaction along with its buddy's IP address. The primary buddy becomes the coordinator of a simplified commit protocol between the two buddies. Both buddies perform the transaction and commit or abort together.

The dispatcher maintains metadata about the freshness of data items in the different clusters. The dispatcher increments a version counter for each data item after it has been modified. Any two service providers (clusters) with the latest version of the requested data items can be selected as a buddy. Note, that the database maintained by the two clusters must agree on the requested data item versions but may be different for the other data items.

Unfortunately, the buddy system required greenfield engineering to leverage the new algorithms. This current work allows a developer who has deployed a document-oriented database in the hope of high availability to regain some consistency.

III. MOTIVATING EXAMPLE

We demonstrate our work using a Ticketing Reservation System (TRS). The TRS uses web services to provide a variety of functionalities to the patrons who are attending a performance. To understand the impacts on a real organizations' data we used the New York Philharmonic Orchestra's data for the past 10 years. We simplified their relational data model to allow for a better illustration of the challenges in moving from the relational to a semi-structured data model. Figure 1 shows the Entity Relationship (ER) model we used for our experimentation. In the model, each

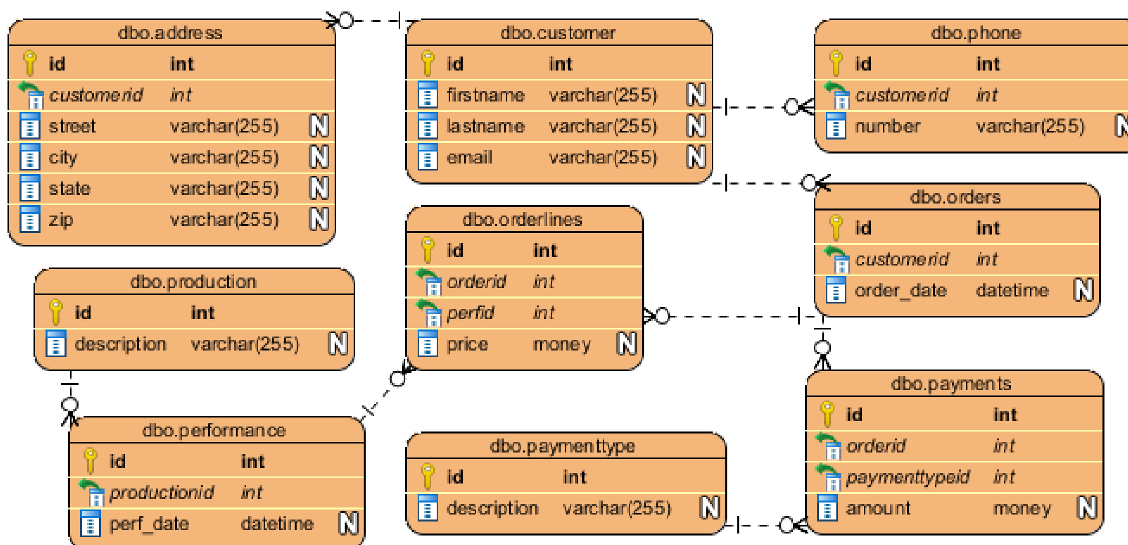

Figure 1. Relational Model

individual customer can have many addresses, many phone numbers and many orders tracked by the system. Each order can have many order lines entities and many payments entities. The order table stores a record of each order for tickets purchased by the individual customer. The order line entity has a many-to-one relationship to a performance record. The performance record represents a specific performance of the orchestra of a production. The production has the description of the pieces performed and the orchestra members. For example, on a weekend one production would have two performances. This relationship is represented by a many-to-one relationship from the performance table to the production table. In our research, we take the relational model in Figure 1 and convert the model into a CouchDB [3] data model.

IV. SEMI-STRUCTURED DATA, SCHEMAS & VALIDATION

There are two main formats used in semi-structured data stores; JavaScript Object Notation (JSON) and Extensible Markup Language (XML). JSON documents are in a format that is easily read by the JavaScript programming language. XML documents are an older format that allows any language to create well-formed documents by creating a language of tags to mark the data. The XML Schema format has matured to the level of being governed by a standards body where the JSON Schema [21] is relatively new and is not governed by a standards body as of yet. XML Schema is governed by the World Wide Web Consortium (W3C) [22].

Both Schema formats allow you to define custom entities, attributes, and the hierarchy of the entities stored in a single document. The two formats diverge in relation to references across documents. XML Schema allows one document to reference the existence of data in another document where JSON schema validation is only within a single document.

The two document-oriented databases we analyzed in this research use the JSON document format but do not support JSON schemas. Both systems support a document validation function that fires before a document is inserted or updated. In the case of MongoDB, there is a declarative structure that can limit the domain of the data type using enumerations and regular expressions. CouchDB and the Cloudant system allow unlimited validation functions that parse the records using JavaScript code. The functions can throw exceptions that stop the data operation from completing.

V. ROOTED TREES AND PARTITIONING

The data model shown in Figure 1 can be partitioned in many ways. We could store every object in a single document representing the complete hierarchy. The problem with this approach is normalization. We will have many copies of the same facts if the graph is not a complete directed

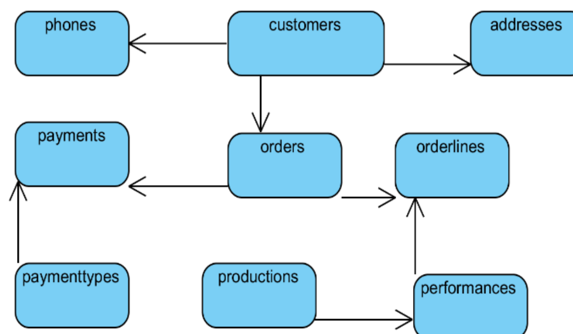

Figure 2. Directed Graph of Data Model

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-607-1

acyclic rooted graph [23]. This type of graph can be referred to as a rooted tree.

We can turn the ER diagram into a graph by using foreign keys as direct edges that travel from the one node to the many nodes. Each table in the ER diagram becomes a node in the graph. Once we have the graph, our algorithm can try each node, and if it can visit every other node and we do not have a cycle then our start node can be the root, and we have a rooted tree. If we have a cycle, then we had foreign keys that pointed in both directions between two entities. Cycles are only possible if we are starting from a relational database that allows constraint validation at transaction time instead of action time. Most relational databases do not allow transaction time constraint enforcement. In the case of cycles, we can merge the entities as they are truly one-to-one relationships. Figure 2 shows a directed graph generated from our data model in Figure 1.

In the case of Figure 1, we do not have a single rooted tree. We have three subtrees each with their own root; starting from customers, payment types, and productions. In this case, we would be required to store duplicate values across several documents depending on the root we choose. If we choose customers for the root node, then we will duplicate payment type, production, and performance information. This can lead to an update anomaly if we modify an attribute in one of those nodes but do not update all nodes that contain the duplicate information.

To eliminate the vulnerability of an update anomaly we need to partition the document into 3 sperate documents. Clearly, the three possible roots belong in their own document. We can continue traversal from these nodes to include other nodes in the separate documents. Unfortunately, we end up with two nodes (payments and order lines) that are placed into two separate documents. To decide which document these nodes should be stored in we will turn to the design documents and pull the required semantics from those documents.

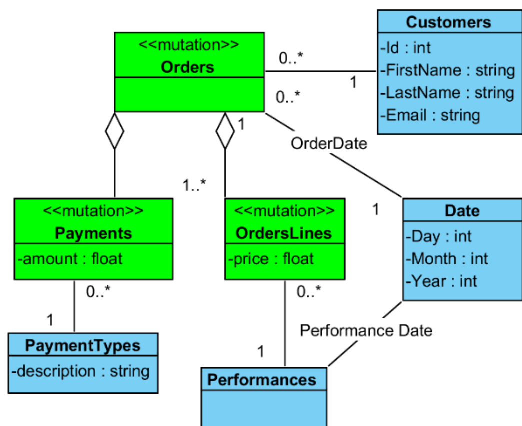

Figure 4. Class Diagram for “Write Order” Activity

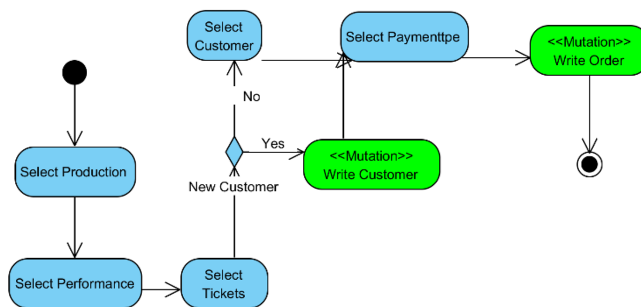

Figure 3. UML Activity Diagram for a Transaction

VI. UML SEMANTICS

Additional semantics for the data model can be acquired from the integration of the matching UML Activity and Class diagrams. UML provides an extensibility mechanism that allows a designer to add new semantics to a model. A stereotype allows a designer to extend the vocabulary of UML in order to represent new model elements [24]. We utilize this mechanism to understand the read and write semantics of activities that consume and generate the data in our data model. Figure 3 is an activity diagram with two stereotypes used to model activities that are read-only and activities that write and update data. The activity model is the main type of transaction that reads and writes the data in our data model. This transaction model is the process of purchasing a ticket for a specific performance. The “Write Order” activity modifies data as part of the transaction. This ability is represented by the stereotype of “Mutation” The

Algorithm 1. Partition Algorithm

INPUT: ER Diagram, Activity Diagrams (XMI representation of UML class diagram) and Class Diagrams (XMI representation of UML class diagram)
OUTPUT: document partition

```
1    docPartions = empty array
2    foreach activityDiagrm in activityDiagrams
3    foreach activity in activtiyDiagram
4    if activity is a mutation
5    foreach rootedSubtree
6    documentFound = FALSE
7    foreach entity in rootedSubtree orderby tree nav
8    if entity in activity and class is a mutation
9      if NOT documentFound
10       push new curDocument on docPartitions
11       documentFound = TRUE
12     else
13       add readonlyMidBranches to curDocument
14     add entity to curDocument
15   else
16     if documentFound
17       add entity to readonlyMidBranches
```

Copyright (c) IARIA, 2018.

"Write Customer" activity can also mutate data but is not required in every execution path.

```
(['red', 'green', 'blue'].indexOf(doc.field) >= 0)
```

Figure 5. Enumeration Validation

Each activity in the activity diagram has a matching UML class diagram that represents the internal structure of the code that manages the activity. Figure 4 displays the matching class diagram for the "WriteOrder" activity. We utilize the same stereotypes as we used in the activity diagram to model which elements are read-only and which can be modified in the activity. We use Algorithm 1 to choose the proper document partition based on the semantics of the UML model.

Algorithm 1 first creates an empty array to hold the document partitions. The algorithm takes a complete set of activity diagrams and matching class diagrams and navigates through the activity diagrams. The UML diagrams are passed into the algorithm in XMI format. XMI is a standard XML format for representing UML diagrams. Having the model in XML allows us to automate our algorithm, as any programming language can read the XML representation of the model. If the activity diagram is a mutation, then the algorithm will loop over the rooted subtrees from the ER diagram and will add the path from the beginning mutated document to the last mutated document. In our example application, we did not have overlap across the partitions. There is the possibility of overlap, and in that case, the documents will need to be merged to ensure each entity is only in one document. The overlap should be straightforward as it should represent different workflows to generate similar data. For example, in the motivating example for this research, we could have an activity diagram for a self-service web transaction to purchase a ticket and a phone order transaction with a back-office system operator. In both workflows, the partition documents would be almost exact.

In our example four partitions were created:

1. customers, addresses, phones
2. orders, orderlines, payments
3. payment types
4. performances, productions

The new structure eliminates both update and deletion anomalies. Deletion anomalies occur when a fact is lost because all related facts are deleted. With the rooted tree structure, the parent's facts need to exist by definition for the child facts to exists.

VII. Document Validation

Now that we have solved the normalization problem through the partitioning of our documents, we want to ensure that updates are validated for domain consistency. There are three types of domain consistency we are concerned with:

1. Simple Data Types – Simple data types including integers, floats, dates, times and strings.
2. Enumerations – Enumerations are limitations of the valid instances of a simple data type. For example, we could have an attribute of enumeration type color that takes in three possible strings: 'Blue,' 'Red,' 'Green.'
3. Referential Integrity – In relational databases, we used foreign keys to link column values to tuples stored in another table.

In CouchDB and the IBM cloud-hosted version Cloudant, design documents are just JSON documents stored in the database. This means we can add design documents via the HTTP interface programmatically. We developed an application that will iterate through our relational model and generate a design document per partitioned document to enforce our three domain consistency types. JSON Schema could be used for the first two domain consistency concerns but not for referential integrity. JSON Schema does not have the notion of referential integrity, and the validation function does not have access to other documents. So instead of trying to implement JSON schema validation in the JavaScript validation function, we took a novel approach that allows us to solve all three of the potential domain consistency issues.

To enforce the simple data types, we were able to use the built-in JavaScript parse methods such as parseInt, parseFloat.and Date.parse. To enforce the enumerations, we read the enumerations from the information schema of the database model and generate a validation test such as is shown in Figure 5. The left-hand side of the code includes a list of the possible enumerated values.

Referential integrity is handled in two ways depending on the partitioning of the document. If the foreign entity is stored in a separate document, then we handle the situation similarly to how we handled the enumerations. For each read all the possible values from the document store and generate validation check to ensure the new value is one of the possible options. If the foreign entity is in the current document tree, we navigate the document to check for its existence.

The challenge with our foreign key solution is in timing. For example, in our motivating example when a new performance is created, no new "orderlines" entities can be written without a new version of the design document being generated that includes the new performance in the valid list. To solve this problem, we implemented a client application that was written in Java. The application executes on a local machine in the end-user organization location. The application utilizes the continuous changes API in CouchDB and Cloudant to receive change notifications on the lookup tables. The continuous changes API allows the application to see the changes as they come in using a single HTTP connection between the application and the database service. When the application sees a change in the lookup data, it will generate a new revision of the design document to include the changed values in the validation function for the foreign key checks. This allows our validation function to have a low latency between the time new facts are inserted into the

document store and the time they can be used in related documents.

Our solution works well except in the case of surrogate identifiers in parent entities. Surrogate ids are used when there is not a natural identifier. Entities that use surrogate identifiers tend to have a large number of entities in the collection, and our solution does not work when the foreign key list is large. In our motivating example, the customer's entity has a surrogate identifier for the id attribute. When we partitioned the document so that the orders entity is stored in a different document from the customer entity the validation function for the orders needs to have a list of valid customers. In practice, the number of customers would be too large to handle this way. To solve this problem, we merge the two documents, so we end up with a single document that covers the complete rooted tree consisting of customers, addresses, and payments. This solution does not break the normalization and simplifies the validation so that the validation can happen in a single document.

VIII. CONCLUSION

In this paper, we propose algorithms for semi-structured document normalization and domain value correctness and validation. We develop a test implementation to automate our implementation and validate that your solutions provide the guarantees for the normalization of the semi-structured data and for the consistency of domain values. Our solutions are based on navigating the relationships in both ER and UML diagrams and using additional semantics applied to the models.

In this research, we studied a specific application domain related to the entertainment industry. We believe the algorithms can be applied to other application domains without a significant amount of modification. Future work needs to test our algorithms in other application domains to ensure the work applies across different application domains. We also plan to add additional guarantees of correctness for these semi-structured documents.

REFERENCES

[1] S. Gilbert and N. Lynch, "Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News,* vol. 33, pp. 51-59, 2002.

[2] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer,* vol. 45, pp. 37-42, 2012.

[3] The Apache Software Foundation, "couchDB relax," 2017. [Online]. Available: http://couchdb.apache.org/. [Accessed 27 August 2017].

[4] MongoDB, Inc., "MongoDB for Giant Ideas," 2017. [Online]. Available: https://www.mongodb.com/. [Accessed 31 August 2017].

[5] International Business Machines, "IBM Cloudant," 2017. [Online]. Available: https://www.ibm.com/analytics/us/en/technology/cloud-data-services/cloudant. [Accessed 31 August 2017].

[6] E. F. Codd, The Relational Model for Database Management, Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[7] F. Heidenreich, C. Wende, and B. Demuth, "A Framework for Generating Query Language Code," *Electronic Communications of the EASST,* 2007.

[8] B. Demuth, H. Hußmann and S. Loecher, "OCL as a Specification Language for Business Rules in Database Applications," in *The Unified Modeling Language. Modeling Languages, Concepts, and Tools.*, Springer, 2001, pp. 104-117.

[9] M. T. Ozsu and P. Valduriez, Principles of Distributed Database Systems, 3rd ed., Springer, 2011.

[10] H. Jung, H. Han, A. Fekete and U. Rhm, "Serializable snapshot isolation," *PVLDB,* pp. 783-794, 2011.

[11] Y. Lin, B. Kemme, M. Patino Martıınez and R. Jimenez-Peris, "Middleware based data replication providing snapshot isolation," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data, ser. SIGMOD ’05*, New York, NY, 2005.

[12] Y. Breitbart and H. F. Korth, "Replication and consistency: being lazy helps sometimes," *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of database systems, ser. PODS ’97,* pp. 173-184, 1997.

[13] K. Daudjee and K. Salem, "Lazy database replication with ordering," in *Data Engineering, International Conference on*, Boston, MA, 2004.

[14] S. Jajodia and D. Mutchler, "A hybrid replica control algorithm combining static and dynamic voting," *IEEE Transactions on Knowledge and Data Engineering,* vol. 1, pp. 459-469, 1989.

[15] D. Long, J. Carroll and K. Stewart, "Estimating the reliability of regeneration-based replica control protocols," *IEEE Transactions on,* vol. 38, pp. 1691-1702, 1989.

[16] L. Irun-Briz, F. Castro-Company, A. Garcia-Nevia, A. Calero-Monteagudo and F. D. Munoz-Escoi, "Lazy recovery in a hybrid database replication protocol," in *In Proc. of XII Jornadas de Concur-rencia y Sistemas Distribuidos*, 2005.

[17] A. Lakshman and P. Malik, "Cassandra: a decentralized structured," *SIGOPS Oper. Syst. Rev.,* vol. 44, pp. 35-40, 2010.

[18] A. Olmsted and C. Farkas, "High Volume Web Service Resource Consumption," in *Internet Technology and Secured Transactions, 2012. ICITST 2012*, London, UK, 2012.

[19] A. Olmsted and C. Farkas, "The cost of increased transactional correctness and durability in distributed databases," in *13th International Conference on Information Reuse and*, Los Vegas, NV, 2012.

[20] A. Olmsted and C. Farkas, "Coarse-Grained Web Service Availability, Consistency and Durability," in *IEEE International Conference on Web Services*, San Jose, CA, 2013.

[21] Jjson-schema Organisation, "JSON Schema," 2017. [Online]. Available: http://json-schema.org/. [Accessed 27 August 2017].

[22] World Wide Web Consortium, "Schema," 2017. [Online]. Available: https://www.w3.org/standards/xml/schema. [Accessed 27 August 2017].

[23] Wikipedia, "Rooted graph," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Rooted_graph#CITEREFGrossYellenZhang2013. [Accessed 30 August 2017].

[24] O. M. Group, "Unified Modeling Language: Supersturcture," 05 02 2007. [Online]. Available: http://www.omg.org/spec/UML/2.1.1/. [Accessed 08 01 2013].