Future Generation Computer Systems 80 (2018) 384–399

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Testing of transactional services in NoSQL key-value databases

María Teresa González-Aparicio [a], Muhammad Younas [b,*], Javier Tuya [a], Rubén Casado [c]

[a] Computing Department, University of Oviedo, Gijón, Spain
[b] Oxford Brookes University, Oxford, United Kingdom
[c] Accenture Digital Spain, Madrid, Spain

HIGHLIGHTS

- Design and development of the new framework with context-aware transaction model that takes into account contextual requirements of NoSQL clients and system level setting in relation to the data consistency.
- Design and modelling of a real life big data about London bus services in such a way that can be represented and stored in NoSQL key/value databases.
- Analyse the impact of the big data requirements and characteristics such as availability and velocity on the consistency of NoSQL databases. Note that current literature does not investigate into this issue.
- Test and evaluate the proposed framework using a widely used NoSQL key/value database, Riak, and a real (open/big) data from the Council of London for public transportation of bus services.

ARTICLE INFO

Article history:
Received 14 July 2016
Received in revised form 26 April 2017
Accepted 1 July 2017
Available online 17 July 2017

Keywords:
Transactions
Testing
Context-aware
CRUD
NoSQL key/value database
Riak
Data consistency

ABSTRACT

Transactional services guarantee the consistency of shared data during the concurrent execution of multiple applications. They have been used in various domains ranging from classical databases through to service-oriented computing systems to NoSQL databases and cloud. Though transactional services aim to ensure data consistency, NoSQL databases prioritize efficiency/availability over data consistency. In order to address these issues various transaction models and protocols have been proposed in the literature. However, testing of transactions in NoSQL database has not been addressed. In this paper, we investigate into the testing of transactional services in NoSQL databases in order to test and analyse the data consistency by taking into account the characteristics of NoSQL databases such as efficiency, velocity, etc. Accordingly, we develop a framework for testing transactional services in NoSQL databases. The novelty and contributions are that we develop a context-aware transactional model that takes into account contextual requirements of NoSQL clients and the system level setting in relation to the data consistency. This can assist NoSQL application developers in choosing between transactional and non-transactional services based on their requirements of the level of data consistency. The framework also provides ways to analyse the impact of the big data requirements and characteristics (e.g., velocity, efficiency) on the data consistency of NoSQL databases. The evaluation and testing are carried out using a widely used NoSQL key/value database, Riak, and a real (open) and big data from the Council of London for public transportation of the London bus services.

1. Introduction

Software services have become key sources for enhancing organizational and business processes as well as economic growth in the 21 century. Building on the service-oriented computing technologies, cloud computing has emerged as a new digital infrastructure for hosting and provisioning of modern services. Cloud services are provided using various models such as Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS).

With the popularity of cloud computing there has been a significant increase in the number of cloud users and services and in the volume of data they generate such as online social media, web searching/browsing, customers reviews, road traffic and weather data [1]. Such a large-scale service-generated data has led to a new concept called big data [2] (in terabytes and petabytes). The volume refers to the sheer amount of data to be stored and processed by

* Corresponding author.
E-mail addresses: maytega@uniovi.es (M.T. González-Aparicio), m.younas@brookes.ac.uk (M. Younas), tuya@uniovi.es (J. Tuya), ruben.casado.tejedor@accenture.com (R. Casado).

http://dx.doi.org/10.1016/j.future.2017.07.004
0167-739X/

the storage systems. It represents one of the biggest challenges for the new era of information technology structures due to its order of magnitude (in terabytes and petabytes). These new structures should be able to create horizontal scalability and to distribute data along the different nodes dynamically. Variety refers to the different formats of data that data storages have to deal with (texts, images, raw data, etc.). Velocity is the speed at which data are created, stored, analysed and visualized. Veracity is the trustworthiness of the value obtained from the data associated to a specific application.

The above characteristics and requirements take big data beyond the processing capabilities of traditional relational databases. Therefore, a new generation of databases, named NoSQL ("Not only SQL" or "No SQL"), has been emerged as the core technology for storing and processing big data. NoSQL databases are generally schema-free and use CRUD (Create, Read, Update and Delete) operations for processing data. They replicate data and do not support transactions such as ACID (Atomicity, Consistency, Isolation, Durability) properties. Data replication provides high availability and efficiency but it may result in data inconsistency in the case of updating (replicated) data.

In order to ensure stronger consistency of data and reliability of applications of NoSQL databases various transactional services (based on different models and protocols) have been developed [3,4]. However, they neither investigate into the testing of transactional services nor the consistency of data in NoSQL databases.

We believe that testing of transactional services and data consistency are important in order to systematically explore their behaviour and efficiency and to detect anomalies in data that may occur as a result of concurrent transactions [5]. But the process of testing transactions in NoSQL databases is complicated due to several reasons. First, transactions are more complex compared to classical transactions as they have to deal with replicated data which is distributed across different nodes of NoSQL databases. This situation is further complicated by the prospect of (high frequency) updates to the data which is the consequence of the velocity of big data. Second, transactions should be executed/managed in a way such that it maintains data consistency but without severely affecting the availability and efficiency of data. Third, transactions should be adapted to the needs of users/applications—e.g., when do users prefer strong consistency over availability/efficiency or vice a versa. Fourth, there is no standard model or framework that captures the requirements (characteristics) of big data as well as the properties of transactions.

In this paper we propose a new framework for testing transactional services in NoSQL key-value databases. Due to the complexity of the environment the framework involves design and development of various components or building blocks which include: development of a context-aware transactional system; design of a test data using NoSQL database; and design of test cases for testing the framework. The framework is implemented as a middleware layer over the NoSQL key-value database, Riak (by Basho). The testing of transactional services is carried out using a real (open/big) London bus services data from the Council of London transportation system [6].

The key contributions of this paper are as follow:

- Design and development of the new framework with context-aware transaction model that takes into account contextual requirements of NoSQL clients and system level setting in relation to the data consistency.
- Design and modelling of a real life big data about London bus services in such a way that can be represented and stored in NoSQL key/value databases.
- Analyse the impact of the big data requirements and characteristics such as availability and velocity on the consistency of NoSQL databases. Note that current literature does not investigate into this issue.
- Assist application developers in choosing between transactional and non-transactional NoSQL operations based on their preference and requirements of availability and consistency.
- Test and evaluate the proposed framework using a widely used NoSQL key/value database, Riak, and a real (open/big) data from the Council of London for public transportation of bus services.

The remainder of the paper is structured as follows. Section 2 presents an overview of NoSQL databases and related work on transactional services. Section 3 describes the proposed framework. Section 4 describes the type of contexts which are tackled in our architecture. Section 5 presents the application scenario and the datasets for testing the proposed framework. Section 6 presents the modelling of test data and the transaction management. Section 7 presents the evaluation and testing of the proposed framework. Conclusion is described in Section 8.

2. Related work

This section first illustrates the different types of NoSQL databases. It then explains key-value NoSQL database, Riak, which is used in the development and evaluation of the proposed framework. It also reviews existing work on transactional services in NoSQL databases.

2.1. NoSQL databases

Relational databases enforce relationship between data tables (relations) and support ACID properties that guarantee strong consistency of data and concurrency of transactions. SQL has been widely used in relational databases. But with the emergence of new technologies such as web services, service-oriented computing and cloud there has been significance increase in the amount of data which is generated through applications or services such as online social media, web searching/browsing, customers reviews, road traffic and weather data. In such applications it is difficult to maintain strong consistency (as in relational databases) as well as availability and scalability. According to the CAP theorem [6], consistency, availability and partition tolerance cannot be guaranteed simultaneously. This has led to a new trend in databases, named as NoSQL databases.

NoSQL databases process large volume of data and generate results in real time such as analysis of millions of tweets or processing of live road traffic data. Therefore, such applications demand high response time, scalability and availability. Different NoSQL databases follow different data models and provide different levels of consistency, availability and efficiency. The most common NoSQL databases types are Document databases, Key-value databases, Column store and Graph databases [7].

A key-value store, referred to as schemaless [8], consists of a set of key-value pairs. The values could be text strings or more complex lists and sets. Data searches are performed against keys, not values. This fact implies that only supports simple CRUD (Create, Read, Update, Delete) operations. Some examples are Dynamo (Amazon), Voldemort (LinkedIn), Redis, BerkeleyDB and Riak. A document database is a key-value store. Unlike the key-value stores, the value is a document expressed in standard data exchange format, such as XML, BSON or JSON. Keys and values are searchable. MongoDB and CouchDB are two examples of document oriented. In column stores, the columns of a table are partitioned

into groups, which are usually accessed together. These groups are named as column families. On disk, a table is stored by column families. Some examples are Bigtable (Google), Hypertable, Cassandra (Facebook), SimpleDB (Amazon) and DynamoDB. In graph databases, conceptual objects are represented as nodes in a network, edges between nodes represent object relationships, and properties express object attributes. Neo4j, InfoGrid, AllegroGraph, InfiniteGraph are examples of this technology.

The proposed framework is focused on key/value databases. More specifically it is built around Riak for two main reasons. Firstly, Riak is an open source key/value database. Secondly, it is based on Dynamo, which has been used for large systems such as Amazon. Riak works on a cluster which is made of multiple physical nodes. Each node is logically divided into virtual nodes. The set of key/value pair is assigned over different virtual nodes by a hash function. Data scalability and availability are achieved through a partition and replica technique. Each pair is replicated at N virtual nodes, which are located in distinct physical nodes. Moreover, key/value pairs are grouped in a namespace called "bucket". This is to allow storing different pairs with the same key but in different buckets. Buckets are grouped in another namespace named "bucket type", where a set of system behaviour properties could be established. For instance, properties like the number of replicas (N) and the level of consistency/availability could be initialized at the bucket type—i.e., to determine when a read ('r') or a write ('w') operation will be considered successful or not. For instance, if r and w have values lower than N, then the system will never reject the operation as long as there are at least r and w nodes available. Several instances of Dynamo are set with (3, 2, 2) values as a common configuration for (N, r, w). These are believed to provide satisfactory levels of performance, consistency and availability [9]. In general, consistency is maintained by a quorum technique and a decentralized replica synchronization protocol. The system will provide stronger consistency for $(r + w > N)$ over $(r + w \leq N)$.

2.2. Transaction services for NoSQL databases

In order to achieve the objectives of high availability, scalability and efficiency, NoSQL databases follow different design principle than that of classical relational databases. For instance, they adopt key/value data models, where relationships between data entities are not strictly enforced as in relational databases. Operations of the NoSQL query language have been simplified to Get/Put operations. ACID transactions are not guaranteed as NoSQL databases prioritize availability over consistency. Examples of such NoSQL databases are BigTable [10], Facebook Cassandra [11], Windows Azure [12] and PNUTS [13].

Realizing the importance of transactional services, several approaches have been proposed to implement transactions in NoSQL databases. Implementation of transactional services has been proposed at three different layers such as data store, middleware and client side. Systems such as Spanner [14], COPS [15], Granola [16] and Warp [17] have been developed to support transactions at the data store level. But this may compromise scalability and availability [18]. Middleware approaches include Google Megastore [19], G-Store [20], Deuteronomy [21], CloudTPS [22], pH1 [23], CumuloNimbo [24] and [25]. Such approaches implement transactional services at the middleware level which is an interface between clients and database. That is, concurrency control and ACID properties are managed at the middleware level. In the client layer approach, APIs are developed that send and receive metadata from the client's applications. Percolator [26], ReTSO [27] and the system in [4] are some examples.

The above models provide different levels of consistency in NoSQL databases. These include: strong consistency or linearizability (global real-time ordering) [28,29], sequential consistency (global ordering) [30], causal + consistency (combination of causal consistency and convergent conflict handling) [15], causal consistency (partial orderings between dependent operations) [31], FIFO consistency (partial ordering of an execution thread) [32], per-key sequential consistency (global ordering of operations for each key) [13], and eventual consistency (convergence to an agreement, which does not order concurrent operations) [33,34]. The authors in [3] propose a hierarchical transactional model where transaction operations are classified into four different data consistency levels. Considering the hierarchy from the strongest to the weakest data consistency level, the transaction management protocols applied to each level are as follows: Snapshot Isolation based on serializable transactions where ACID properties are ensured (SR level), Causal Snapshot Isolation (CSI) referred to as the fork-join model [35], Causal Snapshot Isolation with concurrent commutative updates (CSI-CM) and asynchronous updates (ASYNC).

Similar to pH1 [23], ReTSO [27], Percolator [26], and CumuloNimbo [24], our proposed framework aims to exploit timestamp and snapshot isolation techniques in order to ensure serialization and concurrency of transactions in Riak. Spanner [14] also adopts similar techniques in order to manage transactions that are distributed across different data centres. Other techniques such as Deuteronomy [21] use locks in order to manage concurrent transactions. Though locking technique is useful in ensuring consistency of data they have negative effects on the efficiency of transactions.

In summary, existing transactional systems implement different techniques to control concurrency, consistency and availability. Nevertheless, they lack of context-awareness in relation to client's requirements. Further, they do not investigate into the testing of transactions and data consistency.

2.3. Context-aware transactions

Applications and services should be aware of the context where they are performed. The aim is to provide more feasible and reliable outcomes. Dey [36] defined context as "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves". In addition, the term context-aware computing was first discussed by Schilit and Theimer [37]. The definition of context-aware given by Dey [36] is "A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task". The idea is that any software application, which has been designed with a context-aware architecture, should take into account the context as a key element during its execution [38].

There is a scarcity of literature on context-aware transactions systems based on NoSQL databases. For this reason, it is provided a general prospective of this type of systems but could be considered for systems based on NoSQL databases. Context-aware transaction systems have been applied to different fields. In mobile applications, it should be considered changes in the network states from on-line to off-line and vice versa if transactions are taking place [39–41]. There are Radio Frequency IDentification (RFID) systems [42] which should provide a context-aware intelligence in order to guarantee a secure transaction verification. In mobile commerce transactions, a context could be characterized by different points of view such as social, system, application and physical information [43]. Physical context involves aspects like spatial features (e.g. location). System and application contexts are related to the state of a set of entities, a computing entity (e.g. network and device) and an application entity (e.g. transaction) respectively. Social context considers data such as relationships with a group of people and/or personality traits, among others.

3. The proposed framework: an overview

This section provides an overview of the proposed framework for testing transactional services in NoSQL key-value databases. Due to the complexity of the testing process the framework involves design and development of various components or building blocks which include: development of a context-aware transactional system, design of a test data using NoSQL database, and design of test cases for testing the framework.

We design a context-aware transaction system for NoSQL key-value databases, in general, and Riak, in particular. It takes into account context information in the processing of transactions. A context could be defined as "a feature F is contextual for an action A if F constrains A, and may affect the outcome of A, but is not a constituent of A" [44]. Unlike existing transactional services, the context-aware transaction system captures the main requirements of NoSQL database transactions. This fact will imply that a context will impose a set of limitations for a given situation.

The transaction model is implemented as a separate layer over the NoSQL key-value database, Riak (by Basho). Riak is chosen as it is widely used and is based on the DynamoDB. Amazon, one of the biggest e-commerce services providers, uses DynamoDB [9] for its online services such as shopping carts, session management, product catalogue and customers preferences.

The testing of the proposed transactional services is carried out using a real big data from the Council of London transportation system and its iBus; an Automatic Vehicle Location (AVL) System. This dataset is chosen as it complies with the characteristics of big data such as volume, velocity and variety. However, the original data from the London (UK) transportation system cannot be straightforwardly stored in Riak. We therefore tailor the data according to the design principles of the Riak.

The following sections explain the detailed design and development of the different components of the proposed framework.

3.1. Context-aware transactional system

This section first presents the architecture of the context-aware transactional system. It then illustrates the execution protocol for the context-aware transactions.

3.2. Architecture design

The architecture of the proposed transactional system is depicted in Fig. 1. This based on extension to our previous work [45]. It has been designed following a combination of both a client layer and a middleware layer. This separates the transactional system from the underlying data store of the NoSQL database, Riak. Such separation of transactional system from the data store allows its portability to other NoSQL key/value databases.

The main components of the architecture are explained as follows.

Client layer: It receives every request from the client (user) that needs to be executed. The requests are forwarded to the "Client Manager". Initially, this module will create a thread for each request. It also takes care of the whole thread management process related to the processing of transactions.

Middleware layer: This layer comprises different components (of modules) of the system. The Coordinator manages the overall execution of CRUD (Create, Read, Update and Delete) operations which are grouped into transactions. That is, each transaction comprises different CRUD operations. Coordinator receives requests from the Client Manager. Every request is then sent to the Transaction Controller (TC). TC creates a new transaction and a sequence number for transaction ID. Sequence numbers are assigned to transactions

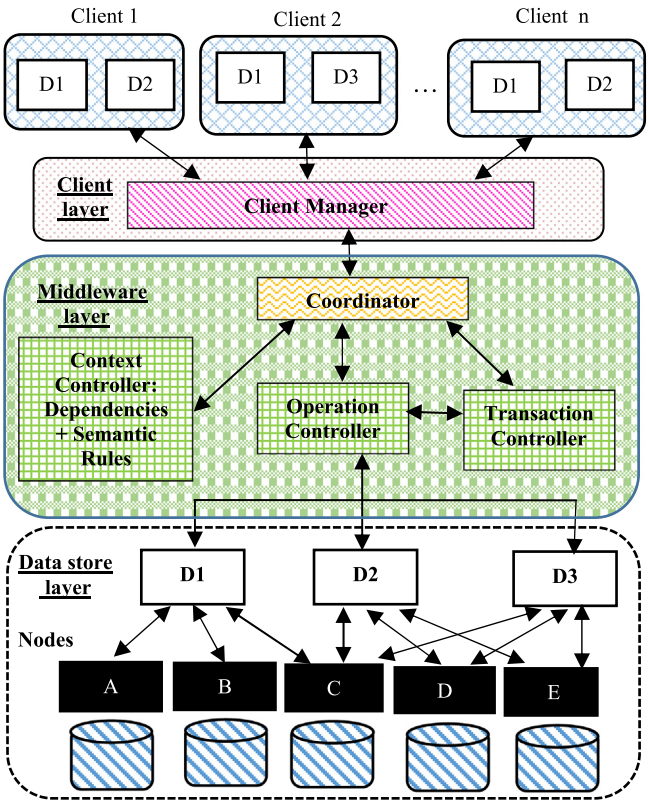

Fig. 1. Architecture of the context-aware transactional system.

in an monotonically increasing order [21,25]. Concurrency of multiple transactions, accessing shared data, can be handled using several techniques such as Two-Phased [46] technique or Snapshot isolation. Our transactional system uses the latter technique which deals with multi-version concurrency—this is more appropriate for managing multiple replicas of same data in NoSQL databases. Several write or read–write operations can simultaneously manipulate the same data and may result in write–write or read–write conflicts. Coordinator and TC therefore control concurrent data access in order to prevent data constraint violations such as Read Skew, Write Skew or lost update [47,48].

The Context Controller manages context information during the execution of CRUD operations of a transaction. The Coordinator and TC in cooperation with the Context Controller can decide on the outcome of a transaction if it meets the required context. The decision on meeting the context information is based on internal and external context. An internal (or implicit) context refers to a set of NoSQL database internal parameters (N, r, w) which can be configured according to the level of consistency and availability required by the client application. External (or client) context refers to functional and non-functional client requirements. A further explanation about what is understood by context information has been tackled more deeply below (Section 5).

The Operation Controller is in charge of the communication with the NoSQL key/value database during the execution of transactions.

3.3. Execution protocol

The different components of the (above) architecture communicate with each other in order to execute transactional and non-transactional CRUD operations [45]. We devise an execution protocol which comprises various steps. These are explained through a generalized scenario which is depicted in Fig. 2.

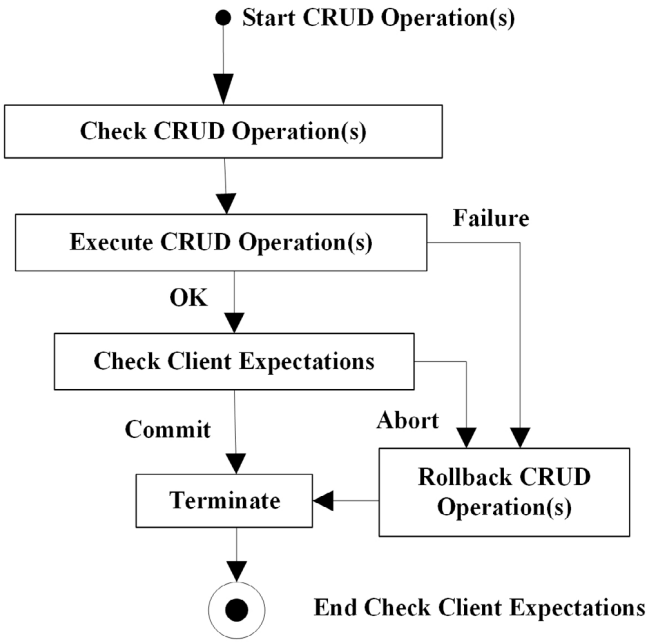

Fig. 2. Execution protocol—example scenario.

The execution protocol proceeds according to the following steps:

1. Clients submit CRUD operation(s) to the Client Manager which forwards them to the Coordinator. This event marks the start of CRUD operations.
2. Check CRUD Operation(s): The Coordinator checks the type of operations (transactional or non-transactional) and their contextual requirements.
3. Execute CRUD Operation(s): The Coordinator sends the information to the Operation Controller and Transaction Controller in order to execute the operation(s). The successful execution of operations depends on meeting the functional requirements (i.e., performing required tasks) as well as the client expectation (see Step 4). If the operations runs successfully and meets client expectation then they are considered to be 'successful' or committed. If not, they are rolled back or aborted.

Check Client Expectations: Operation Controller and Transaction Controller contact the Context Controller which checks client expectations. The process of checking client expectation is illustrated in Fig. 3. There are different steps involved in making decisions on the outcome of the operations which is based on context information (External Context (EC) and Internal Context (IC)), and the Client Expectations. This is one of the key components as it represents the logic-control to deal with multi-version data, context requirements and decisions made by users in relation to the commitment or abortion of transactional/non-transactional CRUD operations. Once IC and EC are evaluated, the decision on the outcome of the operations is then made. If IC and EC meet client's requirements, then the operation(s) are committed, otherwise they are aborted by the Transaction Controller. In addition, our system also provides users with the flexibility of making the final decision on the outcome of the execution of operations. That is, users may accept the outcomes even they do not exactly meet the IC and EC.

4. Rollback CRUD Operation: This operation implies to undo completed operation(s) and then terminate them. Undo is required when the Context Controller confirms that the outcome of the operation(s) has not achieved the contextual requirements after its execution. Undo is also performed if a system failure results in an unexpected outcome.
5. Terminate: This is to terminate the execution process of CRUD operation(s).

4. Context

The Context is a centric element which contributes to fulfil client expectations. This role is played by the "Context Controller" in our architecture. Context has been classified into two types, named as Internal Context (IC) and External Context (EC). The IC allows to establish some NoSQL database properties such as the level of consistency and availability (IC). Data relationships obey functional and non-functional application requirements. For this purpose, semantic rules and relations (dependencies) between specific application parameters in the NoSQL system should be guaranteed (EC). In the end, contexts have an influence on the behaviour of non-transactional/transactional CRUD operations.

4.1. Internal context

IC information is based on the configuration or setting of the NoSQL system properties (also called system settings). Specifically, in Riak there are more than twenty properties. Our work is centred on replication and multiversion data. Some NoSQL database properties which are related to those features are the following:

- An object could have different values on different nodes. These objects are called siblings. If this situation is allowed, then the property named as "allow_mult" should be set to true.
- Tracking the history of object updates could be done with timestamp, vector clocks or dotted version vectors. The aim is to sort out object conflicts. The mechanism timestamp is centred on chronological time, but vector clocks and dotted version vectors are geared towards sequences of events. The property named as "dvv_enabled" should be active for dotted version vectors. Moreover, the property "last_write_wins" indicates if an object's timestamp is used in case of write conflicts.
- Replication properties: "n_val" is the number of copies of each object in the cluster by default is three; "r" is the number of servers to respond to a read request by default is "quorum" (n_val/2 +1); "w" is the number of servers to respond to a write request by default is also "quorum".

An internal context is defined as an array of key-value pairs (p, v), where 'p' is the name of a property, and 'v' its value. For instance, if we would like to have three copies of an object out of 3, a number of read responses equal to 1, and siblings, then the system settings should be set as [("n_val", 3), ("r", 1), ("allow_mult", true)],

4.2. External context

There are some semantic aspects of a data model that cannot be only represented with parameters and values. For this purpose, the introduction of the EC into the framework expects to fill this gap. EC information includes both functional and non-functional requirements. The EC is represented by two types of logical expressions named as semantic rules and dependencies, which are explained below.

A semantic rule is defined as a logical expression that some specific function arguments have to obey. The combination of two logical expressions (two semantic rules) will form another

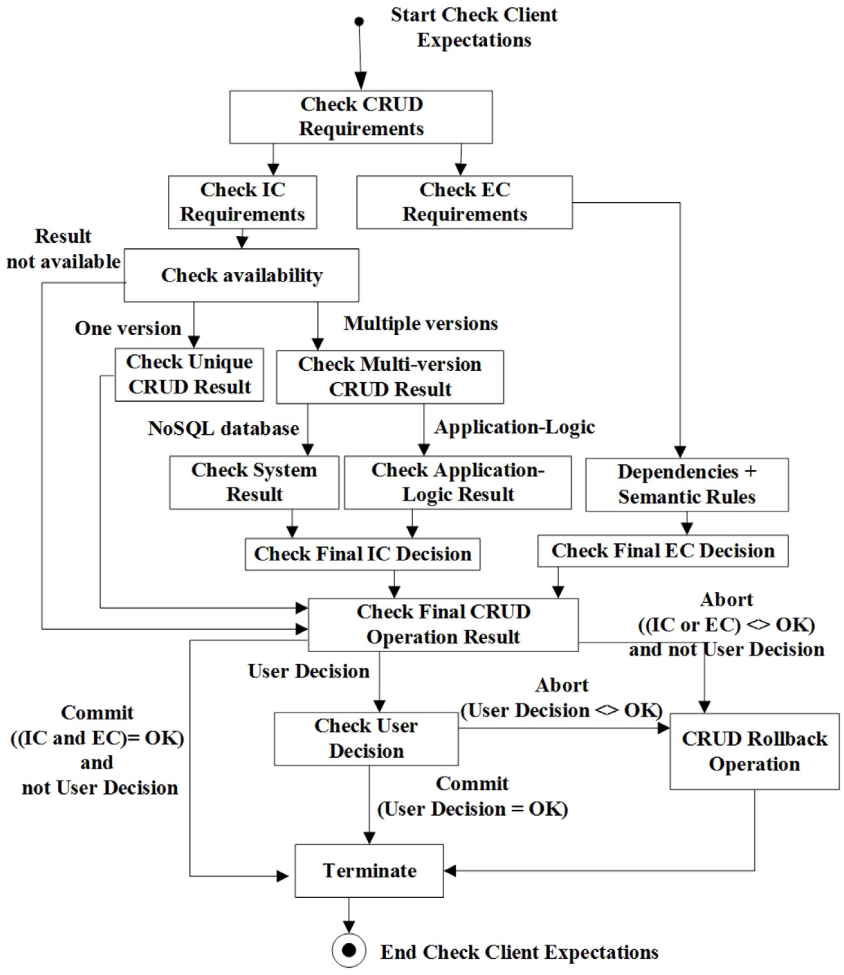

Fig. 3. Logic-control for transactional/non-transactional CRUD operations.

semantic rule. A semantic rule is defined with a list of three terms. The first and second term are parameters and the third one represents a logical expression between the first term and the second term.

A dependency arises when there are relations between two or more application parameters. It is defined with a list of $k+1$ terms (dependency, parameter1, parameter2, …, parameter$_k$), The first term represents the type of relation between the k parameters.

For instance "Transport for London" [49] could consider the following context:

- Semantic rule 1 (sr1): "the arrival time has to be a positive number", then the semantic rule is expressed like ("current_arrival_time", "0", ">").
- Semantic rule 2 (sr2): "the arrival time has to decrease", then ("current_arrival_time", "previous_arrival_time", ">").
- Semantic rule 3: Semantic rule 1 "and" Semantic rule 2, then (sr1, sr2, "and").
- Dependency 1: "a display device should visualize the arrival time according to the time left for the arrival", then it is expressed like ("==", "current_arrival_time", "expected_arrival_time").
- Dependency 2: "new data should be visualized in the display device every thirty seconds" (non-functional requirement), then it is like ("==", "frequency_arrival_time", "30").

4.3. API description

An API is designed for the communication between the application and the server. The communication procedure consists in setting internal and external contexts first, and after that performing transactional/non-transactional CRUD operations.

The definition of a CRUD operation has to use the following class:

- Class "CRUDOperation": it stores information in relation with a CRUD operation, such as: the type of CRUD operation ("C", "R", "U", "D"), a boolean value to identify if it is transactional ("true") or non-transactional ("false"), a boolean value to identify if there is a user decision ("true") or not ("false"), and an array with key-value pairs (p, v) for each parameter 'p' to be set with a value 'v'.

Then, a client request receives both contexts and a CRUD operation. For this purpose, the class "ClientRequest" defines a client request as follows:

- Class "ClientRequest": the attributes represent the information which can be sent and received to/from the server, i.e. the contexts and CRUD operations. Initially, an instance of this class should receive information in relation with internal context (an array of key-value pairs) and external context (two arrays of key-value pairs). After that, the methods allow to start, execute and end a CRUD operation. Moreover, it provides a method to consider the user decision as part of the operation.

Table 1
A Crud operation request.

```
//Defining the contexts
u = [(property_1, value_1), …, (property_q, value_q)] // Set 'q' system properties
v = [semantic_rule_1, …, semantic _rule_n)] // Set 'n' semantic rules
w = [dependency_1, …, dependency_m] // Set 'm' dependencies

//Creating a request and setting contexts
ClientRequest client = new ClientRequest (u, v, w);

//Defining a CRUD operation
t = type of CRUD operation // {"C", "R","U","D"}
is_transactional = transactional/non-transactional // {true, false}
user_decision = user decision // {true, false}
x = parameters [(param1, v1), …, (paramn, vn)] // Set CRUD operation parameters

CRUDOperation op = new CRUDOperation(t, is_transactional, user_decision, x)

//Executing a CRUD operation
client.startCRUDOperation(op)
client.executeCRUDOperation(op)
client.executeUserDecision(op)
client.endCRUDOperation(op)
```

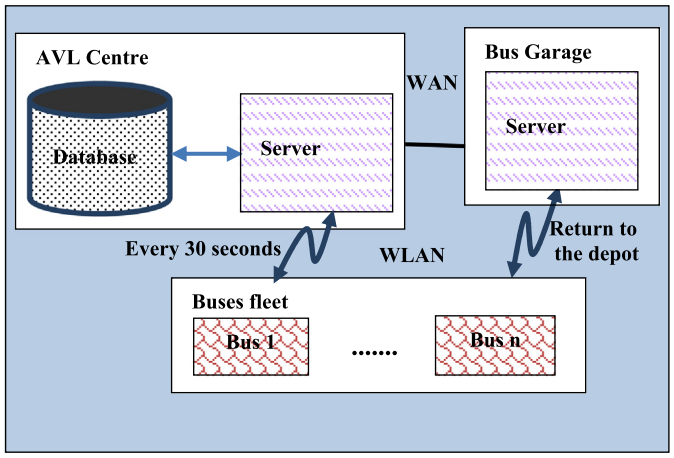

Fig. 4. AVL system.

In the end, Table 1 represents the procedure to execute transactional/non-transactional CRUD operations requests.

5. Application scenario and test data

This section describes the application scenario and test data which are used to evaluate the proposed framework. These are chosen such that they capture the requirements and characteristics of transactions and NoSQL big data.

- The dataset is based on London Datastore; this is an open and big data about London buses transport fleet, and is provided by the Greater London Authority [50].
- The application scenario is built around the iBus system; an Automatic Vehicle Location (AVL) System [51].

In the iBus system, a special unit OBU (on-board unit) is installed in every bus that stores information received from different sources (GPS, door sensors, etc.) in a real time. Such information is then sent from the OBU to a server in an AVL centre every 30 s approximately. When buses return to their depot, information from the OBU's memory is transferred to another server located at the garage, and then it is sent to the AVL centre. An overview of the iBus system components is represented in Fig. 4.

The data stored in a data stores in JSON format and is related to the location of the buses (latitude and longitude) on the road network and journey times such as estimated arrival times at destinations. After that, data are sent to and visualized along different electronic screens which are located on the buses and at the bus stops. The goal is to keep the passengers informed about the remaining time for the arrival of a specific bus, the current time stamp, etc. In addition, such data is also to the Internet so that people can see real time bus information on websites.

Using the iBus and AVL system, the (round-the-clock) operation of London buses generates large volume of data which accumulate to big data. This data is part of the London Datastore which has more than 500 different datasets that are openly available for use and distribution [52]. This data is in line with the 3V model of big data and is suitable for testing the proposed framework. First, such data has a high velocity given that it is sent to the (AVL) system and then distributed to different sources in a real time. This is also important for the analysis of the effects of data arrival (with high velocity) on the consistency of the database. Second, due to the nature of the information, different data structures have been used for its representation, thus fulfilling the 'variety' feature of big data. Third, the operation of the high number of bus lines and the enormous amount of bus stops [53] generate a large amount of data (i.e., 'volume' feature of big data). Specifically, there are over 19,000 bus stops, 700 bus routes and 8000 buses. Moreover, hundreds of thousands of bus arrival predictions are generated in time approximately [49]. Indeed, the system provides bus arrival information for a period of time of 30 min, including destinations, and data refresh every 30 s at source.

Datasets in the transport of London have their information centred on bus stops. There are five different types of datasets. If no real-time is needed, then it will return information named as reference data. There are two datasets classified into this group named as "Stop dataset" and "Baseversion dataset". The aim is to provide context to the real-time data whenever it is necessary. On the contrary, a combination of real-time and reference data will be provided. This last group includes three datasets, named as "Prediction dataset", "Flexible messages dataset" and "URA version dataset". Our test case is focused on the "Prediction dataset". This dataset provides the estimated arrival time for buses at bus stops, and reference data for routes and bus stops. Some fields are the following: arrival time to station (real-time), period of valid time for the arrival prediction (real-time) and reference data from the stop dataset.

In the end, it is used the "Prediction dataset" from the London Datastore, and it is stored in a NoSQL key-value databases such as Riak. From that dataset, it is possible to know data in relation with bus stops (reference data), bus arrival predictions and disruptions (real-time data). In the next section, we model and store data in Riak, and their process using transactional and non-transactional CRUD operations.

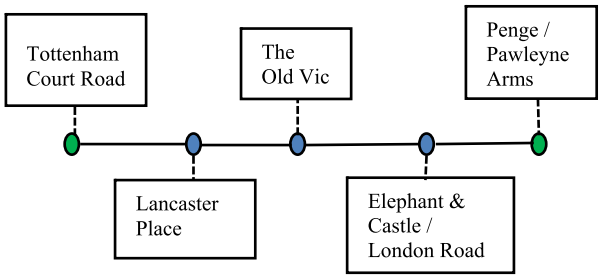

Fig. 5. A snippet from the bus line 176 in London city.


```
{
  "$type":"Tfl.Api.Presentation.Entities.Prediction, Tfl.Api.Presentation.Entities",
  "id":"-432165831",
  "operationType":1,
  "vehicleId":"LJ04YWV",
  "naptanId":"490000254D",
  "stationName":"Waterloo Station  / Waterloo Road",
  "lineId":"176",
  "lineName":"176",
  "platformName":"D",
  "direction":"inbound",
  "bearing":"143",
  "destinationNaptanId":"",
  "destinationName":"Penge",
  "timestamp":"2016-05-22T19:34:18.435Z",
  "timeToStation":846,
  "currentLocation":"",
  "towards":"Elephant & Castle or Kennington",
  "expectedArrival":"2016-05-22T19:48:25Z",
  "timeToLive":"2016-05-22T19:48:55Z",
  "modeName":"bus"
}
```


Fig. 6. One arrival prediction record for the line 176.

6. Data modelling and transaction management

We model a representative set of bus operation and data as it is not feasible to include all the bus operations and their data in this paper. The proposed framework has been tested on one of the bus lines with connections in the area of Waterloo (in London) [54], i.e., bus number 176 from 'Tottenham Court Road' to 'Penge/Pawleyne Arms'. It consists of 54 bus stops between the origin and the destination. Nevertheless, the example has been centred only on those bus stops which have other bus lines connections (bus number 68, 168, 171, 172 and 188). This is shown in Fig. 5.

Fig. 6 represents a real example in JSON format. It is one data record about the arrival information of bus line 176 that travels between two consecutive bus stations [55]. It includes the following fields: the name of the station from which the bus leaves "Waterloo Station/Waterloo Road" ("station name"), the name of the station towards which the bus is travelling "Elephant & Castle or Kennington" ("towards"), how much time (in seconds) is left to arrive at the station ("timeToStation"), prediction time for the arrival ("expectedArrival"), and maximum time that the prediction will be considered as valid ("timeToLive").

The goal is to keep the different buses informed about the route they should follow and the arrival time estimation at each bus stop [50]. Indeed, the arrival time information from the 21 buses from the Waterloo area could generate large volume of daily and monthly data. The data volume can increase to an order of magnitude of many terabytes if we consider data about more than 400 buses of the London bus transport fleet.

A database in Riak has been created in order to process the information. It has been populated with data from the "Prediction dataset" belong to the Transport of London. According to the information from the above example, it has been designed three entities in Riak, named as "Bus stop", "Line route" and "Bus route", which are represented in Fig. 7. The aim is to map the information we are interested in into the NoSQL database. In general terms, the "Bus stop" entity sets the geographical location for the bus stop (reference data), the "Bus route" entity identifies the sequence of bus stops which made the bus route (reference data), and the "Bus arrival prediction" entity provides the arrival time estimation to the next bus stop (real-time). The data model design for the three entities has been specified deeply in the next section (Section 6.1).

Approximately, the amount of information that the NoSQL database has to store is as follows: there are about 700 bus routes, which provide a service to 19,000 bus stops. They are covered with a fleet of 8000 buses approximately. The system estimates bus arrival predictions for each bus stop every 30 s for a period of 30 min in advance. Therefore, it is highly likely that the system generates bus arrival predictions beyond 300,000 per minute ($19{,}000 * 8000 * 2$).

6.1. Data modelling

The data model defines some units of information (or entities) that represent the data items obtained from the London Datastore. The entities named as "LineRoute" and "BusStop" respectively represent the route for a specific bus line and the bus connections with other lines at a specific bus stop. Indeed, a bus route has been established as a sequence of bus stops (locations) ("LineRoute"). Moreover, in modelling the data, it should be considered that a bus route may suffer variations due to possible disruptions on the road network and bus stop ("BusStop") at any time (e.g., traffic jams, delays, accidents, etc.). In the case of disruptions the central system (of the iBus) can set alternative routes for the bus lines that face disruptions.

Further, each bus route has been represented with an entity named as "BusRoute". It consists of a set of tuples with information about the different locations that the bus has already travelled through. Moreover, the expected journey time between two locations (bus stops) is represented in the database. This is due to the fact that it has been estimated from an algorithm based on the location of the bus. The different entity fields are represented in Table 2.

The locations ("id_origin", "id_destination", "id_location", "id_current_location" and "id_next_location") are identified by the NaPTAN (National Public Transport Access Nodes), which is applied by the Britain's national system to link points of access to public transport [56].

We use the NoSQL key/value database to store the information represented by the above entities. However, each NoSQL key/value database imposes certain restrictions on the way data is represented and stored. For instance, different NoSQL key/value databases represent data types differently. We use the NoSQL key/value database, Riak, that supports proprietary data types, called CRDTs ("map", "register", "set", etc.) [57].

Moreover, all entities should include properties that represent their internal context. This is defined using the "Bucket Type" structure in Riak. For instance, it is used to set the level of consistency for read ("level_r") and write operations ("level_w"), and the level of availability ("n_val"). A general schema of such entity is shown in Table 3.

However, the sequence of locations in "LineRoute" and "BusRoute" could not be stored in a "set" due the fact that the insertion order is not preserved. These entities have been modified as follows:

- The entity "LineRoute" has been split into two entities, named as "Line" and "LineRoute".

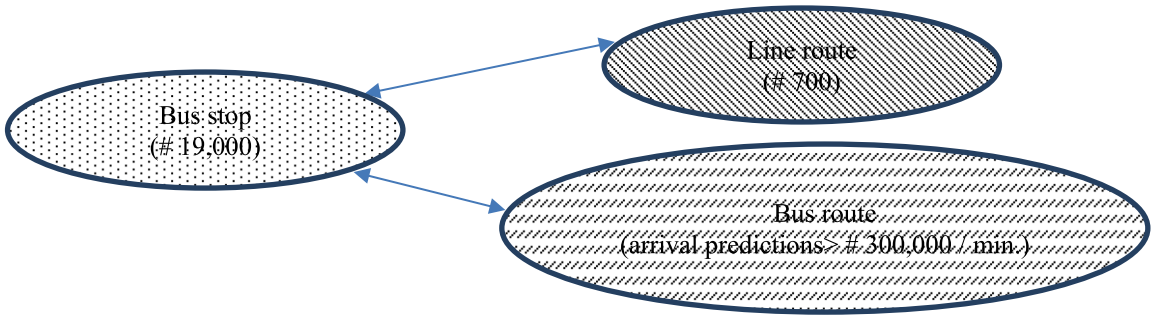

Fig. 7. Riak entities for transport in London.

Table 2
Entities description.

Entity	Fields
"LineRoute"	• Bus line identifier ("id_line"). • Name of the bus line ("line_name"). • Identification of the origin location ("id_origin"). • Identification of the destiny location ("id_destination"). • Direction of the journey "inbound" or "outbound" ("direction"). • Sequence of locations of the bus (route) from the origin to the destination following the specified direction:{id_location1, …}
"BusStop"	• Bus stop identifier ("id_location"). • Name of the bus stop ("location_name"). • Set of lines connected to the bus stop: {id_line1", "id_line2", …} • Presence of any disruption at the bus stop ("has_location_disruption").
"BusRoute"	• Bus identifier ("id_bus"). • Direction of the journey "inbound" or "outbound" ("direction"). • Set of tuples with information of the route for a specific bus at each bus stop: {("id_current_location", "id_next_location", "time_to_station", "expected_arrival_time", "time_to_live"), …}

Table 3
Entity properties.

Entity = bucket_type + bucket + level_r + level_w + n_val + {<other_properties>}

- The key of the entity "Line" is made of the concatenation of a line identifier and a type of direction ("id_line" +"direction"). This is due to the fact that every line could be followed in two directions ("inbound" or "outbound").
- A new key for the entity "LineRoute" is defined as the concatenation of "id_bus", "direction" and "id_current_location". Every instance of the entity "LineRoute" represents an association between a pair of locations in the route of a line. Therefore, the set of all instances belong to a specific line represents the whole route.
- Every instance of the entity "BusRoute" represents an association between a pair of locations in the current route for a bus line, the stamp time at the current physical location ("stamp_time"), the remaining time from the location to the next one in seconds ("time_to_station"), the expected time that the bus should arrive ("expected_arrival_time"), and the time until the approximation time is considered as valid ("time_to_live"). These are explained in Fig. 5.

Based on the above, a new data model is designed and is represented in Table 4.

6.2. Transaction management

Current iBus and AVL operations are not executed as transactions. We model such operations as CRUD operations which are executed as part of transactions using the Riak database. Recall that the benefit of using transactions is to ensure consistency of data and correctness (reliability) of concurrently running applications.

As shown in Fig. 4, iBus regularly receives data about location of a bus on the road network, journey time, estimated arrival times at destination, etc. This data is sent to a server in an AVL centre. From there the data is then sent to various sources such as display screens at bus stops, on board 'next-stop signage' screen within the buses, and the websites. This process involves a sequence of different operations. We simulate the execution of such operations as transactions using the proposed context-aware transactional system which is presented in Section 3.

We define a Global Journey Transaction (GJT) that consists of a set of subtransactions called Location Journey Transaction (LJT). That is, GJT $= \{LJT_1, LJT_2, \ldots LJT_n\}$. Each LJT contains a sequence of CRUD operations; LJT $= \{op_1, op_2, \ldots op_n\}$. For instance, op_1 can write data to the Riak database in order to update the current location of a bus on a road network. Similarly, op_2 can read the database in order to find out the arrival time of a bus at a bus stop.

We define the scope of a Location Journey Transaction to be the set of operations which are executed when a bus moves from one bus stop and arrives (stops) at another bus stop. For example, LJT_1 can cover the operations related to bus number 176 travelling from Tottenham Court Road to Lancaster Place, and LJT_2 covers operations from Lancaster Place to The Old Vics (see Fig. 5). LJT completes (commits) when a bus arrives at a bus stop and all its operations are executed and data is stored in the database. It may need to be cancelled (compensated) and aborted if its operations cannot be executed successfully.

When a bus starts its journey, the system creates a transaction, GJT and its subtransactions, LJT, that automatically execute all the operations involved in the iBus and AVL system during the travel period of the bus. We represent a bus as a client in the proposed system. That is, every client (bus) receives periodically from the

Table 4
Description of new entities.

Entity	Fields
"Line"	• Identification of the line in a specific direction: "id_line" +"direction" • Identification of the line ("id_line"). • Name of the bus line ("line_name"). • Direction of the journey "inbound" or "outbound" ("direction"). • Identification of the origin location ("id_origin"). • Identification of the destiny location ("id_destination"). • Presence of any disruption in the line ("has_line_disruption").
"LineRoute"	• Identification of the line route: "id_line" +"direction" +"id_origin_location" • Identification of the line ("id_line"). • Direction of the journey "inbound" or "outbound" ("direction"). • Identification of the origin location ("id_origin_location"). • Identification of the next location ("id_next_location").
"BusInRoute"	• Identification of the bus route: "id_bus" +"id_line" +"direction" +"id_current_location" • Direction of the journey "inbound" or "outbound" ("direction"). • Identification of the current bus stop ("id_current_location"). • Identification of the next bus stop ("id_next_location"). • Current time ("time_stamp"). • Remaining time to the next bus stop ("time_to_station"). • Expected arrival time to the next bus stop ("expected_arrival_time"). • Valid time ("time_to_live").

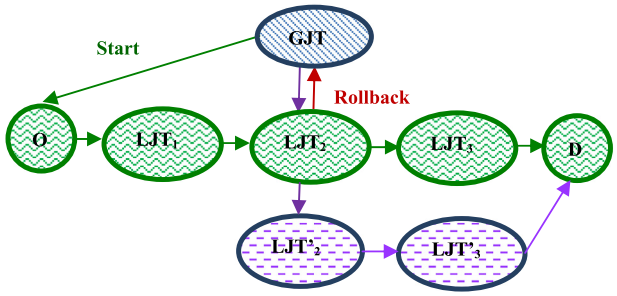

Fig. 8. An example of a "Global Journey Transaction".

AVL Centre which route should follow in the city of London at the depot, e.g., once a day. Initially, when a bus starts its journey, it will send a request to the module "Client manager" (see Fig. 1 Section 3). After that, the request will be sent to the module "Coordinator", which should act as a controlling mechanism for the client's journey. It will create a transaction named as "Global Journey Transaction" (GJT) and should manage the overall transaction.

During the journey a bus may face disruptions due to traffic jams, accidents, road work, etc. In case of disruptions the AVL system may devise an alternative route for the bus. With alternative route some of the LJT may need to be cancelled. Fig. 8 shows an example scenario of the GJT and LJTs in relation to the bus that travels from an origin stop (O) to a destination stop (D). The green path represents the initial route which has been planned by the system from the origin to the destination. Suppose that a transport problem (or disruption) occurs before the bus arrives at the second bus stop (associated with LJT_2). In this case, if the system has to devise an alternative route (purple path in Fig. 8) for the bus then completed operations (of LJT_2) need to be cancelled (compensated). That is the information (e.g., expected arrival time of a bus) already sent to the sources (display panels at bus stops or 'next-stop signage' screen in the bus) will not be valid (or consistent) anymore.

Contexual information of GJT and LJT

This section explains the contextual information related to GJT and LJT. Initially, every time a "Global Journey Transaction" (GJT) is created a new context is linked to it. Context can be external and internal. External Context (EC) represents data in relation with the client's requirements. Internal Context (IC) refers to a set of parameters which should identify the transaction, control concurrency requests, and other properties in order to tune CRUD operations performance in the NoSQL key/value database (scalability, fault-tolerance and availability). A GJT will be identified by a unique sequence number. Some of the properties are presented in Table 5.

Table 5
Properties of a "Global Journey Transaction" (GJT).

GJT context	
External	Internal
• Public transport identifier • Origin and destination • Maximum journey time	• Level of availability • Level of consistency

Context information related to the LJT reflects the conditions before starting the route from the current bus stop to the following one. The LJT will be identified by the concatenation of the global transaction number and another unique sequence number. ("Number of GJT" + sequence number). Moreover, the availability and consistency properties will be inherited from the global transaction context. The external context will represent information in relation to the current bus stop. The properties for both contexts are shown in Table 6.

Table 6
Properties of a "Location Journey Transaction" (LJT).

LJT context	
External	Internal
• Current bus stop • Next bus stop • Time to the next station • Expected arrival time • Maximum arrival time • Disruptions	• Level of availability • Level of consistency

Examples of transactional operations:

The proposed transactional system can process requests from different users (bus central station management or bus crew or passengers). Such requests are executed atomically on a single view of the NoSQL database. For this reason, every request will be linked to a transaction. Therefore, it is necessary to define what types of requests, a set of entities and CRUD operations are linked to each of them. The Tables 7 and 8 present such information.

Table 7
Requests to the OBU.

Types of requests	Entity	CRUD operations
Next bus location	LineRoute	**Input:** id_bus, id_line, direction, id_current_location **Read**: id_next_location
	BusInRoute	**Input:** id_bus, id_line, direction, id_current_location **Update:** id_next_location, timestamp, time_to_station, expected_arrival_time, time_to_live
Check disruption	BusStop	**Input:** id_location **Read:** has_location_disruption
	Line	**Input:** id_location **Read:** has_line_disruption

Table 8
Requests to the AVL centre.

Types of requests	Entity	CRUD operations
Change route	LineRoute	**Input:** id_line, direction, id_current_location **Update:** id_next_location
	BusInRoute	**Input:** id_bus, id_line, direction, id_current_location **Update:** id_next_location, time_stamp, time_to_station, expected_arrival_time, time_to_live
Change arrival time	BusInRoute	**Input:** id_bus, id_line, direction, id_current_location **Update:** time_stamp, time_to_station, expected_arrival_time, time_to_live
Set disruption	Line	**Input:** id_line, direction **Update:** has_line_disruption
	BusStop	**Input:** id_location **Update:** has_location_disruption

7. Evaluation and testing

This section presents the experiments in order to evaluate the proposed framework. It employs software testing techniques in order to check the correctness of transactional and non-transactional CRUD operations of the proposed framework according to the application requirements and data consistency. Several test cases are designed based on information about the proposed framework or the Software Under Test (SUT). We adopt the software testing strategy which is based multi-dimensional criteria [58]. The generation of test cases is based on the execution protocols presented in Figs. 2 and 3 (Section 3). Every path of the graph (in Figs. 2 and 3) is represented a test case. A path represents a sequence of states which are to be checked or tested during the execution of every CRUD operation of a transaction. The testing strategy has been developed according to the following concepts:

1. ***Test basis***: it represents all sources from which the requirements of a system could be inferred. In our case, this represents the logic-control and execution of transactional/non-transactional CRUD operations and their behaviour or outcomes (Figs. 2 and 3 in Section 3).
2. ***Test items***: These are the minimal units to be tested independently. The aim is to design test cases which represent different possibilities during the execution of CRUD operations (of a transaction), in order to detect possible faults or inconsistency of data. Some examples are given below.
    - **Single version of the data, no user decision and a rollback operation**: In this case the outcome (or data value) is unique, i.e. only one version of the requested data is provided by the NoSQL database when a transaction (of a NoSQL application) is executed. The acceptance of the outcome depends on: (a) the contextual information of a transaction—i.e., it is possible that NoSQL application may reject the outcome due to the fact that it does not meet contextual information of a transaction. Therefore, this process will undo the execution of the operation (rollback); or (b) user's own view or the level of consistency of the data—i.e. it is also possible that the user may accept the outcome even though it does not meet the contextual information. The flow of the operation is as follow: (''Check CRUD Requirements'' – ''Check IC Requirements'' – ''Check Unique CRUD Result'' – ''Check availability'' – ''Check Final IC Decision'' – ''Check Final CRUD Operation Result'' – ''CRUD Rollback Operation'' – ''Terminate'').
    - **Single version of the data, no user decision and with a successful end**: In this case the outcome is unique, i.e. only one version of the requested data is provided by the NoSQL database. Indeed, the application should accept it due to the fact that it obeys client's requirements or contextual information. Moreover, the user may accept the outcome depending on the requirement of the level of consistency needed. The flow is as follow: (''Check CRUD Requirements'' – ''Check IC Requirements'' – ''Check Unique CRUD Result'' – ''Check availability'' – ''Check Final IC Decision'' – ''Check Final CRUD Operation Result'' – ''Terminate'').
    - **Multi-version of the data, no user decision and a rollback operation**: In this case the outcome is not unique, i.e. several versions of the requested data are provided by the NoSQL database. This situation is more complex. It is more likely that NoSQL application rejects the outcome as there are more than one version. Therefore, this process will undo the execution of the operation (rollback). But the user may accept the outcome if any of the data version meets user's requirement in relation to the level of data consistency. The flow is as follow: (''Check CRUD Requirements'' – ''Check IC Requirements'' – ''Check Multi-version CRUD Result'' – ''Check System Result'' – ''Check consistency/availability'' – ''Check Final IC Decision'' – ''Check Final CRUD Operation Result'' – ''CRUD Rollback Operation'' – ''Terminate'').
3. ***Test conditions***: It represents the execution conditions linked to any ''Test item''. Therefore, the internal context of the system must be configured in order to tune a number of parameters in relation to data consistency and availability. In Riak, there are two parameters which determine how

CRUD operations will work: (1) the number of nodes where data must be replicated (N) to guarantee a specific level of consistency/availability, and (2) the level of success of read operation, r ($r \leq N$), and write operation, w ($w \leq N$). The goal is to measure the influence of the aforementioned parameters over the "Test items".

4. ***Test coverage item***: It is the set of properties from the "Test conditions" to which some specific values have been assigned. In the proposed framework, the datacenter was made of 5 nodes, the number replicas is equal to 3 ($N = 3$), and different levels of consistency for read ($r \in \{1, 2, 3\}$) and write CRUD operations ($w \in \{1, 2, 3\}$) have been tackled.
5. ***Test case***: It is a combination of "Test coverage item", where parameters will be assigned to a specific value.
6. ***Test suite***: It represents the set of test cases.

The experiments have been executed over one machine with the following hardware/software features: a CPU core with 2.4 GHz Intel(R) Core(TM) i7-5500, the operating system Ubuntu 14.04 LTS of 64 bits, Eclipse Luna 4.4.2 as IDE (Integrated Development Environment), Oracle Java 7 as the programming language, and the client API supported by the NoSQL key/value Riak (by Basho) 2.1.1 as database management. The simulation has been run over a cluster of five nodes over one CPU, the NoSQL key/value store Riak, and a number of client's requests.

Several experiments have been performed with several concurrent journey requests. The experiments simulate the bus travel scenario of line 176, wherein information is sent/stored to/in the central database Riak and then communicated with the various output sources such as display panels at bus stops, 'next-stop signage' screen in the buses and websites showing real time information about buses. Note that in our experiments we do not design the display panels at bus stops or websites. Instead, these are simulated.

Experiments have been performed in order to measure the degree of correctness from the system response with different data arrival rates (velocity) and different levels of consistency. The level of data correctness depends on both the arrival speed of the data to the system (NoSQL database), and how strong or weak data consistency is set.

During the experiments, different timetables data (bus location, arrival times, journey times, etc.) were regularly written into the NoSQL key/value database. This is to emulate the arrival of new data from the iBus system in relation to the line 176 (Fig. 5, Section 5). The simulation reveals that the information sent to the output sources (e.g., display panels at bus stops) could be distorted due to the following factors: the frequency of updating timetable data in the NoSQL database and the existence of multi-version data.

Firstly, the arrival rate of data (velocity) has been analysed during the testing procedure. Specially, if the arrival of new data for a specific bus stop is not recorded as a part of a transaction (in an atomic manner), then data would be inconsistent and there would be mixture of new and old information shown on the output sources and display panels at bus stops, etc.

Secondly, the NoSQL key/value database has an asynchronous replication policy. This fact reflects that old and new data could coexist in different nodes at the same time. This fact should be taken into account when the selection of the newest version is crucial and some nodes are failing. Indeed, the accuracy in the outcome will improve as the level of write operations consistency increases. This is due to the fact that it is compulsory that the newest data are updated in a higher number of nodes. Therefore, the selection of a stale version is highly likely to arise as the level of write operation consistency decreases.

7.1. Test cases using the bus data

Tests have been based on a collection of data, which has been taken from the London Datastore for a specific bus line 176 at different intervals of time. A sample data is represented in Table 9.

Table 9
A snippet from the open data of the bus line 176.

	From high street/maple road to high street/green lane		
Timestamp	13:29	13:30	13:31
Time to station	633	573	712
Expected arrival time	13:40	13:40	13:43

In the above table, "Timestamp" is the time from the system at which data were collected. "Time to station" represents the number of seconds that the bus journey is highly likely to take from the current station to the next one. "Expected arrival time" is the time that the bus should arrive to the next station. Moreover, it could be observed that the "Time to station" value should decrease as the bus is approaching to the next bus stop. This situation is shown if we compare data from the "Timestamp" equals to 13:29 and 13:30. However, if a delay is expected then "Time to station" and "Expected arrival time" should increase (dependency relation). Indeed, this possibility is shown when the timestamp is equal to 13:30 with an arrival time at 13:40 (573 sec.), but one minute after (13:31) the bus would arrive at 13:43 (712 s). This implies that it is important to write data which belongs to a specific interval of time as a whole in the NoSQL database (i.e. write operation should be part of a transaction). Therefore, data which belong to different "Timestamp" must not be overlapped in the NoSQL database. On the contrary, data visualization on the screen could be mixed up and could become inconsistent. For example, it is possible that data from the "Timestamp" 13:31 are presented on the screen but "Time to station" remains with the value 573 instead of 712. In summary, the fields "Timestamp", "Time to station" and "Expected arrival time" should obey certain relationships between them and their values should be updated as part of a transaction.

Several experiments have been conducted with several read and write operations. Each read and write request has been associated with a transaction. Several problems arose during the execution of the tests. Firstly, if the execution time between reads and writes is very short, data are likely to be overlapped in the NoSQL database. It could produce a temporal inconsistent state of data. Secondly, if no read operations are executed between any pair of write operations, then some data could be missed by the user (this is called read skew in the literature).

Our framework uses the context information in order to preserve the consistency between concurrent read requests. For this reason, the client (of the system) should check the relations between data, what it has been called in this paper as the dependencies for the external context and the semantic rules. However, this example is focused on the control for the correctness of the relationships between the different fields mentioned above. To ensure this, we have devised an algorithm in Table 10, where "Timestamp 1" represents a system time before the "Timestamp 2" (dependency relation).

Table 10
External context algorithm.

```
if (Timestamp 1 ≥ Timestamp 2)
    then data arrival failure
    else if (Time to station 1 >Time to station 2)
            then if (Exp. arrival time 1 ≠ Exp. arrival time 2)
                    then data arrival failure
                    else data seem to be right
                 endif
            else //It seems to be a delay
                 if (Exp. arrival time 1 >Exp. arrival time 2)
                    then data arrival failure
                    else data seem to be right
                 endif
            endif
endif
```

The above observations reveal the possibility of two problems that could lead to a wrong or inconsistent data due to a high arrival rate of data in the database. Firstly, it should be taken into account that if a delay is not expected ("Time to station 1 >Time to station 2") then the bus should arrive on time ("Expected arrival time 1 == Expected arrival time 2"). If a delay happens ("Time to station 1 <Time to station 2") then the bus would not arrive on time ("Exp. arrival time 1 <Exp. arrival time 2"). In both situations, there is a dependency relation between a bus arrival time and the presence/absence of a delay. Therefore, data arrival may fail as follows:

A. When the expected arrival time from the previous state is different from the expected arrival time from the current state ("Exp. arrival time 1 ≠ Exp. arrival time 2")
B. When the expected arrival time from the previous state could be higher than the expected arrival time from the current state ("Exp. arrival time 1 >Exp. arrival time 2") respectively.

As a consequence, data would be inconsistent of a write operation (in both situations, A & B) is not executed as part of transaction. This is due to the fact that the different values of data ("Timestamp", "Time to station" and "Expected arrival time") which belong to the same period of time could be mixed up from one period to the next one.

7.2. Experiments with different ratios of read and write consistency

Different experiments were performed in order to take into account the following ratio of reads to writes operations [59]: 70:30, 80:20 and 90:10, and nine levels of consistency which result from the combination of different values from the r and w parameters ($r \in [1,3]$, $w \in [1,3]$) named as "Case 1", "Case 2" and so on ("Case 1":"$r = 1; w = 1$"; "Case 2": "$r = 1; w = 2$"; "Case 3":"$r = 1; w = 3$"; "Case 4":"$r = 2; w = 1$"; "Case 5":"$r = 2; w = 2$"; "Case 6":"$r = 2; w = 3$"; "Case 7":"$r = 3; w = 1$"; "Case 8":"$r = 3; w = 2$"; "Case 9":"$r = 3; w = 3$"). Moreover, it is considered that the NoSQL database receives the information periodically at a constant rate similar to a real system (iBus system). However, users could read the information at any time from the screen or from Internet. Therefore the arrival of read requests has been performed randomly. Moreover, during the simulation, a thousand of CRUD operations (of transactions) were executed at different ratios, where write operations were executed at a constant rate (10 ms), and read operations at different instants in time between two write operations (1–10 ms).

The experiment, in Fig. 9 shows the number of write operations which could produce an inconsistent state in the NoSQL key/value database according to the algorithm in Table 10. Therefore, it has been checked to see the violation of any of the following context-aware relationships:

1. The timestamp of one period (Timestamp 1) must be lower than the next period (Timestamp 2): (Timestamp 1 <Timestamp 2).
2. The expected arrival time between two consecutive periods should be the same if no delays occur, and the time to station between them should decrease: (Timestamp 1 <Timestamp 2) and (Time to station 1 >Time to station 2) and (Exp. arrival time 1 == Exp. arrival time 2).
3. The expected arrival time between two consecutive periods should increase if a delay occurs, and also the time to station (Timestamp 1 <Timestamp 2) and (Time to station 1 ≤ Time to station 2) and (Exp. arrival time 1 ≤ Exp. arrival time 2) should increase.

If the NoSQL database is set with the strongest level of consistency (Case 9), and 300 write operations are executed out of 1000 ("70:30"), then the number of write inconsistencies fluctuates between 24 and 30. The proportion of write inconsistencies is similar in both "80:20" and "90:10". However, it is important to take into account the impact that a write inconsistency could cause in read operations. This problem has been represented in Fig. 10, in terms of the number of read inconsistencies. Looking on Case 9 and "70:30" again, it could be observed that 42 reads out of 700 will violate any of the three context-aware situations. Specially, the experiment highlighted that this problem seems to become more obvious in the ratio "90:10", where the number of erroneous read operations could fluctuate between 700 and below 760. However, this result has not been represented in the figure due to its big difference in comparison to the other ratios ("70:30" and "80:20"). The number of erroneous read operations varies from 29 (Case 4) to 46 (Case 1) in 70:30, and from 39 (Case 8) to 45 (Case 3) in 80:20.

This result could be due to fact that the interval of time between write operations is not big enough. Indeed, the system seems not to have time to reach a stable state before the next write operation arrives. Therefore, if read operations are concurrently executed with the write operations then they may provide misleading (or inconsistent) data.

The experiment has been conducted by writing a set of different pieces of data ("Timestamp", "Time to station" and "Expected arrival time") in the NoSQL database at a constant rate. During this operation, there are two possibilities:

1. The set could be treated as a whole write unit on its own, i.e., no concurrent read/write operations should be allowed in the middle of the operation.
2. The different pieces of information could be written in sequential order and independently one from another, i.e., concurrent read/write operations should be allowed in the middle of the operation.

The first option implies to implement a mechanism that locks data during write operations. However, it could introduce delays and would affect the system response. The second option appears to work faster but it could provide misleading data to the user. This is because old and new data can co-exist during while data are updated in the database.

In our experiments, we measured the influence of two options by taking into account interval of time between write operations (velocity of data updates) and the level of consistency. As shown in Figs. 9 and 10, the number of inconsistencies in read and write operations remains high for all levels of consistency (Case 1 to Case 9), and also for all read to write proportions ("70:30", "80:20" and "90:10"). Therefore, the velocity of write operations does not allow the database to reach a stable state. It will have an impact on the system results, which will be propagated to the electronic panels for their visualization. For this reason, we propose to control the introduction of new data in the database according to the algorithm

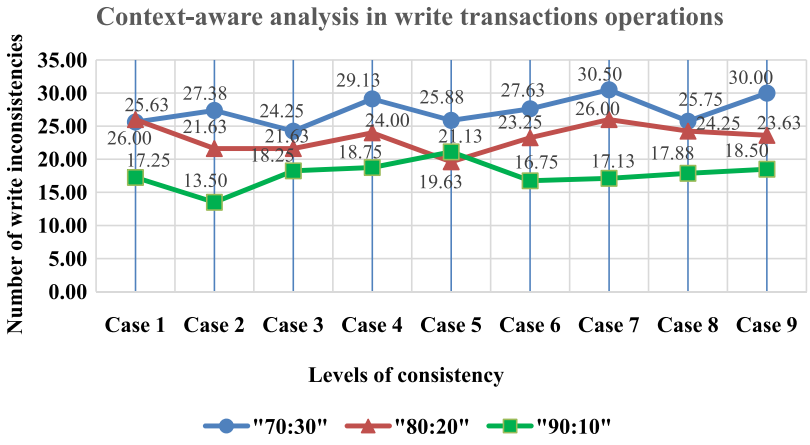

Fig. 9. Analysis of context-aware write inconsistencies.

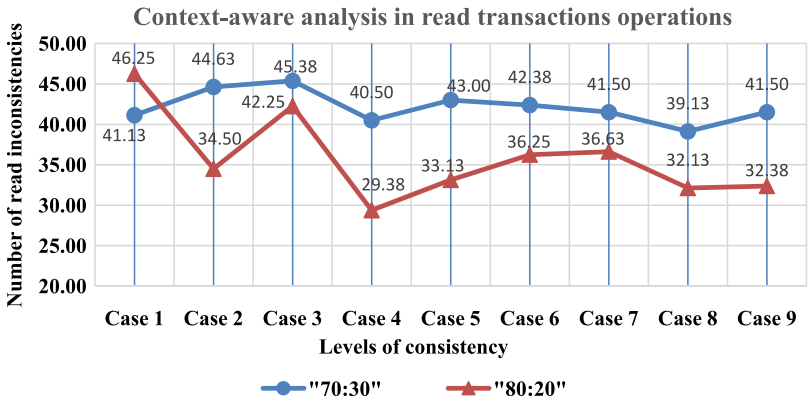

Fig. 10. Analysis of context-aware read inconsistencies.

in Table 10 (context-aware) in case of write inconsistencies. In this case, some write operations should need to be undone (transactions) to keep the database in a consistent state. As a conclusion, it should be look into a trade-off between the interval of time for write operations and the number of transactions needed, in order to provide appropriate outcomes (or consistent data).

8. Conclusion

NoSQL databases are emerged as new data management systems in order to manage and process large volume of data in an efficient and scalable manner. However, they compromise on the level of data consistency and other classical database features such as relationships, transactions, and integrity constraints. This paper focused on transactional services which provide NoSQL databases with enhanced consistency as well as concurrency and reliability. In particular, it addressed a critical issue of testing transactional services in NoSQL databases which has not been addressed in the current literature.

The novel features of this paper are to design and develop a new framework that takes account of context-aware transactions and a real big data from the London Datastore about the London bus services. Test data was modelled in such a way that can be represented and stored in NoSQL key/value databases. The framework has the potential to analyse the impact of the big data requirements and characteristics (e.g., availability and velocity) on the consistency of NoSQL databases. It can also assist application developers in choosing between transactional and non-transactional NoSQL operations based on their preference and requirements as well as contextual information.

Acknowledgements

We would like to express our gratitude to the Oxford Brookes University, Oxford, UK, and two financial sponsors of this work: the Spanish Research, Development and Innovation Plan supported by the Ministry of Economy and Competitiveness (TIN2013-46928-C3-1-R and TIN2016-76956-C3-1-R) and the Principality of Asturias (GRUPIN14-007).

References

[1] M. Chen, et al., Big data: A survey, Mob. Netw. Appl. 19 (2014) 171–209.
[2] M.A. Mohamed, et al., Relational vs. nosql databases: A survey, Int. J. Comput. Inf. Technol. 3 (2014) 598–601.
[3] A. Tripathi, B. Thirunavukarasu, Design and Evaluation of a Transaction Model with Multiple Consistency Levels for Replicated Data, 2015.
[4] V. Padhye, A. Tripathi, Scalable transaction management with snapshot isolation for NoSQL data storage systems, IEEE Trans. Serv. Comput. 8 (2015) 121–135.
[5] S. Kiljan, et al., Evaluation of transaction authentication methods for online banking, Future Gener. Comput. Syst. (2016).
[6] S. Gilbert, N. Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, ACM SIGACT News 33 (2002) 51–59.
[7] A. Moniruzzaman, S.A. Hossain, Nosql database: New era of databases for big data analytics-classification, characteristics and comparison, 2013, arXiv preprint arXiv:1307.0191.
[8] P.J. Sadalage, M. Fowler, NoSQL Distilled: A Brief Guide To the Emerging World of Polyglot Persistence, Pearson Education, 2012.
[9] G. DeCandia, et al., Dynamo: amazon's highly available key-value store, SIGOPS Oper. Syst. Rev. 41 (2007) 205–220.
[10] F. Chang, et al., Bigtable: A distributed storage system for structured data, ACM Trans. Comput. Syst. 26 (2008) 1–26.

[11] A. Lakshman, P. Malik, Cassandra: structured storage system on a P2P network, in: presented at the Proceedings of the 28th ACM symposium on Principles of distributed computing, Calgary, AB, Canada, 2009.
[12] D.G. Campbell, et al., Extreme scale with full SQL language support in microsoft SQL Azure, in: presented at the Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, Indianapolis, Indiana, USA, 2010.
[13] B.F. Cooper, et al., PNUTS: Yahoo!'s hosted data serving platform, Proc. VLDB Endow. 1 (2008) 1277–1288.
[14] J.C. Corbett, et al., Spanner: Google’s globally distributed database, ACM Trans. Comput. Syst. 31 (2013) 1–22.
[15] W. Lloyd, et al., Don't settle for eventual: scalable causal consistency for wide-area storage with COPS, in: presented at the Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, Cascais, Portugal, 2011.
[16] J. Cowling, B. Liskov, Granola: low-overhead distributed transaction coordination, in USENIX ATC'12, Boston, MA, 2012.
[17] R. Escriba, et al., Warp: Multi-Key Transactions for Key-Value Stores, 2013.
[18] J. Baker, et al., Megastore: Providing Scalable, Highly Available Storage for Interactive Services, in: CIDR, 2011, pp. 223–234.
[19] J. Baker, et al., Megastore: Providing Scalable, Highly Available Storage for Interactive Services, in: Innovative Data system Research, CIDR, Asilomar, California, 2011, pp. 223–234.
[20] S. Das, et al., G-Store: a scalable data store for transactional multi key access in the cloud, in: presented at the Proceedings of the 1st ACM symposium on Cloud computing, Indianapolis, Indiana, USA, 2010.
[21] J.J. Levandoski, et al., Deuteronomy: Transaction Support for Cloud Data, in: CIDR, 2011, pp. 123–133.
[22] K. Chitra, B. Jeevarani, Cloud TPS: Scalable transaction in the cloud computing, Int. J. Eng. Comput. Sci. 2 (2013) 2280–2285.
[23] F. Coelho, et al., pH1: middleware transacional para NoSQL, in: IEEE 33rd International Symposium on Reliable Distributed Systems, SRDS, 2014, pp. 115–124.
[24] R. Jimenez-Peris, et al., CumuloNimbo: A cloud scalable multi-tier SQL database, IEEE Data Eng. Bull. 38 (2015) 73–83.
[25] A.E. Lotfy, et al., A middle layer solution to support ACID properties for NoSQL databases, J. King Saud Univ. Comput. Inf. Sci. 28 (2016) 133–145.
[26] D. Peng, F. Dabek, Large-scale incremental processing using distributed transactions and notifications, in: OSDI, 2010, pp. 1–15.
[27] F. Junqueira, et al., Lock-free transactional support for large-scale storage systems, in: Dependable Systems and Networks Workshops, DSN-W, 2011 IEEE/IFIP 41st International Conference on, 2011, pp. 176–181.
[28] V. Vafeiadis, et al., Proving correctness of highly-concurrent linearisable objects, in: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, 2006, pp. 129–136.
[29] M.P. Herlihy, J.M. Wing, Linearizability: A correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (1990) 463–492.
[30] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Trans. Comput. 28 (1979) 690–691.
[31] M. Ahamad, et al., Causal memory: Definitions, and implementation, and programming, Distrib. Comput. 9 (1995) 37–49 1995/03/01.
[32] R.J. Lipton, J.S. Sandberg, PRAM: A Scalable Shared Memory, Princeton University, Department of Computer Science, 1988.
[33] S. Burckhardt, et al., Eventually consistent transactions, in: Programming Languages and Systems, Springer, 2012, pp. 67–86 ed.
[34] W. Vogels, Eventually consistent, Commun. ACM 52 (2009) 5.
[35] Y. Sovran, et al., Transactional storage for geo-replicated systems, in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011, pp. 385–400.
[36] A.K. Dey, Understanding and using context, Pers. Ubiquitous Comput. 5 (2001) 4–7.
[37] B.N. Schilit, M.M. Theimer, Disseminating active map information to mobile hosts, IEEE Netw. 8 (1994) 22–32.
[38] J.-y. Hong, et al., Context-aware systems: A literature review and classification, Expert Syst. Appl. 36 (2009) 8509–8522.
[39] S. Vaupel, et al., A generic architecture supporting context-aware data and transaction management for mobile applications, in: Proceedings of the International Workshop on Mobile Software Engineering and Systems, 2016, pp. 111–122.
[40] G. Bernard, et al., Mobile databases: A selection of open issues and research directions, ACM SIGMOD Rec. 33 (2004) 78–83.
[41] M. Younas, S.K. Mostéfaoui, Context-aware mobile services transactions, in: Advanced Information Networking and Applications, AINA, 2010 24th IEEE International Conference on, 2010, pp. 705–712.
[42] D. Ma, N. Saxena, A context-aware approach to defend against unauthorized reading and relay attacks in RFID systems, Secur. Commun. Netw. 7 (2014) 2684–2695.
[43] P. Pushpa, Customer Context Based Transactions in Mobile Commerce Business Environment, in: e-Business Engineering, ICEBE, 2016 IEEE 13th International Conference on, 2016, pp. 208–213.
[44] A.N. Steinberg, G. Rogova, Situation and context in data fusion and natural language understanding, in: Information Fusion, 2008 11th International Conference on, 2008, pp. 1–8.
[45] M.T. Gonzalez-Aparicio, et al., A new model for testing CRUD operations in a NoSQL databases, in: presented at the IEEE 30th International Conference on Advanced Information Networking and Applications, Crans-Montana, Switzerland, 2016.
[46] Database Transaction Models for Advanced Applications, Morgan Kaufmann Publishers Inc, 1992.
[47] H. Berenson, et al., A critique of ANSI SQL isolation levels, SIGMOD Rec. 24 (1995) 1–10.
[48] S. Revilak, et al., Precisely Serializable Snapshot Isolation (PSSI), in: presented at the Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, 2011.
[49] TfL Live Bus River Bus Arrivals API Documentation v2.1 Available: https://tfl.gov.uk/corporate/terms-and-conditions/live-bus-departure-information.
[50] Greater London Authority (GLA) - London datastore. Available: https://api.tfl.gov.uk.
[51] N. Hounsell, et al., Data management and applications in a world-leading bus fleet, Transp. Res. C 22 (2012) 76–87.
[52] Open Knowledge Foundation.
Available: http://okfn.org/, http://www.opendefinition.org/okd/.
[53] Transport for London. Available: https://tfl.gov.uk/corporate/about-tfl/what-we-do/buses.
[54] Buses from Waterloo. Available: http://content.tfl.gov.uk/bus-route-maps/waterloo-120915.pdf.
[55] Transport for London unified API. Available: https://api.tfl.gov.uk/Line/176/Arrivals?app_id=&app_key=.
[56] National Public Transport Access Nodes (NaPTAN). Available: https://data.gov.uk/dataset/naptan.
[57] I. Basho Technologies. (2011–2017). Developing with Riak KV - Datatypes. Available: http://docs.basho.com/riak/kv/2.2.0/developing/data-types/.
[58] R. Casado, et al., Multi-dimensional criteria for testing web services transactions, J. Comput. System Sci. 79 (2013) 1057–1076.
[59] A. Dey, et al., YCSB+T: Benchmarking web-scale transactional databases, in: Data Engineering Workshops, ICDEW, 2014 IEEE 30th International Conference on, 2014, pp. 223–230.

María Teresa González-Aparicio is an Associate Lecturer in Computing at the Department of Computer Science, Oviedo University, Spain. Her research interests include NoSQL databases, transactions in cloud and services computing and software testing. She received a Ph.D. in Computer Science from the University of Oviedo, Spain. She has published in international journals and conferences.

Muhammad Younas is a Senior Lecturer in Computing at the Department of Computing and Communication Technologies, Oxford Brookes University, UK. His research interests include Web technologies, cloud and services computing, pervasive and mobile information systems. He received a Ph.D. in Computer Science from the University of Sheffield, UK. He has published more than hundred papers in international journals and conferences. He is on the editorial and advisory boards of international journals and is also involved in the steering, organizing and program committees of refereed international conferences and workshops.

Javier Tuya is a professor at the University of Oviedo, Spain, where he is the research leader of the Software Engineering Research Group. He received his Ph.D. in Engineering from the University of Oviedo in 1995. He is Director of the Indra-Uniovi Chair, member of the ISO/IEC JTC1/SC7/WG26 working group for the recent ISO/IEC/IEEE 29119 Software Testing standard and convener of the corresponding AENOR National Body working group. His research interests in software engineering include verification & validation and software testing for database applications and services. He is a member of the IEEE, IEEE Computer Society, ACM and the Association for Software Testing (AST).

Rubén Casado received a B.Sc. degree in Computer Science in 2005, a M.Sc. in Computing in 2008 and a PhD in Software Systems in 2013 from University of Oviedo, Spain. He has worked as a researcher and teaching assistant at the University of Oviedo, where he is currently a member of the Software Engineering Research Group. He has also collaborated as a visiting researcher with the Oxford Brook University (Oxford, UK), and the LORIA/INRIA team (Nancy, France). He has been involved in many research projects, including those with national (TIN, PCTI) and international scope (FP7, H2020). He has also been the leader of the Big Data research program at Treelogic, Spain. Currently he is working as a Big Data Manager at the Accenture Digital, Madrid, Spain.