Big Data and Information Analytics
©American Institute of Mathematical Sciences
Volume **1**, Number **2&3**, April & July **2016**

doi:10.3934/bdia.2016004
pp. **185–216**

HOW DO I CHOOSE THE RIGHT NOSQL SOLUTION? A COMPREHENSIVE THEORETICAL AND EXPERIMENTAL SURVEY

Hamzeh Khazaei, Marios Fokaefs, Saeed Zareian
Nasim Beigi-Mohammadi, Brian Ramprasad, Mark Shtern
Purwa Gaikwad and Marin Litoiu

Center of Excellence for Research in Adaptive Systems
York University
Toronto, Ontario, Canada

(Communicated by Jianhong Wu)

Abstract. With the advent of the Internet of Things (IoT) and cloud computing, the need for data stores that would be able to store and process big data in an efficient and cost-effective manner has increased dramatically. Traditional data stores seem to have numerous limitations in addressing such requirements. NoSQL data stores have been designed and implemented to address the shortcomings of relational databases by compromising on ACID and transactional properties to achieve high scalability and availability. These systems are designed to scale to thousands or millions of users performing updates, as well as reads, in contrast to traditional RDBMSs and data warehouses. Although there is a plethora of potential NoSQL implementations, there is no one-size-fit-all solution to satisfy even main requirements. In this paper, we explore popular and commonly used NoSQL technologies and elaborate on their documentation, existing literature and performance evaluation. More specifically, we will describe the background, characteristics, classification, data model and evaluation of NoSQL solutions that aim to provide the capabilities for big data analytics. This work is intended to help users, individuals or organizations, to obtain a clear view of the strengths and weaknesses of well-known NoSQL data stores and select the right technology for their applications and use cases. To do so, we first present a systematic approach to narrow down the proper NoSQL candidates and then adopt an experimental methodology that can be repeated by anyone to find the best among short listed candidates considering their specific requirements.

1. **Introduction.**

The term “NoSQL” was first coined in 1998 by Carlo Strozzi for his RDBMS, Strozzi NoSQL [37]. However, Strozzi used the term simply to distinguish his solution from other relational database management systems (RDBMS), which utilize SQL. He used the term NoSQL just for the reason that his database did not expose a SQL interface. Recently, the term NoSQL (meaning ‘Not only SQL’) has come to describe a larger class of databases, which do not have the same

2010 *Mathematics Subject Classification.* Primary: 68P20, 68P15; Secondary: 68M20, 68M14.
Key words and phrases. NoSQL datastores, key-value datastores, wide-column datastores, graph databases, document datastores, future trends, systematical selection, YCSB, performance evaluation, distributed datastores.

properties as traditional relational databases and are generally not queried with SQL.

Recently the term "NoSQL" has revived in a different context, generally known as the era of *Big Data*. Big Data is defined as a collection of data sets, which are enormously large and complex that conventional database systems cannot process within desired time [9]. For instance, storing and processing daily tweets at Twitter demand significant data storage, processing, and data analytics capabilities. Although conventional SQL-based databases have proven to be highly efficient, reliable, and consistent in terms of storing and processing structured (or relational) data, they fall short of processing Big Data, which is characterized by large volume, variety, velocity, openness, absence of structure, and high visualization demands among others [9]. Internet-born Companies like Google, Amazon and Facebook have invented their own data stores to cope with the big data that appear in their applications and have inspired other vendors and open source communities to do similarly for other use cases [16].

Figure 1 shows the the Big Data challenges and the corresponding features of NoSQL systems that try to address them. On one hand, in the domain of Big Data, we are obviously talking about very large data sets that should be available to large number of users at the same time (*volume*). There is also the need for fast data retrieval to enable real-time and critically efficient data analysis (*velocity*) and the data comes in a much greater *variety* of formats beyond structured and relational data. On the other hand, NoSQL data stores can accommodate the large volume of data and users by partitioning the data in many storage nodes and virtual structures, thus overcoming infrastructure constraints and ensuring basic availability. Additionally, NoSQL data stores relax the transactional properties of user queries by abandoning the ACID system for the BASE (Basic availability, Soft state, Eventual consistency) [32] system. This way there is less blocking between user queries in accessing particular data structures, a fact which is also supported by data partitioning. Finally, NoSQL data stores come in many flavors, namely data models, to accommodate the data variety that is present in real problems.

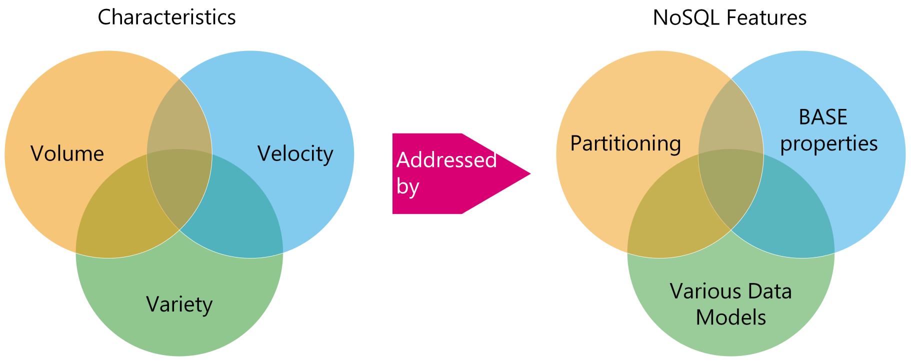

FIGURE 1. Big Data characteristics and NoSQL System features.

In this paper, we survey the domain of NoSQL in order to present an overview of the relevant technologies to store and process big data. We present these technologies according to their features and characteristics. For each category, we present a historical overview of how they came to be, we review their key features and competitive characteristics, we discuss some of their most popular implementations and, finally, we identify some future directions and trends for the evolution and advancement of NoSQL systems. In the second part of the paper, we demonstrate how to customize and leverage Yahoo! Cloud Serve Benchmark (YCSB) [12] for performing performance evaluation.

Our work aims to provide a comprehensive overview of state-of-the-art solutions for big data stores, but also to promote and motivate further research and advancement by concisely presenting, what both the academia and the industry have identified as important issues to be addressed in the future. Additionally, through the qualitative comparison and more importantly, providing the ability of quantitative comparison, the goal is to guide developers to select a solution fit for their needs. Finally, this paper contributes several extensions to YCSB, in terms of new drivers for specific implementations, updates for existing drivers and a new write-intensive workload. These extensions are made public with the goal to facilitate and further advance the benchmark and evaluation capabilities of YCSB for the rest of the research community.

The rest of the paper is organized as follows. In Section 2, we describe the common concepts, data models and terminologies in the NoSQL word. In Section 3, we first briefly introduce the classical classes of NoSQL solutions; then we elaborate each class along with top-rated sample datastores. In section 5 we demonstrate how to customize YCSB for new solutions or specific requirements for each user. Finally, in Section 6 we conclude the paper with a summary of new findings and observations of future trends in the area of NoSQL datastores.

2. **NoSQL technologies.**

NoSQL systems range in functionality from the simplest distributed hashing, as supported by the popular Memcached [22, 15], a distributed and open-source memory caching system, to highly scalable partitioned tables, as supported by Googles BigTable [11]. In fact, BigTable, Memcached, and Amazons DynamoDB [36] provided a "proof of concept" that inspired many of the data stores we describe here [12]. Memcached demonstrated that in-memory indexes can be highly scalable, distributing and replicating objects over multiple nodes. DynamoDB pioneered the idea of eventual consistency as a way to achieve higher availability and scalability; data fetched are not guaranteed to be up-to-date, but updates are guaranteed to be propagated to all nodes eventually. BigTable demonstrated that persistent record storage could be scaled to thousands of nodes, a feat that most of the other systems aspire to.

NoSQL systems generally have six key characteristics [10]:

1. the ability to horizontally scale CRUD operations throughput over many servers,
2. the ability to replicate and to distribute (i.e.,partition or shard) data over many servers,
3. a simple call level interface or protocol (in contrast to a SQL binding),
4. a weaker concurrency model than the ACID transactions of most relational (SQL) database systems,
5. efficient use of distributed indexes and RAM for data storage, and
6. the ability to dynamically add new attributes to data records.

2.1. **Transactional properties and performance.**

In order to guarantee the integrity of data, most of the classical database systems support transactions. This ensures consistency of data in all levels of data management. These transactional characteristics are also known as ACID (Atomicity, Consistency, Isolation, and Durability). However, scaling out of ACID-compliant systems has been shown to be impossible. This is due to conflicts that can arise between the different aspects of high availability in distributed systems, which are not fully solvable, known as the CAP theorem [28]:

- **Strong Consistency** implies that a system is in a consistent state after the execution of an operation. A Distributed system is typically considered to be consistent, if, after an update operation of some writer, all readers see the updates in some shared data source.
- **Availability** means that all clients can always find at least one copy of the requested data, even if some of the machines in a cluster are down,
- **Partition-tolerance** is understood as the ability of the system to continue operating normally in the presence of network partitions. These occur if two or more "islands" of network nodes arise which (temporarily or permanently) cannot connect to each other. Partition tolerance is also understood as the ability of a system to cope with the dynamic addition and removal of nodes (e. g. for maintenance purposes; removed and again added nodes are considered a network partition on their own in this notion).

The CAP-Theorem postulates that only two of the three different aspects of scaling out can be achieved fully at the same time (see Fig. 2).

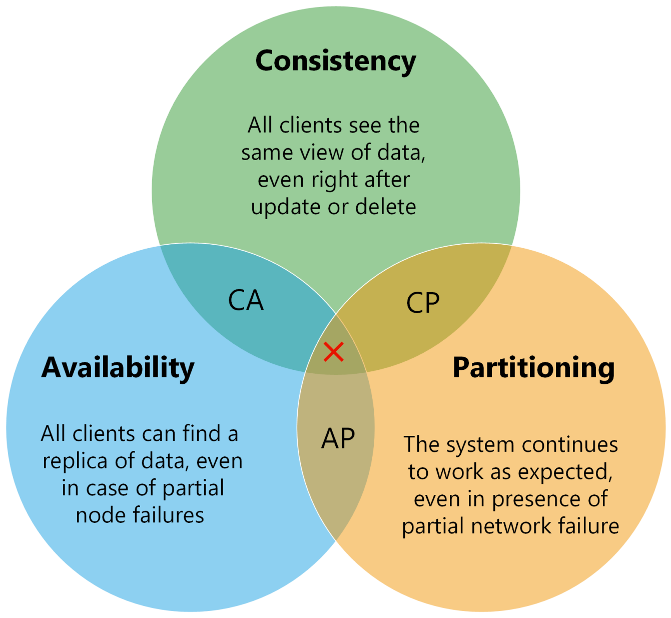
Consistency
All clients see the same view of data, even right after update or delete
CA
CP
×
Availability
Partitioning
AP
All clients can find a replica of data, even in case of partial node failures
The system continues to work as expected, even in presence of partial network failure

FIGURE 2. Visualization of CAP theorem.

Many of the NOSQL databases have loosened up the requirements on Consistency in order to achieve better Availability and Partitioning. This resulted in systems known as BASE (Basically Available, Soft-state, Eventually consistent). These have no transactions in the classical sense and introduce constraints on the data model to enable better partition schemes. Cattell [10] classifies NoSQL databases according to the CAP theorem.

While it is always the case that NoSQL stores claim to be faster in terms of performance than RDBMS systems, especially when concerning big data, it should be noted that not all NoSQL datastores are created alike where performance is concerned. Users, e.g., system architects and IT managers, are wise to compare NoSQL databases in their own environments using data and user interactions that are representative of their expected production workloads before deciding which NoSQL database to use for new application [1]. However, using performance evaluation tools designed specifically for NoSQL systems could help to narrow the list dramatically.

In this paper, we aim to provide both theoretical and experimental comparison among popular NoSQL solutions in order to assist users for choosing the right technology that is appropriate for their specific needs. We will employ Yahoo Cloud Serve Benchmarking (YCSB) [12] for the performance evaluation of selected NoSQL stores.

3. **Data store categories.**

There has been a number of approaches to classify NoSQL databases according to various criteria. In the context of our work, we adopt the classification based on the supported data model.

- **Key-value stores.** These are probably the simplest form of database management systems. They can only store pairs of keys and values, as well as retrieve values when a key is known. These simple systems are normally not adequate for complex applications. On the other hand, it is exactly this simplicity, that makes such systems attractive in certain circumstances. For example, resource-efficient key-value stores are often applied in embedded systems or as high performance in-process databases.
- **Document stores.** Document stores, also called document-oriented database systems, are characterized by their schema-free organization of data. Records (i.e., document) do not need to have a uniform structure, i.e. different records may have different columns. The types of the values of individual columns can be different for each record and columns can have more than one value (arrays). Records can have a nested structure and document stores often use internal notations, which can be processed directly in applications, mostly JSON.
- **Graph databases.** Graph DBMS, also called graph-oriented DBMS or graph databases, represent data in graph structures as nodes and edges, which represent relationships between nodes. They allow easy processing of data in that form, and simple calculation of specific properties of the graph, such as the number of steps needed to get from one node to another node.
- **Wide Column Stores.** Wide column stores, also called extensible record stores, store data in records with an ability to hold very large numbers of dynamic columns. Since the column names as well as the record keys are not fixed, and since a record can have billions of columns, wide column stores can be seen as two-dimensional key-value stores. Wide column stores share the characteristic of being schema-free with document stores, however the implementation is significantly different.

3.1. **Key-value data stores.**

Key-value stores are the most common and simplest NoSQL data stores that store pairs of keys and values, where values are retrieved when keys are known. Key-value stores are mainly used when there is a need for higher performance than SQL databases, but not for the accompanying rigid relational data models and the complex query functionalities of SQL and other NoSQL databases. Fig 3 represents a typical object in such datastores; this object is for loop detector sensors embedded in Ontario highways that measure the speed, volume and length of cars periodically[42].

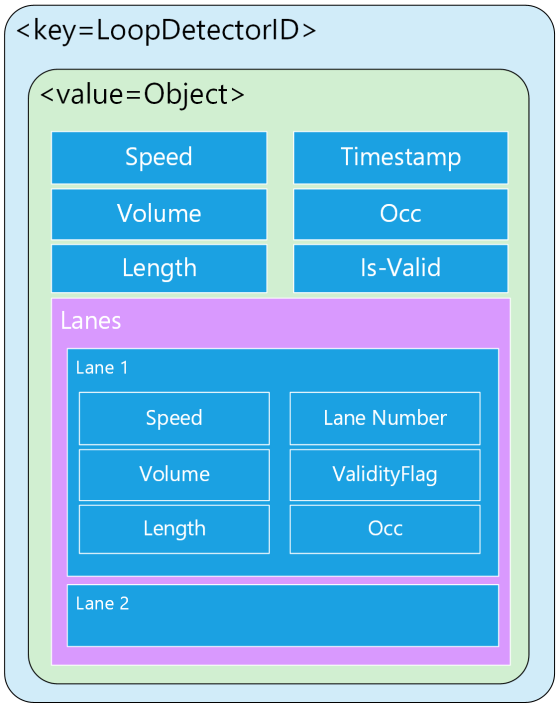

FIGURE 3. A Key value data model for traffic data.

Key-value stores can be categorized into three types in terms of storage option including temporary, permanent, and hybrid stores. In temporary stores, all the data are stored into memory, hence the access to data is fast. However, data will be lost if the system is down. Permanent key-value stores ensure high availability of data by storing the data on the hard disk but with the price of lower speed of I/O operations on the hard disk. Hybrid approaches combine the best of both temporary and permanent types by storing the data into memory and then writing the input to the hard disk when a set of specified conditions are met.

Berkeley DB [29] is one of the simplest examples of NoSQL key-value stores that provides fundamental functionalities of a key-value store and serves as a base for developing other advanced key-value stores such as Project Voldemort [14], used by LinkedIn and Amazon DynamoDB [36]. Berkeley DB provides ACID guarantees offered by SQL databases, while it provides a simple database with good performance and lightweight footprint.

Key-value stores are generally good solutions if you have a simple application with only one kind of object, and you only need to look up objects up based on one attribute. Table 1 shows system properties comparison among key-value stores that we cover in this section. It has been compiled form [20, 44] and each of the candidates' website.

TABLE 1. System properties comparison among Memcached, Redis and Project Voldemort.

Name	**Memcached**	**Redis**	**Voldemort**
Description	key-value cache	a key-value data structure server	A key-value Database
Data storage	volatile memory	volatile memory and persistent	Database
Data type	string	data structures	JSON and Java objects
Query language	Memcached-protocol	RESTful API calls, memcached-protocol and lua	API calls
Initial Release	2003	2009	?
License	BSD License	BSD License	Apache License
Implementation language	C, Java	C	Java
Secondary index	no	no	no
Composite key	no	no	no
MapReduce	no	no	no
Replication mode	none	master-slave replication	symmetric replication
Sharding	yes	yes	yes
Consistency	yes	yes	yes
Atomicity	yes	no	yes
Full text search	no	no	no
Transaction concept	yes (ACID)	yes	conditional
Concurrency	yes	yes	yes
Durability	yes	yes	yes
Value size max	1 MB	512 MB	?

As opposed to classical classification that only considers data model for grouping NoSQL data stores, we incorporate other criteria to have more well-defined classes. We categorize a NoSQL solution as a key-value stores if it has all the following features:

- support only for simple data types (no notion of documents);
- search only based on keys (no composite keys);
- no support for full text search;
- no support for secondary index.

Due to mentioned characteristics, some data stores that used to be considered key-value stores, such as Riak or Oracle NoSQL, no longer fall into our key-value class; for example, the latest version of Riak is almost a document store with providing secondary indexes, composite keys, full text search and complex values [7].

3.1.1. *Memcached.* Memcached [15, 22] is a high-performance, distributed memory object caching system, generic in nature, but originally intended for use in speeding up dynamic web applications by alleviating database load. Memcached has been improved to include features analogous to the other key-value stores: persistence, replication, high availability, dynamic growth, backup, and so on. Memcached clients can store and retrieve items from servers using keys. These keys can be any character strings. Typically, keys are MD5 sums or hashes of the objects being stored/fetched. The identification of the destination server is done at the client side using a hash function on the key. Therefore, the architecture is inherently scalable as there is no central server to consult while trying to locate values from keys [22]. Basically, Memcached consists of the following components:

- client software, which is given a list of available memcached servers,
- a client-based hashing algorithm, which chooses a server based on the "key" input,
- server software, which stores the values with their keys into an internal hash table, and
- server algorithms, which determine when to throw out old data (if out of memory), or reuse memory.

3.1.2. *Redis.* REmote DIctionary Server (Redis) is an in-memory database where data are stored on the memory for faster performance. In Redis, complex objects such as lists and sets can be associated with a key. In Redis, data have time-to-live (TTL) values that can be set, after which keys are removed from memory. Redis uses locking for atomic updates and performs asynchronous replications.

Persistence in Redis is achieved in two ways: one is called snapshotting which is a semi-persistent durability mode where the dataset is asynchronously transferred from memory to disk from time to time, written in RDB dump format. Since version 1.1 the safer alternative is an append-only file (AOF) that is written as operations modifying the dataset in memory are processed. Redis is able to rewrite the AOF in the background in order to avoid an indefinite growth of the AOF file.

By default, Redis syncs data to the disk at least every 2 seconds, with more or less robust options available if needed. In the case of a complete system failure on default settings, only a few seconds of data would be lost. In applications that do not need durability of data, Redis performs very well compared to writing the data into the disk for any changes in the data. Since version 2.8, lexicographically range queries are possible, assuming elements in a sorted set are all inserted with the same identical score. As the primary storage of Redis is memory, Redis might not be the right option for data-intensive applications with dominant read operations because the maximum Redis data set cant be bigger than memory [34].

Redis implements the Publish/Subscribe messaging paradigm where senders (publishers) are not programmed to send their messages to specific receivers (subscribers). Rather, published messages are characterized into channels, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more channels, and only receive messages that are of interest, without knowledge of what (if any) publishers there are. This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology[34].

Redis supports consistent hashing, however only Redis Cluster, which has been available since April 2015, can fully leverage such partitioning. Consistent hashing implementation in Redis provides the ability to switch to other Redis instance if the preferred instance for a given key is not available. Similarly if we add a new instance, part of the new keys will start to be stored on the new instance[34].

3.1.3. *Project voldemort.* Voldemort is a distributed database, where tasks are only achieved through traditional CRUD requests such as `GET`, `PUT` and `DELETE`. Voldemort provides multi-version concurrency control (MVCC) for updates. It updates replicas asynchronously and, as a result, it does not guarantee consistent data. However, it can guarantee an up-to-date view, if you read a majority of replicas. Keys and values can be more complex objects such as JSON objects, maps and lists in addition to simple scalar values [25]. In terms of consistency, there is no guarantee that at given read time data is consistent among different node stores. However, it applies versioning using vector clocks and read-repair options. Voldemort is used by LinkedIn Inc. and it provides simple key-value store concentrating on industry-level performance and efficiently [10].

Voldemorts partitioning scheme relies on consistent hashing to distribute the load across multiple nodes. In consistent hashing, the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Instead of mapping a node to a single point in the ring, each node gets assigned to multiple points in the ring. To this end, Voldemort uses the concept of "virtual nodes". A virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. Project Voldemort allows namespaces for key-value pairs called "stores", in which keys are unique. Keys are associated with exactly one value and values are allowed to contain complex structures. Basic operations in Voldemort are atomic to exactly one key-value pair [16]. Serialization in Voldemort is pluggable so any format can be supported by implementing a Serializer class that handles the translation between bytes and objects.

3.1.4. *Summary and future trend.* Key-value stores support basic operations over relatively simple keys and values. They are only able to search based on key and do not support advanced searching capabilities such as compound keys, secondary indexes, or full text search. The main use case for key-value stores is providing a distributed cashing system with backup capabilities. They all provide scalability by distributing keys on different nodes using various hashing techniques. Voldemort uses MVCC and the others use locks for concurrency control.

Key-value stores tend to support complex data as their value and provide more searching capabilities. Riak and Oracle NoSQL are examples that have paved such a direction. They now support composite keys, secondary indexes, full-text search and handling complex values, similar to what document stores do. More specifically, three distinctive paths can be imagined for the near future of key-value stores: first, to become document stores that not only support simple key-values but also handle documents; second, adding graph data model to introduce themselves as multi-model NoSQL datastores (OrinetDB is an example in this regard); third to be more focused on in-memory capabilities to serve as a fully distributed pure key-value caches (e.g., Redis and Memcached).

3.2. **Document stores.** Inspired by Lotus Notes, document databases were, as their name implies, designed to manage and store documents. These documents are encoded in a standard data exchange format such as XML, JSON (Java Script Option Notation), YAML (YAML Ain't Markup Language), or BSON (Binary JSON). Unlike the simple key-value stores described above, the value column in document databases contains semi-structured data specifically attribute name/value pairs. A single column can house hundreds of such attributes, and the number and type of attributes recorded can vary from row to row (schema free). Also, unlike key-value stores, both keys and values are fully searchable in document databases. The also usually support complex keys and secondary indexes.

Also referred to as document-oriented database, a document store allows the inserting, retrieving, and manipulating of semi-structured data. The term "document store" may be confusing; while these systems could store documents in the traditional sense (articles, Microsoft Word files, etc.), a document in these systems can be any kind of "pointerless object". They store documents which allow values to be nested documents or lists as well as scalar values, and the attribute names are dynamically defined for each document at runtime. Unlike the key-value stores, these systems generally support secondary indexes and multiple types of documents (objects) per database. Most of the databases available under this category provide data access typically over HTTP protocol using RESTful API or over Apache Thrift protocol for cross-language interoperability. Like other NoSQL systems, the document stores do not provide ACID transactional properties [10, 16, 43].

Table 2 shows system properties comparison among document stores that we cover in this section. It has been generated form the materials in their official documentations and [20, 44].

3.2.1. *CouchDB.* Apache CouchDB is a flexible, fault-tolerant database, which supports data formats such as JSON and AtomPub. CouchDB stores documents in "collections" that form a richer data model compared to its counterparts. Collections comprise the only schema in CouchDB, and secondary indexes must be explicitly created on fields in collections. A document has field values that can be scalar (text, numeric, or boolean) or compound (a document or list). Queries are done with what CouchDB calls "views", which are defined with Javascript to specify field constraints. View model is a method of aggregating and reporting on the documents in a database, and are built on-demand to aggregate, join and report on database documents. The indexes are B-trees, so the results of queries can be ordered or value ranges. Queries can be distributed in parallel over multiple nodes using a map-reduce mechanism. However, CouchDBs view mechanism puts more burden on programmers than a declarative query language [4].

TABLE 2. System properties comparison among CouchDB, MangoDB, DynamoDB and RavenDB.

Name	**CouchDB**	**MongoDB**	**RavenDB**
Description	A document store inspired by Lotus Notes	One of the most popular document stores	.NET-based Document Store
Data storage	file system	volatile memory file system	Esent
Query language	JavaScript, REST, Erlang	API calls, Java Scripts, REST	API calls, Direct object access, HTTP, JSON, JavaScript, REST
Initial Release	2005	2009	2010
License	Apache V2	AGPL V3	AGPL V3
Database as service	no	no	no
Language	Erlang	C++	C#
Server OS	Android, BSD, Linux, OS X, Solaris, Windows	Linux, OS X, Solaris, Windows	Windows
Data type	JSON	JSON	JSON, BSON, BLOB
Secondary index	yes (via views)	yes	yes
APIs	RESTful HTTP/JSON API	proprietary protocol using JSON	.NET Client API, RESTful HTTP API
Partitioning method	sharding	sharding	sharding
Replication methods	master-master and master-slave replication	master-slave replication	master-master, master-slave, async and multi-source replication
MapReduce	yes	yes	yes
Integrity model	MVCC, BASE, ACID	BASE	ACID, BASE, eventually consistent
Consistency concept	eventual consistency	eventual and immediate consistency	eventual consistency
Transaction concept	no (atomic operations within a single doc)	no (atomic operations within a single doc)	yes
Concurrency	yes (optimistic locking)	yes	yes
Durability	yes	yes (optional)	yes
Value size max	4 GB	16 MB	2 GB

The map function gets a document as a parameter, performs some calculation, and produces data based on the views criteria. The data structure produced by the map function is a triplet consisting of the document id, a key and a value result. After the map function has been executed, the results are passed to an optional reduce function to be aggregated on the view. As all documents of the database are processed by a views functions this can be time consuming and resource intensive for large databases. Therefore a view is not created and indexed when write operations occur but on demand and updated incrementally when it is requested again [16, 4].

CouchDB provides asynchronous replication to achieve scalability and does not use sharding. The replication process operates incrementally where only modified data since the last replication gets transmitted to the next one. To provide durability, all updates on documents and indexes are flushed to disk on commit, by writing to the end of a file. Therefore, together with the MVCC mechanism, it is claimed that CouchDB provides ACID semantics at the document level. The single update operations are either executed to completion or fail/rollback so that the database never contains partly saved or updated documents. CouchDb does not guarantee consistency, since each client sees a self-consistent view of the database. All replicas are always writable and they do not replicate with each other by themselves[10, 43].

3.2.2. *MongoDB.* MongoDB is a database between relational databases and non-relational database. MongoDB is a GPL open source document store written in C++. Like CouchDB, it provides indexes on collections, it is lockless, and it provides a document query mechanism. However, there are key differences between the two: CouchDB provides MVCC on documents, while MongoDB provides atomic operations on fields; MongoDB supports dynamic queries with automatic use of indices, like RDBMSs; MongoDB supports automatic sharding, distributing load/data across "thousands of nodes" with automatic failover and load balancing, a feature inspired by Googles BigTable; while CouchDB achieves scalability through asynchronous replication, MongoDB achieves it through sharding (however an extension of CouchDB called CouchDB Lounge supports sharding). To shard a collection, MongoDB uses a shard key. A shard key is either an indexed field or an indexed compound field that exists in every document in the collection. MongoDB divides the shard key values into chunks and distributes the chunks evenly across the shards. To divide the shard key values into chunks, MongoDB uses either range based partitioning or hash based partitioning [27].

MongoDB stores data in a binary JSON-like format called BSON. BSON supports boolean, integer, float, date, string and binary types. Client drivers encode the local languages document data structure (usually a dictionary or associative array) into BSON and send it over a socket connection to the MongoDB server (in contrast to CouchDB, which sends JSON as text over an HTTP REST interface). MongoDB also supports a GridFS specification for large binary objects, e.g. images and videos. These are stored in chunks that can be streamed back to the client for efficient delivery.MongoDB supports master-slave replication with automatic failover and recovery.

A replica set is a group of *mongod*[1] instances that host the same data set. One mongod, the primary one, receives all write operations. All other mongod instances, secondary ones, apply operations from the primary so that they have the same data

[1]*mongod* is the primary daemon process for the MongoDB system. It handles data requests, manages data access, and performs background management operations.

set. The primary accepts all write operations from clients. A replica set can have only one primary. In other words, replication (and recovery) is done at the shard level. Replication is asynchronous for higher performance, so some updates may be lost on a crash [27].

The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data. There are two tools that allow applications to represent these relationships: references and embedded documents. References store the relationships between data by including links or references from one document to another. Broadly speaking, these are normalized data models (Figure 4).

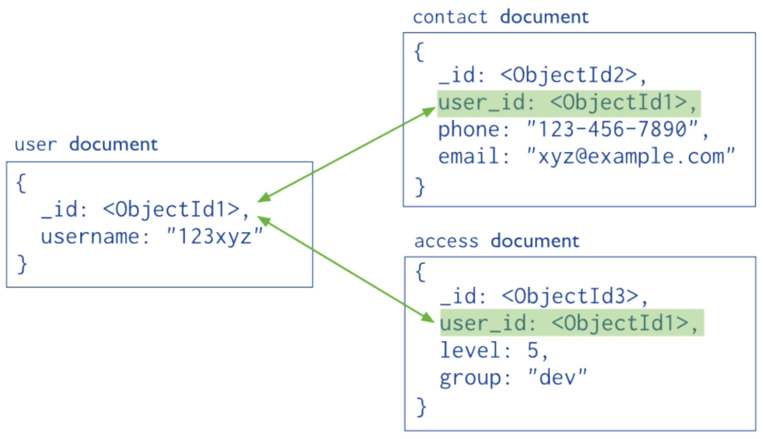

FIGURE 4. Normalized data model design in MongoDB [27].

Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures in a field or array within a document. These denormalized data models allow applications to retrieve and manipulate related data in a single database operation (Figure 5).

3.2.3. *RavenDB*. RavenDB is a transactional, open-source Document Database written in .NET. Data in RavenDB is stored in a schemaless manner as JSON documents, and can be queried efficiently using Linq queries from the .NET code or by REST requests using other tools. Internally, RavenDB makes use of indexes, which are automatically created based on the usage, or created explicitly by the consumer. RavenDB offers replication and sharding support. It is a database technology based on a client-server architecture; data is stored on a server instance and data requests from one or more clients are made to that instance.

In RavenDB every document has metadata attached to it. By default, this metadata only contains data that is used internally by RavenDB. With RavenDB, each document has its own unique global ID, in the sense that if one attempted to store two different entities under the same ID, the second write would overwrite the first one without any warning. The convention in RavenDB is to have a document ID that is a combination of the collection name and the entity's unique ID within the collection. This convention is enforced by default within RavenDB by pluralizing

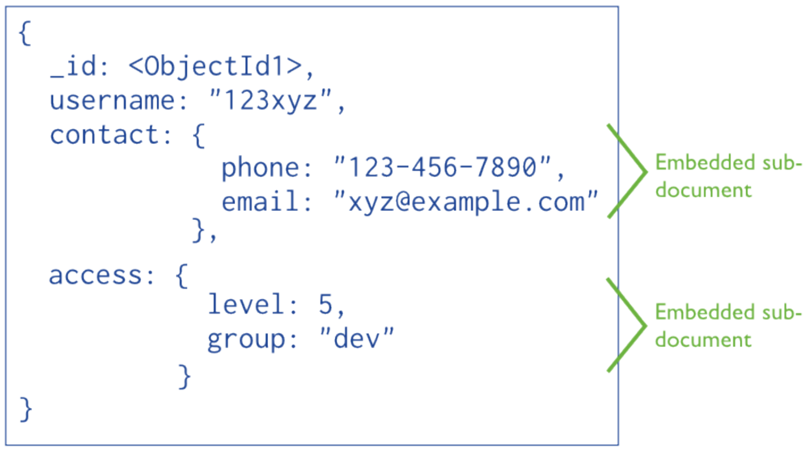
```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
            level: 5,
            group: "dev"
          }
}
```

Figure 5. Denormalized data model design in MongoDB [27].

the name of the object class being saved, and adding an auto-incremented number to it. However, this document ID convention is not mandatory: document IDs are independent of the entity type, and therefore don't have to contain the name of the collection they are assigned to.

One of the design principles that RavenDB adheres to is the idea that documents are independent, meaning all data required to process a document is stored within the document itself. However, this doesn't mean that RavenDB does not support related documents. RavenDB offers three approaches to solve the relation problem namely, "denormalization", "includes" and hybrid approaches. Each scenario will need to use one or more of them. When applied correctly, they can drastically improve performance, reduce network bandwidth and speedup development. "Denormalization" and "includes" are corresponding to denormalized and reference approaches in MongoDB. The hybrid approach is the combination of the two by having a short version of the related document inside the master document and providing a reference to the long version of related document. There are no strict rules as to when to use which approach, but the general idea is to give it a lot of thought, and consider the implications each approach has [18].

RavenDB supports replication and to enable it, a built-in replication bundle should be activated, when creating a new database. On every transaction commit, RavenDB will look up the list of replication destinations. For each of the destinations, the replication bundle will query the remote instance for the last document that was replicated to that instance. Next, it starts sending batches of updates that occurred since the last replication. Replication happens in the background and in parallel. RavenDB contains built-in support for sharding. It will handle all aspects of sharding for users, and they only need to define the shard function (how to actually split the documents among multiple shards) [18].

3.2.4. *Summary and future trend.* To enhance the performance of common queries and updates, document stores usually have full support for secondary indexes. These indexes allow applications to store a view of a portion of the collection in an efficient data structure. Most indexes store an ordered representation of all values of a field or a group of fields. Indexes may also enforce uniqueness, store objects in a geospatial representation, and facilitate text search. Full text search is another capability that most of document stores support and adopt various technique to make the searching efficient and low cost.

Document stores already implemented some notion of reference documents in their data model. Recently some document stores such as OrientDB [30] and ArangoDB [5] matured this feature to support graph data model as well so that introduce themselves as multi-model stores. This trend is not specific to document stores and can be seen in other classes to mitigate a problem known as "polyglot persistence" in which different types of NoSQL solutions are leveraged to satisfy all business requirements in the data layer [24, 9].

3.3. **Graph databases.** Graph databases are designed to fill the gap between graph data modeling requirements and the tabular abstraction in traditional databases. Graph databases employ Graph theory concepts like nodes and edges. Nodes are entities in the data domain and edges are the relationship between two entities. Nodes can have properties or attributes to describe them. Graph databases help us to implement graph processing requirements in the same query language level as we use for fetching graph data without the extra layer of abstraction for graph nodes and edges. This means less overhead for graph-related processing and more flexibility and performance. Performance and flexibility in such databases are becoming more important nowadays, especially with the emergence of new widely used applications such as social media or e-commerce software. Big companies can store and model users' interests and recommend more related advertisement to them. Additionally, graph databases make it easier to implement complex traversals in graph datasets. Figure 6 depicts a data example in a graph database.

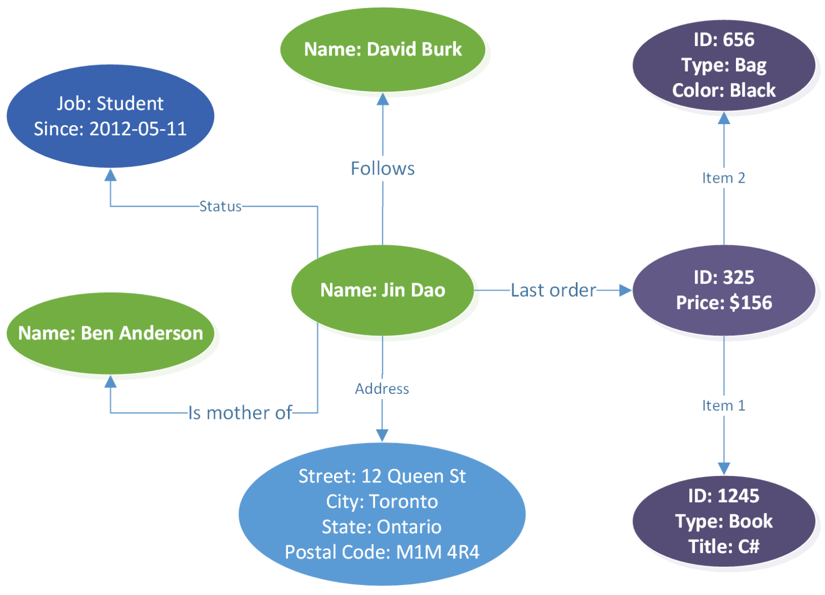

FIGURE 6. A Graph data model.

Graph databases are growing fast in industry. Examples of relevant use cases include social networks analysis, authentication and authorization systems, e-commerce recommender systems and fraud detection in financial services. Such demanding use cases can be implemented easier using graph databases. Based on online reports [2], the popularity of graph databases has increased six times since 2013. This report implies that by 2017, 25% of the enterprise solutions will use graph database solutions.

McColl et al. [26] have conducted a performance evaluation on multiple database systems but focused more on graph datasets with different characteristics or injected a graph-related experiment (i.e., PageRank) in each evaluation. Also, Jouili et al. [23] introduced their own bench-marking system (instead of using YSCB) and implemented popular workloads for graph datasets such as shortest path between nodes and neighborhood exploration experiment. In both of them, we observed a lack of implementation of enough general scenarios to let us compare implementations of graph data systems.

Table 3 shows the features comparison among graph databases that we cover in this section [21, 44].

3.3.1. *Neo4j.* Neo4j is a well-known project of a graph storage since 2007. It has various native APIs in most of programming languages such as Java, Go, PHP, and others. Neo4j has its own database file structure for storing files, which is flexible enough to be embedded inside Java applications. Neo4j is fully ACID compatible and schema-free.

Neo4j uses its own query language called Cypher [38] that is inspired by SQL, but it supports syntax related to graph nodes and edges. Cypher designed to be declarative language, which is more focused on the data to be retrieved rather than commands needed to be used for retrieval. This makes Cypher queries richer for graph and node model definitions.

Neo4j supports data replication in a master-slave fashion which ensures fault-tolerance against server failures. owever, it does not allow for data partitioning in multiple servers. This means the data size should be less than the capacity of the server. Nevertheless, Neo4J supports replication by which it replicates the whole data on another cluster node.

3.3.2. *Titan.* Titan is an open-source project with focus on serializing and compacting graph data structure, rich graph data modeling, and execution of graph queries in a more efficient way [6]. This database system implements modular interfaces for data persistence, data indexing, and client access. Titan can perform its storage over various NoSQL databases such as HBase, Cassandra and Oracle Berkeley database while Neo4j stores its data on its proprietary file structures. Titan's main feature against Neo4j is its scalability and real-time processing of data. Titan supports various secondary indexing systems from Lucence[2], Solr[3] and ElasticSearch[4].

[2]`https://lucene.apache.org/core`

[3]`http://lucene.apache.org/solr`

[4]`https://www.elastic.co`

TABLE 3. System properties comparison among Neo4j, Titan and, OrientDB.

Name	**Neo4j**	**OrientDB**	**Titan**
Description	open source graph database	multi-model DBMS (Document, Graph, Key/Value)	a Graph DBMS optimized for distributed clusters.
Data Storage	disk Persistence, volatile memory	disk Persistence, volatile memory, remote	disk Persistence, volatile memory, remote
Data model	graph DBMS	document store, Graph DBMS and, Key-value store	graph DBMS
Query Language	Cypher query language, Java API, RESTful HTTP API	Java API, RESTful HTTP/JSON API, Tinkerpop	Java API, TinkerPop
Initial Release	2007	2010	2012
License	AGPL V3 (Commercial) / BSD (Community)	Apache V2	Apache V2
Language	Java	Java	Java
Secondary index	yes (via Apache Lucene)	yes	yes
Composite key	no	yes	yes
MapReduce	no	no (but achievable with distributed queries)	yes (via Faunus)
Replication methods	master-slave replication (commercial)	master-slave replication	symmetric replication
Sharding	no	yes	yes
Data schema	schema free	schema free	yes
SQL	no	no	yes
Consistency	eventual Consistency, immediate consistency	unknown	eventual Consistency, immediate Consistency
Atomicity	yes	yes	yes
Transaction concept	ACID	ACID	ACID
Concurrency	yes	yes	yes
Value size max	1 TB	2 GB	2 GB

Titan presents its abstraction layer on top of commonly used NoSQL database systems. Thus, Titan can inherit back-end storage systems capabilities by integrating with them. One good example can be data partitioning which is not implemented in Neo4j but it is ready to use in Titan, simply by using Cassandra or HBase. This project provides abstractions to work easily with a large volume of edges and vertices in big datasets.

Titan can be used in two ways: embedded or remote. In the first case, Titan is embedded inside the Java application as a library and can negotiate with the storage back-end form application. In the second case, Titan can be accessible with Tinkerpop stack API [3].

3.3.3. *OrientDB.* OrientDB is another open-source solution for storing various models of data structure. One of the supported models in this database system is Graph. This database system supports relationships and that is the key feature to enter to the domain of Graph database systems. This feature is implemented by document pointers. Document pointers are values inside JSON documents that point to other documents. In this way, pointers can operate like edges in graph data structures. Instead of providing JOIN functionality, OrientDB uses document pointers to connect documents with a linear cost for big data [30].

There are two special features in OrientDB: “dataset profiling”, which is implemented using roles for database users, and an “indexing algorithm”, which is inspired from B+ tree and Red-Black tree.

Like Neo4j, OrientDB supports RESTful access to data. In addition to REST, OrientDB supports SQL in the same manner that SQL is used in a relational database, while Neo4j uses the Cypher query language. OrientDB supports TinkerPop Blueprints API [3] to access graph data. This feature is also implemented in Titan. OrientDB also can be embedded inside Java application like Neo4j.

OrientDB uses horizontal partitioning for bigger databases stored on multiple servers (i.e., sharding). Each chunk of data, i.e. shard, contains smaller subsets of data, i.e. some rows. Besides sharding, OrientDB supports multi-master feature. Other solutions, like HBase, have one master server and some secondary nodes, which may create throughput bottlenecks on the master side. In multi-master configurations, the client can connect to any of the database cluster servers and synchronization is handled by OrientDB. Thus, the total throughput of the cluster is equal to the throughput of all database cluster servers instead of the single master server. Multi-master feature is unique in OrientDB among all graph solutions.

3.3.4. *Summary and future trend.* In summary, graph databases are trending upwards and are becoming more popular in the industry. One of the reasons for their popularity is that they can model some industrial problems in a more convenient way. We investigated three graph storage and processing solutions. Titan only provides abstraction layer for graph storage on top other key-value solutions like Cassandra. OrientDB and Neo4j have their backend storage. While OrientDB supports SQL, Neo4j has an exclusive query declaration language called Cypher. OrientDB was the only solution that supports the concept of a multi-master topology for maximizing storage throughput. All of the solutions support durability and ACID features and secondary indexes. Titan does not support any data partitioning feature, but Giraph and OreintDB support sharding.

One future trend for graph datastores is the provision of more graph-related functionality in their query languages. Such functionalities can be added in their query language like Cypher in Neo4j or by adding more functions to SQL language. Also another concern in graph databases to be addressed in future is their elasticity tolerance. Graph databases support more complex and resource-intensive queries than the other types of databases, so their scaling can cause more issues in queries.

3.4. **Wide column stores.** Although column-oriented data stores are defined as being non-relational, it can be argued that they are the equivalent of Big Data for relational databases, since they exhibit similarities on a conceptual level. In this sense, they have retained the notions of *tables*, *rows* and *columns*. This creates the notion of a *schema*, explicit from the client's perspective, but rather flexible from the data store's point of view, which enables the execution of queries on the data. Nevertheless, the underlying philosophy, architecture and implementation are quite different from traditional RDBMS. The basic difference is that the notion of tables is primarily for interacting with clients, while the storage, indexing and distribution of data is taken care by a file and a management system.

Almost all of the popular wide-column data store systems have inherited the data model proposed by Google for the BigTable system [11]. In BigTable, a table is defined as a "sparse, distributed, persistent multidimensional sorted map". A cell, a unit of data, is uniquely identified by its row key, its column key and its timestamp. The row keys are lexicographically sorted and consecutive keys are grouped into tablets, which dictates how data is stored in the distributed underlying file system, thus implementing and exploiting the locality property of the data to improve performance of reads and writes. Given this, the design of the row key is essential to enable efficient reads of short ranges of row keys (i.e., scan) without having to "wake up" a large number of machines. This way the row key defines the load balancing of the data. Transactional consistency in BigTable is guaranteed on a row level; every read and write on a single row is serializable and timestamped, thus allowing concurrent updates to the same row.

The data in each row is organized in column families. Each family is of the same data type, to allow for data compression, and may have an arbitrary number of columns. The number of column families is usually small to minimize the amount of shared metadata. Columns define the properties of access control and resource accounting. Columns may be added or deleted from column families without affecting the data. However, the removal of a column family will change the schema of the data. Since the model does not guarantee transactions across multiple rows, if a column family is to be deleted and there is corresponding data in multiple rows, then this data cannot be deleted automatically. Figure 7 represents a sample table in a wide-column datastore.

Under the hood, most wide-column datastores are based on a distributed file system. More specifically, BigTable is built on top of the GFS (Google File system). The data is persistently (and immutably) stored in files called *SSTables*. Updates are handled in an intermediate, cache-like structure called the *MEMTABLE*. Periodically the updates are pushed to the SSTables. When a read request is received, both the MEMTABLE and the sequence of SSTables are checked.

The infrastructure of a BigTable cluster contains a Chubby [8] server acting as master and Tablet servers acting as data nodes. Chubby is responsible among others for access control, maintain tablets and assign data to tablets. Tablet servers maintain a collection of tablets and a tablet maintains a sequence of SSTables. All servers (master and data) also contain metadata to address locality issues and update the topology (with the addition or removal of servers) and logs about the

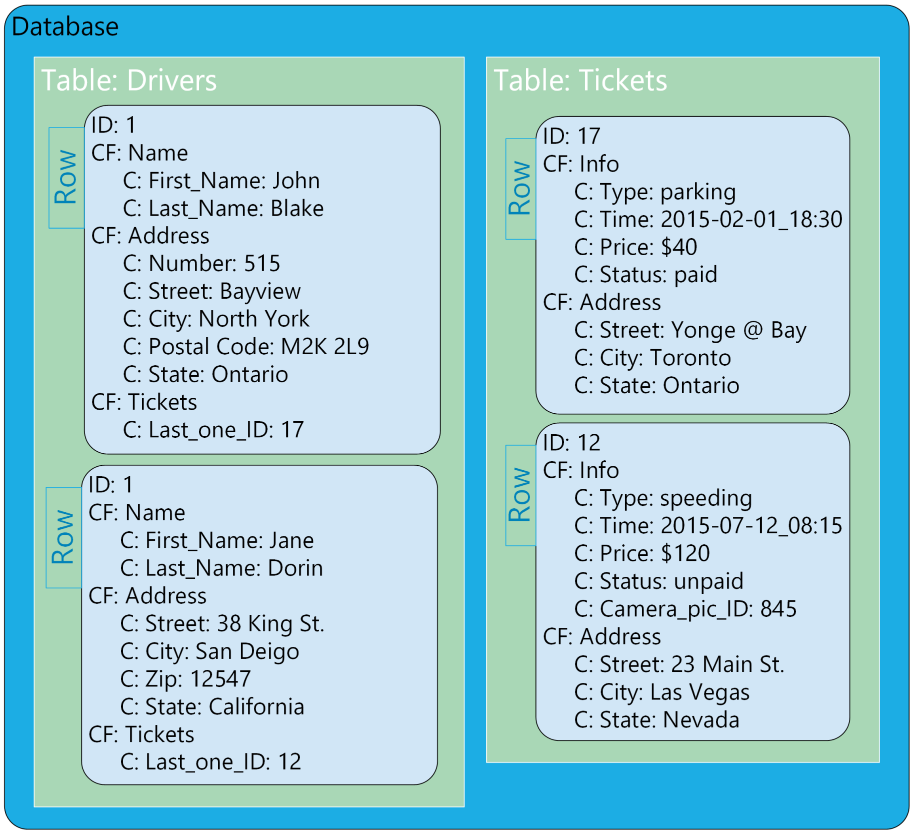

FIGURE 7. A Wide-Column data model for drivers/ticket information.

operations on the data. The user tables, the ones used by clients to access the data, correspond to a schema representation of the underlying tablets.

Wide-column stores are designed for data that spans billions of rows and millions of columns. Additionally, they support cells of arbitrarily large size. They are more efficient than traditional relational databases for applications that scan a few columns of many rows, because this needs to load significantly less data than reading the whole row. Moreover, rows can be of large size and their keys of high complexity, which makes storing to and retrieving data from a single machine possible. Finally, wide-column stores allow different access level for different columns.

In this paper, we review four independent implementations of BigTable, namely Apache HBase [41], Apache Cassandra [40], Apache Accumulo [39] and Hypertable [19], ranked as the most popular ones by DB-engines [20]. Table 4 compares the presented implementations based on some of their basic properties. It has been generated with information from DB-engines [20] and vsChart [44].

TABLE 4. System properties comparison among HBase, Cassandra, Accumulo and Hypertable.

Name	**HBase**	**Cassandra**	**Accumulo**	**Hypertable**
Description	based on Apache Hadoop and on concepts of BigTable	based on ideas of BigTable and DynamoDB	featuring cell level security and server-side computation	An open source BigTable implementation based on Hadoop
Data model	Column-oriented Key-value	Column-oriented Key-value	Column-oriented Key-value Schema-less	Column-oriented
Initial Release	2008	2008	2011	2009
License	Apache V2	Apache V2	Apache V2	BSD
Language	Java	Java	Scala	C++
Server OS	Linux, Unix, Windows (through an emulator)	BSD, Linux, OS X, Windows	Unix	Linux, OS X, Windows (unofficially)
Data schema	schema free	schema free	schema free	schema free
SQL	no	no	no	no
Secondary index	no	yes (for some queries)	yes	yes (for some queries)
APIs and other access methods	Java, REST, Thrift, XML	CQL (proprietary), Thrift	Java, REST, Thrift	C++, REST, Thrift
Partitioning method	sharding	sharding	sharding	sharding
Replication methods	Master-slave	Peer to Peer, Asynchronous, Multi-source	Multi-master	Master-slave
MapReduce	yes	yes	yes	yes
Integrity model	Log replication	BASE	MVCC	MVCC
Consistency concept	immediate consistency	event/imm consistency	immediate consistency	immediate consistency
Transaction concept	no (atomic for single row)	no (atomic for single row)	no (atomic for single row)	no (atomic for single row)
Concurrency	yes	yes	yes	yes
Durability	yes	yes	yes	yes
Value size	2 TB	2 GB	1 EB	unknown

3.4.1. *Apache HBase.* Apache HBase is the NoSQL wide-column store for Hadoop, the open-source implementation of MapReduce for Big Data analytics. The purpose of HBase is to support random, real-time read and write access to very large tables with billions of rows and millions of columns. Just like BigTable, HBase runs on top of HDFS (Hadoop Distributed File System) and is deployed on commodity hardware serving as the Master server, responsible for cataloging and load balancing, and RegionServers, acting as the data nodes. HBase provides the same level of transactional support as BigTable; consistency is guaranteed on the row level. Finally, just like BigTable is tailored to support MapReduce, HBase is built in a way to conveniently support Hadoop jobs.

3.4.2. *Apache cassandra.* Cassandra is designed to be a fault-tolerant, highly reliable and available data store. The premise is that failures may happen both in software and hardware and they are practically inevitable. For this reason, Cassandra's architecture allows for fast recovery to ensure high data availability. To achieve this, two entities are central; data centers and clusters. A data center is a collection of homogeneous nodes that hold a number of redundant replicas of the data for high availability. Nodes in data centers are always local, i.e. do not span multiple physical locations. Clusters, on the other hand, are collections of data centers and can span across multiple locations. The nodes in a cluster are connected in a peer-to-peer network and communication information among each other every second using a gossip protocol to mediate failures. Cassandra also employs the concepts of commit log and SSTables, in a similar manner as BigTable, to ensure ACID actions.

3.4.3. *Apache accumulo.* Accumulo shares the notion of the Master server, TabletServers and Tablets with BigTable and is also built on top of HDFS like HBase. Accumulo offers a number of improved features over the native BigTable implementation. It offers server-side programming capabilities to define functions that can happen at any stage of the data management process, including when reading or writing in the disk. Furthermore, unlike BigTable, Accumulo allows for access control on the level of individual cells and it shreds the requirement that a single row should fit in memory as it allows for very large cell values, practically of unlimited size. Additionally, using Zookeeper, Accumulo allows for the definition of multiple Master servers, so that it is tolerant against Master failures.

3.4.4. *Hypertable.* Hypertable inherits from BigTable the notions of Master and Range servers (equivalent to the RegionServers of HBase and data nodes of BigTable) and also the Hyperspace, which plays the role of the BigTable's Chubby server. However, unlike the other three aforementioned implementations, Hypertable possesses a unique feature; instead of being tied to a specific underlying file system, Hypertable abstracts its interface by directing all the requests for actions on the file system level through a broker process. By developing and contributing different brokers, Hypertable can be connected to any file system. Another unique feature is the ability of the RangeServers to allocate their memory depending on whether the workload is write- or read-heavy.

3.4.5. *Summary and future trend.* Wide column stores are tightly coupled with analytical frameworks, like MapReduce and Hadoop. Their design aims for high availability and high velocity in retrieving data to enable fast analytics. In addition, the flexible design of the data schema with the column families of variable length (i.e. different records may have different number of columns per column family) enables targeted retrieval of data for more efficient queries and analysis.

In their latest versions, the presented stores have worked towards implementing or improving the following key aspects of their features:

1. fast reads and high availability,
2. faster writes, and
3. improved security.

All three Apache-supported solutions (HBase[5], Accumulo[6] and Cassandra) have implemented methods to improve data retrieval. More specifically, HBase is offering increased availability by replicating regions across different RegionServers[7]. In practice, a RegionServer holds the data of a region for reading and writing along with *read-only* replicas of other regions. This way if a RegionServer is busy, down or unusually slow, a client request may be served by another server holding a replica. The reads in this paradigm are *timeline-consistent*, implying that requests for newly added data are served by the primary RegionServers and replicas are updated periodically. Similarly, Accumulo offers data center replication with eventual consistency[8]. Cassandra offers *rapid read protection*[9] through data replication across nodes and the concept of *eager retries.* In Cassandra, the nodes exchange states within the cluster through the gossip protocol, so the cluster knows if a node is down. When a node is being slow, the system would resend the user request to replica nodes, before the query times out.

With respect to faster writes, HBase has implemented a multithreaded method for writing multiple WALs (write-ahead logs) on a given RegionServer, to increase write throughput [10]. In addition, HBase has adopted Accumulo's cell level access control to increase security[11]. Finally, HBase[12] and Cassandra[13] have proposed novel, hybrid compaction methods to reduce compaction time and improve read efficiency.

4. **How to select the right one.** Out of the 3Vs of Big Data (i.e., Volume, Velocity and Variety), velocity and variety are the most requested ones for NoSQL solutions rather than volume. It is known that 90% of web based companies will likely never rise to the volume levels implied by NOSQL (please see Figure 8). When coming to the selection of a NoSQL solution, the good thing is that there are several types of Big Data databases to pick from. The bad news, however, is that there is no one type that does everything the best.

Stefan Edlich [13] compiled six groups of aspects/questions for those who are about to leverage or migrate to a NoSQL solution. However, we describe the main elements in choosing the proper NoSQL datastore in three classes as follows:

- **Data Model and access pattern.** First of all, the data model is required to be identified. This cuts the target domain down to one of the classes that we described earlier in this paper. If the data is strongly relational and has some

[5]`https://blogs.apache.org/hbase/entry/start_of_a_new_era`
[6]`http://hortonworks.com/blog/apache-accumulo-1-7-0-released/`
[7]`https://issues.apache.org/jira/browse/HBASE-10070`
[8]`https://issues.apache.org/jira/browse/ACCUMULO-378`
[9]`http://digbigdata.com/apache-cassandra-2-features/`
[10]`https://issues.apache.org/jira/browse/HBASE-8755`
[11]`https://issues.apache.org/jira/browse/HBASE-6222`
[12]`https://issues.apache.org/jira/browse/HBASE-7667`
[13]`http://www.datastax.com/wp-content/uploads/2013/09/WP-DataStax-WhatsNewC2.0.pdf`

characteristics of big data then NewSQL solutions[14] with horizontal scalability are the right solutions to choose which is out of the scope of this paper. Column oriented, document-like, graph, object, multi-values, JSON and BLOBS are among data types that can be handled with one of the four groups of NoSQL solutions that we investigated in this paper. Then some data-related features such as volume, complexity, schema flexibility, durability and data access pattern are taken into account to further narrow the candidates. The access pattern including distribution of read/write and random vs sequential access needs a careful consideration. Figure 8 shows the comparison between NoSQL solutions in terms of data complexity and size.

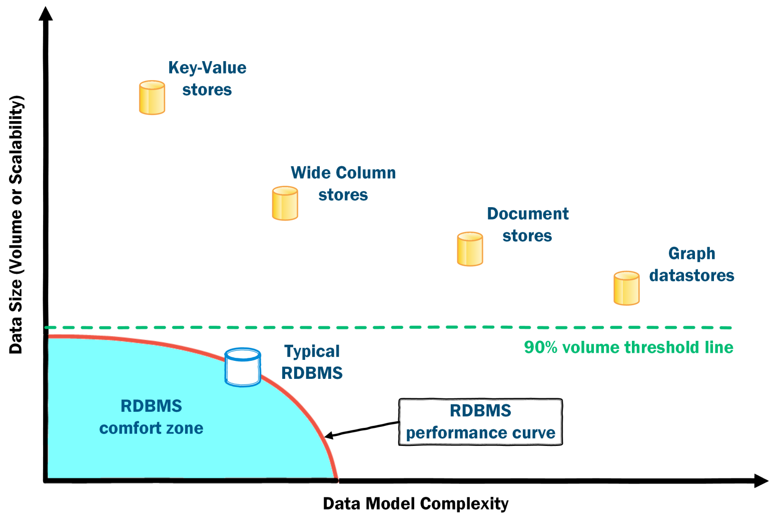

FIGURE 8. NoSQL solutions; complexity vs size.

- **Query Requirements.** Next, the required query capabilities should be identified; is SQL preferred or is LINQ[15] a better fit? Is MapReduce appropriate or do we need higher level query or script languages? What is the significance of Ad-Hoc queries, secondary indexes, full text search, aggregations, views or column oriented functions in our business? For instance, get, put and delete functions are best supported by key-value systems. Aggregation becomes much easier while using column-oriented systems rather than the conventional row oriented databases. The former use tables but do not have joins. Mapping data from object-oriented software becomes easy using a Document oriented NoSQL database such as XML or JSON as they use structured document formats.
- **Non-functional properties.** Large number of concerns and features fall into this category, chief among them is performance. Usually performance translates into latency (read, write, modify or insert) and throughput. Performance is a non-functional property that depends on other non-functional

[14]MySQL, Postgres, Percona provides distributed solutions for relational data.
[15]LINQ (Language Integrated Query) is a uniform query syntax in C# and VB.NET used to save and retrieve data from different sources. It is integrated in C# and VB.

properties including, partitioning, replication (synchronous vs asynchronous), horizontal scalability, load balancing, auto-scalability, consistency technique (e.g., ACID, BASE or adjustable consistency), CAP trade-off (i.e., the ability to tune CA, CP and AP) and concurrency mechanisms (e.g., locks, MVCC, ACID or None). The way all these properties are designed and implemented in a solution has direct impact on the overall performance. Other important non-functional factors include elasticity, DB simplicity (i.e., installation, configuration, upgrade, maintenance and development), security (i.e., authentication, authorization, validity), license model, vendor reliability, community support and total cost of ownership (e.g., license cost, scaling cost, sysadmin cost, operational cost, safety, backup and restore costs, disaster management and monitoring cost). Implementation language helps to determine how fast a database will process. Typically NoSQL databases written in low level languages such as C/C++ and Erlang will be the fastest. On the other hand, those written in higher level languages such as Java make customization easier. We have compiled such non-functional capabilities for the selected solutions in this paper in Tables 1, 2, 3 and 4.

After all this systematic comparison and contrast, a real and independent performance evaluation is required to make sure the final candidate(s) fits our needs. Database vendors usually measure productivity of their solutions with custom hardware and software settings designed to demonstrate the advantages of their solutions. Experiments done by others might also not fulfill one's particular requirements; everyone has their own specific workload and access patterns. Therefore, any potential user is advised to conduct an independent and unbiased experiment to determine whether the final candidate is the real fit. We suggest users to leverage YCSB [12] that is the de facto standard for performance evaluation of NoSQL solutions. YCSB has the built-in implementation of a large number of solutions and can generate various types of workloads. In section 5, we demonstrate how to implement new interfaces for new or unimplemented solutions and also show how to modify the existing ones to work with new version of a solution. Moreover, we show how to customize workloads to imitate the real workload in the user business.

5. **Performance evaluation.**

Benchmarking is widely used for evaluating computer systems, and benchmarks exist for a variety of levels of abstraction, from the CPU, to the database software, to complete enterprise systems. A few efforts in the area of big data benchmarks emerged, such as YCSB [12], PigMix [45], CALDA [31] and GraySort [17]. While some are focused on one or a subset of components and tasks typical for big data systems, others are based on specific map-reduce-style systems. Rabl et al. [33] presented "BigBench" an end-to-end big data benchmark that includes a data model, synthetic data generator and workload description. Among all, we choose YCSB to deal with performance evaluation of selected NoSQL datastores. YCSB has a plugin-based architecture and can be extended or customized easily.

As we mentioned in section 4, the approach in this paper is not to compare a subset of solutions under a pre-defined set of conditions; rather we describe how to leverage, customize and extend YCSB to do the performance analysis in your environment and under your specific workload. Figure 9 shows the high level architecture of YCSB.

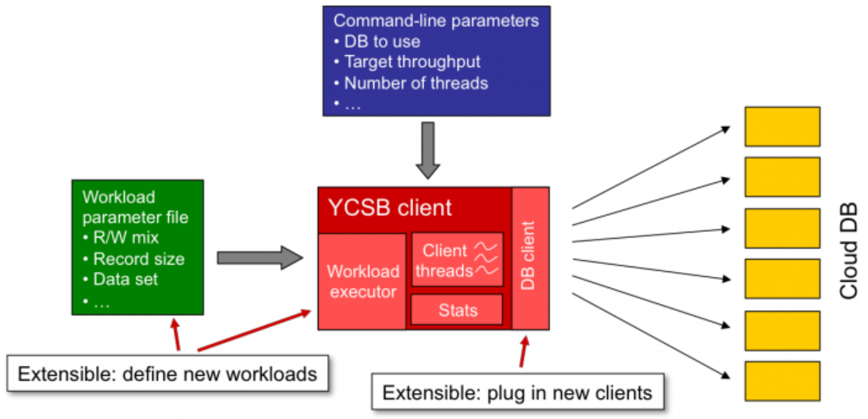

FIGURE 9. YCSB Client Architecture [12].

Our own experience with selecting a NoSQL solution was in the context of our Connected Vehicles and Smart Transportation[16] project [42]. For this project, we needed a NoSQL solution to act as a warehouse and host various types of traffic data. We were also in need of very efficient write operations under heavy load. Using a systematic approach, similar to what presented in this paper, HBase was selected as the first candidate. Then, we needed to examine the performance of HBase in our environment. So, we set up a cluster and tuned it according to our needs on SAVI cloud [35]. Next, we tried YCSB to examine the datastore, but the driver was not working properly. We modified the HBase driver to work with the specific version that we were interested, and then we introduced a heavy write workload[17] based on the YCSB core workload to imitate our data ingestion and access pattern. Thanks to acceptable performance by HBase for our project, we selected it as the final solution for the warehouse.

This experiment motivated us to do the research presented in this work and make it available for other users. Our edition of YCSB (i.e., ASRL-YCSB) contains all our development, fully commented, and is publicly available[18].

We have examined other NoSQL datastores discussed in this paper including Redis, Voldemort, Memcached, MongoDB, CouchDB, RavenDB, OrientDB, Titan, Cassandra and Accumulo to verify the YCSB compatibility against the latest version of these datastores. These experiments have been done with minimum configuration and under the default deployment that makes the results disqualified for any comparison. Due to the popularity of Neo4J and the lack of its support in YCSB, we developed a new driver for Neo4J. ASRL-YCSB repository[19] contains the tested drivers for latest versions[20] of the above-mentioned datastores.

5.1. **Benchmark configuration.** We leverage SAVI cloud [35], which is an OpenStack-based academic and experimental testbed in Canada, for conducting the performance evaluation. Table 5 shows the specification of machines that we used for the experiment.

[16]`http://cvst.ca`
[17]We refer to this workload as workloadg.
[18]ASRL-YCSB can be obtained from here: `https://github.com/ceraslabs/ASRL-YCSB`.
[19]`https://github.com/ceraslabs/ASRL-YCSB`
[20]The versions are the latest as of July 2015.

Table 5. Virtual Machines (VM) specifications.

Name	**Extra Large (Xlarge)**	**Large**	**Medium**	**Small**
vCPU	8	4	2	1
Disk (GB)	160	80	40	20
RAM (GB)	16	8	4	2

Table 6 shows the specifications of our environment.

Table 6. Configuration parameters in performance evaluation.

Description	**YCSB Client**	**HBase**	**Neo4J**
VM flavor	xlarge	xlarge	All flavors
No. of VMs	1	4	1
Version	Yahoo YCSB 2010	1.0.0	2.2.3
OS	Ubuntu 14.04 64B	Ubuntu 14.04 64B	Ubuntu 14.04 64B

5.2. **Results.** In this section, we present the results that we obtained under our conditions for HBase. We also present the result of Neo4J to test the new developed driver under a highly stressed environment. Table 7 describes the workloads that we used for the performance evaluation. We added workload G (i.e., "workloadg") to examine HBase cluster under extensive insert and seldom reads. We used synthetic data provided by YCSB. The data size (i.e., each record) is 1 KB comprising of 10 fields, each of which 100 bytes and 24 bytes for the key. For each "read" operation, one record will be retrieved using a random key. The "insert" command is hashed and not ordered. The way the YCSB client does the "scan" is that it will pick a start key, and then request a number of records; this works fine even for hashed insertion. By default, the scan length size for each call is a uniformly random number between 1 and 100 records. Requests (i.e. operations) are distributed based on Zipfian distributions. By default, the YCSB client inserts 1000 records into datastore during the "load" phase and then will do 1000 operations against the datastore during the "run" phase. All these values can be adjusted according to the target environment. We set load and run size to one million records and operations respectively. The full specifications of our experiment can be found in "script" directory as shell scripts[21].

Figure 10 and 11 show the read and update latency (or response time) for workloads A, B and F under different target throughput.

Figure 12 shows the comparison of read-modify-write delay on small, medium, large and extra large machines. As it can be seen, after upgrading to the next larger machine, the delay decreases almost linearly with the factor of two, which is also the factor with which the individual resources of a VM (CPU, memory and disk) are multiplied by. For other operations, such as read, update and insert the improvement was almost equivalent.

Figures 13, 14, 15 and 16 show the results for HBase cluster. We ran the experiment using one YCSB client with 40 active threads and the client buffering was

[21] `https://github.com/ceraslabs/ASRL-YCSB/tree/master/scripts`

TABLE 7. Workload specifications.

	Workloads				
Operations	**A**	**B**	**E**	**F**	**G**
Read	0.5	0.95	0	0.5	0.05
Update	0.5	0.05	0	0	0
Scan	0	0	0.95	0	0
Insert	0	0	0.05	0	0.95
Read-Modify-Write	0	0	0	0.5	0

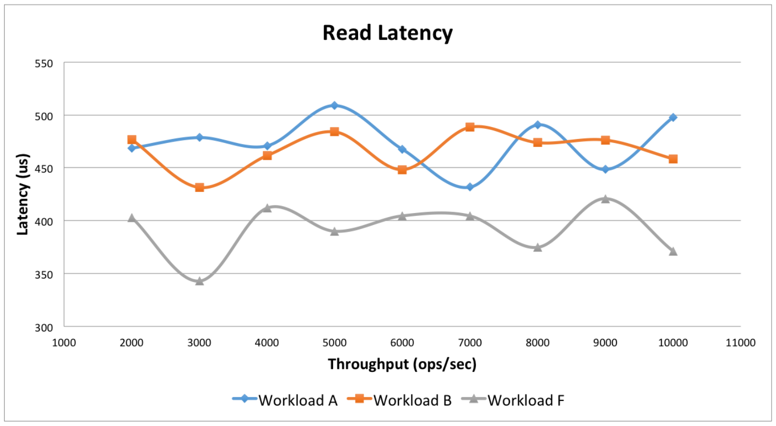

FIGURE 10. Read latency vs target throughput for Neo4J.

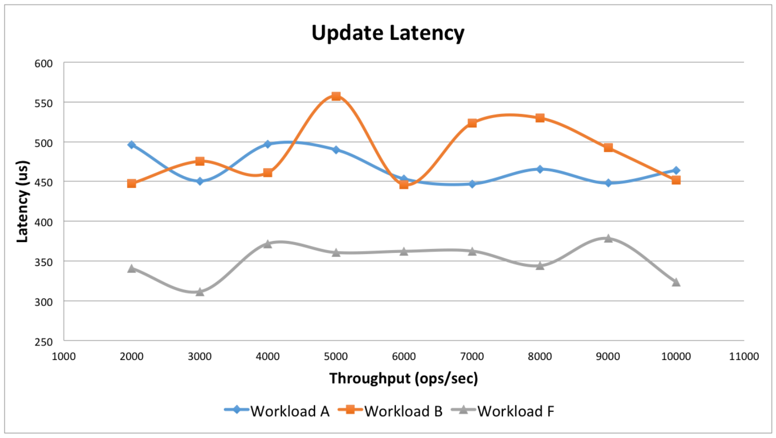

FIGURE 11. Update latency vs target throughput for Neo4J.

disabled. Figure 16 shows the actual throughput versus target throughput and as can be seen, the maximum throughput was obtained for workload A.

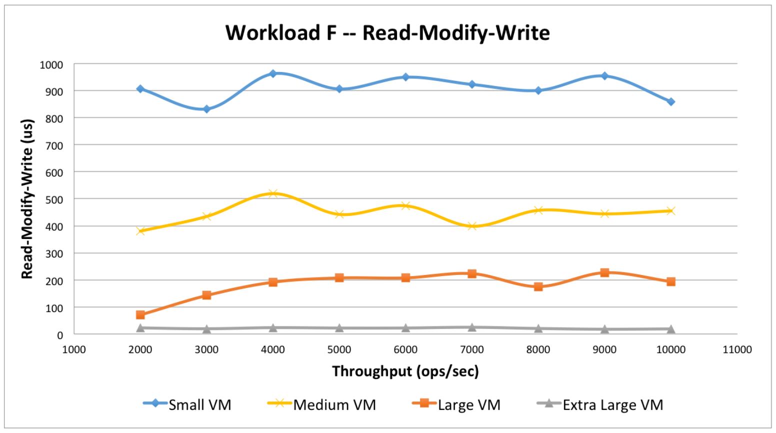

FIGURE 12. Read-Modify-Write delays for four configurations; running Neo4J on small, medium, large and extra large machine.

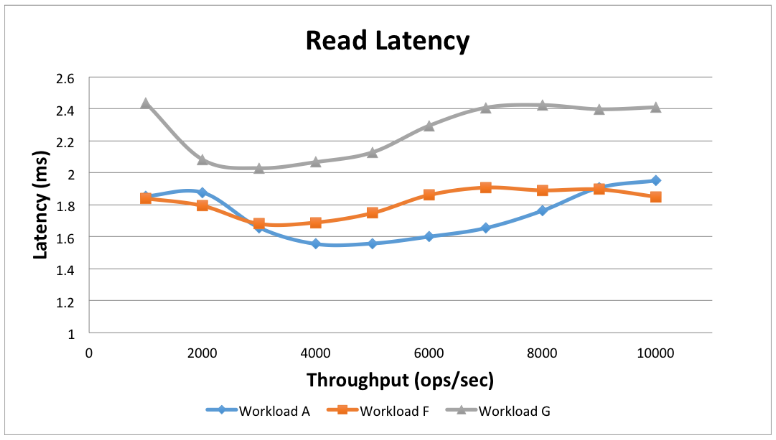

FIGURE 13. Read delay for HBase.

6. **Conclusion.** Big data has been forcing businesses to leverage new types of datastores that are more performant, economical, reliable and scalable compared to traditional RDBMS solutions. Selecting the right datastore is not a trivial task due to diversity and lack of standard benchmarks in this domain. There exist research works and experiments to compare and contrast various solutions but none of them are truly generalizable and applicable for other interested parties. Most of those works have been done for pre-selected solutions under controlled conditions.

In this paper however, we provided a comprehensive comparison for major classes in NoSQL world in a systematic manner. For each class, we elaborated on the main functionalities and characteristics and then introduced 3 to 4 dominant solutions in that class. We introduced new criteria to redefine traditional classifications of NoSQLs in a more distinctive manner. Then, as the final step, we demonstrated the methodology for customizing and configuring YCSB to be repeatable for a different target environment; new drivers and one new workload have been implemented

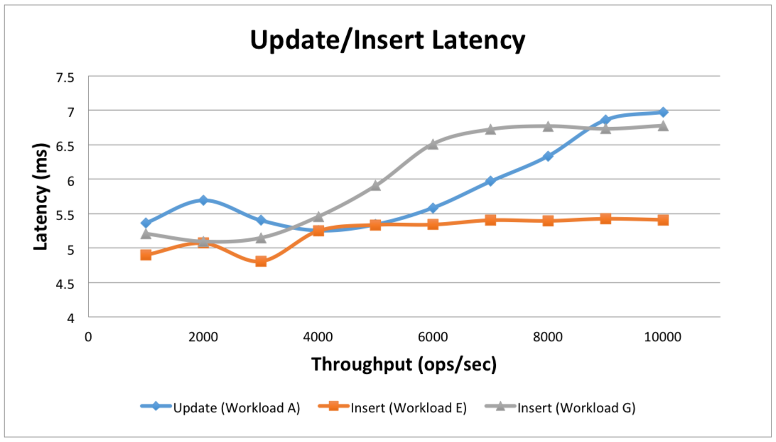

FIGURE 14. Update and Insert delays for HBase.

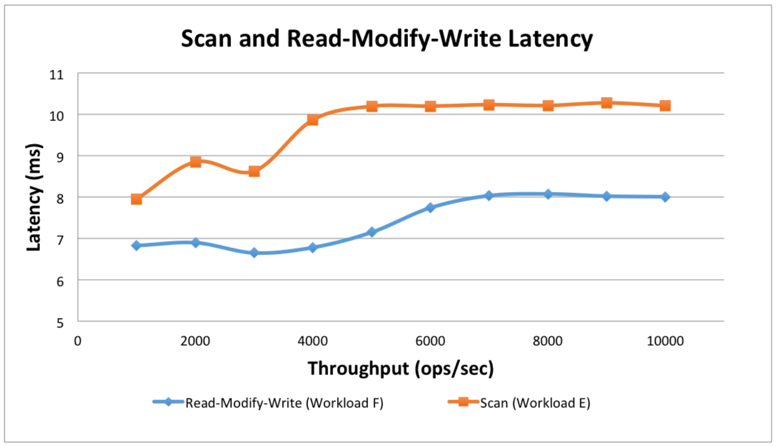

FIGURE 15. Scan and Read-Modify-Write delays for HBase.

to materialize the tailoring process. This will help businesses to choose the right solution(s) for their specific data and environment with high confidence.

Acknowledgments. This research was supported by Fuseforward Solutions Group Ltd., the Natural Sciences and Engineering Council of Canada (NSERC), and the Ontario Research Fund for Research Excellence under the Connected Vehicles and Smart Transportation (CVST) project.

REFERENCES

[1] Y. Abubakar, T. S. Adeyi and I. G. Auta, Performance evaluation of nosql systems using ycsb in a resource austere environment, *Performance Evaluation*, **7** (2014), 23–27.
[2] P. Andlinger, 2015, URL`http://db-engines.com/en/blog_post/43`.
[3] Apache Software Foundation, Apache tinkerpop, 2015, URL`http://tinkerpop.incubator.apache.org`.
[4] Apache Software Foundation, Technical overview of apache couchdb, 2015, URL`http://wiki.apache.org/couchdb/TechnicalOverview`.

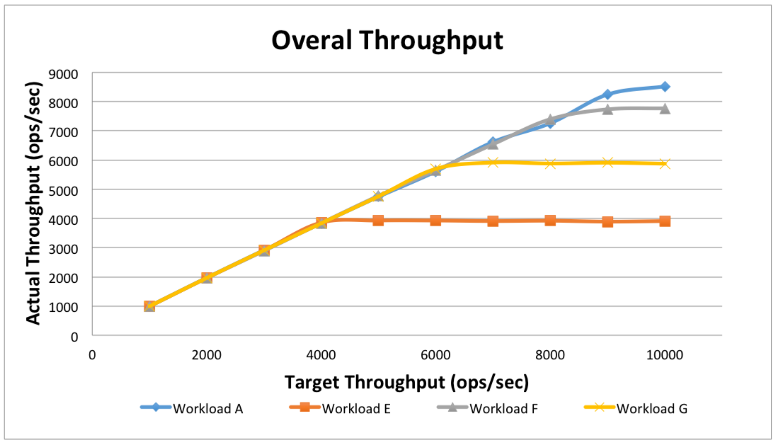

FIGURE 16. Target throughput versus actual throughput for HBase.

[5] ArangoDB GmbH, Arangodb documentation, 2015, URL`https://www.arangodb.com/documentation`.

[6] Aurelius LLC, Titan architecture overview, 2015, URL`http://s3.thinkaurelius.com/docs/titan/0.9.0-M2/arch-overview.html`.

[7] Basho Technologies, Inc, Riak docs, 2015, URL`http://docs.basho.com/riak/latest/intro-v20`.

[8] M. Burrows, The chubby lock service for loosely-coupled distributed systems, in *Proceedings of the 7th symposium on Operating systems design and implementation*, USENIX Association, 2006, 335–350.

[9] R. Casado and M. Younas, Emerging trends and technologies in big data processing, *Concurrency and Computation: Practice and Experience*, **27** (2015), 2078–2091.

[10] R. Cattell, Scalable sql and nosql data stores, *ACM SIGMOD Record*, **39** (2010), 12–27.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Gruber, Bigtable: A distributed storage system for structured data, *ACM Transactions on Computer Systems (TOCS)*, **26** (2008), p4.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan and R. Sears, Benchmarking cloud serving systems with ycsb, in *Proceedings of the 1st ACM symposium on Cloud computing*, ACM, 2010, 143–154.

[13] S. Edlich, A. Friedland, J. Hampe, B. Brauer, M. Brückner, S. Edlich, A. Friedland, J. Hampe, B. Brauer and M. Brückner, Nosql.

[14] A. Feinberg, Project voldemort: Reliable distributed storage, in *Proceedings of the 10th IEEE International Conference on Data Engineering*, 2011.

[15] B. Fitzpatrick, Distributed caching with memcached, *Linux journal*, **2004** (2004), p5.

[16] S. K. Gajendran, A survey on nosql databases, *University of Illinois*.

[17] J. Gray, Graysort benchmark, 2015, URL`http://sortbenchmark.org`.

[18] Hibernating Rhinos., Ravendb - the open source nosql database for .NET, 2015, URL`http://ravendb.net/docs/article-page/3.0/csharp/start/getting-started`.

[19] Hypertable Inc, Hypertable, 2014, URL`http://hypertable.org/`.

[20] S. IT, Knowledge base of relational and nosql database management systems, 2015, URL`http://db-engines.com`.

[21] S. IT, System properties comparison neo4j vs. orientdb vs. titan, 2015, URL`http://db-engines.com/en/system/Neo4j`.

[22] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur et al., Memcached design on high performance rdma capable interconnects, in *Parallel Processing (ICPP), 2011 International Conference on*, IEEE, 2011, 743–752.

[23] S. Jouili and V. Vansteenberghe, An empirical comparison of graph databases, in *Social Computing (SocialCom), 2013 International Conference on*, 2013, 708–715.

[24] J. Klein, I. Gorton, N. Ernst, P. Donohoe, K. Pham and C. Matser, Performance evaluation of nosql databases: A case study, in *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*, PABS '15, ACM, New York, NY, USA, 2015, 5–10.

[25] LinkedIn, Project voldemort, 2015, URL`http://www.project-voldemort.com`.

[26] R. C. McColl, D. Ediger, J. Poovey, D. Campbell and D. A. Bader, A performance evaluation of open source graph databases, in *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, ACM, New York, NY, USA, 2014, 11–18.

[27] MongoDB Inc., Mongodb 3.0 manual, 2015, URL`http://docs.mongodb.org/manual`.

[28] A. Moniruzzaman and S. A. Hossain, Nosql database: New era of databases for big data analytics-classification, characteristics and comparison, arXiv preprint arXiv:1307.0191.

[29] M. A. Olson, K. Bostic and M. I. Seltzer, Berkeley db., in *USENIX Annual Technical Conference, FREENIX Track*, 1999, 183–191.

[30] Orient Technologies, Top 10 key advantages for going with orientdb, 2015, URL`http://orientdb.com/why-orientdb/`.

[31] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden and M. Stonebraker, A comparison of approaches to large-scale data analysis, in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, ACM, 2009, 165–178.

[32] D. Pritchett, Base: An acid alternative, *Queue*, **6** (2008), 48–55.

[33] T. Rabl, A. Ghazal, M. Hu, A. Crolotte, F. Raab, M. Poess and H.-A. Jacobsen, Bigbench specification v0. 1, in *Specifying Big Data Benchmarks*, Springer, 2014, 164–201.

[34] RedisLabs, Redis, 2015, URL`http://redis.io/documentation`.

[35] SAVI, Smart Applications on Virtual Infrastructure, Cloud platform, 2015, URL`http://www.savinetwork.ca`.

[36] S. Sivasubramanian, Amazon dynamodb: A seamlessly scalable non-relational database service, in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ACM, 2012, 729–730.

[37] C. Strozzi, Nosql–a relational database management system, 2015, URL`http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/HomePage`.

[38] Technology, Cypher query language, 2015, URL`http://neo4j.com/docs/stable/cypher-query-lang.html`.

[39] The Apache Foundation, Apache accumulo, 2015, URL`http://accumulo.apache.org/`.

[40] The Apache Foundation, Welcome to apache cassandra, 2015, URL`http://cassandra.apache.org/`.

[41] The Apache Foundation, Welcome to apache hbase, 2015, URL`http://hbase.apache.org/`.

[42] A. Tizghadam and A. Leon-Garcia, Connected Vehicles and Smart Transportation - CVST Platform, 2015, URL`http://cvst.ca/wp/wp-content/uploads/2015/06/CVST.pdf`.

[43] G. Vaish, *Getting started with NoSQL*, Packt Publishing Ltd, 2013.

[44] vsChart.com, The comparison wiki: Database list, 2015, URL`http://vschart.com/list/database/`.

[45] P. Wiki, Pig mix benchmark, 2015, URL`https://cwiki.apache.org/confluence/display/PIG/PigMix`.

Received October 2015; revised December 2015.

E-mail address: `hkh@yorku.ca`
E-mail address: `fokaefs@yorku.ca`
E-mail address: `zareian@yorku.ca`
E-mail address: `nbm@yorku.ca`
E-mail address: `brian.ramprasad@gmail.com`
E-mail address: `mark@cse.yorku.ca`
E-mail address: `gaikwadpurwa@gmail.com`
E-mail address: `mlitoiu@yorku.ca`