2014 IEEE 10th World Congress on Services

NoSQL Systems for Big Data Management

Venkat N Gudivada
Weisburg Division of Computer Science
Marshall University
Huntington, WV, USA
gudivada@marshall.edu

Dhana Rao
Biological Sciences Department
Marshall University
Huntington, WV, USA
raod@marshall.edu

Vijay V. Raghavan
Center for Advanced Computer Studies
University of Louisiana at Lafayette
Lafayette, LA, USA
vijay@cacs.louisiana.edu

***Abstract*—The advent of Big Data created a need for out-of-the-box horizontal scalability for data management systems. This ushered in an array of choices for Big Data management under the umbrella term NoSQL. In this paper, we provide a taxonomy and unified perspective on NoSQL systems. Using this perspective, we compare and contrast various NoSQL systems using multiple facets including system architecture, data model, query language, client API, scalability, and availability. We group current NoSQL systems into seven broad categories: Key-Value, Table-type/Column, Document, Graph, Native XML, Native Object, and Hybrid databases. We also describe application scenarios for each category to help the reader in choosing an appropriate NoSQL system for a given application. We conclude the paper by indicating future research directions.**

***Keywords*-Data Models, NoSQL, NewSQL, Big Data, Graph Databases, Document Databases, Native XML Databases**

I. Introduction

Until recently, relational database management systems (RDBMS) were the mainstay for managing all types of data irrespective of their naturally fit to the relational data model. The emergence of Big Data and mobile computing necessitated new database functionality to support applications such as real-time logfile analysis, cross-selling in eCommerce, location based services, and micro-blogging. Many of these applications exhibit a preponderance of insert and retrieve operations on a very large scale. Relational databases were found to be inadequate in handling the scale and varied structure of data.

The above requirements ushered in an array of choices for Big Data management under the umbrella term NoSQL [1], [2], [3]. NoSQL (meaning 'not only SQL') has come to describe a large class of databases which do not have properties of traditional relational databases and are generally not queried with SQL (structured query language). NoSQL systems provide data partitioning and replication as built-in features. They typically run on cluster computers made from commodity hardware and provide horizontal scalability.

Developing applications using NoSQL systems is quite different from the process used with RDBMS [4]. NoSQL databases require developer-centric approach from the application inception to completion. For example, data modeling is done by the application architect or developer, whereas in RDBMS based applications, data architects and data modelers complete the data modeling tasks. They begin by constructing conceptual and logical data models. Database transactions and queries come into play only in the design of a physical database. In contrast, NoSQL database approaches begin by identifying application queries and structure the data model to efficiently support these queries. In other words, there is a strong coupling between the data model and application queries. Any changes to the queries will necessitate changes to the data model. This approach is in stark contrast to the time-tested RDBMS principles of logical and physical data independence.

Traditionally, data is viewed as a strategic and shared corporate resource with well established policies and procedures for data governance and quality control. In contrast, NoSQL systems promote *data silos*, each silo geared towards meeting the performance and scalability requirements of just one or more applications. This runs against the ethos of enterprise data integration, redundancy control, and integrity constraints [5].

Given the above backdrop, following are the contributions of this paper. We provide a taxonomy and unified perspective to help the reader understand the functionality, strengths and limitations of NoSQL databases. Using this perspective, we compare and contrast various NoSQL systems using multiple facets which include system architecture, data model, query language, client API, scalability, and availability. We group the current NoSQL systems into seven broad categories: Key-Value, Table-type/Columnar, Document, Graph, Native XML, Native Object, and Hybrid databases. Members of these categories are not always disjoint, with some systems falling into more than one category. We also describe application scenarios for each category to help the reader choose an appropriate NoSQL system for a given application. We conclude the paper by indicating future research direction.

We begin with a discussion of the principal features

978-1-4799-5069-0/14 $31.00 © 2014 IEEE
DOI 10.1109/SERVICES.2014.42

of NoSQL systems (section II). In section III, we describe concepts that are essential to understanding NoSQL systems from both a technical standpoint and business perspective. Sections IV through X describe the classes of NoSQL systems along with their application scenarios. Section XI concludes the paper.

II. Principal Features of NoSQL

The need for NoSQL database solutions primarily emanated from the requirements of eCommerce, mobile computing and location based services, Web services, and social media applications. eCommerce applications require predictive analytics, personalization, dynamic pricing, superior customer service, fraud and anomaly detection, real-time order status through supply chain visibility, and Web server access log analysis. All of the above functions are characterized by massive data which need to be fused from multiple sources and processed in real-time.

A. Big Data Characteristics

Data that is **too big and complex** to capture, store, process, analyze, and interpret using the state-of-the-art tools and methods is referred to as *Big Data*. It is characterized by five Vs: Volume, Velocity, Variety, Value, and Veracity. Data size (*Volume*) is in the order of terabytes or even petabytes, and is rapidly heading towards exabytes. *Velocity* refers to the speed at which the data is generated. Enterprise Big Data is typically heterogeneous (*Variety*) and is comprised of structured, semi-structured, and unstructured data. *Value* refers to extracting useful and actionable information from massive data sets. Due to diversity in sources and evolution of data through intermediate processing, issues of security, privacy, trust, and accountability need to be addressed using secure data provenance (*Veracity*).

B. Flexible Data Models

All RDBMS databases are based on the relational data model. In contrast, each class of NoSQL system employs a different data model. NoSQL data models inherently support horizontal data partitioning across processors in a cluster or even processors across data centers.

RDBMS applications typically work with fixed database schema, though the schema may evolve slowly. Schema change is a well managed activity and the application goes through rigorous software testing. In contrast, many NoSQL systems treat schema as a construct that inherently evolves over time. These systems also require schema that allows great variation in row data without incurring the NULL value problems.

A NoSQL data model closely reflects an individual application need rather than needs of several applications. The data model naturally mirrors the way a subject matter expert (SME) thinks about the data in the domain in the context of a specific application. In other words, a NoSQL application works with a data model that is specifically designed and optimized for that application.

C. Eventual Consistency

Create/Insert, Read, Update, and Delete (CRUD) are four operations that databases are expected to provide and execute efficiently. NoSQL data models are structured to support massively concurrent insert and read operations relative to updates. The notion of *eventual consistency* is embraced to support extremely fast insert and read operations. Eventual consistency implies that applications are aware of the *non-repeatable read* issue due to latency in consistency. Some NoSQL systems employ data models to efficiently support only insert and read to the exclusion of update and delete.

D. Partial Record Updates

RDBMS data models are optimized for row-wise processing based on the assumption that applications process most of an entity's attributes. In contrast, some NoSQL data models are designed to efficiently perform column-wise processing to support computing aggregates on one or two attributes across all entities. For example, data models of column-oriented NoSQL databases resemble a data warehouse star schema where the fact table at the center stores de-normalized row data and each dimension table stores all columns of a column-family.

E. Application Synthesized Relationships

RDBMS data models are designed to capture one-to-one, one-to-many, and many-to-many relationships between entities. In contrast, many NoSQL data models do not explicitly and inherently model relationships (graph data model based NoSQL systems are an exception). Most NoSQL systems do not support *relational join operation*. NoSQL applications are expected to synthesize relationships by processing the data. If the data is inherently rich in relationships, many NoSQL data models are a poor fit.

F. Optimized MapReduce Processing

In RDBMS applications, employing MapReduce processing requires moving the data from the RDBMS to another memory address space within the same machine or to a different one. In contrast, some NoSQL systems feature MapReduce as a native functionality to obviate the data movement.

G. Native Support for Versioning

Mainstream, off-the-shelf RDBMS do not provide support for data versioning. However, an RDBMS data model can be designed to support versioning of an attribute by including two additional attributes (begin date and end date). This is a convoluted approach with strong performance penalties. In contrast, some NoSQL data models are designed to provide native support for versioning.

H. New Query Languages

RDBMS SQL queries in practice tend to be complex, involve joins across multiple tables, consume substantial database resources to execute, and are expensive to write, debug, and maintain. In contrast, many NoSQL systems avoid SQL altogether. Some provide a SQL-like query language such as XQuery, while others require writing programs for querying the database.

NoSQL databases take a multi-faceted approach to querying including allowing only simple queries through interfaces such as REST API, caching data in main memory, or using database stored procedures as queries. For example, in document-oriented NoSQL systems, a database query is a JSON document (technically, a JavaScript expression) specified across all the documents in the collection. For graph model based NoSQL systems, queries involve traversing graph structures. Interfaces for graph traversal include JavaScript-based interface (JIG), and special query languages such as SPARQL, RDFS++, and Cypher.

I. Horizontal Scalability

RDBMS for Big Data applications are forced to use distributed storage. Reasons for this include data that exceeds the disk size limits, need for partitioned table storage, recovery from hard disk failures through data redundancy, and data replication to enable high availability. A typical RDBMS query requires join operations across multiple tables. How fast a query is processed is limited by how quickly data can be moved from hard disks to primary storage, and the amount of data moved. Database update operations are usually run as database transactions. If the table data is partitioned, update operations run slowly due to contention for exclusive locks.

Securing and releasing write locks, coordinating this across various disks, and ACID (atomicity, consistency, isolation, and durability) compliance slows down transaction throughput. Long running transactions exacerbate the transaction throughput further. *Vertical scaling* is often proposed as a solution to the transaction throughput problem, which involves adding more CPUs, more cores per CPU, and additional main memory. However, vertical scaling quickly reaches its saturation point. Other techniques for improving transaction throughput include relaxing ACID compliance and using methods such as disabling database logging.

Horizontal scalability, in contrast, refers to scaling by adding new processors complete with their own disks (aka nodes). Closely associated with horizontal scaling is the partitioning of data. Each node/processor contains only a subset of the data. The process of assigning data to each node is referred to as *sharding*, which is done automatically in some NoSQL systems.

It is easier to achieve horizontal scalability with NoSQL databases. The complexity involved in enforcing ACID compliance simply does not exist for most NoSQL systems as they are ACID non-compliant by design. Furthermore, only insert and read operations dominate in NoSQL databases as update and delete operations fade into insignificance in terms of volume. NoSQL systems also achieve horizontal scalability by delegating two-phase commit required for transaction implementation to applications. For example, MongoDB does not guarantee ACID compliance for concurrent operations. This may infrequently result in undesirable outcomes such as *phantom reads*.

III. NoSQL Concepts

NoSQL databases draw upon several concepts and techniques to realize flexible data modeling and associated query languages, and horizontal scalability. These concepts include but are not limited to shared-nothing architecture, Hash trees, consistent hashing, REST API, Protocol buffers (Protobuf), Apache Thrift, JSON and BSON, BASE, MapReduce, vector clocks, column family, keyspace, memory-mapped files, and the CAP theorem.

A. Shared-Nothing Architecture

In this architecture, each node is self-sufficient and acts independently to remove single point of resource contention. Nodes share neither memory nor disk storage. Database systems based on shared-nothing architecture can scale almost linearly by adding new nodes assembled using inexpensive commodity hardware components. Data is distributed across the nodes in a non-overlapping manner, which is called *sharding*. Though this concept existed for long with its roots in distributed databases, it gained prominence with the advent of NoSQL systems.

B. Hash Trees and Consistent Hashing

A *hash tree* (aka *Merkle tree*) is a tree in which every non-leaf node is labeled with the hash of the labels of its child nodes. Hash trees enable efficient and secure verification of data transmitted between computers for

veracity. They are also used to efficiently determine the differences between a document and its copy.

In traditional hashing, a change in the number of slots in the hash table results in nearly all keys remapped to new slots. If K is the number of keys and n is the number of slots, *consistent hashing* guarantees that on average no more than K/n keys are remapped to new slots.

C. REST API, Protobuf and Apache Thrift

Representational State Transfer (REST) is an HTTP API. It uses four HTTP methods – GET, POST, PUT and DELETE – to execute different operations. Many NoSQL systems enable client interaction through this API.

Protocol buffers (Protobuf) is a method for efficiently serializing and transmitting structured data. It is used to perform remote procedure calls (RPC) as well as for storing data. Thrift is a software framework and an interface definition language (IDL) for cross-language services and API development.

D. JSON and BSON

JavaScript Object Notation (JSON) is a lightweight, text-based, open standard format for exchanging data between a server and a Web application. Though it is originally derived from the JavaScript language, it is a language-neutral data format. *BSON* is a format for binary-coded serialization of JSON-like documents. Compared to Protobuf, BSON is more flexible with schema but not as space efficient.

E. BASE Properties

In RDBMS, the *consistency* property ensures that all transactions transform a database from one valid state to another. Once a transaction updates a database item, all database clients (i.e., programs and users) will see the same value for the updated item.

ACID properties are to RDBMS as BASE is to NoSQL systems. BASE refers to basic availability, soft state, and eventual consistency. Basic availability implies disconnected client operation and delayed synchronization, and tolerance to temporary inconsistency and its implications. Soft state refers to state change without input, which is required for eventual consistency. *Eventual consistency* means that if no further updates are made to an updated database item for *long enough* period of time, all clients will see the same value for the updated item.

F. Minimizing Data Movement with MapReduce

MapReduce is a computational paradigm for processing massive datasets in parallel if the computation fits a pattern characterized by three steps: map, shard and reduce. The *map process* involves several parallel map processes concurrently processing different parts of data and each process produces (key, value) pairs. The *shard process* (second step) acts as a barrier to ensure that all mapper processes have completed their work, collects the generated key-value pairs from each mapper process, sorts them, and partitions the sorted key-value pairs. Next, the shard process assigns each partition to a different *reduce* process (third step). Each reduce process essentially receives *all* related key-value pairs and produces one result.

G. Vector Clocks

A *vector clock* is an algorithm to reason about events based on event timestamps. It is an extension of multi-version concurrency control (MVCC) used in RDBMS to multiple servers. Each server keeps its copy of vector clock. When servers send and receive messages among themselves, vector clocks are incremented and attached with messages. A partial order relationship is defined based on server vector clocks, and is used to derive causal relationships between database item updates.

H. Column Families and Keyspaces

A *column family* is a collection of rows, and each row can contain different number of columns. *Row keys* are unique within a column family, but can be reused in other column families. This enables storing unrelated data about the same key in different column families. Some NoSQL systems use the term column-family to refer to a group of related columns within a row.

A keyspace is a data container. It is similar to the schema concept in RDBMS. Keyspaces are used to group column families together. Data replication is specified at the keyspace level. Therefore, data with different replication requirements reside in separate keyspaces.

I. Memory-Mapped Files

A *memory-mapped file* is a segment of virtual memory which is associated with an operating system file or file-like resource (e.g., a device, shared memory) on a byte-for-byte correspondence. Memory-mapped files increase I/O performance especially for large files.

J. CAP Theorem

Consistency, availability, and partition tolerance (CAP) are the three primary concerns that determine which data management system is suitable for a given application. Consistency feature guarantees that all clients of a data management system have the same view of data. Availability assures that all clients can always read and write. Finally, partition tolerance ensures that the system works well with data that is distributed across physical network partitions. The CAP theorem states that it is impossible for any data management

system to achieve all these three features at the same time. For example, to achieve partition tolerance, a system may need to give up consistency or availability.

IV. Key-Value Databases

The defining characteristics of key-value databases include real-time processing of Big Data, horizontal scalability across nodes in a cluster or data centers, reliability and high availability. Their use cases include applications where response is needed in milliseconds. They are used for session management for Web applications; configuration management; distributed locks; messaging; personalization of user experience and providing engaging user interactions in social media, mobile platforms, and Internet gaming; real-time bidding and online trading; Ad servers; recommendation engines; and multi-channel retailing and eCommerce.

As the name suggests, these systems store data as key-value pairs or maps (aka dictionaries). Though all of them store data as maps, they differ widely in functionality and performance. Some systems store data ordered on the key, while others do not. Some keep entire data in memory and while others persist data to disk. Data can also be distributed across a cluster of nodes.

Representative systems in this category include Memcached, Aerospike, Redis, Riak, Kyoto Cabinet, Membase, Amazon DynamoDB, CouchDB, BerkeleyDB, EHCache, Apache Cassandra [6], and Voldermot. Table I summarizes characteristics of some key-value databases.

A hallmark of key-value database is rapid applications development and extremely fast response times even with commodity type processors. For example, 100K - 200K simple write/read operations have been achieved with an Intel Core 2 Duo, 2.6 GHz processor. In another case, replacement of a RDBMS with Redis for user profile management (100K+ profiles with over 10 filter attributes) was reduced from 3 - 6 sec to 50 ms.

V. Table-type/Column Databases

RDBMS are *row-based systems* as their processing is row-centric. They are designed to efficiently return entire rows of data. Rows are uniquely identified by system generated *row ids*. As the name implies, column databases are column-centric. Conceptually, a columnar database is like an RDBMS with an index on every column without incurring the overhead associated with the latter. It is also useful to think of column databases as nested key-value systems.

Column database applications are characterized by tolerance to temporary inconsistency, need for versioning, flexible database schema, sparse data, partial record access, and high speed insert and read operations. When a value changes, it is stored as a different version of the same value using a timestamp. In other words, the notion of update is effectively nonexistent. Partial record access contributes to dramatic performance improvements for certain applications. Columnar databases perform aggregate operations such as computing maxima, minima, average, and sum on large datasets with extreme efficiency.

Recall that a column family is a set of related columns. Column databases require predefining column families, and not columns. A column family may contain any number of columns of any type of data, as long as the latter can be persisted as byte arrays. Columns in a family are logically related to each other, and are physically stored together. Performance gain is achieved by grouping columns with similar access characteristics into the same family. Database schema evolution is achieved by adding columns to column families. A column family is similar to the column concept in RDBMS.

Systems in this category include Google BigTable (available through Google App Engine), Apache Cassandra, Apache HBase, Hypertable, Cloudata, Oracle RDBMS Columnar Expression, Microsoft SQL Server 2012 Enterprise Edition. Table II summarizes characteristics of some columnar databases.

VI. Graph Databases

A graph data model is at the heart of graph databases [7]. In some applications, relationships between objects is even more important than the objects themselves. Relationships can be static or dynamic. Such data is called *connected data*. Twitter, Facebook, Google, and LinkedIn data are naturally modeled using graphs. In addition, graph data models are used in other industries including airlines, freight companies, healthcare, retail, gaming, oil and gas. Graph databases are also popular for implementing access control and authorization subsystems for applications that serve millions of end users.

Graph databases include FlockDB, InfiniteGraph, Titan, Microsoft Trinity, HyperGraphDB, AllegroGraph, Affinity, OrientDB, DEX, Facebook Open Graph, Google Knowledge Graph, and Neo4J. Table III summarizes characteristics of two graph databases.

VII. Document Databases

These databases are not the document/full-text databases in the traditional sense. They are also not content management systems. Document databases are used to manage semi-structured data mostly in the form of key-value pairs packaged as JSON documents.

Table I
KEY-VALUE DATABASES

Name	Salient Characteristics
Memcached	Shared-nothing architecture, in-memory object caching systems with no disk persistence. Automatic sharding but no replication. Client libraries for popular programming languages including Java, .Net, PHP, Python, and Ruby.
Aerospike	Shared-nothing architecture, in-memory database with disk persistence. Automatic data partitioning and synchronous replication. Data structures support for string, integer, BLOB, map, and list. ACID with relax option, backup and recovery, high availability. Cross data center replication. Client access through Java, Lua, and REST.
Cassandra	Shared-nothing, master-master architecture, in-memory database with disk persistence. Key range based automatics data partitioning. Synchronous and asynchronous replication across multiple data centers. High availability. Client interfaces include Cassandra Query Language (CQL), Thrift, and MapReduce. Largest known Cassandra cluster has over 300 TB of data in over 400-node cluster.
Redis	Shared-nothing architecture, in-memory database with disk persistence, ACID transactions. Supports several data structures including sets, sorted sets, hashes, strings, and blocking queues. Backup and recovery. High availability. Client interface through C and Lua.
Riak	Shared-nothing architecture, in-memory database with disk persistence, data teated as BLOBs, automatic data partitioning, eventually consistency, backup and recovery, and high availability through multi data center replication. Client API includes Erlang, JavaScript, MapReduce queries, full text search, and REST.
Voldemort	Shared-nothing architecture, in-memory database with disk persistence, automatic data partitioning and replication, versioning, map and list data structures, ACID with relax option, backup and recovery, high availability. Protocol Buffers, Thrift, Avro and Java serialization options. Client access through Java API.

Table II
COLUMN DATABASES

Name	Salient Characteristics
BigTable	A sparse, persistent, distributed, multi-dimensional sorted map. Features strict consistency and runs on distributed commodity clusters. Ideal for data that is in the order of billions of rows with millions of columns.
HBase	Open Source Java implementation of BigTable. No data types, and everything is a byte array. Client access tools: shell (i.e., command line), Java API, Thrift, REST, and Avo (binary protocol). Row keys are typically 64-bytes long. Rows are byte-ordered by their row keys. Uses distributed deployment model. Works with Hadoop Distributed File System (HDFS), but uses filesystem API to avoid strong coupling. HBase can also be used with CloudStore.
Cassandra	Provides eventual consistency. Client interfaces: phpcasa (a PHP wrapper), Pycassa (Python binding), command line/shell, Thrift, and Cassandra Query Language (CQL). Popular for developing financial services applications.

Table III
GRAPH DATABASES

Name	Salient Characteristics
Neo4J	In-memory or in-memory with persistence. Full support for transactions. Nodes/vertices in the graph are described using properties and the relationships between nodes are *typed* and relationships can have their own properties. Deployed on compute clusters in a single data center or across multiple geographically distributed data centers. Highly scalable and existing applications have 32 billion nodes, 32 billion relationships, and 64 billion properties. Client interfaces: REST, Cypher (SQL-like), Java, and Gremlin.
AllegroGraph	Full read concurrency, near full write concurrency, and dynamic and automatic indexing of committed data, soundex support, fine granular security, geospatial and temporal reasoning, and social network analysis. Online backups, point-in-time recovery, replication, and *warm standby*. Integration with Solr and MongoDB. Client interfaces: JavaScript, CLIF++, and REST (Java Sesame, Java Jena, Python, C#, Clojure, Perl, Ruby, Scala, and Lisp).

Each document is an independent entity with potentially varied and nested attributes. Documents are indexed by their primary identifiers as well as semi-structured document field values. Document databases are ideal for applications that involve aggregates across document collections.

Systems in this category include MongoDB, CouchDB, Couchbase, RavenDB, and FatDB. Document databases often integrate with full-text databases such as Solr, Lucene, and ElasticSearch. For example, ElasticSearch provides real-time response to document queries in JSON format; RavenDB uses Lucene. Table IV summarizes characteristics of some document databases.

VIII. Native XML Databases

These databases store documents in native XML format. Query languages supported include XQuery, XSLT, and XPath. Some provide support for JSON. Systems in this category include BaseX, eXist, MarkLogic Server, and Sedna.

The distinguishing feature of MarkLogic Server is security. For this reason, it is widely used in security industry. It provides client access through REST and Java and JSON is supported. REST API is partitioned into three categories: client API for CRUD and search; management API for instrumentation; and packaging API for configuration. MarkLogic Server recently rebranded as Enterprise NoSQL.

IX. Native Object Databases

Object databases, as the name implies, use object models to store data. These systems combine database functionality with object-oriented programming language capabilities. They allow creating and modifying objects within the database system. Some systems are tightly integrated with an object-oriented programming language, while others feature database-specific programming language. This eliminated the *impedance mismatch* problem of RDBMS since the same data model is used by the programming language and the database. A subclass within object databases is called object-relational. These systems combine traditional RDBMS capabilities with object databases. Native object databases are ideally suited for complex data often found in computer-aided design, manufacturing, spatial, graphics, and multimedia applications.

Query languages for object databases are typically declarative. Object database queries typically run much faster since no RDBMS joins are involved. Object Query Language (OQL) is the result of standardization effort spearheaded by the Object Data Management Group.

Current systems in this category include Versant Object Database, Versant JPA, FastObjects, db4o, and WakandaDB. The first three are marketed by Versant and db4o is an open source system. Versant Object Database targets applications with high performance and concurrency requirements. Client access is provided through interfaces for Java, .NET and C++. Versant JPA is an object database with JPA 2.0 compliant client interface. FastObjects primarily provides object-based persistence for .NET applications.

X. Hybrid Systems

Systems in this category are those that evolved from the traditional RDBMS or those that fall into more than one category discussed above. Systems in this category include PostgreSQL, VoltDB, OrientDB, Aerospike, ArangoDB, and Spanner. On Amazon EC2, VoltDB achieved 95 thousand transactions per second (TPS) in a Node.js application running on 12 nodes; and 34 million TPS with 30 nodes. Table V summarizes characteristics of some hybrid databases.

XI. Conclusions

Unprecedented data volumes, connected data, performance and scalability requirements of modern Big data-driven applications effectively challenged the practice that RDBMS is the only approach for data management. Consistency, availability, and partition tolerance are the three primary concerns that determine which data management system is suitable for a given application.

Netflix moved from Oracle RDBMS to Apache Cassandra, and achieved over a million writes per second across the cluster with over 10,000 writes per second per node while maintaining the average latency at less than 0.015 milliseconds. Total cost of Cassandra set up and running on Amazon EC2 was at around $60 per hour for a cluster of 48 nodes.

Cisco recently replaced an Oracle RAC solution for master data management with Neo4J. Query times were reduced from minutes to milliseconds in addition to expressing queries on connected data with ease. This application has 35 million nodes, 50 million relationships, and 600 million properties.

As the above two cases exemplify, NoSQL data models are designed for efficiently supporting insert and read operations; storing sparse data and column-wise processing; enabling disconnected client operation and delayed synchronization; and tolerance to temporary inconsistency and its implications.

A more typical scenario for NoSQL systems to gain ubiquitous use will come from using an array of data management systems, each naturally suited for the type of data and operations, all data management systems abstracted away through a Web or application server. Furthermore, Web servers will manage access control and authorization centrally. Also, the migration from

Table IV
DOCUMENT DATABASES

Name	Salient Characteristics
MongoDB	No transaction support. Only modifier operations offer atomic consistency. Lack of isolation levels may result in phantom reads. Uses memory-mapped files storage. Support is available for geospatial processing and MapReduce framework. Indexing, replication, GridFS, and aggregation pipeline. JavaScript expressions as queries. Client access tools: JS Shell (command line tool), and drivers for most programming languages. Suitable for applications that require auto-sharding, high horizontal scalability for managing schema-less semi-structured documents. Stores documents in BSON format and data is transferred across the wire in binary format.
CouchDB	Open Source database written in Erlang. JSON format for documents. Client access tools: REST API, CouchApps (an application server), and MapReduce. JavaScript is used for writing MapReduce functions.
Couchbase	Incorporates functionality of CouchDB and Membase. Data is automatically partitioned across cluster nodes. All nodes can do both reads and writes. Used in many commercial high availability applications and games.

Table V
HYBRID DATABASES

Name	Salient Characteristics
PostgreSQL	Recent versions provide support for JSON and key-value support through an add-on called *hstore*. Suitable for applications that primarily depend on RDBMS for data management, but also face a need for scalable means for managing key-value data.
VoltDB	In-memory database running on a single thread. Eliminates the overheads associated with locking and latching in multi-threaded environments. Uses snapshots to save data to the disk. Database can be restored to a previous state using snapshots. Data is distributed across several servers. Supports transactions. Supports only a subset of ANSI/ISO SQL. Migration to VoltDB will require rewriting some of the existing SQL queries. Client interfaces: JSON API, Java, C++, C#, PHP, Python, Node.js, Ruby, and Erlang.
VoltCache	A key-value database implemented on top of VoltDB. Client access is through a Memcached compatible API.
OrientDB	Provides document, key-value, and graph databases functionality.
ArangoDB	Provides document, key-value, and graph databases functionality.
Aerospike	Hybrid system like OrientDB, but also provides traditional RDBMS functionality.
Google Spanner	Multi-version, globally-distributed, and synchronously-replicated database. Spanner supports externally-consistent distributed transactions.

RDBMS to NoSQL systems is eased as the latter return results in JSON format.

Though NoSQL systems are predominantly used for new applications which are characterized by horizontal scalability, high performance, relaxed and eventual consistency, it is also likely that existing applications will begin to use NoSQL through reegineering process. The current upheaval in the data management systems will help promote using the system that closely matches the application needs. Services such as Amazon EC2 will make NoSQL systems more economical and within reach for all organizations, both small and big.

REFERENCES

[1] R. Cattell, "Scalable sql and nosql data stores," *SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, May 2011.

[2] V. Benzaken, G. Castagna, K. Nguyen, and J. Siméon, "Static and dynamic semantics of NoSQL languages," *SIGPLAN Not.*, vol. 48, no. 1, pp. 101–114, Jan. 2013.

[3] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. a. Paulo, J. Pereira, and R. Vilaça, "MeT: Workload aware elasticity for NoSQL, booktitle = Proceedings of the 8th ACM European Conference on Computer Systems, series = EuroSys '13, year = 2013, isbn = 978-1-4503-1994-2, location = Prague, Czech Republic, pages = 183–196, numpages = 14, publisher = ACM, address = New York, NY, USA."

[4] D. McCreary and A. Kelly, *Making Sense of NoSQL: A guide for managers and the rest of us*. Manning Publications, 2013.

[5] C. Mohan, "History repeats itself: Sensible and NonsenSQL aspects of the NoSQL hoopla," in *Proceedings of the 16th International Conference on Extending Database Technology*, ser. EDBT '13. New York, NY, USA: ACM, 2013, pp. 11–16.

[6] V. Abramova and J. Bernardino, "NoSQL databases: MongoDB vs Cassandra," in *Proceedings of the International C* Conference on Computer Science and Software Engineering*, ser. C3S2E '13. New York, NY, USA: ACM, 2013, pp. 14–22.

[7] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O'Reilly, 2013.