Journal of Database Management
Volume 31 • Issue 3 • July-September 2020

The Rise of NoSQL Systems:
Research and Pedagogy

Akhilesh Bajaj, University of Tulsa, USA
Wade Bick, University of Tulsa, USA

ABSTRACT

Transaction processing systems are primarily based on the relational model of data and offer the advantages of decades of research and experience in enforcing data quality through integrity constraints, allowing concurrent access and supporting recoverability. From a performance standpoint, they offer joins-based query optimization and data structures to promote fast reads and writes, but are usually vertically scalable from a hardware standpoint. NoSQL (Not Only SQL) systems follow different data representation formats than relations, such as key-value pairs, graphs, documents or column-families. They offer a flexible data representation format as well as horizontal hardware scalability so that Big Data can be processed in real time. In this review article, we review recent research on each type of system, and then discuss how teaching of NoSQL may be incorporated into traditional undergraduate database courses in information systems curricula.

KEYWORDS

Column Family Database, Database Consistency, Database Course, Database Partitioning, Graph Database, Information Systems Curriculum, JSON Database, Key Hashing, Key-Value Database, NoSQL, Relational Database, XML Database

INTRODUCTION

Starting from the 1980-s, the relational model of data has dominated the storage and retrieval of commercial data in the business world. Relational database management systems (DBMS) by vendors such as Oracle, IBM, Microsoft and open source offerings such as MySQL have evolved into stable platforms that use the SQL (Structured Query Language) standard for developing online transaction processing applications (OLTP), which constitute the bulk of information systems used by businesses.

However, the explosive growth of Web 2.0 and the Internet of Things (IoT) has led to a proliferation of massive amounts of data, often termed big data, that needs to be processed in real time (van der Aalst, 2018). The structuring of information into normalized tables with subsequent joins for querying, the overhead required for checking integrity constraints, and the enablement of data recoverability has led to performance bottlenecks when big data is stored on relational DBMSs. The first NoSQL (Not only SQL) applications were developed by the initial users of big data, such as Google that began development of Bigtable in 2004 (Whitchcock, 2005). Over time, several different NoSQL data models have emerged based on the needs of different domains. In each case, they have emerged because of

DOI: 10.4018/JDM.2020070104

faster performance and support of evolving data schemas. As of now, a plethora of NoSQL systems exist, though relational DBMSs are still by far the most popular (see usage data for various DBMSs at https://db-engines.com/en/ranking). As each of these systems has evolved, they have raised their own set of research questions.

From a teaching standpoint, the undergraduate curricula in Information Systems have contained a course on Database Management starting from the 1980s, and the most recent ACM IS 2010 curriculum specifies that this course largely deal with designing a normalized relational schema and building applications using the structured query language (SQL) standard (Topi et al., 2010). However, recent industry trends indicate that the adoption of NoSQL systems is increasing by organizations who are increasingly willing to try out these new data stores in a polyglot approach to managing their data storage (Simone, 2019). At the same time, many of the cloud vendors that offer NoSQL stores also offer relational stores which are increasing in popularity. The CIO of Amazon describes how their relational Aurora product is the fastest growing service in the history of Amazon Web Services (AWS) in (Vogel, 2019).

Given this changing landscape, a review of open research questions in SQL and NoSQL systems is needed. Another question that needs exploration is: should IS curricula adopt NoSQL as part of their database course, perhaps at the expense of teaching relational concepts? In this work we describe current and evolving research questions for each type of NoSQL system. Next, we contrast relational versus NoSQL systems, based on their application development methods, their application domains and the level of standards available. Using this review, we analyze how NoSQL can fit into an IS database curriculum over the near to midterm horizon, and if it is appropriate to replace relational concepts with NoSQL concepts in a core database course.

The rest of this paper is organized as follows. In Section 2, we describe the different categories of data models available today and describe open research questions for each model. In Section 3, we describe the application development methods commonly used for each type, as well as the domains of common application. Section 4 contains an analysis of appropriate curricular content for each type in an undergraduate IS (Information Systems) curriculum. We conclude in Section 5 with a discussion on the future importance of each type of system.

REVIEW OF DIFFERENT DATA MODELS AVAILABLE

First, we describe research related to consistency and availability issues, followed by data partitioning research. Next, we look at the following data models that are in wide use today: the relational model, the key-value (KV) model, the column family model, the graph model, the JSON model and the XML (eXtensible Markup Language) model.

ACID, BASE and the CAP Theorem

The consistency, availability, and partition tolerance (CAP) theorem was introduced in the early 2000-s as the size of the Internet was exploding (Gilbert & Lynch, 2002). In its simplest form, it asserts that a database implementation can only maintain two of the three concepts listed in the CAP acronym. For example, if a database system prioritizes consistency and partition tolerance, it will sacrifice some amount of availability. Trying to maintain high availability and partition tolerance would cause consistency issues. Network partitioning is a requirement for database systems being accessed today over networks, so partition tolerance is a basic requirement. Hence the tradeoff when building or selecting a system is considering the continuum with guaranteed consistency on one end, and guaranteed availability on the other end (Brewer, 2012).

The well-known ACID (Atomicity, Consistency, Isolation, and Durability) model signifies the four requirements for transactions in more traditional relational systems (Gray & Reuter, 1992). Systems that follow a pure ACID model are at one end of the CAP continuum, where consistency is always guaranteed. However, in a big data context, this occurs at the expense of availability. At the

other end of the CAP continuum is BASE (Basically Available, Soft-state, Eventually consistent) (Pritchett, 2008). This is a set of design principles where availability is paramount, as in big data real time systems where data partitions ensure that locally consistent data is always available, but global consistency can be accomplished at a later time (V. Gudivada, Apon, & Rao, 2018). Collectively, NoSQL systems tend to be on the side of BASE to different degrees.

The highest level of consistency is strict serializability of transactions. This assures that all transactions can be ordered strictly, so that lost updates and dirty reads are eliminated. For non-transactional systems, this is called linearizability or strong consistency (Davoudian, Chen, & Liu, 2018). Sequential consistency is the next lower level where eventually a total ordering is created, but a read operation may see a dirty value. Since providing consistency over a distributed system is very expensive, many NoSQL systems use atomic aggregates where denormalized data is stored together on the same node. Eventual consistency is the next level of relaxation of consistency, where ordering is not enforced, but ultimately all the copies of the data converge (Vogels, 2009). This allows for real time speeds of big data stores, but dirty writes and reads are a possibility here. Strategies used here include the last write wins rule where every write is timestamped. In some variants of this, the values of each attribute are timestamped, for example in Apache Cassandra and Amazon Dynamo (Lakshman & Malik, 2010). In addition, data can also be repaired while reading inconsistent values. In this strategy, the read operation for a data item gets to see all the data values with their timestamps on different nodes, and then sends back updates to each node with the latest data value (DeCandia et al., 2007). This strategy is used in several systems, including Dynamo, Cassandra and Voldemort.

There are different approaches to data replication, for example, master-slave (Couchbase, MongoDB, Espresso), master-master (BDR for PostgreSQL, GoldenGate for Oracle) and masterless (Dynamo, Cassandra). In a master-slave replication architecture, database clients send the data to one known master partition, and the system in turn updates the slave partitions at a later time. As pointed out recently in (Gonzalez-Aparicio, Younas, Tuya, & Casado, 2019), pushing the tradeoff envelope between availability and consistency for NoSQL stored continues to be an ongoing research area.

Data Partitioning Schemes

Data in distributed systems can be split horizontally, vertically or functionally. Horizontal splits occur when the data schema is preserved across each partition, but only a small subset of data is on each segment or *shard.* Vertical partitioning occurs when a subset of fields in the schema are in each partition, usually based on field access performance of the system. Functional partitioning occurs when the entire schema is split based on some functionality, for example all accounts payable records may be in one partition.

When data is partitioned, the performance scaling is usually horizontal, so that subsequent shards or partitions are loaded onto low cost hardware, and scaling up is indefinite. The design goals for partitioning strategies include minimizing inter-partition requests, load balancing of read and write requests across nodes after normalizing for node capacity and finally facilitating the insertion or removal of new nodes in the system with minimum disruption to partitions (Huang, Wang, Zhong, Song, & Yu, 2015; Schall & Härder, 2015; Stonebraker, Brown, Zhang, & Becla, 2013).

Key based sharding is the most common method of horizontal data partitioning, except in the case of graph databases, where graph vertices that are strongly connected are grouped onto one node. Research in key based sharding may be divided into workload-unaware and workload-aware sharding. Workload-unaware sharding is static in nature, and optimizes storage. Example strategies include range-based sharding where key ranges determine the destination partition for a data item as in MongoDb (Kookarinrat & Temtanapat, 2015). The disadvantages here are that some partitions may have more data since distribution of data items amongst range buckets may not be uniform, and the maintenance of a central mapping structure is required that maps range values to actual nodes.

Hashing is another method, where data item keys are hashed and the hash value is used to determine the storage node. A disadvantage here is the increase in multi partition queries, since data

items that are queried together often may have very different hash values for their keys. Another disadvantage of simple hashing is the removal and insertion of nodes. A solution to this is consistent hashing, where nodes are also given several virtual positions based on a hash function and data items are allocated based on proximity of their hashed key value to the hashed value of the node. This distributes the load so that the removal or addition of nodes is easier. Hashing schemes to meet the design goals of partitioning, described above, continue to be an ongoing area of research. For example, a consistent hashing scheme with bounded loads is proposed in (Mirrokni, Thorup, & Zadimoghaddam, 2018) for dynamic load balancing, where the goal is to minimize the maximum load on any node, and to minimize the number of data items to be moved if any node is added or removed.

Relational Model

Proposed in 1970 (Codd, 1970), the relational model is the most popular model in existence for the storage and access of alpha-numeric data. Decades of research have created secure and reliable systems that allow concurrent access to users in an organization, secure recovery in case of software or hardware failure and declarative access to data via SQL. Role based access is also a feature of relational models. While earlier iterations were optimized for more writes to the database, also called online transaction processing (OLTP), newer iterations provide storage that is optimized for online analytic processing (Chaudhuri & Dayal, 1997), where roll-up and drill-down queries are optimized across measures of interest such as sales figures along different dimensions.

The relational schema needs to be usually defined at the onset of deploying an application, and is not amenable to frequent evolution (V. N. Gudivada, Rao, & Raghavan, 2016). Read and write performance can be scaled up in relational systems via vertical scaling (adding more expensive hardware) though this is expensive and reaches a ceiling quickly (Vaquero, Rodero-Merino, & Buyya, 2011). The relational database market is over $40 billion and expected to constitute over 80% 0f the entire database market till 2022(Doherty, 2013; Wells, 2019). Over 100 relational database management system vendors exist, including Oracle, Microsoft SQL server, IBM DB2 and MySQL.

From a research standpoint, relational systems are still an active area of research in many areas. Hardware advances, such as NVMe (Non Volatile Memory Express) drives that deliver significantly lower latency and higher bandwidth for I/O applications continue to appear, and exploring how these advances impact relational database design going forward is an important research question. For example, it was found in (Xu et al., 2015) that NVMe backed systems delivered up to eight times faster client-side performance over enterprise-class SATA (serial advanced technology attachment) solid state drives. Indexing schemes to get better read and write performance from relational stores also continue to be an area of research. A survey of 48 indexing techniques was presented in (Gani, Siddiqa, Shamshirband, & Hanum, 2016), with a view to managing big data on different data models. For example, a compact steiner index was presented in (Li, Feng, Zhou, & Wang, 2011) to facilitate keyword searches in relational databases.

Another ongoing area of research is associating ontologies with relational data. This has become easier since OWL (One World Language or Web Ontology Language) became the de-facto standard for creating class-based ontologies. A survey of tools to map ontologies to relational data is presented in (Moldovan et al., 2015). Ontologies can be used to create more natural query interfaces. For example, usage of an OWL schema to create fuzzy queries on scalar data in a relational database is presented in (Martínez-Cruz, Noguera, & Vila, 2016). A natural language query interface using ontologies is proposed in (Lei et al., 2018). Mapping semantic constraints in OWL to relational database implementation mechanisms such as triggers is another current area of research (Achpal, Kumar, & Mahesh, 2016). A useful summary of ontologies for knowledge modeling and information retrieval is available in (Munir & Anjum, 2018).

A third area of research is the emergence of NewSQL databases that combine the throughput performance of NoSQL systems with the ACID properties of relational systems (Duggirala, 2018). Many of these use some form of in-memory storage to enhance I/O performance. Application domains

of NewSQL databases are described in (Almassabi, Bawazeer, & Adam, 2018), and include archival data warehouses, real-time and streaming systems and systems that support time stamped information.

Key Value Model

Data in the key value model is stored as simple key-value pairs, where a key is generated using any partitioning algorithm, such as a hash algorithm, and the value is stored as an opaque binary BLOB (binary large object). Once the application has the key, the BLOB can be made to appear as a list, or a dictionary, or another structure, depending on the system and the application. The performance of key based querying is independent of the amount of data stored. The KV data model is very extensible in a deployed application, so that the value in the next KV pair does not have to match the format of the value of an earlier KV pair. KV systems lend themselves well to horizontal scaling (adding cheaper machines to a pool of resources) since data can be sharded more easily and shards can be stored on different servers. Aggregate querying is also facilitated using a MapReduce program that runs queries in parallel across multiple shards and combines the result (Dean & Ghemawat, 2008). Big data applications that require real time reading and writing performance in milliseconds benefit from the KV model. Over 50 vendors exist for KV applications, including Aerospike, Redis, Memcached, Riak and DynamoDB.

While applications such as web session logs and shopping carts need primarily key based querying, most applications require some form of value-based querying. Rather than require the application to parse the value object and write query code, recent KV systems support indexing and querying values of certain data types, such as lists (redis and Aerospike) or table rows (HyperDex and Spinnaker).

In-memory KV systems (such as Memcached and Voldemort) offer very fast access, and are used for cloud caching or web session information. Persistent systems (such as Riak and Oracle NoSQL) offer solid state disk storage, while hybrid systems (such as Aerospike and Redis) offer in memory performance with persistent storage if certain conditions are met (Davoudian et al., 2018). Ongoing research in KV systems focuses on developing access control models for data access (Moreno, Fernandez, Fernandez-Medina, & Serrano, 2018) and teasing out data semantics in the value portion (Rudnicki, Cox, Donohue, & Jensen, 2018). While KV systems offer fast query performance, the issue of data quality in the value portion is an issue since integrity constraints have to be written at the application level.

Column Family Model

Unlike relational systems that are relatively immutable when it comes to schema evolution, column family systems model data as multiple families of columns, where usually intra-column mutability is facilitated. Column families are not easily mutable once created for an application, but columns within a family can be easily added or taken away. This allows for the data schema to change easily as the application evolves, so that each "row" within a column family can have a flexible number of columns. Columns within a family are physically stored together and easily accessed quickly. In essence, a nested KV model can be used to represent a column family schema, where each "row" has one key, and then nested key-value pairs to store the data. New values do not replace old ones, but are instead time stamped, so a triple <*row-key, column-key, timestamp*> can be used to access data (Davoudian et al., 2018; Wiese, Waage, & Brenner, 2019). Computing aggregate metrics on large databases that have temporal data, such as financial applications, is a classic application of column family databases. Optimizing aggregate queries is an ongoing research question for column family systems (Storey & Song, 2017). For example, storing the results of expected searches on big data, so that, say, all in-box mail messages related to 'picnics' are in one row and all messages related to user 'johnDoe21' are in another row is another example of the application of column family databases. An aggregate search can then be run very fast for retrieving messages based on search criteria. Notable systems include Cassandra, HBase, BigTable and HyperTable.

The partitioning of column family systems can be by column families (vertical) or by rows (horizontal). Storage is usually in-memory, using a data structure such as an LSM (log-structured merge) tree for each column family (Wu, Xu, Shao, & Jiang, 2015). Copies of these trees are kept in persistent storage, usually immutably, so that the most recent copy is used to access the data. Updates become faster because of the immutability of old values in older copies. The MapReduce programming model pioneered by Google works well with the column family model, and there is ongoing research to alter the model to accommodate different types of domains. For example, processing Geospatial data in HBase using MapReduce is covered in (Gao, Yue, Wu, & Zhang, 2017). A two-phased MapReduce algorithm to facilitate replication of data in data warehouses is presented in (Barkhordari & Niamanesh, 2018). A summary of the different MapReduce variants used in Column Family systems is presented in (Seera & Taruna, 2018).

Graph Model

The native graph model stores data as vertices with edges. Usually vertices represent things or events, and edges represent relationships between these things or events. Attributes can be used to further describe the vertices and the edges. Ongoing research in this model involves two broad areas: storage of graph data, and optimizing deep queries.

Systems such as early versions of twitter FlockDB stored their graph data in adjacency lists on a relational system such as MYSQL. Adjacency lists store a single list of neighbor vertices for each vertices in a non-directed graph, and two lists (in and out) for each vertex for a directed graph. This storage causes data redundancy, since each link information is stored once for each vertex in the link. Updates are faster with this kind of storage over a matrix representation, since a vertex can be easily added by concatenation of a list for the new vertex. A variation of this is called an edge list, where the source and destination vertice IDs of each edge are stored in a two column relational database table, that may be indexed using, say, a B-Tree (Comer, 1979). In this case, data redundancy is lower since each piece of information is retained only once in the database.

The most popular graph storage format is a compressed sparse row (CSR) where two integer arrays are used (Qian, Childers, Huang, Guo, & Wang, 2018). The first is an edge array that maps an edge ID to the ID of its destination vertex, and the second is a vertex array that maps each vertex to the ID of its first outgoing edge. The main advantage of CSR is the contiguous storage of outgoing edges of each vertex in main memory, thereby leading to a reduced need for secondary storage access. The cost of edge insertion or removal is O(*m)* where *m* is the number of edges. CSR is commonly found in many graph engines such as GraphChi (Kyrola, Blelloch, & Guestrin, 2012) where billions of edges can be managed. Recent advances in the area include G-store where trillion edge graphs are optimized (Kumar & Huang, 2016).

The second area of research in graph databases is the querying of large graphs. A good survey of modern query languages for graph databases is presented in (Angles et al., 2017). Graph queries can be real time queries, relating to a small portion of a large graph, or off line computations that optimize queries covering over large portions of the graph. The storage mechanisms for these two needs are different, in that the former uses an online model, where updates and queries over small portions of the graph are optimized, while the latter may actually take the graph offline and optimize for quick traversal over the graph. Systems like GraphCHI and Microsoft Trinity support both aspects of graph querying. The three most popular query languages for graphs are SPARQL (Harris & Seaborne, 2013), Neo4J's Cypher and Apache's Gremlin. All three employ graph pattern matching, where graph patterns with variables for edges and nodes are matched with actual data in a graph schema. Both homomorphic (SPARQL) and isomorphic (Neo4j) match strategies may be supported when it comes to criteria for finding matches between patterns and actual data. A second aspect of queries is finding paths between nodes that satisfy certain criteria such as existence of the path, satisfaction of a regular expression or more complex queries (Libkin, Martens, & Vrgoč, 2016).

The selection of proper standardized semantics for these aspects of graph queries, across languages, is an ongoing research issue.

Document Model (JSON and XML)

The data model used here is a variant of the key-value model, where the value is not an opaque BLOB, but is instead a structured file format, usually JSON (JavaScript Object Notation) or XML (eXtensible Markup Language). Each value is a file with its own attributes or fields. The data schema can evolve since subsequent files need not have the same fields as prior ones. Documents that are semi structured with widely varying fields whose field structure and data types can change are suitable for the document model. Indices can be built for different attributes. Aggregate querying across document collections is optimized in these systems (V. N. Gudivada et al., 2016). Over 40 vendors for document systems exist, including MongoDB, CouchDB, CouchBase, DynamoDB. Since the difference between the KV model and the document data model lies in how the value is stored, many systems offer a choice of both. An example query language in this domain is N1QL (non first normal form query language) in CouchBase that supports declarative querying across documents.

Database management systems supporting the XML standard go beyond just document model systems. XML schemas can be validated using namespaces or document type definitions (DTD), so a schema can be enforced on data. At the same time, because of the extensibility, a new DTD can be defined easily, thereby supporting the evolution of the schema as the domain changes. Rule-based data validation can also be done using the Schematron standard (Van der Vlist, 2007). Unlike add-on modules to some systems that store XML files as Character Large Object (CLOB) files, data is stored in these systems natively as an XML file (V. N. Gudivada et al., 2016).

XPath and XQuery extensions are standards that can support reads and writes to the data. Other accompanying standards such as XForms and XProc also may be implemented in these systems, thereby supporting an evolving standard in data processing and application development. Standards based storage, access and processing allows applications to be more easily moved across different native XML systems and allows for better recovery and security policies. Examples of native XML systems include MarkLogic, BaseX, and Sedna. An ongoing area of research in document-based systems is polyglot storage, so that the relationships between documents may be stored in, say, a graph representation, while the data itself is stored in an underlying JSON or XML format (Oliveira & del Val Cura, 2016; Sadalage & Fowler, 2013).

Table 1 summarizes the ongoing research questions in the different NoSQL data models discussed in this section.

Next, we briefly describe the application development methods that may be used for the different models, and how this may impact pedagogical content in IS.

APPLICATION DEVELOPMENT AND DOMAINS

Relational Model

There is a great deal of mathematical theory associated with the design of relational schemas, primarily based on the notion of functional dependencies between attributes in the schema (Kent, 1983). Reducing redundancy of the data that is stored with appropriate forms of normalization, the relational schema is widely taught in undergraduate database courses in both IS and computer science curricula. Semantic models like the extended entity relationship model (EER) have also been developed to bridge the gap between natural language descriptions of domain requirements and a relational schema design (Engels et al., 1992; Teorey, Yang, & Fry, 1986). Thus, the notion of designing a schema for improved data quality in the application is well understood when using the relational model. Inbuilt constraint enforcement mechanisms such as primary key and referential integrity constraints and triggers are built in to most relational OLTP systems, further enhancing their

Table 1. Ongoing research areas in NoSQL systems

NoSQL Area of Research	Ongoing Research Questions
Consistency/Availability	Algorithms to push the consistency-availability envelope
Partitioning Schemes	Schemes to reduce inter-partition requests, load balance read/write requests across nodes and reduce disruption to partitions when nodes are inserted or removed.
Relational Model	Memory management changes because of advances in secondary storage latency and bandwidth. Indexing schemes for improved read/write performance. Mapping of ontologies to relational schemas and integrity constraints such as referential integrity and triggers. Increase throughput and read/write latency to NoSQL levels while still allowing for relational schemas.
Key Value Model	Providing data access security to roles. Capturing application semantics and integrity constraints in the value portion.
Column Family Model	Balance latency of aggregate queries with storage requirements. Apply column family data model to different domains. Modify MapReduce algorithms for increased performance.
Graph Model	Formulate graph storage methods so that storage space is lower, link insertion and removal is faster and access is faster. Standardize and optimize query language constructs for small segment queries and larger traversal queries.
Document Model	Develop systems with polyglot storage to capture links between document segments. Evolving standards in JSON to match those of XML for document schema validation and querying.

abilities to facilitate data quality for transaction processing applications with alpha numeric data, as is currently found in most functional business areas (Reiter, 1988). Role based security mechanisms with a fine-grained level of control, the ability to support concurrent users and recovery from system faults are also built into these systems.

Typically, the performance scaling for these systems is vertical, in that faster speed is usually obtained through increasing the main memory, disk access speeds and processing speeds of existing hardware. This limitation implies that relational systems typically are not suitable for very high throughput applications where input data is generated from programs, such as data from Internet of Things (IoT) appliances, or webserver logs. Instead, they are more suitable for applications where data is being input by several humans or machines in a typical business setting, such as customer relationship management, banking, etc.

Column Family Systems

Column family systems like Cassandra do allow the creation of a primary key. The key is used to ensure uniqueness of records, and also to determine the location of the partition that will house that record. This facilitates searches by key. However, there is no support for referential integrity, triggers or other data quality checks. This means that data is often stored in duplicate locations, leading to redundancy but very fast reads since no joins are needed.

Columns within a family can be dynamically or statically typed, and families are extensible, which supports applications with an evolving data schema. An application development framework for column family systems is more involved than KV systems, since there is some structure to data (columns), though normalization is not an option. Since time stamps are assigned to each update/delete/insert mutation with micro second granularity, big data applications such as financial transactions and streaming multimedia where time stamping is important are suitable for column family systems.

Graph Systems

The logical model for graph systems is vertex-edge information which is often physically implemented in different formats. Support for transactions or data partitioning varies based on the system, and there seems to be no converging standard at present for how to develop applications. Graph systems are very useful where large datasets of vertex-edge information is needed, as in recommender systems, security access systems, social networks, etc.

Document Systems

Document systems that are non-XML usually store information as Key-JSON file pairs. Systems like CouchDB offer both a key-binary value pair and a key-JSON file storage model. Systems like MongoDB and CouchDB also allow system specific document validation features, though no standard exists. JSON has advantages over XML in that it can be parsed with JavaScript functions versus using an XML parser and it supports arrays, so less coding is required to move data to and from a JSON file. For AJAX style applications, JSON is easier to use than XML since no looping through the XML document object model is required. JSON files also tend to be smaller and faster to transmit and retrieve in the case of big data applications. Applications that require flexible schemas, have mainly alpha numeric data and require fast throughput on the web are good candidates for JSON based document systems.

XML systems store data in native XML format, which means they can also store data with varying degrees of structure, similar to JSON based systems. However, in XML, attributes can be put into the opening tags in XML, thereby delineating metadata from actual data. Application development is the most mature here amongst all the NoSQL systems. First, data schema design can be undertaken using constraints such as primary key and referential integrity constraints, with automated checking using Schematron. Every new documents' schema can also be checked against a DTD or a namespace (Bray, Hollander, & Layman, 1999).

Second, query-based applications can be written using the Xpath/XQuery standards, so applications are portable across systems. Third, front end client applications can be built using XForms, that supports common graphic user interface (GUI) architectures. Standards-based development permits the usual advantages of greater availability of technical personnel, portability across different XML platforms and easier outsourcing of development. XML based systems are slower than JSON based systems, but provide better support for document heavy applications. Typical applications for XML systems include workflow systems with very long transactions and evolving schemas, automated supply chain systems and any application with large amounts of semi-structured data that is mainly alphanumeric.

Key Value Systems

KV systems are applicable in very high throughput, big data environments where real time response is essential. Examples include web session data, auction systems, dynamic customized user interface generation for a user, with thousands of users concurrently logged in, recommendation engines, etc. The design of data in KV systems is very flexible, since the storage structure is an opaque binary object. The selection of what type of key to use is somewhat important, though often a hash key may be sufficient. Thus, in terms of data quality enforcement, all mechanisms must be coded in the application code, unlike the relational model with built in constraints and triggers. Applications where the primary lookup is via keys are especially well suited for KV systems, especially if the data schema evolves constantly.

Table 2 summarizes the discussion above and contrasts the application development in he different data models.

Next, we explore the extent to which NoSQL needs to be incorporated into a database course in Information Systems.

Table 2. Application domains and development for relational and NoSQL data models

Data Model	Logical Representation	Major Characteristics	Application Development	Application Areas
Relational Model	Relations, Attributes	Integrity constraints, fine grained security, SQL standard, strong ACID support	Strong normal form design theory, SQL standard to write code, vertical scaling	OLTP and OLAP systems, multiuser systems usually within an organization. Query speed and throughput not as critical as data quality. Typically not used for big data applications.
Column Family Model	Columns which can be dynamically added to a family	Primary key, but no normalization support	Lots of duplication of data, since each query often becomes its own family. Leads to very fast reads.	Big data applications where time stamping is important, and data still needs to be organized into columns. Includes financial applications and data warehouses.
Graph Model	Nodes and edges, with attributes for both	No normalization or other standardized integrity constraint support. Supports graph oriented operations such as shortest-path, minimum spanning tree, cluster analysis, etc.	Identifying what to model as nodes versus edges, what attributes to capture. Developing queries using the language provided by the particular system.	Big data application where richness of relationships is important. Examples include recommender systems and social network analysis.
JSON Systems	Key-Value pairs with a JSON file as each value	Use JSON syntax to capture data structure, which is lightweight, though lacks meta data specification support.	Structure the data as JSON files, and Uses JavaScript to process quickly. Data integrity constraint checking is unstandardized and needs to be coded.	Web based big data applications where there is lots of network traffic and the data model evolves, but is mainly alphanumeric. Examples include web based sharing of large numbers of documents, text searches on large databases, etc.
XML Systems	Key-Value pairs with an XML file as each value	Use XML syntax to model data structure, structure can be validated against DTD or namespace, and values can be validated using a standard like Schematron.	Structure the data fields, and list integrity constraints. Develop applications using Schematron and XForms. Set up queries using XQuery and XPath.	Data applications where data sharing is essential and the data model evolves, but where real time speed is not critical and data quality and integrity checking is more important. Examples include multi organizational supply chain systems, and other OLTP software that spans multiple databases.
Key-Value Systems	Key-value pairs where the value is an opaque binary value	Keys are partitioned and usually hash values. Maximum flexibility since each value can be anything, including multimedia. No checking for integrity constraints	Decide on the key partitioning scheme and which objects to implement as values. Decide on data partition mechanisms so local data is always consistent for application.	Very fast real time performance in big data sets for writes and reads, independent of size of data. Works well for distributed applications, as long as localized data partitions are consistent. Any kind of application that requires non alpha numeric data with very flexible structure. Examples include large scale video databases, music databases, image databases, etc.

HOW SHOULD NOSQL DATABASES BE INCORPORATED INTO AN IS CURRICULUM?

The overall market for big data software and services is estimated to grow from $ 42 billion to $103 billion from 2017 to 2027 (Columbus, 2018). The market for NoSQL databases is relatively small, and though growing relatively fast, likely to remain at under 10% of the overall database market till 2022, as per a report in (Wells, 2019). An analysis of data published by IDC, the market intelligence firm, indicates that the total database market in 2022 will be approximately $40.4 billion with relational databases accounting for about $33 billion (Mullins, 2019). The NoSQL market in the same analysis is estimated to be approximately $3.7 billion in 2022. This implies that storage and usage of transactional data, stored in a relational database, in the business world will dominate information generation and usage in the near future.

The ACM/AIS curriculum for IS schools recommends a course termed "Data and Information Management" (IS 2010.2) with an intent to provide "an introduction to the core concepts in data and information management" (Topi et al., 2010). The recommended learning goals incorporate conceptual modeling (*e.g.* entity-relationship model), relational database design with normalization levels, with a primary focus on OLTP systems. Understanding OLAP, semi-structured and unstructured data management is listed in goals 19 and 21 (out of a total of 21 learning goals). One of the authors of this paper has been teaching an IS database class at various US based Universities since 1995, and this is typical of the current syllabi used in IS curricula, where ER or EER (extended ER) modeling, normalization, SQL and application development on a relational platform are the bulk of the course, with one or two lectures regarding warehousing and semi-structured or unstructured data. These components conform to the well understood requirement that University level courses offer knowledge with a "long shelf life" that is independent of any particular IS artifact or system.

The discussion of NoSQL systems in Sections 2 and 3 of this work indicates that NOSQL has undergone a bottom-up evolution, where solutions were put together to solve performance and scaling issues in emerging web and big data scenarios. There is a lack of underlying theory guiding database design and standards similar to SQL, except to some degree in the case of XML systems. Application development on NoSQL systems also seems to be largely based on the features provided by each vendor, versus a set of standards. Finally, there are many different types of NoSQL systems, depending on the big data application domain.

An early work on incorporating NoSQL into a traditional database course argued that NoSQL needs to be added because of "industry changes" and suggested that students be asked to do the same project on relational and a NoSQL platform (Stanier, 2012). More recently, a case was proposed that could be used as a NoSQL project in a traditional database course, using CouchDB (Fowler, Godin, & Geddy, 2016). The goal behind this project was to show students how an evolving data schema can be captured in CouchDB, and sample JavaScript queries were provided. Pre- and post- test measures showed that students had a better understanding of why NoSQL is helpful in a social networking context and how it supports an evolving data schema. However, as shown in (Topi et al., 2010), the current AIS curriculum for a database course has little room to add new content, without taking away content. Further, the challenge is to teach "long shelf life" intellectual content, which can be a challenge in a landscape that evolved bottom-up and has few standards.

Should a new course for NoSQL be created? A recent work described the creation of such a course in the curriculum at Rose-Hulman Institute of Technology (Mohan, 2018). It was argued that in computer science, the emphasis on course creation is based on realistic use case scenarios and problem-based learning (Walker & Slotterbeck, 2002), and not just on theory-based learning. The course exposed students to the underlying motivations behind NoSQL, and a variety of NoSQL databases (document, graph, KV and Columnar). Students had to work on an application on each system and reflect on the differences and applicability of each system. A final project required

students to develop an information system using a NoSQL system they select. Student feedback was positive on the course.

While concepts such as scaling, big-data performance and multiple system application development can be covered in a computer science course, are they relevant in an IS curriculum? As per the 2010 IS Curriculum in IS, the goal of IS programs is to produce individuals who can use technology to make the enterprise work better (Topi et al., 2010) pp. 374. Given the fluid landscape of NoSQL models, and the continued preponderance of relational systems, we recommend that IS courses in the area of databases do not significantly change their content at present. However, the fast growth of NoSQL in big data applications implies that at the very least, IS student should be aware of the contrast between relational and NoSQL systems, and how application development would occur in a NoSQL scenario. One option here is to utilize a lecture in the traditional IS database course to discuss these issues, perhaps with example scenarios. A second choice would be to introduce a NoSQL course as a more technical elective in the curriculum, where students build applications in a variety of NoSQL systems, along the lines of the course described in (Mohan, 2018).

CONCLUSION

NoSQL systems evolved from the ground-up to help manage latency and throughput in big data applications where much of the data comes from automated sources, such as a database of web pages maintained by a search engine, or web server logs to customize user experience. Several data models, such as column families, key-value pairs, graphs and document-based systems have emerged to represent data in different domains. They share common traits such as emphasizing low latency and high throughput at the expense of data consistency and management of data integrity. In this work, we highlighted ongoing research questions for each of these data models, and summarized current techniques used to develop applications on the different data models' platforms.

The historic evolution of relational databases systems offers some guidance on the future of application development in the NoSQL landscape. The relational model was proposed in (Codd, 1970), and the relational calculus shortly thereafter (Codd, 1971). The emergence of SQL (Chamberlin, 1980; Pirotte, 1979) allowed the implementation of relational systems on the university campus and eventually into industry. At the same time, the rise of conceptual modeling (Chen, 1976) allowed the creation of an application development methodology that bridged the gap between end-user requirements in an enterprise and the design and implementation of transaction processing systems. Database courses in computer science curricula dealt with learning the architecture of relational DBMSs, from the point of view of the algorithms related to enforcing the ACID requirement. Courses in IS curricula evolved differently and dealt with translating business requirements into a normalized design, using an intermediate conceptual model such as the E-R model, and applying SQL to create applications on an existing DBMS platform.

In the area of NoSQL systems, there has been an absence of a top-down theory that has driven the evolution of the different data models. Bottom-up principles such as horizontal scaling, data partitioning and aggregating denormalized data have been the driving factors, motivated by real world data handling requirements that have mushroomed for specific big data applications. However, it is important to note that while big data application usage grows, the market for NoSQL systems is still projected to be a small fraction of the overall DBMS market, in the future. While University courses in computer science are starting to appear that discuss architectural issues, there has not been an application development methodology that has emerged, specific to NoSQL, that would warrant a course in IS curricula at present. IS students need to be aware of NoSQL principles and where NoSQL can be applied, but relational design and SQL continues to remain a core skill for IS majors that cannot be replaced.

REFERENCES

Achpal, A., Kumar, V. B., & Mahesh, K. (2016). Modeling Ontology Semantic Constraints in Relational Database Management System. *Paper presented at the International MultiConference of Engineers and Computer Scientists*. Academic Press.

Almassabi, A., Bawazeer, O., & Adam, S. (2018). Top NewSQL Databases and Features Classification. *International Journal of Database Management Systems*, *10*(2), 11–31. doi:10.5121/ijdms.2018.10202

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., & Vrgoč, D. (2017). Foundations of modern query languages for graph databases. *ACM Computing Surveys*, *50*(5), 68. doi:10.1145/3104031

Barkhordari, M., & Niamanesh, M. (2018). Chabok: A Map-Reduce based method to solve data warehouse problems. *Journal of Big Data*, *5*(1), 40. doi:10.1186/s40537-018-0144-5

Bray, T., Hollander, D., & Layman, A. (1999). Namespaces in XML. *World Wide Web Consortium.* Retrieved from http://www.w3.org/TR/1999/REC-xml-names-19990114

Brewer, E. (2012). CAP Twelve years Later: How the. *Computer*, *45*(2), 23–29. doi:10.1109/MC.2012.37

Chamberlin, D. D. (1980). *A summary of user experience with the SQL data sublanguage. IBM Thomas J*. Watson Research Division.

Chaudhuri, S., & Dayal, U. (1997). An overview of data warehousing and OLAP technology. *SIGMOD Record*, *26*(1), 65–74. doi:10.1145/248603.248616

Chen, P. P.-S. (1976). The entity-relationship model—Toward a unified view of data. *ACM Transactions on Database Systems*, *1*(1), 9–36. doi:10.1145/320434.320440

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, *13*(6), 377–387. doi:10.1145/362384.362685

Codd, E. F. (1971). A data base sublanguage founded on the relational calculus. *Paper presented at the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*. Academic Press. doi:10.1145/1734714.1734718

Columbus, L. (2018). 10 Charts That Will Change Your Perspective Of Big Data's Growth. *Forbes*.

Comer, D. (1979). Ubiquitous B-tree. *ACM Computing Surveys*, *11*(2), 121–137. doi:10.1145/356770.356776

Davoudian, A., Chen, L., & Liu, M. (2018). A survey on NoSQL stores. *ACM Computing Surveys*, *51*(2), 40. doi:10.1145/3158661

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107–113. doi:10.1145/1327452.1327492

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., . . . Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *Paper presented at the ACM SIGOPS Operating Systems Review*. Academic Press. doi:10.1145/1294261.1294281

Doherty, S. (2013). *The Future Of Enterprise Data: RDBMS will be there*. Wired.

Duggirala, S. (2018). NewSQL Databases and Scalable In-Memory Analytics. *Advances in Computers*, *109*, 49–76. doi:10.1016/bs.adcom.2018.01.004

Engels, G., Gogolla, M., Hohenstein, U., Hülsmann, K., Löhr-Richter, P., Saake, G., & Ehrich, H.-D. (1992). Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering*, *9*(2), 157–204. doi:10.1016/0169-023X(92)90008-Y

Fowler, B., Godin, J., & Geddy, M. (2016). Teaching Case Introduction to NoSQL in a Traditional Database Course. *Journal of Information Systems Education*, *27*(2), 99–103.

Gani, A., Siddiqa, A., Shamshirband, S., & Hanum, F. (2016). A survey on indexing techniques for big data: Taxonomy and performance evaluation. *Knowledge and Information Systems*, *46*(2), 241–284. doi:10.1007/s10115-015-0830-y

Gao, F., Yue, P., Wu, Z., & Zhang, M. (2017). Geospatial data storage based on HBase and MapReduce. *Paper presented at the 2017 6th International Conference on Agro-Geoinformatics*. Academic Press. doi:10.1109/Agro-Geoinformatics.2017.8047040

Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, *33*(2), 51–59. doi:10.1145/564585.564601

Gonzalez-Aparicio, M. T., Younas, M., Tuya, J., & Casado, R. (2019). Evaluation of ACE properties of traditional SQL and NoSQL big data systems. *Paper presented at the Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. Academic Press. doi:10.1145/3297280.3297474

Gray, J., & Reuter, A. (1992). *Transaction processing: concepts and techniques*. Elsevier.

Gudivada, V., Apon, A., & Rao, D. L. (2018). Database systems for big data storage and retrieval. In Handbook of Research on Big Data Storage and Visualization Techniques (pp. 76–100). Hershey, PA: IGI Global. doi:10.4018/978-1-5225-3142-5.ch003

Gudivada, V. N., Rao, D., & Raghavan, V. V. (2016). Renaissance in database management: Navigating the landscape of candidate systems. *Computer*, *49*(4), 31–42. doi:10.1109/MC.2016.115

Harris, S., & Seaborne, A. (2013). sparql 1.1 query language. Recommendation. *World Wide Web Consortium*.

Huang, X., Wang, J., Zhong, Y., Song, S., & Yu, P. S. (2015). Optimizing data partition for scaling out NoSQL cluster. *Concurrency and Computation*, *27*(18), 5793–5809. doi:10.1002/cpe.3643

Kent, W. (1983). A simple guide to five normal forms in relational database theory. *Communications of the ACM*, *26*(2), 120–125. doi:10.1145/358024.358054

Kookarinrat, P., & Temtanapat, Y. (2015). Analysis of range-based key properties for sharded cluster of mongodb. *Paper presented at the 2015 2nd International Conference on Information Science and Security (ICISS)*. Academic Press. doi:10.1109/ICISSEC.2015.7370983

Kumar, P., & Huang, H. H. (2016). G-store: high-performance graph store for trillion-edge processing. *Paper presented at the SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Academic Press. doi:10.1109/SC.2016.70

Kyrola, A., Blelloch, G., & Guestrin, C. (2012). GraphChi: Large-Scale Graph Computation on Just a PC. *Paper presented at the 10th USENIX Symposium on Operating Systems Design and Implementation OSDI 12*. Academic Press.

Lakshman, A., & Malik, P. (2010). Cassandra: A decentralized structured storage system. *Operating Systems Review*, *44*(2), 35–40. doi:10.1145/1773912.1773922

Lei, C., Özcan, F., Quamar, A., Mittal, A. R., Sen, J., Saha, D., & Sankaranarayanan, K. (2018). Ontology-Based Natural Language Query Interfaces for Data Exploration. *IEEE Data Eng. Bull.*, *41*(3), 52–63.

Li, G., Feng, J., Zhou, X., & Wang, J. (2011). Providing built-in keyword search capabilities in RDBMS. *The VLDB Journal*, *20*(1), 1–19. doi:10.1007/s00778-010-0188-4

Libkin, L., Martens, W., & Vrgoč, D. (2016). Querying graphs with data. [JACM]. *Journal of the Association for Computing Machinery*, *63*(2), 14. doi:10.1145/2850413

Martínez-Cruz, C., Noguera, J. M., & Vila, M. A. (2016). Flexible queries on relational databases using fuzzy logic and ontologies. *Information Sciences*, *366*, 150–164. doi:10.1016/j.ins.2016.05.022

Mirrokni, V., Thorup, M., & Zadimoghaddam, M. (2018). Consistent hashing with bounded loads. *Paper presented at the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. Academic Press. doi:10.1137/1.9781611975031.39

Mohan, S. (2018). Teaching NoSQL Databases to Undergraduate Students: A Novel Approach. *Paper presented at the 49th ACM Technical Symposium on Computer Science Education*. Academic Press. doi:10.1145/3159450.3159554

Moldovan, D., Antal, M., Valea, D., Pop, C., Cioara, T., Anghel, I., & Salomie, I. (2015). Tools for mapping ontologies to relational databases: A comparative evaluation. *Paper presented at the 2015 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*. Academic Press. doi:10.1109/ICCP.2015.7312609

Moreno, J., Fernandez, E. B., Fernandez-Medina, E., & Serrano, M. A. (2018). A Security Pattern for Key-Value NoSQL Database Authorization. *Paper presented at the 23rd European Conference on Pattern Languages of Programs*. Academic Press. doi:10.1145/3282308.3282321

Mullins, C. S. (2019). The New World Of Database Technologies. *Paper presented at the Data Summit*. Retrieved from http://conferences.infotoday.com/documents/331/A101_Sherman.pdf

Munir, K., & Anjum, M. S. (2018). The use of ontologies for effective knowledge modelling and information retrieval. *Applied Computing and Informatics*, *14*(2), 116–126. doi:10.1016/j.aci.2017.07.003

Oliveira, F. R., & del Val Cura, L. (2016). Performance evaluation of NoSQL multi-model data stores in polyglot persistence applications. *Paper presented at the Proceedings of the 20th International Database Engineering & Applications Symposium*. Academic Press. doi:10.1145/2938503.2938518

Pirotte, A. (1979). Fundamental and secondary issues in the design of non-procedural relational languages. *Paper presented at the Fifth International Conference on Very Large Data Bases*. Academic Press. doi:10.1109/VLDB.1979.718139

Pritchett, D. (2008). Base: An acid alternative. *Queue*, *6*(3), 48–55. doi:10.1145/1394127.1394128

Qian, C., Childers, B., Huang, L., Guo, H., & Wang, Z. (2018). CGAcc: A Compressed Sparse Row Representation-Based BFS Graph Traversal Accelerator on Hybrid Memory Cube. *Electronics (Basel)*, *7*(11), 307. doi:10.3390/electronics7110307

Reiter, R. (1988). On integrity constraints. *Paper presented at the 2nd Conference on Theoretical Aspects of Reasoning about Knowledge*. Academic Press.

Rudnicki, R., Cox, A. P., Donohue, B., & Jensen, M. (2018). Towards a methodology for lossless data exchange between NoSQL data structures. *Paper presented at the Ground/Air Multisensor Interoperability, Integration, and Networking for Persistent ISR IX*. Academic Press. doi:10.1117/12.2307717

Sadalage, P. J., & Fowler, M. (2013). *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education.

Schall, D., & Härder, T. (2015). Dynamic physiological partitioning on a shared-nothing database cluster. *Paper presented at the 2015 IEEE 31st International Conference on Data Engineering*. Academic Press. doi:10.1109/ICDE.2015.7113359

Seera, N. K., & Taruna, S. (2018). Leveraging MapReduce with Column-Oriented Stores: Study of Solutions and Benefits. In Big Data Analytics (pp. 39-46): Springer.

Simone, S. (2019). Experts Predict Trends for Data Architecture in 2019.

Stanier, C. (2012). Introducing NoSQL into the database curriculum. *Paper presented at the 10th International Workshop on the Teaching, Learning and Assessment of Databases*. Academic Press.

Stonebraker, M., Brown, P., Zhang, D., & Becla, J. (2013). SciDB: A database management system for applications with complex analytics. *Computing in Science & Engineering*, *15*(3), 54–62. doi:10.1109/MCSE.2013.19

Storey, V. C., & Song, I.-Y. (2017). Big data technologies and management: What conceptual modeling can do. *Data & Knowledge Engineering*, *108*, 50–67. doi:10.1016/j.datak.2017.01.001

Teorey, T. J., Yang, D., & Fry, J. P. (1986). A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, *18*(2), 197–222. doi:10.1145/7474.7475

Topi, H., Valacich, J. S., Wright, R. T., Kaiser, K., Nunamaker, J. F. Jr, Sipior, J. C., & de Vreede, G.-J. (2010). IS 2010: Curriculum guidelines for undergraduate degree programs in information systems. *Communications of the Association for Information Systems*, *26*(1), 18.

van der Aalst, W. M. (2018). Responsible Data Science in a Dynamic World. *Paper presented at the IFIP International Internet of Things Conference*. Academic Press.

Van der Vlist, E. (2007). *Schematron*. O'Reilly Media, Inc.

Vaquero, L. M., Rodero-Merino, L., & Buyya, R. (2011). Dynamically scaling applications in the cloud. *Computer Communication Review*, *41*(1), 45–52. doi:10.1145/1925861.1925869

Vogel, W. (2019). Amazon Aurora ascendant: How we designed a cloud-native relational database. All Things Distributed. Retrieved from https://www.allthingsdistributed.com/2019/03/Amazon-Aurora-design-cloud-native-relational-database.html

Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, *52*(1), 40–44. doi:10.1145/1435417.1435432

Walker, E. L., & Slotterbeck, O. A. (2002). Incorporating realistic teamwork into a small college software engineering curriculum. *Journal of Computing Sciences in Colleges*, *17*(6), 115–123.

Wells, J. (2019). Key Database Trends Now and for the Future.

Whitchcock, A. (2005). Google's Big Table. Retrieved from https://web.archive.org/web/20060616203323/http://andrewhitchcock.org/?post=214

Wiese, L., Waage, T., & Brenner, M. (2019). CloudDBGuard: A framework for encrypted data storage in NoSQL wide column stores. *Data & Knowledge Engineering*. doi:10.1016/j.datak.2019.101732

Wu, X., Xu, Y., Shao, Z., & Jiang, S. (2015). LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. *Paper presented at the 2015 USENIX Annual Technical Conference (USENIX 15)*. Academic Press.

Xu, Q., Siyamwala, H., Ghosh, M., Suri, T., Awasthi, M., Guz, Z., . . . Balakrishnan, V. (2015). Performance analysis of NVMe SSDs and their implication on real world databases. *Paper presented at the 8th ACM International Systems and Storage Conference*. ACM Press. doi:10.1145/2757667.2757684

Akhilesh Bajaj is a Chapman Professor of Computer Information Systems at the University of Tulsa. He has a B Tech (Chem. E.) from the Indian Institute of Technology, MBA from Cornell University and a PhD from the University of Arizona. He has published in several academic journals including Management Science, Journal of Association of Information Systems, IEEE Transactions on Knowledge and Data Engineering, Journal of Information Systems and Information Systems. He is on the editorial board of several IS journals and has taught courses from the undergraduate to the Executive level on database design and development, web programming, and IS strategy.

Wade Bick earned a degree in electrical engineering at the University of Southern California and has worked at Teradyne for over 25 years. Positions included memory applications, ASIC test, systems engineering, and software verification. He is currently enrolled in the Master's of Data Analytics Program at the University of Tulsa. Research interests include databases, analytics, and neural networks. Outside of academics, he spends time with his family, enjoys wood working and officiates swim meets for USA Swimming.