EVALUATION CRITERIA FOR SELECTING NOSQL DATABASES IN A SINGLE-BOX ENVIRONMENT

Ryan D. L. Engle, Brent T. Langhals, Michael R. Grimaila, and Douglas D. Hodson

Air Force Institute of Technology, Wright-Patterson Air Force Base, OH 45433 USA

ABSTRACT

In recent years, NoSQL database systems have become increasingly popular, especially for big data, commercial applications. These systems were designed to overcome the scaling and flexibility limitations plaguing traditional relational database management systems (RDBMSs). Given NoSQL database systems have been typically implemented in large-scale distributed environments serving large numbers of simultaneous users across potentially thousands of geographically separated devices, little consideration has been given to evaluating their value within single-box environments. It is postulated some of the inherent traits of each NoSQL database type may be useful, perhaps even preferable, regardless of scale. Thus, this paper proposes criteria conceived to evaluate the usefulness of NoSQL systems in small-scale single-box environments. Specifically, key value, document, column family, and graph database are discussed with respect to the ability of each to provide CRUD transactions in a single-box environment.

KEYWORDS

Data storage systems, databases, database systems, relational databases, data models, NoSQL

1. INTRODUCTION

Over the past decades, a class of database systems, generally termed NoSQL, have emerged as popular alternatives to traditional Relational Database Management Systems (RDBMSs) for modern, distributed data storage and retrieval applications. These new databases were designed to manage the volume, variety, and velocity commonly associated with Big Data applications. Specifically, NoSQL was conceived to serve large numbers of simultaneous users across numerous distributed systems while handling all types of structured, semi-structured, and unstructured data [1] [2] [3]. Traditional relational systems have coped poorly with these challenges. Yet, despite the focus on Big Data distributed environments, not all modern data storage and retrieval applications have such requirements. Personal, educational, and small business databases, as an example, may be much smaller in volume and housed on single devices, yet still require high speed transactions, the ability to store and manipulate a variety of data types, all the while supporting many simultaneous users. Often relational-based solutions are chosen simply because they are familiar to the user or already "in place" while NoSQL systems are dismissed because they are considered useful only for addressing large scale problems on distributed platforms. It is, therefore, not surprising little research has been dedicated toward examining the applicability of NoSQL systems for single box environments. This study aims to start a discussion about how, when and which types of NoSQL databases can excel for small scale applications.

For this study, a single box environment is defined as a single computer system (mobile device, laptop, personal computer, workstation, or server) serving one to multiple simultaneous users and may be connected to a network. The box may contain multiple processors, cores, and hard disks, and terabytes (or more) of RAM scaling up to the top performance single systems available. In contrast, a system that is distributed across multiple motherboards (i.e. a server rack) is not, for the

purposes of this study, considered a single box. Additionally, this paper considers only the baseline capabilities of key-value, document, column family, and graph data stores, not specific commercial or open sourced implementations of each. The following sections review the defining characteristics of each NoSQL type and then proposes criteria for evaluating the merits of each with respect to operations in a single box environment.

2. BACKGROUND

In order to effectively evaluate NoSQL database usefulness for small scale, single box solutions, it is important to first identify the characteristics which still apply outside of distributed environments. NoSQL systems encompass a variety of data systems outside of Codd's relational data model for storing and retrieving data [4]. The acronym *NoSQL* has been defined as both "No SQL," referring to the exclusion of a relational model and SQL, and "Not Only SQL," implying that some implementations may include relational elements under the hood. In an attempt to describe NoSQL more formally, Cattell identified six key features of NoSQL systems: 1) horizontal scalability, 2) replication across many servers, 3) simple interface or protocol, 4) "weaker" transaction concurrency than RDBs, 5) efficient storage using RAM and distributed indexes, and 6) the flexibility to add additional attributes to existing data records [3] [5]. However, Cattell's definition is biased toward expectations of distributed, multi-box implementation as his NoSQL characteristics of horizontal scalability, replication, and to a lesser extent, "weaker" concurrency transactions are not relevant to the single box solutions.

One popular way to characterize NoSQL has been to examine its approach to satisfying Brewer's Consistency, Availability, and Partition tolerance (CAP) theorem [6] [7]. Most NoSQL systems have been designed to sacrifice consistency in exchange for high availability in a partitioned environment [3] [8]. However, in the context of a single box, applying the CAP theorem provides little value. For instance, if the system is online, it is presumed to be available. If the system is available, only one consistent "view" is presented to the users. It is similarly difficult to evaluate single-box NoSQL databases based solely on ability to ensure Atomic, Consistent, Isolated, and Durable (ACID) transaction concurrency. As suggested by both Cattell and Brewer, NoSQL systems balance concurrency in distributed systems through an approach known as Basically Available, Soft state, and Eventually consistent (BASE). In short, the BASE approach provides no guarantees that the distributed database will be available from all access points or that it will provide a consistent data view to its users at a given time [9] [10]. However, for the same reasons as the CAP theorem, the BASE concurrency model has little applicability when discussing NoSQL within a single box environment. For single box solutions, it can be argued that BASE effectively equals ACID.

Perhaps, then, the defining characteristic for NoSQL data stores is the use of aggregate stores or an aggregate-oriented model. The aggregate concept is a helpful way to contrast NoSQL database types with each other as well as with the relational databases and implies a certain level of knowledge exists regarding what data is stored and how it will be retrieved. An aggregate is formally defined as a composite data object that is considered the atomic unit for Create, Read, Update, and Delete (CRUD) operations. This concept originated with databases and software development in the 1970s and is also related to Evans's recent work with domain driven design. In the NoSQL context, aggregates may vary widely in size and composition, ranging from individual binary values representing status flags to MPEG video files and their associated metadata. Treating data as aggregates enables data stores to take advantage of locality and denormalization to improve data retrieval performance [11] [12] [13] [14].

Empowering the aggregate model concept is the ability of NoSQL to accept data without any prerequisite data modeling, unlike relational databases where a schema or predefined logical view of the database must exist before data can be imported [14] [15]. Thus, the NoSQL database structure, i.e. the physical aggregate, emerges as data is added [16] [17]. Additionally, NoSQL

databases excel at storing and retrieving semi-structured and unstructured data which do not fit well inside a traditional RDB. Unstructured data may include plain text format or multimedia, while semi-structured data may be organized into JSON or CSV formats. Both JSON and CSV formats impose some structure on data, but the contents can vary. Semi-structured data is also referred to as having a hybrid structure. RDBs primarily operate on structured data, which is data that is easily organized into a rectangular table and normalized. In contrast, NoSQL databases can store and retrieve all data types efficiently [2] [3] [1] [18] [19] [20] [21].

The following subsections describe the baseline capabilities of four NoSQL database types. The capabilities described are considered baseline because they are not specific to any particular implementation. That is, the capabilities and attributes described are expected to be common to the applicable database type. Unless indicated, the defining NoSQL characteristics outlined thus far are assumed to apply to each database type.

2.1. Key Value Databases

Key-value data models store and retrieve data as key-value pairs. The key is a unique identifier and the value is the data associated with the key. These pairs are similar to maps, dictionaries, and associative arrays which use non-integer indexing to organize data. In this data model, a value composes the aggregate. Additionally, aggregates are isolated and independent from each other. Thus, no relationships are stored in this data model. Furthermore, few limitations are placed on what data types can be stored as values. Values may contain strings, integers, maps, lists, sets, hashes, queues, Binary Large Objects (BLOBs), or a composite object of these types [22] [1] [13].

KV databases treat aggregates as opaque atomic units once they are imported. "Opaque" means the database has little knowledge about what comprises the value stored or how it is structured. However, this feature provides for great flexibility in storage, simplicity for querying, and shifts responsibility for data integrity outside of the database. Additionally, KV databases generally do not include a complex query processor. CRUD operations are accomplished using put, get, and delete operations. Thus, complex queries must be handled at the application layer outside the database [1].

2.2. Document Databases

The document model is in many ways similar to the KV model. Document models organize and store data in a document structure consisting of a set of key-value pairs. More formally, a document is a self-describing, i.e., key-value pairs, hierarchical tree data structure consisting of scalar values, maps, lists, other documents, and collections. A collection is a group of documents and often pertains to a particular subject entity. The aggregate is the document in this model. The inclusion of keys in the aggregate provides the self-describing aspect of this object [3] [13] [23]. Also like the KV model, most data types can be stored in a document model including Boolean values, integers, arrays, strings, dates, and BLOBs among others. Additionally, document models employ a unique identifier to distinguish individual, top-level documents. While a document is similar in concept to a row in a relational database, it does not natively store relationships between documents with the exception of nested documents [3] [13] [23].

A few more aspects of the document model differ from the KV model. Document models provide aggregate transparency which enables access during read, update and delete operations to attributes/data elements stored within an aggregate. This characteristic is unlike the opaque nature of KV models. Additionally, document stores typically include a query processor that can perform complex queries such as searching for a range of values, accessing keys within a document, or handling conditional query statements like those common to SQL. Yet, like a KV model, responsibility for data integrity and any relational consistency is placed outside the database itself. Furthermore, document models often include indexing to speed up searches. Lastly, attributes can be added to existing documents [3] [13] [23].

2.3. Column Family Databases

The column family, or column-oriented, model organizes data into a multidimensional map based of the Decomposition Storage Model (DSM). Originally the DSM organized data into columns which were associated by a unique identifier known as a surrogate. In this model, the column is the basic storage unit and composed of a name and a value, much like a key-value pair. For column family databases, the aggregates consist of columns assembled together and are referred to as column families. Rows are composed of a unique key and one or more columns and/or column families [24] [25] [1] [13] [23].

Though the terminology is similar to the relational model, a row in this model is actually a two-level map.
Figure **1** presents an example consisting of two rows, to illustrate the two-level map properties of the column family database. The first row, row key "1235," contains a single column consisting of a column key designated as "name" and its value is "Grad Student A." The second row, row key"1236," contains a slightly more complex column family. In the column family, there are four keys,"first Author Name," "second Author Name," "third Author Name," and "fourth Author Name" and four associated values, "Grad Student A," "Professor X," "Professor Y," and "Professor Z."

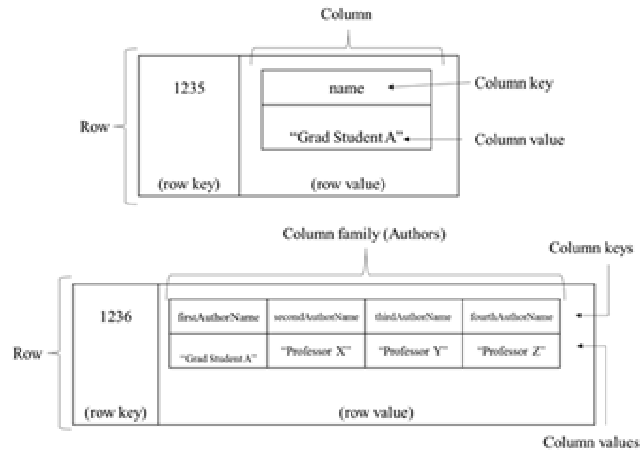

Figure 1. Examples of two rows in a column-oriented database.

In this model, the row value is the aggregate. Additionally, column family models provide aggregate transparency, like the document model, to provide access to individual columns within the aggregate. Furthermore, columns can be added, updated, or excluded from rows without updating a predefined schema. However, column families usually must be defined before they are used. Finally, column family databases often include a query processor to facilitate searching and retrieval [26] [1] [27] [13] [23].

2.4. Graph Model Database

Property graph models are common implementations of the more general graph model. Property graph models store and retrieve data using two primary modeling objects: nodes and edges. A node represents an entity and stores any attributes as properties. Likewise, an edge represents a relationship between one or two nodes. Edges have an associated direction between nodes and may also include properties. Properties for either nodes are edges are stored as key-value pairs. [1] [13] [16].

Graph models share many characteristics with other NoSQL data models, but have some unique traits. Graph models support most primitive data types such as Boolean, byte, short, int, long, float, double, and char types. Naturally, arrays of these primitives are also permitted. Storage of BLOBs are permitted, but are not as well suited for the graph model. Graph models are said to be relationship oriented and most appropriate for heavily linked data. Additionally, they are unique from the other NoSQL data models because there are two elements (nodes and edges) which can comprise an aggregate. Furthermore, nodes and edges may be defined in a schema or added as necessary [1] [13] [16].

Graph models are considered to represent relationships more naturally than other data models. Graphs can overcome the impedance mismatch commonly found in relational models of data objects. The object-relational impedance mismatch describes the difference between the logical data model and the tabular-based relational model resulting in relational databases. This mismatch has long created confusion between technical and business domains [1] [16].

Figure 2. Example of a property graph model using authors and their relationships.

Figure 2 presents an example of a property graph depicting a few relationships between the authors and their associated departments. In this figure, two kinds of nodes, “Author” and “Department,” and nine relationships are displayed. Each “Author” node has two properties: “name” and “title.” Similarly, the “Department” nodes contain one property called “name.” The values for each property are shown in the diagram. Relationships are interpreted by starting with one node and following a directed edge to its related node. For example, the Author named “Professor X” ADVISES the Author named “Grad Student A.” Additionally, the Author named “Professor X” WORKS IN the Department named “Systems Engineering & Management”. The Author’s “duties” in this relationship are to “Perform[s] research” and “teach[es] courses.” The other relationships are interpreted using the same method.

Understanding the how each NoSQL database type operates is the first step toward evaluating the merits of each in a single box environment. However, consistent evaluation criteria must be applied to enable the correct selection of NoSQL type. The following section proposes evaluation criteria designed to evaluate and select the best NoSQL database for use in a single box environment.

3. SINGLE BOX NOSQL EVALUATION CRITERIA

This section describes criteria by which the merits of the four NoSQL database types can be examined. The criteria presented in figure 3 were derived from NoSQL database traits that remain relevant for a single box environment. Thus, many of the typical NoSQL characteristics associated

with large scale data and applications are excluded. For example, horizontal scalability was not considered because this characteristic pertains to large-scale, distributed systems only. Additionally, the identified criteria focus on data storage and retrieval operations and mechanisms available within the databases themselves. That is, the database type does not rely on an external application to provide the feature. Moreover, to provide clarifying examples of how the criteria are to be used, a database system designed for Unmanned Aircraft Systems (UAS) log data is discussed. For this UAS, log data is generated by the periodic sampling of various subsystem status variables. In this context, an *aggregate* is a collection of all the variables of interest sampled at a given moment in time. "Of interest" refers to the variables intended to be stored, retrieved, or updated in the DBMS. Likewise, an *element* would be a single sample of a variable, such as engine speed, altitude, or heading. Furthermore, a *transaction* is a CRUD operation performed on a single aggregate and a *query* is a set of transactions executed on one or more aggregates and/or elements.

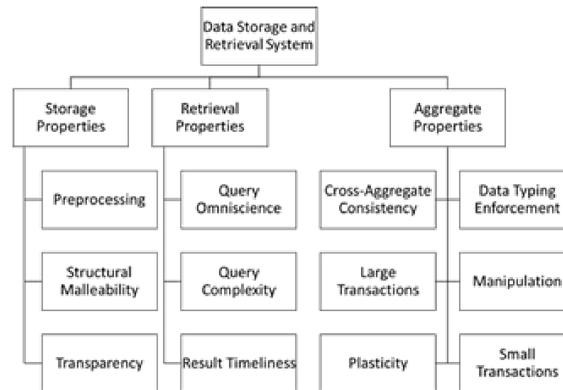

FIGURE 3: NoSQL Evaluation Criteria

The proposed evaluation criteria are logically grouped by the type of operations they influence. First, *Storage Properties* refers to characteristics of NoSQL databases that affect either extent to which data must be manipulated prior to storage or the ability to create/delete aggregates and then view them once created. The second characteristic, *Retrieval Properties*, captures how well each database type can return stored data. The power of the NoSQL aggregate model stems from storing related data physically together, allowing efficient retrieval. Implicitly then, certain aspects of the stored data and more importantly how it is expected to be returned must be known a priori. For retrieval operations, the characteristics of interest include query complexity, the amount of a priori knowledge known, and the timelines required for query returns. The last set evaluation criteria involve properties of the aggregates generated by each NoSQL database type. They include issues of consistency, data typing, ability to handle varied transaction size, and ability to update and manipulate each aggregate. Additionally, this set includes criteria involving properties which cannot be exclusively categorized as storage or retrieval that is some of these criteria incorporate aspects related to both storage and retrieval. The following sections discuss each of these criteria in greater detail and outline the relative importance for each.

3.1. Storage Properties

The first storage property is *preprocessing* and includes all preprocessing and associated operations required to import/load data into the DBMS. Examples of preprocessing may include: deciding/selecting which variables to store in the DBMS, filtering the selected variables from the raw data, transforming filtered variables into a format suitable for loading, and loading transformed data into the DBMS. When evaluating this criterion, one should consider the ability to perform preparation work before data can be loaded into the DBMS. If unwilling (or unable), then this criterion is important because the DBMS is expected/required to facilitate data preparation,

preprocessing, or loading of raw data. However, if the user is willing and able to handle the preprocessing, then this criterion is less important because the DBMS is not expected/required to facilitate this process.

The second storage property is *structural malleability* which refers to the DBMS's ability to add/remove "types" of aggregates to and from the DBMS. For example, aggregate types for a UAS database could include: engine subsystem, pilot inputs, aircraft telemetry, and others. If, perhaps, a new sensor system was added to the UAS, a new aggregate type might need to be added to the DBMS to store and organize data for this system. When considering this criterion for a UAS database, it would be important to think about the likelihood a new subsystem (thus new aggregate) would be added to or removed from the aircraft. If likely, then this criterion is important because a solution to support aggregate type changes is needed. If not, then this criterion is less important.

The final storage property is *transparency.* In this case, transparency describes the DBMS's ability to store aggregates such that individual elements within the aggregate can be viewed and retrieved during read transactions. For example, assume a DBMS is designed using an aggregate model that contains all variables for a sample in time. Thus, engine speed, altitude, landing gear, tail number, timing information, etc. are stored in each aggregate. When a DBMS supports transparency, it is possible to search for and retrieve engine speed from an aggregate. Without transparency, searches are limited to retrieving the entire aggregate, not the specific value for engine speed. Users should consider if they care about retrieving individual elements, such as engine speed, from the data. If so, then this criterion is important because there is a need to access individual data elements within aggregates. If not, then this criterion is less important.

3.2. Retrieval Properties

The first retrieval property criterion is *query complexity* which describes a DBMS's ability to perform both simple and complex queries. A simple query retrieves data using only a unique identifier or a range of identifiers and/or values. Complex queries involve additional conditions enforced on the operations. For example, a simple query might ask 1) if the engine speed ever exceeded 5500 RPM or 2) if the engine speed exceeded 5500 RPM on the mission flown today. An example complex query, however, may ask 1) if the engine fan drew more than 33 Amps while the cooling fan was set to auto or 2) what is the average coolant temperature during a mission for each aircraft. Users should consider whether they need the database to perform sorting, grouping, mathematical calculations, and conditional searches on their data sets. If yes, query complexity is important. If not, query complexity is less important.

The second retrieval criterion is *query omniscience* and it refers to the degree to which the complete set of possible queries is known by the user before system is implemented. For example, if the user knows every question that will ever be asked about the data and does not anticipate unexpected events or circumstances will require unique investigations, then this criterion is important because the user will not need the DBMS to support additional queries/questions in the future. However, if the possibility exists for new unanticipated queries to be developed or for existing queries to be changed, then this criterion is not as important.

Finally, *result timeliness* is the last retrieval criterion. Result timeliness refers to how quickly the results are provided to the user after a request is made. For example, a query is described, run, and produces results within 10 seconds of starting to run. If the user expects results within seconds or less, then result timeliness is important. However, if the user is willing to wait minutes or longer for a result, then result timeliness is less important.

3.3. Aggregate Properties

The first evaluation criterion for aggregate properties is *cross-aggregate consistency*. This criterion refers to the DBMS's ability to perform cascading updates to data and relationships. Assume a UAS was flown on multiple distinct flights and the log data and tail number for each flight is loaded into

a DBMS as part of several different aggregates. Additionally, the DBMS stores the relationship between the UAS and each flight as part of these different aggregates. Under such a scenario, queries can determine which flights were flown by a particular UAS as well as any associated log data by looking for tail numbers across the aggregates. Queries can also retrieve/update properties about relationships such as the start/end date and time of all missions flown by a particular UAS using its unique tail number. Later, someone realizes the wrong tail number was recorded for a given flight, resulting in an update which affects all aggregates containing data from the affected UAS. Cross-aggregate consistency involves the process responsible for updating each aggregate with the appropriate tail number. When updates to stored data are required, cross-aggregate consistency is important, because the DBMS is expected to enforce consistency among data and stored relationships. If the user can tolerate some inconsistencies or cross aggregate consistency can be managed in another way, then cross aggregate consistency is less important.

Next, *data typing enforcement* describes the extent a DBMS applies schema enforcement for data typing during transactions. For example, flap angle is recorded in degrees with floating-point precision by the UAS. Assume an acceptable reading is 5.045 degrees. In contrast, landing gear status is recorded as either up or down (Boolean: 1 or 0). Data typing enforcement ensures flap angle is stored in and retrieved from the DBMS as 5.045 rather than being rounded to an integer value of 5. Likewise, landing gear status is stored and retrieved appropriately as either a 0 or 1. These data types (float and Boolean) can be specified in schemas and enforced by some DBMSs. Users should consider the data type and precision requirements. If the user's elements have well-known types and precision and the user wishes to ensure they are maintained within the DBMS, then data typing enforcement is important. If not, it is less important.

The third aggregate evaluation criterion is *large transactions* which refers to the DBMS ability to store, retrieve, and update large aggregates quickly (within a few seconds). For this criterion, a large aggregate is defined as being larger than 1 terabyte (TB) in size. Assume for a UAS example, the combined size of all the variables of interest collected in a sampling period is 1.5 TB. In other words, an aggregate would be 1.5 TB. In this situation, DBMS support for large aggregate transactions is important. If not, then this criterion is less important.

Conversely, the fourth aggregate evaluation criterion is *small transactions* which describes a DBMS's ability to store, retrieve, and update small aggregates quickly. A small aggregate is defined for this criterion as being smaller than 1 kilobyte (kB). In the UAS context, if the combined size of all the sampled variables is 800 bytes, then the resultant aggregate would be 800 bytes. In this situation, small aggregate transactions are important, because the resultant aggregate is considered small (<1kB). If the resultant aggregate was larger than 1 kB, then the small transaction performance would be less important. It should be noted that users may have the need for both large and small aggregate transactions. Additionally, some DBMS excel only at large scale, while others perform well at both.

The fifth evaluation criterion is *manipulation*. Manipulation refers to the DBMS's ability to update elements within stored aggregates independently from other aggregates and elements. This behavior may be desirable depending on what relationships exist between stored aggregates. For example, assume a UAS flight occurred and the log data is loaded into a DBMS. Later it is discovered that the timing signal was initially corrupt resulting in 10 aggregates containing incorrect timing information. Since subsequent timing information was accurate, it can be used to calculate the timing information for the corrupted periods. To update this information in the DBMS, the appropriate timing element within the affected aggregates must be changed. The manipulation property enables these update operations to occur independently without the DBMS automatically performing cascading update to all aggregates. This ability to independently perform updates distinguishes manipulation from cross aggregate consistency. Users should consider whether updates to elements, such as timing, in existing aggregates are likely. If so, manipulation is important for the database solution to provide. If not, this ability is less important than others.

Finally, *plasticity* pertains to the DBMS's ability to add or remove elements within stored aggregates. For example, assume a UAS flight took place and the log data is loaded into a DBMS. Adequate GPS data was collected for the entire flight. Now the user wishes to use the existing GPS timing information to calculate the UTC time for each sample and store the result as a new element with the aggregate. The plasticity property enables the DBMS to add a "Calculated UTC time" element to each aggregate. Similarly, assume a UAS flight occurred and the log data is loaded into a DBMS. Later it is determined that the *Engine Speed* element was included but contained corrupt data for this mission. The user wishes to remove this element from all of the aggregates for the flight. Plasticity enables the DBMS to remove elements from existing aggregates. To evaluate the need for plasticity, users should considerations whether the need exists for the DBMS to support adding or removing elements in existing aggregates. If the need exists, plasticity is important. Otherwise, it is not as important for the database to provide this ability.

The proposed evaluation criteria were designed to help a user assess the performance characteristics required for a given application against the general strengths and weaknesses of each NoSQL database type when constrained to a single box environment. Of course, each implementation of a NoSQL database type, e.g., Mongo, Couch, and Azure Document DB, may possess slightly different abilities. However, it is expected the evaluation criteria would point the assessor toward the most suitable NoSQL database type, from which a specific application can be chosen. At the very least, the criteria could be used to rule out incompatible NoSQL databases. The following section explains how the criteria may be used.

4. Evaluation Criteria Usage

Based upon the descriptions of each NoSQL database type, it is believed certain inherent capabilities remain relevant for small scale, single box solutions. To evaluate each databases strengths and weaknesses, 12 evaluation criteria were developed to objectively compare each general database type. For each evaluation criteria discussed, **Error! Reference source not found.** provides a notional matrix of how well each NoSQL database type supports each criterion. An assessment of Good, Fair, or Poor was recorded for each NoSQL database type, however these are only a representative assessment and could have just as easily consisted of other distinct descriptors such as values from one to three or other similar labels. The important aspect of the matrix is to evaluate the ability of each database type against the evaluation criteria and develop a matrix like **Error! Reference source not found.**. From there, users need to assess the relative importance of each criteria for the application they intend use. For example, users who value Cross-Aggregate consistency would (according to **Error! Reference source not found.**) consider Graph databases to be good, but Key Value, Document, and Column databases as poor. Each evaluation criterion is assessed in a similar manner.

Table 1. Criteria List and Associated NoSQL Database Type

	KV	Document	Column	Graph
Preprocessing	Fair	Fair	Fair	Fair
Structural Malleability	Fair	Good	Fair	Good
Transparency	Poor	Good	Good	Good
Query Omniscience	Poor	Fair	Poor	Fair
Query Complexity	Poor	Good	Fair	Good
Result Timeliness	Good	Good	Fair	Good
Cross-Aggregate Consistency	Poor	Poor	Poor	Good
Data Typing Enforcement	Poor	Fair	Poor	Fair
Large Aggregate Transactions	Fair	Fair	Fair	Poor
Small Aggregate Transactions	Good	Good	Fair	Good
Manipulation	Poor	Good	Good	Good
Plasticity	Poor	Good	Good	Good

The following is a hypothetical example of how UAS users may use the evaluation criteria to help select a NoSQL database type. Developers of a UAS database may consider Structural Malleability, Query Complexity, Large Aggregate Transactions, and Manipulation as the most important criteria. Based on the matrix and user defined needs, a Document database may be the best choice as three of the four evaluation criteria are "Good" with the fourth being rated "Fair". In comparison, Key Value databases would not likely be a good choice given it performs poorly for three of the four evaluation criteria. As this hypothetical example suggests, by using the evaluation criteria, certain NoSQL databases can be quickly ruled out and perhaps, depending upon the application, a possible solution is presented.

4. Conclusions

Ever since Codd formally defined the relational model in the 1970s, relational databases and its associated powerful SQL code have been the dominant standard by which data storage and retrieval have been measured. With the advent of modern, Big Data requirements, the relational model has been augmented with a variety of NoSQL type database options. However, for smaller, non-distributed applications, relational databases frequently remain the preferred database system. In many cases this is a prudent choice, but perhaps not always. In certain circumstances, the inherent advantages of NoSQL databases may yield better results regardless of data volume or scale. This paper proposed a set of criteria for evaluating the effectiveness of NoSQL options in a limited, single box environment. It is expected future work would test these criteria and refine as needed. In the end, the goal is to effectively employ the right technology to match the applications data needs.

Acknowledgements

This work was supported in part by the Air Force Office of Scientific Research under grant 18RT0095 in support of Dynamic Data Driven Application Systems, PI: Dr. Erik Blasch.

REFERENCES

[1] R. Hecht and S. Jablonski, "NoSQL evaluation: A use case oriented survey," in International Conference on Cloud and Service Computing (CSC), 2011.

[2] N. Leavitt, "Will NoSQL Databases Live Up to Their Promise?," Computer, pp. 12-14, 25 01 2010.

[3] R. Cattell, "Scalable SQL and NoSQL data stores," ACM SIGMOD Record, vol. 39, no. 4, pp. 12-27, 2011.

[4] E. F. Codd, "A relational model of data for large shared data banks," Communications of the ACM, pp. 377-387, 1970.

[5] M. Stonebraker, "The Case for Shared Nothing," IEEE Database Enginerring Bulletin, vol. 9, no. 1, pp. 4-9, 1986.

[6] E. Brewer, "Towards robust distributed systems," in Principles of Distributed Computing (PODC), 2000.

[7] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," ACM SIGACT News, vol. 33, no. 2, pp. 51-59, 2002.

[8] D. J. Abadi, "Consistency tradeoffs in modern distributed database system design," Computer-IEEE Computer Magazine, pp. 37-42, 02 2012.

[9] D. Pritchett, "BASE: An ACID alternative," Queue, vol. 6, no. 3, pp. 48-55, 2008.

[10] K. Orend, "Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace and Object-relational Persistence Layer," Technische Universitat Munchen, 2010.

[11] J. M. Smith and D. C. Smith, "Database abstractions: aggregation and generalization," ACM Transactions on Database Systems (TODS), pp. 105-133, 1977.

[12] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software Engineering, Westford: Pearson Education, Inc., 2011.

[13] P. J. Sadalage and M. Fowler, NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, Upper Saddle River: Addison-Wesley, 2013.

[14] R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, Boston: Pearson, 2016.

[15] D. Bradbury, "Database Schema vs. Data Structure," 11 03 2017. [Online]. Available: https://stackoverflow.com/questions/42738719/database-schema-vs-data-structure. [Accessed 10 2017].

[16] I. Robinson, J. Webber and E. Eifrem, Graph Databases: New Opportunities for Connected Data, Sebastopol: O'Reilly Media, Inc., 2015.

[17] J. A. Hoffer, R. Venkataraman and H. Topi, Modern Database Management, Pearson, 2016.

[18] Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," in 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), 2013.

[19] A. Joshi, Julia for Data Science, Birmingham: Packt Publishing, 2016.

[20] A. Siddiqa, A. Karim and A. Gani, "Big Data Storage Technologies: A Survey," Frontiers of Information Technology & Electronic Engineering, pp. 1040-1070, 2017.

[21] A. Almassabi, O. Bawazeer and S. Adam., "Top NewSQL Databases and Features Classification," International Journal of Database Management Systems (IJDMS) Vol.10, No.2, April 2018

[22] V. N. Gudivada, D. Rao and V. V. Raghavan, "NoSQL systems for big data management," in 2014 IEEE World Congress on Service (SERVICES), 2014. [23] D. Sullivan, NoSQL for Mere Mortals, Ann Arbor: Addison-Wesley, 2015.

[24] G. P. Copeland and S. N. Khoshafian, "A Decomposition Storage Model," ACM SIGMOD Record, pp. 268-279, 1985.

[25] D. J. Abadi, P. A. Boncz and S. Harizopoulos, "Column-oriented database systems," in Proceedings of the VLDB Endowment, 2009.

[26] D. J. Abadi, "Query execution in column-oriented database systems (Doctoral dissertation)," Massachusetts Institute of Technology, Boston, 2008.

[27] D. J. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos and S. Madden, "The design and implementation of modern column-oriented database systems," Foundations and Trends in Databases, vol. 5, no. 3, pp. 197-280, 2013.

AUTHORS

RYAN D. L. ENGLE was born in Normal, IL, USA in 1979. He earned a B.S. in computer engineering from Wright State University, Dayton, OH in 2007, a M.S. in systems engineering from the Air Force Institute of Technology (AFIT), Dayton OH, in 2015. He is currently a Ph.D. candidate at AFIT. He is commissioned as a Major in the United States Air Force and served as a Computer Developmental Engineer at Wright-Patterson Air Force Base (AFB) in Dayton, OH, a Systems Engineer at Lackland AFB in San Antonio, TX, and a Test Engineer at Eglin AFB, FL. He holds Department of Defense certifications in Systems Engineering, Test & Evaluation, and Program Management.

BRENT T. LANGHALS, Ph.D., is an Assistant Professor of Information Resource Management at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in History from the United States Air Force Academy in 1995 and an M.S. in Information Resource Management in 2001 from AFIT. He completed his Ph.D. in Management Information Systems at the University of Arizona in 2011. His research interests include database, data analytics, human systems integration and vigilance.

MICHAEL R. GRIMAILA, PhD, CISM, CISSP (BS 1993, MS 1995, PhD 1999 Texas A&M) is Professor and Head of Systems Engineering and Management department at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio, USA. He is a member of the Center for Cyberspace Research and consults for various Department of Defense organizations. He holds the Certified Information Security Manager (CISM), the Certified Information Systems Security Professional (CISSP), and the National Security Agency's INFOSEC Assessment Methodology (IAM) and INFOSEC Evaluation Methodology (IEM) certifications. His research interests include computer engineering, computer and network security, data analytics, mission assurance, quantum communications and cryptography, and systems security engineering. He can be contacted via email at Michael.Grimaila@afit.edu.

DOUGLAS D. HODSON, Ph.D., is an Associate Professor of Computer Engineering at the Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio USA. He received a B.S. in Physics from Wright State University in 1985, and both an M.S. in Electro-Optics in 1987 and an M.B.A. in 1999 from the University of Dayton. He completed his Ph.D. at the AFIT in 2009. His research interests include computer engineering, software engineering, real-time distributed simulation, and quantum communications.