Performance Optimizations of NoSQL Databases in Distributed Systems

Tristyn Maalouf

A Senior Thesis submitted in partial fulfillment
of the requirements for graduation
in the Honors Program
Liberty University
Spring 2020

Acceptance of Senior Honors Thesis

This Senior Honors Thesis is accepted in partial
fulfillment of the requirements for graduation from the
Honors Program of Liberty University.

_____
Dr. Melesa Poole, Ph.D.
Thesis Chair

_____
Dr. Robert Tucker, Ph.D.
Committee Member

_____
Dr. David Schweitzer, Ph.D.
Assistant Honors Director

_____
Date

Abstract

Databases store information about a system and provide a mechanism for data to be accessed and manipulated. While advancements in the 1970s provided a relational database model that has persisted to this day, web-scale era mass data needs surfacing in the 1990s and the early 2000s revealed limitations in the scalability of the relational model. As systems grew and transitioned into distributed architectures to support mass data storage and parallel processing, a complete overhaul of distributed computing technologies evolved that fundamentally departed from the relational data model in favor of the NoSQL data model. The course of this research details the scaling problems encountered by relational databases and the NoSQL solutions that made web-scale systems possible.

Keywords: SQL, NoSQL, horizontal scaling, distributed systems

Performance Optimizations of NoSQL Databases in Distributed Systems

Introduction

A database comprises the methodology and implementation for which data are stored, structured, and manipulated in a system. Most often, databases are implemented when data must be persistent, that is, must remain available even when the process that created the data has ended. Historically this has been done using a relational database management system, or an RDBMS. The relational database model utilizes a spreadsheet-like approach to organizing data by storing it into tables and includes a straightforward querying language called SQL, or Structured Querying Language, to access and manipulate the data within a database. The relational database model became widely known and used beginning in the late 1970s and 1980s when the technology matured and was commercialized by some of the world's largest technology companies like Oracle (Tiwari, 2011). However, this development predates the invention of the World Wide Web, created by Tim Berners-Lee in 1990, which marked the beginning of the web-scale era where applications began to be simultaneously accessed by millions of users around the world (Andrews, 2013).

While RDBMSs had served and continue to serve small and medium size applications extremely well, the unforeseen rates of data generation brought about during the transition into web-scale systems led to challenges with the relational model, namely, datasets that could not be physically stored on a single server and instead had to be distributed across many servers. This can be quantified by some of the world's largest technology companies such as Google and Amazon, which respectively had 900,000 and 450,000 servers running across their worldwide server farms in 2013 ("Facts and Stats," 2013). Fast forward three years to 2016 and Gartner

estimates placed the Google server count at 2.5 million, almost three times their 2013 count (Google Data Center FAQ, 2017). As data generation rates skyrocketed in the early 2000s, companies like Google and Amazon realized that the relational database model would not be able to scale horizontally across thousands of servers to support billions of users as relational databases tend to favor vertical scaling, or increasing the computational power on a small number of servers as needs increase. This spurred a period of research that ultimately resulted in the rise of NoSQL databases, which can stand for either *Not only SQL* or *No SQL* to indicate that some departure has been made from the relational model, either in part or in whole (Foote, 2018).

The following research describes the history of SQL and NoSQL databases, each model's core architecture and properties, and why the properties of SQL limit its horizontal scalability while the NoSQL model thrives in a distributed, horizontally scaled system. The terms SQL model and relational database model will be used interchangeably throughout the course of this research.

History of SQL and NoSQL Technologies

To fully understand NoSQL, it is important to first discuss the problem that led to its rise in popularity and use. As previously mentioned, the use of SQL and the relational database model emerged in the late 1970s. This was due to the combined research efforts of Ray Boyce, Ted Codd, and Donald Chamberlin, whose work became well-known after Boyce and Chamberlin published "SEQUEL: A Structured English Query Language" in 1974 (as cited in Chamberlin, 2012). The invention of SQL and the relational database model on which it relies marked a transformational improvement in the ability of developers to define and manipulate

data. With the help of these technologies, a significant abstraction in the data management process was introduced through the definition of an intuitive data structure and a simple querying language that could carry out complex inner processes. To the surprise of it its creators, the popularity and use of SQL grew rapidly after its commercialization in the 1970s and 1980s, leading to its eventual formalization by the ANSI and ISO standards groups (Chamberlin, 2012). From the time of its commercialization onwards, SQL became industry-standard practice for database management and is now a fundamental component of a formal computer science education.

The concept of a NoSQL database, or a database that departs from the relational architecture and properties in any way, is not as modern as it may seem, as lots of data storage applications were in existence throughout many domains prior to the RDBMS and SQL advancements. However, the emergence of NoSQL as a widely used and well-known technology was not brought about until the need for mass scalability by web-era applications introduced distributed and parallel computing in the 1990s and early 2000s (Tiwari, 2011). An early indicator of this need was recognized by search engine company Inktomi, which struggled to scale its relationally based systems and eventually collapsed. This problem was also encountered by search engine company Google, founded only two years after Inktomi, which experienced similar problems with efficient processing, effective parallelization, and scalability as its relational systems grew and evolved (Tiwari, 2011). To solve these problems, Google completely redesigned their system architecture by creating a distributed filesystem, a column-oriented NoSQL data store, a distributed coordination system, and a MapReduce-based parallel-execution algorithm, effectively optimizing each layer of their application stack (Tiwari, 2011).

Four papers were released by Google in the early 2000s detailing each of these optimizations which laid the groundwork for other companies to replicate Google's work and transition their systems into a maintainable NoSQL-based web-scale architecture. These four advancements will be defined and discussed in greater detail in later sections to demonstrate why this transition fundamentally relied on a departure from the relational database model in favor of NoSQL technologies.

After Google released the papers detailing their system overhaul, developers at Yahoo! soon replicated Google's design which ultimately resulted in an open-source distributed computing stack called Apache Hadoop. This system was soon adopted by some of the world's largest websites like Yahoo! and Facebook which were facing similar challenges in scaling their relational data architectures at the time (Vance, 2009). In 2007, online shopping giant Amazon published similar work detailing its NoSQL datastore Dynamo, which also departed from the relational model in order to support the scalability of Amazon's systems as they struggled to manage the spike in activity during holiday seasons using a relational architecture (Brockmeier, 2012). It is at this time that the term NoSQL began to gain traction, although it was reportedly first used in 1998 when developer Carlo Strozzi created a lightweight, open-source relational database that did not use SQL (Foote, 2018). While there has been some ambiguity since the term's inception concerning whether the acronym translates to No SQL or Not only SQL, both are generally accepted and refer to the same technology (Foote, 2018). With two web giants, Google and Amazon, actively utilizing NoSQL technologies to support their worldwide systems and an open-source counterpart Apache Hadoop becoming more popular with smaller-scale organizations, NoSQL gradually became a widely known and implemented technology for

storing and managing data, especially in large distributed systems where the horizontal scaling capability of NoSQL is most utilized.

SQL and NoSQL Data Architectures and Core Properties

Simply stated, NoSQL is an umbrella term for any data storage system that doesn't follow the well-established RDBMS architecture or properties (Tiwari, 2011). To follow the RDBMS architecture, a database must be organized into tables with rows and columns representing pre-defined relationships between the data objects. To follow the RDBMS core properties, the database must comply with the ACID components of atomicity, isolation, consistency, and durability (Amazon, n.d.). It is important to note here that a database must only digress from the RDBMS model in one of these categories, architecture or properties, to be considered a NoSQL database, although many digress in both. An example of this is the Neo4j graph database, which does not follow the RDBMS data architecture but does, in fact, maintain the ACID properties (Tiwari, 2011).

Difference Between SQL and NoSQL Data Architectures

SQL. The relational database provides a straightforward storage approach using rows, columns, and tables much like a spreadsheet. In a relational database, tables are representative of entities, or objects, in a system, each entity holding its own data. Each column determines a certain attribute of that entity, with one column reserved as a unique identifier for the row, or primary key. These entities, represented through tables, can then interact by referencing one another through foreign keys, which are references to the primary key of another table. The database tables must be row-oriented, which simply means that all data within a row are stored together. SQL can then be used to search and manipulate a database through queries

and insert/update statements so long as the developer has accurate knowledge of the tables, their attributes, and how they are connected through primary and foreign keys.

NoSQL. NoSQL diverts from SQL entirely and represents a much broader set of data architectures whose only uniting property is that they do not follow the relational model described above. As NoSQL has grown and evolved, four general categories have emerged to represent the different data architectures available; however, many combinations of these categories have been created and are often referred to as multimodal NoSQL databases. The four main NoSQL data architectures are described in detail in the following sections. Popular implementations of each category are included along with several examples of web-scale companies that have used the NoSQL category to scale their systems.

Key-value stores. Key-value stores are used to represent key-value pair data. The pairs are stored in a HashMap or an associative array with unique keys and a pointer to the data value for that key. Instead of using a query language like SQL, key-value stores access data using the HashMap get, put, and delete operations, which have an amortized time complexity of O(1), or constant time, lending very high performance (Tiwari, 2011). The key-value NoSQL data architecture does not require that values adhere to a pre-defined schema but rather stores each value as a binary large object (BLOB). This means that for some keys the value could point to a string, while other keys will point to a document, integer, or some other data type (Saravanan, 2019). This provides a minimally structured architecture that allows greater flexibility in storing data.

Representing key-value data in a relational database results in a two-column database where the unique key is in one column and the value is in the other column. However, all values

must adhere to the same predefined data structure, unlike the NoSQL approach which provides greater flexibility for value types. To access a key-value pair, a SQL query must be used which finds a particular key by searching the entire key column in linear time, or using an index for faster performance when one is available. Thus, key-value stores provide a significant performance improvement for searches, insertions, and deletions compared to the relational model because the key-value NoSQL approach relies on constant time operations whereas the relational approach relies on linear time operations and is dependent on its indexing technology to execute queries faster.

Nonetheless, systems with well-defined entities and relationships are often best defined upfront using a relational database in small systems. If entities have many attributes that need to be accessed separately, storing all of these attributes together as one BLOB related to a single key can create confusion when accessing the data and may make more sense simply using a relational architecture and SQL. For this reason, key-value databases are best implemented in systems where the data and relationships between them are more varied. A common use of key-value store databases is in caching, which provides an in-memory reference of the most-used data in an application to reduce disk accesses and boost performance. Because the data stored in a cache can vary widely and should be accessible extremely quickly for the cache to function effectively, a key-value data store can perform well in this situation (Tiwari, 2011). Amazon's original Dynamo database falls under this category, as well as the popular Cassandra, Voldemort, and Redis databases. The Redis database has been implemented at Snapchat, StackOverflow, and Craigslist as a part of their fundamental data architectures (Redis, n.d.).

Columnar databases. Columnar databases, also called wide-column or column-oriented data stores, are similar to the relational architecture in that the data are stored in rows and columns, however they differ in that the orientation of the database relies on the columns, not the rows. This means that instead of storing database information on disk in row-groups, the data of each column are stored together in a column-group. To ensure that related information can still be viewed as a unit as in the row-oriented approach, each unit of data is structured as a key/value pair where the primary identifier, called primary key in relational databases but a row-key in columnar databases, is the same across column entries even though they are not physically located together on disk. To illustrate this, consider the following data stored relationally in Table 1.

Table 1

Relational database storing employee names and bonuses

ID	Last	First	Bonus
1	Doe	John	3000
2	Smith	Sam	1000
3	Beck	Jane	9000

In this relational, row-oriented database, the data are stored on disk by row-groups as displayed in Figure 1.

```
Primary key: 1
        Last:   Doe
        First:  John
        Bonus:  3000
Primary key: 2
        Last:   Smith
        First:  Sam
        Bonus:  1000
Primary key: 3
        Last:   Beck
        First:  Jane
        Bonus:  9000
```

Figure 1. Representation of data storage locations on disk using the relational model.

In a NoSQL columnar approach, the data would be stored on disk by column groups as shown in Figure 2.

```
Column group: Last
        row-key 1:      Doe
        row-key 2:      Smith
        row-key 3:      Beck
Column group: First
        row-key 1:      John
        row-key 2:      Sam
        row-key 3:      Jane
Column group: Bonus
        row-key 1:      3000
        row-key 2:      1000
        row-key 3:      9000
```

Figure 2. Representation of data storage locations on disk using the columnar model.

As Figure 2 displays, the data in the columnar approach are stored in column-oriented groups, and each entry is a key-value pair. Column-oriented data stores also provide column-families to keep related columns nearby, so in this example the first and last name entries could be combined into a column-family bucket that stores the first and last name together. This transforms the database to Figure 3.

```
Column-family 'Name'
        row-key 1:
                Last:   Doe
                First:  John
        row-key 2:
                Last:   Smith
                First:  Sam
        row-key 3:
                Last:   Beck
                First:  Jane

Column group: Bonus
        row-key 1:      3000
        row-key 2:      1000
        row-key 3:      9000
```

Figure 3. Columnar model using column-family buckets.

An important characteristic that most columnar databases maintain is a sorted order, which is key to their efficient processing capabilities. The units of data are sorted by row-key, as Figures 2 and 3 display. As data grows on a given node, this sorted order is maintained. That is, if a new data item is added to a previously existing row-key, say an employee who formerly had no bonus now has a $1000 bonus, the row-key for that employee must be inserted into the bonus column-group in sorted order, not simply appended at the end of the group. As a node becomes full, it splits into multiple nodes, which also maintain a sorted order with each other, providing one continuously sorted column-store even across nodes. Because the database maintains this sorted-order property, data seeks by row-key value are extremely efficient because the distributed architecture can quickly determine which node a row-key is on (Tiwari, 2011).

Several additional advantages result in the columnar approach. First, null values are not stored when a value does not exist within a column. This contrasts relational models which allocate space for each column of a row entry and enter a null placeholder if there is no value for that row-column value, which can be overwritten later if the data state changes. Second, database columns do not need a-priori definition or declaration as they would in a relational database.

Because the columns can accept any data type so long as it can be persisted to an array of bytes, maintaining columns with varied data-type entries is possible (Tiwari, 2011). One final advantage that column-oriented data storage provides is that aggregate functions on column stores access contiguous memory. While a relational model would have to skip through the entire database of information to find and add each employee's bonus into one aggregate sum, if for example the average employee bonus was requested, the columnar approach stores all of the bonuses together so that the information that is relevant to the search is accessible in contiguous memory and can be processed much faster (Monash, 2011).

Columnar databases help distributed databases optimize performance by storing related information together in a new way, namely by column-groups as opposed to row-groups. However, to piece the data of a particular member back into a single unit, multiple column-families must be accessed to retrieve all instances of the row-key and the data it contains. For example, in the database in Figure 4, both the name column-family and the bonus column-group would need to be accessed to combine related information on row-key 2 back into a single unit. Depending on how the data will most often be accessed, a relational model may be more efficient for smaller, minimally distributed systems because related data for each key will be stored together. For a highly distributed web-scale architecture, however, processing data in a row-oriented fashion is less efficient than processing it by column-groups. Just as indexes are used in relational databases to mitigate inefficiencies and provide quicker access to database information, indexes can be kept on NoSQL columnar databases as well to make re-grouping row-key entries faster.

Some popular columnar databases include HBase, Hypertable, and Bigtable, which is the column-oriented sorted data store originally developed by Google that will be discussed in greater detail in later sections. Besides Google's use of Bigtable, HBase has been used by Facebook, Hulu, and Yahoo!, amongst others (Tiwari, 2011).

Document-style databases. Document-style databases are like key-value stores in that each document is assigned a unique identifier internally which corresponds to all of the data for that record (similar to the key-value BLOB). However, each document maintains a greater structure than the key-value BLOB as document data can be stored in JSON, XML (eXtensible Markup Language), or BSON (Binary Encoding Of JSON). Like columnar databases, document-style databases only store information on the non-null attributes of an entry and do not pay attention to non-required attributes that are not included in a particular entry. This means that space is not wasted on null attributes and allows a system to hold more entries than it would in a SQL approach in the case of a sparse database. Document databases provide document indexing using the primary identifier of the document as well as the properties of the document which boosts search performance (Tiwari, 2011). Figures 4 and 5 demonstrate a simple SQL database and its NoSQL document store representation, respectively.

ID	Name	Cell	Resume	Job History	Skills	Hobbies	Website	References	Cover Letter
1	Seth Ames	(424) 431-9441	Sames.txt	...					
2	Kaylie Martin	(416) 203-4431	Kmartin.pdf	...	C++, Java, ...		www.kmartin.com		KMartinCL.pdf
3	Sandra Soo	(930) 990-4249							
4	Richard Frank	(810) 890-4056	Rfrank.pdf	...					RFrankCL.pdf
5	Kevin Patel	(469) 339-5995							
6	Ethan James	(952) 294-3617	Ejames.txt			Web dev...	www.ejames.com		
7	Julia Kissel	(277) 877-5601	Jkissel.docx						
8	Miranda Benz	(357) 694-0295	Mbenz.pdf	...					
...									

Figure 4. Example single-table SQL database holding job applicant records.

```
{
  "candidates": [
    {
      "id":           1,
      "name":         "Seth Ames",
      "cell":         "(424)-431-9441"
    },
    {
      "id":           2,
      "name":         "Kylie Martin",
      "cell":         "(416)-203-4431",
      "resume":       "Kmartin.pdf",
      "Job_History":  "Job history info...",
      "Skills":       "C++, Java, ...",
      "Website":      "www.kmartin.com",
      "Cover_Letter": "KMartinCL.pdf"
    }
  ]
}
```


Figure 5. Document-style representation of the data from two entries of the database in Figure 4.

In Figure 5, a sparse row-entry, with the id of 1, and a dense row-entry, with the id of 2, are shown to demonstrate how document data stores ignore non-existent data points, while the SQL representation in Figure 4 still allocates space for each attribute even though many of the columns are empty for most of the entries.

Document-style databases are like relational databases in that the data for an object or entity are stored together, not split apart as in columnar databases. A major disadvantage of document databases, however, is that they cannot perform joins like a relational database can. This can make data analysis and querying expensive, because the more loosely defined structure provides less information for algorithms to work with when searching for or grouping data (Saravanan, 2019). As previously noted, however, many NoSQL databases store specific metadata and indexes to make querying the data more efficient. Nonetheless, in a join-heavy, minimally distributed system a document-style database will likely perform much more poorly than a relational database.

Two of the most popular document-style databases include MongoDB and CouchDB. MongoDB has been implemented by Intuit and Github, amongst others, and CouchDB has been implemented at Apple, BBC, Cern, and more (Tiwari, 2011).

Graph databases. Graph databases model data using nodes and relationships where a node represents an entity or object and a relationship represents the connection between two nodes. Graph databases predetermine relationships, meaning that the database does not have to figure out how nodes are connected at query time using primary and foreign keys but rather has direct links to any related nodes. Because of this, resolving relationship paths can occur significantly faster in graph databases compared to a relational model.

Consider, for example, the following relational model of persons, department members, and departments as visualized in Figure 6. To find the names of all people in the persons table given a department id of 111, the database must first search the dept_members table for that department id and gather all corresponding person id's which can then be used to search the persons table and find matches. This results in a lot of searching over irrelevant entries to find the needed information.

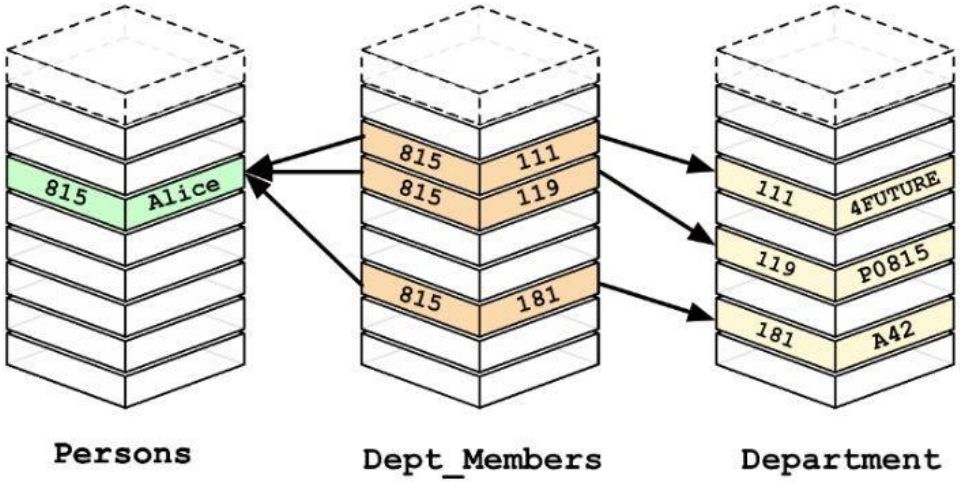

Figure 6. Adapted from "The Basics of NoSQL Databases – and Why We Need Them", by N. Saravanan, 2019, Reprinted from https://www.freecodecamp.org/news/nosql-databases-5f6639ed9574/. Reprinted with permission.

In the graph database visualized by Figure 7, predetermined relationships between these three entities eliminate searches over irrelevant data and allow for immediate access when resolving relational paths.

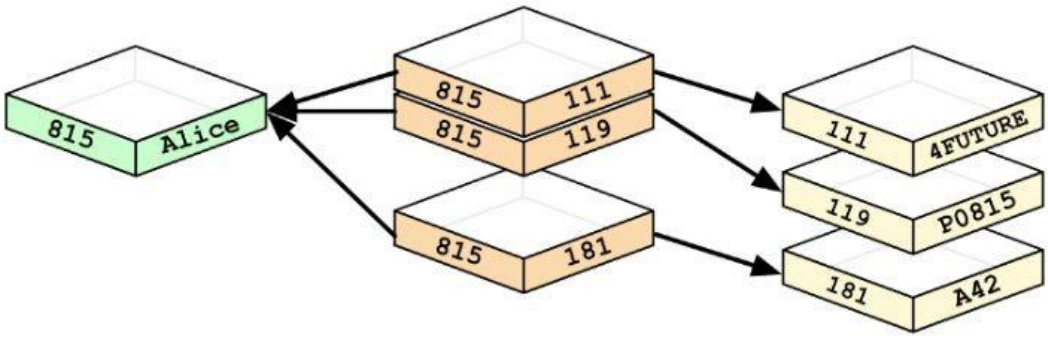

Figure 7. Adapted from "The Basics of NoSQL Databases – and Why We Need Them", by N. Saravanan, 2019, Reprinted from https://www.freecodecamp.org/news/nosql-databases-5f6639ed9574/. Reprinted with permission.

One disadvantage to the graph approach, however, is that changing a relationship between two nodes requires a regressive update to the entire relationship path because when a

link is broken, the rest of the path cannot remain intact or the data will become dirty (Saravanan, 2019). Thus, a database with persistent relationships that will be infrequently changed once created could be suitable for a graph database. Two well-known graph databases include Neo4j, which has been implemented at Walmart (Neo4j, n.d.), and FlockDB, which has been implemented at Twitter (Tiwari, 2011). Because social networks are easily represented using graphs, they have become a very common use-case for storing data in a graph database.

NoSQL data architectures summary. These four categories make up the primary approaches to structuring data using a NoSQL model and each provide their own strengths and tradeoffs when compared with relational systems. It is important to note that most complex data ecosystems implement many types of databases, often both SQL and NoSQL models, in order to optimize different aspects of a system. As the following sections will demonstrate, while relational models are at times optimal in small systems, their poor horizontal scalability severely limits their use in large distributed systems.

Difference Between SQL and NoSQL Core Properties

The four NoSQL categories allow for more tailored data structures that better fit the characteristics of the data they store; however, NoSQL's advantage in distributed systems relies on its ability to scale horizontally, which is a direct result of its core properties.

ACID properties. The core properties governing relational databases are atomicity, consistency, isolation and durability. A database cannot be considered relational if it does not uphold these properties. Atomicity implies that transactions must be all-or-nothing, that is, all of the changes of a transaction must complete, or the entire transaction gets rolled back. For example, in a banking system, if a credit is made to account B successfully, the corresponding

debit from account A must complete. If a failure occurs that prevents the account A debit from completing, the entire transaction is rolled back, and the account B credit is reverted to its original state. Consistency requires that the state of data for an entire system remains the same at the beginning and end of a transaction. Building on the first example, if the sum of accounts A and B prior to the transaction equals $100, after completing the transaction between these accounts their combined sum must remain $100. Isolation seeks to hide intermediate states of transactions from other transactions so that multiple transactions can be run concurrently. Drawing on the original example once again, the isolation property ensures that any extraneous transactions looking at accounts A and B see the transferred funds in either account A or account B, but not in both, nor in neither. Finally, durability ensures that upon completion of a transaction, changes to the data are persistent and cannot be lost even in the event of a system failure. This is often enforced by logging transactions before completing them, so that if the system fails after the logging but before the transaction completes, the necessary information is available to restore the database to its previous state (IBM, 2019). Together, these properties define the "highest level of transactional integrity in database systems" making relational databases highly reliable and fault-proof (Tiwari, 2011). As further sections will demonstrate, however, the rigidity that the ACID properties impose on relational databases produce severe limitations in their ability to scale horizontally and support large distributed systems.

CAP theorem. Barring exceptions like Neo4J, which more closely adheres to the Not only SQL naming convention as it meets some but not all of the qualifications of an RDBMS, most NoSQL datastores do not maintain ACID properties and instead adhere to the CAP theorem, which stands for Consistency, Availability, and Partition Tolerance and states that of

these three properties, one must be compromised to allow for the other two in large, distributed systems. The CAP theorem was originally developed by Eric Brewer, one of the founders of Inktomi, which was mentioned earlier as one of the first companies to run into the issues of RDBMSs in large distributed systems. While Inktomi did not survive the transition, the research Brewer conducted in his attempt to scale Inktomi laid the foundation for other companies to finish what he started. Brewer originally shared his research regarding the CAP theorem during a keynote at the ACM Symposium on the Principles of Distributed Computing in 2000 (Brewer, 2000).

The specifics of the three CAP domains are as follows. Consistency in CAP is not the same as its ACID counterpart. Here, consistency means that reads and writes are consistent so that concurrent operations both see a valid and consistent data state. This is more aligned with the isolation and atomicity properties of ACID and ultimately means there can be no stale data (Tiwari, 2011). Availability implies that the system is ready to serve the moment it's needed. Minor delays and minimal hold-ups are not tolerated in the CAP definition of availability. If a system is not ready on demand, it is not available (Tiwari, 2011). Partition tolerance deals with the fact that NoSQL databases scale out, not up, implying that the data are partitioned into many units instead of one, stronger unit as storage needs increase. Due to this design, partition tolerance is "the ability of a system to continue to serve requests in the event that one or more of its cluster members become unavailable" (Tiwari, 2011, p. 174). In other words, failing nodes do not cause a failing system.

CAP not only differs from ACID in the composition and definition of its properties, but more importantly in that the CAP theorem recognizes a compromise is necessary to accomplish

the efficiency and parallelization of a distributed system while ACID is a rigid set of rules that cannot be compromised (Brewer, 2012). While the CAP theorem at first garnered backlash by those who argued that a compromise in any of the three categories was not a viable solution, his theorem was proven by Seth Gilbert and Nancy Lynch (2002), which solidified that NoSQL was the way forward for large, distributed systems.

Why Distributed Systems Need CAP

It is not impossible to maintain the ACID properties in a large, distributed system. It is extremely complex and involves challenges regarding multi-database isolation, blocking, and resource unavailability, but strictly speaking it is possible (Tiwari, 2011). While the four ACID properties of atomicity, consistency, isolation, and durability will exist in such a system, the core CAP property of availability is still compromised because resources are locked frequently, and sometimes for long periods of time if a transaction is complex, making all other clients wait for the needed resources to become available. In many cases, the wait times are significant which is an unrealistic compromise in a system serving millions of users in real-time. As the following sections will discuss, consistency is a much more realistic compromise in the CAP system as availability is crucial and partition failures are inevitable due their reliance on hardware. Once consistency is compromised, however, the ACID properties are no longer maintained, and the system must transition into a NoSQL solution.

Google's Distributed Architecture Optimizations

With an understanding of the underlying differences between SQL and NoSQL architectures and properties, the optimizations originally published by Google and later open-

sourced can now be discussed in detail to provide an in-depth understanding of how a NoSQL data architecture was created to support web-scale distributed applications.

Google File System and the CAP Compromise

The foundational layer of Google's architecture overhaul, detailed by Ghemawat, Gobioff, and Leung (2003), describes the system as "a scalable distributed file system for large distributed data-intensive applications" (p. 1). The authors note that to accomplish the Google File System, hereafter referred to as GFS, they had to make a "marked departure from some earlier file system assumptions" which led them to "reexamine traditional choices and explore radically different design points" (p. 1). From a high-level, the GFS architecture consists of the GFS master-server and many GFS distributed servers called chunkservers which contain chunks, or file fragments. The GFS master contains only metadata that maps 64-Bit unique identifiers for chunks called chunkhandles to their respective chunkservers. The chunkservers each contain many chunks.

The request model involves a client which sends the GFS master the chunk index and filename of the data it is looking for, gaining the chunkhandle and the byte range of the chunk data in return. The client can then use this chunkhandle to access the appropriate GFS chunkserver which then returns the chunk data that the user is ultimately seeking. To maintain the state of the database, the GFS master also interacts regularly with the chunkservers independent of client requests in a process called *heartbeat messages* so that it can give the chunkservers instructions and collect their states (Ghemawat et al., 2003). This process is visualized by the authors in Figure 8.

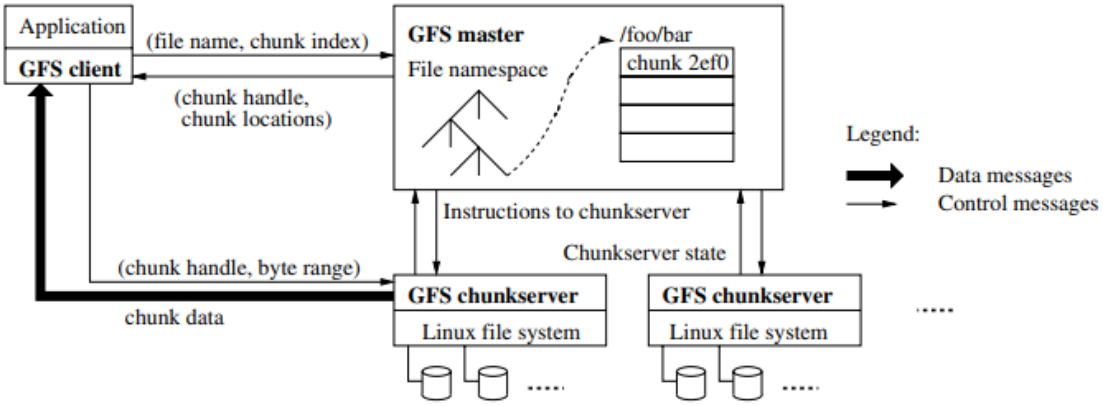

Figure 8. Adapted from "The Google file system", by Ghemawat, Gobioff, and Leung, 2003, *ACM SIGOPS Operating Systems Review*, *37*(5), 3. Reprinted with permission.

The authors relate that the GFS has a "relaxed consistency model that supports our highly distributed applications well but remains relatively simple and efficient to implement" (Ghemawat et al., 2003, p. 1). That is, the engineers opted to compromise the consistency of the GFS to allow for availability and partition tolerance, as per the CAP theorem. To accomplish this, the GFS replicates each chunk in three separate chunkservers which must reside on different racks in the server farm. Three is only the default replication rate, and chunks can be replicated in higher numbers if requested by overriding the default configuration when storing the chunk to chunkservers. If a chunkserver does fail, it is designed to restore its state and start-up in seconds no matter what caused its termination. This ability, plus the fact that there are several if not many replicas of each chunk, ensures that data are always available on demand despite potential node failures throughout the system. Because multiple replicas are stored, it is possible that some may not be as up-to-date as others if a recent failure has occurred on the chunkserver where a particular replica resides. This means that the data state across replicated chunks can have disparities implying a lack of consistency for any read/write operations accessing different, unequal replicas at the same time. Because the heartbeat message process can quickly resolve

consistency issues due to chunkserver failures, this was not an unreasonable compromise for the engineers designing the GFS (Ghemawat et al., 2003). Later sections will show how the third layer of Google's data architecture overhaul, the distributed coordination system, also bolsters the consistency of the data to help offset the implications of compromise in this category.

Bigtable, A Column-Family-Oriented NoSQL Data Store

The second optimization Google made to overcome the scaling challenges of relational databases was Bigtable, a distributed storage system for structured data. While this description may seem to imply that Bigtable is relational because it is described as a storage system for structured data, the authors explicitly state otherwise by noting that "Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage" (Chang et al., 2008, p. 1). A more technical definition of Bigtable describes it as a "sparse, distributed, persistent multi-dimensional sorted map" that is "indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes" (p. 1). When cross-referenced with the four NoSQL categories, the Bigtable design falls under the columnar or column-oriented data store architecture. As a *sorted* column-oriented map, Bigtable maximizes the processing optimizations available with column-oriented data stores.

Thus, Bigtable is itself a NoSQL data structure, as it departs from the relational model and adopts the column-oriented structure instead. The authors of the original Bigtable paper note that it relies on the GFS to handle the physical implementation of its storage in a distributed fashion (Chang et al., 2008). Just as Hadoop is the open-source corollary to the GFS, HBase is

the popular, open-source, sorted ordered column-family store database that was developed after Bigtable's release to make this technology accessible to a wider audience (Tiwari, 2011).

Distributed Coordination System Using the Chubby Lock Service

The next layer of Google's architectural overhaul, the chubby lock service for distributed coordination, became a necessity after distributed systems were recognized as the way forward in web-scale data storage. However, this layer does not relate directly to the advantages of NoSQL in distributed systems, so it will only be described in brief. Essentially, now that system data could successfully be stored and accessed over thousands of nodes using the GFS and Bigtable, some guidelines needed to be put in place to ensure that conflicts between clients accessing the same resources at the same time were minimized. Because consistency was the chosen compromise in the GFS, the chubby lock service is not a fault-proof solution, but instead provides a mechanism to bolster consistency as much as possible across nodes.

The problem that a large, distributed, highly accessed system faces with consistency is as follows. Client A and B both want to access resource R, which is some data in one of the nodes on a distributed system. Client A sends a request to resource R, but a failure occurs. Then client B sends a request to resource R, and resource R begins communicating with B. After the failure with client A is resolved, resource R fulfills client A's requests, based on the logs that were saved before the failure, and at this point may be sending back data that client B manipulated in some way, thereby giving client A incorrect data to its original request.

To mitigate this, a lock system was put in place to help resource R know when it should reject a request using advisory locks. The use of the word *advisory* here is important because oftentimes locks are implemented in a strict sense, meaning that if client A has the lock to

resource R, no other clients can access resource R until client A releases it and it is their turn to obtain it. Because consistency is a viable compromise in web-scale distributed architectures, locks follow less strict guidelines. For example, in the example with clients A and B and resource R, the lock system may reject client A's request because it knows that it has an advisory lock from B which it gained when it began processing the request from B. Then client A simply must resend its request to R but now knows that it is getting a response that may have changed from whatever happened during client B's request. Alternatively, client A's request can be forwarded to a different replica of the data so that it is more likely that it will obtain the information it originally requested, although concurrent operations could also have occurred on the replica as well. In this example consistency is still compromised but the clients can know when an advisory lock may be affecting the results of a request in the case of a failure amidst concurrent operations. A deeper description of the chubby lock system implantation of advisory locks can be found in the original 2006 paper written by Google engineer Mike Burrows.

MapReduce-based Parallel Algorithm Execution Environment

The final optimization in Google's original distributed data architecture overhaul was the MapReduce-based parallel algorithm execution environment. This will also be described in brief as it leverages the distributed architecture made available by NoSQL and the CAP compromise but in and of itself is not part of the NoSQL architecture. Instead, it was the initial algorithm used to efficiently process nodes in parallel and reduce the results of each node into a final output for the client. Since the release of the original paper outlining the MapReduce algorithm (Dean & Ghemawat, 2004), Google and many other distributed architecture systems have advanced to

more modern and superior parallel processing algorithms. However, at their core, all of these algorithms rely on the storage of data across many nodes so that parallel processing is possible in the first place.

The MapReduce algorithm derives its name from its two main stages: map and reduce. The process is as follows. A client sends a request for data to the distributed system, for example, a count of every instance of the word ‘NoSQL’ across the entire GFS. From a user perspective, this is analogous to typing ‘NoSQL’ into the Google search bar. The GFS master sends the instruction down to the cluster members through a heartbeat message, and the cluster members then process the instruction in parallel. To process the instruction, they search their contents and map out all of the instances of ‘NoSQL’ that are found. Then, the results of all of the cluster members are reduced into a final output, and this data is sent back to the client (Dean & Ghemawat, 2004). By splitting the search field into many nodes, searching them in parallel, and combining the results back into a final output, the distributed system is able to efficiently serve client A’s request while a non-distributed system, or a system that aimed to keep the number of nodes as small as possible to maximize vertical scaling, would process the requests much more slowly because each node would have significantly more work to do as it searched its massive archive of data (Dean & Ghemawat, 2004). A high view of the MapReduce algorithm is visualized in Figure 9.

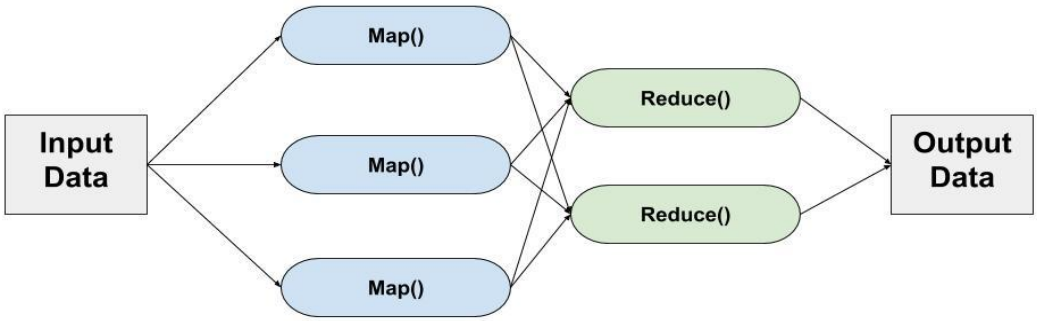

Figure 9. A high-level view of the Map-Reduce algorithm.

Summary of Google's Distributed Architecture Optimizations

The original data architecture overhaul initially published by engineers at Google but soon replicated by developers around the world fundamentally departs from the relational, ACID-constricted model and instead adopts a NoSQL, CAP-oriented approach to allow for optimal efficiency and parallelization. The GFS compromises on consistency to allow for high availability and partition tolerance in its distributed data storage methodology. Bigtable switches the data architecture layer from a row-oriented SQL structure to a column-oriented NoSQL structure. The Chubby Lock system bolsters consistency as much as possible with its advisory locks, and MapReduce provides an efficient algorithm for running operations on separate cluster members in parallel and then reducing the results into one final output. This all fundamentally relies on the distributed system, which is where NoSQL makes its entrance as a horizontally-scaling alternative to the rigid, vertically scaling relational model.

Conclusion

The above sections explore the origins of NoSQL as it relates to web-era scalability and detail the underlying reasons why SQL could not be scaled in these large, distributed systems. The ultimate cause of this limitation relies on the CAP theorem, which proves that distributed

systems require a compromise in order to support mass storage needs and efficient processing through parallelization. Because NoSQL data stores are governed by CAP while SQL data stores are governed by ACID, NoSQL is able to scale to meet the demands of distributed computing while SQL is not. The first major transition from a relationally structured architecture into a NoSQL architecture in a web-scale company happened at Google, whose research helped the entire web-scale landscape to adopt the principles and properties of NoSQL in their own scalability transitions.

References

Amazon. (n.d.). The relational database. Retrieved from https://aws.amazon.com/relational-database/

Andrews, E. (2013). Who invented the internet?. Retrieved from https://www.history.com/news/who-invented-the-internet

Brewer, E. (2000, July). Towards robust distributed systems. *Symposium on Principles of Distributed Computing*. Portland, OR.

Brewer, E. (2012). CAP twelve years later: How the "rules" have changed. *Computer*, *45*(2), 23–29. doi: 10.1109/mc.2012.37

Brockmeier, J. (2012). Amazon takes another pass at NoSQL with DynamoDB. Retrieved from https://readwrite.com/2012/01/18/amazon-enters-the-nosql-market/

Burrows, M. (2006). The Chubby lock service for loosely coupled distributed systems. *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA.

Chamberlin, D. D. (2012). Early history of SQL. *IEEE Annals of the History of Computing*, *34*(4), 78–82. doi: 10.1109/mahc.2012.61

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., … Gruber, R. E. (2008). Bigtable. *ACM Transactions on Computer Systems*, *26*(2), 1–26. doi: 10.1145/1365815.1365816

Dean, J., & Ghemawat, S. (2004). MapReduce: simplified data processing on large clusters. *Symposium on Operating System Design and Implementation* (6th ed.). San Francisco, CA.

Facts and Stats of World's largest data centers. (2013). Retrieved from https://storageservers.wordpress.com/2013/07/17/facts-and-stats-of-worlds-largest-data-centers/

Foote, K. D. (2018). A brief history of non-relational databases. Retrieved from https://www.dataversity.net/a-brief-history-of-non-relational-databases/

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The Google file system. *ACM SIGOPS Operating Systems Review*, *37*(5), 29. doi: 10.1145/1165389.945450

Gilbert, S., & Lynch, N. (2002). Brewers conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, *33*(2), 51. doi: 10.1145/564585.564601

Google data center FAQ. (2017). Retrieved from https://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq

IBM. (2019). ACID properties of transactions. Retrieved from https://www.ibm.com/support/knowledgecenter/en/SSGMCP_5.4.0/product-overview/acid.html

Monash, C. (2011). Columnar compression vs. column storage. Retrieved from http://www.dbms2.com/2011/02/06/columnar-compression-database-storage/

Neo4j. (n.d.). Who uses Neo4j?. Retrieved from https://neo4j.com/who-uses-neo4j/

Redis. (n.d.). Who's using Redis?. Retrieved from https://redis.io/topics/whos-using-redis

Saravanan, N. (2019). The basics of NoSQL databases-and why we need them. Retrieved from https://www.freecodecamp.org/news/nosql-databases-5f6639ed9574/.

Tiwari, S. (2011). *Professional NoSQL*. Hoboken, NJ: Wiley.

Vance, A. (2009). Hadoop, a Free Software Program, Finds Uses Beyond Search. Retrieved from https://www.nytimes.com/