information

Review

A Review of Polyglot Persistence in the Big Data World

Pwint Phyu Khine * and Zhaoshun Wang

Department of Computer Science and Technology, School of Computer and Communication Engineering, University of Science and Technology Beijing (USTB), Beijing 10083, China; zhswang@ustb.edu.cn

* Correspondence: pwintphyukhinecs@outlook.com or pwintphyukhine@xs.ustb.edu.cn; Tel.: +86-13051513579 (China) or +95-095-311-798 (Myanmar)

Received: 10 March 2019; Accepted: 4 April 2019; Published: 16 April 2019

Abstract: The inevitability of the relationship between big data and distributed systems is indicated by the fact that data characteristics cannot be easily handled by a standalone centric approach. Among the different concepts of distributed systems, the CAP theorem (Consistency, Availability, and Partition Tolerant) points out the prominent use of the eventual consistency property in distributed systems. This has prompted the need for other, different types of databases beyond SQL (Structured Query Language) that have properties of scalability and availability. NoSQL (Not-Only SQL) databases, mostly with the BASE (Basically Available, Soft State, and Eventual consistency), are gaining ground in the big data era, while SQL databases are left trying to keep up with this paradigm shift. However, none of these databases are perfect, as there is no model that fits all requirements of data-intensive systems. Polyglot persistence, i.e., using different databases as appropriate for the different components within a single system, is becoming prevalent in data-intensive big data systems, as they are distributed and parallel by nature. This paper reflects the characteristics of these databases from a conceptual point of view and describes a potential solution for a distributed system—the adoption of polyglot persistence in data-intensive systems in the big data era.

Keywords: polyglot persistence; big data; ACID; BASE; CAP theorem; SQL; NoSQL

1. Introduction

Big data has passed beyond its teenage years. According to MGI [1] "Big data refers to datasets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze". For handling data management in the digital universe [2], the characteristics of the first 3 Vs (Volume, Variety, and Velocity) of big data are regarded as the biggest challenges. Before the age of big data, SQL data stores were the perfect standard for centralized systems. However, big data require scale-out horizontality for the purpose of scalability, which is in stark contrast with Structured Query Language's (SQL's) inability to scale out well, i.e., shrinking or stretching resources as required. Huge volume requires scalability, whereas variety requires that handling different types of data structures. Data access velocity requires low latency and high throughput. Moreover, these types change in different application systems from time to time. Data sources must provide veracity assurance in order to retrieve data reliably for business processes (data insight). These big data characteristics impose the necessity of using different kinds of SQL and Not-Only SQL (NoSQL) databases in a big data universe.

Emerging big data challenges have been predicted from different perspectives since the early days of big data. The earliest predicted challenges included those of big data integration for unorganized, real-world, sometimes schemaless, complex multidisciplinary databases, completeness and consistency of integrated data, and the dire need for a non-SQL query. The need to handle the unbelievable hugeness of the information in terms of scale, scope, distribution, heterogeneity, and supporting technologies has likewise been predicted [3]. Some components of the ACID (Atomicity, Consistency, Isolation, Durability) properties guaranteed by SQL databases need to be compromised in order to achieve better availability and scalability in the distributed system [4]. The relational data model can no longer efficiently handle all the current needs for analysis of large and often unstructured datasets. NoSQL ("Not only SQL") databases with BASE (Basically Available, Soft State, and Eventual consistency) characteristics have emerged to fill this gap.

The CAP (Consistency, Availability, Partition Tolerant) conjecture [5] explains why SQL is not sufficient for a distributed system. Since the ACID characteristics are difficult to fulfill in a distributed manner, many researchers are considering new types of database systems with relaxed characteristics for implementation in distributed systems. The CAP conjecture is theoretically proven in [6]. It is assumed that it is impossible to implement all three properties—consistency, availability, and partition tolerance—in a distributed system. Therefore, consistency levels have to be compromised for databases that have to be operated in a distributed system. This reality is causing the database community to shift towards the development of new types of database systems with relaxed consistency properties, especially "eventual consistency" [7], that are different from the traditional relational SQL databases [8]. Eventual consistency means that all of the different states of a distributed system will become consistent after a period of time, even if they were inconsistent during network partitions.

However, NoSQL databases are not perfect, in that most of them are evolving technologies. Moreover, they are specific to certain niches in their areas of application. For this reason, the polyglot persistence approach [9] is applied in distributed data-intensive big data applications. Polyglot persistence generally entails the use of different types of databases, as appropriate, in different parts of the target system. The concept of using different languages in one application domain is called "Polyglot programming" [10]. The same concept is applied to "Polyglot Persistence" for the use of different types of databases in one application system, instead of forcing all data to fit into only one database type. However, there may be differences between approaches to implementing polyglot persistence, although the term "polyglot persistence" is used to describe different types of data storage technology being used in an organization or an application. "Polyglot Persistence allows one database to process one part of the application and use the same data which are used by the different database for processing another part of the same application".

Reference [11] suggests "Polyglot Persistence" to be the implementation of different types of data stores in education services. However, Polyglot Persistence is just a concept. Therefore, there is more than one way to accomplish polyglot persistence in the big data world. Using different types of data stores in data-intensive systems is just one of these. The state-of-the-art big data architectures of today apply polyglot persistence in their architectural concepts when building data-intensive application systems, such as social networks, e.g., Facebook and Twitter; scientific projects, e.g., square kilometer array telescope, which will produce 1TB raw data on a daily basis; and search engines, e.g., Google, Yahoo, and Bing.

The rest of this paper is structured as follows. Section 2 reviews the concepts and inspiration behind the emergence of NoSQL databases, and the question as to why "Polyglot Persistence" should be applied in data-intensive systems is presented from technical and business perspectives. Section 3 describes the theories that have motivated the emergence of different data stores and their relationships with polyglot persistence. The details of polyglot persistence approaches are discussed in Section 4. Section 5 discusses polyglot persistence as the de facto solution concept in big data applications, and conclusions are drawn in Section 6.

2. No Cure-All Database Solution Changes with Big Data World

There are many changes happening in different areas of the digital universe, which have, in turn, provided huge volumes of data that cannot be easily handled by SQL. Drastic increases in machine main memory, requests to handle non-OLTP ad-hoc queries, orientation towards shared-nothing architectures, the need for systems with self-everything modes, the growing popularity of little non-multipurpose languages (such as Python and PHP), the emergence of new markets like stream processing frameworks, the need for warehouse system enhancements (which could lead to a whole other research topic), proliferation of scientific projects, and the emergence of systems that need semi-structured data, text analysis, and unstructured data like images and video files are, just to name a few, demonstrate that a one-size-fits-all approach does not sit well in big data [12]. DBMSs are mainly considered based on a data model that outlines the database's logical structure. This process standardizes how the data elements are related to each other. There are three generations of database technologies—navigational (hierarchical, network databases), relational (SQL databases), and post-relational (NoSQL and NewSQL databases). Before the big data era, relational SQL was the most dominant type. However, the database world has witnessed a shift with the emergence of the big data era.

Changes in the overall database and data management landscape include ubiquity of structured and unstructured data variety, extended developer demands, advancements in memory (increasing cores and threads per chip leading more powerful and cheaper memory), network (advancements in distributed and parallel systems), hardware such as SSD—Solid State Drives—and data-driven architecture shifts in computing, especially with the wide adoption of cloud computing. Using shared nothing architecture eliminates the tedious resource sharing problems of memory, processor and disk. However, the network becomes a new bottleneck [13]. Therefore, SQL is no longer a one-size-fits-all solution [14] today, from both a technical and business perspective. There are significant technical changes that indicate that relational databases are not the answer to everything in today's world.

2.1. Technical and Business Perspective: A Brief Look at SQL and NoSQL Stores in the Big Data Landscape

Data is not homogenous from the most fine-grained level to the more coarse-grained level, i.e., from data types (Integer, Float, Character, String) to data formats (JSON, XML, Arvo, Parquet). The content of the data is constantly changing, in addition to being corruptible and mutable. Data is heterogeneous by all means [15]. In the software development lifecycle, an SQL schema is difficult to change after the design phase. There are differences between real-world entities and object relationships. Relational databases manipulate data stored in them with ACID characteristics, which can offer absolute data integrity.

Theoretically, SQL databases work seamlessly in standalone systems. However, when confronted with big data systems with a distributed nature, they are unable to handle what big data have requested due to their innate problems. The demand for more insight reveals SQL's shortcomings, such as sparse dimensionality, design inflexibility, limited data types support, and performance [16,17]. SQL database incompetency can be summarized as sparse dimensionality with null value problem, design inflexibility, limited data type support, the problem of join and aggregation operations, end-user performance and quality decline. Moreover, business needs to blend different types of data sources from different domains to use for a particular goal, e.g., acquiring data from some major book store websites will help enhance the movie recommender system to consider the user's favorite novel adaption to the movie.

Overall, there are two business demands that SQL databases cannot handle in the digital universe. Users now want data of different types for business—be it structured, semi-structured, unstructured, or raw data. Many of today's users want data directly from the source to find hidden insight—some of them wanting the most granular level of data. Both of these requirements cannot be fulfilled by SQL databases for two reasons. Firstly, a very large portion of the data (approximate 70% to 90%) in the business world today is unstructured data, while SQL can only handle the structured portion of data. Secondly, their concept is based on the clear line between OLTP and OLAP. Most data are structured before storing them as transaction data in SQL databases. These data are retrieved, transformed, and summarized to store in a separate data warehouse for the analytical process. However, when users want to retrieve insight from transaction data or raw data directly, they are unable to do so because of the rigorous design restrictions in dominant relational SQL databases.

2.2. (Relational) SQL Databases

Relational database (SQL) concepts mostly rely on basic features of system R developed in the early 1980s—features such as disk-oriented storage and indexing structures, multithreading for log-based recovery (e.g., recording every action in the database triggered by the transactions), lock-based concurrency control (e.g., two-phase locking), latency hiding (e.g., make data present in the memory to reduce access latency). The separation between data, transactions, and applications are first introduced by the Integrated Data Store which became the background work for relational (SQL) databases. It also introduced an idea about the elimination of redundant data and the first CRUD (Store/Create, Retrieve, Update, Delete) statements [18]. Relational databases, which are also called SQL databases, define and manipulate the data side by using SQL (structured query language) and became dominant in the database world.

Relational database theory was initiated by Codd [19] based on the relation sets in the form of tables with tuples (rows) and attributes (columns). To ensure the integrity and eliminate redundancy, SQL databases use normalization by splitting and trimming many tables based on entities and the relationships between them. SQL databases only ask what to fetch when a query is requested because they already have well-defined methods on how to transfer data in a tabular format (normal forms) by normalization rules. Many SQL vendors implement record-oriented storage systems ("row-store" or "row-oriented" architecture), where records are contiguously placed in storage. All the attributes of a record will be pushed to the disk in a single disk write. This write-optimization is very effective in OLTP (Online Transaction Processing) applications.

Relational SQL database lives in a shared environment and manipulates data stored in them based on the ACID characteristics—(Atomicity, Consistency, Isolation, and Durability) [20] for absolute data integrity. Jim Gray [21] deliberated the transformational state of transactions and defined the atomicity, consistency, and durability for databases, i.e., "atomicity (all or nothing), durability (effects survive failures) and consistency (a correct transformation)". Isolation comes from performing atomic transactions using concurrency control, such as locking—e.g., deadlock detection for synchronization [22]. SQL databases try to maintain ACID characteristics, regardless of standalone or distributed systems, usually imposing consistency at the end of each operation. However, due to the changing data world, ACID characteristics need to be negotiated.

2.3. NoSQL Data Stores

SQL databases cannot fulfill all the requirements of big data in the expanding digital universe. To solve the challenges imposed by the big data era, new types of database systems have begun to emerge for each specialized area. Not all of them have the ability of SQL databases, such as Just-in-time compiler, uniform query language, or ACID integrity. To differentiate them from SQL databases, they are termed as NoSQL (initially named as "No-to-SQL—NoSQL", later changed into "Not only SQL"—NoSQL). Due to their non-formality with traditional database standardization, they are sometimes termed as "data stores" instead of traditional "databases", and are thus called "NoSQL data stores."

The shortcomings of SQL databases are that they are not consistent with various applications that have emerged in this new digital era. The "one size fits all" concept with a single code line for all DBMS service is not a good approach for diverse data stores today. NoSQL database system development is designed to handle high rates of schemaless data capture and updates. The demise of the "one size fits all" concept, the development of market segments incompatible with the rigid relational model, and the need for rethinking data models and query languages for specialized engines in vertical markets based on the overall changes in the database and data management landscape have motivated the emergence of new database technologies [23].

NoSQL refers to a series of database management concepts that do not conform to SQL query language (without a query optimizer). NoSQL data stores do not provide reliable data dictionaries. They process data in a distributed manner without fixed table schemas, avoid computation intensive operations such as joins, store data in replication, and typically scale horizontally [24]. Most NoSQL are open-sourced. In general, NoSQL is schemaless, avoids join operations and handles redundancy by "embedding" and "linking". Embedding means that subclass entities are denormalized (collapsed) into superclass entities for the sake of efficient retrieval and consistency control. "Linking" means adding a key to the object when referencing objects and their relationships are static (e.g., book and publisher). In these ways, the mapping between memory structure and database structure is simplified [25].

Big Data systems are distributed in nature and cannot be handled with traditional tools and techniques; they have to scalable and flexible. As a result, the underlying file system for databases or data stores in the big data era must meet these requirements. Therefore, design assumptions (as failure is inevitable in distributed systems) and parameters (I/O operation, file block sizes) are revisited. The Google File System (GFS) [26], a scalable distributed file system with a fault-tolerant mechanism, is the first prominent pioneer to handle such large distributed data-intensive applications. It uses bigger file blocks in nodes and replication to handle node failure on clustered storage, with the master for handling slave nodes. However, GFS is a prosperity file system. Hadoop [27], which uses an elephant as its symbol, is the most widely used open source framework for big data batch processing that makes use of HDFS (Hadoop Distributed File System) [28]. HDFS borrows many concepts from Google's GFS (Google File System). Hadoop can be considered a de facto standard for big data batch processing systems. HDFS uses a MapReduce approach [29]—storing data as key-value pairs, mapping keys-value pairs, shuffling, sorting, and transforming—reducing the paired keys-values to get the desired output results. HDFS assures scalability and availability by replicating data between nodes (one for the original copy, one for the inter-rack node, and one for the extra-rack node by default) in a distributed system, avoiding the single node failure of SQL databases. Figure 1 provides the general structure layer of NoSQL data stores.

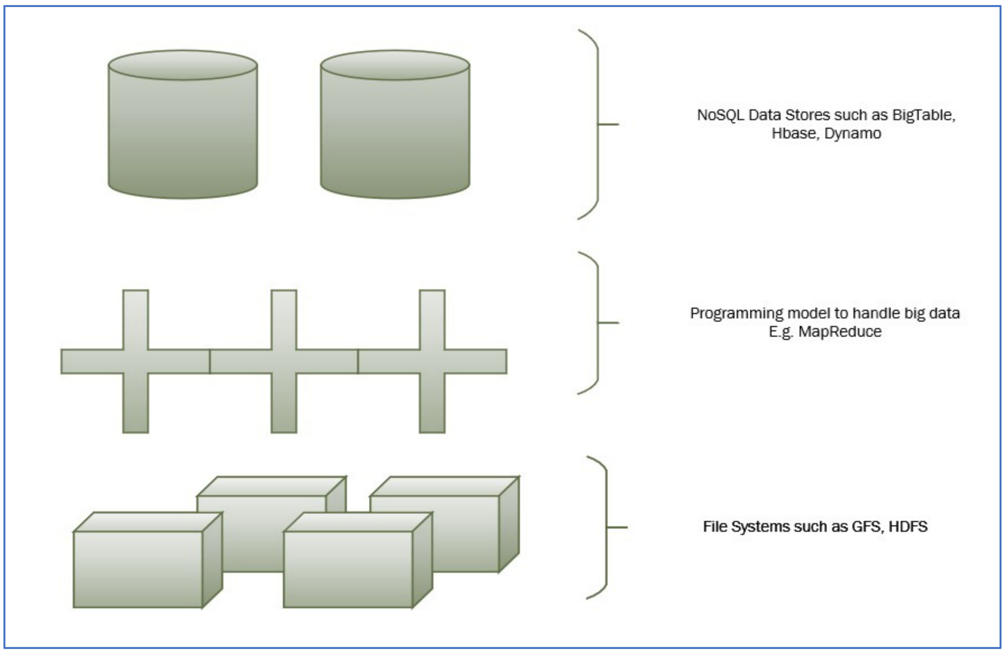

NoSQL Data Stores such as BigTable, Hbase, Dynamo
Programming model to handle big data E.g. MapReduce
File Systems such as GFS, HDFS

Figure 1. General Layer in Not-Only Structured Query Language (NoSQL) data stores. GFS, Google File System; HDFS. Hadoop Distributed File System.

As data are intentionally denormalized and replicated across nodes in the cluster to achieve higher availability, NoSQL databases can only offer BASE (Basically Available, Scalable, Eventually consistent) characteristics instead of ACID. With NoSQL databases, the scalability of databases for big data can be significantly relaxed. Many NoSQL databases, such as BigTable and HBase, use HDFS or GFS as their basic File System and built their conceptual models above them. Hive can work as a data warehouse solution for big data [30,31]. Many of the NoSQL databases today are targeted to semi-structured and unstructured data with high data complexity, which is the weak point of RDBMS. Conceptually, there is no data storage scale limitation for a NoSQL following the assumption that NoSQL databases must be scalable. However, in practice, there are some limitations—for example, how many nodes they can handle.

2.4. Types of NoSQL Data Stores

To support the demands of big data characteristics, NoSQL databases implement different structures and different data handling techniques. There are four famous types of NoSQL databases: Key-value stores, Columnar stores, Documentation stores, Graph stores [32], in order of ascending complexity. Different types of NoSQL stores are described in Figure 2.

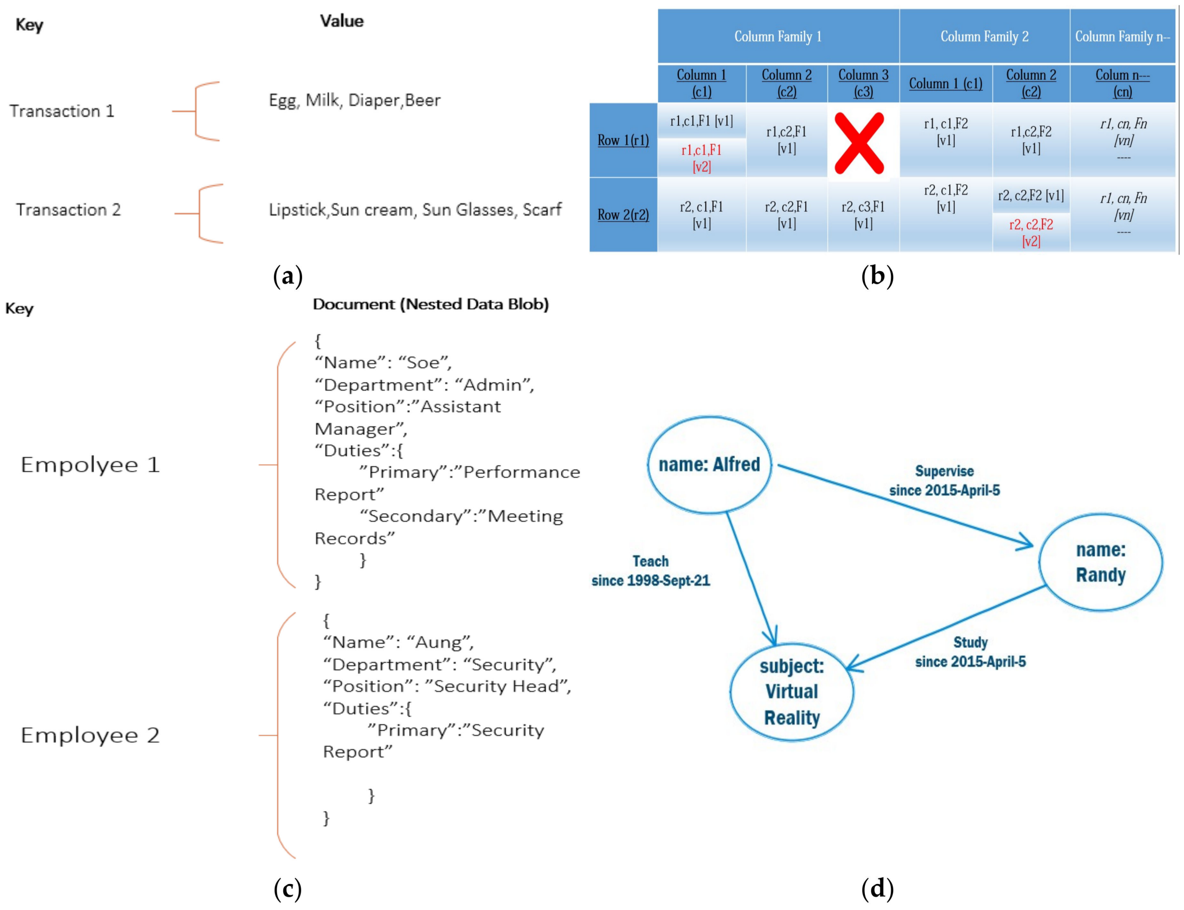


Figure 2. Types of NoSQL data stores. (**a**) Key-Value Store; (**b**) Columnar Store; (**c**) Document Store; (**d**) Graph Store.

1. Key-Value Stores

Distinct Characteristic: Key-Value Stores are the simplest among the NoSQL data stores.

Basic Model: All data are simplified into unique keys and their values (data blobs) using a hash table. Keys and values can be simply stored as a "String" in a list or dictionary structures—like a dictionary with many words that have many definitions.

Query and Schema: They have no definite (standardized) query language and no predefined schema. Some data models provide index values and handle semi-structured data like RDF (Resource Description Framework) and XML (eXtensible Makeup Language).

Suitable Usage: Transactional data that are not well suited in RDBMS [33], such as sale records, will work well in key-value stores. They are particularly good at working in aggregates, i.e., data blobs. Typical applications—dictionaries, lookup tables, query caches, e.g., Redis [34], Dynamo [35], Voldemort [36]. Figure 2a describes a key-value store structure, which is very suitable for transactional data.

2. Columnar Stores

Distinct Characteristic: Columnar stores are widely used to handle the read-heavy workload. Instead of being row-based like relational SQL databases, new types of big data stores, such as Cassandra, with column orientation and a read-heavy workload are called columnar stores [37,38]. These stores handle the "columns with null value" problem of SQL databases by giving each row to define their own column.

Basic Model: The key structure of a typical column store consists of row_key, column family name, column qualifier and timestamp for versioning. Sometimes, the columnar store will provide secondary indexes. By default, the most recent value in the specified column is retrieved. The design is dependent on access paths and multiple tables create data redundancy.

Query and Schema: They support the MapReduce programming model as the query style over the distributed file system, such as GFS or HDFS, e.g., BigTable [39] or HBase [40]. The number of tables in a columnar is much larger than that in RDBMS. Data Integrity and query processing have to be processed at the coding level. Columns can be defined in an ad-hoc manner, although the column family has to be defined in advance. The latest value is retrieved when there are more than one-row versions. The common structure of the columnar store is described in Figure 2b. The latest row version is illustrated in red.

Suitable Usage: Columns-based stores are used in OLAP systems, data mining, data analytics, and web crawling. Columnar stores are oriented toward a "Read-heavy" workload, so only required attributes will be read during a given query. All other unnecessary attributes can be avoided into bringing them into memory, resulting in sizable performance advantages.

3. Document Stores

Distinct Characteristics: Document stores are key-documented as values with the search index.

Basic Model: Each record is considered to be a document. Everything inside a document is searchable, although indexes are very large. Document stores are similar to versioned documents with nested key-values. When a new document is added, everything inside a document is automatically indexed and provide a flexible schema.

Query and Schema: Each document store has an API or query language that specifies the path or path expression to any node or group of nodes. Document stores are typically schemaless and can change their structure on the fly. They can handle complex documents structures and formats, such as binary format files like the Portable Document Format (PDF), MS Word, JavaScript Object Notation (JSON) and/or Binary JSON (BSON), and/or XML, using different types of data blobs.

Suitable Usage: Typical applications are online streaming, Web pages, images, and video storage, and natural language processing, e.g., MongoDB [41], CouchDB [42], MarkLogic [43]. Figure 2c illustrates a document store.

4. Graph Stores

Distinct Characteristic: Graph Stores use a "node-relationship" structure mostly useful for a graph structure, such as a social network.

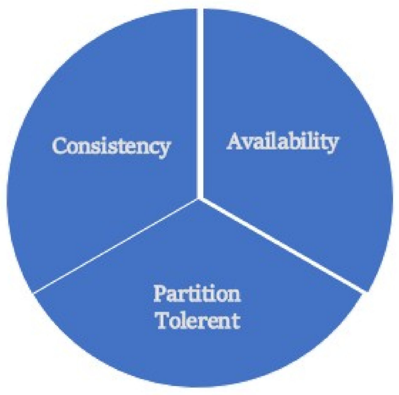
Basic Model: Graph stores are composed of vertices (nodes) and edges (relationships) with supporting properties—a key-value pair where data are stored. Graph stores are based on the concept of Graph theory. A node can be a person, a thing, or a place. Directed Acyclic Graph (DAG) is used for Provenance, i.e., data lineage [44]. An example Graph Store is depicted in Figure 2d.

Query and Schema: Instead of join operations, databases are traversed by following the relationships between nodes. Previously, they did not have standard query language; however, query languages for graph databases are slowly maturing e.g., Cypher [45], SPARQL [46]. It is difficult to distribute the components of a graph among a network of servers as graphs become larger. Unlike other NoSQL databases, Graph databases are not cluster-friendly.

Usage: e.g., Facebook with more than two billion users. Other than social networking, graph stores are used in network and cloud management, security and access control management, logistics, and the manufacturing of complex goods like airplanes and rockets. Examples of Graph stores include Neo4j, MarkLogic, and Amazon's Neptune [47].

3. CAP Theorem of Big Data Era—Relationship of Polyglot Persistence with CAP Theorem

In the era of relational databases, a database without an ACID guarantee is impossible to accept. However, there is a big change in the basic assumption of the distributed system, especially in web services systems, in the big data era. The idea of being fault-tolerant is very different from the traditional belief of 100% fault-tolerance in the distributed system concepts. The concept now accepted by most is that failures are inevitable. Therefore, even in the event of failure, the services should still be operating by degrading performance elegantly [48]. This general idea of learning to embrace the failure in data centers and distributed systems makes NoSQL databases become acceptable in the current web service-oriented era. NoSQL databases work well with the CAP Theorem. The CAP theorem proved that only two out of three properties can be achieved in the distributed systems, although Network partitions are rare. Consistency in CAP is different from the "Consistency property" of ACID characteristics in SQL. SQL's Consistency guarantees data integrity for all database connections (e.g., using a two-phase commit). However, Consistency in the CAP theorem is related to how fast data changes appear in a distributed system. The meanings of CAP characteristics are summarized in Appendix A. When partitions are present, availability can be enhanced by permitting the soft state and eventual consistency in the system. Regardless of the ACID of SQL or BASE, in NoSQL has to face the challenges with big data's volume, variety, and velocity. The CAP Theorem and its associations are depicted in Figure 3.



Brewer's CAP theorem (Consistency, Availability, and Network Partitioning)—conjectured in 2000, proved in 2002, revised in 2012, and extended in 2012—is inherent in the nature of the distributed systems during network partitions.

- In Brewer's CAP conjecture, only two out of three properties can be achieved—CA or CP or AP.
- CAP conjecture is proved as CAP theorem in 2002 that consistency or availability is chosen in the presence of partition tolerant.
- Brewer also reminds us not to blindly forfeit partition tolerance and states that the core of CAP is partition recovery time—(to decide whether to emphasize consistency or availability) and added latency problem.
- The PACELC, an extension of CAP theorem, showed the compromise between consistency and latency due to replication in detail.
- Data-intensive systems now offer strong consistency and best effort availability or strong availability and best effort consistency or sacrifice both C and A for better performance.

Figure 3. CAP Theorem and its associations.

3.1. BASE Descending from the CAP Theorem

Traditional DBMS research is mostly about ACID properties. The "C" and "I" in ACID can be exchanged for "Availability" as a fundamental tradeoff with graceful performance degradation in CAP.

The CAP theorem is proved using two network models: asynchronous and partially synchronous. Distributed web services need to be transactional (atomic, linear), available, and fault-tolerant. In a semi-synchronous network, there will be inconsistency in read/write operations when message communication is lost. However, this semi-synchronous model can limit the time, i.e., how long inconsistency continues [49]. These read/write operations will become consistent after some time. This conceptualizes the "eventually consistent" property of BASE. "The use of commutative operations, which make it easy to restore consistency after a partition heals [50] the concept of "delayed exceptions" and allows more operations to be considered commutative and simplifies eventual consistency during a partition".

Put simply, data will eventually become consistent, although they are currently inconsistent, displaying stale data in some partitions. The CAP theorem introduced the BASE alternative instead of ACID properties in a distributed system, i.e., the "Basically Available", "Soft-State", and "Eventual Consistency" properties of NoSQL databases [51]. BASE differ mainly from ACID properties by assuming that the system might be in different states. Even so, these different states will eventually be consistent after a point in time. Eventual Consistency is enough for most applications, except those that need high availability and low latency, i.e., applications that need fast reads and writes. The characteristic of the BASE is summarized in Appendix B.

CAP theorem with ACID and BASE characteristics are deeply interrelated—orienting consistency and availability respectively. Horizontal scaling of the NoSQL database system is concerned with data partitioning tolerance known as sharding. A shard is an individual partition of data. In general, it can be considered that most NoSQL databases can fulfill either CP or AP using different consistency levels. SQL databases can be considered as having a CA property (without partition tolerance, as they do not provide it inherently). The ACID property can be fully fulfilled if there is no partition in NoSQL data stores. If partitions are present, then improving availability is possible by allowing soft state and eventual consistency.

There are two reasons why the relationship between CAP and ACID is more complex. Firstly, CAP and ACID contain the same constituents but have different concepts. Secondly, availability can affect only some parts of ACID properties. In CAP, each partition can have atomic operations when availability is oriented, as these operations simplify recovery. The "Consistency" property in CAP and ACID is often misunderstood as the same—but they have different meanings. CAP's Consistency simply means "atomic consistency", referring to single-copy consistency—a property of a single request-response sequence that is a subset of ACID consistency. Consistency in ACID refers to the transaction preservation of all database rules, such as preserving unique keys.

Fulfilling ACID's C in the relational model requires a general range of mechanisms for data and transactions, including referential integrity, data integrity, constraint integrity, and Multi-version concurrency control (MVCC). Therefore, the ACID Consistency property cannot be fulfilled across partitions in CAP, i.e., to maintain the data invariants. Partition recovery needs to restore ACID consistency but restoring invariants and restricting operations during recovery is problematic. Nonetheless, designers chose the soft state due to the expense in maintaining durability. By detecting and correcting durable operations in partitions, durability can be reversed during partition recovery. Overall, having ACID transactions in partitions facilitates easier recovery and provides a framework for compensating transactions. Due to these reasons, both ACID and BASE databases are used in the appropriate parts of data-intensive big data systems to fulfill big data characteristics.

3.2. CAP Compromises

Although original CAP theorem suggests three design options—CA, AP, and CP. Many NoSQL data stores have opted for AP systems (that is, sacrificing Consistency at different levels in practice. CAP partitions and latency are closely related. Time out has to be considered for partitions and partition effects have to be mitigated into Consistency and Availability. How to balance these properties is dependent on each application requirement. In general, every distributed big data system has to make a tradeoff between availability and consistency, i.e., how to trade consistency (harvest) for availability (yield) [52], in a way, the best-effort behavior for consistency, within the latency bounds. Consistency, Availability and Partition Tolerance properties can, in reality, be a continuum. Choices can be made "ad hoc", e.g., sacrificing consistency can gain faster responses in a more scalable manner.

PACELC is an extension of the CAP theorem and considers that "as soon as a storage system replicates data, a tradeoff between consistency and latency arises. it exists even when there are no network partitions." [53] PACELC makes the compromise between Availability and Consistency if there is a Partition (i.e., PAC), or Else makes the compromise between Latency and Consistency if there is no partition (E for Else, LC). The tradeoff between Availability and Consistency has to be considered based on CAP's Partition (i.e., P = Partition tolerance + the existence of a network partition itself). DBMSs differ on which properties they are oriented towards—PA or PC (availability oriented or consistency when a partition occurs), and EL or EC (oriented lower latency or consistency in normal operation when there is no partition).

Some of the researchers, such as Stonebreaker, do not agree with degrading consistency to be able to work in a distributed database. They suggested that once consistency is compromised, other characteristics of ACID have the possibility to be compromised [54]. Therefore, they suggest another solution: NewSQL. NewSQL maintains ACID characteristics; however, these characteristics can work in a distributed systems environment with heterogeneous types of data. C-Store [55] is a columnar store which does not compromise the BASE. VoltDB [56] is an instance of a NewSQL database. However, this paper will assume NewSQL as a type of NoSQL with ACID characteristic for the sake of simplicity. Therefore, a NoSQL database may or may not have BASE characteristics. NoSQL systems exhibit the ability to store and index arbitrarily big data sets while enabling a large number of concurrent user requests.

Different types of databases—NoSQL, RDBMS (SQL), and NewSQL (new types of databases without giving up ACID but not relational SQL databases)—are categorized based on their characteristics—for instance, NoSQL—Dynamo, Cassandra, Riak as (PA\EL) systems, BigTable, HBase as (PC\EC), MongoDB (PA\EC), VoltDB as (PC\EC), NewSQL's H-Store, Megastore as (PC\EC) systems. One mistaken assumption is that distributed DBMSs need to reduce consistency in the absence of a partition. The real CAP permits complete ACID with high availability when there is no partition. New SQL databases follow full ACID characteristics. The tradeoff between consistency and latency occurs as soon as replication is done for high availability.

CAP is just a small inlet in the ocean of distributed system theory. CAP was initially based on a wide area network. Because, at present, the data world is heading toward wireless sensor networks, more unreliable (unpredictable) message losses and latencies will become commonplace. The CAP theorem should also be considered from this new network perspective [57].

3.3. Varying Consistency Levels, NoSQL, SQL (Strong Consistency) and Polyglot Persistence

Different databases have different levels of consistency, especially those in NoSQL. Most of them offer a form of eventual consistency [58]—either eventual consistency or strong eventual consistency [59]. Eventual consistency generally means that all data items in all replicas become consistent if there is no new update. A data store is considered to have strong eventual consistency if two replicas of a data item that applied the same set of updates are in the same state. ACID ensures that the transactions in the consistent states guarantee integrity. RDBMS usually works with some form of strong consistency, such as linearization (illusion of a single copy) or serialization (transactions work in the form of a serial order), or both. NoSQL databases try to provide at least eventual consistency.

When observing data updates from the client side, consistency can be roughly categorized into strong consistency and weak consistency (with an inconsistency window). Eventual consistency is a specific type of weak consistency that can be further divided into causal consistency, read your Write, session consistency, monotonic read consistency, and monotonic write consistency. These properties can be combined, e.g., monotonic read with session-level consistency. A popular combination is the monotonic read and reads your writes found in the eventual consistency system. As the CAP theorem implicitly suggests the relaxation of transactions, which make it possible to lower the bar from the stringent strong consistency level, it becomes art for the NoSQL data stores to choose the appropriate level of consistency in order to offer the best performance or meet the requirements of the target system. The Read/Write Consistency level in Cassandra can be tuned to meet the client's requirements.

The consistency of the future Internet [60] provides an example of NoSQL databases with varying consistency levels. Some of the NoSQL stores offer different levels of consistency e.g., Redis offers strong consistency for the single instance but eventual consistency in handling multiple instances. MongoDB allows for a choice between strong consistency and eventual consistency, which can be classified into "strong consistency, causal consistency, session consistency, and eventual consistency".

NoSQL databases can differ (adjust) between what they want to be oriented towards. For example, Cassandra is an AP that can, however, be tuned to CP when not in network partitioning. Neo4j provides causal consistency and oriented availability. Neo4j uses causal consistency and oriented availability, which resides in the CA part of the CAP theorem. However, it does not support partitioning. CAP theorem highlights that different parts of the application need different types of data stores and the CAP characteristics in them may have different nuances. When implementing polyglot persistence, the consistency level offered by data stores has to be taken into consideration.

4. Depicting Polyglot Persistence

4.1. No One Cure-Them-All Database Resulting in Polyglot Persistence

The long-term goal of the database community is to be able to manage all the varieties of structured, semi-structured, and unstructured data from repositories in enterprise and over the web, not just managing structured data with a well-defined schema. In pre-big data times, business intelligent systems could only rely on structured data manipulated by SQL databases for data insight. Since the arrival of the big data era, it is possible for them to retrieve more value (insight, hindsight, foresight) from previously untouched semi-structured and unstructured data by storing them in NoSQL databases. NoSQL databases are a maturing technology and not a cure-all solution. They have their own flaws.

Key-Values cannot be used effectively if (1) there are many different relationships of data, (2) if there are multikey transactions, (3) if the operations are by sets, and (4) if queries are executed on values. Columnar is not suitable when the schema keeps on changing, and the query uses heavy SUM and AVERAGE operations. There is no sophisticated filtering query; join operations have to be handled by code, and no mechanism for supporting multi-record consistency. Document stores are not suitable for applications where queries change frequently and graph stores only work well only when the applications have complex relationships to handle. NoSQL stores are non-transactional, thereby creating difficulties in the advent of failures and/or concurrent updates. The lack of atomicity causes the problem in multiple rows of updates [61]. As mentioned earlier, time or place specific applications require the use of temporal and spatial databases, which is still the Achilles' heel for both SQL and NoSQL.

There have been debates about which database type is better—SQL or NoSQL. There are attempts to completely dominate the market both from current SQL vendors and new NoSQL vendors. The main strategy is trying to extend its data models by adding more functionality to handle more types of data. Many vendors are trying to enhance their capabilities. For example, the SQL-Server 2016 (Prominent SQL vendor) and MarkLogic (emerging NoSQL vendor) added more properties to handle time in their databases.

Existing computational techniques can still be applied in (sometimes with extensions) in Big Data problems such as logical data independence and a uniform and powerful SQL query language from RDBMS. The challenge is to combine the substantial features of relational models with novel solutions for handling big data challenges. The most NoSQL provide for now is BASE although many NoSQL vendors are trying hard to enhance their properties to near ACID (trying to give the feeling of using one central store) as much as possible. There has not existed any database that could handle all types and structures of all data, until now. All in all, there is currently no "cure-all solution" database or data stores. Upon this reflection, approaches to polyglot persistence implementation need to be considered. Table 1 shows the sequence of the step leading to Polyglot Persistence.

Table 1. The sequence of the Steps leading to Polyglot Persistence Review.

Source Conjecture	Different file system ➔ (specialized) different type of data stores, ➔ polyglot persistence [generality]
Formulation	Related theory—CAP Theorem ➔ derived theorem—PACELC, different consistency models, relationship with transactions ➔ BASE or ACID of different data stores ➔ Polyglot Persistence
Methods	Current Solutions for achieving Polyglot PersistenceSuggestions to improve current solutions for Polyglot Persistence
Comparison	Polyglot persistence characterization, big data architectures comparison and possible co-operations

4.2. *Evolution of Polyglot Persistence*

Instead of using one type of data store and one type of consistency, Polyglot Persistence is oriented toward flexibility and extensibility, i.e., polyglot persistence should be oriented towards the "easy to change" aspects of data, applications, and data stores. The original meaning of polyglot persistence is using suitable NoSQL data stores in different parts of one system. One part of the applications process data and output the result being stored to suitable data stores. Another part of the system takes that data to process and outputs another result to store in another type of data store. As most NoSQL stores abandon join operations, most of these are done in aggregate execution.

NoSQL databases are based on set theory (Graph databases works with graph theory and DAG for operations). NoSQL databases have weak consistency. Some of them give last write wins instead of isolation. Transactions are managed by application level and are based on Complex code (not all NoSQL databases have a defined query language). Many of them are based on a Simple MapReduce function. They provide a flexible Schema (easy evolution), Scale-out (theoretically unlimited scalability), Shared-nothing (parallelizable), fault-tolerant (by replication) [62]. Their conceptual model and logical model are often combined; further, their code and physical models are combined.

As they are different in nature, their usage will become different. For example, ACID-based relational (SQL) databases, such as PostgreSQL, are safe for bank transactions of online shopping websites. This database is not suitable for an online shopping cart because users have a very short attention span. However, light-weight NoSQL key-value databases, such as Redis, are suitable for a shopping cart because they are availability-oriented. Therefore, more than one single database type needs to be applied in just one application domain.

4.2.1. The simplest form of Polyglot Persistence

As mentioned before, currently, there is no single SQL or NoSQL database that can fulfill the requirement of big data—volume, variety, and speed—for winning the CAP theorem, although both SQL and NoSQL vendors are striving very hard by adding more properties to become cure-all data stores. However, there is not one that stands out from the rest, as each has its own advantages and disadvantages. As an alternative, the use of different database types together in one system has become prevalent. Polyglot Persistence is simple at its core—use appropriate data stores in appropriate parts of the system [63]. An example of a Polyglot Persistence E-commerce application [64] is shown in Figure 4. Availability-oriented activities, such as a shopping cart, are stored in a key-value store. Financial data that require ACID and are oriented toward strong consistency are stored in a relational data store. However, this process is easier said than done. We will discuss how to choose appropriate data stores in the next section.

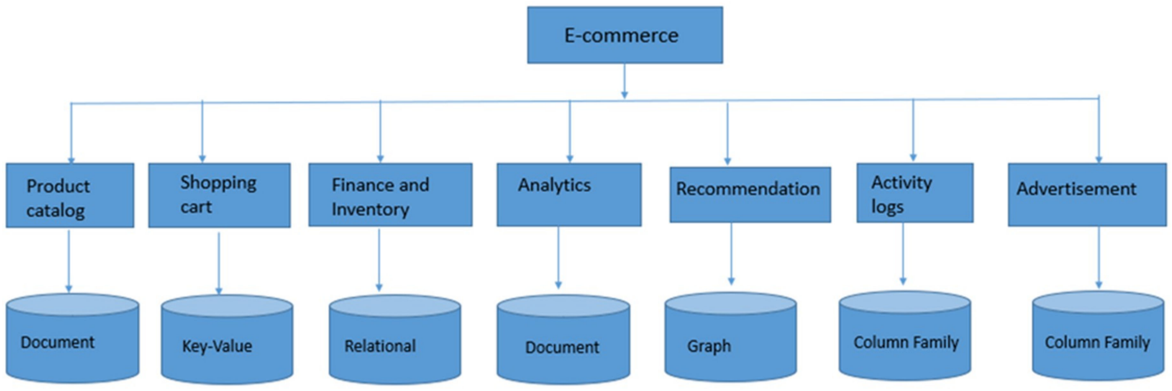


Figure 4. E-commerce System Example for Polyglot Persistence [40].

4.2.2. Choosing NoSQL Databases for Different Parts of the System

The popularity of new applications, like social networking, and the concept of making everything digitalized drive the emergence of NoSQL with diverse data model categories. Therefore, determining which database is proper (SQL's relational or NoSQL's data models) for a specific domain will merely depend on the requirements of the target applications. There are multi-model databases that provide integrated features of these NoSQL databases e.g., OrientDB, although they are not at the level that can solve for polyglot persistence. Each SQL and NoSQL database have their own advantages and weaknesses. NoSQL databases have some similar features, such as no requirement for fixed rows in a table schema, having horizontal scaling, having the ability to increase performance by connecting more nodes together to work as a single unit, elimination of join operation for better performance, relaxing concurrency models, accepting multi versions concurrency, being schemaless by allowing dynamically adding new fields to arbitrary data records, and utilizing shared-nothing architecture, i.e., many NoSQL databases, except graph store, are designed to work on a cluster of commodity servers sharing neither RAM nor disks.

When choosing which NoSQL databases to be used for the target applications (or different parts of one application domain), [65] one must give the criteria for considerations: Data Model, CAP Support, Multi Data Center Support, Capacity, Performance, Query API, Reliability, Data Persistence, Rebalancing, and Business Support. Based on these considerations, appropriate NoSQL databases for storing datasets of the specific domain have to be chosen from technical, business, system domain, and environmental viewpoints. NoSQL databases differ and have to be chosen for appropriate data stores for use in the target application's domain, e.g., graph stores for social media applications. Not only types of NoSQL have to be chosen, but which NoSQL products in that type will be used must also be determined, as they are best suited in different data structures (representative of the variety feature in big data), e.g., MongoDB or MarkLogic as document stores, or HBase or Cassandra to be used in the system as a Columnar. Maintaining all available replicas eventually produces a significant performance overhead. They also introduced the idea of orienting AP or CP as requested by the user. Relational databases, which use SQL as the standard query language, are based on relational theory. These databases work with nested transactions and offer strong consistency. They have difficult evolution because of their defined schema, scale-up (limited), shared-something (disk, memory, processor) and clear separation between their conceptual models, logical models, and physical models (code and query are separated).

NoSQL databases are used for different reasons [66]. To store semi-structured data that has a high degree of variability, flexible NoSQL data models are used, e.g., key-value data stores (schemaless—to store tuples arbitrary schemas). NoSQL databases also fill the need for a different query language or API to access the data in a more comfortable manner, e.g., MongoDB's API to query semi-structured documents. To avoid a myriad of exchanges between the client and server side needed in a relational database, a NoSQL with graph traversal model is used. For instance, Graph databases can perform traversals of a graph with a single invocation avoiding abundant client-server communication using DAG. The most prominent use of NoSQL is to fill the scalability (scale in—scale-out) void of SQL databases in distributed systems. Specific technical research areas for NoSQL include design patterns (how to represent connections of data from the real world), two-phase commit (for atomic operations), graph partitioning (by Data Mining and conceptual modeling), developing standard query languages, and visualization. When implementing polyglot persistence with different stores, it is very important to know the nature of the subsystem to decide the required consistency level, then decide the required data stores type for that subsystem. After taking its consistency into consideration, the system can be developed such that the application code can fill the gap to handle the inconsistency in the best way.

4.2.3. Polyglot Persistence Implementation Examples

There is no single SQL or NoSQL database that can fulfill the requirement of big data, and there is no cure-all data multi-model data store or universal query language for all data stores. As a result, Polyglot Persistence features, i.e., using the different types of data stores in subsystems of a single data-intensive system, have become prevalent. Nowadays, not only big data-intensive systems but even small- or medium-sized web applications have to use more than one type of data store—appropriate data stores in appropriate parts of the system. Polyglot Persistence becomes inevitable. Reference [67] is a proof-of-concept "prototype implementation" of polyglot persistence applied to Healthcare Information Systems. polyHIS [68] is a real-world polyglot-persistent framework that combines relational, graph, and document data models (use of different types of databases in different parts of the healthcare information system) to accommodate the information variety nature of the domain. A Polyglot Persistence concept can also be used to enhance the performance of the Energy Data Management System (EDMS) [69]. The simplest implementation of Polyglot Persistence is very domain specific, as the code is written at the application level to work for different types of databases in different parts of one data-intensive big data system.

There are other studies on easier implementations of Polyglot Persistence. Reference [70] demonstrates a hybrid data store architecture using a data mapper (a set of workers, a data store wrapper object factory, data store wrapper repository, and a connection pool) for mapping application data to its destination data store. In polyglot persistence, different types of databases have different schema types. The schema shift from one type of database to another is handled by application source code. However, this is very domain specific. ExSchema is a tool that read code based on the project structure, update methods, and the declaration of local variables, and changes it to another schema [71]. This process automatically discovers the external data schemas from the source code for polyglot persistence applications.

A major concern of polyglot persistence is how to query data across different data stores that provide different query languages and have different data models. The CoherentPaaS project studies the use of the federated query languages, such as CloudMdsQL [72] for querying these heterogeneous data stores. CloudMdsQL, a functional SQL-like language, can query both SQL and NoSQL databases with a single query. An example implementation is demonstrated with graph, document, and RDBMS. The query is translated into the native queries of targeted data stores and optimized. These databases are still in initial prototype stages in the PaaS cloud structure and have not covered every NoSQL type yet. However, the idea to develop a universal query language as "the golden rule" for both SQL and NoSQL may be the solution to polyglot persistence. Polyglot Persistence is one of the de facto solutions in the current digital era for manipulating a wide range of databases to handle different types of data. CoherentPaaS provides "scalable holistic transactions" across SQL and NoSQL data stores, such as document stores, key-value stores, and graph stores. The CoherentPaaS project is intended for the development of a single query language and a uniform programming model. It tries to provide ACID-based global transactional semantics for the assistant of Polyglot Persistence. [73] introduces the concept for a Polyglot Persistence Mediator (PPM) as a new kind of middleware service layer, which allows for runtime decisions on routing data to different backends, according to schema-based annotations for efficient use. Automated polyglot persistence is illustrated using online newspaper publishing as an example, based on service level agreements defined over the functional and non-functional requirements of database systems.

4.3. Approaches to Achieve Polyglot Persistence

Reference [74] gives the three different types of polyglot persistence approaches found in research and industry: the application-coordinated polyglot persistence pattern, polyglot persistence microservices, where storage decisions are made from loosely coupled services, and polyglot database services, which forward requests based on distribution rules. However, the authors summarize the current approaches to achieve polyglot persistence into three general ways, based on solution in which they are oriented: (1) domain (similar with application), (2) query language, and (3) other solutions, such as framework, middleware, or developing multi-model data stores.

4.3.1. Domain-Oriented Solutions

In domain attribute-based solutions, the most important domain attributes (features) can be retrieved from the most important functions of the system with the help of domain experts or using feature selection methods. The nature of this sub-system or function is to decide which is the best-suited data store for them. This domain-oriented approach will first decide the features, then decide the types of data stores suitable for them based on the functions of the subsystems, and then use them as appropriate. This is currently the most used polyglot persistence implementation style. In legacy systems, domain code can be read to decide which type of data stores are suitable when reengineering the system.

4.3.2. Query Language-Based Solutions

There are suggestions that a declarative query language for NoSQL data stores should be developed to solve these problems once and for all [75]; however, there is not enough research in that direction. There are two possible ways to approach polyglot persistence from query language. One of the ideas is to develop a universal query language and another is to develop a query converter to transform queries from one database model to another. If there were a universal query language for all data stores, the polyglot persistence model would have been solved. However, such a solution has not existed yet, despite the attempt to do so. Reference [76] suggests the establishment of a declarative query language for NoSQL databases. It further divided databases into analytical and operational, with NoSQL and NewSQL. As mentioned in Section 4.2, there are solutions, which perform like some query language converters, that convert from one query language to another; however, these solutions are still in the embryo stage and mostly convert from one or two query languages to another.

4.3.3. Other Solutions: General Framework/Middle Ware/Multi-Models

The Apache Drill [77] can be considered a general framework for handling polyglot persistence. Apache Drill is not considered a database but rather a query layer that works with several underlying data sources. Apache Drill transforms a query into human-readable syntaxes, such as standardized SQL, or non-standardized MongoQL, into a logical plan using a query parser. It is transformed into a physical plan for execution based on data sources and dataflow DAG.

The Polyglot Persistence Mediator (PPM) described above makes runtime decisions on routing data at different backends using schema-based annotations. The unified REST interface, which takes into account the different data models, schemas, consistency concepts, transactions, access-control mechanisms, and query languages to expose cloud data stores through a common interface without restricting their functionality, is also suggested [78].

Reference [79] gives suggestions to build a unified system to query, index and update multi-model data in a unified fashion. It suggests the systems should have model-agnostic storage, multi-model query processing, and model-agnostic transactions with the in-memory data structure. It also points out the main challenges for building a multi-model system: diversity (query optimization and transaction model mainly work on a single model), extensibility (to identify the boundary of data—several types of data, i.e., relation, JSON, XML and graphs, data based on time time-series, streaming) and flexibility (a schema should be automatically discovered, schema-less for storage, and schema-rich for queries). Overall, there are many open research areas to develop possible ways for polyglot persistence implementation.

4.4. Polyglot Persistence in Big Data Architectures

Data in the use cases were typically collected into databases or log files. Relational databases were applied for storage of important user-related data (Facebook's MySQL, LinkedIn's Oracle). NoSQL databases or in-memory stores were used for intermediate or final storage of data analysis results (LinkedIn's Voldemort, Netflix's Cassandra, and EV-cache). Special log facilities were used for the storage of stream-based data (Facebook's Scribe, LinkedIn's Kafka, Netflix's Chukwa, and the Packet capture driver in the Network measurement case). Structured data was typically saved into a distributed file system (e.g., Hadoop HDFS) [80].

Big Data architecture suggestions for data-intensive applications all utilize polyglot persistence. Renowned big data architectures to conquer (or to alleviate) the CAP theorem—both Lambda Architecture [81] and Kappa Architecture [82]—accept polyglot persistence. For handling data, the original example implementations use different kinds of databases in their systems, e.g., Twitter (Lambda) and LinkedIn (Kappa). They not only accept polyglot persistence, they even try to leverage data from transitional data stores and analytical data warehouses.

Lambda separates batch layers for historical data—using the batch processing frameworks, such as Hadoop, and real-time layers for real-time data—using streaming frameworks such as Storm. The data stores at the batch layer could be anything appropriate for the systems. The serving layer ensures that data in the batch layer are historical data. Kappa architecture does not separate layers for handling batch and real-time. This architecture uses the window to decide the time frame of old or new data, storing data in one layer, such as using Kafka based on ordered logs. The processing is done on all the stored data (first job). If there is new code, the processing is done from the start of the retained data (second job), and the first job is deleted (the data is processed by them) when the second job is caught up. All transactions stored in data stores will only have CR—create, replicate operations—instead of very basic CRUD—Create, Replicate, Update, Delete—operations. A significant deviation from classic CRUD is that there are no update operations. Only new data records will be added to the datasets, which signals that new updates have been made in the record. Delete operations will be handled when databases (or warehouses) are purged (e.g., garbage collection).

Both architectures use different kinds of databases in different layers, e.g., Lambda. However, it has not been researched yet as to which architectures are more suited for which types of systems, which kinds of database combinations are more suited for which specific applications, and how polyglot persistence can be done with them. Their advantages and disadvantages are still left as open research areas for academics and professionals. These architectures accept data immunity concepts and agree that code changes are inevitable. They seek to retrieve the ultimate query operations as query = function (all data) and make a separation between data, data stores, and processing. As the focus of the discussion—both of these innovative architectures implicitly apply the polyglot persistence successfully—especially making the work done by application code: "In the foreseeable future, relational databases will continue to be used alongside a broad variety of nonrelational data stores—an idea that is sometimes called polyglot persistence" [83].

However, applying polyglot persistence in data-intensive (big data) system architecture is not without difficulty. Carrying data from these different types of database sources is already a very obvious challenge. For example [84], Kappa architecture uses logs for integrating data from different databases and systems. Building data pipelines for dealing with different types of data stores for data ingestion needs extra attention, as these pipelines can easily result in complicated structures and make maintenance difficult. Therefore, Kappa architecture relies on a unified log (a general pipeline) that will consume data from every data store for scraping data. The current business area that is applying Kappa architecture, such as the Telco industry, takes into consideration the building of these data pipelines before implementing their prized advanced analytics algorithms [85]. Reference [86] provides an approach and an algorithm for schema and data transformation to support dynamic data storage and polyglot persistence: "The approach uses an intermediate canonical model to ensure the flexibility and extensibility of the implementation. The transformation algorithm is implemented as a Lambda architecture with a batch and speed layer to support live applications without downtime and the need for code changes".

5. Discussion

Polyglot Persistence is applied in Lambda and Kappa architectures implicitly, as these architectures extract and process data from different types of databases appropriate for different parts of the system without adding further complexity. Reference [87] suggested a new term called "polyglot processing", as the attention is shifted onto how data is manipulated and queried as polyglot persistence has been an accepted concept. It also pointed out that "Polyglot Persistence" in reality is an idea that is yet to mature. Using different types of databases in different parts of the system illustrates the need to handle data integration and mitigation and the need to separately perform data maintenance for each type of database. Data integration require different databases to communicate with each other. Access control and authentication of the NoSQL data stores are also yet to be improved. Every software engineer has been encountering data integration and mitigation problems since SQL. NoSQL databases have more challenges than SQL databases as NoSQL databases have to make critical choices for implementation even within the same NoSQL family (e.g., should one implement Cassandra or HBase columnar for this system?), thereby maturing the query languages of NoSQL due to the lack of a globally reliable query language. Data integration for different data sets and data mitigation from one system to another (e.g., Hive for mitigating from SQL to NoSQL) need to be specially handled. Because neither a unified query language nor a multi-model has yet been developed, the widely used polyglot persistence method of domain code-based solutions, i.e., dividing different parts of data stores for different parts of the system, has to be used.

Currently, polyglot persistence has been implemented based on the domain system, i.e., the domain system is divided into different parts, each part using appropriate data stores. These domain code-based solutions for implementing polyglot persistence can be improved as follows. After deciding or retrieving the most important features from the target domain, develop a system structure style that can easily change from one database data store to another and track the possible feature evolution over time. When dividing the different parts of the system, consider their requirements for CAP in each subsystem and decide the best database for that part (decide the type of the data store to use in that subsystem, e.g., a key-value for sale transaction records, a user profile for document store type, RBDMS for financial). Then, consider which database solution to use (or database configuration if the database solution is already decided). This choice should be based on the main function of CAP characteristics and the required consistency in transactions (e.g., Voldemort for key-value sale transactions for easier market basket analysis, MongoDB for user profiles, and PostgreSQL with ACID compliance for some kind of strong consistency models for instance serializable). Reference [57] suggests different ways for the system to be segmented into components that provide different types of guarantees to circumvent the CAP dilemma. Consistency is a safety property and Availability is a liveness property. Systems can be partitioned into different dimensions. Data partitioning can be different for different types of data: shopping cart for Availability, billing for strong consistency, operational partitioning (availability for read-only operations), functional partitioning (services can be divided into subservices with their own requirements), user partitioning (social network applications try to partition users to gain high availability between friend groups), and hierarchical partitioning (different levels of performance—better availability for top-level—root, better consistency for lower level—leaves). Finally, the domain application code should assist the required transactional consistency. However, it may lead to simpler or more complicated manipulation of domain code. In legacy applications, it is also possible to decide what type of data stores is most suitable by reading the domain code. The code would be interpreted to figure out which type of data stores it used based on the domain features, code structure, and functions.

Polyglot persistence, which implements different suitable databases in different parts of a system because of data variety and complexity, is becoming prominent. For now, Polyglot Persistence can be considered a de facto for Big Data systems with acceptance in Lambda and Kappa architectures. These Big Data architectures not only accept polyglot persistence, but they also try to leverage the transactional data stores and analytical data warehouses to extract surplus value. This encourages possible database repository change such as data lake [88], architecture changes such as lambda and kappa, and the acceptance of different types of databases, i.e., "polyglot persistence" in the big data era. However, the future of Polyglot persistence still has many challenges. Disadvantages of the Polyglot Persistence approach are described in [89]. There is no unified (standardized) query or query language, and no consistency guarantee, i.e., cross-database consistency is particularly difficult because of referential integrity. Interoperability is difficult because of new versions of different databases, and they also have a different integration layer. Logical redundancy is difficult to avoid. Logical redundancy increases administrative overhead and has a more complex configuration, e.g., a security configuration. This process recommends a polyglot persistence approach "only if several diverse data models have to be supported and the maintenance overhead can be managed".

Getting information from heterogeneous sources with multiple formats is polyglot persistence and integrating mitigation between different types of SQL and NoSQL, and the development of a universal query language for all types of databases is an open research area for polyglot persistence. Figure 5 illustrated the possible ways to implement polyglot persistence. Currently, the possible ways to achieve polyglot persistence can be summarized as using application code to handle different types of databases in data-intensive systems, e.g., by using application code or with lambda and kappa architecture, developing unified query language as a golden rule for new emerging NoSQL database types, e.g., CloudMdsQL, from CoherentPaaS, and developing frameworks for query translation: to translate from one type of NoSQL query language to another, e.g., Polyglot Persistence Mediator (PPM) or developing multi-model solutions or general frameworks, such as Apache Drill, or middleware, such as PPM. However, most of these approaches are still in their infancy and left as open research areas.

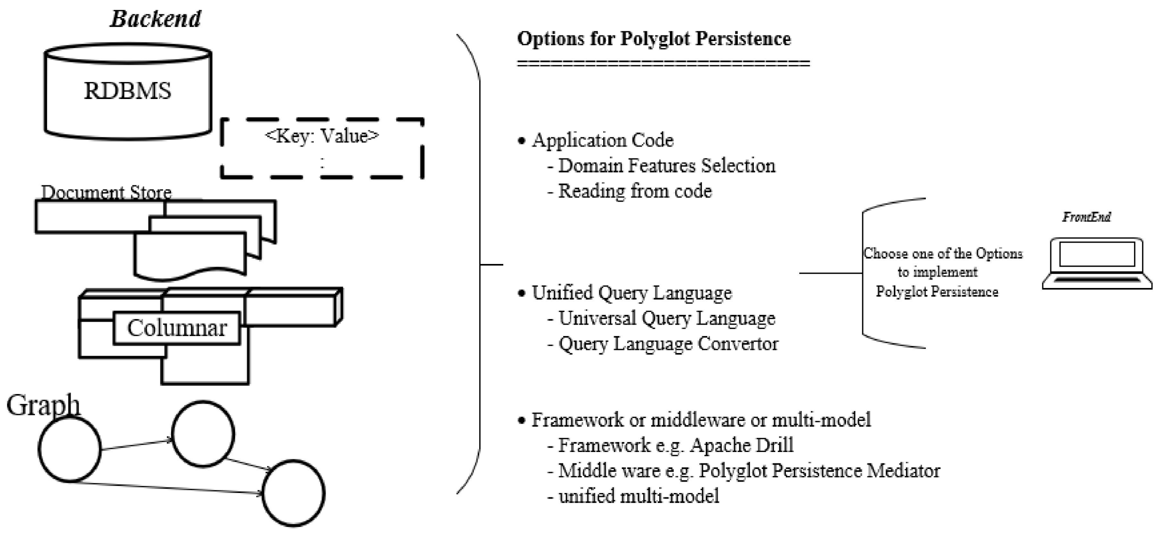

Backend
RDBMS
<Key: Value>
:
Document Store
Columnar
Graph
Options for Polyglot Persistence
• Application Code
- Domain Features Selection
- Reading from code
• Unified Query Language
- Universal Query Language
- Query Language Convertor
• Framework or middleware or multi-model
- Framework e.g. Apache Drill
- Middle ware e.g. Polyglot Persistence Mediator
- unified multi-model
Choose one of the Options to implement Polyglot Persistence
FrontEnd

Figure 5. Approaches to implementing Polyglot Persistence.

6. Conclusions

When looking at the overall big data landscape from a bird's eyes view, it can be stated that: 1. Concepts are changing and evolving over time; 2. technologies are constantly evolving, and their basic concepts can be changed from time to time, and 3. synergy effects between the academic and business worlds are more obvious. Theorems behind big data are evolving constantly. There are now revolutionary ideas, such as assuming failures are inevitable in distributed systems and developing mechanisms for handling these failures. CAP conjecture and its theorem have been proven to drive the emergence of distributed NoSQL databases; with this extension, empowering the concepts behind these distributed databases becomes more wide-ranging and complete.

NoSQL data stores are continuously evolving and enhancing their capabilities, just as SQL does in big data landscape, for handling heterogeneous, varied, and complex data. The development behind big data technologies has more obvious synergy effects both from academic and business fields. Access control and authentication of the NoSQL data stores are also yet to be improved. Every software engineer has been encountering data integration and mitigation problems since SQL. NoSQL databases have more challenges than SQL databases as NoSQL databases have to make critical choices for implementation even within the same NoSQL family (e.g., should one implement Cassandra or HBase columnar for this system?), thereby maturing the query languages of NoSQL due to the lack of a globally reliable query language. Data integration for different data sets and data mitigation from one system to another (e.g., the hive for mitigating from SQL to NoSQL) need to be specially handled.

The CAP concept allows the new types of distributed databases to decide which one can handle scalability reliably. However, NoSQL is not the answer to all requirements of the distributed system, either. NoSQL handles scalability but cannot guarantee the security, reliability, and integrity like SQL databases. Although database vendors from both the SQL and NoSQL sides are trying hard to one day become a cure-all solution, implicitly or explicitly, neither SQL nor NoSQL is currently the perfect solutions. No cure-all databases have yet appeared. CAP theorems drive the emergence of NoSQL databases. In addition, data variety encourages the use of different types of databases in different parts of the application domain, e.g., Polyglot Persistence in the big data era. Concepts and mechanisms accepted today in big data may be left behind and replaced, adapted, or combined with new ones. Therefore, embracing and adapting theories and technology changes is the correct mindset for big data in this digital universe.

Funding: This research was funded by the National Key Research and Development plan 2017 for High-Performance Computing, grant number 2017YFB0202303.

Acknowledgments: Author thanks the supervisor Zhao Shun Wang for all the understanding and encouragement.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Appendix A. CAP Characteristics

- **Consistency:** All clients will access the same (consistent) data at the same time. The same view of the data is provided to users by propagating the most recent updated version of data to all nodes, even if data are on replicated partitions of the database. If data are inconsistent in all nodes, the user has to wait until the latest version is available (maintain consistency of data).
- **Availability:** Non-filing nodes in the system will always give a response to a request. Every client can always read and write to the databases, even when there is a failure in communication (partitions are present) within replicated parts of the database. Availability is also the reason the data may or may not be consistent in all partitions resulting in different versions of data in different physical servers.
- **Partition tolerance:** Exception of total network failure, non-failing nodes in the working network partition must provide valid (atomic) responses. Segments of systems (non-failing nodes) work

well and give responses (whether consistency oriented or availability oriented) even in the presence of temporary communication failure across the physical network.

Appendix B. BASE Characteristics

- **Basically Available**: Data from a database (responses to a request) will always be available to a user when they are needed by replicating multiples copies of data blocks. The hindrances in ACID, such as deadlock, will be avoided. However, the received data may or may not be consistent in all nodes and may show multiple versions of data to different users who request the same data.
- **Soft State**: The system state might be changing to get an eventual state per operation even where there is no input to the database system. This "continue changing states" nature makes the system always remain in a soft state. Consistency responsibility is delegated to the application level.
- **Eventual Consistency**: The system will not guarantee the consistency of all data after each operation. Data in different nodes may not be consistent at all time. Instead, the received changes for data will eventually be propagated to all nodes in the system, after a point of time.

References

1. McKinsey Global Institute, Big Data: The Next Frontier for Innovation, Competition, and Productivity; 2011. Available online: https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/big-data-the-next-frontier-for-innovation (accessed on 15 April 2019).
2. Laney, D. 3D Data Management: Controlling Data Volume, Velocity, and Variety. Application Delivery Strategies 2001, No. February 2001; Gartner: 2001. pp. 1–4. Available online: https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf (accessed on 15 April 2019).
3. Al-Jaroodi, J.; Mohamed, N. Characteristics and requirements of big data analytics applications. In Proceedings of the 2016 IEEE 2nd International Conference on Collaboration and Internet Computing, IEEE CIC, Pittsburgh, PA, USA, 1–3 November 2016; pp. 426–432.
4. Vogels, W. Eventually Consistent. *Queue* **2008**, *6*, 14. [CrossRef]
5. Brewer, E.A. Towards Robust Distributed Systems (Abstract). In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, Portland, OR, USA, 16–19 July 2000.
6. Gilbert, S.; Lynch, N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News* **2002**, *33*, 51. [CrossRef]
7. Petersen, K.; Spreitzer, M.J.; Terry, D.B.; Theimer, M.M.; Demers, A.J. Flexible Update Propagation for Weakly Consistent Replication. *ACM SIGOPS Operat. Syst. Rev.* **2004**. [CrossRef]
8. Gehrke, J.; Ailamaki, A.; Hellerstein, J.M.; Franklin, M.J.; Kossmann, D.; Sarawagi, S.; Doan, A.; Stonebraker, M.; Carey, M.J.; Ioannidis, Y.E.; et al. The Claremont Report on Database Research. *Commun. ACM* **2009**. [CrossRef]
9. Polyglot Persistence. Available online: https://martinfowler.com/bliki/PolyglotPersistence.html (accessed on 5 Janauary 2019).
10. Polyglot Programming. Available online: http://memeagora.blogspot.com/2006/12/polyglot-programming.html (accessed on 5 January 2019).
11. Hwang, J.S.; Lee, S.; Lee, Y.; Park, S. A Selection Method of Database System in Bigdata Environment: A Case Study from Smart Education Service in Korea. *Int. J. Adv. Soft Comput. Its Appl.* **2015**, *7*, 9–21.
12. Hachem, N.; Helland, P.; Stonebraker, M.; Madden, S.; Abadi, D.J.; Harizopoulos, S. The End of an Architectural Era: It's Time for a Complete Rewrite. In Proceedings of the 33rd International Conference on Very Large DataBases, Vienna, Austria, 23–27 September 2007.
13. Dean, J. *Big Data, Data Mining, and Machine Learning*; Wiley: New York, NY, USA, 2014.
14. Stonebraker, M.; Çetintemel, U. "One Size Fits All": An Idea Whose Time Has Come and Gone. In Proceedings of the International Conference on Data Engineering, Tokyo, Japan, 5–8 April 2005.
15. Friends Don't Let Friends Build Data Pipelines. Available online: https://www.abhishek-tiwari.com/friends-dont-let-friends-build-data-pipelines/ (accessed on 10 March 2019).

16. Buyya, R.; Calheiros, R.N.; Dastjerdi, A.V. *Big Data: Principles and Paradigms*; Elsevier: Atlanta, GA, USA, 2016.
17. Cuzzocrea, A. Warehousing and Protecting Big Data: State-Of-The-Art-Analysis, Methodologies, Future Challenges. In Proceedings of the International Conference on Internet of Things and Cloud Computing ICC'16, Cambridge, UK, 22–23 March 2016.
18. Bachman, C.W. Software for Random Access Processing. *Datamation* **1965**, *11*, 36–41.
19. Codd, E.F. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* **1983**, *26*, 64–69. [CrossRef]
20. Haerder, T.; Reuter, A. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.* **1983**, *15*, 287–317. [CrossRef]
21. Banothu, N.; Bhukya, S.; Sharma, K.V. Big-Data: Acid versus Base for Database Transactions. In Proceedings of the International Conference on Electrical, Electronics, and Optimization Techniques, ICEEOT 2016, Chennai, India, 3–5 March 2016.
22. Kohler, W.H. A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. *ACM Comput. Surv.* **1981**, *13*, 149–183. [CrossRef]
23. Jagadish, H.V.; Naughton, J.F.; Doan, A.; Balazinska, M.; Abadi, D.; Dean, J.; Franklin, M.J.; Halevy, A.Y.; Widom, J.; Suciu, D.; et al. The Beckman Report on Database Research. *Commun. ACM* **2016**, *59*, 92–99.
24. Sadalage, P.J.; Fowler, M. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*; Addison-Wesley: Boston, MA, USA, 2012.
25. Storey, V.C.; Song, I.Y. Big Data Technologies and Management: What Conceptual Modeling Can Do. *Data Knowl. Eng.* **2017**, *108*, 50–67. [CrossRef]
26. Gobioff, H.; Ghemawat, S.; Leung, S.-T. The Google File System. *ACM SIGOPS Oper. Syst. Rev.* **2003**, *37*, 29.
27. White, T. *Hadoop: The Definitive Guide*, 3rd ed.; O'Reilly: Boston, MA, USA, 2012.
28. Shvachko, K.; Kuang, H.; Radia, S.; Chansler, R. The Hadoop Distributed File System. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST2010, Incline Village, NV, USA, 3–7 May 2010.
29. Dean, J.; Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the 6th Usenix Symposium on Operating Systems Design and Implementation (OSDI '04), San Francisco, CA, USA, 6–8 December 2004.
30. Thusoo, A.; Sarma, J.S.; Jain, N.; Shao, Z.; Chakka, P.; Zhang, N.; Antony, S.; Liu, H.; Murthy, R. Hive—A Petabyte Scale Data Warehouse Using Hadoop. In Proceedings of the International Conference on Data Engineering, Long Beach, CA, USA, 1–6 March 2010.
31. Thusoo, A.; Sarma, J.; Jain, N. Hive: A Warehousing Solution over a Map-Reduce Framework. In Proceedings of the 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), Long Beach, CA, USA, 1–6 March 2010.
32. Tudorica, B.G.; Bucur, C. A Comparison between Several NoSQL Databases with Comments and Notes. In Proceedings of the RoEduNet IEEE International Conference, Iasi, Romania, 23–25 June 2011.
33. Han, J.; Kamber, M.; Pei, J. *Data Mining: Concepts and Techniques*, 3rd ed.; Elsevier: Atlanta, GA, USA, 2012.
34. Redis. Available online: https://redis.io/ (accessed on 19 February 2019).
35. DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS Oper. Syst. Rev.* **2007**, *41*, 205. [CrossRef]
36. Sumbaly, R.; Kreps, J.; Gao, L.; Feinberg, A.; Soman, C.; Shah, S. Serving Large-Scale Batch Computed Data with Project Voldemort. In Proceedings of the 10th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, 14–17 February 2012.
37. Lakshman, A.; Malik, P. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Oper. Syst. Rev.* **2010**, *44*, 35. [CrossRef]
38. Why Cassandra Doesn't Need Vector Clocks. Available online: https://www.Datastax.Com/Dev/Blog/Why-Cassandra-Doesnt-Need-Vector-Clocks, (accessed on 10 February 2019).
39. Chang, F.; Dean, J.; Ghemawat, S.; Hsieh, W.C.; Wallach, D.A.; Burrows, M.; Chandra, T.; Fikes, A.; Gruber, R.E. Bigtable: A Distributed Storage System for Structured Data. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Seattle, DC, USA, 6–8 November 2006.

40. Vora, M.N. Hadoop-HBase for Large-Scale Data. In Proceedings of the 2011 International Conference on Computer Science and Network Technology, ICCSNT 2011, Harbin, China, 24–26 December 2011.
41. MongoDB. Available online: https://www.mongodb.com/ (accessed on 10 February 2019).
42. CouchDB. Available online: http://couchdb.apache.org/ (accessed on 10 February 2019).
43. MarkLogic. Available online: https://www.marklogic.com/ (accessed on 10 February 2019).
44. Nan, X.; Vicknair, C.; Wilkins, D.; Zhao, Z.; Chen, Y.; Macias, M. A Comparison of a Graph Database and a Relational Database. In Proceedings of the 48th Annual Southeast Regional Conference, Oxford, MS, USA, 15–17 April 2010.
45. Francis, N.; Green, A.; Guagliardo, P.; Libkin, L.; Lindaaker, T.; Marsault, V.; Plantikow, S.; Rydberg, M.; Selmer, P.; Taylor, A. Cypher: An Evolving Query Language for Property Graphs. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018.
46. Marx, E.; Saleem, M.; Lytra, I.; Ngomo, A.C.N. A Decentralized Architecture for SPARQL Query Processing and RDF Sharing: A Position Paper. In Proceedings of the 12th IEEE International Conference on Semantic Computing, ICSC 2018, Laguna Hills, CA, USA, 31 January–2 February 2018.
47. Amazon's Netptune. Available online: https://aws.amazon.com/neptune/ (accessed on 20 February 2019).
48. Robbins, J.; Krishnan, K.; Allspaw, J.; Limoncelli, T. Resilience Engineering: Learning to Embrace Failure. *Commun. ACM* **2012**, *55*, 40.
49. Brewer, E. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer* **2012**, *45*, 23–29. [CrossRef]
50. Brewer, E. A Certain Freedom: Thoughts on the CAP Theorem. In Proceedings of the 29th ACM SIGACT-SIGOPS, Zurich, Switzerland, 25–28 July 2010.
51. Gribble, S.D.; Chawathe, Y.; Brewer, E.A.; Gauthier, P.; Fox, A. Cluster-Based Scalable Network Services. *ACM SIGOPS Oper. Syst. Rev.* **1997**, *31*, 78–91.
52. Fox, A.; Brewer, E.A. *Harvest, Yield, and Scalable Tolerant Systems*; Hot Topics in Operating Systems: Rio Rico, AZ, USA, 2003.
53. Abadi, D. Consistency Tradeoffs in Modern Distributed Database System Design: CAP Is Only Part of the Story. *Computer* **2012**. [CrossRef]
54. MIT Prof. Michael Stonebraker: "The Traditional RDBMS Wisdom is All Wrong". Available online: https://blog.jooq.org/2013/08/24/mit-prof-michael-stonebraker-the-traditional-rdbms-wisdom-is-all-wrong/ (accessed on 10 January 2019).
55. O'Neil, P.; Cherniack, M.; Abadi, D.J.; Tran, N.; Lau, E.; Batkin, A.; Madden, S.; Chen, X.; O'Neil, E.; Stonebraker, M.; et al. C-Store: A Column-Oriented DBMS. In Proceedings of the 31st VLDB Conference, Trondheim, Norway, 30 August–2 September 2005.
56. VoltDB. Available online: https://www.voltdb.com (accessed on 10 February 2019).
57. Gilbert, S.; Lynch, N. Perspectives on the CAP Theorem. *Computer* **2012**, *45*, 30–36. [CrossRef]
58. Vogels, W. Eventually Consistent. *Commun. ACM* **2009**, *52*, 40–44. [CrossRef]
59. Shapiro, M.; Sutra, P. Database Consistency Models. In *Encyclopedia of Big Data Technologies*; Sakr, S., Zomaya, A.Y., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 591–601.
60. Diogo, M.; Cabral, B.; Bernardino, J. Consistency Models of NoSQL Databases. *Future Internet* **2019**, *11*, 43. [CrossRef]
61. Myter, F.; Scholliers, C.; de Meuter, W. A CAPable Distributed Programming Model. In Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Boston, MA, USA, 7–8 November 2018; pp. 88–98.
62. Cattell, R. Scalable SQL and NoSQL Data Stores. *ACM SIGMOD Rec.* **2011**, *39*, 12. [CrossRef]
63. De Araújo, A.M.C.; Times, V.C.; da Silva, M.U. PolyEHR: A Framework for Polyglot Persistence of the Electronic Health Record. In Proceedings of the 17th International Conference on Internet Computing and Internet of Things, Las Vegas, NV, USA, 25–28 July 2016.
64. Srivastava, K.; Shekokar, N. A Polyglot Persistence Approach for E-Commerce Business Model. In Proceedings of the 2016 International Conference on Information Science, ICIS 2016, Kochi, India, 12–13 August 2016.
65. Han, J.; Haihong, E.; Le, G.; Du, J. Survey on NoSQL Database. In Proceedings of the 2011 6th International Conference on Pervasive Computing and Applications, ICPCA 2011, Port Elizabeth, South Africa, 26–28 October 2011.

66. Jimenez-peris, R.; Pati, M. Transactional Processing for Polyglot Persistence. In Proceedings of the 2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA), Crans-Montana, Switzerland, 23–25 March 2016.
67. Prasad, S.; Sha, M.S.N. NextGen Data Persistence Pattern in Healthcare: Polyglot Persistence. In Proceedings of the 2013 4th International Conference on Computing, Communications and Networking Technologies, ICCCNT 2013, Tiruchengode, India, 4–6 July 2013.
68. Kaur, K.; Rani, R. Managing Data in Healthcare Information Systems: Many Models, One Solution. *Computer* **2015**, *48*, 52–59. [CrossRef]
69. Prasad, S.; Avinash, S.B. Application of Polyglot Persistence to Enhance Performance of the Energy Data Management Systems. In Proceedings of the 2014 International Conference on Advances in Electronics, Computers, and Communications, ICAECC 2014, Bangalore, India, 10–11 October 2014.
70. Shah, C.; Srivastava, K.; Shekokar, N. A Novel Polyglot Data Mapper for an E-Commerce Business Model. In Proceedings of the 2016 IEEE Conference on e-Learning, e-Management and e-Services, IC3e 2016, Langkawi, Malaysia, 10–12 October 2016.
71. Castrejón, J.; Vargas-Solar, G.; Collet, C.; Lozano, R. ExSchema: Discovering and Maintaining Schemas from Polyglot Persistence Applications. In Proceedings of the IEEE International Conference on Software Maintenance, ICSM, Eindhoven, The Netherlands, 22–28 September 2013.
72. Kolev, B.; Valduriez, P.; Bondiombouy, C.; Jiménez-Peris, R.; Pau, R.; Pereira, J. CloudMdsQL: Querying Heterogeneous Cloud Data Stores with a Common Language. *Distrib. Databases* **2015**, *34*, 463–503. [CrossRef]
73. Schaarschmidt, M.; Gessert, F.; Ritter, N. Towards Automated Polyglot Persistence. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16*; Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS): Hamburg, Germany, 2015.
74. Gessert, F.; Ritter, N. Scalable Data Management: NoSQL Data Stores in Research and Practice. In Proceedings of the 2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, 16–20 May 2016.
75. Bussler, C. SQL for NoSQL Databases: Déjà Vu. In Proceedings of the Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA, 4–7 January 2015.
76. Bach, M.; Werner, A. Standardization of NoSQL Database Languages. In *Communications in Computer and Information Science*; Springer: New York, NY, USA, 2014.
77. Hausenblas, M.; Nadeau, J. Apache Drill: Interactive Ad-Hoc Analysis at Scale. *Big Data* **2013**, *1*, 100–104. [CrossRef] [PubMed]
78. Gessert, F.; Friedrich, S.; Wingerath, W.; Schaarschmidt, M.; Ritter, N. Towards a Scalable and Unified REST API for Cloud Data Stores. In *Lecture Notes in Informatics (LNI), Proceedings—Series of the Gesellschaft für Informatik (GI)*; Gesellschaft für Informatik e.V.: Bonn, Germany, 2014.
79. Lu, J.; Liu, Z.H.; Xu, P.; Zhang, C. UDBMS: Road to Unification for Multi-Model Data Management. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; Springer: Cham, Switzerland, 2018.
80. Pääkkönen, P.; Pakkala, D. Reference Architecture, and Classification of Technologies, Products, and Services for Big Data Systems. *Big Data Res.* **2015**, *2*, 166–186. [CrossRef]
81. How to Beat the CAP Theorem. Available online: http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html (accessed on 20 December 2018).
82. Questioning the Lambda Architecutre. Available online: https://www.oreilly.com/ideas/questioning-the-lambda-architecture (accessed on 30 December 2018).
83. Kleppmann, M. *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems*; O'Reilly: Boston, MA, USA, 2017.
84. Kreps, J. *I Heart Logs*; O'Reilly: Boston, MA, USA, 2015.
85. O'Reilly Media, Inc. *Big Data Now: Applying the Kappa Architecture to the Telco Industry*; O'Reilly: Boston, MA, USA, 2016; pp. 33–40.
86. Vanhove, T.; Sebrechts, M.; Van Seghbroeck, G.; Wauters, T.; Volckaert, B.; De Turck, F. Data Transformation as a Means towards Dynamic Data Storage and Polyglot Persistence. *Int. J. Netw. Manag.* **2017**, *27*, e1976. [CrossRef]
87. Polyglot Processing. Available online: http://datadventures.ghost.io/2014/07/06/polyglot-processing (accessed on 19 December 2018).

88. Khine, P.P.; Wang, Z.S. Data Lake: A New Ideology in Big Data Era. *ITM Web Conf.* **2018**, *17*, 3025. [CrossRef]
89. Wiese, L. Polyglot Database Architectures. In Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB, Trier, Germany, 7–9 October 2015.