Data Storage Management in Cloud Environments: Taxonomy, Survey, and Future Directions

YASER MANSOURI, ADEL NADJARAN TOOSI, and RAJKUMAR BUYYA, The University of Melbourne, Australia

Storage as a Service (StaaS) is a vital component of cloud computing by offering the vision of a virtually infinite pool of storage resources. It supports a variety of cloud-based data store classes in terms of availability, scalability, ACID (Atomicity, Consistency, Isolation, Durability) properties, data models, and price options. Application providers deploy these storage classes across different cloud-based data stores not only to tackle the challenges arising from reliance on a single cloud-based data store but also to obtain higher availability, lower response time, and more cost efficiency. Hence, in this article, we first discuss the key advantages and challenges of data-intensive applications deployed within and across cloud-based data stores. Then, we provide a comprehensive taxonomy that covers key aspects of cloud-based data store: data model, data dispersion, data consistency, data transaction service, and data management cost. Finally, we map various cloud-based data stores projects to our proposed taxonomy to validate the taxonomy and identify areas for future research.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Data Management, Data Storage, and Data Replication

Additional Key Words and Phrases: Data management, data storage, data replication, data consistency, transaction service, and data management cost

ACM Reference format:
Yaser Mansouri, Adel Nadjaran Toosi, and Rajkumar Buyya. 2017. Data Storage Management in Cloud Environments: Taxonomy, Survey, and Future Directions. *ACM Comput. Surv.* 50, 6, Article 91 (December 2017), 51 pages.
https://doi.org/10.1145/3136623

1 INTRODUCTION

The explosive growth of data traffic driven by social networks, e-commerce, enterprises, and other data sources has become an important and challenging issue for IT enterprises. This growing speed is doubling every two years and augments tenfold between 2013 and 2020—from 4.4 to 44 ZB. The challenges posed by this growth of data can be overcome with aid of using cloud computing services. Cloud computing offers the illusion of infinite pool of highly reliable, scalable, and flexible

Authors' addresses: Y. Mansouri, A. N. Toosi, and R. Buyya, are with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Parkville Campus, VIC 3010, Australia; email: yase@student.unimelb.edu.au, {anadjaran,rbuyya}@unimelb.edu.au.

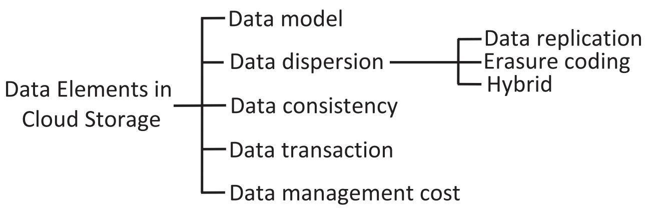

Fig. 1. Data elements in cloud storage.

computing, storage, and network resources in a pay-per-use manner. These resources are typically categorized as Infrastructure as a Service (IaaS), where Storage as a Service (StaaS) forms one of its critical components.

StaaS provides a range of cloud-based data stores (*data stores* for short) that differs in data model, data consistency semantic, data transaction support, and price model. A popular class of data stores, called Not only SQL (NoSQL), has emerged to host applications that require high scalability and availability without having to support the ACID properties of relational database (RDB) systems. This class of data stores—such as PNUTS (Cooper et al. 2008) and Dynamo (DeCandia 2007)—typically partitions data to provide scalability and replicates the partitioned data to achieve high availability. *Relational data store*, as another class of data stores, provides a full-fledged relational data model to support ACID properties, while it is not as scalable as NoSQL data store. To strike a balance between these two classes, NewSQL data store was introduced. It captures the advantages of both NoSQL and relational data stores and initially was exploited in Spanner (Corbett et al. 2013).

To take the benefits of these classes, application providers store their data either in a single or multiple data stores. A single data store offers the proper availability, durability, and scalability. But reliance on a single data store has risks like vendor lock-in, economic failure (e.g., a surge in price), and unavailability as outages occur, and probably leads to data loss when an environmental catastrophe happens (Borthakur et al. 2011). Geo-replicated data stores, on the other hand, mitigate these risks and also provide several key benefits. First, the application providers can serve users from the best data store to provide adequate responsiveness since data is available across data stores. Second, the application can distribute requests to different data stores to achieve load balance. Third, data recovery can be possible when natural disaster and human-induced activities happen. However, the deployment of a single or multiple data stores causes several challenges depending on the characteristics of data-intensive applications.

Data-intensive applications are potential candidates for deployment in the cloud. They are categorized into transactional (referred to as *online transaction processing* (OLTP)) and analytical (referred to as *online analytical processing* (OLAP)) that demand different requirements. OLTP applications embrace different consistency semantics and are adaptable with row-oriented data model, while OLAP applications require rich query capabilities and compliance with column-oriented data model. These requirements are faced with several challenges, and they mandate that we investigate the key elements of data management in data stores as shown in Figure 1. The first five key elements are mature topics in the context of distributed systems and require how to apply them to data stores with possible modifications and adoptions if needed. The last element, *data management cost*, is a new feature for cloud storage services and it is important for users to optimize it while their Service Level Agreements (SLAs) are guaranteed.

The first element is *data model* that reflects how data is stored in and retrieved from data stores. The second element is *data dispersion* with three schemes. *Data replication* scheme improves availability and locality by moving data close to the user, but it is costly due to usually storing three replicas for each object in data stores. *Erasure coding* scheme alleviates this overhead, but it requires structural design to reduce the time and cost of recovery. To make a balance between these benefits and shortcomings, a combination of both schemes is exploited. We clarify these schemes and identify how they influence availability, durability, and user-perceived latency.

Other elements of data management are *data consistency* and *data transaction* that refer to coordination level between replicas within and across data stores. Based on CAP theorem (Gilbert and Lynch 2002), it is impossible to jointly attain Consistency, Availability, and Partition tolerance (referred to the failure of a network device) in distributed systems. Thus, initially data stores provide *eventual consistency*—all replicas eventually converge to the last updated value—to achieve two of three these properties: availability and partition tolerance. Eventual consistency is sometimes acceptable, but not for some applications (e.g., e-commerce) that demand strong consistency in which all replicas receive requests in the same order. To obtain strong consistency, the recent endeavours bring transactional isolation levels in NoSQL/NewSQL data stores at the cost of extra resources and higher response time.

The last element is *data management cost* as the key driver behind the migration of application providers into the cloud that offers a variety of storage and network resources with different prices. Thus, application providers have many opportunities for *cost optimization* and *cost trade-offs*, such as storage vs. bandwidth, storage vs. computing, and so on.

The main contributions of this paper are as follows:

- Comparison between cloud-based data stores and related data-intensive networks,
- Delineation on goals and challenges of intra- and inter-cloud storage services, and determination of the main solutions for each challenge,
- Discussion on data model taxonomy in three aspects: *data structure*, *data abstraction*, and *data access model*; and comparison between different levels/models of each aspect,
- Providing a taxonomy for different schemes of data dispersion, and determining when (according to the specifications of workload and the diversity of data stores) and which scheme should be used,
- Elaboration on different levels of consistency and determination on how and which consistency level is supported by the state-of-the-art projects,
- Providing a taxonomy for transactional data stores, classifying them based on the provided taxonomy, and analyzing their performance in terms of throughput and bottleneck at message,
- Discussion on the cost optimization of storage management, delineation on the potential cost trade-offs in data stores, and classifying the existing projects to specify the research venue for future.

This survey is divided into seven sections. Section 2 compares cloud-based data stores to other distributed data-intensive networks and then discusses the architecture, goals, and challenges of a single and multiple cloud-based data stores deployments. Section 3 describes a taxonomic of data model, and Section 4 discusses different schemes of *data dispersion*. Section 5 elaborates on data consistency in terms of *level*, *metric*, and *model*. Section 6 details the taxonomy of transactional data stores, and Section 7 presents the cost optimization of storage management. Section 8 concludes the paper and presents future directions in the context of data storage management.

2 OVERVIEW

This section discusses a comparison between cloud-based data stores and other data-intensive networks (Section 2.1), the terms used throughout this survey (Section 2.2), the characteristics of

Table 1. Comparison between Data-intensive Networks in Charachteristics and Objectives

Property	Cloud-Based Data Stores	Data Grids	Content Delivery Network (CDN)	Peer to Peer (P2P)
Purpose	Pay-as-you-go model, on-demand provisioning and elasticity	Analysis, generating, and collaboration over data	File sharing and content distribution	Improving user-perceived latency
Management Entity	Vendor	Virtual Organization	Single organization	Individual
Organization	Tree-based (Wang et al. 2014)† Fully optical Hybrid	Hierarchy Federation	Hierarchy	Unstructured Structured Hybrid (Venugopal et al. 2006)
Service Delivery	IaaS, PaaS, and SaaS	IaaS	IaaS	IaaS
Access Type	Read-intensive Write-intensive Equally of both	Read-intensive with rare writes	Read-only	Read- intensive with frequent writes
Data Type	Key-value Document-based Extensible record Relational	Object-based (Often big chunks)	Object-based (e.g., media, software, script, text)	Object/file-based
Replica Discovery	HTTP requests Replica Catalog	Replica Catalog	HTTP requests	Distributed Hash Table††, Flooded requests
Replica Placement	Section 4.1.6	Popularity Primary replicas	A primary copy Caching	Popularity without primary replica
Consistency	Weak and Strong	Weak	Strong	Weak
Transaction Support	Only in relational data stores (e.g., Amazon RDS)	None	None	None
Latency Management	Replication, caching, streaming	Replication, caching, streaming	Replication, caching, streaming	Caching, streaming
Cost Optimization	Pay-as-you-go model (in granularity of byte per day for storage and byte for bandwidth)	Generally available for not-for-profit work or project-oriented	Content owners pay CDN operators, which, in turn, pays ISPs to host contents	Users Pay P2P to receive sharing files.

†Tree-based organization has a flexible topology, while fully optical (consisting of a "pure" optical switching network) and hybrid (including switching network of electrical packet and optical circuit) organizations have a fixed topology. Google and Facebook deploy fat tree topology, a variant of tree topology, in their datacener architecture. †† Distributed hash table is used for structured organization and flooded requests for the unstructured organization.

data-intensive applications deployed in data stores (Section 2.3), and the main goals and challenges of a single and multiple data stores leveraged to manage these applications (Section 2.4).

2.1 A Comparison of Data-Intensive Networks

Table 1 highlights similarities and differences in characteristics and objectives between cloud-based data stores and (i) Data Grid in which storage resources are shared among several industrial/educational organizations as Virtual Organization (VO), (ii) Content Delivery Network (CDN) in which a group of servers/datacenters (DCs) are located in several geographical locations to serve users contents (i.e., application, web, or video) faster, and (iii) Peer-to Peer (P2P) in which a peer (i.e., server) shares files with other peers.

Cloud-based data stores share more overlaps with Data Grids in the properties listed in Table 1. They deliver more abstract storage resources (due to more reliance on virtualization) for different types of workloads in an accurate economic model. They also provide more elastic and scalable resources for different demands in size. These opportunities result in the migration of data-intensive

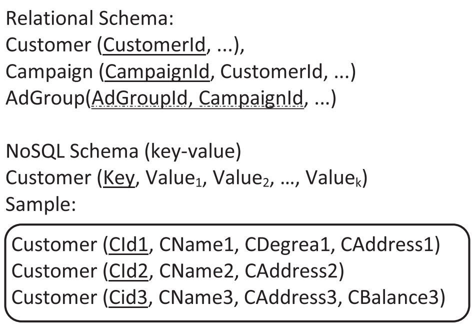
Relational Schema:
Customer (CustomerId, ...),
Campaign (CampaignId, CustomerId, ...)
AdGroup(AdGroupId, CampaignId, ...)
NoSQL Schema (key-value)
Customer (Key, Value1, Value2, ..., Valuek)
Sample:
Customer (CId1, CName1, CDegrea1, CAddress1)
Customer (CId2, CName2, CAddress2)
Customer (Cid3, CName3, CAddress3, CBalance3)

Fig. 2. Relational and NoSQL schemas.

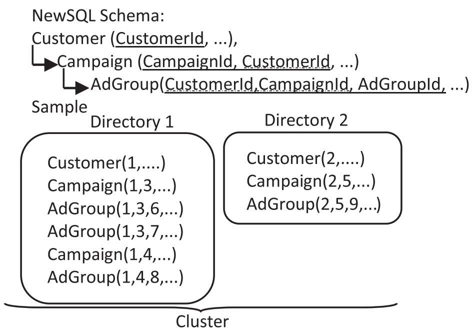

Fig. 3. NewSQL Schema (Shute et al. 2013).

applications to the clouds and cause two categories of issues. One category is more specific to cloud-based data stores and consists of issues such as vendor lock-in, multi-tenancy, network congestion, monetary cost optimization, and so on. Another category is common between cloud-based data stores and Data Grids and includes issues like data consistency and latency management. Some of these issues require totally new solutions, and some of them have mature solutions and may be applicable either wholly or with some modifications based on the different properties in cloud-based data stores.

2.2 Terms and Definitions

A *data-intensive application* system consists of applications that generate, manipulate, and analyze a large amount of data. With the emergence of cloud-based storage services, the data generated from these applications are typically stored in *single data store* or *Geo-replicated data stores* (several data stores in different worldwide locations). The data is organized as *dataset*, which is created and accessed across DCs by *users/application providers*. *Metadata* describe the dataset with respect to the several attributes such as name, creation time, owner, replicas location, and so on.

A dataset consists of a set of *objects* or *records* that are modeled in relational databases (RDBs) or NoSQL databases. The data store that manages RDBs and NoSQL databases are, respectively, called as *relational data store* and *NoSQL data store*. As shown in Figure 2, RDBs have fixed and predefined fields for each object whereas NoSQL databases do not. NoSQL data stores use *key-value* data model (or its variations such as graph and document) in which each object is associated with a pair of *key* and *value*. Key is unique and is used to store and retrieve the associated value of an object. NewSQL data stores follow a *hierarchical data model*, which introduces a *child-parent* relation between each pair of table where the child table (`Campaign` in Figure 3) borrows the primary key of its parent (e.g., `Customer`) as a prefix for its primary key. In fact, this data model has a *directory table* (e.g., `Customer` in Figure 3) in the top of hierarchical structure and each row of directory table together with all rows in the descendant tables (e.g., `Campaign` and `AdGroup`) constructs a *directory*.

A *complete* or *partial* replica of the dataset is stored in a single data store or different data stores across Geo-distributed DCs based on the required QoS (response time, availability, durability, monetary cost). An operation to update an object can be initially submitted to a predefined replica (called *single master*) or to any replicas (*multi-master*) based on predefined strategy (e.g., the closest). Replicas are *consistent* when they have the same value for an object. Replicas are in *weak/strong* consistency status if replicas return (probably) *different/same* values for a read operation. A *transaction* is a set of reads and writes, and it is *committed* if all reads and writes are conducted (on all replicas of data); otherwise, it is *aborted*.

Service Level Agreement (SLA) is a formal commitment between the cloud provider (e.g., Amazon and Microsoft Azure) and the application providers/users. For example, current cloud providers compensate users with service credit if the availability of a data store would be below a specific value in percent.

2.3 Data-intensive Applications

In respect to the cloud characteristics, two types of data-intensive applications can be nominated for the cloud deployment (Abadi 2009).

Online transaction processing (OLTP) applications must guarantee ACID properties and provide an "all-or-nothing" proposition that implies each set of operations in a transaction must complete or no operation should be completed. Deploying OLTP applications across data stores is not straightforward, because achieving the ACID properties requires acquiring distributed locks, executing complex commit protocols and transferring data over network, which in turn causes network congestion across data stores and introduces network latency. Thus, OLTP should be adapted to ACID properties at the expense of high latency to serve reads and writes across data stores. *Online Analytical Processing* (OLAP) applications usually use read-only databases and often need to handle complex queries to retrieve the desired data for data warehouse. The updates in OLAP are conducted on regular basis (e.g., per day or per week), and their rate is lower than that of OLTP. Hence, the OLAP applications do not need to acquire distributed locks and can avoid complex commit protocols.

Both OLTP and OLAP should handle a tremendous volume of data at incredible rates of growth. This volume of data is referred to as *big data*, which has challenges in five aspects: *volume* refers to the amount of data; *variety* refers to the different types of generated data; *velocity* refers to the rate of data generation and the requirement of accelerating analysis; *veracity* refers to the certainty of data; and *value* refers to the determination of hidden values from datasets. Schema-less NoSQL databases easily cope with two of these aspects via providing an infinite pool of storage (i.e., volume) in different types of data (i.e., variety). For other aspects, NoSQL data stores are controversial for OLTP and OLAP. From the velocity aspect, NoSQL data stores like BigTable (Chang et al. 2008), PNUTS (Cooper et al. 2008), Dynamo (DeCandia 2007), and Cassandra (Lakshman and Malik 2010) facilitate OLTP with reads and writes in low latency and high availability at the expense of weak consistency. This is a part of the current article to be discussed. From velocity, veracity and value aspects (the last two aspects are more relevant to OLAP), OLAP requires frameworks like Hadoop,[1] Hive (Thusoo et al. 2010), and Pig,[2] as well as algorithms in big data mining to analysis and optimize the complex queries in NoSQL data stores. It is worth to mention these frameworks lack rich query processing on the cloud and change into the data model is a feasible solution that is out of the scope of this article. In the rest of this section, we focus on goals and challenges of data stores that suit OLTP applications.

2.4 Architecture, Goals, and Challenges of Intra-Cloud Storage

This section first describes a layered architecture of data store and then discusses the key goals and challenges of deploying a single data store to manage data-intensive applications.

2.4.1 Architecture of Intra-Cloud Storage. Cloud storage architecture shares a storage pool through either a dedicated Storage Area Network (SAN) or Network Attached Storage (NAS).[3]

[1]http://wiki.apache.org/hadoop.
[2]http://hadoop.apache.org/pig/.
[3]A NAS is a single storage device that operates on file system and makes TCP/IP and Ethernet connections. In contrast, a SAN is a local network of multiple devices that operates on disk blocks and uses fiber channel interconnections.

Table 2. Intra-Cloud Storage Goals

Goals	Techniques (Examples)	Section(s)
Performance and cost saving	Combination of different storage services	Section 7
Fault tolerance and availability	Replication (Random replication, Copyset, MRR)	Section 4.1
	Erasure coding	Section 4.2
Multi-tenancy	Shared table (saleforce.com (Lomet 1996))	—
	Shared process (Pisces (Shue et al. 2012), ElasTras (Das et al. 2013a))	
	Shared machine (Commercial DaaS, like Amazon RDS)	
Elasticity and load balancing	Storage-load-aware (Dynamo, BigTable)	—
	Access-aware ((Chen et al. 2013))	

The architecture is composed of a distributed file system, Service Level Agreement (SLA), and interface services. It divides the components by physical and logical functions boundaries and relationships to provide more capabilities (Zeng et al. 2009). In the layered architecture, each layer is constructed based on the services offered by its underneath layer. These layers from bottom to top are as follows:

(1) *Hardware layer* consists of distributed storage servers in a cluster of several racks, and each of which has disk-heavy storage nodes. (2) *Storage management* provides services for managing data in the storage media. It consists of the fine-grained services like data replication and erasure coding management, replica recovery, load balancing, consistency, and transaction management. (3) *Metadata management* classifies the metadata of stored data in a global domain (e.g., storage cluster) and collaborates with different domains to locate data when it is stored or retrieved. For example, *Object Table* in Window Azure Storage (Calder et al. 2011) has a primary key containing three properties: *AccountName*, *PartitionName*, and *ObjectName* that determine the owner, location, and name of the table, respectively. (4) *Storage overlay* is responsible for storage virtualization that provides data accessibility and is independent of physical address of data. It converts a logical disk address to the physical address by using metadata. (5) *User interface* provides users with primitive operations and allows cloud providers to publish their capabilities, constraints, and service prices to help subscribers to discover the appropriate services based on their requirements.

2.4.2 Goals of Intra-Cloud Storage. Table 2 introduces the five main goals of data-intensive applications deployment in a single data store. These goals are as follows:

- *Performance and cost saving*. Efficient utilization of resources has a direct influence on cost saving as data stores offer the pay-as-you-go model. To achieve both these goals, a combination of storage services varying in price and performance yields the desired performance (i.e., response time) with a low cost as compared to relying on one type of storage service. To make an effective combination, the decision on when and which type of storage services to use should be made based on *hot*- and *cold*-spot statuses of data, respectively, receiving many and a few read/write requests, and the required QoS. Current data stores do not guarantee SLA in terms of performance and only measure latency within the storage service (i.e., the time between when a request was served at one of the storage nodes and when the response left the storage node) and the latency over the network (i.e., for the operation to travel from the VM to the storage node and for the response to travel back) via enabling logging.[4]
- *Fault tolerance and availability.* Failures arise from *faulty hardware* and *software* in two models. *Byzantine* model presents arbitrary behavior and can be survived via $2f + 1$ replicas (f

[4]Log format in Amazon: http://docs.aws.amazon.com/AmazonS3/latest/dev/LogFormat.html.

is the number of failures) along with non-cryptographic hashes or cryptographic primitives. *Crash-stop* model silently stops the execution of nodes and can be tolerated via *data replication* and *erasure coding* in multiple servers located in different racks, which in turn are in different domains. This failure model has two types: *correlate* and *independent*. The random placement of replicas used in current data stores only tackles independent failures. Copyset (Cidon et al. 2013) replicates an object on a set of storage nodes to tolerate correlated failures, and multi-failure resilient replication (MRR) (Liu and Shen 2016) places replicas of a single chunk in a group of nodes (partitioning into different sets across DCs) to cope with both types of failure.

- *Multi-tenancy*. Multi-tenancy allows users to run applications on shared hardware and software infrastructure so their isolated performance is guaranteed. It brings benefits for cloud providers to improve infrastructure utilization by eliminating to allocate the infrastructure to the maximum load; consequently, cloud providers run applications on and store data in less infrastructure. This, in turn, reduces the monetary cost for users. However, multi-tenancy causes *variable and unpredictable performance*, and *multi-tenant interference and unfairness* (Shue et al. 2012), which happen to different multi-tenancy models (from the weakest to the strongest): *shared table* model in which applications/tenants are stored in the same tables with an identification field for each tenant, *shared process* model in which applications share the database process with an individual table for each tenant, and *shared machine* model in which applications only share the physical hardware, but they have an independent database process and table. Among these models, shared process has the best performance as a database tenant is migrated, while shared machine brings inefficient resource sharing among tenants and shared table is not suitable for applications with different data models (Das et al. 2013a). Examples of each model are shown in Table 2.
- *Elasticity and load balancing*. Elasticity refers to expansion (scaling up) and consolidation (scaling down) of servers during load changes in a dynamic system. It is orthogonal with load balancing in which workloads are dynamically moved from one server to another under skewed query distribution so all servers handle workloads almost equally. This improves the infrastructure utilization and even reduces monetary cost. There are two approaches to achieve load balancing. *Storage-load-aware* approach uses *key range* (i.e., distributing tuples based on the value ranges of some attributes) and *consistent hash-algorithm* (DeCandia 2007) techniques to distribute data and balance load across storage nodes. These techniques, used by almost all existing data stores, are not effective when data-intensive applications experience hot- and cold-spot statuses. These applications thus deploy the *access-load-aware* approach for load balancing (Chen et al. 2013). Both approaches use either *stop and copy* migration, which is economic in data transferring, or *live* migration, which is network-intensive task but incurs fewer service interruptions when compared to the former migration technique (Tran et al. 2011). Nevertheless, current cloud providers support auto-scaling mechanisms in which the type, maximum and minimum number of database instances should be defined by users. Thus, they cannot provide a fine-grained SLA in this respect.

2.4.3 Challenges of Intra-Cloud Storage.

Table 3 introduces what challenges application providers confront with the deployment of their applications within a data store. These challenges are as follows:

- *Unavailability of services and data lock-in*. *Data replication* across storage nodes in an efficient way (instead of random replica placement widely used by current data stores) and *erasure coding* are used for high availability of data. Using these schemes across data stores

Table 3. Intra-Cloud Storage Challenges

Challenges	Solutions	Section(s)/References
Unavailability of services and data lock-in	Data replication Erasure coding	Section 4.1 Section 4.2
Data transfer bottleneck	**(i)** Workload-aware partitioning (for OLTP) **(ii)** Partitioning of social graph, co-locating the data of a user and his friends along with the topology of DC (for social networks) **(iii)** Scheduling and monitoring of data flows	(Kumar et al. 2014; Kamal et al. 2016; Tang et al. 2015; Chen et al. 2016; Zhou et al. 2017) (Al-Fares et al. 2010; Das et al. 2013b; Shieh et al. 2011)
Performance unpredictability	**(i)** Data replication and redundant requests **(ii)** Static and dynamic reservation of bandwidth **(iii)** Centralized and distributed bandwidth guarantee **(iv)** Bandwidth guarantee based on network topology and application communications	(Stewart et al. 2013; Shue et al. 2012) (Ballani et al. 2011; Xie et al. 2012) (Jeyakumar et al. 2013; Ballani et al. 2011; Popa et al. 2013; Guo et al. 2014) (Lee et al. 2014)
Data security	**(i)** Technical solutions (encryption algorithms, audit third party (ATP), digital signature) **(ii)** Managerial solutions **(iii)** A combination of solutions (i) and (ii)	(Shin et al. 2017)

also mitigates data lock-in in the face of the appearance of new data store with a lower price, mobility of users, and change in workload that demands data migration.

- *Data transfer bottleneck*. This challenge arises when data-intensive applications are deployed within a single data store. It reflects the optimality of data placement that should be conducted based on the characteristics of the application as listed for OLTP and social networks in Table 3. To reduce more network congestion, data flows should be monitored in the switches, and they should be then scheduled (i) based on the data flow prediction (Das et al. 2013b), (ii) when data congestion occurs (Al-Fares et al. 2010), and (iii) for integrated data flows (Shieh et al. 2011) rather than individual data flow. In addition, data aggregation (Costa et al. 2012), novel network typologies (Wang et al. 2014), and optical circuit switching deployment (Chen et al. 2014) are other approaches to drop network congestion.
- *Performance unpredictability*. Shared storage services face the challenges like *unpredictable performance* and *multi-tenant unfairness*, which result in degrading the response time of requests. In Table 3, the first two solutions solve challenge with respect to storage, where replicas placement and selection should be considered. The last two solutions cope the challenges related to network aspect, where *fairness* in allocated network bandwidth to cloud applications and *maximizing of network bandwidth utilization* should be taken into consideration. These solutions are: (i) *static* and *dynamic* reservation of bandwidth where the static approach cannot efficiently utilize bandwidth, (ii) *centralized* and *distributed* bandwidth guarantee where the distributed approach is more scalable, and (iii) *bandwidth guarantee* based on the network topology, and application communications, which is more efficient (Lee et al. 2014). Such solutions suffer from data delivery within deadline, and they thus are suitable for batch applications, but not for OLTP applications, which require the completion of data flows within deadline (Vamanan et al. 2012).
- *Data security*. This is one of the strongest barriers in the adoption of public clouds to store users' data. It is a combination of (i) data integrity, protecting data from any unauthorized operations; (ii) data confidentiality, keeping data secret in the storage (iii) data availability, using data at any time and place; (iv) data privacy, allowing data owner to selectively reveal

their data; (v) data transition, detecting data leakage and lost during data transfer into the storage; and (vi) data location, specifying who has jurisdiction and legislation over data in a transparent way. Many solutions were proposed in recent decades to deal with concerns over different security aspects except *data location* where data is probably stored out of users' control. These solutions are not generally applicable, since unlike the traditional systems with two parties, the cloud environment includes three parties: users, storage services, and vendors. The last party is a potential threat to the security of data, since it can provide a secondary usage for itself (e.g., advertisement purposes) or for governments.
Solutions to relieve concerns over data security in the cloud context can be *technical*, *managerial*, or *both*. In addition to technical solutions as listed in Table 3, managerial solutions should be considered to relieve security concerns relating to the geographical location of data. A combination of both types of solutions can be: (i) designing location-aware algorithms for data placement as data are replicated to reduce latency and monetary cost, (ii) providing location-proof mechanisms for user to know the precise location of data (e.g., measuring communication latency and determining distance), and (iii) specifying privacy acts and legislation over data in a transparent way for users. Since these acts, legislation, and security and privacy requirements of users are a fuzzy concept, it would be relevant to design a fuzzy framework in conjunction of location-aware algorithms. This helps users to find storage services, which are more secure in respect to rules applied by the location of data.

A better approach to achieve the discussed goals and avoid the above challenges is to store data across data stores. In the rest of the section, we discuss the benefits of this solution as well as the challenges that arise from it.

2.5 Architecture, Goals, and Challenges of Inter-Cloud Storage

This section describes the layered architecture of Inter-Cloud storage deployment along with its key benefits and challenges.

2.5.1 Architecture of Inter-Cloud Storage. The architecture of inter-cloud storage does not follow standard protocols. So far, several studies exploit multiple data stores diversity for different purposes with a light focus on the architecture of Inter-cloud storage (e.g., RACS (Abu-Libdeh et al. 2010) and ICStore (Cachin et al. 2010)). Spillner et al. (2011) focused more on the Inter-cloud storage architecture that allows user to select a data store based on service cost or minimal downtime. Inspired by this architecture, we pictorially clarify and discuss a layered inter-cloud architecture as shown in Figure 4.

- *Transport layer* represents a simple data transport abstraction for each data store in its underneath layer and transfers data to data store specified in the upper layer by using its transport module. This layer implements a file system (e.g., the Linux Filesystem in Userspace[5]) to provide *protocol adaptor* for the data store.
- *Data management layer* provides services for managing data across data stores and consists of services such as *load balance*, *replication*, *transaction*, *encryption*, and *decryption*. Load balance service monitors the submitted requests to each data store and reports to replication service to store a new replica or write/read the object in/from data store with the lowest load. Replication and transaction services can be configured in the layered architecture and collaborate with each other to provide the desired consistency. Encryption

[5]http://fuse.sourceforge.net.

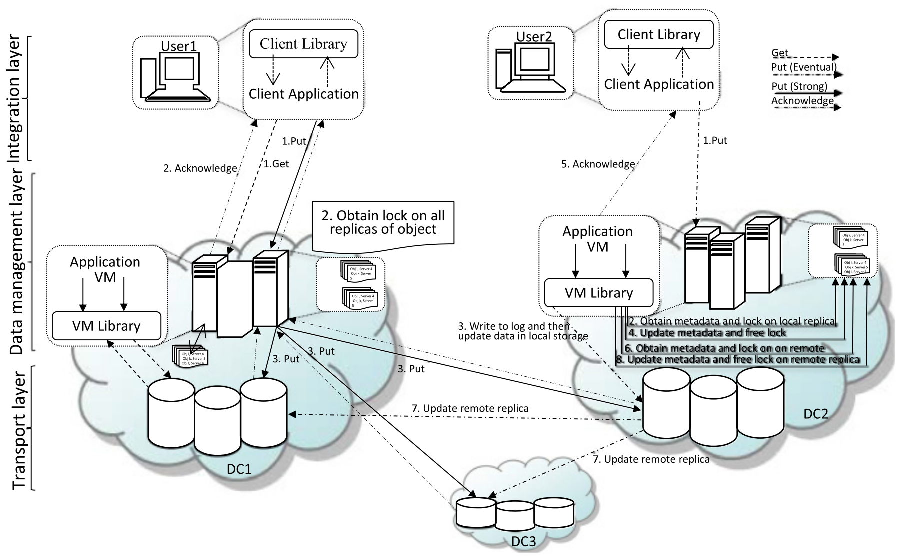

Fig. 4. Inter-cloud storage architecture.

and decryption services can be deployed to make data more secure and reduce the storage and network overheads.

- *Integration layer* allows user to access and manipulate the data through *client library* (API), which is exploited by the client application to issue operations to data stores (often the closest one).
- *Metadata component* plays as an essential component in the architecture and collaborates with data management layer for writing (respectively, reading) objects in (respectively, from) data stores. It contains *object metadata* (e.g., in the form of XML) to determine the location of object replicas and is stored by all DCs in which the application is deployed.

As an example in Figure 4, an issued Get (read) request from client library is usually directed to the closest DC (e.g., DC1), and the requested object is searched in metadata stored at DC. If the object exists, then the application code in VM retrieves the object from the local data store and returns it to the client. Otherwise, the DC sends the request to DC, which hosts the object according to the metadata. For a Put (write) request, after finding the location of replicas, first the object is written in VM's application and the metadata in VM is changed, and then the VM library invokes the Put in client library to store the object in local data store via *transport layer*. Finally, the Put is propagated to other replicas to guarantee eventual consistency (Put in DC2) or, in a simple way, all replicas are locked and the Put is synchronously propagated to them to provide strong consistency (Put in DC1).

2.5.2 *Goals of Inter-Cloud Storage.* Table 4 lists the key goals of leveraging Geo-replicated data stores. These are as follows:

- *High availability, durability, and data lock-in avoidance.* These are achievable via data replication scheme across data stores owned by different cloud providers. A 3-way data replication provides availability of 7 nines (Mansouri et al. 2013), which is adequate for most applications. This is inadequate for protecting data against *correlated* failures, while two replicas are sufficient for guarding data against *independent* failures (Cidon et al. 2015). *Erasure coding* scheme is another way to attain higher durability even though it

Table 4. Inter-Cloud Storage Goals

Goals	Techniques/Schemes	Section(s)
High availability, durability, and data lock-in avoidance	Data replication Erasure coding	Section 4.1 Section 4.2
Cost benefit	Exploitation of pricing differences across data stores and time-varying workloads	Section 7
Low user-perceived latency	Placing replicas close to users Co-locating the data accessed by the same transactions Determining the location and roles of replicas (master and slave) in a quorum-based configuration (Sharov et al. 2015)	—
High data confidentiality, integrity, and auditability	Cryptographic protocols with erasure coding and RAID techniques	—

degrades data availability in comparison to data replication. Both schemes prevent data lock-in.

- *Cost benefit.* Due to *pricing differences* across data stores and *time-varying workloads*, application providers can diversify their infrastructure in terms of vendors and locations to optimize cost. The cost optimization should be integrated with the demanding QoS level like availability, response time, consistency level, and so on. This can be led to a trade-off between the cost of two resources (e.g., storage vs. computing) or the total cost optimization based on a single-/multi-QoS metrics (Section 7).
- *Lowe user-perceived latency.* Application providers achieve lower latency through deploying applications across multiple cloud services rather than within a single cloud services. Applications can further decrease latency via techniques listed in Table 4. In spite of these efforts, users may still observe the latency variation, which can be improved through caching data in memory (Nishtala et al. 2013)), issuing redundant reads/writes to replicas (Wu et al. 2015)), and using feedback from servers and users to prevent requests redirection to saturated servers (Suresh et al. 2015)).
- *High data confidentiality, integrity, and auditability.* Using *Cryptographic protocols with erasure coding and RAID techniques* on top of multiple data stores improves security in some aspects as deployed in HAIL (Bowers et al. 2009) and DepSky (Bessani et al. 2011). In such techniques, several concerns are important: scalability, cost of computation and storage for encoding data, and the decision on where the data is encoded and the keys used for data encryption are maintained. A combination of *private* and *public* data stores and applying these techniques across public data stores offered by different vendors improve data protection against both insider and outsider attackers, especially for insider ones who require access to data in different data stores. Data placement in multiple data stores, on the other hand, brings a side effect, since different replicas are probably stored under different privacy rules. Selection of data stores with similar privacy acts and legislation rules would be relevant to alleviate this side effect.

2.5.3 Challenges of Inter-Cloud Storage. The deployment of data-intensive applications across data stores is faced with the key challenges as listed in Table 5. These are discussed as follows:

- *Cloud interoperability and portability.* Cloud interoperability refers to the ability of different cloud providers to communicate with each other and agree on the data types, SLAs, and so on. Cloud portability means the ability to migrate application components and data across cloud providers regardless of APIs, data types, and data models. Table 5 lists solutions for this challenge.

Table 5. Inter-Cloud Storage Challenges

Challenges	Solutions	Example	Section(s)
Cloud interoperabiliy and portability	Standard protocols for IaaS	n/a†	—
	Abstraction storage layer	CASL (Hill and Humphrey 2010)	
	Open API	JCloud††	
Network congestion	Dedicating redundant links across DCs	n/a	—
	Store and forward approach	Postcard (Feng et al. 2012; Wu et al. 2017)	
	Software-Defined Networking (SDN)	B4 (Jain et al. 2013; Wu et al. 2017)	
Strong consistency and transaction guarantee	Heavy-weight coordination protocols	See Appendices C, D, and E	Section 5, Section 6
	Contemporary techniques		

†n/a: not applicable, †† JCloud: https://jclouds.apache.org/.

- *Network congestion.* Operating across Geo-DCs causes *network congestion*, which can be *time-sensitive* or *non time-sensitive*. The former, like interactive traffic, is sensitive to delay, while the latter, like transferring big data and backing up data, is not so strict to delay and can be handled within deadline (Zhang et al. 2015) or without deadline (Jain et al. 2013). The first solution for this challenge, listed in Table 5, is expensive, while the second solution increases the utilization of network. The last solution is Software-Defined Networking (SDN) (Kreutz et al. 2015). SDN separates *control plane* that decides how to handle network traffic, and *data paths* that forwards traffic based on the decision made from control plane. All these solutions answer a part of this fundamental question: *how to schedule the data transfer so it is completed within a deadline and budget subject to the guaranteed network fairness and throughput for jobs that processes the data.*
- *Strong consistency and transaction guarantee.* Due to high communication latency between DCs, coordination across replicas to guarantee strong consistency can drive users away. To avoid high communication latency, some data stores compromise strong consistency at the expense of application semantics violations and stale data observations. Others, on the other hand, provide strong consistency in the cost of low availability. To achieve strong consistency without compromising with availability and scalability, coordination across replicas should be reduced or even eliminated.

Our survey article focuses on some of these discussed goals and challenges. In particular, the rest of the article discusses the five elements of data storage management specified in Figure 1.

3 DATA MODEL

Data model reflects how data is logically organized, stored, retrieved, and updated in data stores. We thus study it from different aspects and map data stores to the provided data model taxonomy in Figure 5.

3.1 Data Structure

Data structure affects the speed of assimilation and information retrieving. It has three categories. (i) *Structured* data refers to data that defines the relationship between the fields specified with a name and value, for example, RDBs. It supports a comprehensive query and transaction processing facilities. (ii) *Semi-structured* is associated with the special form of structured data with a specific schema known for its application and database deployments (e.g., document and extensible DBs). It supports primitive operations and transaction facilities as compared to structured data. (iii) *Unstructured* data refers to data that have neither pre-defined data model nor organized in a pre-defined way (e.g., video, audio, and heavy-text files). It takes the simplest data access model,

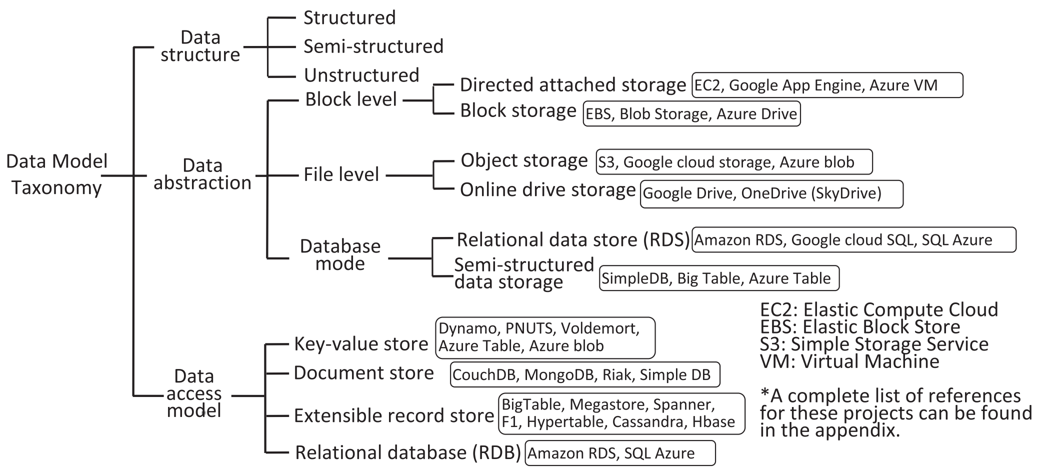
Data Model Taxonomy
Data structure
Structured
Semi-structured
Unstructured
Data abstraction
Block level
Directed attached storage
EC2, Google App Engine, Azure VM
Block storage
EBS, Blob Storage, Azure Drive
File level
Object storage
S3, Google cloud storage, Azure blob
Online drive storage
Google Drive, OneDrive (SkyDrive)
Database mode
Relational data store (RDS)
Amazon RDS, Google cloud SQL, SQL Azure
Semi-structured data storage
SimpleDB, Big Table, Azure Table
Data access model
Key-value store
Dynamo, PNUTS, Voldemort, Azure Table, Azure blob
Document store
CouchDB, MongoDB, Riak, Simple DB
Extensible record store
BigTable, Megastore, Spanner, F1, Hypertable, Cassandra, Hbase
Relational database (RDB)
Amazon RDS, SQL Azure
EC2: Elastic Compute Cloud
EBS: Elastic Block Store
S3: Simple Storage Service
VM: Virtual Machine
*A complete list of references for these projects can be found in the appendix.

Fig. 5. Data model taxonomy.

that is, key-value, that delivers high scalability and performance at the cost of sacrificing data consistency semantic and transaction support.

In these logical data structures, data is internally organized row-by-row, column-by-column closely related to database normalization, or combination of both schemes- called hybrid—within a block. *Structured data* can be organized in all schemes, *semi-structured* in row-by-row and hybrid schemes, and *Unstructured data* in a row-by-row scheme.

3.2 Data Abstraction

This refers to different levels of storage abstraction in data stores. These levels are as follows:

(1) *Block-level* provides the fastest access to data for virtual machine (VM). It is classified into (i) *directed-attached storage* coming with a VM that provides highly sequential I/O performance with a low cost, and (ii) *block storage* that pairs with a VM as a local disk.
(2) *File-level* is the most common form of data abstraction due to its ease of management via simple API. It is provided in the forms of (i) *object storage* that enables users to store large binary objects anywhere and anytime, and (ii) *online drive storage* that provides users with folders on the Internet as storage.
(3) *Database mode* offers storage in the forms of *relational data store* and *semi-structured data storage*, which, respectively, provide users with RDB and NoSQL/NewSQL databases. RDB exploiting the SQL standard does not scale easily to serve large web applications but guarantees strong consistency. In contrast, NoSQL provides horizontal scalability by means of shared nothing, replicating, and partitioning data over many servers for simple operations. In fact, it preserves BASE (Basically Available, Soft state, Eventually consistent) properties instead of ACID ones to achieve higher performance and scalability. NewSQL—as a combination of *RDB and NoSQL*—targets delivering the scalability similar to NoSQL, meanwhile maintaining ACID properties.

Table 6 compares different levels of storage abstractions in several aspects as well as their applicability. This comparison indicates that as the storage abstraction (ease of use) level increases, the cost and performance of storage reduce.

Table 6. Comparison between Different Storage Abstractions

Ease of use†	Scalability	Performance††	Cost [Applicability]
File-level	File-level	Block-level	Block-level [OLAP applications]
Database mode	Block-level	Database mode	Database mode [OLTP applications with low latency queries]
Block-level	Database mode	File-level	File-level [Backup data and web content static]

†The levels of storage abstract are listed from high to low for each aspect listed in each column. ††Performance is defined in terms of *accessibility*.

Table 7. Comparison between NoSQL (The First Three Databases) and Relational Databases

Database	Simplicity	Flexibility	Scalability	Properties	Data Structure	SLA	Application [Query] type
key-value	High	High	High	–	(Un-/)structured	Weak	OLAP [Simple]
Document	High	Moderate	Moderate	BASE	Semi-structured	Weak	OLAP [Moderate]
Extensible record	High	High	High	ACID	(Semi-/)structured	Weak	OLTP [Moderate, repetitive]
Relational	Low	Low	Low	ACID	Structured	Weak	OLTP [Complex]

3.3 Data Access Model

This reflects storing and accessing model of data that affect on consistency and transaction management. It has four categories as follows:

(1) *Key-value database* stores keys and values that are indexed by keys. It supports primitive operations and high scalability via keys distribution over servers. Data can be retrieved from the data store based on more than one attribute if additional key-value indexes are maintained.
(2) *Document database* stores all kinds of documents indexed in the traditional sense and provides primitive operations without ACID guarantee. It thus supports *eventual consistency* and achieves scalability via *asynchronous* replication, *shard* (i.e., horizontal partition of data in the database), or both.
(3) *Extensible record database* is analogous to table in which columns are grouped, and rows are split and distributed on storage nodes (Cattell 2011) based on the primary key range as a *tablet* representing the unit of distribution and load balancing. Each cell of the table contains multiple versions of the same data that are indexed in decreasing timestamps order, thereby the most recent version can always read first (Sakr et al. 2011). This scheme is called NewSQL, which is equivalent with *entity group* in Megastore (Baker et al. 2011), *shard* in Spanner (Corbett et al. 2013), and *directory* in F1 (Shute et al. 2013).
(4) *Relational database* (RDB) has a comprehensive pre-defined scheme and provides manipulation of data through SQL interface that supports ACID properties. Except for *small-scope* transactions, RDB cannot scale the same as NoSQL.

Table 7 compares different databases in several aspects and indicates which type of application and query can deploy them. NoSQL databases offer horizontal scalability and high availability by scarifying *consistency semantic*, which makes them unsuitable for the deployment of OLTP applications. To provide stronger consistency semantic for OLTP applications, it is vital to carefully partition data within and especially across data stores and to use the mechanisms that exempt or minimize the coordination across transactions. Moreover, these databases have several limitations relating to big data for OLAP applications. All these disk-based data stores cannot suitably facilitate OLAP applications in the concept of *velocity* and thus most commercial vendors combine them with in-memory NoSQL/relational data stores (e.g., Memcached,[6] Redis,[7] and RAMCloud (Rumble

[6]https://memcached.org/.
[7]https://redis.io/.

et al. 2014)) to further improve performance. These databases also cannot be completely fitted with the concept of *velocity* relating to big data for OLAP applications, which receive data with different formats. It is thus required a platform to translate these formats into a canonical format. OLAP applications can combat the remaining limitations (i.e., veracity and value) arising from using these databases in the face of large data volumes via designing indexes to retrieve data. Moreover, NoSQL data stores guarantee weak SLA in terms of availability (Sakr 2014) and auto-scaling, and without any SLA in the case of response time. In fact, they measure response time as data is stored and retrieved, and they also replicate data across geographical locations on users' request.

Another class of database is relational. This class of database compromise high availability and scalability for stronger consistency semantics and for providing higher query processing facilities. Current cloud providers offer this class of database in the form of Database-as-a-Service (DaaS) such as Amazon RDS, SQL Azure, and Google cloud SQL. DaaSs allow application providers to use the full capabilities of MySQL, MariaDB, PostgreSQL, Oracle, and Microsoft SQL Server DB engines. In fact, DaaSs support complex queries, distributed transactions, stored procedures, views, and so on, while they confront with several limitations mainly elasticity, strong consistency, transaction management across geographical locations, and live migration for purposes like multi-tenancy and elasticity. Moreover, similar to NoSQL databases, DaaSs support almost the same SLA and only can scale out/down based on manual rules defined by users according to the workload. DaaSs require more complex mechanisms to guarantee SLA, because they should manage resources like CPU, RAM, and disk. For more details on challenges and opportunities, the readers are referred to the survey on cloud-hosted databases proposed by Sakr (2014).

Figure 5 provides examples of discussed classes of databases, and Table 1 in the Appendix summarizes them in several main aspects. For more details on these databases, readers are referred to Sakr et al. (2011).

4 DATA DISPERSION

This section discusses the second element of data management in storage, data dispersion schemes, as shown in Figure 1.

4.1 Data Replication

Data replication improves availability, performance (via serving requests by different replicas), and user-perceived latency (by assigning requests to the closest replica) at the cost of replicas coordination and storage overheads. This is affected by facets of data replication based on the taxonomy in Figure 6.

4.1.1 Data Replication Model. There are two replication models for fault-tolerant data stores (Pedone et al. 2000): The first model is *state machine replication* (SMR) in which all replicas receive and process all operations in a *deterministic way* (in the same order) using *atomic broadcast* (Section 6.1.3). This implies SMR is abort-free and failure transparency, which means if a replica fails to process some operations those are still processed in other replicas. However, SMR has low throughput and scalability for read and write (RW) transactions, since all servers process each transaction. Thus, the scalability and throughput are confined by the processing power of a server. Scalable-SMR (S-SMR) (Bezerra et al. 2014) solves this issue across data stores via (i) partitioning database and replicating each partition and (ii) using cache techniques to reduce communication between partitions without compromising consistency. SMR and S-SMR are suitable for contention-intensive and *irrevocable transactions* that require abort-free execution.

The second model is Deferred-update replication (DUR). It resembles single/multi-master replication approach and scales better than SMR due to locally executing RW transactions on a server

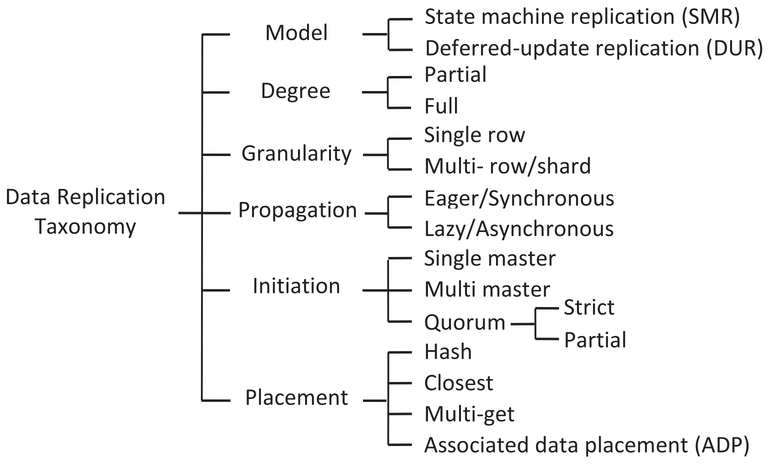

Fig. 6. Data replication taxonomy.

Table 8. Comparison between Replication Models

Replication Models	Advantages	Disadvantages	Consistency semantic
State machine replication†	Failure transparency Abort-free	Low throughput and scalability for RW transactions	Linearizability
Deferred-update replication	High throughput and scalability for RO transactions	Stale data Replica divergence	non-Linearizability

†It is also called *active replication.*

Table 9. Comparison between Full and Partial Replications

Properties	Comparison	Reason/Description
Scalability	Partial > Full	Due to access to a subset of DCs not all DCs
Complexity	Partial > Full	Due to the requirement for the exact knowledge where data reside
Storage Cost	Partial < Full	Due to data replication in a subset of data stores
Applicability	Partial > Full	If read requests come from a specific DCs, or when objects are write-intensive
	Partial < Full	If read requests come from all DCs, or when transactions are global

and then propagating updates to other servers. Thus, in DUR, the RW transactions do not scale with the number of replicas in comparison to the read-only (RO) transactions executed only on a server without communication across servers using a *multiversion* technique. Scalable-DUR (S-DUR) (Sciascia et al. 2012) and Parallel-DUR (P-DUR) (Pacheco et al. 2014) allow RW transactions to scale with the number of servers and the number of cores available for a replica, respectively. In respect to pros and cons summarized in Table 8, the scalability and throughput of transactions can be improved through borrowing the parallelism in DUR and abort-free feature in SMR (Kobus et al. 2013).

4.1.2 Data Replication Degree. Data replication can be either *partial* or *full*. In partial (respectively, full) replication each node hosts a *portion* (respectively, *all*) of data items. For example, in the context of Geo-DCs, *partial (respectively, full) replication* means that certain (respectively, all) DCs contain a replica of certain (respectively, all) data items.

Table 9 shows that partial replication outperforms full replication in storage services cost due to access to a subset of data stores deployed across DCs. It is also better than full replication in scalability, because full replication is restricted by the capacity of each replica that certifies and processes transactions. These advantages demand more complex mechanisms for consistency and transaction support in partial replication, which potentially degrades response time. Many partial replication protocols provide such mechanisms at the expense of communication cost same as full replication (Armendáriz-Iñigo et al. 2008). This is due to unpredictable *overlapping* transactions in which the start time of transaction T_i is less than the commit time of transaction T_j and the intersection of write set T_i and T_j is not empty. The deployment of *genuine partial replication* solves such issue and enforces a transaction to involve only the subset of servers/DCs containing the desired replicas for coordination. In terms of applicability, partial replication is suitable for write-intensive objects (due to submitting each request to a subset of DCs (Shen et al. 2015)), and full replication is proper for execution of global (multi-shard) transactions.

Therefore, the characteristics of workload and the number of DCs are main factors in making a decision on what *data replication degree* should be selected. If the number of DCs is small, then full replication is preferable; otherwise, if global transactions access few DCs, then partial replication is a better choice.

4.1.3 Data Replication Granularity. This defines the level of data unit that is replicated, manipulated, and synchronized in data stores. Replication granularity has two types: *single row* and *multi-row/shard*. The former naturally provides horizontal data partitioning, thereby allowing high availability and scalability in data stores like Bigtable, PNUTS, and Dynamo. The latter is the first step beyond single row granularity for new generation web-applications that require attaining both high scalability of NoSQL and ACID properties of RDBs (i.e., NewSQL features). This type of granularity has an essential effect on the scalability of transactions, and according to it, we classify transactions from granularity aspect in Section 6.1.5.

4.1.4 Update Propagation. This reflects *when* updates take place and is either *eager/synchronous* or *lazy/asynchronous*. In eager propagation, the committed data is simultaneously conducted on all replicas, while in lazy propagation the changes are first applied on the master replica and then on slave replicas. Eager propagation is applicable on a single data store like SQL Azure (Campbell et al. 2010) and Amazon RDS, but it is hardly feasible across data stores due to response time degradation and network bottleneck. In contrast, lazy propagation is widely used across data stores to improve response time.

4.1.5 Update Initiation. This refers to *where* updates are executed in the first place. Three approaches for update initiation are discussed as follows:

Single master approach deploys a replica at the closest distance to the user or a replica receiving the most updates as the master replica. All updates are first submitted to the master replica and then are propagated either eagerly or lazily to other replicas. In single master with lazy propagation, replicas receive the updates in the same order accepted in the master and might miss the latest versions of updates until the next re-propagation by the master. Single master approach has advantages and disadvantages as listed in Table 10. These issues can be mitigated somehow by *multi-master* approach in which every replica can accept update operations for processing and in turn propagates the updated data to other replicas either eagerly or lazily. Thus, this approach increases the throughput of read and write transactions at the cost of stale data, while replicas might receive the updates in the different order, which results in replicas divergence and thus the need for conflict resolution.

Quorum approach provides a protocol for availability vs. consistency in which writes are sent to a write set/quorum of replicas and reads are submitted to a read quorum of replicas. The set of read quorum and write quorum can be different and both sets share a replica as coordinator/leader. The reads and writes are submitted to a *coordinator replica*, which is a single master or multi-master.

Table 10. Comparison between Update Initiation

Update initiation	Advantages	Disadvantages	Data store(s)
Single Master†	Strong consistency†† Conflict avoidance	Low throughput of RW transactions Single point of failure and bottleneck	Amazon RDS◇ PNUT
Multi-master	High throughput of RO and RW transactions, not performance bottleneck	Stale data Replica divergence	Window Azure Storage (WAS)
Quorum	Adaptable protocol for availability vs. consistency	Determining the location of coordinator Determining the read and write quorum	Cassandra, Riak, Dynamo, Voldemort

†It is also called *primary backup*. ††Strong consistency is provided with *eager propagation*, not lazy. ◇ Amazon RDS: https://aws.amazon.com/rds/.

This protocol suffers from the disadvantages in determining the coordinator location and the quorum of write and read replicas as addressed when workload changes (Sharov et al. 2015). Though this classical approach guarantees strong consistency, many Geo-replicated data stores achieve higher availability at a cost of weaker consistency via its adaptable version in which a read/write is sent to all replicas and is considered successful if the acknowledgements are received from a quorum (i.e., majority) of replicas. This adapted protocol is configured with write (W) and read quorum (R) in synchronous writes and reads. The configuration is determined in (i) *strict* quorum in which any two quorums have non-empty intersection (i.e., $W + R > N$, where N is the number of replicas) to provide strong consistency, and (ii) partial quorum in which at least two quorums should not overlap (i.e., $W + R < N$) to support weak consistency. Generally speaking, (i) a rise in $\frac{W}{R}$ improves the consistency, and (ii) a raise in W reduces availability and increases durability.

4.1.6 Replica Placement. This is related to the mechanism of replica placement in data store and is composed of four categories. (1) *Hash mechanism* is intuitively understood as a random placement and determines the placement of objects based on the hashing outcome (e.g., Cassandra). It effectively balances the load in the system, however, it is not effective for a transactional data store that requires co-located multiple data items. (2) *Closest mechanism* replicates a data item in the node that receives the most requests for this data item. Although closest mechanism decreases the traffic in the system, it is not efficient for a transactional data store, because the related data accessed by a transaction might be placed in different locations. (3) *Multiget* (Nishtala et al. 2013) seeks to place the associated data in the same location without considering the localized data serving. (4) *Associated data placement* (ADP) (Yu and Pan 2015) makes the strike between *closest* and *multiget* mechanisms.

As discussed previously, using each strategy differed on various aspects of replication can affect the latency, consistency and transaction complexity, and even monetary cost. Among these, the key challenge is how to make a trade-off between consistency and latency, where update *initiation* and *propagation* are main factors. Data stores provide lesser latency and better scalability as these factors mandate weaker consistency (Table 2 in the Appendix).

4.2 Erasure Coding

Cloud file system uses *erasure coding* to reduce storage cost and to improve availability as compared to data replication. A (k, m)-erasure coding divides an object into k equally sized chunks that hold original data along with m extra chunks that contain data coding (parity). All these chunks are stored into $n = k + m$ disks, which increase the storage overhead by a factor of $1/r = k/n < 1$ and tolerates m faults, as opposed to $m - 1$ faults for m–way data replication with a factor of $m - 1$ storage overhead. For example, a (3,2)-erasure coding tolerates two failed replicas with a 2/3(=66%) storage overhead as compared to three-way replication with the same fault tolerance and a 200% storage overhead. To use erasure coding as an alternative to data replication, we need to investigate it in the following aspects as shown in Figure 7.

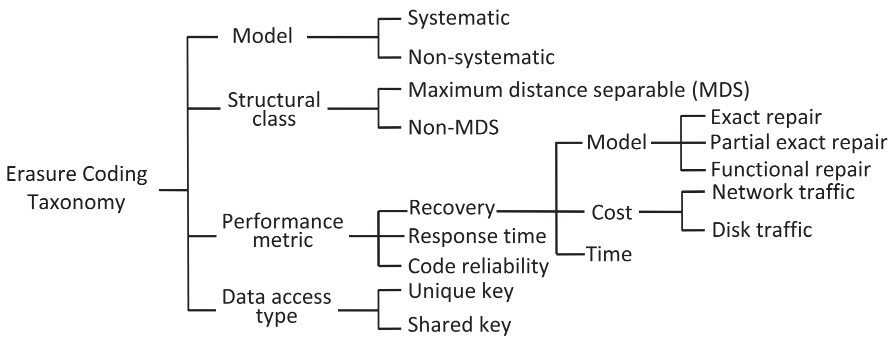

Fig. 7. Erasure coding taxonomy.

4.2.1 Model. Erasure coding has two models. *Systematic* model consists of *(original) data* and *parity* chunks, which are separately stored in $n - m$ and m storage nodes. *Non-systematic* model includes the coded data (not original data), which is stored in n nodes. Systematic and non-systematic codes are, respectively, relevant for *archival* and *data-intensive* applications due to, respectively, low and high rate of reads. Systematic codes seem more suitable for data stores, because they decode data when a portion of data is unavailable. Comparatively, non-systematic codes decode data whenever data are retrieved due to storing the coded data not the original data. This may degrade the response time.

4.2.2 Structural Class. This represents the reconstruct-ability of code that is largely reliant on erasure coding rate (i.e., r), including two classes. The first class is (k,m)-maximum distance separable (MDS) code in which the data is equally divided into k chunks, which are stored in $n = k + m$ storage nodes. A (k,m)-MDS code tolerates any m of n failed nodes and rebuilds the unavailable data from any k surviving nodes, known as MDS-*property*. A MDS code is optimal in terms of reliability vs. storage overhead while it incurs significant recovery overheads to repair failed nodes. The second class is non-MDS code. Any code that is not MDS code is called non-MDS code. This code tolerates less than m failed nodes and typically rebuilds the failed nodes from less than k surviving nodes. Thus, compared to MDS codes, non-MDS codes are (i) more economical in network cost to rebuild the failed nodes, which makes them more suitable for deploying across data stores and (ii) less efficient in the storage cost and fault-tolerance.

4.2.3 Performance Metrics. Erasure coding is evaluated based on the following performance metrics that have received significant attention in the context of cloud.

(1) *Recovery* refers to the amount of data retrieving from disk to rebuild a failed data chunk. Recovery is important in the following aspects.

- *Recovery model* in which a failed storage node is recovered through survivor nodes has three models (Suh and Ramchandran 2011). The first is *exact-repair* in which the failed nodes are exactly recovered, thus lost parity with their exact original data are restored. The second is *partial exact-repair* in which the data nodes are fixed exactly and parity nodes are repaired in a functional manner by using random-network-coding framework (Dimakis et al. 2010). This framework allows to repair a node via retrieving functions of stored data instead of subset of stored data so the code property (e.g., MDS-property) is maintained. The third is *functional repair* in which the recovered nodes contain different data from that of the failed nodes while the recovered system preserves the MDS-code property. Workload characteristics and the deployed code model determine which recovery model satisfies

Table 11. Comparison between Replication, Reed-Solomon Code, and Local Recostruction Code

Schemes	Structural Class	Recovery Cost†	Storage Overhead	Applicability
(m-way) Replication	n/a	1	(m-1)X	3-way replication in WAS, S3, and Google storage
(k,m)-RS	MDS	K	(1+$\frac{m}{k}$)X	(6,3)-RS in Google and (10,4)-RS in Facebook
(k,l,r)-LRC	Non-MDS	[r+1,r+l]	(1+$\frac{l+r}{k}$)X	(12,2,2)-LRC in WAS (Calder et al. 2011)

†n/a: not applicable, *Recovery time*: (m-way) Replication < (k,l,r)-LRC < (k,m)-RS. *Durability*: (m-way) Replication < (k,m)-RS < (k,l,r)-LRC.

the application requirements. Exact- and partial exact-repair are appropriate for archival applications while functional repair is suitable for non-secure sensitive applications, because it requires the dynamics of repairing and decoding rules that result in information leakage.

- *Recovery cost* is the total number of required chunks to rebuild a failed data chunk. It consists of *disk traffic* and *network traffic* affected by network topology and replication policy (Zhang et al. 2010). Recently, the trade-off between recovery cost and storage cost takes considerable attention in the context of clouds as discussed in the following.

  A (k,m)-Reed-Solomon (RS) code contains k data chunks and m parity chunks, where each parity chunk is computed from k chunks. When a data chunk is unavailable, there is always a need of any subset of k chunks from $m + k$ chunks, as recovery cost, to rebuild the data chunk. This code is used in Google ColossusFS (Ford et al. 2010) and Facebook HDFS (Muralidhar et al. 2014) within a DC (a XOR-based code–using pure XOR operation during coding computation—across DCs). In spite of the RS code optimality in reliability vs. storage overhead, it is still unprofitable due to high bandwidth requirements within and across data stores. Hitchhiker (Rashmi et al. 2014) mitigates this issue without compromise on storage cost and fault tolerance via the adapted RS code in which a single strip RS code is divided into two correlated sub-stripes.

  Similar to MDS-code, *regenerating* and non-MDS codes (Wu et al. 2007) alleviate the network and disk traffics. Regenerating codes aim at the optimal trade-off between storage and recovery cost and come with two optimal options (Rashmi et al. 2011). The first is the *minimum storage regenerating* (MSR) codes, which minimize the recovery cost keeping the storage overheads the same as that in MDS codes. NCCloud (Chen et al. 2014) uses *functional* MSR, as a non-systematic code, and maintains the same fault tolerance and storage overhead as in RAID-6. It also lowers recovery cost when data migrations happen across data stores due to either transition or permanent failures. The second is the *minimum bandwidth regenerating* (MBR) codes that further minimize the recovery cost, since they allow each node to store more data.

  A (k, l, r) Local Reconstruction Code (LRC) (Huang et al. 2012) divides k data blocks into l local groups and creates a *local parity* for each local group and $r = \frac{k}{l}$ *global parities*. The number of failure it can tolerate is between $r + 1$ and $r + l$. HDFS-Xorbas (Sathiamoorthy et al. 2013) exploits LRC to make a reduction of 2x in the recovery cost at the expense of 14% more storage cost. HACFS (Xia et al. 2015) code also uses LRC to provide a *fast code* with low recovery cost for *hot* data and exploits Product Code (PC) (Roth 2006)) to offer a compact code with low storage cost for *cold* data. Table 11 compares RS-code and LRC, used in current data stores, with data replication in the main performance metrics.

- *Recovery time* refers to the amount of time to read data from disk and transfer it within and across data stores during recovery. As recovery time increases, *response time* grows, resulting in a notorious effect on the data availability. Erasure codes can improve recovery time through two approaches. First, erasure codes should reduce network and disk traffics to which RS codes are inefficient as they read all data blocks for recovery. However, *rotated* RS (Khan et al. 2012) codes are effective due to reading the requested data only. LRC and XOR-base (Rashmi et al. 2014) codes are also viable solutions to decrease recovery time. Second, erasure codes should avoid retrieving data from hot nodes for recovery by replicating hot nodes' data to cold nodes or caching those data in dynamic RAM or solid-state drive.

(2) *Response time* indicates the delay of reading (respectively, writing) data from (respectively, in) data store and can be improved through the following methods. (i) *Redundant requests* simultaneously read (respectively, write) n coded chunks to retrieve (respectively, store) k parity chunks (Shah et al. 2014). (ii) *Adaptive batching* of requests makes a trade-off between delay and throughput, as exploited by S3 (Simple Storage Service) and WAS (McCullough et al. 2010). (iii) *Deploying erasure codes across multiple data stores* improves availability and reduces latency. Fast Cloud (Liang and Kozat 2014) and TOFEC (Liang and Kozat 2016) use the first two methods to make a trade-off between throughput and delay in key-value data stores as workload changes dynamically. Response time metric is orthogonal to data monetary cost optimization and is dependent on three factors: scheduling read/write requests, the location of chunks, and parameters that determine the number of data chunks. Xiang et al. 2014 considered these factors and investigated a trade-off between latency and storage cost within a data store.

(3) *Reliability* indicates the *mean time to data loss* (MTTDL). It is estimated by standard Markov model (Fine et al. 1998) and is influenced by the speed of block recovery (Sathiamoorthy et al. 2013) and the number of failed blocks that can be handled before data loss. LRCs are better in reliability than RS codes, which in turn, are more reliable than replication (Huang et al. 2012) with the same failure tolerance. As already discussed, failures in data stores can be *independent* or *correlated.* To relieve the later failure, the parity chunks should be placed in different racks located in different domains.

4.2.4 Data Access Type. There are two approaches to access chunks of the coded data: *unique key* and *shared key* (Liang and Kozat 2016), in which a key is allocated to a chunk of coded data and the whole coded data, respectively. These approaches can be compared in three aspects. (1) *Storage cost*: both approaches are almost the same in the storage cost for writing into a file. In contrast, for reading chunks, shared key is more cost-effective than unique key. (2) *Diversity in delay*: with unique key, each chunk, treated as an individual object, can be replicated in different storage units (i.e., server, rack, and DC). With shared key, chunks are combined into an object and very likely stored in the same storage unit. Thus, in unique (respectively, shared) key, there is a low (respectively, high) correlation in the access delay for different chunks. (3) *Universal support*: unique key is supported by all data stores, while shared key requires advanced APIs with the capability of partial reads and writes (e.g., S3).

4.3 Hybrid Scheme

Hybrid scheme is a combination of data replication and erasure coding schemes to retain the advantages of these schemes while avoiding their disadvantages for data redundancy within and across data stores. Table 12 compares two common schemes in performance metrics to which

Table 12. Comparison between Replication and Erasure Coding Schemes

Schemes	Availability	Durability	Recovery	Storage Overhead	Repair Traffic	Read/Write latency
Replication	High	Low	Easy	>1X	=1X	Low for hot-spot objects with small size
Erasure Coding	Low	High	Hard	<1X	>1X	Low for cold-spot objects with large size

Table 13. Comparison between the State-of-the-Art Projects Using Different Redundancy Schemes

Projects	Redundancy Scheme	Contributing Factors	Objective(s)
DepSky (Bessani et al. 2011)	Replication	AR†, Pr	High availability, integrity, and confidentiality
Spanner (Wu et al. 2013)	Replication	OS, AR, Pr	Cost optimization and guaranteed availability
CosTLO (Wu et al. 2015)	Replication	OS, AR, Pe	Optimization of variance latency
SafeStore (Kotla et al. 2007)	Erasure coding	AR, Pr	Cost optimization
RACS (Abu-Libdeh et al. 2010)	Erasure Coding	AR	Cost optimization and data-lock in
HAIL (Bowers et al. 2009)	RAID technique	n/a	High availability and integrity
NCCloud (Chen et al. 2014)	Network Codes	n/a	Recovery cost optimization of lost data
CDStore (Li et al. 2015)	Reed-Solomon	n/a	Cost optimization, security and reliability
CAROM (Ma et al. 2013)	Hybrid (RAC)	AR	Cost optimization
CHARM (Zhang et al. 2015)	Hybrid (ROC)	AR, Pr	Cost optimization and guaranteed availability
HyRD (Mao et al. 2016)	Hybrid (ROC)	OS, Pr, Pe	Cost and latency optimization

†n/a: (not applicable), OS: (object size), AR: (access rate), Pr: (price), and Pe (performance).

three factors contribute into when and which scheme should be deployed: Access rate (AR) to objects, object size (OS), price (Pr), and performance (Pe) of data stores.

These factors have a significant effect on the *storage overhead*, *recovery cost*, and *read/write latency*. As indicated in Table 12, replication incurs storage overhead more than erasure coding especially for large objects, while it requires less recovery cost due to retrieving the replica from a single server/data store instead of fragmented objects from multiple servers/data stores. Erasure coding is more profitable in read/write latency (i) for cold-spot objects, since update operations require re-coding the whole object, and (ii) for large objects, because the fragmented objects can be accessed in parallel from multiple server/data stores. For the same reasons, replication is more efficient for hot-spot objects with small size like metadata objects. Thus, cold-spot objects with large size should be distributed across cost-effective data stores in the form of erasure coding, and hot-spot objects with small size across performance-efficient data stores in the form of replication.

We classify the hybrid scheme into two categories. (i) *Simple hybrid* stores an object in the form of either replication or erasure coding (ROC) or replication and erasure coding (RAC) during its lifetime. (ii) *Replicated erasure coding* contains replicas of each chunk of coded objects and its common form is *double coding*, which stores two replicas of each coded chunk of the object. Compared to ROC, double coding and RAC increase storage cost two times, but they are better in availability and bandwidth cost due to retrieving the lost chunk of the object from the server, which has a redundant copy. Table 13 summarizes projects using common redundancy or hybrid schemes. Neither workload characteristics nor data stores diversities (in performance and cost) are fully deployed in these projects using hybrid scheme. It is an open question to investigate the effect of these characteristics and diversities on the hybrid scheme.

5 DATA CONSISTENCY

Data consistency, as the third element of data management in storage, means that data values remain the same for all replicas of a data item after an update operation. It is investigated in three main aspects, *level*, *metric*, and *model*, as shown in Figure 8. This section first describes different

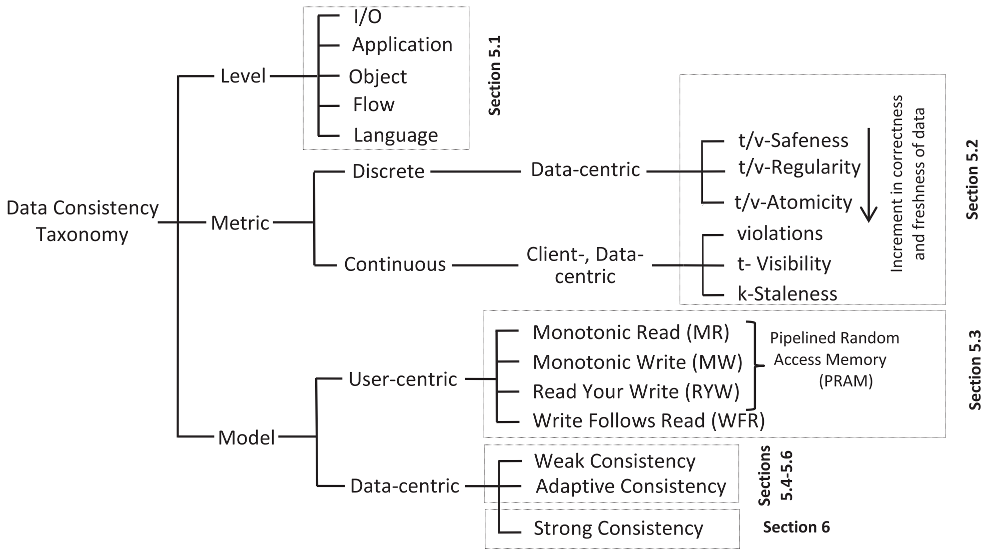

Fig. 8. Data consistency taxonomy.

consistency levels and their pros and cons (Section 5.1). Then it defines the consistency metrics to determine how much a consistency semantic/model is stronger than another (Section 5.2). Finally, it discusses consistency models from the user-perspective (Section 5.3) and from the data store perspectives (Sections 5.4–5.6).

5.1 Consistency Level

Distributed data stores rely on different levels of data consistency (Alvaro et al. 2013), as shown in Figure 8.

I/O-level consistency allows a clear separation between low-level storage and application logic. It simplifies the development of the application and the complexity of distributed programming. However, I/O-level consistency requires conservative assumptions like concurrent write-write and read-write on the same data, resulting in inefficiency. It should also execute writes and reads in a serial order due to its unawareness of the application semantics. *We focus on this level of consistency in this article.*

application-level consistency exploits the semantics of the application to ensure the concreteness of invariants[8] without incurring the cost of coordination among operations. Thus, it imposes a burden on the developers and sacrifices the generality/reusability of the application code.

Object-level consistency makes a trade-off between *efficiency* and *reusability*, respectively, degraded by I/O- and application-level consistency. It provides the convergence of replicas to the same value without any need of synchronization across replicas via Conflicted-free Replicated Data Types (CRDTs) (Shapiro et al. 2011), in which the value of objects can change in an *associative*, *commutative*, and *idempotent* fashion. Though object-level consistency removes concerns of the reusability of application-level consistency, it requires mapping the properties of the application to invariants over objects by developers.

[8]The term invariant refers to a property that is never violated (e.g., primary key, foreign key, a defined constraint for the application—for example, an account balance $x \geq 0$).

Flow-level consistency is an extension of object-level consistency and requires a model to obtain both the semantic properties of *dataflow component* and the dependency between interacting components.[9] Some components are insensitive to message order delivery as a semantic property,[10] and they are *confluent* and produce the same set of outputs under all ordering of their inputs (Alvaro et al. 2014). Flow-level consistency demands manual definition for confluent components, resulting in error-prone. But it is more powerful than object-level consistency and it has more applicability than object- and language-level consistency. Indigo (Balegas et al. 2015) exploits flow-level consistency based on confluent components that contain the application-specific correctness rules that should be met.

Finally, *language-level* consistency integrates the semantics and dependencies of the application and maintains a long history of invariants to avoid distributed coordination across replicas. The CALM principle (Alvaro et al. 2011) shows a strong connection between the need of distributed coordination across replicas and *logical monotonicity*. *Bloom* language (Alvaro et al. 2011) deploys this principle and translates *logical monotonicity* into a practical program that is expressible via *selection*, *projection*, *join*, and *recursion* operations. This class of program, called *monotonic program*,[11] provides output as it receives input elements, and thus guarantees eventual consistency under any order of inputs set. Unlike Bloom, QUELEA language (Sivaramakrishnan et al. 2015) maps operations to a fine-grained consistency levels such as *eventual*, *causal*, and *ordering* and transaction isolation levels like *read committed* (RC), *repeatable read* (RR) (Berenson et al. 1995), and *monotonic atomic view* (MAV) (Bailis and Ghodsi 2013).

5.2 Consistency Metric

This determines how much a consistency model is stronger than another and is categorized into *discrete* and *continuous* from data store perspective (i.e., data-centric) and user perspective (i.e., client-centric).

Discrete metrics are measured with the maximum number of time unit (termed by *t-metric*) and data version (termed by *v-metric*). As shown in Figure 8, they are classified into three metrics (Anderson et al. 2010): (i) *Safeness* mandates that if a read is not concurrent with any writes, then the most recent written value is retrieved. Otherwise, the read returns any value. (ii) *Regularity* enforces that a read concurrent with some writes returns either the value of the most recent write or concurrent write. It also holds *safeness* property. (iii) *Atomicity* ensures the value of the most recent write for every concurrent or non-concurrent read with write.

Continuous metrics, shown in Figure 8, are defined based on *staleness* and *ordering*. The former metric is expressed in either *t-visibility* or *k-staleness* with the unit of probability distribution of *time* and *version lag*, respectively. The latter one is measured as (i) *the number of violations* per time unit from data-centric perspective and (ii) the *probability distribution of violations* in the forms of MR-, MW-, RYW-, WFR-violation from client-centric perspective as discussed later.

5.3 Consistency Model

This is classified into two categories: *user-* and *data-centric*, which, respectively, are vital to application and system developers (Tanenbaum and Steen 2006).

[9]A component is a logical unit of computing and storage and receives streams of inputs and produces streams of outputs. The output of a component is the input for other components, and these streams of inputs and outputs implement the flow of data between different services in an application.
[10]The semantic property is defined by application developers. For example, developers determine confluent and non-confluent paths between components based on analysis of a component's input/output behavior.
[11]Non-monotonic program contains aggregation and negation queries, and this type of program is implementable via block algorithms that generate output when they receive the entire inputs set.

User-centric consistency model is classified into four categories as shown in Figure 8. *Monotonic Read* (MR) guarantees that a user observing a particular value for the object will never read any previous value of that object afterward (Yu and Vahdat 2002). *Monotonic Write* (MW) enforces that updates issued by a user are performed on the data based on the arrival time of updates to the data store. *Read Your Write* (RYW) mandates that the effects of all writes issued by users are visible to their subsequent readers. *Write Follows Read* (WFR) guarantees that whenever users have recently read the updated data with version *n*, then the following updates are applied only on replicas with a version$\geq$ *n*. Pipelined Random Access Memory (PRAM) is the combination of MR-, MW-, and RYW-consistency and guarantees the serialization in both of reads and writes within a session. Brantner et al. (2008) designed a framework to provide these client-centric consistency models and the atomic transaction on Amazon S3. Also, Bermbach et al. (2011) proposed a middle-ware on eventually consistent data stores to provide MR- and RYW-consistency.

Data-centric consistency aims at coordinating all replicas from the internal state of data store perspective. It is classified into three models. *Weak consistency* offers low latency and high availability in the presence of network partitions and guarantees *safeness* and *regularity*. But it causes a complicated burden on the application developers and caters the user with the updated data with a delay time called *inconsistency window* (ICW). In contrast, *strong consistency* guarantees simple semantics for the developer and *atomicity*. But it suffers from long latency, which is eight times more than that of weak consistency, and consequently its performance in reads and writes diverges by more than two orders of magnitude (Terry et al. 2013). *Adaptive consistency* is switching between a range of weak and strong consistency models based on the application requirements/constraints like availability, partition tolerance, and consistency.

The consistency (**C**) model has a determining effect on achieving availability (**A**) and partition tolerance (**P**). Based on the CAP theorem (Gilbert and Lynch 2002), data stores provide only two of these three properties. In fact, data stores offer only CA, CP, or AP properties, where CA in CAP is a better choice within a data store due to rare network partition, and AP in CAP is a preferred choice across data stores. Recently, Abadi (2012) redefined CAP as PACELC to include latency (**L**) that has a direct influence on monetary profit and response time, especially across data stores, where the latency between DCs might be high. The term PACELC means that if there is a network partition (**P**) then there is a choice between **A** and **C** for designers, else (**E**) the choice is between **L** and **C**. Systems like Dynamo and Riak leave strong consistency to achieve high availability and low latency. Thus, they are PA/EL systems. Systems with full ACID properties attain stronger consistency at the cost of lower availability and higher latency. Hence, they are PC/EC systems. See Table 1 in the Appendix for more examples.

5.4 Eventual Consistency

In this section, we first define the eventual consistency model (Section 5.4.1). Then, we discuss how this model is implemented and describe how the conflicts that arise from this model are solved (Section 5.4.2).

5.4.1 Definition. Eventual consistency is defined as all replicas eventually converge to the last update value. It purely supports *liveness*, which enforces that all replicas eventually converge based on the operations order, while lacking *safety*, which determines the correct effects of operations and leads to incorrect intermediate results. The safety property is assessed in terms *t-visibility* and *k-staleness* as *inconsistency window* (ICW), which is affected by the communication latency, system load, and replicas number. Bailis et al. (2012) analytically predicted the value of ICW via probabilistically bounded staleness (PSB) based on the quorum settings (Section 4.1.5) for quorum-based data stores. Wada et al. (2011) also experimentally measured ICW with different configurations in terms

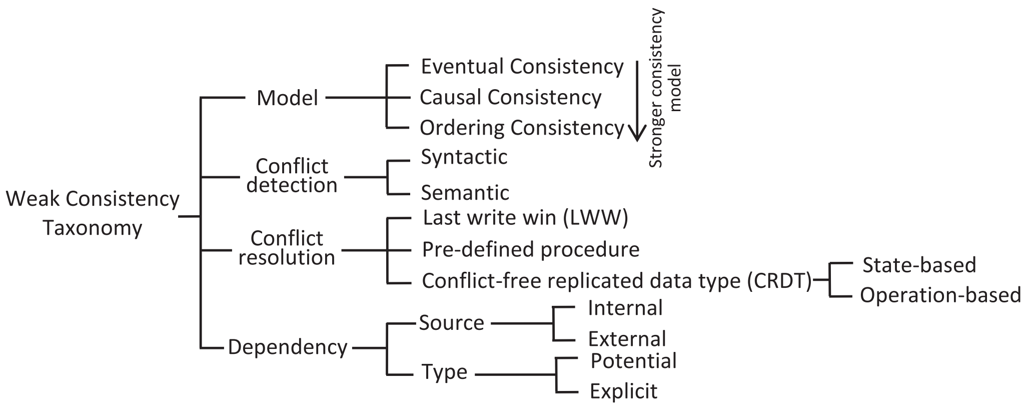

Fig. 9. Weak consistency taxonomy.

of readers and writers in different threads in the same/different processes on the same/different VMs for S3, Google App Engine (GAE), and SimpleDB. They concluded that S3 and GAE are "good enough" in data freshness with eventual consistency option, while SimpleDB is not. Zhao et al. (2015) also experimentally measured ICW on the virtualization-based DaaS and concluded that the workload characteristics has more effects on ICW as compared to communication latency across geographical locations. It is worth to note that neither NoSQL databases nor DaaSs guarantee a specific SLA in terms of data freshness.

5.4.2 Implementation. Eventual consistency-based data stores employ optimistic/lazy replication in which (i) the operation is typically submitted to the closest replica and logged/remembered for the propagation to other replicas later, and (ii) replicas exchange the operation or the effect of operation among each other via *epidemic/gossip protocol* (Demers et al. 1987) in the background (Saito and Shapiro 2005). Operations are partially ordered by deploying vector clocks. This leads to data *conflicts*, which happen as operations are simultaneously submitted to the same data in multi-master systems.

There are four approaches to deal with conflicts. (i) *Conflict detection* approaches strengthen the application semantic and avoid the problems arising from ignoring conflicts. These approaches are classified into *syntactic* and *semantic* (Saito and Shapiro 2005). The syntactic approach relies on the logical or physical clock, whereas the semantic approach works based on the semantic knowledge of operations such as invariants, commuting updates (i.e., CRDTs), and pre-defined procedure. (ii) Conflicts *prohibition* is attainable via blocking or aborting operations and using a single master replica, which comes at the expense of low availability. (iii) Conflicts *ignorance* and *reduction* are achievable by the following conflict resolution techniques (Figure 9) to guarantee *safety*.

(1) *Last Write Win (LWW)* (Thomas 1979) ignores conflicts, and the update with the highest timestamp is accepted (e.g., Riak (by default), SimpleDB, S3, and Azure Table). It causes lost updates (i.e., updates with less timestamp as compared with winner update) and the violation of expected semantics. (2) *Pre-defined procedure* merges two versions of a data item to a new one according to application-specific semantic as used in Dynamo. The merged data must be *associative* and *commutative* for guaranteeing eventual/causal consistency. Albeit the pre-defined procedure solves conflicts without the need of the total order, it is error-prone and lacks generality. Some data stores use *application-specific precondition* (i.e., a condition or predicate that must always be true just prior

to the execution of other conditions) to determine *happened-before dependencies* among requests when the causal consistency comes as a need.

(3) *Conflict-free replicated data types* (CRDTs) avoid the shortcomings of the previously discussed approaches and provide eventual consistency in the presence of node failure and network partition without cross replicas coordination. CRDTs enforce the convergence to the same value after all updates are executed on replicas, and they are either *operation-based* or *state-based* (Shapiro et al. 2011).

In *state-based* CRDT, the local replica is first updated and then the modified data is transmitted across replicas. State-based CRDT pursues a partial order $\leq_v$ (e.g., integer order) with *least upper bound* (LUB) $\sqcup_v$ (e.g., maximum or minimum operation between integer numbers) that guarantees *associative* (i.e., $(a_1 \sqcup_v a_2) \sqcup_v a_3 = a_1 \sqcup_v (a_2 \sqcup_v a_3)$), *commutative* (i.e., $a_1 \sqcup_v a_2 = a_2 \sqcup_v a_1$), and *idempotent* (i.e., $a_1 \sqcup_v a_1 = a_1$) properties, for each value of object a_1, a_2, a_3 (e.g., integer numbers). Such CRDT is called *Convergent Replicated Data Type* (CvRDT) and is used in Dynamo and Riak. CvRDT can tolerate out-of-order, repeatable, and lost messages as long as replicas reach the same value. Thus, CvRDT achieves eventual consistency without any coordination across replicas, but it comes at the expense of monetary cost and communication bottleneck for transferring large objects particularly across data stores. Almeida et al. (2015) addressed this issue by propagating the effect of recent update operations on replicas instead of the whole state; meanwhile all properties of CvRDT are maintained. As an example of CvRDT, consider Grow-only set (G-set) that supports only *union* operations. Assume a partial order $\leq_v$ on two replicas of G-set S_1 and S_2 is defined as $S_1 \leq_v S_2 \iff S_1 \subseteq S_2$ and *union* is performed as $S_1 \cup S_2$. Since the *union* operation preserves the mentioned three properties, G-set is a CvRDT.

In *operation-based* CRDT, first the update is applied to the local replica, and then it is asynchronously propagated to the other replicas. Operation-based CRDT demands a reliable communication network to submit all updates to every replica in a delivery order $\leq_v$ (specified by data type) with *commutative* property (Shapiro et al. 2011), as utilized in Cassandra. If all concurrent operations are commutative, then any order of operations execution converges to an identical value. Such data type is called *Commutative Replicated Data Type* (CmRDT) and is more useful than CvRDT in terms of data transferring for applications that span write-intensive replicas across data stores. This is because CmRDT demands less bandwidth to transfer *operation* across replicas, as compared to CvRDT that transfers the effect of the operation. For instance, G-set is also CmRDT, because *union* is commutative. Similar to CvRDT, CmRDT allows the execution of updates anywhere, anytime, and in any order, but they have a key shortcoming in guaranteeing integrity constraints and invariants across replicas.

5.5 Causal and Causal+ Consistency

We first introduce a formal definition of causal and causal+ consistency models (Section 5.5.1), followed by a description of the source and type of dependencies found in this model (Section 5.5.2). We then discuss the state-of-the-art projects supporting these models of consistency (Section 5.5.3).

5.5.1 Causal Consistency Definition. Causal consistency maintains the merits of eventual consistency, while respecting to the causality order among requests applied to replicas. It is stronger and more expensive than eventual consistency due to tracking and checking dependencies. It defines Lamport's *"happens-before"* relation (Gray and Lamport 2006) between operations o_1 and o_2 as $o_1 \rightsquigarrow o_2$. Potential causality $o_1 \rightsquigarrow o_2$ maintains the following rules (Ahamad et al. 1995). *Execution thread*: If o_1 and o_2 are two operations in the same thread of execution, then $o_1 \rightsquigarrow o_2$ if o_1 happens before o_2. *Read from*: If o_1 is a write, and o_2 is a read and returns the value written by o_1, then $o_1 \rightsquigarrow o_2$. *Transitivity*: if $o_1 \rightsquigarrow o_2$ and $o_2 \rightsquigarrow o_3$, then $o_1 \rightsquigarrow o_3$. Causal consistency does not support concurrent operations (i.e., $a \not\rightsquigarrow b$ and $b \not\rightsquigarrow a$). According to this definition, the write operation happens if all write operations having causal dependency with the given write have occurred before. In other words, if $o_1 \rightsquigarrow o_2$, then o_1 must be written before o_2. Causal+ is the combination of *causal* and *convergent conflict resolution* to ensure *liveness* property. This consistency model allows users locally receive the response of read operations without accessing remote data store, meanwhile the application semantics are preserved due to enforcing causality on operations. However, it degrades scalability across data stores for write operations, because each DC should check whether the dependencies of these operations have been satisfied or not before their local commitment. This introduces a trade-off between *throughput* and *visibility*. *Visibility* is the amount of time that a DC should wait for checking the required dependencies among the write operations in the remote DC, and can be influenced by *network latency* and *DC capacity for checking dependencies*.

5.5.2 Dependency Source and Type. Dependencies between operations are represented by a graph in which each vertex represents an operation on variables and each edge shows the causality of a dependency between two operations. The source of dependencies can be *internal* or *external* (Du et al. 2014a). The former refers to causal dependencies between each update and previous updates in the same session, while the latter relates to causal dependency between each update and updates created by other sessions whose values are read in the same session. COPS (Lloyd et al. 2011), Eiger (Lloyd et al. 2013), and Orbe (Du et al. 2013) track both dependency sources. Dependency types can be either *potential* or *explicit* for an operation (as in Eiger and ChainReaction (Almeida et al. 2013)) or for a value (as in COPS and Orbe). Potential dependencies capture all possible influences between data dependencies, while explicit dependencies represent the semantic causality of the application level between operations. The implementation of potential dependencies in modern applications (e.g., social networks) can produce large metadata in size and impede scalability due to generating large dependencies graph in the degree and depth. The deployment of explicit dependencies, as used in Indigo (Balegas et al. 2015), alleviates these drawbacks to some extent, but it is an ad-hoc approach and cannot achieve the desired scalability in some cases (e.g., in social applications). This deployment is made more effective with the help of garbage collection, as used in COPS and Eiger, in which the committed dependencies are eliminated and only the nearest dependencies for each operation are maintained.

5.5.3 Causally Consistent Data Stores. Causal consistency recently received significant attention in the context of Geo-replicated data stores. **COPS** (Lloyd et al. 2011) provides causal+ consistency by maintaining metadata of causal dependencies. In COPS, a read is locally submitted, and the update operations become visible in a DC when their dependencies are satisfied. COPS supports a causally consistent read-only (RO) transactions, which return the version of objects. To do so, it maintains a full list of all dependencies and piggybacks them when a client issues read operations, as opposed to maintaining the nearest dependencies for providing causal+ consistency. **Eiger** (Lloyd et al. 2013) supports the same consistency model for the column-family data model and provides RO and WO transactions. It maintains fewer dependencies and eliminates the need for garbage collection as compared to COPS. **ChainReaction** (Almeida et al. 2013) uses a variant of chain replication (van Renesse and Schneider 2004) to support causal+ consistency. Contrary to COPS requiring each DC to support *serliazability* (see Section 6), it uses two logic clocks (LCs) for each object: the first one is a global LC, and the second one is local LC that determines which local replica can provide causal consistency. ChainReaction provides RO transactions the same as COPS whilst averting 2 round trip times (RTTs) (as required in COPS) by deploying a sequencer in each DC. Although the sequencer reduces the number of RTT (at most 1 RTT) in the case of RO transactions, it reduces scalability and increases the latency for all updates by 1 RTT within DC. **Orbe** (Du et al. 2013) deploys DM-protocol (exploiting a matrix clock) to support basic reads and writes and DM-clock protocol (deploying physical clock to track dependencies) to provide RO transactions the same as those in ChainReaction that completes in 1 RTT. Orbe avoids the downsides of ChainReaction due to not using a centralized sequencer. **GentleRain** (Du et al. 2014b) eliminates dependencies checking and reduces the metadata piggybacked to each update propagation. This innovation achieves throughput analogous with that in eventual consistency and reduces storage and communication overheads. To achieve such aims, GentleRain uses physical time and allows a DC to make the update visible if all partitions within the DC have seen all updates up to the remote update timestamp. Nevertheless, this technique deteriorates updates visibility in remote DCs. **Bolt-on causal consistency** (Bailis et al. 2013) inserts a layer between clients and data stores to provide causal consistency according to the semantics of the application, not the deployment of LCs or physical clocks. The discussed projects are summarized in Table 3 in the Appendix.

5.6 Ordering and Adaptive-Level Consistency

We first define ordering consistency model and how it is provided. We then discuss projects that enable application providers to switch between a range of consistency models based on their requirements.

5.6.1 Ordering Consistency. As discussed earlier, eventual consistency applies the updates in different orders at different replicas and causal consistency enforces partial ordering across replicas. In contrast, *ordering consistency*—also called *sequential consistency*—provides a global ordering of the updates submitted to replicas by using a logical clock to guarantee monotonic reads and writes. In fact, *ordering consistency* mandates a read operation from a location to return the value of the last write operation to that location. Ordering consistency is guaranteed through deploying *chain replication* (van Renesse and Schneider 2004) or a master replica, which is responsible for ordering writes to an object and then propagating the updates to slave replicas as provided per key in PNUTS.

5.6.2 Adaptive-level Consistency. Adaptive-level consistency switches between weak and strong consistency models based on the requirements of applications to reduce response time and monetary cost. The following data stores leverage adaptive-level consistency.

CRAQ (Terrace and Freedman 2009) switches between strong, causal, and eventual consistency for reading objects replicated in a chain replication topology. CRAQ's current implementation relies on placing chains within a DC with the capability of stretching on multiple DCs. It provides a single row (object) consistency and *mini-transactions* that update multiple objects in a single or multiple chains. **Pileus** (Terry et al. 2013) offers a broad spectrum of consistency levels between strong and eventual based on SLAs like latency. It supports all kinds of transactions, but does not scale well, because all writes are assigned to the primary replica without automatic movement to other DCs in the face of workload changes. **Gemini** (Li et al. 2012) switches between strong and eventual consistency based on *blue* and *red* operations, which are, respectively, executed on different and same order at different DCs. It also introduces the concept of *shadow* operations to increase blue operations for improving response time and throughput. The consistency level of such operations can be manually assigned by developers based on a specified method as in Gemini or based on the SLA as in Pileus. These methods are non-trivial and cumbersome, and **SIEVE** (Li et al. 2014) alleviates this by having an automatic assignment mechanism that exploits application code, invariants, and CRDTs. SIEVE incurs low run-time overheads, but it lacks scalability as applications code becomes large. The discussed projects are summarized in Table 4 in the Appendix.

6 TRANSACTIONS

This section discusses the fourth element of data management in cloud storage, *transaction*. A *transaction* is defined as a set of reads and writes followed by a *commit* if the transaction is completed or an *abort* otherwise. A transaction has four properties: (i) *atomicity* guarantees *all-or-nothing* outcomes for the set of operations, (ii) *consistency* ensures any transaction transits the database from one valid state to another, (iii) *isolation* ensures the concurrent execution of transactions leads to a database state that would be obtained if transactions were executed one after the other, (iv) *durability* means that once a transaction is committed, all the changes have been recorded to a durable data store. These four properties are called ACID properties. Of these properties, *isolation* concurrency level (*isolation level* for short) refers to a control mechanism for executing two concurrent transactions accessing the same data item. Isolation level expresses the consistency semantic provided by the transaction. Data stores initially provided eventual consistency, which is suitable for certain applications (e.g., social networks), while some classes of applications (e.g., e-commerce) demand strong consistency. Data stores, therefore, shifted to guarantee single row transactions (e.g., SimpleDB (Wada et al. 2011) and PNUTS (Cooper et al. 2008)), single shard transactions (e.g., SQL Azure (Campbell et al. 2010)), and multi-shard transactions where shards are geographically replicated (e.g., Spanner (Corbett et al. 2013)).

Serialiazability as the strongest isolation level is not scalable across data stores, because it requires strict coordination between replicas probably located far from each other across data stores. Such isolation level has overheads like latency increment, throughput reduction, and unavailability in the case of a network partition. Thus, data stores leave serialiazability, and they resort to (i) offer weaker isolation levels, (ii) partition and replicate data accessed by transactions at a limited number of servers/DCs, and (iii) exploit techniques to optimize coordination as the key requirement to maximize scalability, availability, and performance. Ardekani et al. (2013) stated the following criteria for achieving these purposes across data stores. (1) *Wait-free query* (WFQ) means that a read-only (RO) transaction always commits without synchronizing with other transactions. (2) Genuine Partial Replication (GPR) decreases the synchronization and computational time. (3) *Minimal commitment synchronization* is achievable if synchronization is not avoidable (e.g., two transactions with write-write conflicts must be synchronized). (4) *Forward freshness* is attainable if a transaction is allowed to read the committed object version after the transaction starts. Unfortunately, classical concurrency protocols (e.g., 2PL+2PC) are not scalable and purely applicable across the data stores to achieve the goals discussed previously. To understand how to effectively deploy classical protocols, we study the main aspects of transactional data stores in the following section.

6.1 Transactional Data Stores

This section details a taxonomy of transactional data stores as shown in Figure 10.

6.1.1 Architecture. To build an ACID transactional data store, a stacked layer consisting of *transaction and replication layers* is used. The transaction layer consists of, shown in Figure 11(b), (i) *concurrency control mechanism* (Kung and Robinson 1981) that schedules a transaction within each shard to guarantee *isolation*, and (ii) an *atomic commitment protocol* that coordinates distributed transactions across shards to provide *atomicity*. The replication layer is responsible for synchronizing replicas in the case of strong consistency. Figure 11 shows different architectures that indicate how transaction and replication layers are configured (Agrawal et al. 2013). Notable systems like Spanner (Corbett et al. 2013)) preserves the *top-down* architecture (Figure 11(a)), while Replicated Commit (Mahmoud et al. 2013) follows the *reversed top-down* architecture (Figure 11(b)) to achieve lower latency. In the *flat* architecture (Figure 11(c)), both layers access the storage layer, and there is no clear border between them as deployed in Megastore (Baker

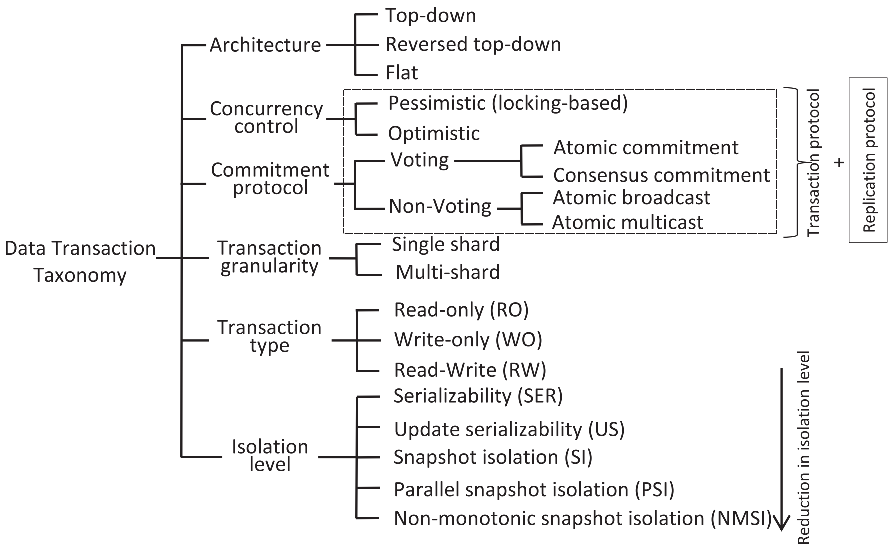

Fig. 10. Data transaction taxonomy.

et al. 2011). The separation of the two layers in the architecture brings advantages like modularity, clarity of semantic, and less message-exchange.

6.1.2 Concurrency Control. This schedules transactions accessing the same data so the *isolation* property is guaranteed and is categorized into *pessimistic (locking-based)* and *optimistic*. In the *pessimistic* protocol, a transaction first locks (e.g., via 2 Phase Lock (2PL)) the shared data accessed by concurrent transactions, and then operates on data. This protocol serializes transactions upon the occurrence of conflict incidences and requires a deadlock detection and resolution mechanism. Therefore, it increases response time especially across data stores, reduces the throughput of data store (expressed as the number of committed transactions per time unit), hurts availability if the lock is held by a failed node, and suffers from *thrashing* (i.e., when the number of transactions is high, many of them become blocked and only a few are in progress). Nevertheless, it eases write-write conflicts detection without maintaining the transaction metadata and works best when conflicts among transactions are short running. On the contrary, in *optimistic* protocol (optimistic concurrency control (OCC)), all transactions are executed separately though each of them before commitment needs to pass a *certification procedure* in which the write set of transaction T that is being validated against the read and write sets of other active transactions in the system. Thus, any read-write or write-write conflicts result in OCC to be aborted. To avoid such procedure, MVOCC (Vo et al. 2012) combines multi-version concurrency control (MVCC) and OCC to detect conflicts via the version number of data. Thus, MVOCC always commits RO transactions, while uses the version number of data to check their conflicts for update transactions. However, OCC provides low latency at the cost of weak concurrency control, fits when transactions are long running and avoids drawbacks arising from the pessimistic protocol though it wastes resources due to restarting transaction and requiring exclusive lock during final 2 Phase Commit (2PC) (Agrawal et al. 1987). MaaT (Mahmoud et al. 2014), as a re-design of OCC, eliminates locks during 2PC to reduce aborts rate, avoids MVCC to make efficient use of memory, and improves throughput; meanwhile, MaaT maintains the *no-thrashing* property of OCC.

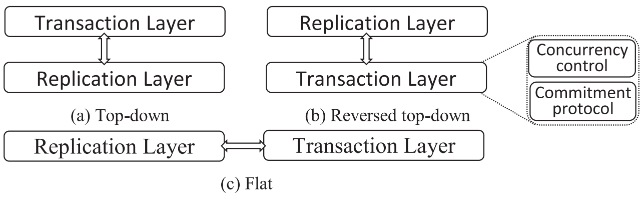

Fig. 11. Transaction architectures in cloud storage.

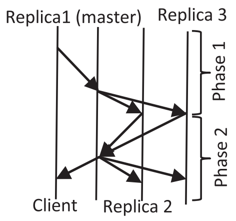

Fig. 12. Execution of basic Paxos.

6.1.3 Commitment Protocol. This refers the way transactions terminate while guaranteeing *atomicity* and *consistency*. It is generally categorized into two types as follows:

(1) *Voting* protocols agree on updates based on voting all replicas as in *atomic commitment protocol* (e.g., 2PC) or quorum of replicas as in *consensus commitment protocol* (e.g., Paxos commitment (Gray and Lamport 2006)). In 2PC protocol, a node is designated as the *coordinator/master* node and other nodes in the system as *participants*, *cohorts*, or *workers*. This protocol consists of (i) a *voting phase* in which a coordinator node prepares all participant nodes to take either commit ("yes") or abort ("no") vote on the transaction, and (ii) *commit phase* in which the coordinator decides whether to commit or abort the transaction based on vote of participant nodes in the previous phase, and then notifies about its decision to all the participant nodes. This protocol suffers from blocking and non-resilience to node failures. By contrast, Paxos protocol is fault-tolerant, non-blocking, and a consensus algorithm for achieving on a single value among a set of replicas and tolerates a variety of failures like duplicated, lost, and reordered messages as well as the failure of nodes. In the deployed applications across data stores, storage and computing nodes take the responsibility of four basic roles in Paxos: *client* are application servers, *proposers* are coordinators, *acceptors* are storage nodes, and all nodes are *learners*. As shown in Figure 12, this protocol requires 2 RTTs to reach consensus on a value and consists of two phases: the first establishes the coordinator/master for an update for specific record issued by a client, and the second tries to reach a consensus value across a majority of acceptors and writes the specified value for a specific record.

(2) *Non-voting* protocols work based on a group communication (GC) primitive in which, unlike 2PC, all nodes receive the certificate vote and then locally perform a commit or abort operation. The GC is mainly categorized into two primitives. *Atomic broadcast* (Défago et al. 2004) propagates messages to *all* DCs and ensures that all DCs agree on the set of received messages and their order. It offers serializability in the case of full replication but hampers scalability, which can be relieved via pairing it with genuine partial replication (GPR). By contrast, *atomic multicast* (Schiper et al. 2009) propagates the message to a subset of DCs, which can be *genuine* or *non-genuine* (Guerraoui and Schiper 2001). Genuine protocols are expensive in terms of delivering message and receiving acknowledgement, while non-genuine ones can deliver messages in 1 RTT and can work well except when the load and the number of DCs are high in the system (Schiper et al. 2009). Moreover, Genuine and non-genuine protocols are identical in *minimality property*, which implies messages are only sent to nodes—servers or DCs—hosting the desired replicas. Note that, atomic multicast implementation using atomic broadcast protocol does not satisfy the minimality property, and thus it is *non-genuine*.

6.1.4 Replication Protocol. Data stores rely on a *replication protocol* to guarantee a serial ordering of write commits of different transactions. They use replication protocol like basic Paxos (Lamport 1998), Viewstamped (Oki and Liskov 1988) (equivalent to Multi-Paxos-based) or atomic broadcast (Défago et al. 2004). These protocols are expensive at (i) throughput as a function

Table 14. Comparison between Different Consensus Replication Protocols

Performance metrics	Basic Paxos†	Fast Paxos	Generalized Paxos	Paxos Batching	Replication Replication	Speculative Paxos	NOPaxs
Latency (message delay)	4	3	2	>4	2|4††	2	2
Message at bottleneck	$2n$	$2n$	$2n$	$2n$	2	2	2
Quorum size	$> n/2$	$> 2n/3$	$> 2n/3$	$> n/2$	$> 2n/3 \| n/2$	$> 3n/4$	$> n/2$

†Multi-Paxos is an optimization for basic Paxos by reserving the master replica for several Paxos instances; thus it can avoid phase 1 to achieve two message delays for agreement on a consensus value.
††Inconsistent Replication (IR) uses the fast path (slow path), which requires 2 (4) message delays with a quorum size of $> 2n/3$ ($> n/2$) *consensus* operations. These operations can execute in any order, while inconsistent operations can execute in different order at each replica and IR can complete them in 2 message delays with a quorum size of $> n/2$.

of the load on a bottleneck replica (e.g., leader replica), and (ii) latency as a function of the number of message delays in the protocol—that is, from when client sends a request to replica until it receives a reply. As shown in Figure 12, the message flow in a leader-based Paxos is: client→leader→replicas →leader→client; thus the latency (i.e., *message delays*) is 4 and throughput (i.e., *bottleneck at message– here master replica is a bottleneck*) is $2n$, where n is the number of replicas. Fast Paxos (Lamport 2006) reduces this latency by sending requests from the client to replicas instead of through a distinguished replica as a leader. It reduces latency to 3 (client→replicas →leader→client) at the cost of requiring larger quorum sizes as compared to that for Paxos. Generalized Paxos (Lamport 2005) is an extension of Fast Paxos and exploits *commutative property* between operations to commit a request in two message delays. EPaxos (Moraru et al. 2013) is another replication protocol that is built on Fast Paxos and Generalized Paxos to achieve low latency and high throughput. Unlike these protocols, Inconsistent Replication (IR) (Zhang et al. 2015) is fault-tolerant without guaranteeing any consistency.

As discussed, the fundamental difference between replication protocols is an extra RTT or a large quorum to order conflicts. Replication protocols recently provide *ordering* in network layer and leave reliability to the replication layer. This results in a fewer message delay and higher throughput. Speculative Paxos (Ports et al. 2015) assumes a best effort ordering at the cost of application-level roll-back, while NOPaxos (Li et al. 2016) guarantee ordering in network layer to avoid roll-back/transaction abort. Table 14 summarizes consensus replication protocols in performance metrics.

6.1.5 Transaction Granularity and Type. One trend in data stores to achieve scalability is to limit the number and location expansion of partitions accessed by a transaction. We call this limitation as *transaction granularity* and classify it as *single shard/partition* and *multi-shard/partition*. The execution and commitment of single shard and multi-shard transactions, respectively, involves a shard within a server (e.g., SQL Azure (Campbell et al. 2010)) and more than one shard replicated across several servers in a data store or several data stores (e.g., Spanner). Most today's data stores support multi-shard, and some of them pose limitations like the pre-declared read/write set performed by transactions in Sinfonia (Aguilera et al. 2007) and Calvin (Thomson et al. 2012).

Transaction type is divided into three categories. *Read-only* (RO) transactions conduct read operations on any updated replicas without locking; meanwhile the incoming writes are not blocked. The original distributed RO transactions (Chan and Gray 1985), as deployed in Spanner, always take 2 RTTs, and before starting they wait until all the involved servers/DCs guarantee that all the transactions have been committed. In contrast, in Eiger (Lloyd et al. 2013) the RO transactions take 1 RTT by maintaining more metadata. *Write-only* (WO) transactions only contain write operations and deploy concurrency controls to avoid write-write conflicts. *Read-write* (RW) transactions consist of reads and writes and avoid write-read and write-write conflicts by using concurrency controls and commitment protocols.

6.1.6 Isolation Levels. This refers to the concurrency between transactions. As the isolation level is weaker, the concurrency level is higher but with more anomalies (Adya 1999; Berenson et al. 1995). In the following, we discuss different isolation levels from strong to weak as shown in Figure 10.

- *Serializability* (SER) guarantees that every concurrent execution of the committed transactions is equivalent to the serial execution of transactions (i.e., one after another). It is typically implemented via 2PL for full replication, which impedes scalability and increases response time especially across data stores. So, SER is provided in the case of partial replication or genuine partial replication (GPR) (e.g., P-store (Schiper et al. 2010)), in which all transaction types might require a certification procedure and go through a synchronization phase. If so, then the transaction aborts (e.g., Sinfonia and P-store); otherwise, the transaction is *wait-free query* (WFQ) as in S-DUR (Sciascia et al. 2012). Note that *strict* SER (SSER, also called *linearizability*) deploys the physical clock to order transactions.
- *Update serializability* (US) (Hansdah and Patnaik 1986) provides guarantees analogous to those in SER for update transactions, but it leads RO transactions to observe non-monotonic snapshots[12] (i.e., two RO transactions may observe different order of the committed update transactions). In fact, US makes the support of observing a snapshot equivalent to some serial execution of the partially ordered history of update transactions. Extended (E) US (Peluso et al. 2012), as a stronger variant of US, holds the same semantic not only for committed update transactions but also for those are executing and may abort later due to either write-write or write-read conflicts. This property guarantees that applications do not act in an unexpected manner because of non-serializable snapshots observation.
- *Snapshot isolation* (SI) never aborts RO transactions and guarantees that a transaction reads the most recent version of data through MVCC (Kung and Robinson 1981). However, it blocks write transactions to avoid write-write conflicts and enhances responsiveness of these transactions at the expense of *write skew* anomaly[13] Berenson et al. (1995) due to ignoring read-write conflicts. It also implements WFQ but not in GPR scheme (Ardekani et al. 2014). Due to its simple implementation, it is supported by database vendors (e.g., Microsoft SQL Server and Oracle) and most Geo-replicated data stores.
- *Parallel snapshot isolation* (PSI) (Sovran et al. 2011) is similar to SI and suitable to deploy across data stores. In PSI, the commit order of non-conflicting transactions[14] can vary among replicas, since the transactions are causally ordered. In fact, a transaction is propagated to other replicas after all transactions committed before it starts. Thus, PSI enforces that transactions observe the local data that may be stale. Unlike SI, PSI neither supports GPR and nor monotonic snapshots of transactions, which is the main hindrance of SI regard to scalability (Ardekani et al. 2013).
- *Non-monotonic snapshot isolation* (NMSI) (Ardekani et al. 2013) works under GPR scheme and preserves WFQ, *minimal commitment synchronization*, and *forward freshness*. NMSI is more scalable and takes any snapshot as compared to PSI, because PSI neither is implementable under GPR and nor is preservable for the forward freshness property. The Jessy

[12]A snapshot is a logical copy of data consisting of all committed updates and is created when a transaction is committed.
[13]The readers are referred to articles by Adya (1999) and Berenson et al. (1995) on details of anomalies.
[14]Both write-write and write-read conflicts are avoided in SER, SI, and US, while only write-write conflicts are prevented in PSI and NMSI.

Table 15. Anomalies in Different Isolation Levels

Anomalies	SSER	SER	US	SI	PSI	NMSI	MAV	RA	RC	RUC
Dirty Read	N[1]	N	N	N	N	N	N	N	N	Y
Repeatable Reads	N	N	N	N	N	N	N	N	Y	Y
Read Skew	N	N	N	N	N	N	N	Y	Y	Y
Dirty Writes	N	N	N	N	N	N	N	N	N	N
Lost Updates	N	N	N	N	N	N	Y	Y	Y	N
Write Skew	N	N	N	Y	Y	Y	Y	Y	Y	Y
Causality violation	N	N	N	N	Y	Y	Y	Y	Y	Y
Non-Mon. Snap.[2]	N	N	Y	N	Y	Y	Y	Y	Y	Y
True Time violation	N	Y[1]	Y	Y	Y	Y	Y	Y	Y	Y

[1] N: disallowed anomaly and Y: allowed anomaly. [2] Non-Monotonic Snapshot (long fork).

protocol (Ardekani et al. 2013) implements NMSI under GPR and uses 2PC and the versioning mechanism for concurrency control.

Compared to the isolation levels discussed previously, several weaker isolation levels are also implemented in data stores. *Read Uncommitted* (RUC) (as in PNUTS, Dynamo, and BigTable) totally orders writes to each object, and *Read Committed* (RC) disallows transactions to access uncommitted or intermediate version of data items. *Monotonic Atomic View* (MAV) (Bailis and Ghodsi 2013) ensures that if a transaction T_j reads a version (v) of an object that transaction T_i wrote, then a later read by T_j returns a value whose version $v' \geq v$. *Read Atomic* (RA) (Bailis et al. 2014) enforces all or none of each transaction's updates are visible to others. RA prevents all anomalies avoided by RC with the help of multi-versioning and a small metadata per write operations. Unlike RUC, RC, and MAV, it also prevents *fractured read* anomaly, which happens if transaction T_i writes variables x_m and y_n, then T_j reads x_m and y_k, where $k < n$. Both MAV and RA are useful for the applications requiring secondary indexes, foreign key, and materialized view maintenance.

There are several ways to mutually compare isolation levels. The **first** is to understand what anomalies they allow (Table 15). For example, SSER, SER, and US prevent *write-skew (short fork)* anomaly due to checking write-write and read-write conflicts. The **second** is that they can be compared based on scalability, which is classified into *highly available transaction* (HAT) and non-HAT (Bailis and Ghodsi 2013). Anomalies like *lost update* and *write skew*, and semantics such as *concurrent update* and *bounds on data recency*—real-time insurance on writes/reads—are impossible to be prevented by HAT systems. Thus, SER, SI, PSI, and RR are not HAT-complaint. In contrast, HAT systems preclude *dirty read and write*, and thus RA, RU, and RC are HAT-complaint. The **third** is they can be analyzed via a programmatic tool like G-DUR (Ardekani et al. 2014) that implements the most discussed isolation levels.

6.2 Coordination Mechanisms in Transactional Data Store

This section discusses different mechanisms that coordinate transactions within and across the state-of-the-art data stores.

6.2.1 Heavy-Weight Protocols-Based Transactions. Transactional data stores usually partition data into *shards* and then replicate each shard across servers for fault-tolerance and availability. To guarantee transactions with strong consistency, they deploy a *distributed transaction protocol*, which implements concurrency control via a write-ahead log (log for short). Each shard is designated to a log, which is replicated across data stores. This log is divided into *log positions*, which are uniquely numbered in ascending order (Agrawal et al. 2013). To guarantee the serial ordering of write commits of different transactions into logs, transactional data stores commonly integrate a costly *replication protocol* with strong consistency like Paxos. In the Paxos deployment, participants compete to access a *log position* in the log, and only one of them is allowed to commit and others are required to abort. Thus, these data stores enforce strong consistency in both *transaction protocol*, which provides a serial ordering of transactions across shards, and *replication protocol*, which ensures a serial ordering of writes/reads across replicas in each shard. This redundancy increases latency (message delays) and degrades throughput especially when heavy-weight protocols like 2PL+2PC and OCC+2PC with Paxos are deployed. To alleviate this issue, the following solutions can be leveraged. The first is to use replication protocols with lower latency and higher throughput as discussed before. The second is to replicate and partition data at a limited number of servers/DCs when the data accessed by multi-shard transactions. This solution mandates workload-aware data partitioning approach rather than the commonly used data partitioning mechanisms like *random (hash-based)*, *round robin*, and *key range*. This approach is classified into *scheme-dependent*, which is coarse-grained and unsuitable for dynamic workload, and *scheme-independent* (Kamal et al. 2016) that is fine-grained and deploys graph partitioning to cater a range of workloads in particular dynamic workload.

6.2.2 Contemporary Techniques. Sole reliance on heavy-weight protocols deteriorates response time and causes *spurious coordination* as a transaction unnecessarily delays or reorders the execution of transactions based on their natural arrival order. To relieve these drawbacks, the following contemporary techniques are used. The first is *dependency-tracking technique* and is used in the *linear transactions protocol* (Escriva et al. 2012). This protocol allows transactions to locally commit on the server. ROCOCO (Mu et al. 2014) uses this technique to reorder interfered transactions rather than aborting or blocking them. Warp (Escriva et al. 2013) uses ROCOCO to improve throughput and latency as compared to 2PL and OCC. The second technique is *transaction decomposition*. Lynx (Zhang et al. 2013b) uses this technique and decomposes each transaction to several chains based on the application semantics and the table scheme by using transaction chopping (Shasha et al. 1995). Homeostasis protocol (Roy et al. 2015) also dynamically extracts application semantics to determine acceptable upper-bound inconsistency of the system. This helps it to locally execute transactions across DCs without communication as long as inconsistency does not violate the correctness of transactions. The third technique is the following set of *logical properties*, which minimizes or exempts coordination across transactions: (i) *Commutativity* guarantees reordering operations, which do not influence the outcome of transactions (Lloyd et al. 2013). *Monotonic programs* ensure deterministic outcome for any order of operations on objects (Alvaro et al. 2011) and support operations such as *selection, projection, and join* through a declarative language like Bloom. (iii) *CRDTs*, as already discussed, guarantee convergent outcomes irrespective of the order of updates applied on each object. (iv) *Invariants* like primary keys, foreign keys, and integrity constraints reduce coordination between transactions including *confluent* operations that can be executed without coordination with others. However, contemporary techniques are error-prone, ad-hoc, and some of them, like monotonic programs and CRDTs, are not well developed. Tables 5 and 6 in appendix compare the state-of-the-art projects in several aspects and indicates which concurrency mechanism they used.

7 DATA MANAGEMENT COST

This section discusses the last element of data management in storage: *data management cost*. This element is influenced by *price*, which is a new and important feature of data stores as compared to traditional distributed systems like cluster and grid computing. Data stores offer a variety of pricing plans for different storage services with a set of performance metrics. Pricing plans offered by data stores are typically divided in two categories (Naldi and Mastroeni 2013): *bundling price* (also called *quantity discount*) and *block rate pricing*. The first is observed in most data stores (e.g., Google Drive) and is recognized as a non-linear pricing, where unit price changes with quantity to follow *fixed cost* and *per-unit charge*. The second category divides the range of consumption into sub-ranges and in each sub-range unit price is constant as observed in Amazon. This category is a special form of the *multi-part tariffs* scheme in which the fixed price is zero. Note that the standard form of multi-part tariffs consists of a fixed cost plus multi-ranges of costs with constant cost in each range. One common form of this scheme is *two-part tariffs* that are utilized in data stores with a fixed fee for a long term (currently 1 or 3 years) plus a per-unit charge. This model is known a *reserved pricing model* (e.g., as offered by Amazon RDS) as opposed to an *on-demand pricing model* in which there is no fixed fee and its per-unit charge is more than that in the reserved pricing model. All pricing plans offered by the well-known cloud providers follow *concavity property* that implies as the more resources the application providers buy the cheaper the unit price is. The unit price for storage, network, and VM, respectively, are often GB/month, GB, and instance per unit time.

A cloud provider offers different services with the same functionality while performance is directly proportional to price. For example, Amazon offers S3 and RRS as online storage services but RRS compromises redundancy for lower cost. Moreover, the price of same resources across cloud providers is different. Thus, given these differences, many cost-based decisions can be made. These decisions will become complicated especially for applications with time-varying workloads and different QoS requirements such as availability, durability, response time, and consistency level. To do so, a joint optimization problem of resources cost and the required QoS should be characterized. Resources cost consists of: (i) *storage cost* calculated based on the duration and size of storage the application provider uses, (ii) *network cost* computed according to the size of data the application provider transfers out (reads) and in (writes) to data stores (typically data transfer into data stores is free), and (iii) *computing cost* calculated according to duration of renting a VM by application providers. In the rest of the section, we discuss the cost optimization of data management based on a single QoS or multi-QoS metrics and cost trade-offs.

7.1 Cost Optimization Based on a Single QoS Metric

Application providers are interested into the selection of data stores based on a single QoS metric so the cost is optimized or does not go beyond the budget constraint. This is referred to as a cost optimization problem based on a single QoS metric and is discussed in the following.

(1) *Cost-based availability and durability optimization.* Availability and durability are measured in the number of nines and achieved by means of usually triplicate replication in data stores. Chang et al. (2012) proposed an algorithm to replicate objects across data stores so users obtain the specified availability subject to budget constraint. Mansouri et al. (2013) proposed two dynamic algorithms to select data stores for replicating non-partitioned and partitioned objects, respectively, with the given availability and budget. In respect to durability, PRCR (Li et al. 2012) uses the duplicate scheme to reduce replication cost while achieving the same durability as in triplicate replication. CIR (Li et al. 2011) also dynamically increases the number of replicas based on the demanding reliability with the aim of saving storage cost.

(2) *Cost-based consistency optimization.* While most of the studies explored consistency-performance trade-off (Section 5 and Section 6), several other studies focused on lowering cost with adaptive consistency model instead of a particular consistency model. The *consistency rationing* approach (Kraska et al. 2009) divides data into three categories with different consistency

levels assigned and dynamically switches between them in run time to reduce the resource and the penalty cost paid for the inconsistency of data, which is measured based on the percentages of incorrect operations due to using lower consistency level. Bismar (Chihoub et al. 2013) defines the consistency level on operations rather than data to reduce the cost of the required resources at run time. It demonstrates the direct proportion between the consistency level and cost. C3 (Fetai and Schuldt 2012) dynamically adjusts the level of consistency for each transaction so the cost of consistency and inconsistency is minimized.

(3) *Cost-based latency optimization.* User-perceived latency is defined as (i) a constant in the unit of RTT, distance, and network hops, or (ii) latency cost that is jointly optimized with monetary cost of other resources. Latency cost allows the latency metric to be changed from a discrete value to continuous one, thereby achieving an accurate QoS in terms of latency constraint and easily making a trade-off between latency and other monetary costs. OLTP (respectively, OLAP) applications satisfy the latency constraint by optimization of data placement (respectively, data and task placement). This placement has an essential effect on optimizing latency cost or satisfying it as a constraint as well as in reducing resources cost.

7.2 Cost Optimization Based on Multi-QoS Metric

Application providers employ Geo-replicated data stores to reduce the cost spent on storage, network, and computing under multi-QoS metrics. In addition, they may incur *data migration cost* as a function of the data size transferring out from data store and its corresponding network cost. Data migration happens due to application requirements, the change of data store parameters (e.g., price), and data access patterns. The last factor is the main trigger for data migration as data transits from *hot-spot* to *cold-spot* status (defined in Section 2) or the location of users changes as studied in "Nomad" (Tran et al. 2011). In Nomad, the changes in users' location are recognized based on simple policies that monitor the location of users when they access an object. Depending on the requirements of the applications, cost elements and QoS metrics are determined and integrated in the classical cost optimization problems as linear/dynamic programming (Cormen et al. 2009), k-center/k-median (Jain and Dubes 1988), ski-rental (Seiden 2000), and so on. In the following, these features and requirements are discussed.

(1) For a file system deployment, a key decision is to store a data item either in cache or storage at an appropriate time while guaranteeing access latency. Puttaswamy et al. (2012) leveraged EBS and S3 to optimize the cost of file system and they abstracted the cost optimization via a ski-rental problem. (2) For data-intensive applications spanning across DCs, the key decision is which data stores should be selected so the incurred cost is optimized while QoS metrics are met. The QoS metrics for each data-intensive application can be different; for example, online social applications suffice causal consistency, while a collaborative document editing web service demands strong consistency. (3) For online social network (OSN), the key factor is replica placement and reads/writes redirection, while "social locality" (i.e., co-locating the user's data and her friends' data) making a reduction in access latency is guaranteed. In OSN, different policies to optimize cost are leveraged: (i) minimizing the number of slave replicas while guaranteeing social locality for each user (Pujol et al. 2010), (ii) maximizing the number of users whose locality can be preserved with a given number of replicas for each user (Tran et al. 2012), (iii) graph partitioning based on the relations between users in OSN (e.g., cosplay (Jiao et al. 2016)), and (iv) selective replication of data across DCs to reduce the cost of reads and writes (Liu et al. 2013). (4) The emergence of content cloud platforms (e.g., Amazon Cloud-Front[15] and Azure CDN[16]) help to build a cost-effective cloud-based

[15]"Amazon CloudFront", https://aws.amazon.com/cloudfront/.
[16]"Azure CDN", https://azure.microsoft.com/en-us/services/cdn/.

content delivery network (CDN). In CDN, the main factors contributing into the cost optimization are replicas placement and reads/writes redirection to appropriate replicas (Chen et al. 2012).

7.3 Cost Trade-Offs

Due to several storage classes with different prices and various characteristics of workloads, application providers are facing with several cost trade-offs as follows.

Storage-computation trade-off. This is important in scientific workloads in which there is a need for the decision on either storing data or recomputing data based on the size and access patterns. Similar decision happens to the privacy preservation context that requires a trade-off between encryption and decryption of data (i.e., computation cost) and storing data (Zhang et al. 2013a). The trade-off can be also seen in video-on-demand service in which video transcoding[17] is a computation-intensive operation, and storing a video with different formats is storage-intensive. Incoming workload on the video and the required performance for users determine whether the video is transcoded on-demand or stored with different formats.

Storage-cache trade-off. Cloud providers offer different tiers of storage with different prices and performance metrics. A tier, like S3, provides low storage cost but charges more for I/O, and another tier, like EBS and Azure drive, provides storage at higher cost but I/O at lower cost (Chiu and Agrawal 2010). Thus, as an example, if a file system frequently issues reads and writes for an object, it is cost-efficient to save the object in EBS as a cache, or in S3 otherwise. This trade-off can be exploited by data-intensive applications in which the generated intermediate/pre-computed data can be stored in caches such as EBS or memory attached to VM instances.

Storage-network trade-off. Due to significant *differences in storage and network costs* across data stores and *time-varying workload* on an object during its lifetime, acquiring the cheapest network and storage resources at the appropriate time of the object lifetime plays a vital role in the cost optimization. Simply placing objects in a data store with either the cheapest network or storage for their whole lifetime can be inefficient. Thus, storage-network trade-off requires a strategy to determine the placement of objects during their lifetime in which the status of objects change from hot-spot to cold-spot and vice versa. This was studied in a dual cloud-based storage architecture (Mansouri and Buyya 2016) as well as in distributed data stores for a limited number of replicas for each object (Mansouri et al. 2017). This trade-off also comes as a matter in the recovery cost in erasure coding context, where *regenerating* and *non-MDS* codes are designed for this purpose.

Reserved-on demand storage trade-off. Amazon RDS and Dynamo data stores offer *on-demand* and *reserved* database (DB) instances and confront the application providers with the fact that how to combine these two types of instances so the cost is minimized. Although this trade-off received attention in the context of computing resources (Wang et al. 2013), it is worthwhile to investigate the trade-off in regard to data-intensive applications, since (i) the workload of these two is different in characteristics, and (ii) the combination of on-demand DB instances and different classes of reserved DB instances with various reservation periods can be more cost-effective. Table 7 in the Appendix summarizes state-of-the-art studies in respect to this section.

8 SUMMARY AND FUTURE DIRECTIONS

This section presents a summary and discusses future directions (see Table 16) of different aspects of data management in cloud-based data stores (data stores for short).

Intra- and Inter-data store services. The deployment of OLTP applications within and across data stores brings benefits and challenges for application providers as summarized in Tables 2–5. To

[17]Video transacoding is the process of converting a compressed digital video format to another.

Table 16. Future Direction for Cloud-Based Data Stores

Data store Aspect	Future Directions
Intra- and Inter-data stores services	• Designing algorithms to guarantee the time of data retrieval from storage nodes within a range of time via (i) data replication in several storage nodes owned by different vendors, and (ii) redundant requests against replicas while considering optimizing the monetary cost. • Replicating data across storage nodes instead of random replication to avoid *correlated* and *independent* failures while considering consistency costs, joining and leaving storage nodes. • Reducing unpredictability of response time in multi-tenant storage services via replicating data in storage nodes and submitting redundant read/write requests against them while considering the status of data, cold- and hot-spot, I/O and CPU load. • Designing auto-scaling mechanisms in which the range of required database instances should be determined based on the workload and the type of instance, not by users who determine currently in commercial data stores. • Designing scheduling algorithms to complete data delivery within deadline and budget by considering the size and price of bandwidth. • Designing a fuzzy and self-learning framework for ensuring data security while considering managerial solutions, acts, legislation, and privacy over data placement within and across data stores.
Data Model	• Analysing the relationship between entities in applications to identify their relationship and then placing the associated data in servers/DCs at the close distance • Guaranteeing SLA in terms of response time in both classes of data store via dynamic allocation of bandwidth to requests, selection of adaptive consistency semantic, adaptive replication of data based on their workload.
Data Dispersion	• Borrowing the parallelism in deferred-update replication and abort-free features in state machine replication to improve the scalability and throughput of transactions. • Designing algorithms to replicate data in full or partial degree based on the number of DCs, globality of transactions, and the size of data. • Defining a monetary cost function including latency as a utility to capture it as a continuous value instead of discrete and the cost of storage services, and then applying this cost function in Quorum-based replication approach. • Constructing codes with small repair bandwidth and a minimal number of nodes participating in the repair process of the failed chunk for cold objects, and constructing codes for hot objects with the help of queuing and coding theory to read and to construct the failed chunk from several data chunks in parallel. • Utilizing fully the characteristics of workload and data stores in performance and price to leverage hybrid scheme of data dispersion.
Data consistency and transaction management	• Extension of Conflicted-free Replicated Data Types (CRDTs) in operation types and make the richer relation between them to exempt the coordination between data replicas. • Analysing the semantic of the application to provide consistency at the language level. • Designing replication protocols with the capability of *ordering* request against replicas at the *network level* with the small quorum size to increase throughput– see Table 14. • Designing adaptive isolation levels for applications based on the required response time and available budgets. As response time is tighter, weaker isolation level is chosen. • Designing adaptive algorithms with the help of graph partitioning techniques and clustering algorithms to reduce distributed transactions, and in turn, improve response time.
Data management cost	• Designing algorithms to make a trade-off between performance criteria like availability and durability and monetary cost including storage, read, write, and potential migration costs. • Designing light-weight algorithms to optimize data management cost consisting of storage, read, write, potential migration cost across different storage classes based on the status of objects, that is, hot- and cold-spot. • Designing online and off-line algorithms to make decisions on the number and type of required database instances as well as when on-demand or reserved database instances are deployed.

better achieve the goals of intra-data store (Table 2), there are some interesting tracks to explore as identified in Table 16. However, storing data within data stores faces application providers with challenges as shown in Table 3. A way to tackle these challenges, especially *data transfer bottleneck* and performance unpredictability, is to deploy *workload-aware data placement* while considering the topology of data stores rather than randomly placing data in storage nodes of commercial data stores. Moreover, such solutions require data delivery within a deadline especially for OLTP applications. In respect to another challenge, *data security*, it would be useful to design a fuzzy framework in which acts, legislation, security, and privacy requirements are considered. A complementary approach to these promising solutions is storing data across data stores owned by different vendors. However, the cross-deployment of data stores raises key challenges (Table 5). The first one is how to ease the movement across data stores owned by different vendors. This requires the design of a common data model and standard APIs for different cloud databases to help application providers. The second is how to handle network congestion. The ideal solutions for this challenge are to complete data transfer within a *budget* and *deadline* especially for OLTP applications that demand high response time. Last, concurrency control is another challenge for which the solutions imply to focus on the following data elements as outlined in Figure 1.

Data model. Relational data model provides ACID properties for applications, whereas it impedes scalability. In contrast, NoSQL data model makes data stores more scalable for data-intensive applications. NewSQL takes benefits of both data models. It also mitigates the latency effects of using remote data, because a hierarchical scheme of the related data is made, and then the data is placed in servers/DCs, which are at the close distance. Thus, it would be very useful to analyze the workload of the application and build a hierarchical scheme of related data. This helps to reduce cross-coordination and confine transactions to a limited number of servers/DCs, and eases processing complex queries over entities. Furthermore, both classes of commercial data stores using these data models support weak SLA for a limited performance criteria like availability and durability. Hence, there is a venue to provide better SLA in respect to response time, auto-scaling, monetary cost (Table 16).

Data dispersion. Though data replication has been widely investigated in the field of databases, it is worthwhile to investigate which model, degree, granularity, and propagation of data replication is deployed for achieving specific performance metrics in the cloud-based data stores (Table 16). In respect to erasure coding, cloud-based data stores require codes that are more efficient in network rather than storage to save monetary cost as deployed across data stores and reduce network congestion as used within data stores. The pure use of each scheme of data dispersion may not be efficient in terms of performance metrics. A better alternative is to deploy a hybrid scheme based on the characteristics of workloads and data stores that were not fully investigated in state-of-the-art projects (Table 13).

Data consistency and transaction management. To provide consistent data and support ACID transactions, a concurrency control mechanism should be deployed. This significantly affects the overall performance of response time, availability, and even monetary cost. Thus, it is worth deploying strategies to reduce or exempt coordination across replicas especially across DCs. For this purpose, the concurrency control mechanisms are performed in different levels. The concurrency control in *I/O* level is a widely studied research topic in OLTP applications. The proposed protocols in this level are either lock-free or lock-based, which make a trade-off between response time and memory consumption. To reduce the need for concurrency in I/O level, it would be relevant (i) to replicate associated data in the servers/DCs that are at a close distance and (ii) order requests against replicas in the network level instead of replication protocol, which has received considerable attention recently. The concurrency mechanisms in the level of *object* and *flow* are appealing, since they exempt/reduce the need of coordination. They are in their infancy and currently support a limited number of data objects (e.g., *counters*, some specific *sets* types) and operation types (e.g., increment and decrements for counters, union and intersect for sets). These objects also lack rich relations and operations among themselves that affect the amount of information maintained and propagated by each replica. The concurrency mechanisms in application and language levels can reduce the need of coordination but they are ad hoc and error-prone.

Data management cost. Monetary cost is a key factor for application providers to move their data into storage infrastructure. The optimization of the monetary cost is a vital criterion for application providers. The contributing factors in this optimization are: the pricing models, the duration of used resources, the characteristics of workload, and the required QoS depending on the types of applications. We discussed this optimization based on either single/multi QoS metrics. The existing studies dealt with the cost optimization problems in which some cost elements and QoS metrics are considered. Thus, this is a venue to study the cost optimization problems, which are complimentary to those already studied. Further work is also needed to investigate when these problems lead to cost trade-offs like storage vs. network, storage vs. cache, and storage vs. computation.

Furthermore, due to different classes of storage (for example, S3 Standard, S3 Standard-Infrequent Access, Amazon Glacier, Amazon Reduced Redundancy Storage (RRS)) differentiating in price and performance, it would be interesting to investigate how to replicate objects during their lifetime based on the writes, reads, and size of objects so the cost is optimized and the required SLA is satisfied. In fact, the objects should be migrated across these classes within or across data stores for this purpose. This mandates to determine the migration time(s) for objects as their statuses change from hot-spot to cold-spot and vice versa.

ACKNOWLEDGMENTS

The authors would like to thank Chenhao Qu, Mara Alejandra Rodrguez, Shashikant Ilager, Rodrigo N. Calheiros, Sareh Fotuhi Piraghaj, and the anonymous reviewers for the insightful comments for improvement.

REFERENCES

Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 2 (Feb. 2012).

Daniel J. Abadi. 2009. Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull.* 32, 1 (2009), 3–12.

Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. 2010. RACS: A case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10).* ACM, New York, NY, 229–240.

Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. Dissertation. Cambridge, MA.

Divyakant Agrawal, Amr El Abbadi, Hatem A. Mahmoud, Faisal Nawab, and Kenneth Salem. 2013. *Managing Geo-replicated Data in Multi-datacenters.* Springer, Berlin, 23–43.

Rakesh Agrawal, Michael J. Carey, and Miron Livny. 1987. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Data. Syst.* 12, 4 (Nov. 1987), 609–654.

Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: A new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev.* 6 (Oct. 2007), 5:1–5:48.

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: Definitions, implementation, and programming. *Distrib. Comput.* 9, 1 (01 Mar 1995), 37–49.

Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10).* USENIX Association, Berkeley, CA, 19–19.

Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2015. *Efficient State-Based CRDTs by Delta-Mutation.* Springer International Publishing, Cham, 62–76.

Sérgio Almeida, João Leitão, and Luís Rodrigues. 2013. ChainReaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13).* ACM, New York, NY, 85–98.

Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M. Hellerstein. 2013. Consistency without borders. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13).* ACM, New York, NY, Article 23, 23:1–23:10.

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. 2014. Blazes: Coordination analysis for distributed programs. In *Proceedings of the 30th IEEE International Conference on Data Engineering.* 52–63.

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency analysis in bloom: A CALM and collected approach. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research.* 249–260.

Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. 2010. What consistency does your key-value store actually provide? In *Proceedings of the 6th International Conference on Hot Topics in System Dependability (HotDep'10).* USENIX Association, Berkeley, CA, 1–16.

Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2013. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems (SRDS'13).* IEEE Computer Society, Washington, DC, 163–172.

Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2014. G-DUR: A middleware for assembling, analyzing, and improving transactional protocols. In *Proceedings of the 15th International Middleware Conference (Middleware'14).* ACM, New York, NY, 13–24.

Masoud Saeida Ardekani, Pierre Sutra, Marc Shapiro, and Nuno M. Preguiça. 2013. On the scalability of snapshot isolation. In *Proceedings Euro-Par 2013 Parallel Processing—19th International Conference.* 369–381.

J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. González de Mendívil, and F. D. Muñoz Escoí. 2008. SIPRe: A partial database replication protocol with SI replicas. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC'08).* ACM, New York, NY, 2181–2185.

Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.* 7, 3 (Nov. 2013), 181–192.

Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems. *Proc. VLDB Endow.* 3 (Nov. 2014), 185–196.

Peter Bailis and Ali Ghodsi. 2013. Eventual consistency today: Limitations, extensions, and beyond. *Queue* 11, 3, Article 20 (March 2013), 20:20–20:32.

Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2012. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.* 5, 8 (April 2012), 776–787.

Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR'11).* 223–234.

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting consistency back into eventual consistency. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15).* ACM, New York, NY, Article 6, 6:1–6:16.

Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM'11).* ACM, New York, NY, 242–253.

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'95).* ACM, New York, NY, 1–10.

David Bermbach, Markus Klems, Stefan Tai, and Michael Menzel. 2011. MetaStorage: A federated cloud storage system to manage consistency-latency tradeoffs. In *Proceedings of IEEE International Conference on Cloud Computing (CLOUD'11).* 452–459.

Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2011. DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11).* ACM, New York, NY, 31–46.

C. E. Bezerra, F. Pedone, and R. V. Renesse. 2014. Scalable state-machine replication. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* 331–342.

Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. 2011. Apache Hadoop goes realtime at facebook. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'11).* ACM, New York, NY, 1071–1080.

Kevin D. Bowers, Ari Juels, and Alina Oprea. 2009. HAIL: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09).* ACM, New York, NY, 187–198.

Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. 2008. Building a database on S3. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08).* ACM, New York, NY, 251–264.

Christian Cachin, Robert Haas, and Marko Vukolic. 2010. *Dependable storage in the Intercloud.* Technical Report. Research Report RZ, 3783.

Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11).* ACM, New York, NY, 143–157.

David G. Campbell, Gopal Kakivaya, and Nigel Ellis. 2010. Extreme scale with full SQL language support in microsoft SQL azure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10).* ACM, New York, NY, 1021–1024.

Rick Cattell. 2011. Scalable SQL and NoSQL data stores. *SIGMOD Record* 39, 4 (May 2011), 12–27.

A. Chan and R. Gray. 1985. Implementing distributed read-only transactions. *IEEE Trans. Softw. Eng.* SE-11, 2 (Feb 1985), 205–212.

Chia-Wei Chang, Pangfeng Liu, and Jan-Jan Wu. 2012. Probability-based cloud storage providers selection algorithms with maximum availability. In *Proceedings of the 41st International Conference on Parallel Processing.* 199–208.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 4:1–4:26.

Fangfei Chen, Katherine Guo, John Lin, and Thomas F. La Porta. 2012. Intra-cloud lightning: Building CDNs in the cloud. In *Proceedings of the IEEE INFOCOM*. 433–441.

H. Chen, H. Jin, and S. Wu. 2016. Minimizing inter-server communications by exploiting self-similarity in online social networks. *IEEE Trans. Parall. Distrib. Syst.* 27, 4 (April 2016), 1116–1130.

Haopeng Chen, Zhenhua Wang, and Yunmeng Ban. 2013. Access-load-aware dynamic data balancing for cloud storage service. In *Proceedings of the 6th International Conference on Internet and Distributed Computing Systems—Volume 8223 (IDCS'13)*. Springer-Verlag, New York, 307–320.

H. C. H. Chen, Y. Hu, P. P. C. Lee, and Y. Tang. 2014. NCCloud: A network-coding-based storage system in a cloud-of-clouds. *IEEE Trans. Comput.* 63, 1 (Jan 2014), 31–44.

Kai Chen, Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, Xitao Wen, and Yan Chen. 2014. OSA: An optical switching architecture for data center networks with unprecedented flexibility. *IEEE/ACM Trans. Netw.* 22, 2 (April 2014), 498–511.

H. E. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Prez. 2013. Consistency in the cloud: When money does matter! In *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. 352–359.

D. Chiu and G. Agrawal. 2010. Evaluating caching and storage options on the Amazon web services cloud. In *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing*. 17–24.

Asaf Cidon, Robert Escriva, Sachin Katti, Mendel Rosenblum, and Emin Gun Sirer. 2015. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*. USENIX Association, Santa Clara, CA, 31–43.

Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. 2013. Copysets: Reducing the frequency of data loss in cloud storage. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*. USENIX, San Jose, CA, 37–48.

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*2 (Aug. 2008), 1277–1288.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 8:1–8:22.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*, 3rd ed. MIT Press.

Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. 2012. Camdoop: Exploiting in-network aggregation for big data applications. In *Presented as Part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*. USENIX, San Jose, CA, 29–42.

Anupam Das, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Curtis Yu. 2013b. Transparent and flexible network management for big data processing in the cloud. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX, Berkeley, CA.

Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013a. ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.* 38, 1, Article 5 (April 2013), 5:1–5:45.

DeCandia. 2007. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220.

Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *Comput. Surveys* 36, 4 (Dec. 2004), 372–421.

Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*. ACM, New York, NY, 1–12.

A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. 2010. Network coding for distributed storage systems. *IEEE Trans. Info. Theory* 56, 9 (Sept 2010), 4539–4551.

Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. 2013. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13)*. ACM, New York, NY, Article 11, 11:1–11:14.

Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014b. Closing the performance gap between causal consistency and eventual consistency. In *Proceedings of the Workshop on the Principles and Practice of Eventual Consistency (PaPEC'14)*.

Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014a. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–13.

Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'12)*. ACM, New York, NY, 25–36.

Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2013. Warp: Multi-key transactions for key value stores. *United Networks, LLC, Technical Report* (2013).

Yuan Feng, Baochun Li, and Bo Li. 2012. Postcard: Minimizing costs on inter-datacenter traffic with store-and-forward. In *Proceedings of the 32nd International Conference on Distributed Computing Systems Workshops (ICDCSW'12)*. IEEE Computer Society, Washington, DC, 43–50.

Ilir Fetai and Heiko Schuldt. 2012. Cost-based data consistency in a data-as-a-service cloud environment. In *Proceedings of the IEEE 5th International Conference on Cloud Computing (CLOUD'12)*. IEEE Computer Society, Washington, DC, 526–533.

Shai Fine, Yoram Singer, and Naftali Tishby. 1998. The hierarchical hidden Markov model: Analysis and applications. *Mach. Learn.* 32, 1 (July 1998), 41–62.

Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. 61–74.

Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (June 2002), 51–59.

Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *ACM Trans. Data. Syst.* 31, 1 (March 2006), 133–160.

Rachid Guerraoui and André Schiper. 2001. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.* 254, 1–2 (March 2001), 297–316.

Jian Guo, Fangming Liu, Xiaomeng Huang, John C. S. Lui, Mi Hu, Qiao Gao, and Hai Jin. 2014. On efficient bandwidth allocation for traffic variability in datacenters. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'14)*. 1572–1580.

R. C. Hansdah and L. M. Patnaik. 1986. *Proceedings of the International Conference on Database Theory (ICDT'86)*. Springer, Berlin, 171–185.

Zach Hill and Marty Humphrey. 2010. CSAL: A cloud storage abstraction layer to enable portable cloud applications. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Scie (CLOUDCOM'10)*. IEEE Computer Society, Washington, DC, 504–511.

Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *Proceedings of the USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, 15–26.

Anil K. Jain and Richard C. Dubes. 1988. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ.

Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a globally deployed software defined wan. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 3–14.

Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. 2013. EyeQ: Practical network performance isolation at the edge. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, Berkeley, CA, 297–312.

Lei Jiao, Jun Li, Tianyin Xu, Wei Du, and Xiaoming Fu. 2016. Optimizing cost for online social networks on geo-distributed clouds. *IEEE/ACM Trans. Netw.* 24, 1 (Feb. 2016), 99–112.

Joarder Kamal, Manzur Murshed, and Rajkumar Buyya. 2016. Workload-aware incremental repartitioning of shared-nothing distributed databases for scalable OLTP applications. *Future Gen. Comput. Syst.* 56, C (March 2016), 421–435.

Osama Khan, Randal C. Burns, James S. Plank, William Pierce, and Cheng Huang. 2012. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*.

Tadeusz Kobus, Maciej Kokocinski, and Pawel T. Wojciechowski. 2013. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS'13)*. IEEE Computer Society, Washington, DC, 286–296.

Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. 2007. SafeStore: A durable and practical storage system. In *Proceedings of the USENIX Annual Technical Conference (ATC'07)*. USENIX Association, Berkeley, CA, Article 10, 10:1–10:14.

Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.* 1 (Aug. 2009), 253–264.

Diego Kreutz, Fernando M. V. Ramos, Paulo Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2015. Software-defined networking: A comprehensive survey. *Pract. Proc. IEEE* 103, 1 (2015), 14–76.

K. Ashwin Kumar, Abdul Quamar, Amol Deshpande, and Samir Khuller. 2014. SWORD: Workload-aware data placement and replica selection for cloud data management systems. *VLDB J.* 23, 6 (2014), 845–870.

H. T. Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Trans. Data. Syst.* 2 (June 1981), 213–226.

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40.

Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169.

Leslie Lamport. 2005. *Generalized Consensus and Paxos.* Technical Report MSR-TR-2005-33. Microsoft Research. 60 pages.

Leslie Lamport. 2006. Fast paxos. *Distrib. Comput.* 19, 2 (2006), 79–103.

Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. 2014. Application-driven bandwidth guarantees in datacenters. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 467–478.

Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14).* USENIX Association, Berkeley, CA, 281–292.

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12).* USENIX Association, Berkeley, CA, 265–278.

Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just say NO to paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16).* USENIX Association, 467–483.

Mingqiang Li, Chuan Qin, and Patrick P. C. Lee. 2015. CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15).* USENIX Association, Santa Clara, CA, 111–124.

Wenhao Li, Yun Yang, Jinjun Chen, and Dong Yuan. 2012. A cost-effective mechanism for cloud data reliability management based on proactive replica checking. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'12).* IEEE Computer Society, Washington, DC, 564–571.

Wenhao Li, Yun Yang, and Dong Yuan. 2011. A novel cost-effective dynamic data replication strategy for reliability in cloud data centres. In *Proceedings of the 9th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC'11).* IEEE Computer Society, Washington, DC, 496–502.

Guanfeng Liang and Ulaş C. Kozat. 2014. Fast cloud: Pushing the envelope on delay performance of cloud storage with coding. *IEEE/ACM Trans. Netw.* 22, 6 (Dec. 2014), 2012–2025.

G. Liang and U. C. Kozat. 2016. On throughput-delay optimal access to storage clouds via load adaptive coding and chunking. *IEEE/ACM Trans. Netw.* 24, 4 (Aug 2016), 2168–2181.

G. Liu, H. Shen, and H. Chandler. 2013. Selective data replication for online social networks with distributed datacenters. In *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP'13).* 1–10.

J. Liu and H. Shen. 2016. A low-cost multi-failure resilient replication scheme for high data availability in cloud storage. In *IEEE 23rd International Conference on High Performance Computing (HiPC'16).* 242–251.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11).* ACM, New York, NY, 401–416.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13).* USENIX Association, Berkeley, CA, 313–328.

D. Lomet. 1996. Replicated indexes for distributed data. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems.* 108–119.

Yadi Ma, Thyaga Nandagopal, Krishna P. N. Puttaswamy, and Suman Banerjee. 2013. An ensemble of replication and erasure codes for cloud file systems. In *Proceedings of the IEEE INFOCOM.* 1276–1284.

Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.* 9 (July 2013), 661–672.

Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2014. MaaT: Effective and scalable coordination of distributed transactions in the cloud. *Proc. VLDB Endow.* 5 (Jan. 2014), 329–340.

Yaser Mansouri and Rajkumar Buyya. 2016. To move or not to move: Cost optimization in a dual cloud-based storage architecture. *J. Netw. Comput. Appl.* 75 (2016), 223–235.

Yaser Mansouri, Adel Nadjaran Toosi, and Rajkumar Buyya. 2013. Brokering algorithms for optimizing the availability and cost of cloud storage services. In *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'13).* 581–589.

Y. Mansouri, A. Nadjaran Toosi, and R. Buyya. 2017. Cost optimization for dynamic replication and migration of data in cloud data centers. *IEEE Trans. Cloud Comput.* (2017). DOI: https://doi.org/10.1109/TCC.2017.2659728

Bo Mao, Suzhen Wu, and Hong Jiang. 2016. Exploiting workload characteristics and service diversity to improve the availability of cloud storage systems. *IEEE Trans. Parall. Distrib. Syst.* 27, 7 (2016), 2010–2021.

John C. McCullough, John Dunagan, Alec Wolman, and Alex C. Snoeren. 2010. Stout: An adaptive interface to scalable cloud storage. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, 4–4.

Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 358–372.

Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting more concurrency from distributed transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 479–494.

Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. 2014. f4: Facebook's warm BLOB storage system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 383–398.

Maurizio Naldi and Loretta Mastroeni. 2013. Cloud storage pricing: A comparison of current practices. In *Proceedings of International Workshop on Hot Topics in Cloud Services (HotTopiCS'13)*. ACM, New York, NY, 27–34.

Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling memcache at facebook. In *Presented as Part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*. USENIX, 385–398.

Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC'88)*. ACM, New York, NY, 8–17.

L. Pacheco, D. Sciascia, and F. Pedone. 2014. Parallel deferred update replication. In *Proceedings of the IEEE 13th International Symposium on Network Computing and Applications*. 205–212.

Fernando Pedone, Matthias Wiesmann, André Schiper, Bettina Kemme, and Gustavo Alonso. 2000. Understanding replication in databases and distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems*. 464–474.

Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luis Rodrigues. 2012. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Proceedings of the 2012 IEEE 32nd International Conference on Distributed Computing Systems (ICDCS'12)*. IEEE Computer Society, Washington, DC, 455–465.

Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. 2013. ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing. In *Proceedings of the ACM SIGCOMM Conference on SIGCOMM (SIGCOMM'13)*. ACM, New York, NY, 351–362.

Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Oakland, CA, 43–57.

Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. 2010. The little engine(s) that could: Scaling online social networks. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM'10)*. ACM, New York, NY, 375–386.

Krishna P. N. Puttaswamy, Thyaga Nandagopal, and Murali Kodialam. 2012. Frugal storage for cloud file systems. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, New York, NY, 71–84.

K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A "Hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. *SIGCOMM Comput. Commun. Rev.*4 (Aug. 2014), 331–342.

K. V. Rashmi, N. B. Shah, and P. V. Kumar. 2011. Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction. *IEEE Trans. Info. Theory* 57, 8 (Aug 2011), 5227–5239.

Ron Roth. 2006. *Introduction to Coding Theory*. Cambridge University Press, New York, NY.

Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. 2015. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM, New York, NY, 1311–1326.

Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX Association, Berkeley, CA, 1–16.

Yasushi Saito and Marc Shapiro. 2005. Optimistic replication. *Comput. Surveys* 37, 1 (March 2005), 42–81.

Sherif Sakr. 2014. Cloud-hosted databases: Technologies, challenges and opportunities. *Cluster Comput.* 17, 2 (June 2014), 487–502.

S. Sakr, A. Liu, D. M. Batista, and M. Alomari. 2011. A survey of large scale data management approaches in cloud environments. *IEEE Commun. Surveys Tutor.* 13, 3 (2011), 311–336.

Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. XORing elephants: Novel erasure codes for big data. *Proc. VLDB Endow.* 5 (March 2013), 325–336.

N. Schiper, P. Sutra, and F. Pedone. 2009. Genuine versus non-genuine atomic multicast protocols for wide area networks: An empirical study. In *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems.* 166–175.

Nicolas Schiper, Pierre Sutra, and Fernando Pedone. 2010. P-store: Genuine partial replication in wide area networks. In *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems (SRDS'10).* IEEE Computer Society, Washington, DC, 214–224.

Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. 2012. Scalable deferred update replication. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN'12).* IEEE Computer Society, Washington, DC, 1–12.

Steven S. Seiden. 2000. A guessing game and randomized online algorithms. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC'00).* ACM, New York, NY, 592–601.

Nihar B. Shah, Kangwook Lee, and Kannan Ramchandran. 2014. The MDS queue: Analysing the latency performance of erasure codes. In *Proceedings of the IEEE International Symposium on Information Theory.* 861–865.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS'11).* Springer-Verlag, Berlin, Heidelberg, 386–400.

Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. 2015. Take me to your leader! Online optimization of distributed storage configurations. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1490–1501.

Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.* 3 (Sept. 1995), 325–363.

M. Shen, A. D. Kshemkalyani, and T. Y. Hsu. 2015. Causal consistency for geo-replicated cloud storage under partial replication. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW'15).* 509–518.

Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the data center network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11).* USENIX Association, Berkeley, CA, 309–322.

Youngjoo Shin, Dongyoung Koo, and Junbeom Hur. 2017. A survey of secure data deduplication schemes for cloud storage systems. *Comput. Surveys* 49, 4, Article 74 (Jan. 2017), 441–446.

David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12).* USENIX Association, Berkeley, CA, 349–362.

Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. 2013. F1: A distributed SQL database that scales. *Proc. VLDB Endow.* 11 (Aug. 2013), 1068–1079.

K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. *SIGPLAN Not.* 50, 6 (June 2015), 413–424.

Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11).* ACM, New York, NY, 385–400.

J. Spillner, G. Bombach, S. Matthischke, J. Muller, R. Tzschichholz, and A. Schill. 2011. Information dispersion over redundant arrays of optimal cloud storage for desktop users. In *Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing (UCC'11).* 1–8.

Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. 2013. Zoolander: Efficiently meeting very strict, low-latency SLOs. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC'13).* USENIX, San Jose, CA, 265–277.

C. Suh and K. Ramchandran. 2011. Exact-repair MDS code construction using interference alignment. *IEEE Transactions on Information Theory* 57, 3 (March 2011), 1425–1442.

Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15).* USENIX Association, Berkeley, CA, 513–527.

Andrew S. Tanenbaum and Maarten van Steen. 2006. *Distributed Systems: Principles and Paradigms (2nd ed.).* Prentice-Hall, Inc.

J. Tang, X. Tang, and J. Yuan. 2015. Optimizing inter-server communication for online social networks. In *Proceedings of the IEEE 35th International Conference on Distributed Computing Systems*. 215–224.

Jeff Terrace and Michael J. Freedman. 2009. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIX'09)*. USENIX Association, Berkeley, CA, 11–11.

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based service level agreements for cloud storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 309–324.

Robert H. Thomas. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.* 2 (June 1979), 180–209.

Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. ACM, New York, NY, 1–12.

A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. 2010. Hive - A petabyte scale data warehouse using hadoop. In *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE'10)*. 996–1005.

Duc A. Tran, Khanh Nguyen, and Cuong Pham. 2012. S-CLONE: Socially-aware data replication for social networks. *Comput. Netw.* 56, 7 (2012), 2001–2013.

Nguyen Tran, Marcos K. Aguilera, and Mahesh Balakrishnan. 2011. Online migration for geo-distributed storage systems. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. USENIX Association, Berkeley, CA, 15–15.

Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. 2012. Deadline-aware datacenter TCP (D2TCP). In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'12)*. ACM, New York, NY, 115–126.

Robbert van Renesse and Fred B. Schneider. 2004. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation—Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, 7.

Srikumar Venugopal, Rajkumar Buyya, and Kotagiri Ramamohanarao. 2006. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Comput. Surv.* 38, 1, Article 3 (June 2006), 1–53.

Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. 2012. LogBase: A scalable log-structured database system in the cloud. *Proc. VLDB Endow.* 5, 10 (June 2012), 1004–1015.

Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. 2011. Data consistency properties and the trade-offs in commercial cloud storages: The consumers' perspective. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR'11)*. 134–143.

Ting Wang, Zhiyang Su, Yu Xia, and M. Hamdi. 2014. Rethinking the data center networking: Architecture, network protocols, and resource sharing. *IEEE Access* (2014).

Wei Wang, Baochun Li, and Ben Liang. 2013. To reserve or not to reserve: Optimal online multi-instance acquisition in IaaS clouds. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC'13)*. 13–22.

Yunnan Wu, Alexandros G. Dimakis, and Kannan Ramchandran. 2007. Deterministic regenerating codes for distributed storage. In *Proceedings of the Allerton Conference on Control, Computing, and Communication*. 1–5.

Yu Wu, Chuan Wu, Bo Li, Linquan Zhang, Zongpeng Li, and Francis C. M. Lau. 2015. Scaling social media applications into geo-distributed clouds. *IEEE/ACM Trans. Netw.* 23, 3 (June 2015), 689–702.

Yu Wu, Zhizhong Zhang, Chuan Wu, Chuanxiong Guo, Zongpeng Li, and Francis C. M. Lau. 2017. Orchestrating bulk data transfers across geo-distributed datacenters. *IEEE Trans. Cloud Comput.* 5, 1 (2017), 112–125.

Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. 2013. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 292–308.

Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. 2015. A tale of two erasure codes in HDFS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, 213–226.

Yu Xiang, Tian Lan, Vaneet Aggarwal, and Yih Farn R. Chen. 2014. Joint latency and cost optimization for erasurecoded data center storage. *SIGMETRICS Perform. Eval. Rev.* 2 (Sept. 2014), 3–14.

Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Kompella. 2012. The only constant is change: Incorporating time-varying network reservations in data centers. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'12)*. ACM, New York, NY, 199–210.

Boyang Yu and Jianping Pan. 2015. Location-aware associated data placement for geo-distributed data-intensive applications. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'15)*. 603–611.

Haifeng Yu and Amin Vahdat. 2002. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.* 20, 3 (Aug. 2002), 239–282.

Wenying Zeng, Yuelong Zhao, Kairi Ou, and Wei Song. 2009. Research on cloud storage architecture and key technologies. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human (ICIS'09).* ACM, New York, NY, 1044–1048.

Hong Zhang, Kai Chen, Wei Bai, Dongsu Han, Chen Tian, Hao Wang, Haibing Guan, and Ming Zhang. 2015. Guaranteeing deadlines for inter-datacenter transfers. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15).* ACM, New York, NY, Article 20, 14 pages.

Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15).* ACM, New York, NY, 263–278.

Q. Zhang, S. Li, Z. Li, Y. Xing, Z. Yang, and Y. Dai. 2015. CHARM: A cost-efficient multi-cloud data hosting scheme with high availability. *IEEE Trans. Cloud Comput.* 3, 3 (July 2015), 372–386.

X. Zhang, C. Liu, S. Nepal, S. Pandey, and J. Chen. 2013a. A privacy leakage upper bound constraint-based approach for cost-effective privacy preserving of intermediate data sets in cloud. *IEEE Trans. Parall. Distrib. Syst.* 24, 6 (June 2013), 1192–1202.

Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. 2013b. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of ACM SIGOPS 24th Symposium on Operating Systems Principles (SOSP'13).* 276–291.

Zhe Zhang, Amey Deshpande, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. 2010. Does erasure coding have a role to play in my data center. *Microsoft Res. MSR-TR-2010*-52 (2010).

L. Zhao, S. Sakr, and A. Liu. 2015. A framework for consumer-centric SLA management of cloud-hosted databases. *IEEE Trans. Services Comput.* 8, 4 (July 2015), 534–549.

Jingya Zhou, Jianxi Fan, Jin Wang, Baolei Cheng, and Juncheng Jia. 2017. Towards traffic minimization for data placement in online social networks. *Concurr. Comput.: Practice Exp.* 29, 6 (2017). DOI: 10.1002/cpe.3869.

Received October 2016; revised August 2017; accepted August 2017