Open Access

Open Journal of Databases (OJDB)
Volume 1, Issue 1, 2014

www.ronpub.com/journals/ojdb
ISSN 2199-3459

Eventual Consistent Databases: State of the Art

Mawahib Musa Elbushra [A], Jan Lindström [B]

[A] College of Graduate Studies, Sudan University of Science &Technology, Steen Street block 85 house no. 207, 111 Khartoum, Sudan, mawahib.elbushra@hotmail.com
[B] SkySQL – The MariaDB Company, Tekniikantie 12, FIN-02150 Espoo, Finland, jan.lindstrom@skysql.com

ABSTRACT

One of the challenges of cloud programming is to achieve the right balance between the availability and consistency in a distributed database. Cloud computing environments, particularly cloud databases, are rapidly increasing in importance, acceptance and usage in major applications, which need the partition-tolerance and availability for scalability purposes, but sacrifice the consistency side (CAP theorem). In these environments, the data accessed by users is stored in a highly available storage system, thus the use of paradigms such as eventual consistency became more widespread. In this paper, we review the state-of-the-art database systems using eventual consistency from both industry and research. Based on this review, we discuss the advantages and disadvantages of eventual consistency, and identify the future research challenges on the databases using eventual consistency.

TYPE OF PAPER AND KEYWORDS

Research review: *eventual consistency, consistency models, cloud databases, distributed databases*

1 INTRODUCTION

Cloud computing and big data have become increasingly popular and are changing our way of thinking about the world by providing new insights and creating new forms of value. The research of cloud data management is to address the challenges in managing large collections of data in the cloud computing environment, and in identifying information of value to business, science, government, and society. The huge volume of data in cloud computing environments poses major challenges, including data storage at Petabyte scale, massively parallel query execution, facilities for analytical processing, online query processing, resource optimization, data privacy and security.

Consistency is an important area of study in distributed systems. A consistency model in distributed systems is a guarantee about the relation between an update to an object and the access to an updated object. In this paper, our focus will be on the eventual consistency model, which is particularly important in the RDBMS and "NoSQL" worlds.

The literature of distributed systems defines several popular consistency models. They include: linearizability [33]; serializability [10, 30, 47] that ensures a global ordering of transactions; sequential consistency [50] that ensures a global ordering of operations [34]; causal consistency [3, 36] that ensures partial orderings between dependent operations; eventually consistent transactions [41, 49, 50] that ensure that different orders of updates in all copies eventually converge to the same value, and session consistency [44].

Eventual consistency is a consistency model, which is used in many large distributed databases. Such databases require that all changes to a replicated piece of data eventually reach all affected replicas. Furthermore, the conflict resolution is not handled in these databases, and the responsibility is pushed up to the application authors in the event of conflicting updates.

Eventual consistency is a specific form of weak consistency: the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value [49]. If no failures occur, the maximum size of the inconsistency window can be determined based on the factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme.

A few examples of eventually consistent systems are:

- DNS
- Asynchronous master/slave replication on an RDBMS, e.g. MariaDB (www.mariadb.org)
- Memcached in front of MariaDB, which caches reads

The most popular system that implements eventual consistency is DNS (Domain Name System). Updates to a domain name are distributed according to a configured pattern and time-controlled caches. Eventually, all clients will see the same state. Eventual consistency means that given enough time, over which no changes are performed, all updates will propagate through the system and all replicas will be synchronized. At any given point of time, there is no guarantee that the data accessed is consistent, thus the conflicts have to be resolved.

In this paper, we make the following major contributions:

1. We present the history of eventual consistency and define rigorously eventual consistency.
2. Based on the literature, we review the databases using eventual consistency both from research and industry. To the authors' knowledge, this is the first paper presenting a review on the databases using eventual consistency.
3. We evaluate the database systems reviewed. To the authors' knowledge, this is the first paper trying to evaluate the database systems using eventual consistency.
4. Based on the research work above, we discuss the advantages and disadvantages of eventual consistency.
5. Finally, we identify future research issues on eventual consistency.

The rest of this paper is organized as follows. In Section 2, we present the history of eventual consistency and some related systems using eventual consistency. Based on the literature we review databases using eventual consistency in Section 3. Section 4 evaluates the reviewed systems, and Section 5 identifies the advantages and disadvantages of eventual consistency. We present future research issues in Section 6, and conclusions of this paper are given in section 7.

2 HISTORY OF EVENTUAL CONSISTENCY

Eventual consistency states that in an updatable replicated database, eventually all copies of each data item converge to the same value. The origin of eventual consistency can be traced back to Thomas' majority consensus algorithm [46]. The term was coined by Terry et al. [44] and later on popularized by Amazon in their Dynamo system, which supported only eventual consistency [26, 27, 43].

The CAP theorem, also called as Brewer's theorem by its author Dr. Erik A. Brewer, was introduced at PODC 2000 [14, 15]. The theorem was formally proven by Gilbert and Lynch [29]. Brewer introduced consistency, availability and partition tolerance as three desired properties of any shared-data system, and made the conjuncture that maximally two of them can be guaranteed in one time [16, 17].

In general, this theorem perfectly matches the needs of today's internet systems. Ideally, we expect a service to be available during the whole time period of network connection. Therefore, if a network connection is available, the service should be available as well [24, 45, 48, 51]. If the number of servers is increased, the probability of server failure or of network failure is also increased. A system hence needs to take this into account and be designed in such a way that these failures are transparent for the client and the impact of such failure is minimized.

The abbreviation of the CAP theorem comes from the following three properties:

- *Consistency*: This property requires that each operation executed in a distributed system, where data is spread among many servers, ends with the same result as if executed on one server with all data.
- *Availability*: This property requires that in a distributed system sending a request to any functional node should be enough for a requester to get the response. By complying with this

property, a system is tolerant to failure of any nodes, which are caused, for instance, by network throughput issues.

- *Partition Tolerance*: A distributed system consists of many servers interconnected by a network. A frequent requirement is distributing the system across more data centers to eliminate the failure of one of them. During network communication, failures are frequent. Hence, a system needs to be fail-proof against an arbitrary number of failed messages among servers. Temporary communication interruption among a server set must not cause the whole system to respond incorrectly [29].

Next we define eventual consistency informally.

DEFINITION 1: *Eventual consistency*.

- *Eventual delivery*: An update executed at one node evenly executes at all nodes.
- *Termination*: All update executions terminate.
- *Convergence*: Nodes that have executed the same updates eventually reach an equivalent state (and stay).

EXAMPLE 1: Consider a case where data item *R=0* on all three nodes. Assume that we have the following sequence of writes and commits: *W(R=3) C W(R=5) C W(R=7) C* in node 0. Now read on node 1 could return *R=3* and read from node 2 could return *R=5*. This is eventually consistent as long as eventually read from all nodes return the same value. Note that this final value could be *R=3*. Eventual consistency does not restrict the order in which the writes must be executed.

To understand eventual consistency deeper, we establish some precise terminology and we do this similarly as in [19]. For uniformness, we require that all operations are part of a transaction and thus all operations are inside the transactions. We can describe the interaction between transactions and the database by the following three types of operations (query-update interface):

- Updates $u \in U$ issued by the transactions
- Pairs (q, v) representing a query $q \in Q$ issued by the transaction together with a response $v \in V$ by the database system.
- The end of transaction operations issued by the transactions.

Formally, we can represent the activity as a stream of operations, which form a history.

DEFINITION 2: A *history H* for a set of transactions T and a query-update interface (Q, V, U) is a map H, which maps each transaction $t \in T$ and a client to a finite or infinite sequence H(t) operation from alphabet $\sum = U \cup (Q \times V) \cup \{end\}$.

Furthermore, we need to define a program order, i.e., the order in which operations are executed on a transaction.

DEFINITION 3: *Program order*. For a given history H, we define a partial order $\prec_p$ over events in H such that $e \prec_p e'$ iff e appear before e' in some sequence H (t).

Then we need to define an equivalence relation.

DEFINITION 4: *Factoring*: We define an equivalence relation $\sim_t$ over events such that $e \sim_t e'$ iff transaction (e) = transaction (e'). For any partial order $\prec$ over events, we say that $\prec$ factors over $\sim_t$ iff for any events x and y from different transactions $x \prec y$ implies $x' \prec y'$ for any x, y such that $x \sim_t x'$ and $y \sim_t y'$. This induces a corresponding partial order on the transactions.

With the following formalization, we can specify the information about relationships between events declaratively, without referring to implementation-level concepts, such as replicas or messages.

Eventual consistency relaxes other consistency models by allowing queries in a transaction *t* to see only a subset of all transactions that are globally ordered before *t*. It does so by distinguishing between a visibility order (a partial order that defines what updates are visible to a query), and an arbitration order (a partial order that determines the relative order of updates).

DEFINITION 5: A history H is *eventually consistent* if there exist two partial orders $\prec_v$ (the visibility order) and $\prec_a$ (the arbitration order) over events in H, such that the following conditions are satisfied for all events $e_1, e_2, e \subset E_H$:

1. *Arbitration extends visibility*: if $e_1 \prec_v e_2$ then $e_1 \prec_a e_2$.
2. *Total order on past events*: if $e1 \prec_v e$ and $e_2 \prec_v e$, then either $e_1 \prec_a e_2$ or $e_2 \prec_a e_1$.
3. *Compatible with program order*: if $e_1 \prec_p e_2$ then $e_1 \prec_v e_2$
4. *Consistent query results*: for all $(q, v) \in E_H$, $v = q^{\#}$ (apply ($\{e \in H) \| e \prec_v q\}$, $\prec_a$, s_0)). Thus the query returns the state as it results from applying all preceding visible updates (as determined by the

visibility order) to the initial state, in the order given by the arbitration order.

5. *Atomicity*: Both $\prec_v$ and $\prec_a$ factor over $\sim_t$.
6. *Isolation*: If $e_1 \notin$committed (E_H) and $e_1 \prec_v e_2$, then $e_1 \prec_p e_2$. That is, events in uncommitted transactions are visible only to later events by the same client.
7. *Eventual delivery*: For all committed transactions t, there exist only finitely many transactions $t' \in T_H$ such that $t \not\prec_v t'$.

The reason why eventual consistency can tolerate temporary network partitions is that the arbitration order can be constructed incrementally, i.e. it may remain only partially determined for some time after a transaction commits. This allows conflicting updates to be committed even in the presence of network partitions.

Some database solutions support the Availability and Partition tolerance of Brewer's CAP theorem. These database solutions do not support consistency in the same way as the relational database systems do, but they support eventual consistency where data is replicated to the remaining nodes at any given time, as Cassandra does. These systems, along with the others, mainly focus on achieving as low latency as possible by combined with as high performance as possible [35, 45, 52].

There are other database solutions that focus on supporting Consistency and Partition tolerance, and partially supporting Availability. Their partition tolerance may often be obtained by mirroring database clusters between different data centers. The main advantage is the possibility to achieve quicker response by splitting the workload into different sub-tasks, and these sub-tasks are then executed simultaneously across all available nodes/servers [32, 40].

The consistency level may be important for some systems like a stock market. The stock prices and number of stocks available will always have to be up to date. It is the same principle for an e-commerce website - it would not be good for the business if the customer finds out that the product is out of stock only after he or she submitted the payment.

Eventual consistency means that *writes* to one replica will eventually appear at other replicas, and if all replicas have received the same set of *writes*, they will have the same values for all data. This weak form of consistency does not restrict the ordering of operations on different keys in any way, thus forcing programmers to reason about all possible orderings and exposing many inconsistencies to users. For example, under eventual consistency, after Alice updates her profile, she might not see that update after a refresh. Or, if Alice and Bob are commenting back-and-forth on a blog post, Carol might see a random non-contiguous subset of that conversation.

Burckhardt et al. [19, 20, 21] proposed a novel consistency model based on eventually consistent transactions, which are ordered by two order relations (visibility and arbitration) rather than a single order relation. The consistency model establishes a handful of simple operational rules for managing replicas, versions and updates, based on graphs called revision diagrams. These authors have also proved a theorem, which states that the revision diagram rules are sufficient to guarantee eventual consistency.

Bailis et al. [8] stated that dozens of architects support eventual consistency, and this can be taken as a reference of that the eventual consistency had done a "good enough job". An application designer needs to know how database consistency is obtained and what the costs of each inconsistency or anomalies are, in order to decide if she/he needs to implement the eventual consistency with high availability in the application. Dealing with abnormalities, consistency is intuitive and depends on thinking in the correct sequence, and is therefore more difficult than high consistency.

In [3] Abdallah et al. proposed a new atomic commitment protocol that contains single-phase and is non-blocking. However, this method requires that all participants are ruled by a rigorous concurrency control. Therefore, while sites are autonomous on decision, it assumes exactly the same method on all sites. Furthermore, rigorous concurrency control, where transaction does not release any locks until it commits or aborts, decreases the concurrency.

In [13] Bermbach and Tai proposed a novel approach to benchmark staleness in distributed data stores. It was implemented in Amazon S3. The approach has one writer periodically writing a local timestamp plus a version number to the storage system, which considers the difference between the timestamp versions. This achieved satisfactory results. The work provides a criterion for the application developer to determine if consistency in the data store eventually provides guarantees of acceptable consistency. However, they found that S3 frequently violates monotonic read consistency.

In [25] Cooper et al. described *PNUTS*, a massively parallel and geographically distributed database system for Yahoo!'s web applications. PNUTS provides data storage organized as hashed or ordered tables, low latency for large numbers of concurrent requests including updates and queries, and novel per-record consistency guarantees. The consistency model is a per-record timeline consistency, i.e. all replicas of a given record apply all updates to the record in the same order. This provides a consistency model that is between the two extremes of serialized transactions and eventual consistency.

In [38] Merrel et.al. presented *Bitbox*, which is an application that synchronizes distributed repositories of data. It can be used as a backup or sharing application, similarly to popular cloud-based storage systems. Bitbox supports arbitrary and changing topologies, thus allowing residential gateways to be used as caches for synchronizing nomadic devices that connect only periodically. The Bitbox synchronization scheme achieves strong eventual consistency.

In [6] Anderson et.al. presented *Pahoehoe* that is designed to support extreme availability, and offers a key-value-based get-put interface. Pahoehoe is composed of three main entities: proxies, key lookup servers (KLS), and fragment servers (FS). On a *put*, the proxy splits the value into multiple erasure-coded fragments. The FSs are responsible for storing the fragments, which form the bulk of the data. The KLSs maintain a mapping of the user-provided keys to the locations of corresponding fragments. In a typical setup, each data center has a few KLSs for availability and many FSs for reliability and scaling capacity.

Currently, Pahoehoe only guarantees the eventual consistency and can tolerate temporary inconsistency, because the availability is paramount for our initial applications. Its protocols are eager in that they provide a useful result as soon as possible, thus offering a highest availability. For example, a *put* returns success as soon as it has updated any one of the KLSs and a minimum number of FSs, thus ensuring that the value is durable. The remainder of the *put* completes in the background. A *get* will try the list of values referenced by the first responding KLS, from newest to oldest, and will return as soon as it succeeds. If none of the referenced values is available, the *get* tries contacting other KLSs. Thus, *puts* can return success before they are complete and repeated *gets* may sometimes return earlier versions after newer ones.

Pahoehoe is a partition-tolerant storage system, where key-value pairs can be kept in a redundant manner. The novelty with this system is that the redundancy is achieved using erasure-coding rather than normal replication. Eventual consistency is achieved by regularly trying to spread data items that do not have a satisfactory level of redundancy. Conflicts will not occur in the system since there are no integrity constraints, and concurrent *put* operations for the same key are ordered according to timestamps. However, Pahoehoe is not really a database system based on the authors' categorization.

3 DATABASES USING EVENTUAL CONSISTENCY

In this section, we review the databases using eventual consistency. To the authors' knowledge this review contains all currently available and published database systems supporting eventual consistency.

3.1 MongoDB

MongoDB [1, 37] is a document-oriented NoSQL DBMS written in C++ and developed by 10gen. The word mongo in its name comes from the word humongous [1]. MongoDB focuses on ease of use, performance and high scalability. MongoDB is available for Windows and Unix-like environments.

MongoDB uses a binary form of JSON called Binary JSON, or BSON, to store data. BSON is designed to be easily and efficiently traversed and parsed. Users use regular JSON, which is then transformed into the BSON format. When data is retrieved, it is again transformed into regular JSON. A JSON document is zero or more key-value pairs, and a MongoDB document is simply a JSON document. Since MongoDB uses JSON, it is schema-less. This means that there is no grouping of documents, which has exactly the same keys, like in the relational model. Instead, similar documents with different key-value pairs are stored together in collections. A database, in its turn, can be seen as a collection of collections.

MongoDB supports indexing on any attribute of a document, similar to how RDBMS offer indexing on any column. Indexes are implemented using B-Trees [3]. MongoDB indexes are created from JavaScript shell by using the *ensureIndex()* function. Indexes can be created on simple keys, embedded keys and entire documents. MongoDB uses JSON as its query languages. A JSON query is a JSON document, which describes what is to be searched for.

In MongoDB, *replica sets* are used as the replication strategy, instead of the conventional *master-slave* replication. Replica Sets improves master-slave replication with failover capabilities. A replica set is a cluster of MongoDB nodes, and consists of a *primary* node and multiple *secondary* nodes. The primary *node* is responsible for answering queries, and secondary *nodes* periodically update their data by reading logs from the primary node.

If a primary node is down, one of the secondary nodes is chosen as new primary. The secondary node calls for an election among secondary nodes, when it cannot reach the primary node. Nodes in the system are classified by a priority scheme that ranges from 1 (high) to 0 (low). The priority setting affects elections, and nodes will prefer to vote for the nodes with the highest priority value. If the old primary comes back to life, it will act as a secondary node and update its data according to the new primary's log.

Replicates can be used for scaling out reads and writes. In read scale out, secondary nodes will respond to requests for reading data. Because replication is asynchronous, and there is always a time interval between a write request reaching the primary node and the read request reaching a secondary node, data can be inconsistent. When scaling out writes, secondary nodes will accept conflicting operations without negotiating with the primary node. In this case, data replicated from the primary node will always take preference over the locally written data. Therefore, updates to secondary nodes might be unused due to replication.

From the point of view of client applications, whether a MongoDB instance is running as a single server or a replica set is transparent, read operations to a replica set by default return results from the primary, and are consistent with the last write operation. Applications may configure the read preference based on a per-connection basis, and prefer that the read operations return the replicas on the secondary node. When reading from a secondary, a query may return data that reflects a previous state. This feature is sometimes characterized as the eventual consistency because the secondary member's state will eventually reflect the primary's state. MongoDB cannot guarantee strict consistency for read operations from secondary members. To guarantee the consistency for reads from secondary members, one can configure the client and driver to ensure that write operations succeed on all members before reads complete successfully.

MongoDB uses a readers-writer lock, which allows concurrent read access to a database but exclusive write access to a single write operation. Before the version 2.2 of MongoDB, this lock was implemented on a per-MongoDB basis. Since the version 2.2, the lock is implemented at the database level. One approach to increasing concurrency is to use sharding. In some situations, reads and writes will yield their locks. If MongoDB predicts that a page is unlikely to be in memory, operations will yield their lock while the pages load. The use of lock is expanded greatly in 2.2.

MongoDB offers the following C-A tradeoff options:

- For writes:
  - Write to a master, which may be the only master for the shard, is scalable.
- For reads:
  - Read from the master guarantees consistency at the cost of performance.
  - Read from a slave may return old data but with higher performance.

3.2 CouchDB

CouchDB [5] is also a document-oriented NoSQL database management system, developed and maintained by the Apache Software Foundation. CouchDB is written in the functional programming language Erlang. The name CouchDB is derived from its developers' idea of it being easy to use. At CouchDB server startup, the phrase "It's time to relax" is outputted on the console. What makes CouchDB unique is its RESTful API, which supports the database access over HTTP.

CouchDB stores JSON documents in a binary format, like MongoDB. CouchDB stores documents directly to its databases, and its database files have an extension .couch. Each document has a unique ID, which can be assigned manually when inserting documents, or automatically by CouchDB. There is no maximum number of key-value pairs for documents and there is no maximum size; the default max size is 4 GB, but this can be changed by editing the CouchDB configuration file.

CouchDB is normally queried by direct identifier lookups, or by creating MapReduce "views", which CouchDB runs to create an index for querying or computing other attributes. In addition, the ChangesAPI of CouchDB shows documents in the order they were last modified. CouchDB replicates the document versions between nodes, thus making the CouchDB databses an eventually consistent system. Because of the CouchDB append-only value mutation, individual instances will not lock. When distributed, CouchDB will not allow updating the same document without a preceding version number, and conflicts must be manually resolved before concluding a write.

CouchDB uses a B-tree storage engine for all internal data, documents, and views. In CouchDB, MapReduce is used to compute the results of a view. MapReduce makes use of two functions, "map" and "reduce," which are applied to each document in isolation. The two functions produce key/value pairs, and CouchDB insert them into the B-tree storage engine. Documents and results in CouchDB are accessed and viewed by key or key range. CouchDB uses Multi-Version Concurrency Control (MVCC) to provide concurrent access to the database. CouchDB documents are versioned. Changing a document means that CouchDB creates an entirely new version of that document and saves it over the old one. After doing this, CouchDB ends up with two versions of the same document, one old and one new.

Let us consider a set of requests wanting to access a document. The first request reads the document. While this is being processed, a second request changes the document. Since the second request includes a completely new version of the document, CouchDB can simply append it to the database without having to wait for the read request to finish. When a third request wants to read the same document, CouchDB will point it to the new version that has just been written. During this whole process, the first request could still be reading the original version.

Maintaining consistency inside a single database node is quite easy. On the contrary, maintaining consistency between multiple database servers is difficult. If a client performs a write operation on server A, how do we make sure that this is consistent with server B, or C, or D? For relational databases, this is a very complex problem, and whole books are needed for discussing its solutions. One could use multi-master, master/slave, partitioning, sharding, write-through caches, and all sorts of other complex methods for achieving consistency between multiple database servers.

The operations of CouchDB take place within the context of a single document. CouchDB achieves eventual consistency between multiple databases by using incremental replication. Incremental replication is a process where document changes are periodically copied among servers. Considering a case where the same document is changed in two different databases and this change is synchronized with each other. For this situation, CouchDB's replication system offers automatic conflict detection and resolution. When CouchDB detects that a document has been changed in both databases, it flags this document as being in conflict.

When two versions of a document conflict during replication, the winning version is saved as the most recent version in the document's history. The losing version is not deleted. Instead, CouchDB saves this as a previous version in the document's history, so that it can be accessed. This happens automatically and consistently, and both databases will make exactly the same choice. It is up to the application to handle conflicts in a way that makes sense for your application. You can leave the chosen document versions in place, revert to the older version, or try to merge the two versions and save the result.

3.3 Amazon SimpleDB

Amazon [4] is a public cloud computing provider, and offers services (AWS) based on the IaaS approach. Amazon AWS (Amazon Web Services) is a set of Web Services (WS) [5], and relies on the cloud computing infrastructure for delivering its services. These services can be accessed using REST (Representational State Transfer) and SOAP (Simple Object Access Protocol) protocols.

Within a number of services provided by Amazon, EC2 (Elastic Compute Cloud) and S3 (Simple Storage Service) are the most popular and well-known services. Other services have also been developed around these basic services such as EBS (Amazon Elastic Block Store), AWS Management Console, etc. one of the latest services provided by Amazon consists in Cloud watch for monitoring the applications that are running in the cloud.

Amazon services are paid according to the user's consumption (number of requests, amount of bandwidth, etc.). However, in February 2011, Amazon released a free tier account for the developers in order to foster the creation of applications based on their cloud infrastructure. In the context of mobile technologies, Amazon provides support for Android. *Amazon SimpleDB* service works with S3 [2] and EC2 [1], and provides the ability to store, process and query data sets in the cloud. Each dataset is organized into domains, and can run queries across all of the data stored in a particular domain. Domains are collections of items that are defined by attribute-value pairs

Amazon SimpleDB stores multiple geographically distributed copies of each domain to offer high availability and data durability. A successful write means that all copies of the domain will durably persist. Amazon SimpleDB supports two read consistency options: eventually consistent reads and consistent reads. The Eventually Consistent option gives the best read performance and it is used by default. However, an eventually consistent read might not return the most recently completed write. Consistency across all copies of data is usually reached within a second; repeating a read after a short time should return the updated data. Amazon SimpleDB also provides the flexibility and control when requesting a consistent read. A consistent read returns a result, and this result reflects all writes that received a successful response prior to the read.

Amazon SimpleDB is not a relational database and sacrifices complex transactions and relations (i.e., joins) in order to provide unique functionality and performance. However, Amazon SimpleDB does offer transactional semantics such as: Conditional put and conditional delete are new operations, which were added in February, 2010. They address a problem that arises when accessing SimpleDB concurrently. Considering a simple program that uses SimpleDB to store a counter, i.e. a number that can be incremented, the program must do three things:

- Retrieving the current value of the counter from SimpleDB.
- Adding one to the value.

- Storing the new value in the same place as the old value in SimpleDB.

If this program runs while no other programs access SimpleDB, it will work correctly. However, it is often desirable for software applications (particularly web applications) to access the same data concurrently. When the same data is accessed concurrently, a race condition arises, which would result in a undetectable data loss.

Consistent read was a new feature that was released at the same time as conditional put and conditional delete. As the name suggests, consistent read addresses problems that arise due to SimpleDB's eventual consistency model. Considering the following sequence of operations:

1. Program *A* stores some data in SimpleDB.
2. Immediately after that, *A* requests the data it just stored.

SimpleDB's eventual consistency only guarantees that Step 2 reflects the complete set of updates in Step 1, or none of those updates. Consistent read can be used to ensure that the data retrieved in Step 2 reflect changes in Step 1.

The reason why inconsistent results can arise when the consistent read operation is not used is that SimpleDB stores data in multiple locations (for availability), and the new data in Step 1 might not be written at all locations when SimpleDB receives the data request in Step 2. In that case, it is possible that the data request in Step 2 is serviced at one of the locations where the new data has not been written.

Amazon discourages the use of consistent read, unless it is required for correctness. The reason for this recommendation is that the rate, at which consistent read operations are serviced, is lower than for regular reads.

3.4 DynamoDB

DynamoDB [26, 27] is a NoSQL database service. All data items are stored on Solid State Drives (SSDs), and are replicated across 3 Availability Zones for high availability and durability. With DynamoDB, one can offload the administrative burden of operating and scaling a highly available distributed database cluster.

DynamoDB is different from the traditional NoSQL solutions in that it maintains the relational model of tables. Availability is increased with multiple replicas distributed geographically across three different Availability Zones in order to maintain a fault-tolerant architecture.

This is much like MongoDB's replica sets in order to ensure that, if one node goes down, the data is still available in another geographically distributed node. As a consequence, data along the network is increased. DynamoDB also uses a solid state storage method to further improve the performance. This increases the speed of reads and writes, and aims to minimize the amount of latency when performing operations on the server.

DynamoDB stores multiple copies of each data item to ensure durability. When you receive an "operation successful" response to your write request, the server ensures that the write is durable on several servers. However, it takes time for the update to propagate to all copies. The data is eventually consistent, and this means that a read request immediately after a write operation might not show the latest version. However, DynamoDB offers the option to request the latest version of the data.

When one reads data (GetItem, BatchGetItem, Query or Scan operations), the response might not reflect the results of the latest completed write operation (PutItem, UpdateItem or DeleteItem), and the response might include old versions of data. By default, the Query and GetItem operations use eventually consistent reads, but one can optionally request strongly consistent reads. BatchGetItem operations are eventually consistent by default, but one can specify strongly consistent on a per-table basis. Scan operations are always eventually consistent.

When one client issues a strongly consistent read request, DynamoDB returns a response with the most up-to-date data that reflects the updates from all prior related write operations, to which DynamoDB returned a successful response. A strongly consistent read might be less available in the case of a network delay or outage. For the query or get item operations, you can request a strongly consistent read result by specifying optional parameters in your request.

DynamoDB supports a "conditional write" where you specify a condition when updating an item. DynamoDB writes the item if and only if the specified condition is met; otherwise, it returns an error. DynamoDB also provides an "atomic counter" feature where you can send a request to add or subtract from an existing attribute value without interfering with another simultaneous write request. For example, a web application might want to maintain a counter per visitor to its site. In this case, the client only wants to increment a value regardless of what the previous value was.

3.5 Riak

Riak (http://docs.basho.com/riak/latest/) is an open-source, fault-tolerant key-value NoSQL database. It implements the principles from Amazon's Dynamo paper [26], and shows a heavy influence from Dr. Eric Brewer's CAP Theorem. Written in Erlang, Riak is known for its ability to distribute data across nodes by using consistent hashing in a simple key/value scheme in namespaces called buckets.

Riak supports a REST API through HTTP and Protocol Buffers for basic PUT, GET, POST, and DELETE operators. Additional query choices are offered, including secondary indices, Riak Search using the Apache Solr Engine with Solr client query APIs, and MapReduce. MapReduce has native support for both JavaScript and Erlang. Riak evenly distributes data across nodes with consistent hashing and can provide an excellent latency profile, even in the case of multiple node failures. Key/Values can be stored in memory, disk, or a combination, depending on which pluggable backend one chooses.

Riak also supports the feature of checking if the server is available. An instantiation of the client will automatically execute a “Client Ping” command to ensure that the node defined by the client is available for requests. This will provide some reference to the users about whether they need to check their installation before continuing. When multiple datacenters are used on replication, one cluster acts as a "primary cluster". The primary cluster handles replication requests from one or more "secondary clusters". If the datacenter with the primary cluster goes down, a secondary cluster can take over as the primary cluster. There are two modes of operation: fullsync and real-time. In fullsync mode, a complete synchronization occurs between primary and secondary cluster(s), by default every 360 minutes. In real-time mode, continual, incremental synchronization occurs - replication is triggered by new updates.

Riak provides the highest degree of flexibility, and allows to trade off availability and consistency on a per-request basis. It achieves such a feature by allowing reads and writes with three different parameters: (N) for nodes, (W) for writes, and (R) for reads. N represents the number of nodes where data will be replicated. W is the number of nodes that must be written successfully before a response is returned. R is the number of nodes from which data must be read in order to reply to a request.

Let us consider an example of a simple Riak cluster with five nodes and a default quorum of 3, which means every data item is stored on 3 nodes. In this setup, reads use a quorum of 2 to ensure at least two copies, and writes also use a quorum of 2 to enforce strong consistency. When data is written with a quorum of 2, Riak sends the write request to all three replicas anyway, but returns a successful reply when two of them respond with a successful write.

Every key belongs to N primary virtual nodes (vnodes), which are running on the physical nodes assigned to them in the ring. Secondary virtual nodes are run on nodes, which are close to the primaries in the key space and stand in for primaries when they are unavailable (also called fallbacks). The basic steps of a request in Riak are as follows:

1. Determining the vnodes responsible for the key from the preference list
2. Sending a request to all the vnodes determined in the previous step
3. Waiting until enough requests return the data to fulfill the read quorum (if specified) or the basic quorum
4. Returning the value to the client

In a typical failure scenario, at least one node fails and two replicas are intact in the cluster. Clients can expect that reads with an R of 2 will still succeed, until the third replica comes back up again.

3.6 DeeDS

DeeDS [7, 28, 31] is a prototype of a distributed, active real-time database system. It aims to provide a data storage for real-time applications, which may have hard or firm real-time requirements. As database, DeeDS uses OBST (Object Management system of STONE) [22] and TDBM (DBM with transactions), which replaces the OBST storage manager. One main reason for introducing TDBM is to add support of nested transaction into DeeDS. TDBM is a transaction processing data store with a layered architecture [18], and provides DeeDS with:

- Nested Transactions
- Volatile and persistent databases
- Support for very large data items

To meet real-time constraints, all operations supported by DeeDS have to be predictable. This is ensured by avoiding delays for disk access, network communication and distributed commit through main memory residency, full replication and local commit of transactions. Local commit means that transactions are allowed to commit on a node by updating only the local database of that node.

The other nodes are informed eventually. This behavior not only avoids the unpredictable execution time of distributed commit protocols like the “Two

Phase Commit Protocol", but also weakens global consistency. Instead of immediate global consistency, DeeDS supports eventual global consistency. Local commit also introduces some concurrency problems like concurrent updates of different replicas belonging to the same object.

To handle these problems, DeeDS uses conflict detection and forward conflict resolution, which resolves conflicts without rolling back transactions [7]. Conflict resolution is done deterministically on all nodes so that global consistency is reached eventually, if there are no new updates to the database. Local consistency at each node is ensured at all times by the pessimistic concurrency control offered by OBST/TDBM.

To achieve better portability, an extra layer called DOI (Deeds Operating systems Interface) is used between DeeDS and the operation system. This makes it possible to run Deeds not only on POSIX compliant systems like UNIX or LINUX, but also on real-time OSE Delta.

3.7 Zatara

The *Zatara* database [22] is a distributed database engine that features an abstract query interface and plug-in-able internal data structures. Zatara is designed for the framework, where it is flexible enough to be used by any software application, and guarantees data integrity and achieves high performance and scalability.

In *Zatara*, nodes are organized in groups, each group contains at least two nodes, and the actual size of the group depends on the developer. A node has a NodeID and a GroupID. NodeIDs are 32 bit integers, and GroupIDs are 16 bit integers. The developer chooses between two key storage caches: in the first cache, data is stored in a single node and is not resistant to single node failure; in the other cache, data is stored in persistent storage and is resistant to node failures, and the eventual consistency is used between nodes. The distributed database ZATARA also tries to address most of the limitations presented in other systems, and proves that it is technically possible to scale almost linearly as long as there are no ACID requirements.

ZATARA uses the algorithm of the consistent hashing. With the algorithm, the client will read or write the information from/on a particular node. If a node is not accessible, a decision on what to do further is based on the class of the requested key. The keys that are stored persistent can be read/write from another node in the group. Consistent hashing does not guarantee a fair data distribution across nodes. When adding new nodes, some keys must be redistributed. In order to perform the consistent hashing, the client should have an overview of the infrastructure.

4 EVALUATION OF SYSTEMS

MongoDB is a cross-platform document-oriented NoSQL database system, and uses BSON to store data. as its data mdoel. MongoDB is free and open source software, and has official drivers for a variety of popular programming languages and development environments. Web programming language *Opa* also has built-in support for MongoDB, and offers a type-safety layer on top of MongoDB. There are also a large number of unofficial or community-supported drivers for other programming languages and frameworks.

CouchDB is an open source NoSQL database, and uses JSON as its data mdoel, JavaScript as its query language and HTTP as API. CouchDB was first released in 2005 and later became an Apache project in 2008. One of CouchDB's distinguished features is multi-master replication. The features of MongoDB and CouchDB are summarized in the Table 1.

Table 1: MongoDB and CouchDB features

Feature	**MongoDB**	**CouchDB**
Interface	Custom	HTTP/REST
Data Model	BSON, NOSQL	JSON, NOSQL
Storage Model	Caching	
Consistency	Strong + eventual	Eventual
Collection	Collection	
Replication	Master slave	Multi master
Concurrency	Update in place	MVCC
Transactions	No atomicity	Atomicity
Availability	Open	Open
Query language	Javascript	Javascript, REST, Erlang

Amazon SimpleDB is a distributed database written in Erlang by Amazon.com. It is used as a web service with Amazon Elastic Compute Cloud (EC2) and Amazon S3, and is part of Amazon Web Services. It was announced on December 13, 2007.

Amazon DynamoDB is a fully managed proprietary NoSQL database service that is offered by Amazon.com as part of the Amazon Web Services portfolio. DynamoDB uses a similar data model as Dynamo, and derives its name also from Dynamo, but has a different underlying implementation: DynamoDB has a single master design. DynamoDB was announced by Amazon CTO Werner Vogels on January 18, 2012.

Riak is an open-source, fault-tolerant key-value NoSQL database, and implements the principles from

Amazon's Dynamo. Riak uses the consistent hashing to distribute data across nodes, and buckets to store data.

Both DeeDS and ZATARA are the result from research projects and not yet mature enough for production usage. The features of DynamoDB, SimpleDB and Riak are summarized in the Table 2.

Table 2: DynamoDB, SimpleDB and Riak features.

Feature	**DynamoDB**	**SimpleDB**	**Riak**
Interface	Table	REST, SOAP	Erlang
Data Model	key-value		Key-value
Storage Model	API		backend
Consistency	strong + eventual	strong + eventual	configurable eventual
Collection	Collection of key-value		
Replication	master slave	master slave	Master less multisite replication
Concurrency	Optimistic		
Transactions	Atomicity		
Availability	Commercial	Commercial	Open
Query language	API calls		Erlang Map-reduce

We use the following criteria to evaluate the database systems that support the eventual consistency:

- Popularity
- Maturity
- Consistency
- Use cases

4.1 Popularity

We evaluate the popularity of the presented database systems based on DB-Engines ranking (http://db-engines.com/en/ranking). The DB-Engines Ranking ranks database management systems according to their popularity.

- At the beginning of 2014, MongoDB was ranked 7th with a score of 96.1. In February 2014, it is ranked 5th with the score 195.17.
- At the beginning of 2014, CouchDB was ranked 16th. In February 2014, CouchDB is ranked 19th with the score 23.34.
- At the beginning of 2014, Riak was ranked 27th. In February 2014, Riak is ranked 30th with the score 10.77.
- At the beginning of 2014, DynamoDB was ranked 35th with a score of 7.20. In February 2014 DynamoDB is ranked 33rd with the score 8.36.
- At the beginning of 2014, SimpleDB was ranked 46th. In February 2014 SimpleDB, is ranked 48th with the score 3.30.

According to this ranking, MongoDB is clearly the most popular and widely known database system supporting the eventual consistency

4.2 Maturity

Based on the authors' research, MongoDB is clearly the most mature database system using eventual consistency. It has a large user and customer base and is actively developed. MongoDB has official drivers for several popular programming languages and development environments. There are also a huge number of unofficial or community-supported drivers for other programming languages and frameworks.

Riak is available for free under the Apache 2 License. In addition, Riak uses Basho Technologies to offer commercial licenses with subscription support and the ability for MDC (Multi Data Center) Replication. Riak has official drivers for Ruby, Java, Erlang, Python, PHP, and C/C++. There are also many community-supported drivers for other programming languages and frameworks.

CouchDB is a NoSQL database. CouchDB uses JSON to store data, supports MapReduce query functions in JavaScript and Erlang. CouchDB was first released in 2005 and became an Apache project in 2008. The replication and synchronization features of CouchDB make it ideal for mobile devices, where network connection is not guaranteed but the application must keep on working offline. CouchDB is also suited for applications with accumulating, occasionally changing data, on which pre-defined queries are to be run and where versioning is important (CRM, CMS systems, for example). The master-master replication is an especially interesting feature of CouchDB, which allows easy multi-site deployments. CouchDB is clearly a mature system and used in production environments.

DynamoDB is a commercially managed NoSQL database service, offered by Amazon.com as part of the Amazon Web Services portfolio. There is also a local development version of DynamoDB, with which developers can test DynamoDB-backed applications locally. The programming languages with DynamoDB binding include Java, Node.js, .NET, Perl, PHP, Python, Ruby, and Erlang. Therefore, DynamoDB is a mature and production-quality service.

Amazon SimpleDB is on the Beta phase and thus we do not suggest its use in production.

ZATARA and DeeDS are in the research phase and there are no publicly available systems for testing.

Therefore, they are at most in the Alpha phase and we do not recommend their use in production as well.

4.3 Consistency

From earlier research, we know that Amazon SimpleDB's inconsistency window for eventually consistent reads was almost always less than 500ms [49], while another study found that Amazon S3's inconsistency window lasted up to 12 seconds [2, 12]. However, to the author's knowledge, there is not a widely known and accepted workload for the databases using eventual consistency. Therefore, the comparison of consistency or inconsistency must be based solely on system features.

From a point of view of consistency, Riak offers the most configurable consistency feature, which allows selecting the consistency level. MongoDB, SimpleDB and DynamoDB offer the possibility to read the latest version of the data time, thus providing strong consistency as well as eventual consistency. All other systems offer only eventual consistency, and may return an old version of the data when performing read operations.

4.4 Use cases

MongoDB has been successfully used on operational intelligence, especially on storing log data, creating pre-aggregated reports and in hierarchical aggregation. Furthermore, MongoDB has been used on product management systems to store product catalogs, manage inventory and category hierarchy. In content management systems, MongoDB is used to store metadata, asset management and store user comments on content, like blog posts and media.

Riak has been successfully used on simple high read-write applications for session storage, serving advertisements, storing log data and sensor data. Furthermore, Riak has been used in content management and social applications for storing user accounts, user settings and preferences, user events and timelines, and articles and blog posts.

The replication and synchronization capabilities of CouchDB are well suited in mobile environment, where network connection is not guaranteed, but the application must keep on working offline. CouchDB is also ideal for the applications with accumulating, occasionally changing data, on which pre-defined queries are to be run, and where versioning is important. CRM, CMS systems are the examples of such applciations. CouchDB has an especially interesting feature: master-master replication, which allows easy multi-site deployments.

SimpleDB is well suited for logging, online games, and metadata indexing. However, one cannot use SimpleDB for aggregate reporting: there are no aggregate functions such as SUM, AVERAGE, MIN, etc. in SimpleDB. Metadata indexing is a very good use case for SimpleDB. One can also have data stored in S3 and use SimpleDB domains to store pointers to S3 objects with more information about them.

Another class of applications, for which SimpleDB is ideal, is sharing information between isolated components of an application. SimpleDB also provides a way to share indexed information, i.e., the information that can be searched. A SimpleDB item is limited in size, but one can use S3 for storing bigger objects, such as images and videos, and point to them from SimpleDB. This could be called the metadata indexing.

5 ADVANTAGES AND DISADVANTAGES OF EVENTUAL CONSISTENCY

5.1 Advantages

Eventual consistency is easy to achieve and provides some consistency for the clients [11]. Building an eventually consistent database has two advantages over building a strongly-consistent database: (1) It is much easier to build a system with weaker guarantees, and (2) database servers separated from the larger database cluster by a network partition can still accept writes from applications. Unsurprisingly, the second justification is the one given by the creators of the first generation NoSQL [9] systems that adopted eventual consistency.

Eventual consistency is often strongly consistent. Several recent projects have verified the consistency of real-world eventually consistent stores [12]. One study found that Amazon SimpleDB's inconsistency window for eventually consistent reads was almost always less than 500ms [49], while another study found that Amazon S3's inconsistency window lasted up to 12 seconds [2, 12]. Other recent work shows similar results from Cassandra, where the inconsistency window is around 200ms [37].

5.1 Disadvantages

While eventual consistency is easy to achieve, the current definition is not precise [8, 39]. Firstly, from the current definition, it is not clear what the state of eventually consistent databases is. A database always returning the value 42 is eventually consistent, even if 42 were never written.

One possible definition would be that eventually all accesses return the last updated value, and thus the database cannot converge to an arbitrary value [49]. Even this new definition has another problem: what values can be returned before the eventual state of the database is reached?

If replicas have not yet converged, what guarantees can be made on the data returned? In this case, the only possible solution would be to return the last known consistent value. The problem here is how to know what version of data item was converged to the same state on all replicas [4].

Eventual consistency requires that writes to one replica will eventually appear at other replicas, and that if all replicas have received the same set of writes, they will have the same values for all data. This weak form of consistency does not restrict the ordering of operations on different keys in any way, thus forcing programmers to reason about all possible orderings and exposing many inconsistencies to users. For example, under eventual consistency, after Alice updates her profile, she might not see that update after a refresh. Or, if Alice and Bob are commenting back-and-forth on a blog post, Carol might see a random non-contiguous subset of that conversation.

When an engineer builds an application on an eventually consistent database, the engineer needs to answer several tough questions every time when data is accessed from the database:

- What is the effect on the application if a database read returns an arbitrarily old value?
- What is the effect on the application if the database sees modification happen in the wrong order?
- What is the effect on the application if a client is modifying the database as another tries to read it?
- And what is the effect that my database updates have on other clients, which are trying to read the data?

That is a hard list, and developers must work very hard in order to answer these questions. Essentially, an engineer needs to manually do the work to make sure that multiple clients do not introduce inconsistency between nodes.

One way to address these questions at least partly is to use a stronger version of eventual consistency. Let us define the strong eventual consistency.

DEFINITION 6: Strong *Eventual consistency*.

- *Eventual delivery*: An update executed at one node evenly executes at all nodes.
- *Termination*: All update executions terminate.
- *Strong Convergence*: Nodes that have executed the same updates *have* equivalent state.

To the authors' knowledge, there is currently no database system that uses strong eventual consistency. This could be because it is harder to implement.

Eventual consistency represents a clear weakening of the guarantees that traditional databases provide, and places a requirement for software developers. Designing applications, which maintain correct behavior even if the accuracy of the database cannot be relied on, is hard. In fact, Google addressed the pain points of eventual consistency in a recent paper on its F1 database [42] and noted:

"We also have a lot of experience with eventual consistency systems at Google. In all such systems, we find developers spend a significant fraction of their time building extremely complex and error-prone mechanisms to cope with eventual consistency and handle data that may be out of date. We think this is an unacceptable burden to place on developers and that consistency problems should be solved at the database level."

6 RESEARCH ISSUES

For the future research, one interesting direction is to design encapsulated solutions that offer good isolation for common scenarios. Examples are use of convergent and commutative replicated data types, and convergent merges for non-commutative operations. Another direction is scenario-specific patterns, such as compensations and queued transactions, which can be leveraged to achieve high availability, and provides consistency that applications can reason about.

Based on this review, it is clear that there is a need for a stronger consistency level that can provide the most of the CAP features. Strong eventual consistency is a step in this direction, but in our opinion more research is needed. The most important research question is: What is the strongest consistency level that can provide the essence of CAP. This study could also be extended to find out what potential stronger consistency guarantees or isolation levels can be provided for transactions containing multiple statements.

Another important research question is what kind of workload would best emulate and measure the performance and inconsistency window of eventual consistent databases. "Availability" in the CAP sense means that every node remains being able to read and write even when it is not able to communicate with the rest of the system. This is more than desirable, but it is easy to see the impossibility highlighted by the CAP theorem: If a node cannot communicate with anything else, of course it cannot remain consistent.

There is an excellent alternative: A system, which keeps some, but not all, of its nodes being able to read and write during a partition, is not available in the CAP sense, but is still available in the sense that clients can talk to the nodes that are still connected. In this way fault-tolerant databases with no single point of failure can be built without using eventual consistency.

Developers should not have to deal with eventual consistency. Vendors should stop hiding behind the CAP theorem as a justification for eventual consistency. New distributed, consistent systems like Google Spanner concretely demonstrate the falsity of a trade-off between strong consistency and high availability.

The next generation of commercial distributed databases with strong consistency will not be easy to build, but they will be much more powerful and usable than their predecessors. Like the first generation, they will have true shared-nothing distributed architectures, fault tolerance and scalability. However, rather than accepting weak eventual consistency, they will adopt far stronger models like ACID transactions or strong eventual consistency, making them more powerful and productive tools in the enterprise.

7 CONCLUSIONS

In this paper, we have presented a history of eventual consistency, and defined eventual consistency rigorously. We have reviewed several database systems that use eventual consistency and presented their significant features. Based on this review, we have evaluated these systems and discussed the advantages and disadvantages of eventual consistency and identified the future research issues.

Clearly, there are several very mature and popular database systems using eventual consistency. Most of these are actively developed and there is a strong community behind them. We believe that we will see more database systems in the future using eventual consistency or strong eventual consistency.

REFERENCES

[1] MongoDB: 10gen Inc. *Agile and Scalable*. http://www.mongodb.org/, 2011.

[2] Amazon. *Simple Storage Service (S3)*. http://aws.amazon.com/s3/, Aug. 2009.

[3] Mustaque Ahamad, Gil Neiger, Prince Kohli, James Burns, and Phil Hutto: *Causal memory: Definitions, implementation, and programming*. Distributed Computing, 9(1), 1995.

[4] Tom J. Ameloot, Jan Van den Bussche: *Deciding eventual consistency for a simple class of relational transducer networks*. ICDT 2012, pp. 86-98

[5] Anderson, C. J., Lehnardt, J., and Slater, N. 2010. *CouchDB: The Definitive Guide*. Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. January 2010, First Edition.

[6] Eric Anderson, Xiaozhou Li, Arif Merchant, Mehul A. Shah, Kevin Smathers, Joseph Tucek, Mustafa Uysal, Jay J. Wylie: *Efficient eventual consistency in Pahoehoe, an erasure-coded key-blob archive*. DSN 2010, pp.181-190.

[7] S. F. Andler, J. Hansson, J. Mellin, J. Eriksson, and B. Eftring: *An overview of the DeeDS real-time database architecture*. In Proc. of 6th International Workshop on Parallel and Distributed Real-Time Systems, 1998.

[8] Bailis, P., and Ghodsi, A: *Eventual consistency today: limitations, extensions, and beyond*, In communications of the ACM vol. 56, no. 5, PP. 55-63, May 2013.

[9] Bartholomew Daniel: *SQL vs. NOSQL*. Linux J., 2010, July 2010.

[10] Philip A. Bernstein and Nathan Goodman: *Concurrency control in distributed database systems*. ACM Computer Surveys, 13(2), June 1981.

[11] Philip A. Bernstein, Sudipto Das: *Rethinking Eventual Consistency*, SIGMOD 2013, June 22–27.

[12] Bermbach, D. and Tai S: *Eventual Consistency: How soon is eventual?* In Proc. of ACM MW4SOC '11 and 6 other workshop on Service Oriented Computing, New York, December, 2011, no.1.

[13] Bermbach David, Jörn Kuhlenkamp, Bugra Derre, Markus Klems, Stefan Tai: *A Middleware Guaranteeing Client-Centric Consistency on Top of Eventually Consistent Datastores*. IC2E 2013, pp. 114-123.

[14] Brewer, E: *PODC keynote*. http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf, 2000.

[15] Brewer, E.: *Towards Robust Distributed Systems*, (invited Talk) Principles of Distributed Computing, Portland, Oregon, SIGOPS, And SIGACT News, July 2000.

[16] Brewer, E.: *CAP twelve years later: How the "rules" have changed*. IEEE Computer, vol. 45, no. 2, pp. 23-29, February 2012.

[17] Brewer, E.: *Towards robust distributed systems*. In Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, 2000, pp. 710.

[18] Brachman, B., G. Neufeld: *TDBM: A DBM Library with Atomic Transactions*, In Proc. USENIX, San Antonio, 1992.

[19] Burckhardt, S., Leijen, D., Fähndrich, M.,Sagiv, M.: *Eventually Consistent Transactions*, ESOP 2012, pp. 67-86,

[20] Burckhardt, S., Manuel Fähndrich, Daan Leijen, Benjamin P. Wood: *Cloud Types for Eventual Consistency*. ECOOP 2012, pp. 283-307.

[21] Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang: *Understanding Eventual Consistency*. TechReport MSR-TR-2013-39. March 2013. http://research.microsoft.com/apps/pubs/default.aspx?id=189249.

[22] Bogdan Carstoiu and Dorin Carstoiu: *Zatara, the Plug-in-able Eventually Consistent Distributed Database*. AISS, 2(3), 2010.

[23] E. Casais, M. Ranft, B. Schiefer, D. Theobald, W. Zimmer: *STONE – An Overview*, Forschungszentrum Informatik (FZI), Germany, 1992.

[24] Cerami, E., S. Laurent: *Web services essentials*, O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2002.

[25] Cooper Brian, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni: *PNUTS: Yahoo!'s hosted data serving platform.* VLDB, August 2008.

[26] Decantia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W: *Dynamo: Amazon's highly available key-value store*. In Proceeding 21st ACM Symposium on Operating Systems Principles (SOSP), pp. 205-220, 2007.

[27] DynamoDB, http://aws.amazon.com/dynamodb/

[28] Eriksson, D.: *How to implement Bounded-Delay replication in DeeDS*, B.Sc. dissertation, Department of Computer Science, Högskolan i Skövde, 1998.

[29] Lynch, S. Gilbert, N: *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. ACM SIGACT News. 2002, 33(2), p. 5159.

[30] Gray, J.N., Lorie, R.A., Putzolu, G.R., and Traiger, I.L: *Granularity of locks and degrees of consistency in a shared data base*. In Modelling in Data Base Management Systems, IFIP, January 1976, pp. 365-294,

[31] Gustavsson, S., S. F. Andler: *Continuous Consistency Management in Distributed Real-Time Databases with Multiple Writers of Replicated Data*. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'05), Denver, Co, USA, 2005.

[32] Hale, C.: *You can't sacrifice partition tolerance*; Available from http://codahale.com/you-cant-sacrificepartition-tolerance.

[33] Herlihy, Maurice P. and Jeannette M. Wing.: *Linearizability: A correctness condition for concurrent objects.* ACM TOPLAS, 12(3), 1990.

[34] Lamport Leslie: *How to make a multiprocessor computer that correctly executes multiprocess programs*. IEEE Trans. Computer, 28(9), 1979.

[35] Lipton Richardand Jonathan S. Sandberg: *PRAM: A scalable shared memory.* Technical Report TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[36] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, David, G. Andersen: *Don't settle for eventual: scalable causal consistency for wide-area storage with COPS*. SOSP 2011, pp. 401-416.

[37] Membrey P. Plugge E. and Hawkins T: *The Definitive Guide to MongoDB.* Apress, 2010.

[38] E. Le Merrer, N. Le Scouarnec, Straub. G.: Bitbox: eventually consistent file sharing. NETYS, May 2013.

[39] Rahman, M., Golab, W., AuYoung, A., Keeton, K. and Wylie, J.: *Toward a principled framework for benchmarking consistency*. Workshop on Hot Topics in System Dependability, 2012.

[40] Marc Shapiro: *A Principled Approach to Eventual Consistency*. WETICE 2011, p. 1.

[41] Marc Shapiro, Bettina Kemme: *Eventual Consistency*. Encyclopedia of Database Systems, 2009, pp. 1071-1072.

[42] Shute Jeff, Vingralek Radek, Samwel Bart, Handhy Ben, Whipkey Chad, Rollins Eric, Oancea Mircea, Littlefield Kyle, Menestina David, Ellner Stephqan, Cieslewicz John, Rae Ian, Stancescu Traian, Apte Himani: *F1: A*

Distributed SQL Database That Scales, VLDB, 2013.

[43] Sivasubramanian, S. *Amazon DynamoDB: a seamlessly scalable non-relational database service*. In proceeding SIGMOD International Conference on Management of Data, ACM New York, NY, USA, 2012, pp. 729-730.

[44] Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M.J., Theimer, M.M., Welch, B. B.: *Session guarantees for Weakly Consistent Replicated Data*. In PDIS, 1994: pp. 140-149.

[45] Tharakan, R.: *Brewers CAP Theorem on distributed systems*, Scalable Web Architecture, February 14, 2010.

[46] Thomas, R. H.: *A majority consensus approach to concurrency control for multiple copy databases*. ACM Trans. on Database Systems, vol. 4, no. 2, pp. 180–209, June 1979.

[47] Traiger, I. L. Gray, J., Galtieri, C. A. and Lindsay, B. G.: *Transactions and consistency in distributed database systems*, In ACM Transactions on Database Systems, New York, vol. 7, no. 3, pp. 323–342, September 1982.

[48] Vaquero, L. M., Rodero-Merino, L., Caceres, J., and Lindner. M.: *A Break in the Clouds: Towards a Cloud Definition*. ACM SIGCOMM Computer Communication Review, vol. 39, no. 1, pp. 50–55, January 2009.

[49] Vogels, W.: *Scalable Web services: Eventually Consistent*, ACM Queue, vol. 6, no. 6, pp. 14-16, October 2009.

[50] Vogels, W.: *Eventually consistent*, Communications of the ACM, vol. 52, no. 1, pp. 40–44, January 2009.

[51] Wawa, H., Fekete, A., Zhao, L., Lee, K., A. and Liu, A.: *Data consistency and the tradeoffs in commercial cloud storage: the consumers' perspective*. In Proceedings of the Conference on Innovative Data Systems Research. Asilomar, CA, USA, January 2011.

[52] Feng Yan, Alma Riska, Evgenia Smirni: *Fast Eventual Consistency with Performance Guarantees for Distributed Storage*. ICDCS Workshops 2012, pp. 23-28.

AUTHOR BIOGRAPHIES

Mawahib Elbushra received her MSc on Computer Science from the College of Graduate Studies, Sudan University of Science & Technology. She is currently aiming PhD on Computer Science in Sudan University of Science &Technology. Her research interests include cloud databases, distributed databases and eventual consistency.

Dr. Jan Lindström is the principal engineer at SkySQL working on InnoDB storage engine and Galera cluster. Before joining SkySQL he was software developer for IBM DB2 and development manager for IBM solidDB core development. He joined IBM with the acquisition of Solid Information Technology in 2008. Before joining Solid in 2006, Jan worked on Innobase and spent almost 10 years working in the database field as a researcher, developer, author, and educator. He has developed experimental database systems, and has authored, or co-authored, a number of research papers. His research interests include real-time databases, in-memory databases, distributed databases, transaction processing and concurrency control. Jan has an MSc. and Ph.D. in Computer Science from the University of Helsinki, Finland.