Managing Scalable Persistent Data

F. D. Muñoz-Escoí, J. R. García-Escrivá, M. R. Pallardó-Lozoya, J. Esparza-Peidro

Institut Universitari Mixt Tecnològic d'Informàtica
Universitat Politècnica de València
46022 València (SPAIN)

{fmunyoz,rgarcia,rpallardo,jesparza}@iti.upv.es

Technical Report ITI-SIDI-2011/003

Managing Scalable Persistent Data

F. D. Muñoz-Escoí, J. R. García-Escrivá, M. R. Pallardó-Lozoya, J. Esparza-Peidro

Institut Universitari Mixt Tecnològic d'Informàtica
Universitat Politècnica de València
46022 València (SPAIN)

Technical Report ITI-SIDI-2011/003

e-mail: {fmunyoz,rgarcia,rpallardo,jesparza}@iti.upv.es

Abstract

Distributed applications should be able to manage dynamic workloads; i.e., the amount of client requests per time unit may vary frequently and servers should rapidly adapt their computing efforts to those workloads. This implies that these applications should be able to scale out without problems in order to handle workload peaks and to reduce their number of replicas when the workload diminishes, at least when a pay-per-use utility model is assumed. Cloud systems provide a solid basis for this kind of applications. This paper surveys different techniques being used in different modern systems in order to increase the scalability and adaptability in the management of persistent data. Those techniques follow two basic principles: (a) to minimise distributed coordination, and (b) to eliminate any sources of delay in local operation service. These principles are implemented following six complementary mechanisms: (1) to replicate data in order to improve read access parallelisation, (2) to partition the database in order to increase update access concurrency, (3) to relax the resulting replica consistency in order to ensure network-partition tolerance, (4) usage of simple operations in order to reduce concurrency control efforts, (5) declaration of simple schemas, thus reducing the dependency on elaborate indexing techniques and eliminating the need of join operations, and (6) to bound coordination for directly achieving the first stated principle.

1 Introduction

Scalability, *pay-per-use utility model* and *virtualisation* are the three key characteristics of the *cloud computing* paradigm [91]. Many modern distributed applications are service-oriented and can be easily deployed in a cloud infrastructure. One of the main difficulties for achieving scalability in a cloud system can be found in the management of persistent data, since data have traditionally been stored in secondary memory and replicated in order to overcome failures. As a result, this management necessarily implies noticeable delays.

The aim of this paper is to describe the techniques and mechanisms used in modern distributed systems in order to guarantee an acceptable level of adaptability and scalability in data management. Most of them have been inspired by current cloud systems. Some of such systems have designed very efficient solutions for some specific kinds of applications that, unfortunately, are unable to satisfy the requirements of other applications.

Although there are already some papers [43, 48, 49, 88] that state which are the principles that must be followed in order to enhance data management scalability, all of them have assumed a given kind of applications. Therefore, their lists of proposals are only partial or too specific. Thus, whilst [43, 48] recommend to simplify transactions in order to reduce synchronisation needs, by means of simple operations that only update a single item (an approach followed in some cloud systems [31, 35, 41]), other papers [88] and systems [66, 30, 63] still consider that regular transactions are recommendable and they need to be supported, contradicting the former. Besides, other papers [49] argue that the key elements could be the

use of *idempotent actions* and *asynchronous propagation*, generating as a result a *relaxed consistency* that should be assumed by application programmers. Hence, no complete agreement on the set of mechanisms to be used in order to obtain a scalable system exists nowadays.

This paper takes an ample snapshot of the current approaches. Its main aim is to provide a general description of those solutions for the readers interested in this field providing an initial set of references to follow in order to obtain further details about each of these systems. Additionally, we provide a general comparison of those systems and point to possible new trends in this area.

The rest of the paper is structured as follows. Section 2 presents the common characteristics of the assumed system model in most scalable systems. Section 3 presents the general mechanisms to achieve scalability. Later, Sections 4 to 8 describe each one of such recommendations and discuss the trade-offs they introduce. Section 9 summarizes the main characteristics of the most relevant scalable systems and, finally, Section 10 concludes the paper.

2 System Model

Most modern data-centres for any kind of cloud infrastructure assume the following characteristics:

- The system is composed by multiple machines. There could be tens of machines in a minimal configuration, but they could easily reach the thousands, or even millions.
- In each data-centre network latencies are small and have little variance. A partially synchronous distributed system can be assumed due to this fact in that environment.
- Nodes may fail by crashing. Most data-centres are carefully monitored and provide a trusted environment where Byzantine failures may not arise.
- Replication generally follows a *Read-One Write-All-Available* (ROWAA) [21] approach. Although some other quorum-based approaches exist, *Jiménez-Peris et al.* [52] proved that, among them, a ROWAA method is the best alternative in order to improve both performance and scalability. This implies that read-only requests can be directly served by a single replica and that no inter-server interaction is needed in that case. On the other hand, update requests should eventually reach all servers that hold any copy of the updated data. This does not necessarily imply that all system nodes need to co-operate when an update request is started, since the amount of data item copies could be smaller than the number of system nodes.

Additionally, some scalable applications might require their deployment in multiple data-centres, thus reaching a world-wide scope. In those cases, the inter-data-centres links usually have non-negligible latencies and a severely limited bandwidth (at least, when they are compared with internal networks in regular data-centres).

3 Scalability Mechanisms

Intuitively, a distributed system is *scalable* if it is able to increase its computing power in order to deal with increasing workloads. To this end, two different approaches exist: vertical scalability and horizontal scalability. In the *vertical* case, the computing capacity of each node should be increased; for instance, expanding its main memory or replacing its CPU or disk with a newer and faster or larger one, respectively. In the *horizontal* case, the system is extended including additional nodes to it. So, the larger the number of nodes, the better the system's computing service becomes.

Although vertical scalability cannot be ignored, current designs for scalable data management are concentrated in the horizontal variant. In an ideal world, when the set of nodes that compose the distributed system was extended, a linear scalability would be obtained; i.e., the workload that could be managed in a larger system would be directly proportional to the amount of new nodes that have been added to it. Moreover, such trend should be maintained infinitely.

Some characteristics to be considered in the design of scalable data systems have been suggested in different papers [48, 43] and they have been widely accepted and followed in the design of modern data scalable systems. Let us recall those recommendations and analyse in the next sections which different approaches have been adopted in current systems to implement them. Note that none of these characteristics comes for free, since there are some trade-offs between them and also when they are confronted with the regular properties that clients expect in any distributed service.

The mechanisms to be considered are:

M1 *Replication*. Data must be replicated. This allows that different server nodes hold copies of the data and each of such servers could be placed close to a given set of clients, minimising thus the time needed for propagating the requests and replies exchanged by clients and servers.

Replication is also the key principle to achieve *failure transparency* [89]; i.e., when any of the components failed, the user would not be able to perceive such failure. To this end, redundancy (replication) is mandatory.

Replication is able to ensure linear scalability when read-only requests are considered. Note that in that case the workload can be proportionally divided among all data replicas, and no interaction is needed among them.

Unfortunately, not all operations being requested by the applications would be read-only and updates always need some interactions among servers in a ROWAA model. Those update propagations introduce non-negligible delays and they might prevent the system from scaling. As a result, different complementary rules should be considered for minimising those delays.

M2 *Data partitioning*. Since the set of data being managed in a modern cloud system could easily reach Petabyte sizes [31, 4], it is impossible to maintain an entire copy of all these data in each of the server nodes. As a result, some kind of *partial replication* [21] should be adopted; i.e., only some subset of the data is stored in each server, and the updates do not need to be propagated to all system nodes.

However, partial replication introduces the risk of requiring multiple nodes for serving a single read-only query, since the set of items to be accessed might not be allocated to a single server. So, some care has to be taken in order to distribute the set of data among the set of system nodes. When the set of possible queries is known in advance, a refinement of the partial replication strategy can be considered: to partition the data in disjoint subsets, assigning each data subset to a different team of server nodes. Such approach is known as *database partitioning* [84], and has been recommended in most systems maintaining large stores [4, 7, 17, 30, 35, 36, 38, 41, 48, 78, 88]. In order to minimise service delays, these systems recommend a *passive replication* [28] model; i.e., there is a *primary* node that directly manages all update requests forwarded to a given partition, applying later those updates (once ordered by the primary) to the rest of replicas in that partition. Thus, conflicts among concurrent requests can be locally managed by this single server and the need of coordination with other replicas is eliminated (if we only consider the steps related to conflict detection and transaction ordering).

M3 *Relaxed consistency*. In a distributed system, *consistency* usually refers to bounding the divergence among the states of multiple replicas of a given piece of memory. There are different consistency models [70]. The strongest ones require a complex coordination among replicas but provide a very comfortable model for the application programmer (almost identical to that of a single machine), while the most relaxed ones are able to admit multiple differences among replicas' states and they minimise the coordination needed by system nodes, but they are very hard to programme.

Regarding consistency, the key for guaranteeing a minimal delay when client operations should be managed by a replicated data store is to select a relaxed consistency model. Most modern data systems have adopted the *eventual consistency* [92] model or recommend [43] similar relaxed models. Such model requires that, in the absence of further updates, the states of all item replicas eventually converge. If we consider that previous principles have advised a partitioned store with a passive replication model, this allows us to use *lazy propagation* [27] (also known as *asynchronous replication* [49]) of updates through secondary replicas in a trivial way; i.e., the update propagation can be done once the results of an operation execution have been reported to the client.

M4 *Simple operations*. If data operations are protected by transactions, data stores should provide concurrency control mechanisms in order to guarantee isolation, logs for ensuring data recoverability, and different kinds of indexes for locating the data to be accessed. All these managements demand a lot of computing resources. So, at a glance, one should try to avoid such costs in a scalable system, as recommended in different papers [48, 43]. The immediate effect of such attempt might be to eliminate transactions or to simplify them, only allowing single-item accesses in each operation or transaction.

Single-item operations do also simplify the design of partitioned databases. Note that if all operations are compelled to access a single item, no algorithm will be needed to obtain a perfect database partitioning since all possible partition schemes are valid.

In order to further relax communication guarantees, some papers [48, 49, 26, 43, 5] also recommend that these operations were *idempotent*; i.e., that their effects do not depend on how many times the operation is executed. Thus, if unreliable communication protocols were used, application semantics can be ensured with an *at-least-once* message delivery policy.

A final requirement that simplifies the design of recovery protocols for previously failed replicas consists in guaranteeing that all updating operations were *commutative*, as suggested in [49, 57]. This recommendation is specially important in systems that assume an asynchronous multi-primary replication model.

M5 *Simple schemas*. Relational databases provide an SQL interface that is assumed by most programmers when they need to use a database. Unfortunately, a relational schema admits some operations (joins, for instance) that would be difficult to support in a distributed environment where the database has been partitioned (as recommended in *Mechanism M2*) and the amount of server coordination steps needs to be minimised. Because of this, many scalable data stores [31, 35, 41, 59] have renounced to the relational model and have adopted a simpler single-table *key-value* [41] schema.

M6 *Limited coordination*. In spite of needing a minimal server coordination, scalable data stores should maintain some meta-data (for instance, which are the current data partitions and which has been the assignment of primary replicas to each partition) whose availability is critical. So, meta-data is also replicated but it cannot follow the loosely consistent model described above for the regular store contents: its consistency should be strong. As a result, there is a small set of critical meta-data that demands a different and specialised management. Unfortunately, this data-set requires a strong coordination between the nodes that store it. There is no agreement on the name given to the set of nodes that manage these meta-data, although the term *kernel set* may suggest what it is intended for.

In practice, any coordination needed by the set of nodes that hold the entire database could be delegated to this *kernel set*. Since the general aim is to reduce inter-node coordination as much as possible, the communication costs in the *kernel set* should have also a minimal impact on the overall performance. To this end, the *kernel set* is regularly deployed in some specialised clusters (more than one, in order to ensure its liveness in case of failure), executing all of them the same tasks (i.e., mirroring the service of any request) following a *state machine* [77] model or using the *Paxos* [61] protocol, ensuring thus a strong consistency. The updates applied on this *kernel set* are later propagated to the rest of the system, or will be known by those other nodes when they refresh the state of a cached version of such data.

One of the first proposals for an architecture with a component similar to a *kernel set* was presented in a paper from Baldoni *et al.* describing the design of a three-tier replication architecture [18] for the active replication model [77]. Its middle-tier (equivalent to the *kernel set* described here) was responsible for all the coordination required in order to guarantee a *linearisable* [50] (i.e., strong) consistency and was deployed as a cluster of nodes. On the other hand, regular replicas were placed at the third tier in that architecture and they were asynchronously accessed by the middle tier. Thus, replicas could be accessed even across WANs, and they did not require any coordination among them, but only with the middle tier.

Examples of *kernel sets* of this kind in scalable data stores are: the Chubby service [29] in Google's Bigtable clouds, Elastra's Metadata Manager and Master (MMM) [38] component, ZooKeeper [53]

in Cassandra-based [59] and Yahoo!'s [35, 34] systems, the Paxos Service component [67] in the Boxwood architecture, etc.

In order to sum up, *Mechanism M1 (Partial replication)* does not admit any objection since all distributed systems require failure transparency [89] and this demands some kind of replication. On the other hand, all remaining mechanisms (*M2 to M6*) do not perfectly match the regular deployment of common data services in a distributed system. So the following sections discuss different trade-offs raised by those recommendations.

4 Data Partitioning

Database partitioning [84, 42] was presented in the eighties as one of the best approaches to provide scalability in the management of persistent data. Assuming regular distributed transactions, a "*shared nothing*" [84] approach provides an adequate basis for replicating data and, when server interaction is adequately minimised by the replication protocol, for enhancing the scalability.

The concept presented in this first paper [84] only refers to not sharing any physical device nor resource between database machines, and this is also applicable to other database replication techniques (e.g., all protocol families based on total order broadcast that have been described and compared in [95]). However, other more recent papers [85] refine such concept and present it as a synonym of (or, at least, the basis for) "*sharding*" or *horizontal partitioning* [42]. We follow this last approach in this section.

There are several issues (presented as DPI, i.e., *data partitioning issue*, in the following list) that should be considered when horizontal partitioning is used in a distributed or replicated database:

DPI1 *Difficulty of database design.* A perfect database partitioning is able to increase performance, distributing the workload among multiple servers and thus boosting scalability. In that optimal case, each transaction does only access a single partition and it can be directly served by a single node. However, this is almost impossible to achieve in the general case, since the set of transactions that applications may execute on a given database could be dynamic and difficult to forecast. So, several transactions could require data located in multiple partitions and this would raise concurrency control and data propagation problems, thus reducing the achieved level of scalability.

Fortunately, most modern applications access relational databases using stored procedures [72] and this permits to know in advance the kind of transactions that each application will use over the stored data. With that information, the task of partitioning is simplified. For instance, different *transaction conflict classes* [23] can be defined, and each partition receives the items belonging to a given conflict class. Moreover, research on partitioning approaches is also maturing and good results in this area are already available [37].

DPI2 *Difficulty of load balancing.* The main focus of a database partitioning approach is to maximise the percentage of transactions that do only access a single database partition. However, once the database is partitioned in this way, it could be difficult to receive the same workload in each partition, since any load balancer in these systems is conditioned by the data stored and managed in each node. So, a partitioning solution should be able to consider this fact in its partitioning criteria. Note also that not all servers would have the same computing performance. As a result, per-partition workloads should be monitored and, when a reconfiguration is needed, partitions might be re-assigned to the servers that better match their needs.

DPI3 *Re-partitioning overhead.* The set of nodes that maintain the database might vary dynamically [80] and this might pose problems if the set of available partitions needs to be restructured. Note that in current systems, scalability is announced as "*infinite*" and this implies that the set of data being stored is also progressively enlarged, requiring periodical evaluations of the set of existing partitions in a given database. When new database servers join the system and new partitions need to be added, it has to be decided which data should be migrated to them and which nodes will be responsible for those new partitions. Similar problems should be managed when the set of partitions needs to be reduced, since the data originally placed in the removed partitions should be transferred to some nodes that will be their new managers (enlarging their own partitions).

DPI4 *Number of messages*. In the original paper [84], a *shared nothing* architecture was compared against others that share disk or main memory. In those cases the partitioned architecture requires more messages than its alternatives, since not all the resources are locally available as in those other systems.

A partitioned database requires that each of its fragments or partitions were replicated in order to be fault-tolerant. Therefore, each update request needs to be propagated to all replicas, demanding more messages for completing such task. Additionally, if partitioning is not perfect and each transaction (or, at least, part of them) accesses items placed in different partitions, data propagation will be also needed to manage those transactions. So, potentially, the required number of messages to serve transactions in a partitioned environment could be high.

Nonetheless, modern data scalable systems have eliminated many of these potential problems suggesting simple operations [5, 26, 43, 48] instead of regular transactions. So, updates are only allowed over single items, whilst queries are not restricted in the general case. Thus, one can ensure that update operations do only require a single partition and may be directly managed by a single server. As a result, the number of messages needed by this kind of operations can be easily bound. Queries will also follow a pre-processing in the *map-reduce* [40] paradigm (to be described in Section 6) that allows their parallelisation. Thus, although they might require a large set of messages, their response time will still be short and users will not complain.

DPI5 *Data directory*. Some directory should exist in the data store to find out which is the set of nodes that maintain each database partition. Moreover, such information should be known by all database system nodes. The aim is to get the address of the storing node for each item in a minimal time, without requiring any message exchange in the ideal case. This is also known as the *routing* [56] component of a data storage architecture and its aim is different from that of the *kernel set* component, since the latter mainly manages coordination/synchronisation tasks.

The routing functionality is easily achievable if the identifiers of all system nodes are known and a deterministic (hash or indexing) function is used for matching item IDs onto node IDs, as it has been done in *structured peer-to-peer systems* [10], for instance. So, in the regular case, no actual physical directory is needed: the deterministic function is enough, and that function can be embedded in the proxies being used by clients. As a result, the first message associated to a given operation being emitted by any client is already sent to the partition manager node able to directly answer it, thus minimising communication costs.

DPI6 *Locality of reference*. Finally, observe that in systems with a world-wide deployment (i.e., those with multiple data-centres spread over the earth), each data-centre should hold the partitions mostly accessed by clients placed near it. This requires some access monitorisation, but it will be able to further reduce the message propagation delays between clients and servers. This property was highly recommended in classical distributed databases where horizontal partitioning was applied [97], and such recommendation has been kept also in modern scalable systems [5, 17, 56].

As we have seen, the design of a partitioned data store is not trivial and it might require distributed coordination when that partitioning effort does not generate a perfectly matched distribution, compelling some transactions or operations to access multiple partitions. Because of this, other distributed architectures were proposed for achieving scalability. One of such alternatives is to hold the database in a shared set of disks that could be managed by a large set of servers [80]. That architecture has been recently proposed in the Hyder system [22] as an ideal approach for managing relational databases in cloud platforms. Its best advantage is its simplified programming model, since applications receive and use a perfect single-system image and do not need to use any distribution-aware procedure. Besides providing that transparent interface to application programmers, the architecture described in [22] does not need to internally use any distributed termination protocol nor distributed concurrency control mechanism. It maintains the database as a log of transaction intentional records; i.e., the log itself is the database. This implies that write operations can be implemented as append-only actions, making possible the usage of fast flash memory devices, further reducing the transaction completion time.

With a shared-disk architecture, it is possible to parallelise the service of concurrent transactions among the set of database servers sharing such network-reachable set of disks. To this end, Hyder adopts an

optimistic concurrency control with multi-versioning. Because of this, the usage of locks is eliminated and no delays are introduced in transaction service. Additionally, transaction records are directly written to disk before deciding whether the transaction can be committed or not. At transaction termination time, the transaction records can be checked for conflicts against other concurrent transactions already committed. Therefore, any server is able to decide on its own the transaction's fate using simple deterministic rules.

However, the shared disk (even using the fastest ones) is the main bottleneck in those architectures, and unlimited scalability is impossible. Sooner or later, the set of servers that concurrently process the incoming transactions will saturate the available disk bandwidth. Although this is a severe constraint, the simulations given in [22] report that workloads up to 100000 TPS can be served without problems in the Hyder system.

The adoption of a shared-disk strategy, as proposed in [22], makes sense for relational databases that need to use regular ACID transactions whose set of operations might not be known in advance. Note that a system of this kind does not use some of the mechanisms suggested in Section 3: there are no simple operations (*Mechanism M4*) nor a simple schema (*Mechanism M5*). Otherwise, when most of the recommendations are followed, partitioning seems to still be the preferable alternative.

5 Relaxed Consistency

As it has been already said in Section 3, replication is a highly recommended mechanism in order to develop a scalable system. Because of this, for each operation being served, the amount of interaction among replicas should also be minimised. Such minimal interaction forces to adopt a *passive* (i.e., *primary-backup*) [28] replication model, since a multi-master one would require a distributed concurrency control mechanism and it would have increased the synchronisation of such replicas and the amount of messages being exchanged for serving each operation.

Even with a passive model there are communication costs that should be considered: those needed to propagate the updates once a given operation has been executed in the primary replica. To this end, multiple papers [35, 39, 41, 43, 44, 49, 54, 57, 92] have advised the adoption of an asynchronous update propagation (i.e., that those updates were propagated once the primary had completed the transaction and replied to the client), thus leading to a relaxed replica consistency. With it, the primary or master replica always holds the latest state, but secondary replicas may have a given number of pending updates; i.e., updates not yet applied. So, the state in those backup replicas will not be the latest one and the replica consistency observed by clients reading from them is relaxed. In some cases, this asynchronous update propagation is the basis for an *eventual consistency* [92] model; i.e., one where the states of all replicas will eventually converge (for instance, when no further updates are received in a given interval of time).

In spite of this, the design outlined in the last two paragraphs is still able to provide *sequential consistency* [60] at the server side, since all replicas will be able to see the same sequence of updates and such updates are consistent with the execution of a given programme (in this case, that of the primary replica), as required by that consistency model. However, such executions are not *linearisable* [50], since once a primary replica has generated and read a new value v_2 for a given item, any secondary is still able to read afterwards an older value v_1 on its local replica; i.e., *read inversions* [16] (forbidden in the linearisable model) are possible.

Those read inversions might be perceived by user applications, generating thus a relaxed consistency model for client processes. Because of this, several systems (as Amazon SimpleDB and Google AppEngine) present in their API different consistency guarantees that can be selected by the application [94]. Surprisingly, the benchmarks used in [94] have not detected important differences in operation completion time nor throughput between the alternative consistency models supported by these systems.

But consistency may be further reduced in order to improve scalability and service time. To this end, one of the constraints stated above might be removed: the adoption of a passive replication model. Such replication model was assumed since, in the regular case, most of the update operations to be executed in a given data partition are conflictive among them and this would demand a distributed concurrency control between all master replicas in a given partition. However, no concurrency control is needed when all operations are *commutative* and this has also been another recommendation given in several papers [49, 57, 74]. So, in that case, multiple replicas in the same partition are able to directly manage update operations,

in parallel, thus balancing the updating workload among them and increasing the system productivity. Later, using asynchronous propagation, those updates are applied in all other replicas. As a result of this, consistency might be now very relaxed (since different replicas follow different sequences of updates violating thus the *causal* [6], *PRAM* [64] and even *cache* [47] consistency models) and, when all operations have been applied in all replicas, the states of such replicas will converge.

Note that many operations are not commutative, and because of this, they should be executed in the order they have been requested by clients. However, there are simple rules that allow the translation of those operations into others that are commutative. For instance, regarding operations over numerical data types, the operation to be executed could be applied onto the current value of the items that are its operands; if the result (assuming that it would be stored in a given data item A) is greater than A's previous value, the operation may be translated into an addition to A, or into a subtraction otherwise. So, when the results of such operations are propagated to other replicas, they are not expressed as the new value for the updated data items but as subtractions or additions to be applied on their current value. As a result, their application order is no longer important, since they are now commutative. Following this approach, all numerical operations would be translated into commutative operations allowing their propagation and application without problems.

Therefore, many people identifies *eventual consistency* as the assumed consistency for any cloud system. However, this might depend on the requirements of each application to be deployed in these systems. For instance, Google Megastore [17] ensures strong consistency and a SQL-like interface using synchronous update propagation. It is layered on top of the key-value stores (e.g., Bigtable [31]) developed at Google, so each application may choose its intended level of consistency, using the system that better matches its requirements. In a similar way, there are different Microsoft's papers describing several cloud-related projects [30, 49]. Thus, whilst the first one explicitly mentions that the database is partitioned and a passive replication is used [30] in order to manage each partition (and, as such, this system may still provide a sequential consistency), the latter [49] recommends commutative operations and a relaxed eventual consistency. So, it seems that it is recommendable to support different levels of consistency, adapting it to the requirements stated by each application.

Flexible consistency has been managed in different ways in several systems. The basic solution consists in providing a relaxed support in the data store and delegating other alternatives to specialised protocols [58] executed at the application level able to reinforce the resulting consistency, if needed. Other solutions [57] propose a flexible consistency management that is able to adapt the resulting consistency level to the requirements stated by applications.

The approach described in [57] consists in associating consistency requirements to the sets of data managed by applications, instead of associating them to the transactions that access such data. Data is divided into three categories (A, B and C). Category A contains data for which a consistency violation results in large penalty costs (critical data). It implements *serialisability* [21]; i.e., *sequential consistency* [60] according to [70]. Category B varies its consistency requirements depending on another factor, e.g., the availability of that data, their probability of conflicts, different time constraints, etc. This is named *adaptive consistency*. Category C comprises data that tolerates inconsistencies, at least temporarily. It ensures *session consistency* [90]. Different sample applications are considered to illustrate how categories are selected. For instance, in an electronic commerce application, the economical value of each item determines its category; in an auction system, the remaining time till the end of the auction gives each item's category; in a collaborative editing application, the probability of conflicts in accesses to each file determines each file's category.

6 Simple Operations

Multiple proposals [31, 43, 48] for scalable data stores have stated that the operations to be served by such systems should be as simple as possible, accessing a single item or entity in the ideal case. This makes sense since the simpler the operations are, the better the performance would be, so that each operation would require less effort to be completed, therefore improving the achievable concurrency level and the degree of service parallelisation.

However, not all the operations in a given application could be always so simple. So, we will distinguish

between update operations and queries in the following subsections, describing each variant separately. Additionally, transactions cannot be dropped in all applications. So, they are also described in Subsection 6.3.

6.1 Updates

Many web-based applications follow a *simple-operation* model for interacting with data [26, 35]; i.e., most update operations in those applications only require a single data item. For instance, *webmail applications* do not demand transactional semantics to an e-mail delivery, even when it is targeted to multiple destinations. Such e-mail delivery might be seen as an insert of a new item (a new "*column*") onto the information of a given entity (if we assume that e-mail accounts are managed as entities, each one with a different key). So, an interface based on simple operations is enough to manage such events. Note that high levels of scalability are demanded by these modern webmail systems since they currently manage millions of users (e.g., in systems as *gmail*, *Yahoo! mail*, or *hotmail*, to name a few), and allowing large amounts of data stored per user/account (some GBs in most cases).

The use of simple operations combined with *Mechanism M2 (Data partitioning)* provides an excellent basis to overcome the constraints imposed by the CAP theorem [46]. Recall that the CAP theorem proves that it is impossible to achieve simultaneously a service system that is consistent, available and network-partition-tolerant in both asynchronous and partially synchronous systems. At least one of these three properties should be dropped when the other two are guaranteed. Therefore, we can still build a consistent and (partially) available system even when network partitions arise. Note that data partitioning might distribute all item replicas of a given data partition in the same data-centre, promoting reference locality (i.e., *DPI6* as presented above). So, the consistency among all item replicas for a given data partition can still be ensured. With this assumption, since most operations do only access a single item, consistency is guaranteed for updates. If the network is partitioned, several database partitions may still remain reachable in each resulting network component. As a result, part of the updating operations may still find their intended target items available, and a reasonable capacity and quality of service can still be delivered; i.e., system progress is not lost. On the other hand, query operations that try to access a large set of database items might receive only a partial answer in case of a network partition. In this case consistency is sacrificed, but for many applications that relaxed answer may be enough.

Although a model based on simple operations (constrained to single-item updates) is optimal regarding scalability, it immediately implies that regular transactions are not usable since their aim was to encompass a sequence of accesses ensuring atomicity (and also isolation, consistency and durability) to their effects, and now the operations being managed only encompass a single item, but not many. So, multi-operation atomicity is intentionally sacrificed in order to improve performance. Once transactions have been discarded, complex concurrency control mechanisms, query optimisation, buffer management and other transaction management tasks can also be dismissed. Therefore, operation management is simplified and it will not generate any delays, further improving scalability.

Many systems that follow this *simple operation* principle use as their underlying storage a distributed file system based on *immutable items*. Immutable items compel the generation of new item versions on each update. As a result, each write operation is served as a log of the new values, and this further accelerates the update completion time, as it was proven by general-purpose *log file systems* [76]. On the other hand, logging might induce a slight overhead in subsequent read accesses, since the locating and positioning steps that have been saved in the writing phase compel the usage of indexes that introduce an additional level of indirection in the read steps. Nonetheless, such log indexes are maintained in main memory and the logs are periodically compacted, thus removing any overhead in read accesses. Systems like Cassandra [59] and the Google File System [44] follow this approach based on immutable items.

Some research papers [5, 26, 43, 48, 49, 66] have recommended that update operations were *idempotent*. Therefore, communication among system nodes could be asynchronous and faster, and even it might tolerate unreliable protocols. This approach reduced communication costs, and it would enhance the resulting scalability. With it, servers do only need to re-attempt the execution of each request until such requests are executed, achieving an *exactly-once* semantics with an *at-least-once* implementation. This also simplifies the development of recovery procedures. Unfortunately, this recommendation seems to contradict a previous one regarding *Mechanism M3*: operations need to be *commutative*, and to this end, they were

converted into operations that should be executed exactly once and that were not *idempotent*. The solution to this apparent conflict was already described in [49]: *idempotency* can be logically achieved when the servers are able to recall whether the incoming requests have already been executed. If so, they are simply discarded. The key to achieve this consists in tagging each request with an appropriate identifier and in guaranteeing that all attempts for a given client request will use the same identifier. Therefore, once the request is executed, all servers will be able to discard subsequent repetitions.

6.2 Queries

All we have described up to now is concerned about update operations. At a glance, queries do not seem to introduce any problem since a read-only request simply requires to access a single node (recall that we have assumed a ROWAA model). So, no server coordination is needed in this case. But in the regular case, a query is not a "simple" operation; i.e., most common queries get an answer composed by many items and this might require to contact many system nodes (i.e., many database partitions). Hopefully, assuming *Mechanism M3* the consistency being granted in a cloud system is, in the general case, quite relaxed. Therefore, in order to enhance scalability some means should be designed for promoting the maximal collaboration between server nodes in query answers, dividing the work among as many servers as possible and mixing their replies without incurring in any complex coordination among them. This aim has been accomplished with the *MapReduce* [40] programming paradigm for queries, originally developed at Google. It can be complemented with a high-level programming language, specially intended for a *map-reduce* paradigm and providing a declarative interface. Sawzall [74] is an example of this kind of languages.

In this paradigm, the intended computation is divided in two sequential phases. The first one ("map") consists in applying a mapping function onto each of the key-value pairs of a given table in order to generate several intermediate key-value pairs. The second phase applies a "reduce" function onto all intermediate pairs with a same value in their intermediate key. As a result, either zero or one pair is generated for each of such sets.

MapReduce is usually employed on top of Bigtable [31] that uses GFS [44] as its underlying file-system. Since Bigtable data is structured in tablets, and tablets are distributed onto multiple servers, it is quite easy to parallelise the execution of the "map" function. As a result, parallelisation is automatic if Bigtable is used as a data container. The intermediate key-value pairs generated by the "map" function can be stored in GFS and processed later on by other processes that execute the "reduce" function.

Hadoop [15, 13] is an open-source project, developed by The Apache Software Foundation, that provides support for the map-reduce paradigm. This certifies that this paradigm has gained wide adoption for efficient programme development onto large data stores. The Hadoop project has developed other components needed for supporting such programming paradigm, such as a distributed store that replicates data (HDFS [81], similar to GFS [44]), a high-level programming language (Pig [71], comparable to Google's Sawzall [74]) and a coordination manager (ZooKeeper [53] inspired in Chubby [29], Paxos [61] and Boxwood [67]). Many companies have adopted Hadoop as their tool for managing scalable data stores, being *Yahoo!* [34] one of the best examples since it has been directly implied in Hadoop development.

6.3 Transactions

In spite of the good performance of *map-reduce* queries and *simple* (i.e., single-entity) updates, several authors still consider mandatory the usage of some kind of transactions encompassing a given sequence of operations, since most programmers are used to them. So, here we find a trade-off between scalability (with a new programming paradigm associated to it) and reliability (with classical transactions as its main building block).

To begin with, the concept of *minitransaction* was proposed in [5] and it is able to merge both trends. On one hand, it adopts a template with a minimal set of steps to build each transaction. Therefore, it is not a simple update operation as those described above but it is also short-lived in all cases. On the other hand, it may access data placed in multiple nodes, and ensures atomicity on these accesses. Concretely, a minitransaction consists of a set of *read items*, a set of *compare items*, and a set of *conditional-write items*. Each item specifies a memory node and an address range within that memory node; compare and

conditional-write items also include the data values to be used. Items must be chosen before the minitransaction starts executing. Upon execution, a minitransaction does the following: (1) it reads the locations specified by the read items, (2) it compares the locations stated in its compare items section, if any, against the data in the compare items (equality comparison), (3) if all comparisons succeed, or if there are no compare items, it writes into the locations in the conditional-write items, and (4) it finally returns to the application the locations read in (1) and the results of the comparisons in (3). With a careful selection of data structures, the resulting Sinfonia system [5] illustrates how highly scalable services can be developed using the minitransaction concept. Although the Sinfonia paper is not focused in data scalability, it shows that transactions still make sense in the development of scalable services.

A second group of papers [4, 38, 39, 68] proposes the usage of some kind of transaction to encompass the update of multiple items in systems originally intended for simple updates. This is particularly interesting for collaborative applications [39] where most of the tasks need to update a sequence of items atomically, and this kind of support is not present in regular cloud systems that have used *Mechanism M4*. To this end, the authors of [39] propose the *KeyGroup* concept, able to dynamically encompass a set of items and ensure atomicity on their updates, with some synchronisation needed among the nodes that hold such data items.

In a third group we can find data scalable systems that are not only interested in scalability, but also in maintaining compatibility with existent standards. One example is SQL Azure [30] that needs to maintain interfaces compatible with those of SQL Server regarding database access. Assuming this constraint, transactions can not be eliminated. Note that a large group of companies that currently use or will use cloud services are interested in the *Platform-as-a-Service (PaaS)* model, and they plan to use those cloud services as a *database outsource* solution. Therefore, in those cases, an SQL interface is mandatory in order to reduce the costs of such migration. Scalability is a recommendable plus, but not the single objective of those solutions.

Other scalable systems have followed this SQL with regular transactions path. A second example is the *H-Store* system [54] based on replication, horizontal partitioning, and main-memory storage, or the *C-Store* one [86] based on vertical partitioning. Those systems follow some of the recommendations given in [88] for what should be, according to its authors, an efficient new architecture for OLTP. Those recommendations are:

1. *Main memory storage*. Since common databases in small and medium companies do not manage a huge amount of data, and current main memories are large, most databases will fit in main memory. As a result, main memory should be the default storage medium.

2. *No resource control*. Since modern OLTP transactions are not interactive and can be completed in a few milliseconds (or even microseconds assuming main-memory storage), multi-threading and concurrency control could be dropped.

3. *Shared-nothing architectures*. Grid and blade deployments are common nowadays. Instead of sharing the database among many processors in a single computer, it is better to partition it among a set of computers interconnected by a high-speed network.

4. *High availability*. Partitions may be replicated in order to guarantee high availability. Logging (i.e., redo logs) in secondary storage can be also eliminated, since recovery is ensured by the existing replicas. Only an undo log should be maintained, but it can be maintained in main memory, and only whilst the transaction is still alive.

SQL and regular ACID transactions are mandatory in these systems, since these classical characteristics are still demanded by a large set of modern applications. Note that [88] is breaking some of the assumptions initially given in our paper. We were interested in extremely scalable system storage; i.e., cloud providers try to achieve huge scalability levels since they need to support the data of a large set of companies that require their services. Therefore, the amount of data being considered is almost infinite. However, if we consider each isolated client company, the rule will be that assumed by [88]. These are two different views of the same problem, and the solutions adopted in each of them are quite different.

Nonetheless, the VoltDB [93] database manager developed as an evolution of H-Store [54] announces 45 times higher throughput than conventional OLTP DBMSs and with an almost linear scalability (in the

range from 1 to 12 machines). So, the principles outlined in [88] seem to be appropriate for small to medium systems.

7 Simple Schemas

As a consequence of *Mechanism M4 (Simple operations)*, a simplified database schema seems convenient. Since regular ACID transactions might be abandoned when *Mechanism M4* is used, there will be no need to maintain a relational database schema. Scalable data stores should look for other database schemas better tailored for simple update operations, and also to facilitate database partitioning as needed by *Mechanism M2*. Therefore, many cloud systems [8, 9, 12, 14, 19, 31, 35, 39, 41, 59, 75] use a single data store with a key-value schema, and is some of them [14, 31, 35, 59] their "value" field can be structured at will.

The main aim of any scalable approach should be to eliminate the need of server coordination and to reduce as much as possible the time needed to serve each operation. A key-value store is able to achieve these objectives. Server coordination is eliminated for updates if operations are simplified and do only access a single data item, as assumed in *Mechanism M4*. The key being used to locate the item directly provides the address of the primary server devoted to each data item, assuming a deterministic location-directory service. So, updates can be directly forwarded to the server that will manage them and no coordination nor additional message propagation step is needed to start an update operation. On the other hand, queries that manage a large set of data can be easily parallelised by the complementary map-reduce paradigm. So, their service time is also minimal, and scalability is not endangered.

But other researchers [85] have criticised the fact of eliminating the relational model and its supporting relational data stores in these new scalable systems. Indeed, the contents of [85] propose an adaptation of the relational database management systems in order to increase its performance and eliminate their service delays, thus providing an operation response time for complete transactions comparable to that of simple operations in key-value stores. That adaptation follows the advices previously given in [88] that have been outlined in Section 6: main-memory storage, data partitioning, no logging, no buffering, and no concurrency control.

Other cloud systems, as *SQL Azure* [30], *Relational Cloud* [36] or the unbundled kernel described in [66, 63], also maintain data assuming a relational database schema and provide an SQL interface to scalable applications. In [66, 63] the database service is re-structured in two independent layers: the *Transactional Component* (TC) and the *Data Component* (DC), being TC placed above DC. In such architecture, TC manages concurrency control and undo/redo recovery, whilst DC manages the physical storage structure, supporting a relational schema. Thus, DC should be involved in the indexing, caching, and physical recovery tasks. Although both transactions and a relational schema are assumed in [66, 63], its two-layer design suggests that transactions and storage should be managed independently and optimised on their own.

The effects of simplified schemas on performance has been partially evaluated in [96], where several recommendations for avoiding memory wastes are discussed. Although its results are intended for general schemas, that paper shows that using three basic rules (to recycle the free space available in indexes as a caching space; to avoid locality waste, i.e., to place together those rows or fields that will be used together; and to avoid encoding waste, i.e., to select appropriate types for each field, once the stored data is known), it is possible to achieve important improvements on performance (up to 8 times faster query time) and on memory consumption (up to 17 times lower when all rules are followed, although this depends on the concrete data store being considered).

All these proposals show that there is no complete agreement on the adequacy of a simple key-value interface and schema in order to maintain and access persistent data.

8 Limited Coordination

Traditional distributed database management systems [21] required *2-phase* [62] or *3-phase commit* [82] in order to terminate a distributed transaction. This was needed in order to guarantee their ACID properties. Those termination protocols required that all involved nodes participate in the termination procedures,

communicating their agreement on such completion. Unfortunately, this would introduce an excessive delay in a modern system that intends to be scalable.

Modern database replication protocols [95], as initially proposed in [3], have partially eliminated this overhead using *total-order broadcast* [33] for propagating transaction writesets, and limiting server coordination to a single round of update propagation, avoiding both 2PC and 3PC. Most of these protocols only need local concurrency control. Using it, they have also simplified their conflict evaluation rules and have thus removed any other need of coordination. But most of them still require synchronous eager update propagation in order to maintain consistency (i.e., the server that has directly executed the transaction should wait until writeset delivery time in order to reply to the client) and this introduces a non-negligible delay in transaction completion time.

Because of this, the remaining scalability principles outlined in this paper have progressively been proposed and adopted. They have contributed to eliminate delays in transaction/operation service time. Therefore, an optimal deployment that minimises server coordination is based on a perfect database partitioning that ensures that every operation or transaction will only access the items stored in a single partition. Besides this, the replication protocol and the assumed replica consistency must be able to admit an asynchronous update propagation; i.e., writesets could be transferred to other replicas and applied there once the results of the transaction or operation have been returned to the client application.

Thus, *replication management* will not require any server interaction that introduces client-perceivable delays when these characteristics are ensured:

- *Perfect database partitioning*. It allows that all operation/transaction items were available at a single server. With that, it is not necessary to check for the items maintained in other servers in order to complete each transaction.
- *Asynchronous update propagation*. Since this defers propagation after the transaction or operation completion and the client is unaware of such costs. Moreover, this also makes possible to batch multiple updates in a single propagation message.

Although this might eliminate all server interactions regarding replica management, there are other aspects of a data store system that should be considered. One of them is administration. Administration tasks may be needed at application deployment time, but also when the workload being supported by any application requires some kind of deployment reconfiguration in order to follow the application *service level agreement*. Additionally, node failures should be monitored and correctly managed. All these activities imply updates in several kinds of meta-data being used for administration tasks. Such meta-data, as regular application data, needs to be replicated in order to overcome failures. But, as shown in *Mechanism M3*, regular application data usually tolerates a relaxed consistency model, since the aim of most applications is to be able to scale without problems, providing acceptable service to as many clients as possible. On the other hand, meta-data needs strong consistency, since the items maintained in this second store are critical to maintain an appropriate quality of service for applications. Therefore, meta-data recovers the need for strong consistency and the adoption of heavy replication approaches in order to manage such second kind of data. Those protocols still impose a non-negligible need of coordination among the servers that manage these activities.

To solve this issue, a special component (that we have called *kernel set*) is used in most cloud systems. It is able to coordinate server interaction and to hold and administer reduced amounts of critical data. Scalability is not an issue in this second store, since not many processes will need to concurrently access such data. The important issue here is to guarantee data availability and to ensure strong consistency.

Since *kernel sets* are specialised in coordination, any component of a cloud system may manage (or delegate) any distributed coordination using the interfaces provided by such special module. Examples can be found in the papers that describe Chubby [29] or ZooKeeper [53].

9 Scalable Systems

Table 1 summarises the main characteristics of several scalable systems that are described in the sequel, showing which combinations of the mechanisms explained in this paper are actually used in real systems.

To this end, Section 9.1 groups the set of data stores that strictly follow the *key-value* schema suggested by *Mechanism M5*, whilst Sections 9.2 to 9.6 list in alphabetical order and describe some of the systems that do not follow such recommendation. The list has selected a single system for each different combination of the mechanisms and parameters shown in Table 1 in order to be concise. Nonetheless, at the end of each of those sections, other systems that also share the same configurations are also cited. Finally, Section 9.7 gives a summary and some suggestions for further research.

	Systems					
Mechanisms	Key-value stores	G-Store	Hyder	Megastore	SQL Azure	VoltDB
Data partitioning (*M2*)	Hor.(+Vert.)	Horiz.	No	Horiz.	Horiz.	Horiz.
Consistency (*M3*)	Eventual	Eventual	Strong	Multiple	Sequent.	Sequent.
Update prop.	Async.	Async.	Cache-upd	Sync.	Async.	None
Simple operations (*M4*)	Yes	Base	No	No	No	No
Concur. ctrl.	No	mutex	MVCC	MVCC	Yes	No
Isolation	No	Serial	MVCC	MVCC	Serializ.	Serial
Transactions	No	KeyGroup	Yes	Yes	Yes	Yes
Simple schema (*M5*)	KeyValue	KeyValue	Log	KeyValue	No	No
Coordination (*M6*)	Minimal	Medium	Medium	Medium	Medium	Low
Admin. tasks	Yes	Yes	No	Yes	Yes	Yes
Transac. start	No	Yes	No	No	No	bcast
Dist. commit	No	At times	Cache-upd	Yes	At times	No

Table 1: Characteristics of some scalable data management systems.

9.1 Key-value Stores

The term key-value store encompasses a rather large set of data storing systems, with some common attributes. Because of the diversity of their approaches, some authors have identified the need to further classify them into several subcategories. There is some controversy about such taxonomy, as well as the systems included in each variant, or even if a concrete system should be considered as a key-value store. We present the following classification inspired in [87], as well as the information collected from different sources:

- *Simple key-value stores*: Systems which store single key-values pairs, and provide very simple insert, delete and lookup operations. The different values can be retrieved by the associated keys, and that is the only way of retrieving objects. The values are typically considered as blob objects, and replicated without further analysis. It is the simplest approach and provides very efficient results. Some examples are Dynamo [41], Voldemort [83], Riak [19] and Scalaris [79].

- *Document stores*: Systems which store documents, complex objects mainly composed of key-value pairs. Some systems allow even nested documents, but relationships between documents are normally dropped. The core system is still based on a key-value engine, but extra lookup facilities are provided, since objects are not considered just black boxes. In this way, the documents are indexed and can be retrieved by simple query mechanisms based on the provided key-value pairs. These systems are supposed to provide the same benefits as simple key-value stores but supporting more complex data and query procedures. Some examples are SimpleDB [9], CouchDB [12], MongoDB [75] and Terrastore [25].

- *Tabular stores*: They are also known as (wide) column-based stores. These systems store multidimensional maps indexed by a key, which somehow provides a tabular view of data, composed of rows and columns. These maps can be partitioned vertically and horizontally across nodes. Some works [1, 65] suggest that column-oriented data stores are very well suited for storing extremely

big tables without observing performance degradation, and some real implementations have proved such thesis. Some examples are Bigtable [31], HBase [14], Hypertable [51], Cassandra [11, 59] and PNUTS [35].

All these data stores typically implement all recommended scalability mechanisms cited in Section 3. As a result, they are able to reach the highest scalability levels.

At a glance, their main limitation is their relaxed consistency that might prevent some applications from using those systems. However, it is worth noting that these data stores are a single component of a cloud-provider architecture and that, when needed, other pieces of such architecture are still able to mask such problems; i.e., to enforce stricter consistency guarantees, for instance.

As shown in Table 1 there are a few minor limitations on some of the listed recommendations. For instance, distributed coordination might be useful in some cases. That coordination is only needed when some administrational tasks are executed (new nodes join the system, partitions are re-arranged, etc). However, no coordination mechanism is natively implemented by these data stores, but by other components of the general architecture, as illustrated by Chubby [29] in Google's systems or ZooKeeper [53] in Yahoo!'s ones. Therefore, the *no-distributed-coordination* ideal cannot be achieved by these systems in all cases although coordination is not required for dealing with regular client operations.

9.2 G-Store

In some cases, modern web-based applications still demand some transactional support in order to encompass multiple operations in a single transaction. This may happen [39] in some applications as online gaming, collaborative editing, etc. Because of this, some data storing systems have partially sacrificed the usage of simple operations and have mixed it with the possibility of defining multi-entity transactions. *G-Store* [39] has been one of the first systems providing such support.

G-Store is a layer placed on top of a regular key-value store. So, applications may still use the functionality and reach the scalability levels described in Section 9.1. However, when needed, these applications may provide atomicity guarantees to a sequence of operations that access multiple entities/keys. To this end, a *KeyGroup* is dynamically created and several transactions can be started on it. A "grouping protocol" is needed to create the *KeyGroup*. The resulting group will have a leader that will manage all subsequent transactional steps. In order to create the group, the leader follows an algorithm similar to the one required in a mutual exclusion algorithm with a coordinator; i.e., it multicasts a request message to all the followers that answer granting an exclusive lock. Later, a final release message is also multicast by the leader when the group is dissolved. Therefore, G-Store transactions seem to be a bit stronger than regular ACID ones, since concurrency control is provided through mutual exclusion, and the resulting isolation is not only serialisable (providing the logical image of a serial order, but allowing concurrency), but a real serial order. Although an algorithm for distributed commit is not explicitly used in this system, the protocols for creating and dissolving key-groups have a comparable coordination cost. Moreover, when transactions are used, their operations should be monitored and propagated by the group coordinator, further compromising its performance (since it is a centralisation point that could be overloaded).

So, the example provided by G-Store is able to balance the excellent levels of scalability of key-value stores (when the target applications are comfortable with such interface) with a better transactional support (when multi-operation atomicity is demanded and extreme scalability is not needed). Note that the *KeyGroup* abstraction demands non-negligible coordination efforts and the resulting scalability is clearly reduced when transactions are used.

9.3 Hyder

Although database partitioning was already proposed in the eighties [84] as a mechanism able to enhance scalability, it was not the unique proposal for this aim. Contemporaneously, other different architectures appeared with good scalability levels, as those [80] based on multiple servers sharing a single data store (e.g., with multi-ported disks). Such alternative proposal has been recovered in the Microsoft *Hyder* [22] system.

Hyder's architecture still assumes relational databases. It does not follow horizontal partitioning and uses a shared set of networked flash stores. Note that horizontal partitioning is not always easy in a relational database. For instance, a many-to-many relation that needs to be used by many transactions accessing many of its rows will be almost impossible to partition and distribute [22].

Inter-server communication is kept to a minimum in this architecture since each server is able to manage its transactions using only local concurrency control mechanisms based on multi-versioning. Besides this, distributed commit is unneeded since each transaction does only use a single server.

Another important aim of this system is the usage of NAND flash memory in order to store its persistent data, since its performance and costs are improving at a fast pace. However, flash memory imposes severe restrictions on the way storage is administered: this memory does only support a limited (although progressively larger) amount of per-location writings, and although read and update operations are allowed on a per-page granularity (with typical page sizes from 512 bytes to 4096 bytes), erasure operations are only allowed on a per-block granularity (and each block groups from 32 to 128 consecutive pages). This leads to the usage of log-based file systems on this kind of storage devices and a complete redesign of several components of a database management system [22, 24].

As a result, the design of Hyder is based on a log-structured store directly mapped to a shared flash storage. The key here is that the own log is the database, and the recently used records are cached in main memory. Although distributed commit protocols are avoided, since each transaction can be served by a single node (no remote subtransaction is needed), once update transactions are completed, messages should be sent to remote nodes in order to update their caches.

In order to sum up, Hyder is able to ensure a good scalability level without respecting all suggestions given by the mechanisms described in our paper. It does not need *data partitioning*, since the complete data store is remotely stored and shared by all servers. Since that remote storage is shared and it provides a logical single image, the overall *consistency is strong*. Data are actually replicated at the file system level, but details on the replication strategy are not discussed in [22]. The users are not compelled to use *simple operations* in their applications. Transactions are admitted and a SQL-like interface might be provided on top of Hyder (although, again, details are not discussed in [22]). The logical *database schema* is not restricted by default. Write operations are always kept as appended records at the end of the database log. This is the unique implementation of the database. So, actually, the schema being used is only focused on performance: write operations do not imply any delay, and read operations (queries) are managed using local caches in each server node. This is one of the best "schemas" regarding performance. *Coordination* is unneeded for dealing with distributed commitment. Therefore, some of our proposed mechanisms have not been adopted in Hyder, but its used alternatives provide similar scalability levels.

Nonetheless, the use of a networked data store could introduce a bottleneck for achieving extreme scalability levels. Since all database servers might manage concurrently a lot of transactions, their updates might saturate the available network (in order to reach the store) or storage bandwidths. Additionally, the paper states that updates are also transferred to the remaining servers in order to directly update their caches, and this might introduce a non-negligible coordination effort (lower than that required by distributed commit protocols but still significant).

As a result, the achievable scalability level is lower than that of a key-value store, although such drawback is compensated by its transactional support (that does only require local management; making possible a regular ACID functionality).

9.4 Megastore

Similar to G-Store, Google *Megastore* [17] is a layer placed on top of a key-value database (concretely, Bigtable) with the aim of accepting regular ACID transactions with an SQL-like interface in a highly scalable system.

The *entity group* [48] abstraction is used in order to partition the database. Each entity group defines a partition, and each partition is synchronously replicated (i.e., with synchronous update propagation) and ensures strong consistency among its replicas. Replicas are located in different data-centres. Therefore, each entity group is able to survive regional "disasters". On the other hand, consistency between different entity groups is relaxed and transactions that update multiple entity groups require a distributed commit protocol. So, inter-entity-group transactions are penalised and they will be used scarcely.

Transactions use multi-versioned concurrency control. They admit three different levels of consistency: *current* (i.e., strong), *snapshot* and *inconsistent* (i.e., relaxed). Quoting the paper [17]:

> Current and snapshot reads are always done within the scope of a single entity group. When starting a current read, the transaction system first ensures that all previously committed writes are applied; then the application reads at the timestamp of the latest committed transaction. For a snapshot read, the system picks up the timestamp of the last known fully applied transaction and reads from there, even if some committed transactions have not yet been applied. Megastore also provides inconsistent reads, which ignore the state of the log and read the latest values directly. This is useful for operations that have more aggressive latency requirements and can tolerate stale or partially applied data.

So, this system ensures strong consistency thanks to its synchronous update propagation but it is also able to by-pass pending update receptions when the user application may deal with a relaxed consistency, thus improving performance.

Note that Megastore is bound to the schema provided by Bigtable. So, it is compelled to use a simple schema that is not appropriate for relational data management. So, some "denormalisation" rules are needed for translating regular SQL database schemas, implementing them in Megastore/Bigtable. To this end, Megastore DDL includes a "STORING" clause and allows the definition of both "repeated" and "inline" indexes. The former are able to index the values stored in an attribute/field of a given entity. Note that such attributes may hold a list of values in Bigtable. The latter (i.e., inline indexes) "*are useful for extracting slices of information from child entities and storing the data in the parent for fast access. Coupled with repeated indexes, they can also be used to implement many-to-many relationships more efficiently than by maintaining a many-to-many link table*".

Regarding coordination, as we have previously seen, distributed commit protocols are needed at times in Megastore: when the involved items belong to different entity groups, since each group has a different coordinator node. Besides this, every regular Megastore transaction uses a simplified variant of the Paxos [61] protocol for managing update propagation. Therefore, coordination needs are strong in this system, compromising extreme scalability.

9.5 SQL Azure

As it has been shown in the last three subsections, ACID transactions seem to be needed in highly scalable systems, and different solutions to this lack have been proposed in G-Store, Hyder and Megastore. However, none of those solutions have left the *simple schema* recommended in *Mechanism M5*. Because of this, all those systems still provide a good scalability level but they are unable to manage a fully compliant SQL interface, and such functionality is required by a large set of companies that plan to migrate their IT services to the cloud with a minimal programming effort. Note that in this case, most of those companies are more interested in database outsourcing (due to the saving in system administration tasks and in hardware renewal costs) than in extreme scalability. *Microsoft SQL Azure* [30, 69] fills this void.

Obviously, in this kind of systems *Mechanism M5* should be forgotten and a regular relational schema must be adopted instead. This implies a small sacrifice in scalability, since relational databases need specialised management regarding buffering, query optimisation, concurrency control, etc. in order to guarantee all ACID properties, providing an acceptable performance level. However, although *Mechanism M4* should also be dropped, since the aim is to fully support ACID SQL transactions, all other mechanisms (i.e., *M1*, *M2*, *M3* and *M6*) discussed in our paper are still followed.

To this end, databases are passively replicated and horizontally (or, at least, with a table-level granularity) partitioned; i.e., there is a primary server in each database logical partition, allowing thus the concurrent execution (without any needed coordination) of multiple transactions that do only access the items placed in single partitions. This might allow an asynchronous propagation of updates to the secondary replicas of each logical database partition, reducing the perceived transaction completion time, although Microsoft papers do not explain whether synchronous or asynchronous propagation is used.

Asynchronous update propagation would provide a slightly relaxed consistency model, but this is still compliant with the *generalised* [2] *serialisable* (and *snapshot* [20]) isolation levels (Recall that both isolation levels –serialisable and snapshot– are supported in Microsoft SQL Server 2008 and SQL Azure

ensures compatibility with it.) Indeed, such implemented consistency model could be tagged as *sequential* [60] since with an asynchronous passive replication it is easy to guarantee that all replicas see the same sequence of updates, and those updates follow the FIFO order set by the primary.

A distributed commit protocol might be needed, but only in case of transactions that have needed to access items placed in more than one database partition. This will not be the regular case in this system.

Several research papers have been published related to database management designs from Microsoft for cloud deployments [66, 63] but it is unclear whether such designs perfectly match the current implementation of SQL Azure. Anyway, the system is commercially available, showing that its level of scalability is adequate for the set of its target applications (mainly interested in SQL Server compatibility).

There are other prototypes or systems that share most of the aims of SQL Azure (support for ACID SQL transactions using a relational model in a scalable data store). One of them is *Relational Cloud* [36], whose main effort is placed on ensuring a perfect partitioning scheme in order to guarantee the best performance levels once the regular workload for a given database is known. Therefore, that system uses a new partitioning algorithm (inspired in that of [37]) and the resulting distribution considers both horizontal and vertical partitioning.

9.6 VoltDB

VoltDB [93] and its previous prototype *H-Store* [54] have similar aims to those of SQL Azure: to maintain a relational schema and ACID SQL transactions in a scalable data store. However, there is an important difference between both proposals: SQL Azure is the data store for a cloud provider company, whilst VoltDB is not targeted for cloud environments. This implies that virtualisation is not considered in the VoltDB design.

As it has been already mentioned at the end of Section 6, VoltDB/H-Store follows the design recommendations given in [88] for improving the scalability of relational DBMSs: horizontal partitioning, main-memory storage, no resource control, no multi-threading, shared-nothing architecture (complemented with partitioning) and high-availability (replication).

As in most of the systems described previously in Section 9 one of the key characteristics to enhance scalability is a perfect database partitioning. VoltDB also shares this requirement. Once the set of transactions that the target applications need is known, a horizontal data partitioning criterion is used. With this, distributed commit protocols are not needed and the required distributed coordination efforts are minimal: they only consist in monitoring the state of each node, reacting to node failures.

Local concurrency control mechanisms are also avoided. To this end, multi-threading is discarded and all transactions are served sequentially. Since most modern applications are not interactive, their transactions can be implemented as stored procedures and executed directly from begin to end without any pause. Since one of the other recommendations given in [88] is to deploy the database in main-memory storage, transaction completion time is minimal and high availability is ensured through active replication. To this end, the same stored procedure is invoked in all replicas of the involved database partition at once, being locally executed in each of them and avoiding any update propagation among them.

According to some performance results [73] for OLAP workloads, the scalability levels of this kind of systems is promising.

On the other hand, at a glance, it seems that such good performance levels might only be achieved for specific kinds of transactions. Since concurrency is avoided in this architecture, a workload dominated by long transactions would be difficult to manage in this architecture, whilst it would not introduce any problem in some other systems studied in this survey (that tolerate concurrency without danger). Despite this, the VoltDB architecture illustrates that there are multiple ways to achieve a given objective (SQL transactions in a scalable system) and that all of them make sense.

As a related work, it is worth noting the existence of the *C-Store* [86] proposal that shares some of the characteristics of the H-Store and VoltDB systems, but introducing an important difference: partitioning is vertical instead of horizontal. So, C-Store proposes a column-oriented partitioning aiming at query optimisation (instead of write optimisations, as done in regular relational database systems). It still maintains a relational schema, but data is partitioned, indexed and stored in order to rapidly answer the typical queries existent in the (known) applications that should access those data. A given column may be stored multiple times, in different projection sets (one per important query to be answered). So, replication is implicit in

this storage model. Write operations, despite not being the main focus in this system, are also fast. To this end, insertions are first made in a separate row-oriented *writeable store* (WS), being later moved and indexed into the "regular" column-oriented *read-optimised store* (RS) of this system. This architecture shows that the relational model could still be adapted to and used when query optimisation is the main objective.

9.7 Summary and Possible New Trends

Multiple scalable data storing systems have been cited in this paper. Most of them have been able to implement the mechanisms (fully or, at least, partially) recommended in order to ensure good levels of horizontal scalability.

The key principle followed by all these mechanisms is to avoid distributed coordination as much as possible. Indeed, this had already been recommended by distributed system textbooks (e.g., [89]) as the key for designing decentralised algorithms, that are necessary if scalability is the aim.

Unfortunately, when coordination is reduced, some troubles may arise. The first is an excessively relaxed consistency model, that may introduce some difficulties for developing correct programmes [49]. Hopefully, the data store is not the single component of a cloud system and other complementary layers are able to ensure a usable consistency level, as it has been compared and surveyed in [94]. The second trouble consists in providing a programming interface that was not adequate for all user applications. Thus, whilst simple update operations and a map-reduce query paradigm are easily parallelised and ensure impressive performance levels, not all web-based applications may follow such model and some of them still require regular ACID transactions to interact with data. Because of this, some systems (already described in Sections 9.3 to 9.6) or architectural layers [26] have given a *traditional* SQL programming interface.

At a glance, the performance and scalability bounds of key-value stores seem to be higher than those of SQL-compliant systems. Indeed, the former are able to directly implement more scalability mechanisms than the latter. However, we have found a single paper [55] comparing the performance of some of these systems, and it shows that there are not significant differences among SQL Azure and the Amazon's storage (although the latter was deployed directly on its distributed file system S3 [8] that shares most of its characteristics with the Dynamo research prototype). Both systems are far better than other configurations where regular relational DBMSs were used as the underlying cloud storage system, even in the Amazon's cloud architecture.

Regarding possible new trends in scalability mechanisms, we have to state again that the driving principles consist in (a) limiting coordination needs and (b) eliminating any local barriers that could enlarge operation service time in each node. All the mechanisms presented in our paper try to comply with either one or both principles. So, any additional scalability mechanisms should be complementary to those already presented.

A possible research line in this field is related to how to ensure local data persistency incurring in minimal delays. We have already seen how to reduce writing costs using immutable entities, leading to entity versioning and log-based file systems. This approach was already recommended in [43] and implemented in several systems (e.g., Bigtable [31], GFS [44], Cassandra [59], ...). However, some systems (e.g., the SwissBox [7] architecture with its Crescando [45] storage system, H-Store [54] and Cloudy [56]) have gone one step beyond and they are able to implement the storage layer in main-memory. Therefore, they need to have multiple replicas of each data item in order to ensure persistency in case of failure, and this requires a non-negligible coordination effort in order to propagate the updates. So, this design has eliminated local delays but it has also re-introduced communication needs (but such update propagation steps are also needed in all other scalable systems, since high availability is always a must). Hopefully, some new storing technologies are emerging (as Phase-Change Memory [32]) that ensure persistency with an access time similar to that of current RAM. This will provide a solution to what is looked for in this research line, but at the expense of a database manager redesign [32].

Finally, another interesting research line that complements pure scalability mechanisms is that of configurability; i.e., to be able to adapt a data storage system to the requirements of each application besides ensuring a high scalability. To this end, a modular architecture is needed, and the *Cloudy* [56] system is a good starting example for this trend. Cloudy is able to provide three different APIs: Key/Value, SQL, and XQuery; three different kinds of storage: main-memory hash maps, Berkeley DB, or MXQuery; and to accept different policies for managing other of its modules: routing, partitioning, consistency, load-balancing,

etc. Therefore, the selection of appropriate choices for each module seems easy, and the resulting data management could be tailored to the requirements of each application. Most of the policies and mechanisms described in that system match the recommendations outlined in our paper.

10 Conclusions

The current paper discusses which are the essential mechanisms in order to improve the scalability of persistent data storing services. It describes such mechanisms and provides some pointers to systems and research papers that have adopted them or have proposed other complementary techniques.

We have identified two main principles: (1) to reduce distributed coordination to a minimum and (2) to eliminate any sources of delay in local operation service. Those principles are concreted in a series of scalability mechanisms that should be implemented in data storing systems. These mechanisms are: (1) to replicate data in order to improve read access parallelisation, (2) to partition the database in order to increase update access concurrency, (3) to relax the resulting replica consistency in order to ensure network-partition tolerance, (4) to use simple operations in order to reduce concurrency control efforts, (5) to use simple schemas in order to reduce the dependency on elaborate indexing techniques and to eliminate the need of *join* operations, and (6) to bound coordination for directly achieving the first stated principle.

Most key-value stores are able to directly implement all these recommended mechanisms, but provide a data consistency model that might be too relaxed, needing special care for programming using it [49]. Additionally, simple operations and schemas seem to prohibit the usage of ACID transactions and relational stores. So, other systems were designed trying to by-pass some of these mechanisms but still achieving comparable levels of scalability. We have presented a short summary of each of these systems, providing references that would be useful to deepen in the knowledge of this field.

Acknowledgements

This work has been partially supported by EU FEDER and Spanish MICINN under research grant TIN2009-14460-C03-01. The authors are grateful to M. Idoia Ruiz-Fuertes, Leticia Pascual-Miret, Emili Miedes and J. Enrique Armendáriz-Íñigo for their comments on preliminary versions of this paper.

References

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Intnl. Conf. on Mngmnt. of Data (SIGMOD)*, pages 967–980, New York, NY, USA, 2008. ACM Press.

[2] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In *16th Intnl. Conf. on Data Eng. (ICDE)*, pages 67–78, San Diego, CA, USA, March 2000. IEEE-CS Press.

[3] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases. In *3rd Intnl. Euro-Par Conf.*, volume 1300 of *Lect. Notes Comput. Sc.*, pages 496–503, Passau, Germany, 1997. Springer.

[4] Divyakant Agrawal, Amr El Abbadi, Shyam Antony, and Sudipto Das. Data management challenges in cloud computing infrastructures. In *6th Intnl. Wshop. on Databases in Networked Information Systems (DNIS)*, pages 1–10, Aizu-Wakamatsu, Japan, March 2010.

[5] Marcos Kawazoe Aguilera, Arif Merchant, Mehul A. Shah, Alistair C. Veitch, and Christos T. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.*, 27(3), 2009.

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

[7] Gustavo Alonso, Daniel Kossmann, and Timothy Roscoe. SwissBox: An architecture for data processing appliances. In *5th Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 32–37, Asilomar, CA, USA, January 2011.

[8] Amazon Web Services LLC. Amazon simple storage service (S3). URL: http://aws.amazon.com/s3/, March 2011.

[9] Amazon Web Services LLC. Amazon SimpleDB. URL: http://aws.amazon.com/simpledb/, 2011.

[10] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36:335–371, December 2004.

[11] Apache Software Foundation. The Apache Cassandra project. URL: http://cassandra.apache.org, 2011.

[12] Apache Software Foundation. The Apache CouchDB project. URL: http://couchdb.apache.org, 2011.

[13] Apache Software Foundation. Apache Hadoop wiki. URL: http://wiki.apache.org/hadoop/, April 2011.

[14] Apache Software Foundation. The Apache HBase book. URL: http://hbase.apache.org/book/, April 2011.

[15] Apache Software Foundation. Hadoop map reduce project. URL: http://hadoop.apache.org/mapreduce/, April 2011.

[16] Hagit Attiya. Robust simulation of shared memory - 20 years after. *Bull. EATCS*, (100):100–114, February 2010.

[17] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *5th Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 223–234, Asilomar, CA, USA, January 2011.

[18] Roberto Baldoni, Carlo Marchetti, and Alessandro Termini. Active software replication through a three-tier approach. In *21st Symp. on Reliab. Distrib. Sys. (SRDS)*, pages 109–118, Osaka, Japan, October 2002. IEEE-CS Press.

[19] Basho Technologies, Inc. An introduction to Riak. URL: http://wiki.basho.com/An-Introduction-to-Riak.html, 2011.

[20] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *Intnl. Conf. on Mgmnt. of Data (SIGMOD)*, pages 1–10, San José, CA, USA, May 1995. ACM Press.

[21] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[22] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - a transactional record manager for shared flash. In *5th Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 9–20, Asilomar, CA, USA, January 2011.

[23] Philip A. Bernstein, David W. Shipman, and James B. Rothnie Jr. Concurrency control in a system for distributed databases (SDD-1). *ACM Trans. Database Syst.*, 5(1):18–51, 1980.

[24] Philippe Bonnet and Luc Bouganim. Flash device support for database management. In *5th Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 1–8, Asilomar, CA, USA, January 2011.

[25] Sergio Bossa. Terrastore: A scalable, elastic, consistent document store. http://code.google.com/p/terrastore/, 2011.

[26] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *Intnl. Conf. on Mngmnt. of Data (SIGMOD)*, pages 251–264, Vancouver, BC, Canada, June 2008. ACM Press.

[27] Yuri Breitbart and Henry F. Korth. Replication and consistency: being lazy helps sometimes. In *16th ACM Symp. on Princ. of Database Sys. (PODS)*, PODS '97, pages 173–184, New York, NY, USA, 1997. ACM.

[28] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Optimal primary-backup protocols. In *6th Intnl. Wshop. on Distr. Alg. (WDAG)*, volume 647 of *Lect. Notes Comput. Sc.*, pages 362–378, Haifa, Israel, 1992. Springer.

[29] Michael Burrows. The Chubby lock service for loosely-coupled distributed systems. In *7th Symp. on Operat. Sys. Design and Implem. (OSDI)*, pages 335–350, Seattle, WA, USA, November 2006. USENIX Assoc.

[30] David G. Campbell, Gopal Kakivaya, and Nigel Ellis. Extreme scale with full SQL language support in Microsoft SQL Azure. In *Intnl. Conf. on Mngmnt. of Data (SIGMOD)*, pages 1021–1024, New York, NY, USA, 2010. ACM.

[31] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. In *7th Symp. on Operat. Sys. Design and Implem. (OSDI)*, pages 205–218, Seattle, WA, USA, November 2006. USENIX Assoc.

[32] Shimin Chen, Phillip Gibbons, and Suman Nath. Rethinking database algorithms for phase-change memory. In *5th Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 21–31, Asilomar, CA, USA, January 2011.

[33] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[34] Brian F. Cooper, Eric Baldeschwieler, Rodrigo Fonseca, James J. Kistler, P. P. S. Narayan, Chuck Neerdaels, Toby Negrin, Raghu Ramakrishnan, Adam Silberstein, Utkarsh Srivastava, and Raymie Stata. Building a cloud for Yahoo! *IEEE Data Eng. Bull.*, 32(1):36–43, 2009.

[35] Brian F. Cooper, 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. *PVLDB*, 1(2):1277–1288, 2008.

[36] Carlo Curino, Evan P. C. Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: A database-as-a-service for the cloud. In *5th Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 235–240, Asilomar, CA, USA, January 2011.

[37] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[38] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An elastic transactional data store in the cloud. *CoRR*, abs/1008.3751, 2010.

[39] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: a scalable data store for transactional multi key access in the cloud. In *1st ACM Symposium on Cloud Computing (SoCC)*, pages 163–174, Indianapolis, Indiana, USA, June 2010.

[40] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Simp. on Operat. Syst. Design and Impl. (OSDI)*, pages 137–150, San Francisco, CA, USA, December 2004. USENIX Assoc.

[41] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *21st ACM Symp. on Operat. Sys. Princ. (SOSP)*, pages 205–220, Stevenson, Washington, USA, October 2007.

[42] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The Gamma database machine project. *IEEE Trans. Knowl. Data Eng.*, 2(1):44–62, 1990.

[43] Shel Finkelstein, Dean Jacobs, and Rainer Brendle. Principles for inconsistency. In *4th Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2009.

[44] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th ACM Symp. on Operat. Sys. Princ. (SOSP)*, pages 29–43, Bolton Landing, NY, USA, October 2003.

[45] Georgios Giannikis, Philipp Unterbrunner, Jeremy Meyer, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. Crescando. In *Intnl. Conf. on Mngmnt. of Data (SIGMOD)*, pages 1227–1230, Indianapolis, Indiana, USA, June 2010. ACM Press.

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

[47] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, March 1989.

[48] Pat Helland. Life beyond distributed transactions: an apostate's opinion. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 132–141, Asilomar, CA, USA, January 2007.

[49] Pat Helland and David Campbell. Building on quicksand. In *4th Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2009.

[50] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[51] Hypertable, Inc. Hypertable. http://www.hypertable.org, 2011.

[52] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, 2003.

[53] Flavio Paiva Junqueira and Benjamin Reed. The life and times of a ZooKeeper. In *28th Annual ACM Symp. on Princ. of Distrib. Comp. (PODC)*, page 4, Calgary, Alberta, Canada, August 2009. ACM Press.

[54] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[55] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Intnl. Conf. on Mngmnt. of Data (SIGMOD)*, pages 579–590, Indianapolis, Indiana, USA, June 2010. ACM Press.

[56] Donald Kossmann, Tim Kraska, Simon Loesing, Stephan Merkli, Raman Mittal, and Flavio Pfaffhauser. Cloudy: A modular cloud storage system. *PVLDB*, 3(2):1533–1536, 2010.

[57] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

[58] Konstantinos Krikellas, Sameh Elnikety, Zografoula Vagena, and Orion Hodson. Strongly consistent replication for a bargain. In *26th Intnl. Conf. on Data Eng. (ICDE)*, pages 52–63, Long Beach, CA, USA, March 2010. IEEE-CS Press.

[59] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[60] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

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

[62] Butler W. Lampson. Atomic transactions. In *Advanced Course: Distributed Systems*, volume 105 of *Lect. Notes Comput. Sc.*, pages 246–265. Springer, 1980.

[63] Justin J. Levandoski, David Lomet, Mohamed F. Mokbel, and Kevin Keliang Zhao. Deuteronomy: Transaction support for cloud data. In *5th Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 123–133, Asilomar, CA, USA, January 2011.

[64] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[65] Ziyang Liu, Bin He, Hui-I Hsiao, and Yi Chen. Efficient and scalable data evolution with column oriented databases. In *14th Intnl. Conf. on Extend. Database Techn. (EDBT)*, pages 105–116, New York, NY, USA, 2011. ACM Press.

[66] David B. Lomet, Alan Fekete, Gerhard Weikum, and Michael J. Zwilling. Unbundling transaction services in the cloud. In *4th Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2009.

[67] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *6th Simp. on Operat. Syst. Design and Impl. (OSDI)*, pages 105–120, San Francisco, CA, USA, December 2004. USENIX Assoc.

[68] Francisco Maia, José Enrique Armendáriz-Íñigo, María Idoia Ruiz-Fuertes, and Rui Oliveira. Scalable transactions in the cloud: Partitioning revisited. In *12th Intnl. Symp. on Distrib. Obj., Middleware and Appl. (DOA)*, volume 6427 of *Lect. Notes Comput. Sc.*, pages 785–787, Hersonissos, Crete, Greece, October 2010. Springer.

[69] Microsoft Corp. Windows Azure: Microsoft's cloud services platform. URL: http://www.microsoft.com/windowsazure/, March 2011.

[70] David Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, 1993.

[71] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Intnl. Conf. on Mngmnt. of Data (SIGMOD)*, pages 1099–1110, Vancouver, BC, Canada, June 2008. ACM Press.

[72] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.

[73] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Intnl. Conf. on Mngmnt. of Data (SIGMOD)*, pages 165–178, Providence, Rhode Island, USA, June 2009. ACM Press.

[74] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[75] Eelco Plugge, Tim Hawkins, and Peter Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkely, CA, USA, 1st edition, 2010.

[76] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[77] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[78] Lutz Schubert. The future of cloud computing: Opportunities for European cloud computing beyond 2010. Expert Group Report, January 2010. European Commission, Information Society and Media.

[79] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: Reliable transactional P2P key/value store. In *7th ACM SIGPLAN Wshop. on Erlang*, pages 41–48, New York, NY, USA, 2008. ACM.

[80] Kurt A. Shoens. Data sharing vs. partitioning for capacity and availability. *IEEE Database Eng. Bull.*, 9(1):10–16, 1986.

[81] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *26th IEEE Conf. on Mass Storage Syst. and Techn. (MSST)*, Incline Village, NV, USA, May 2010. IEEE-CS Press.

[82] Dale Skeen. Nonblocking commit protocols. In *Intnl. Conf. on Mngmnt. of Data (SIGMOD)*, pages 133–142. ACM Press, April 1981.

[83] SNA LinkedIn Team. Project Voldemort web site. http://project-voldemort.com/, 2011.

[84] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[85] Michael Stonebraker. SQL databases v. NoSQL databases. *Commun. ACM*, 53(4):10–11, April 2010.

[86] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A column-oriented DBMS. In *31st Intnl. Conf. on Very Large Data Bases (VLDB)*, pages 553–564, Trondheim, Norway, August 2005. ACM Press.

[87] Michael Stonebraker and Rick Cattell. Ten rules for scalable performance in "simple operation" datastores. *Commun. ACM*, 54(6), June 2011.

[88] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In *33rd Intnl. Conf. on Very Large Data Bases (VLDB)*, pages 1150–1160, Vienna, Austria, September 2007. ACM Press.

[89] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[90] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *3rd Intnl. Conf. Paral. and Distrib. Inform. Sys. (PDIS)*, pages 140–149, Austin, Texas, USA, September 1994. IEEE-CS Press.

[91] Luis Miguel Vaquero, Luis Rodero-Merino, Juan Cáceres, and Maik A. Lindner. A break in the clouds: towards a cloud definition. *Computer Communication Review*, 39(1):50–55, 2009.

[92] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[93] VoltDB, Inc. VoltDB technical overview: Next generation open-source SQL database with ACID for fast-scaling OLTP applications. Downloadable from: http://voltdb.com/_pdf/VoltDBTechnicalOverviewWhitePaper.pdf, May 2010.

[94] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data consistency properties and the trade-offs in commercial cloud storages: The consumers' perspective. In *5th Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 134–143, Asilomar, CA, USA, January 2011.

[95] Matthias Wiesmann and André Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Trans. Knowl. Data Eng.*, 17(4):551–566, 2005.

[96] Eugene Wu, Carlo Curino, and Samuel Madden. No bits left behind. In *5th Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 187–190, Asilomar, CA, USA, January 2011.

[97] Clement T. Yu and C. C. Chang. Distributed query processing. *ACM Comput. Surv.*, 16(4):399–433, 1984.