Pallardó Lozoya, MR.; Esparza Peidro, J.; García Escriva, JR.; Decker, H.; Muñoz Escoí, FD. (2011). Scalable data management in distributed information systems. Lecture Notes in Computer Science. 7046:208-217. doi:10.1007/978-3-642-25126-9_31.

Scalable Data Management in Distributed Information Systems*

M. Remedios Pallardó-Lozoya, Javier Esparza-Peidro, José-Ramón García-Escrivá, Hendrik Decker, Francesc D. Muñoz-Escoí

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
46022 Valencia (SPAIN)
{rpallardo,jesparza,rgarcia,hendrik,fmunyoz}@iti.upv.es

Abstract. In the era of cloud computing and huge information systems, distributed applications should 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. Cloud systems provide a solid basis for this kind of applications but most of the traditional relational database systems are unprepared to scale up with this kind of distributed systems. This paper surveys different techniques being used in modern SQL, NoSQL and NewSQL systems in order to increase the scalability and adaptability in the management of persistent data.

Keywords: Cloud Computing, Scalability, Data Management, High Availability, Distributed System, NoSQL

1 Introduction

Scalability, *pay-per-use utility model* and *virtualisation* are the three key characteristics of the *cloud computing* paradigm. 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-based information 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.

In order to increase scalability while maintaining persistent data, some systems simplify or even eliminate transactions [1,2,3,4,5] in order to reduce synchronisation needs, avoiding full ACID guarantees and using simple operations that only update a single item. In contrast, other papers [6] and systems [7,8,9] still consider that regular transactions are recommendable and they need to be supported, contradicting the former. The first kind of systems is known as

* The original publication is available at: http://www.springeronline.com . DOI: http://dx.doi.org/10.1007/978-3-642-25126-9_31. This paper was published by Springer in Lecture Notes in Computer Science, vol. 7046, pp. 208–217.

NoSQL systems, whilst systems that maintain the relational model with transactions and ACID guarantees that improve scalability applying a set of new mechanisms, are becoming to be known as *NewSQL* systems.

Besides, other papers [10] argue that the key elements could be the use of *idempotent actions* and *asynchronous propagation*, generating thus a *relaxed consistency* that should be assumed by application programmers. Therefore, no complete agreement on the set of mechanisms to be used in order to obtain a scalable system exists nowadays.

The aim of this paper is to provide a general description of the techniques and mechanisms that the application designer of a modern distributed information system must follow in order to guarantee an acceptable level of adaptability and scalability in data management. We also outline a general comparison of some of these systems.

The rest of the paper is structured as follows. Section 2 presents the general mechanisms to achieve scalability. Later, Section 3 summarises the main characteristics of some relevant scalable systems and, finally, Section 4 concludes the paper. An extended version of this paper can be found in [11].

2 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 memory or upgrading its CPU. In the *horizontal* case, the system is extended including additional nodes to it and, in an ideal world, when the set of nodes that compose the system was extended, a linear scalability would be obtained. Nowadays the *horizontal* approach is the most used and, from now on, we are going to assume this type of scalability in this paper.

Some characteristics of scalable data systems have been suggested in different papers [1,2] and they have been widely followed in the design of modern data scalable systems. The aim of this section is recall those recommendations and to note that none of these characteristics (or as we call, mechanisms) comes for free, since there are some trade-offs between them.

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 the time needed for propagating the requests and replies exchanged by clients and servers. If replication is used, *failure transparency* will be provided so, when any of the components failed, the user would not be able to perceive such failure.
Assuming a ROWAA (read one, write all available) model, replication is able to ensure linear scalability when read-only requests are considered. Note that in that case the workload can be divided among all data replicas, and no interaction is needed among them.

Unfortunately, updates always need some interactions among servers. 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 [3,12], 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* should be adopted; i.e., only some subset of the data is stored in each server.
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, when the set of possible queries is known in advance, a refinement of the *partial replication* strategy called *database partitioning* [13] can be considered: to partition the data in disjoint subsets, assigning each data subset to a different server.
Database partitioning has been recommended in most systems maintaining large stores [12,14,15,8,4,16,17,5,1,6] which, in order to minimise service delays, recommend a *passive* (or primary-backup) replication model. In that model, conflicts among concurrent requests can be locally managed by the *primary* 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. The strongest models require a complex coordination among replicas but provide a very comfortable view 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* [18] model. 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* [19] (also known as *asynchronous replication*) of updates.

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 durability, and different levels of buffering and caching in order to maintain an acceptable performance level. All these managements demand a high computing effort and many I/O accesses, so it is recommended [1,2] to avoid such costs in a scalable system. The immediate effect of such attempt might be to eliminate transactions or to simplify them, only allowing single-item accesses in each transaction.

Single-item operations do also simplify the design of partitioned databases because operations will not need to access more than one partition and therefore, no algorithm will be needed to obtain a perfect database partitioning. Moreover, if we turn these operations into *idempotent* ones [1,10,20,2], we will achieve 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* [10]. 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 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 (*Mechanism M2*) and the amount of server coordination steps needs to be minimised. Because of this, many scalable data stores [3,4,5,21] have renounced to the relational model and have adopted a simpler single-table *key-value* [5] schema.
These systems are becoming to be known as *NoSQL* systems.

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 and should be managed by a specialised mechanism that provides strong coordination between nodes. As there is no agreement on the name given to the set of nodes that manage these meta-data, we suggest the term *kernel set* for that mechanism.
Recent examples of *kernel sets* in scalable data stores are: the Chubby service [22] in Google's Bigtable clouds, Elastra's Metadata Manager and Master (MMM) [17] component, ZooKeeper [23] in Cassandra-based [21] and Yahoo!'s [4] systems, the Paxos Service component [24] in the Boxwood architecture, etc.

In order to sum up, notice that *Mechanism M1 (Partial replication)* does not admit any objection since all distributed systems require failure transparency and it compels 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.

3 Scalable Systems

This section presents several scalable systems, describing which combinations of the mechanisms explained above are actually used in them. To this end, Section

Table 1. Characteristics of some scalable data management systems.

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

3.1 groups the set of data stores that follow the *key-value* schema suggested by *Mechanism M5*, whilst Sections 3.2 to 3.5 describe some of the systems that do not follow such recommendation. Table 1 summarises these relationships.

3.1 Key-value Stores

The term *key-value* store encompasses a large set of data storing systems, with some common attributes. Following *Stonebraker et al.* [25], as well as the information collected from different sources, *key-value* stores can be classified in three types:

- *Simple key-value stores*: Systems which store single key-value 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 [5], Voldemort, Riak and Scalaris.
- *Document stores*: Systems which store documents, complex objects mainly composed of key-value pairs. 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. Some examples are SimpleDB [26], CouchDB, MongoDB and Terrastore.
- *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. Column-oriented

data stores are very well suited for storing extremely big tables without observing performance degradation, as some real implementations have proved. Some examples are Bigtable [3], HBase, Hypertable, Cassandra [21] and PNUTS [4].

These data stores typically implement all recommended scalability mechanisms cited in Section 2. As a result, they are able to reach the highest scalability levels. Their main limitation is a relaxed consistency that might prevent some applications from using those systems. However, other modules in a cloud system are still able to mask such problems, enforcing stricter consistency guarantees.

3.2 Megastore

Google *Megastore* [15] is a layer placed on top of a *key-value* database (concretely, Bigtable [3]) with the aim of accepting regular ACID transactions with an SQL-like interface in a highly scalable system.

The *entity group* [1] 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) [15]. 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.

Regarding coordination, as we have previously seen, distributed commit protocols are needed at times in Megastore. Besides this, every regular Megastore transaction uses a simplified variant of the Paxos [27] protocol for managing update propagation. Therefore, coordination needs may be strong in some cases.

3.3 Hyder

Hyder [28] is a *data sharing* system that enhances scalability sharing a single data store between multiple servers. Its architecture still assumes relational databases and horizontal partitioning. It uses a shared set of networked flash stores (notice

that the use of a networked data store could introduce a bottleneck for achieving extreme scalability levels). Furthermore, 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 read/write operations and storage is administered. So, the usage of log-based file systems and a redesign of several components of the database management system [28,29] are needed. As a result, the design of Hyder is based on a log-structured store directly mapped to a shared flash storage that acts as the database: write operations will be always kept as appended records at the end of the database log while read operations (queries) are managed using local caches in each server node.

Although *Coordination* is unneeded (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 strong consistency and a good scalability level without respecting all suggestions given by the mechanisms described in our paper. Nevertheless, data are actually replicated at the file system level and transactions are admitted with a SQL-like interface that might be provided on top of Hyder (although details are not discussed in [28]), making possible a regular ACID functionality.

3.4 SQL Azure

As it has been shown in the last subsections, ACID transactions may be needed in highly scalable systems, and different solutions to this lack have been proposed in Megastore and Hyder. 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* [8,30] 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. 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 and asynchronous propagation of updates to the secondary replicas are allowed, reducing the perceived transaction completion time.

A distributed commit protocol might be needed, but only in case of transactions that access items placed in multiple database partitions. This will not be the regular case in this system.

3.5 VoltDB

VoltDB [31] and its previous prototype *H-Store* 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 public cloud provider company, whilst VoltDB is mainly designed for a private cluster.

VoltDB/H-Store follows the design recommendations given in [6] 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).

Its database partitioning schema is designed by the developer, who decides if a table must be horizontally partitioned by a certain column (*partition column*) or if, in contrast, a table must be replicated. 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 joins and failures. Furthermore, since most modern applications are not interactive, their transactions can be implemented as stored procedures and executed sequentially from begin to end without any pause, discarding multi-threading and local concurrency control mechanisms.

In summary, VoltDB is designed for systems that need SQL and transactional support with full ACID guarantees and whose data fits in main memory. Unfortunately, this system is not allowed to grow elastically: if new nodes have to be added while the system is running, it is halted in order to be reconfigured. Notice that since concurrency is avoided in this architecture, a workload dominated by long transactions would be difficult to manage in this system.

4 Conclusions

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

We have seen that most key-value stores are able to directly implement all these recommended mechanisms, but provide a data consistency model that

might be too relaxed. Additionally, simple operations and schemas seem to prohibit the usage of ACID transactions and relational stores. So, other systems were designed to by-pass some of these mechanisms but still achieving comparable levels of scalability. We have also 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 supported by EU FEDER and Spanish MICINN under research grants TIN2009-14460-C03-01 and TIN2010-17193.

References

1. Helland, P.: Life beyond distributed transactions: an apostate's opinion. In: 3rd Biennial Conf. on Innov. Data Syst. Research (CIDR), Asilomar, CA, USA (2007) 132–141
2. Finkelstein, S., Jacobs, D., Brendle, R.: Principles for inconsistency. In: 4th Biennial Conf. on Innov. Data Syst. Research (CIDR), Asilomar, CA, USA (2009)
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: A distributed storage system for structured data. In: 7th Symp. on Operat. Syst. Design and Implem. (OSDI), Seattle, WA, USA, USENIX Assoc. (2006) 205–218
4. Cooper, B.F., Baldeschwieler, E., Fonseca, R., Kistler, J.J., Narayan, P.P.S., Neerdaels, C., Negrin, T., Ramakrishnan, R., Silberstein, A., Srivastava, U., Stata, R.: Building a cloud for Yahoo! IEEE Data Eng. Bull. **32** (2009) 36–43
5. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: 21st ACM Symp. on Operat. Syst. Princ. (SOSP), Stevenson, Washington, USA (2007) 205–220
6. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era (it's time for a complete rewrite). In: 33rd Intnl. Conf. on Very Large Data Bases (VLDB), Vienna, Austria, ACM Press (2007) 1150–1160
7. Lomet, D.B., Fekete, A., Weikum, G., Zwilling, M.J.: Unbundling transaction services in the cloud. In: 4th Biennial Conf. on Innov. Data Syst. Research (CIDR), Asilomar, CA, USA (2009)
8. Campbell, D.G., Kakivaya, G., Ellis, N.: Extreme scale with full SQL language support in Microsoft SQL Azure. In: Intnl. Conf. on Mngmnt. of Data (SIGMOD), New York, NY, USA, ACM (2010) 1021–1024
9. Levandoski, J.J., Lomet, D., Mokbel, M.F., Zhao, K.K.: Deuteronomy: Transaction support for cloud data. In: 5th Biennial Conf. on Innov. Data Syst. Research (CIDR), Asilomar, CA, USA (2011) 123–133
10. Helland, P., Campbell, D.: Building on quicksand. In: 4th Biennial Conf. on Innov. Data Syst. Research (CIDR), Asilomar, CA, USA (2009)
11. Muñoz-Escoí, F.D., García-Escrivá, J.R., Pallardó-Lozoya, M.R., Esparza-Peidro, J.: Managing scalable persistent data. Technical Report ITI-SIDI-2011/003, Instituto Tecnológico de Informática, Universitat Politècnica de València, Spain (2011)
12. Agrawal, D., El Abbadi, A., Antony, S., Das, S.: Data management challenges in cloud computing infrastructures. In: 6th Intnl. Wshop. on Databases in Networked Information Systems (DNIS), Aizu-Wakamatsu, Japan (2010) 1–10

13. Stonebraker, M.: The case for shared nothing. IEEE Database Eng. Bull. **9** (1986) 4–9
14. Alonso, G., Kossmann, D., Roscoe, T.: SwissBox: An architecture for data processing appliances. In: 5th Biennial Conf. on Innov. Data Syst. Research (CIDR), Asilomar, CA, USA (2011) 32–37
15. Baker, J., Bond, C., Corbett, J.C., Furman, J.J., Khorlin, A., Larson, J., Léon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing scalable, highly available storage for interactive services. In: 5th Biennial Conf. on Innov. Data Syst. Research (CIDR), Asilomar, CA, USA (2011) 223–234
16. Curino, C., Jones, E.P.C., Popa, R.A., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., Zeldovich, N.: Relational cloud: A database-as-a-service for the cloud. In: 5th Biennial Conf. on Innov. Data Syst. Research (CIDR), Asilomar, CA, USA (2011) 235–240
17. Das, S., Agrawal, D., El Abbadi, A.: ElasTraS: An elastic transactional data store in the cloud. CoRR **abs/1008.3751** (2010)
18. Vogels, W.: Eventually consistent. Commun. ACM **52** (2009) 40–44
19. Breitbart, Y., Korth, H.F.: Replication and consistency: being lazy helps sometimes. In: 16th ACM Symp. on Princ. of Database Syst. (PODS). PODS '97, New York, NY, USA, ACM (1997) 173–184
20. Brantner, M., Florescu, D., Graf, D.A., Kossmann, D., Kraska, T.: Building a database on S3. In: Intnl. Conf. on Mngmnt. of Data (SIGMOD), Vancouver, BC, Canada, ACM Press (2008) 251–264
21. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. Operating Systems Review **44** (2010) 35–40
22. Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. In: 7th Symp. on Operat. Syst. Design and Implem. (OSDI), Seattle, WA, USA, USENIX Assoc. (2006) 335–350
23. Junqueira, F.P., Reed, B.: The life and times of a ZooKeeper. In: 28th Annual ACM Symp. on Princ. of Distrib. Comp. (PODC), Calgary, Alberta, Canada, ACM Press (2009) 4
24. MacCormick, J., Murphy, N., Najork, M., Thekkath, C.A., Zhou, L.: Boxwood: Abstractions as the foundation for storage infrastructure. In: 6th Simp. on Operat. Syst. Design and Impl. (OSDI), San Francisco, CA, USA, USENIX Assoc. (2004) 105–120
25. Stonebraker, M., Cattell, R.: Ten rules for scalable performance in "simple operation" datastores. Commun. ACM **54** (2011) 72–80
26. Amazon Web Services LLC: Amazon SimpleDB. Available from: http://aws.amazon.com/simpledb/ (2011)
27. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16** (1998) 133–169
28. Bernstein, P.A., Reid, C.W., Das, S.: Hyder - a transactional record manager for shared flash. In: 5th Biennial Conf. on Innov. Data Syst. Research (CIDR), Asilomar, CA, USA (2011) 9–20
29. Bonnet, P., Bouganim, L.: Flash device support for database management. In: 5th Biennial Conf. on Innov. Data Syst. Research (CIDR), Asilomar, CA, USA (2011) 1–8
30. Microsoft Corp.: Windows Azure: Microsoft's cloud services platform. URL: http://www.microsoft.com/windowsazure/ (2011)
31. 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 (2010)