Data Management Challenges in Cloud Computing Infrastructures [*]

Divyakant Agrawal Amr El Abbadi Shyam Antony Sudipto Das

University of California, Santa Barbara
{agrawal, amr, shyam, sudipto}@cs.ucsb.edu

Abstract. The challenge of building consistent, available, and scalable data management systems capable of serving petabytes of data for millions of users has confronted the data management research community as well as large internet enterprises. Current proposed solutions to scalable data management, driven primarily by prevalent application requirements, limit consistent access to only the granularity of *single objects, rows, or keys*, thereby trading off consistency for high scalability and availability. But the growing popularity of "cloud computing", the resulting shift of a large number of internet applications to the cloud, and the quest towards providing data management services in the cloud, has opened up the challenge for designing data management systems that provide consistency guarantees at a granularity larger than *single rows and keys*. In this paper, we analyze the design choices that allowed modern scalable data management systems to achieve orders of magnitude higher levels of scalability compared to traditional databases. With this understanding, we highlight some design principles for systems providing scalable and consistent data management as a service in the cloud.

1 Introduction

Scalable and consistent data management is a challenge that has confronted the database research community for more than two decades. Historically, distributed database systems [15, 16] were the first generic solution that dealt with data not bounded to the confines of a single machine while ensuring global serializability [2, 19]. This design was not sustainable beyond a few machines due to the crippling effect on performance caused by partial failures and synchronization overhead. As a result, most of these systems were never extensively used in industry. Recent years have therefore seen the emergence of a different class of scalable data management systems such Google's Bigtable [5], PNUTS [6] from Yahoo!, Amazon's Dynamo [7] and other similar but undocumented systems. All of these systems deal with petabytes of data, serve online requests with stringent latency and availability requirements, accommodate erratic workloads, and run on cluster computing architectures; staking claims to the territories used to be occupied by database systems.

One of the major contributing factors towards the scalability of these modern systems is the data model supported by these systems, which is a collection of *key-value*

[*] This work is partially funded by NSF grant NSF IIS-0847925.

pairs with consistent and atomic read and write operations only on *single keys*. Even though a huge fraction of the present class of web-applications satisfy the constraints of *single key* access [7, 18], a large class of modern Web 2.0 applications such as collaborative authoring, online multi-player games, social networking sites, etc, require consistent access beyond *single key* semantics. As a result, modern *key-value* stores cannot cater to these applications and have to rely on traditional database technologies for storing their content, while scalable *key-value* stores drive the in-house applications of the corporations that have designed these stores.

With the growing popularity of the "cloud computing" paradigm, many applications are moving to the cloud. The *elastic* nature of resources and the *pay as you go* model have broken the infrastructure barrier for new applications which can be easily tested out without the need for huge upfront investments. The sporadic load characteristics of these applications, coupled with increasing demand for data storage while guaranteeing round the clock availability, and varying degrees of consistency requirements pose new challenges for data management in the cloud. These modern application demands call for systems capable of providing scalable and consistent data management as a service in the cloud. Amazon's SimpleDB (http://aws.amazon.com/simpledb/) is a first step in this direction, but is designed along the lines of the *key-value* stores like Bigtable and hence does not provide consistent access to multiple objects. On the other hand, relying on traditional databases available on commodity machine instances in the cloud result in a scalability bottleneck for these applications, thereby defeating the scalability and elasticity benefits of the cloud. As a result, there is a huge demand for data management systems that can bridge the gap between scalable *key-value* stores and traditional database systems.

At a very generic level, the goal of a scalable data management system is to sustain performance and availability over a large data set without significant over-provisioning. Resource utilization requirements demand that the system be highly dynamic. In Section 2, we discuss the salient features of three major systems from Google, Yahoo!, and Amazon. The design of these systems is interesting not only from the point of view of what concepts they use but also what concepts they eschew. Careful analysis of these systems is necessary to facilitate future work. The goal of this paper is to carefully analyze these systems to identify the main design choices that have lent high scalability to these systems, and to lay the foundations for designing the next generation of data management systems serving the next generation of applications in the cloud.

2 Analyzing Present Scalable Systems

Abstractly, a distributed system can be modeled as a combination of two different components. The *system state*, which is the distributed meta data critical for the proper operation and the health of the system. This state requires stringent consistency guarantees and fault-tolerance to ensure the proper functioning of the system in the presence of different types of failures. But scalability is not a primary requirement for system state. On the other hand is the *application state*, which is the application specific information or data which these systems store. The consistency, scalability and availability of the *application state* is dependent purely on the requirements of the type of application that the system aims to support, and different systems provide varying trade-offs between different attributes. In most cases, high scalability and high availability is given a higher priority. Early attempts to design distributed databases in the late eighties and early nineties made a design decision to treat both the *system state* and *applications state* as a cohesive whole in a distributed environment. We contend that the decoupling of the two states is the root cause for the high scalability of modern systems.

2.1 System State

We refer to the meta data and information required to correctly manage the distributed system as the *system state*. In a distributed data management system, data is partitioned to achieve scalability and replicated to achieve fault-tolerance. The system must have a *correct* and *consistent* view of the mappings of partitions to nodes, and that of a partition to its replicas. If there is a notion of the master amongst the replicas, the system must also be aware of the location of the master at all times. Note that this information is in no way linked to the data hosted by the system, rather it is required for the proper operation of the entire system. Since this state is critical for operating the system, a distributed system cannot afford any inconsistency or loss. In a more traditional context, this corresponds to the system state in the sense of an operating systems which has a global view about the state of the machine it is controlling.

Bigtable's design [5] segregates the different parts of the system and provides abstractions that simplify the design. There is no data replication at the Bigtable layer, so there is no notion of replica master. The rest of Bigtable's *system state* is maintained in a separate component called Chubby [3]. The *system state* needs to be stored in a consistent and fault-tolerant store, and Chubby [3] provides that abstraction. Chubby guarantees fault-tolerance through log-based replication and consistency amongst the replicas is guaranteed through a Paxos protocol [4]. The Paxos protocol [14] guarantees safety in the presence of different types of failures and ensures that the replicas are all consistent even when some replicas fail. But the high consistency comes at a cost: the limited scalability of Chubby. Thus if a system makes too many calls to Chubby, performance might suffer. But since the critical system meta data is considerably small and usually cached, even Chubby being at the heart of a huge system does not hurt system performance.

In PNUTS [6], there is no clear demarcation of the *system state*. Partition (or *tablet*) mapping is maintained persistently by an entity called the *tablet controller*, which is a single pair of active/standby servers. This entity also manages tablet relocation between different servers. Note that since there is only one *tablet controller*, it might become a bottleneck. Again, as with Chubby, an engineering solution to move the *tablet controller* away from the data path, and caching of mappings is used. On the other hand, the mapping of tablets to its replicas is maintained by the Yahoo! Message Broker (YMB) which acts as a fault-tolerant guaranteed delivery publish-subscribe system. Fault-tolerance in YMB is achieved through replication – at a couple of nodes, to commit the change, and more replicas are created gradually [6]. Again, better scalability is ensured through limiting the number of nodes (say two in this case) requiring synchronization. The per-record master information is stored as meta data for the record. Thus, the *system state* in PNUTS is split between the *tablet controller* and the *message broker*.

On the other hand, Amazon's Dynamo [7] uses an approach similar to peer-to-peer systems [17]. Partitioning of data is at a per-record granularity through consistent hashing [13]. The key of a record is hashed to a space that forms a ring and is statically partitioned. Thus the location of a data item can be computed without storing any explicit mapping of data to partitions. Replication is done at nodes that are neighbors of the node to which a key hashes to, a node which also acts as a master (although Dynamo is *multi-master*, as we will see later). Thus, Dynamo does not maintain a dynamic *system state* with consistency guarantees, a design different compared to PNUTS or Bigtable.

Even though not in the same vein as scalable data management systems, Sinfonia [1] is designed to provide an efficient platform for building distributed systems. Sinfonia [1] can be used to efficiently design and implement systems such as distributed file systems. The *system state* of the file system (e.g. the inodes) need to be maintained as well as manipulated in a distributed setting, and Sinfonia provides efficient means for guaranteeing consistency of these critical operations. Sinfonia provides the *minitransaction* abstraction, a light weight version of distributed transactions, supporting only a small set of operations. The idea is to use a protocol similar to Two Phase Commit (2PC) [10] for committing a transaction, and the actions of the transaction are piggy backed on the messages sent out during the first phase. The light weight nature of *minitransactions* allow the system to scale to hundreds of nodes, but the cost paid is a reduced set of operations.

Thus, when it comes to critical *system state*, the designers of these scalable data management systems rely on traditional mechanisms for ensuring consistency and fault-tolerance, and are willing to compromise scalability. But this choice does not hurt the system performance since this state is a very small fraction of the actual state (*application state* comprises the majority of the state). In addition, another important distinction of these systems is the number of nodes communicating to ensure the consistency of the *system state*. In the case of Chubby and YMB, a commit for a general set of operations is performed on a small set of participants (five and two respectively [3, 6]). On the other hand, Sinfonia supports limited transactional semantics and hence can scale to a larger number of nodes. This is in contrast to traditional distributed database systems, which tried to make both ends meet, i.e., providing strong consistency guarantees for both *system state* and *application state* over any number of nodes.

2.2 Application State

Distributed data management systems are designed to host large amounts of data for the applications which these systems aim to support. We refer to this application specific data as the *application state*. The *application state* is typically at least two to three orders of magnitude larger than the *system state*, and the consistency, scalability, and availability requirements vary based on the applications.

Data Model and its Implications. The distinguishing feature of the three main systems we consider in this paper is their simple data model. The primary abstraction is a table of items where each item is a *key-value* pair. The value can either be an uninterpreted string (as in Dynamo), or can have structure (as in PNUTS and Bigtable). Atomicity is supported at the granularity of a single item – i.e., *atomic read/write* and *atomic* *read-modify-write* are possible to only individual items and no guarantee is provided across objects. It is a common observation that many operations are restricted to a single entity, identifiable with a primary key. However, the disk centric nature of database systems forces relatively small row lengths. Consequently, in a traditional relational design, logical single entities have to be split into multiple rows in different tables. The novelty of these systems lie in doing away with these assumptions, thus allowing very large rows, and hence allowing the logical entity to be represented as a single physical entity. Therefore, *single-object* atomic access is sufficient for many applications, and transactional properties and the generality of traditional databases are considered an overkill. These systems exploit this simplicity to achieve high scalability.

Restricting data accesses to a *single-object* results in a considerably simpler design. It provides designers the flexibility of operating at a much finer granularity. In the presence of such restrictions, application level data manipulation is restricted to a single compute node boundary and thus obviates the need for multi-node coordination and synchronization using 2PC or Paxos, a design principle observed in [11]. As a result, modern systems can scale to billions of data tuples using horizontal partitioning. The logic behind such a design is that even though there can be potentially millions of requests, the requests are generally distributed throughout the data set, and all requests are limited to accesses to a single object or record. Essentially, these systems leverage *inter-request* parallelism in their workloads. Once data has been distributed on multiple hosts, the challenge becomes how to provide fault-tolerance and load distribution. Different systems achieve this using different techniques such as replication, dynamic partitioning, partition relocation and so on. In addition, the *single key* semantics of modern applications have allowed data to be less correlated, thereby allowing modern systems to tolerate the non-availability of certain portions of data. This is different from traditional distributed databases that considered data as a cohesive whole.

Single Object Operations and Consistency. Once operations are limited to a single key, providing single object consistency while ensuring scalability is tractable. If there is no object level replication, all requests for an object arrive at a single node that hosts the object. Even if the entire data set is partitioned across multiple hosts, the *single key* nature of requests makes them limited to a single node. The system can now provide operations such as *atomic reads*, *atomic writes*, and *atomic read-modify-write*.

Replication and Consistency. Most modern systems need to support *per-object* replication for high availability, and in some cases to improve the performance by distributing the load amongst the replicas. This complicates providing consistency guarantees, as updates to an object need to be propagated to the replicas as well. Different systems use different mechanisms to synchronize the replicas thereby providing different levels of consistency such as *eventual consistency* [7], *timeline consistency* [6] and so on.

Availability. Traditional distributed databases considered the entire data as a cohesive whole, and hence, non availability of a part of the data was deemed as non-availability of the entire systems. But the *single-object* semantics of the modern applications have allowed data to be less correlated. As a result, modern systems can tolerate non-availability of certain portions of data, while still providing reasonable service to the rest of the data. It must be noted that in traditional systems, the components were cohesively bound, and non-availability of a single component of the system resulted in the entire system becoming unavailable. On the other hand, modern systems are loosely coupled, and the non-availability of certain portions of the system might not affect other parts of the system. For example, if a partition is not available, then that does not affect the availability of the rest of the systems, since all operations are *single-object*. Thus, even though the system availability might be high, record level availability might be lower in the presence of failures.

2.3 The Systems

In Bigtable [5], a single node (referred to as *tablet server*) is assigned the responsibility for part of the table (known as a *tablet*) and performs all accesses to the records assigned to it. The *application state* is stored in the Google File System (GFS) [9] which provides the abstraction of a scalable, consistent, fault-tolerant storage for user data. There is no replication of user data inside Bigtable (all replication is handled at the GFS level), hence it is by default *single master*. Bigtable also supports *atomic read-modify-write* on *single keys*. Even though scans on a table are supported, they are best-effort without providing any consistency guarantees.

PNUTS [6] was developed with the goal of providing efficient read access to geographically distributed clients while providing serial *single-key* writes. PNUTS performs explicit replication to ensure fault-tolerance. The replicas are often geographically distributed, helping improve the performance of web applications attracting users from different parts of the world. As noted earlier in Section 2.1, Yahoo! Message Broker (YMB), in addition to maintaining the *system state*, also aids in providing application level guarantees by serializing all requests to the same key. PNUTS uses a *single master* per record and the master can only process updates by publishing to a single broker, as a result providing *single-object time line consistency* where updates on a record are applied in the same order to all the replicas [6]. Even though the system supports *multi-object* operations such as range queries, no consistency guarantees are provided. PNUTS allows the clients to specify their consistency requirements for reads: a read that does not need the guaranteed latest version can be satisfied from a local copy and hence has low latency, while reads with the desired level of freshness (including read from latest version) are also supported but might result in higher latency.

Dynamo [7] was designed to be a highly scalable key-value store that is highly available to reads but particularly for writes. This system is designed to make progress even in the presence of network partitions. The high write availability is achieved through an asynchronous replication mechanism which acknowledges the write as soon as a small number of replicas have written it. The write is eventually propagated to other replicas. To further increase availability, there is no statically assigned coordinator (thereby making this a *multi master* system), and thus, the *single-object* writes also do not have a serial history. In the presence of failures, high availability is achieved at the cost of lower consistency. Stated formally, Dynamo only guarantees *eventual consistency*, i.e. all updates will be eventually delivered to all replicas, but with no guaranteed order. In addition, Dynamo allows multiple divergent versions of the same record, and relies on application level reconciliation based on vector clocks.

2.4 Design Choices

So far in this section, our discussion focussed on the current design of major internet-scale systems. We anticipate more such key-value based systems will be built in the near future, perhaps as commodity platforms. In such cases, there are a few issues that need to be carefully considered and considerable deviation from the current solutions may be appropriate.

Structure of Value. Once the design decision to allow large values in key-value pairs is made, the structure imposed on these values becomes critical. At one extreme, one could treat the value as an opaque blob-like object, and applications are responsible for semantic interpretation for read/writes. This is in fact the approach taken in Dynamo. Presumably this suits the needs of Amazon's workload but is too limited for a generic data serving system. On the other hand, PNUTS provides a more traditional flat row like structure. Again, the row can be pretty large and frequent schema changes are allowed without compromising availability or performance. Also, rows may have many empty columns as is typical for web workloads. In Bigtable, the schema consists of column families and applications may use thousands of columns per family without altering the main schema, effectively turning the value into a 2D structure. Other choices that should be considered include restricting the number of columns, but allowing each column to contain lists or more complex structures. This issue needs to be studied further since the row design based on page size in no longer applicable, and hence more flexibility for novel structures is available.

System Consistency Mechanism. As discussed earlier, maintaining consistency of the *system state* is important for these systems. One obvious problem is to how to keep track of each partition assignment and consensus based solutions seem to be a good solution. But to add more features to the system, there is a need for reliable communication between partitions, e.g. supporting batched blind writes. PNUTS resorts to a reliable message delivery system for this purpose and hence is able to support some features such as key-remastering. This issue also needs further study since it might bring unnecessary complexity and performance problems unless carefully designed.

Storage Decoupling. Given that data is partitioned with a separate server responsible for operations on data within each partition, it is possible to store the data and run the server on the same machine. Clearly this avoids a level of indirection. However we think such close coupling between storage and servers is quite limiting since it makes features such as secondary indexes very hard to implement and involves much more data movement during partition splitting/merging. It would be better to follow a design where data is replicated at the physical level with a level of indirection from the server responsible for that partition. This is applicable even if there are geographically separated logical replicas since each such replica can maintain local physical replicas which would facilitate faster recovery by reducing the amount of data transfer across data centers. This design will need some mechanism to ensure that servers are located as close as possible to the actual data for efficiency while not being dependent on such tight coupling.

Exposing Replicas. For systems which aim for availability or at least limited availability in the face of network partitions, it makes sense to allow applications to be cognizant of the underlying replication mechanism. For systems, with limited availability, allowing the application to specify freshness requirements allows easy load spreading as well as limited availability. This is the case in both PNUTS and Dynamo. But in these setting we think designers should strongly consider adding support for multi-versioning, similar to that supported in Bigtable. These versions are created anyway as part of the process and the design decision is to store them or not. Note that old versions are immutable anyway and when storage servers are decoupled as discussed above, this allows analysis applications to efficiently pull data without interfering with the online system and also allowing time-travel analysis.

3 The Next Generation of Scalable Systems

In this section, we summarize the main design principles that allow *key value* stores to have good scalability and elasticity properties. We then discuss the shortcomings of such *key value* stores for modern and future applications, and lay the foundation for discussion of design principles of the next generation of scalable data management systems supporting complex applications while providing scalable and consistent access to data at a granularity large than *single keys*. The design of such stores is paramount for the success of data rich applications hosted in the cloud.

3.1 Scalable Design Principles

In this section, we highlight some of the design choices that have lent scalability to the *key value* stores:

– **Segregate System and Application State.** This is an important design decision that allows dealing differently with different components of the system, rather than viewing it as one cohesive whole. The *system state* is critical and needs stringent consistency guarantees, but is orders of magnitude smaller than the *application state*. On the other hand, the *application state* requires varying degrees of *consistency* and operational flexibility, and hence can use different means for ensuring these requirements.
– **Limit interactions to a Single physical machine.** Limiting operations to the confines of a single physical machines lends the system the ability to horizontally partition and balance the load as well as data. In addition, failure of certain components of the system does not affect the operation of the remaining components, and allows for graceful degradation in the presence of failure. Additionally, this also obviates distributed synchronization and the associated cost. This design principle has also been articulated in [11] and forms the basis for scalable design.
– **Limited distributed synchronization is practical.** Systems such as Sinfonia [1] and Chubby [3] (being used at the core of scalable systems such as Bigtable [5] and GFS [9]) that rely on distributed synchronization protocols for providing consistent data manipulation in a distributed system have demonstrated that distributed synchronization, if used in a prudent manner, can be used in a scalable data management system. The system designs should limit distributed synchronization to the minimum, but eliminating them altogether is not necessary for a scalable design.

The above mentioned design principles will form the basis for the next generation of scalable data stores.

3.2 Moving beyond Single Key semantics

A large class of current web-applications exhibit *single key* access patterns [7, 18], and this is an important reason for the design of scalable data management systems that guarantee *single key* atomic access. But a large number of present and future applications require scalable and consistent access to more than a single key. For example, let us consider the example of an online casino game. Multiple players can join a game instance, and the profiles of the participants in a game are linked together. Every profile has an associated balance, and the balance of all players must be updated consistently and atomically as the game proceeds. There can be possibly millions of similar independent game instances which need to be supported by the system. Additionally, the load characteristics of these applications can be hard to predict. Some of these applications might not be popular and hence have low load characteristics, while sudden popularity of these applications can result in a sudden huge increase in the load on the system [8, 12]. The cloud computing paradigm provides efficient means for providing computation for these systems, and for dealing with erratic load patterns. But since these applications cannot be supported by *key value* stores like Bigtable or Simple DB, they have to rely on traditional databases, and traditional database servers running on commodity machine instances in the cloud often become a scalability bottleneck. A similar scalability challenge is confronted by the movement of more and more web applications to the cloud. Since a majority of the web applications are designed to be driven by traditional database software, their migration to the cloud results in running the database servers on commodity hardware instead of premium enterprise database servers. Additionally, porting these applications to utilize *key value* stores is often not feasible due to various technical as well as logistic reasons. Therefore, modern applications in the cloud require a next generation data storage solution that can run efficiently on low cost commodity hardware, while being able to support high data access workload and provide consistency granularity and functionality at a higher granularity compared to *single key* access.

3.3 Concluding Remarks

Among the primary reasons for the success of the cloud computing paradigm for utility computing are *elasticity*, *pay as you go* model of payment, and use of commodity hardware in a large scale to exploit the economies of scale. Therefore, the continued success of the paradigm necessitates the design of a scalable and elastic system that can provide data management as a service. This system should efficiently and effectively run on commodity hardware, while using the *elasticity* of the cloud to deal with the erratic workloads of modern applications in the cloud, and provide varying degrees of consistency and availability guarantees as per the application requirements. The spectrum of data management systems has the scalable *key value* stores on one end, and flexible, transactional, but not so scalable database systems on the other end. Providing efficient data management to the wide variety of applications in the cloud requires bridging this gap with systems that can provide varying degrees of consistency and scalability. In this paper, our goal was to lay the foundations of the design of such a system for managing "clouded data".

References

1. Aguilera, M.K., Merchant, A., Shah, M., Veitch, A., Karamanolis, C.: Sinfonia: a new paradigm for building scalable distributed systems. In: SOSP. pp. 159–174 (2007)
2. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison Wesley, Reading, Massachusetts (1987)
3. Burrows, M.: The Chubby Lock Service for Loosely-Coupled Distributed Systems. In: OSDI. pp. 335–350 (2006)
4. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: PODC. pp. 398–407 (2007)
5. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. In: OSDI. pp. 205–218 (2006)
6. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. 1(2), 1277–1288 (2008)
7. 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: SOSP. pp. 205–220 (2007)
8. von Eicken, T.: Righscale Blog: Animoto's Facebook Scale-up. `http://blog.rightscale.com/2008/04/23/animoto-facebook-scale-up/` (April 2008)
9. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. In: SOSP. pp. 29–43 (2003)
10. Gray, J.: Notes on data base operating systems. In: Operating Systems, An Advanced Course. pp. 393–481. Springer-Verlag, London, UK (1978)
11. Helland, P.: Life beyond distributed transactions: an apostate's opinion. In: CIDR. pp. 132–141 (2007)
12. Hirsch, A.: Cool Facebook Application Game – Scrabulous – Facebook's Scrabble. `http://www.makeuseof.com/tag/best-facebook-application-game-scrabulous-facebooks-scrabble/` (2007)
13. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: STOC. pp. 654–663 (1997)
14. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (1998)
15. Lindsay, B.G., Haas, L.M., Mohan, C., Wilms, P.F., Yost, R.A.: Computation and communication in R*: a distributed database manager. ACM Trans. Comput. Syst. 2(1), 24–38 (1984)
16. Rothnie Jr., J.B., Bernstein, P.A., Fox, S., Goodman, N., Hammer, M., Landers, T.A., Reeve, C.L., Shipman, D.W., Wong, E.: Introduction to a System for Distributed Databases (SDD-1). ACM Trans. Database Syst. 5(1), 1–17 (1980)
17. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM. pp. 149–160 (2001)
18. Vogels, W.: Data access patterns in the amazon.com technology platform. In: VLDB. pp. 1–1. VLDB Endowment (2007)
19. Weikum, G., Vossen, G.: Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. Morgan Kaufmann Publishers Inc. (2001)