UCSB Computer Science Technical Report 2008-18, November 2008.

Clouded Data: Comprehending Scalable Data Management Systems

Sudipto Das Shyam Antony Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara
{sudipto, shyam, agrawal, amr}@cs.ucsb.edu

ABSTRACT

Managing petabytes of data for millions of users has been a challenge for big internet based enterprises such as Google, Yahoo!, and Amazon. Even though database management systems have a long history of managing enterprise level data and information, they are deemed to be unsuitable in this context. This resulted in an architectural redesign of data management systems with an eye towards the requirements of high scalability, high availability, and low latency while providing weaker consistency and lower application generality. In this paper, we try to comprehend what is fundamentally different in the internet-scale applications that allowed these data management systems to achieve orders of magnitude higher levels of scalability compared to traditional databases. With an understanding of these modern systems, we also make an attempt to predict future application requirements and raise two fundamental questions: where do we really stand in terms of scalable data management, and how far are we from providing scalable data management as a service, just as computing is provided as a service in large scale infrastructures?

1. INTRODUCTION

Recent years have seen the emergence of large scala data management systems such Google's Bigtable[1], PNUTS [7] from Yahoo!, Amazon's Dynamo [8] and some 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 such as flash crowds, offer variants of a key-value data model highly amenable to partitioning, run on cluster computing architectures; thus staking claims to some of the territories that used to be occupied by database systems. In this paper, we analyze some of the salient features of these systems and their current and future impact on large scale data management systems.

Historically, distributed database systems were the first generic solution that dealt with data split across multiple machines. These systems continued to maintain the semantics of their centralized counterparts by ensuring global serializability. This made good short term sense since applications could be ported with no effort, but was not sustainable beyond a few machines due to the crippling effect on performance cause by partial failures and synchronization overhead. This point is very well recognized and installations with significant amounts of data with database backends rarely use such a setup for online transactions. Data is typically partitioned with individual online transactions restricted to one or few partitions. Analysis queries which need a global view are run offline. Thus the frequently cited problem of distributed transactions, such as large coordinated commits, while true, are rarely used in practical installations. But database engines continue to retain various assumptions forced by their goal of closely adhering to the design of their centralized counterparts.

The data model supported by the modern systems is a collection of key-value pairs with read and write operations on individual items. While the systems vary in the structure imposed on the value, it can be readily seen that these semantics naturally permit data partitioning. Furthermore, for operations in the key-value model, each partition can be assigned to a separate database server and no tight coupling is needed among these servers. Note that this eliminates the need for any coordinated commit. In other words, this setup is among the best case scenarios for distributed database systems. Nonetheless, distributed databases have been found inadequate, resulting in the emergence of systems like Bigtable, Dynamo and PNUTS. We believe that the core problem lies in dealing with workload dynamicity and repartitioning. In typical partitioned databases, repartitioning is a relatively infrequent operation, usually done with manual intervention and auto tuning tools. The implicit assumption is that the server managing each shard is provided with enough resources to meet the performance and availability goals of the system. In a system with petabytes of data, this would result in an enormous amount of overprovisioning when the load distribution among the data items is not predictable. The solution is obvious – there is a critical need for rapid dynamic repartitioning while simultaneously sustaining performance and availability, and this is where a traditional partitioned database based setup would collapse.

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. The resource utilization requirement demands that the system be highly dynamic. In Section 2, we discuss the salient features of three major systems from Google, Yahoo!, and Amazon, that meet these goals. The design of these systems is interesting not only from the point of view of what concepts they use (e.g. consensus protocol) but also what concepts they eschew (e.g. distributed two phase locking, disk page centric size limitations etc.). Some aspects reflect individual

[1]Even though Google's solution is a combination of Chubby [3], Google File System [9], and Bigtable [5], for simplicity, we refer to it as Bigtable.

choices, while other aspects are inherent to meet the data management challenges outlined above. Careful analysis of the latter is necessary to facilitate future work.

Note that our discussion above is entirely driven by the assumption that the workload consists of simple operations on individual items. In situations where this assumption does not hold, many applications have to rely on traditional databases. For example, in online stores, shopping cart and session management can be handled with a key-value system but the checkout process may have to fall back on a traditional database to facilitate the various expressive queries needed during order processing. Furthermore, these operations cannot be simply pushed offline since activities such as order or payment processing demands an online setup. In Section 3, we address the question of whether this hybrid mechanism of using scalable systems for simple operations and relying on databases for everything else (with comparatively smaller workloads) can last. We analyze some potential applications of the future as well as the repercussions of the sheer increase in scale of the checkout-style workload. We predict that the need for scalable systems enhanced to support carefully chosen operations beyond simple read/write on individual items will arise. Once such systems are available as commodity systems that can be purchased, the utility of databases engines in their current form becomes questionable. Of course, building such commodity systems will involve tremendous engineering as well as research challenges. What better community can one think of to meet these challenges but a rejuvenated database community!

2. ANALYSIS

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 which the distributed system stores. 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.

Even though not all systems under consideration provide a clear demarcation between the two types of state, each of them does employ different techniques for maintaining the two different types of state. Such a demarcation is necessary to improve our understanding of the systems and demonstrates how the systems deal with these states differently, since the correctness and consistency requirements of the two are very different.

2.1 System State

The meta data and information required to correctly manage the distributed system is referred to as the *system state.* In a distributed data management system, when data is partitioned to achieve scalability, and replicated to achieve fault-tolerance, the system must have a consistent view of the mappings of partitions to nodes, and that of a partition to its replicas. If the application data is *dynamically* partitioned to provide better load-balancing and resource utilization or replicas are relocated to deal with failures, the corresponding *system state* needs to maintain the *correct* and *consistent* view of the mappings. In addition, 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 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 based consensus protocol [4]. All the replicas in Chubby are equivalent and store a consistent view of the system. The Paxos protocol [12] at the core 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 as it typically has few replicas (five) and a single master. Thus if a system makes too many calls to Chubby, it might become a bottleneck. 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 its performance.

In PNUTS [7], 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 created gradually [7]. 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 [8] uses an approach similar to peer-to-peer systems [15]. Partitioning of data is at a per-record granularity through consistent hashing [10]. 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 the master (although Dynamo is *multi-master*, as we will see later). Thus, Dynamo does not maintain a dynamic *system state* with consistency guarantees on that, a design different compared to PNUTS or Bigtable. Minimal membership maintenance is done through administrator intervention. But lesser state comes at a cost: there can potentially be an issue with over provisioning (given the static partitioning of hash space), dealing with which is a challenge for Dynamo.

Even though not in the same vein as scalable data management systems, we consider Sinfonia [1], a system from HP Laboratories, since it 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 filesystem (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 a concept of *minitransactions*, 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) for committing a transaction, and the actions of the transaction are piggy backed on the messages sent out during the first phase. Thus, only a small set of operations is supported in Sinfonia. 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. Since no data management system in the same vein as Bigtable or PNUTS is built on Sinfonia, we limit our discussion of Sinfonia more as a representative system to maintain *system state*.

Thus, when it comes to critical *system state*, the system designers 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 consistency of the *system state*. In the case of Chubby and YMB, a commit for a general set of operations is carried on a small set of participants (five and two respectively [3, 7]). On the other hand, Sinfonia provides limited transactional support 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 huge 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 an order of magnitude larger than the *system state*, and the consistency, scalability and availability requirements vary based on the applications.

2.2.1 Data Model

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* is 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 (or blobs). Hence, even for single entities, there is a need to normalize data to span multiple rows in different tables. The novelty of these systems lie in doing away with these assumptions, thus allowing very large rows. In addition, *single-object* atomic access is sufficient for many applications, and thus transactional properties and the generality of traditional databases are an overkill. These systems exploit this simplicity to achieve high scalability.

Restricting data accesses to a *single-object* results in considerably simpler design for providing data management functionality. Now, instead of viewing the entire table (or database) as a single entity, it allows the 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. As a result, modern systems can scale to millions 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 now is in providing fault-tolerance and load distribution. Different systems achieve this using different techniques such as replication, dynamic partitioning, partition relocation and so on. In addition, *single-object* semantics of the modern applications have allowed data to be less correlated, thereby allowing modern systems to tolerate 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 object, providing single object consistency while ensuring scalability is tractable. If there is no object level replication, then consistency can be achieved with more ease. All requests for an object arrive at a single node that hosts the object, and even if the entire data set is partitioned across multiple hosts, the *single-object* 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*. *Atomic read-modify-writes* can also be used by the application to ensure some form of isolation.

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 increases the complexity for providing consistency guarantees, as updates to an object need to be propagated to the replicas as well. Furthermore, traditional techniques for recovery [17] are needed, but are simpler due to absence of undo operations. Different systems use different mechanisms to synchronize the replicas thereby providing different levels of consistency such as *eventual consistency*, *timeline consistency* and so on. In addition, the systems can also be classified into two classes based on the number of nodes that can concurrently write to a single object: *single master* systems where only one replica has the privilege of processing updates, and *multi master* systems where multiple nodes can concurrently process updates on the same object.

2.2.2 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 GFS which is treated as a scalable, consistent, fault-tolerant storage for user data, while providing scalable access to the data. There is no replication of user data inside Bigtable (all replication is handled at the storage level i.e., GFS, with some coordination from Bigtable for logging and recovery), hence it is by default *single master*. Bigtable also supports *atomic read-modify-write* on *single records*. Even though scans on a table are supported, they are best-effort without providing any consistency guarantees.

PNUTS [7] is developed with the goal to provide efficient read access to geographically distributed clients while providing serial *single-key* writes. PNUTS explicitly performs replication to ensure fault-tolerance. The replicas are often geographically distributed, helping improve 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 [7]. Even though the system supports *multi-object* operations such as range queries, no consistency guarantees are provided. PNUTS allows the clients to specify the 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 desired level of freshness (including read from latest version) are also supported but might result in higher latency.

Dynamo [8] 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 returns success to the application as soon as a small number of replicas have replied. 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 [11].

2.3 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.

Support for Scripting. The second design decision is determining the operations that can be supported on the server side. With relation-like rows, one could push various filtering, such as projection or selection, to the system, before a query is answered, as supported by PNUTS. On the other hand, Bigtable allows clients to ship their own script which is then executed inside the system, i.e, a client ships a Sawzall script (similar to prepared statements in DBMS) and the system executes it with the object named in the request as input. This kind of scripting provides the potential for executing relatively complex operations without too much network traffic. The ability to add such support, leads us to speculate that an efficient design, in which the object structure resembles a row with column families and each family storing common data structures such as short lists or maps that can be manipulated or filtered by client-provided scripts.

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.

3. DATA MANAGEMENT TRENDS

In this section, we consider some potential trends in future applications, and the resultant challenges that will have to be faced by scalable systems.

Hybrid Systems. Anecdotal evidence suggests that many medium scale data management companies use a hybrid of a database engine and a custom solution. The custom solutions handle scalability intensive events whereas the database back-end handles reliability, persistence, etc. This is probably the only reasonable way at the current time since the scalable systems discussed in the paper are "in-house" and are not yet available commodity products. We predict that such commodity systems will be available soon and investing in such solutions may provide a way to the scalable maintenance of data without significant development effort.

Surprisingly even companies with "in-house" scalable data management solutions continue to use a hybrid mixture of such systems and database systems for their varied application tasks. Database systems are used in managing more critical data such as order processing or payment management while scalable systems are used for applications such as web indexing or shopping cart management. From one point of view, such hybrid solutions are very welcome since they avoid the often repeated fallacy of shoe horning all data into one complex system. However, it is important to analyze the nature of the workloads being handled by high scalable systems with simple data models, and those handled by traditional databases. If the analysis reveals that this hybrid data path is sufficient both in the short term as well as the long term, more effort should be spend in tuning these systems rather than making significant architectural changes. We underscore that database systems provide excellent features such as well defined recovery, rock solid stability, declarative interfaces with high expressive power, and efficient query optimization [13].

However, a careful analysis from a long term point of view should be made before allowing databases to reside in various important data paths. Even though the current scale of the applications may not be affected by such hybrid solutions, it is also necessary to determine if that will continue to hold in the long run. We are aware of one installation in which a database was used in data exhibiting high update rate (such as streaming click events) and after a while became practically unusable and various ad hoc solutions had to be forced into the data path.

We think that systems will soon start emerging that occupy the middle ground more efficiently, i.e., systems which inherit features from both traditional and current scalable systems. This trend can also be observed in some recent research [16]. Not just one big system, but various systems occupying different places in the consistency, availability, and scalability spectrum, with well defined interfaces allowing them to be purchased as separate commodities and composable to create efficient data paths.

Future Application Trends: The basis of the systems discussed in this paper is that atomic access to single objects is sufficient for many applications. This is indeed true with the current generation of web scale applications – but will that assumption continue to hold in the future? Clearly atomic access to a large amount of data is unlikely to be supported. But we can think of cases in which a small number of objects will need some kind of a synchronized access for a short period of time. We provide a couple of examples where this trend is likely to show up and how systems may evolve to handle such situations.

With the growing popularity of Web 2.0 application, the concept of collaborations have been put forward. Consider the case of collaborative authoring (such as Wikipedia or Google Docs) where multiple users are concurrently accessing a single document with the intent to collaboratively author the document. If we consider each document to be one problem instance, there can be potentially millions of such problem instances. But it must be noted that in most of the cases, the number of users collaborating in a problem instance is small. Currently conflict resolution in these systems are rather ad hoc. For example, in the context of Google Docs, if there are concurrent modifications of the same document, the application tries to merge the modifications. This seems to work well in many cases, but occasionally it does garble the documents when two users are concurrently modifying the same text. Similarly, GMail admits concurrent work on the same draft email and performs application level conflict resolution, which is to create two distinct drafts of the same email. Existing systems are thus providing acceptable but ad hoc support for collaboration, and it remains to be seen if such application level coordination without support from the data management service will be sufficient with more complex operations and heavier workloads.

Another interesting example is the increasing popularity of online gaming, especially online multi-player games. With financial incentives involved, this can grow into a lucrative industry potentially attracting hundreds of thousands of users. This is evident in a recent news article about a free version of the Scrabble game becoming heavily popular in the Facebook social networking website [14]. Again, such applications are similar to the collaborative authoring case – a small number of users in the same instance, and potentially millions of such instances. The present design paradigm (*single-object* accesses) shows that a scalable design can be achieved while providing consistency guarantees, as long as accesses are limited to a single node. So, a simple design would require all users associated with a single game instance to map onto the same node (which might be a logical node), and provide consistency guarantees on operations restricted to this single node. But is this enough, when users would like to join or leave ongoing games? Let us take an example of an online casino kind of application where a single user, with a finite amount of money in his cart, can be concurrently associated to multiple game instances where he is investing money in different games from the cart. He can concurrently lose as well as win money, and while these transactions are being processed, he is also investing money on other games he might be interested in. Evidently, this is a hard problem to tackle in a scalable system. But then again, the question we need to consider is whether these applications need high scalability or current hybrid solutions are sufficient?

Infrastructural Trends. In addition to the new applications, there is a new infrastructural trend as well [6]. Popularly known as *Cloud Computing* or *Utility Computing*, it is the paradigm of providing computation and storage as a service. Companies such as Amazon have come up with models such as *Elastic Compute Cloud* (EC2) and *Simple Storage Service* (S3) where the users need not be aware of the exact details of which node is performing the desired computation, or which node is actually storing the data. The users pay per use, and hence need not worry about over-provisioning for peak load, or maintaining the systems to deal with failures. Once complexities are abstracted, the application developer can concentrate on the development of the application logic.

Even though most of the applications being hosted in the *clouds* are generally computationally intensive data analysis problems, it is predicted that enterprises such as financial institutions might outsource their business logic to the cloud. In such a model of *utility computing*, it would be possible to complement computing and raw storage with a data management system which is provided as a service [2]. Stated in other words, can we have a system which provides data management as a service, scale to millions of users if the need arises, or provide consistency guarantees on *multi-object* operations on a small set. For example, an application developer designing an online shopping mall can use the data management service to maintain shopping carts at the scale of millions (while not requiring high consistency and durability), and use the same service for order processing in the scale of hundreds of thousands (requiring high consistency, durability and fault-tolerance). The question we need to answer is: do our present day systems live up to these requirements?

4. CONCLUSION

In this paper, we analyze how the design of modern data management systems has allowed them to scale to millions of users and petabytes of data, while providing high availability, and reasonable consistency which is sufficient for the applications they aim to serve. It is evident that these systems have been designed with a complete understanding of the characteristics of the applications which they aim to serve. This realization of the application requirements helped decide on the primary and secondary design goals and has led to the widespread success of these systems. With the lessons learned from these systems, we try to analyze the present state of the art in data management solutions and discuss whether these solutions would be enough to handle the application requirements of the future. Considering the new generation of collaborative applications, and the new infrastructural paradigms like cloud computing, the question we want to ponder is how far are we from supporting scalable modern applications and providing data management as a service just as enterprises provide computing and storage as a service?

5. REFERENCES

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, pages 159–174, 2007.

[2] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *SIGMOD*, pages 251–264, 2008.

[3] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI*, pages 335–350, 2006.

[4] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, pages 398–407, 2007.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006.

[6] The claremont report on database research. `http://db.cs.berkeley.edu/claremont/claremontreport08.pdf`, May 2008.

[7] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *VLDB*, pages 1277–1288, 2008.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[10] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

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

[13] J. M.Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[14] Facebook scrabble application hit by legal woes. `http://www.dailyprincetonian.com/archives/2008/01/21/news/19889.shtml`.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[16] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[17] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., 2001.