Building a Database in the Cloud

Matthias Brantner[◇] Daniela Florescu[†] David Graf[◇]
Donald Kossmann[◇♣] Tim Kraska[♣]

28msec Inc.[◇]
{firstname.lastname}@28msec.com

Oracle[†]
dana.florescu@oracle.com

Systems Group, ETH Zurich[♣]
{firstname.lastname}@inf.ethz.ch

Abstract

There has been a great deal of hype about cloud computing. Cloud computing promises infinite scalability and high availability at low cost. Currently, Amazon Web Services is the most popular suite of cloud computing services, but other vendors such as Adobe, Google, and Microsoft are also appearing on the market place. The purpose of this paper is to demonstrate the opportunities and limitations of using cloud computing as an infrastructure for general-purpose Web-based database applications. The paper studies alternative consistency protocols in order to build database services on top cloud storage services. Furthermore, the paper studies alternative client-server and indexing architectures. Both the performance (response time) and cost trade-offs are studied.

1 Introduction

The Web has made it easy to provide and consume content of any form. Building a Web page, starting a blog, and making both searchable for the public have become a commodity. Arguably, the next wave is to make it easy to provide *services* on the Web. Services such as Flickr, YouTube, SecondLife, or Myspace lead the way. The ultimate goal, however, is to make it easy for everybody to provide such services — not just the big guys. Unfortunately, this is not yet possible.

Clearly, there are non-technical issues that make it difficult to start a new service on the Web. Having the right business idea and effective marketing are at least as difficult on the Web as in the real world. There are, however, also technical difficulties. One of the most crucial problems is the cost to operate a service on the Web, ideally with 24×7 availability and acceptable latency. To run a large-scale service like YouTube, several data centers all around the world are needed. But, even running a small service with a few friends involves a hosted server and a database which both need to be administrated. Running a service becomes particularly challenging and expensive if the service is successful: Success on the Web can kill! In order to overcome these issues, *utility computing* (aka cloud computing) has been proposed as a new way to operate services on the Internet [27].

The goal of cloud computing is to provide the basic ingredients such as storage, CPUs, and network bandwidth as a commodity by specialized utility providers at low unit cost. Users of these utility services do not need to worry about scalability because the storage provided is virtually infinite. In addition, utility computing provides full availability; that is, users can read and write data at any time without ever being blocked; the response times are (virtually) constant and do not depend on the number of concurrent users, the size of the database, or any other system parameter. Furthermore, users do not need to worry about backups. If components fail, it is the responsibility of the utility provider to replace them and make the data available using replicas in the meantime. Another important reason to build new services based on utility computing is that service providers only pay for what they get; i.e., pay by use. No investments are needed upfront and the cost grows linearly and predictably with the usage. Depending on the business model, it is even possible for the service provider to pass the cost for storage, computing, and networking to the end customers because the utility provider meters the usage.

The most prominent utility service today is AWS (Amazon Web Services) with its simple storage service, S3, and elastic computing cloud, EC2, as the most popular representatives. Adobe Share and Microsoft's SQL Server Data Services (SSDS) are additional examples for such commercial utility services. Furthermore, Google has recently provided a suite of services towards the same goal including MegaStore and Google AppEngine. Today, AWS and in particular S3 are most successful for multimedia objects. Smugmug (www.smugmug.com), for instance, is implemented on top of S3 [8]. Furthermore, S3 is popular as a backup device. For instance, there already exist products to backup data from a MySQL database to S3 [26]. As Google's AppEngine demonstrates, utility computing is quickly evolving as a candidate platform in order to develop general-purpose Web-based applications. The goal of this paper is to explore alternative options and architectures in order to leverage utility computing services for such applications.

The first contribution of this paper is to define an (abstract) API for utility services. This API can be implemented effectively using AWS and other utility service offerings. As all these offerings vary significantly today and there do not seem to be any standardization activities, such a reference API is important as a basis for all further developments in this area.

The second contribution of this paper is the development of alternative protocols in order to ensure the consistency of data stored using utility services such as S3. Currently, S3 only provides a low level of consistency, referred to as *eventual consistency* and has no support to coordinate and synchronize parallel access to the same data. As a result, this paper presents a number of protocols in order to orchestrate concurrent updates to S3. To this end, this paper adapts the protocols presented in [7] to our reference API for utility services. Furthermore, this paper presents and evaluates two additional protocols, locking and snapshot isolation, and their implementation on top of that reference API. This paper also contains some sketches in order to discuss the correctness of the protocols.

The third contribution of this paper is to study alternative client-server architectures in order to build Web-based applications on top of utility services. One big question is whether the application logic should run on the client or should run on a server provided by the utility services (e.g., EC2). The second question concerns the implementation and use of indexes; for instance, is it beneficial to push query processing into the cloud and use services such as SimpleDB or SSDS as opposed to implementing indexing on the client-side. Obviously, the experimental results depend strongly on the current prices and utility service offerings; nevertheless, there are fundamental trade-offs which are not likely to change.

Finally, a fourth contribution of this paper is to discuss alternative ways to implement the reference API for utility services based on AWS. Ideally, a utility service provide would directly implement the API, but as mentioned above, such standardization is not likely to happen in the foreseeable future.

Again, we would like to emphasis that we are aware that the results presented in this paper capture merely a snapshot (October 2008) of the current state-of-the-art in utility computing. Given the success of AWS and the recent offerings by Google, it is likely that there the utility computing market and its offerings will evolve quickly. Nevertheless, we believe that the techniques and trade-offs discussed in this paper are fundamental and are going to continue to stay relevant for a longer period of time; in a similar way as the performance results of a paper written twenty years ago are still relevant today even though the underlying hardware has changed significantly over the last 20 years.

The remainder of this paper is organized as follows: Section 2.2 describes AWS (i.e., S3, SQS, EC2 and SimpleDB) and defines an (abstract) API for utility services which forms the basis for the remainder of the paper. Even though there are many utility service offerings today, AWS is described in detail because AWS was used for the experiments and because AWS has the most complete and mature suite of services. Section 3 presents the proposed architecture to build Web-based database applications on top of utility services. Sections 4 and 5 present the protocols to implement reads and writes on top of S3 at different levels of consistency. Section 6 gives implementation details of these protocols using AWS. Section 7 summarizes the results of experiments conducted using the TPC-W benchmark. Section 8 discusses related work. Section 9 contains conclusions and suggests possible avenues for future work.

2 Cloud Computing

Cloud computing allows users to access technology-enabled services from the Internet (”in the cloud”) without owning the technology infrastructure that supports them. The range of services today varies from basic infrastructure services, e.g. providing storage space, to rather specialized services for payment, identity authentication and others. This section reiterates the advantages of using cloud services. Furthermore, this section provides an overview of the services offered today by using Amazon Web Services as an example. Finally, this section presents a generalized API that is used as a basis for the remainder of this paper and in order to enable to efficiently build database applications in the cloud.

2.1 Why Cloud Computing?

So far there is no standard definition of cloud or utility computing. Some people refer to utility computing as pricing model and to cloud computing as an architecture. It is even possible to encounter opposite definitions wherein utility computing is seen as the infrastructure service and cloud computing refers to the commercialization of those. This paper uses both terms interchangeably in order to refer both to the pricing model and the technical infrastructure.

There are many types of cloud services. At the core, cloud computing provides hardware resources which can be rented and consumed in a fine-grained manner; i.e., CPU cycles and storage space. The business model is *pay-per-use*. That is, users need not make an investment upfront and pay for the hardware resources they consume. On top of these basic services, cloud computing providers offer other, higher-level services such as authentification services, queues, or payment services. Again, no investment for software licenses need to be made by the user; instead the usage of the services is metered and billed accordingly.

The success of cloud computing is based on an economy of scale. The cloud computing provider can offer services to millions of users at a lower price than users can accomodate these services themselves. In addition to the price, the quality of the service is a major incentive to use cloud services. Specifically, the cloud provider is responsible for guaranteeing high availability and reliability. Users expect almost 100% availibity and (virtually) constant response time independent of the number of concurrent users. Furthermore, users of cloud storage services do not need to worry about backups. If components fail, it is the responsibility of the provider to replace them and make the data available using replicas in the meantime. Furthermore, users of cloud services do not need to worry about scalability because the offer is virtually infinite. The IT cost grows linearly with the business, rather than in a step-wise function as for traditional computing in which businesses need to buy hardware in the granularity of machines (rather than CPU cycles). In summary, the goal of cloud computing is to provide more for less.

While the advantages of cloud computing are compelling, there are also important disadvantages. Although some services are quite powerful, they might not directly satisfy the requirements for a certain task. A specific concern for operators of Web-based database applications is that storage providers usually give only relaxed consistency guarantees. Although this is sufficient for certain scenarios, it does not suit every application. If an application requires higher guarantees, it must be build these on top, as part of the application or by using other services. Second, infrastructure hosted in the cloud is often slower because of network latency. E.g. storage services are orders of magnitudes slower than locally attached disks. Furthermore, similar services might not be homogeneously priced and even for the same service prices may be discontinuous in usage due to mass discounts. Both issues, latency and prices, introduce a complete new set of trade-offs which have to be made for building applications inside the cloud. The purpose of this paper is study these trade-offs.

2.2 AWS

More and more providers appear on the cloud computing market place (reference cloud overview). The most prominent provider today is Amazon with its Amazon Web Services (AWS). Amazon not only offers the most complete stack of services, but makes it especially easy to integrate different services. Therefore, this section describes AWS in more detail as the most prominent representative for the wide range of today's offerings. The focus of this short survey is on infrastructure services (e.g., storage and CPU cycles) because those form the foundation for building Web-based applications. More specialized services like Amazon's payment service are beyond the scope of this work.

2.2.1 Storage Service

Amazon's storage service is named S3 standing for **S**imple **S**torage **S**ystem. Conceptually, it is an infinite store for objects of variable size (minimum 1 Byte, maximum 5 GB). An object is a byte container which is identified by a URI. Clients can read and update S3 objects remotely using a SOAP- or REST-based interface, e.g., *get(uri)* returns an object and *put(uri, bytestream)* writes a new version of the object. A special *getIfModifiedSince(uri, timestamp)* method allows retrieving the new version of an object only if the object has changed since the specified timestamp. Furthermore, user-defined metadata (maximum 4 KB) can be associated to an object and can be read and updated independent of the rest of the object.

In S3, each object is associated to a bucket. That is, when a user creates a new object, the user specifies into which bucket the new object should be placed. S3 provides several ways to *scan* through objects of a bucket. For instance, a user can retrieve all objects of a bucket or only those objects whose URIs match a specified prefix. Furthermore, the bucket can be the unit of security: Users can grant read and write authorization to other users for entire buckets. Alternatively, access privileges can be given on individual objects.

S3 is not for free. It costs USD 0.15 to store 1 GB of data for one month if the data is stored in the USA. Storing 1 GB in a data-center in Europe costs USD 0.18. In comparison, a 160 GB disk drive from Seagate costs USD 70 today. Assuming a two-year life time of a disk drive, the cost is about USD 0.02 per GB and month (power consumption is not included). Given that disk drives are never operated at 100 percent capacity and considering mirroring, the storage cost of S3 is in the same ballpark as that for regular disk drives. Therefore, using S3 as a backup device is a no-brainer. Users, however, need to be more careful to use S3 for live data because every read and write access to S3 comes with an additional cost of USD 0.01 (0.012 for Europe) per 10,000 *get* requests, USD 0.01 (0.012 for Europe) per 1,000 *put* requests, and USD 0.10 to USD 0.17 per GB of network bandwidth consumed (the exact rate depends on the total monthly volume of a user). For this reason, services like smugmug use S3 as a persistent store, yet operate their own servers in order to cache the data and avoid interacting with S3 as much as possible [8].

Another reason to make aggressive use of caching is latency. Table 1 shows the response time of *get* requests and the overall bandwidth of *get* requests, depending on the *page size* (defined below). These experiments were executed using a Mac (2.16 GHz Intel Core Duo with 2 GB of RAM) connected to the Internet and S3 via a fast Internet connection. (The results of a more comprehensive performance study of S3 are reported in [17].) The results in Table 1 support the need for aggressive caching of S3 data; reading data from S3 takes at least 100 msecs (Column 2 of Table 1) which is two to three orders of magnitudes longer than reading data from a local disk. Writing data to S3 (not shown in Table 1) takes not much more time as reading data. While latency is an issue, S3 is clearly superior to ordinary disk drives in terms of *throughput*: Virtually, an infinite number of clients can use S3 concurrently and the response times shown in Table 1 are practically independent of the number of concurrent clients.

Column 3 of Table 1 shows the bandwidth a European client gets when reading data from S3 (S3 servers located in the USA). It becomes clear that an acceptable bandwidth can only be achieved if data are read in relatively large chunks of 100 KB or more. Therefore, small objects should be clustered into *pages* and a whole page of small objects should be the unit of transfer. The same technique to cluster *records* into *pages* on disk is common practice in all state-of-the-art database systems [19] and we

Page Size [KB]	*Resp. Time [secs]*	*Bandwidth [KB/secs]*
10	0.14	71.4
100	0.45	222.2
1,000	2.87	348.4

Table 1: Resp. Time, Bandwidth of S3, Vary Page Size
External Client in Europe

adopt this technique for this study.

Amazon has not published details on the implementation of S3 and it does not give any guarantees. Taking [13] as a reference for Amazon's design principles (even though [13] describes a different system), it seems that S3 replicates all data at several data centers. Each replica can be read and updated at any time and updates are propagated to replicas asynchronously. If a data center fails, the data can nevertheless be read and updated using a replica at a different data center; reconciliation happens later on a *last update wins* basis. This approach guarantees full read and write availability which is a crucial property for most Web-based applications: No client is ever blocked by system failures or other concurrent clients. Furthermore, this approach guarantees persistence; that is, the result of an update can only be undone by another update. Additional guarantees are not assumed in this paper. The purpose of this work is to show how such additional consistency guarantees can be provided on top of S3. In order to exploit S3's (apparent) last update wins policy, all protocols make sure that there is sufficient time between two updates to the same S3 object (i.e., several seconds, Section 4).

2.2.2 Servers

Amazon's offering for renting computing hardware is named **E**lastic **C**omputing **C**loud, short EC2. Technically, the client gets a virtual machine which is hosted on one of the Amazon servers. Like S3, EC2 is not for free. The cost varies from USD 0.10 to 0.80 per hour depending on the configuration. For example, the cheapest machine costs USD 0.10, the second cheapest machine costs USD 0.2 per hour, but this machine comes with 5 times as high computing power as the cheapest machine. Independent of the configuration, the fee must be paid for the time the machine is leased regardless of how heavily the machine is used.

One interesting aspect of EC2 is that the network bandwidth from an EC2 machine to other Amazon Services is free. Nevertheless, the "per request" charges are applicable in any case. For example, if a user makes a request from an EC2 machine to S3, then the user must pay USD 0.01 per 10,000 *get* requests. As a result, it seems advantageous to have large block sizes for transfer between EC2 and S3.

EC2 is built on top of XEN an open source hypervisor [33]. It allows to operate a variety of linux/unix images. To the best of our knowledge, EC2 does not support any kind of virtual machine migrations from one EC2 machine to another EC2 machine. As a result, it is difficult to implement fault-tolerance using EC2. If a machine goes down, the state is lost unless it was stored somewhere else, e.g., on S3. From a performance perspective, it is attractive to run applications on EC2 if the data is hosted on S3 because the latency of the communication between EC2 and S3 is much faster than between an external client and EC2. Table 2 shows the response times of EC2/S3 communication, again varying the page size. Comparing Table 2 with 1, the performance difference

Page Size [KB]	*Resp. Time [secs]*	*Bandwidth [KB/secs]*
10	0.035	333.3
100	0.046	2173.9
1,000	0.128	7812.5

Table 2: Resp. Time, Bandwidth of S3, Vary Page Size
EC2 Client

Operation	*Time [secs]*
send	0.31
receive	0.16
delete	0.16

Table 3: Response Times of SQS

becomes apparent. Nevertheless, S3 is still much slower than a local disk, even when used from an EC2 machine.

In addition to the reduction of latency, EC2 is also attractive to implement missing infrastructure services such as a global transaction counter. The use of EC2 for this purpose is discussed in more detail in Section 6.

2.2.3 Queues

Amazon also provides a queue service named **S**imple **Q**ueueing **S**ystem short SQS. SQS allows users to manage a (virtually) infinite number of queues with (virtually) infinite capacity. Each queue is referenced by a URI and supports sending and receiving messages via a HTTP or REST-based interface. The maximum size of a message is 8 KB. Any bytestream can be put into a message; there is no pre-defined schema. Each message is identified by a unique id. Messages can only be deleted by a client if the client received the message before. Another important property of SQS is, that it supports the locking of a message for up to 2 hours. Amazon changed the prices for SQS recently: The prices as of (2008-10-01) are USD 0.01 per 10,000 requests. Furthermore, the network bandwidth costs at least USD 0.10 per GB of data transferred, unless the requests originated from an EC2 machine. As for S3, the cost for the consumed network bandwidth decreases, the more data is transferred. USD 0.10 per GB is the minimum for heavy users.

Table 3 lists the round trip times of the most critical SQS operations used in this study; i.e., the operations that impact the performance of a Web-based application built using SQS. Each call to SQS either returns a result (e.g., *receive* returns messages) or returns an acknowledgment (e.g., *send, delete*). The round trip time is defined as the total (wallclock) time between initiating the request from the application and the delivery of the result or ack, respectively. For these experiments, the message size was fixed to 100 Bytes, but the sensitivity to the message size is low.

Again, Amazon has not published any details on the implementation of SQS. It seems, however, that SQS was designed along the same lines as S3. The messages of a queue are stored in a distributed and replicated way, possibly on many machines in different data centers. Clients can initiate requests at any time; they are never blocked by failures or other clients and will receive an answer (or ack) in constant time. For instance, if one client has locked all messages of a queue as part of a *receive* call, then another concurrent client which initiates another *receive* call will simply get an empty set of messages as a result. Since the queues are stored in a distributed way, SQS only makes a best-effort when returning messages in a FIFO manner. That is, there is no guarantee that SQS returns the first message of a queue as part of a *receive* call or that SQS returns the messages in the right order. Although SQS is designed to be extremely reliable, Amazon enforces deletions of messages after 4 days in the queue and retains the right to delete unused queues.

2.2.4 Indexing Service

Amazon SimpleDB is a recent extension to AWS family of services. At the time of writing this paper, SDB was still in beta and only available for a restricted set of users. With SimpleDB, Amazon offers a service to run simple queries on structured data. In SimpleDB, each item is associated to a domain which has associated attributes. SimpleDB automatically indexes an item as it is added to a domain. Each item can have up to 256 attribute values and each attribute value can range from 1 to 1,024 bytes. SimpleDB does not require pre-defined schemas so that any item with any kinds of attributes can be inserted and indexed.

Unfortunately, SimpleDB has a number of limitations. First, SimpleDB does not support any kind of bulkloading: every item must be inserted individually, which makes bulkloading both slow and expensive. Second, SimpleDB only supports *text* values (e.g., *strings*). If an application requires integers, dates, or floating point numbers, those values must be encoded properly. Third, the size limit of 1,024 bytes per attribute turns out to be too restrictive for many applications. Amazon recommends the use of S3 for anything larger, but obviously those data types cannot be indexed in S3. Another restriction is the expressiveness of the SimpleDB query language. The language allows for simple comparisons, negation, range expressions and queries on multiple attributes, but does not support joins or any other complex operation. Compared to a traditional database system, SimpleDB only provides *eventual consistency* guarantees in the same way as S3: That is, there are no guarantees *when* committed updates are propagated to all copies of the data.

As all services of AWS, SimpleDB is not for free. Unlike S3 and EC2, the cost is difficult to predict as it depends on the machine utilization of a request. Amazon charges USD 0.14 per machine hour consumed, USD 1.5 per GB of structured data stored per month, plus network bandwidth in the range of USD 0.10 to USD 0.17 per GB for communication with external (non-EC2) clients, depending on the consumed volume per month.

2.3 Reference Cloud API for Database Applications

This section describes an API that abstracts from the details of cloud services such as those offered by AWS. All protocols and techniques developed in the remainder of this paper for the development of Web-based database applications are based on calls to this API. This API specifies not only the interfaces but also the guarantees the different services should provide. Section 6 shows how this API can be implemented on top of AWS.

2.3.1 Storage Services

The most important building block is a reliable store. The following is a list of methods which can be easily implemented using today's cloud services (e.g., AWS and Google), thereby abstracting away from vendor specifics. This list can be seen as a greatest common divisor for all current cloud service providers and the minimum required in order to build stateful applications:

- ***put****(uri as string, payload as binary, metaData as key-value-pairs) as void:* Stores an item (or object) identified by the given URI. An item consists of the payload, the meta-data, and a timestamp of the last update. If the URI already exists, the item gets overwritten without warning; otherwise a new item is created.
- ***get****(uri as string) as item:* Retrieves the item (payload, meta-data, and timestamp) associated to the URI.
- ***get-metadata****(uri as string) as key-value-pairs:* Returns the meta-data of an item.
- ***getIfModifiedSince****(uri as string, timestamp as time) as item:* Returns the item associated to the URI only if the timestamp of the item is greater than the timestamp passed as a parameter of the call.
- ***listItems****(uriPrefix as string) as items:* Lists all items whose URIs match the given URI prefix.
- ***delete****(uri as string) as void:* Deletes an item.

For brevitity, we do not specify all error codes. Furthermore, we do not specify the operations for granting and removing access rights because implementing security on cloud services is beyond the scope of this paper.

In terms of consistency, we expect the cloud storage service to support *eventual consistency* [30]. Again, eventual consistency seems to be the standard offered by most cloud services today because it allows to scale out and provide 100 percent availability at low cost.

2.3.2 Machine Reservation

Building customized services in the cloud requires to run personal code in the cloud. The current trend is to provide either a hosting platform for a specific language like Google's AppEngine [18] or a facility to start a virtualized machine as done in Amazons EC2. Hosting platforms are often restricted to a certain programming language and normally hide details about the way how the code gets executed. Since virtualized machine are more general we propose the API to allow to start and shutdown machines as follows:

- ***start****(machineImage as string) as machineDescription:* Starts a given image and returns the virtual machine information such as virtual machine id and ip address.
- ***stop****(machineId as string) as void:* Stops the virtual machine with the machineId

For brevity, methods for creating own images are not described.

2.3.3 Simple Queues

As shown in Section 4, queues are an important building block in order to create Web-based database applications in the cloud. A queue should support the following operations (again, error codes are not specified for brevity):

- ***createQueue****(uri as string) as void:* Creates a new queue with the given URI.
- ***deleteQueue****(uri as string) as void:* Deletes a queue.
- ***listQueues****() as strings:* Returns the URIs as string of all queues.
- ***sendMessage****(uri as string, payload as binary) as integer:* Sends a message with the payload as the content to the queue and returns the MessageID. The MessageID is an integer (not necessarily ordered by time).
- ***receiveMessage****(uri as string, N as integer) as message:* Retrieves N messages from the queue. If less than N messages are available, as many messages are returned as possible. A message is returned with its MessageId and payload, of course.
- ***deleteMessage****(uri as string, messageId as integer) as void:* Deletes a message (identified by MessageId) from a queue (identified by its URI).

Queues should be reliable and never lose a message. (Unfortunately, SQS does not fulfil this requirement because it deletes messages after four days. Workarounds are described in Section 6.) Simple queues do not make any FIFO guarantees. That is, a *receiveMessage* call may return the second message without the first message. Furthermore, Simple Queues do not make any guarantees that all messages are available at all times; that is, if there are N messages in the queue, then it is possible that are *receiveMessage* call asking for N messages returns less than N messages. All protocols that are built on top of these Simple Queues must respect these flaws. As will become clear in Sections 4 and 5, the performance of a protocol improves the better the queue conforms to the FIFO principle and the more messages the queue returns as part of a *receiveMessage* calls. Some protocols require stricter guarantees; those protocols must be implemented on top of Advanced Queues.

2.3.4 Advanced Queues

Compared to Simple Queues, Advanced Queues provide stronger guarantees. Specifically, Advanced Queues are able to provide all messages to a user at any moment. As a consequence, requests to these queues are expected to be more expensive and Advanced Queues may temporarily not available. An additional difference between Advanced queues and Simple Queues is the availability of operators in the Advanced queue which allow to filter messages. Furthermore, Advanced Queues provide a way to attach a user-defined key (in addtion to the MessageID) to each message. This key allows further filtering of messages. Again, the details of our implementation of Advanced Queues on top of AWS are given in Section 6. The API of Advanced Queues is given in the following:

- ***createAdvancedQueue****(uri as string) as void:* Creates a new Advanced Queue with the given URI.
- ***deleteAdvancedQueue****(uri as string) as void:* Deletes a queue.
- ***sendMessage****(uri as string, payload as binary, key as integer) as integer:* Sends a message with the payload as the content to the queue and returns the MessageID. The MessageID is an integer that is ordered according to the arrival time of the message in the queue (messages received earlier have lower MessageID).

Furthermore, Advanced Queues support the attachment of user-defined keys to messages (represented as intergers) in order to carry out further filtering.

- ***receiveMessage****(uri as string, N as integer, messageIdGreaterThan as integer) as message:* Returns the top N messages whose MessageID is higher than the "messageIdGreaterThan." If less than N such messages exist, the Advanced Queue returns *all* matching messages. Unlike Simple Queues, the *receiveMessage* method of Advanced Queues respects the FIFO principle.

- ***receiveMessage****(uri as string, N as integer, keyGreaterThan as integer, keyLessThan as integer):* Retrieves the top N messages whose key matches the specified key value range. In terms of FIFO and completeness guarantees, this version of the *receiveMessage* operation behaves exactly like the *receiveMessage* operation which filters on MessageID.

- ***receiveMessage****(uri as string, N as integer, olderThan in seconds):* Retrieves the top N messages in the queue which are older than *olderThan* seconds.

- ***deleteMessage****(uri as string, messageId as integer) as void:* Deletes a message (identified by MessageId) from a queue (identified by its URI).

- ***deleteMessages****(uri as string, keyLessThan as integer, keyGreaterThan as integer) as void* Deletes all messages whose key are in the specified range.

2.3.5 Locking Service

The locking service implements a centralized service to keep track of read- and write-locks. A client (identified by a ClientID) is able to acquire a *shared lock* on a resource specified by a URI. Furthermore, users can acquire *exclusive locks*. Following conventional wisdom, several different clients can hold shared locks at the same time whereas exclusive locks can only be held by a single client and exclusive locks preclude the granting of shared locks. Locks are only granted for a certain timeframe. Expiring locks after a specified timeout, ensures the liveliness of the system in case a client which holds a lock crashes. As exclusive locks are rather often used, shared locks are only required for the 2 Phase Locking protocol presented in Section5.5. Fortunately, exclusive locks can be easily implemented on top of SQS as shown in Section 6, but implementing it as a dedicated cloud service as part of the cloud infrastructure may be more efficient than implementing it on top of basic cloud services.

The Locking Service API has the following operations:

- ***setTimeOut****(prefix as string, timeout as integer) as void:* Sets the timeout of locks for resources identified by a certain URI prefix.

- ***acquireXLock****(uri as string, ClientId as string) as boolean:* Acquires an exclusive lock. Returns true if the lock was granted and false, otherwise.

- ***acquireSLock****(uri as string, ClientId as string) as boolean:* Acquires a shared lock. Returns true, if the lock was granted, false, otherwise.

- ***releaseLock****(uri as string, ClientId as string) as boolean:* Releases a lock.

In terms of reliability, it is possible that the locking service fails. It is also possible that the locking service loses its state as part of such a failure. If a locking service recovers after such a failure it much refuse all *acquireXLock* and *acquireSLock* requests for the maximum timeout period in order to guarantee that all clients who hold locks can finish their work as long as they are holding the locks.

2.3.6 Advanced Counters

The advanced counter service is a special service designed for implementing the Generalized Snapshot Isolation protocol (Section 5). As its name implies, the Advanced Counter Service implements counters which are incremented with every *increment* call. Each counter is identified by a URI. A special feature of the Advanced Counter Service is that it allows to *validate* counter values and that it allows to get the highest *validated counter value*. If not explicitly validated, counter values are automatically validated after a specified timeout period. As shown in Section 5.4, this validation feature is important in order to implement the commit protocol of snapshot isolation. In summary, the API contains the following operations:

- ***setTimeOut****(prefix as string, timeout as integer) as void:* Sets the timeout of all counters for a certain URI prefix.
- ***increment****(uri as string) as integer:* Increments the counter, identified by the *uri* parametner, and returns the current value of the counter. If not counter with that URI exists, a new counter is created and initialized to 0.
- ***validate****(uri as string, value as integer) as void:* Validates a counter value. If not called explicitly, the counter value is validated automatically considering the timeout interval after the *increment* call that created that counter value.
- ***getHighestValidatedValue****(uri as string) as integer:* Returns the highest validated counter value.

Like the Lock Service, all protocols must be designed respecting that the Advanced Counter Service can fail at any time. When the Advanced Counter Service recovers, it resets all counters to 0. However, after restart the Advanced Counter Service refuses all requests (returns error) for the maximum timeout period of all counters. The maximum timeout period of all counters must, thus, be stored persistently and recoverable (e.g., in services like S3).

3 Database Architecture Revisited

As mentioned in Section 2.2.1, cloud computing promises infinite scalability, availability, and throughput. This section shows that many textbook techniques to implement tables, pages, B-trees, and logging can be applied to implement a database on top of the Cloud API of Section 2.3. The purpose of this section is to highlight the commonalities between a disk-based and cloud-based database system. The reader should not be surprised by anything said in this section. Sections 4 and 5, then, highlight the differences.

3.1 Client-Server Architecture

Figure 1 shows the proposed architecture of a database implemented on top of the Cloud Storage API described in Section 2.3.1. This architecture has a great deal of

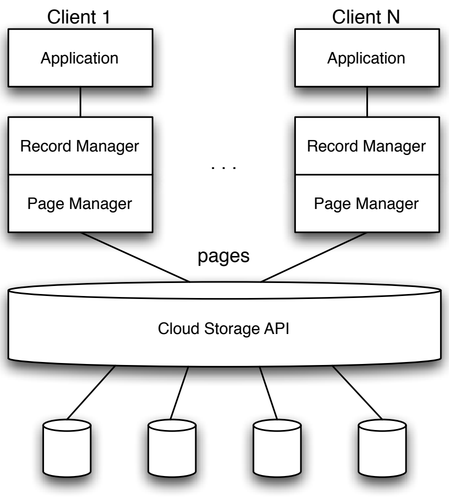

Figure 1: Shared-disk Architecture

commonalities with a distributed shared-disk database system [29]. The unit of transfer and buffering is a *page*. The difference is that pages are stored in the cloud persistently, rather than on a disk that is directly controlled by the database system. In this architecture, pages are implemented as *items* in the Storage Service API of Section 2.3.1. Consequently, pages are identified by a URI.

As in traditional database systems, a page contains a set of records or index entries. Following the general DB terminology, we refer to records as a bytestream of variable size whose size is constrained by the page size. Records can be relational tuples or XML elements and documents. Blobs can be stored directly on the Storage Service or using the techniques devised in [6]; all these techniques are applicable in a straightforward way so that Blobs are not discussed further in this paper.

Within a client, there is a stack of components that support the application. This work focuses on the two lowest layers; i.e., the record and page managers. All other layers (e.g., the query processor) are not affected by the use of cloud services and are, thus, considered to be part of the application. The *page manager* coordinates read and write requests to the Storage Service and buffers pages in local main memory or disk. The *record manager* provides a record-oriented interface, organizes records on pages, and carries out free-space management for the creation of new records. Applications interact with the record manager only, thereby using the interface described in the next subsection.

Throughout this work, we use the term *client* to refer to software artifacts that retrieve pages from and write pages back to the Cloud Storage Service. It is an interesting question on whether parts of the *client* application stack should run on machines provided by the cloud service provider (e.g., EC2 machines) and which parts of the application stack should run on machines provided by end users (e.g., PCs, laptops, mobile phones, etc.). Indeed, it is possible to implement Web-based database architecture using this architecture without using any machines from the cloud service provider. Section 7 explores the performance and cost trade-offs of different client-server configurations in more detail.

A related question concerns the implementation of indexes: One option is to use

SimpleDB (or related services) in order to implement indexing. An alternative is to implement B-trees on top of the cloud services in the same way as traditional database systems implement B-trees on top of disks. Again, the trade-offs are studied as part of performance experiments presented in Section 7. In order to avoid confusion, this section describes how B-tree indexes can be implemented on top of the Cloud Service API; integrating services like SimpleDB (SDB) for indexing is straightforward and a detailed description is, thus, omitted.

Independent of whether the client application stack runs on machines of users or on, say, EC2 machines, the architecture of Figure 1 is designed to support thousands if not millions of clients. As a result, all protocols must be designed in such a way that any client can fail at any time, possibly without ever recovering. As a result, clients are *stateless*. They may cache data from the cloud, but the worst thing that can happen if a client fails is that all the work of that client is lost.

In the remainder of this section, the record manager, page manager, implementation of (B-tree) indexes, and logging are described in more detail. Meta-data management such as the management of a catalogue which registers all collections and indexes is not discussed in this paper. It is straightforward to implement in this architecture in the same way as the catalogue of a traditional (relational) database is stored in the database itself. Furthermore, security is not described because it is beyond the scope of this paper. Some notes on alternative security models are given in [7], but a more comprehensive coverage of all security issues is an important avenue for future work.

3.2 Record Manager

The record manager manages records (e.g., relational tuples). Each record is associated to a collection (see below). A record is composed of a key and payload data. The key uniquely identifies the record within its collection. Both key and payload data are bytestreams of arbitrary length; the only constraint is that the size of the whole record must be smaller than the page size. (The implementation of Blobs is not addressed in this paper, as mentioned in the previous section.)

Physically, each record is stored in exactly one page which in turn is stored as a single *item* using the Cloud Store API. Logically, each record is part of a collection (e.g., a table). In our implementation, a collection is identified by a URI. All pages in the collection use the collection's URI as a prefix. The record manager provides functions to create new records, read records, update records, and scan collections.

Create(key, payload, uri): Creates a new record into the collection identified by *uri*. There are many alternative ways to implement free-space management [24], and they are all applicable in this context. In our implementation, free-space management is carried out using a B-tree; this approach is sometimes also referred to as *index-organized table*. That is, the new record is inserted into a leaf of a B-tree. The key must be defined by the application and it must be unique.

If the key is not unique, then *create* returns an error (if the error can be detected immediately) or ignores the request (if the error cannot be detected within the boundaries of the transaction, Section 4.3). In order to implement keys which are guaranteed to be unique in a distributed system, we used *uuids* generated by the client's hardware in our implementation.

Read*(key as uuid, uri as string):* Reads the payload data of a record given the key of the record and the URI of the collection.

Update*(key as uuid, payload as binary, uri as string):* Update the payload information of a record. In this study, all keys are immutable. The only way to change a key of a record is to *delete* and re-*create* the record.

Delete*(key as uuid, uri as string):* Delete a record.

Scan*(uri as string):* Scan through all records of a collection. To support scans, the record manager returns an *iterator* to the application.

In addition to the *create, read, update,* and *scan* methods, the API of the record manager supports *commit* and *abort* methods. These two methods are implemented by the page manager, described in the next section. Furthermore, the record manager exposes an interface to probe indexes (e.g., range queries): Such requests are either handled by services like SimpleDB (straightforward to implement) or by B-tree indexes which are also implemented on top of the cloud infrastructure. The implementation of such "client-side" B-tree indexes is described in Section 3.4.

3.3 Page Manager

The page manager implements a buffer pool directly on top of the Cloud Storage API. It supports reading pages from the service, pinning the pages in the buffer pool, updating the pages in the buffer pool, and marking the pages as updated. The page manager also provides a way to create new pages. All this functionality is straightforward and can be implemented just as in any other database system. Furthermore, the page manager implements the *commit* and *abort* methods. We use the term *transaction* for a sequence of read, update, and create requests between two commit or abort calls. It is assumed that the write set of a transaction (i.e., the set of updated and newly created pages) fits into the client's main memory or secondary storage (e.g., flash or disk). If an application commits, all the updates are propagated to the cloud via the *put* method of the Cloud Storage Service (Section 2.3.1) and all the affected pages are marked as *unmodified* in the buffer pool. How this propagation works is described in Section 4. If the application aborts a transaction, all pages marked *modified* or *new* are simply discarded from the buffer pool, without any interaction with the cloud service. We use the term transaction liberally in this work: Not all the protocols presented in this paper give ACID guarantees in the DB sense. The assumption that the write set of a transaction must fit in the client's buffer pool can be relaxed by allocating additional overflow pages for this purpose using the Cloud Storage Service; discussing such protocols, however, is beyond the scope of this paper and rarely needed in practice.

The page manager keeps copies of pages from the Cloud Storage Service in the buffer pool across transactions. That is, no pages are evicted from the buffer pool as part of a *commit*. An *abort* only evicts modified and new pages. Pages are refreshed in the buffer pool using a *time to live* (TTL) protocol: If an unmodified page is requested from the buffer pool after its time to live has expired, the page manager issues a *get-if-modified-since* request to the Cloud Storage API in order to get an up-to-date version, if necessary (Section 2.3.1).

3.4 B-tree Indexes

Again, as mentioned several times earlier, there are two fundamentally different ways to implement indexes. First, cloud services for indexing such as SimpleDB can be leveraged. Second, indexes can be implemented on top of the page manager. This section describes the second approach using B-trees as an example. The first approach is straightforward. The trade-offs of the two approaches are studied in Section 7.

B-trees can be implemented on top of the page manager in a fairly straightforward manner. Again, the idea is to adopt existing textbook database technology as much as possible. The root and intermediate nodes of the B-tree are stored as pages on the storage service (via the page manager) and contain *(key, uri)* pairs: *uri* refers to the appropriate page at the next lower level. The leaf pages of a primary index contain entries of the form *(key, payload)*; that is, these pages store the records of the collection in the index-organized table (Section 3.2). The leaf pages of a secondary index contain entries of the form *(search key, record key)*. That is, probing a secondary index involves navigating through the secondary index in order to retrieve the *keys* of the matching records and then navigating through the primary index in order to retrieve the records with the payload data.

As mentioned in Section 3.1, holding locks must be avoided as much as possible in a scalable distributed architecture. Therefore, we propose to use B-link trees [22] and their use in a distributed system as proposed by [23] in order to allow concurrent reads and writes (in particular splits), rather than the more mainstream *lock-coupling protocol* [3]. That is, each node of the B-tree contains a pointer (i.e., URI) to its right sibling at the same level. At the leaf level, this chaining can naturally be exploited in order to implement scans through the whole collection or through large key ranges.

A B-tree is identified by the URI of its root page. A collection is identified by the URI of the root of its primary index. Both URIs are stored persistently as meta-data in the system's catalogue on the cloud service. (Section 3.1). Since the URI of an index is a reference to the root page of the B-tree, it is important that the root page is always referenced by the same URI. Implementing this requirement involves a slightly modified, yet straightforward, way to split the root node. Another deviation to the standard B-tree protocol is that the root node of a B-tree can be empty; it is not deleted even if the B-tree contains no entries.

3.5 Logging

The protocols described in Sections 4 and 5 make extensive use of *redo* log records. In all these protocols, it is assumed that the log records are *idempotent*; that is, applying a log record twice or more often has the same effect as applying the log record only once. Again, there is no need to reinvent the wheel and textbook log records as well as logging techniques are appropriate [19]. If not stated otherwise, we used the following (simple) redo log records in our implementation:

- *(insert, key, payload):* An *insert* log record describes the creation of a new record; such a log record is always associated to a collection (more precisely to the primary index which organizes the collection) or to a secondary index. If such an *insert* log record is associated to a collection, then the *key* represents the key value of the new record and the *payload* contains the other data of the record. If the *insert* log record is associated to a secondary index, then the *key* is the value of the search key of that secondary index (possibly composite) and the payload is the primary key value of the referenced record.

- *(delete, key):* A *delete* log record is also associated either to a collection (i.e., primary index) or to a secondary index.

- *(update, key, afterimage):* An *update* log record must be associated to a data page; i.e., a leaf node of a primary index of a collection. An update log record contains the new state (i.e., after image) of the referenced record. Diffing, logical logging, or other optimized logging techniques are not studied in this work for simplicity; they can be applied to cloud databases in the same way as to any other database system. Entries in a secondary index are updated by deleting and re-inserting these entries.

By nature, all these log records are idempotent: In all three cases, it can be deduced from the database whether the updates described by the log record have already been applied. With such simple *update* log records, however, it is possible that the same update is applied twice if another update overwrote the first update before the second update. This property can result in indeterminisms as shown in Section 4.3. In order to avoid these indeterminisms, more sophisticated logging can be used such as the log records used in Section 5.2.

If an operation involves updates to a record and updates to one or several secondary indexes, then separate log records are created by the record manager to log the updates in the collection and at the secondary indexes. Again, implementing this functionality in the record manager is straightforward and not different to any textbook database system.

Most protocols studied in this work involve *redo* logging only. Only the protocol sketched in Section 5.4 requires *undo* logging. *Undo* logging is also straightforward to implement by keeping the *before image* in addition to the *after image* in *update* log records, and by keeping the last version of the record in *delete* log records.

4 Basic Commit Protocols

The previous section showed that a database implemented on top of cloud computing services can have a great deal of commonalities with a traditional textbook database system implemented on top of disks. This section addresses one particular issue which arises when concurrent clients commit updates to records stored on the same page. If no care is taken, then the updates of one client are overwritten by the other client, even if the two clients update different records. The reason is that the unit of transfer between clients and the Cloud Storage Service in the architecture of Figure 1 is a page, rather than an individual record. This issue does not arise in a (shared-disk) database system because the database system coordinates updates to the disk(s); however, this coordination limits the scalability (number of nodes/clients) of a shared-disk database. This issue does not arise in the way that cloud storage services are used conventionally because today those are mostly used to store large objects (e.g., multi-media objects) so that the unit of transfer can be the object; for small records, clustering several records into pages is mandatory in order to get acceptable performance (Section 2.2.1). Obviously, if two concurrent clients update the same record, then the last updater wins. Protocols to synchronize concurrent update transactions are sketched in Sections 5.4 and 5.5.

The protocols devised in this section are applicable to all architectural variants described in Section 3; i.e., independent of which parts of the client application stack are executed on end users' machines (e.g., laptops) and which parts are executed on cloud machines (e.g., EC2 machines). Again, the term *client* is used in order to abstract from these different architectures. The protocols are also applicable to the two different ways of implementing indexes (SimpleDB vs. client-side B-trees); again, this section refers to the implementation of "client-side B-trees" because it is the more complicated variant and in order to avoid confusion. The protocols designed in this section preserve the main features of cloud computing: clients can fail anytime, clients can read and write data at constant time, clients are never blocked by concurrent clients, and distributed

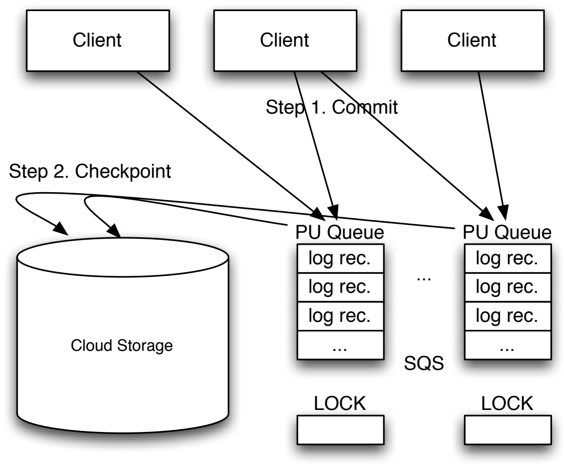

Figure 2: Basic Commit Protocol

Web-based applications can be built on top of the cloud, without the need to build or administrate any additional infrastructure. Again, the price to pay for these features is reduced consistency: In theory, it might take an undetermined amount of time before the updates of one client become visible at other clients. In practice, the time can be controlled, thereby increasing the cost (in $) of running an application for increased consistency (i.e., a reduced propagation time).

4.1 Overview

Figure 2 demonstrates the basic idea of how clients commit updates. The protocol is carried out in two steps:

- In the first step, the client generates log records for all the updates that are committed as part of the transaction and sends them to the queues.
- In the second step, the log records are applied to the pages using our Store API. We call this step checkpointing.[1]

This protocol is extremely simple, but it serves the purpose. Assuming that the queue service is virtually always available and that sending messages to the queues never blocks, the first step can be carried out in constant time (assuming a constant or bounded number of messages which must be sent per commit). The second step, checkpointing, involves synchronization (Section 4.3), but this step can be carried out asynchronously and outside of the execution of a client application. That is, end users are never blocked by the checkpointing process. As a result, virtually 100 percent read, write, and commit availability is achieved, independent of the activity of concurrent clients and failures of other clients.

The protocol of Figure 2 is also resilient to failures. If a client crashes during *commit*, then the client resends all log records when it restarts. In this case, it is possible

[1]We use the word *checkpointing* for this activity because it applies updates from one storage media (queues) to the persistent storage. There are, however, important differences to traditional DBMS checkpointing. Most importantly, checkpointing is carried out in order to reduce the recovery time after failure in traditional DBMSes. Here, checkpointing is carried out in order to make updates visible.

that the client sends some log records twice and as a result these log records may be applied twice. However, applying log records twice is not a problem because the log records are idempotent (Section 3.5). If a client crashes during commit, it is also possible that the client never comes back or loses uncommitted log records. In this case, some log records of the commit have been applied (before the failure) and some log records of the commit will never be applied, thereby violating atomicity. Indeed, the basic commit protocol of Figure 2 does not guarantee atomicity. Atomicity, however, can be implemented on top of this protocol as shown in Section 5.1.

In summary, the protocol of Figure 2 preserves all the features of cloud computing. Unfortunately, it does not help with regard to consistency. That is, the time before an update of one client becomes visible to other clients is unbounded in theory. The only guarantee that can be given is that *eventually* all updates will become visible to everybody and that all updates are durable. This property is known as *eventual consistency* [30]. In practice, the freshness of data seen by clients can be controlled by setting the checkpoint interval (Section 4.5) and the TTL value at each client's cache (Section 3.1). Setting the checkpoint interval and TTL values to lower values will increase the freshness of data, but it will also increase the ($) cost per transaction (see experiments in [7]). Another way to increase the freshness of data (at increased cost) is to allow clients to receive log records directly from the queue, before they have been applied to the persistent storage as part of a checkpoint.

The remainder of this section describes the details of the basic commit protocol of Figure 2; i.e., committing log records to queues (Step 1) and checkpointing (Step 2).

4.2 PU Queues

Figure 2 shows that clients propagate their log records to so-called *PU queues* (i.e., **P**ending **U**pdate queues). These PU queues are implemented as Simple Queues using the API defined in Section 2.3.3. In theory, it would be sufficient to have a single PU queue for the whole system. However, it is better to have several PU queues because that allows multiple clients to carry out checkpoints concurrently: As shown in Section 4.3, a PU queue can only be checkpointed by a single client at the same time. Specifically, we propose to establish PU queues for the following structures:

- Each B-tree (primary and secondary) has one PU queue associated to it. The PU queue of a B-tree is created when the B-tree is created and its URI is derived from the URI of the B-tree (i.e., the URI of the root node of the B-tree). All *insert* and *delete* log records are submitted to the PU queues of B-trees.

- One PU queue is associated to each leaf node of a primary B-tree of a collection. We refer to these leaf nodes as *data pages* because they contain all the records of a collection. Only *update* log records are submitted to the PU queues of data pages. The URIs of these PU queues are derived from the corresponding URIs of the data pages.

If services like SimpleDB are used in order to index the data, then the first category of PU Queues for B-trees are not needed. Records can directly be inserted into and deleted from such SimpleDB indexes. The second kind of PU Queues for data pages are still needed in such an architectural variant.

The pseudo code of the commit routine is given in Algorithm 1. In that algorithm, $R.Log$ refers to the log record generated to capture all updates on Record R. $C.Uri$ refers to the URI of the collection to which a Record R belongs to; this URI coincides with the URI of the root page of the collections primary index. $P.Uri$ is the URI of the page in which a record resides. $R.key$ is the key value of Record R.

Algorithm 1 Commit Protocol

```
1:  for all modified records $R$ do
2:    $C \leftarrow$ collection of $R$
3:    if $R$ has no assigned page (i.e., $R$ is a new record) then
4:      sendMessage($C$.Uri, $R$.Log);
5:    else
6:      $P \leftarrow$ page of $R$
7:      sendMessage($P$.Uri, $R$.Log);
8:    end if
9:    for all $C$.SecondaryIndexes $S$ do
10:     if $R$ has a modified value for $S$ then
11:       sendMessage($S$.Uri, pair($R$.Key, $V$));
12:     end if
13:   end for
14: end for
```

4.3 Checkpoint Protocol for Data Pages

Checkpoints can be carried out at any time and by any node (or client) of the system. A checkpoint strategy determines when and by whom a checkpoint is carried out (Section 4.5). This section describes how a checkpoint of *update* log records is executed on data pages; i.e., leaf nodes of the primary index of a collection. The next section describes how checkpoints of *insert* and *delete* log records are carried out for B-trees. (If services like SimpleDB are used for indexing, then checkpointing for index *inserts* and *deletes* is not necessary because the *inserts* and *deletes* can be applied directly.)

The input of a checkpoint is a PU queue. The most important challenge when carrying out a checkpoint is to make sure that nobody else is concurrently carrying out a checkpoint on the same PU queue. For instance, if two clients carry out a checkpoint concurrently using the same PU queue, some updates (i.e., log records) might be lost because it is unlikely that both clients will read the exactly same set of log records from the PU queue (Section 2.3.3). In order to synchronize checkpoints, the Cloud Lock Service (Section 2.3.5) is used. When a client (or any other authority) attempts to do a checkpoint on a PU queue, it tries first to acquire an exclusive lock for the PU queue URI. If that lock request is granted, then the client knows that nobody else is concurrently applying a checkpoint on that PU queue and proceeds to carry out the checkpoint. If it is not granted, then the client assumes that a concurrent client is carrying out a checkpoint and simply terminates the routine (no action required for this client).

Per definition, the exclusive lock is only granted for a specific time-frame. During this time period the client must have completed the checkpoint; if the client is not finished within that timeout period, the client aborts checkpoint processing and propagates no changes. Setting the timeout for holding the exclusive lock during checkpointing a data page is critical. Setting the value too low might result in starvation because no checkpoint will ever be completed if the PU queue has exceeded a certain length. Furthermore, the timeout must be set long enough to give the Cloud Storage Service enough time to propagate all updates to a data page to all replicas of that data page.

On the other hand, a short timeout enables frequent checkpoints and, thus, fresher data. For the experiments reported in Section 7, a timeout of 30 seconds was used.

Algorithm 2 Checkpoint Protocol

```
Require: page $P$, $PropPeriod$ , $X \leftarrow$ maximum number of log records per checkpoint
 1: if acquireWriteLock($P$.Uri) then
 2:   $StartTime \leftarrow$ CurrentTime()
 3:   $V \leftarrow$ get-if-modified-since($P$.Uri, $P$.Timestamp)
 4:   if $V \neq Null$ then
 5:     $P \leftarrow V$
 6:   end if
 7:   $M \leftarrow$ receiveMessge($P$.Uri, $X$)
 8:   for all messages $m$ in $M$ do
 9:     apply $m$ to $P$
10:   end for
11:   if CurrentTime() - $StartTime < LockTimeOut - PropPeriod$ then
12:     put($P$.Uri, $P$.Data)
13:     for all messages $m$ in $M$ do
14:       deleteMessage($P$.Uri, $m$.Id)
15:     end for
16:   end if
17: end if
```

The complete checkpoint algorithm is given in Algorithm 2. The algorithm first gets an exclusive lock (Line 1) and then re-reads the data page, if necessary (Lines 2-6). In order to find out whether the current version of the data page in the client's cache is fresh, each data page contains the timestamp of the last checkpoint as part of its page header. After that, X update log records are read from the PU Queue (Line 7). X is a parameter of this algorithm and depends on the timeout period set for holding the exclusive lock (the longer, the more log records can be applied) and the checkpoint interval (Section 4.5, the longer, the more log records are available). In our implementation, X was set to 256. After reading the log records from the PU Queue, the log records are applied to the local copy of the page in the client's cache (Lines 8-10) and the modified page is written back to the Cloud Storage Service (Line 12). Writing back the page to the Cloud Storage Service involves propagating the new version of the data page to all replicas of the data page inside the cloud. This propagation must happen within the time out period of the exclusive lock in order to avoid inconsistencies created by concurrent checkpointing clients. Unfortunately, Cloud Storage providers do not give any guarantees how long this propagation takes, but, for instance, Amazon claims that five seconds is a safe value for S3 (one second is the norm). Accordingly, Line 11 considers a *PropPeriod* parameter which is set to five seconds in all experiments reported in Section 7. There are other ways to avoid inconsistencies with concurrent checkpointing clients; these protocols involve the use of Advanced Counters or Advanced Queues; going into the details is beyond the scope of this paper. Finally, at the end of Algorithm 2, the log records are deleted from the PU Queue (Lines 13-15).

In Line 9 of Algorithm 2, it is possible that the data page must be split because the records grew. For brevity, this paper does not describe all the details of splitting pages. In the implementation with index-organized tables, splitting data pages is the same as splitting index nodes and carried out along the lines of [23] so that clients which read a page are not blocked while the page is split. If SimpleDB is used to index the pages, then a similar protocol can be used, thereby linking the data pages. In Line 12, the *put* method to the Cloud Storage Service is considered to be atomic. The exclusive lock obtained in Line 1 need not be released explicitly because it becomes available automatically after the timeout period.

This protocol to propagate updates from a PU queue to the Cloud Storage Service is safe because the client can fail at any point in time without causing any damage. If the client fails before Line 12, then no damage is made because neither the PU Queue nor the data page have changed. If the client fails after Line 12 and before the deletion of all log records from the PU Queue, then it is possible that some log records are applied twice. Again, no damage is caused in this case because the log records are idempotent (Section 3.5). In this case, indeterminisms can appear if the PU queue contains several *update* log records that affect the same *key*. As part of a subsequent checkpoint, these log records may be applied in a different order so that two different versions of the page may become visible to clients, even though no other updates were initiated in the meantime. These indeterminisms can be avoided by using the extended logging mechanism for *monotonic writes* described in Section 5.2 as opposed to the simple log records described in Section 3.5.

4.4 Checkpoint Protocol for Client-side B-trees

As mentioned in Section 4.2, checkpointing is only needed for client-side (B-tree) indexes. No checkpointing is required if indexing is implemented using services like SimpleDB. For client-side B-tree indexes, there is one PU queue associated to each B-tree: This PU queue contains *insert* and *delete* log records for that B-tree. Primary and secondary indexes are checkpointed in the same way; only the leaf nodes of a primary index (i.e., data pages) are treated specially. Checkpointing a client-side B-tree is more complicated than checkpointing a data page because several (B-tree) pages are involved in a checkpoint and because splitting and deleting pages are frequent. Nevertheless, the basic ideas are the same and can be summarized in the following protocol sketch:

1. Obtain the token from the LOCK queue (same as Step 1, Section 4.3).

2. Receive log records from the PU queue (Step 2, Section 4.3).

3. Sort the log records by key.

4. Take the first (unprocessed) log record and navigate through the B-tree to the leaf node which is affected by this log record. Reread that leaf node from S3 using S3's *get-if-modified* method.

5. Apply all log records that are relevant to that leaf node.

6. If the timeout of the token received in Step 1 has not expired (with some padding for the *put*), *put* the new version of the node to S3; otherwise terminate (same as Step 5, Section 4.3).

7. If the timeout has not expired, delete the log records which were applied in Step 5, from the PU queue.

8. If not all log records have been processed yet, goto Step 4. Otherwise, terminate.

As part of Step 5, nodes might become empty or be split. Again, we cannot describe all the details in this paper due to space constraints and refer the interested reader to the technical report. As mentioned in Section 3.4, our implementation adopts the techniques of [23] to make sure that concurrent readers are not blocked by splits and deletions carried out by a checkpoint.

4.5 Checkpoint Strategies

The purpose of the previous two sections was to show *how* checkpoints are implemented. The protocols were designed in such a way that anybody can apply a checkpoint at any time. This section discusses alternative *checkpoint strategies*. A checkpoint strategy determines *when* and *by whom* a checkpoint is carried out. Along both dimensions, there are several alternatives.

A checkpoint on a Page (or Index) X can be carried out by the following authorities:

- *Reader:* A reader of X.
- *Writer:* A client who just committed updates to X.
- *Watchdog:* A process which periodically checks PU queues.
- *Owner:* X is assigned to a specific client which periodically checks the PU queue of X.

In this work, we propose to have checkpoints carried out by *readers* and *writers* while they work on the page (or index) anyway. Establishing *watchdogs* to periodically check PU queues is a waste of resources and requires an additional infrastructure to run the *watchdogs*. Likewise, assigning *owners* to PU queues involves wasting resources because the owners must poll the state of their PU queues. Furthermore, owners may be offline for an undetermined amount of time in which case the updates might never be propagated from the PU queue to S3. The advantage of using *watchdogs* and assigning *owners* to PU queues is that the protocols of Sections 4.3 and 4.4 are simplified (no LOCK queues are needed) because no synchronization between potentially concurrent clients is required. Nevertheless, we believe that the disadvantages outweigh this advantage.

The discussion of whether checkpoints should be carried out by *readers* or *writers* is more subtle and depends on the second question of *when* checkpoints should be carried out. In this work, we propose to use writers in general and readers only in exceptional cases (see below). A writer initiates a checkpoint using the following condition:

- Each data page records the timestamp of the last checkpoint in its header. For B-trees, the timestamp is recorded in the metadata (Section 2.2.1) associated to the root page of the B-tree. For B-trees, the S3 maintained metadata, rather than the root page, is used to store this information because checkpointing a B-tree typically does not involve modifying the root and rewriting the whole root in this event would be wasteful. The timestamp is taken from the machine that carries out the checkpoint. It is not important to have synchronized clocks at all machines; out-of-sync clocks will result in more or less frequent checkpoints, but they will not affect the correctness of the protocol (i.e., eventual consistency at full availability).

- When a client commits a log record to a data page or B-tree, the client computes the difference between its current wallclock time and the timestamp recorded for the last checkpoint in the data page / B-tree. If the absolute value of this difference is bigger than a certain threshold (*checkpoint interval*), then the *writer* carries out a checkpoint asynchronously (not blocking any other activity at the client). The *absolute* value of the difference is used because out-of-sync clocks might return outrageous timestamps that lie in the future; in this case, the difference is negative.

The *checkpoint interval* is an application-dependent configuration parameter; the lower it is set, the faster updates become visible, yet the higher the cost (in USD) in order to carry out many checkpoints. The trade-offs of this parameter were studied in [7]. Obviously, the checkpoint interval must be set to a significantly larger value than the *timeout* on the LOCK queue for checkpoint processing used in the protocols of Sections 4.3 and 4.4. For a typical Web-based application, the checkpoint interval should be set to, say, 10-15 seconds whereas timeouts on LOCK queues should be set to 1-2 seconds. Clearly, none of the protocols devised in this work are appropriate to execute transactions on hot-spot objects which are updated thousands of times per second.

Unfortunately, the *writer-only* strategy has a flaw. It is possible that a page which is updated once and then never again is never checkpointed. As a result, the update never becomes visible. In order to remedy this situation, it is important that readers also initiate checkpoints if they see a page whose last checkpoint was a long time ago: A reader initiates a checkpoint randomly with a probability proportional to $1/x$ if x is the time period since the last checkpoint; x must be larger than the checkpoint interval. (The longer the page has not been checkpointed after the checkpoint interval expired, the less likely a checkpoint is needed in this approach.) Initiating a checkpoint does no block the reader; again, all checkpointers are carried out asynchronously outside of any transaction. Of course, it is still possible that an update from a PU queue is never checkpointed in the event that the data page or index is neither read nor updated; we need not worry about this case, however, because the page or index is garbage in this case.

The proposed checkpointing strategy makes decisions for each data page and each index individually. There are no concerted checkpointing decisions. This design simplifies the implementation, but it can be the source for additional inconsistencies. If a new record is inserted, for instance, it is possible that the new record becomes visible in a secondary index on S3 before it becomes visible in the primary index. Likewise, the query *select count(*) from collection* can return different results, depending on the index used to process this query. How to avoid such phantoms and achieve serializability is discussed in Sections 5.4 and 5.5 ; unfortunately, serializability cannot be achieved without sacrificing scalability and full availability of the system.

5 Transactional Properties

The previous section showed how *durability* can be implemented on top of a Cloud Storage Service. No update is ever lost, updates are guaranteed to become visible to other clients (*eventual consistency* [30]), and the state of records and indexes persist until they are overwritten by other transactions. This section describes how additional transactional properties can be implemented. Again, the goal is to provide these additional properties at as low as possible additional cost (monetary and latency), and without sacrificing the basic principles of cloud computing: scalability, availability, and no need to operate an additional infrastructure. This section revisits the Atomicity and Monotonicity protocols of [7] and shows how they can be implemented using the services of Section 2.3. Additionally, this section presents two new protocols, locking and snapshot isolation, which were not presented in [7]. Both of these protocols implement high levels of consistency (in fact, locking implements strong consistency). The trade-offs of all protocols are studied with help of the performance experiments presented in Section 7. All protocols described in this section are layered on top of the basic protocol described in the previous section.

5.1 Atomicity

Atomicity implies that *all* or *none* of the updates of a transaction become visible. Atomicity is not guaranteed using the Basic Commit Protocol depicted in Figure 2. If a client fails while processing a commit of a transaction, it is possible that the client already submitted some updates to the corresponding PU queues whereas other updates of the transaction are lost due to the failure.

Fortunately, atomicity can be implemented using additional ATOMIC Queues. An ATOMIC Queue is associated to each client and implemented using the Simple Queue Service (Section 2.3.3). In fact, it is possible that a client maintains several such queues in order to execute several transactions concurrently. For ease of presentation, however, we assume that each client has a single ATOMIC queue and that a client executes transactions one after the other. (Of course, several operations can be executed by a client as part of the same transaction.)

The commit protocol that ensures atomicity with help of ATOMIC Queues is shown in Algorithm 3. The idea is simple. First, all log records are sent to the ATOMIC Queue, thereby remembering the message ids of all messages sent to the ATOMIC Queue in the LogMessageIDs set (lines 1-3). For efficiency, it is possible that the client packs several log records into a single message to the ATOMIC Queue, thereby exploiting the maximum message size of the Cloud Simple Queue Service; to facilitate presentation, this trick is not shown in Algorithm 3. Once the client has written all log records of the transaction to its ATOMIC Queue, the client sends a special *commitLog* record to the ATOMIC Queue (line 5). At this point, the transaction is successfully committed and recoverable. Then, the client executes the basic commit protocol of Algorithm 3 (line 6); that is, the client sends the log records to the corresponding PU Queues so that they can be propagated to the data and index pages in subsequent checkpoints. If the basic commit protocol was carried out successfully, the client removes all messages from its ATOMIC Queue, hereby using the LogMessageIDs set (lines 7-9).

When a client fails, the client executes the recovery protocol of Algorithm 4 when it restarts. First, the client reads all log messages from its ATOMIC Queue (lines 1-6). Since Simple Queues are used and these queues do not give any completeness guarantees, probing the ATOMIC Queue is carried out several times until the client is guaranteed to have received all messages from its ATOMIC Queue. (If an Advanced Queue is used, this iterative probing process is simplified.) Once the client has read all log records from its ATOMIC Queue, the client checks whether its last transaction was a winner; i.e., whether a *commitLog* record was written before the crash (Line 7 of Algorithm 4). If it was a winner, then the client simply re-applies all the log records according to the basic commit protocol and deletes all log records from its ATOMIC Queue (Lines 10-12). For loser transactions, the client simply deletes all log records from its ATOMIC Queue.

Algorithm 3 Atomicity - Commit

```
Require: $AtomicQueueUri \leftarrow$ Atomic Queue URI for the Client
1: $LogMessageIds \leftarrow \emptyset$
2: for all modified records $R$ do
3:     $LogMessageIds$.add(sendMessage($AtomicQueueUri$, $R$.Log))
4: end for
5: $LogMessageIds$.add(sendMessage($AtomicQueueUri$, CommitLog));
6: execute basic commit protocol (Algorithm 1)
7: for all $i$ in $LogMessageIds$ do
8:     deleteMessage($AtomicQueueUri$, $i$)
9: end for
```

Algorithm 4 Atomicitiy - Recovery

```
Require: $AtomicQueueUri \leftarrow$ Atomic Queue URI for the Client
1: $LogMessages \leftarrow \emptyset$
2: $M \leftarrow$ receiveMessge($AtomicQueueUri$, $\infty$)
3: while $M.size() \neq 0$ do
4:     $LogMessages$.add($M$)
5:     $M \leftarrow$ receiveMessge($AtomicQueueUri$, $\infty$)
6: end while
7: if CommitLog $\in LogMesssages$ then
8:     execute basic commit protocol (Algorithm 1)
9: end if
10: for all $l$ in $LogMessages$ do
11:     deleteMessage($AtomicQueueUri$, $l$.MessageId)
12: end for
```

Of course, clients can fail after restart and while scanning the ATOMIC Queue. Such failures cause no damage. It is possible that log records are propagated to PU queues twice or even more often, but that is not an issue because the application of log records is idempotent. If a client fails permanently, then a different client (or some other service) must periodically execute the recovery protocol of Algorithm 4 for inactive ATOMIC Queues. If the *commitLog* record is deleted last in Lines 10-12 of Algorithm 4, then several recovery processes on the same ATOMIC Queue can be carried out concurrently (and fail at any moment) without causing any damage. More details for concurrent recovery may be found in Section 5.3.

5.2 Consistency Levels

Tanenbaum and van Steen describe different levels of consistency in their book [30]. The highest level of consistency is *strict consistency*. Strict consistency mandates that "every read on a data item x returns a value corresponding to the result of the most recent write on x" [30]. Strict consistency can only be achieved by synchronizing the operations of concurrent clients; isolation protocols are discussed in the next sections (Sections 5.4 and Sections 5.5). The weaker levels of consistency such as *monotonic reads* and *read your writes* are trivial to implement with an Advanced Queue Service as described in Section 2.3.4. Therefore, we forgo to discuss the algorithms in more detail. The implementation of weaker consistency levels without relying on Advanced Queues have already been outlined in [7] and are also not further discussed.

5.3 Atomicity for Strong Consistency

In Section 5.1 we presented how atomicity can be achieved. This protocol guarantees that either all or none of the updates become visible as long as the client recovers at some point. Unfortunately, this protocol gives no guarantees on how long such a recovery will last and therefore also how long a inconsistent state can exist. For building protocols with stronger consistency guarantees we need to minimize the inconsistency timeframe. One solutions to this problem is to allow clients to recover for failures from other clients. Another problem encountered in a highly distributed system with thousands of clients is, that in the worst case a single client is able to block the whole system (Sections 5.4 and 5.5 will demonstrate this issue in more detail). Transaction timeouts are one solution to this problem and are applied here. As the higher consistency levels require Advanced Queues anyway, the atomicity protocol presented here also makes use of Advanced Queue features.

The revisited code for Atomocity is shown in Algorithm 5 and Algorithm 6. The main differences are:

- The protocol only requires two ATOMIC Queues for all clients: the ATOMIC COMMIT Queue and the ATOMIC Queue. PU messages are send to the ATOMIC Queue whereas commit messages are send to ATOMIC COMMIT Queue. We separated those queues just for simplicity and performance.
- The commit message is only send if the transaction is faster as the transaction timeout. Therefore, long-running transactions are aborted.
- All messages carry the ClientID as additional key.

The revised recovery algorithm is presented in Algorithm 6. In order to enable clients to recover for failures from other clients, clients are required to check on commit

Algorithm 5 Advanced Atomicitiy - Commit

```
Require: $AtomicQueueUri$, $AtomicCommitQueueUri$, $ClientId$, $TransactionStartTime$, $TransactionTimeout$
 1: $LogMessageIds \leftarrow \emptyset$
 2: for all modified records $R$ do
 3:     $LogMessageIds$.add(sendMessage($AtomicQueueUri$, $R$.Log, $ClientId$))
 4: end for
 5: if CurrentTime() $- TransactionStartTime < TransactionTimeout$ then
 6:     $C \leftarrow$ sendMessage($AtomicCommitQueueUri$, CommitLog, $ClientId$);
 7:     execute basic commit protocol (Algorithm 1)
 8:     for all $i$ in $LogMessageIds$ do
 9:         deleteMessage($AtomicQueueUri$, $i$)
10:     end for
11:     deleteMessage($AtomicCommitQueueUri$, $C$)
12: end if
```

messages from others. If we find an old message, meaning older than a timeframe called AtomicTimeout, it is assumed that recovery is required (line 1). To specify what old in this context means is crucial. Considering too young messages as old might result in false positives, on the other hand setting a too high value for the AtomicTimeout value might hold the database in an inconsistent state for too long. We found that 5s is a good value. If a client determines it has to do recovery, it tries to receive the recovery lock. If the client is able to get the lock (line 3), the client is allowed to perform the recovery steps (line 4-11). If not, the transaction is aborted as the database is in recovery (line 15). To allow just one client to recover is the easiest way of recovery but also might result in longer unavailability times of the system. If those outages are unacceptable or/and clients are assumed to crash more often, solutions include parallel recovery from several clients up to partitioning the ATOMIC Queues to consistency entities. For simplicity, we do not further discuss those strategies. The actual recovery is similar to the recovery mechanism described before. The only difference is, that we need to make sure, only to recover for a certain commit message. To do this kind of group by we use the feature of the Advanced Queues, to filter the messages according to the user-defined key (line 6).

Again, crashes during the recovery do not harm, as the logs are idempotent. Also, if the lock-timeout for the recovery expires or a transaction simply takes longer for the atomic commit, it does no damage for the same reason.

5.4 Generalized Snapshot Isolation

The idea of snapshot isolation is to serialize transactions in the order of the time they started [5]. When a transaction reads a record, it initiates a time travel and retrieves the version of the object as of the moment when the transaction started. For this purpose, all log records must support undo-locking (Section 3.5) and log records must be archived even beyond checkpointing. Generalized snapshot isolation relaxes snapshot isolation in the sense that it is not required to use the latest snapshot [14]. This allows higher concurrency in the system as several transactions are able to validate at the same time and in the case of read-only transactions a snapshot is more likely to be handled by the cache.

We therefore propose a protocol applying generalized snapshot isolation (GSI).

Algorithm 6 Advanced Atomicitiy - Recovery

```
Require: $AtomicQueueUri$, $AtomicCommitQueueUri$, $AtomicTimeout$
 1: $M \leftarrow$ receiveMessge($AtomicCommitQueueUri$, $AtomicTimeout$)
 2: if $M$.size() $> 0$ then
 3:   if acquireXLock($AtomicCommitQueueUri$, $ClientId$) then
 4:     for all $m$ in $M$ do
 5:       $LogMessages \leftarrow$ receiveMessage($AtomicQueueUri$, $\infty$, $m$.Key-1,
          $m$.Key+1)
 6:       execute basic commit protocol for LogMessages (Algorithm 1)
 7:       for all $l$ in $LogMessages$ do
 8:         deleteMessage($AtomicQueueUri$, $L$.MessageId))
 9:       end for
10:       deleteMessage($AtomicCommitQueueUri$, $m$.MessageId))
11:     end for
12:     releaseXLock($AtomicQueueUri$, $ClientId$)
13:   else
14:     abort() //Database in recovery
15:   end if
16: end if
```

GSI requires that every transaction is started by a call to BeginTransaction, presented in Algorithm 7. BeginTransaction checks first if recovery is required, to ensure that a consistent state exists. Afterwards, the protocol retrieves a valid SnapshotId from the Advanced Counter service. This ID represents the snapshot the transaction is working on.

Algorithm 7 GSI - BeginTransaction

```
Require: $DomainUri$
 1: execute advanced atomic recovery protocol
 2: $SnapshotId \leftarrow$ getHighestValidatedValue($DomainUri$)
```

To ensure that every read/write is performed on the correct snapshot it is also required to apply or rollback logs from the PU queues. Algorithm 8 shows the necessary modifications to the BufferManager for Pages. If the BufferManager receives a get request for a Page, it also requires the SnapshotId. Depending on the fetched Page being older or younger than the SnapshotId, log records are rolled back or normally applied. To enhance efficiency and reliability, we again assume Advanced Queues, where the client sets the message key to the SnapshotId during the commit. The interested reader might have noticed, that this also requires a slight modification of the Atomicity protocol: Log messages sent to the ATOMIC Queue require to hold on to the SnapshotId as well. This is required in the case of a recovery.

The commit shown in Algorithm 9 forms the last step of the GSI. It consists of two steps, the validation phase and the actual commit phase. The validation phase ensures that we do not have two conflicting writes for our snapshot whereas the commit phase is the actual commit. To validate a transaction correctly it is required that no other transaction validates the same record at the same time. The protocol ensures this by acquiring a lock for every record in the writeset (line 4-9). If it is not possible to acquire a lock for one of the records, the transaction aborts. Deadlocks are avoided by sorting the

Algorithm 8 GSI BufferManager - Get Page

```
Require: $SnapshotId$, $PageUri$
1: $P \leftarrow$ fetch(PageUri)
2: if $SnapshotId < P$.SnaptshotId then
3:     $M \leftarrow$ receiveMessage($P$.Uri, $\infty$, $SnapshotId$, $P$.SnaptshotId + 1)
4:     rollback $M$ from $P$
5: else if $SnapshotId > P$.SnaptshotId then
6:     $M \leftarrow$ receiveMessage($P$.Uri, $\infty$, $P$.SnaptshotId, $SnapshotId$ + 1)
7:     apply $M$ to $P$
8: end if
9: $P$.SnaptshotId $\leftarrow SnapshotId$
```

records according to the page Uri and RecordId. Once all locks are acquired, the protocol checks the PU queues for conflicting updates. Therefore, a CommitId is received by incrementing the Advanced Counter. Then, every PU queue which might contain conflicting updates is checked by retrieving all messages from the the used SnapshotId up to the CommitId. If one of the received log records conflicts with one of the records in the writeset the transaction aborts. Otherwise, the validation phase was successful and the actual commit phase starts. The commit is similar to the one of the Advanced Atomicity protocol - as only difference, the CommitId is set as the message key. Once the atomic commit finishes, some cleanup is performed; that is, locks are released and the CommitId gets validated to become a new valid Snapshot (lines 19 - 23). The TransactionTimeout starts with the commit and not with the BeginTransaction. This is an optimization applied to allow for long-running read requests.

Finally, to fully enable GSI the counter and lock timeout have to be chosen carefully. If they are too small, it is still possible that a client works on an inconsistent snapshot. To guarantee consistency in the sense of snapshot isolation, the timeouts should be set to a value bigger than 2 * TransactionTimeout + AtomicityTimeout + NetworkLatency. As this might be quite a long period, higher concurrency in snapshot isolation is achieved by explicitly releasing locks and validating the counter.

For the purpose of snapshot isolation, log records can be garbage-collected from the archive of log records if all the transactions using the snapshot id of the log record or a younger id have either committed or aborted. Keeping track of those transactions can be achieved by using an additional counter. Alternatively, garbage collection can be done by enforcing the read and write time to be part of the transaction time. Log records older than the TransactionTimeout can then be automatically garbage-collected.

Furthermore, our algorithms are extended to reuse an existing snapshot if it is determinable that a transaction is read-only. Doing so allows answering read-transaction from the cache and therefore reducing transaction time and cost.

5.5 2-Phase-Locking

Next to snapshot isolation we also implemented 2-Phase-Locking (2PL) which allows serializability. Our 2PL protocol is rather traditional: It uses the lock-service to acquire read- and write-locks and to propagate read- to write-locks. For simplicity, we abort transactions if they are not able to receive a lock. This is only possible because we do fine-grained locking on record-level and do not expect a lot of conflicts. For other scenarios, waiting might be required together with deadlock detection on the locking

Algorithm 9 GSI - Commit

```
Require: $DomainUri$, $SnapshotId$, $ClientId$, $WriteSet$
 1: $TransactionStartTime \leftarrow$ CurrentTime()
 2: //Validation Phase
 3: sort $WriteSet$ according to the PageUri and RecordId
 4: $P \leftarrow \emptyset$
 5: for all records $r$ in $WriteSet$ do
 6:     $P$.add($r$.Page)
 7:     $RecordUri \leftarrow$ concat($P$.Uri, $R$.Id)
 8:     if NOT acquireXLock($RecordUri$, $ClientId$) then
 9:         abort()
10:     end if
11: end for
12: $CommitId \leftarrow$ increment($DomainUri$)
13: for all pages $p$ in $P$ do
14:     $M \leftarrow$ receiveMessage($p$.Uri, $\infty$, $SnapshotId$, $CommitId$)
15:     if $M$ contains a log for a item in WriteSet then
16:         abort()
17:     end if
18: end for
19: //Commit Phase
20: execute advanced atomic commit protocol
21: validate($DomainUri$, $CommitId$)
22: for all record $r$ in $WriteSet$ do
23:     $RecordUri \leftarrow$ concat($P$.Uri, $R$.Id)
24:     releaseXLock($RecordUri$, $ClientId$)
25: end for
```

service. Atomicity for the 2PL works in the same lines as the GSI except that the TransactionTimeout starts with the begin of the transaction and not with the commit. So also read-only transactions underly a timeout, which can be an important drawback for certain scenarios. As the protocol design closely follows the text-book description, we do not discuss it in further detail.

6 Implementation on Amazon Web Services

This section describes the implementation of the API of Section 2.4 using services available from Amazon. Although we restrict it to Amazon, other providers usually offer similar features and allow for adopting the discussed methods.

6.1 Storage Service

The API for the reliable storage service was already designed in the lines of storage services from Google (BigTable), Amazon (S3) and others. It just requires a persistent store with eventual consistency properties for storing large objects. The mapping of the API to Amazon S3 is straightforward (compare Section 2.2.1) and not discussed in detail. However, note that Amazon makes use of a concept called Bucket. Buckets can be seen as big folders for a collection of objects identified by the Uri. We do not require buckets and assume the bucket name is some fixed value.

6.2 Machine Reservation

The Machine Reservation API consists of just two methods, *start* and *stop* of a virtualized machine. Consequently, Amazon Elastic Compute Cloud (EC2) is a direct fit to implement the API and additionally offers much more functionality. Furthermore, EC2 also enables us to experiment with alternative cloud service implementations and to build services which are not offered directly in the cloud. This not only saves a lot of money as transfer cost between Amazon machines is for free, but also makes intermediate service requests faster due to reduced network latency.

6.3 Simple Queues

Simple Queues do not make any FIFO guarantees and do not guarantee that all messages are available at all times. One way to implement Simple Queues is using Amazon SQS although SQS is not reliable because it deletes messages after 4 days. If we assume that for any page a checkpoint happens in less than 4 days, SQS is suitable for implementing Simple Queues. Unfortunately, another overhead was introduced with the SQS version from 2008-01-01; that is, it is not possible to delete messages directly with the message id. Instead, it is required to receive the message before deleting it. Especially for the atomicity this change of SQS made the protocol to cause more overhead and consequently to be more expensive. Alternatively, Simple Queues can be implemented by S3 itself. Every message is written to S3 using the page URI, a timestamp and ClientId to guarantee a unique message id. By doing a prefix scan per page URI, all messages can be retrieved. S3 guarantees reliability and has ”no” restriction on the message size, which allows big chunks (especially useful for atomicity). It is therefore also a good candidate for implementing Simple Queues. Last but not least, Simple Queues can be achieved by the same mechanism as the Advanced Queues, see next section.

6.4 Advanced Queues

Advanced queues are most suited to be implemented on top of EC2. The range of possible implementations is huge. A simple implementation is to build simple in-memory queue system and hold several replicas on different EC2 instances. This gives some reliability although it is not perfect. More advanced versions involve replication over data-center boundaries, flushing data to S3, using Paxos protocols [21] [10] or multiphase protocols [25]. Advanced Queues can also be achieved by combining SQS with some counter services and fail-over strategies. However, building reliable queues is research for itself and we restrict our implementation to simple in-memory, replicated queues.

6.5 Locking Service

AWS does not directly provide a locking service. Instead, it refers to SQS to synchronize processes. Indeed, SQS can be used as a locking service, as it allows to retrieve a message exclusively. Thus locks can be implemented on top of SQS by holding one queue per lock with exactly one message. The timeout of the message is set to the timeout of the lock. If a client is able to receive the lock message, the client was able to receive the lock. Unfortunately, Amazon states that it deletes messages older than 4 days and even queues that have not been used for 30 consecutive days, thus, it is required to renew lock messages within 4 days. This is especially problematic as creating such a message is a critical task. A crash during deletion/creation can either result in an empty queue or in a queue with two lock messages. We therefore propose to use a locking service on EC2. Implementations for such a locking service are wide-ranging: The easiest way is to have a simple fail-over lock service. Thus one server holds the state for all locks. If the server fails it gets replaced by another server with a clean empty state. However, this server has to reject lock requests, until all the lock timeouts since the failure have expired. Hence, all locks would have been automatically returned anyway. The lock manager service does not guarantee 100 percent availability, but it guarantees failure resilience. Other possibilities include more complex synchronization mechanisms like the ones implemented in Chubby [9] or ZooKeeper [1]. Again, we implement the simplest solution and build our services on top of EC2 using the described fail-over protocol. Developing our own locking service on EC2 gives the additional advantage of easily implementing shared, exclusive locks and the propagation from shared to exclusive without using an additional protocol. However, it seems quite likely that such a locking service will be offered by many cloud providers in the near future. Literature already states that Google, Yahoo already use such a service internally.

6.6 Advanced Counters

As for the locking service, using EC2 is the best way to implement advanced counters. If a server which hosts one (or several) counter(s) fails, then new counter(s) must be established on a new server; To always ensure increasing counter values, counters work with epoch numbers. This implies that if the counter fails, the epoch number is increased. Every counter value is prefixed with this epoch number. If a machine instance fails, a new machine replaces the service with a clean state, but with a higher epoch number. In other words, the counter service is not reliable and when it fails it loses its state. Again, like for the lock service, the fail-over time has to be longer than the counter-validation time. This ensures for GSI that requesting the highest validated value does not reveal an inconsistent state.

7 Performance Experiments and Results

7.1 Software and Hardware Used

We implemented the cloud API of Section 2.3 on AWS (S3 and EC2) as discussed in the previous section. Furthermore, we implemented the protocols presented in Sections 4 and 5 on top of this cloud API and the alternative client-server architecture variants described in Section 3. This section presents the results of experiments conducted with this implementation and the TPC-W benchmark.

More specifically, we implemented the following consistency protocols:

- *Naïve:* As in [7], this approach is used as a baseline. With this protocol, a client writes all dirty pages back to S3 at commit time, without using queues. This protocol is subject to lost updates because two records located on the same page may be updated by two concurrent clients. As a result, this protocol does not even fulfill eventual consistency. It is used as a baseline because it corresponds to the way that cloud services like S3 are used today.
- *Basic:* The basic protocol depicted in Figure 2. As stated in Section 4, this protocol only supports *eventual consistency*.
- *Atomicity:* The atomicity protocol of Section 5.1 in addition to the *Monotonicity* protocols which are specified in detail in [7] on top of the *Basic* protocol.
- *Locking:* The Locking protocol as described in Section 5.5. This protocol implements strong consistency.
- *Snapshot Isolation:* The Snapshot Isolation protocol as described in Section 5.4. This protocol fulfills the Snapshot Isolation level as defined in [4].

As discussed in Section 3, there are several alternative client-server architectures and ways to implement indexing on top of cloud services like AWS. In this study, the following configurations were used:

- *EU-BTree:* The whole client application stack of Figure 1 is executed on “end user” machines; i.e., outside of the cloud. For the purpose of these experiments, Linux boxes with two AMD 2.2 GHz processors located in Europe were used as such “end user” machines. In this configuration, client-side “B-tree” indexes are used in order to effect indexing; that is B-tree index pages are shipped between the EU machines and S3, as specified in Section 4.
- *EU-SDB:* The client application stack is executed on “end user” machines. Indexing is effected using SimpleDB (Section 2.2.4).
- *EC2-BTree:* The client application stack is installed on EC2 servers. On the “end user” side, only TPC-W requests are initiated. Indexing is done using a “client-side” B-tree; that is, EC2 servers ship B-tree index pages from and to S3.

- *EU-SDB:* The client application stack is installed on EC2 servers. Indexing is carried out using SimpleDB.

All five consistency protocols studied (Naïve, Basic, Atomicity, Locking, and Snapshot Isolation) support the same interface at the record manager as described in Section 3.2. Furthermore, all four client-server and indexing configurations support the same interfaces. As a result, the benchmark application code is identical for all variants.

In all experiments reported, (TPC-W benchmark) requests were initiated from a single machine running a single process (and thread) located in Europe. Again, this machine which simulated an "end user" machine was a Linux box with two AMD processors and 6 GB RAM. Depending on the configuration (EU vs. EC2), this server communicated directly with basic cloud services such as S3 or communicated with an EC2 server which had the benchmark application installed and communicated in turn with basic cloud services. In the scalability experiments, we used ten EC2 servers and the "end user" machine was used to issue up to 20 concurrent streams of requests.

If not stated otherwise, we used the following parameter settings for the three tuning parameters. The page size was set to 109 KB. The TTL was set to 100 secs. The checkpoint interval was set to 45 secs.

In all experiments here, response time refers to the end-to-end wall clock time from the moment a request was initated at the end user's machine until the request was fully processed and the answer was consumed by the end user's machine. Cost refers to the charges of Amazon for using EC2 and S3. Requests to advanced services such as Advanced Queues, counters, and locks were priced at USD 0.01 per 1000 requests.

7.2 TPC-W Benchmark

To study the trade-offs of the alternative consistency protocols and architecture variants, we use the TPC-W benchmark [31]. The TPC-W benchmark models an online bookstore and a mix of different so-called *Web Interactions (WI)*. Each Web Interaction corresponds to a *click* of an online user; e.g., searching for products, browsing in the product catalog, or shopping cart transactions. Overall, the TPC-W benchmark is read-heavy, but it involves a significant number of *updating* Web Interactions, too. In all experiments, we measured the *average response time* in secs for each Web Interaction and the *average cost* in milli-dollars per Web Interaction. We choose those measures and not the WIPS measures of the TPC-W benchmark in order to allow comparisons between the latency and cost trade-offs of the alternative consistency protocols and architecture variants.

Throughout this section, we do not present error bars and the variance of response times and cost. Since the TPC-W benchmark contains a mix of operations with varying profile, such error bars would merely represent the artefacts of this mix. Instead, we present separate results for read WI (e.g., searching and browsing) and update WI (e.g., shopping cart transactions) whenever necessary. As shown in [7], in general the variance of response times and cost for the same type of TPC-W operation is not high using cloud computing and we made the same observation in this performance study.

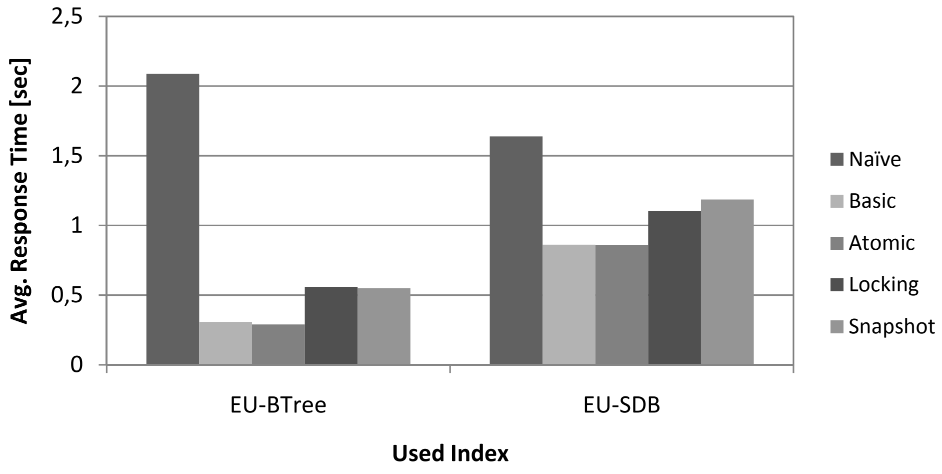

Figure 3: Avg. Response Time (secs), EU-BTree and EU-SDB

7.3 Response Time Experiments

7.3.1 EU-BTree and EU-SDB

Figure 3 shows the average response times per Web Interaction (WI) of the TPC-W benchmark for the EU-BTree and EU-SDB configurations. As shown in [7], the variance of response times (and cost, reported in later experiments) is not high; of course, there are differences between read WIs (e.g., browsing) and update WIs (e.g., shopping cart transactions), but within each category of Web Interactions the variance is low. This observation could be confirmed throughout the experiments reported in this paper so that the error bars are not shown.

Comparing the alternative protocols, the results are not surprising. For read-only Web Interactions (e.g., browsing in the TPC-W benchmark), all protocols behave in exactly the same way. The differences are due to updating Web Interactions (e.g., putting an item into a shopping cart). In order to highlight these differences better, Figure 4 shows the average response time for the update Web Interactions only. It becomes clear that *Naïve* which is used as a baseline, has the worst performance because it writes all updated pages directly to S3 for update actions (e.g., putting items into the shopping cart), which is expensive. The *Basic* and *Atomic* protocols have similar response times because they have similar commit routines. In fact, *Atomic* has a lower response time because it is faster to send log records to the ATOMIC queue using batching than to send each individual log record to the various PU Queues (Section 5.1). Furthermore, in the Atomic protocol of Algorithm 3, sending the log records to the PU Queues (Lines 6-9) is done in an asynchronous way so that that is not part of the response time as seen by the end user. (It does add to the *cost*, as shown in Section 7.4.) The *Locking* and the *Snapshot Isolation* protocols take obviously longer than *Basic* and *Atomic*: Somewhat surprisingly, the performance of these two protocols is almost the same, with *Locking* being a bit faster.

Comparing the client-side B-Tree and the SDB configuration in Figure 3, it can be seen that for most protocols (all, except *Naïve*), the ”BTree” configuration is about twice as fast as the ”SDB” configuration. Here, the differences can be explained due to caching as part of the read WIs. B-Tree pages can effectively be cached so that many

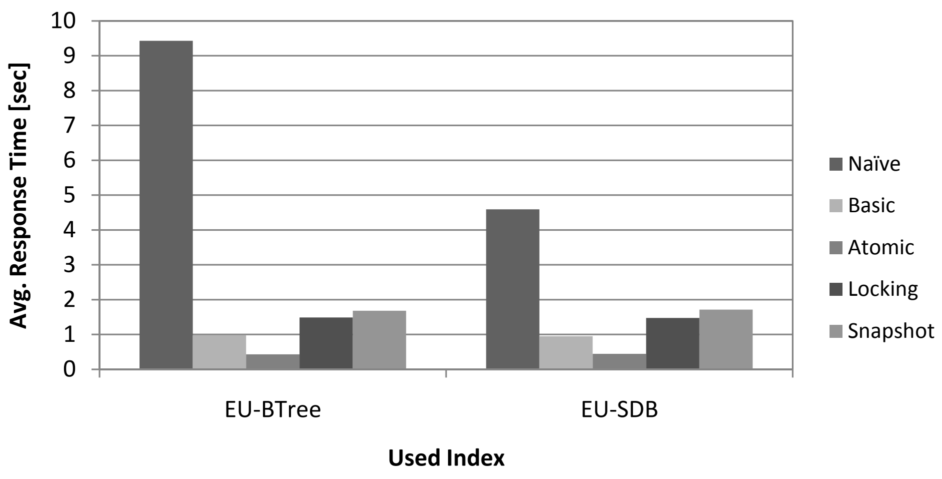

Figure 4: Avg Write Response Time (secs), EU-BTree and EU-SDB, Update WIs Only

index probes can be executed by the client without any interaction with S3. On the other hand, probing a SimpleDB index always involves communication with the Amazon's SimpleDB service because SimpleDB information cannot be cached as effectively. As shown in Figure 4, the caching effect is only advantageous for probing: Updates are as expensive with the client-side B-tree as with SimpleDB because it takes roughly as much time to send the update log records for B-trees to the corresponding queues as it is to directly update the SimpleDB index. An exception is *Naïve* for which it is much more expensive to write whole index pages back to S3 (in addition to whole data pages) than to update the SimpleDB index directly.

7.3.2 EC2-BTreee and EC2-SDB

Figure 5 shows the average response time per WI for the alternative consistency protocols on the EC2-BTree and EC2-SDB configurations. Again, since all the consistency protocols behave in the same way for read interactions, Figure 6 shows the average response times, considering update interactions only.

Comparing the different consistency protocols, the same trends can be observed for the EC2 configurations as for the EU configurations and for the same reasons: *Naïve* has the worst performance, *Basic* and *Atomic* have the best performance, and *Locking* and *Snapshot* are somewhere in between. In the EC2 configuration, however, the differences between the *Basic*, *Atomic*, *Locking*, and *Snapshot* protocols are much smaller than for the EU configurations because response times are dominated by the communication between the end user's machine and the EC2 server: Compared to that, the Amazon-internal communication between an EC2 server and S3 is fast so that the additional communication required for the *Locking* and *Snapshot* protocols does not hurt performance in a significant way. The only exception is *Naïve* which causes significant latency even in the EC2 to S3 communication because it ships whole pages, rather than log records only.

Comaparing the BTree and SDB configurations in Figure 5, it can be seen that again BTree is faster than SDB (except for *Naïve*), but that the differences are smaller. The reason is that both EC2 to SimpleDB and EC2 to S3 communication is fast as compared to EC2 to end user machine communication so that the performance impact

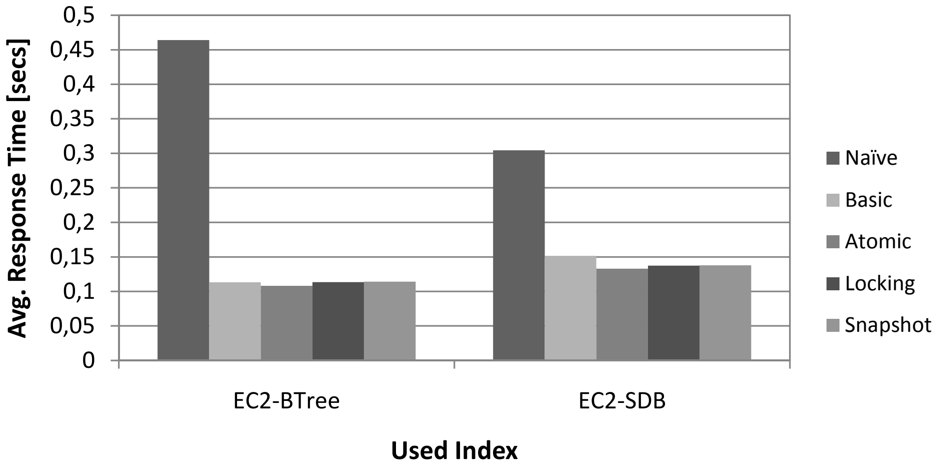

Figure 5: Avg Response Time (secs), EC2-BTree and EC2-SDB

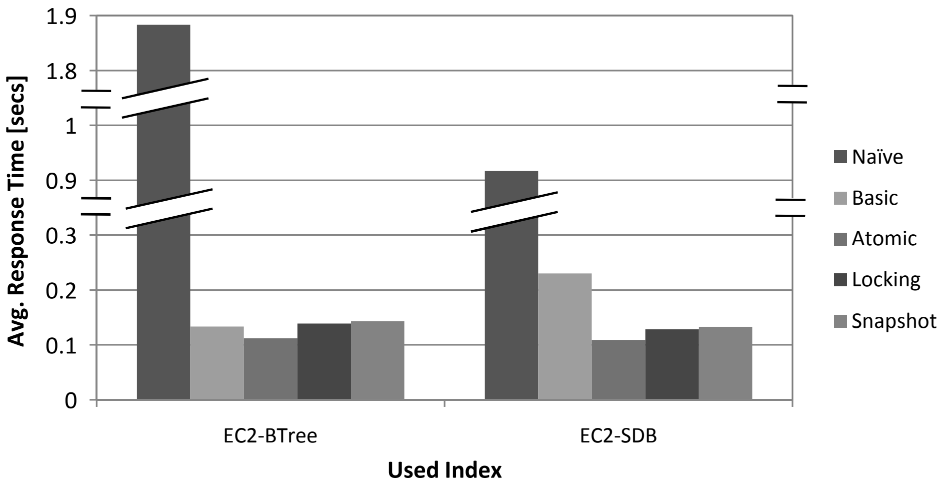

Figure 6: Avg Response Time (secs), EC2-BTree and EC2-SDB, Update WIs Only

of BTree vs. SDB is relatively small in this experiment.

Comparing Figures 3 and 5, it can be seen that significantly better response times can be achieved in the EC2 configurations than in the EU configurations. Again, this result is not surprising and can be explained easily. Since the EC2 machines are much closer to the data, the EC2 configurations achieve lower response times in the same way as *stored procedures* are more efficient than regular client-server interactions in a traditional DB application scenario. In other words, running the client application stack of Figure 1 on EC2 machines is a form of *query shipping* (i.e., moving the application logic to the data).

7.4 Cost of Consistency Protocols

7.4.1 EU-BTree and EU-SDB

Figure 7 shows the average cost per WI for the alternative protocols in the EU configurations. The findings can be summarized in a fairly straight-forward way: The cheapest protocols are *Naïve* and *Basic* because they pay no or little fees for carrying out synchronization or coordinating commits using cloud services such as Simple Queues, Advanced Queues, Counters, or Locks. The other protocols are significantly more expensive because of their extensive interaction with cloud services in order to achieve higher levels of consistency. Furthermore, in terms of cost, there is almost no difference between the BTree and SDB configurations.

One of the most important findings of this work is that *response time* and *cost* are not necessarily correlated when running a Web-based database application in the cloud. Comparing Figure 7 with Figure 3, this observation becomes clear. The *Atomic* protocol does very well in the response time experiments because most of its work (flushing ATOMIC Queues to PU Queues and checkpointing) are carried out asynchronously so that the extra work has no impact on the response times of Web Interactions as experienced by users. Nevertheless, this work must be paid for (independently of whether it is executed asynchronously or synchronously). Likewise, the BTree configurations did better than the SDB configurations in the response time experiments because of caching. While caching was important to improve performance, caching has only little effect in order to reduce cost in these experiments. The reason is that network transfer (as saved with the help of caching) is very cheap in the Amazon pricing model as compared to the per request cost in order to, e.g., enqueue or dequeue a message. Obviously, the pricing model can change in the future, but the important observation is that it is in general not possible to predict cost in $ from other performance metrics such as response time or throughput.

For brevity, the break-down between read and write transactions is not shown. In terms of overall cost, write transactions clearly dominate, again, because of the high fees for requests to cloud servers such as queues, counters, and locks in order to orchestrate the updates. Again, this observation is in contrast to the findings of Section 7.3: In the response time experiments, the overall response times were dominated by read requests because the read requests were much more frequent.

7.4.2 EC2-BTree and EC2-SDB

Figure 8 shows the cost per WI in milli-dollars for the various consistency protocols in the EC2 configurations. Comparing the consistency protocols and comparing the client-side B-tree implementation with indexing using SimpleDB, the same observa-

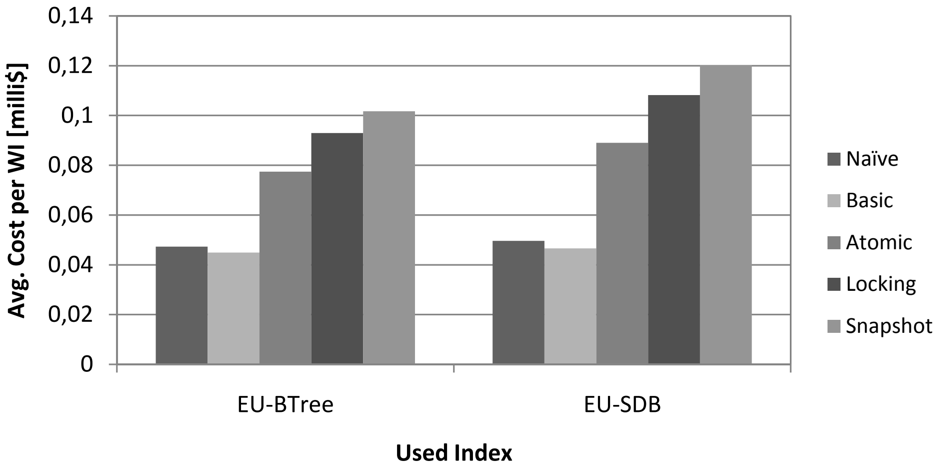

Figure 7: Cost per WI (milli$), EU-BTree and EU-SDB

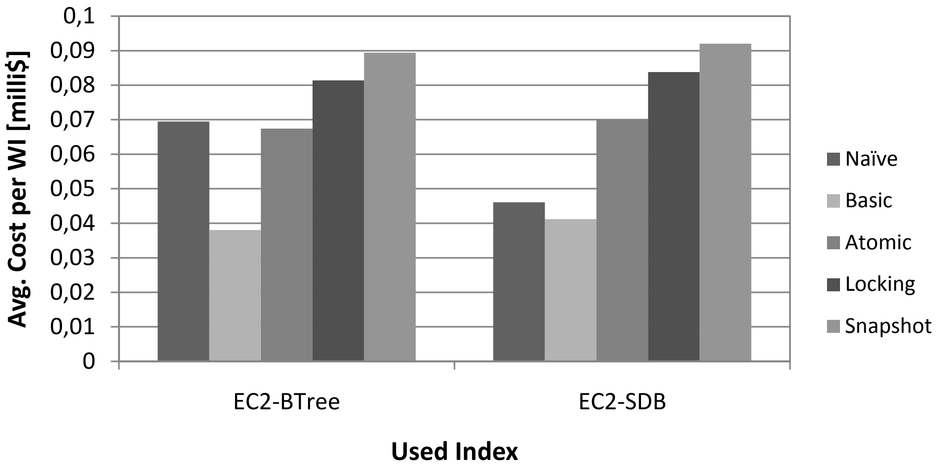

Figure 8: Cost per WI (milli$), EC2-BTree and EC2-SDB

tions can be made as for the results shown in Figure 7. The most interesting observations can be made by comparing Figure 8 with Figure 7. The expectation would be that pushing the client application stack on to EC2 servers would increase the cost because EC2 server must be paid whereas the end users' machines are free resources. Surprisingly, this does not seem to be the case as the cost are almost identical. In fact, in these experiments, the EC2 configurations are even a bit cheaper in some cases.

The first reason for the price competitiveness for the EC2 configurations was that the cost to rent CPU cycles on EC2 servers is relatively small compared to the cost per request in order to interact with other cloud services (S3, queues, counters, and locking services). As a result, using the free end user machines does not have such a cost impact. The reason why EC2 is even cheaper than EU is an artefact of our particular experiments: As mentioned in Section 4, checkpoints are carried out periodically given a specified *checkpoint interval*. In all our experiments, the *checkpoint interval* was set to 45 seconds. Since EC2 runs more TPC-W transactions per second than EU (Section

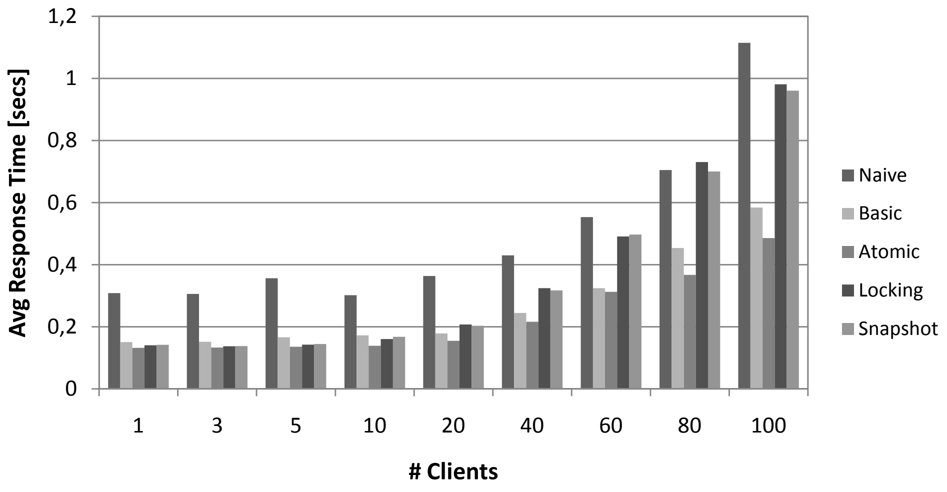

Figure 9: Avg. Response Time (secs), EC2-SDB, Vary Number of TPC-W Clients

7.3), EC2 runs more TPC-W transactions per checkpoint. Since checkpoints are a significant cost factor, EC2 gets a significant advantage from this artefact. It would have been fair to have different checkpoint interval parameter settings for the EC2 and EU configurations, but we did not do that for the sake of uniformity.

7.5 Vary Load on EC2 Server

In all the previous experiments, there was only one concurrent client so that the EC2 server was only lightly loaded in the EC2 configurations. This section studies the response times and cost characteristics of an EC2 server with increasing load (i.e., multi-programming level).

Figure 9 shows the average response time per WI of the TPC-W benchmark for a varying number of concurrent clients. In this experiment, the number of concurrent clients was varied from 1 to 100 because only one EC2 server was used. It hast to be mentioned, that the experiment was carried out without a wait time between requests. Obviously, the higher the multi-programming level (i.e., number of concurrent clients), the higher the average response time. Overall, an EC2 server is easily able to sustain about 20 concurrent TPC-W clients. After that, the response time degrades visibly.

Figure 10 shows the cost per WI for the alternative protocols, thereby varying the number of clients. Here, only the cost induced for renting the EC2 server is reported: The costs for other cloud services (e.g., queues) are factored out. As mentioned in Section 7.4, the cost for other cloud services dominate the overall AWS bill so that factoring these costs out of Figure 10 highlights the differences in cost induced by varying the number of clients. Not surprisingly, Figure 10 shows that the cost per WI decreases with an increasing number of concurrent clients. The reason is obvious because the EC2 server must be paid for, independently on how heavily it is used (Section 2.2.2). Again, at 20 concurrent TPC-W clients, the EC2 server is saturated so that at this point the best $ / WIPS results can be achieved.

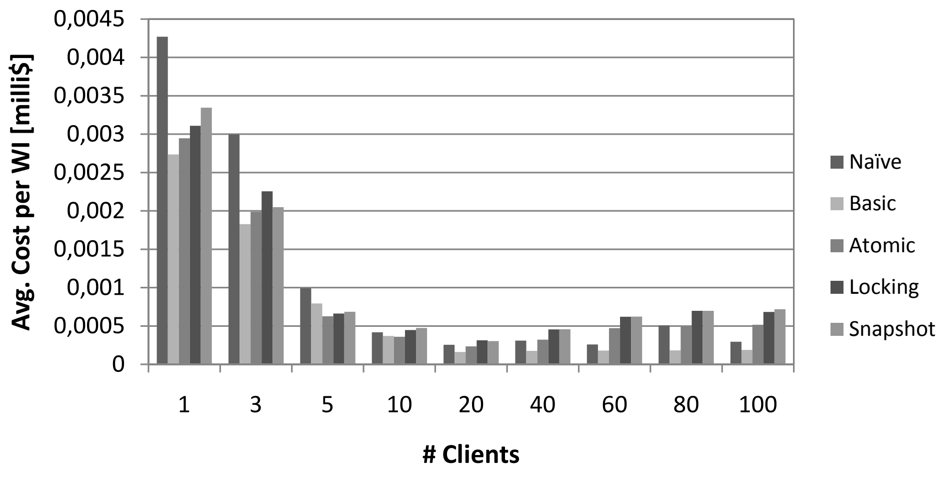

Figure 10: Cost per WI (milli$), EC2-SDB, Vary Number of TPC-W Clients

7.6 Tuning Parameters

Reconsidering the discussions of Sections 4 to 6, there are three important tuning parameters:

- *Checkpoint Interval:* Defines the interval when the PU messages from the queues are written to the page. The lower the value is set, the faster updates become visible.
- *Page Size:* Page size as in traditional database systems.
- *TTL:* Time to live is another parameter to control the freshness of the data. It determines the time a page in the cache is valid.

The Checkpoint Interval parameter was already studied in [7]. The results of [7] are directly applicable to the experimental environment used in this work. The Checkpoint Interval is an effective way to trade cost with freshness of data. The remainder of this section studies alternative settings for the other two parameters.

7.6.1 Page Size

Figure 11 shows the cost per WI for the alternative protocols in the EU-BTree configuration with a varying pagesize. With an increasing pagesize, the cost decreases slightly, but the effects are not significant. This observation could also be made for all other configurations. The reason for this slight decrease is that less interactions with S3 are needed with an increasing page size. On the negative side, higher costs for network bandwidth are incurred, but again, network bandwidth is cheap in AWS compared to the per request costs.

Figure 12 shows the average response time with a varying pagesize, again in the EU-BTree configuration. With an increasing page size, the response time drops, again because a larger pagesize reduces the number of interactions with S3. For the EU-BTree, the reduction in response time can be quite significant with an increasing pagesize; for the other configurations this effect is less pronounced. If the client application

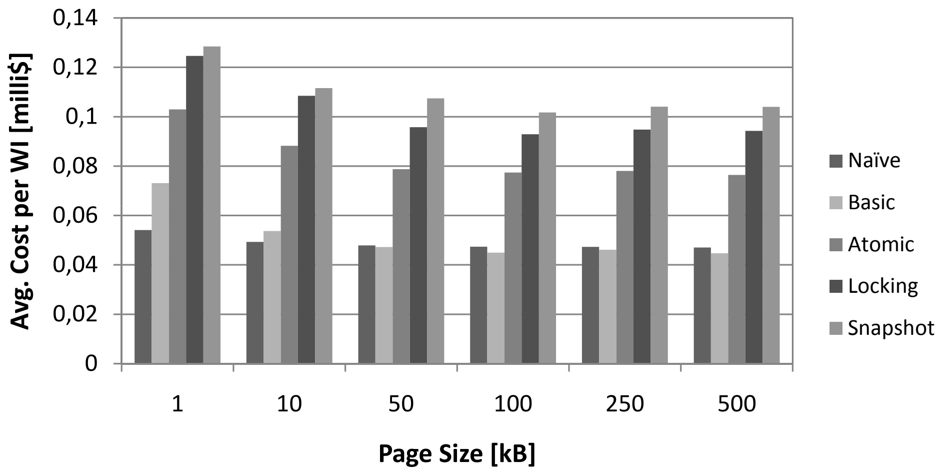

Figure 11: Cost per WI (milli$), EU-BTree, Vary Page-Size

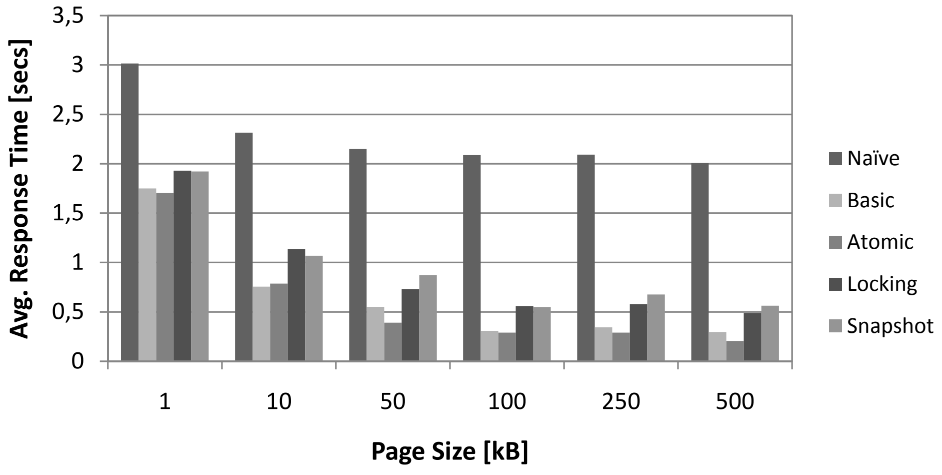

Figure 12: Avg. Response Time(secs), EU-BTree, Vary Page-Size

stack is hosted on an EC2 server, the effect is smaller because EC2-S3 communication is fast. If SimpleDB is used as an index, the effect is less pronounced because SimpleDB is probed in the granularity of individual entries so that the page size parameter is only applicable to data pages which is less significant.

7.6.2 Time-to-live

In order to complete the sensitivity analysis on alternative parameter settings, Figures 13 and 14 show the cost and response times of the alternative protocols in the EU-BTree configuration with a varying TTL parameter. The TTL parameter controls how long a page can be cached in a client's buffer pool before checking whether a new version of the page is available on S3. Obviously, the higher the TTL parameter is set, the more effective caching becomes, resulting in lower response times and lower cost. On the negative side, a high TTL parameter setting may result in stale data. Figures 13

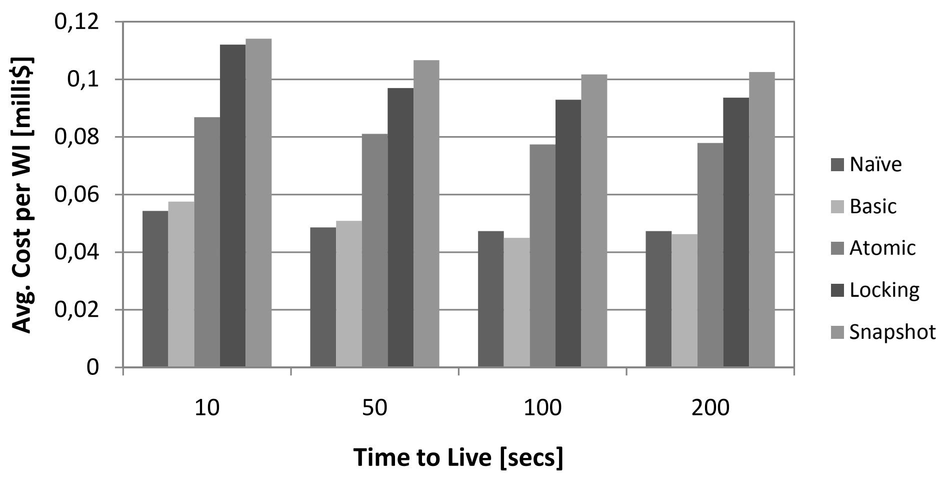

Figure 13: Cost per WI (milli$), EU-BTree, Vary TTL

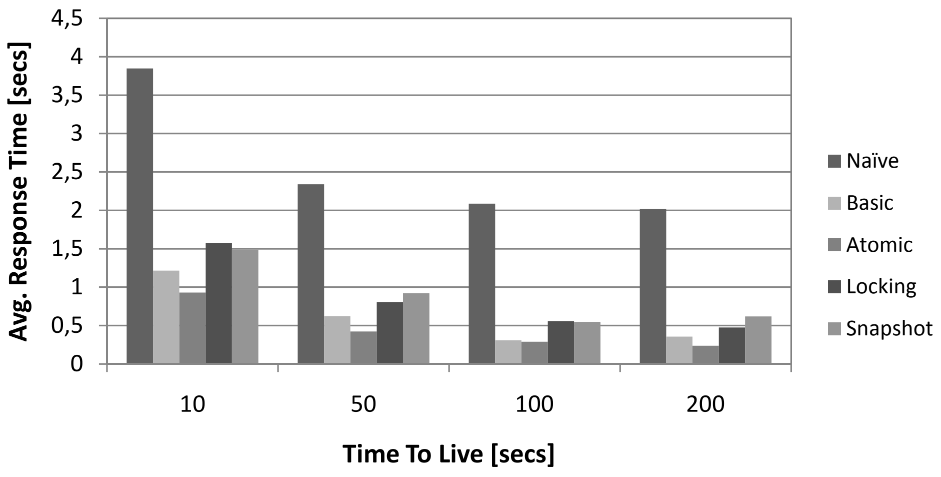

Figure 14: Avg. Response Time (secs), EU-BTree, Vary TTL

and 14 show that for the TPC-W benchmark the TTL parameter is not critical - almost all protocols have cost and response times almost independent of the TTL parameter setting. We made the same observation for all the other configurations (i.e., EC2-BTree, EC2-SDB, and EU-SDB).

7.7 Bulkloading Times

As mentioned in Section 2.2.4, SimpleDB does not offer a bulkloading facility. As shown in Table 4, this deficit has a direct impact on the time and cost it took us to create the indexes for the TPC-W experiments in this work. For SimpleDB, each *object* (e.g., *product*) must be inserted individually, which is both tedious, resulting in a high response time, and expensive, resulting in a high bill, just for bulkloading the data. With a client-side B-tree implementation, index entries can be inserted in the granularity of whole index pages which is more than one order of magnitude faster and cheaper

Index	*Time [secs]*	*Cost [Milli-Dollar]*
EU-BTree	23.3	0.2
EU-SDB	672.7	7.8

Table 4: Cost (milli$) and Time (secs) to Bulkload 1000 Products

than the ”record by record” approach.

8 Related Work

In [7] the authors studied how to build a database on S3. Compared to the work here, the architecture was specifically restricted to Amazon’s S3 and SQS. By defining a generalized utility API and adopting and formalizing the protocols from [7] to this API the results are more applicable to other providers as well. Furthermore, this paper presents and evaluates two additional protocols, locking and snapshot isolation, and their implementation on top of that reference API. In [7] the application logic was only executed on the client. This paper extends the work and also includes a study about the different client-server architectures, index-usages and their trade-offs. Additionally, various implementation possibilities for the used services are sketched out.

This work was mainly inspired by the recent development of Amazon’s Dynamo [13] and Google’s BigTable [11] towards ultra-scalable storage systems. Both systems provide just *eventual consistency* guarantees. Recently, work got published on storage services providing stronger guarantees. This includes Yahoo PNUTS [12] and Google’s MegaStore [16]. Both systems are able to work on certain snapshots of the data and have some concurrency control mechanism for a restricted group of objects. Unfortunately, both systems are not yet publically available (MegaStore is most likely the service attached to Google’s AppEngine, but not exposed as a single service) and therefore cannot be studied in detail here. Another restriction of those services is, that the consistency is only guaranteed for a smaller set of objects. As this might be sufficient for some applications, for others it might not. The protocols presented in this work do not rely on such assumptions. Nevertheless, studying the possibilities to directly guarantee different consistency levels inside the cloud storage is part of our future work.

In the distributed systems and database literature, many alternative protocols to coordinate reads and writes to (replicated) data stored in a distributed way have been devised. The authoritative references in the DB literature are [5] and, more recently, [32]. For distributed systems, [30] is the standard textbook which describes the alternative approaches and their trade-offs in terms of consistency and availability. This work is based on [30] and applies distributed systems techniques to data management with utility computing and more specifically S3. To our best knowledge, this is the first attempt to do so for S3. The only other work on S3 databases that we are aware of makes S3 the storage module of a centralized MySQL database [2].

Utility computing has been studied since the nineties; e.g., the OceanStore project at UC Berkeley. Probably, its biggest success has come in the Scientific community where it is known as *grid computing* [15]. Grid computing was designed for very specific purposes; mostly, to run a large number of analysis processes on Scientific data. Amazon has brought the idea to the masses. Even S3, however, is only used for specific purposes today: large multi-media objects and backups. The goal of this paper is to broaden the scope and the applicability of utility computing to general-purpose Web-based applications.

Supporting scalability and churn (i.e., the possibility of failures of nodes at any time) are core design principles of peer-to-peer systems [20]. Peer-to-peer systems also enjoy similar consistency vs. availability trade-offs. We believe that building databases on utility computing such as S3 is more attractive for many applications because it is easier to control security , to control different levels of consistency (e.g., atomicity and monotonic writes), and provide latency guarantees (e.g., an upper bound for all read and write requests). As shown in Figure 1, S3 serves as a centralized component which makes it possible to provide all these guarantees. Having a centralized component like S3 is considered to be a “no-no” in the P2P community, but in fact, S3 is a distributed (P2P) system itself and has none of the technical drawbacks of a centralized component. In some sense, this work proposes to establish data management overlays on top of S3 in a similar way as the P2P community proposes to create network overlays on top of the Internet.

Postponing updates is also not a new idea and was for the first time studied in the context of databases systems in [28]. The idea between log queues and differential files is quite similar, although the authors do not deal with a completely distributed environment.

9 Conclusion

Web-based applications need high scalability and availability at low and predictable cost. No user must ever be blocked by other users accessing the same data or due to hardware failures at the service provider. Instead, users expect constant and predictable response times when interacting with a Web-based service. Utility computing (aka cloud computing) has the potential to meet all these requirements. Cloud computing was initially designed for specific workloads. This paper showed the opportunities and limitations to apply cloud computing to general-purpose workloads, using AWS and in particular S3 for storage as an example. The paper showed how the *textbook* architecture to build database systems can be applied in order to build a cloud database system. Furthermore, the paper presented several alternative consistency protocols which preserve the design philosophy of cloud computing and trade cost for a higher level of consistency. Finally, an important contribution of this paper was to study alternative client-server and indexing architectures to effect applications and index look-ups.

The experimental results showed that cloud computing and in particular the current offerings of providers such as Amazon are not attractive for high-performance transaction processing if strong consistency is important; such application scenarios are still best supported by conventional database systems. Furthermore, while indeed virtually infinite scalability and throughputs can be achieved, the cost for performance (i.e., $ per WIPS in the TPC-W benchmark) is not competetive as traditional implementations that are geared towards a certain workload. Cloud computing works best and is most cost-effective if the workload is hard to predict and varies significantly because cloud computing allows to provision hardware resource on demand in a fine-grained manner. In traditional database architectures, the hardware resources must be provisioned for the expected peak performance which is often orders of magnitudes higher than the average performance requirements (and possibly even the real peak performance requirements).

In summary, we believe that cloud computing is a viable candidate for many Web

2.0 and interactive applications. With the current trends and increased competition on the cloud computing provider market, we expect that cloud computing will become more attractive in the future.

From our point of view, this work is just the beginning towards the long-term vision to implement full-fledged database systems on top of cloud computing services. This work only scratched the surface. Clearly, there are many database-specific issues that still need to be addressed. There are still a number of optimization techniques conceivable in order to reduce the latency of applications (e.g., caching and scheduling techniques). Furthermore, query processing techniques (e.g., join algorithms and query optimization techniques) and new algorithms to, say, bulkload a database, create indexes, and drop a whole collection need to be devised. For instance, there is no way to carry out chained I/O in order to scan through several pages on S3; this observation should impact the design of new database algorithms for S3. Furthermore, building the right security infrastructure will be crucial for the success of an information system in the cloud.

References

[1] Apache Software Foundation. ZooKeeper, 2008. http://hadoop.apache.org/zookeeper/.

[2] M. Atwood. A storage engine for Amazon S3. MySQL Conference and Expo, 2007. http://fallenpegasus.com/code/mysql-awss3.

[3] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Inf.*, 9(1):1–21, 1977.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. of ACM SIGMOD*, pages 1–10, Jun 1995.

[5] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, 1987.

[6] A. Biliris. The performance of three database storage structures for managing large objects. In *Proc. of ACM SIGMOD*, pages 276–285, Jun 1992.

[7] M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *Proc. of the ACM SIGMOD*, pages 251–264, 2008.

[8] D. Brunner. Scalability: Set Amazon's servers on fire, not yours. Talk at ETech Conf., 2007. http://blogs.smugmug.com/don/files/ETech-SmugMug-Amazon-2007.pdf.

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

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

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of OSDI*, pages 205–218, 2006.

[12] B. 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 *Proc. of VLDB*, 2008.

[13] 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 *Proc. of SOSP*, pages 205–220, Oct 2007.

[14] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database Replication Using Generalized Snapshot Isolation. In *Proc. of SRDS*, pages 73–84, 2005.

[15] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, Amsterdam, 2004.

[16] J. Furman, J. S. Karlsson, J.-M. Leon, S. N. Alex Lloyd, , and P. Zeyliger. *Megastore: A Scalable Data System for User Facing Applications*. 2008. Presentation at SIGMOD Products Day, Vacouver, Canada.

[17] S. Garfinkel. An evaluation of Amazon's grid computing services: EC2, S3, and SQS. Technical Report TR-08-07, Harvard University, 2007.

[18] Google. Google App Engine, 2008. http://code.google.com/appengine/.

[19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1994.

[20] J. Hellerstein. Architectures and algorithms for interent-scale (P2P) data management. In *Proc. of VLDB*, page 1244, Aug 2004.

[21] L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[22] P. Lehman and B. Yao. Efficient locking for concurrent operations on B-trees. *ACM TODS*, 6(4):650–670, 1981.

[23] D. Lomet. Replicated indexes for distributed data. In *Proc. of PDIS*, pages 108–119, Dec 1996.

[24] M. McAuliffe, M. Carey, and M. Solomon. Towards effective and efficient free space management. In *Proc. of ACM SIGMOD*, pages 389–400, Jun 1996.

[25] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Upper Saddle River, NJ, 1999.

[26] RightScale LLC. Redundant MySQL set-up for Amazon EC2, November 2007. http://info.rightscale.com/2007/8/20/redundant-mysql.

[27] W. Ross and G. Westerman. Preparing for utility computing: The role of IT architecture and relationship management. *IBM Systems Journal*, 43(1):5–19, 2004.

[28] D. G. Severance and G. M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Trans. Database Syst.*, 1(3):256–267, 1976.

[29] M. Stonebraker. The case for shared nothing. *IEEE Data Eng. Bulletin*, 9(1):4–9, 1986.

[30] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, NJ, 2002.

[31] TPC. TPC Benchmark W. Specification Version 1.8 of TPC Council, 2002.

[32] G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann, San Mateo, CA, 2002.

[33] Xen. Xen hypervisor, 2008. http://www.xen.org/.