# Building a Database on S3

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 Amazon's simple storage service (S3). S3 provides infinite scalability and high availability at low cost. Currently, S3 is used mostly to store multi-media documents (videos, photos, audio) which are shared by a community of people and rarely updated. The purpose of this paper is to demonstrate the opportunities and limitations of using S3 as a storage system for general-purpose database applications which involve small objects and frequent updates. Read, write, and commit protocols are presented. Furthermore, the cost ($), performance, and consistency properties of such a storage system are studied.

## Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design; H.2.4 [**Database Management**]: Systems—*Concurrency , Distributed databases*

## General Terms

Algorithms, Design, Experimentation, Performance

## Keywords

Cloud Computing, Database, AWS, Concurrency, Eventual Consistency, Storage System, Cost Trade-Off, Performance, SQS, S3, EC2, SimpleDB

## 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*SIGMOD'08,* June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...$5.00.

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 $\times$ 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 [17].

The goal of utility 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, as the most popular representative. Today, AWS and in particular S3 are most successful for multi-media objects. Smugmug (www.smugmug.com), for instance, is implemented on top of S3 [6]. Furthermore, S3 is popular as a backup device. For instance, there already exist products to backup data from a MySQL database to S3 [16]. In summary, S3 is already a viable option as a storage medium for large objects which are rarely updated. The purpose of this work is to explore whether S3 (and related utility computing services) are also attractive for other kinds of data (i.e., small objects) and as a general-purpose store for Web-based applications.

While the advantages of storage systems like S3 are compelling,

there are also important disadvantages. First, S3 is slow as compared to an ordinary locally attached disk drive. Second, storage systems like S3 were designed to be cheap and highly available (see above), thereby sacrificing consistency [7]. In S3, for instance, it might take an undetermined amount of time before an update to an object becomes visible to all clients. Furthermore, updates are not necessarily applied in the same order as they were initiated. The only guarantee that S3 gives is that updates will *eventually* become visible to all clients and that the changes persist. This property is called *eventual consistency* [19]. If an application has additional consistency requirements, then such additional consistency guarantees must be implemented on top of S3 as part of the application.

The purpose of this paper is to explore how Web-based database applications (at any scale) can be implemented on top of utility services like S3. The paper presents various protocols in order to store, read, and update objects and indexes using S3. The ultimate goal is to preserve the scalability and availability of a distributed system like S3 and achieve the same level of consistency as a database system (i.e., ACID transactions). Unfortunately, it is not possible to have it all because of Brewer's famous CAP theorem [10]. Given the choice, this work follows the distributed systems' approach, thereby preserving scalability and availability and maximizing the level of consistency that can be achieved under this constraint. As a result, we will not even try to support ACID transactions because we feel that it is not needed for most Web-based applications [21], whereas scalability and availability are a must. We will show how certain transactional properties (e.g., atomicity and durability) can be implemented; however, we will only sketch (Section 5.3) what it takes to implement full ACID transactions and why these approaches do not scale.

We are aware that the results presented in this paper capture merely a snapshot (November 2007) of the current state-of-the-art in utility computing. Given the success of S3, it is likely that there will be competitors with different features on the market place soon. Furthermore, it is likely that Amazon itself will add features and possibly provide new services with additional guarantees. In fact, Amazon already changed the interface of one of its services (SQS) and introduced a new service (SimpleDB) in early 2008. Nevertheless, we believe that the results presented in this paper will remain applicable because they study fundamental tradeoffs of distributed data management based on utility computing. If Amazon decides to provide additional features and guarantees, then Amazon itself will need to use similar protocols as those studied in this work.

In summary, this paper makes the following contributions: (Here S3 is used as a placeholder for more general utility computing by Amazon and other providers.)

- Show how small objects which are frequently updated by concurrent clients can be implemented using a distributed storage system like S3.
- Show how a B-tree can be implemented on top of a storage system like S3.
- Present protocols that show how different levels of consistency can be implemented using S3.
- Present the results of performance experiments with the TPC-W benchmark in order to study the cost (response time and $) of running a Web-based application at different levels of consistency on S3.

The remainder of this paper is organized as follows: Section 2 describes AWS (i.e., S3, SQS, and EC2). Section 3 presents the proposed architecture to build Web-based database applications on top of AWS. Sections 4 and 5 present the protocols to implement reads and writes on top of S3 at different levels of consistency. Sections 3-5 also cover the implementation of a B-tree on S3. Section 6 summarizes the most significant results of performance experiments with the TPC-W benchmark. Section 7 discusses related work. Section 8 contains conclusions and suggests possible avenues for future work.

# 2. WHAT IS AWS?

This section presents the functionality and properties in terms of performance and consistency of three services of the Amazon Web Services (AWS): S3, SQS, and EC2. Recently, SimpleDB was added to the AWS family of services; unfortunately, too late to be studied as part of this work. AWS is currently the most prominent provider of utility computing. AWS is used in the remainder of this study as a basis for studying the development of Web-based applications on utility computing. Other providers such as Adobe Share are beginning to appear on the market place. The results of this work are applicable to all utility services which provide a read/write interface in order to persist data in a distributed system.

## 2.1 S3

S3 is Amazon's **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 simply 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 *get-if-modified-since(uri, timestamp)* method allows to retrieve the new version of an object only if the object has changed since the specified timestamp. This feature is useful in order to implement caching based on a TTL protocol (Section 3.3). Furthermore, user-defined metadata (maximum 4 KB) can be associated to an object and can be read and updated independently of the rest of the object. This feature is useful, for instance, to record a timestamp of the last change (Section 4.5).

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. 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 per 10,000 *get* requests, USD 0.01 per 1,000 *put* requests, and USD 0.10 to USD 0.18 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 [6].

Another reason to make aggressive use of caching is latency. Table 1 shows the response time of *get* requests and the overall

| *Page Size [KB]* | *Resp. Time [secs]* | *Bandwidth [KB/secs]* |
|---|---|---|
| 10 | 0.14 | 71.4 |
| 100 | 0.45 | 222.2 |
| 1,000 | 3.87 | 258.4 |

**Table 1: Resp. Time, Bandwidth of S3, Vary Page Size**

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

**Table 2: Response Times of SQS**

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 [9].) 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 about three times as long 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 client gets when reading data from S3. 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 [11] and we adopt this technique for this study.

Amazon has not published details on the implementation of S3 and it does not give any guarantees. Taking [7] as a reference for Amazon's design principles (even though [7] 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 SQS

SQS stands for **S**imple **Q**ueueing **S**ystem. 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. (We use the HTTP interface in this work.) As of November 2007, the maximum size of a message is 256 KB using the REST-based interface and 8 KB for the HTTP interface. Any bytestream can be put into a message; there is no pre-defined schema. Each message is identified by a unique id. Based on that id, a message can be read, locked, and deleted from a queue. More specifically, SQS supports the following methods which are relevant for this work:

- *createQueue(uri):* Creates a new queue.
- *send(uri, msg):* Sends a message to the queue identified by the *uri* parameter. Returns the id of the message in that queue. The message id becomes an integral part of the message and is visible to clients which receive the message.
- *receive(uri, number-of-msg, timeout):* Receives *number-of-msg* messages from the top of a queue. *number-of-msg* must be 256 or smaller. If the queue has less than *number-of-msg* messages or only a subset of the messages are available (e.g., due to node failures at Amazon), then SQS may return less than *number-of-msg* messages. The returned messages are locked for the specified *timeout* period; i.e., the messages are not visible via *receive* calls (or other read operations) to other clients during this *timeout* period. This feature makes it possible to ensure that each message is processed at most once, and yet, avoid that messages are lost if a client fails while processing a message.
- *delete(uri, msg-id):* Deletes a message from a queue based on its message id. Typically, this method is called after a client has completely processed the message.
- *addGrant(uri, user):* Allow another user to send and receive messages to/from a queue.

In addition to these methods, there are several other methods, but none of these other methods are used in this study.

Like S3, SQS is not for free. The cost is USD 0.01 to send 1,000 messages. Furthermore, the network bandwidth costs at least USD 0.10 per GB of data transferred. 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 2 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. We have made experiments and as a rule of thumb it can be expected that SQS

returns only every tenth relevant message. For instance, if a queue contains 200 (unlocked) messages and a client asks for the top 100 messages, then SQS is likely to return about 20 messages as a result for that request. An important assumption made in this work is that no messages are ever lost. In fact, this property does not hold for SQS because SQS deletes messages after fifteen days. Fortunately, in all situations where this assumptions is made, there are work-arounds which are not reported in this paper for brevity and ease of presentation.

## 2.3 EC2

EC2 stands for **E**lastic **C**omputing **C**loud. EC2 is a service which allows clients to rent machines (CPU + disks) for a client-specified period of time. Technically, the client gets a virtual machine which is hosted on one of the Amazon servers. The cost is USD 0.10 per hour (i.e., USD 72 per month), regardless of how heavily the machine is used. One interesting aspect of EC2 is that all requests from EC2 to S3 and SQS are free. From a performance perspective, it is attractive to run applications on EC2 if the data is hosted on S3 because that way the computation is moved to the data (i.e., query shipping and stored procedures). EC2 is also attractive to implement a distributed infrastructure such as a global transaction counter (Section 5.3). The experiments reported in this paper (Section 6) were carried out using our own servers and without EC2. Studying the trade-offs of EC2 is beyond the scope of this paper, but definitely an important avenue for future work.

# 3. USING S3 AS A DISK

As mentioned in Section 2.1, utility computing with S3 promises infinite scalability, availability, and throughput. Furthermore, S3 has a similar *read / write* interface as an ordinary disk drive. Thus, S3 looks like a great candidate to implement a database. 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 S3. So, the purpose of this section is to highlight the commonalities between a disk-based and S3-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 an S3 database. This architecture has a great deal of commonalities with a distributed shared-disk database system [18]. Clients retrieve pages from S3 based on the pages' URIs, buffer the pages locally, update them, and write them back. As mentioned in Section 2.1, the unit of transfer is a *page* which contains typically several 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 S3 or using the techniques devised in [5]; 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 S3 and are, thus, considered to be part of the application. The *page manager* coordinates read and write requests to S3 and buffers pages from S3 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

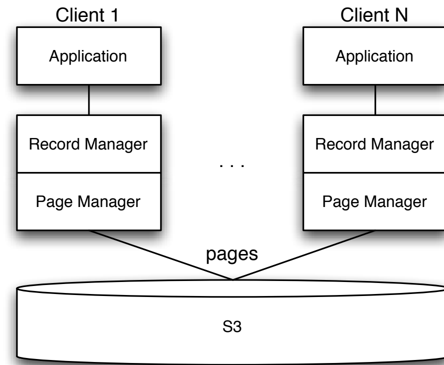


**Figure 1: Shared-disk Architecture**

in the next subsection.

Throughout this work, we use the term *client* to refer to software artifacts that retrieve pages from S3 and write pages back to S3. Depending on the application, the software architecture of Figure 1 can be implemented by different hardware configurations. For instance, the page manager, record manager, and parts of the application could be executed on EC2 or on machines in a separate data center which is the configuration used by smugmug [6] and possibly other large-scale applications. Alternatively, it is possible that the whole client stack is installed on, say, laptops or handheld computers (e.g., mobile phones) in order to implement a fancy Web 2.0 application in which users share data via S3. In such a configuration, no additional infrastructure is needed in addition to AWS in order to operate the Web 2.0 application. This configuration, therefore, fits well the needs of providers of new services on the Web, as described in the introduction. For ease of presentation, such a configuration is assumed in the remainder of this paper. Hence, if not stated otherwise, it is assumed that application, record, and page manager run on a single machine and in a single process. The techniques studied in this paper, however, are also applicable to other configurations.

All protocols proposed in this work were designed to support a large number of concurrent clients; possibly in the thousands or even millions. It is assumed that the utility provider (i.e., S3) can support such a large number of clients and guarantees (pretty much) constant response times, independent of the number of concurrent clients. Furthermore, the utility provider guarantees a high degree of availability and durability; that is, clients can read and write pages from/to S3 at any moment in time without being blocked and updates are never lost by the utility provider. All these properties must be preserved in the application stack at the client. As a result, the protocols must be designed in such a way that any client can fail at any time without blocking any other client. That is, clients are never allowed to hold any locks that would block the execution at other clients. Furthermore, clients are *stateless*. They may cache data from S3, but the worst thing that can happen if a client fails is that all the work of that client is lost. Obviously, fulfilling all these requirements comes at a cost: eventual consistency [19]. That is, it might take a while before the updates of one client become visible at other clients. Furthermore, ANSI SQL-style isolation and serialization [3] are impossible to achieve under these requirements. Following [21], we believe that strict consistency and ACID transactions are not needed for most Web-based applications, whereas scalability and availability are a must. The goal of this paper is to

bring as much DB technology as possible into this environment, to maximize consistency, and to achieve other transactional guarantees (e.g., atomicity and monotonicity). Furthermore, this paper shows the limits and implications of implementing full DB-style transactional support in such an architecture (Section 5.3).

In the remainder of this section, the record manager, page manager, implementation of indexes, logging, and security issues are described in more detail. Metadata 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 on top of S3 in the same way as the catalogue of a relational database is stored in the database itself.

## 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 are bytestreams of arbitrary length; the only constraint is that the size of the whole record must be smaller than the page size. Physically, each record is stored in exactly one page which in turn is stored as a single object in S3. Logically, each record is part of a collection (e.g., a table). In our implementation, a collection is implemented as a bucket in S3 and all the pages that store records of that collection are stored as S3 objects in that bucket. A collection is identified by a URI. The record manager provides functions to create new objects, read objects, update objects, 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 [15], and they are all applicable to an S3 database. 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. 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, uri):** Reads the payload information of a record given the key of the record and the URI of the collection.

**Update(key, payload, uri):** 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, uri):** Delete a record.

**Scan(uri):** 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 B-tree indexes (e.g., range queries); the implementation of B-trees is described in Section 3.4.

## 3.3 Page Manager

The page manager implements a buffer pool for S3 pages. It supports reading pages from S3, 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 on S3. 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 (flash or disk). If an application commits, all the updates are propagated to S3 and all the affected pages are marked as *unmodified* in the client's 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 client's buffer pool, without any interaction with S3. We use the term transaction liberally in this work: Our protocols do not give ACID guarantees in the DB sense, as motivated in Section 3.1. 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 on S3; discussing such protocols, however, is beyond the scope of this paper and rarely needed in practice.

The page manager keeps copies of S3 pages 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* request to S3 in order to get an up-to-date version, if necessary (Section 2.1).

## 3.4 B-tree Indexes

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 S3 (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 [13] and their use in a distributed system as proposed by [14] in order to allow concurrent reads and writes (in particular splits), rather than the more mainstream *lock-coupling protocol* [2]. 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 metadata in the system's catalogue on S3 (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 in S3 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 [11]. 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 S3 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 protocols sketched in Sections 5.3 require *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.

## 3.6 Security

Obviously, security is a concern in the open architecture shown in Figure 1. Everybody has access to S3, but of course, not everybody should have access to a client's personal data. Fortunately, S3 gives clients control of the data. A client who owns a collection, implemented as an S3 bucket, can give other clients read and/or write privileges to the collection (i.e., bucket) or individual pages of that collection. Unfortunately, S3 does not support fine-grained security and flexible authorization using, e.g., SQL views. To do that, an additional security infrastructure is needed which can (luckily) be implemented on top of S3, too. Designing and implementing such an infrastructure is an important avenue for future research.

One important feature of the architecture of Figure 1 is that clients need not trust the utility provider. All data can be stored in S3 in an encrypted form. In order to give different sets of clients access to different sets of pages, different encryption keys can be used. In this case, the header of the page indicates which key must be used to decrypt and re-encrypt the page in the event of updates, and this key must be shared by all the clients who may access that page.

Using SQS, furthermore, it is possible to implement several security scenarios. It is possible, for example, to assign a curator for a collection. In this scenario, all updates must be approved by the curator before they become visible to other clients. While the updates wait for approval, clients continue to see the old version of the data. Although, this paper will not come back to the implementation of such scenarios, their implementation using SQS should become clearer after reading the next section.

# 4. BASIC COMMIT PROTOCOLS

The previous section showed that an S3 database system can have a great deal of commonalities with a traditional textbook database system. 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 S3 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 S3 is used conventionally because today S3 is mostly used to store large objects so that the unit of transfer to S3 can be the object; for small objects, clustering several objects into pages is mandatory in order to get acceptable performance (Section 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 Section 5.3.

The protocols designed in this section preserve the features of utility computing with S3: clients can fail anytime, clients can read and write data at constant time, clients are never blocked by concurrent clients, and distributed Web-based applications can be built on top of AWS only, 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 SQS.[1]

- In the second step, the log records are applied to the pages stored on S3. We call this step checkpointing.[2]

[1] In fact, S3 itself could also be used in order to implement a queue. We use SQS because it is cheaper for our purposes.

[2] We use the word *checkpointing* for this activity because it applies updates from one storage media (SQS) to the persistent storage (S3). 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.

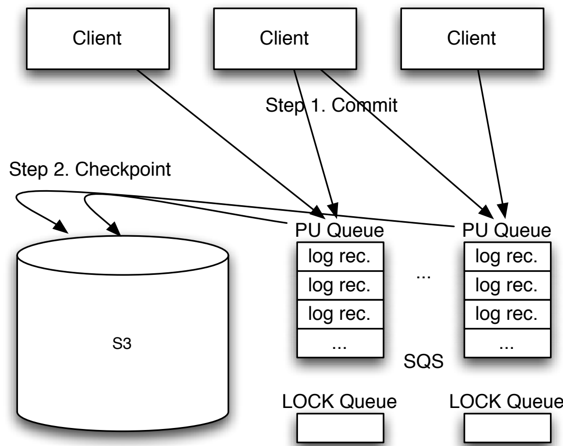


**Figure 2: Basic Commit Protocol**

This protocol is extremely simple, but it serves the purpose. Assuming that SQS is virtually always available and that sending messages to SQS 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. Furthermore, no additional infrastructure is needed to execute this protocol; SQS and S3 are both utility services provided by AWS, and the hope is that similar utility services will be supported by other providers soon.

The simple 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 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 simple protocol of Figure 2 preserves all the features of utility 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* [19]. 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 (Section 6.5). Another way to increase the freshness of data (at increased cost) is to allow clients to receive log records directly from SQS, before they have been applied to S3 as part of a checkpoint. For brevity, this latter technique is not elaborated in more detail in this paper.

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

## 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 on B-trees.

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 2.2). One solution to synchronize checkpoints is to designate machines to carry out checkpoints on particular PU queues. This approach is referred to as *watchdog* or *owner* in Section 4.5, but it is not used in this work because it does not degrade gracefully in the event that one of these designated machines fail (Section 4.5). The alternative is to allow any client which has write permission to carry out checkpoints and to implement a lock in order to guarantee that two clients do not checkpoint the same PU queue concurrently.

As shown in Figure 2, such a lock can be implemented using SQS. The idea is to associate a LOCK queue to each PU queue. Both, the PU queue and the LOCK queue are created at the time a new page is created in S3. At every point in time, a LOCK queue contains exactly one message which can be regarded as a *token*. This token message is created when the LOCK queue is created and nobody is allowed to *send* messages to a LOCK queue or *delete* the token message. When a client (or any other authority) attempts to do a checkpoint on a PU queue, it tries first to receive and lock the token message from the LOCK queue of that PU queue. If the client receives the token message, then the client knows that nobody else

is concurrently applying a checkpoint on that PU queue and proceeds to carry out the checkpoint. As part of the *receive* request to the LOCK queue, the client sets a timeout to lock the token message. During this timeout period the client must have completed the checkpoint; if the client is not finished in that timeout period, the client aborts checkpoint processing and propagates no changes. If the client does not receive the token message at the beginning, the client assumes that somebody else is carrying out a checkpoint concurrently. As a result, the client terminates the routine.

In summary, *update* log records are applied in the following way:

1. *receive(URIofLOCKQueue, 1, timeout)*. If the token message is returned, continue with Step 2; otherwise, terminate. The *timeout* on the token message should be set such that Steps 2 to 4 can be executed within the timeout period.

2. If the leaf page is cached at the client, refresh the cached copy with a *get-if-modified* call to S3. If it is not cached, *get* a copy of the page from S3.

3. *receive(URIofPUQueue, 256, 0)*. Receive as many log records from the PU queue as possible. (256 is the upper bound for the number of messages that can be received within one SQS call.) The timeout can be set to 0 because the log records in the PU queues need not be locked.

4. Apply the log records to the local copy of the page.

5. If Steps 2 to 4 were carried out within the timeout specified in Step 1 (plus some padding for the *put*), *put* the new version of the page to S3. If not, terminate.

6. If Step 5 was successful and completed within the timeout, delete all the log records which were received in Step 3 from the PU queue using the *delete* method of SQS.

In Step 4, 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 nodes. In our implementation, splitting pages is carried out along the lines of [14] so that clients which read a page are not blocked while the page is split. In Step 5, the *put* method to S3 is considered to be atomic. The token from the LOCK queue received in Step 1 need not be unlocked explicitly; the token becomes automatically visible again, after the timeout expires.

This protocol to propagate updates from an SQS queue to S3 is safe because the client can fail at any point in time without causing any damage. If the client fails before Step 5, then no damage is made because neither the state on SQS nor on S3 have changed. If the client fails after Step 5 and before the deletion of all log records from SQS in Step 6, 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*. At 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.

Setting the timeout in Step 1 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. On the other hand, a short timeout enables frequent checkpoints and, thus, fresher data. For the experiments reported in Section 6, a timeout of 2 seconds was used.

## 4.4 Checkpoint Protocol for B-trees

As mentioned in Section 4.2, there is one PU queue associated to each B-tree for *inserts* and *deletes* to 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 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 [14] 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 2.1) 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 wall-clock 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 are studied in Section 6.5. 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 Section 5.3; 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 S3 with the help of SQS. No update is ever lost, updates are guaranteed to become visible to other clients (*eventual consistency* [19]), 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), but without sacrificing the basic principles of utility computing: scalability, availability, and no need to operate an additional infrastructure. It is shown that atomicity and all client-side consistency levels described in [19] can be achieved under these constraints whereas *isolation* and *strict consistency* cannot. The protocols described in this section are layered on top of the basic protocols described in the previous section.

## 5.1 Atomicity

Atomicity involves 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 which are associated to each client. Each client maintains one or several of such ATOMIC queues on SQS; for ease of presentation, we assume that each client has a single ATOMIC queue. Rather than committing log records directly to the PU queues, the client commits the log records to its ATOMIC queue first. For this purpose, every log record is annotated with an additional field which carries an *id* of that commit: This *id* must uniquely identify a transaction on that client; different clients can use the same *id* for different transactions. For efficiency, the client packs as many log records as possible into a single message to the ATOMIC queue; it is not necessary to send the log records to the ATOMIC queue individually. Once the client has written all log records of the transaction to its ATOMIC queue, the client sends a special *commit(id)* record to the ATOMIC queue, thereby using the same *id* as in the log records. In that commit record, the client also indicates the number of log records that were committed so that they can all be recovered more safely from the ATOMIC queue. After that, the client starts submitting all log records to the PU queues just as in the basic commit protocol. If there are no failures, then the client deletes all log records from the ATOMIC queue. That is, the commit operates in three steps which must be carried out in this order:

- Send all log records to the ATOMIC queue. The *commit* record is sent last.

- Send all log records to the corresponding PU queues. Delete a message with log records from the ATOMIC queue after all the log records packed into that message have been sent to the corresponding PU queues.

- Delete the *commit* record from the ATOMIC queue.

After the first step, the commit is complete. The second and third steps can be carried out asynchronously; the application can continue and does not need to wait until these steps are completed.

When a client fails, the client checks its ATOMIC queue at restart. Winners are all log records which carry the same *id* as one of the *commit* records found in the ATOMIC queue; all other log records are losers. Losers are deleted immediately from the ATOMIC queue and never propagated to a PU queue. Winners are propagated to the corresponding PU queue and deleted after they have been propagated to the PU queue. A *commit* record may only be deleted from the ATOMIC queue after all the log records of the corresponding transaction have been propagated to PU queues. 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.

## 5.2 Consistency Levels

Tanenbaum and van Steen describe different levels of consistency in their book [19]. 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 $x$" [19]. Strict consistency can only be achieved by synchronizing the operations of concurrent clients; isolation protocols are discussed in the next section (Section 5.3). As stated in [21], strict consistency is never needed in practice and even considered harmful. This section discusses how the other (weaker) levels of consistency described in [19] can be achieved. The focus is on so-called client-side consistency models [19] because they are the basis for the design of most Web-based services [21]:

**Monotonic Reads:.** *"If a client [process]*[3] *reads the value of a data item $x$, any successive read operation on $x$ by that client will always return the same value or a more recent value"* [19]. This property can be enforced by keeping a record of the highest commit timestamp for each page which a client has cached in the past. If a client receives an old version of a page from S3 (older than a version the client has seen before), the client can detect that and reread the page from S3.

**Monotonic Writes:.** *"A write operation by a client on data item $x$ is completed before any successive write operation on $x$ by the same client"* [19]. This level of consistency can be implemented by establishing a counter for each page (or index) at a client (as for monotonic reads) and incrementing the counter whenever the client commits an update to that page (or index). The pairs *(client id, counter value)* of the latest updates of each client are stored in the header of each page and in the log records. As a result, the log records can be ordered during checkpointing and out of order log records can be detected in the event that SQS does not return all relevant records of a PU queue (Section 2.2). If an out-of-order log record is found during checkpointing, that log record is not applied and its application is deferred to the next checkpoint.

[3][19] uses the term *process*. We interpret that term as *client* in our architecture. If we would interpret this term as *transaction*, then all the consistency levels would be fulfilled trivially.

**Read your writes:.** *"The effect of a write operation by a client on data item $x$ will always be seen by a successive read operation on $x$ by the same client"* [19]. This property is automatically fulfilled in the architecture of Figure 1 if *monotonic reads* are supported.

**Write follows read:.** *"A write operation by a client on data item $x$ following a previous read operation on $x$ by the same client, is guaranteed to take place on the same or a more recent value of $x$ that was read"* [19]. This property is fulfilled because writes are not directly applied to data items; in particular, the *posting a response* problem described in [19] cannot occur using the protocols of Section 4.

In similar ways, several data-centric consistency levels defined in [19] can be implemented on S3 (e.g., FIFO consistency). Going through the details is beyond the scope of this work.

## 5.3 Isolation: The Limits

Multi-version (e.g., snapshot isolation [3]) and optimistic concurrency control (e.g., BOCC [22]) appear to be good candidates to implement isolation and, thus, strict consistency in an S3 database system. Indeed, many aspects of these protocols can be implemented without sacrificing scalability and availability. In the following, we will sketch our implementation of snapshot isolation and BOCC on S3 and show their limitations.

*Snapshot Isolation:* The idea of snapshot isolation is to serialize transactions in the order of the time they started [3]. 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. When a transaction commits, it compares its write set to the write sets of all transactions that committed earlier and started later; the intersection must be empty. To the best of our knowledge, snapshot isolation has not been implemented in a distributed system yet. The *time travel* can be implemented in the same way using S3 as in a traditional database system. The commit involves synchronization; essentially, a strict 2PL protocol must be applied on the PU queues in the *commit* phase. The real problem of implementing snapshot isolation in a distributed system is establishing a global counter in order to put a global order on all transactions. Whenever a transaction begins or commits, the transaction must increment this global counter. Such a counter can be implemented on top of S3 (and EC2), but it may become a bottleneck and is a single point of failure in the system.

*BOCC:* Like snapshot isolation, backward-oriented concurrency control [22] can be implemented in an S3-based database. Since BOCC makes stronger guarantees (it supports serializability), however, the limitations of implementing BOCC in a distributed system are even higher. Like snapshot isolation, the implementation of BOCC involves the use of a global counter to mark the beginning and commit of all transactions. Furthermore, BOCC requires that only one transaction commits at the same time. This requirement can be prohibitive with thousands of concurrent clients so that only relaxed versions of BOCC are conceivable.

# 6. EXPERIMENTS AND RESULTS

## 6.1 Software and Hardware Used

We implemented the protocols presented in Sections 4 and 5 and conducted experiments in order to study their trade-offs in terms of latency and cost ($). These are the two critical metrics when using S3 as compared to an ordinary disk drive: In all other metrics, S3

beats conventional technology. This section reports on experiments carried out with the following configurations of increasing levels of consistency:

- *Basic:* The basic protocol depicted in Figure 2. As stated in Section 4, this protocol only supports *eventual consistency*.
- *Monotonicity:* The protocols described in Section 5.2 on top of the basic protocol. These protocols support full client-side consistency (i.e., monotonic reads and writes, read your writes, and write follows read).
- *Atomicity:* The atomicity protocol of Section 5.1 in addition to the *Monotonicity* protocols on top of the *Basic* protocol. This is the highest level of consistency supported by the protocols presented in this work.

We do not study snapshot isolation and BOCC because they require additional infrastructure and do not have the same properties in terms of scalability and availability.

As a baseline, we implemented a "naïve" way to use S3 as a store. In this Naïve approach, the commit operation writes all dirty pages directly back to S3, rather than carrying out the basic two step protocol of Figure 2. This Naïve way to use S3 is subject to lost updates; so, it will not even implement the eventual consistency level achieved in the basic protocol.

All four variants studied (Naïve, Basic, Monotonicity, and Atomicity) support the same interface at the record manager as described in Section 3.2. As a result, the benchmark application code is identical for all four variants. Furthermore, the implementation of *read*, *write*, *create*, *index probe*, and *abort* operations in the record manager, page manager, and B-tree index are identical. The variants only differ in their implementation of *commits* and *checkpoints*.

This section only reports on results carried out with a single client that ran on a Mac with a 2.16 MHz Intel processor (Section 2). We also carried out experiments with varying numbers of clients. The results (cost per transaction, latency) are the same as the results with a single client, so the multi-client results are not discussed here for brevity. In all experiments reported here, the page size of data pages was fixed to 100 KB and the size of B-tree nodes was 57 KB. The TTL parameter of the client's cache was set to 100 seconds, and the cache size was limited to 5 MB. In Experiments 1 and 2, the *checkpoint interval* was set to 15 seconds. In Experiment 3, this parameter was varied. Since we are not AWS premium customers, the cost per GB of network traffic was USD 0.18 in all experiments.

## 6.2 TPC-W Benchmark

To study the trade-offs of the alternative protocols, we used a sub-set of the TPC-W benchmark [20]. The TPC-W benchmark models an online bookstore with queries asking for the availability of products and an update workload that involves the placement of orders. In all experiments reported here, we used a complex customer transaction that involves the following steps: (a) retrieve the customer record from the database; (b) search for six specific products; (c) place orders for three of the six products. In all cases, customers and products were chosen randomly.

The purpose of the experiments was to study the running times and cost ($) of transactions for different consistency levels. Experiments 1 and 2, consequently, report on running times and cost. Furthermore, the benchmark was used in order to study the impact of the *checkpoint interval* parameter on the cost.

| | *Avg.* | *Max.* |
|---|---|---|
| Naïve | 11.3 | 12.1 |
| Basic | 4.0 | 5.9 |
| Monotonicity | 4.0 | 6.8 |
| Atomicity | 2.8 | 4.6 |

**Table 3: Running Time per Transaction [secs]**

## 6.3 Experiment 1: Running Time [secs]

Table 3 shows the average and maximum execution times in seconds per transaction. The absolute numbers are high. These high execution times, however, were expected: As mentioned in Section 2.1, S3 has about two to three orders of magnitude higher latency than an ordinary local disk drive and interaction with S3 and SQS dominated the overall running times in all our experiments. Despite these high execution times, we believe that the results are acceptable in an interactive environment such as a Web 2.0 application. Each transaction simulates about twelve *clicks* of a user (e.g., searching for products, adding a product to the shopping cart) and none of these clicks (except for the *commit*) takes longer than a second.

Somewhat surprisingly, the higher the level of consistency, the lower the overall running times. The reason lies in details of the various commit protocols. *Naïve* has the highest running time because it writes all affected pages of the transactions directly to S3. The commit (and, thus, the transaction) is complete once this process has been carried out. The other approaches are faster because they propagate log records only to SQS; these log records, of course, are much smaller than the pages. *Atomicity* has the fastest commit because it sends less messages to SQS as part of a commit because the log records can be batched as described in Section 5.1. In all approaches, the latency of the *commit* can be reduced by sending several messages in parallel to S3 and SQS; in the current implementation, the messages are sent one by one.

Table 3 also shows the *maximum* running times. The variances are fairly low. In all our experiments over the last weeks, the variance of S3 and SQS was negligible. Significant variance in running times were only caused by caching effects. Furthermore, *Monotonicity* has a higher overall running time if pages must be re-read from S3 due to a consistency violation (e.g., monotonic read).

## 6.4 Experiment 2: Cost [$]

Table 4 shows the overall cost per 1000 transactions. This cost was computed by running a large number of transactions (several thousands), taking the cost measurements of AWS, and dividing the total cost by the number of transactions. Comparing Tables 3 and 4, a somewhat inverse effect can be seen. While the latency decreases with an increasing level of consistency (due to peculiarities of the protocols), the cost in $ clearly increases. For the highest level of consistency (*Atomicity*), the cost per transaction is almost twenty times as high as for the *Naïve* approach which is used as a baseline. In particular, the interaction with SQS can become expensive. A great deal of the cost is spent to carry out checkpoints and/or to process the ATOMIC queue for the *Atomicity* approach (Column 3 of Table 4). This cost can be reduced by setting the checkpoint interval to a larger value (Experiment 3); thereby reducing the freshness of data. The cost to process transactions cannot be tuned and depends fully on the protocol used and the (application and/or user-defined) activity of the transaction.

Again, the absolute values in Table 4 are less significant than the differences between the various variants. For a bookstore, a transactional cost of about 3 milli-dollars (i.e., 0.3 cents) is probably affordable because transactional costs in this order of magnitude are

| | *Total* | *Chckp. + Atomic Q.* | Transaction |
|---|---|---|---|
| Naïve | 0.15 | 0 | 0.15 |
| Basic | 1.8 | 1.1 | 0.7 |
| Monotonicity | 2.1 | 1.4 | 0.7 |
| Atomicity | 2.9 | 2.6 | 0.3 |

**Table 4: Cost per 1000 Transactions [$]**

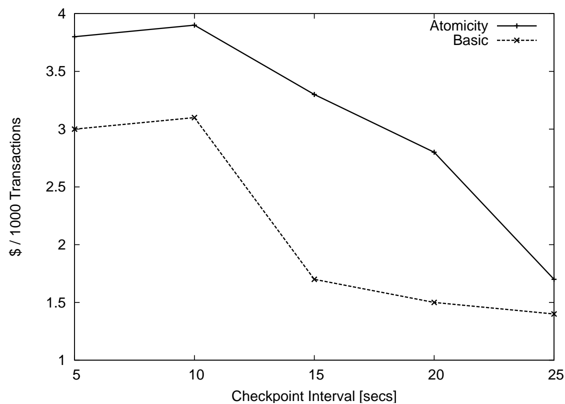


**Figure 3: Cost per 1000 Transacts., Vary Checkpoint Interval**

only incurred if the client carries out complex updates (i.e., orders books). For many Web 2.0 applications, such costs are too high so that they need to move to lower levels of consistency. Overall, however, we expect the transactional costs to decrease in the future, following Moore's law and with competitors to AWS appearing on the market place.

## 6.5 Experiment 3: Vary Checkpoint Interval

Figure 3 shows the (total) cost in $ per 1000 transactions as a function of the checkpoint interval for *Basic* and *Atomicity*. *Monotonicity* (not shown) is somewhere in between *Basic* and *Atomicity*; *Naïve* (not shown) is independent of the checkpoint interval. The running times of transactions are also independent of this parameter because checkpoints are carried out outside of transactions.

Increasing the checkpoint interval is a good way to control cost. In the workload of our experiments, a checkpoint interval below 10 seconds effectively involved initiating a checkpoint for every update that was committed; so, setting the checkpoint interval below 10 seconds does not make much sense. With a checkpoint interval above 10 seconds, the cost is quickly reduced. The curve flattens after about 20 secs; for a checkpoint interval of $\infty$, the curve would converge to about $ 7 per 1,000 transactions which is the base cost to execute a transaction in our workload (Table 4). While the overall shape of the curve is similar independent of the workload, the exact cut-off points and the best setting of the checkpoint interval depends on the workload and in particular on the skew in the update pattern. Of course, the right setting of this parameter also depends on the freshness of the data that the application requires.

# 7. RELATED 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 [4] and, more recently, [22]. For distributed systems, [19] is the standard textbook which describes the alternative approaches and their trade-offs in terms of consistency and availability. This work is based on [19] 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 [1]. As mentioned in Section 5, we do not even try to provide strict consistency and DB-style transactions because these techniques do not scale at the Internet level and because they are simply not needed for most modern Web-based applications.

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* [8]. 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 multimedia 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 [12]. 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 (Section 3.6), 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.

# 8. CONCLUSION

Web-based applications need high scalability and availability at low and predictable cost. No client must ever be blocked by other clients accessing the same data or due to hardware failures at the service provider. Instead, clients expect constant and predictable response times when interacting with a Web-based service. Utility computing has the potential to meet all these requirements. Utility computing was initially designed for specific workloads. This paper showed the opportunities and limitations to apply utility computing to general-purpose workloads, using AWS and in particular S3 for storage as an example. As of today, utility computing is not attractive for high-performance transaction processing; such application scenarios are best supported by conventional database systems. Utility computing, however, is a viable candidate for many Web 2.0 and interactive applications.

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 utility services. This work only scratched the surface and provided low-level protocols to read and write data from and to a storage services like S3. Clearly, there are many database-specific issues that still need to be addressed. We are aware that our approach to abandon strict consistency and DB-style transactions for the sake of scalability and availability is controversial and requires further discussion and evaluation. There might be scenarios for which ACID properties are more important than scalability and availability. 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 S3-based information system. Finally, EC2 and SimpleDB deserve further investigation. EC2 might be a way to reduce latency and cost for certain workloads; SimpleDB might be an attractive way to implement an index for S3 data. Unfortunately, both EC2 and SimpleDB are quite expensive.

# 9. ADDITIONAL AUTHORS

# 10. REFERENCES


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

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

[3] H. Berenson et al. A critique of ANSI SQL isolation levels. In *Proc. of ACM SIGMOD*, pages 1–10, Jun 1995.

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

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

[6] 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.

[7] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *Proc. of SOSP*, pages 205–220, Oct 2007.

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

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

[10] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *SIGACT News*, 33(2):51–59, 2002.

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

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

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

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

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

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

[17] 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.

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

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

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

[21] W. Vogels. Data access patterns in the Amazon.com technology platform. In *Proc. of VLDB*, page 1, Sep 2007.

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