CS 655: Advanced Topics in Distributed Systems
Paper Critique 4
Hessah Alsaaran

Paper
Building a database on S3. In Proceedings of the 2008 ACM SIGMOD International Conference on

Summary
Cloud computing can provide high levels of scalability, availability, and throughput to distributed
systems. According to the CAP theorem, fully consistent DB systems cannot be deployed in such settings.
In addition, it is not desired to maintain throughput and reduce cost. So the authors of this paper built a
DB system in the cloud settings for specific target applications: web-based applications that highly
benefit these three features provided by cloud computing and can settle for a weakly consistent DB. The
system was built using three Amazon services: Amazon Web Services (AWS), Simple Storage Service
(S3), and Simple Queuing System (SQS).

S3 is designed to store large objects that are rarely updated. However, the authors are defying that
by using it to store the DB records; small objects that are frequently updated. But to minimize the cost of
using S3 when retrieving or updating the records, and to increase its bandwidth, multiple records are
packed into a single object (page) that is the unit of transfer. Buckets are used to group objects in S3,
which are used here to represent collections (tables). Pages are organized as B-trees in each collection.
Also to minimize the cost incurred from updates, the authors resort to eventual consistency.

The system is built using the client-server architecture; multiple concurrent clients can retrieve or
update pages in the S3 DB. All updates (insert, delete, or update) are stored in log records that are inserted
in Pending Updates (PU) queues upon transaction commits. These PU queues are created in SQS for each
collection and records’ pages. According to a set of protocols defined in the paper, clients can retrieve
these log records and perform the updates in the S3 DB. These protocols are defined to satisfy eventual
consistency, monotonicity, and transaction atomicity.

Critique
The proposed algorithm uses idempotent log records for logging pending updates. This has the
advantage of having a single result, even if the log record was performed more than once due to fault-
tolerance. However, the update operations are simple such that the new value of the payload is passed. So
a simple increment to an attribute must read the record, increment the corresponding attribute locally, and
then commit the update log record (containing the new payload) to the queue. There are multiple
problems here. First, if updates are lazily performed, the read operation that precedes the increment will
read a stale value that might be changed by other clients but still pending in the queue. As a result, the
value written in the new payload will be wrong. So it is better to support more complex update log
records. The second problem happens when the record’s payload is large, while the required update is
small. As a result, useless information is sent over the network.

In the fifth step of the checkpoint protocol for B-trees (Section 4.4), log records are performed on
 corresponding leaf pages. However, the authors did not say that the locks for these pages are acquired to
perform the updates. It is necessary to prevent two clients concurrently updating the same page, thus one
of these versions will be lost.

The time to perform the pending updates is set to when a reader or writer accesses a page and
finds out that it has an old timestamp. An alternative to time stamping is directly checking the page’s
pending queue, and then performing the update operations asynchronously (if any) if it is unlocked. This
results in faster updates (thus better consistency) and fresher local copy of the page. The disadvantage is the probability that the number of retrieved log records from the queue is low, thus more frequent update operations and retrieve requests, which in turn increases the cost. So it’s a trade-off here.

Regarding atomicity, “all or none” of the updates compromising a transaction should be applied. The authors employed a mechanism for the “all” part, but not the “none” part. They assumed that clients always restart, but what happens if the clients don’t restart? There is no mechanism for recalling submitted logs from the queue, or reversing them if they were already applied.

To achieve monotonic writes, the algorithm tries to preserve the order of updates per client. The algorithm uses a counter for each page for each client. When a client commits an update to a page, the client-page-counter is incremented, and the client id along with the counter value is added to the log record. In addition, this pair (client id, counter value) is added (or updated thus reflecting last update by client) to the page header when updated using this log record. Using these counter values, out-of-order log records can be detected and unprocessed in the current checkpoint process. However, the problem with this approach is that the page header may get large, depending on the number of clients updating the same page. In addition, it is accumulative over time. When clients restart, they must know and use the last counter value per each accessed page (could be stored in persistent storage). A better option is to restart counting on client restart, and to have a mechanism for cleaning up the page header. This clean up also accommodates for the situation where clients may not access the page again, in which case it is useless information that can cause storage overhead.

“Read your writes” is claimed to be achieved, but it is not fully explained how it is achieved. I suspect that it is achieved for the following reason: updates are lazily done and a read operation can be done before the update is performed. Unless there is an updated local copy, which is not a good idea since it can be inconsistent because multiple clients can concurrently update the same page, which in turn caused the evolution of this algorithm and is the whole purpose of the updates queue. This point is kind of confusing to me and needs further explanation.

According to [1], S3 also uses eventual consistency to maintain all objects’ replicas. So it’s a double layer of eventual consistency model. So when the algorithm performs an update, it is not necessarily propagated to all S3’s replicas immediately. Thus concurrent access to the different replicas may result in reading different versions, even after the algorithm performed a pending update. Furthermore, the update process can start with a stale replica.

This paper was concerned about storage organization of DB records and update protocols for these records. Query processing was not in the scope of this paper, and it was considered as an upper layer in the client model (i.e. above the record manager). So when the query processor at a client decides to evaluate a query on the DB, it must first gather all the required pages from S3. This may not be efficient in terms of cost, network traffic and available memory at the client, especially when a large portion of the DB is required. When large datasets are used, it is usually better to push the computations to the data instead of pulling the data to the computation. The computation is not distributed and paralleled here.

Additional References: