Global Transactions in the Cloud with Persistent Data Structures

Anton Tayanovskyy and Adam Granicz
{anton.tayanovskyy,adam.granicz}@intellifactory.com

IntelliFactory

Abstract. One of the challenges of cloud programming is achieving the right balance between performance and consistency in a distributed database. In this paper we present a technique to obtain global transactional updates while distributing the bulk of the data in the cloud. The approach is inspired by functional programming: processes use shared memory to allocate immutable data and communicate through a single mutable cell with optimistic concurrency control. The technique is simple to implement on top of existing cloud database systems, offers good performance potential for read-intensive loads, and can partially resolve edit conflicts. In addition, it makes it easy to reason about distributed data and allows to reuse persistent data structure designs. The paper describes the technique and compares it to established approaches such as two-phase commit and eventually consistent databases. It also presents a prototype OCaml client combined with a server running on the Google App Engine[1].

Introduction

One of the challenges of cloud programming is achieving the right balance between performance and consistency in a distributed database. The CAP theorem[1, 2] shows consistency, availability and partition tolerance to be incompatible. A popular compromise is to sacrifice consistency in favor of eventual consistency. Often this is acceptable, and eventually consistent distributed key-value stores such as Amazon's Dynamo[3] are a perfect fit for such applications.

When stronger consistency is required, some form of coordination such as distributed transactions, two-phase commit or locking are called for[4]. Sub-optimal performance of these mechanisms has led major cloud computing providers to exclude global distributed transactions from their offerings. Instead, cloud database systems such as Windows Azure[2] Table Service, Google App Engine data store backed by Bigtable[5] and Amazon SimpleDB[3] provide a weaker form of local transactions, locking, or conditional updates.

The paper presents a technique that builds global transactions on top of such localized concurrency control mechanisms. Our technique is inspired by an

[1] http://www.google.com/enterprise/cloud/appengine/
[2] http://www.microsoft.com/windowsazure/
[3] http://aws.amazon.com/simpledb/

analogy with functional programming: updates always allocate a new copy of a data structure on shared memory, and mutation is restricted to modifying a single shared reference cell. We handle cell edit conflicts by an optimistic conflict resolution strategy. In terms of the CAP theorem, we sacrifice availability for consistency. In particular, in case of node or network failure our system may refuse to service update transactions while maintaining read availability.

The advantage of our technique is that it is simple to implement on top of existing cloud database systems and offers both strong consistency and good performance potential for read-intensive loads. It is especially suited to working with large infrequently updated structures, such as search indexes. The technique also makes the distributed data system easy to reason about, and lets functional programmers reuse familiar persistent data structure designs.

One of the weaknesses is the limited write throughput of the system. Due to optimistic concurrency control, all but one contending update transactions fail and are restarted. We discuss caching access to the shared memory as a simple means to improve write throughput by reducing the amount of work done by failed and restarted transactions, in effect offering partial conflict resolution.

To demonstrate the viability of implementing our design and reusing existing cloud infrastructure, the paper presents a prototype OCaml client combined with a server running on the Google App Engine[4].

Distributed Persistent Data Structures

In this section we present our distributed database design and compare it with the approaches common in the industry, including local non-distributed databases, database clusters with two-phase commit, and eventually consistent data stores.

Database Design

Our computational model involves several processes executing in parallel. The processes may be distributed and have distinct address spaces. Every process has access to unlimited distributed write-once shared memory, enabling it to read and allocate persistent data structures. Communication happens via cells, a distributed equivalent of shared mutable references. In the simplest case there is only one cell. Processes use the cells to share references to the data structures allocated on the distributed shared memory.

The key operation of interest is updating the cell. In general, it involves taking the initial value of the cell, performing lookups from the shared memory, allocating new nodes on the shared memory, and computing the final value of the cell. The goal of our design is to make the updates transactional, giving the system all four ACID properties: atomicity, consistency, isolation, durability.

We use optimistic concurrency control to provide transactional updates. Every cell has a version field and every successful update increments it. An update

[4] http://www.google.com/enterprise/cloud/appengine/

starts by checking out the current value and version of the cell. A commit succeeds if and only if the current version at commit time is the same as it was at the beginning of the update. The design ensures the ACID properties:

– **Atomicity**: Given an atomic update operation on the cells, every transaction either succeeds atomically or fails and is restarted. Failed transactions may allocate nodes on the shared memory but these nodes are not reachable by other processes and can eventually be deleted by the garbage collector.
– **Consistency**: Every time a cell is mutated, its value is set to a pointer representing a consistent data structure. Immutability plays a key role here: once allocated, the data is never mutated, preserving the consistency property. Every transaction thus has a consistent view of the data.
– **Isolation**: Since processes communicate via cells only, before a transaction commits all intermediate nodes are visible only to the process that has allocated them. It should be possible to further isolate them at the transaction level, for example by requiring that a process may execute only one transaction at a time.
– **Durability**: Our design assumes the cells and shared memory implementations to provide durability, thus making sure that a committed transaction is never undone.

Strengths

Our design aims to be simple to understand and easy to implement. It allows to reuse persistent data structures familiar to functional programmers and makes it straightforward to reason about their properties. On the implementation side, the two components the design builds upon, distributed shared write-only memory and reference cells with optimistic concurrency control, are well understood and have many implementations to choose from. Below we assume that a distributed eventually consistent key-value store is utilized as an implementation of shared memory. We also assume a suitable implementation of cells offering consistency and conditional update support.

The strengths of the design include the consistency guarantee and the absence of locking. It has good potential for scaling reads, and makes certain write performance optimizations possible. We discuss these in turn.

Read Performance There are two kinds of reads: reading the cells and reading the shared memory. Given large enough data structures, shared memory reads will dominate. The design has good performance potential here because it does not restrict sharding and replication, making it easy to add more machines to load-balance the shared memory accesses. Reusing a general-purpose distributed key-value store is one easy way to make shared memory reads scalable. A generic distributed key-value store may also be improved on by taking advantage of the data immutability property.

Write Performance While write performance also benefits from the shared memory design, it is severely limited by optimistic concurrency control. Two simultaneous transactions will force one to fail and restart execution.

One mitigating factor is the ability to cache all accesses to the shared memory on the client, which is made legal by the persistent nature of the shared memory. Caching can apply to both lookups and allocations. A restarted transaction can hit the cache and avoid redoing some of the work. This is especially useful for transactions that have a degree of independence. For an example, consider two transactions each of which is adding an entry to a map implemented as a red-black tree. The transactions execute in parallel, then one of them fails and is restarted. With caching the second run will only do a fraction of the work if it encounters shared map nodes. It should be possible to construct an example where this system would outperform the sequential execution of the transactions due to extra parallelism.

Another possible direction for improving write performance is identifying independent updates and interleaving them automatically. The theoretical framework for this is provided by the *linearizability* concept proposed by Herlihy and Wing [6]. A set of concurrent updates is linearizable if it is provably equivalent to a sequential update series, where the equivalence notion makes use of the semantics of the data structure being updated. As identifying such updates uses more information than caching, it has a greater potential to improve performance. We have not yet attempted to exploit linearizability in our prototype implementation, leaving it for future work.

Weaknesses

The primary weakness of the proposed design is the extra theoretical complexity imposed by the immutable data structures, typically requiring $O(log\ N)$ operations in place of $O(1)$ operations. Weaknesses also include issues with system availability, lack of transaction support across cells, absence of nested transactions, the need to reclaim space with garbage collection, and lack of fairness guarantees in case of contending updates.

According to the CAP theorem, a distributed database system can only achieve two of the three properties: consistency, availability, and partition tolerance. Our design is no exception: it preserves consistency and partition tolerance at the cost of availability.

In particular, network failures will cause update transactions to explicitly fail at the initiating client. For example, the server responsible for maintaining the shared cell might go down, causing both reads and writes to become unavailable. If a cluster of servers is used for this purpose, network failures can still prevent them from communicating, making writes unavailable. In this situation a typical cloud computing system will attempt to recover the failed nodes and connectivity automatically. Upon success, there will be no data or consistency loss. Higher availability can be achieved by extra redundancy in the cells and key-value store components.

More problems stem from optimistic concurrency in the handling of contending updates. Nested transactions are not supported as they would cause an infinite cascade of restarts. There are also no fairness guarantees: if update transactions are continuously started faster than they are being committed, a process can perpetually fail to commit.

Another weakness is the necessity of garbage collection to reclaim space occupied by unreachable nodes. A mark-and-sweep strategy is possible and can be done periodically in the background, impeding writes but not affecting read performance. An alternative with more predictable performance is reference counting, perhaps combined with prohibiting loops in the allocated data structures.

Alternatives

We now compare our design to some popular alternatives used by the industry to handle data in a cloud computing environment.

Local Databases A simple, conventional solution is to run a single local database without replication. With a single database, there is no need to worry about partitioning and distributed transaction support. The solution works well until the system hits availability constrains imposed either machine failure or its throughput capacity.

Master-Slave Replication Another approach is setting up a cluster where one machine acts as an authoritative master and handles all writes, and one or more machines act as slaves that help handle read traffic. This design improves upon the read availability and throughput of a local database, but has a single point of failure and a single bottleneck for write traffic. Our design has similar characteristics. However, the immutability constraint makes coordinating shared memory much easier than coordinating relational database replicas. In addition, our consistency guarantees are stronger: reading a read replica may produce incorrect out-of-date results in case of network failure, which is not possible with write-once shared memory.

Two-Phase Commit Database clusters with true distributed transactions often use two-phase commit protocol. The advantage is providing a familiar database interface while obtaining read and write availability in presence of node failures. The main disadvantage is complexity. In case of strong strict two-phase locking, the system can cause deadlocks, and requires a special transaction manager role. Commitment ordering relaxes the locking requirement but arguably retains the complexity of the design. The complexity inherent in coordinating distributed transactions with two-phase commit also limits their performance.

Eventually Consistent Data Stores Another alternative to the CAP dilemma is to drop consistency in favor of eventual consistency, obtaining better performance, availability and partition tolerance. Eventually consistent data stores are well understood and have many implementations to chose from. Despite the benefits, they are not appropriate for applications that require stronger consistency. Eventual consistency may also make application correctness more difficult to reason about. Our design aims to reclaim some of the performance benefits of these data stores while offering stronger consistency guarantees.

Implementation

We have created a prototype implementation[5] of our distributed database design in OCaml. The implementation includes a small Python service providing the shared memory and cell services on top of Google Application Engine. The implementation addresses the essentials, including concurrency control and high availability replication, but does not provide garbage collection.

Shared Memory Service

Shared memory is modeled by the following OCaml signature:

```
module type MEMORY =
sig
  type key
  val find : key -> string
  val insert : string -> key
end
```

We assume that stored data nodes can be serialized to the `string` type. Another requirement is the capability to generate new unique keys, which closely resembles the need to allocate memory in a functional language runtime.

Immutability greatly simplifies reasoning about the shared memory. The only implementation caveat involves an eventually consistent back-end: it may be required to retry failed `find` operations. This may be necessary when we manage to obtain a valid key before it propagates to all machines participating in the cloud.

Cells Service

The shared mutable cell interface takes the following form:

```
module type CELL =
sig
  val read : unit -> string
  val update : (string -> string) -> unit
end
```

[5] The code can be found at http://bitbucket.org/IntelliFactory/ifl-2011

The `read` operation returns the current state of the cell. The `update f` operation tries to replaces a state `t` with `f t`. If mismatched versions indicate that the update was preempted by another transactions, it is restarted. Both operations may fail indicating exceptional situations such as communication failure or too many update attempts.

Implementing `CELL` requires concurrency control: we need a distributed lock service, single-row transactions, or a way to perform atomic updates conditional on the value of the version field. Fortunately, these building blocks are present in several major cloud offerings, including Google App Engine, Amazon SimpleDB, and Windows Azure. They are also available from most relational database systems.

Sharded Data Structures

The obvious way to store a data structure is to serialize it and store it in a single `string`, but this approach restricts member lookup to be linear in the structure size. It is asymptotically more efficient to combine serialization with sharding.

For an example of a sharded data structure, consider the familiar purely functional lists. The list is either an empty list, or a cons-cell with a value and a pointer to the next list. We can construct a database list analogously, representing pointers as database keys, and packaging it with database access utilities in the following signature:

```
module type LIST =
sig
  type elem
  type list
  type node = Nil | Cons of elem * list
  val find : list -> node
  val insert : node -> list
end
```

We include this signature together with a functorized implementation requiring a `MEMORY` structure to model allocations on distributed memory.

These simple modifications apply to many purely functional data structures and algorithms. For another example consider maps. The following OCaml signature describes the interface:

```
module type MAP =
sig
  type key
  type value
  type map
  val empty : unit -> map
  val add : key -> value -> map -> map
  val find : key -> map -> value option
  val remove : key -> map -> map
end
```

Efficient implementations include red-black and AVL tree-based maps[7], but for demonstration purposes a simple list-based implementation will do. To derive a database implementation, let us start with normal in-memory code. Here is how the `remove` function might look like:

```
module ListMap (...) : MAP =
struct
  type map = (key * value) list
  let rec remove k m =
    match m with
      | [] -> []
      | (k', v) :: rest ->
        if k = k' then remove k rest
          else (k, v) :: remove k rest
  ...
end
```

The database version of the remove function is obtained from the in-memory version by substituting ordinary lists with database lists, and adding database lookup (assuming an `L : LIST`):

```
let rec remove k m =
  match L.find m with
    | L.Nil -> m
    | L.Cons ((k', v), rest) ->
      if k = k'
      then remove k rest
      else L.insert (L.Cons ((k, v), remove k rest))
```

Other functions are obtained analogously.

Typed Database Client

The typed database client is an OCaml functor parameterized by `MEMORY`, `CELL` and necessary serialization services, constructing a module satisfying the `DB` signature:

```
module type DB =
sig
  type t
  val read : unit -> t
  val update : (t -> t) -> unit
end
```

App Engine Backend

As a proof of concept, we have implemented `CELLS` and `MEMORY` as web services running on the Google App Engine (GAE). `MEMORY` is straightforward as the data store available to GAE applications easily subsumes the key-value store functionality. `CELLS` is slightly more involved as it requires optimistic concurrency control. Fortunately, GAE offers a transaction interface that allows to express the versioning logic. The relevant Python code is:

```
def update(entityId, version, value):
  def transact(entityId, version, value):
    entity = get(entityId)
    if entity.version == version:
      entity.value = value
      entity.version = version + 1
      entity.put()
    else:
      raise ConcurrentUpdateError()
  return db.run_in_transaction(transact, entityId, version, value)
```

It must be noted that GAE transactions are limited in scope. In particular, they cannot affect arbitrary entities, but can only edit entities within a single entity group. Entity groups behave like horizontal database table partitions, and they can be nested but not otherwise overlapping. Entity group membership is decided at entity creation time. In our `CELLS` implementation the transaction spans a single entity only, which is a trivial case of an entity; the code is therefore correct.

Other systems such as Amazon SimpleDB do not provide transactions but have a mechanism for conditional updates. Such updates atomically fail if an entity field has an unexpected value during the update. A `CELLS` service can be implemented as a conditional update by using a check on the `version` field.

Related Work

Distributed transactions have been a topic of active research since at least the 1970s [4]. Bernstein et al [8] give an good overview of the theory and issues involved in distributed databases, covering serializability, two-phase commit, atomic commitment and discussing replication. A more recent textbook treatment can be found in [9]. The idea of restricting mutation in the data store has also been attempted early, for example in the Jasmine/JStore system [10].

More recently, a number of eventually consistent systems have been proposed and implemented [11, 3, 5] in an attempt to provide higher availability and performance.

Our contribution lies in proposing an easy to implement hybrid system that combines the consistency of classical database transactions with the scalability of eventually consistent data stores.

Conclusions

We have presented a design for a distributed database with transactional updates and good read performance potential. The database can be implemented on top of standard solutions available from the major cloud computing providers, such as the eventually consistent distributed key-value stores and local transactions. The approach utilizes data structures familiar from purely functional programming by reinterpreting them as recipes for efficient distributed database partitioning. Because of the overhead of optimistic concurrency control, our approach is especially well suited to infrequently updated structures, although caching can be used to improve the write throughput of the system. Our prototype implementation shows the approach to be applicable to existing cloud technology, in particular the Google App Engine.

We conclude that restricting mutation greatly simplifies providing global transaction support in distributed databases.

As future work, we plan to extend our implementation to support multiple cloud storage providers while maintaining a uniform programming interface. Other directions include exploring undo capabilities by retaining keys of all past versions of the stored objects, exploiting linearizability of concurrent updates, investigating garbage collection algorithms for reclaiming space occupied by unreachable nodes, and benchmarking performance on problems of practical interest.

References

1. Brewer, E.A.: Towards robust distributed systems. In: Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing. PODC '00, New York, NY, USA, ACM (2000) 7
2. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News **33** (2002) 51–59
3. DeCandia, G. and Hastorun, D. and Jampani, M. and Kakulapati, G. and Lakshman, A. and Pilchin, A. and Sivasubramanian, S. and Vosshall, P. and Vogels, W.: Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review **41** (2007) 205–220
4. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21** (1978) 558–565
5. Chang, F. and Dean, J. and Ghemawat, S. and Hsieh, W.C. and Wallach, D.A. and Burrows, M. and Chandra, T. and Fikes, A. and Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS) **26** (2008) 1–26
6. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS) **12** (1990) 463–492
7. Okasaki, C.: Purely functional data structures. Cambridge University Press, New York, NY, USA (1998)
8. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)

9. Weikum, G., Vossen, G.: Transactional Information Systems - Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann Publishers (2002)
10. Wiebe, D.: A distributed repository for immutable persistent objects. SIGPLAN Not. **21** (1986) 453–465
11. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in bayou, a weakly connected replicated storage system. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles. SOSP '95, New York, NY, USA, ACM (1995) 172–182