Protocols for Online, Consistent Remastering in SLOG

Johann Miller

Abstract

We describe two protocols for maintaining consistency guarantees during remastering in the SLOG system. Remastering moves ownership of data between server regions, allowing the system to adapt to changes in access patterns. The first protocol takes a high-level approach that is computationally inexpensive, but potentially limits system throughput. The second protocol is more granular, but has a higher overhead. We prove correctness for both protocols, and compare them with benchmarks. Under normal network latency their performance was equivalent, while the lower level approach performed better in an extreme case.

Introduction

SLOG [1] is a geo-replicated database that takes advantage of regional locality in access patterns to offer low-latency on some transactions. In general, geo-replicated databases must pay a cross-region coordination cost to write data and maintain consistency guarantees (See PACELC [2]). In SLOG, entries of the database are assigned to a 'home' region. Any transaction that accesses data owned by a single region can execute without any cross-region coordination. In use cases with regional locality, data entries can be owned by regions close to the clients that access them.

However, the regional locality may not be static. For SLOG to be robust to changes in the access pattern, it must be able to remaster entries between homes. The challenge lies in coordinating the transfer of ownership between regions. To maintain the strict serializability guarantee of SLOG, only one region may own the data entry at once. Furthermore, all replicas must order all transactions from the old home before any from the new home. One way to achieve this by disabling transaction processing during the remaster (0 homes own the entry). Of course this harms both throughput latency. Here, we compare two implementations of online remastering that preserve SLOG's serializability guarantee.

Remastering Protocols

Both protocols are potential implementations of remastering in the SLOG system. The system keeps two pieces of metadata along with every data entry: the region where it is currently mastered, and a counter. The counter increases every time the data is remastered. Both these protocols use the counters to maintain strict serializability. Remastering itself is inexpensive in SLOG- it only requires updating the metadata. No data has to move, since replicas still process the transactions that they didn't master. The significant task is processing the transactions that access remastered data.

All transactions in SLOG are ordered in a local log for the region that masters the data they access. If a transaction accesses data in multiple regions, it is broken up and placed in all of the involved local logs. Local logs are shipped to all replicas, so the order of transactions within a local log is

preserved. At the replicas, the local logs are combined to form a local view of the global log. In the absence of remastering, transactions in separate local logs access mutually disjoint data, and thus don't need to be relatively ordered in the global log. However, with remastering, transactions can arrive from the local log of a new master ahead of the transactions from the old master that access the same data, and thus be inserted into the global log in an incorrect order. More details are available in [1].

These protocols resolve this situation by aborting or reordering transactions. Both protocols run on every transaction in the global log before the transaction can request locks. They only process single-home and lock-only transactions (the divided parts of mutli-home transactions). The protocols are deterministic, so each replica can run them without coordination.

Per-LocalLog Algorithm

```
VerifyCounters(txn, local_log_id):
If any counters behind current:
        Abort
Else If queue[local_log_id] not empty:
        Insert txn at the back of queue[local_log_id]
Else If any counters ahead current
        Insert txn at the head of queue[local_log_id]

RemasterOccured(old_local_log_id, new_local_log_id):
While queue[new_local_log_id] not empty:
        curr_txn = head of queue[new_local_log_id]
        If counters behind current:
                Abort curr_txn
        Else if counters ahead of current:
                Return
        Else:
                Send curr_txn for locks
```

Per-Granule Algorithm

```
VerifyCounters(txn):
if any counters behind current:
        return ABORT
else if no counters ahead:
        can_execute = true
        for key in txn.keys_in_partition:
                if queues contains key:
                        if queues[key].first().counter() <= txn.counters[key]
                                can_execute = false
                                break
        if can_execute:
                return VALID
```

```
for key in txn.keys_in_partition:
        insert txn into queues[key], in order of counters, after ties


Remaster(key, counter):

if the txn at the head of queues[key] is waiting for this counter:
        for key in txn.keys_in_partition:
                if txn is head of queues[key] and matches or below the counter:
                        Unblock txn
                        Recursively try to unblock the head of the queue for each
                        key of the txn
```

Strict Serializabilty

We show that both protocols will output equivalent transaction orders at every replica, and that the order reflects the order of submission to the system. Thus this protocol upholds SLOG's strict serializability guarantee.

Notation

T_1 *{A:(R_1, 1), B:(R_1, 2)}* is a transaction accessing keys A and B, both with master 1 and counters 1 and 2 respectively.

T_R *{A:(R_1, 1)->(R_2,2)}* is a remaster of key A from replica 1 with counter 1 to replica 2 with counter 2.

Per Local Log

Lemma 1

If T_1 is ahead of T_2 in the same local log, then at all replicas T_1 will be released first unless T_1 is aborted.

Proof

If T_2 was sent first, then it was released by the remaster manager while T_1 was blocked. But both are from the same local log, so T_2 would be behind T_1 in the queue and transactions are only unblocked from the front.

Lemma 2

T_R {A:(R_a, n)->(R_b, n+1)} is in R_a's local log. If T_1 {A:(R_a, n)} is placed later in the local log, it will be aborted at all replicas.

Proof

T_R will acquire its locks first, and update the counter to n+1. T_1's counters will be checked when it is removed from the queue (which might pass) and again once it has acquired locks. The second test will fail.

Lemma 3

If T_1 is aborted due to a low counter at one replica, it will be independently aborted at all replicas.

Proof
This abort implies that some T_R exists earlier in T_1's local log that increases the counter. All replicas see these transactions in the same order, so by Lemma 2, T_1 will abort.

Lemma 4
T_R {A:(R_a, n)->(R_b, n+1)} is in R_a's local log. T_1 {A:(R_b, n+1)} will be released after T_R.
Proof
If T_1 is released, then A has counter n+1. This implies that T_R has already executed.

Theorem
Using the simple algorithm, transactions which are relatively ordered (access overlapping keys) are released in the same order at every replica, and transactions aborted at one replica will be aborted at all.
Proof
For this not to be the case, there needs to be at least one pair of relatively ordered transactions that are reordered at different replicas. Consider any two transactions T_1 and T_2 that are relatively ordered and are not aborted. Then there are two cases:

1: They are in the same local log. By Lemma 1, the order will be maintained.
2: They are in different local logs, due to a remaster. Lemma 4 shows that T_2 will execute after the remaster transaction. If T_1 is that remaster transaction, the order is already clear. Otherwise, T_1 must precede the T_R in the local log, by Lemma 2.

Thus the order will be preserved at all replicas. The second claim is Lemma 3.

Per-Granule
Lemma 1
If T_1 is aborted at one replica, it will be aborted at all.
Proof
T_1 {B:(R_1, n)} is aborted if and only if T_R {B:(R_1, n)->B:(R_2, n+1)} is placed ahead of it in the same local log.

Lemma 2
If T_1 is ahead of T_2 in a per-key queue and neither abort, then it was submitted to the system before T_2.
Proof
Consider the queue for some key A.
Case 1: Both transactions have equal counters on A. Then T_1 was ordered before T_2 in a local log.
Case 2: T_1 {A:(R_1, n)} has a lower counter than T_2 {A:(R_1, n + c)}. Machine 1 which assigned metadata to T_2 then must have processed some T_R {A:(R_1, n)->(R_2, n+1)}, potentially followed by other remasters until counter n+c was reached. T_1 was not aborted, so it must have been ordered in the local log before the T_R.

Lemma 3
Deadlock cannot occur in the per-key queues.

Proof

Create a dependency graph where every transaction is dependent on transactions ahead of it in the per-key queue. Deadlock implies that a cycle exists in this graph. By Lemma 2, every transaction in the cycle was submitted after the transaction before it, which is impossible.

Lemma 4

Among transactions that are eventually released (no deadlock) and not aborted, relative orders will be equivalent across all replicas.

Proof

Consider any T_1 and T_2, which are relatively ordered. There are 2 cases:

Case 1: T_1 is ordered ahead of T_2 in the same local log. If the counters don't lead to deadlock, then the ordering will be deterministic. Cases:

- Case A: All shared counters of T_2 are less than T_1. In all regions, T_2 is released first.
- Case B: All shared counters of T_2 are equal to or greater than T_1. In all regions, T_1 is released first.
- Case C: Some shared counters of T_2 are less than T_1, and others are greater than or equal. Leads to deadlock, contradiction.

Case 2: The transactions are in different local logs. Since only SH or LO transactions are submitted, no counters can be equal between T_1 and T_2. Cases:

- Case A: All shared counters of T_2 are less than T_1. In all regions, T_2 is released first.
- Case B: All shared counters of T_2 are greater than T_1. In all regions, T_1 is released first.
- Case C: Some shared counters of T_2 are less than T_1, and others are greater. Leads to deadlock, contradiction.

Theorem

Using the per key algorithm, transactions which are relatively ordered (access overlapping keys) are released in the same order at every replica, and transactions aborted at one replica will be aborted at all.

Proof

By Lemma 3, all transactions will release. For any pair of relatively ordered transactions T_1 and T_2 which don't abort, there is an order that all replicas will observe by Lemma 4. Lemma 1 shows that any aborts will be observed at all replicas.

Performance

Experimental Setup

Both protocols were implemented within the SLOG system. The code is freely available at https://github.com/ctring/SLOG. For these benchmarks, the system was deployed to 3 Amazon EC2 t3.large instances in us-east (Virginia), eu-central (Frankfurt), and ap-southeast (Sydney).

Overall Benchmark

This benchmark consisted of 80% single-home and 20% multi-home transactions. Each transaction accessed 30 arbitrary keys, including 10 writes. Remaster transactions were introduced to the system at varying levels. The first result is to be expected- increasing the number of remaster transactions

raises the number of aborts in the system. When the metadata changes often, it's more likely for a transaction to arrive with an old counter.

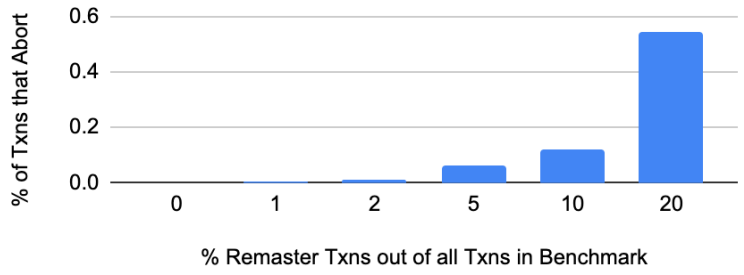

Fig 1: Aborts rise as the amount of remastering increases

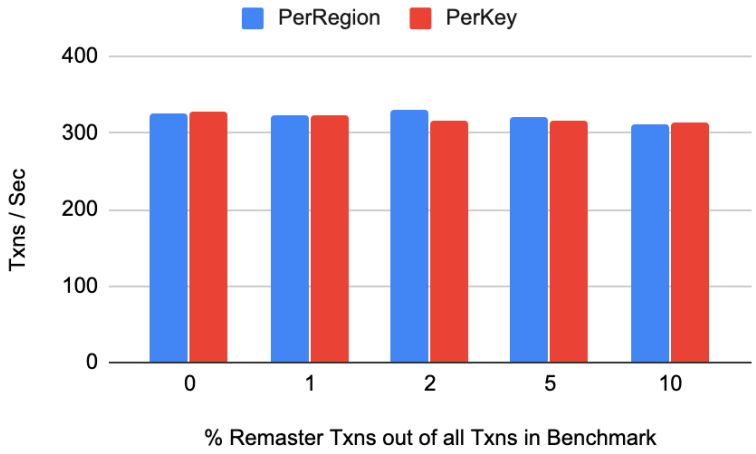

Fig 2: The protocols did not affect throughput

In this benchmark, the choice of protocol did not impact throughput of the system. To explain this, consider that the two protocols only differ once a transaction is marked as 'waiting'. This only occurs when there is a lag in processing between replicas: a transaction is sent to a region that has processed a remaster and is then sent to its home region which has not. In the 1% Remaster Txn benchmark, 0.2% of transactions were marked as 'waiting'. Thus it seems that the replica lag is not very prevalent in the deployed system.

Standalone Benchmark

To compare the protocols in the presence of replica lag, we built a standalone benchmark. 10% of all transactions are sent with high counters, followed by the remaster transactions that unblock them. Each test was run on 10,000 transactions, with 1000 keys present in the system.

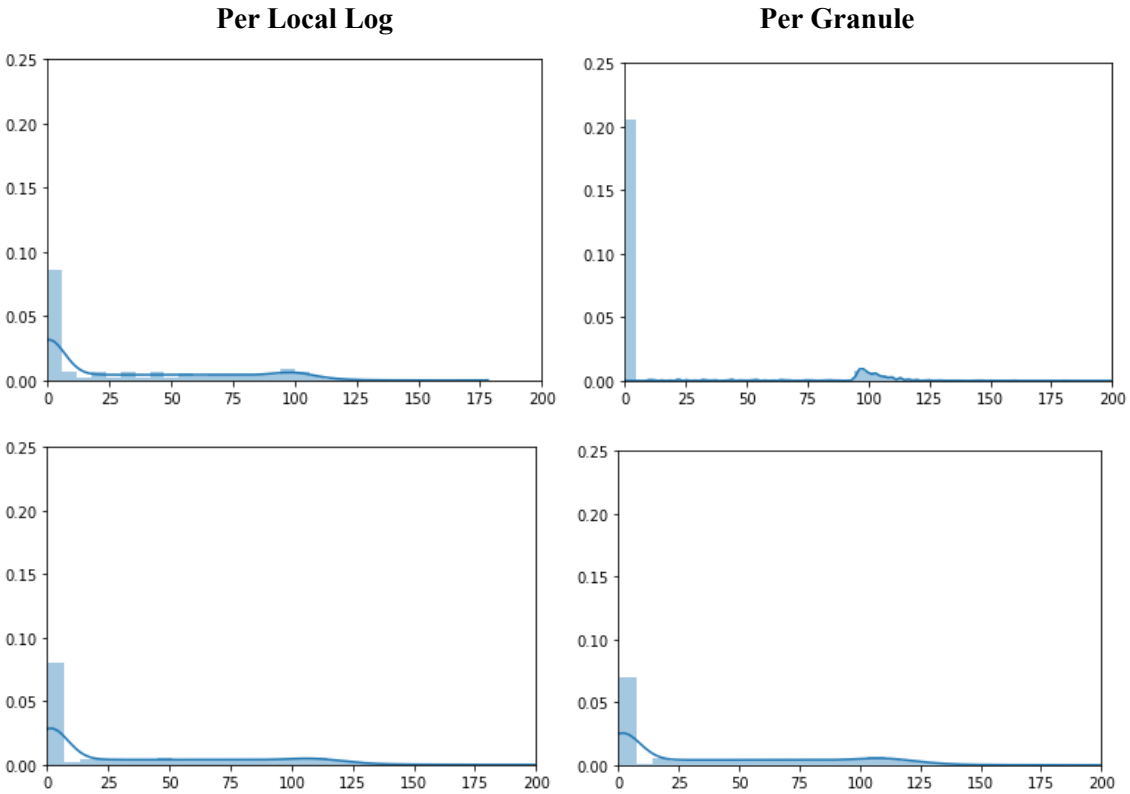

Fig 3: Latency distribution (ms). Top: 5 key access set. Bottom: 50 key access set.

Tests were run with small transactions (5 keys) and large (50 keys). The Per Granule protocol showed a clear advantage on latency for small transactions, but the protocols performed similarly for large transactions. The hump at 100 ms is an artifact of the testing setup; a constant number of transactions were placed between the high-counter transactions and the remaster transactions.

In this experiment, the lower latency did not translate to higher throughput (Fig 4). This is likely due to remastering being single-threaded. In full SLOG, lower latency here would translate to transactions beginning execution sooner, thus freeing locks for other transactions.

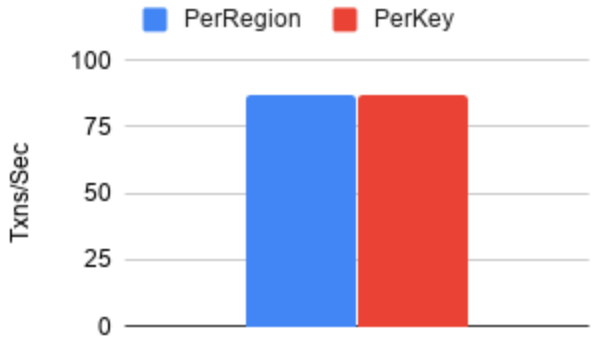

Fig 4: Throughput on 5 key access set

Discussion

We have compared two protocols for online remastering in SLOG. Both maintain SLOG's guarantee of strict serializability, and we found that both perform similarly in a standard setup. To simulate less ideal conditions with network latency or system backup, we tested the protocols in a standalone environment. Here, the lower-level protocol exhibited lower latency on small transactions. It was expected that the lower-level protocol might incur high overhead particularly for large transactions. WIth moderately large transactions (50 keys) we saw some of this effect as the protocol lost it's latency advantage.

Interestingly, any number of remastering protocols that provide equivalent ordering could be used simultaneously by different replicas in SLOG. The two protocols here could be combined to a hybrid that switches between them based on the size of incoming transactions.

These protocols lay the groundwork for dynamic responses to changes in region affinity of the data in the system. They also introduce more potential areas of study. Currently SLOG employs a simple heuristic used in PNUTs that uses the last 3 accesses to data to trigger remastering. However, it is unclear if other protocols might provide more benefit. [5] introduces heuristics which use other factors such as predicted future load to determine the optimal home for data. Compared to its predecessor Calvin [3], SLOG has additional overhead that can lower its performance in the presence of many multi-home transactions. An efficient remastering protocol combined with intelligent remastering could make SLOG extremely robust to changes in access patterns, and able to match or outperform Calvin in nearly all scenarios.

References

[1] https://www.cs.umd.edu/~abadi/papers/1154-Abadi.pdf
[2] https://www.cs.umd.edu/~abadi/papers/abadi-pacelc.pdf
[3] http://cs.yale.edu/homes/thomson/publications/calvin-sigmod12.pdf
[4] http://cs.brown.edu/~mph/HerlihyW90/p463-herlihy.pdf
[5] https://cs.uwaterloo.ca/~kdaudjee/DaudjeeICDE20.pdf

Acknowledgments

A special thanks to Dr. Daniel Abadi for his guidance on this project, and to Cuong Nguyen who contributed ideas and led the development of the SLOG code base.