A Hybrid Approach to a High Throughput, Eventually Consistent Counter for DynamoDB

Ryan Hecht Jawaad Ahmar Ria Haque

Yazan Hunjul Daniel Esemezie

Problem Description

DynamoDB's current approaches to atomic counters present significant challenges to high-throughput demands [1].

DynamoDB achieves atomicity through transactions that use an optimistic concurrency control scheme without locks. As a result, transactions do not wait for other transactions to finish before executing and instead fail when contention occurs [2]. The DynamoDB client uses a retry mechanism with exponential backoff and jitter to resolve these transaction failures, but when the maximum number of retries allowed by the client is exceeded the failure will propagate to the caller. While the potential TPS for transactions with DynamoDB's internal retry strategy is extremely high, there is a non-zero chance that transaction failure errors can propagate to the caller due to transactions that operate on hot partitions.

Many existing products cannot tolerate transaction failures because their API contracts do not include error handling for these specific failure modes. Adding new error types would require releasing a new major version of their API, breaking backward compatibility with existing users. This poses a significant business risk for companies with established user bases, requiring clients to update their integration code simultaneously or risk service disruption.

Existing solutions are incomplete and come with their own disadvantages. Distributed sharded counters reduce but do not fully eliminate contention issues, leaving the product vulnerable to failure. Eventual consistency counters avoid transaction failures but introduce unacceptable latency for applications requiring real-time counter updates. This latency is particularly problematic for services that enforce limits on user actions, such as an API that restricts the number of items a user may create in their account. When a user approaches their limit, the counter must reflect accurate values immediately to prevent limit violations. Eventually consistent counters might allow users to temporarily exceed their allotted limit during the consistency window, creating compliance issues and potentially costly overages that cannot be billed to customers due to API contract limitations.

The industry has a need for a counter solution that provides the immediacy of transactional atomic counters while maintaining DynamoDB's promise of limitless scalability and existing API contracts.

Proposed Solution

A. Solution Overview

Our solution uses both a best-effort counter and an eventual consistency counter. The eventual consistency counter is eventually-accurate and is processed through Lambda functions that consume a DynamoDB stream. The best effort counter may undercount but cannot overcount, healing itself when all services are deleted, and it is reset to zero. Since the best effort counter may only ever undercount or be accurate, the maximum of the best effort counter and the eventually consistent counter is used to enforce creation limits. This serves as an immediate heuristic until the eventually consistent counter catches up.

This design maximizes limit utilization but introduces a small risk of temporarily exceeding limits if the best effort counter undercounts by a significant amount and the eventually consistent counter has not yet stabilized.

During rapid resource creation, the best-effort counter value typically dominates as it updates immediately. This approach is optimal for scenarios where accurate counting is necessary and where minor, temporary limit overages are acceptable.

This approach guarantees limits are never exceeded but may prevent full utilization of available resources until the eventually consistent counter stabilizes. Additional research is needed to determine optimal synchronization intervals between counters and viable strategies to "heal" the best-effort counter.

B. Requirements Satisfied By The Proposed Solution

i. Exactly Once Guarantees

Exactly-once guarantees are widely recognized as impossible in distributed systems. Eventually-exactly-once processing can be broken down into two subproblems: at-most-once processing and at-least-once processing. At-most-once delivery can be achieved through idempotency but at-least-once delivery requires the premise that deliveries will eventually occur, given sufficient time. However, there is no way to guarantee that a system will deliver a message.

ii) At-Most-Once Guarantees

To achieve at-most-once processing, no record can ever be processed more than once, such that double counting never occurs. The primary challenge is that multiple Lambda instances running concurrently may try to process the same record. Our solution uses a conditional transaction that executes only if an idempotency flag on the newly created item (called “counter_processed”) has not been yet been marked as true. In deletion, the condition is whether or not the item exists in the database. If the condition on the transaction fails, then this implies that another Lambda has processed the counter operation on the item and the transaction is aborted, which prevents double counting.

iii) At-Least-Once Guarantees

To achieve at-least-once processing, every record outputted by the DynamoDB stream must eventually be processed, given sufficient time. Lambda functions may fail, time out, or be throttled. To solve this, a retry strategy is used by the DynamoDB Stream to retry failed records. When the retry strategy reaches its maximum number of attempts, the records are sent to a dead letter queue that reprocesses them. In the case of a fatal failure, records are sent to a second dead letter queue that operators can use to restore the system.

iv) Avoiding New API Errors

By decoupling the eventually consistent counter from the main API via a DynamoDB Stream, errors are handled asynchronously and are not propagated to the API or the API's caller.

v) Atomic Processing

A record must either be processed successfully by updating the counter and marking the item as processed, or fail completely. DynamoDB transactions are used to execute both operations as a single unit, ensuring atomicity within the eventual consistency counter. This introduces the potential for transaction failures, but this is ameliorated by the retry strategy.

vi) Reliability, Availability, Durability, Scalability

The system is reliable because the counter achieves eventual consistency. The system remains available because Lambda functions are spawned with extremely short cold-start times in the case of an outage. The system is made durable by its intense error handling, self-healing mechanisms, and self-recovery, which is elaborated on in the following section. Finally, the system is infinitely scalable because an infinite number of Lambda functions can spawn to meet increasing demand.

C. Implementation Details

Resource Creation Flow

1. User requests to create a resource
2. The API service creates the resource in the database with a *counter_processed=false* flag
3. The API immediately increments the best-effort counter
4. The resource creation triggers a DynamoDB stream event
5. A Lambda function processes this event from the stream:
    a. Increments the eventually consistent counter
    b. Sets the *counter_processed* flag to true

Resource Deletion Flow

1. User requests to delete a resource
2. The API immediately decrements the best-effort counter
3. The API sets a *deleted=true* flag on the resource (soft delete)
4. The resource update triggers a DynamoDB stream event
5. A Lambda function processes this event from the stream
    a. Decrements the eventually consistent counter
    b. Completely removes the resource from the database (hard delete)

Error Handling and Recovery

- All updates to the best effort counter are idempotent, meaning on creation a counter may increment by at most once
- If on creation, there is a failure after service creation, undercounting will occur, but since creation precedes counting, overcounting will not occur.
- If on deletion, there is a failure after counting and the API call is retried, undercounting will occur, but since the counter must be decremented at least once for the deleted flag to be set to true, overcounting will not occur.
- If a transaction fails in a lambda, the lambda function will respond with a failure status, and the transaction will be retried.
- If retries are exceeded, the request will be moved to a first dead letter queue where it will be re-inputted to the lambda.
- If an unexpected failure occurs, (such as the database dying completely), then the requests will be moved to a second dead letter queue where service operators can manually ameliorate the problem.
- A transaction may fail in the lambda if another lambda (operating concurrently) has already processed the counter increment or decrement (indicated by the deletion of the item or the *counter_processed* flag being set to true). In this case, the lambda will consume the failure and return a successful response, as this indicates that the counter has already been processed.
- The use of transactions and flags ensures idempotency (at most once) and atomicity, and the use of retries with a dead letter queue ensures at least.

Key Properties of The System

1. Safe Undercounting: The best-effort counter may undercount but never overcounts
2. Self-Healing: When all resources are deleted, both counters reset to zero
3. Idempotent operations in the best-effort counter: Each counter operation happens at most once
4. Transactional atomicity with idempotency: Each counter operation happens *exactly* once and only occurs if the corresponding idempotency flag change or hard-delete also occurs
5. Retries: All counter events will eventually be processed via retries
6. Distributed Safety and Unlimited Scalability: Unlimited Lambda functions can be created to process more counter requests, and to ensure that if one instance goes down the system still functions

7. Dead letter queue: If a part of the eventual consistency system fails, requests will be moved to a dead letter queue to be processed by the system when it regains health

D. Architecture Overview

Refer to Figure 1 for the architecture diagram.

DECISIONS

We made several key decisions:

- Use an eventually consistent counter: We chose to use an eventually consistent counter instead of using only a sharded counter. With a sharded counter, where the transaction must occur within the API, the transaction may fail within the API, continuing to propagate transaction failures to the caller.
- Not using a conflict free replicated data type, specifically positive-negative: We decided against using these because they added complexity, but this could be done in the future along with sharding to increase counter throughput.
- Using a Typescript API over another language (Go): After writing a test API in Go and failing to see a massive increase in TPS, because they were locally constrained by local instances of DynamoDB, a Typescript API was used. This gives a better development experience and type system.
- Using an AWS emulator over fully native local development: Despite adding complexity, it provides a better local replication of the AWS system and achieves higher TPS and a more accurate representation of AWS.
- Using Docker: We chose to use Docker for our project, even though it adds to initial development complexity, because it significantly decreases development complexity in later stages and makes transferring the finished system to AWS easier.
- Using the absence of an item in the database as an idempotency flag: We chose to do this because there is a relationship between the _ being processed and the item no longer existing in the database: an item with its deleted flag set to true will be deleted from the database if and only if the intent to decrement the counter for this item has been processed. This is guaranteed by DynamoDB's fault tolerance and rollback mechanisms.
- Conducting local load testing: Load testing on AWS was prohibitively expensive.

CHALLENGES AND FUTURE CONSIDERATIONS

Challenges

We faced several challenges including but not limited to:

- Lack of native idempotency in non-transactional operations in DynamoDB: We had to implement a custom idempotency method using conditional updates and custom records.
- Lack of high performance compute:. Our TPS during local load testing was limited by our hardware, rather than our software implementation. We were still able to achieve over 2000 TPS, which is over one counter operation per every half-millisecond, exceeding DynamoDB's one millisecond guarantee. However, to test for higher TPS, we would need to run our software on AWS, which is prohibitively expensive.
- Lack of local replicability: Local DynamoDB is effectively a wrapper around SQLite and lacks true parity with the cloud-hosted version. This leads to:
    - Inaccurate performance benchmarking: DynamoDB local doesn't reflect the same throughput behavior, contention handling, or latency characteristics as the real thing.
    - Stream limitations: DynamoDB streams do not behave identically in a local environment (especially under load) which makes testing the Lambda-based counter less deterministic.
- Concurrent Lambda execution: Preventing double-counting in concurrent stream processing was non-trivial but necessary due to the increased processing throughput with concurrent execution.
- Running AWS Lambda functions locally: This required an AWS emulator to simulate concurrent spawning and network calls. Furthermore, getting transaction failures to occur was very difficult because local DynamoDB uses locking transactions rather than optimistic concurrency protocols. It was solved by emulating DynamoDB through local stack.
- Documentation: The AWS documentation is not very detailed, requiring extensive research by reading the original white papers on DynamoDB and other AWS services.
- Satisfying the requirement that no new API errors could be introduced: If transactions can be done at the API level, then an accurate counter can be implemented using transactions with the same immediacy of a best effort counter by distributing the counters in shards and aggregating them when the limit needs to be retrieved. This can reduce transaction contention to increase throughput almost infinitely but still introduces a non-zero chance that transaction failures can occur and be propagated back to the API caller.

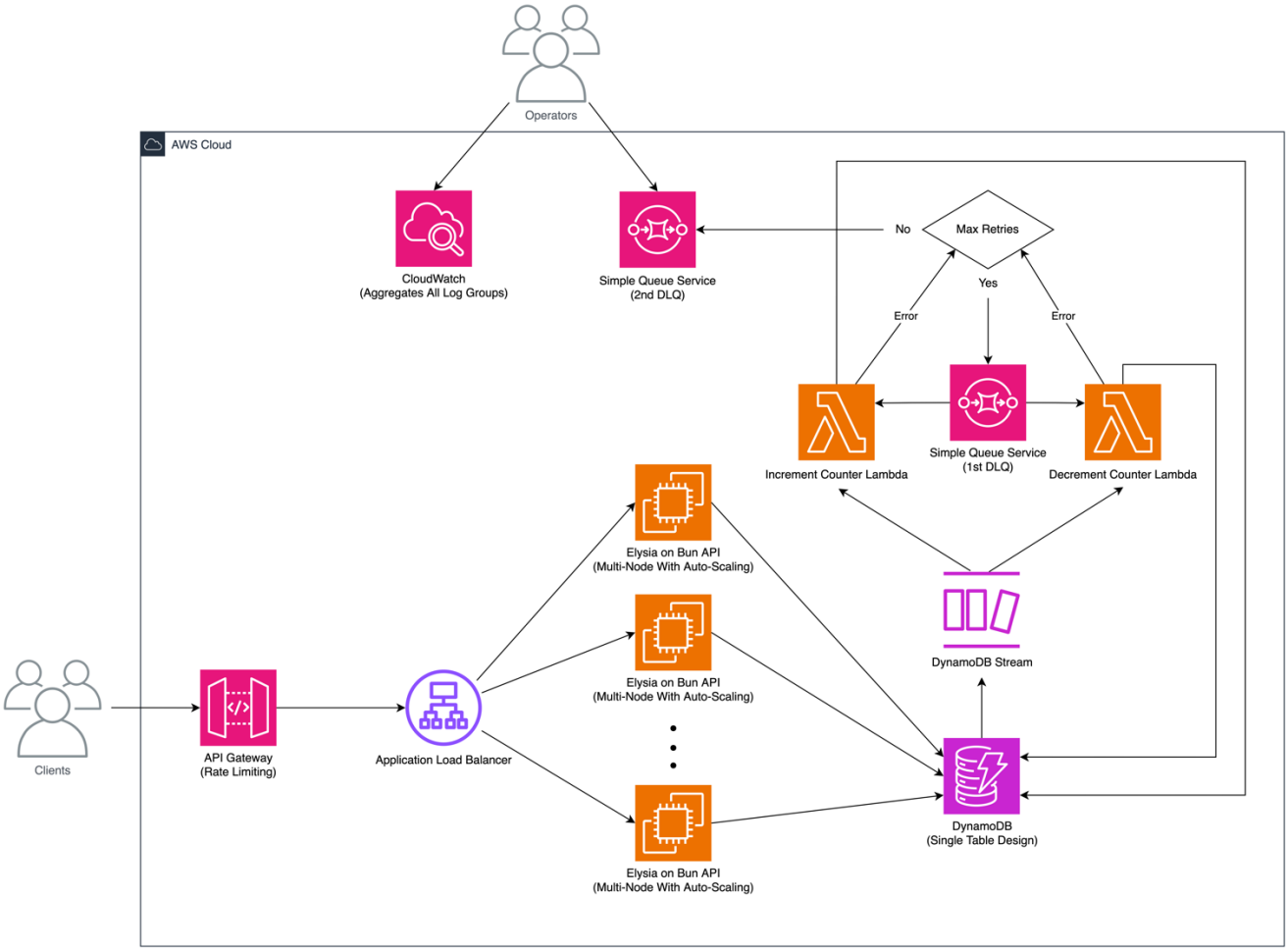

Figure 1
Architecture diagram

Future Considerations

In the future, we would like to investigate the possibility of using a conflict free replicated data type, specifically a positive-negative counter. The throughput of the eventually consistent counter would be immediately doubled in dual operation scenarios and could be further increased by using distributed sharding of the counters and aggregating them when the final count needs to be retrieved.

We would also like to investigate the potential self-healing mechanisms where the eventual consistency counter heals the best effort counter when the eventual consistency counter stabilizes. There needs to be work done to determine how the self-healing process could occur without introducing new errors to the API and how to determine when the eventual consistency counter has stabilized.

Using two counters, one for increments and one for decrements, halving contention

References

[1] Hunter, J., & Gillespie, C. (2023, March 31). Implement resource counters with Amazon dynamodb | AWS database blog. AWS Documentation. https://aws.amazon.com/blogs/database/implement-resource-counters-with-amazon-dynamodb

[2] Idziorek, J., Keyes, A., Lazier, C., Perianayagam, S., Ramanathan, P., Sorenson III, J. C., Terry, D., & Vig, A. (2023). Distributed transactions at scale in Amazon DynamoDB. In Proceedings of the 2023 USENIX Annual Technical Conference (pp. 705-717). USENIX Association. https://www.usenix.org/conference/atc23/presentation/idziorek