Anton Kropp

CassieQ: The Distributed Queue Built On Cassandra

Why use queues?

- Distribution of work
- Decoupling producers/consumers
- Reliability

Existing Queues

- ActiveMQ
- RabbitMQ
- MSMQ
- Kafka
- SQS
- Azure Queue
- others

Advantage of a queue on c*

- Highly available
- Highly distributed
- Massive intake
- Masterless
- Re-use existing data store/operational knowledge

But aren’t queues antipatterns?

Issues with queues in C*

- Modeling off deletes
  - Tombstones
- Evenly distributing messages?
  - What is the partition key?
- How to synchronize consumers?

Existing C* queues

- Netflix Astyanax recipe
  - Cycled time based partitioning
  - Row based reader lock
  - Messages put into time shard ordered by insert time
  - Relies on deletes
  - Requires low `gc_grace_seconds` for fast compaction

Existing C* queues

- Comcast CMB
  - Uses Redis as actual queue (cheating)
  - Queues are hashed to affine to same redis server
  - Cassandra is cold storage backing store
  - Random partitioning between 0 and 100

Missing features

- Authentication
- Authorization
- Statistics
- Simple deployment
- Requirement on external infrastructure

CassieQ

- HTTP(s) based API
- No locking
- Fixed size bucket partitioning
    - Leverages pointers (kafkaesque)
- Message invisibility
    - Azure Queue/SQS inspired
- Docker deployment
- Authentication/authorization
- Ideally once delivery
- Best attempt at FIFO (not guaranteed)

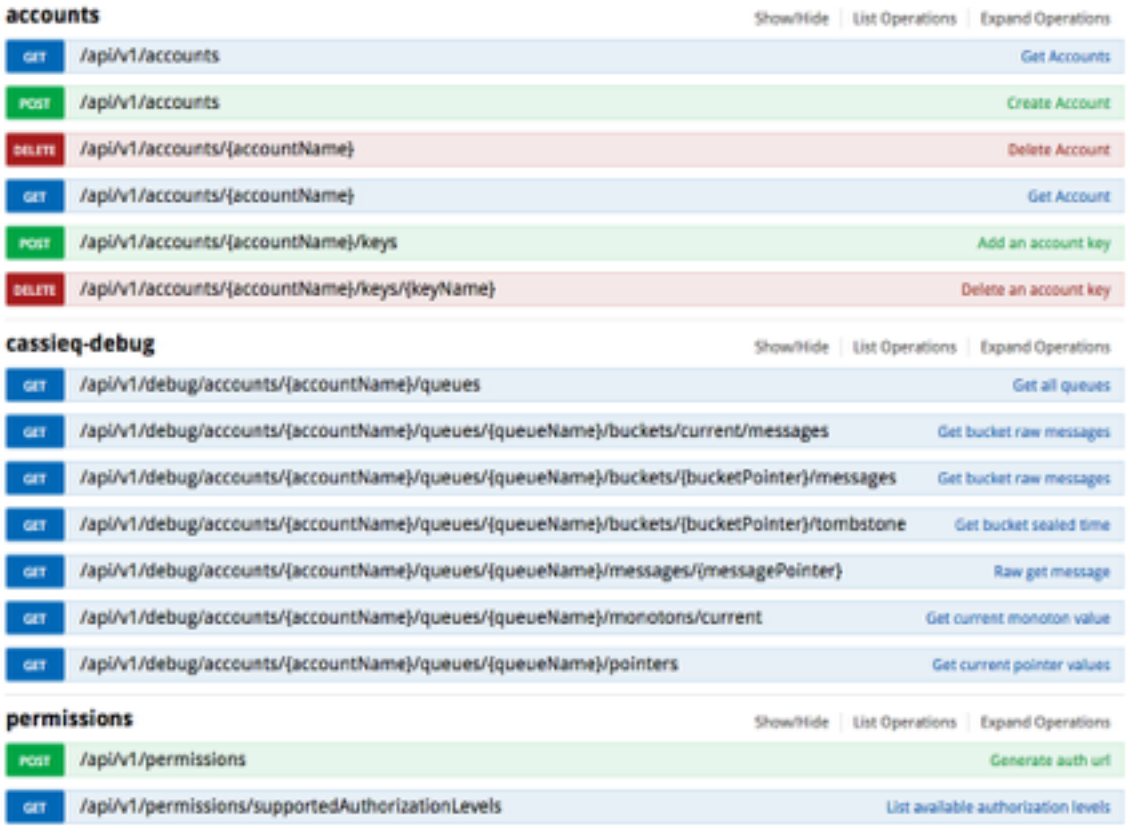
```
docker run -it \
    -p 8080:8080 \
    -p 8081:8081 \
    paradoxical/cassieq dev
```

CassieQ Queue API

cassieq Show/Hide | List Operations | Expand Operations

GET	/api/v1/accounts/{accountName}/queues	Get all account queue definitions
POST	/api/v1/accounts/{accountName}/queues	Create Queue
DELETE	/api/v1/accounts/{accountName}/queues/{queueName}	Delete queue
GET	/api/v1/accounts/{accountName}/queues/{queueName}	Get a queue definition
DELETE	/api/v1/accounts/{accountName}/queues/{queueName}/messages	Ack Message
POST	/api/v1/accounts/{accountName}/queues/{queueName}/messages	Put Message
PUT	/api/v1/accounts/{accountName}/queues/{queueName}/messages	Update Message
GET	/api/v1/accounts/{accountName}/queues/{queueName}/messages/next	Get Message
GET	/api/v1/accounts/{accountName}/queues/{queueName}/statistics	Get queue statistics

CassieQ Admin API



CassieQ workflow

- Client is authorized on an account
  - Granular client authorization up to queue level
- Client consumes message from queue with message lease (invisibility)
  - Gets pop receipt
- Client acks message with pop receipt
  - If pop receipt not valid, lease expired
- Client can update messages
  - Update message contents
  - Renew lease

```
final QueueName queueName = QueueName.valueOf("test_queue");

final AccountName accountName = AccountName.valueOf("test_account");

final CassieqApi client =
        CassieqApi.createClient(server.getBaseUri().toString(),
                                CassieqCredentials.signedQueryString("..."));

client.createQueue(accountName, new QueueCreateOptions(queueName)).execute();

client.addMessage(accountName, queueName, "hi").execute();
```

Lets dig inside

CassieQ internals

TLDR

- Messages partitioned into fixed sized buckets
- Pointers to buckets/messages used to track current state
- Use of lightweight transactions for atomic actions to avoid locking
- Bucketing + pointers eliminates modeling off deletes

CassieQ Buckets

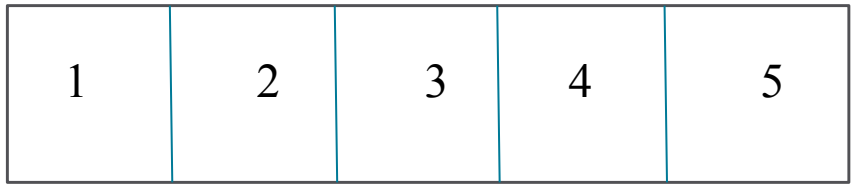
- Messages stored in fixed sized buckets
  - Deterministic when full
  - Easy to reason about
- Why not time buckets?
  - Time bugs suck
  - Non deterministic
  - Can miss data due to time overlaps
- Messages given monotonic ID
  - CAS “id” table
- Bucket # = monotonicId / bucketSize

Pointers to Buckets/Messages

- Reader pointer
  - Tracks which **bucket** a consumer is on
- Repair pointer
  - Tracks first non-finalized **bucket**
- Invisibility pointer
  - Tracks first unacked **message**

Pointers to Buckets

All 3 pointers point to monotonic id value, potentially in different buckets



InvisPointer

ReaderPointer

RepairPointer

Schema

```
CREATE TABLE queue (
  account_name text,
  queuename text,
  bucket_size int,
  version int,
  ...
  PRIMARY KEY (account_name, queuename)
);
```

```
CREATE TABLE message (
  queueid text,
  bucket_num bigint,
  monoton bigint,
  message text,
  version int,
  acked boolean,
  next_visible_on timestamp,
  delivery_count int,
  tag text,
  created_date timestamp,
  updated_date timestamp,
  PRIMARY KEY ((queueid, bucket_num), monoton)
);
```

```
*queueid=accountName:queueName:version
```

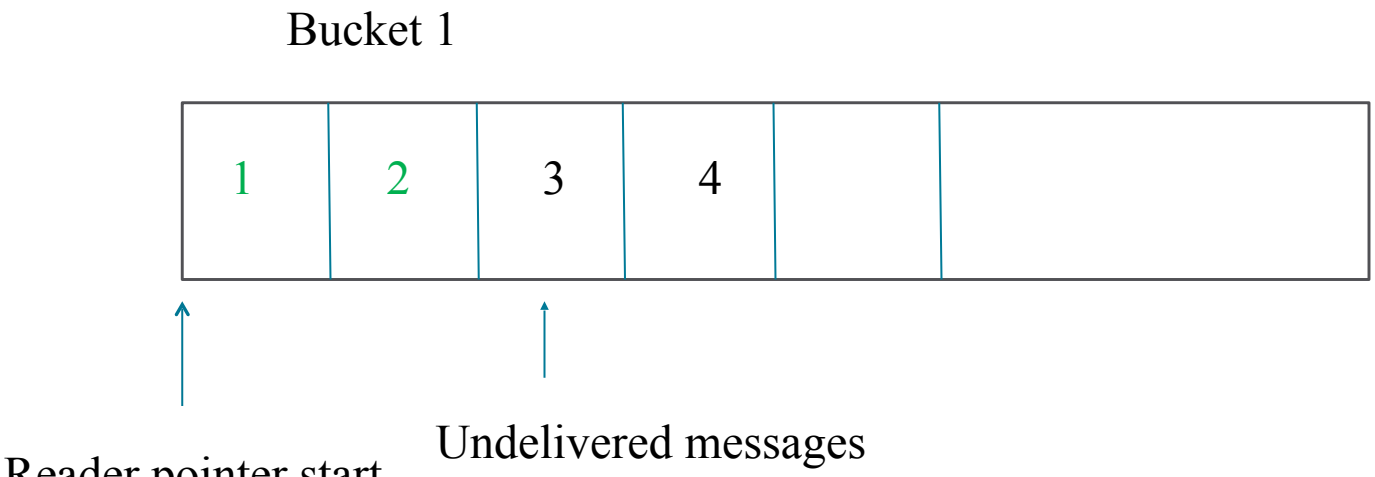
Reading messages

Pointers to Buckets/Messages

- Reader pointer
  - Tracks which **bucket** a consumer is on
- Repair pointer
  - Tracks first non-finalized **bucket**
- Invisibility pointer
  - Tracks first unacked **message**

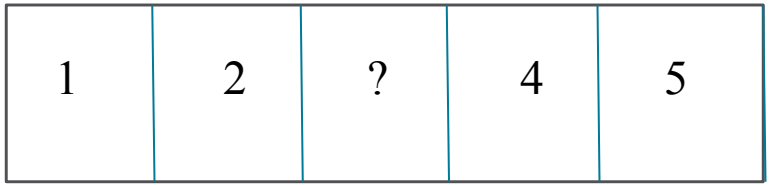
Reading from a bucket

- Read any unacked message in bucket (either FIFO or random)
- Consume message (update its internal version + set its invisibility timeout)
- Return to consumer



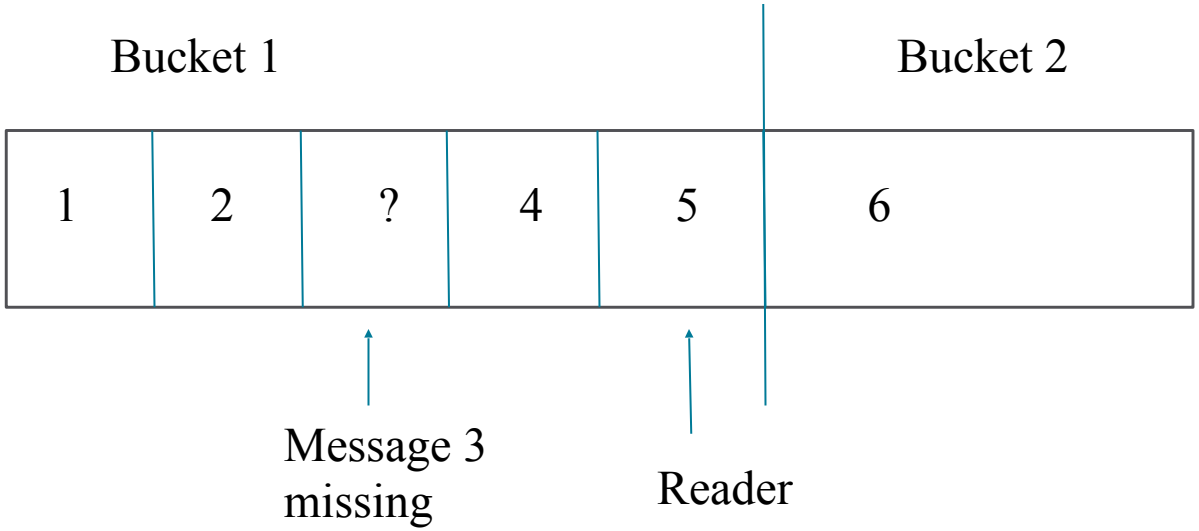
Buckets… complications

- Once a monoton is generated, it is taken
  - Even if a message fails to insert the monoton is taken
- Buckets are now partially filled!
- How to resolve?



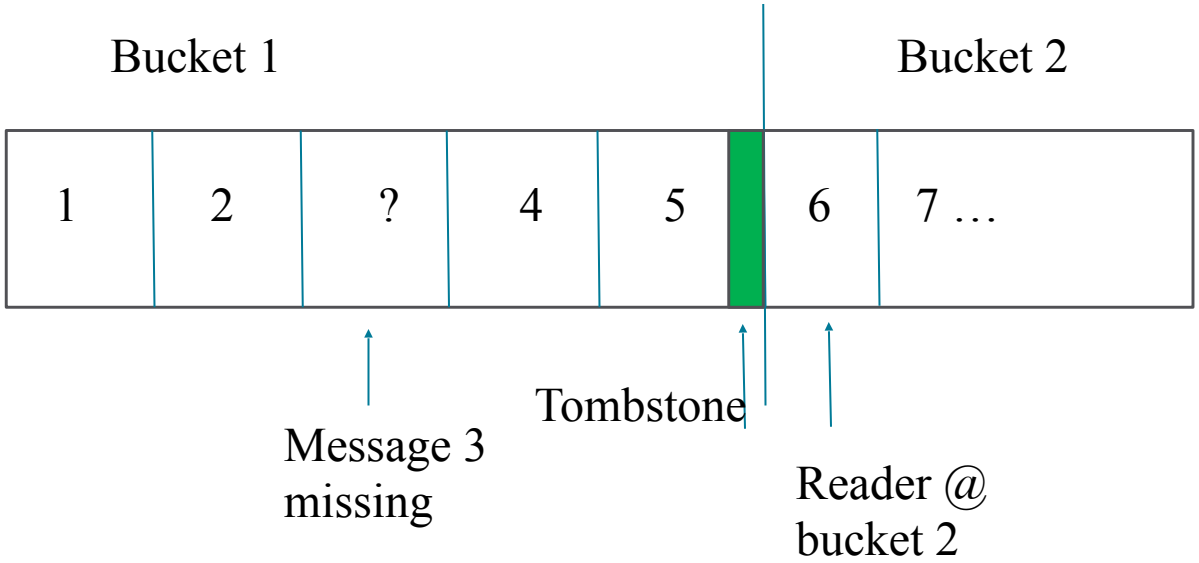
When to move off a bucket?

1. All known messages in the bucket have been delivered at least once
2. All new messages being written in future buckets

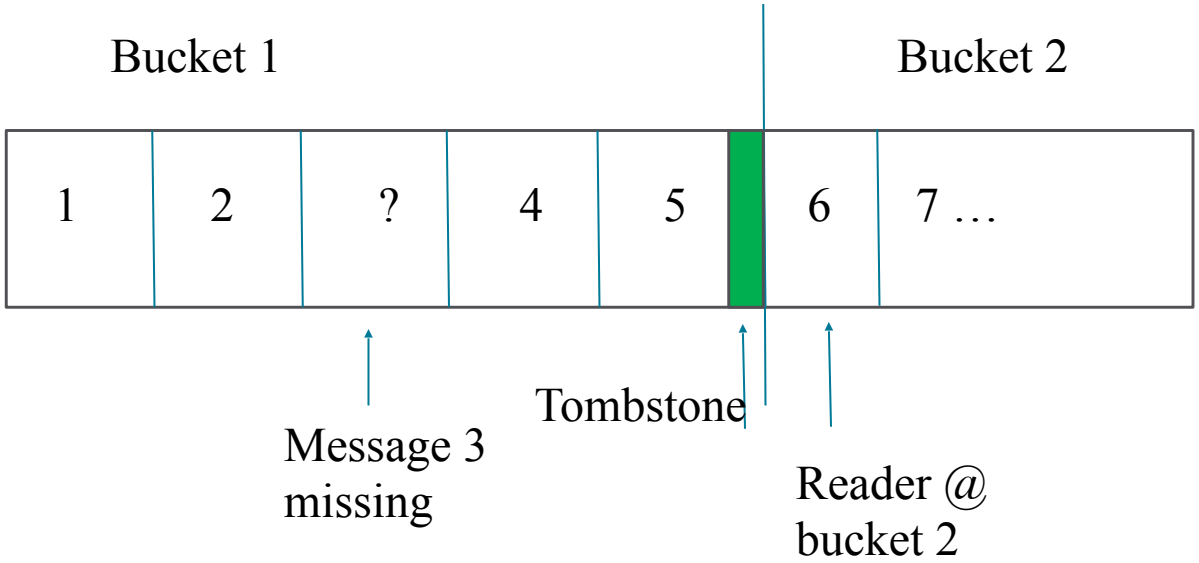


When to move off a bucket?

- Tombstoning (not cassandra tombstoning, naming is hard!)
  - Bucket is sealed, no more writes
- Reader tombstones bucket after its reached



Tombstoning enables us to detect delayed writes



Repairing delayed messages

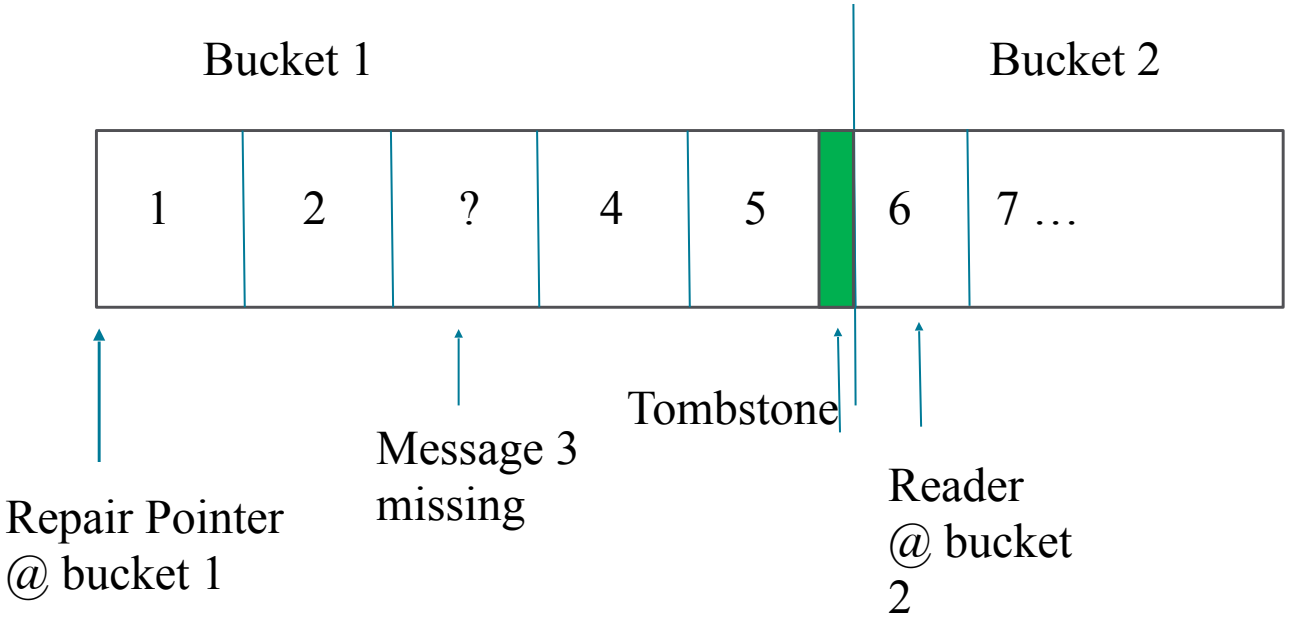
Pointers to Buckets/Messages

- ~~Reader pointer~~
  - Tracks which **bucket** a consumer is on
- Repair pointer
  - Tracks first non-finalized **bucket**
- Invisibility pointer
  - Tracks first unacked **message**

Repairing delayed writes

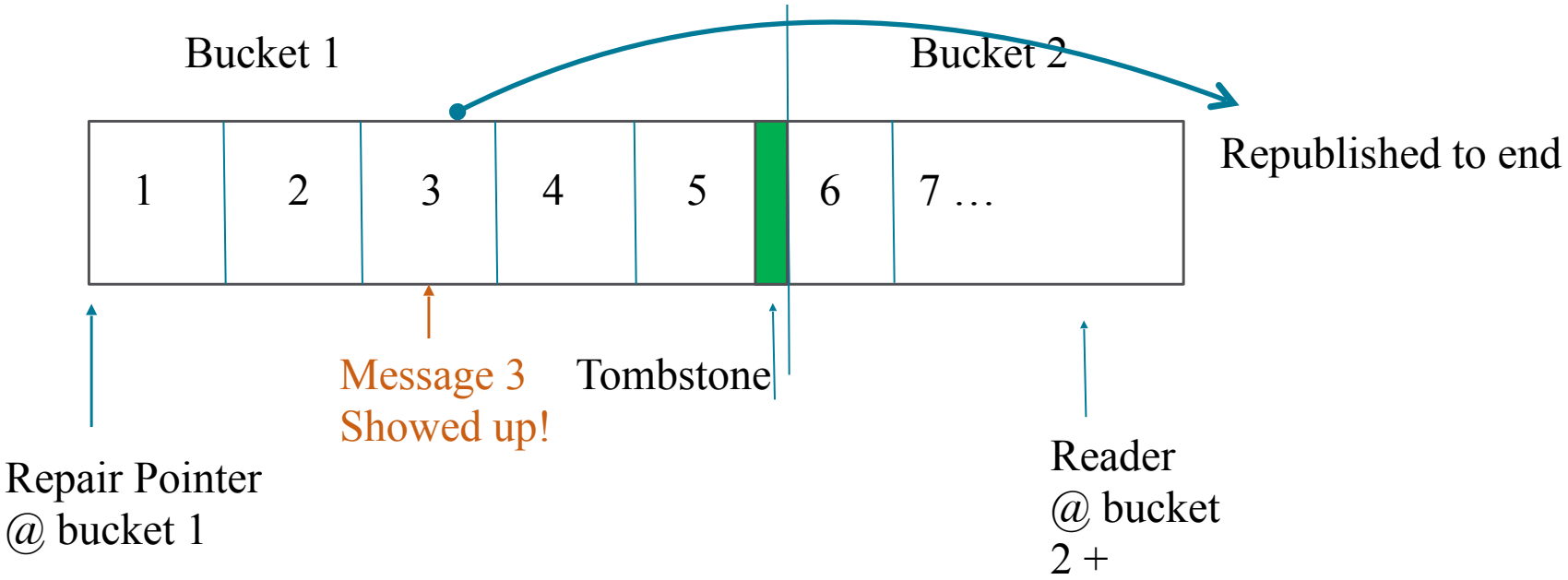
- Scenarios:
  - Message taking its time writing (still alive, but slow)
  - Message claimed monoton but is dead
- Resolution:
  - Watch for tombstone in bucket
  - Wait for repair timeout (30 seconds)
  - If message shows up, republish
  - If not, finalize bucket and move to next bucket (message is dead)

Repairing delayed writes

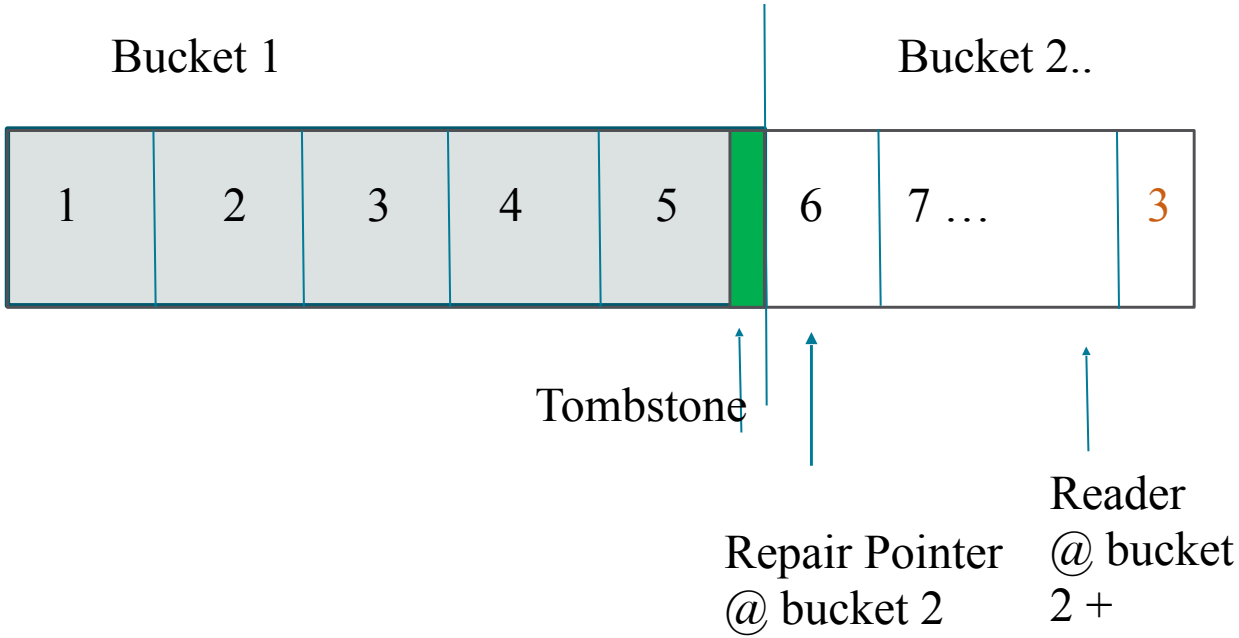


wait 30 seconds…

Repairing delayed writes



Repairing delayed writes



Invisibility

and the unhappy path ☹️

What is invisibility?

A mechanism for message re-delivery

(in a stateless system)

Pointers to Buckets/Messages

- ~~Reader pointer~~
  - Tracks which **bucket** a consumer is on
- ~~Repair pointer~~
  - Tracks first non-finalized **bucket**
- Invisibility pointer
  - Tracks first unacked **message**

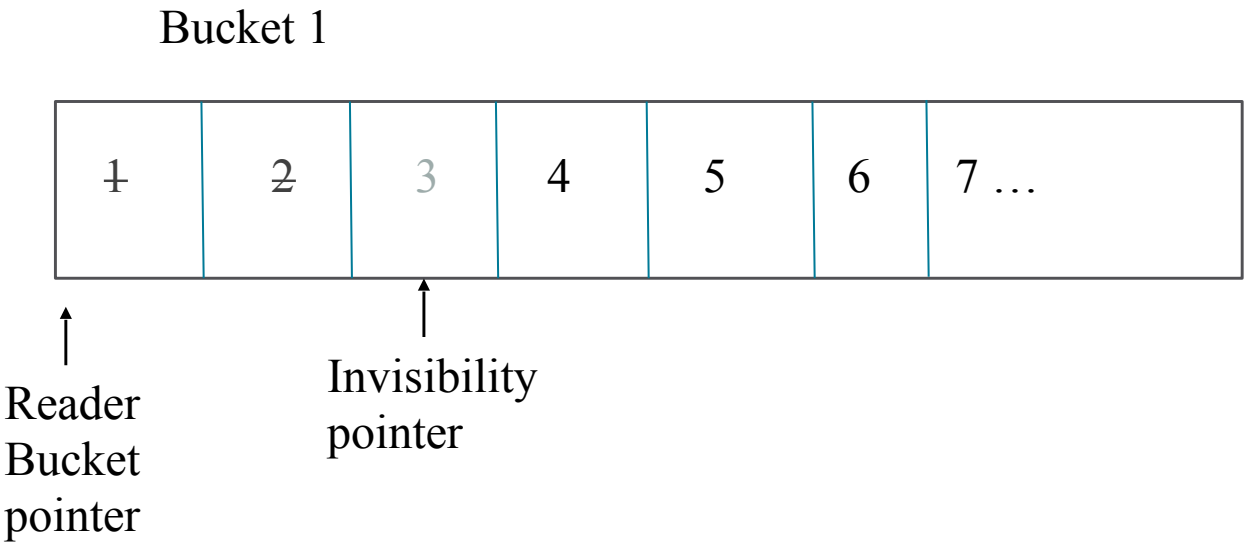
The happy path

- Client consumes message
  - Message is marked as “invisible” with a “re-visibility” timestamp
  - Client gets pop receipt encapsulating metadata (including version)
- Client acks within timeframe
  - Message marked as consumed if version is the same

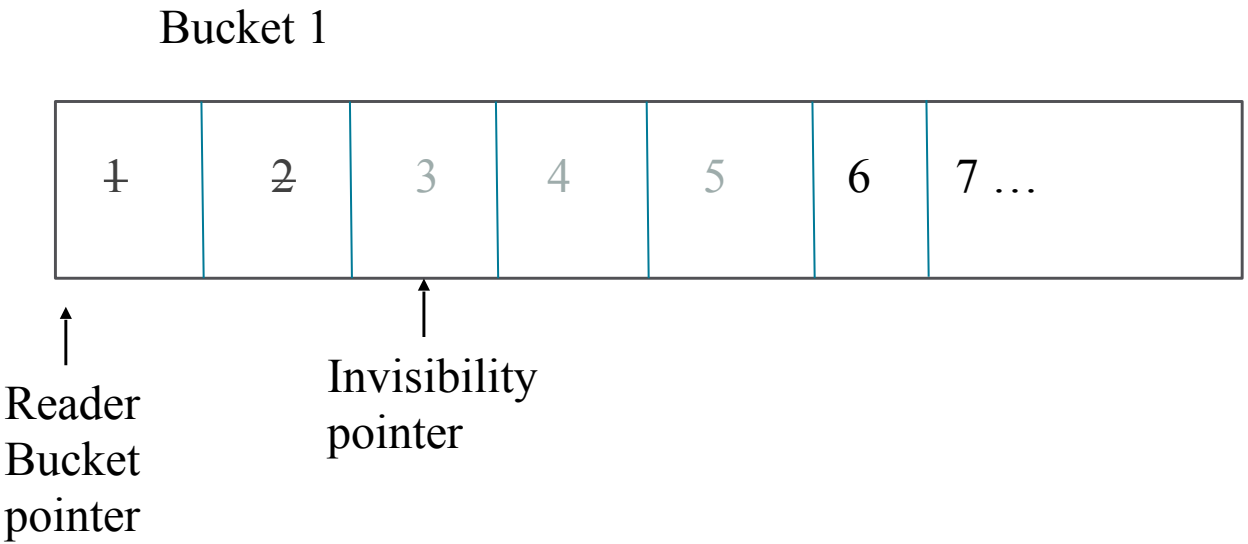
The unhappy path :(

- Client **doesn't** ack within timeframe
- Message needs to be redelivered
- Subsequent reads checks the invis pointer for visibility
  - If max delivers exceeded, push to optional DLQ
  - Else redeliver!

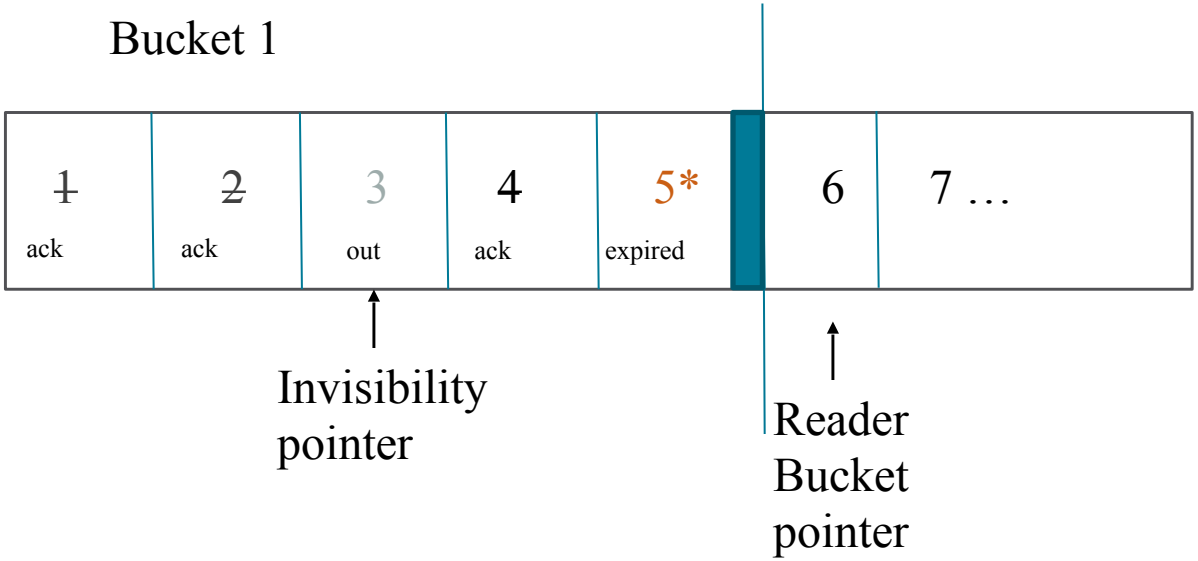
The unhappy path :(



The unhappy path :(



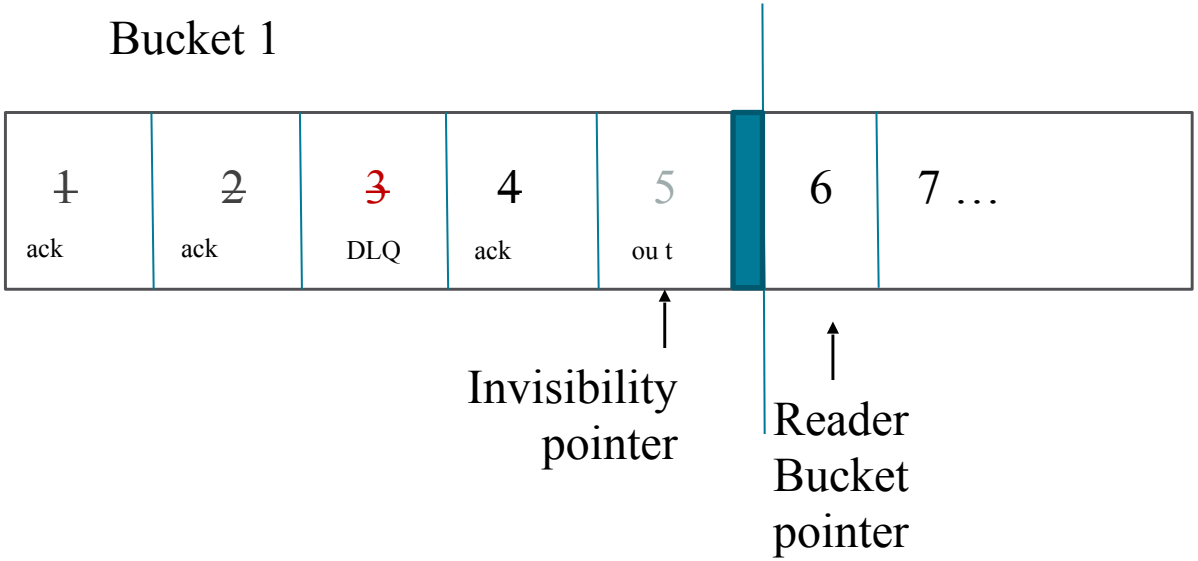
The unhappy path :(



Long term invisibility is bad

- InvisPointer WILL NOT move past a unacked message
- Invisible messages can block other invisible messages
- Possible to starve future messages

The unhappy path :(



Conclusion

- Building a queue on c* is hard
- Limited by performance of lightweight transactions and underlying c* choices
    - compaction strategies, cluster usage, etc
- Need to make trade off design choices
- CassieQ is used in production but in not stressed under highly contentious scenarios

Questions?

or feedback/thoughts/visceral reactions

Contribute to the antipattern @ paradoxical.io

https://github.com/paradoxical-io/cassieq