High Concurrency Architecture

09-2019. TIKI.VN

Contributors

Bùi Anh Dũng
Principal Engineer

Nguyễn Hoàng Bách
Senior Principal Engineer

Phan Công Huân
Senior Software Engineer

Lê Minh Nghĩa
Senior Architect

Trần Nguyên Bản
Principal Engineer

Agenda

- Principles
- Pegasus - Highest throughput API
- Arcturus - High concurrency inventory API
- Conclusion

Principles

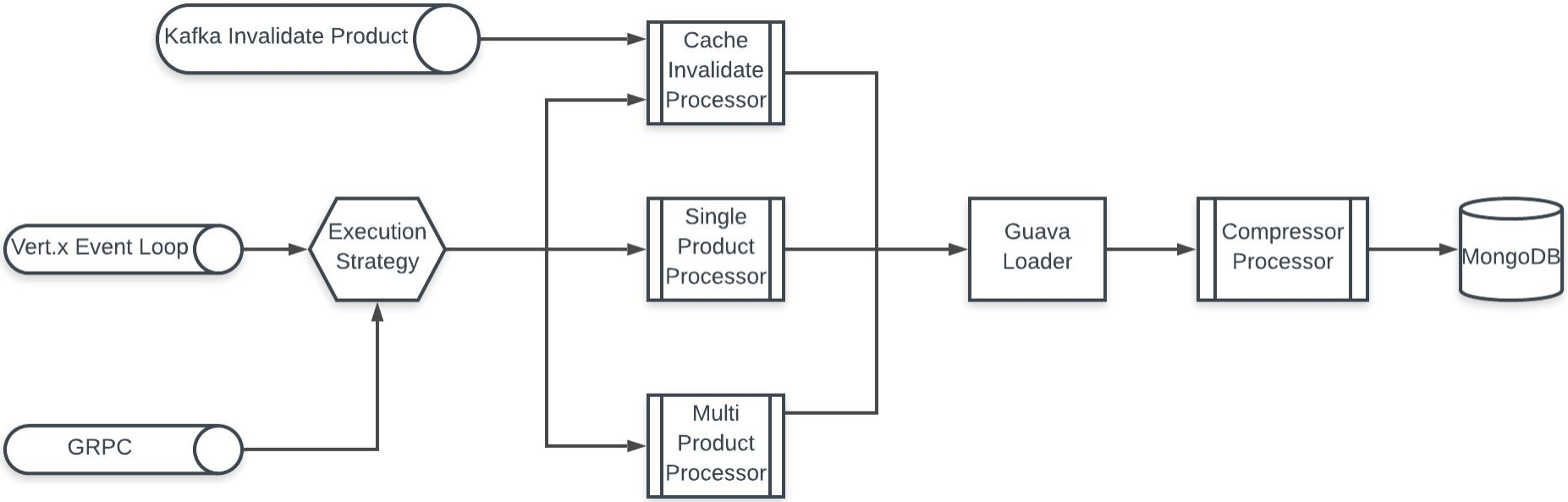
- Use local memory to handle high concurrency transaction
- Non blocking architecture
- Trade-offs consistency vs eventual consistency
- Reliable replication
- Authority: each service owns its problems.

Pegasus - Highest Throughput API at TIKI

Problems:

- Handle the most of traffic to fetch product information
- Has to handle at least 10k request/s, tp95 < 5ms

Pegasus - Architecture



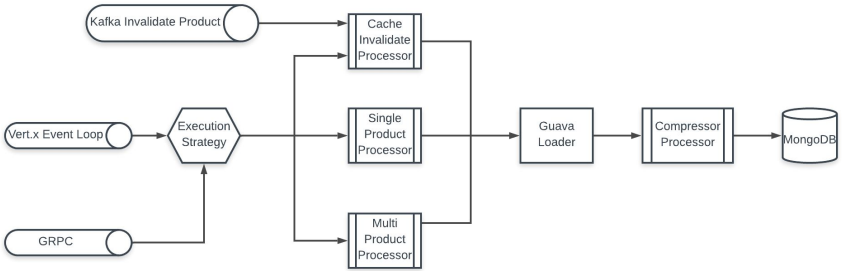
Pegasus - Architecture

Practises:

- Cache product data in local memory
- Subscribe product change event to invalid cache
- Non-Blocking http web server
- Compress to reduce payload size

Technology:

- Java
- Guava - In Memory Cache
- Gridgo - Async IO & Event driven framework



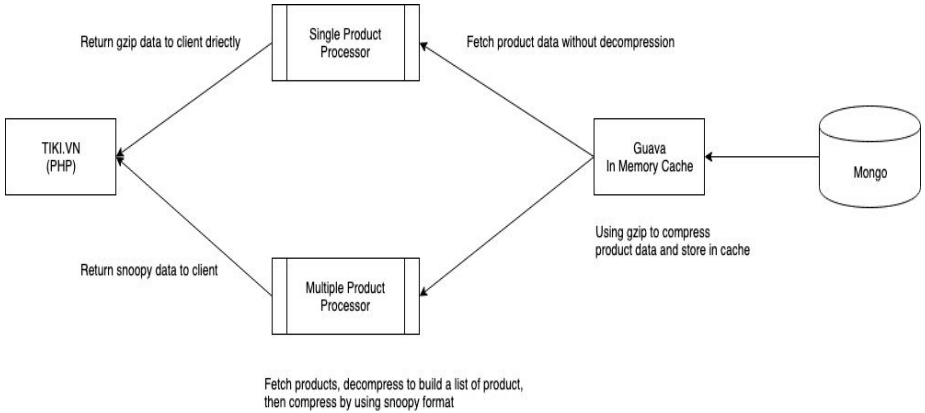
Pegasus - Compression

Two cases:

- Get single product by id
- Get multiple product by a list id

Solutions:

- Using gzip to compress product data to store in cache. Reduce from 200kb text to 3Kb
- Handle single product request: return compressed gzip bytes to client directly
- Handle multiple product request: store plain product data and merge them to build a list of products, then use Snappy format to compress data and return to client
- Gzip format compress better than Snappy and is supported natively by most http client, but Snappy compress faster and use less CPU
- Output cache can be enabled in hot-deal situation

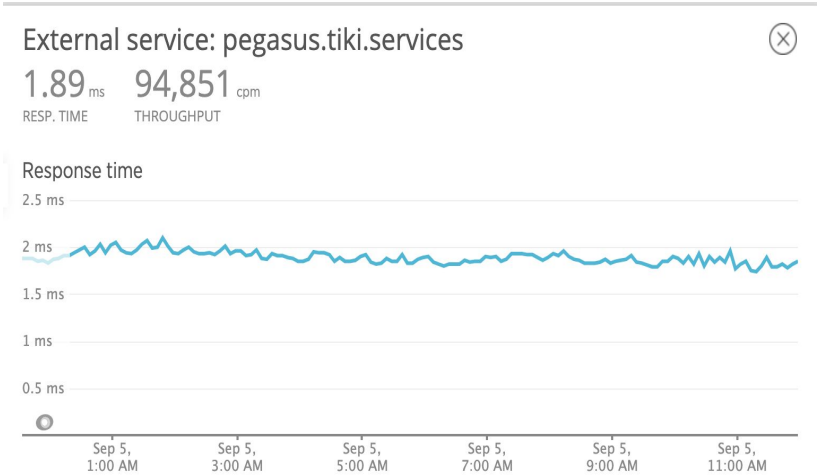


Pegasus - Technology

- Java
- Gridgo
- Guava
- Kafka

Pegasus - Benchmark

```
./wrk -c400 -t12 --latency http://pegasus-query.tiki.services/v1/products/243815
Running 10s test @ http://pegasus-query.tiki.services/v1/products/243815
  12 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     4.81ms    6.05ms 212.42ms   97.13%
    Req/Sec     8.06k   735.74    10.33k    70.85%
  Latency Distribution
     50%    3.83ms
     75%    4.97ms
     90%    6.60ms
     99%   37.53ms
  971596 requests in 10.10s, 2.59GB read
Requests/sec:  96189.34
Transfer/sec:    262.54MB
```



Benchmark use WRK tools, 4VM, L4 LB, GCP:

- 90%<6.6ms
- 96k request/s

Production

- 200k request/minutes (all platforms)
- <2ms

Cache hit: 85%. Increased to 95% with Consistent Hashing With Bounded Load.

Average payload: Single (3KB), Multi (60KB)

Pegasus - Benchmark/Replication

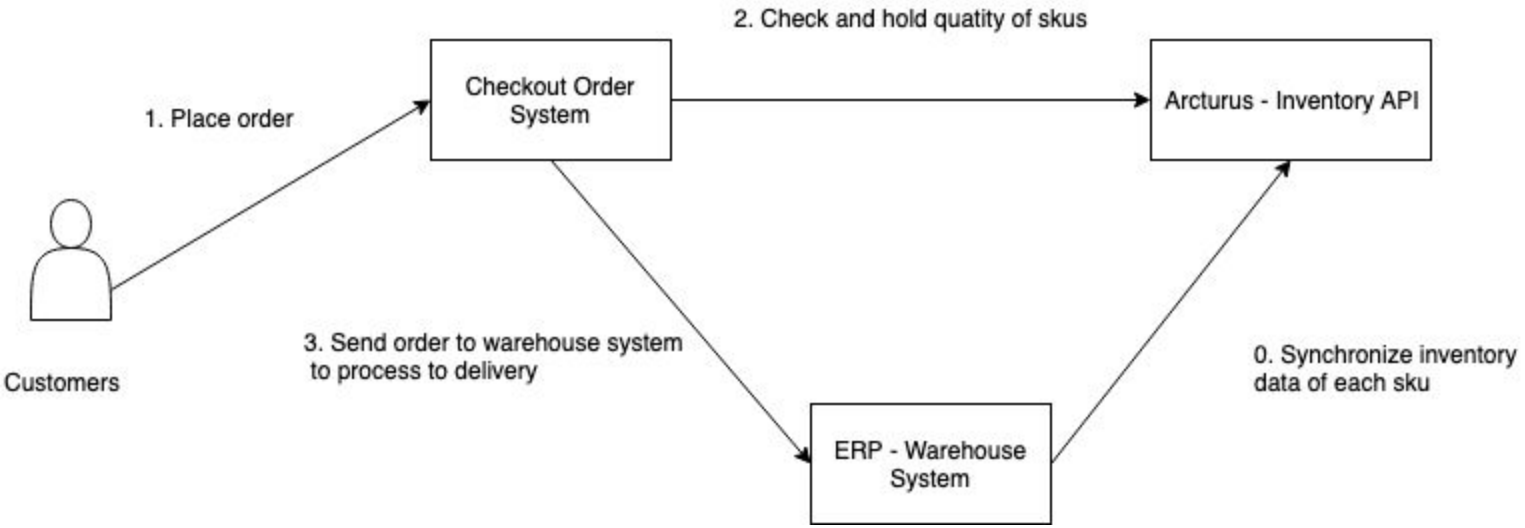
Min Lag Time From Binlog To MongoDB	Max Lag Time From Binlog To MongoDB	Today Message Indexed To DB
36.0 ms	21.4 s	21.5 Mil message

Arcturus - High concurrency Inventory API

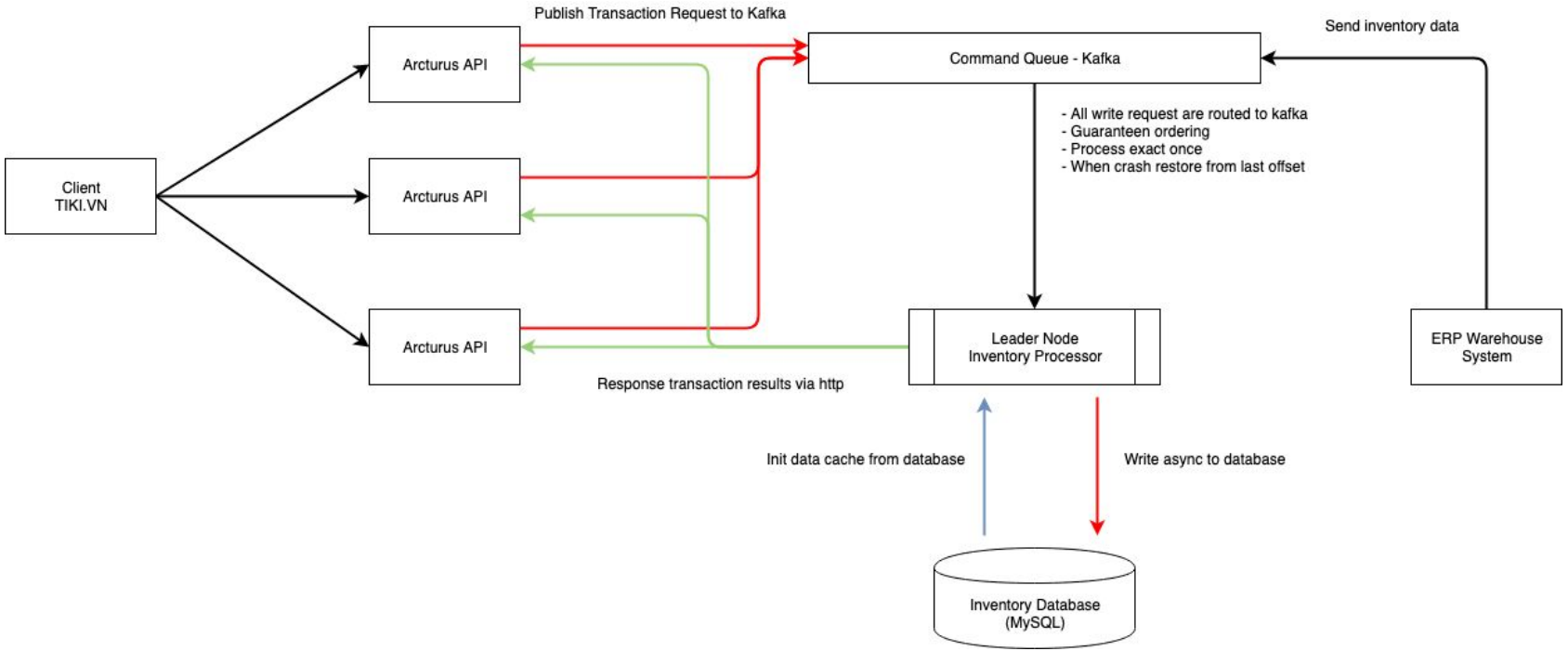
Problems:

- Handle inventory transaction when customers place orders
- Handle high concurrency transaction for extreme hot deals with very cheap and low quantity. Exp: Only ten 1đ iPhone XS, only in ten minutes from 9AM to 9h10 AM.
- Guarantee eventual consistency of inventory data between many systems

Arcturus - Context Diagram



Arcturus - Architecture



Arcturus - Problems and Solutions

Problems:

- Many customers may place order at the same time, especially for hot skus.
- System has to be deterministic and can be recovered after crashing

Solutions:

- All write requests are published to Kafka with only one partition to guarantee the ordering of message and recover if system crashes.
- Non Blocking In Memory Cache for both read and write
- All changes are flushed to database asynchronously
- Recovery based on the offset of write command

Arcturus - In Memory Cache Data Structure

Problem:

- There will be a race condition when many customer place the same skus at the same time
- Building an in memory cache structure that can buffer data changes and flush to database asynchronously

Approach:

- Each record have a *key* (string, bytes…) and a *value* (8 bytes long number).
- A *transaction* is a set of record updates.
- *Transactions* must be ordered (e.g. *key_1* must +3 before can be -1).
- It can be millions of keys and process upto 10k TPS.
- Using ring buffer data structure to guarantee the ordering of changes

	key_0	key_1	key_2	key_3	...	key_n
tx_0	-1		-1	-2		
tx_1		+3				
...						
tx_n		-1	-1			-4

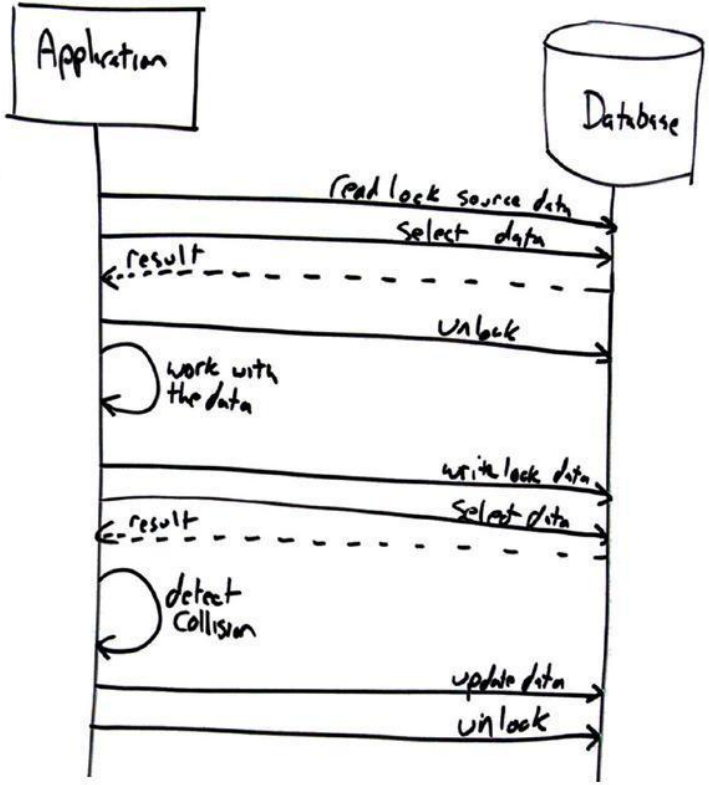
* ***The hardest thing is to keep everything ordered when make it fast enough.***

Arcturus - Old solution

- Multi threading application
- DB transaction, row locking

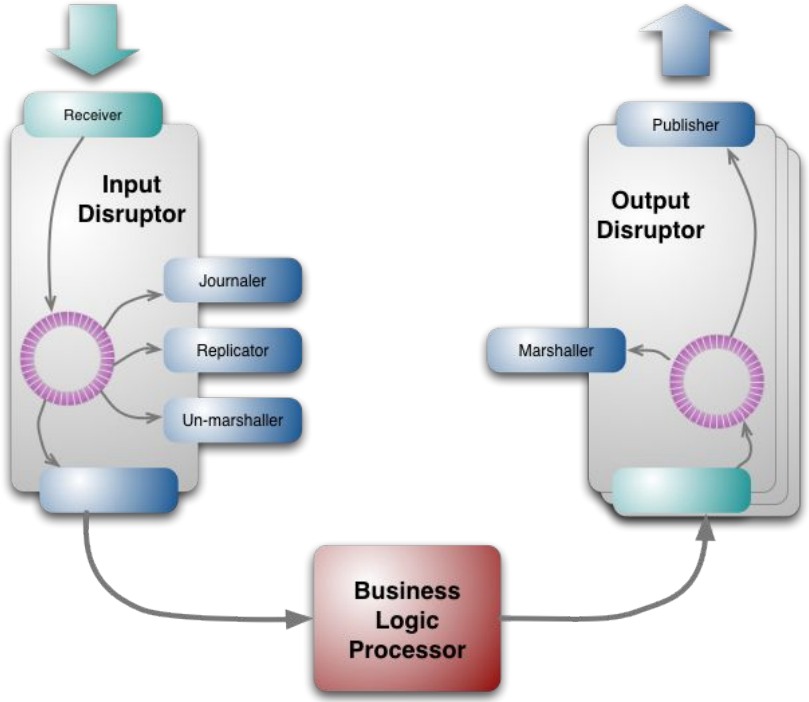
Pros
- Strong consistency
- Simple implementation

Cons
- Slow
- Deadlock/timeout implicit risk



Arcturus - LMax Architecture

- Non Blocking architecture using single thread business logic processor
- Use Ring Buffer data structure to communicate between processors
- Handle million transaction per seconds

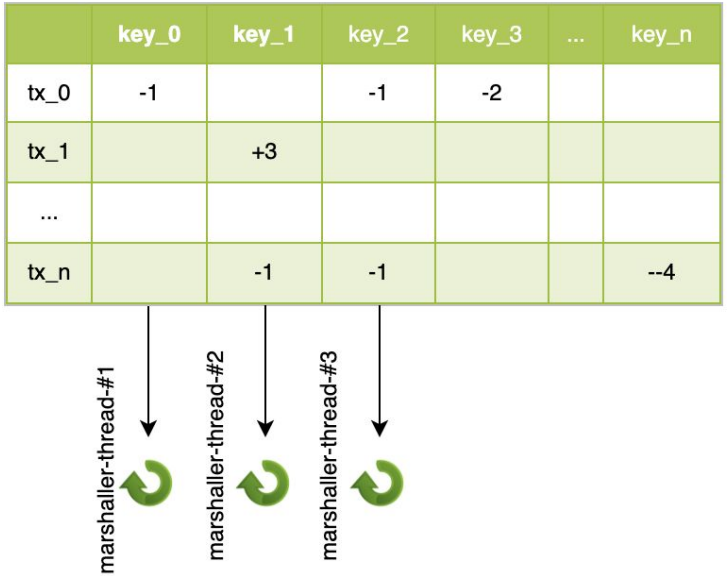


Arcturus - Batching marshaller

vBatching: use multi thread marshaller

- Pros
  - Fast (400k-600k ops/s*)
  - Avoid multi key locking
  - Write only most updated value
- Cons
  - Inconsistent transaction offset

→ use when you just want to make your code as fast as possible.



* *Macbook pro 2016 15" 2.7GHz Core i7, 16G ram. Without I/O, single update per transaction*

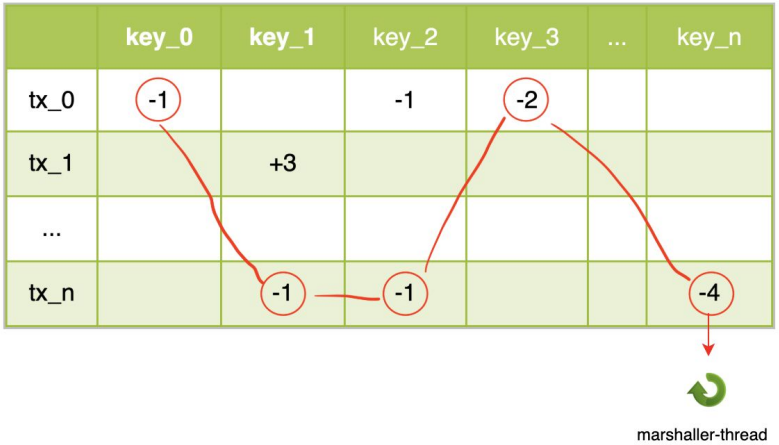
*** *Test without I/O: [Single ops vbatch] Total time running for 1000000 transactions: 1085 ms; at rate: 921,658.99 ops/s*

Arcturus - Batching marshaller

hBatching: use single marshaller thread

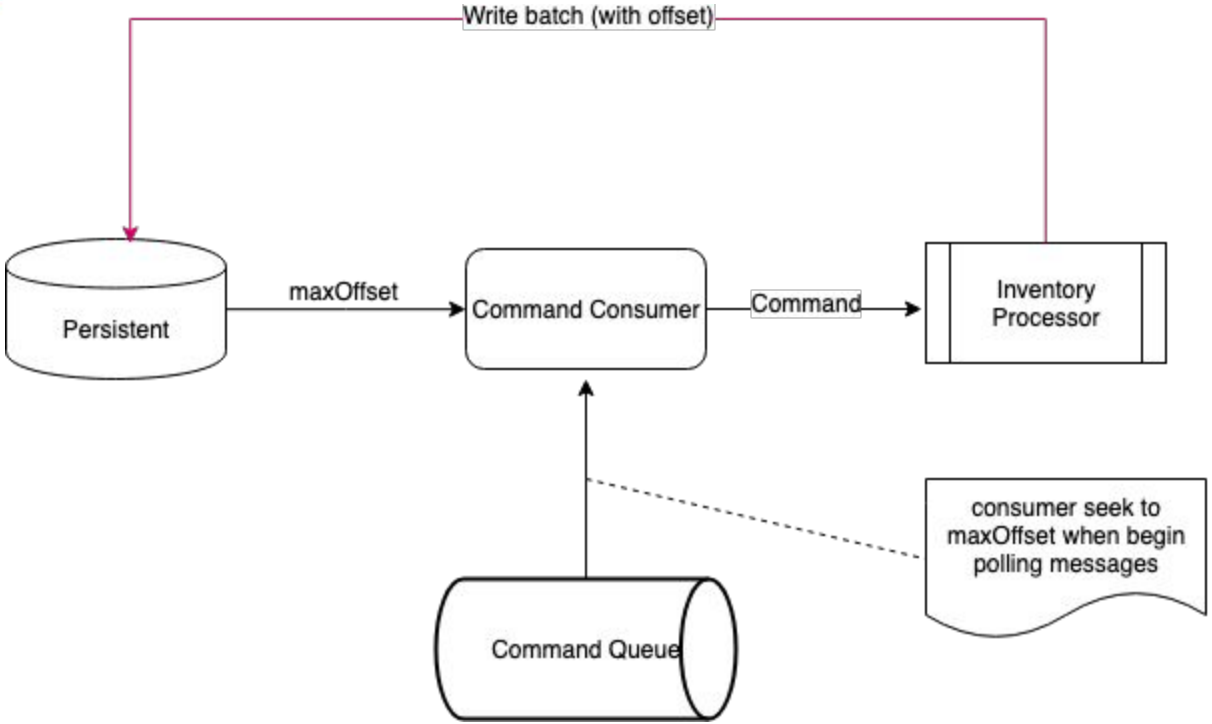
- Pros
  - Ensure transaction offset
  - Ordered db updating
- Cons
  - A little bit slower than vBatching.

→ use when you need to re-create application state after fail.



*** Test without I/O: [Single ops hbatch] Total time running for 1000000 transactions: 1056 ms; at rate: 946,969.70 ops/s

Arcturus - Recovery/Replay



Arcturus - Technology

- Java
- Kafka
- ZeroMQ
- Gridgo
- MySQL
- LMax/Ring Buffer

Conclusion

- Has to trade-off between consistency and eventual consistency
- In Memory is a great way to improve the performance
- Reliable replication is the key to split and scale system
- Non Blocking architecture is a great way to utilize hardware resources efficiently.