Grafana Lokiで構築する
大規模ログモニタリング基盤

CNDT2021

LINE株式会社 Hiroki Sakamoto / @taisho6339

自己紹介

- Title:
  Senior Software Engineer@LINE Corp
- Role:
  Private Cloud開発組織のSRE
- Mission
  Private Cloudを横断した信頼改善
- Interest:
  Kubernetes, 分散システム, 筋トレ, OSS活動
- Twitter: @taisho6339

Lokiとは何か？

like Prometheus but for logs

- ログの保存と検索機能
- ログベースのアラーティング機能
- ログベースのメトリクス作成機能
- マルチテナントのDefault Support

Lokiとは何か？

Explore VRE-Loki-stage4dev

Split | Last 1 hour | Clear all | Run query | Live

Log browser › {service="rabbitmq"} 0.1s

Query type: Range | Instant — Line limit: auto

Help ›

+ Add query | Query history | Inspector

Logs

This datasource does not support full-range histograms. The graph is based on the logs seen in the response.

20, 0 — 11:20 11:25 11:30 11:35 11:40 11:45 11:50 11:55

info

Time | Unique labels | Wrap lines | Dedup: None | Exact | Numbers | Signature — Flip results order

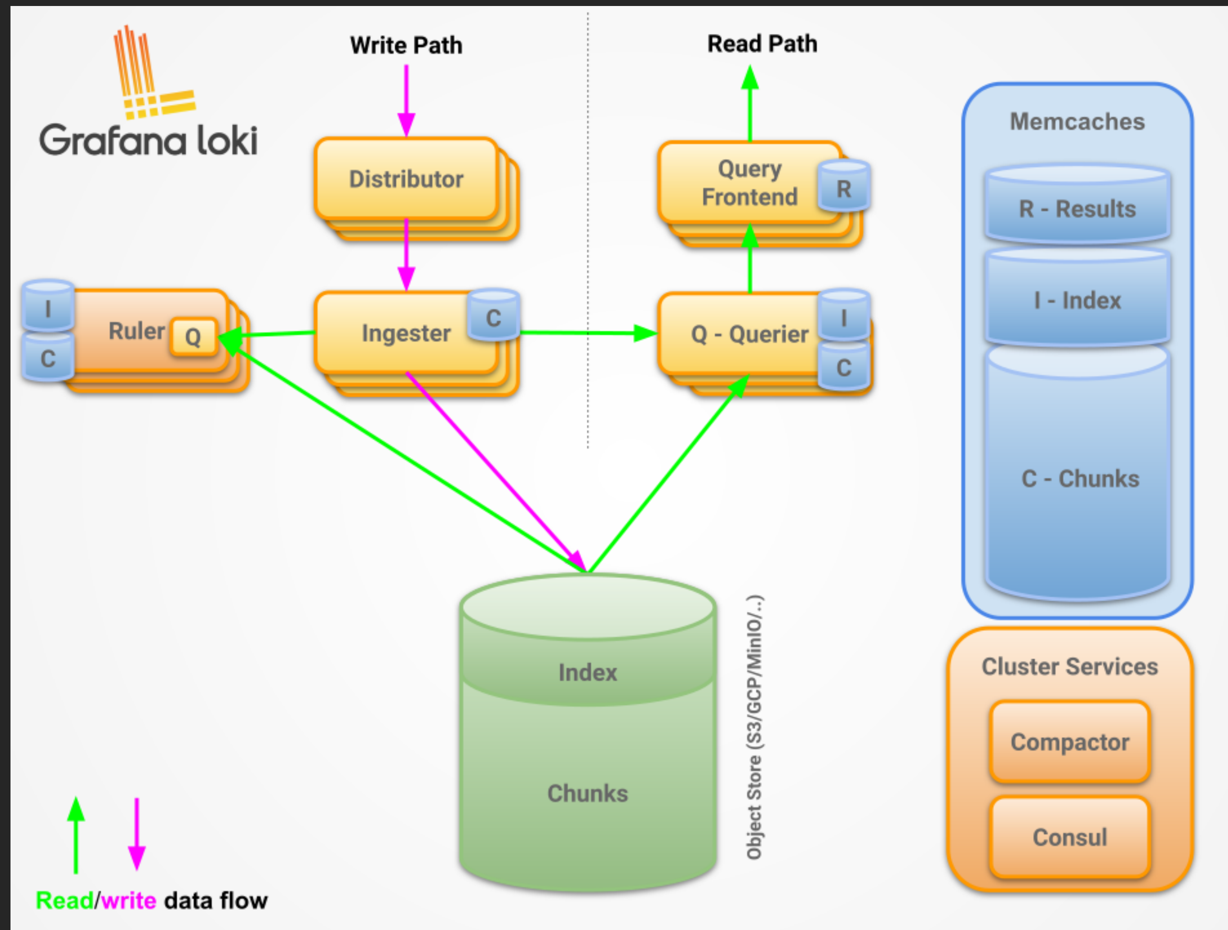
Common labels: rabbitmq stage4dev Line limit: 1000 (40 returned) Total bytes processed: 22.1 kB

› 2021-10-26 11:26:49
› 2021-10-26 11:26:45
› 2021-10-26 11:16:33
› 2021-10-26 11:16:28
› 2021-10-26 11:16:26
› 2021-10-26 11:16:20
› 2021-10-26 11:16:17

Start of range

11:59:46 — 11:15:45

Lokiとは何か？



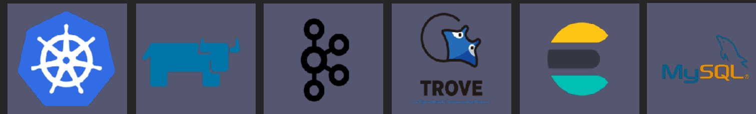
安く大容量のログを保存可能

Private Cloud "Verda" in LINE

is based on OpenStack. since 2016~

FaaS

PaaS



IaaS

Private Cloud “Verda” in LINE

74000+

Virtual Machines

30000+

Physical Machines

4000+

Hypervisors

20 TB / day application logs

Loki is suitable for us!

Lokiの難しさ

Lokiはマイクロサービス

- 各コンポーネント、各キャッシュの仕組みと役割が不明瞭
- ログデータはどこでどんな形式でどのくらい保持されるか不明瞭
- ストレージ障害時はどういう挙動になるのか不明瞭
- 本番で運用するなら何を考慮しないといけないのかが不明瞭

本セッションのゴール

国内で最も詳細に解説することを目指します

- 体系的にLokiのコンポーネントの役割と仕組みを知る
- トラブル時にも原因の特定が迅速にできるようになる
- 自力でキャパシティ管理、パフォーマンスチューニングできるようになる

本セッション想定構成

Loki Version: v2.3.0

Cache: Memcached

Chunk Storage: AWS S3

Index Storage: AWS S3 + BoltDB Shipper

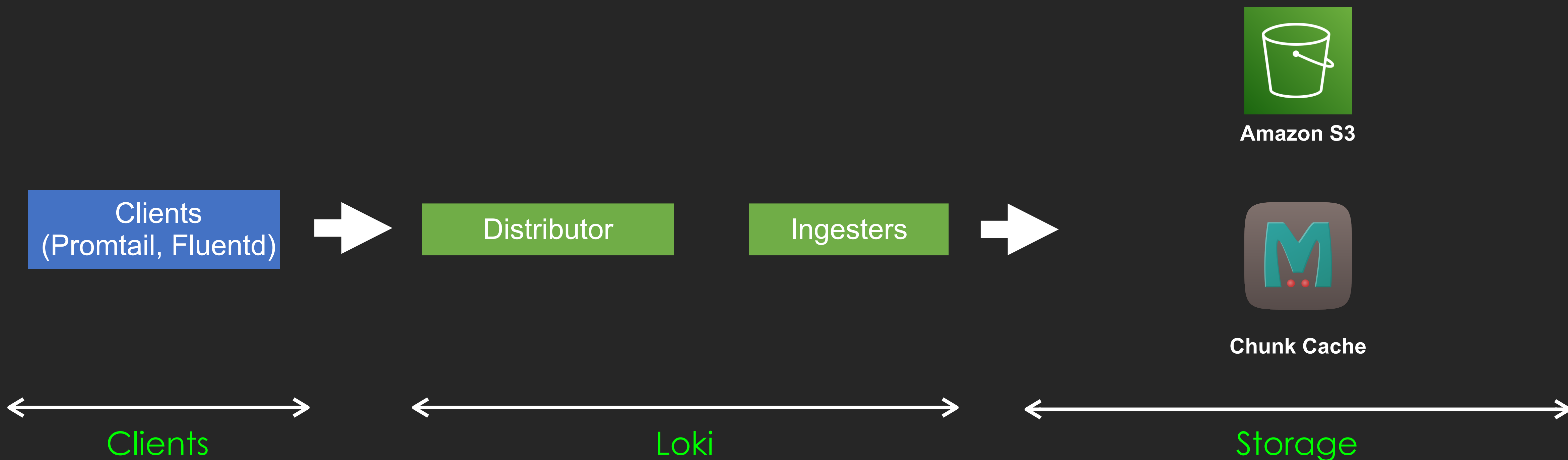
本セッションのRoadmap

1) ログの書き込みプロセスを知る
2) ログの読み込みプロセスを知る
3) 障害時の挙動を知る

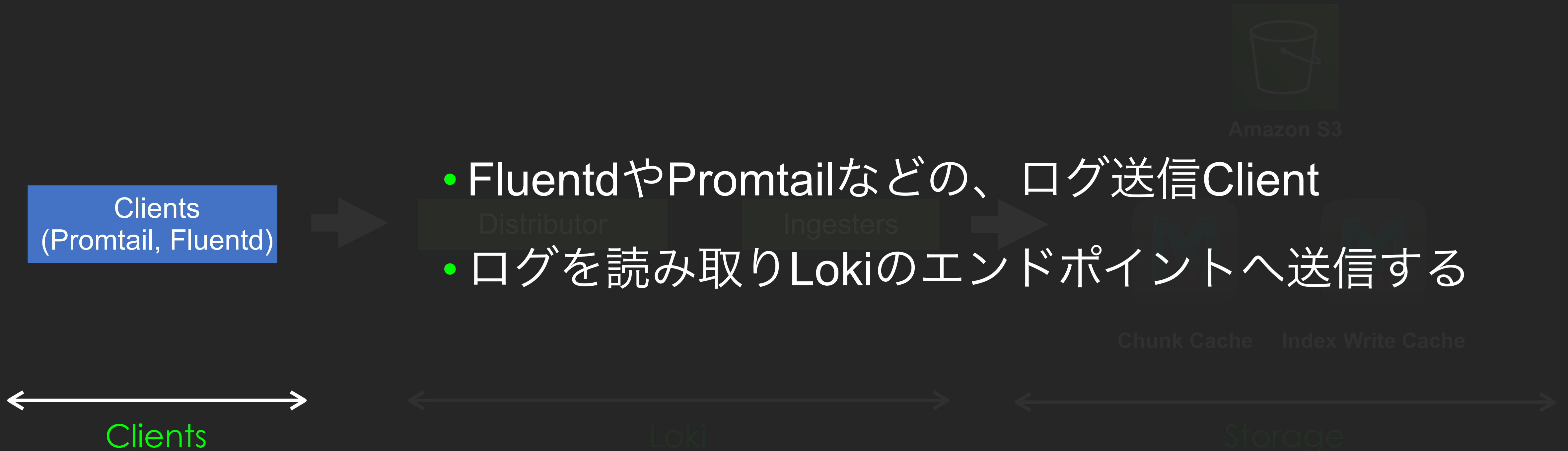
1) ログの書き込みプロセス

Overview

書き込みプロセスの登場人物

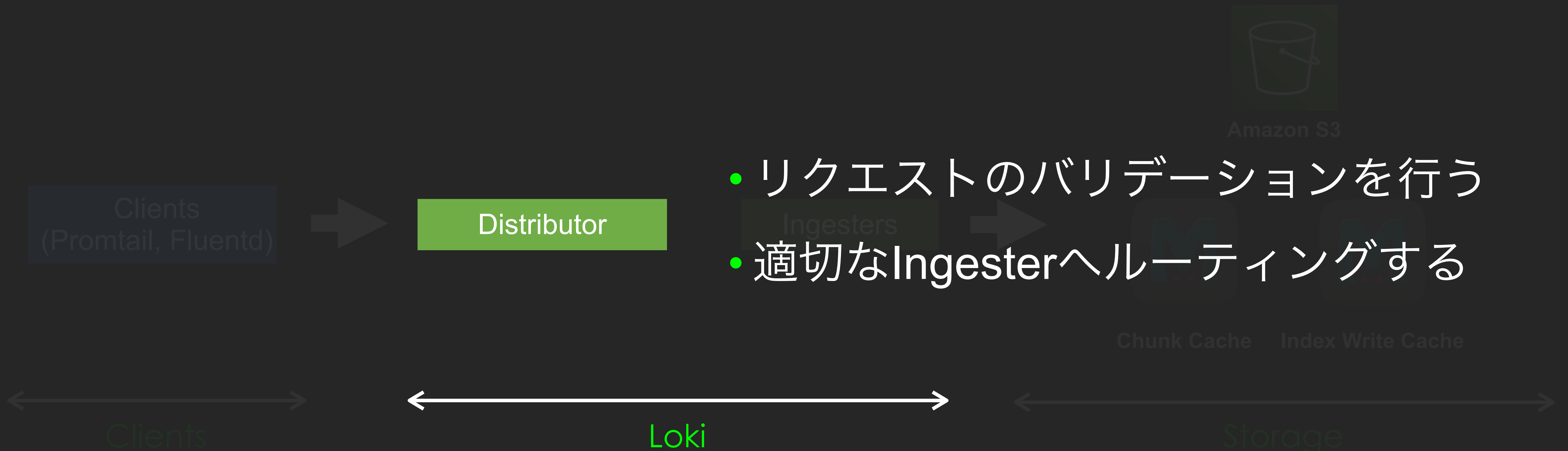


書き込みプロセス Overview



- FluentdやPromtailなどの、ログ送信Client
- ログを読み取りLokiのエンドポイントへ送信する

書き込みプロセス Overview

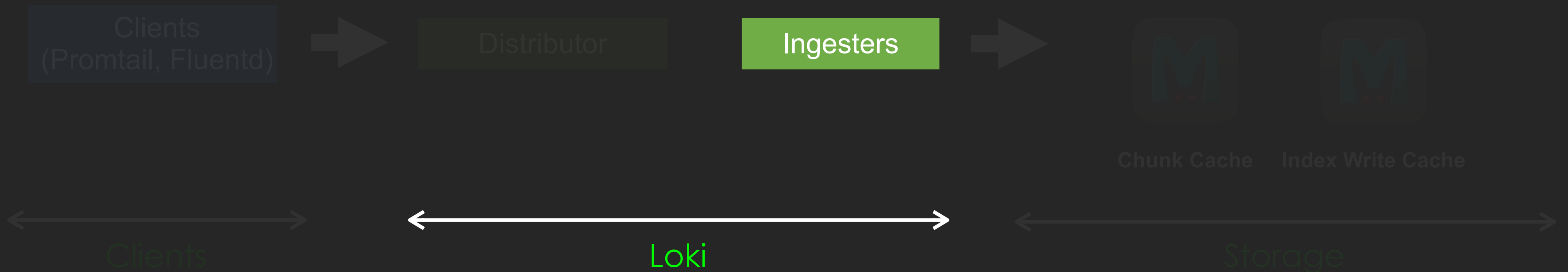


- リクエストのバリデーションを行う
- 適切なIngesterへルーティングする

書き込みプロセス Overview

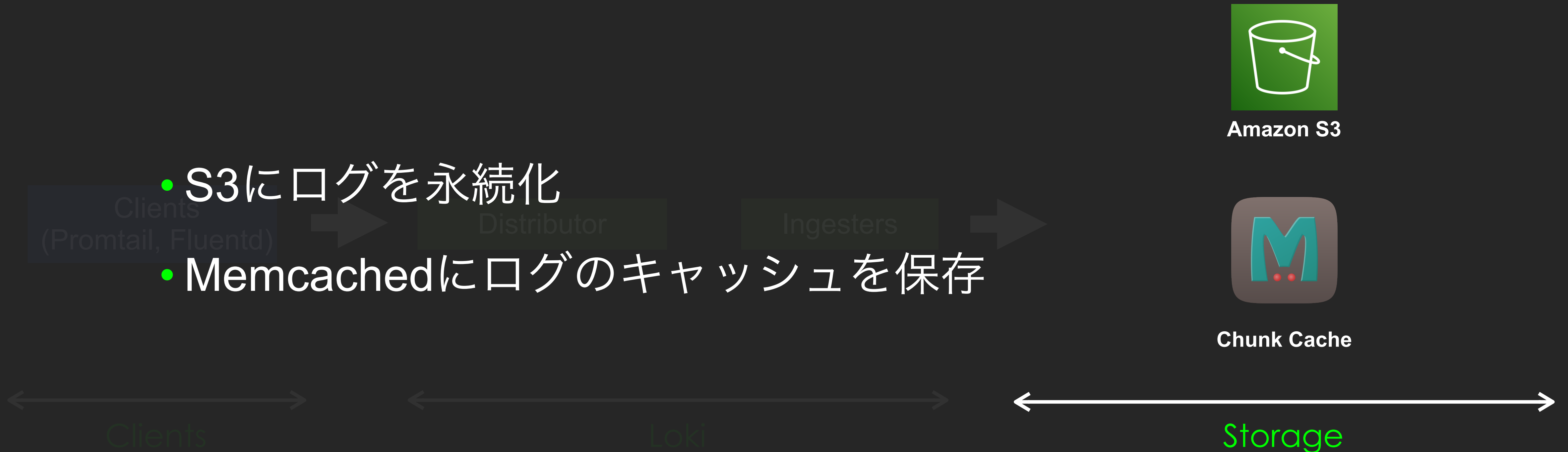
ログを実際にストレージに保存する

一定時間バッファリングしてからストレージに保存



書き込みプロセス Overview

- S3にログを永続化
- Memcachedにログのキャッシュを保存



ClientからDistributorへの送信

Client -> Distributor

TenantIDをRequest Headerに記載

```
HTTP Headers
X-Scope-OrgID : <Tenant ID>
```

Client -> Distributor

Lokiへ送るログデータ構造

{service="keystone", hostname="host1"} 00:00:02 keystone log body
{service="keystone", hostname="host1"} 00:00:03 keystone log body
{service="keystone", hostname="host1"} 00:00:04 keystone log body

Client -> Distributor

Stream TS Log Body

Client -> Distributor

{service="keystone", hostname="host1"} 00:00:02 keystone log body

Stream　TS　Log Body

ログはいくつかのラベルを持つ。
TenantIDとラベルの組み合わせの
一つ一つを、「Stream」と呼ぶ

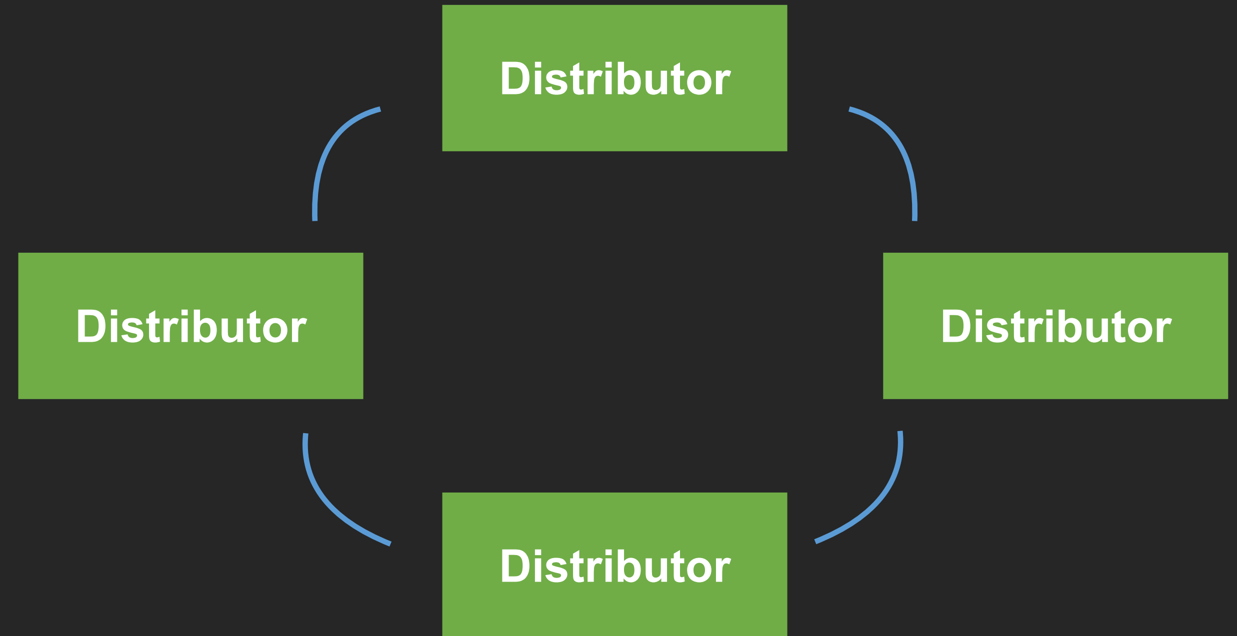
Distributorでのバリデーション

- ラベルの形式は正しいか？
- Rate limitを超えないか？

Distributorでのバリデーション

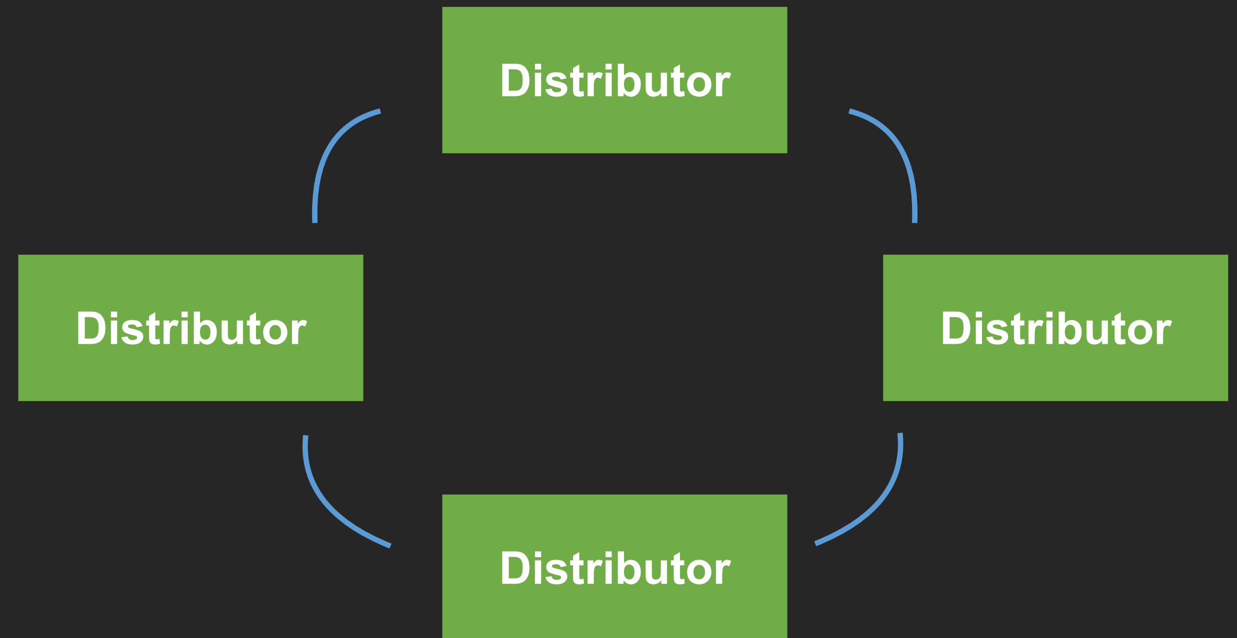
Distributor同士でクラスタリング

- Statusを互いに相互に監視
- ingestion_rate_strategyがglobalならクラスタ全体でingestion rateを制御する



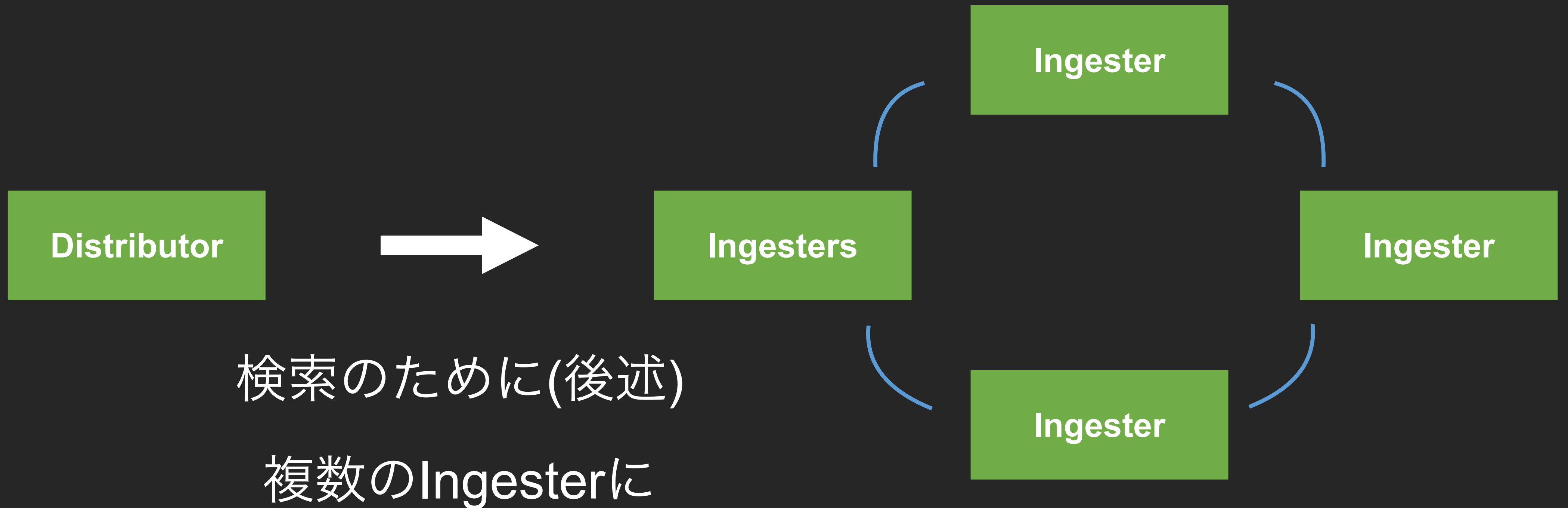
Distributorでのバリデーション

Distributor全体でバリデーション
するためのクラスタリング



DistributorからIngesterへの送信

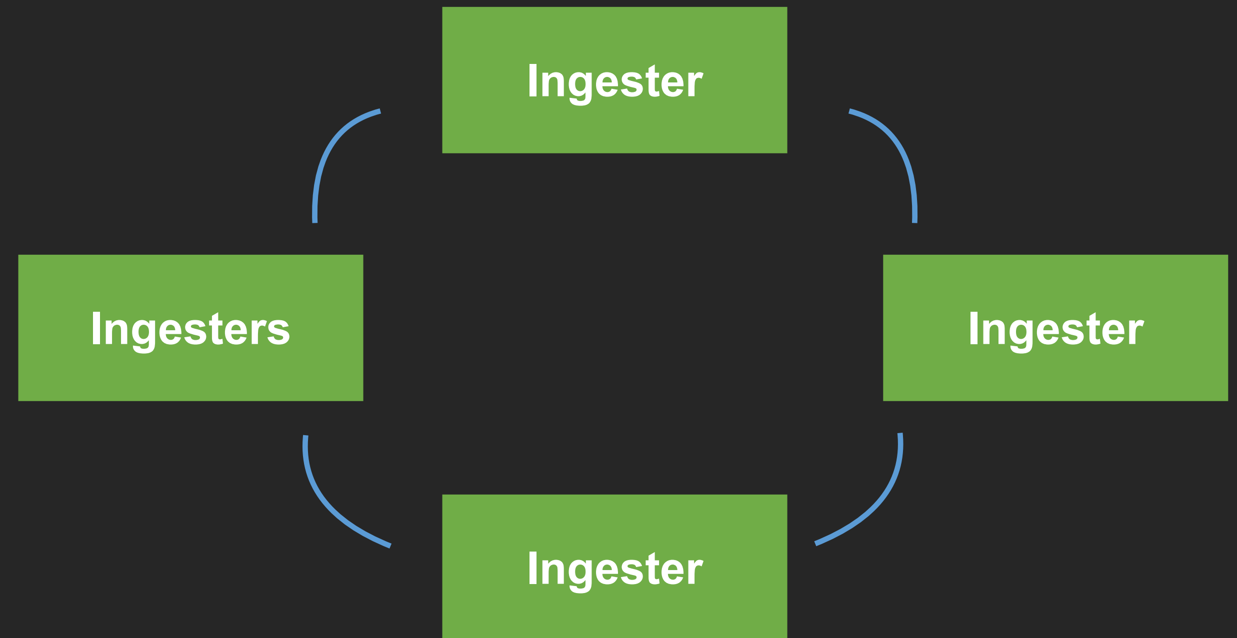
Distributor -> Ingesters



Distributor -> Ingesters

Ingester同士でクラスタリング

- Statusを互いに相互に監視
- Consistent HashのRingになっている



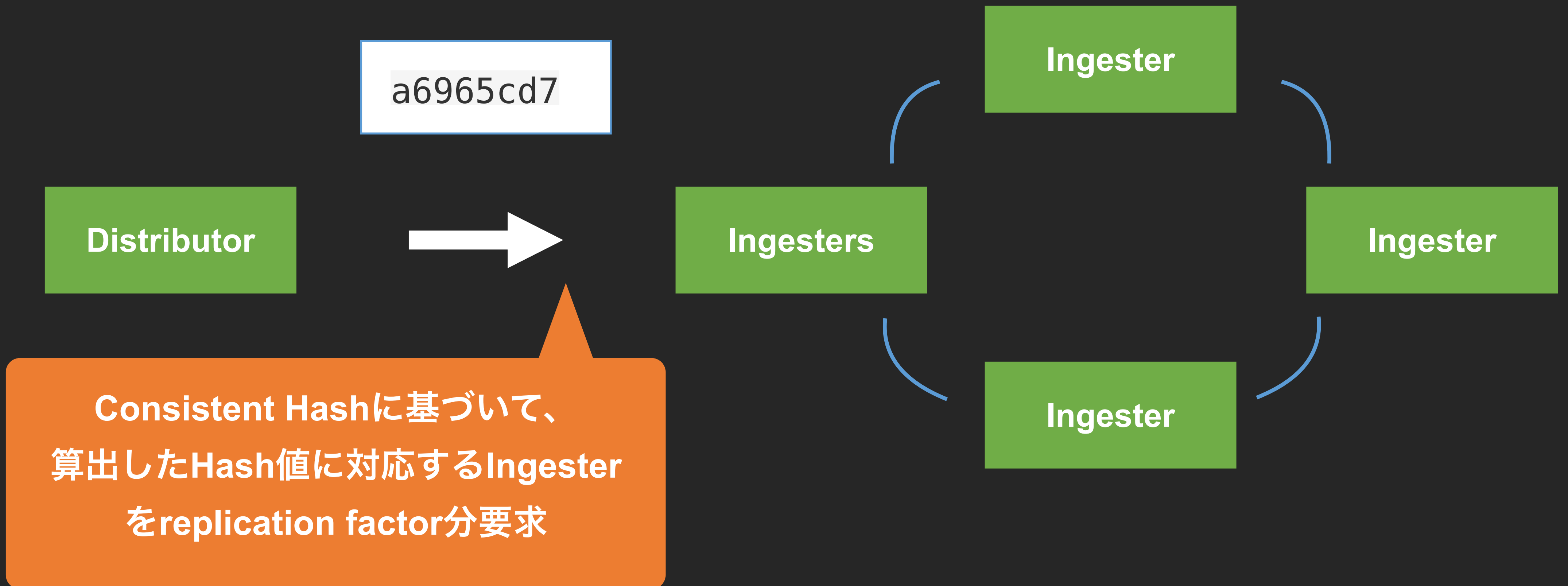
Distributor -> Ingesters

tenantID + {service="keystone", hostname="host1"}

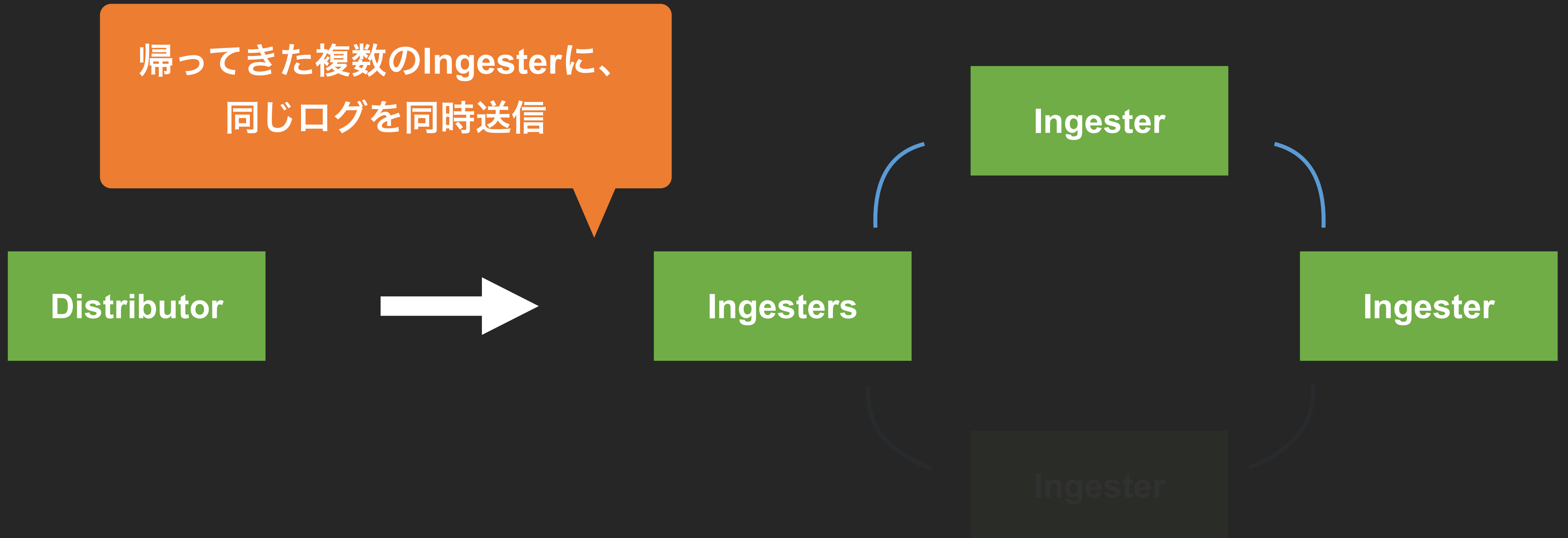
a6965cd7

FNV1-32bitでHash値を生成

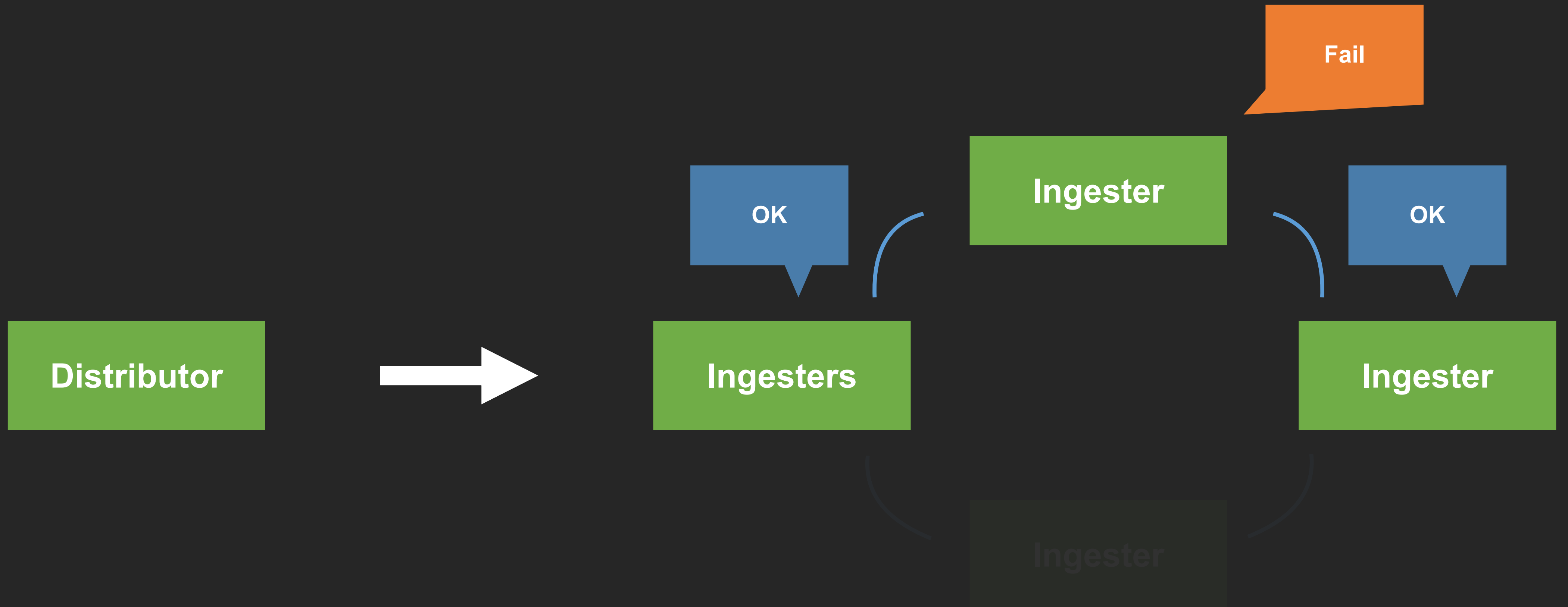
Distributor -> Ingesters



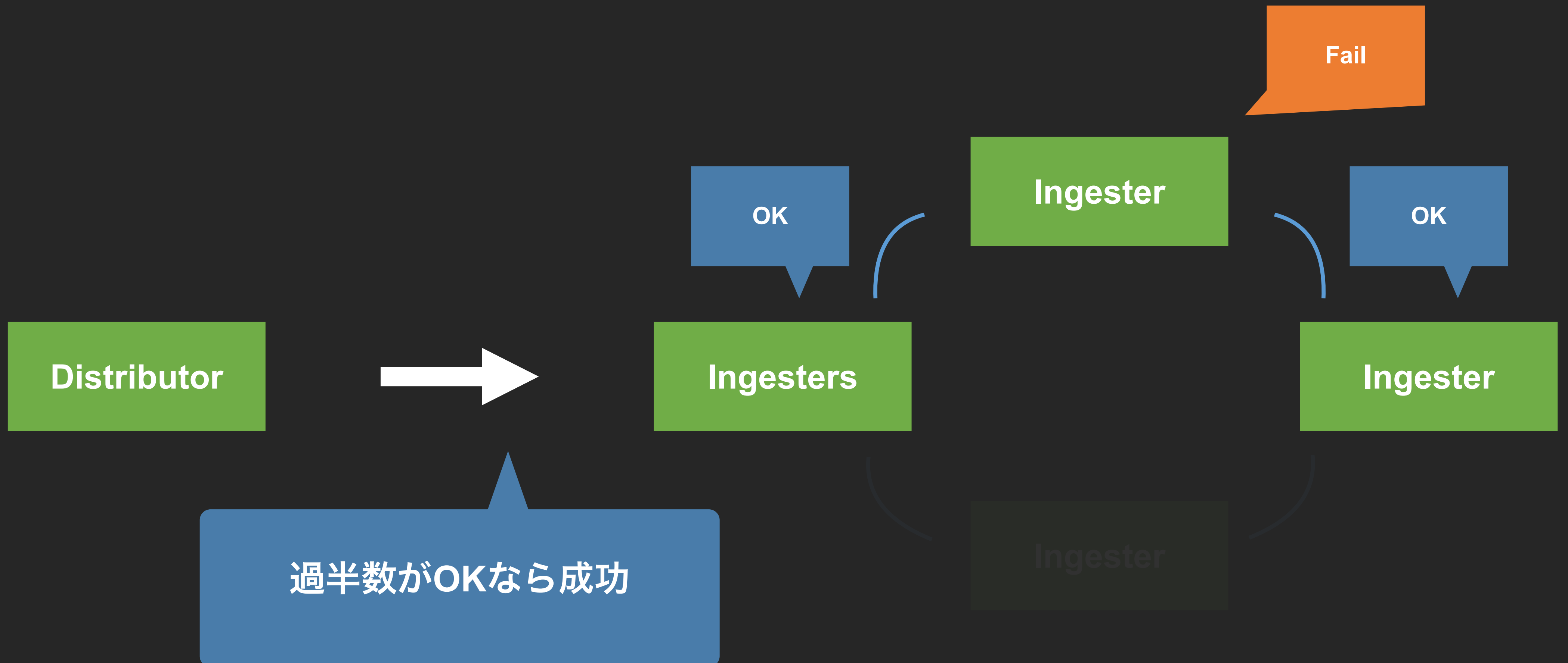
Distributor -> Ingesters



Distributor -> Ingesters

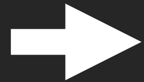


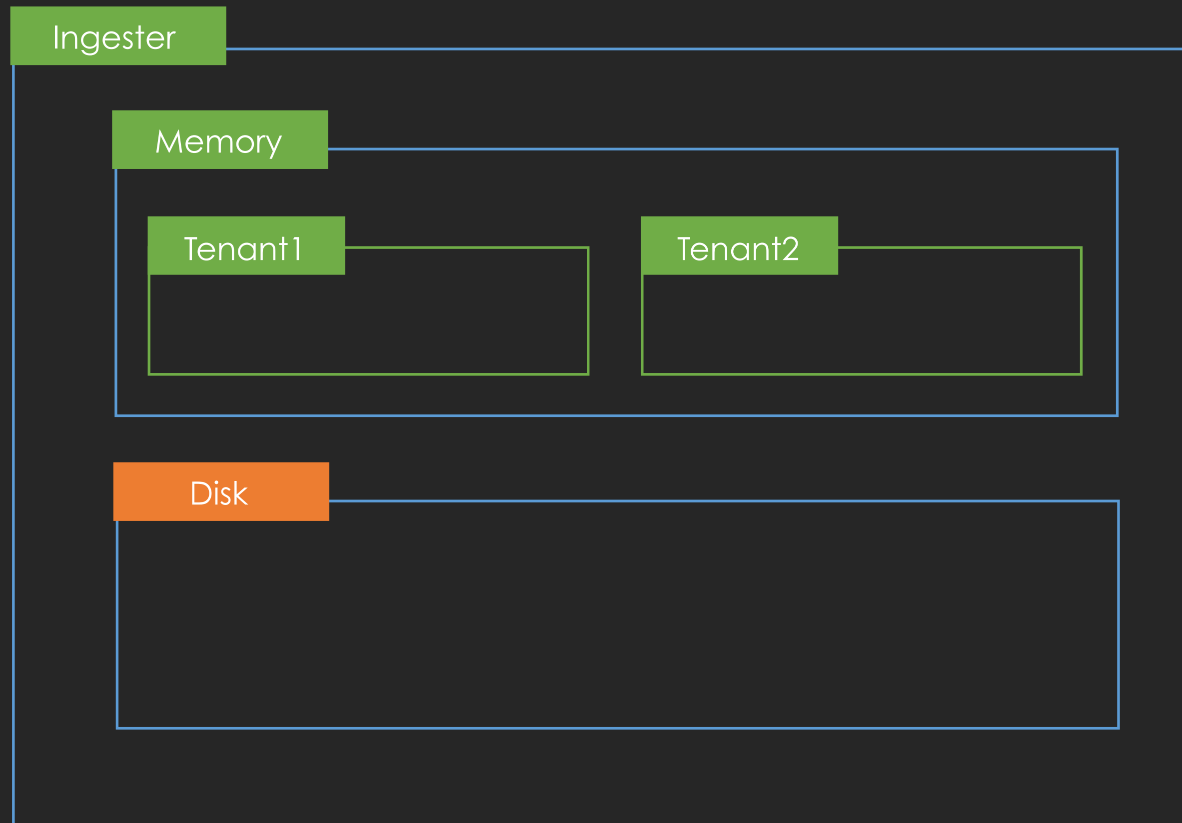
Distributor -> Ingesters



IngesterのRequest Handling

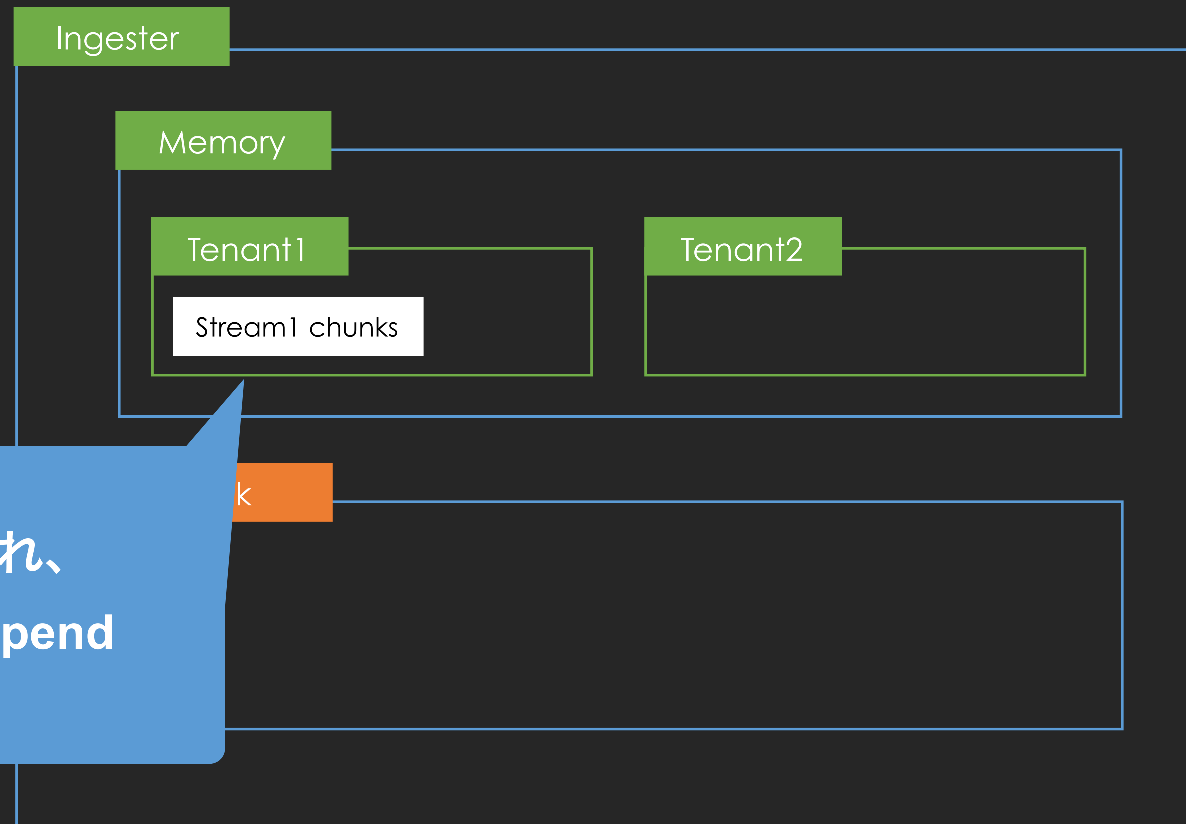
IngesterのRequest Handling



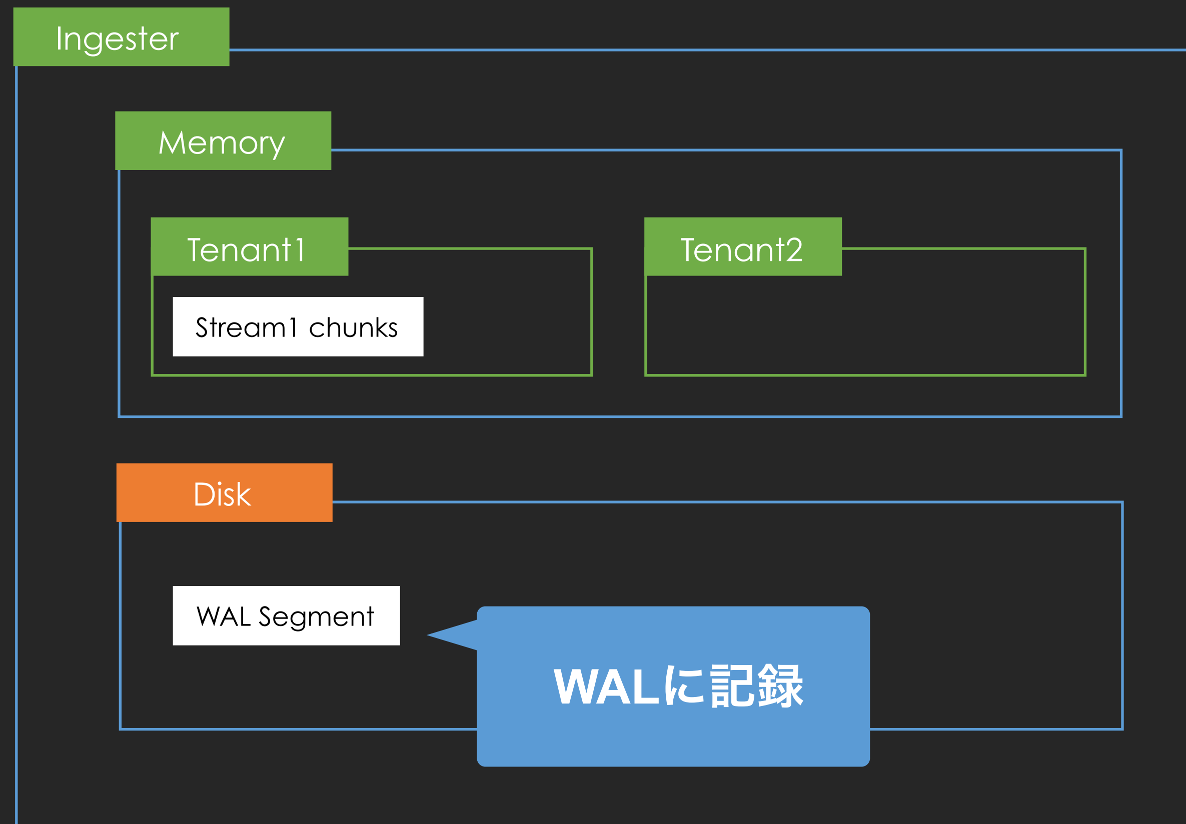


{service=“keystone”, hostname=“host1”}

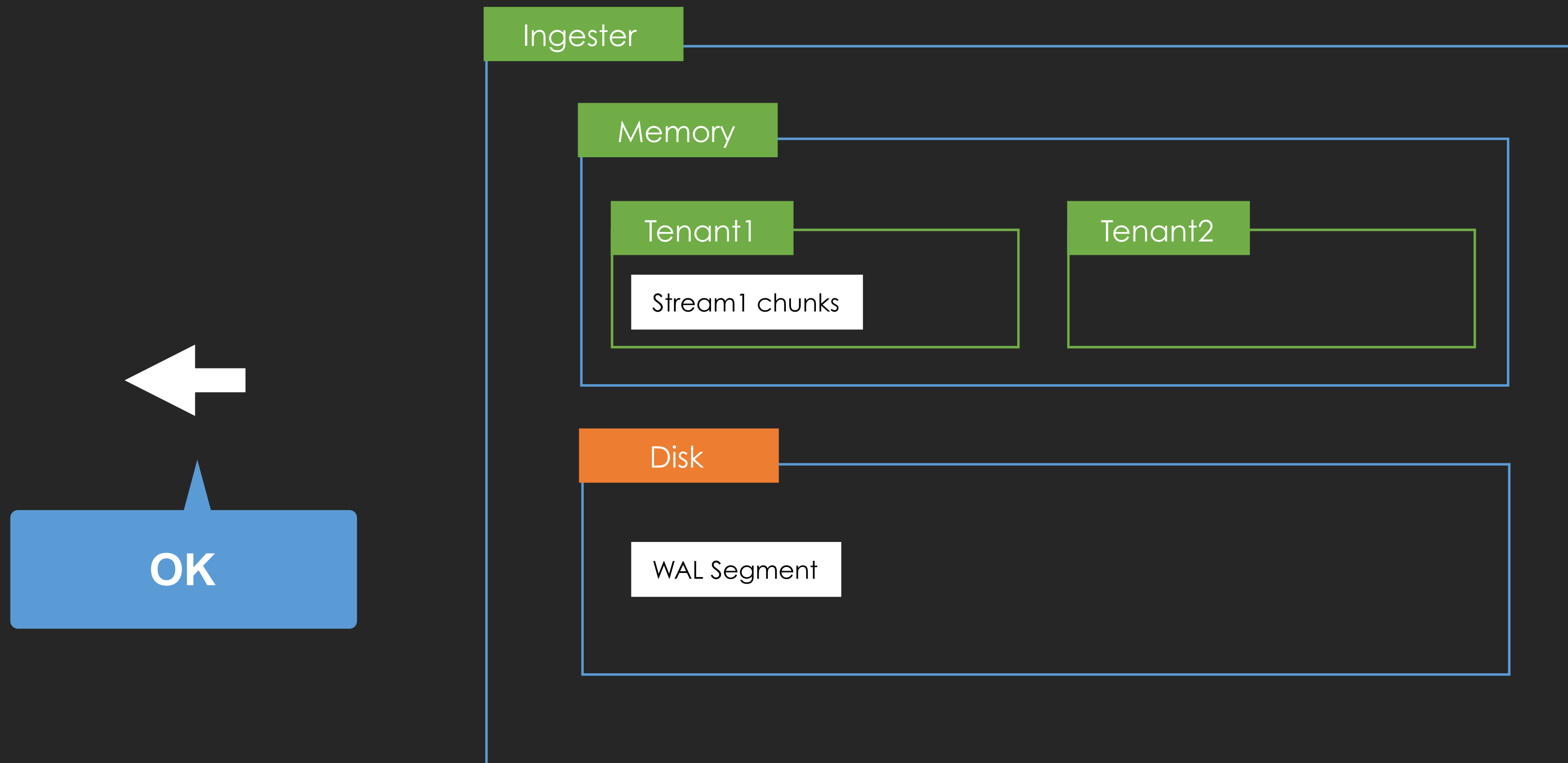
IngesterのRequest Handling



IngesterのRequest Handling



IngesterのRequest Handling



IngesterのRequest Handling

もしStream内で最後に受け取ったログの時間より

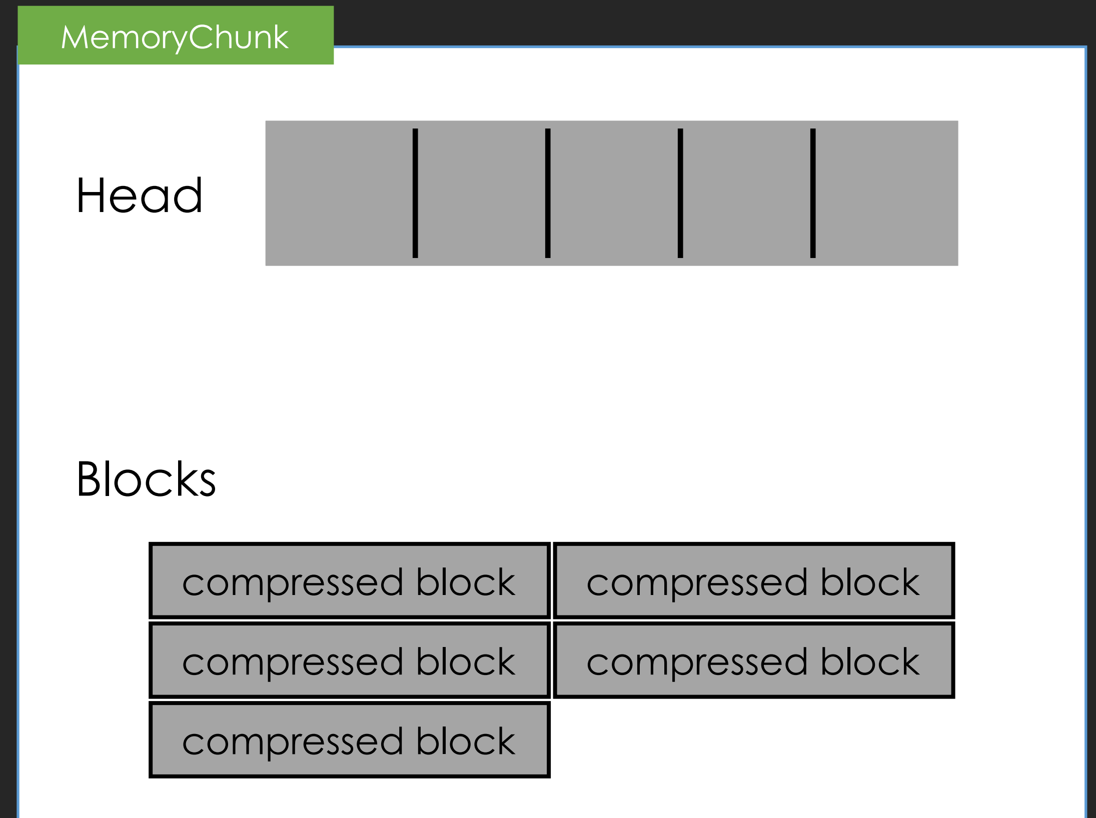
前の時間のログが来た場合はリクエストを失敗させる

Out of order entry error

Chunkバッファリング

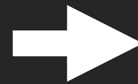
IngesterのChunkバッファリング

- 一定数のログをStreamごとに「Chunk」にまとめる
- Chunkはメモリ上に保存される
- Head、Blocksという配列を保持



IngesterのChunkバッファリング

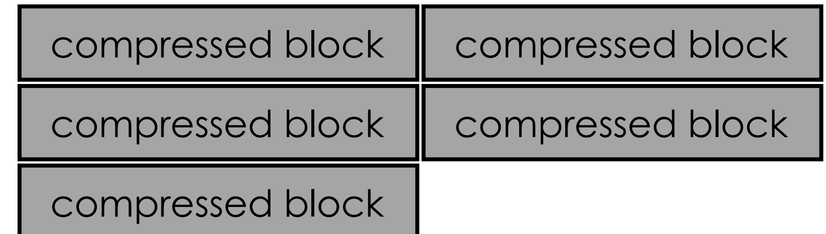
{service=“keystone”, hostname=“host1”}



MemoryChunk

Head

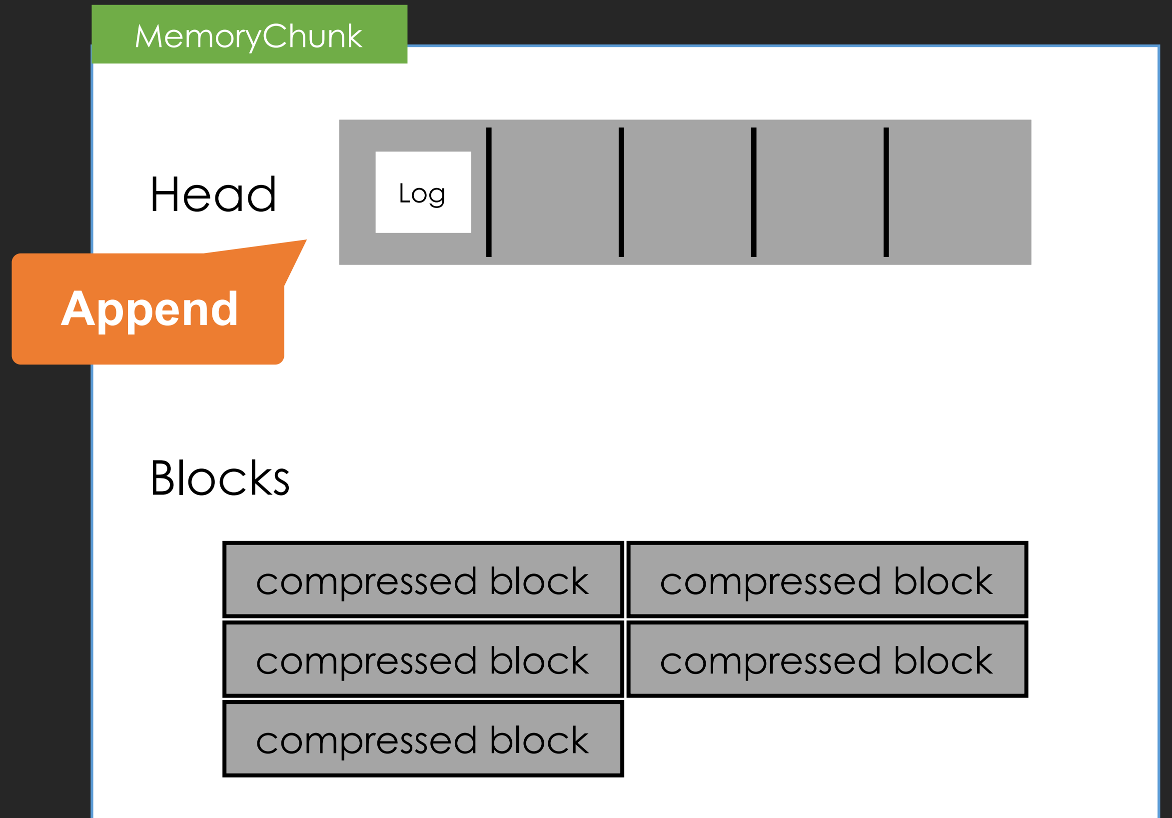
Blocks



IngesterのChunkバッファリング

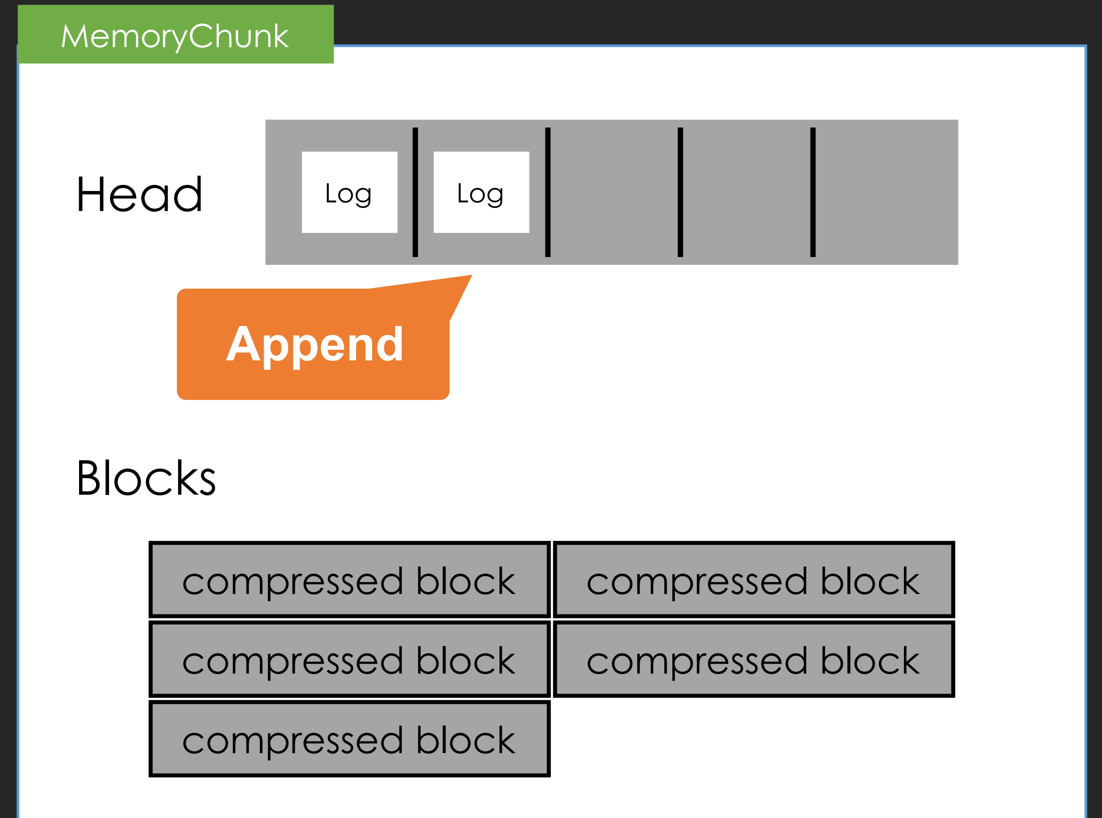
受け取ったログは一旦Headへ

Appendされる



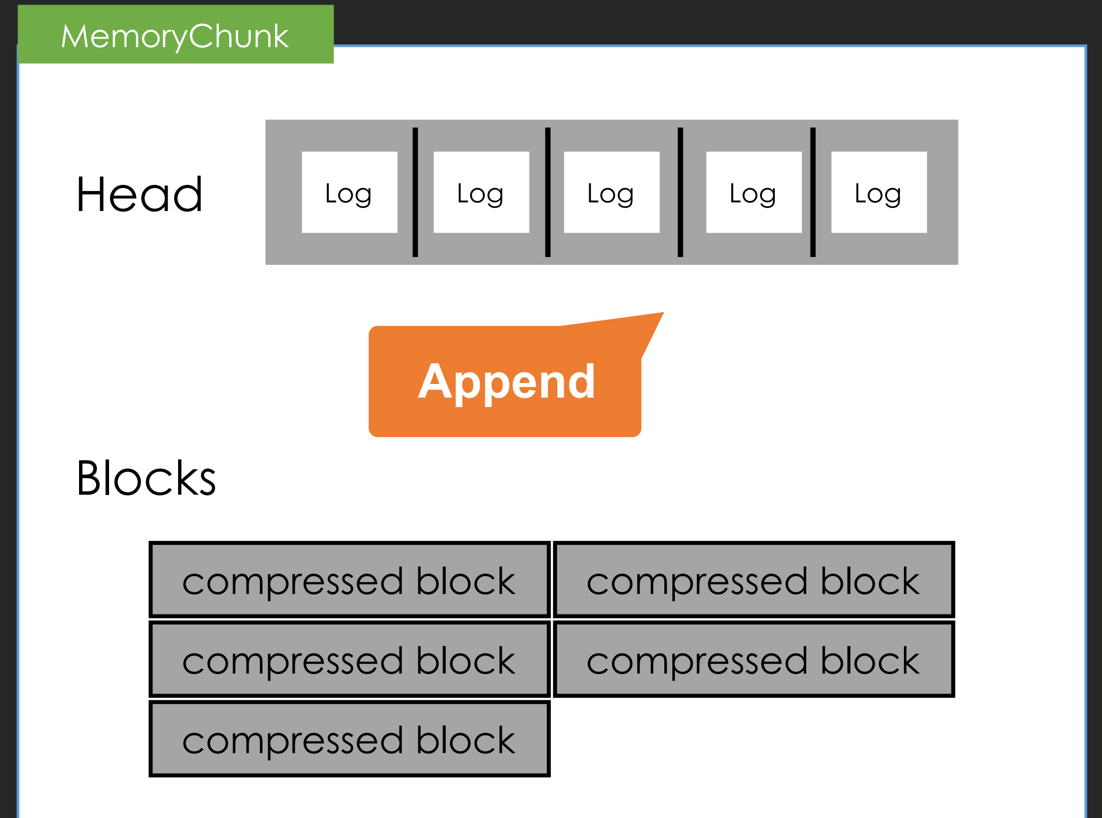
IngesterのChunkバッファリング

1 block size分溜まるまで繰り返す



IngesterのChunkバッファリング

1 block size分溜まるまで繰り返す

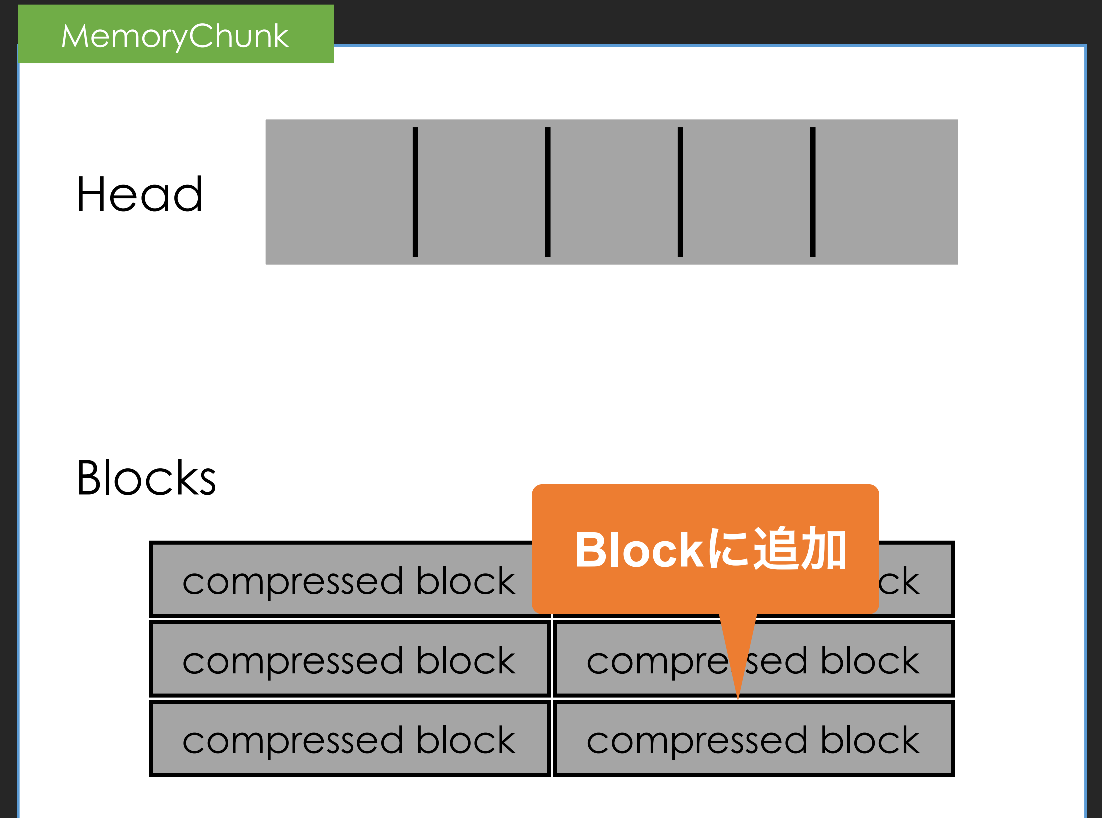


IngesterのChunkバッファリング

一定サイズ溜まったら

設定した形式で圧縮し、

blocksに追加する

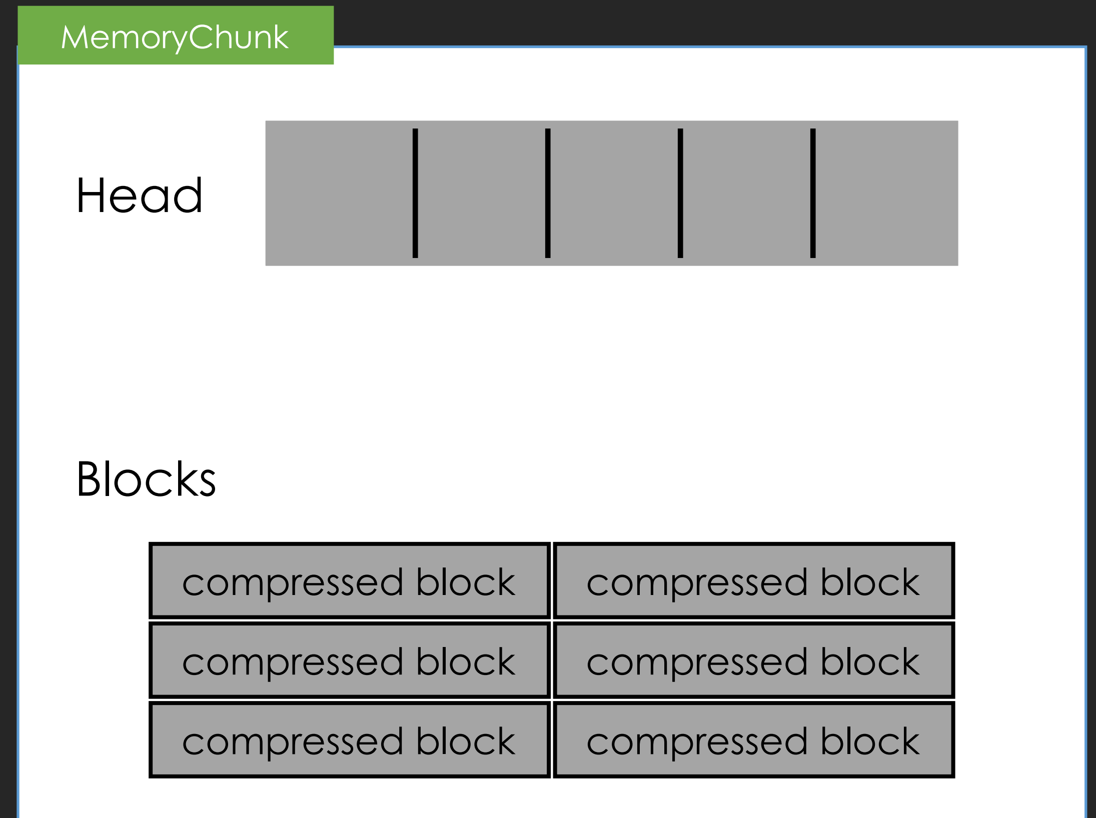


IngesterのChunkバッファリング

Blockが一定数溜まったら

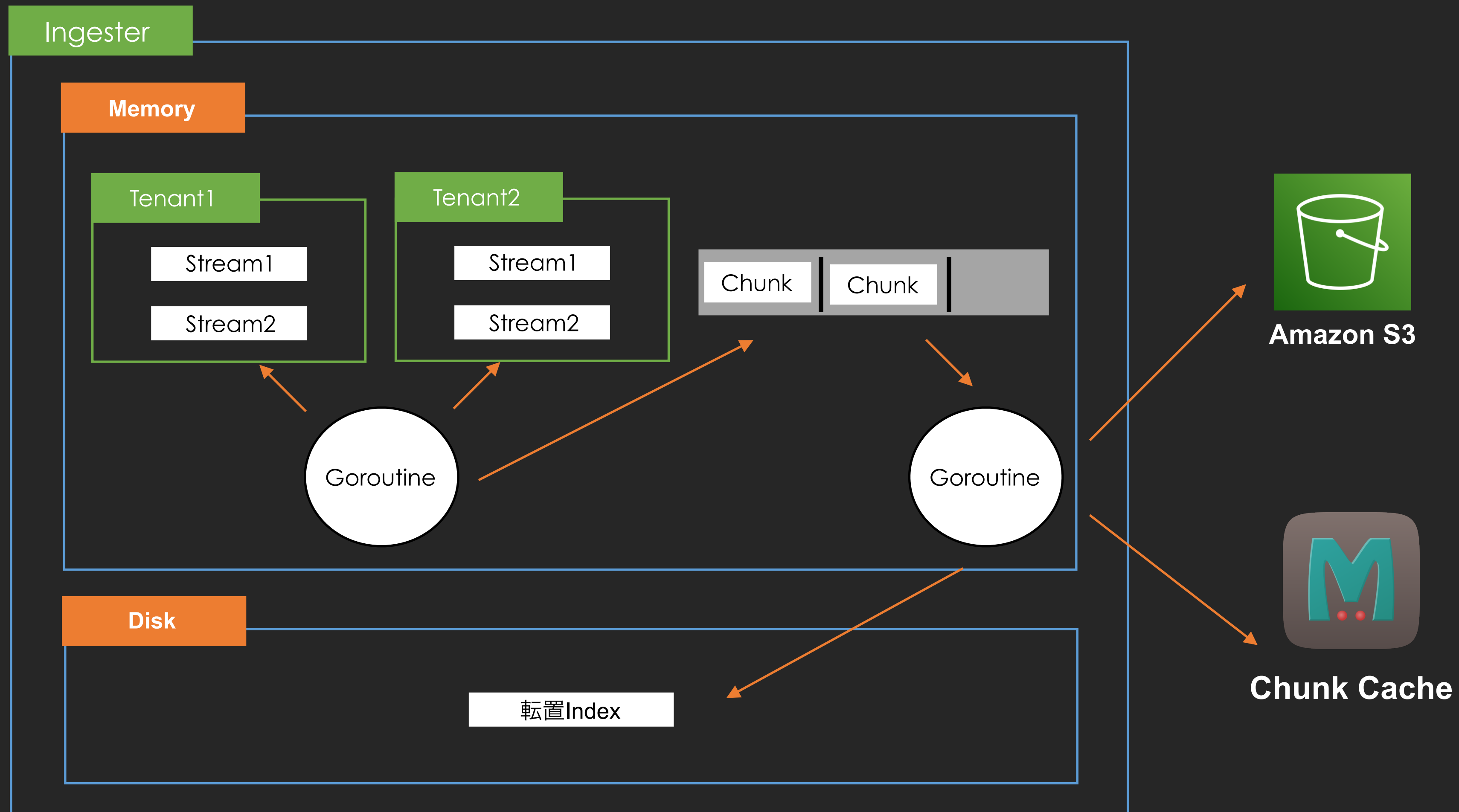
Read Only Modeになり、Flush Queueへ

(Default 10 blocks, target chunk sizeで指定)

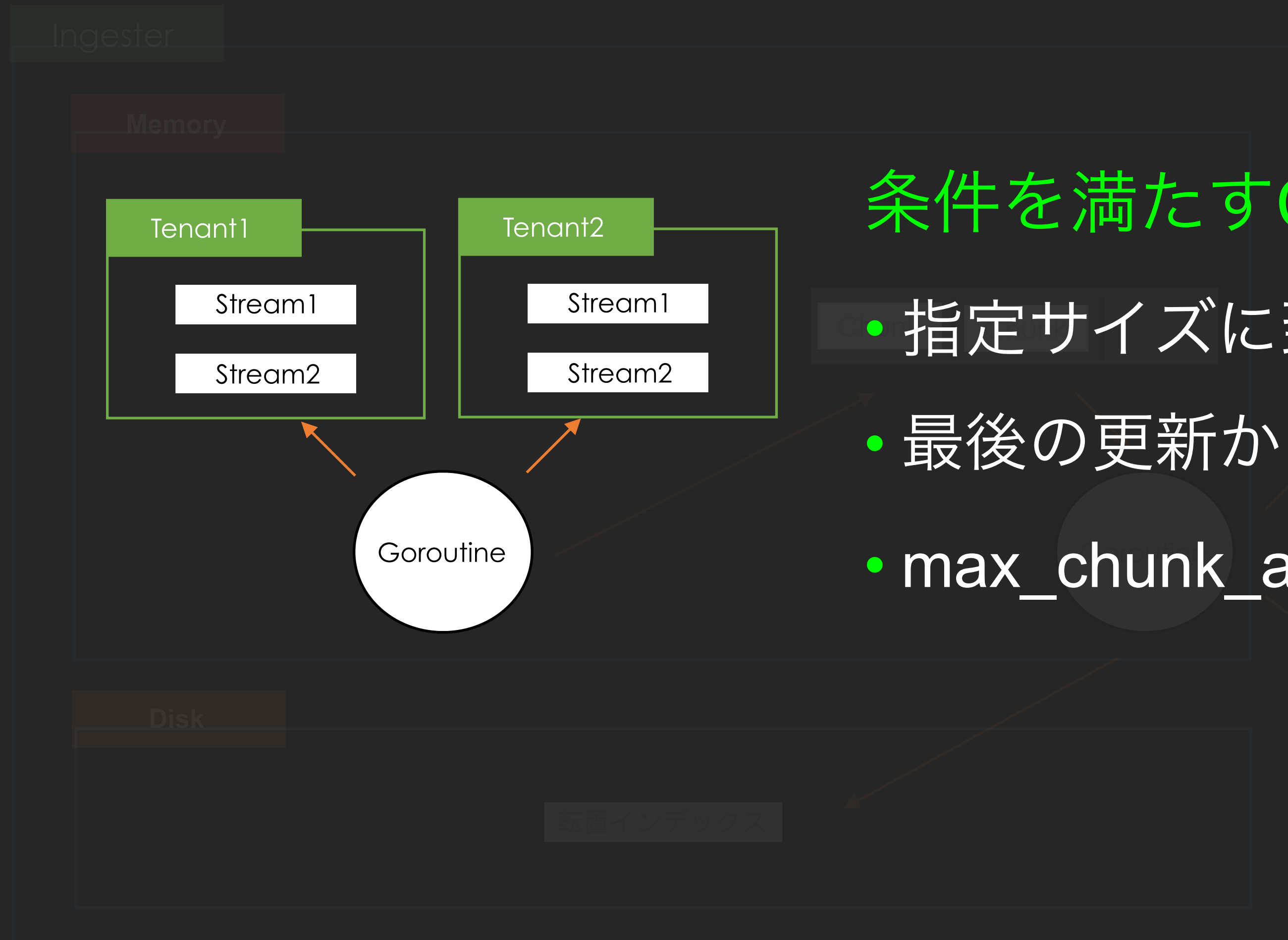


IngesterからStorageへFlush

ChunkのFlush



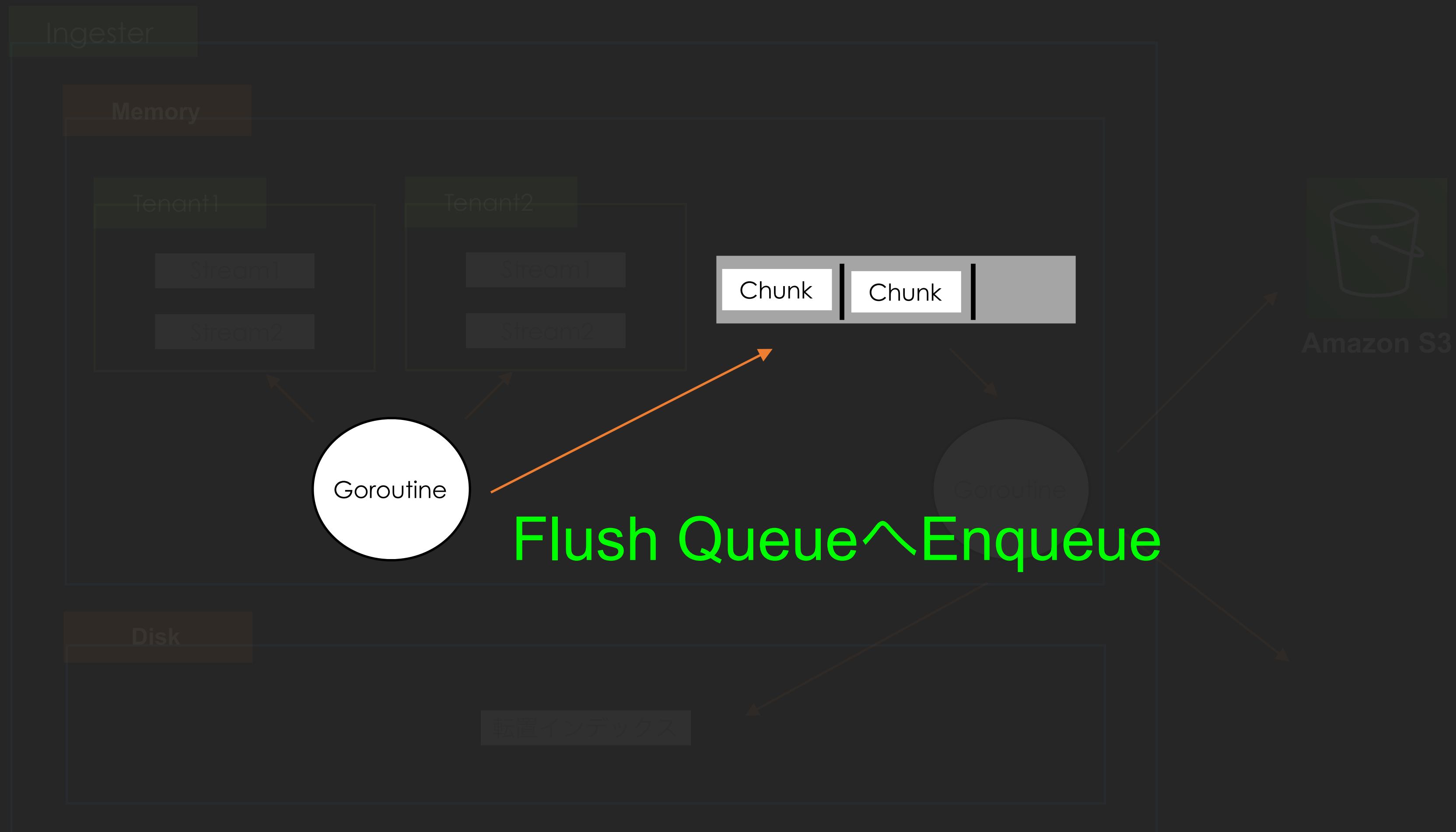
ChunkのFlush



条件を満たすChunkを検知

- 指定サイズに到達
- 最後の更新からchunk idle period経過
- max_chunk_age経過

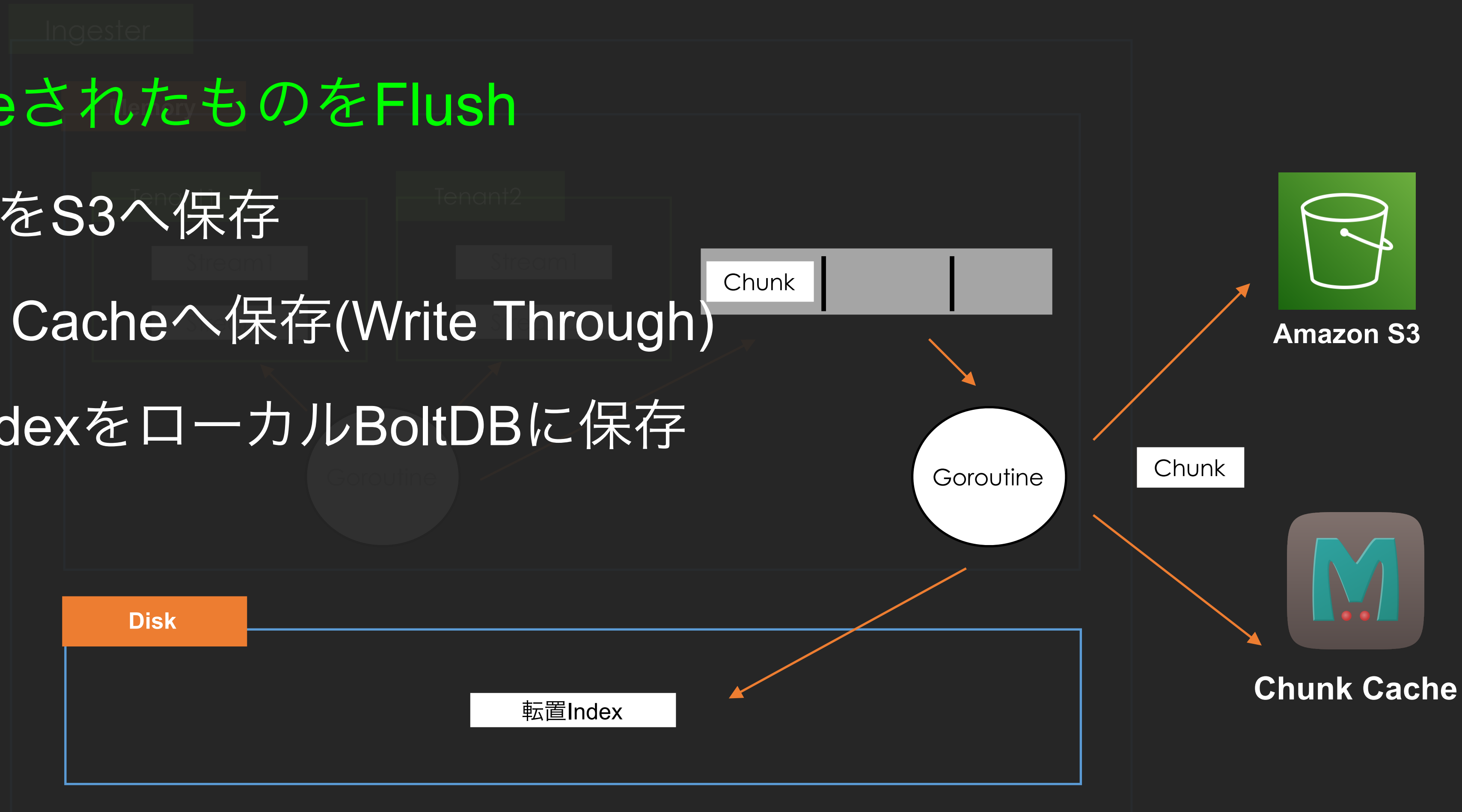
ChunkのFlush



ChunkのFlush

EnqueueされたものをFlush

1. ChunkをS3へ保存
2. Chunk Cacheへ保存(Write Through)
3. 転置IndexをローカルBoltDBに保存



ChunkのFlush

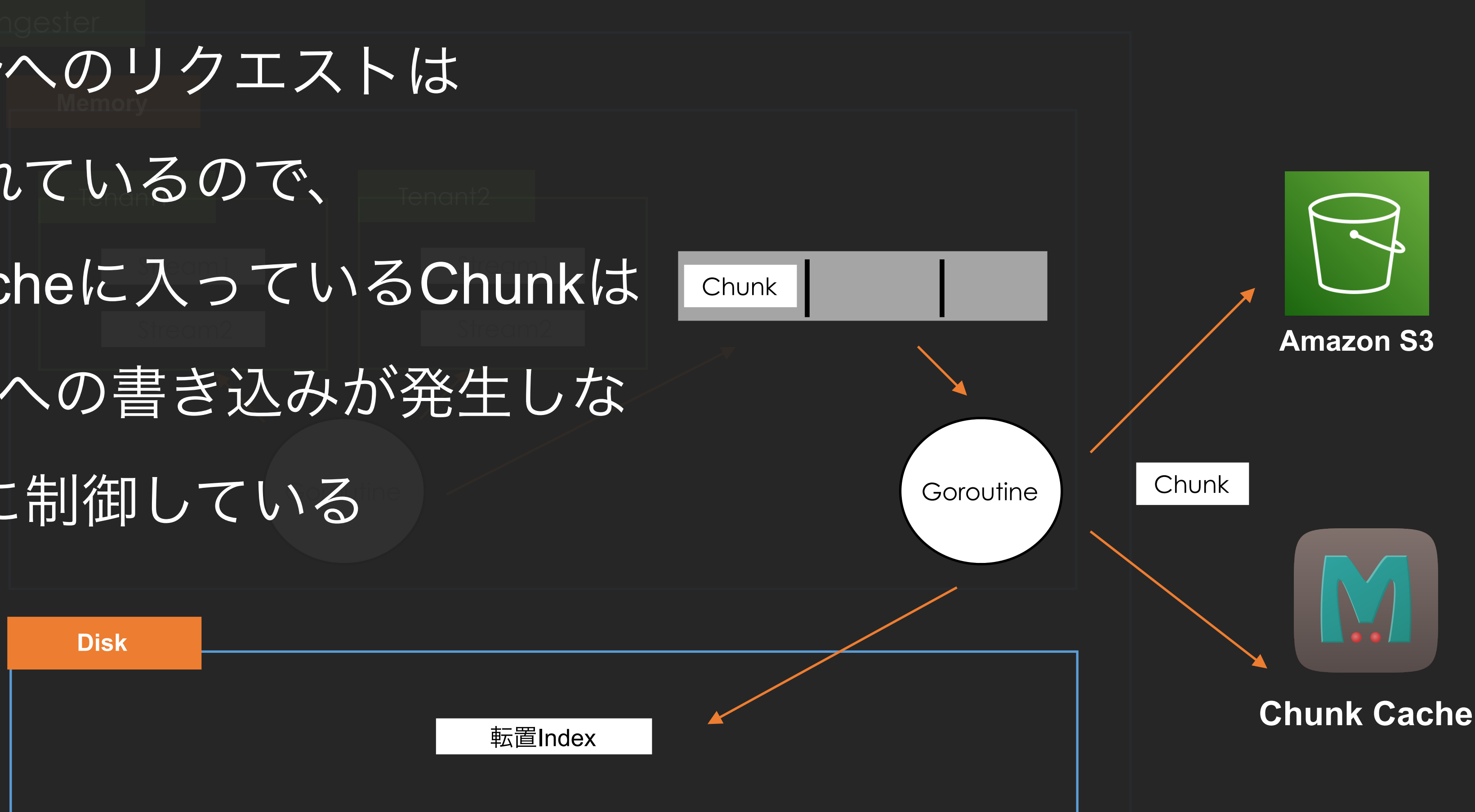
Ingesterへのリクエストは

複製されているので、

既にCacheに入っているChunkは

Storageへの書き込みが発生しな

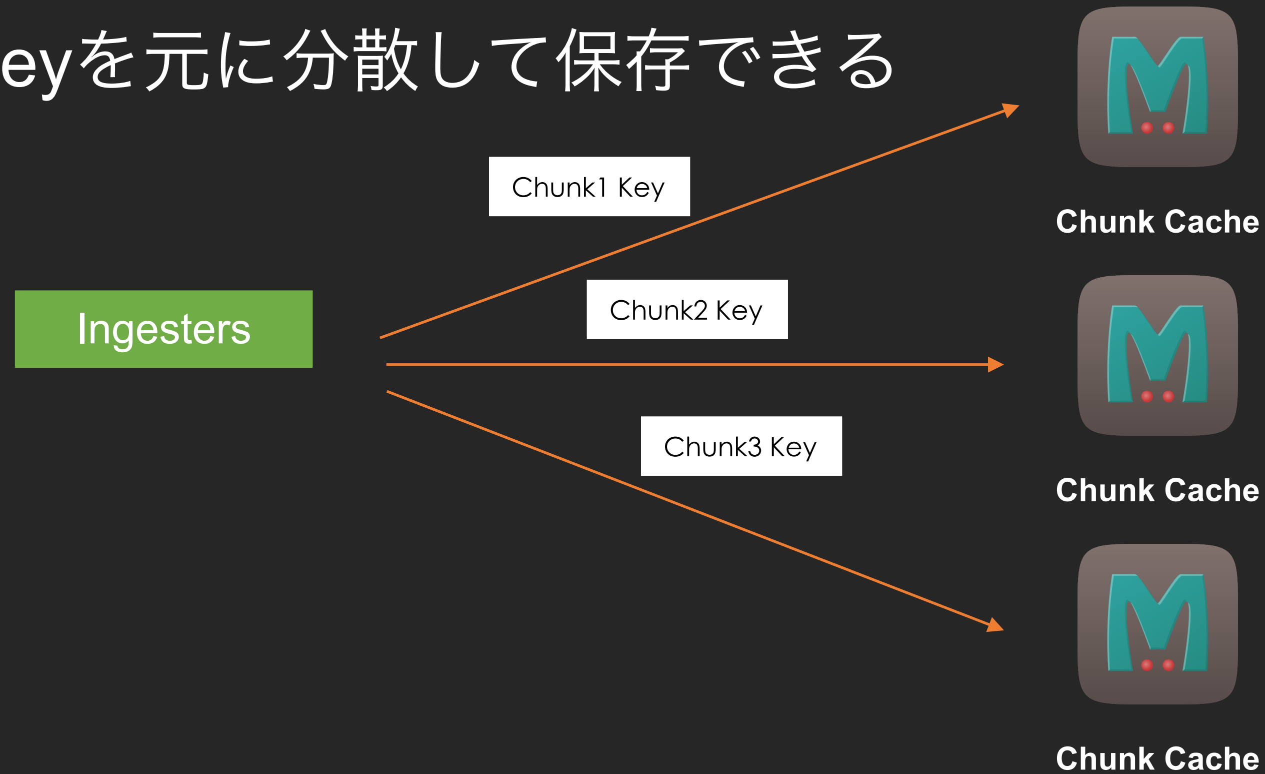
いように制御している



Cacheの負荷分散

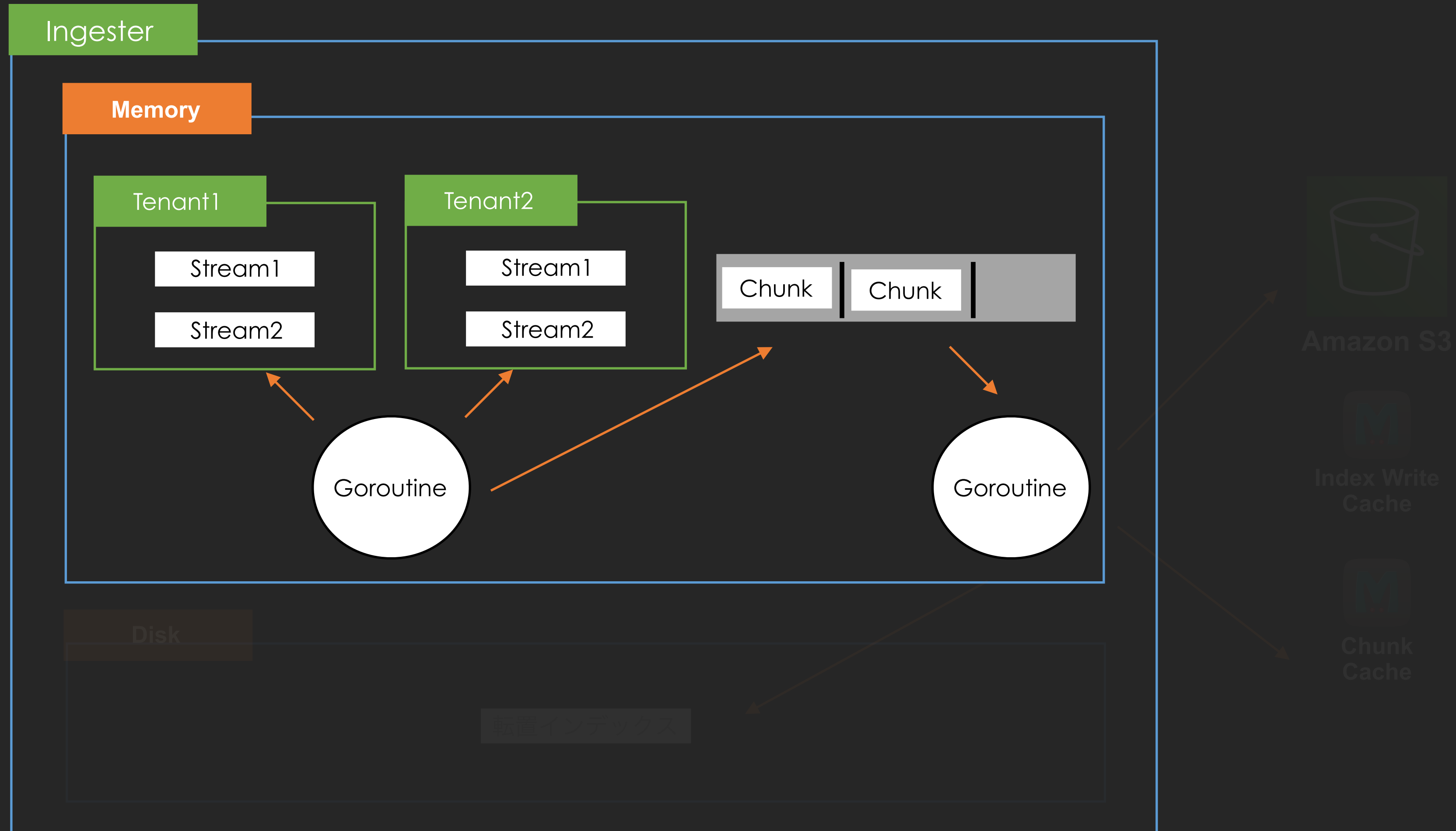
設定によってConsistent Hashにより、

ChunkのKeyを元に分散して保存できる

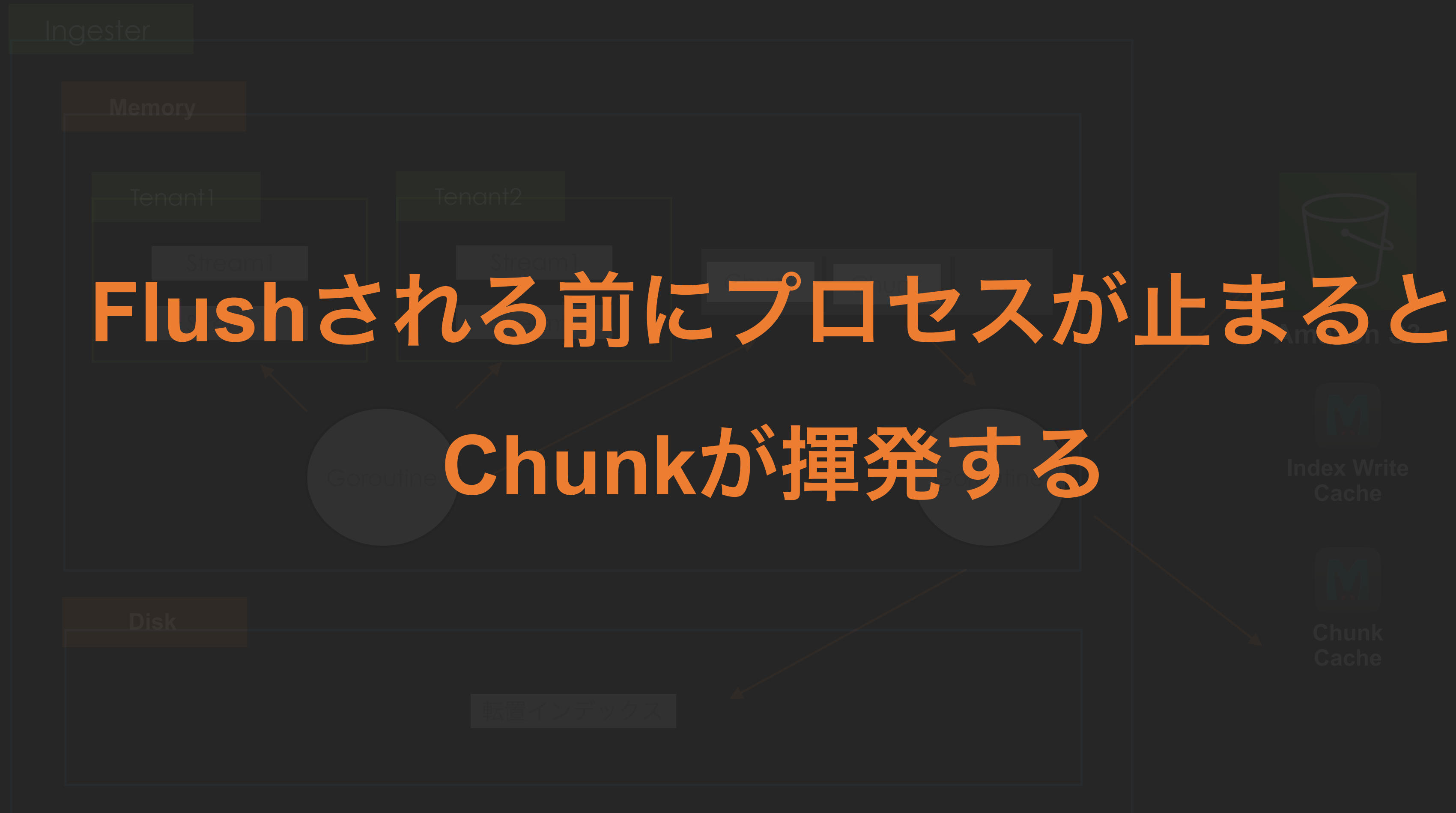


Write Ahead Log

ChunkのFlush(再掲)



IngesterのChunkの持ち方



Flushされる前にプロセスが止まると

Chunkが揮発する

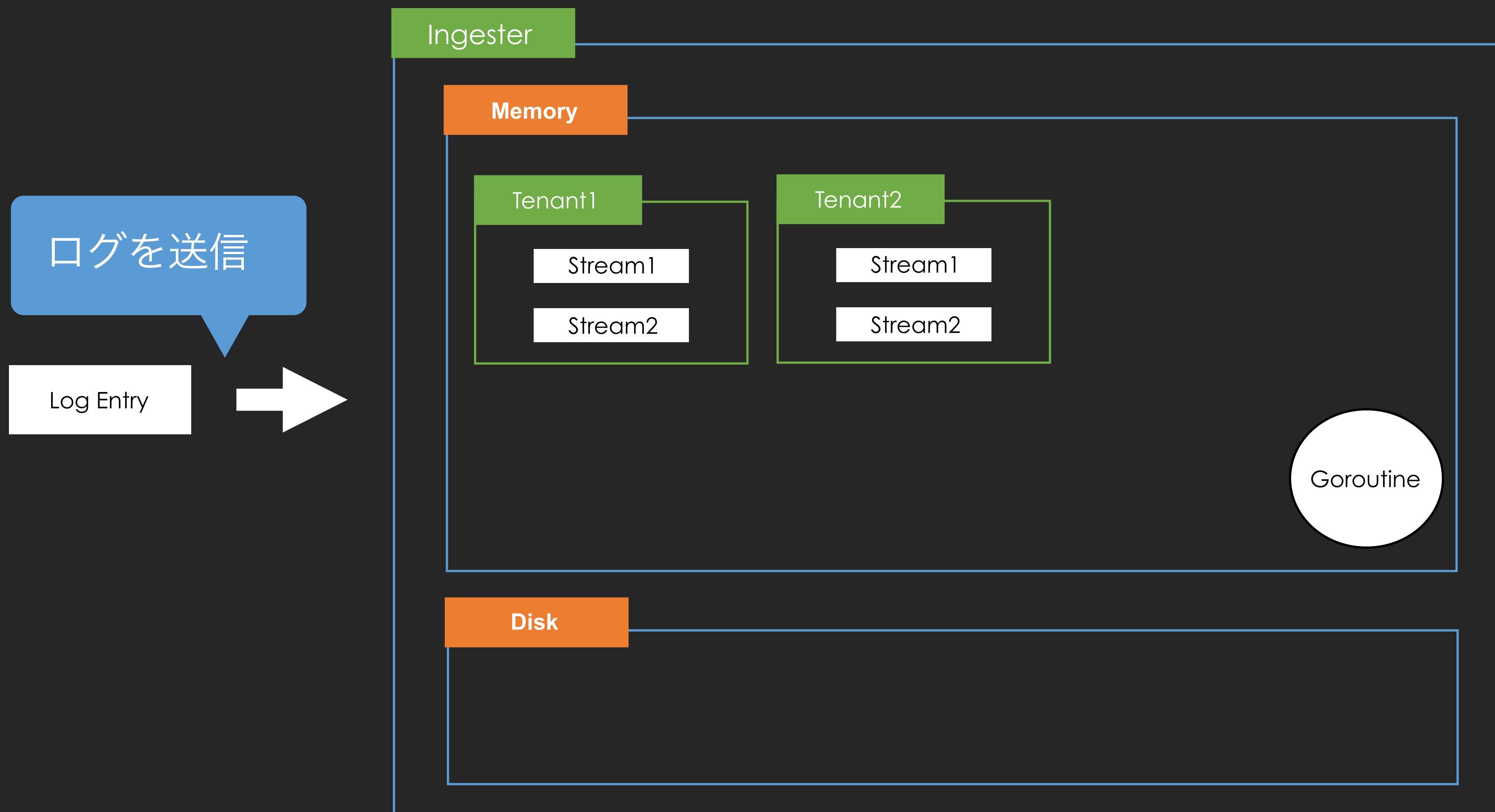
WALの意義

永続化してMemoryの揮発に備える

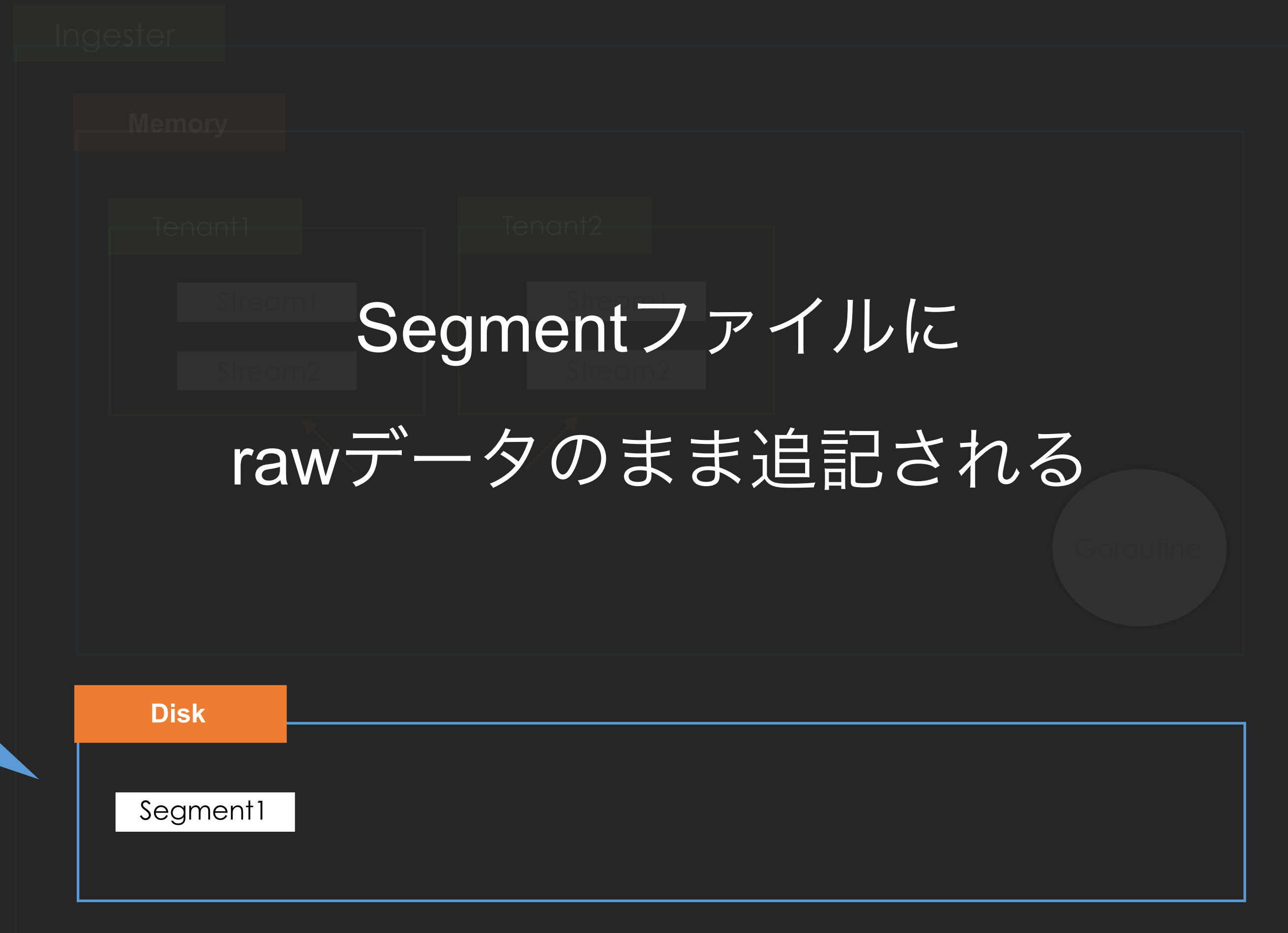
LokiのWALの特徴

- ログ受信時に、MemoryとWAL両方に書き込む
- WAL書き込みが失敗しても処理を失敗させない
- IngesterのプロセスのL起動時にWALからの復旧処理が入る
- 一定期間ごとに不要なWALはパージされる

WALの仕組み



WALの仕組み

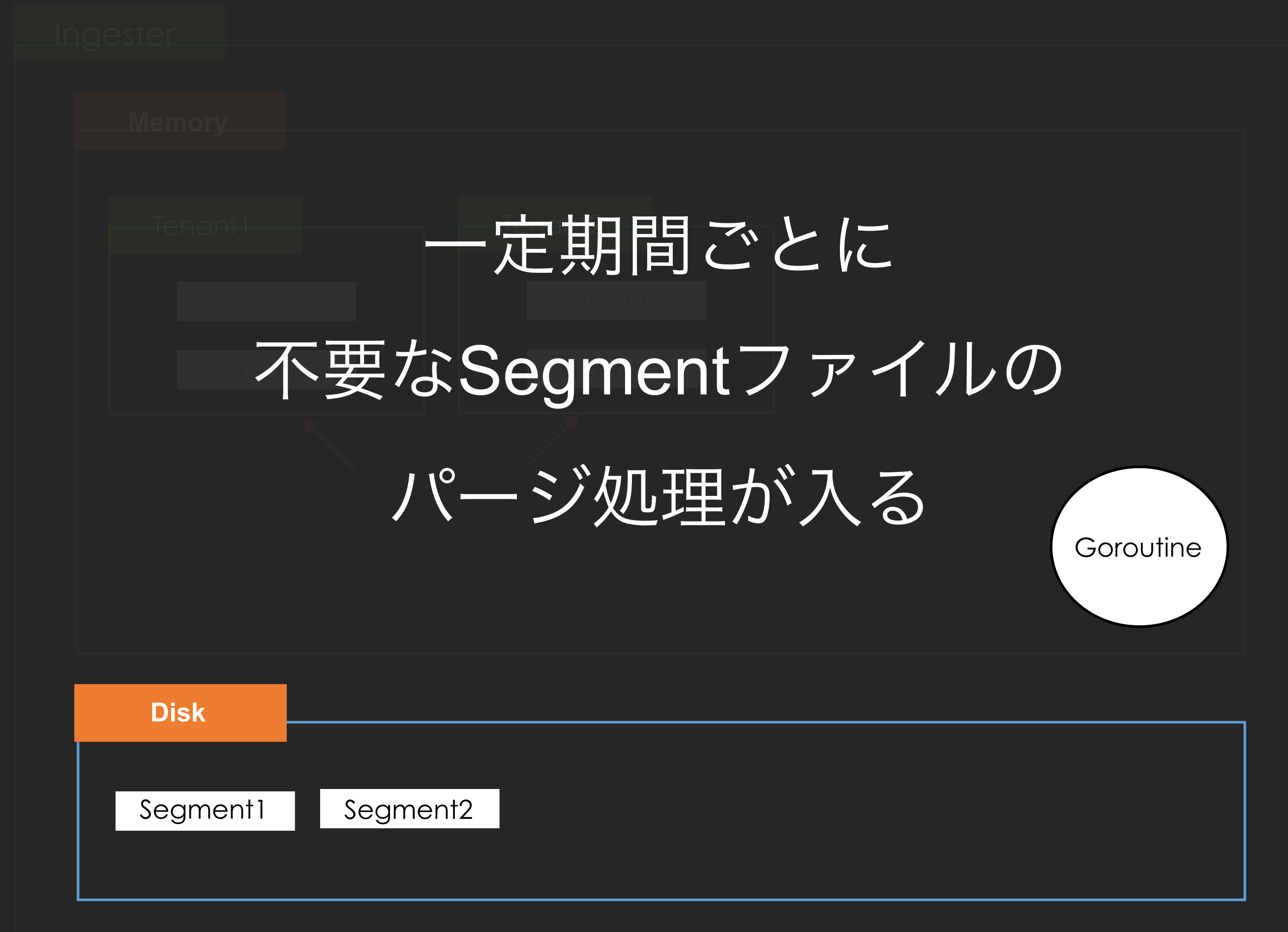


WALの仕組み

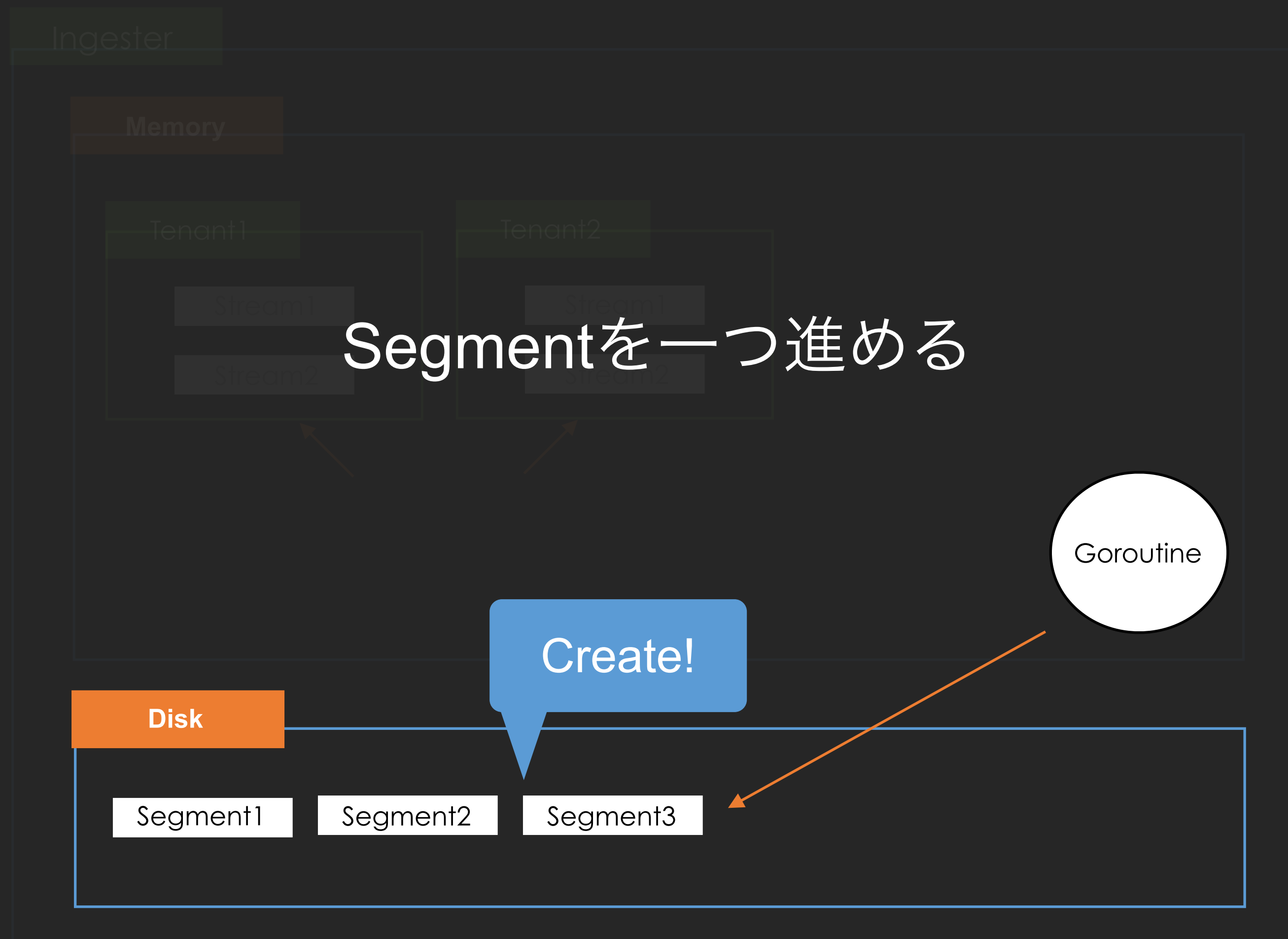
1ファイルが大きくなってくると

別のSegmentファイルを作成

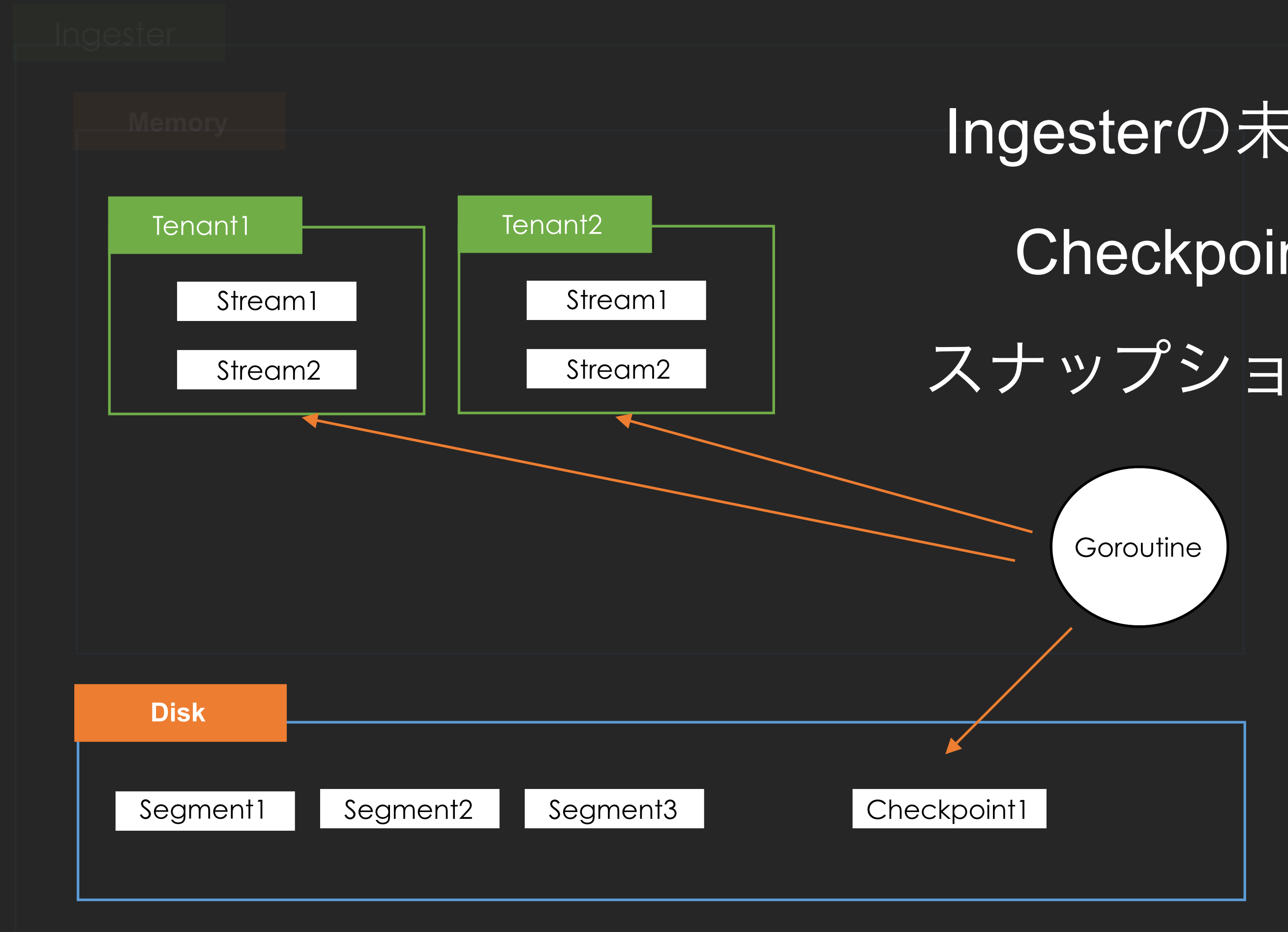
WALの仕組み



WALの仕組み



WALの仕組み

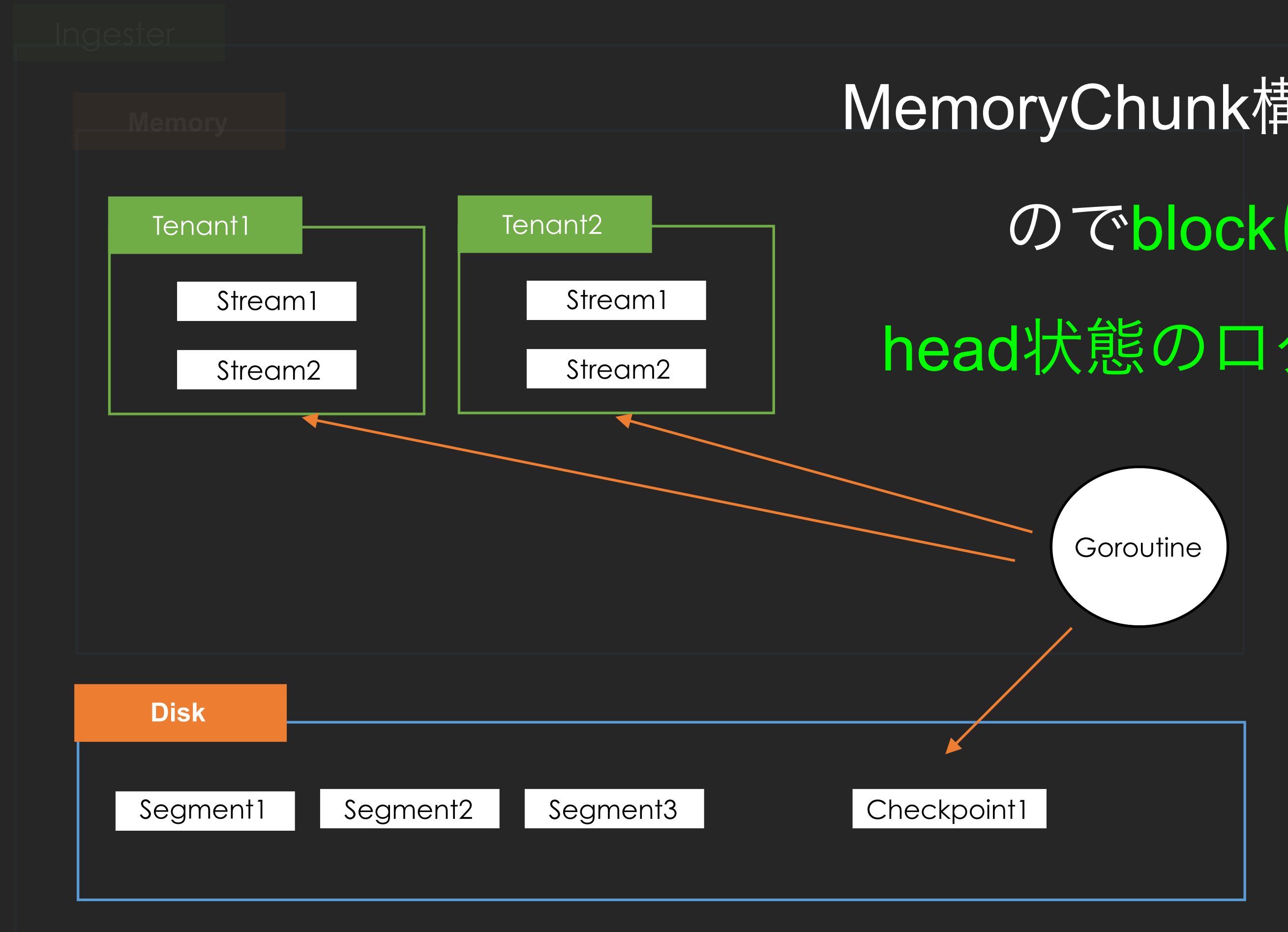


Ingesterの未Flush Chunkを

Checkpointと呼ばれる

スナップショットとして保存

WALの仕組み

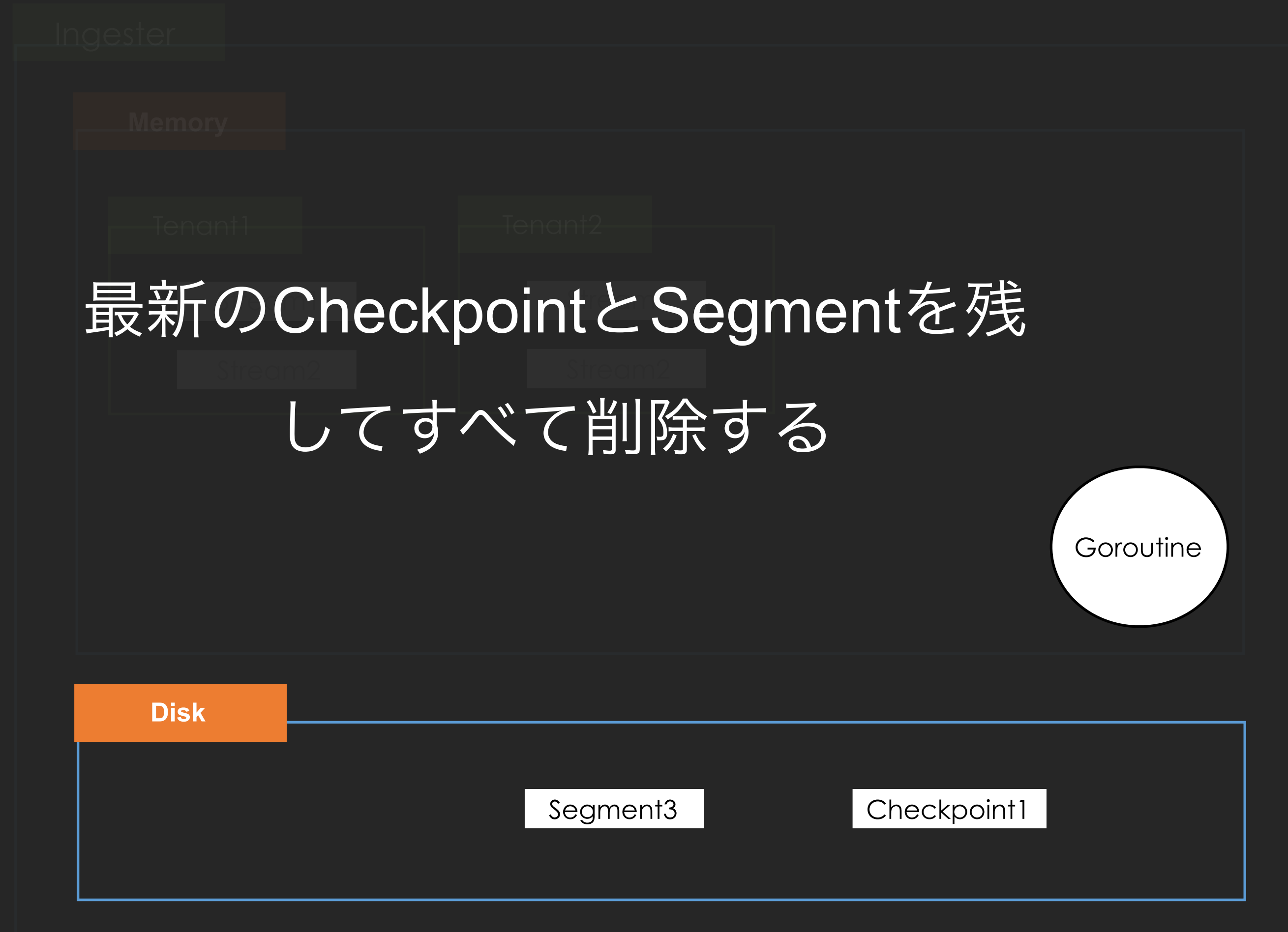


MemoryChunk構造のまま保存する

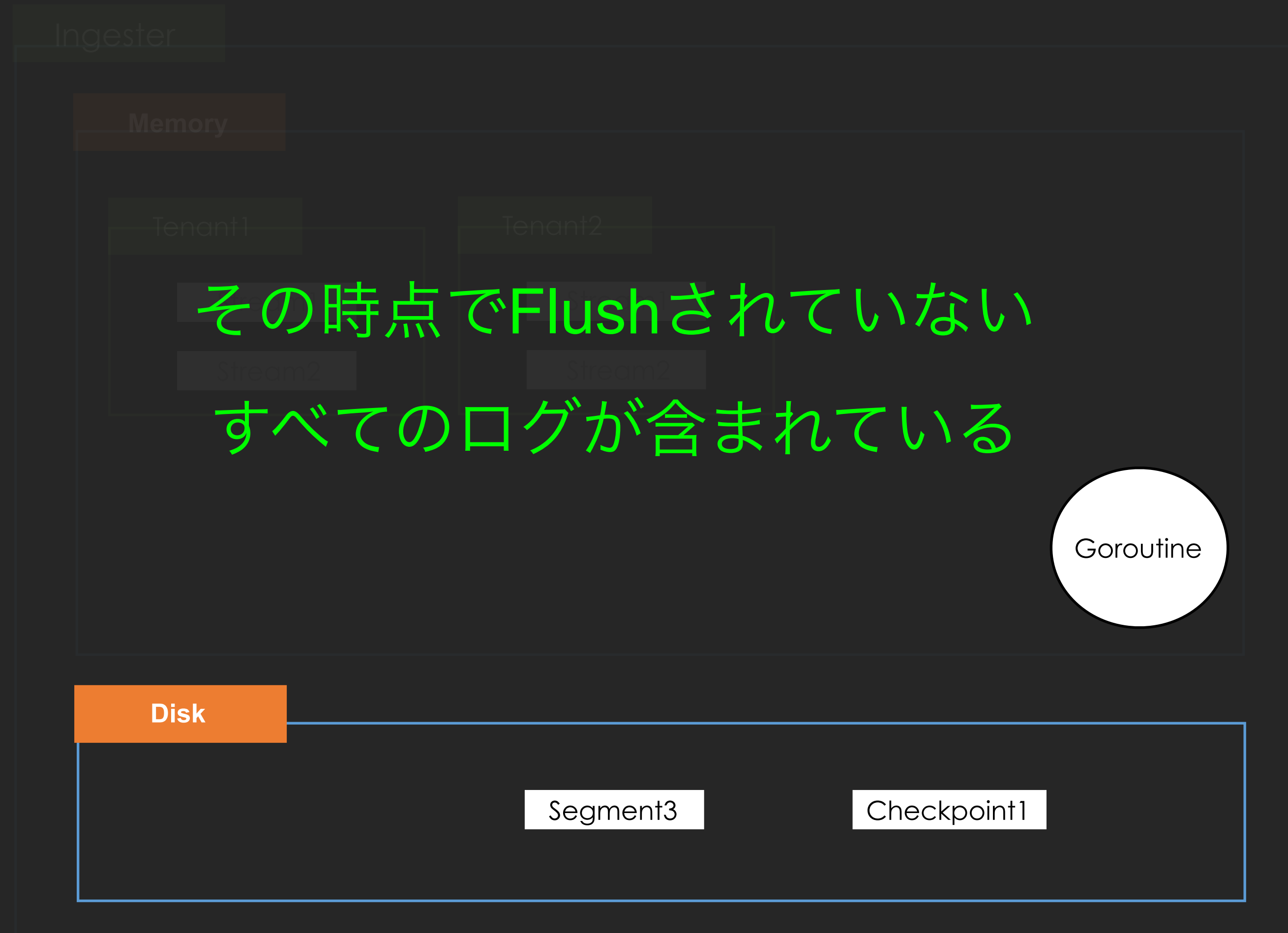
のでblockは圧縮形式、

head状態のログは非圧縮となる

WALの仕組み



WALの仕組み



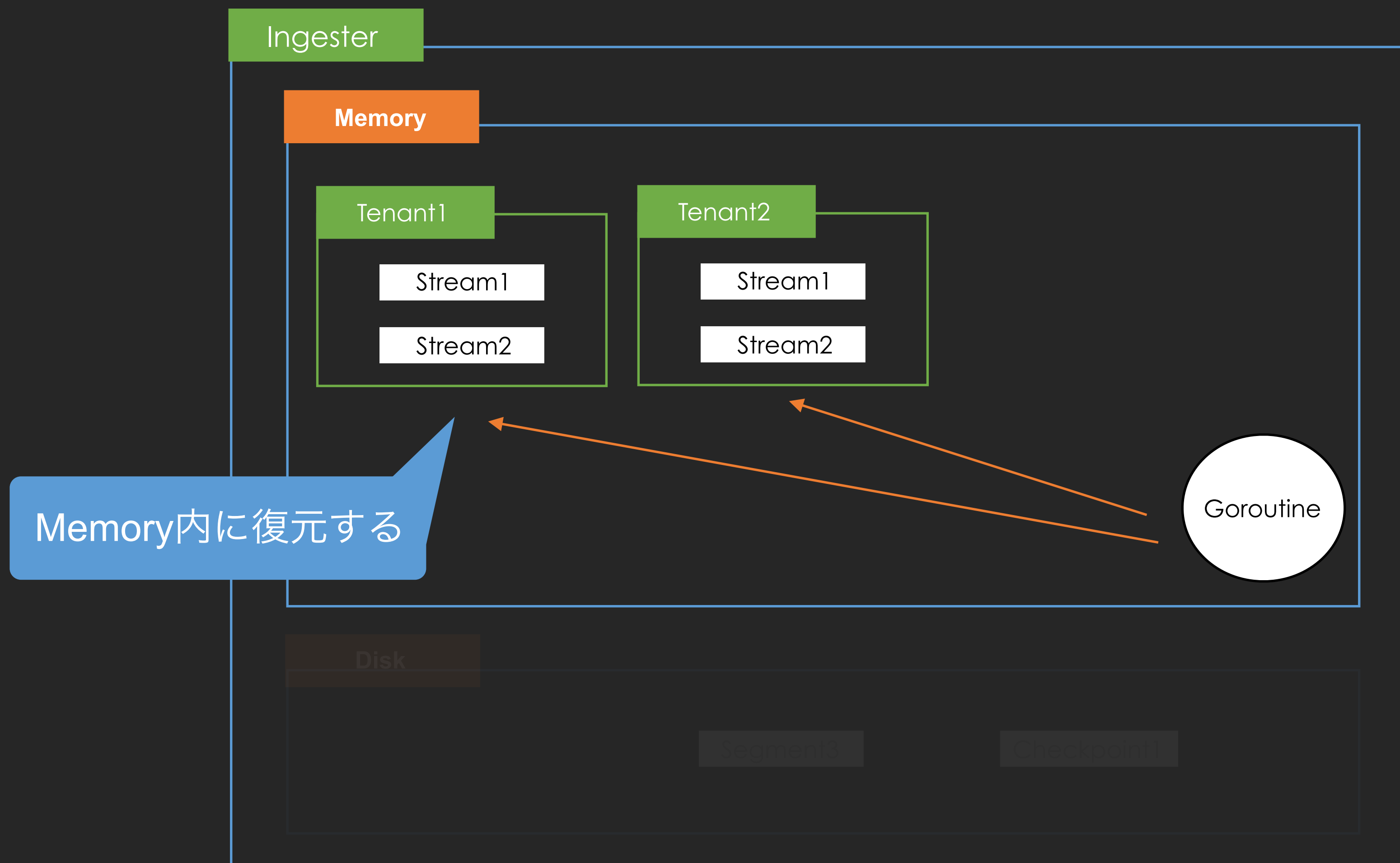
WALの仕組み

Ingesterの起動時には

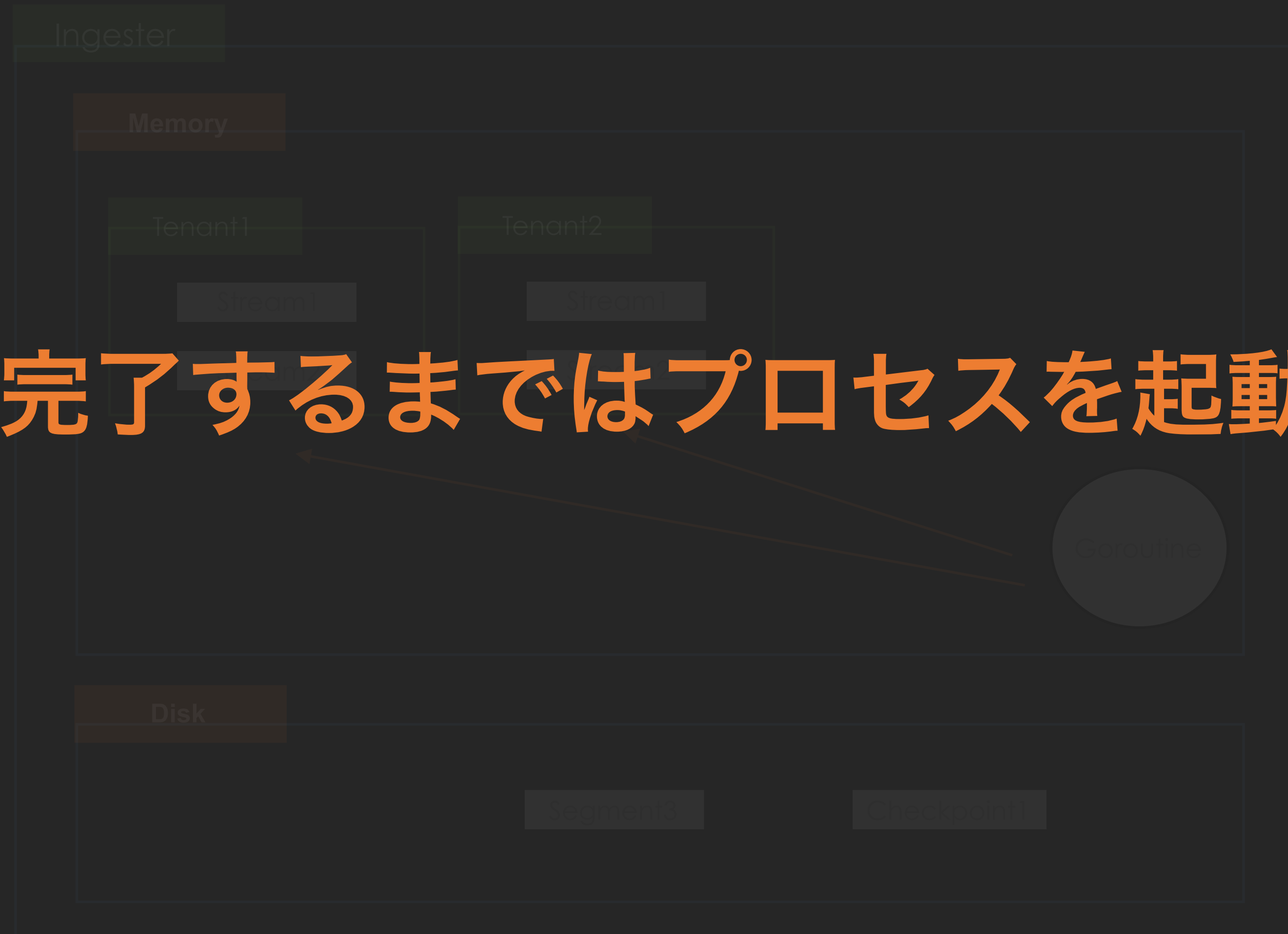
Diskから

SegmentとCheckpointを読み取る

WALの仕組み

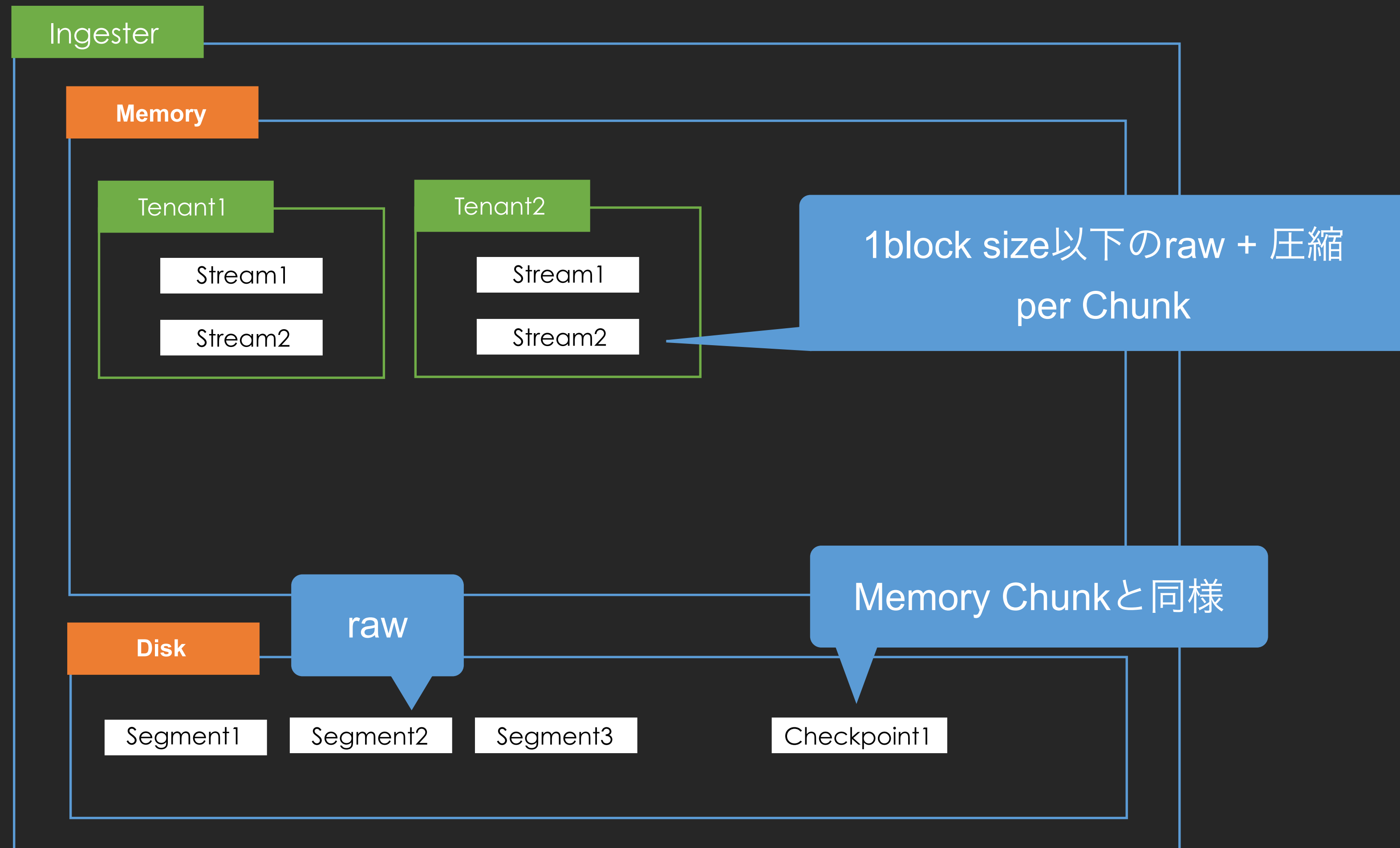


WALの仕組み



復元が完了するまではプロセスを起動しない

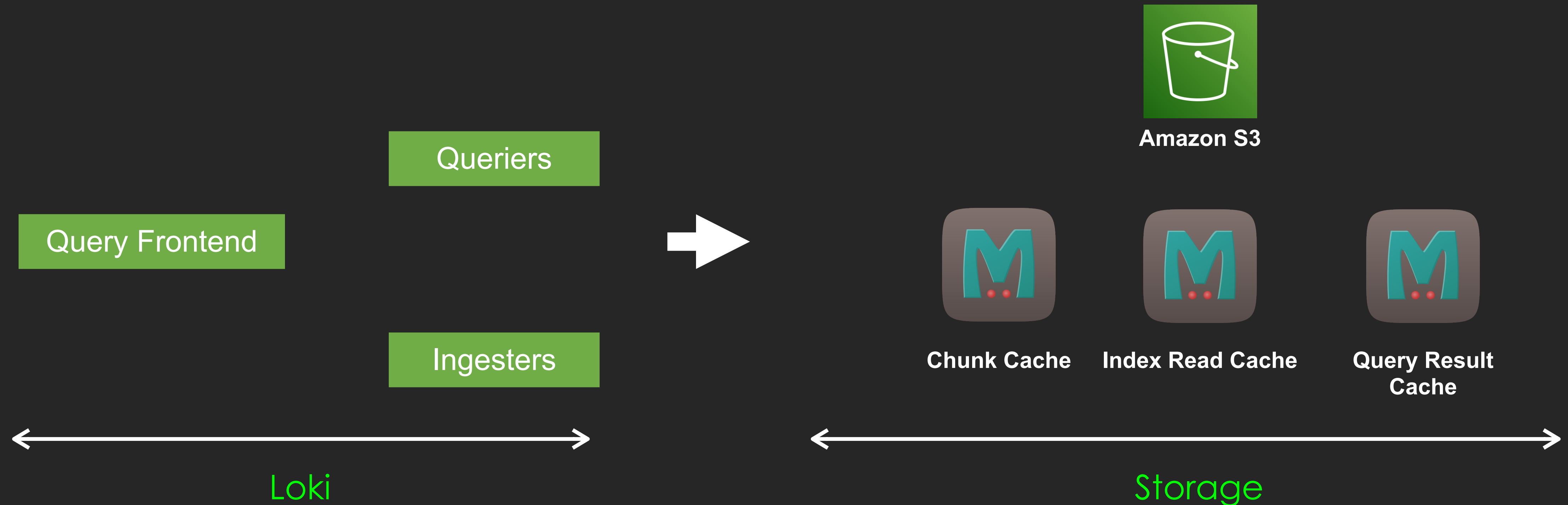
Ingester上のデータとEncode形式



2) ログの読み込みプロセス

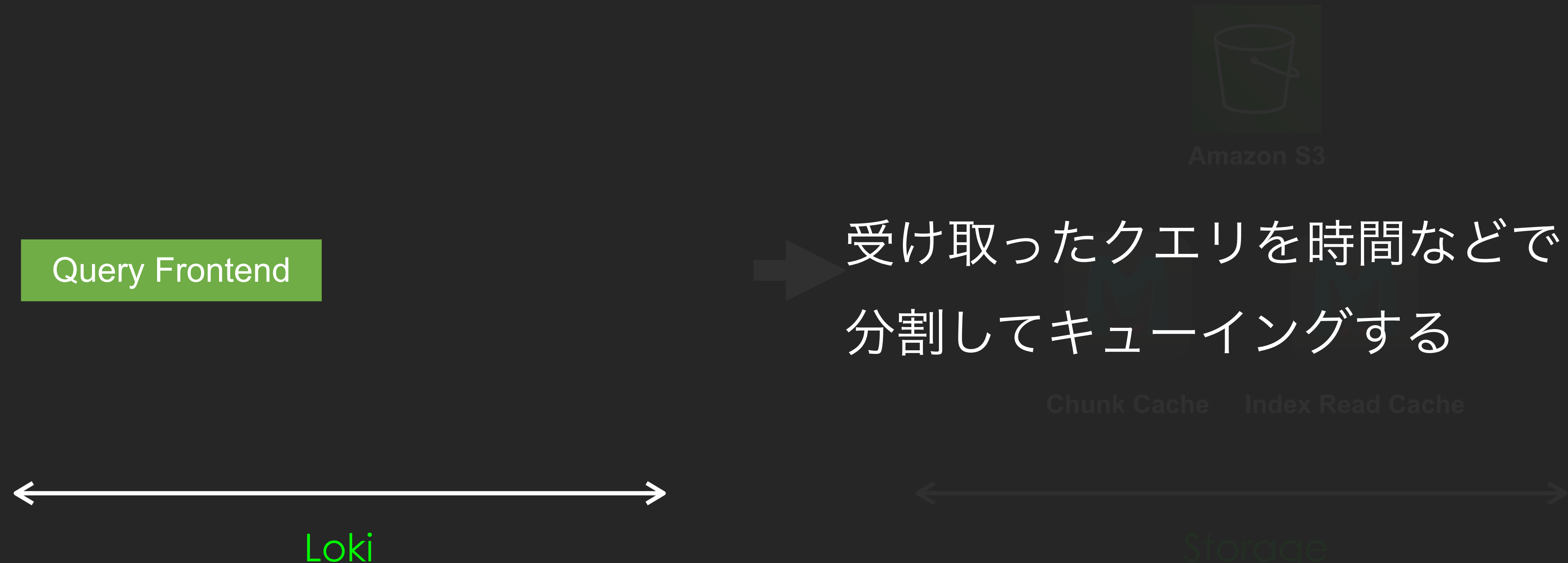
Overview

読み込みプロセスの登場人物

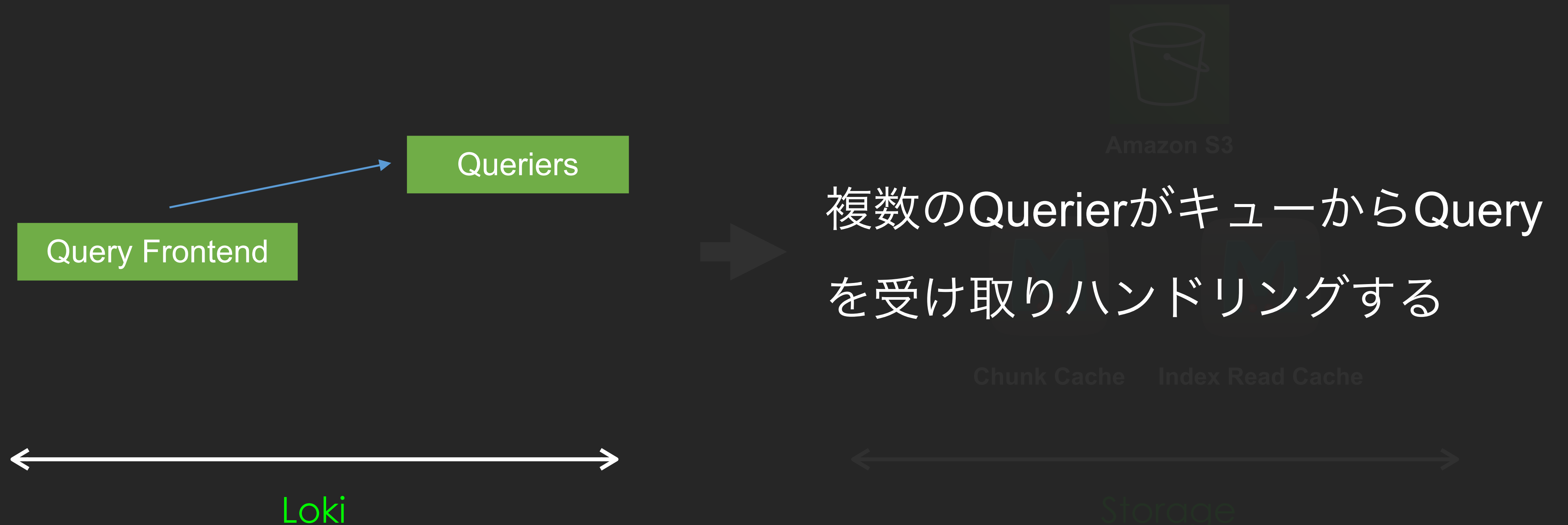


読み込みプロセスの登場人物

読み込みプロセスの登場人物

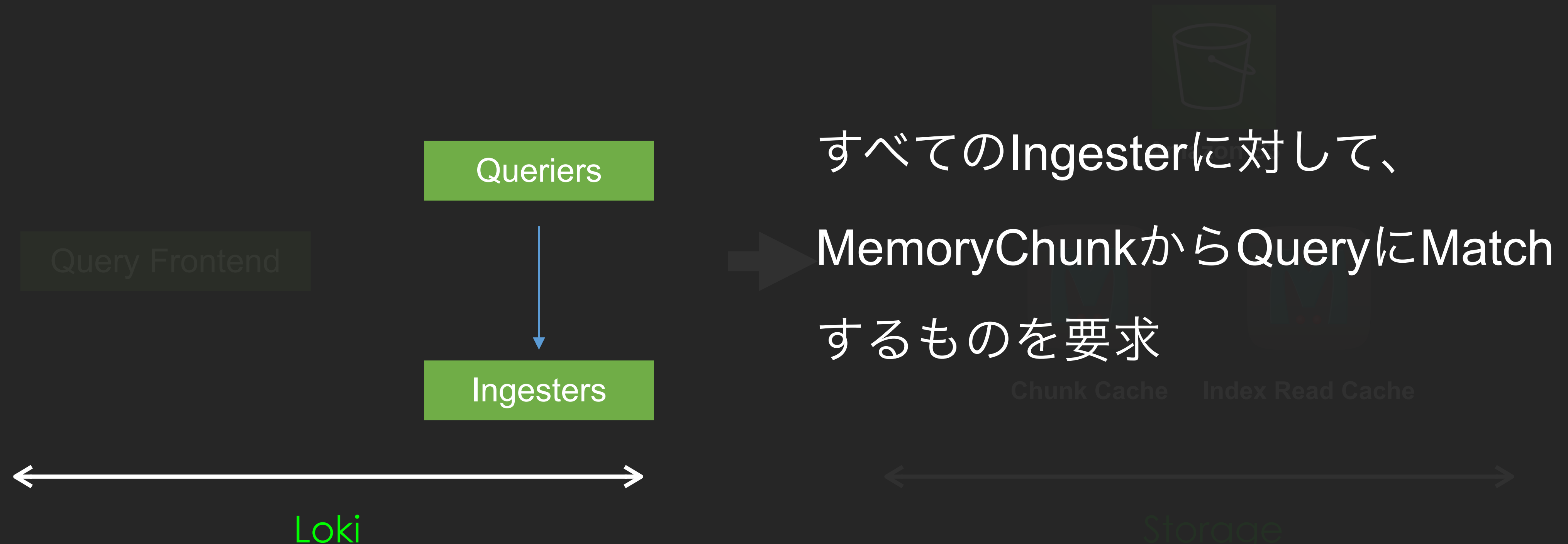


読み込みプロセスの登場人物



複数のQuerierがキューからQueryを受け取りハンドリングする

読み込みプロセスの登場人物



すべてのIngesterに対して、

MemoryChunkからQueryにMatch

するものを要求

読み込みプロセスの登場人物

Queryに対応する転置Indexを取得する

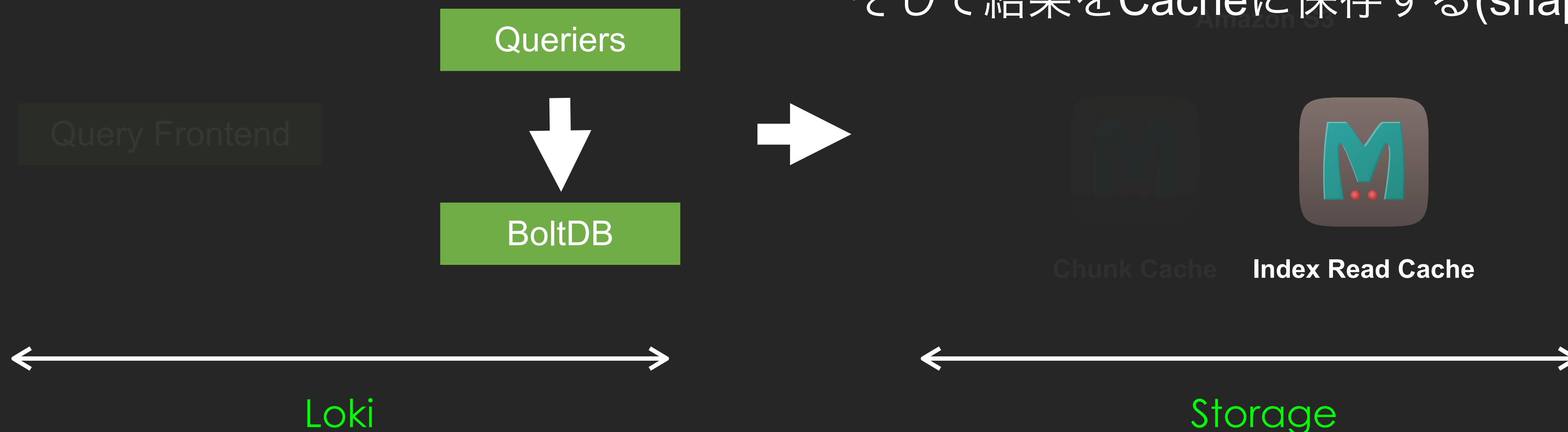
このときCacheに透過的にアクセス

読み込みプロセスの登場人物

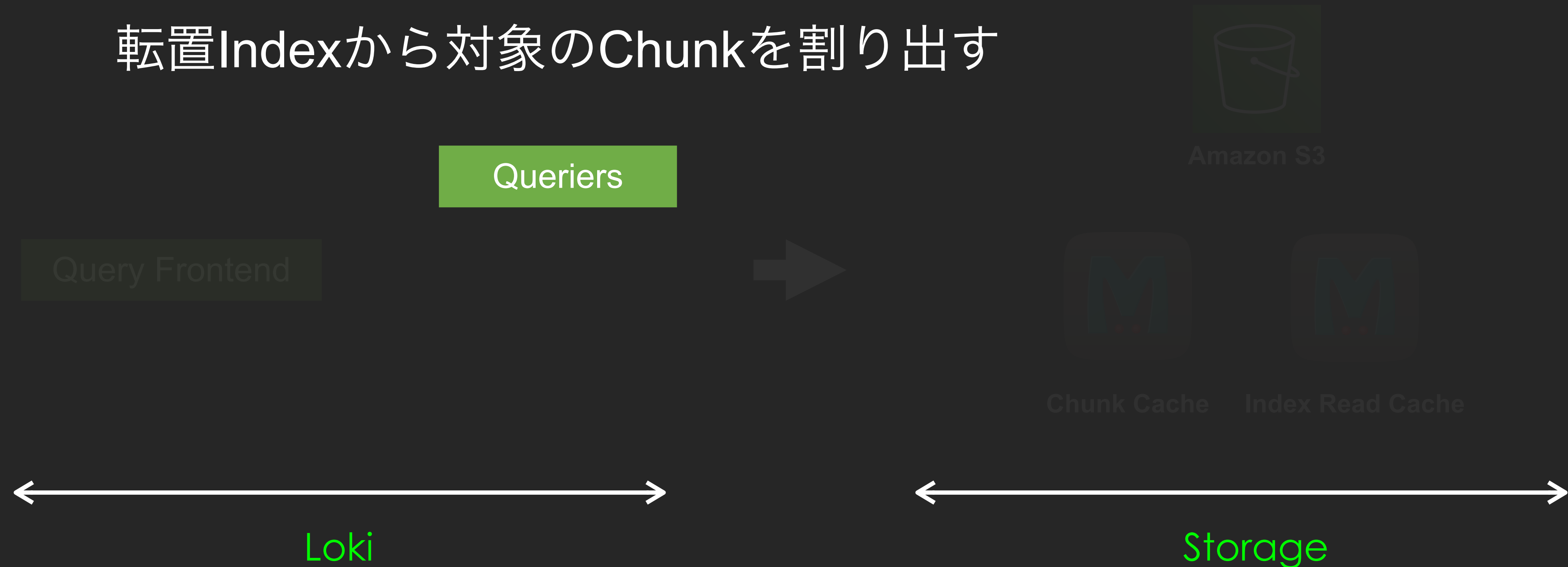
Cacheに存在しない場合は、

ローカルのBoltDBからMatchするIndexを取得

そして結果をCacheに保存する(snappy)

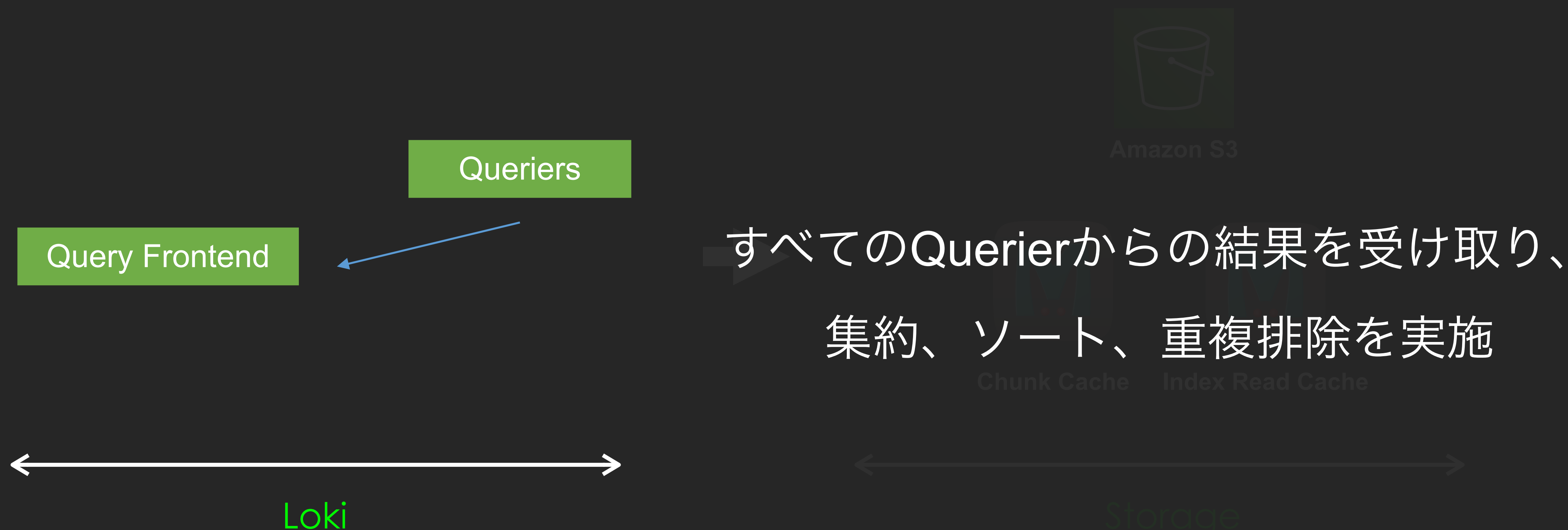


読み込みプロセスの登場人物



読み込みプロセスの登場人物

読み込みプロセスの登場人物



読み込みプロセスの登場人物

結果をQuery Result Cacheへ保存する

読み込みプロセスの登場人物

転置Indexから対象Chunkの選定

転置Indexの目的

S3からChunkを最小労力で取得すること

転置Indexから対象Chunkの絞り込み

1. LabelのKeyとValueの組み合わせからStreamのID (Series ID)を取得する
2. Series IDと時間範囲からChunkのKeyを取得する
3. ChunkのKeyからS3上、Memcached上のパスを割り出し、直接ChunkをDownload

SeriesID

ログ

{service=“keystone”, hostname=“host1”} 00:00:02 keystone log body

SeriesID

9ac2adda49e899b312a9abb895656b1ab26c9858fd500f2ae3983d5309b39363/

key, valueの組み合わせのsha256

Chunk Key

ログ

{service="keystone", hostname="host1"} 00:00:02 keystone log body

FingerPrint

a6965cd7

key, valueの組み合わせのHash値

Chunk Key

Tenant1/a6965cd7:Chunk開始時間:Chunk終了時間

転置Indexの構造イメージ

Label Key-ValueのHashからSeriesIDを引くIndex

Hash Value	Range Value	Value
TenantID + LabelName	Hash(Label Value) + SeriesID	Label Value

転置Indexの構造イメージ

Label Key-ValueのHashからSeriesIDを引くIndex

Hash Value	Range Value	Value
TenantID + LabelName	Hash(Label Value) + SeriesID	Label Value

Hash + Rangeでユニーク行を特定する

転置Indexの構造イメージ

Label Key-ValueのHashからSeriesIDを引くIndex

Hash Value	Range Value	Value
TenantID + LabelName	Hash(Label Value) + SeriesID	Label Value

Range Valueは範囲検索、ソートに利用できる

転置Indexの構造イメージ

テーブルイメージ

Hash Value (TenantID + Label Name)	Range Value (Hash(Label Value) + SeriesID)	Value (Label Value)
Tenant1:service	abc680ab:c79abadeff	keystone
Tenant1:host	cfe960ab:bcfe12ea	hostname1
Tenant1:type	can860ab:c79abadeff	api
Tenant1:service	cdc680ab:c79abadeff	nova
Tenant1:host	bee960ab:bcfe12ea	hostname21
Tenant1:type	abd860ab:c79abadeff	scheduler

転置Indexの構造イメージ

このIndexは

LabelのKeyとValueのパターン分だけ作られる

カーディナリティの高いラベルは

このIndexが大量に作られることになる

転置Indexの構造イメージ

SeriesIDからChunk Keyを引くIndex

Hash Value	Range Value	Value
TenantID + SeriesID	Chunkの開始時間 + Chunk Key	nil

IndexからのChunk Key割り出し

LogQL

{service=“keystone”, hostname=“host1”} |= “level=ERROR”

ラベルマッチ部　　フィルタ部

IndexからのChunk Key割り出し

LogQL

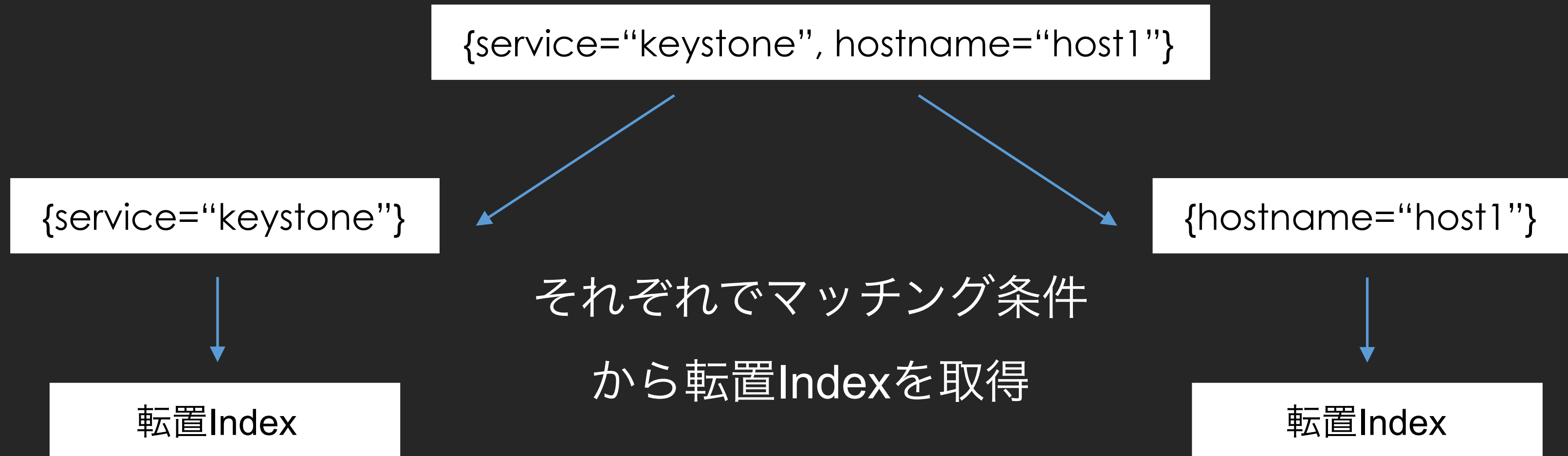
```
{service="keystone", hostname="host1"} |= "level=ERROR"
```

ラベルマッチ部　フィルタ部

Indexを使うのはこの部分の評価

IndexからのChunk Key割り出し

IndexからのChunk Key割り出し



IndexからのChunk Key割り出し

{service=“keystone”}

Hash Value (TenantID + Label Name)	Range Value (Hash(Label Value) + SeriesID)	Value
Tenant1:service	abc680ab:c79abadeff	keystone
Tenant1:host	cfe960ab:bcfe12ea	hostname1
Tenant1:type	c[illegible]860ab:c79abadeff	api
[illegible]	[illegible]b:c79abadeff	nova
Tenant1:host	bee960ab:bcfe12ea	hostname21
Tenant1:type	abd860ab:c79abadeff	scheduler

ヒットするのはこのレコード

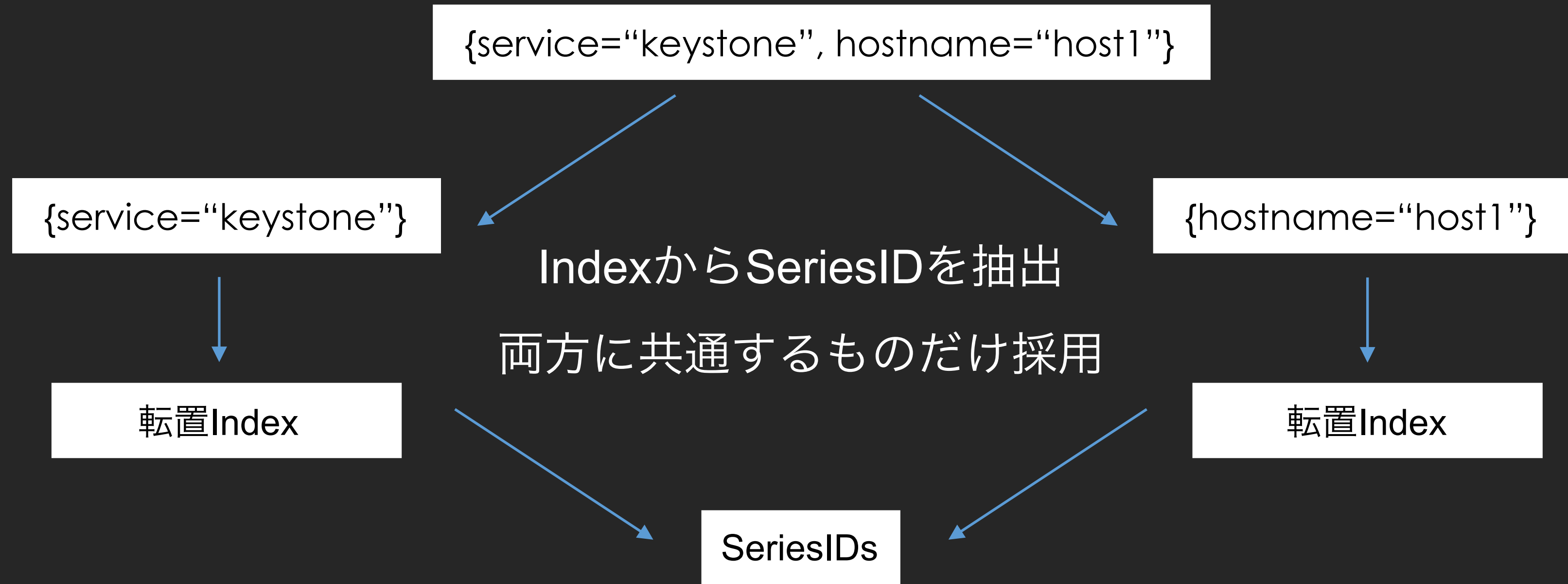
IndexからのChunk Key割り出し

{service="keystone"}

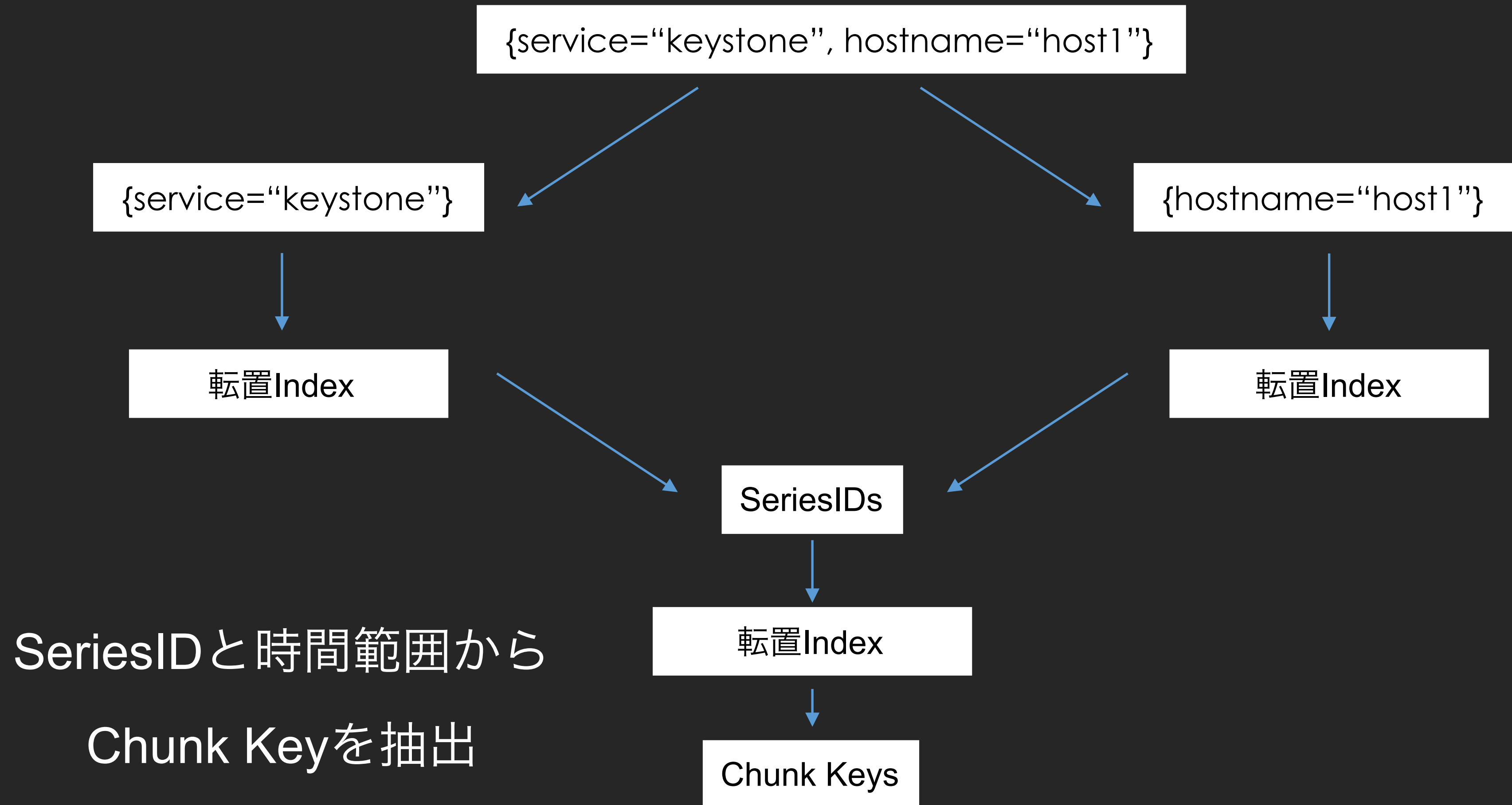
Hash Value (TenantID + Label Name)	Range Value (Hash(Label Value) + SeriesID)	Value
Tenant1:service	abc680ab:c79abadeff	keystone
Tenant1:host	cfe960ab:[illegible]cfe12ea	hostname1
Tenant1:type	can860[illegible]:c79abadeff	api
Te[illegible]	[illegible]b:c79abadeff	nova
Tenant1:host	bee960ab:bcfe12ea	hostname21
Tenant1:type	abd860ab:c79abadeff	scheduler

対象SeriesID

IndexからのChunk Key割り出し



IndexからのChunk Key割り出し



IndexからのChunk Key割り出し

SeriesID = c79abadeff, 範囲=2021/10/26 21:52:00 + 5min

Hash Value (TenantID + SeriesID)	Range Value (Chunk開始時間 + Chunk Key)	Value
Tenant1:c79abadeff	1635252768:chunk1	nil
Tenant1:c79abadeff	1635256368:chunk2	nil
Tenant1:c79abadeff	1635260768:chunk3	nil

IndexからのChunk Key割り出し

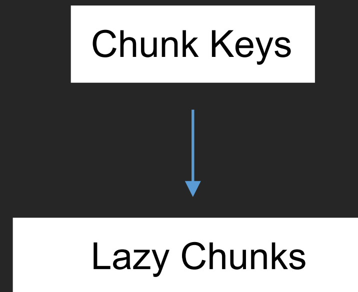
SeriesID = c79abadeff, 範囲=2021/10/26 21:52:00 + 5min

Hash Value (TenantID + SeriesID)	Range Value (Chunk開始時間 + Chunk Key)	Value
Tenant1:c79abadeff	1635252768:chunk1	nil
Tenant1:c79abadeff	163525636[illegible]8:chunk2	nil
Tenant1:c79abadeff	16352[illegible]768:chunk3	nil

対象ChunkのRecord

レスポンスの生成と返却

レスポンスの生成

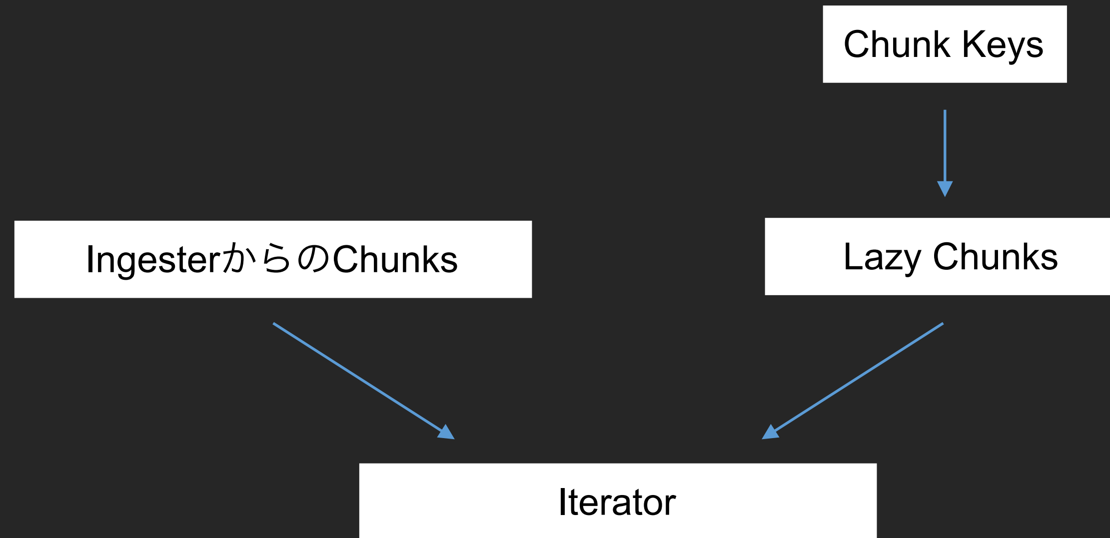


初期状態では実体を持たず、

読み込み命令がされたタイミングでストレージ(キャッシュ)

にChunkを取りに行くLazy Chunkを生成

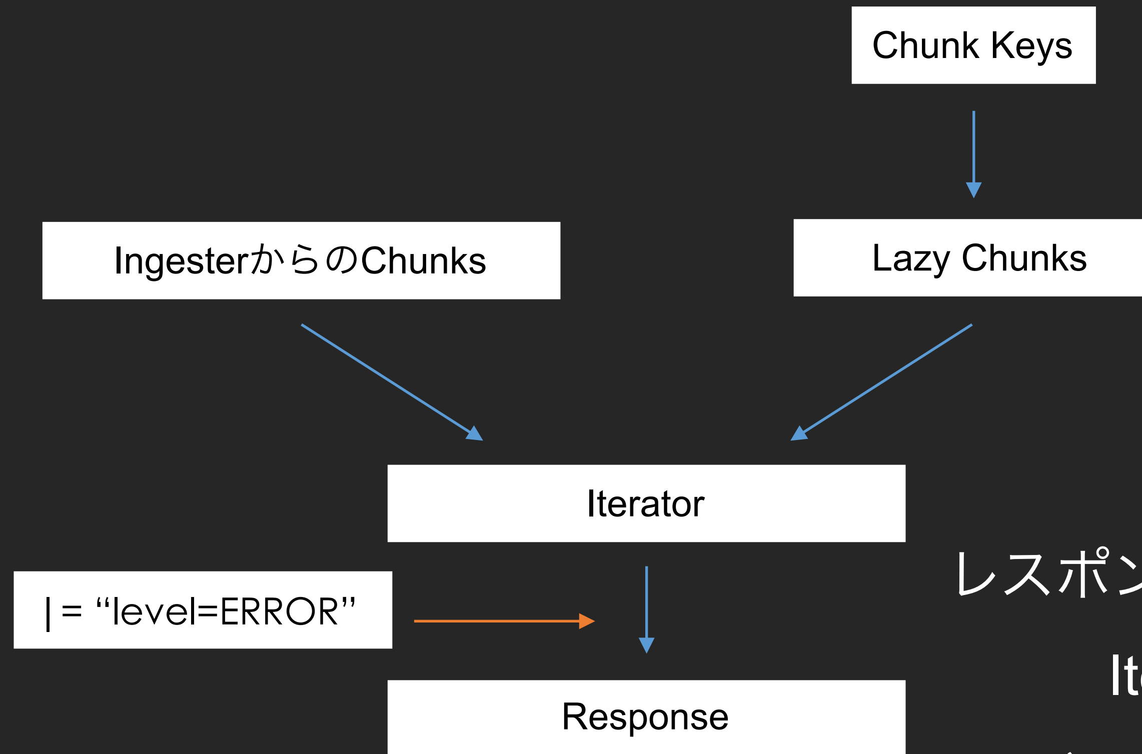
レスポンスの生成



IngesterからのChunkをあわせて、

Iteratorを生成

レスポンスの生成

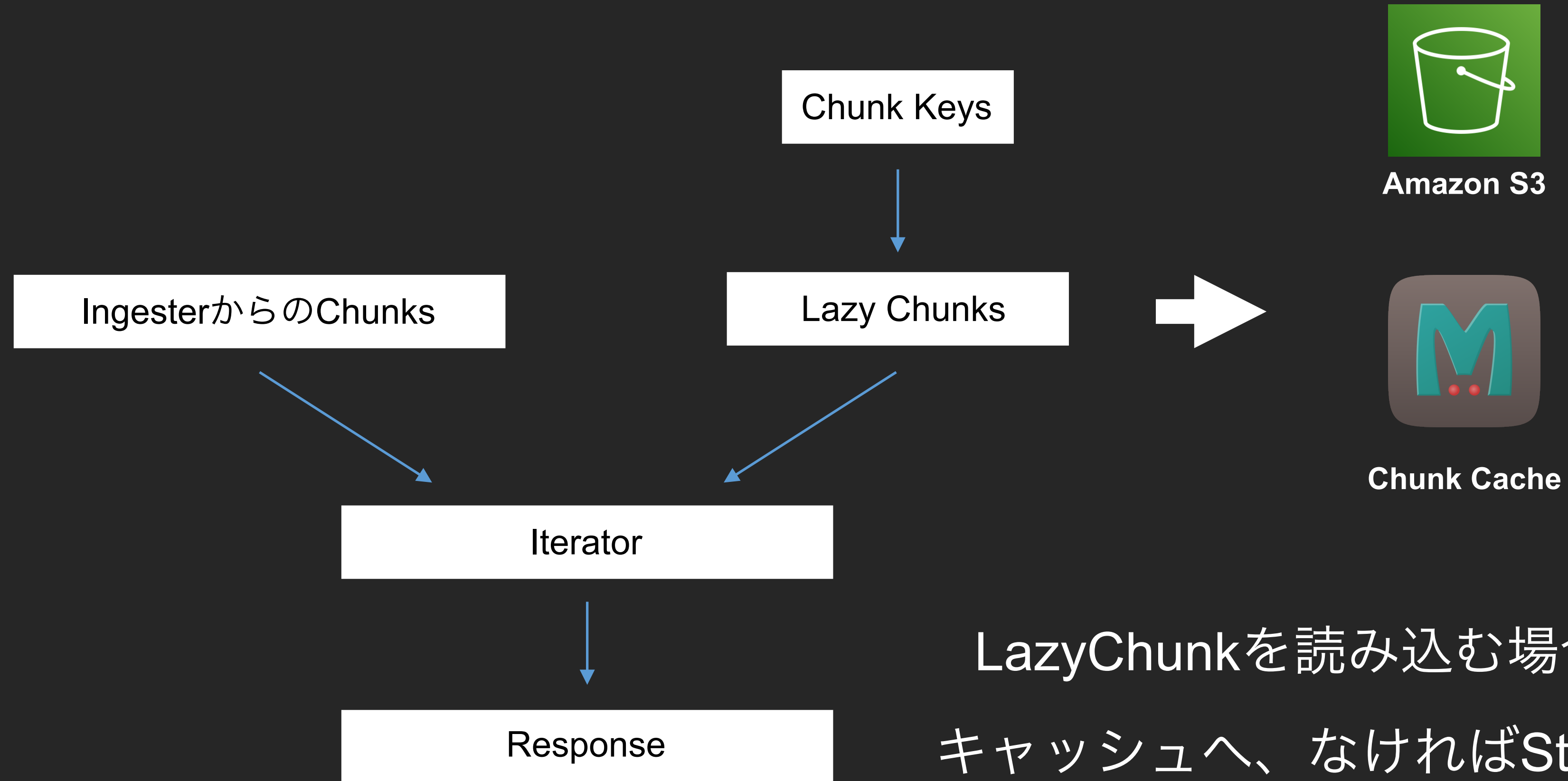


レスポンスが規定件数に達するまで、

Iteratorを読み込んでいく

ログのFilter条件はここで評価される

レスポンスの生成

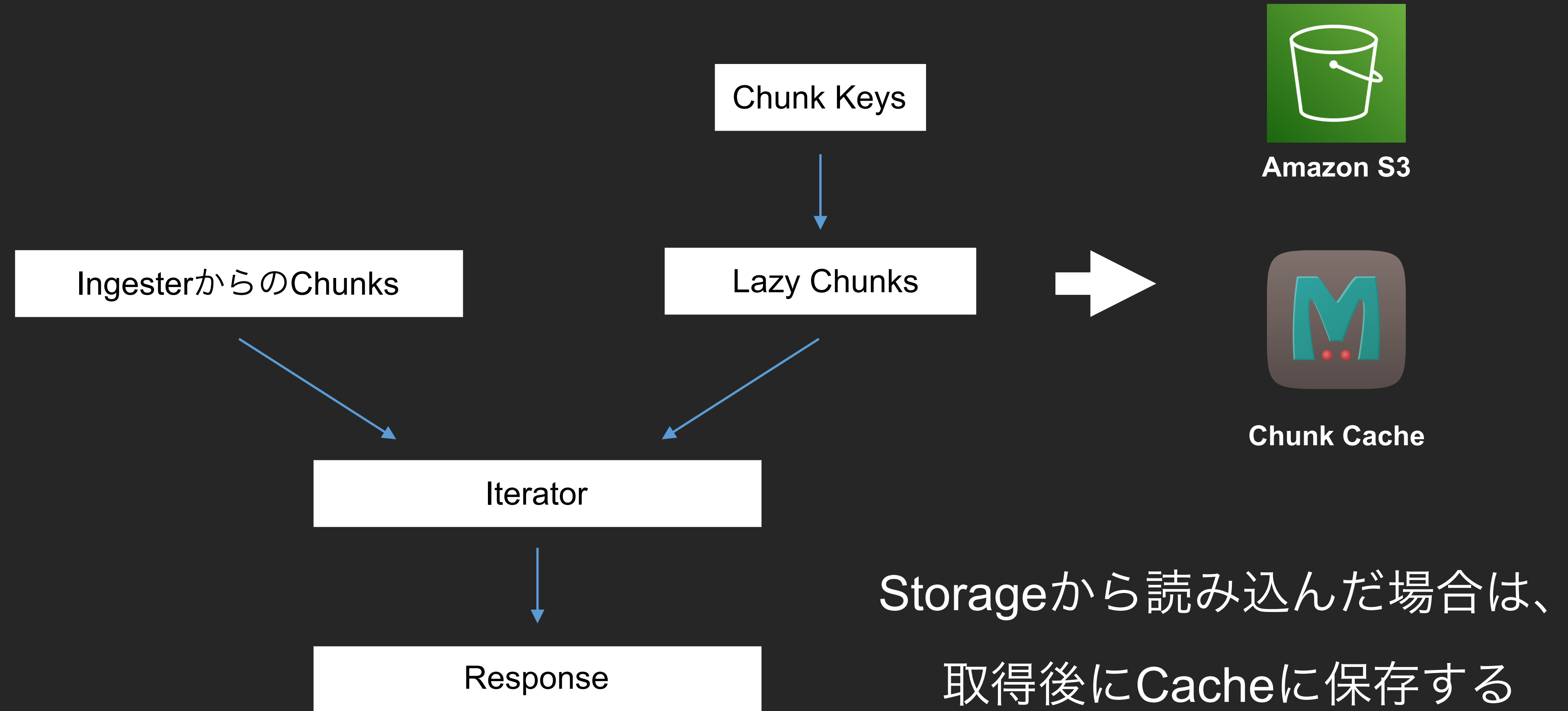


LazyChunkを読み込む場合は、

キャッシュへ、なければStorageへ

Chunkを問い合わせる

レスポンスの生成



Storageから読み込んだ場合は、

取得後にCacheに保存する

Query Sharding

Queryの分割戦略

1. 時間ごとに分割する
    - 1時間分のログを検索する場合、15分で分割する設定なら4つのクエリに分解されて実行される

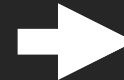
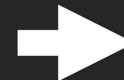
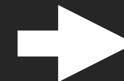
2. 転置IndexをShardingする
    - あらかじめ転置IndexにShard番号をいれておき、QueryFrontendがQueryを分割し、それぞれにShard番号を挿入してQuerierにわたす

転置IndexへのShard番号埋め込み

Shard Number = SeriesID % shard count

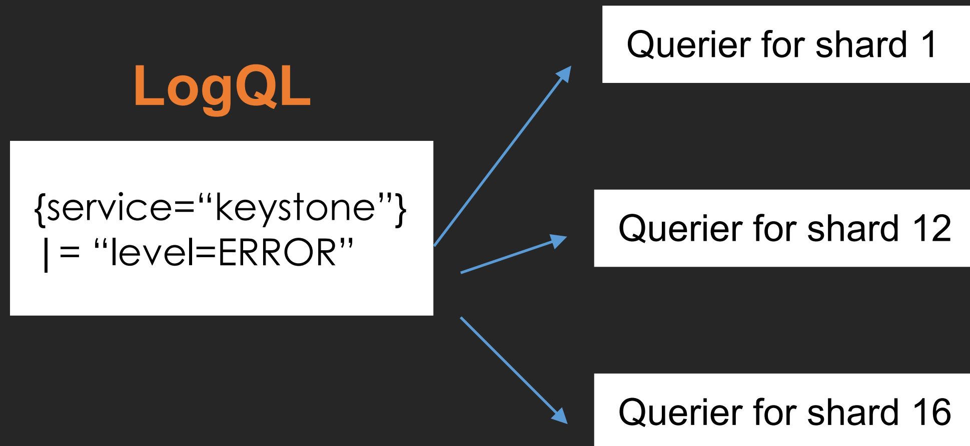
Hash Value	Range Value	Value
TenantID + LabelName	**Shard Number** + Hash(Label Value) + SeriesID	Label Value

転置IndexへのShard番号埋め込み

Stream		Shard Number
{service="keystone", hostname="host1"}		1
{service="keystone"}		12
{service="keystone",hostname="host2"}		16

Shard番号によるQuery分割

QueryをShardで分割



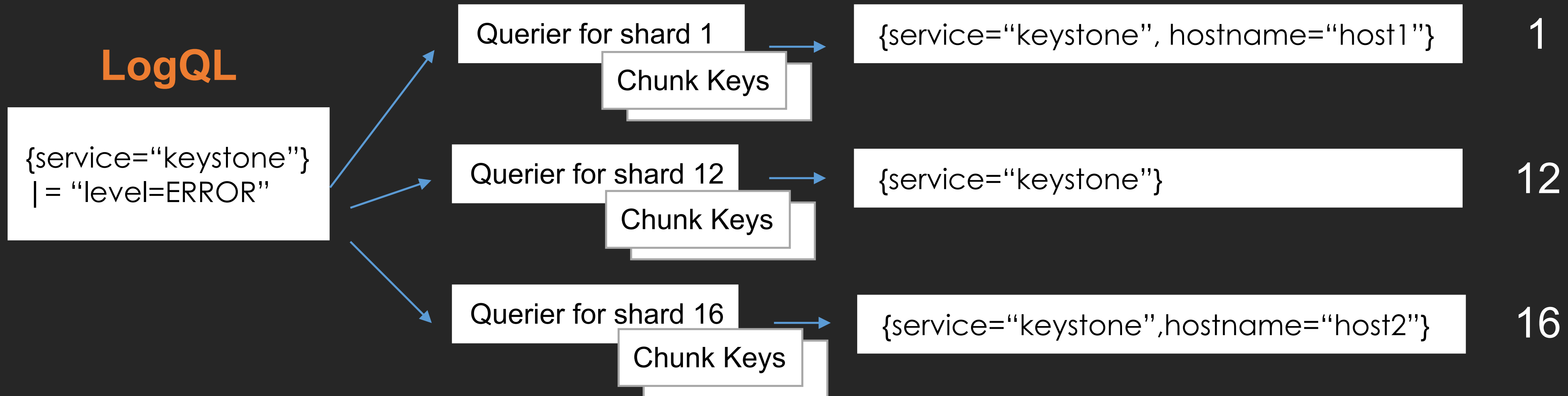
Querier for shard 1

Querier for shard 12

Querier for shard 16

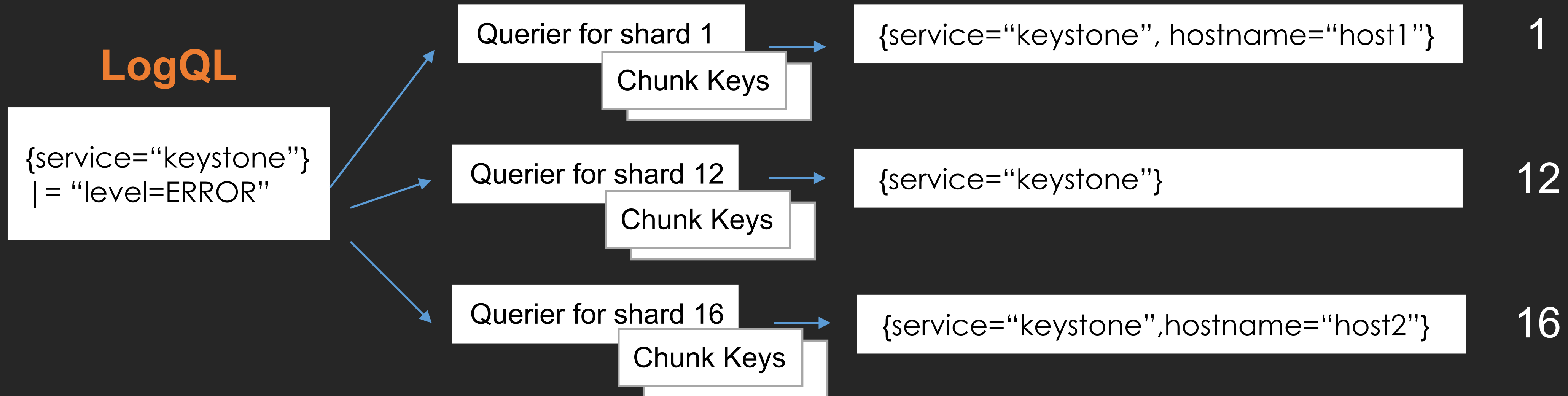
Shard番号によるQuery分割

取れるChunkがshardで分割される



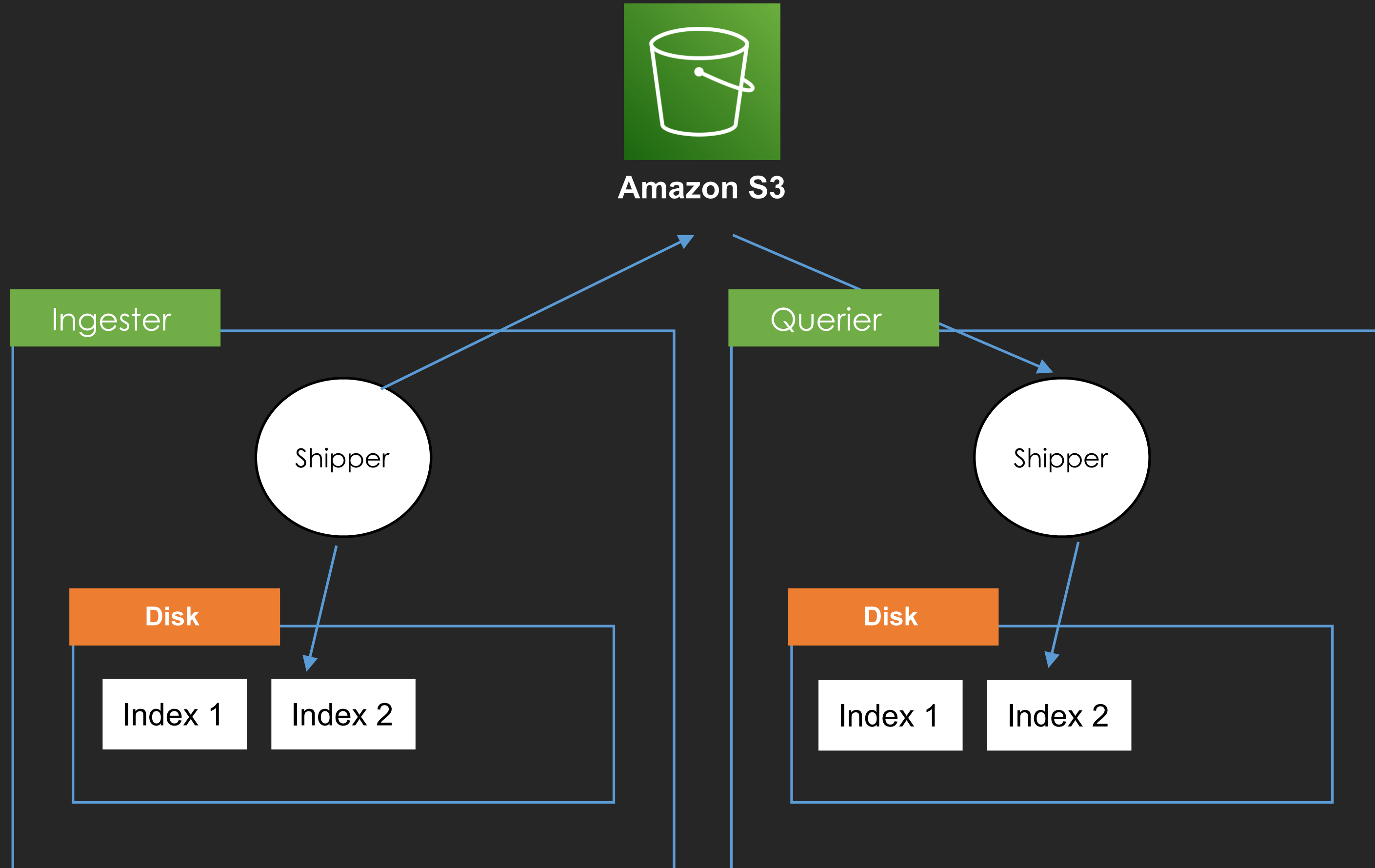
Shard番号によるQuery分割

|= “level=ERROR”のfilter処理を分割処理できる

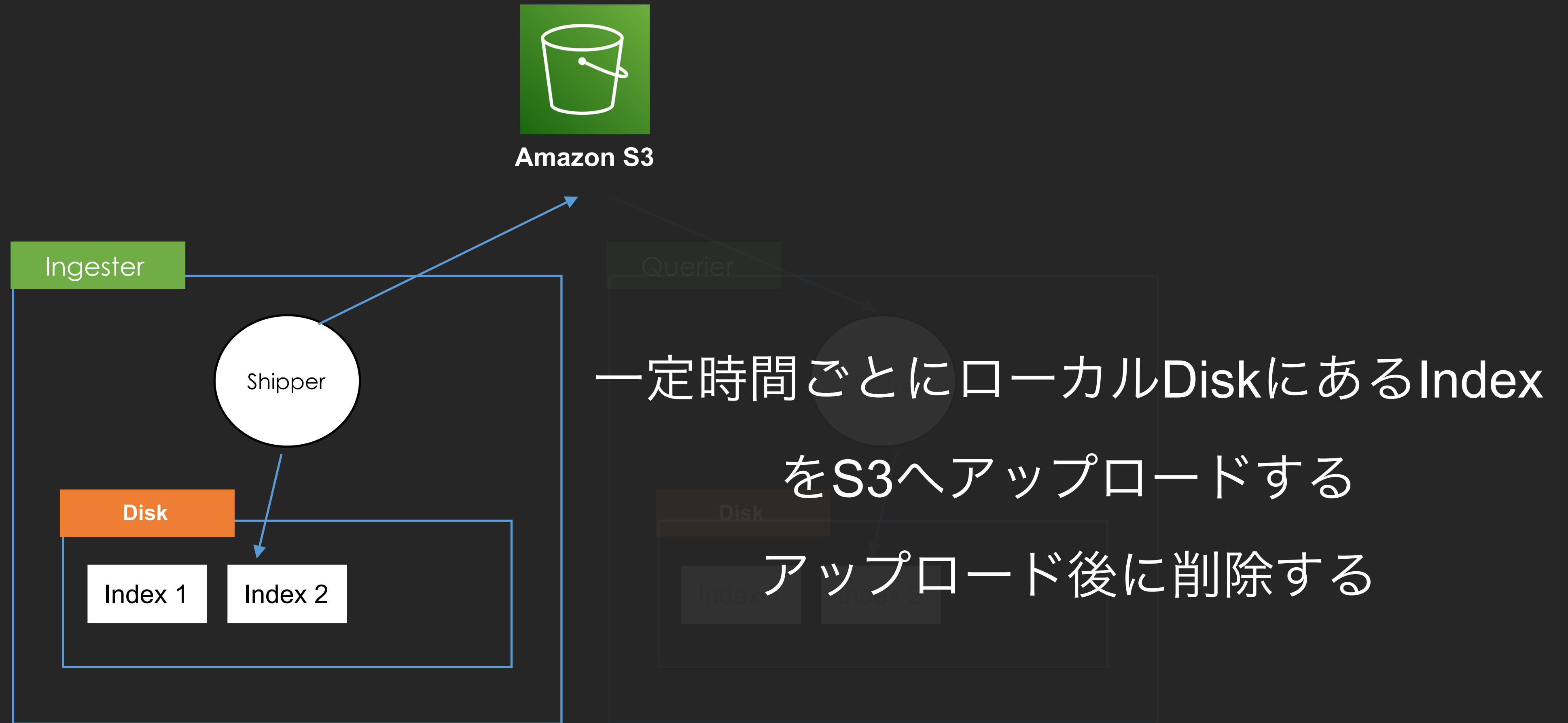


BoltDB ShipperによるIndex管理

BoltDB Shipper



BoltDB Shipper - Ingester side



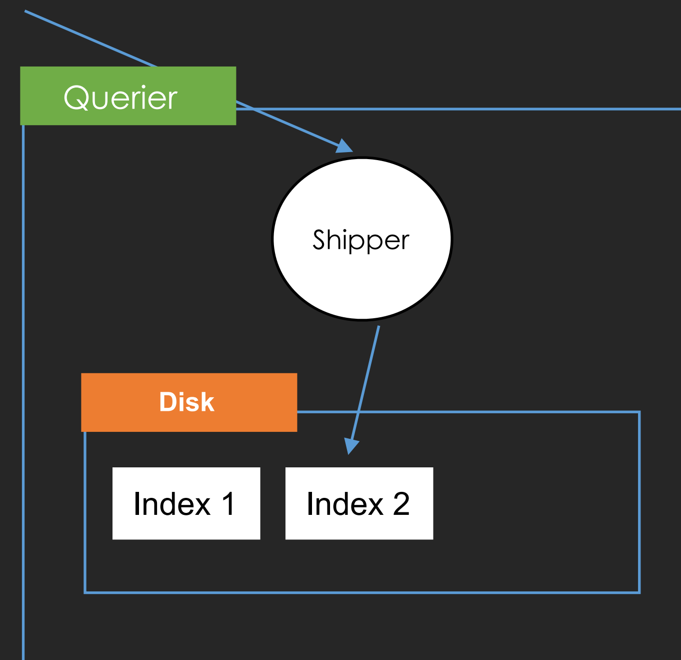
一定時間ごとにローカルDiskにあるIndex

をS3へアップロードする

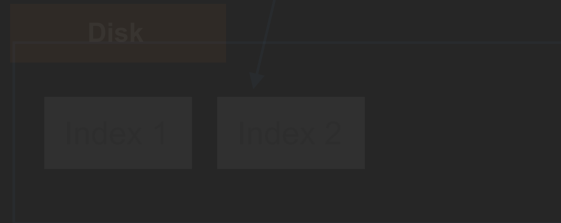
アップロード後に削除する

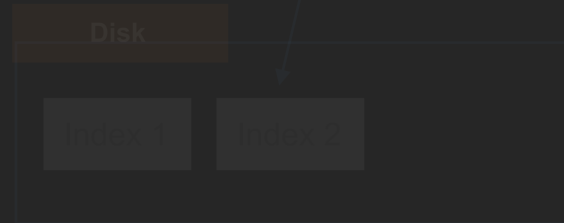
BoltDB Shipper - Querier side

- 起動時にS3にあるIndexをDownload
- Query時に足りないIndexはS3から都度ダウンロード
- 一定時間ごとに最終使用からCacheTTL経過したIndexを削除



BoltDB Shipper





Ingester, Querierはローカルでindexを扱い、

Shipperが非同期でIndexをStorageと同期する

各コンポーネントの役割まとめ

Name	役割	タイプ	データの持ち化	クラスタリング有無
Distributor	バリデーションとIngesterへのルーティング	Stateless		有
Ingester	データのバッファリングとFlush	Stateful	Memory: Chunks(raw + 圧縮) Disk: WAL(raw + 圧縮) 転置Index(圧縮)	有
Query Frontend	クエリの分割、キュー制御	Stateless		無
Querier	クエリの実行	Stateful	Disk: 転置IndexのCache(圧縮)	無
Chunk Cache	Chunkのキャッシュ	Stateful	Memory: Chunks(圧縮)	有(クライアントサイド)
Index Read Cache	IndexのRead用キャッシュ	Stateful	Memory: 転置Index(Snappy)	有(クライアントサイド)
Index Write Cache	同じIndexの書き込みが複数発生しないようにするための制御用キャッシュ (BoltDB Shipperでは不要)	Stateful	Memory: Chunk Key(raw)	有(クライアントサイド)
Query Result Cache	クエリの結果のキャッシュ	Stateful	Memory: Query Result(raw)	有(クライアントサイド)

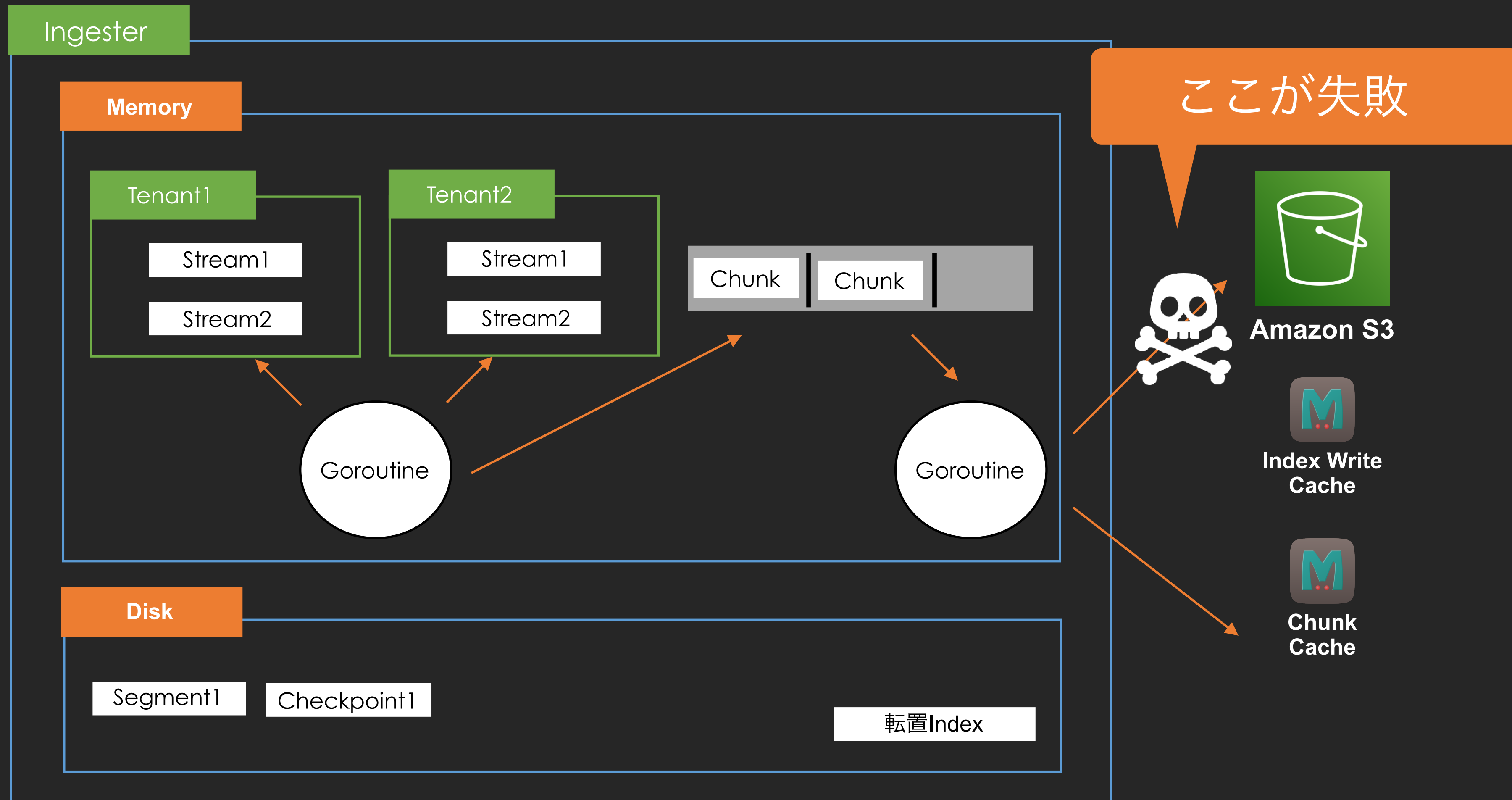
3) 障害時の挙動を知る

書き込み時の耐障害設計

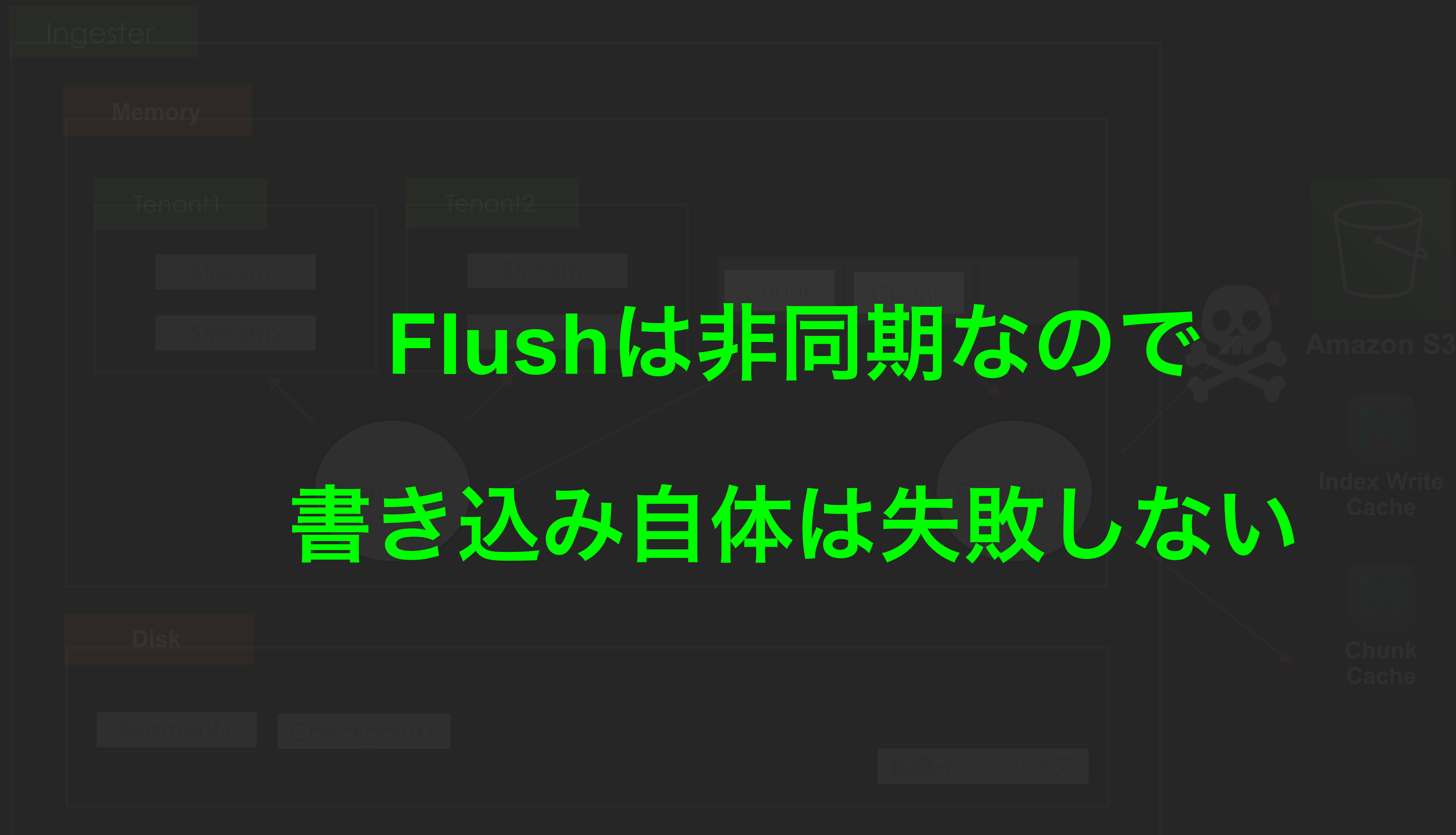
書き込み時の耐障害設計

S3障害時に備える

書き込み時の耐障害設計



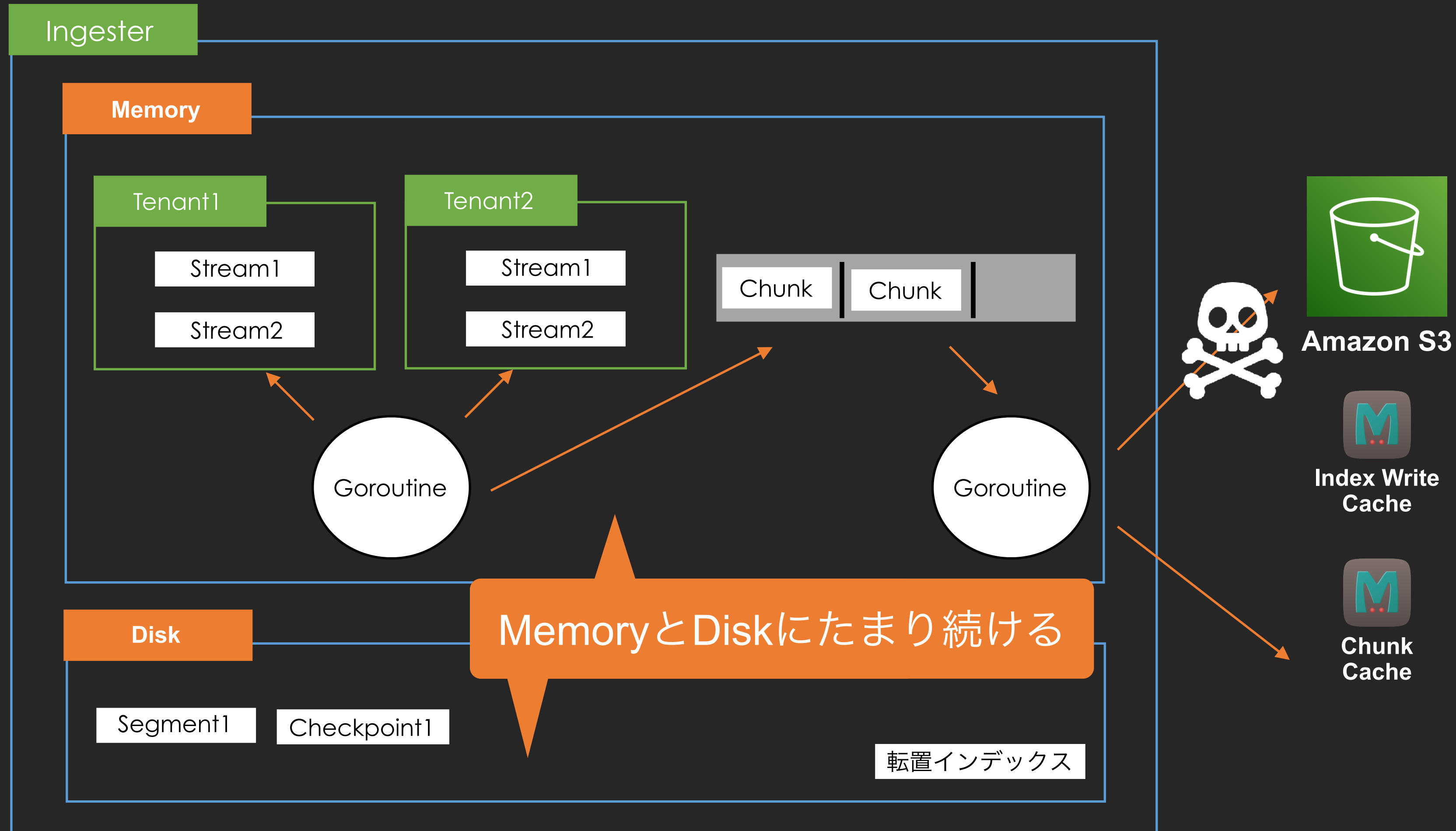
書き込み時の耐障害設計



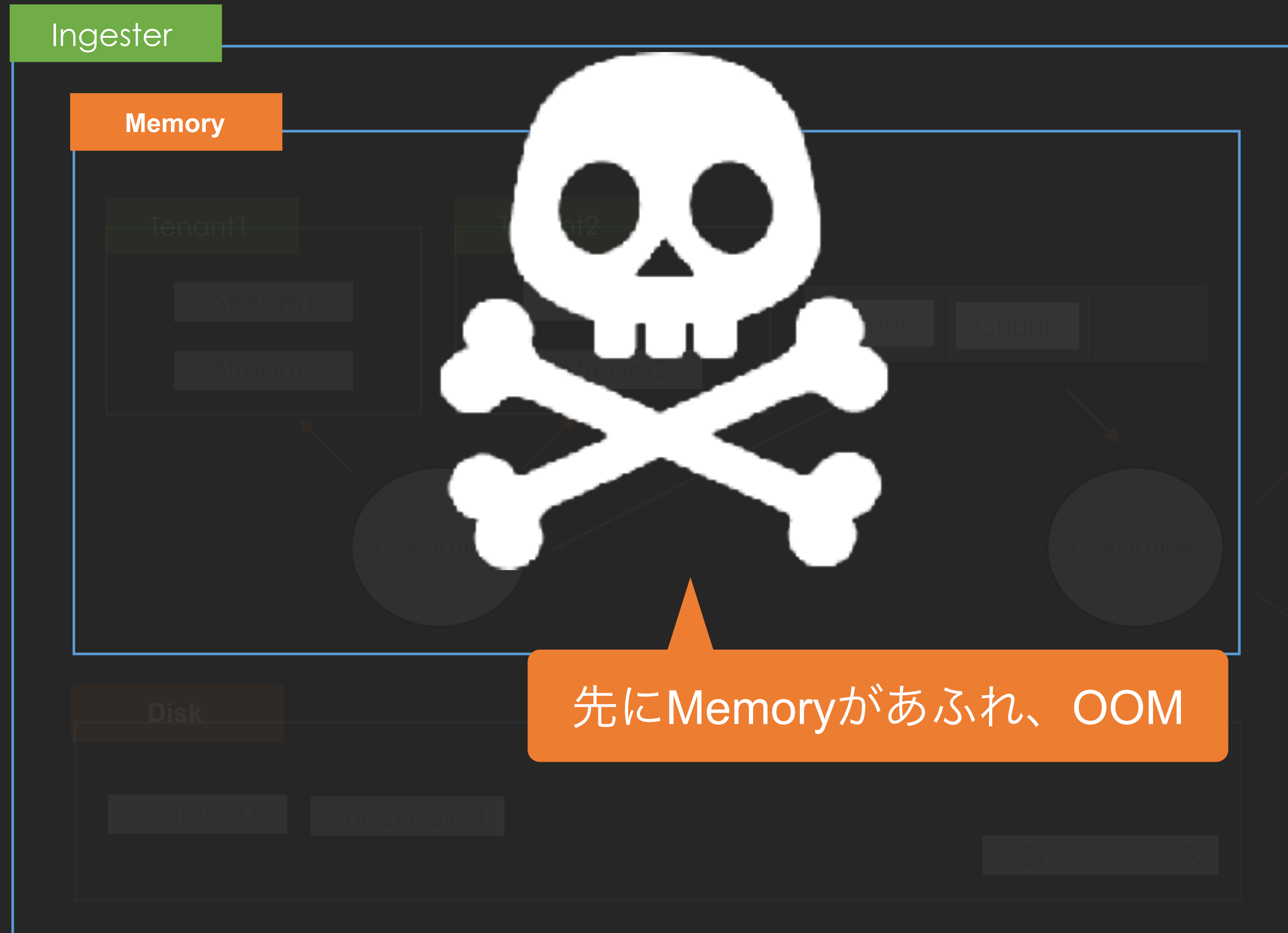
Flushは非同期なので

書き込み自体は失敗しない

書き込み時の耐障害設計



書き込み時の耐障害設計



WALの仕組み(再掲)

Ingester

Memory

Ingesterの起動時には

Diskから

SegmentとCheckpointを読み取る

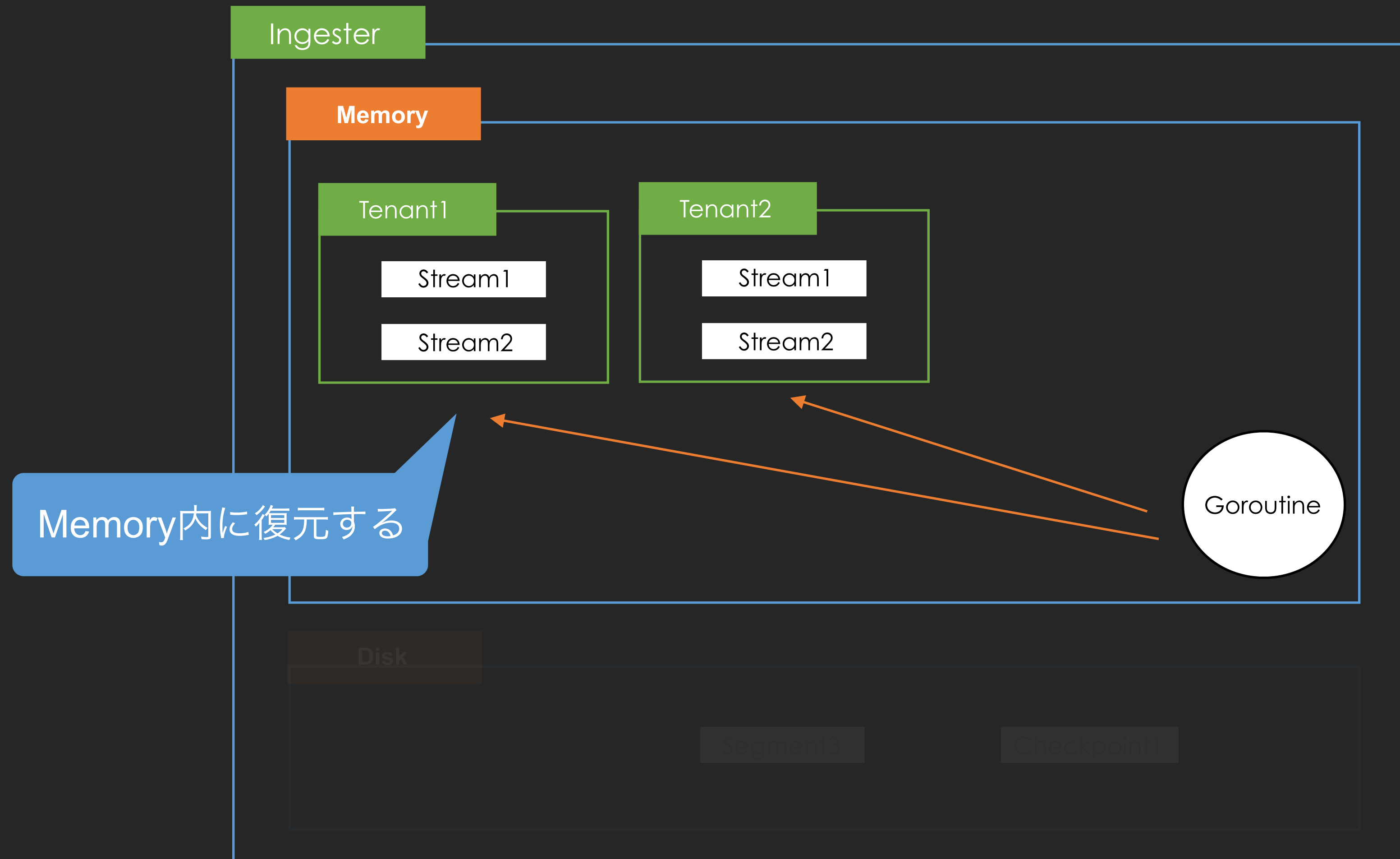
Goroutine

Disk

Segment3

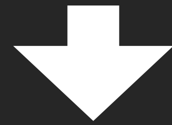
Checkpoint1

WALの仕組み(再掲)



書き込み時の耐障害設計

WALは実質MemoryのSnapshot



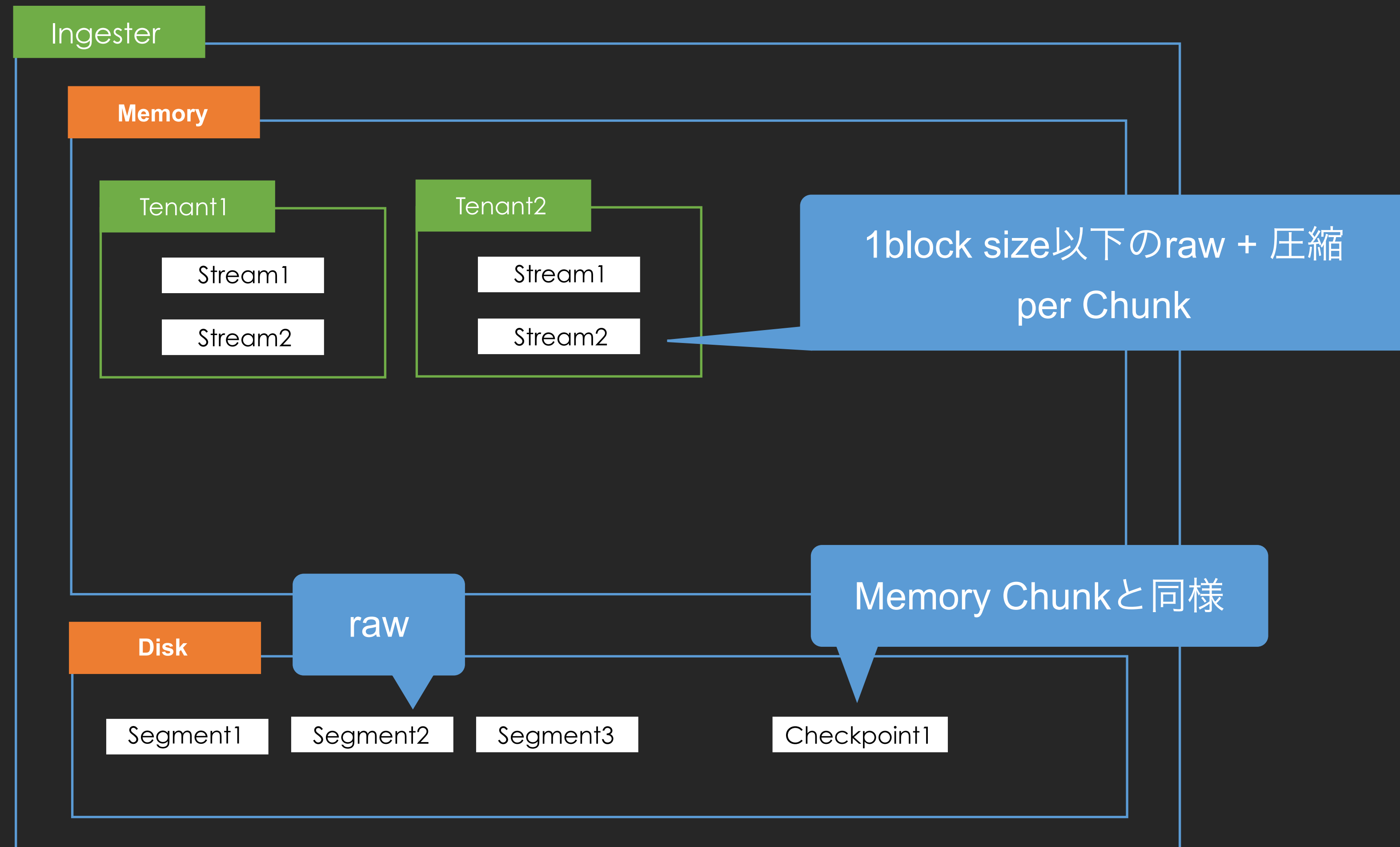
ロード成功しても近いうちにOOM

ロード失敗したらそもそも起動しない

書き込み時の耐障害設計

ボトルネックはIngesterのMemory

Ingester上のデータとEncode形式(再掲)



書き込み時の耐障害設計

対策1. 耐えたい時間分のMemoryを積む

- 時間あたりのログ量 / 圧縮比 * 時間 * replication factor

※圧縮比は弊環境ではgzip圧縮で10~18倍の圧縮比

書き込み時の耐障害設計

1日の流量10TBの環境で1時間耐える(弊社の1リージョン)

1000 / 24 = 41.6 GB / hour

- Replication Factor = 1
- Ingester * 11台
    - Memory: 4GiB
    - Disk: 8GiB(マージンを取ってMemoryの2倍)
- Chunk Cache * 14台
    - Memory: 3GiB

書き込み時の耐障害設計

Replication Factorは1でいいのか？

- FlushされるまでにIngesterプロセスがダウンするとその間だけそこにあったログは欠損する

ある程度割り切りをする

- WALがあるので再起動後にすぐに復旧できる
- 細かい欠損より障害時に稼働継続できる可能性を高める方向に振る

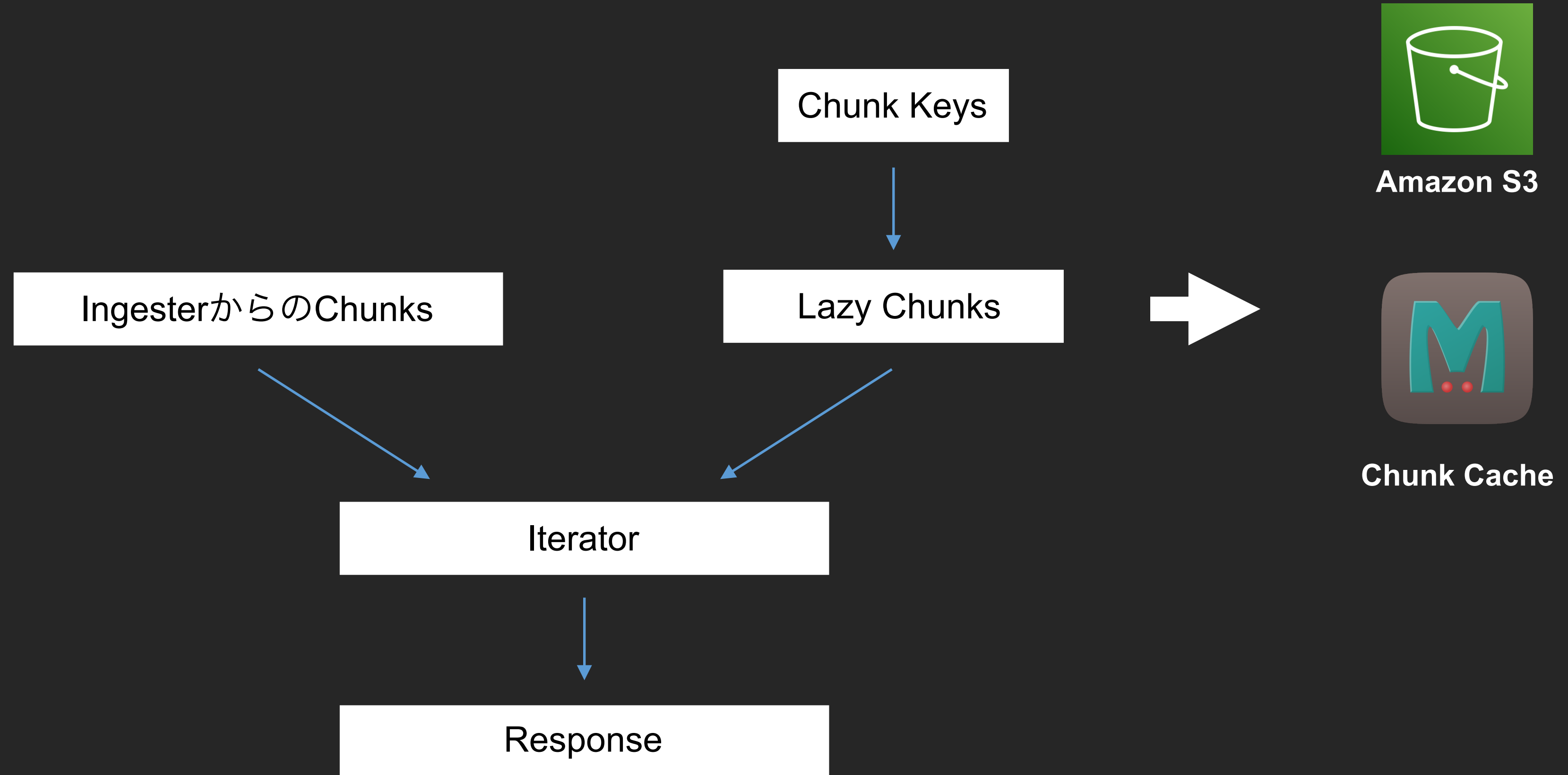
書き込み時の耐障害設計

対策2. 障害時は一時的にWALを無効にして起動する

- WALロードが走らないので、Memoryからあふれる量溜まっていてもプロセス起動できるようになる
- 再有効にするときにログが揮発しないよう、replication factor、update strategyに配慮する

読み取り時の耐障害設計

読み取り時の耐障害設計



読み取り時の耐障害設計

Storage障害時にもログを検索するための条件

- Ingesterが最低一つは健在であること
- 検索結果をCacheかIngesterのデータでカバーできること
- Cacheにない時間範囲をクエリに指定しないこと

まとめ

まとめ

Lokiのコアである書き込みと読み込みプロセスを詳解

- 動作原理がわかったことで、トラブルシューティングやチューニングが可能に
- どこでどのエンコーディングでデータを持つかを把握することでキャパシティプランニングが可能に
- 障害時の挙動を把握することで適切な準備が検討可能に

まとめ

伝えられなかったこと

- ログからのメトリクス生成やアラーティング
- ログのリテンション管理について
- Loki自体のモニタリングについて
- 各コンポーネントのキャパシティ管理について
- Out of order entry問題への対策について

別途本を書きます

ご質問等は

Twitter: @taisho6339

THANK YOU