2024/3/1 OTFSG Tokyo Meetup #2

Kafka Connect Iceberg Sink Connectorを使ってみる

株式会社マイクロアド
永富 安和 (𝕏 @yassan168)

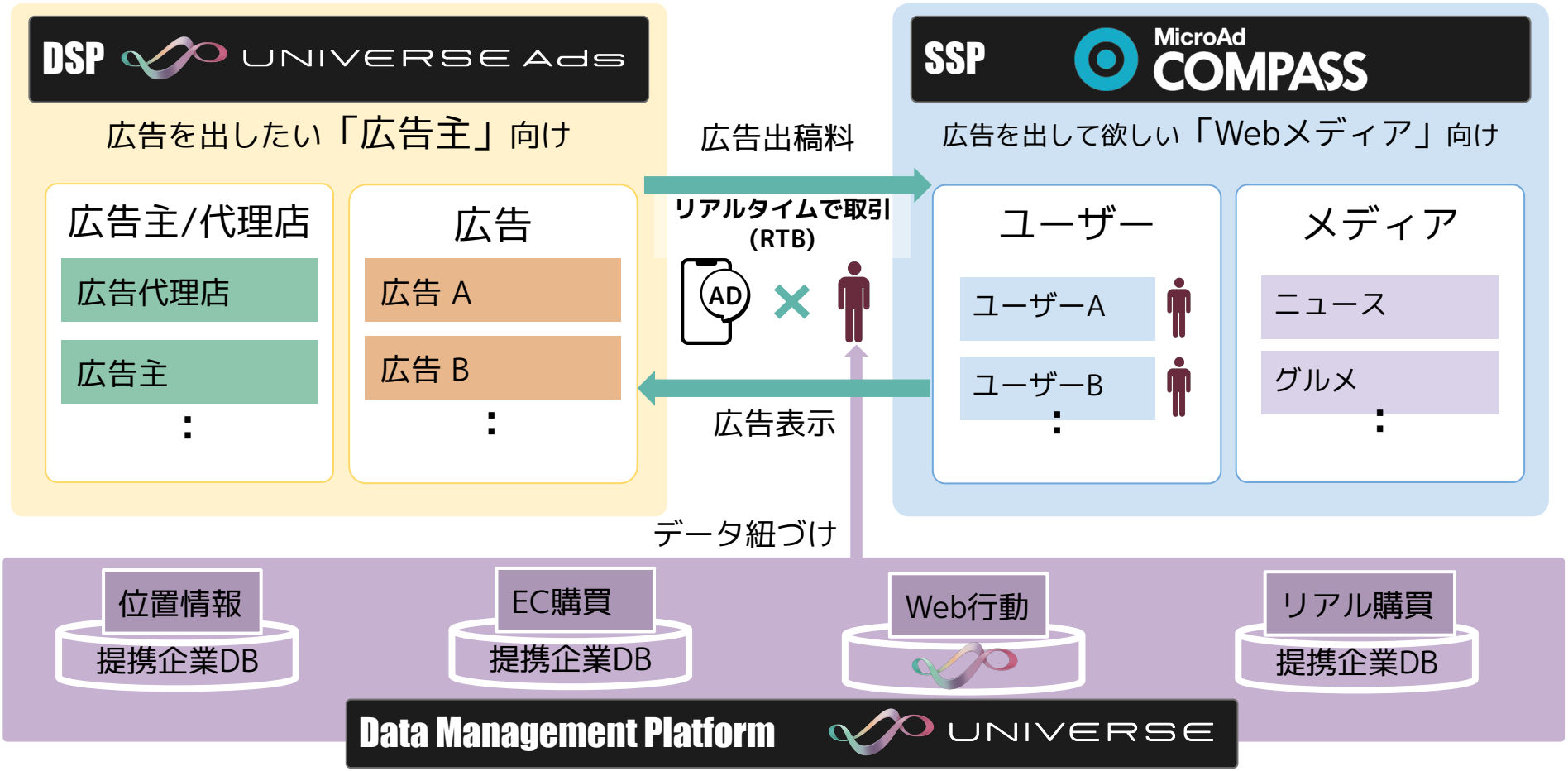
#otfsg_tokyo

おしながき

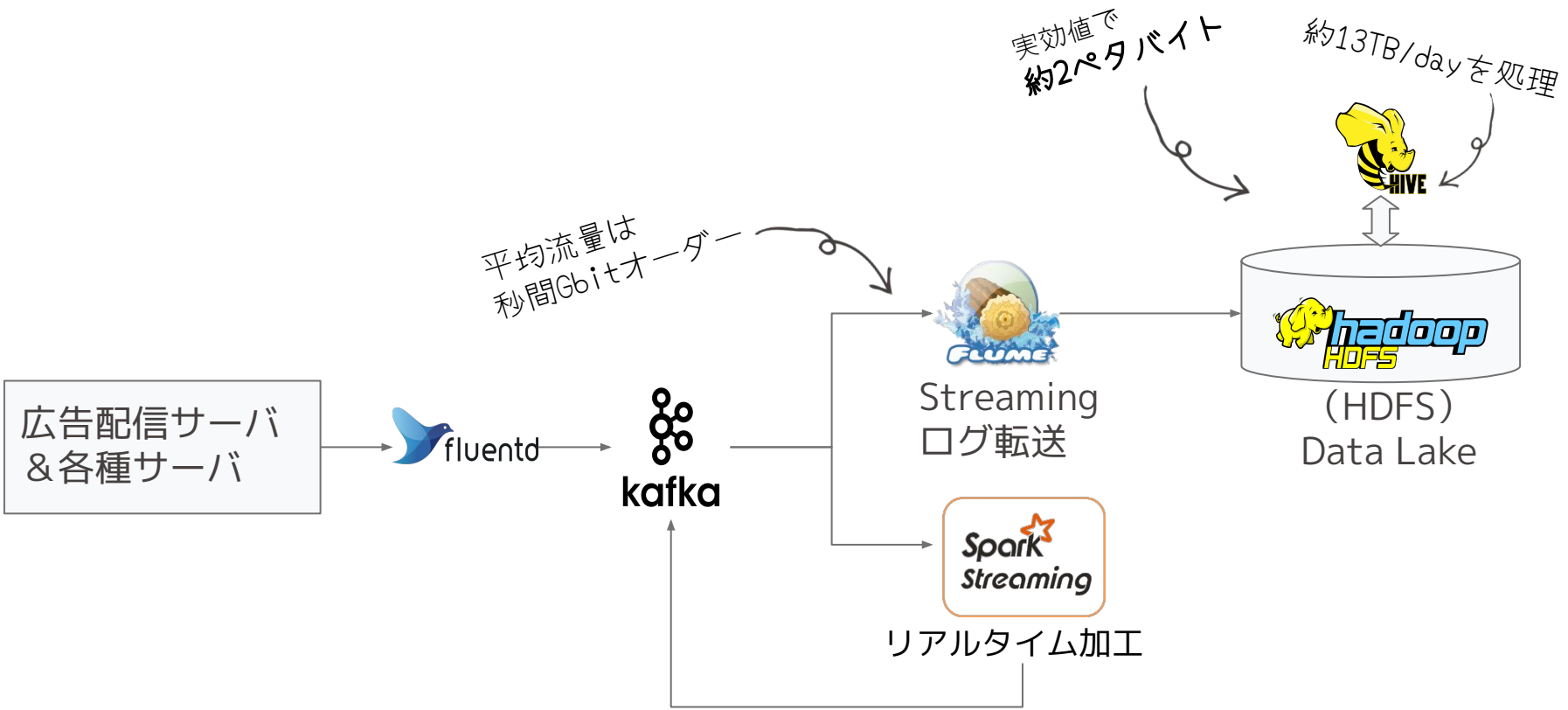
1. 背景説明
2. Kafka Connectをざっくりと
3. Iceberg Sink Connectorの紹介
    a. しくみ
    b. 特徴
    c. 期待している追加機能
4. 今後は
5. この後の雑談したいネタ

事業紹介（データプラットフォーム事業）

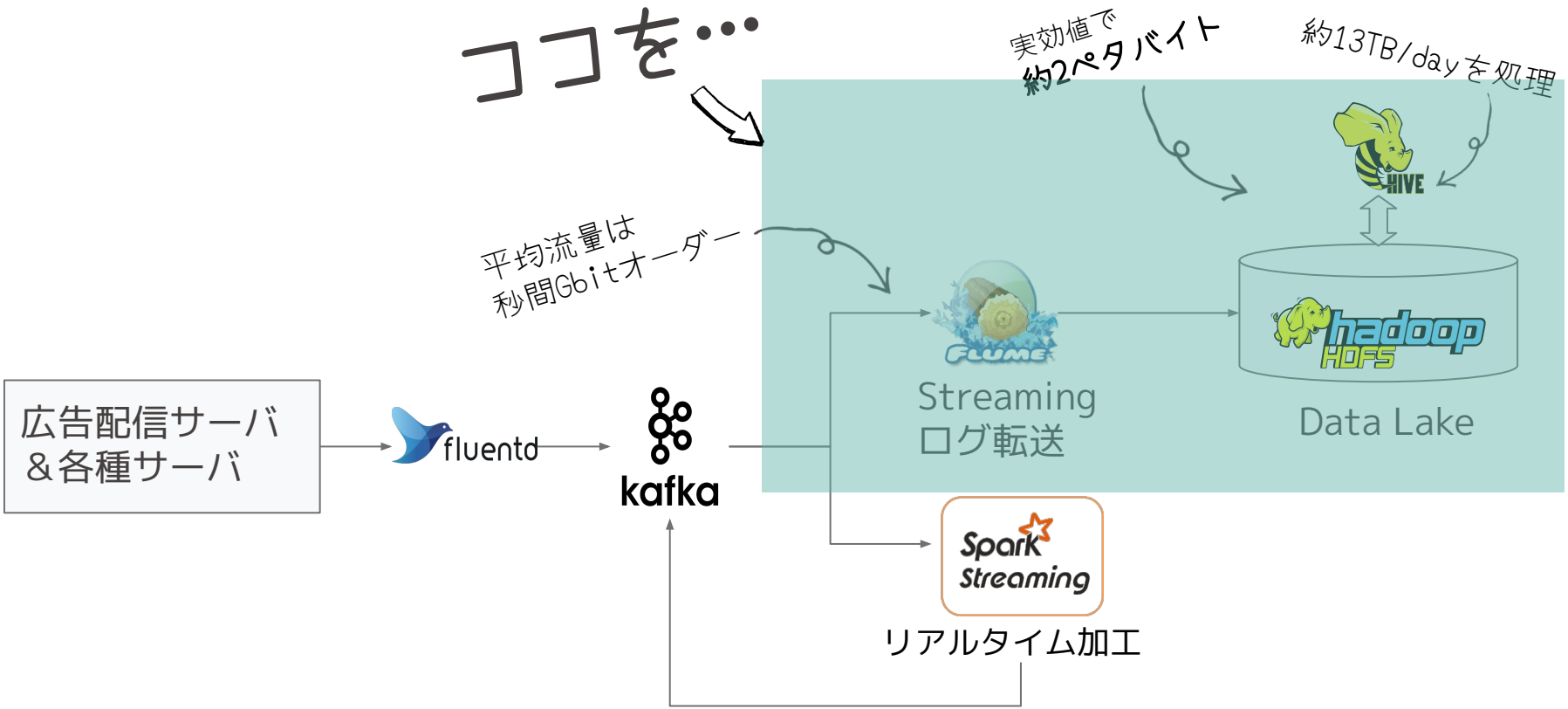
MicroAd
Redesigning the Future Life
#otfsg_tokyo



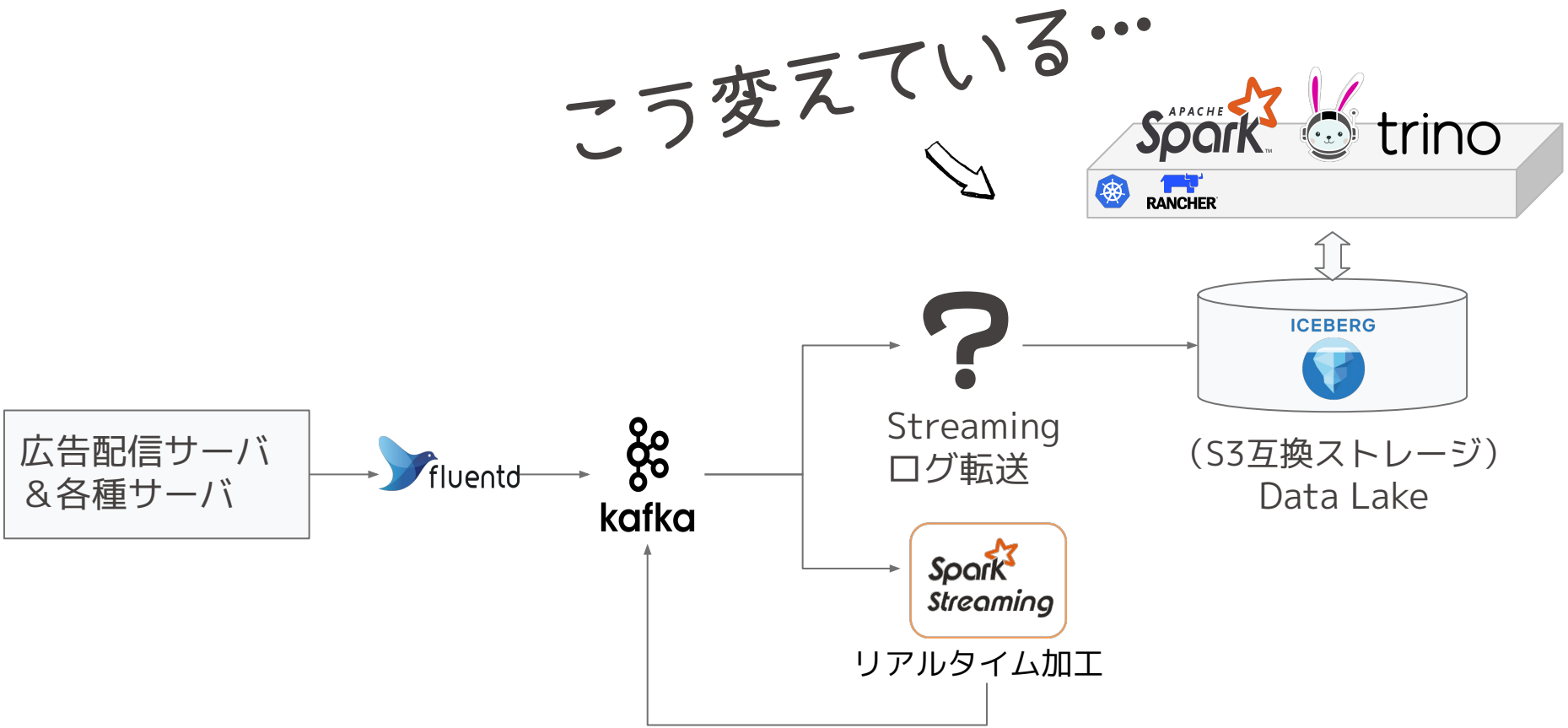
ざっくりデータ基盤のどのへんの話？



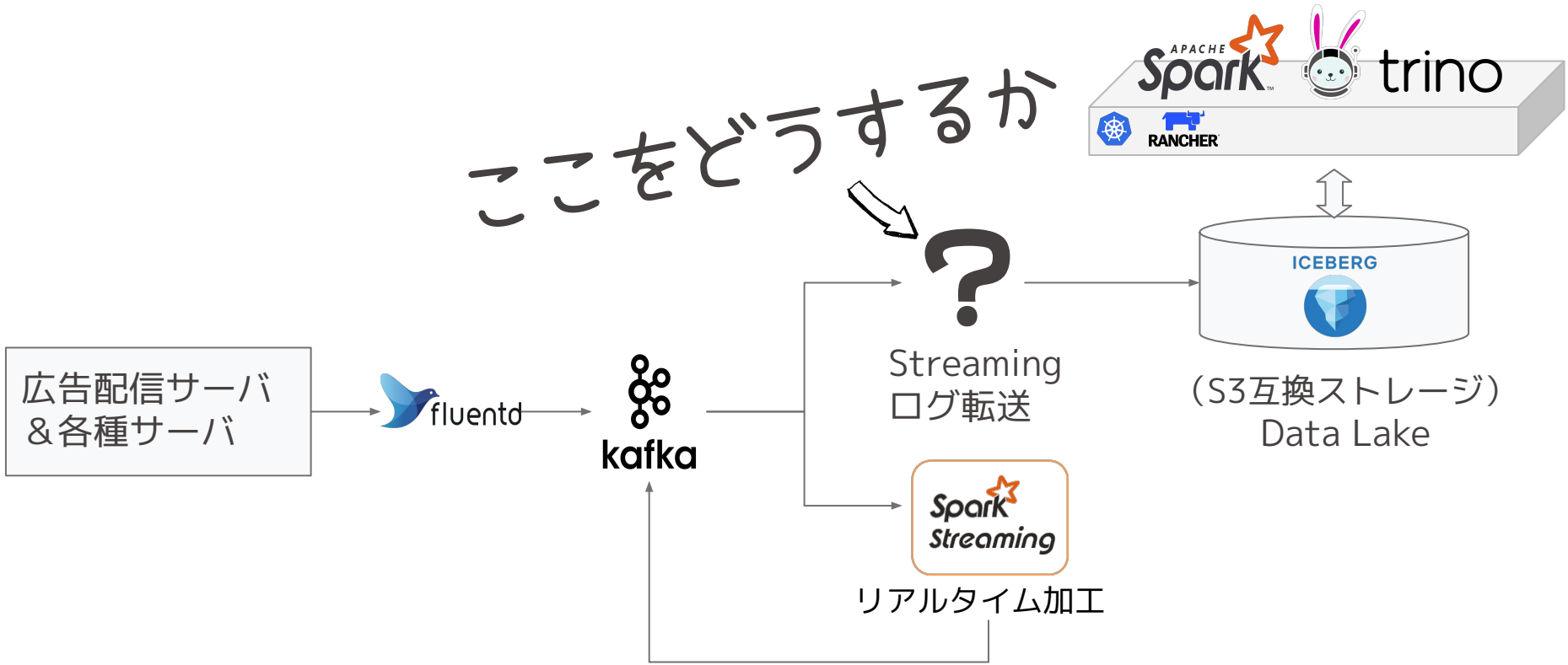
ざっくりデータ基盤のどのへんの話？



ざっくりデータ基盤のどのへんの話？



ざっくりデータ基盤のどのへんの話？



何で変えたいのか？

運用変わらんし
楽っちゃ楽

1. **FlumeがデフォルトでS3をサポートしてない**
   - FlumeにHadoop-AWSモジュールを入れてビルドして、
     Sinkのパスを hdfs:// → s3a:// とするのもアリ???
2. **Sink先はIcebergテーブルを利用するので活かしたい**
   - KafkaのTopicをConsumeして
     直接Icebergテーブルに挿入できるなら変換処理の手間が省ける

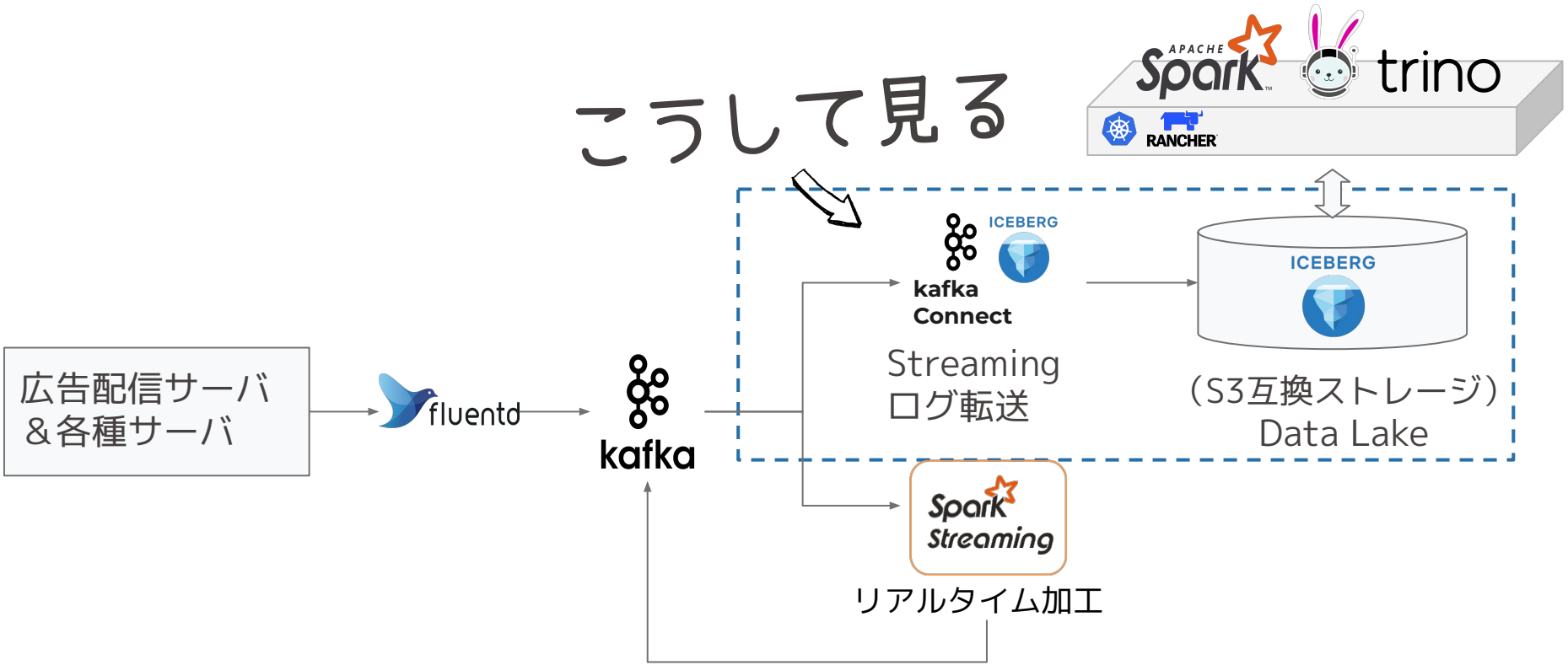
Streamingログ転送に求める事

1. メッセージのデータ型のJSONをデシリアライズ
2. 日時の文字列カラムをパースしてSinkするパスに変換
3. ネストしたデータ構造をフラット化
4. デイリーでローテーションしてるTopic（topic.20240102）を1つのSinkとして扱いたい
5. （出来れば）Icebergテーブルに直接レコードを挿入
6. 可能な限り必要になるコンポーネントは少なく

Streamingログ転送に求める事

1. メッセージのデータ型のJSONをデシリアライズ
2. ~~日時の文字列カラムをパースしてSinkするパスに変換~~
   - 🙌 直接Icebergテーブルに挿入するので考慮不要
3. ネストしたデータ構造をフラット化
4. デイリーでローテーションしてるTopic（topic.20240102）を1つのSinkとして扱いたい
5. （出来れば）Icebergテーブルに直接レコードを挿入
6. 可能な限り必要になるコンポーネントは少なく

という訳で


こうして見る
APACHE Spark
trino
RANCHER
ICEBERG
kafka Connect
Streaming
ログ転送
ICEBERG
（S3互換ストレージ）
Data Lake
広告配信サーバ
＆各種サーバ
fluentd
kafka
Spark Streaming
リアルタイム加工

そもそもKafka Connectとは

Kafka Connectとは、Apache Kafkaの一部（Confluent製品だと誤解してた）で、
データパイプラインを実行・管理するランタイム。

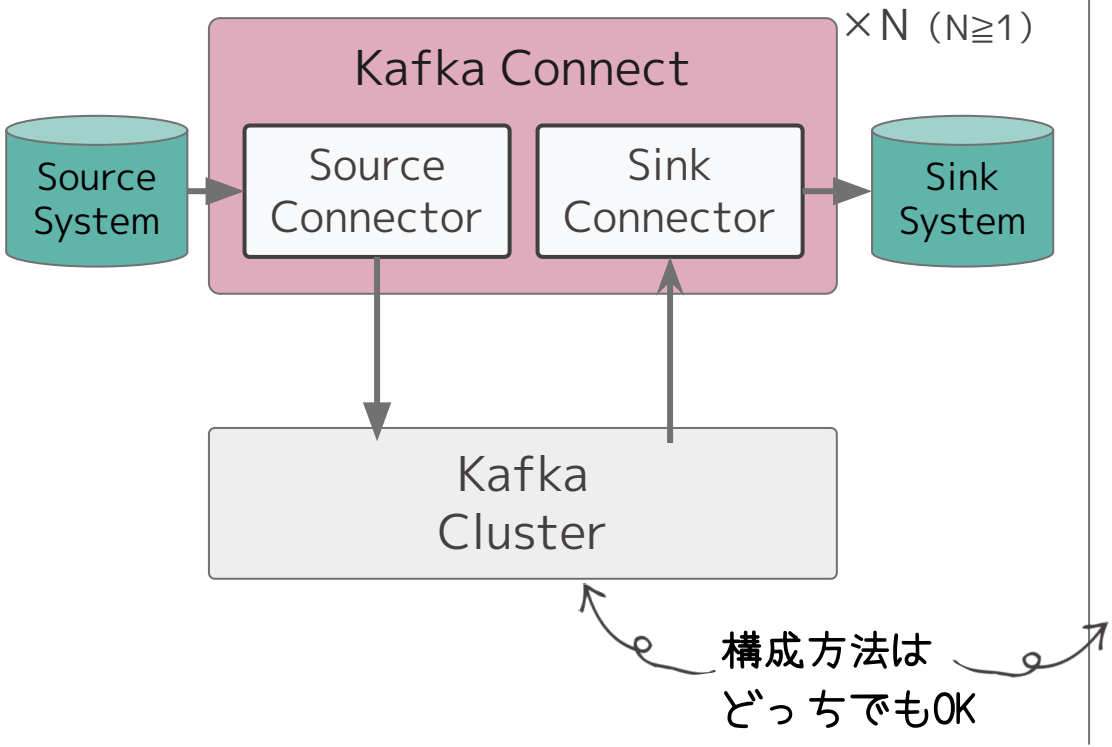
Kafka Connectは、プラグインを組み合わせて複雑なデータパイプラインを構築します。
パイプラインを定義するためのプラグインをコネクタプラグインと呼びます。
コネクタプラグインには以下の種類があります。

- 外部システムからKafkaにデータをImportする **Source Connector**
- Kafkaから外部システムにデータをExportする **Sink Connector**
- Kafka Connectと外部システム間でデータを変換する **Converter**
- Kafka Connectを流れるデータを変換する **Transformation**
- 条件付きで変換を適用する **Predicate**

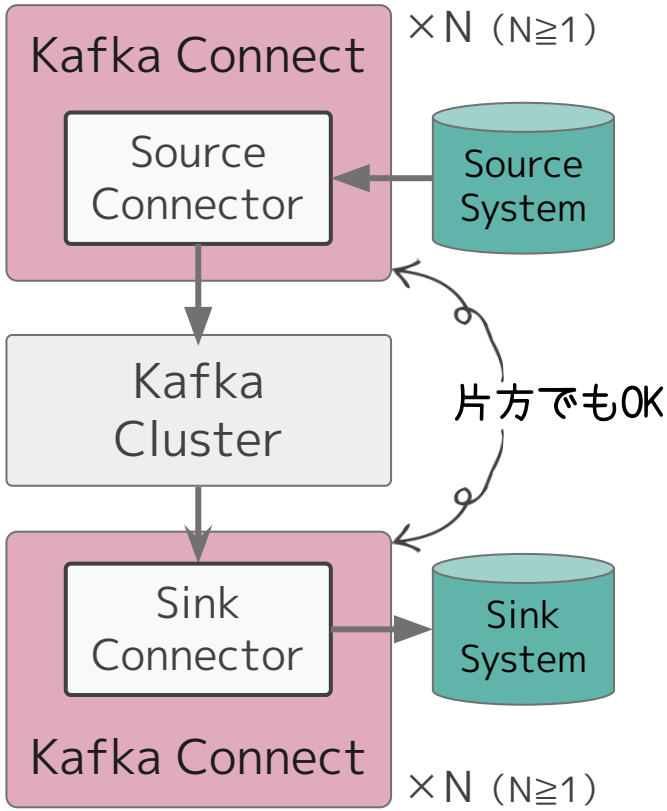
設定変更や操作などはREST APIで行えるので自動化と相性が良い。

そもそもKafka Connectとは

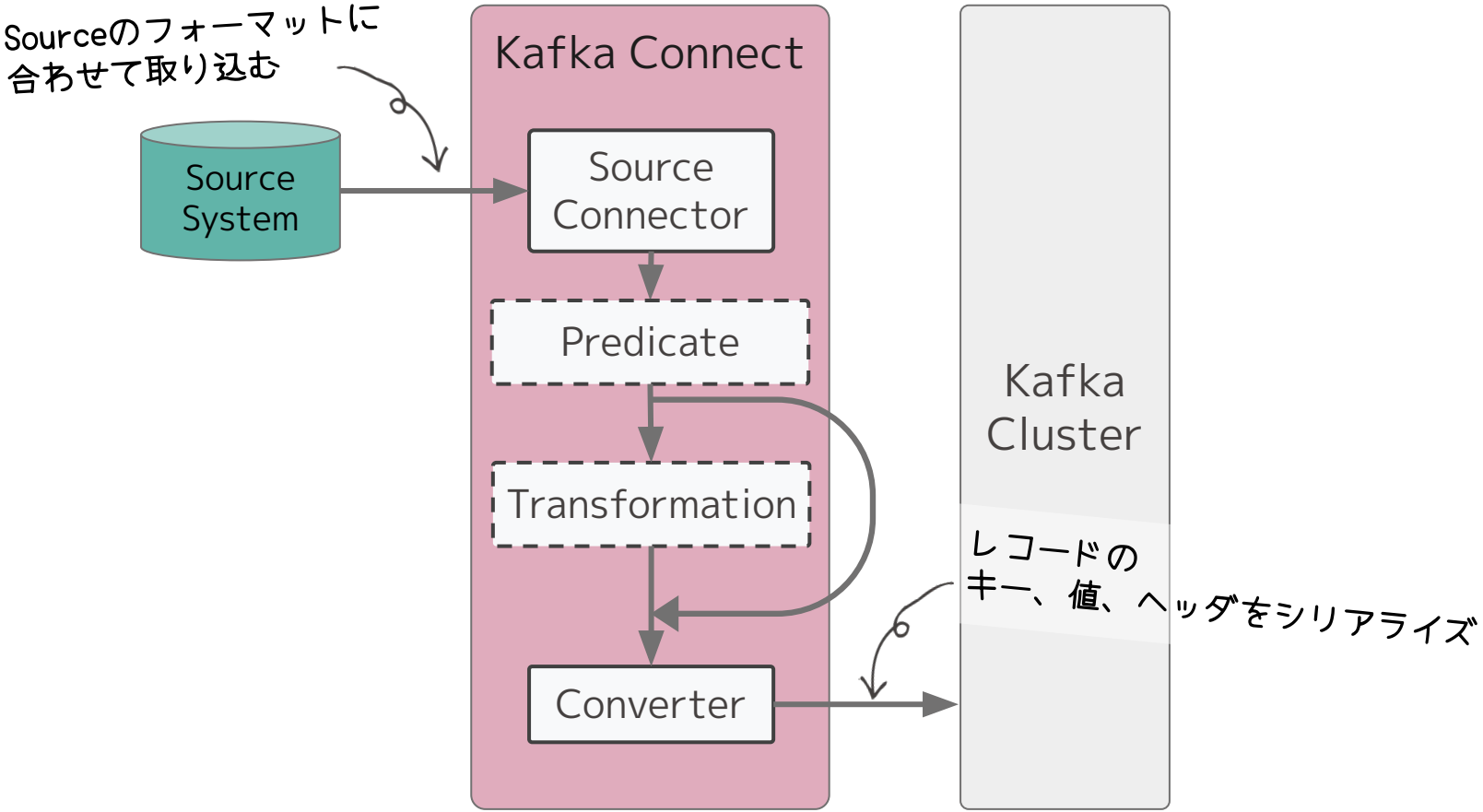
ProducerとConsumerを同じ場所で構成



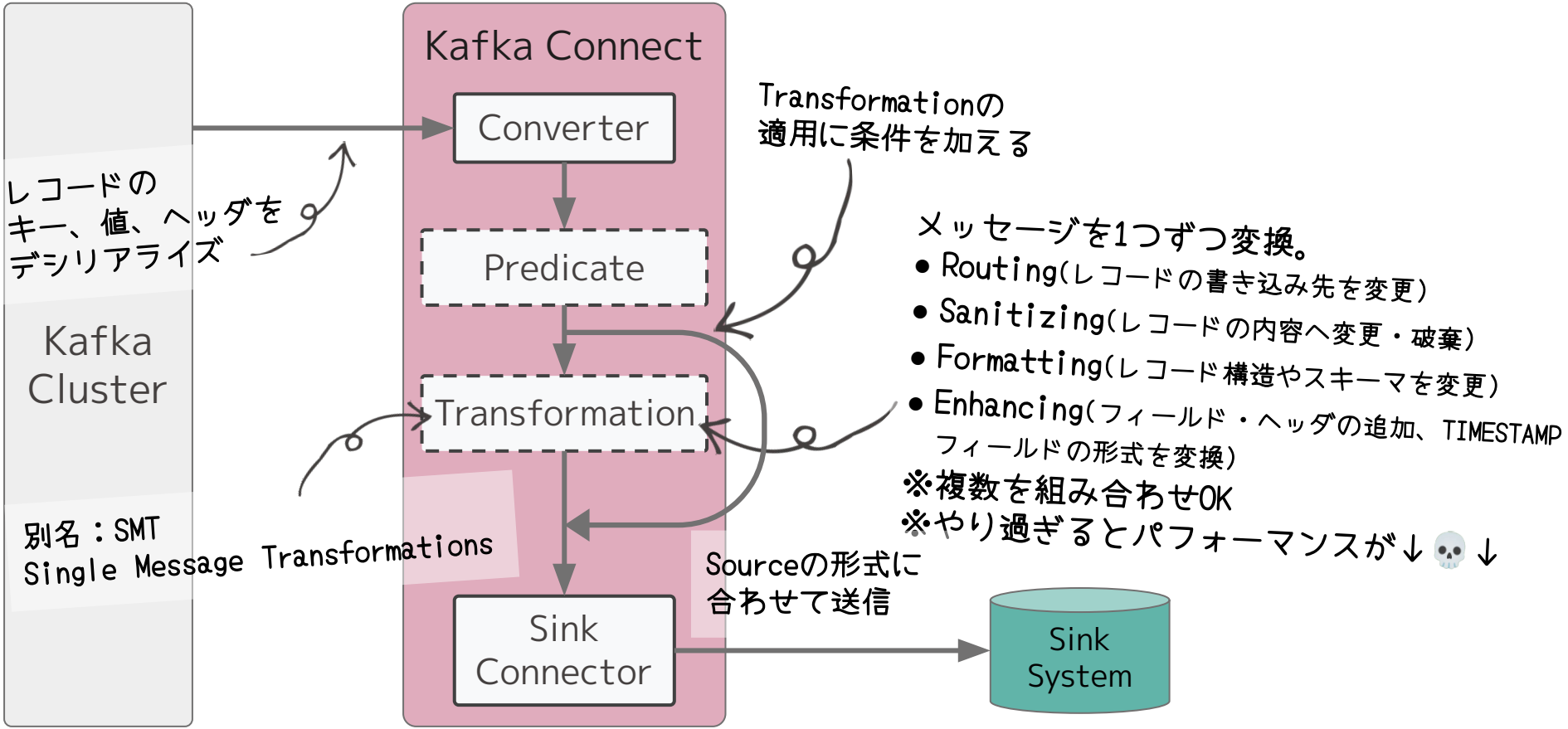
ProducerとConsumerを分けて構成



外部システムからKafkaにImportするまで



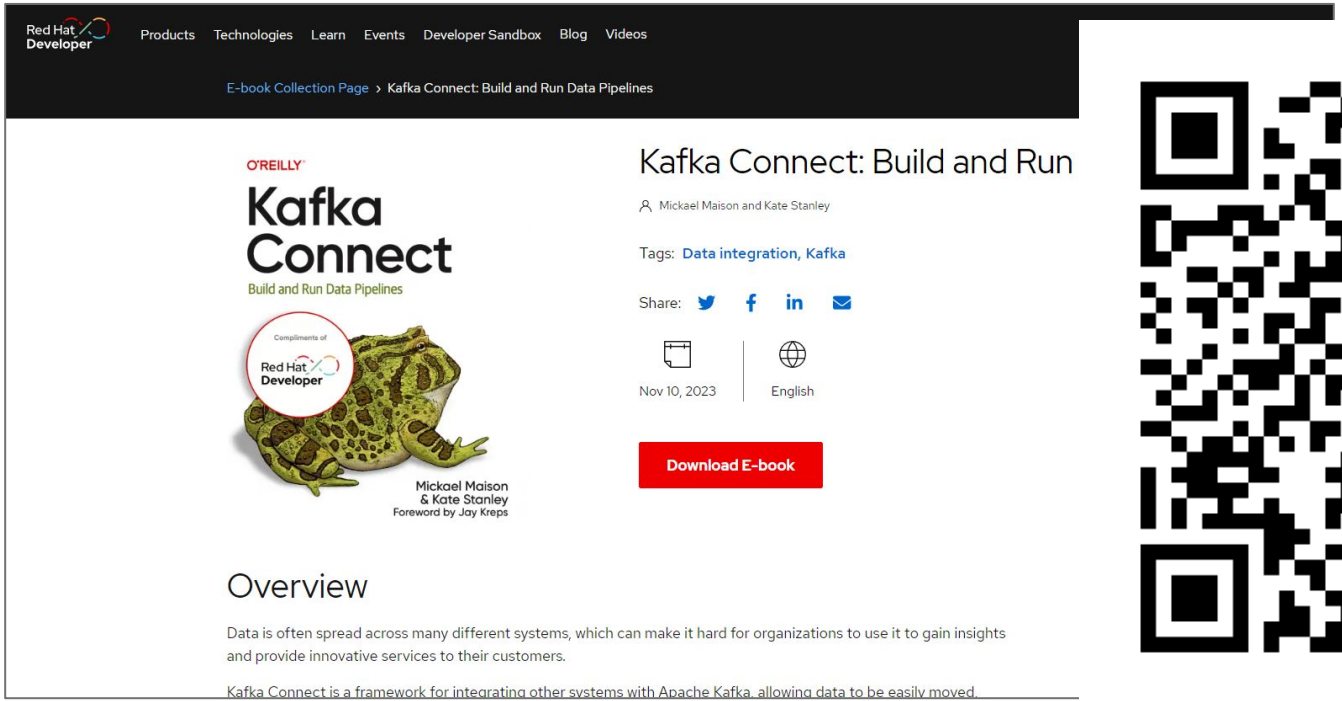
Kafkaから外部システムにExportするまで



続きは、、

以下のパックマンフロッグ本がとても参考になります（日本語版欲しいなぁ。。。）

Kafka Connect: Build and Run Data Pipelines
https://developers.redhat.com/e-books/kafka-connect-build-and-run-data-pipelines



Iceberg Sink Connectorとは

KafkaのTopicをConsumeして、Icebergテーブルに取り込むSink Connector。

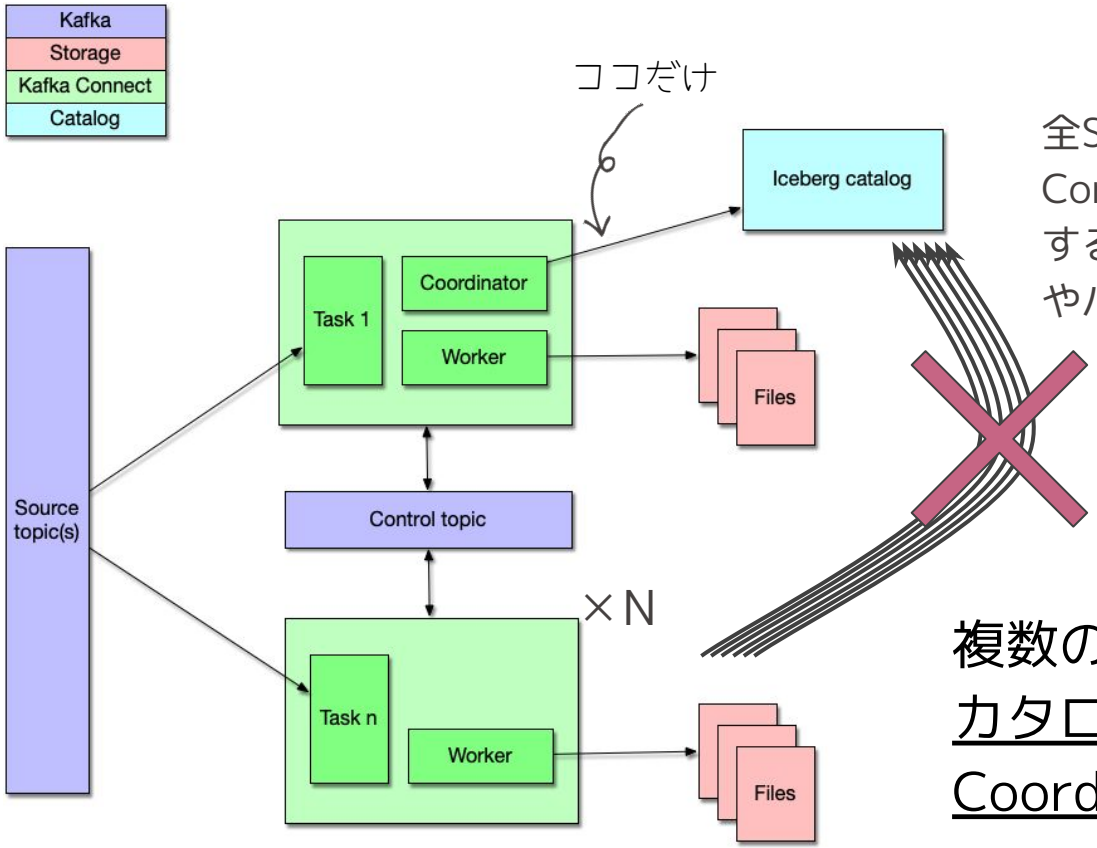
もともと、Tabularの製品だったが、現在Apache Icebergに合流中🎉🎉🎉

apache/iceberg#8701 ・ apache/iceberg#9466 ・ apache/iceberg#9641

主な特徴

- Icebergテーブルへのコミットを一元化するためのコミット調整
- **Exactly-once**（正確に1度だけ）にSinkが可能
- 一度に**複数のテーブルにSink**出来る
- 行の変更（update/delete）、**Upsert**に対応
- テーブルの自動作成と**スキーマの進化**
  ＋「フィールド名」と「Icebergテーブルのカラム」のマッピング

Icebergテーブルへのコミットを一元化するためのコミット調整



全Sink ConnectorのWriterからカタログにCommitしてしまうと大量のSnapshotを作成する事になり、メタデータファイルの肥大化やパフォーマンスの課題に繋がってしまう

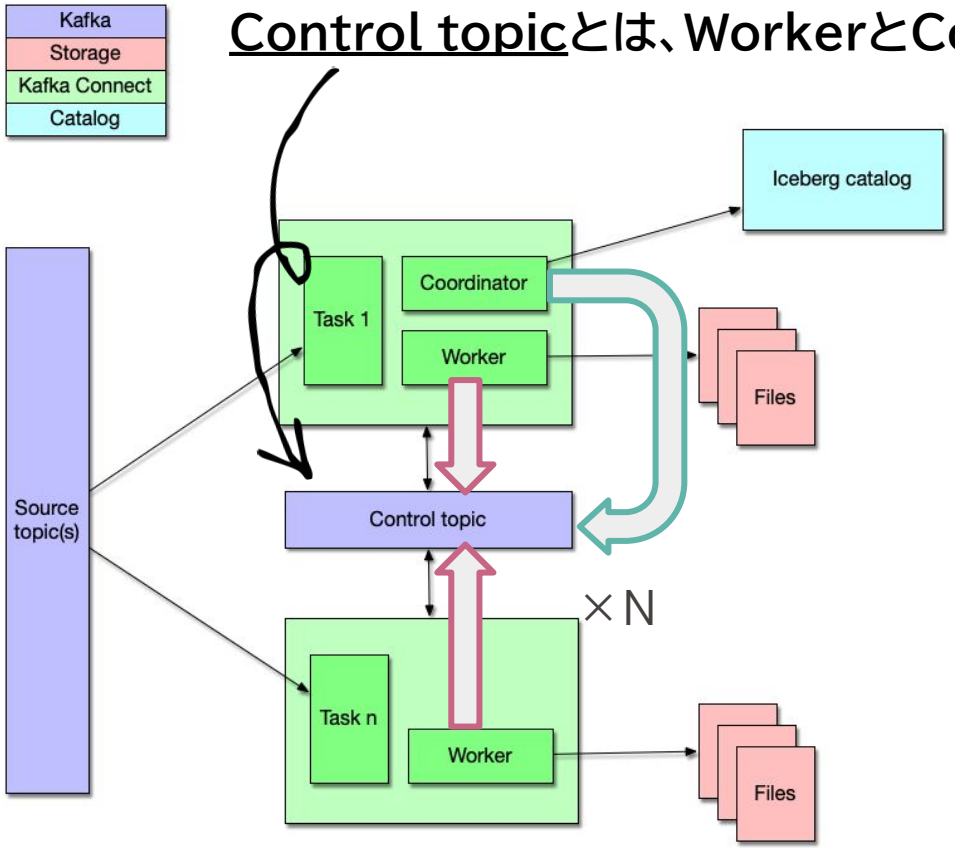
なので。。。

複数のWriterからFileは書込みするが、
カタログにCommitするのは
Coordinatorからの1箇所だけ

cf. github.com/tabular-io/iceberg-kafka-connect/docs/design.md より

Icebergテーブルへのコミットを一元化するためのコミット調整

Control topicとは、WorkerとCoordinator間の通信チャネルを担う。



各々に必要な情報をControl topicにイベントを発行する（混在することになるけど、Worker・Coordinator自身に不要な情報は無視している）。

共用の通信チャンネルを用いることで、途中で落ちても、Kafka Broker側で管理しているControl topicがあるので復旧出来る。

Control topicはAvroを使用してシリアライズされているので、後方互換性を確保しながらスキーマを進化が可能（なので後から仕様変わっても影響が無い）。

cf. github.com/tabular-io/iceberg-kafka-connect/docs/design.md より

Iceberg Sink Connectorのコミットプロセス

Iceberg Sink Connectorのコミットプロセスは以下の順で実行されます。

1. コミットタイマー（iceberg.control.commit.interval-ms）の初期化とチェック
2. コミットプロセスの開始
3. Workerによるデータファイルの準備
4. Coordinatorによるコミットの実行
5. Snapshotプロパティの設定
    a. Control topicのoffset、UUID、完全に処理されたTimestamp VTTS (Valid-Through Timestamp)

Exactly-once（正確に1度だけ）にSinkが可能

Workerは、Kafkaトランザクション内でデータファイルのイベントを送信し、
Source TopicのOffsetをコミットすることで、これを保証します。

Coordinatorは、Control topicのConsumerがコミットされたイベントのみを読み込むように設定し、Control topicのOffsetをIcebergコミットデータの一部として保存することで、これを保証しています。

- Sinkが管理するConsumer GroupのSource topicのOffsetは、
  Control topicに正常に書き込まれたデータファイルイベントに対応する
- OffsetはSnapshotメタデータに保存されるため、
  Control topicのOffsetはIceberg Snapshotに対応する

これってつまり、運用上の注意点でもある

Source TopicのOffsetは、以下の2つの異なるConsumer Groupに保存することになる

- **Sink管理Consumer Group（iceberg.control.group-idで指定しているやつ）**
  - Exactly-OnceでSinkする為に使用
- **Kafka Connect管理Consumer Group（デフォルト名：connect-[コネクタ名]）**
  - Sink管理Consumer Groupが見つからない場合のフォールバック用

なので、（障害などの場合など）
Offsetをリセットしたい場合は、両方のConsumer Groupのリセットが必要

一度に複数のテーブルにSink出来る

もちろん、そのままレコードをテーブルに書き込み出来ますが、
以下の様に複数のテーブルへSinkも出来る。

- **Multi-table fan-out, Static routing**
  - 指定したフィールドの値に応じて、指定するテーブルに書き込む。
    その他のレコードはスキップ。
- **Multi-table fan-out, Dynamic routing**
  - 指定したフィールドの値を名前とするテーブルにレコードを書き込む。
    テーブルが無いならレコードはスキップ。

行の変更（update/delete）、Upsertに対応

⚠️行レベルの更新と削除に対応出来るIcebergテーブルが**Iceberg v2形式が必要**

行の変更

iceberg.tables.cdc-fieldで**指定したフィールドの値**に応じて、
Icebergテーブルへの操作を変更する。

- I：追加操作（append）として機能
- D：等価削除操作（equality delete）として機能
- U：等価削除操作に続いて追加操作を行うことで、更新として機能

Upsertモード

iceberg.tables.upsert-mode-enabled=true とする事で
すべての受信レコードが「更新」として扱われ、各レコードに対して、等しい削除が実行され、その後に追加されます。

テーブルの自動作成とスキーマの進化

メッセージは、Icebergスキーマに最も適合するようにIcebergレコードに変換される。

フィールドは Icebergの名前マッピングと一致するようにマッピングされる。
もし、フィールドの名前マッピングが定義されていない場合は、Iceberg スキーマのフィールド名が使用されます。

流れとしては、、

Source側のSchema（Avro、JSON、Protobuf）
→ Connect側で型変換
→ Icebergテーブルのスキーマと比較
→ 差分があれば、Icebergテーブルをスキーマ進化して追従させる。

エラーハンドリングどうするか

おかしいレコードを違うトピックに流したり、エラーについて通知したい。

Kafka Connect自身の機能には、以下のプロパティで利用できる

- **errors.tolerance**
- **errors.deadletterqueue.context.headers.enable = true**
  - メッセージの拒否理由に関する情報をメッセージ自体のヘッダーに書き込む
  - デッド・レター・キュー上のメッセージを調べるには、Consumer系ツールなら何でも良い（ksql、kafkacatとかTrinoのKafka Connectorでも良さそう）。
- **errors.log.enable = true**
  - メッセージを拒否した理由をログに書き出す

ただ、現状、Iceberg Sink Connectorでは未実装😭

- tabular-io/iceberg-kafka-connect#152
- tabular-io/iceberg-kafka-connect#183

今後は、、

- ネストしたJSONスキーマのフラット化（純粋にはIceberg Sink Connectorとは違うけど）
- スキーマ進化がどの程度まで柔軟にいけるのか確認
  - 追加くらいしか試せてない（ネストしたフィールド側の追加とかは？）
- Schema Registryの無しでどれくらいつらみがあるのか体験
  - そもそもSchema Registryなに使うか問題
    - ConfulentのSchema Registory？ AivenのKarapace？
- 大きめの流量のTopicをConsumeしてどうなるか確認してみたい
  - メタデータまわりの状況
  - Icebergテーブルの最適化（CompactionやSnapshotのExpireの頻度とか）どうするか
- Iceberg Sink Connectorの設定の管理＆反映まわりをGitのPRベースの運用に落とし込む対応の検討

この辺をこの後、雑談したい。

- Kafka または、Kafka ConnectをKubernetesで運用するのしんどない？
  - 構成要素が単純（特にKafka Connect）なので、Kubernetesと相性は良さそう
  - ただ、KubernetesのアップデートやOperatorのアップデートに引きずられるけど、そこをどうするか？
- Sink Connectorサーバ1台あたり、どれくらいさばけるのか？
  - 何Topicくらいまで面倒見られそうか
- Schema Registory無しじゃだめ？
  - Iceberg Sink Connector側でTopicのスキーマ変更を検知してIcebergテーブルをスキーマ進化するなら不要？
  - Graceful Shutdown出来るなら、設定変更に伴うKafka Connectの再起動はズグだし

いかがだったでしょうか。。。

そんな貴方にDocker Composeな検証環境あります。

MicroAd
Redesigning the Future Life
#otfsg_tokyo

実際に触って試したい場合は、以下をどうぞ。

https://github.com/Wuerike/kafka-iceberg-streaming

KakaとKafka Connectを**Redpanda**を使って構成しています（UIがあるのでむっちゃ便利）。
UIからTopicやConnectorの状況や設定の変更も可能。

ストレージには**MinIO**を使い、Icebergカタログは**RESTカタログ**を用いています。

Jupyter Notebookとセットになった**Spark&Iceberg**もあるので、
NotebookからブラウザからIcebergテーブルの操作も可能。

Kafka に流すデータは、**Benthos**を使って生成。
GolangのライブラリのfakerをベースにBenthos固有のBloblangってのを使うと
ダミーデータを生成してデータをPublish出来るので非常に便利。

こんな感じでダミーレコード作成出来る


```
input:
  generate:
    count: 1000 # 生成するメッセージの数
    interval: 1s # メッセージ生成の間隔
    mapping: |
      #!blobl
      let choices = ["debit", "credit", "bank_slip"]
      root.id = uuid_v4()
      root.type = $choices.index(random_int(seed:timestamp_unix_nano()) % $choices.length())
      root.created_at = timestamp_unix().format_timestamp("2006-01-02T15:04:05","UTC")
      root.document = random_int(seed:timestamp_unix_nano(), min:1, max:100).string()
      root.payer = fake("name")
      root.amount = random_int(seed:timestamp_unix_nano(), min:10, max:10000)
```

⬆️が⬇️の様になる


```
{"amount":9424,"created_at":"2024-02-29T12:23:03","document":"75","id":"4f62daea-b880-46ba-9159-215a01405aed","payer":"Miss Aniya Rath","type":"bank_slip"}
{"amount":3114,"created_at":"2024-02-29T12:23:04","document":"31","id":"d564b7ab-b5d2-4741-a375-0f9ae7974e85","payer":"Mrs. Juanita Hermiston","type":"credit"}
{"amount":8695,"created_at":"2024-02-29T12:23:05","document":"100","id":"d153c4d4-f4aa-4a44-827c-7dc6cd0b2bd1","payer":"Dr. Nella Mante","type":"credit"}
```

補足

- Kafka Connect自身に関する情報
  - Kafka Connect Deep Dive – **Converters and Serialization Explained** | Confluent
    https://www.confluent.io/ja-jp/blog/kafka-connect-deep-dive-converters-serialization-explained/
  - Kafka Connect Deep Dive – **Error Handling and Dead Letter Queues** | Confluent
    https://www.confluent.io/blog/kafka-connect-deep-dive-error-handling-dead-letter-queues
  - Iceberg Sink Connectorのドキュメント
    https://github.com/tabular-io/iceberg-kafka-connect/tree/main/docs
  - Iceberg Sink Connector向けのSMTのドキュメント
    https://github.com/tabular-io/iceberg-kafka-connect/blob/main/kafka-connect-transforms
- 検証環境に関連する情報
  - faker：https://pkg.go.dev/github.com/go-faker/faker/v4
  - Benthosで生成出来るダミーデータ仕様
    https://www.benthos.dev/docs/guides/bloblang/functions/#fake