[めもおきば てっくれぽーと] クラウドとセキュリティ TL 流速を支えるエンジニアの総合誌

めもおきば Tech Report

2018
10
October

サーバーレスで実現する

ストリーム処理

徹底活用術

めもおきば TechReport 2018.10

— 目次 —

Aki @ nekoruri

サーバーレスで実現するストリーム処理徹底活用術

サーバーレス技術の代表的ユースケースの一つが、継続して発生する「ストリームデータ」を低遅延で処理するストリーム処理です。

小さい処理を Pub/Sub でパズルのようにつないでいくと、様々なシステムを実現することができる、まさにサーバーレス向きのアーキテクチャです。

その一方で、代表的ユースケースの一つとは言え、もとより技術的困難さが多い分野ということで、様々な苦労を隠蔽してくれるはずのサーバーレス構成において依然としてつらい状況に直面することが多くあります。

というわけで、今回は AWS や Azure におけるサーバーレスストリーム処理について解説します。

ストリーム処理の用途

ストリーム処理は、いわゆる「リアルタイムで何か知りたい」という要求に対応するデータ処理です。具体的には以下のような用途で広く使われています。

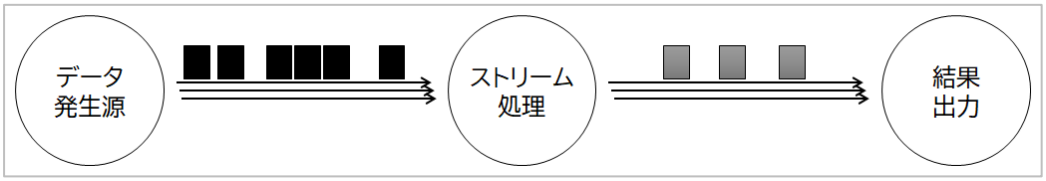
- アクセスログのリアルタイム解析
- 異常検知（例：不正ログインの監視・対策自動化）
- SNS（例：Twitter のタイムライン処理）
- IoT のセンサーデータの集計（例：異常動作を検知して交換品の発注、現在位置の推測）
- 株価変動に基づく自動売買
- サーバ監視のメトリクス集計
- CQRS（コマンド・クエリ責務分離）アーキテクチャにおけるコマンド（書き込み）処理

このような分析処理などを一日などまとまった単位で行うと、どうしても大量のデータを処理する必要があり時間も掛かります。ストリーム処理では、後ほど紹介するデータの不完全さなどいくつかの制約に目を瞑ることによって、小さなデータ処理の繰り返しに分解することができます。これによって、すぐに結果をえられ、それに基づいたアクションを素早く起こすことができます。

さらに、スケーラビリティの観点やサーバーレスの普及により CQRS アーキテクチャが注目されているため、データ分析用途以外においても今後ストリーム処理はより広く使われていくと予想しています。

ストリーム処理の基本

ストリーム処理をとてもざっくり言うと、無限に発生するデータに対して何らかの加工や集計をおこない、少ない遅延で何らかの結果を出力するというものです。



一般的なデータ処理と比較すると、いくつかの大きな違いがあります。

- 小さなデータが届くたびに何らかの処理を行い、それまでに届いたデータに基づいて、何らかの結果を出力します。
- データ発生源からデータが届くには時間が掛かる。
- データの発生順と入れ替わって異なる順番でストリーム処理に届く、あるいはデータの到着にとてもとても時間が掛かる場合がある。また、届いていないデータがあることを確認できない。
- 全てのデータが揃っていない不完全な状態でも、結果を出力しなくてはならない場合があり、その場合は近似値となる。

そのため、イベントが発生した時刻(Event Time)と、処理を行った時刻(Processing Time)という 2 つの時刻に分けて、必要な整合性(もしくは割り切り)を確保する必要があります。また、遅延が前提となる処理であるため、いわゆる組み込み開発系で言われるようなリアルタイム処理との違いを尊重して、ニアリアルタイム処理と呼ばれることもあります。

ストリーム処理は、大きく 2 種類の道具で構成されます。一つは演算処理を実際に行うストリーム処理エンジン、もう一つはデータ発生源とストリーム処理エンジンを接続するメッセージングサービスです。さらにストリーム処理エンジンは、イベント一件ずつ逐次的に「真のストリーム処理」を行うイベント個別型と、時間や件数でイベントを集めてかたまり毎に処理を行うマイクロバッチ型に分けられます。今回扱うのは全てマイクロバッチ型です。

ストリーム処理は、これだけで厚い本が何冊も書けてしまう非常に面白い技術領域なので、是非追いかけてみてください [1]。

[1] 入口としては、@kimutansk 氏の以下の記事やスライドが分かりやすいです。
https://qiita.com/kimutansk/items/60e48ec15e954fa95e1c
ストリーム処理とは何か？＋2016 年の出来事
https://www.slideshare.net/SotaroKimura/ss-72769963
最近のストリーム処理事情振り返り

AWS におけるストリーム処理

AWS でストリーム処理を提供するサービスはいくつかあります。

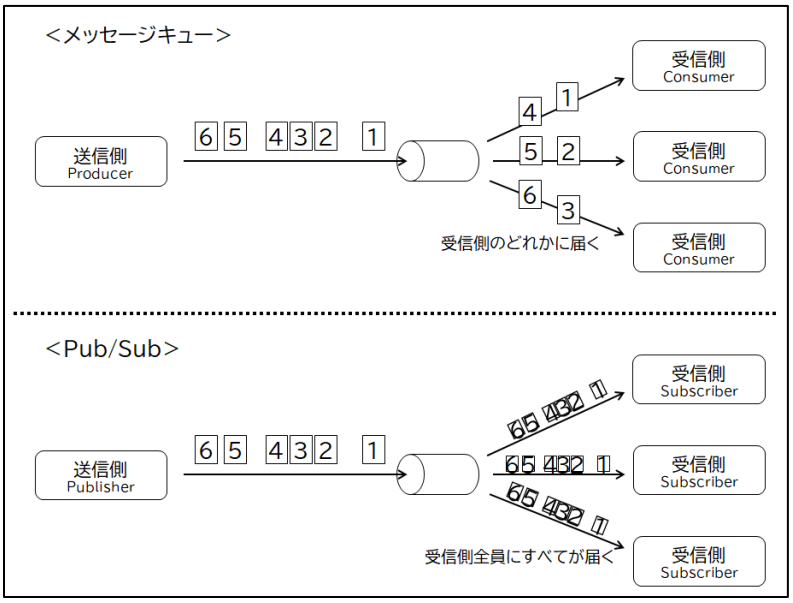
メッセージングサービス	ストリーム処理エンジン
Kinesis Data Streams	Kinesis Data Analytics
EMR（Apache Kafka）	AWS Lambda
※ Kinesis Data Firehose	EMR（Spark Streaming や Apache Fink 等を利用）

もともとストリーム処理という技術分野自体が Apache Hadoop を基盤とする OSS と共に成長してきたこともあり、ストリーム処理エンジンとしての機能を考えるとそれらに一日の長があります。EMR 上でそれらを動かす事もできますが、今回の記事ではサーバーレスありきということでそれらは対象外とします。

Kinesis Data Streams

AWS のストリーム処理サービスにおいて一番重要なものが、メッセージキューを提供する Amazon Kinesis Data Streams です。まず Kinesis Data Streams が中心となり、そこに様々なデータ発生源が集まり、用途に応じたストリーム処理エンジンに流し込みます。

Kinesis Data Streams は、大量のストリームデータ扱うことができる Pub/Sub 型の非同期メッセージングサービスです。混同されがちな一般的なメッセージキュー（Producer/Consumer 型）では、送信者（Producer）から送られたメッセージは、「どれか一つ」の受信者（Consumer）に届きます。Kinesis Data Streams を含む Pub/Sub では、送信者（Publisher）から送られたメッセージの全てが、全ての受信者に届きます。



またデータの欠損・重複については「at least once（少なくとも 1 回）」を提供しているため、基本的にデータが「抜ける」ことはありませんが、同じデータが二回以上届く事があります。そのため受信者は重複したデータが届いても良いような処理（冪等性）にするか、それによる誤差を諦める必要があります。

もう一つの特徴は、受信者が非同期であるということです。言い換えれば Pull 型の Pub/Sub です。受信者のアプリケーションが Kinesis Data Streams に対してポーリングを行い新しいデータを取得します。1 秒あたり 5 回にポーリングが制限されているため、200 ミリ秒より細かい単位でデータを処理したい場合には Kinesis Data Streams を利用することはできません。

非同期という事は、必要な受信者にデータが届くまで Pub/Sub 側でデータを保持し続ける必要があります。Kinesis Data Streams では、受信したストリームデータを最低 24 時間その内部で保存します。それぞれの受信者は、保存データ内のカーソルとも言うべきシャードイテレーターを持ち、新着分までの差分データを取得します。

Kinesis Data Analytics

Amazon Kinesis 一族にはストリーム処理エンジンが用意されています。それが Kinesis Data Analytics です。ストリームデータに対して SQL で様々な集計を行うことに長けています。

最も分かりやすいユースケースがウインドウ集計です。ウインドウ集計では、「直近の 10 分間」などを対象範囲（ウインドウ）として集約関数を適用します。さらに、集計の間隔によって 3 種類に分けられます。

- Tumbling Window 集計：一定時間毎にその間の結果を返します。
  例:10 分ごとに、その 10 分間の間を対象に集計
- Hopping Window 集計:一定時間毎に、直近の期間に対する集計を返します。
  例：1 分ごとに、直近 10 分間の間を対象に主系
- Sliding Window 集計：イベントのタイミングで、直近の期間に対する集計を返します。集計するタイミングが異なるのみなので、Hopping Window とあわせて Sliding Window と呼ぶことも多いです。
  例：ログインしたときに、直近 1 時間での不正ログインの回数を数える。

株価情報を模したデモ用ストリームデータ [2]が用意さているため、これを使って Tumbling Window 集計と Sliding Window 集計を行ったときの SQL を紹介します。なお、これらは全て Developer Guide[3]にあるものです。

[2] 株式銘柄コード（4 文字）、業界分野、差分、株価の 4 項目がひたすら流れてきます。

[3] https://docs.aws.amazon.com/kinesisanalytics/latest/dev/windowed-sql.html

■ Tumbling Window

60 秒毎に、銘柄毎の安値・高値を出力するのが以下の SQL です。

```
CREATE OR REPLACE STREAM "DESTINATION_SQL_STREAM" (
                                  ticker_symbol VARCHAR(4),
                                  Min_Price     DOUBLE,
                                  Max_Price     DOUBLE);

CREATE OR REPLACE PUMP "STREAM_PUMP" AS
  INSERT INTO "DESTINATION_SQL_STREAM"
    SELECT STREAM Ticker_Symbol,
                  MIN(Price) AS Min_Price,
                  MAX(Price) AS Max_Price
    FROM    "SOURCE_SQL_STREAM_001"
    GROUP BY Ticker_Symbol,
             STEP("SOURCE_SQL_STREAM_001".ROWTIME BY INTERVAL '60' SECOND);
```

前半では出力先のストリーム「DESTINATION_SQL_STREAM」を定義しています。後半で肝心のストリーム処理を定義しています。重要となるのは、GROUP BY 句に書かれている以下の部分です。ROWTIME は Kinesis Data Analytics がデータを受け取った時刻（処理を行う時刻とほぼ同じ）ですので、60 秒毎にこの SQL がその期間を対象に実行されることを指示しています。

```
    GROUP BY Ticker_Symbol,
             STEP("SOURCE_SQL_STREAM_001".ROWTIME BY INTERVAL '60' SECOND);
```

■ Sliding Window

1 件データが来る度に、直近 1 分間の安値・高値・平均値を出力するのが以下の SQL です。

```
CREATE OR REPLACE STREAM "DESTINATION_SQL_STREAM" (
                          ticker_symbol VARCHAR(10),
                          Min_Price     double,
                          Max_Price     double,
                          Avg_Price     double);

CREATE OR REPLACE PUMP "STREAM_PUMP" AS
   INSERT INTO "DESTINATION_SQL_STREAM"
     SELECT STREAM ticker_symbol,
                   MIN(Price) OVER W1 AS Min_Price,
                   MAX(Price) OVER W1 AS Max_Price,
                   AVG(Price) OVER W1 AS Avg_Price
     FROM   "SOURCE_SQL_STREAM_001"
     WINDOW W1 AS (
        PARTITION BY ticker_symbol
        RANGE INTERVAL '1' MINUTE PRECEDING);
```

GROUP BY 句の代わりに WINDOW 句が増えました。これが Sliding Window を定義しています。1 分間のウインドウを W1 として定義し、「MIN（Price） OVER W1」のように集約関数の対象期間として指定しています。

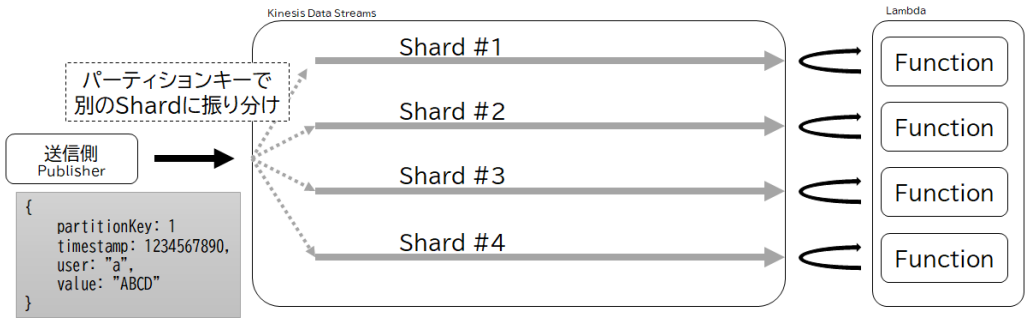
このように、流れてくるデータに対して過去の直近のデータを利用した集計などを簡単に実現できるのが Kinesis Data Analytics です。提供されている SQL で表現できないことはできませんが、Lambda を併用した前処理など機能追加によってできることが増えていくのは間違いないでしょう。

Kinesis Data Streams と Lambda

今のところ AWS からフルマネージドサービスとして提供されているストリーム処理エンジンは Kinesis Data Analytics のみなので、もう少し凝った処理が必要であれば Lambda を組み合わせるか、Spark Streaming や Apache Fink などを EMR 上で動かす事になります。本書では Lambda を取り上げていきます。

AWS におけるストリーム処理の基本は Kinesis Data Streams なので、その先に Lambda を接続することになります。Kinesis Data Analytics ではあまり気にする必要がありませんでしたが、Lambda と組み合わせる場合は Kinesis Data Streams の仕組みをもう少し深く知る必要があります。

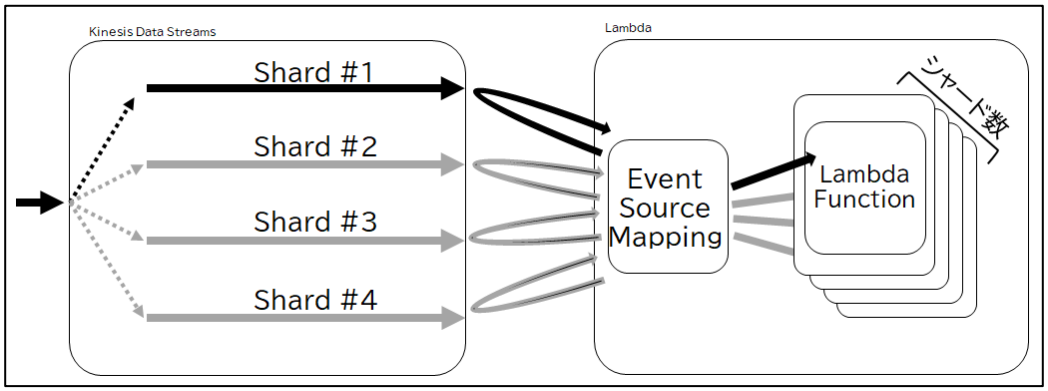
最初にも紹介したとおり Kinesis Data Streams は Pull 型かつ Pub/Sub 型のメッセージングサービスですが、さらに付け加えるなら大量のデータを処理できるようにスケーラブルな分散システムとして設計されています。一つのストリームは、内部では「シャード（Shard）」という単位で分割されて処理されます。Kinesis Data Streams に送られたデータをどのシャードに割り当てるかを決めるのに使われるのが「パーティションキー」です。



ストリームデータはパーティションキーによってどれかのシャードに割り当てられます。同じシャードに追加されたデータ同士はその順序が維持されますが、別のシャードに振り分けられてしまったデータは順序が解らないため、それが必要であればデータの発生時刻やシステムへの到着時刻などをデータ内に保持する必要があります。

データを取得する受信者（Subscriber）側、今回であれば Lambda の Function は、シャードの数だけ並行してデータを受信する必要があります。Lambda の場合は、接続された Kinesis Data

Streams のシャードの数だけ Function が同時実行[4]され、それぞれのシャードからのデータが渡されます。この処理を実際に行うのが「Event Source Mapping」です。



Event Source Mapping は、それぞれのシャードに対応するシャードイテレーターを保持し、各シャードイテレーターによって受信した新しいイベントを引数に、各シャードに対応する Lambda Function を実行します。基本的には 1 秒ごとにポーリングしますが、BatchSize というパラメータによって一度に Lambda に渡すイベントの個数の最大値を指定でき、それ以上のイベントが届いた場合はリトライされます。

Lambda Function には第 1 引数の event として引き渡されます。event.Records に配列として複数のイベントが並んでいますが、さらにその中の kinesis.data に実際のイベントデータが Base64 エンコードされた JSON[5]として格納されています。

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "PartitionKey",
                "sequenceNumber": "〜〜〜〜〜〜",
                "data": "eyJUSUNLR〜〜〜〜〜〜",
                "approximateArrivalTimestamp": 1538917978.966
            },
            (省略)
        },
        ...
    ]
}
```

[4] 実際には（おそらく高速化などのため）、シャード数よりも多くの Lambda Function が同時に「起動」されているようですが、ハンドラ関数が同時に実行されるのはシャード数までとなります。

[5] みんなだいすき eyJ からはじまる文字列ですよ！

というわけで、だいたい処理はこのような感じになります。

```
exports.handler = function(event, context, callback) {
    event.Records.forEach(function(record) {
        var message = new Buffer(record.kinesis.data, 'base64').toString('ascii');
        console.log('message:', message);
        // message を使った実際の処理
    });
    callback();
};
```

あとは、煮たり焼いたり好きなようにできます。

Azure におけるストリーム処理

Azure においても、全体像はだいたい AWS と同じです。

Event Hubs

AWS の Kinesis Data Streams に相当するのが、Azure Event Hubs です。

パーティションキーによって分散するところなど、この二つはほとんど同じような動作をしますが、いくつかの違いがあります。

- Kinesis Data Streams は、「ストリームの名前」という1階層で識別しますが、Azure Event Hubs では「Event Hubs 名前空間」と「Event Hub」という2階層で管理します。アクセス権限や性能などは「Event Hubs 名前空間」という大きな単位で管理します。
- イテレーターの管理は、Kinesis Data Streams ではクライアント側がシャードイテレーターを管理しますが、Azure Event Hubs 側にコンシューマーグループという管理単位が存在します。

Stream Analytics

これも AWS の Kinesis Data Analytics に相当し、SQL でストリームデータの集計ができます。ビルトイン関数などを見ると、本書執筆時点では Azure Stream Analytics のほうが機能が揃っているようです。

Event Hubs と Azure Functions

Pub/Sub メッセージングサービスと FaaS を組み合わせて使うときの仕組みも、おおよそ Kinesis Data Streams＋Lambda の場合とほぼ同じようにできます。

先ほども書いたとおりクライアントがシャードイテレーターを管理するのでは無く、Event Hubs 側でコンシューマーグループという枠組みでイテレーターを管理している点は注意が必要です。Lambda の感覚で、デフォルトのコンシューマーグループに複数の Function をぶら下げてしまうと、どれかの Function だけにイベントが行ってしまい、他の Function にイベントが渡らないことになります。

AWS における Event Source Mapping に代わるものとして、Azure Functions にはより汎用的なバインディングという仕組みが用意されています。管理ポータルからトリガーとして Event Hubs を指定することで、接続に必要なキーを環境変数に保存し、取得先の Event Hubs 名前空間や Event Hub の名前を function.json に保存してくれます。

ここまで設定してしまえば、あとは Lambda と同じように Function の引数として引き渡されます。Kinesis の元の仕様を引っ張っていて Base64 デコードが必要な Lambda と異なり、シンプルにイベント配列として渡されます。

```
module.exports = function (context, eventHubMessages) {
    eventHubMessages.forEach((message, index) => {
        console.log('message:', message);
        // message を使った実際の処理
    });
    context.done();
};
```

あとは、煮たり焼いたり好きなようにできます。

また、バインディングの仕組みを活用して、トリガーで入ってきたイベントに含まれるデータを利用して、CosmosDB などから検索した結果を入力バインディングとして、Function の引数に持ってきたりすることも可能です。

Databricks

Azure には、Apache Spark のフルマネージドサービスとして Azure Databricks が提供されています。本書のスコープからは外れますが、SQL で表現できないような複雑な集計でも、Spark Streaming で実装しフルマネージドなサービスの上で実行することができます。

はまりどころ

さて、ここからはサーバーレスでストリーム処理を実装する上でのはまりどころを書いていきます。

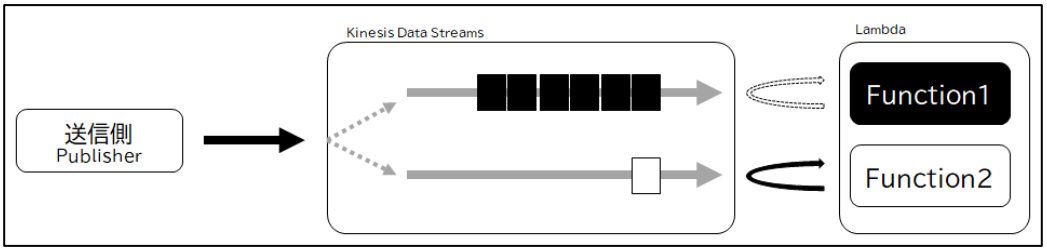
パーティションキーの選び方

Amazon Kinesis Data Streams にしろ Azure Event Hubs にしろ、パーティションキーによって内部で複数のシャードに分散され、さらにシャードを単位として後段の処理が呼び出されます。そのためパーティションキーをどのように選択するかが重要となります。

うまくパーティションキーが設定できていないと、このようなことが発生します。

たとえば、2つのシャードで動かしていた処理があるとします。何らかの理由で Function の処理に時間が掛かるようになっていくと、送信側から入ってくるペースよりも処理できるペースの方が遅くなってしまい、そうなるとストリームの中に未処理のイベントが溜まっていくようになります。

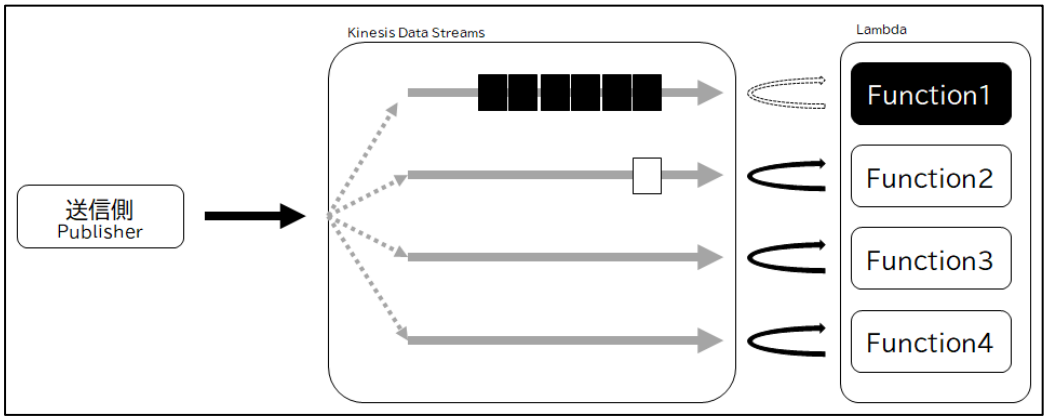
あくまでピーク時だけイベントが多いような場合であれば、ピークを超せばだんだん「処理が追いつく」ようになります。慢性的に処理能力が足りない場合はどんどんストリーム内で未処理のイベントが増え続け、設定された保存期限（デフォルトはどちらも24時間）を超えると古い順にイベントが捨てられます。これは Kinesis Data Streams なり Event Hubs じたいが持つ読み書き量の制限とは別に発生します。



Function の同時実行数を増やすためには、シャード数を増やす必要があります。

ところが、特定のパーティションキーにデータが偏っていると、せっかくシャード数を増やしても同じシャードに振り分けられ、特定の Function のインスタンスにデータが送られてしまい、問題が解決しません。要するに、いわゆる分散 DB[6]における鉄則とまったく同じように、「ホットキー」を避ける必要があります。

[6] そもそも、Pub/Sub メッセージングサービスというもの自体が、決められた時間で自動で削除され、特定の処理・検索だけが行えるデータベースと考えることもできます。



たとえばアクセスログの分析であれば、単純にアクセスされたパスをパーティションキーにしてしまうと、一般的にはトップページのページビューが圧倒的に多いはずで、トップページが振り分けられたシャードと後段の処理ばかり偏ってしまうわけです。

このような問題は、メッセージングサービスを境界として外部のサービスと連携する場合にも問題となります。

IoT プラットフォームの SORACOM には、IoT デバイスから UDP や TCP、HTTP などで送られたイベントを、利用者側のメッセージングサービス[7]に送り込んでくれる SORACOM Funnel というサービスがあります。もともと SORACOM Funnel は大量のデバイスからのデータを集めて送り込んでくれるという設計思想ということで、同じデバイスからのイベントの時系列を保存するためにパーティションキーとしてデバイスを識別できる IMSI[8]を利用していました。

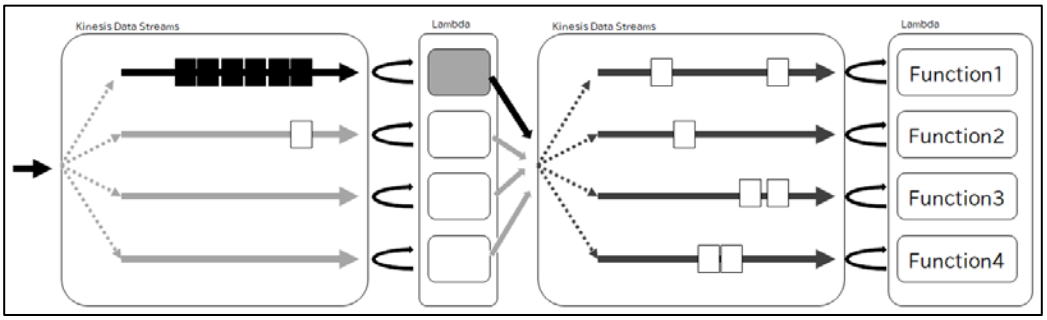
この SORACOM Funnel を利用して、少数のデバイスから大量のイベントを送信するようなシステムを作ったときに問題が起こりました。Kinesis Data Streams に送られたデータの処理が追いつかなくなったときに、シャード数をいくら増やしてもデバイス数までしかイベントが分散されず、また特定のデバイスからのデータが特に多かったことから、上記とまったく同じ状況に陥りました。

回避策として、重い処理を引き受ける別のストリームを作成し、SORACOM Funnel が投げ込むストリームからは別ストリームにパーティションキーを付け替えて投げ込むだけの軽量な処理だけをぶらさげました。

---

[7] Amazon Kinesis Data Streams、Azure Event Hubs だけでなく、Google Cloud Pub/Sub、AWS IoT や、ウイングアーク 1st 社 MotionBoard など様々なサービスに送信することができ、投げ込む際の認証情報なども管理してくれます。
SORACOM Funnel について詳しくは「めもおきば TechReport 2017.08 SORACOM 特集号」をどうぞ！
https://gumroad.com/l/memotr201708

[8] SIM 1 枚ごとに割り当てられている識別子



また、これと並行して SORACOM 社に要望を出し、IMSI の代わりにランダム値をパーティションキーとして使用するオプションが実装されることになりました。もちろん、これは同じデバイスからのイベントの順序関係が維持されなくなるというデメリットを抱えています。このケースではそれが許容できるものだったため、最終的にランダム値を利用するようになっています。

このように、メッセージングサービスを介して外部連携を行うパターンが、サーバーレスの普及と活用によって今後増えて行くと思われますが、そういった場合にはパーティションキーの選択を双方で上手く合意する必要があります。

監視

システム運用において監視は義務です。ですが、ストリーム処理の場合には、リアルタイム処理という見た目に反してほとんど全てのものが非同期システムの連携として動いているため、監視のポイントも従来のような同期型のシステムとは変わってきます。

結論から言ってしまうと、以下を監視すれば良さそうです。

- ストリームに溜まっている未処理イベントの遅延時間

これが最優先監視対象です。

Amazon Kinesis Data Streams であれば IteratorAgeMilliseconds というそのままずばりのメトリクス [9]が存在します。これは、ストリームから取り出した最後のイベントで、そのイベントがストリームに書き込まれた時刻と現在時刻の差分です。言い換えれば、イベントがストリームに入ってから出るまでの時間です。1秒毎に Lambda からポーリングされるため、滞りなく処理できている場合には1秒前後を推移します。

Azure Event Hubs には今のところそのようなメトリクスが用意されていないため、あらかじめイベントに発生時刻（あるいはクラウドへの到着時刻）を含めておき、Event Hubs から取り出した Function 側でイベントの発生時刻と現在時刻を比較する必要があります。

これが増加している場合、他のメトリクスを使って何が起きているかを判断していく必要があります。

[9] ストリーム側にも、Lambda の Event Source Mapping 側にも、どちらも存在します。

■ ストリームの流量

そもそもストリームデータの量が増えているのであれば、それを把握する必要があります。これは AWS も Azure どちらも普通に提供されているので素直に監視しましょう。

■ Function の処理時間

入ってくるストリームデータの量に大差ないのに処理待ちイベントが増えているのであれば、後段の Function の処理に時間が掛かっている可能性が高いです。

Function の平均実行時間だけでなく、最大値や 99%値、95%値なども見ると、「ほとんどは早く終わるけどたまに時間が掛かるイベントがある」「そもそもたまに失敗してタイムアウトまで掛かっている」などが判明します。

■ 一度に取り出したイベントの数

処理時間と合わせて見ておきたいのが、1 回のポーリングで取得したイベントの数です。

AWS も Azure もその最大値を設定できますが、不用意に低かったり高かったりすると割り当てられている CPU やメモリの効率が悪く性能を十分に発揮しきれないことがあります。そのため、Function に一度に渡されるイベントの数を上手く調整してあげることで、シャード数を増やさず[10]に処理性能を引き上げることが可能です。

At least once と向き合う

基本的に Pub/Sub メッセージングサービスは「At least once（少なくとも 1 回）」というモデルを採用しています。既に解説したとおり、読んで字のごとくですが、一つのイベントが少なくとも 1 回配信されます。

同じデータが重複することがある、ということを十分に設計に組み込む必要があります。

たとえば、イベント自体にランダムや連番などユニークとなるキーを含めておき、最終的に集計する直前でキャッシュ系データストアなどによって重複削除したり、より単純に連番で同じか古いものを無視したりといった対応が考えられます。

また Elasticsearch であればドキュメント ID を使うことで、同じデータが何度保存されてもデータが重複せず上書きされるようになります。DynamoDB 等を使う場合も同じようにキー設計をすることが可能です。

あるいは同じデータが複数回来てもサービス上問題がない用途であれば、データベースへの書き込み量などは増えたままになりますがそのまま気にせず処理してしまうという選択肢もあります。

このように「At least once」の性質をデータ処理全体の基本的な設計に組み込む必要があります。

---

[10] なにげに Kinesis Data Streams も 1 シャードあたり月 1500 円ほどと安くありません。

あとがき

今回はサーバーレスから久々に離れて新しいことに取り組んでそれを書こうと思っていたら、何故か気づいたらサーバーレスでストリーム処理をする話になっていました！ふしぎ！

ストリーミング SQL 一族がもっと幅広く使えるようになると、「単純なことをより単純に」実現できるようになると思っているので、もっとみんなで使っていきたいところです。あと、ストリーミング SQL と FaaS で両極端過ぎるので、Spark Streaming をフルマネージドしてくれる Azure Databricks には、ローエンド利用も含めて大きな期待をしています。サーバ建てたくないんじゃあああああああ！

あと Node-RED もストリーム処理エンジンの一種なのですが、私が詳しくないので省略しました。

メガクラウドのサーバーレス系サービスを使ったストリーム処理は、私みたいにこの分野に詳しくない人でも、ちょこっと短いコードを組み合わせるだけでかなり役に立つシステムが作れてしまうので、みんなもっと遊んでみると良いと思います。でも Kinesis 月 1500 円、最低でもシステムの数だけ必要になってしまうので、個人で遊ぶにはもうひと声欲しいのも確かです。Amazon SNS で頑張っても良いのですが、ちょっと違うんですよねえ。

冬コミ、受かっていれば、年末恒例の技術振り返り・予測と、コラム盛り盛りでお届けしたいです。

めもおきば TechReport 2018.10

発行日 2018 年 10 月 8 日 初版 技術書典 5

著者 Aki @nekoruri
aki@nekoruri.jp

発行 めもおきば
http://d.nekoruri.jp/

印刷 キンコーズ