Data engineering patterns on the cloud

A list of 84 cloud native patterns to solve common data engineering problems

waiting for{code}.com

Bartosz Konieczny

Data engineering patterns on the cloud

Bartosz Konieczny

Table of Contents

Copyright . . . 1
Introduction . . . 3
- Who is this book for? . . . 3
- Book organization . . . 4
- Conventions used in the book . . . 5
- Contact . . . 5

Data processing . . . 6
- Batching . . . 7
- Consumer groups . . . 12
- Cooldown period . . . 15
- Data locality . . . 17
- Dead-Letter sink . . . 21
- Ephemeral cluster . . . 25
- Event-driven serverless worker . . . 28
- Exactly-once delivery . . . 31
- Micro-batch processing . . . 35
- No Code data processing . . . 38
- NoSQL Change Data Capture . . . 41
- Ordered delivery . . . 45
- Parallel data upload . . . 50
- Pull processing . . . 53
- Push processing . . . 56
- Reprocessing data from a streaming data source . . . 60
- Reusable job template . . . 64
- Serverless SQL worker . . . 67
- Shuffle service . . . 70
- Single-tenant client . . . 73
- Small data - batch services . . . 76
- Small data - single-node cluster . . . 79
- Streaming serverless SQL processing . . . 81
- Targeted data retrieval from files . . . 84

About the author . . . 87
Index . . . 88

Copyright

Data engineering patterns on the cloud by Bartosz Konieczny waitingforcode.

Published by Bartosz Konieczny waitingforcode.

waitingforcode.com

Cover by Jakub Kędziora.

To Sylwia and Maja
Love you ♡

Introduction

I've started my cloud data engineering journey with a data ingestion project on AW. It turns out, it was also my first contact with data engineering overall, so I felt very excited about this completely new branch. With my last commits to the project's repository, after discovering all these new "data" things, I knew I wanted to settle my career in this field for a while. So I started to look for a new data challenge and found one on GCP.

Although the scope of the services was more limited there, I tried to make some effort and see the ones I wasn't supposed to work with. That's when I had an *aha* moment and I found a lot of similarities with my previous project. From that moment, I've started to consider cloud services in terms of similarities.

That's also how the idea of writing this book was born. Why have I called it with the "patterns" keyword? The concepts described here are present in at least 2 cloud providers, so they're something more general that you can use to solve the problem while moving from one cloud project to another. They're more models than cloud provider-specific features.

Who is this book for?

If you're a data engineer already working with a cloud provider, the book will help extend your cloud knowledge. In your situation, you can learn by analogy between the concepts you know and the ones you want to discover.

If you are a data engineer with an on-premise background and feel comfortable with general data concepts like partitioning or horizontal scaling, you will also find something interesting in the book. The book should be for you a shortcut to map your on-premise knowledge to the flexible world of the cloud.

If you are an aspiring data engineer, the book's content can help you structure and extend your knowledge to the most up-to-date cloud technologies.

If you are a data scientist or data analyst, the content of the book can help you build better data products. Although it targets common data engineering problems, you can leverage this knowledge to innovate and improve your out-of-the-box thinking in your analytic or ML projects.

Finally, if you're a backend software engineer already working on the cloud, you should find something interesting for the services usually shared between data and software engineers, such as object stores, streaming brokers, or NoSQL databases. Maybe you'll discover alternative ways to write your programs?

Even though it's not forbidden to read the book if you aren't one of the following persons, the content can be too difficult to apprehend:

- You want to learn data engineering. The book is not a step-by-step guide. Each pattern details the implementation on different cloud providers, but it doesn't teach data engineering basics.
- You want to learn cloud computing. As above, the book isn't organized as an online course to move you from the "I don't know cloud computing" to the "I know cloud computing" state.

- You are closer to business than technology. Although the book can give you a better picture of what can be done on the cloud, you'll probably be bored with the technical details.

Book organization

The book organizes patterns in 7 categories presented in a separate chapter each. Although it was quite obvious to classify most of them, some were good fit for 2 different categories. For example, the row-level security pattern is present in the Data Warehouse category, but it could also be put into the Data Security section. Nonetheless, most of the patterns are not confusing, and the category-based organization will help you find the ones you're looking for!

Here is the list of the chapters:

- Data processing
- Data storage
- Data security
- Data warehouse
- Data management
- Data orchestration
- Data transfer

Pattern page

Each page starts with a short ***Description*** of the solved problem. Naturally, it might not be a single example, but the purpose of this part is to give a working context rather than share all addressed issues.

After that part, you'll find a ***List of steps*** required to set up the pattern. It's there to give an overall idea of the working details. The list is meant to be generic, so there may be some implementation differences from one cloud provider to another.

Just below the list, you'll find a ***Use for*** section with an extended list of use cases and a ***Look at*** section with the list of warnings regarding the pattern. The goal is to extend the implementation scenario given in the description part and share some attention points to consider before using the pattern.

In the last part, you will see a ***Generic schema design*** and implementation details under the ***Cloud details...*** sections. Each implementation detail will provide you a bunch of links to the cloud provider's documentation where you can find more details if needed.

Conflicts resolution

Some of the cloud services are present in all of the analyzed cloud providers. To avoid repeating them in the ***Cloud details...*** section, you will find them presented only once, always under the same cloud provider. So far it's only the case of Databricks assigned in the book to Azure.

Conventions used in the book

`Fixed width` is used to distinguish technical concepts, configuration properties, code properties, or cloud services names.

Link is a hyperlink leading to the cloud provider documentation, any other resource describing the pattern, or another pattern of the book.

Any quote includes the reference link and is presented as follows:

> Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna...
>
> — https://localhost

Contact

You can address any comments and questions concerning the book to the author at book@waitingforcode.com .

The book updates, errata, and any additional information, please access Updates page

Data processing

Batching

Problem

Your data consumers and producers encounter unexpected latency. You didn't detect any coding issues but found out an unexpected network latency because of the consumers and producers working on individual records at a time.

Use the batching pattern to reduce the number of request-response loops by putting more operations to do in a single request.

For example, a producer client application could put 20 records to write on the server and send them in a single request instead of doing that 20 times (1 record per request).

Mechanisms

For the writer:

1. Configure the batching threshold. It can be expressed as a minimal number of items or a maximal buffering time. Some producers can use both conditions simultaneously and trigger the writing whichever condition comes first.
2. Start the writer that will buffer the records instead of sending them one by one to the data store.
3. The writer communicates with the data store only when the buffering condition from the first point is met.
4. The data store handles the request and generates a response. The response contains the outcome for every operation. The writing action can succeed fully or partially.
5. The writer interprets the response and can retry the delivery of the failed records.

For the reader:

1. Get ids of all items to read. Include the ids in the reading request or query.
2. Send the request to the data store.
3. The data store responds with the list of items corresponding to the ids. The response can be partial, so include missing data. This could happen because of the throttling errors.
4. The reader interprets the response and eventually retries the read operation for the empty ids if the not-matching reason was a temporary error on the server side.

Use for

Reducing the latency in the data enrichment scenario

Fetching multiple items in a single request should be much more efficient than querying an element at a time. This operation reduces the network connection overhead by opening and closing it only once for a bunch of records.

Optimizing storage I/O

Some data stores have an immutable storage that is optimized for large writes. It means that any

insert or update will create a new storage space instead of replacing the existing one. Sending multiple items at once can help optimize the I/O operations on the data store level by reducing the storage management overhead.

Look at

Atomicity

It can be difficult to implement atomic writes, i.e., the all-or-nothing outcome. Even though the response returns a list of successful and failed writes, there is no guarantee that retrying failed records delivery will succeed.

Risk of losing data on the writer's side

Often, the writer will first buffer the events in the memory before sending the batch request. In case of an unexpected runtime failure, the buffered records will be lost.

At-least once delivery

If the producer API only supports retrying the whole buffer and there is no deduplication mechanism in the data store, the writing risks generating duplicates in case of retries.

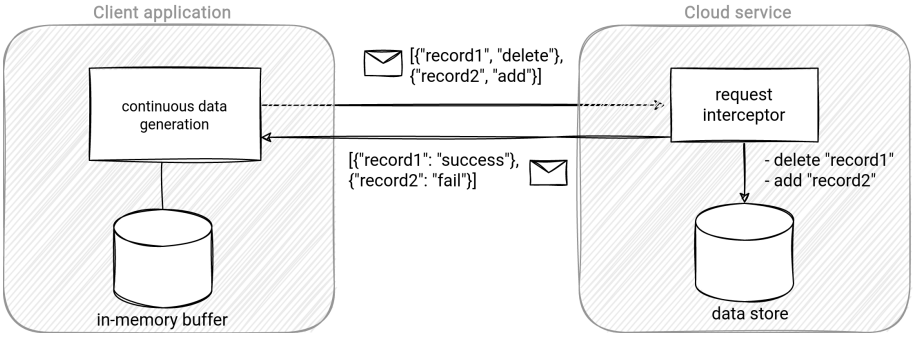
Increased latency

The pattern adds some latency due to the buffering. It might not be the best fit for the request-response scenarios where the client expects to get almost an immediate response.

Batch limits

Batch requests can have limits, such as the number of elements or the request size. Ensure that the API supports this limits natively and flushes the buffer after reading them. Writing the custom code for that may be tricky.

Schema design



Cloud details AWS

Kinesis Data Streams

The `Kinesis Producer Library` (KPL) implements the pattern for data generation. A KPL producer without batching sends each record in an individual HTTP request. On the other hand, a batch-based producer accumulates multiple records in memory and delivers them in a single HTTP call.

The library provides 2 flavors of batching:

- Aggregation. Here the producer writes the buffered records as a single Kinesis Data Streams record.
- Collection. Rather than putting all data to a single record, the producer keeps the records separately and sends them all from a single `PutRecords` operation.

Links:

1. https://docs.aws.amazon.com/streams/latest/dev/kinesis-kpl-concepts.html#kinesis-kpl-concepts-batching

DynamoDB

The DynamoDB client supports writing and reading batch operations.

On the reading side, the `BatchGetItem` operation can get up to 16MB of data and 100 different items in a single call using the primary keys of the records. To reduce the size of the returned elements, you can explicitly define the table attributes to include in the response.

The writing part uses the `BatchWriteItem` operation. A single request supports up to 16MB of data and 25 `put` or `delete` operations that are not atomic. Each individual `put` or `delete` action is atomic, but all of them combined in the same request aren't. Some of them can fail, for example, due to the insufficient table capacity.

Links:

1. https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchGetItem.html
2. https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchWriteItem.html

Kinesis Firehose

This time, the batching implementation is made at the service level instead of the SDK. `Kinesis Firehose` synchronizes the data in-motion with the data-at-rest databases, like for example, Kinesis Data Streams records with an S3 bucket.

Throughput optimization consists of buffering the data and triggering the physical writing action only when the buffering lasts too long or when the buffer is full.

Links:

1. https://docs.aws.amazon.com/firehose/latest/dev/create-configure.html
2. https://docs.aws.amazon.com/firehose/latest/dev/limits.html

Cloud details Azure

Event Hubs

On the Azure side, the Event Hubs producer library supports 2 data generation modes, a record-based or a `batch-based`, which implements the pattern.

The batch-based operation accumulates multiple `EventData` items and delivers them to the broker in

a single operation as an `EventDataBatch` abstraction. The process shares the limits of a single event generation, so 1 MB/second or 1000 events/second.

Links:

1. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-quotas
2. https://github.com/Azure/azure-sdk-for-net/blob/master/sdk/eventhub/Azure.Messaging.EventHubs/README.md#publish-events-to-an-event-hub
3. https://docs.microsoft.com/en-us/javascript/api/@azure/event-hubs/eventhubproducerclient?view=azure-node-latest#createBatch_CreateBatchOptions_
4. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-features

Cosmos DB

When it comes to the data stores, Cosmos DB client libraries support bulk read and write operations. For example, the .NET SDK has a flag enabling or disabling the bulk reads and writes, called `AllowBulkExecution`. If enabled, the SDK groups all operations by physical partition affinity and sends them as batches to the service.

Links:

1. https://devblogs.microsoft.com/cosmosdb/introducing-bulk-support-in-the-net-sdk/

Event Hubs Capture

As AWS for the streaming-to-at-rest storage synchronization scenario, Azure also implements batching directly at the service level. `Event Hubs Capture` is Azure's version of Firehose used to save streaming data in Azure Storage.

The buffer is fully configurable and supports:

- `time window` which defines how long the service can buffer the events to write
- `size window` which sets the maximal size of the buffer

Links:

1. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-capture-enable-through-portal

Cloud details GCP

Pub/Sub

GCP streaming services also support batched data generation. The Pub/Sub client supports batching based on 3 settings:

- `max_messages` - max number of buffered messages
- `max_bytes` - max size of the buffered messages

- `max_latency` - max buffer time

Whenever the writer reaches one of these thresholds, it issues a write request to the service.

Links:

1. https://cloud.google.com/pubsub/docs/publisher#batching

Dataflow

Dataflow is a data processing service that you can use to transform or copy data between 2 data stores by writing some custom code or using Reusable job template pattern.

For example, a job template for streaming data from Pub/Sub to Splunk supports a parameter called `batchCount`. It defines the number of elements delivered at once to the Splunk endpoint. The same parameter also exists for the Pub/Sub to MongoDB synchronization. Besides the number of records, this one also supports the max size of each batch with the `batchSizeBytes` setting.

Links:

1. https://cloud.google.com/dataflow/docs/guides/templates/provided-streaming#cloudpubsubtoavro

BigTable

Another GCP service with batch operations is BigTable. The batching API is transparent to the end user, though. It simply consists of calling `read rows` operation instead of the `read row` for reading, and `mutate rows` instead of `mutate row` for writing.

Links:

1. https://cloud.google.com/bigtable/docs/writing-data
2. https://cloud.google.com/bigtable/docs/reading-data

Consumer groups

Problem

Your small data project has grown and now requires Big Data processing. To keep up with the incoming data, you decided to increase the number of partitions in the streaming broker. You're now looking for a solution to create a dedicated consumer for each partition.

Mechanisms

1. Scale the partitions of the data source (check Data layout - partitions pattern).
2. Set a consumer group property in your data reader application.
3. Package the consumer application.
4. Deploy the package to as many compute nodes as needed. It'll create at least one consumer per partition and automatically load balance the partitions assignment when a new consumer instance joins the group.

Use for

Horizontal scaling

A very popular solution to handle an extra load on the streaming broker side is adding new parallelization units, such as partitions. However, it doesn't scale the consumers automatically. You have to deploy new consumer instances and use the same consumer group id to share the extra capacity.

Automatic data processing parallelization

Although it's technically possible, assigning new consumers instances to the new parallelization units manually requires an additional operationalization overhead. You need to track the current assignments and also deal with the failures to reassign the workload of the failed consumer. The pattern automates this part and delegates it to the service or the SDK.

Look at

Stragglers

The consumer instances can process data at different latencies and some partitions can be late regarding the others. The lateness can be due to some coding issues or unexpected hardware problems of the consumer's node (slower disk writes, etc.). That's why you shouldn't assume the same processing time for each consumer.

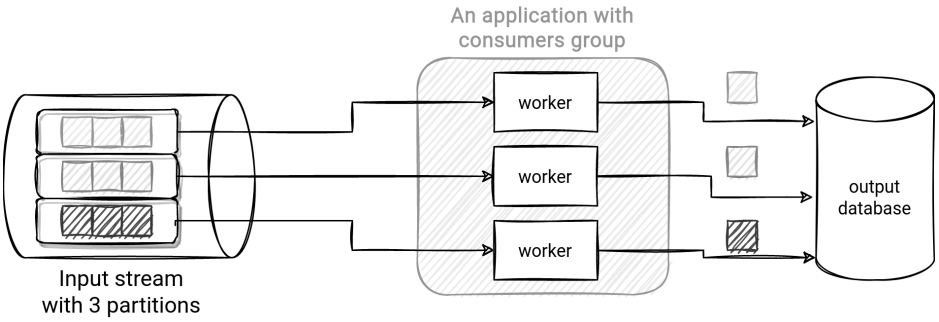
Alive resources

Depending on the implementation, you may need to monitor the number of partitions and decommission idle nodes when the data source scales down, or add new servers when it scales up. It probably won't be required for an Event-driven serverless worker but should be kept in mind for the cluster-based solution.

Rebalancing

Adding new consumers to the group can break stateful semantics if your application keeps some local state in each consumer instance. The new consumers might start processing data from the partitions previously associated elsewhere, and they won't have the state data accumulated so far.

Schema design



Cloud details AWS

Kinesis Data Streams

To implement the pattern in Kinesis you must use the `Kinesis Client Library` (KCL) and define the `application name` attribute. Each worker instance with the same application name will consume one or many disjunct shards of the input stream. The service will also use the application name to create a DynamoDB tracking table with the read positions for all the shards.

Links:

1. https://docs.aws.amazon.com/streams/latest/dev/kinesis-record-processor-implementation-app-java.html

Cloud details Azure

Event Hubs

Event Hubs implements the pattern as a separate component because you create the consumer group explicitly with the CLI `az eventhubs eventhub consumer-group create` command or any official SDK library. The command creates a resource that you associate later with the deployed consumers. The service relies on this association to link a partition to one consumer from the group.

Although it's technically possible to have 5 active consumers from the same consumer group reading one partition, Azure recommends keeping only 1 active consumer per partition. Multiple consumers per partition are more complicated to implement because they process the same messages, and the implementation must take care of the deduplication.

Links:

1. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-features#consumer-groups
2. https://docs.microsoft.com/en-us/cli/azure/eventhubs/eventhub/consumer-group?view=azure-

cli-latest

Cloud details GCP

Pub/Sub

Pub/Sub service implements the pattern by exposing the data from the abstraction called `subscription`. The subscription is a proxy between the consumers and the topic, so all consumers connected to the same subscription share the disjunct messages written to the topic.

Inside each subscription-based consumer group, there can be at most one consumer processing a given message in a given moment. However, the subscription doesn't guarantee exactly-once processing. If the consumer doesn't notify the service about the successful outcome of the processing on time, Pub/Sub will attempt to redeliver it to another consumer instance of the group.

Links:

1. https://cloud.google.com/pubsub/docs/subscriber#at-least-once-delivery

Cooldown period

Problem

The cluster running your batch jobs scales up and down very often in short periods of time. You analyzed the situation and concluded that some of these actions were unnecessary. They happened because the service hadn't had enough time to distribute the work among the new resources. Now, you're looking for a way to give some time for allocating the compute resources to the job.

Mechanisms

1. Configure the scaling metrics for your cluster, like CPU, memory, or disk usage.
2. Define the threshold for each metric that will trigger the scaling action.
3. Set the cooldown period. It's the time interval that should pass between 2 consecutive scaling actions.

Use for

Relevant scaling

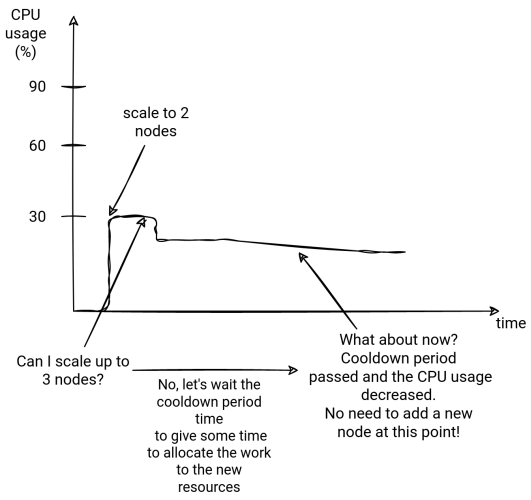
Without the pattern, the cluster manager might not have enough time to redistribute the workload and emit improved scaling metrics. Consequently, the service might continue to add new nodes that will be removed a few minutes later once the workload stabilizes after the first scale-up.

Look at

Value

It's not easy to figure out an accurate cooldown period. Too short value will lead to unnecessary scaling actions, whereas too long will impact the job latency because it won't scale at a good time. Maybe, it'll be necessary to try different configurations before finding the right one.

Schema design



Cloud details AWS

EMR

Under-the-hood, EMR cluster nodes are EC2 instances. Therefore, the EC2 scaling policy configuration is also valid for EMR.

A scaling policy configures the scalability of the cluster by setting the rules composed of the following attributes:

- a name that identifies the rule easier
- a scaling adjustment that defines the number of instances to add (scale-out rule) or terminate (scale-in rule)
- a CloudWatch metric that defines the scaling conditions
- an evaluation period that activates the scaling action only if the CloudWatch metric evaluates to true during that period
- and finally, a `cooldown period` defining when the next scaling action can start

Links:

1. https://docs.aws.amazon.com/autoscaling/ec2/userguide/Cooldown.html
2. https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-automatic-scaling.html

Cloud details GCP

Dataproc

Dataproc manages the autoscaling in the `autoscaling policy`. Each policy defines the following attributes:

- the min and max number of worker and secondary worker instances
- the scaling algorithm parameterized the:
  - scale up factor - it configures the percentage of missing resources added in a single scaling action. For example, if the tasks require an extra 100GB of memory and the scale up factor is set to 0.5 (50%), the scaling action will add an extra 50GB to the cluster.
  - scale down factor - the opposite for the scaling up factor is the `scaling down`. If it's set to 0%, it means that the service will never take back the added resources.
  - `cooldown period` - the cluster will scale up and down more often if it's too short. But it might happen that some of these actions won't be necessary because the cluster won't have enough time to distribute the workload and improve the scaling metrics.

Links:

1. https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/autoscaling#picking_a_cooldown_period

Data locality

Problem

Your new company uses cloud resources from a European region. All jobs are running fine except one that struggles to meet the SLA. After analyzing the monitoring metrics, you noticed that the job copies considerable amounts of data from a different geographical area. You want to avoid this unnecessary operation.

Mechanisms

1. Identify the region where your input data resides.
2. Deploy the job in the same region as the data.

Use for

Execution time optimization

The job can access the data within the same geographical area. Consequently, there is no overhead copying the data from a different geographical location.

Ad-hoc querying optimization

Colocating the serving component with the data can optimize the ad-hoc querying or data exposition scenarios.

Cost reduction

Moving data between regions is an extra cost. Eliminating it or reducing the amount of data to process by performing some pre-aggregation or filtering can help reduce the bill at the end of the month.

Regulatory compliance

In some scenarios, the data may not be authorized to leave a particular geographical area. It can be the case of PII data. You might then be obligated to run the job in the same geographical location.

Look at

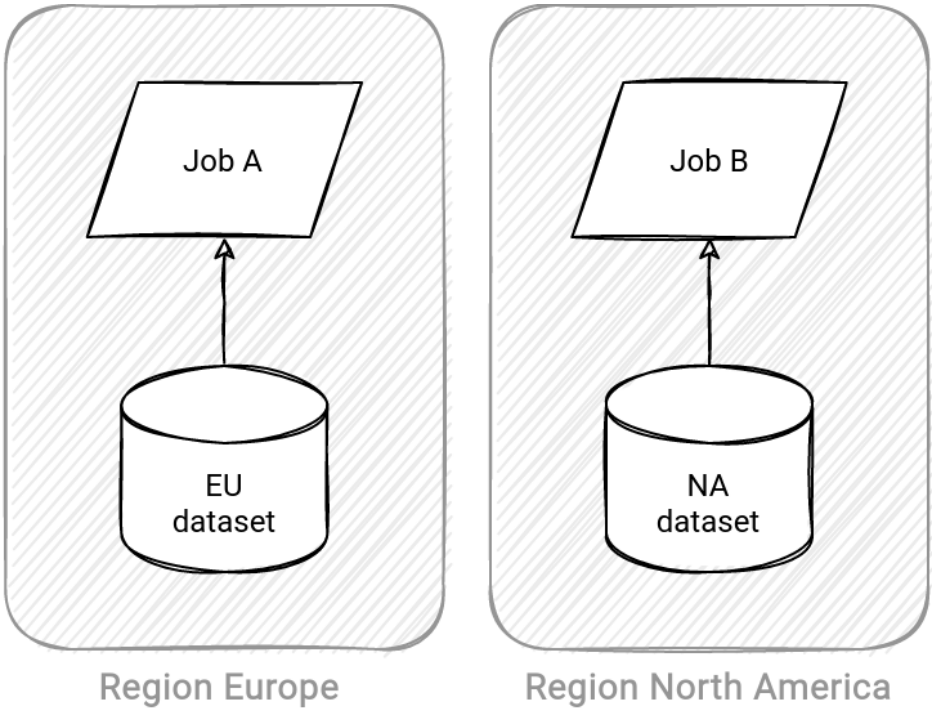
Orchestration effort

The collocation of storage and compute can save costs and optimize execution. However, it can also make your system more complex. For example, writing a job aggregating data across the regional datasets will require an extra orchestration of the partially generated regional aggregates.

Job logic

It won't always be possible to have region-based job instances. In that case, data transfer will still be necessary.

Schema design



Cloud details AWS

EMR

Under the hood, the EMR service uses EC2 instances that are physical nodes living inside AWS regions. Collocating them with the data accelerates the processing time and avoids cross-region data transfer fees.

To set the EMR region, you can use the CLI's `AWS_DEFAULT_REGION` environment variable or the `setEndpoint` method of the SDK.

Links:

1. https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-region.html
2. https://aws.amazon.com/s3/pricing/

Redshift

The pattern doesn't only apply to the pure compute services such as EMR. It's also valid for less compute-intensive operations like the `COPY` command used to load data from S3 to Redshift.

By default, the command assumes that the data resides in the same region as the cluster. If this assumption is not true, you've to set the dataset region in the `REGION AS` attribute. As per documentation, you can incur additional charges:

> Transferring data across Regions incurs additional charges against the Amazon S3 bucket or the DynamoDB table that contains the data.
>
> — COPY from Amazon S3

Links:

1. https://docs.aws.amazon.com/redshift/latest/dg/copy-parameters-data-source-s3.html

Cloud details Azure

Cosmos DB

Cosmos DB is a good example of the pattern used to reduce data serving latency. It's implemented as a fully managed replication process that minimizes the network traffic overhead by bringing the data close to the user's region.

Besides the reading, the database also supports multiple regions for the write operation. However, in this feature you might need to solve an additional concurrency problem of multiple clients working on the same item. To handle it, you must define the `conflict resolution policy` at the container's creation time as:

- The last-writer-wins policy that will accept the most recent operation. The conflict resolution is based on the `_ts` property or a timestamp field from the data.
- A custom policy that is more flexible but requires extra coding effort. In this policy, you could for example, give a preference to the delete operations over the updates from a stored procedure, or resolve the conflicts later by consuming a conflicts feed.

Links:

1. https://docs.microsoft.com/en-us/azure/cosmos-db/distribute-data-globally
2. https://docs.microsoft.com/en-us/azure/cosmos-db/sql/how-to-manage-conflicts

Synapse

As for Redshift, collocating data with the Synapse SQL pool is a recommended strategy to minimize the latency for the loaded data.

Links:

1. https://docs.microsoft.com/en-gb/azure/synapse-analytics/sql/data-loading-best-practices

Cloud details GCP

BigQuery

Although the data locality pattern in BigQuery optimizes the costs and performance, it can also act as a guard and block some invalid operations, such as:

- querying external data sources from a different location
- loading data from a different region than the table
- extracting data to the bucket in a different region than the table

To prevent runtime errors for the 3 scenarios above, you can set up a `Resource Location`

`Restriction` policy and allow resource creation in only some regions.

Links:

1. https://cloud.google.com/bigquery/docs/locations
2. https://cloud.google.com/resource-manager/docs/organization-policy/defining-locations

Dead-Letter sink

Problem

You came back to work after a well deserved weekend. The first thing you saw was a list of errors. After the first analysis you found that the streaming pipeline failed on Friday night. After checking its logs, you discovered that the failure was due to a single unprocessable message (aka *poisson pill message*). You were asked to fix the issue and keep the unprocessable messages aside for further investigation, without perturbing the execution of the job.

Mechanisms

1. Create a data store that will persist the poison pill messages.
2. Set the created data store as the dead-letter sink in the job configuration. The configuration can be defined in the code or at the service level.
3. Start the job.
4. Monitor the dead-letter sink. You should know when the number of unprocessable records goes too high. It's an indicator for the global health of the job.

Use for

Unprocessable records storage

Dead-letter sink is the place where the data consumers send all unprocessable records. It's a different solution than logging the errors in the pipeline with a `try-catch` construction because it's a queryable place. It can then be easy to know how many errors occurred recently and even to replay them.

Temporarily buffer unprocessable records

Data in the dead-letter sink can be bad and not usable, such as badly formatted JSON records. But the sink can also temporarily store invalid records, for example, if they contain a new attribute that was unknown by the job. After integrating this attribute to the job, it's possible to reintegrate the failed records to the main pipeline.

Ignoring errors in different semantics

Although the example discusses the streaming scenario, the pattern also works for batch pipelines.

Look at

Extra workload

Dead-Letter sink is an intrinsic part of your system. It has the same maintenance and monitoring requirements. Otherwise, it may hide the system's real (bad) state or cost you money if you decide to store the unprocessable records forever and do nothing with them.

Coding effort

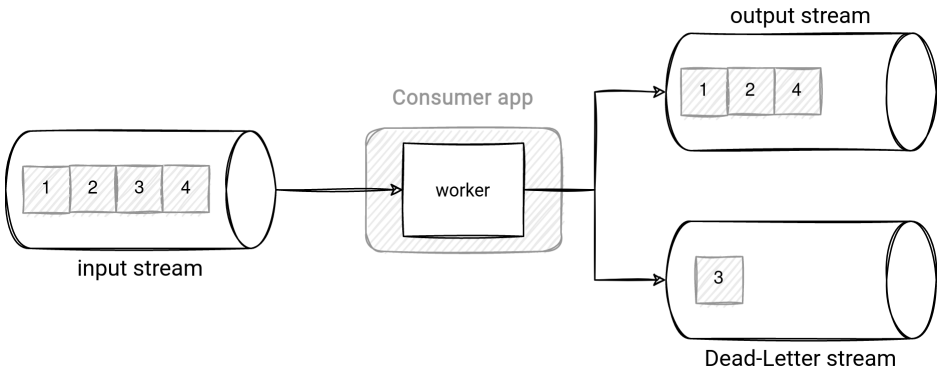
Depending on the type of service, it can require some coding effort. Sometimes, the dead letter

storage is natively exposed from an API. Other times, it must be implemented directly as a custom code part.

Silver bullet

Use the pattern carefully. If none of the input records is processed correctly, the dead-letter sink won't solve anything. It can help spot this problem with appropriate monitoring and alerting but won't fix the data quality issues that can be due to other technical or organizational reasons.

Schema design



Cloud details AWS

Lambda

You can configure a Lambda function to retry a limited number of times in case of failures. After reaching this retries limit, it can either skip the failing record(s) or send them to the configured `on-failure destination` (Dead-Letter Sink pattern).

Besides this relatively new on-failure destination, AWS Lambda also has a Dead-Letter Queue implementation available since 2016. However, AWS recommends using the on-failure destination because it provides a better failure context, including the exception stack trace.

Links:

1. https://aws.amazon.com/blogs/compute/new-aws-lambda-controls-for-stream-processing-and-asynchronous-invocations/
2. https://aws.amazon.com/blogs/compute/introducing-aws-lambda-destinations/
3. https://aws.amazon.com/about-aws/whats-new/2016/12/aws-lambda-supports-dead-letter-queues/

SQS

This message queue service implements the dead-letter pattern with a `redrive policy`. The policy has a property called `maxReceiveCount` to define the maximal number of delivery attempts of a message. Upon reaching this number of retries, the service considers the message as unprocessable and moves it to the dead-letter queue.

It's worth mentioning SQS queue doesn't create the dead-letter queue automatically. You will need to create and configure it aside.

Links:

1. https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-dead-letter-queues.html
2. https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-configure-dead-letter-queue.html

Cloud details Azure

Service Bus

Service Bus queue and topic subscriptions provide a fully managed secondary subqueue called `dead-letter queue`. It's the data store for the messages that cannot be delivered or processed.

There are 2 ways to send erroneous messages to this dead-letter queue. The first method is managed by the service itself. Service Bus marks a message as dead-lettered when:

- it expired
- its session id is `null`
- it reached the allowed number of forwarding actions between queues
- it wasn't correctly processed within the configured retries number
- its header is too big

In addition to these service-managed criteria, a consumer application also can redirect any message to the dead-letter queue at the code level. The client can define here the message payload, the dead-lettering reason, and the error description.

Links:

1. https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dead-letter-queues
2. https://azuresdkdocs.blob.core.windows.net/$web/python/azure-servicebus/latest/azure.servicebus.html?highlight=dead%20letter#azure.servicebus.ServiceBusReceiver.dead_letter_message

Event Grid

Another Azure service supporting the dead-letter pattern is `Event Grid`. It can dead-letter a message when it cannot be successfully delivered within the TTL period or when the number of delivery attempts reaches the allowed number of retries. When one of these conditions happens, the service writes the undelivered event to a Blob created in the Storage Account.

The dead-letter feature is disabled by default.

Links:

1. https://docs.microsoft.com/en-us/azure/event-grid/delivery-and-retry#dead-letter-events
2. https://docs.microsoft.com/en-us/azure/event-grid/manage-event-delivery

Cloud details GCP

Pub/Sub

Pub/Sub supports the dead-letter pattern at the subscription level with an explicitly configured dead-letter topic.

How does it work? If a consumer doesn't acknowledge a message within the maximum delivery attempts (default is 5), the service automatically forwards this message to the configured dead-letter topic.

With the dead-letter topic enabled, each processed message has a metadata field called `delivery_attempt`. It defines how often Pub/Sub tried to deliver this message to a subscriber.

Links:

1. https://cloud.google.com/pubsub/docs/handling-failures
2. https://cloud.google.com/pubsub/docs/reference/rpc/google.pubsub.v1#google.pubsub.v1.ReceivedMessage

Dataflow

The dead-letter pattern also exists in Dataflow. The service defines the job logic with the Apache Beam API and it leverages the API's features to implement the dead-letter pattern.

The first implementation adds a `side output` to the code. The idea is to wrap the processing logic with a `try-catch` block and emit any incorrectly processed record to a separate `PCollection` in the `catch` part. This extra PCollection can be later written to any data store of your choice.

The second approach uses a native dead-letter support in `IO sinks`. For example, the `BigQueryIO` has a method called `getFailedInsertsWithErr()`. It returns all rows that the job didn't succeed to insert into the output table. As for the side output approach, you can take these records and write them to the dead-letter location.

Links:

1. https://beam.apache.org/documentation/programming-guide/
2. https://beam.apache.org/documentation/patterns/bigqueryio/

Ephemeral cluster

Problem

You performed the on-premise-to-cloud migration quickly, without making any paradigm shift. Consequently, your batch pipelines are still running on a big data processing multi-tenant cluster that ended up being the most expensive component of your infrastructure. Moreover, the shared cluster doesn't improve the predictability of the processing because the jobs randomly fail and disturb the work of the unrelated tasks. You want to optimize it and use a more cloud-native solution.

Mechanisms

1. Create a dedicated cluster for each job.
2. Wait for the cluster to be created.
3. Submit the job to the created cluster.
4. Wait for the job to be terminated.
5. Destroy the cluster. This step is optional if the service has an auto-terminate feature that destroys the cluster when there are no more running jobs.

Use for

Cost savings

Unlike a long-running cluster, the ephemeral one is scoped to the job execution time. If the job runs only a few hours a day, you don't need to pay for the whole day.

Flexible configuration

Some jobs may require a different hardware configuration. For example, a heavy-compute logic will need more CPU than a memory-intensive logic. Thanks to this isolated setup, you can easily allocate the resources per job purpose.

Separation of concerns

Each job has its dedicated cluster resources. Therefore, if the job makes a node unhealthy for any reason, this action won't impact other jobs since they're physically separated.

Look at

Execution context loss

Depending on the configuration, you may lose some execution context. If the service implementing the pattern doesn't provide logs and metrics persistence, it'll be hard to investigate the issues after the cluster shutdown. It's also valid for any data stored on the nodes. They're often reused for other clients, so need to be cleaned beforehand.

Effort shift

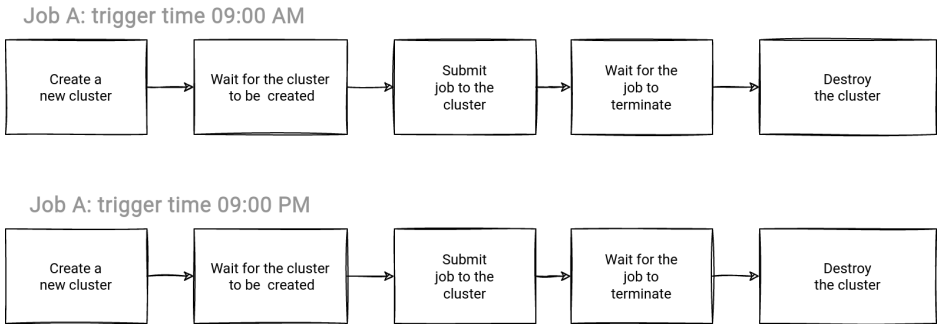
Instead of relying on the assumption of an always-up cluster, you have now to deal with the missing compute environment. During each job scheduling interval, the data orchestration layer

will need to create and destroy the cluster.

Tracking

Instead of having a single monolith component to monitor, you'll have multiple small mini-component stacks. To enable this tracking, it's then important to easily identify the used resources. You can use the Data annotation for this.

Schema design



Cloud details AWS

EMR

EMR implements the pattern with ephemeral clusters, also known as the `transient clusters`.

A transient cluster works in:

- A fully controlled configuration. It requires end-to-end management. It means that you will need to create the cluster, submit the job, wait for the job to terminate, and do the clean up in the end.
- A partially controlled configuration. Here, your responsibility stops at the job submission step. When the job terminates, the service automatically destroys the cluster. To enable this option, you have to specify the `--auto-terminate` flag while creating the cluster.

Links:

1. https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-longrunning-transient.html
2. https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-termination.html

Cloud details Azure

Databricks

When it comes to the pattern's implementation, Databricks also has 2 strategies. The first one uses a fully controlled create-submit-destroy workflow. The cluster can perform self-destruction if the `auto-termination` field is set.

But it's appropriate for the General clusters. Additionally, Databricks comes with a fully managed `Job cluster` type where the cluster lifecycle is tied to the jobs executed on top of it. Databricks will

then take care of the create, submit, and destroy steps of the pattern.

Links:

1. https://docs.microsoft.com/en-us/azure/databricks/clusters/
2. https://docs.microsoft.com/en-us/azure/databricks/clusters/create

Cloud details GCP

Dataproc

The ephemeral clusters pattern also applies to the Dataproc service. The Dataproc is similar to AWS EMR, so is the technique of creating ephemeral clusters.

As for EMR, you can manage the ephemeral clusters with the help of a data orchestrator that will be responsible for managing the cluster lifecycle, including the termination action. Besides, you can also delegate the responsibility of destroying the cluster to the service by defining the max allowed idle time or by using `Dataproc Workflow Templates`.

The `Dataproc Workflow Templates` feature defines a set of jobs (`workflow`) to execute on a Dataproc cluster. Under-the-hood, it'll create an ephemeral cluster for you and destroy it after executing the last job.

Links:

1. https://cloud.google.com/dataproc/docs/concepts/workflows/overview
2. https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/scheduled-deletion

Event-driven serverless worker

Problem

One of your data processing workloads has an unpredictable triggering pattern. The data to process isn't available at the same time every day. You want to implement an efficient data processing logic that won't waste any resources and run only when the data is present. Your data source supports an event-driven trigger.

Mechanisms

1. Write the data processing code and package it in the format expected by the serverless service.
2. Define the event triggering the job and the runtime resources (vCPU, memory).
3. Deploy the job with the trigger to the serverless service.

Use for

Unpredictable data processing

An inefficient solution to process irregularly generated dataset will continuously check for the data to process. It wastes resources and money, especially when there is a big gap between data generation times. If the input data store supports event-driven triggers, they are a more efficient alternative running the job only when there is data to process.

Automatically scalable workloads

The service creates one data processing instance for each input item unless configured differently. It can then seamlessly scale with the increased or decreased load.

Processing without ops skills

With the solution, you can only focus on the data processing logic. You don't have to configure the cluster or set up complex orchestration logic. The service simply triggers your data processing function once the data is available.

Look at

Complex data processing logic

Although the pattern simplifies the triggering part, data processing logic can be more complex to process. Let's take the example of an aggregation job that needs to count the number of lines per user. The job reads 3 files generated at an irregular interval but quite close to each other. Due to the event-driven character, the pattern will create and run 3 independent job instances, making the in-memory aggregation impossible. To support this more complex use case, you will require an extra aggregation layer.

Cloud locking

The event triggers are often limited to fully managed cloud services. It may not be possible to use a virtual machine-backed database or migrate the workload easily to another cloud provider.

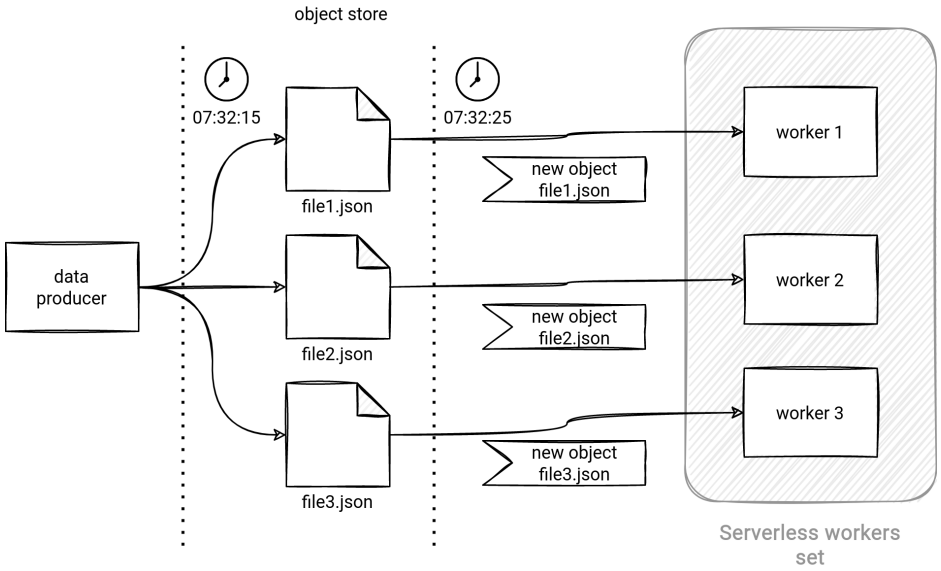
Monitoring effort

It's easier to define the alerting for a predictable data processing workload. For example, we can easily trigger an alert for the input dataset availability. It's much harder for unpredictable workloads because the dataset generation is irregular by nature.

Data sources

Not all cloud data stores are event-driven. The pattern can not work for some NoSQL databases or data warehouses.

Schema design



Cloud details AWS

Lambda

AWS Lambda is a serverless service that runs the function in the response for a data-related event, such as S3 object creation or DynamoDB Streams record writing.

The service creates a function instance when it gets notified about the new data to process. The notification comes from the supported event-based data stores, such as messaging technologies (Kinesis, MSK, SQS) and data-at-rest stores (DynamoDB Streams and S3). Depending on the data source characteristic, the function can behave as:

- An auto-scalable streaming application. For Kinesis Data Streams, the number of instances equals the number of shards in the stream. It's the main scaling unit. Additionally, you can also set a parallelism level to split shard data into multiple instances.
- NoSQL Change Data Capture. In this mode, the function continuously processes all changes made on the table. The data source drives the "continuous" character here. The function doesn't run if no changes are made to the table.

Links:

1. https://docs.aws.amazon.com/lambda/latest/dg/invocation-eventsourcemapping.html

2. https://docs.aws.amazon.com/lambda/latest/dg/with-s3.html

Cloud details Azure

Azure Functions

On Azure, the pattern can be implemented with Azure Function. The service seamlessly connects to messaging services like Event Hubs, Event Grid, or Service Bus, and some data-at-rest stores like CosmosDB, Azure Blob Storage, and Azure Table Storage.

Like for AWS Lambda, an Azure Function can act as an automatically scalable streaming job, Change Data Capture connector, or a simple data processing application working on the blobs (files) from an Azure Storage container.

Links:

1. https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings?tabs=csharp
2. https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-event-hubs-trigger?tabs=python
3. https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-cosmosdb-v2-trigger?tabs=python
4. https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob-trigger?tabs=csharp

Cloud details GCP

Cloud Functions

When it comes to the GCP Cloud Function, it's also similar to the AWS and Azure serverless offerings. You can trigger a Cloud Function to respond to a Pub/Sub message, a Cloud Storage object operation, or a Cloud Firebase item. Depending on the data source, the function can act as a streaming job, Change Data Capture Connector, or a file-based data processing application.

Links:

1. https://cloud.google.com/functions/docs/concepts/events-triggers
2. https://cloud.google.com/functions/docs/calling/storage
3. https://cloud.google.com/functions/docs/calling/pubsub

Exactly-once delivery

Problem

One of your jobs has the at-least-once processing semantic, but you need to deliver the data exactly once to avoid complex deduplication on the consumers' side. You're looking for a way to implement it by relying as much as possible on the features available in your target data store.

Mechanisms

1. Define a unique key for the written record. If the SDK generates it for you, you must ensure it doesn't change upon the retries.
2. Write the record to the output data store supporting the exactly-once delivery.

Use for

Writing each record only once

You can provide a better data quality without the duplicated data.

Avoid implementing data deduplication logic in the consumers

The consumers' codebase stays simple because it doesn't have to deal with duplicates. Moreover, the solution relies on the native features of the output data store and the code isn't complex for the data producers either.

Providing an exactly-once delivery for the retryable data producers

If the producer retries the data delivery in case of a failure, the pattern will help eliminate the duplicates, even with Micro batch pattern.

Look at

Determinism

The id generation must be deterministic. It means that generating the id for the given record multiple times, for example, during the retries, must always generate the same id value.

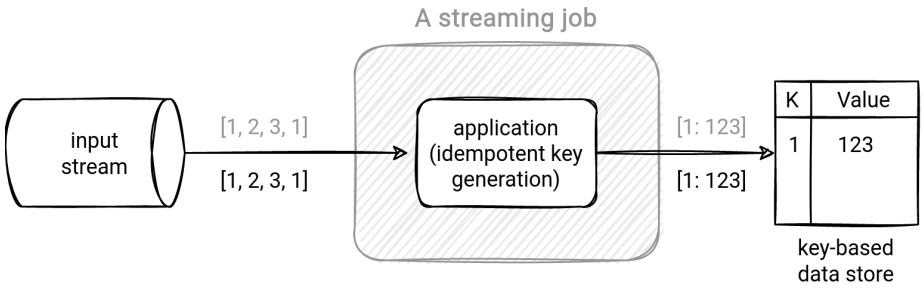
Limits

The perfect exactly-once delivery works for data stores supporting a key-based record identification. If the record key is deterministic, it won't introduce duplicates. Unfortunately, some data stores, such as data warehouses, don't provide a unique way to identify a record.

Time boundary

Sometimes the data store might not provide a key-based identification but can implement the pattern under other conditions. It's the case for the exactly-once delivery based on a time-limited deduplication window. The producer will ignore any duplicated entries written within this window. It's a less perfect implementation since the semantic is time-limited.

Schema design



Cloud details AWS

SQS

SQS can use different types of queues. One of them is `FIFO` and it's the one guaranteeing ordered exactly-once delivery.

However, the service implements the less perfect version of the pattern because it supports the deduplication only within a 5-minutes window. If you try to send the same message after this period, SQS will not detect it as a duplicate.

The producer must provide a unique key for each message to enable the deduplication. It can configure it in 2 ways:

- ask SQS to generate the id as an SHA-256 hash computed from the message body
- explicitly set the id while sending the record

Links:

1. https://aws.amazon.com/sqs/faqs/#fifo-queues
2. https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/FIFO-queues-exactly-once-processing.html

Cloud details Azure

Stream Analytics

Stream Analytics supports the exactly-once delivery for these 3 target data stores:

- CosmosDB. The solution relies on the `upsert` writing mode. It guarantees adding the document to the container only if it doesn't exist. Otherwise, it gets updated with the most recent values.
- SQL. Here, the strategy relies on the Azure SQL capacity to enforce the unicity constraints with the tables keys. Therefore, each written streaming event must have a unique key, defined either as a single field or a combination of fields.
- Azure Table. The implementation also uses the upsert mode. Azure Table identifies each row as a pair of `RowKey` and `PartitionKey` fields. Each inserted row with the same values for these keys will overwrite the previous row if it exists or add a new entry otherwise.

As you can see, Stream Analytics implements the *perfect* version of the pattern relying on the data store guarantees. Although, it can still be broken if the ids generation is not deterministic.

Links:

1. https://docs.microsoft.com/en-us/stream-analytics-query/event-delivery-guarantees-azure-stream-analytics#output-supporting-exact-once-delivery-with-azure-stream-analytics

Cloud details GCP

BigQuery

Even though BigQuery doesn't enforce the data uniqueness at the storage level as an RDBMS could do with primary key constraints or `MERGE` operation, its `Streaming Insert API` has a built-in feature to deduplicate rows.

The deduplication is based on the `insertId` attribute associated to the added row. When defined, the streaming API performs a best-effort data deduplication. It's called so because it doesn't guarantee an overall lack of duplicates. Instead, it has a 1-minute data deduplication window and is an example of the less perfect implementation of the pattern.

But despite the time limits, it can be sufficient to handle some specifc scenarios like retried inserts in a short time period.

Links:

1. https://cloud.google.com/blog/products/gcp/after-lambda-exactly-once-processing-in-cloud-dataflow-part-3-sources-and-sinks
2. https://cloud.google.com/bigquery/streaming-data-into-bigquery

Dataflow

Dataflow supports the exactly-once delivery for Pub/Sub with the `PubsubIO` API. The implementation relies on the Pub/Sub `message id` attribute. If a Dataflow job sees the given id multiple times, it'll process it only once.

The implementation requires the Pub/Sub data producers to set an idempotent message id, so that it doesn't change for the same messages. Otherwise, they would be considered as different by Dataflow.

In addition to the message id, there is also a possibility to use a custom metadata attribute. The `PubSubIO` API exposes a `withIdAttribute` method that "specifies the name of the attribute containing the unique identifier". However, it's a less perfect implementation of the pattern because it works per 10 minutes window for each message.

Links:

1. https://cloud.google.com/dataflow/docs/concepts/streaming-with-cloud-pubsub#efficient_deduplication
2. https://beam.apache.org/releases/javadoc/2.32.0/org/apache/beam/sdk/io/gcp/pubsub/PubsubIO.R

ead.html#withIdAttribute-java.lang.String-

Micro-batch processing

Problem

One of your streaming jobs is slow. After some debugging, you found out that it reads one message at a time and notifies the streaming broker about its successful processing just afterward. You want to optimize that aspect without transforming the streaming job into a batch one.

Mechanisms

1. Configure the number of elements to process in every micro-batch execution.
2. Adapt the data processing job to handle multiple events at once.
3. Process the data.
4. Inform the streaming broker only about the last processed record of the micro-batch.

Use for

Metadata optimization

The pattern optimizes the metadata tracking by checkpointing only the last processed offset. The consumer only needs to checkpoint the number of last read record in the micro-batch.

Simultaneous operations

With the micro-batch semantic, you can apply an operation on multiple events at once and, for example, leverage the Batching pattern to enrich the data from external data sources.

Cost optimization

If the consumer works in the pay-as-you-go mode, reducing the number of invocations might also reduce the price.

Latency reduction

Since the consumer will read more records at once, it will be less impacted by the metadata management and the network traffic cost than a consumer getting one element at a time.

Look at

Error management

Dead letter pattern can be harder to implement while processing multiple elements at once. The implementation must deal with partial results because the processing function can fail at any moment, including after processing the first half of the data.

Delivery semantic

The micro-batch can be a static unit of work. Consequently, successful partial processing can require the whole dataset being reprocessed. It can lead to duplicates in the output data store or a more complex code logic to avoid them.

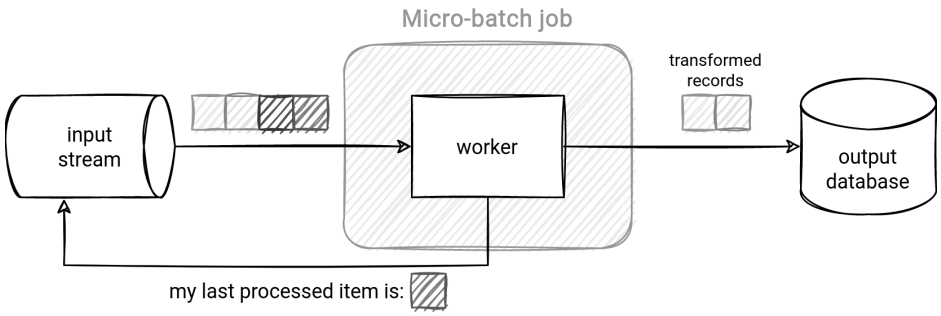
Ordering semantics

If the micro-batch sorts the records and fails after writing half of them to the data store, the retry will introduce out-of-order items.

Compute requirements

Since the job processes multiple items at once, it may need more hardware to do the job.

Schema design



Cloud details AWS

Lambda

A `Kinesis Data Streams` and a `DynamoDB Stream` triggers have a parameter called `batch size` that defines the number of records the function will process in a single invocation. As for many other things on the cloud, there is a limit too. The upper boundary for the batch size is 10 000 records or 6MB of the whole payload.

The `batch size` can work with another micro-batch parameter called `batch window`. The batch window acts more as a backpressure controller because it delays the function invocation. Thanks to that, the function can accumulate more records to process in the next execution. If both the size and window parameters are present, the function triggers whenever the first of them is met.

Links:

1. https://aws.amazon.com/blogs/compute/new-aws-lambda-scaling-controls-for-kinesis-and-dynamodb-event-sources/
2. https://docs.aws.amazon.com/lambda/latest/dg/with-kinesis.html
3. https://aws.amazon.com/premiumsupport/knowledge-center/lambda-batch-small

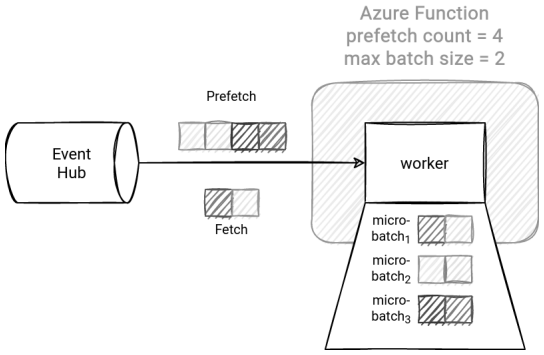
Cloud details Azure

Azure Functions

Azure Function also has 2 parameters impacting the micro-batch semantic:

- `max batch size` to control the amount of data retrieved from an Event Hub
- `prefetch count` to define the number of prefetched records

What's the difference between prefetched and fetched records? The prefetched ones are kept in memory and passed to the function once all fetched elements are processed. Put another way, the function processes up to max batch size records at once but can read more (prefetch count). Thanks to that, the next micro-batch directly from the buffer and decrease the latency.



Links:

1. https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-event-hubs
2. https://stackoverflow.com/questions/57785390/what-is-the-difference-between-prefetch-and-max-batch-size-when-consuming-events

No Code data processing

Problem

Your team is mainly composed of ex-software engineers working now as data engineers. They have great coding skills and can write any pipeline using a distributed data processing framework. However, you're also working with another team where the members are less proficient in programming. You want to find a service they could use to define data processing logic without writing the code.

Mechanisms

1. Define the input dataset.
2. Set the schema and select the transformations for each column to change.
3. Execute the pipeline directly or schedule it for a regular execution. The service is going to translate the logic to an executable code.

Use for

No-code data engineering jobs

The data transformation logic is often expressed as a high-level set of drag & drop-based operations. The service later takes this manually created visual representation of the job and translates it to an executable code.

Data discovery and profiling

Although it's possible to run no-code jobs as regular data processing jobs, it's also possible to use them for more ad-hoc use cases, like data discovery and data profiling. Once again, it'll depend on the personal preferences but sometimes profiling and discovering the data from a User Interface can be faster and more pleasant than writing the code and deploying it as a job from a notebook.

Data access democratization

Since the data access doesn't require programming skills anymore, the pattern can also facilitate spreading data culture within an organization. Still, it will often be a non flexible solution, but the provided features can be enough for non-technical users.

Look at

Extendibility

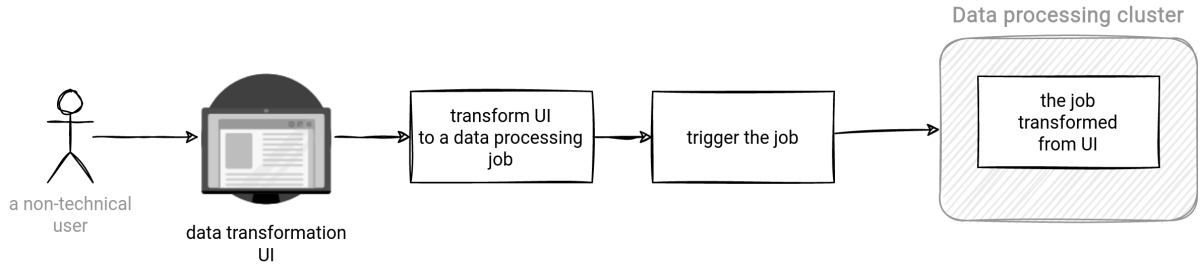
The strength of the pattern is also its weakness. The transformations available from the UI are often the single ones supported by the service.

No-code

Even though it's a no-code solution, it might still require some coding skills. It should be noticeable while transforming a temporary visual-based data exploration pipeline to a regularly

scheduled task integrated into the DevOps practices.

Schema design



Cloud details AWS

Glue Data Brew

In addition to the crawler, job and scheduler, AWS Glue also supports visual data preparation with the `Data Brew` component. Instead of writing Python or Scala code, the users can define their data transformation logic from the User Interface.

To set up the graphical data processing, you have to create a `project` with the following attributes:

- a `recipe` that is a visually defined data transformation composed of steps
- a `dataset` to configure the data source to process

Later the Data Brew service converts these steps to a `job`. The job processes the dataset and writes the outcome to the selected sinks. As of this writing, the service supports RDS, S3, and Redshift as the outputs.

Links:

1. https://docs.aws.amazon.com/databrew/latest/dg/projects.html
2. https://docs.aws.amazon.com/databrew/latest/dg/jobs.recipe.html
3. https://docs.aws.amazon.com/databrew/latest/dg/getting-started.html

Cloud details Azure

Data Factory

At first glance it may sound weird because Data Factory is a well-known data orchestration service. However, besides its data orchestration capabilities, it implements the no code data processing pattern with a special resource called `mapping data flows`.

A mapping data flow declaration is visual and consists of the following steps:

1. Defining the source of the dataset to transform.
2. Configuring the data transformations like derived columns, text parsing, unions, or even joins with other datasets.

3. Setting the output storage in the sink transformation.

Under-the-hood, Data Factory transforms the mapping data flow graph into an Apache Spark job and executes it using a scaled-out cluster.

Links:

1. https://docs.microsoft.com/en-us/azure/data-factory/concepts-data-flow-overview
2. https://docs.microsoft.com/en-us/azure/data-factory/data-flow-sink

Cloud details GCP

Data Fusion

`Data Fusion` is a GCP implementation of the pattern. This no-code visual data transformation service relies on a `pipeline` concept.

A pipeline configures a data source, data transformation logic, and data sinks. Under-the-hood, the visually defined pipeline gets transformed into a `Dataproc` job running on an ephemeral cluster.

Links:

1. https://cloud.google.com/data-fusion/docs/concepts/overview

NoSQL Change Data Capture

Problem

Your system uses a transactional key-value database to store the information about the actions made by the users of your application. So far, you have been reading this database in an ad-hoc way, always accessing one row by the primary key. However, you now have a new use case that requires streaming all the changes made in this table. Changing the table structure is not possible.

Mechanisms

1. Enable the streaming layer in the database.
2. Configure the streaming layer attributes, such as the records content or the retention period.
3. Implement the data consumer as a fully managed service or a custom code running on a virtual machine.

Use for

Deduplicate records

If the CDC source provides the comparative view of new and previous versions of the data, you can use the pattern to implement the data deduplication job.

Synchronize the transaction store with other systems

The transaction store transforms to an events log and enables real-time synchronization with other components of your architecture, opening them for easy format conversion or event-based data processing scenarios.

Real-time data processing

You don't need to write a batch job that will scan the whole table or use a complex key-based partitioning processing logic. Instead, you can process the data continuously and potentially use less compute resources.

Look at

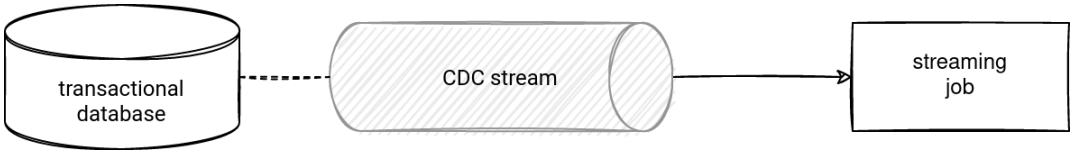
Deletes

Check if the CDC implementation can stream the `delete` operations. If not supported, you might need to switch to `soft deletes` and add a delete flag to the table attributes. Otherwise, the information might be lost.Duplicates. The streaming layer gets all the write operations. Hence if a writer retries the same action multiple times, your CDC consumer will process it multiple times as well. This weakness can become a strength, though, since it can be used in the deduplication logic.

Change detection

You will probably have to implement a logic comparing the streamed record to see what attributes have changed.

Schema design



Cloud details AWS

DynamoDB

DynamoDB implements the CDC pattern that can run in different modes:

- Native DynamoDB Streams component. It's the pull-based part of DynamoDB that supports up to 2 concurrent consumers and 24 hours data retention. The native streams can be processed by an AWS Lambda function or a DynamoDB Streams Kinesis Adapter.
- Kinesis Data Streams for DynamoDB. It can be pull- or push-based and support up to 20 concurrent consumers in enhanced fan-out mode (see Single tenant pattern) with 1 year data retention. Since the data is available in a Kinesis stream, it can be processed by any consumer supporting the stream as a source.

Besides the points mentioned in these short presentations, the 2 modes have important data-related differences. The native streaming capability guarantees no duplicates and ordered processing at an item level. The Kinesis-based solution can have duplicated records and requires an extra effort to implement the ordered processing with the record's timestamp attribute.

When it comes to the stream records, they can be of a different `type`:

- `KEYS_ONLY` - contains only the partition key of the row.
- `NEW_IMAGE` - contains only the new version of the row.
- `OLD_IMAGE` - contains only the old version of the row.
- `NEW_AND_OLD_IMAGES` - contains the old and new versions of the row.

Links:

1. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/streamsmain.html
2. https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_StreamSpecification.html
3. https://aws.amazon.com/blogs/database/dynamodb-streams-use-cases-and-design-patterns/

Cloud details Azure

CosmosDB

Cosmos DB implements the pattern with the `Change Feed` feature that continuously writes all changes made on the documents to a stream.

To consume this data, you've to connect an Azure Function or a process implementing the Change

Feed Processor (CFP) library. There is also a third way with CosmosDB SDK but requires a full management, from the partitions discovery to the checkpoint mechanism. All this is natively provided in the CFP library.

The data processing supports Push and Pull patterns. The latter involves the checkpointing via in-memory or manually persisted `continuation tokens`.

You must be aware the feed doesn't record the `delete` actions. Instead, you can use the soft-deletes mechanism by setting a *delete* attribute and a short TTL attribute to the document. Thanks to the TTL, the service will automatically delete the document, and you'll still have a chance to process it in the feed, since it's an update operation. However, you'll have to consume the change from the feed before it gets deleted by Cosmos DB.

When it comes to the recorded update operations, you also may miss some of them. If multiple concurrent processes are updating a document in a short period of time, you may miss some of the intermediary updates. You can use an add-only writing model to overcome this limitation and materialize each update as a series of writes.

There is no dedicated retention period for the Change Feed. Instead, the documents are retained as long as they aren't deleted from the Cosmos DB containers.

Links:

1. https://docs.microsoft.com/en-us/azure/cosmos-db/change-feed
2. https://docs.microsoft.com/en-us/azure/cosmos-db/sql/change-feed-processor
3. https://docs.microsoft.com/en-us/azure/cosmos-db/sql/read-change-feed
4. https://docs.microsoft.com/en-us/azure/cosmos-db/sql/change-feed-pull-model
5. https://docs.microsoft.com/en-us/azure/cosmos-db/sql/change-feed-design-patterns

Cloud details GCP

Firestore

Even though GCP has a dedicated Change Data Capture service called Datastream, it's reserved for relational databases, whereas the CDC pattern covers the NoSQL storage. That's why you will find the Firestore service here instead.

Cloud Firestore implements the pattern with the support of Cloud Functions triggered as a response to 4 different event types:

- `onCreate` for the insert operation.
- `onUpdate` for the update operation.
- `onDelete` for the delete operation.
- `onWrite` for all of the 3 above actions.

The event types are the triggers you can define in the Cloud Firestore-triggered Cloud Function. Besides them, you also need to specify the `document path`. The path can use a wildcard to listen for

the changes in documents under the collection or include an exact path referencing a specific document.

However, it's a lightweight pattern implementation since the changes are pushed to the function without a persistence layer.

Links:

1. https://firebase.google.com/docs/firestore/extend-with-functions

Ordered delivery

Problem

The dataset processed by your streaming application contains time-dependent events, like create and delete actions. They naturally imply a logical relation because the "create" action must be processed before the "delete" one. Your goal is to implement order-based processing by leveraging the streaming broker or the data producer instead of doing it at the consumer level.

Mechanisms

1. On the data writer side, define the attribute(s) correlating the events, for example, a user id.
2. Use the correlation attribute(s) as the message key to deliver the data to the same physical location.
3. Let the streaming broker put the event with the same key in the same physical location. The ordering should rely on this physical location and/or the correlation attribute.

Use for

Low effort ordering guarantee

The streaming broker natively implements the ordering guarantee by leveraging the physical storage or the attribute. There is no need to implement a windowing logic on the consumer side to process the records in order.

Optimize aggregations

A consumer instance will have all related elements together for the aggregation scenarios. It won't need to redistribute and exchange the data with other consumer instances, saving the network traffic.

Look at

Physical storage coupling

Some of the implementations might rely on the data partitioning and *hash-modulo*-based partitioning function. Changing the storage topology by adding or removing the partitions will break the ordering between the existing and new events.

Failures

Check how the broker behaves for partial writes to the same partition. If it stops ingesting the data at the first failed record, you will need to filter out the successfully added items before sending the retry. If it doesn't stop, you might need to reduce the latency of the data generation (check the "Throughput" point).

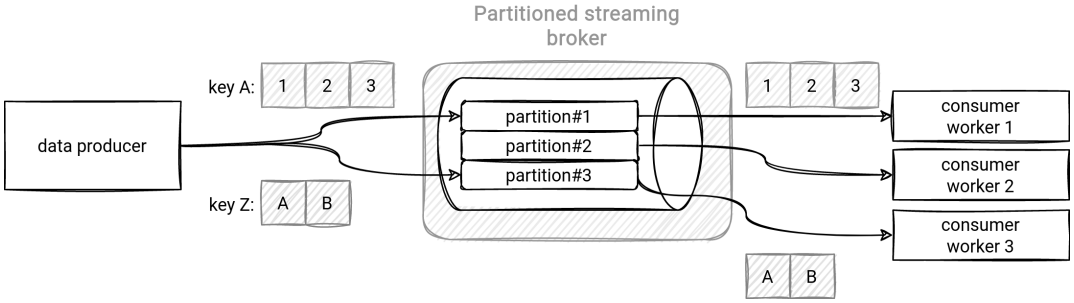
Data skew

The ordering often relies on a relationship between a logical (record attribute) and physical (partition) information. If the record attribute is unbalanced, one partition can be skewed, so contain much more data than the others. It might lead to increased processing time, even if the remaining partitions store much less data.

Throughput

In some implementations, the ordered delivery can increase the latency because the producer will have to send one record for a given key in the request. It can still use the Batching pattern though by delivering only 1 record per correlation key in the batch.

Schema design



Cloud details AWS

Kinesis Data Streams

Kinesis Data Streams implements the pattern with `PutRecord` operation. The delivered record should have an associated `SequenceNumberForOrdering` attribute responsible for the ordering at the shard level. When this parameter is missing, Kinesis Data Streams stores records sorted coarsely by arrival time.

`SequenceNumberForOrdering` should have the sequence number of the previously delivered record. This pseudo-code snippet presents the workflow:

```
records_to_deliver = [record1, record2, record3]
previous_sequence_number = null
for record in records_to_deliver:
  record.sequence_number_for_ordering = previous_sequence_number
  put_result = kinesis_client.put_record(record)
  previous_sequence_number = put_result.sequence_number
```

However, this strategy has an important drawback. It works with the `PutRecord` method, so with the method delivering a single element per request. It also allows only one producer per shard.

An alternative could be the usage of `PutRecords` operation with the explicit partition key and a single record for a given partition key in the request. Additionally, the groups should be sorted by the ordering key:

```
groups_to_deliver = create_groups([(pk1, data1), (pk2, data2), (pk1, data3)]
for group in groups_to_deliver:
  # group 1 = [(pk1, data1), (pk2, data2)]
  # group 2 = [(pk1, data3)]
  result = kinesis_client.put_records(group)
  if result.is_falied:
```

```
    retry_failed_records(result.failed_records)
```

The PutRecords-based solution addresses the throughput issue but doesn't solve the problem of multiple producers sharing the same partition key.

`PutRecords` alone doesn't provide the ordering guarantee:

> The response Records array includes both successfully and unsuccessfully processed records. Kinesis Data Streams attempts to process all records in each PutRecords request. **A single record failure does not stop the processing of subsequent records. As a result, PutRecords doesn't guarantee the ordering of records.** If you need to read records in the same order they are written to the stream, use PutRecord instead of PutRecords, and write to the same shard.
>
> — https://docs.aws.amazon.com/kinesis/latest/APIReference/API_PutRecords.html

Links:

1. https://docs.aws.amazon.com/kinesis/latest/APIReference/API_PutRecord.html
2. https://docs.aws.amazon.com/kinesis/latest/APIReference/API_PutRecords.html
3. https://docs.aws.amazon.com/streams/latest/dev/developing-producers-with-sdk.html
4. https://forums.aws.amazon.com/thread.jspa?threadID=143285
5. https://aws.amazon.com/blogs/database/how-to-perform-ordered-data-replication-between-applications-by-using-amazon-dynamodb-streams/

Cloud details Azure

Event Hubs

Event Hubs producers can also implement the pattern with a one-record-at-a-time strategy. However, they can also use a more efficient version thanks to a different batch operation semantic and partition-based ordering guarantee.

The batch data generation API uses an `event data batch` class to group all data written in a single call. Unlike Kinesis' `PutRecords`, the outcome of this operation is atomic and hence, cannot contain successful and failed writes:

> A set of EventData with size constraints known up-front, intended to be sent to the Event Hubs service in a single operation. When published, the result is atomic; either all events that belong to the batch were successful or all have failed. Partial success is not possible.
>
> — https://docs.microsoft.com/en-us/dotnet/api/azure.messaging.eventhubs.producer.eventdatabatch?view=azure-dotnet

By default, Event Hubs SDK will assign the batched events to the first available partition. As a result, Event Hubs will store the events that should be ordered in different partitions, breaking the ordering guarantee. The delivered batch should then have the `partition id` attribute defined so that all batched events will go to the same partition.

Links:

1. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-python-get-started-send
2. https://docs.microsoft.com/en-us/dotnet/api/microsoft.azure.eventhubs.eventdatabatch.tryadd?view=azure-dotnet#Microsoft_Azure_EventHubs_EventDataBatch_TryAdd_Microsoft_Azure_EventHubs_EventData_
3. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-availability-and-consistency?tabs=dotnet#send-events-to-a-specific-partition
4. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-availability-and-consistency?tabs=dotnet#consistency
5. https://docs.microsoft.com/en-us/dotnet/api/azure.messaging.eventhubs.producer.eventdatabatch?view=azure-dotnet
6. https://docs.microsoft.com/en-us/dotnet/api/azure.messaging.eventhubs.producer.createbatchoptions?view=azure-dotnet

Cloud details GCP

Pub/Sub

Pub/Sub implements ordered delivery thanks to the `ordering_key` attribute. All messages sharing the same `ordering_key` and published to the same regional Pub/Sub instance are delivered and stored in order.

To deal with the failures, Pub/Sub has some built-in features to prevent broken ordering in case of retries:

- Invalidation of the failed subsequence. A failed delivery automatically invalidates all subsequent messages sharing the same ordering key. For example, if the message number 2 of [1, 2, 3, 4] fails, the client library will automatically invalidate the delivery for 3 and 4. It will resume the delivery only after successfully sending the failed record number 2.
- Resuming failed delivery. A `resume publish` function on the client side delivers all failed messages. If we take the previous example, the workflow for an ordering key would look like that:

```
message 1 -> successful delivery
message 2 -> failed delivery; throws an exception
the client calls resume publish function
message 2 -> successful delivery after a retry
message 3 -> successful delivery
message 4 -> successful delivery
```

The SDK doesn't require the data producer to block while sending the messages. Put differently, it doesn't have to stop and wait for the delivery outcome. Instead, it keeps an in-memory buffer for the messages sharing the same ordering key and orchestrates their delivery under-the-hood, allowing the program to continue and retry only the failed keys with the `resume publish` function.

Links:

1. https://cloud.google.com/pubsub/docs/ordering
2. https://medium.com/@imrenagi/understanding-cloud-pub-sub-message-ordering-feature-a3e014b0a5be
3. https://cloud.google.com/pubsub/docs/publisher#python_2
4. https://medium.com/google-cloud/google-cloud-pub-sub-ordered-delivery-1e4181f60bc8
5. https://cloud.google.com/pubsub/docs/samples/pubsub-resume-publish-with-ordering-keys
6. https://github.com/GoogleCloudPlatform/pubsub/blob/master/ordering-keys-prober/src/main/java/com/google/cloud/pubsub/prober/OrderedProber.java

Parallel data upload

Problem

You need to transfer dozens of big files to an object store. After reading the documentation, you're a bit worried because uploading each file would take 1 hour. You are looking for a way to accelerate this process by leveraging multiple cores of your computer.

Mechanisms

1. Identify the method in the object store API supporting parallel upload.
2. Define the size of each uploaded chunk or a global size shared by all chunks.
3. Call the upload method for the divided file.
4. Wait for the file to be fully uploaded.

Use for

Accelerated data upload

By dividing the file into multiple chunks and using all available local compute power to transfer them to the cloud, the data upload process can be much more efficient than doing that on a one-full-file-at-a-time basis.

Resources usage optimization

The multiple CPUs will probably be there anyway. In a classical upload, they'll remain idle, which can be thought of as resources waste.

Resilient upload

The pattern uploads an object splitted into multiple smaller chunks. In the case of an unreliable network, any upload failure will require retrying only the failed fragment (s). For the standard upload, the process should start from the beginning.

Look at

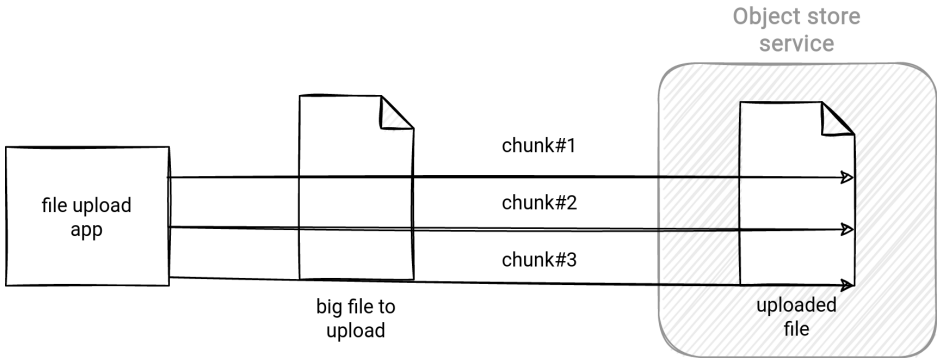
Difficulty

The pattern can be slightly more challenging to implement than uploading the whole file at once. It can require manually handling the chunks' upload result to confirm the final object's creation on the cloud at the end of the upload process.

File checksum

Sometimes it can be impossible to use the same checksum method as an object uploaded with a standard method. For example, GCS multipart upload doesn't support `md5`-based checksum.

Schema design



Cloud details AWS

S3

S3 implements the pattern with a flexible `multipart upload` feature. It supports the following configuration options:

- the minimal number of chunks
- the required minimal file size for the `multipart upload` mode

The upload process consists of 3 steps:

1. Initializing the upload. The client application sends an initialization request to get an upload id.
2. Uploading the chunks. The client can now upload each part of the file by specifying the chunk number and the upload id. S3 returns an ETag value for each successful upload. The client must associate this value with the chunk number to move on to the next step.
3. Completing the upload. Once the client uploads all chunks, it has to issue a completion request and define the upload id, and the list of chunk numbers with associated ETag values.

Links:

1. https://aws.amazon.com/blogs/developer/parallelizing-large-uploads-for-speed-and-reliability/
2. https://aws.amazon.com/premiumsupport/knowledge-center/s3-multipart-upload-cli/
3. https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/s3/transfer/TransferManagerConfiguration.html#setAlwaysCalculateMultipartMd5-boolean-
4. https://docs.aws.amazon.com/AmazonS3/latest/userguide/mpuoverview.html

Cloud details Azure

Blob Storage

Azure Blob API implements the parallel upload pattern with a 2-steps algorithm. In the first step, the client application divides the uploaded file into multiple chunks and transfers each of them separately by assigning the `block id` property to it. The client application uses the `Put Block` operation in this step.

The application must remember each `block id` composing the uploaded file because it's required to notify the service about the upload completion. After uploading the last chunk, the application can call the `Put Block List` operation and pass the saved `block ids`. The Azure service intercepts this request and composes the final blob from the passed block ids.

Links:

1. https://www.andrewhoefling.com/Blog/Post/uploading-large-files-to-azure-blob-storage-in-c-sharp
2. https://docs.microsoft.com/en-us/rest/api/storageservices/put-block
3. https://docs.microsoft.com/en-us/rest/api/storageservices/put-block-list

Cloud details GCP

GCS

In `gsutil`, GCS implements the parallel upload as a special upload type called `parallel composite upload`. In this mode, the client divides the uploaded file up to 32 chunks and transfers them to the cloud concurrently. Once all the chunks are on the cloud, GCS assembles them and composes the final object. At the end of the operation, the service deletes all the chunks and leaves only the final file.

The compose upload behavior is controlled by `parallel_composite_upload_threshold` and `parallel_composite_upload_component_size`. The former activates the parallel composite upload, whereas the latter defines the targeted size of each chunk with respect to the 32 chunks limit.

A drawback of this strategy is the lack of MD5 checksum information about the uploaded file. Any client based on `gsutil` or the `Python SDK` must perform the integrity check with a `crc32c` hash.

Besides the `gsutil`-based strategy, GCS also has an `XML API multipart uploads` working in 3 quite similar steps to S3 and Azure Storage:

1. Initialization. The client gets the `UploadId` to associate with each uploaded part.
2. Upload. It's time for the client to upload each chunk and get the ETag to persist.
3. Completion. To complete the operation, the client sends a request including the `UploadId` and the lists of the ETag and chunk number pairs.

Links:

1. https://cloud.google.com/storage/docs/uploads-downloads
2. https://cloud.google.com/storage/docs/multipart-uploads

Pull processing

Problem

Your data producers generate data continuously. Your jobs need to consume it as soon as possible, preferably from a long-running application.

Mechanisms

1. Initiate the connection to the streaming broker. In this step, you can set the reading starting position or consume only the most recent messages.
2. Call a method asking the broker to send the records from the previously defined reading position.
3. Process the received records and update the previous reading position. The update can be automatically made by the consumer API.
4. Repeat steps 2 and 3 indefinitely. You can add some pause before fetching the new batch of records to implement a static backpressure mechanism.

Use for

Continuous data processing

A pull-based consumer continuously processes the generated data. There is no reason to shut it down since the data comes uninterruptibly. Moreover, stopping and starting the consumer again would add an extra latency which is not welcome for the low latency streaming consumers.

Decoupling the data generation lifecycle from the data processing

When the consumer encounters an unexpected issue, it can go offline and still be able to process the data after showing up.

Implementing scenarios requiring a built-in reprocessing capability

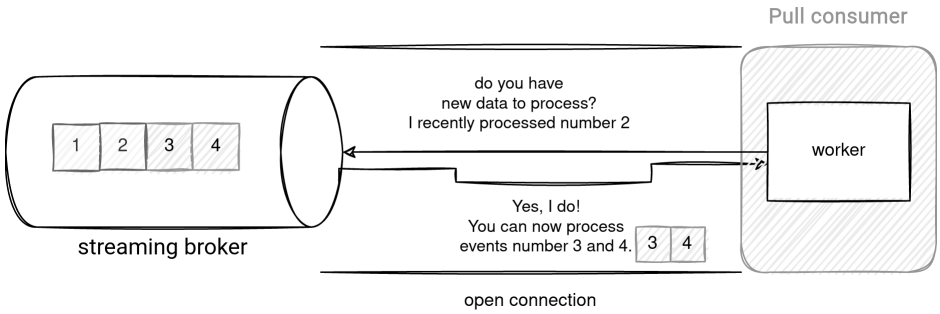
The pull pattern is often associated with append-only log streaming brokers, which provide data retention natively.

Look at

Data generation frequency

The continuous pull is less adapted to occasionally generated data. The consumer will stay idle and waste the resources while waiting for the new data to process.

Schema design



Cloud details AWS

Kinesis Data Streams

Kinesis Data Streams implements the pull pattern as *unregistered consumers without enhanced fan-out mode*. In this method, the consumers pull the messages over HTTP by calling the `GetRecords` endpoint.

Each request to this endpoint returns one of the following:

- an empty list if there is no new data to process
- a list with new records to process, up to 2 MB/sec per shard shared limit
- `null` if the consumed shard has been closed

All unregistered consumers share a 2 MB/sec/shard throughput limit. If the number of concurrent requests reaches this capacity, the service returns a `ProvisionedThroughputExceededException` error.

Links:

1. https://docs.aws.amazon.com/streams/latest/dev/building-consumers.html
2. https://docs.aws.amazon.com/kinesis/latest/APIReference/API_GetRecords.html

Cloud details Azure

Event Hubs

At the high level, Event Hubs consumers work in the pull mode. The service delivers new messages only for the running or restarted consumers. It doesn't happen in the Push pattern where the message broker sends each message only to the up and running targets, without a possibility to catch on them later.

Under-the-hood, an Event Hub consumer must first register to the broker. After that, the consumer can start processing the data and the communication with the broker will be in push mode from this moment, as per documentation:

> Any entity that reads event data from an event hub is an event consumer. All Event Hubs consumers connect via the AMQP 1.0 session and events are delivered through the session as they become available. The client does not need to poll for data availability.

> — https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-features#event-consumers

You'll find more details about the events processing in the Push pattern.

Links:

1. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-java-get-started-send
2. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-features#event-consumers

Cloud details GCP

Pub/Sub

Pub/Sub has 2 pull methods called `Pull` and `StreamingPull`. Their difference consists of the communication details.

Thanks to an optimized network path, `StreamingPull` provides a higher throughput and lower latency. Classical pull mechanism works with the request-response principle, continuously asking the broker for the new messages to process. `StreamingPull` calls the broker only once, to initialize the connection. Later, the broker continuously sends new records to the client through this open connection.

Besides the throughput, there is another important difference. The `Pull` method supports the max number of messages to fetch in every request. This configuration is not supported in the `StreamingPull` because the broker is responsible for the pulling logic.

Despite this backpressure difference the `StreamingPull` is often the default choice in the GCP client libraries.

Links:

1. https://cloud.google.com/pubsub/docs/samples/pubsub-subscriber-sync-pull
2. https://cloud.google.com/pubsub/docs/samples/pubsub-subscriber-async-pull
3. https://cloud.google.com/pubsub/docs/pull

Push processing

Problem

You're looking for a way to process new events as they arrive to your streaming broker. Unfortunately, the data producer is unpredictable, and there can be long gaps between the generated events. You don't want to keep a long-running consumer to process this data because it risks staying idle most of the time.

Mechanisms

1. Deploy your data consumer behind a notification endpoint. The endpoint should be serverless to avoid paying for the resources when there are no events to process.
2. Set the deployed endpoint as a new notification endpoint for the push-based broker.
3. Starting from now, the broker will send each new event to the endpoint.

Use for

Just-In-time processing

The consumers get notified when a new event arrives. If they're running as serverless offerings, they don't need to be running all the time.

Processing irregularly generated data

Long-running consumers are the best choice when the data flows continuously to the system. However, using them for irregular data producers will waste their compute resources and your money.

Avoid polling

The consumer doesn't need to ask the data source for the records to process. It's one request less to exchange.

Look at

The events' volatility

It may be impossible to reprocess an event, for example with a failed consumer and a not configured data retention or a Dead-Letter Queue.

The data availability

New consumers will only process the data produced after their deployment. It might not be possible to read already generated data.

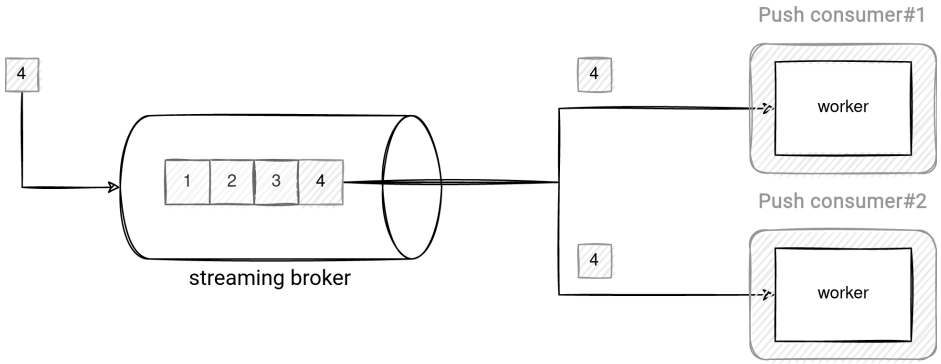
The coupling

The consumers must handle the data at the same pace as the data generation. If there is no alive consumer to process a message, the message may be lost forever.

Cold start

If you rely on a serverless runtime, which helps reduce costs, keep in mind the cold start phenomena. Your data consumer is not up and running along the day, so it has to be started in response to the new events to process. This start time is often called a *cold start* because it adds an extra latency between the real data processing time and the data delivery.

Schema design



Cloud details AWS

Kinesis Data Streams

One of the pull limitations in Kinesis Data Streams consumers is their shared throughput leading to provisioning errors. An approach to mitigate the issue uses throughput reservation for a consumer group called the *enhanced fan-out*.

In this throughout-dedicated mode, the consumer first sends a `SubscribeToShard` request to the broker. Upon accepting the request, the broker pushes the new events to the consumer over the HTTP/2 connection for up to 5 minutes. Passing that time, the consumer must renew his subscription and send the `SubscribeToShard` renewal message.

The push processing consumers based on the enhanced fan-out have a dedicated reading limit of 2 MB/sec per shard.

Links:

1. https://docs.aws.amazon.com/streams/latest/dev/building-consumers.html
2. https://docs.aws.amazon.com/kinesis/latest/APIReference/API_SubscribeToShard.html

Cloud details Azure

Event Hubs

Event Hubs documentation doesn't explain very clearly the processing modes. The `push` pattern mentioned in the documentation applies to the network-level data reading mechanism. On the other hand, the high-level consumers might also fit into the pull pattern category because they're decoupled from the broker and don't lose the message if they go offline for a while.

To be more precise, the component responsible for processing the events is `Event Processor`. It defines several functions that will consume the data, checkpoint the job progress, and load balance the consumers inside the consumer group. In the data reading you won't find a pattern involving repeated fetch calls (pull) to the Event Hubs service. Instead, the processor will simply register itself with the Event Hubs service by invoking a method called start or receive, depending on the SDK. From that moment, the Event Hubs service knows that there is a new consumer reading data from one or multiple partitions. It can then push the data to the consumer's events processing function.

You will find the mention of this mechanism in the documentation of the AMQP protocol:

> After an AMQP 1.0 session and link is opened for a specific partition, events are delivered to the AMQP 1.0 client by the Event Hubs service. This delivery mechanism enables higher throughput and lower latency than pull-based mechanisms such as HTTP GET.
>
> — https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-features#read-events

Links:

1. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-java-get-started-send
2. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-features#common-consumer-tasks
3. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-features#read-events

Event Grid

Event Grid is a more typical example of push processing service on Azure. The service relies on the high-level push mechanism where the consumers are coupled to the data generators.

The service is composed of:

- Publishers. They're custom or cloud-managed clients generating the events.
- Topics. They're the places where the events arrive. Event Grid supports 3 types of topics:
  - The `system topic`. It's fully managed by Azure. It's the place where other Azure services like Blob Storage, Event Hubs, or Key Vault send their events.
  - The `custom topic`. This one is fully managed by the users who can send there any events.
  - The `partner topic`. It's the place where associated 3rd parties can send their events.
- Subscription. It's an intermediate layer between your consumers and the topics.
- Event handler. In other words, event consumers. An event handler can be an application-like service (an Azure Function or a Logic App), or another messaging storage like an Event Hubs or Service Bus.

Whenever a publisher sends a message to a topic, the service delivers it to all subscribed event handlers in a push-based way, so without needing to keep the consumers alive.

Links:

1. https://docs.microsoft.com/en-us/azure/event-grid/delivery-and-retry

Cloud details GCP

Pub/Sub

In Pub/Sub, push processing relies on HTTPS communication between the consumer and the broker.

The broker first delivers each message to the consumer. In its turn, the consumer informs the broker about the outcome of the handled request. If the request is processed successfully, the response is a `2xx` or `102` HTTP code. For any error related to the message processing or the time out, the consumer uses a different status.

For any consumer failure, the broker considers the delivery unsuccessful. Depending on the subscription configuration, it can retry to deliver the message again.

Links:

1. https://cloud.google.com/pubsub/docs/push

Reprocessing data from a streaming data source

Problem

Two days ago, you deployed a new version of your streaming job. Unfortunately, you discovered some bugs in the mapping function afterwards. You fixed them and updated the version running in production. But before restarting the pipeline you must reprocess the data processed with the buggy job.

Mechanisms

1. Stop the buggy job.
2. Fix the bugs and release the new version.
3. Find the first position of the records processed by the buggy job.
4. Deploy the new job and set its processing position to the position found in the previous step. The setting can be made directly from the job level parameters or require interaction with a `checkpoint storage`.

Use for

Bug fixing

Streaming jobs evolve like any other software. Each evolution brings the risk of regression and invalid data delivered to the consumers. If the consumers do care about the data correctness, fixing the bug and redeploying the job won't be enough. The new version will also need to replay the pipeline from the point of failure.

Backfill historical data

Sometimes the data consumers can require historical data generated by the streaming pipeline. If the streaming broker has a long enough retention period, you can simply start the streaming job on the past data. Otherwise, you might need to adapt the pipeline and run it on the data stored at rest.

Look at

Data availability

The data retention period will vary depending on the cloud provider. Check if it's enough regarding your data reprocessing needs. In the case of insufficient retention, you will need to add long-term storage and a job to synchronize streaming data with it.

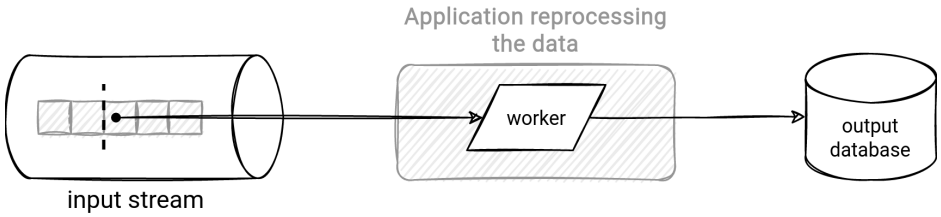
Costs

Longer retention means more expensive storage costs. Sometimes you might prefer a shorter retention period and reprocess the data only from your long-term storage. Often, it'll be cheaper than keeping the streaming data for the reprocessing.

Extra operational effort

Before each job deployment, you need to memorize the first record to process in the new version. Without this information, the reprocessing will be only approximate or you will need to spend some time on searching the replay position.

Schema design



Cloud details AWS

Kinesis Data Streams

To implement the pattern in Kinesis, you must set the consumer's `StartingPosition` attribute that accepts one of the following values:

- A `SequenceNumber`. It's a unique sequence number identifying each record added to the stream shard. Using it requires knowledge of the last successfully processed sequence number.
- A `Timestamp`. It's the timestamp of the first record to process in the job. If no record matches the exact timestamp, the consumer will start with the first record written after this timestamp value.
- `Trim Horizon`. Here, the application will process the very first records available on each shard.
- `Latest`. It's the opposite of the above. The application will process the events written after the consumer's connection to the broker.

Kinesis Data Streams can retain data for 24 hours (default), 7 days (extended data retention), or 365 days (long-term data retention).

Links:

1. https://docs.aws.amazon.com/kinesis/latest/APIReference/API_StartingPosition.html
2. https://aws.amazon.com/kinesis/data-streams/faqs/

Cloud details Azure

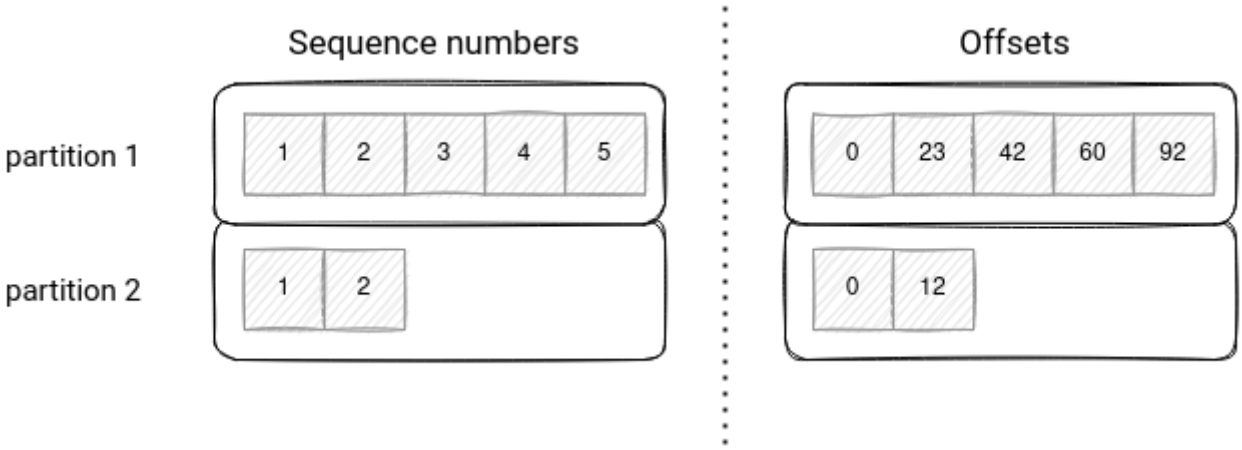
Event Hubs

An Event Hubs consumer is also position-based. You can set the processing start time in the `EventPosition` attribute as one of the following values:

- Offset. It's the relative position of an event inside a partition.
- Sequence number. It represents the logical sequence number of an event inside a partition.

- Enqueued time. It's the enqueued time of the event.
- First available offset in the partition.
- Last available offset in the partition.

The offset and the sequence number are 2 slightly similar concepts. Both represent a numerical position of the event inside the partition but this position stands for different things. The offset represents the physical position, so the starting byte of each event in the partition.



On the other hand, the sequence number is a continuous number of the event within the partition. It increases by 1 whenever the broker appends a new event to the partition. For the reprocessing scenario it should be easier to reason about the sequence numbers than the offset numbers.

Links:

1. https://stackoverflow.com/questions/66284127/azure-eventhub-offset-vs-sequence-number
2. https://docs.microsoft.com/en-us/javascript/api/@azure/event-hubs/eventposition?view=azure-node-latest
3. https://github.com/Azure/azure-event-hubs-spark/blob/master/docs/structured-streaming-eventhubs-integration.md

Cloud details GCP

Pub/Sub

Data reprocessing in Pub/Sub relies on a feature called `Seek`. A consumer can use seek to reach:

- A timestamp. It's time where Pub/Sub received the message. If you want to purge all the messages, even the unprocessed ones, you can also set it to the future!
- A snapshot. It's a backup of the main subscription that stores all unacknowledged messages and all messages produced after the snapshot creation.

To use any of these options, you will have to do some actions first. The timestamp-based seek is supported only for the subscriptions configured with the retention of the already acknowledged messages. To enable it, you must explicitly set the `retain_acked_messages` property to true and configure the retention period in the `message_retention_duration` property.

The snapshot-based strategy requires an explicit action of creating the snapshot ahead of time. The service will delete the snapshot automatically if it reaches a lifespan of 7 days or the oldest unacknowledged message exceeds the `message_retention_duration`.

Links:

1. https://cloud.google.com/pubsub/docs/replay-overview

Reusable job template

Problem

Your data architecture has grown a lot in past months. Recently you've noticed that many batch jobs share the same logic of copying the data within an object store. So far, you've been using different applications, but it makes each small evolution very difficult since you must apply it to multiple places. You're looking for a better way where you could keep the shared logic in a single place and share it with all the jobs. Ideally, the solution should be cloud native.

Mechanisms

1. Define a custom template or use an existing one provided by your cloud provider. It's common to find a ready-to-use template for passthrough jobs that move the data from one location to another without any transformation on the data itself.
2. Deploy the template definition if necessary.
3. For each job, reference the template definition and change all the variable parameters, such as data source location, data sink location, batch size, or number of retries.
4. Deploy the templatized job.

Use for

Centralized logic

Although the pattern is often used for simple jobs like passthrough jobs, it can also be for more complex ones where you code the common business rules only once.

Easier maintenance

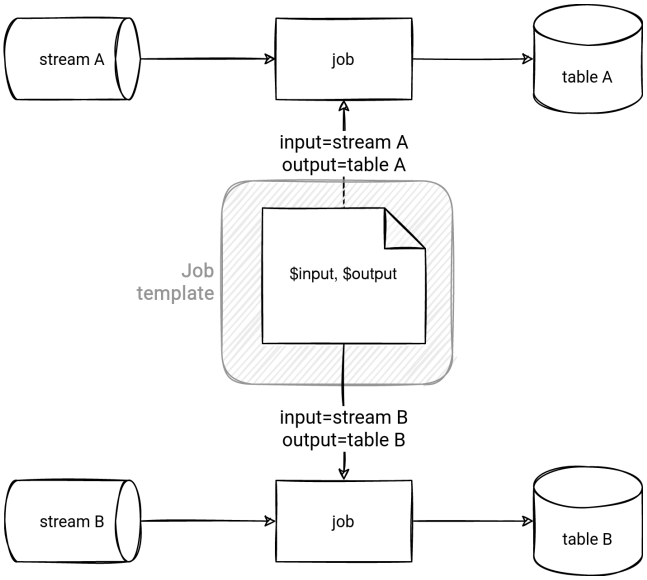
Any change made on the job template will automatically be available to all jobs using it. The single thing to do on your side will be updating the template version in the job itself.

Look at

Flexibility

The idea of sharing the logic sounds appealing. You might be tempted to create a template for the shared input and outputs, but the business logic depends on the executed job. The implementation would probably rely on if-else statements or many different classes, making the template understanding more difficult. Bear this in mind and challenge the templating idea for that scenario. Maybe splitting the project will work better in that case?

Schema design



Cloud details AWS

Glue

AWS Glue implements the pattern with the feature called `Blueprints`. Each blueprint is a combination of:

- a layout script that contains the data transformation logic executed by the job
- a configuration file that provides all necessary input information for the job

The configuration file has a `parameterSpec` section where you can set the input parameters to the layout script's job.

Links:

1. https://docs.aws.amazon.com/glue/latest/dg/developing-blueprints-sample.html
2. https://docs.aws.amazon.com/glue/latest/dg/developing-blueprints-code.html

Lake Formation

Although they have a different purpose, you will also find `Blueprints` in the Lake Formation service. In that context, their goal is to bring the data from an external data store to the AWS Lake Formation environment. To make it happen, you will need to define the database connection, the source data path, and of course, the destination database.

Links:

1. https://docs.aws.amazon.com/lake-formation/latest/dg/workflows-creating.html

Cloud details Azure

Data Factory

Azure provides templates to facilitate the setup of Data Factory pipelines. All you have to do is click

on the "*Pipelines templates*" item from the menu and choose one of the available models in the template gallery. You will find there 3 types of templates:

- `Copy templates` to jobs moving the data between 2 data stores.
- `SSIS templates` to jobs scheduling SSIS packages on an Azure-SSIS Integration Runtime.
- `Transformation templates` to jobs executing some custom data transformation code on Databricks.

You also can define your template and save it to the Template gallery component for private reuse within your organization.

Links:

1. https://docs.microsoft.com/en-us/azure/data-factory/solution-templates-introduction
2. https://docs.microsoft.com/en-us/azure/data-factory/solution-template-databricks-notebook

Cloud details GCP

Dataflow

If you have some repetitive data processing jobs on GCP, you can define them as `Dataflow templates`.

A Dataflow template can be reused later by simply setting the input and output variable parameters during the deployment.

One example of such reuse is a Pub/Sub to BigQuery synchronization template. The template is composed of the parameters for the input Pub/Sub subscription, the output BigQuery table and the dead-letter BigQuery table for the failed records (see Dead Letter Sink pattern).

Links:

1. https://cloud.google.com/dataflow/docs/concepts/dataflow-templates [https://cloud.google.com/dataflow/docs/concepts/dataflow-templates]
2. https://cloud.google.com/dataflow/docs/guides/templates/provided-streaming#cloudpubsubsubscriptiontobigquery

Serverless SQL worker

Problem

In the previous architecture of your data platform, you had a lot of ELT jobs executed on top of an on-premise data warehouse. However, this approach didn't scale very well and you want to replace it with a more scalable solution that supports running SQL queries. The solution shouldn't require any extra infrastructure-related knowledge. Ideally it should let you write a SQL query and execute it without worrying about sizing a cluster.

Mechanisms

1. Write your SQL query and set the data source location. Before running the query, you can verify the amount of processed data and get a price estimate.
2. Run the query.
3. Wait for the query job to complete.

Use for

Serverless SQL data processing

You don't need to worry about the underlying infrastructure. You can only focus on the query logic and delegate the compute environment management to the service.

Simplified data orchestration logic

Since the solution is serverless, you don't need to define steps to create and destroy the cluster in the data orchestration layer either.

Ad-hoc querying

Besides regular jobs, the pattern also works for ad-hoc querying and data exploration steps.

Cost optimization

The pattern can help optimize costs if the human factor required to operate the infrastructure is significant.

Look at

Costs

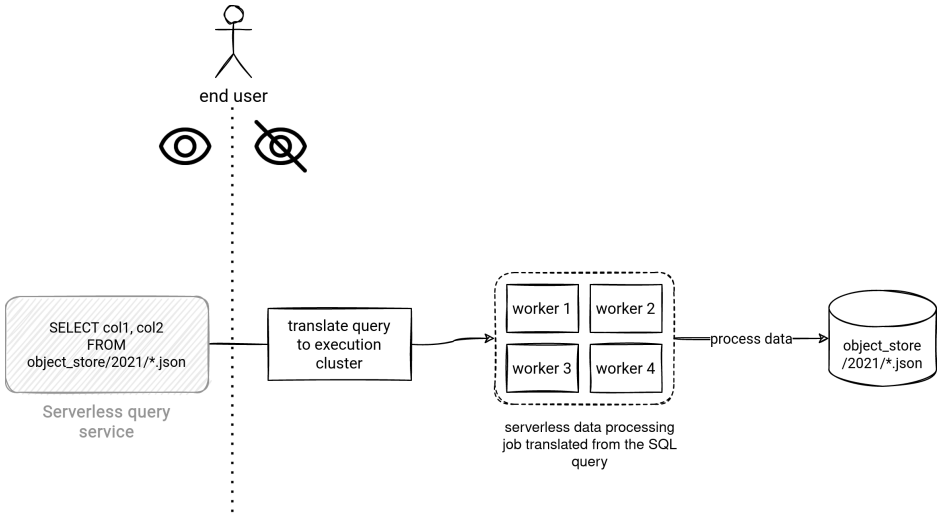
Often the pattern uses a serverless query engine billed per the number of processed bytes. If its execution is uncontrolled or the query must run multiple times without caching, e.g. due to some code regressions, it can be a costly solution.

Data format

To take full advantage of the serverless SQL worker and not explode your cloud bill, you will need to organize your data. The organization includes the storage layout (partitioning and clustering (bucketing)) and columnar file format to eliminate irrelevant parts of the dataset to

the query.

Schema design



Cloud details AWS

Athena

Athena is an AWS implementation of the pattern running SQL queries against the S3 objects. Although its compute environment is fully managed, the service uses Glue Data Catalog to store and retrieve the metadata information about the processed databases and tables.

S3 is the primary data source for Athena but it's possible to work with other data stores like DynamoDB via the `federated query` component. The component executes a set of Lambda functions that read the data sources present in the query.

Links:

1. https://docs.aws.amazon.com/athena/latest/ug/getting-started.html
2. https://aws.amazon.com/athena/faqs/

Cloud details Azure

Synapse

Even though Azure Synapse is a cluster-based data warehouse solution, it exposes a serverless query service called `Serverless SQL Pool` implementing the pattern on Azure. The Pool can natively query the data stored in Azure Data Lake, Cosmos DB, or Dataverse with 2 query modes:

- A pure ad-hoc query where you only have to set the endpoint of the dataset. This technique uses the `OPENROWSET(BULK ⋯)` function to transform Apache Parquet or Delta Lake files into Synapse's internal row format. It also supports other data stores queryable from Synapse, such as Cosmos DB.
- An ad-hoc query against previously created external data sources. Ad-hoc queries are great for data exploration tasks but Synapse can also run regular queries executed within an ETL or ELT

process. However, in that approach, it's recommended to work on external data sources with a defined schema.

Links:

1. https://docs.microsoft.com/en-us/azure/synapse-analytics/sql/on-demand-workspace-overview
2. https://docs.microsoft.com/en-us/azure/synapse-analytics/get-started-analyze-sql-on-demand
3. https://docs.microsoft.com/en-us/azure/synapse-analytics/sql/develop-openrowset
4. https://docs.microsoft.com/en-us/azure/synapse-analytics/sql/query-cosmos-db-analytical-store?tabs=openrowset-key

Cloud details GCP

BigQuery

Like Synapse, BigQuery is also a well-known data warehouse solution. However, thanks to its serverless character, it can also be considered as an implementation of the serverless SQL service pattern.

To query the data, you have 2 choices:

- Bring it to BigQuery by creating the datasets and tables.
- Use one of the available external data sources (Bigtable, Cloud SQL, GCS, Google Drive). For example, a GCS can be:
  - A temporary table. In this configuration, the query directly runs against GCS objects and doesn't create any long-living table in the BigQuery dataset.
  - An external and persistent table. In this mode, the query runs against a persistent BigQuery table referencing a GCS location

BigQuery doesn't require any infrastructure to set up. It simply transforms your SQL queries into the physical execution plan.

Links:

1. https://cloud.google.com/bigquery/external-data-sources
2. https://cloud.google.com/bigquery/docs/interacting-with-bigquery
3. https://cloud.google.com/bigquery/external-data-cloud-storage

Shuffle service

Problem

One of your data processing pipelines relies on autoscaling. However, it struggles to decommission idle nodes from the cluster because they store the intermediate files (shuffle files) needed by the job. These shuffle files must be there to avoid rerunning previously executed successful stages in case of a retry. You want to remove this storage dependency to make your cluster resources more flexible.

Use an external storage for the intermediate files. That way the compute architecture is decoupled from the intermediate files storage. Since these intermediate files are often used in the context of a "shuffle" action on the cluster, we often talk about using a *shuffle service*.

Mechanisms

1. The cluster provides a shuffle data storage service.
2. The tasks write the intermediate data to the dedicated service instead of their local disks.
3. The job uses the intermediate data from the dedicated service, and the cluster can remove any unnecessary nodes in the next steps of the execution.

Use for

Scalability

The component decouples cluster resources from the fault-tolerance mechanism. That way the job can scale more easily, without compromising the resilience.

Smaller hardware requirements

The compute nodes don’t act anymore as the intermediate data storage services. They can have smaller disks attached.

Look at

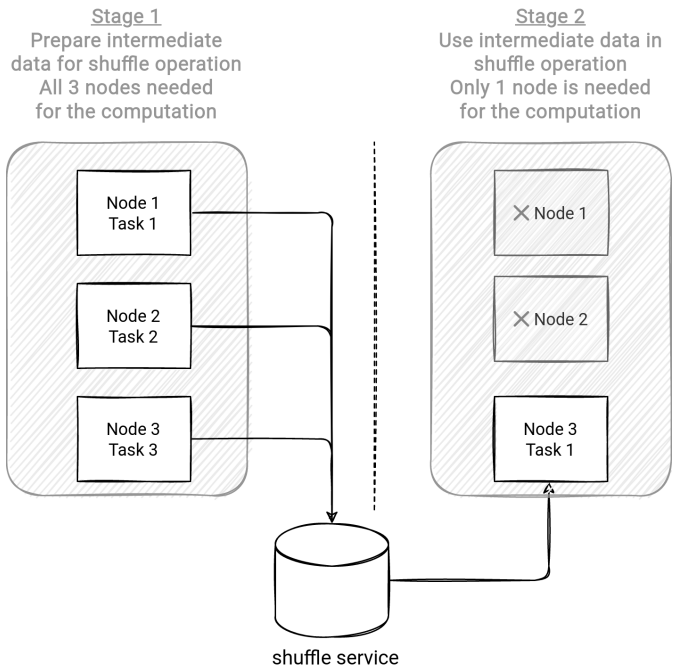
Costs

The shuffle data service can be an extra billable component. It might be worth checking whether it’s not more expensive than keeping the nodes idle in the cluster.

Performance

Writing data across the network can be slower than doing it locally.

Schema design



Cloud details AWS

EMR

EMR automatically enables `Shuffle Service` and Dynamic Resource Allocation (DRA) for Apache Spark jobs on YARN.

The DRA is an Apache Spark feature capable of adding and removing executors (worker process) at runtime, depending on the job progress. If the job has some idle tasks and compute power available, it'll tend to allocate some of the tasks to the unused capacity.

Thanks to the Shuffle Service, if the executor remains idle for too long, the cluster manager can destroy it without worrying about the intermediary files storage. The feature can be used with `EMR Managed Scaling` to enable automatic creation or removal of the nodes on the cluster.

Links:

1. https://aws.amazon.com/blogs/big-data/spark-enhancements-for-elasticity-and-resiliency-on-amazon-emr/
2. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.html#spark-defaults
3. https://aws.amazon.com/blogs/big-data/introducing-amazon-emr-managed-scaling-automatically-resize-clusters-to-lower-cost/

Cloud details Azure

Synapse

In September 2020, Azure announced support for a rewritten shuffle service. As per the announcement note, the service is intended to be more performant for modern hardware and operating systems. However, the feature is still in Preview, and there is not a lot of extra technical information about the concrete optimizations.

Links:

1. https://azure.microsoft.com/en-gb/updates/apache-spark-for-azure-synapse-incluster-caching-and-shuffle-service-preview/

Cloud details GCP

Dataflow

Dataflow jobs can use a component called `Dataflow Shuffle`. The component impacts the execution of the shuffle operations like `GroupByKey`, `CoGroupByKey`, and `Combine`. When enabled any shuffle operation on a Dataflow doesn't use the worker CPU, memory, and disk.

Consequently, Dataflow Shuffle reduces the Persistent Disk storage needs and the CPU and memory consumption of the worker's VMs. Decoupling the storage from the compute also helps the Dataflow service scale without compromising the job fault-tolerance semantics. Moreover, it improves the fault-tolerance because the service can replace any unhealthy node without failing the whole pipeline.

The shuffle component is enabled by default for all batch jobs.

Links:

1. https://cloud.google.com/blog/products/gcp/introducing-cloud-dataflow-shuffle-for-up-to-5x-performance-improvement-in-data-analytic-pipelines
2. https://cloud.google.com/dataflow/docs/guides/deploying-a-pipeline#dataflow-shuffle

Dataproc

Dataproc can delegate the shuffle data storage with the `Enhanced Flexibility Mode`. When enabled, the workers write all the intermediate data to the primary worker nodes of the cluster so that the cluster manager can remove any secondary worker node at any time.

The service provides 2 flexibility modes:

- Primary worker recommended for and only available to Apache Spark jobs.
- Hadoop Compatible File System recommended for jobs with small amounts of data but not adapted for bigger jobs.

In order to take advantage of the Enhanced Flexibility Mode, the cluster must have secondary workers and not use auto-scaling on the primary workers because it could lead to shuffle data loss.

Links:

1. https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/flex

Single-tenant client

Problem

The available throughput of your data source is shared between 3 existing consumers. You're planning to add 2 new ones and are worried about a negative impact on the latency or throughput errors. These unexpected errors might lead to various situations like degrading data freshness or the inability to process the data. You want to solve the issue by reserving a throughout capacity for some of the most important consumers.

Use the single-tenant client pattern that reserves a reading and/or writing capacity to a particular application. Thanks to this dedicated bandwidth, it won't experience any unexpected behavior related to the new clients interacting with the data source.

Mechanisms

1. Create a dedicated consumers group, or a consumer if the data source doesn't support the pattern.
2. Allocate the data source throughput exclusively to the consumer group.

Use for

Prioritizing workloads

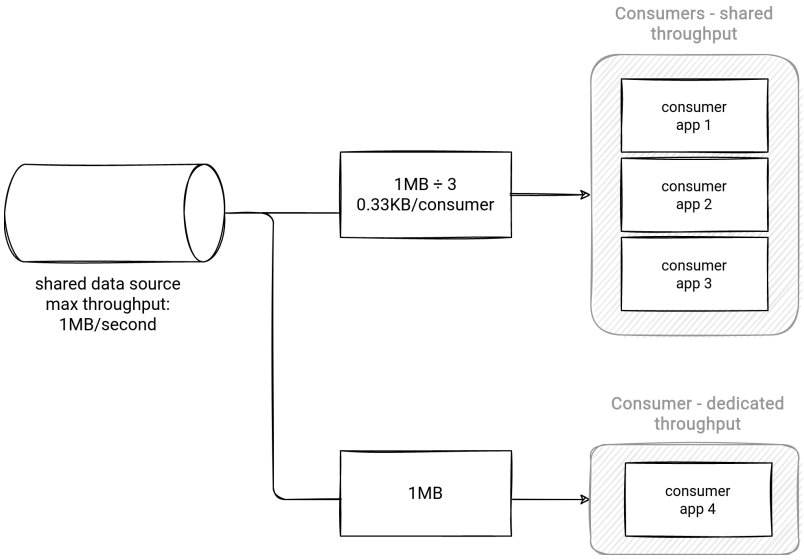
The dedicated throughput can be understood as a definition of the criticality of the consumers in the system. Use it to favor one client over another.Predictive SLA. Since the throughput is not shared the throughput becomes more predictable. A shared throughout rarely guarantees an even distribution of the resources all the time which makes it hard to provide a stable processing time.

Look at

Costs

A dedicated throughput often means an extra billed I/O capacity. Doing that for every consumer might not be the best idea from this standpoint.

Schema design



Cloud details AWS

Kinesis Data Streams

By default, all Kinesis client apps share the same throughput of 2 MB/second/shard for reads and 1 MB/second for writes. In the case of many concurrent operations, some of them may encounter the throughput exceeded exceptions.

The service doesn't support the throughput distribution. Instead, it overcomes this issue with an `Enhanced Fan-Out`. This feature opens a dedicated connection for a given consumer group to guarantee individually allocated compute capacity.

Links:

1. https://aws.amazon.com/blogs/aws/kds-enhanced-fanout/
2. https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/kinesis/model/ProvisionedThroughputExceededException.html

Cloud details Azure

Event Hubs

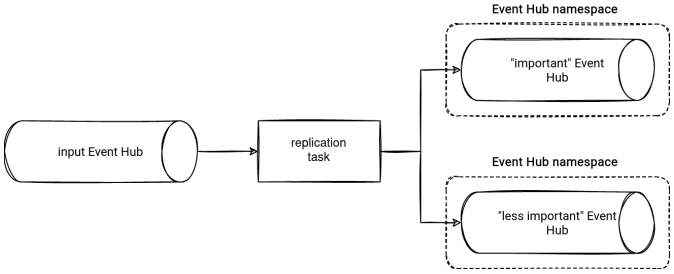
Although it's not possible to dedicate a throughput to a consumer group within an Event Hubs namespace, the service provides an alternative way to implement the pattern by leveraging the available tiers.

Event Hubs has a `Premium` tier that, despite a multi-tenant PaaS character, comes with resource isolation enabling more predictable high-throughput and low latency workloads. Although it doesn't reserve the resources to a specific consumer group, it provides compute, memory, and storage reservation for the namespace to reduce the noisy neighbor risk from other Azure users.

The Premium tier is meant to be a more cost-effective option for mid-range (<120MB/sec) throughput requirements than the `Dedicated` tier, which is by the way another tier available for the pattern. It provides a single-tenant environment, meaning that you get your resources from a fully dedicated pool. Thanks to this physical isolation, it has a better throughput and can handle more

consumer groups (1000 vs 100 in Premium). It's also not limited to the classical 1 MB/second (write) and 2 MB/second (read) I/O. Because of that the Dedicated tier is designed for the most demanding streaming needs, and it's the single one providing 99.99% SLA.

Alternatively, the pattern could be also implemented as a combination of a replication task and multiple Event Hub namespaces.



Links:

1. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-dedicated-overview
2. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-quotas
3. https://techcommunity.microsoft.com/t5/messaging-on-azure-blog/announcing-azure-event-hubs-premium-public-preview/ba-p/2382082
4. https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-federation-configuration

Small data - batch services

Problem

You're working as a data engineer in the context of small data. You don't want to use a data processing framework. Instead, you are looking for a cloud-native solution that can run jobs from Docker images.

Mechanisms

1. Write your data processing logic application.
2. Build the application package.
3. Compose the Docker image with the built package. Deploy the image to a container registry.
4. Define the compute characteristics of the job, like vCPU, memory, or the node type.
5. Submit the job to the batch service by referencing the Docker image in job parameters and the hardware requirements.

Use for

Small data processing

If the dataset fits into a single machine and you don't need to leave the door open for Big Data processing, working with batch services can be the best fit. It doesn't require any Big Data-specific knowledge and let you use any language that can be deployed in a container.

Faster team setup

Depending on the team topology, learning a data processing framework can require extra effort. Using well-known locally running solutions can be an interesting alternative if there are no plans to work on Big Data in the future.

Running distributed processing framework in local mode

Instead of using a data processing service as in the Small data - single node cluster pattern, you can also package your application and deploy it in local mode on the batch service.

Look at

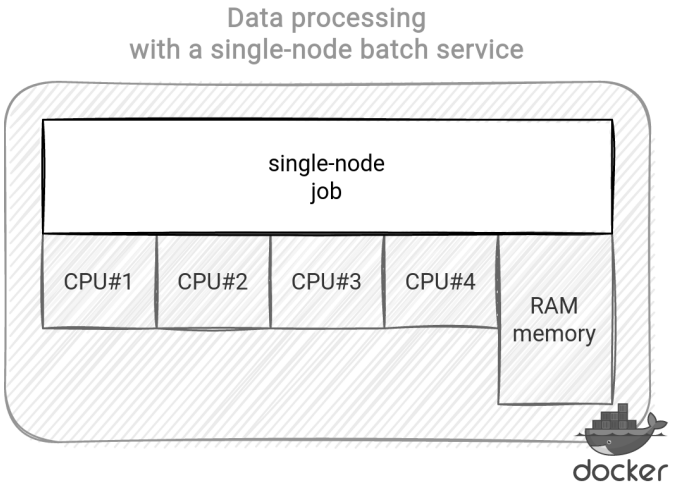
Horizontal scalability

Even though the batch service might support horizontal scalability by enabling cluster mode, adapting the job can be challenging. It would require writing a custom code to handle the parallelism, the dataset splitting and its distribution.

Costs

Depending on the memory and time execution constraints, you may opt for a serverless environment using a physical node. The serverless approach can be cheaper and easier to set up.

Schema design



Cloud details AWS

Batch

The Batch service implements the pattern on AWS. Besides the Docker image runtime introduced in the problem statement, the service supports running shell scripts and Linux executables.

The main executable component is called `job`. The job is described by a `job definition` specifying all required compute resources, like an EC2 (Virtual Machine) type or the amount of vCPU and memory. Optionally, if you decide to upgrade the single-node job to the multi-node parallel job configuration, you can configure the desired number of nodes.

Once defined, the job gets submitted to a `job queue` where it waits for being scheduled on an available `compute environment`.

Links:

1. https://docs.aws.amazon.com/batch/latest/userguide/what-is-batch.html
2. https://docs.aws.amazon.com/batch/latest/userguide/multi-node-job-def.html

Cloud details Azure

Batch

Also, Azure has its Batch service that you can use to implement the pattern. During the setup process, you have to configure the `pool of compute nodes` with the following parameters

- the OS for the nodes (Windows, Linux)
- nodes size (VM type)
- nodes number

As you can already see through this configuration, the service can execute single-node and multi-node workloads.

Like for AWS Batch, the workload is represented by a `job` abstraction with a unique ID, associated pool, and `tasks`. Each task is a data processing command that the service runs on the compute nodes. In the case of a distributed job, each task executes a different operation, for example, processing a separate part of the input dataset.

Links:

1. https://docs.microsoft.com/en-us/azure/batch/batch-technical-overview
2. https://docs.microsoft.com/en-us/azure/batch/quick-create-portal

Small data - single-node cluster

Problem

The data flowing into your system can be processed on a single node. However, you want to leave the door open for the future Big Data scenarios. That's why you decided to use an Open Source distributed data processing framework. You're looking for a way of deploying it in the most cost-efficient manner.

Mechanisms

1. Create a data processing cluster with 1 node. It'll often be the master node.
2. Set the job execution mode to local so that the scheduler won't try to create another node.
3. Submit the job to the cluster.

Use for

Sharing the technical solution for Big and small data scenarios

The distributed data processing framework works in local mode for small data and in distributed mode for Big Data use cases. You can then write the code only once and adapt it to the distributed execution by changing the cluster configuration and the job execution mode.

Low-cost data exploration

You could also use the pattern to perform some initial data exploration on a small subset of data. It should be cheaper than distributed alternatives.

Small data workloads

Even though these workloads won't grow to Big Data, you might still want to use the same technical data processing framework as for bigger volumes of the data.

Look at

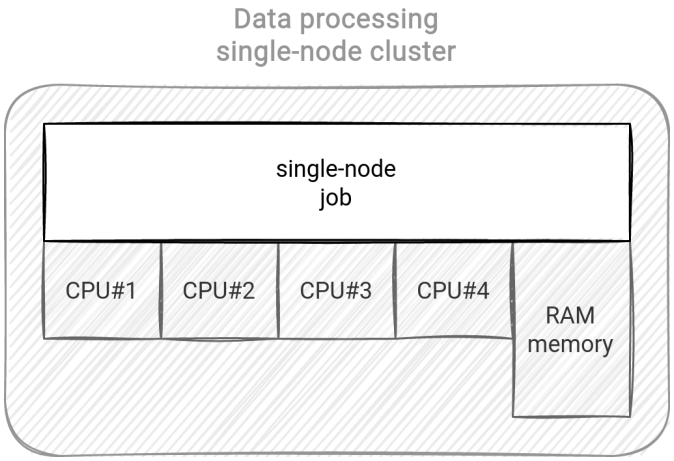
Switch to distributed mode

The small data cluster is composed of a single node and it doesn't support auto-scaling. Changing the execution mode from local to distributed would require using a completely new cluster.

Compromised fault-tolerance

In case of an unhealthy node, the whole job will fail because there is no way to replace the single node and reschedule the failed tasks. The execution context is lost at the failure.

Schema design



Cloud details AWS

EMR

An EMR cluster comprises master, core, and task nodes. To implement the pattern You will need to create the cluster with the master node only.

Links:

1. https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-overview.html
2. https://docs.aws.amazon.com/cli/latest/reference/emr/create-cluster.html

Cloud details Azure

Databricks

Databricks implements the pattern from a special cluster mode called `Single Node`. As the name indicates, the service will create a single-node cluster where the node acts as a driver (coordinator) and worker (jobs runner) at the same time.

Links:

1. https://docs.microsoft.com/en-us/azure/databricks/clusters/single-node

Cloud details GCP

Dataproc

To implement the pattern in Dataproc, you must enable the `single-node` option while creating the cluster. The command will initialize the cluster only with one node acting as both master and worker.

Links:

1. https://cloud.google.com/dataproc/docs/concepts/configuring-clusters/single-node-clusters

Streaming serverless SQL processing

Problem

One of the teams in your company has limited programming skills. Its members are more comfortable with SQL than with Java or Python. Besides, they don't know how to manage the clusters. They have to implement the first real-time job and are looking for some SQL-based and serverless solutions.

Mechanisms

1. Define the data source as a streaming data store. Optionally, you can combine it with a data at-rest storage to enrich the streaming data.
2. Define the data sink.
3. Write the SQL transformation code reading data from the sources and writing it to the sink.
4. Deploy the query and the input-output configuration to the serverless service.

Use for

Declarative pipelines

You can define the data transformation logic by leveraging a query language as SQL.

Data exploration and debugging

Serverless SQL query services are also useful for any ad-hoc querying scenarios. They can help in debugging or data discovery use cases.

Migrating batch jobs

The SQL support on a streaming service opens an easy possibility to migrate the SQL-based batch jobs. Of course, the migration should consider all streaming shortcomings, such as late or duplicated data, but still, it can be easier than rewriting the job in a whole new technology.

Look at

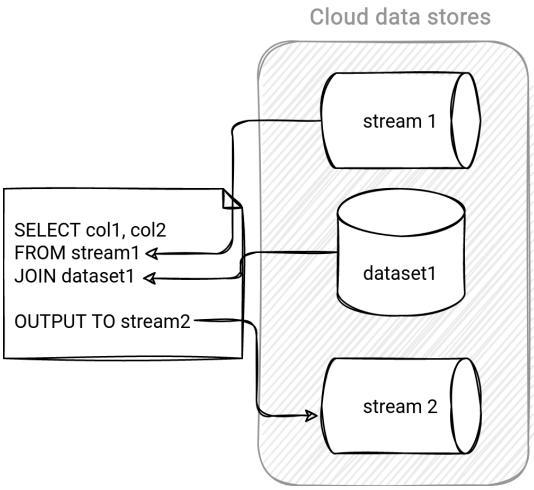
Implementation limits

Not all scenarios are easy to implement in SQL. You can encounter difficulties with dictionary lookups, sequences mapping, or querying external HTTP services.

Extendibility

The fully managed and serverless runtime can be hard to extend. It'll often be limited to the data stores present on a given cloud provider.

Schema design



Cloud details AWS

Kinesis Data Analytics for SQL

`Kinesis Data Analytics` is a serverless streaming processing service with a programmatic API based on Apache Flink and a SQL API implementing the discussed pattern.

The SQL API is composed of:

- An `in-application stream`. It's the place storing the data to transform.
- One or more transformations. Each `SELECT` action executes in the context of an `INSERT` statement. Put another way, data projection adds the output to a different `in-application stream`.
- A `pump`. This operation is required to insert data from one in-application stream to another.

Therefore, to process the data, you will have to issue the following commands:

- `CREATE OR REPLACE STREAM ⋯` to define the final or intermediary stream.
- `CREATE OR REPLACE PUMP ⋯ AS INSERT INTO ⋯` to specify the pump that will move the data of the input stream to the final or intermediary stream.
- `SELECT STREAM ⋯ FROM ⋯` to define the data transformation logic working on the streams.

The service also supports reference datasets from data-at-rest storage like an S3 bucket. They're often used to enrich the streaming data.

Links:

1. https://docs.aws.amazon.com/kinesisanalytics/latest/dev/what-is.html
2. https://docs.aws.amazon.com/kinesisanalytics/latest/dev/how-it-works-input.html
3. https://aws.amazon.com/blogs/big-data/writing-sql-on-streaming-data-with-amazon-kinesis-analytics-part-1/

Cloud details Azure

Stream Analytics

Azure implements the serverless SQL streaming pattern with the `Stream Analytics` service.

To write a Stream Analytics job, you need to follow these 3 steps.

1. At the service level, configure the input data source, like an Event Hubs stream.
2. Still at the service level, set the output, which can be a Storage Account, a streaming, or messaging service (Event Hubs, Service Bus). You can also pass the processed data to a compute service like Azure Functions or a BI tool like Power BI!
3. To terminate, you will need to define the transformation query following the pattern: `SELECT ${my_output} INTO ${my_input} FROM`. The query will read the data from the input data source and write it to the output configured in the previous step.

As Kinesis Data Analytics SQL, Stream Analytics also supports joining in-motion and at-rest datasets.

Links:

1. https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-introduction
2. https://docs.microsoft.com/en-us/azure/stream-analytics/quick-create-azure-cli

Targeted data retrieval from files

Problem

A consumer wants to read a specific range of the file stored on an object store. The consumer wants to avoid downloading the whole file before accessing this range locally.

Mechanisms

1. Identify the range to fetch. The data can be stored aside, for example, in a file mapping the logical to the physical representation of the dataset.
2. Include the range in the query sent to the object store; most of the time, you will use the HTTP Range header for that.

Use for

Targeted data read

There is no need to download the whole file and apply the filtering logic to get only the valuable data. If the consumer knows where the useful data is, it can delegate this filtering responsibility to the service by specifying the bytes range to read.

Parallel reading

A typical parallel reading approach would download the whole file first and divide it among the local threads for concurrent processing. With the targeted data retrieval pattern, these concurrent processes can directly run separate chunks without the prior need to download the whole content.

Look at

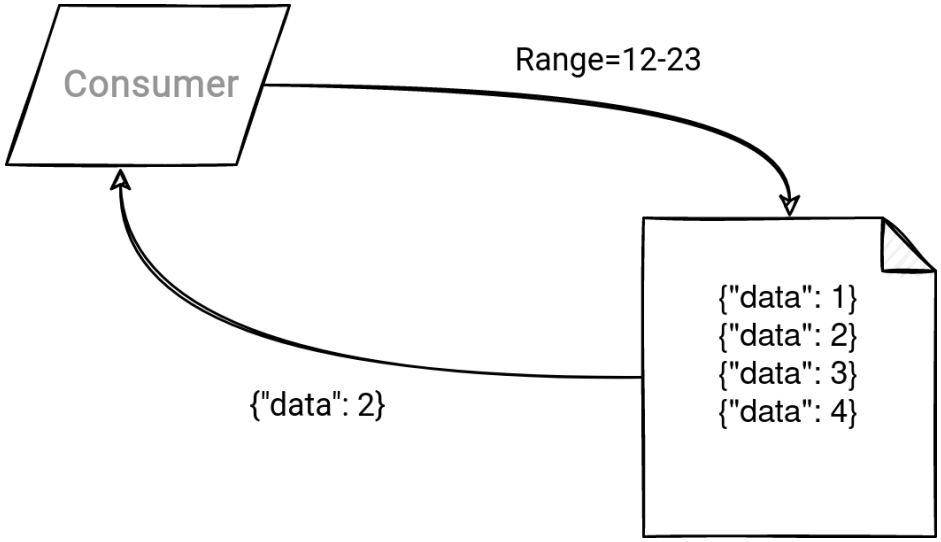
Range identification

Identifying a range can be easy for fixed-length rows. Otherwise, it might require an extra index mapping the logical row id with its physical location in the data file.

Invalidating conditions

The service implementing the pattern can also provide a transcoding feature that will decompress files before returning them to the client. In that case, it can ignore the range parameter because it might be difficult to get the data without decompressing the file beforehand.

Schema design



Cloud details AWS

S3

The pattern is present mainly in the object stores. S3 supports it with the `Range` HTTP header. If defined, the service will return the data included in the specified range. S3 accepts only one range expression per request.

In addition to the `Range` header, the service supports other HTTP headers that the consumers can use to control the downloaded data:

- `If-Match` and `If-None-Match` to return the object only if its entity tag (ETag) is the matches or not the ETag from the header.
- ` If-Modified-Since` and `If-Unmodified-Since` to return the object only if it was modified or not since the specified time.

Links:

1. https://docs.aws.amazon.com/AmazonS3/latest/API/API_GetObject.html#API_GetObject_RequestSyntax
2. https://docs.aws.amazon.com/whitepapers/latest/s3-optimizing-performance-best-practices/use-byte-range-fetches.html
3. https://docs.aws.amazon.com/AmazonS3/latest/API/API_GetObject.html#API_GetObject_RequestParameters

Cloud details Azure

Blob Storage

On Azure, you can fetch a specific part of the blob with the same mechanism as on S3. So if you want to get only a particular part of the blob, you have to define it in the `Range` HTTP header. In addition to this standard header, the service also accepts a custom `x-ms-range` header. It can be helpful to overcome the 32-bit integer limitation for some clients, like .NET.

The service also supports additional *skipping* headers, such as `If-Match`, `If-None-Match`, `If-Modified-Since` and `If-Unmodified-Since`. They have the same meaning as for S3.

Links:

1. https://docs.microsoft.com/en-us/rest/api/storageservices/specifying-the-range-header-for-blob-service-operations
2. https://docs.microsoft.com/en-us/rest/api/storageservices/specifying-conditional-headers-for-blob-service-operations

Cloud details GCP

GCS

Finally, GCS also supports the `Range` HTTP header. However, it's silently ignored for the transcoded files because it's not possible to select a range of compressed data without decompressing it first. Hence, the service always returns the whole file to the consumer in that case.

The service also supports additional *skipping* headers, such as `If-Match`, `If-None-Match`, `If-Modified-Since` and `If-Unmodified-Since`. They have the same meaning as for S3.

Links:

1. https://cloud.google.com/storage/docs/xml-api/get-object-download
2. https://cloud.google.com/storage/docs/transcoding#range

About the author

Bartosz Konieczny is a data engineer by day. In the last years he has been specializing in multi-cloud data engineering by implementing Big Data pipelines on top of AWS, Azure and GCP for various industries (broadcasting, online video sharing platform, oil, worldwide dating app). He is also a certified data engineer for these cloud providers.

By night he shares his knowledge on cloud data services, Apache Spark and data engineering on waitingforcode.com, becomedataengineer.com, and conferences like Spark+AI Summit or Data+AI Summit. He is also part of the first group of Databricks Beacons.

You can reach him on Twitter or at book@waitingforcode.com

Index

A

autoscaling, 15
AWS Athena, 68
AWS Batch, 77
AWS DynamoDB, 9, 42
AWS EMR, 16, 18, 26, 71, 80
AWS Glue
  Data Brew, 39
  Job, 65
AWS Kinesis
  Data Analytics, 82
  Data Streams, 13, 46, 54, 57, 61
    enhanced fan-out, 57
  Firehose, 9
AWS Kinesis Data Streams, 8, 74
  enhanced fan-out, 74
AWS Lake Formation, 65
AWS Lambda, 22, 29, 36
AWS Redshift, 18
AWS S3, 51, 85
AWS SQS, 22, 32
Azure Batch, 77
Azure Blob Storage, 51, 85
Azure Cosmos DB, 10, 19, 42
Azure Data Factory, 39, 65
Azure Databricks, 26, 80
Azure Event Grid, 23, 58
Azure Event Hubs, 9, 13, 47, 54, 57, 61, 74
  Capture, 10
  tiers, 74
Azure Functions, 30, 36
Azure Service Bus, 23
Azure Stream Analytics, 32, 83
Azure Synapse, 19, 68, 71

E

event-driven, 28

G

GCP BigQuery, 19, 33, 69
GCP BigTable, 11
GCP Cloud Functions, 30
GCP Data Fusion, 40
GCP Dataflow, 11, 24, 33, 66, 72
GCP Dataproc, 16, 27, 72, 80
GCP Firestore, 43
GCP GCS, 52, 86
GCP Pub/Sub, 10, 14, 24, 48, 55, 59, 62

I

idempotency, 31

N

No Code, 38
NoSQL, 41

O

object store, 50, 84

S

serverless, 28, 67, 81
streaming
  broker, 60
  consumer, 12, 53, 56, 81
  producer, 31