Dataflow - Michaël Bettan

Definition: Apache Beam managed service for unified stream and batch data processing patterns. Automated infrastructure provisioning and cluster management with horizontal autoscaling.	**Availability**: Regional	**SLA**: 99.9%

Use Cases:
- **Streaming Analytics** for Business insights
- **Real-time AI** for Predictive Analytics or Fraud Detection
- **Sensor and log data process** for System Health
- **ETL pipelines** in general

Do you have an existing Hadoop/Spark ecosystem workload to bring in? Are you open to a modern pipeline development SDK?
→ Dataflow is a great fit (otherwise DataProc if no to both)

Billing: Billed in per second increments, on a per job basis:
- **Dataflow Compute resources**
  - Worker vCPU and Memory
  - Shuffle data processed
  - Streaming Engine Compute Units
- **Dataflow Prime Compute resources**
  - Data Compute Units (DCUs)
- **Other resources**: Persistent Disk, GPUs, Snapshots, Confidential VM

Infrastructure Model

- ***Dataflow Jobs** could be one of the two types **batch** (bounded set of data) or **streaming** (unbounded set of data)*
- *Dataflow Jobs could be invoked via:*
  - ***Predefined Template** (pre-built in Java SDK)*
  - ***Custom Template** (Java, Python, Go or mixed SDK)*
  - ***Dataflow SQL** via BigQuery (depreciated)*
- ***Runner** to execute your pipeline on a backend of your choice such as Apex, Flink, Spark, Dataflow.*
- ***Workers:** compute engine machines execute the data processing with horizontal autoscaling (visible in UI).*
- ***Horizontal Auto Scaling** appropriate number of worker instances for your job, adding or removing workers as needed*

Separating Compute and Storage

- ***A shuffle** is a Dataflow-based operation behind transforms such as GroupByKey, CoGroupByKey, and Combine.*
- ***Dataflow Shuffle service** (batch pipelines only) moves the shuffle operations out of the worker VMs and into the Dataflow service backend, lead to faster execution time*
  - *Reduced consumed resources (CPU, memory, disk) and better auto scaling and fault-tolerance*
- ***Streaming Engine** moves parts of pipeline execution out of the worker and into the service backend, improving autoscaling and data latency.*
  - ***Offloading state storage**: Instead of storing **windowed state** on disks attached to worker VMs, it manages this state in a dedicated service.*
  - ***Optimized shuffling**: Improves the efficiency of data shuffling (redistribution)*
- ***FlexRS** is suitable for Non-time-critical workloads, like daily or weekly jobs with flexible completion times.*
  - *Advanced scheduling in Dataflow Shuffle service.*
  - *Mix of preemptible and on-demand VMs*
  - *Jobs are queued and executed within 6 hours*

Beam SDK Model

- ***Programming Model**: defines a data processing pipeline executed by distributed processing backends.*
- ***Pipeline**: specifies the entire set of operations (reading, transforming, writing), including input data, transformations, and output destinations.*
- ***PCollection**: a distributed, immutable dataset processed within a pipeline. Each transformation creates a new PCollection. Parallel processing supported.*
  - ***Element**: A single data entry (e.g., table row)*
- ***Transforms** **(PTransform)**: operations on PCollections, such as filtering, grouping, or joining data. Each produces a new PCollection.*
  - ***ParDo**: core transform for **parallel processing**. Applies user-defined functions to each element (e.g., filtering or transforming each item).*
  - ***Aggregation**: combines elements, usually by grouping and reducing (e.g., sum or count values)*
  - ***Side Inputs**: Extra data used in parallel processing*
  - ***CoGroupByKey**: Joins PCollections by key, similar to database joins.*
  - ***Flatten**: Merges multiple PCollections into one.*
  - ***Side outputs**: allow a transform to produce multiple outputs, such as splitting data based on a condition*
- ***Pipeline I/Os** are transforms that read from a source and write to a sink.*
- ***User-defined functions (UDF):** custom code typically used in ParDo for element-specific operations.*

Dataflow Prime

- **Vertical autoscaling**: scale workers **memory capacity**
  - Prevents out-of-memory errors.
  - Ensures efficient use of resources.
  - Eliminates the need to specify worker types.
- **Right fitting**: stage specific worker pools using resource hints
  - Uses resource hints (like GPU and RAM) to specify worker resources
  - Prevents over- or under-utilization of resources
  - Improves efficiency and avoids performance bottlenecks
- **Job Visualizer:** see the performance of a Dataflow job and optimize the performance of the job by finding inefficient code, including parallelization bottlenecks

- **Smart Recommendations:** optimize and troubleshoot the pipeline based on the recommendations provided in the Diagnostics tab of the job details page

Security Access Controls

- *Project-wide access (all or nothing). Pipeline data separate*
- ***Dataflow Admin**: creating and managing jobs.*
- ***Dataflow Developer** view, update, and cancel jobs (no access to the underlying data)*
- ***Dataflow Viewer:** read-only access to all resources.*
- ***Dataflow Worker:** execute work units for a pipeline*

Windows

Dividing unbounded data streams into logical chunks for processing and aggregation. **Windowing divides data into time-based, finite chunks**. Essential when doing aggregations using Beam primitives (GroupByKey, Combiners).

- **Processing time**: timestamp when the event is **processed** by the pipeline (same as micro-batching)
- **Event time:** timestamp of when an event **actually occurred** (more complex aggregation logic)

Windowing Types:

- **Tumbling Window**: non-overlapping, consistent and **fixed-size** time intervals (e.g., minute, hourly, daily).
  - **.withAllowedLateness** allows late data within a certain time
- **Sliding Window** (Hopping): Overlapping windows that move forward by a set interval (e.g., a 60-second window sliding every 30 seconds)
  - **Period**: how often a new window begins, influencing the overlap between consecutive windows.
  - **Duration**: length of time each window covers, determining how much data is grouped together.
  - Example: Calculate moving average
- **Session Window**: dynamically determined based on periods of activity followed by inactivity (gaps)
  - Start when data arrives after a gap and end when the specified gap duration is reached
  - Data dependent (not known ahead of time)
  - Examples: user session, user navigation, etc.

Default Behavior: single global window (not practical)

Watermarks

Relationship between the processing timestamp and the event.

Context: Windowing requires deciding when to finalize results, especially with **late-arriving data** → Watermarks required.

Definition: **Estimation of the oldest unprocessed data.** It represents the system's understanding of how complete the data stream is up to a certain point in time. It's the pipeline's expectation of when future data will arrive.

Lag time: difference between when the data was expected and when the data is actually arriving.

Way to track progress in unbounded data. Represents a point in time or a specific offset within a stream

- All data before the watermark has (likely) arrived
- Allows downstream operators to perform time-based operations like windowing
- Event time (based on data timestamps) or Processing time (system clock).
- Ensure consistent and repeatable results
- Tolerates late-arriving data to a certain extent, defined by lateness configurations.
- Stateful operations like aggregations within time windows.
- Handling out-of-order data in real-time applications.

A watermark is a **threshold** that indicates when expecting all of the data in a window to have arrived. If new data arrives with a timestamp that's in the window but older than the watermark, the data is considered late data.

Late Data Identification: data arriving before the watermark is **early**, at the watermark is **on-time**, and after the watermark is **late**.

The default behavior is to **drop late data.**

Watermark Estimation: Dataflow estimates the watermark based on the **oldest unprocessed event time**. This estimate continually updates with each new message.

Data Freshness: metric relates to the input data's watermark. It indicates the **lag between real-time and the oldest unprocessed message** (the watermark). A lower data freshness value indicates more up-to-date processing.

System Latency: measures the time it takes to fully process a message, including any waiting time in the input source.

Triggers

Determine when results are **emitted** for a window.

Default Trigger: afterWatermark. Fires when the watermark passes the end of the window. Often leads to high latency.

Trigger Types:

- **Event Time**: Based on the timestamps of the data elements. Examples: afterWatermark, afterPane.
- **Processing Time:** Based on the worker's clock. Example: afterProcessingTime. Less accurate for event order but provides predictable results emission.
- **Data Volume**: Based on the number of elements in the window. Example: afterCount.

Late Data Handling:

- **allowedLateness**: specifies how long to wait for late data after the watermark. Crucial for accurate results.
- Triggers can fire again for late data, updating previous results.

Accumulation Modes:

- **Accumulating**: All data within the window is included in each trigger firing's result. Can be resource intensive for large windows. Good for accurate sums and counts across the entire window.
- **Discarding**: Only new data since the last trigger firing is included in the result. More efficient for large windows. Suitable for associative and commutative operations (e.g., sums, averages). Requires downstream aggregation if a complete, final result is needed. The transcript emphasizes the performance benefits of discarding for wide windows.

Key Considerations:

- **Latency vs. Completeness**: Choose windowing strategy and trigger based on your application's requirements.

Processing time triggers offer lower latency but sacrifice event-time accuracy.

- **Accumulation mode impact**: Carefully consider the accumulation mode, especially for large windows and stateful operations. Discarding mode can greatly improve efficiency.
- **Late Data is important**: Leverage allowedLateness to handle late data gracefully. Even with non-afterWatermark triggers, the watermark still defines lateness.
- **Downstream aggregation**: If using discarding mode, plan for how you'll aggregate partial results downstream to get a final, complete answer.

Best Practices

- ***Handle Pipeline Errors**: Try-Catch block with a separate side output to a new PCollection to send to another collector (Pub/Sub). Recycling the bad data from the PCollection.*
- ***Secure Pipeline**: Disable public IPs to restrict access to internal systems + VPC service controls to mitigate the risk of data exfiltration + CMEK (Customer-Managed)*
- *Dataflow workers should access Google services, w/o external IPs, using **Private Google Access** or **VPC service controls**. Flag: --enable-private-ip-google-access*

Pipeline Update

- ***Batch pipeline** is not supported (Streaming only is)*
- *Major changes to windowing or triggers → unpredictable*
  - *Stop pipeline with drain for inflight data*
- *Minor changes to windowing fn → Live update with JSON map*
- ***JSON transform mapping** maps the named transforms in your prior pipeline code to names in your replacement pipeline code*
- ***Update Pipeline:** switch the data to the new pipeline. Update Job with transform mapping (JSON file).*

Monitoring

- ***Dataflow UI**: provides insights into watermark values through data freshness and latency metrics, allowing for diagnosing pipeline bottlenecks and input rate fluctuations.*
- ***Dataflow inline monitoring:** directly access job metrics to troubleshoot pipelines at both the step and the worker level.*

Miscellaneous

- ***Graph optimization**: Dataflow optimizes the graph execution by **fusing operations efficiently** and by not waiting for previous steps to finish before starting a new one unless there is a dependency involved.*
- ***Fusion** optimizes pipeline execution by merging multiple steps into single steps. This prevents the materialization of intermediate data, saving memory and processing overhead. While preserving data dependencies, Dataflow may reorder or fuse steps for optimal efficiency*
- ***Introducing a Reshuffle transform** strategically to prevent fusion. This forces Dataflow to treat the steps before and after the Reshuffle as independent stages, enabling parallel processing of the CSV file reading and subsequent operations, ultimately utilizing more workers and improving performance.*
- ***Portability API** (interoperability layer) enables custom containers and multi-language support*
- ***Drain Option**: Stops a Dataflow job gracefully, allowing it to process existing buffered data before shutting down.*
- ***TCP ports 12345 and 12346** are used by Dataflow workers for internal communication*

Pub/Sub as a Source

***PubsubIO.readMessages():** core transform in Dataflow allows you to read messages directly from Pub/Sub subscriptions.*

Subscription Management Best Practices:

- ***Create subscriptions programmatically**: Your Dataflow pipeline should ideally create its own temporary subscriptions for better control and isolation. Avoids conflicts with other applications using the same topic.*
- ***Acknowledge messages explicitly**: Use .withAcknowledgementDeadline() to manage acknowledgement deadlines and prevent message redelivery due to worker failures. Fine-tune this based on your processing time.*
- ***Scale your subscriptions**: Ensure enough pull subscribers in your Dataflow pipeline to keep up with the incoming message rate. Monitor backlog to detect bottlenecks.*

Pub/Sub as a Sink

- ***PubsubIO.writeMessages():** This transform writes messages to a Pub/Sub topic.*
- ***Idempotent Writes (for exactly-once)**: If your downstream consumers require exactly-once delivery, combine PubsubIO.writeMessages() with unique message IDs and logic in your subscribers to handle duplicates.*

Windowing and Watermarks

- ***Impact of Unordered Messages**: Pub/Sub does not guarantee message order within a topic. Dataflow's windowing helps you group messages based on time or other criteria, but watermarks are critical for handling late data.*
- ***Setting Appropriate Watermarks**: Choose watermark strategies carefully (e.g., perfect, heuristic, custom) considering the potential for **late-arriving messages**. A **too-early** watermark can lead to incomplete aggregations, while a **too-late** watermark delays results.*

Error Handling and Dead-Letter Queues

- *Configure Pub/Sub subscription with **a dead-letter topic***
- ***Dataflow Error Handling:** Use try-catch blocks within your DoFns and .apply(MapElements.via(new MyErrorHandlingFn<>())) style error handling pipelines to catch exceptions during message processing. When an error occurs, publish the failing message to the dead-letter topic with error information as attributes.*

Handling failure and retries

- ***Transient errors:** Rely on automatic retries (default is 4) and exponential backoff.*
- ***Persistent errors:** Use dead-letter queues and analyze the failed data to fix the root cause.*

- ***Specific error handling:*** *Implement try-except blocks within your DoFns for custom logic.*
- ***Always design for idempotence*** *to ensure data consistency.*
- ***DirectPipelineRunner*** *for local testing and debugging, but lacks the optimizations of a distributed runner.*

Exactly-Once Processing

- ***Unique Message IDs****: Assign unique identifiers to your messages (either at the publisher side or within Dataflow).*
- ***Idempotent Sinks****: Design your downstream systems or use idempotent Pub/Sub sink implementations that can handle duplicate messages based on their unique IDs. Dataflow provides built-in support for some sinks.*
- ***Transactions*** *: For more complex scenarios, explore using Dataflow's support for streaming transactions, which can provide atomicity across multiple Pub/Sub operations.*

Service Account

Dataflow service agent: *Dataflow uses the* ***Dataflow service account*** *as part of the job creation request, such as to check project quota and to create worker instances on your behalf. Dataflow also uses the Dataflow service account during job execution to manage the job.*

- ***Shared VPC: Compute Network user role*** *required in the host project on either a project level or a subnet level*

Worker service account*: Worker instances use the worker service account to access input and output resources after you submit your job. By default, workers use the* ***Compute Engine default service*** *account associated with your project as the worker service account.*
Best practice: *specify a* ***user-managed service account***

Networking

- ***Shared VPC*** *allows you to run jobs in a network located in a separate host project. Enables centralized network management by designated administrators while allowing other users to manage instances.*
- ***Network options****: operate in the* ***default network*** *or a* ***custom network*** *with defined regions and subnets.*
- ***Ensure sufficient IP addresses*** *in the subnet for the desired number of Dataflow workers.*
- ***Service account permission****s: The Dataflow service agent needs the* ***Compute Network User*** *role*
- *Requirements for Private IP Usage:*
    - *Resources accessed by the pipeline must be within the same VPC network, a shared VPC network, or a network with VPC network peering enabled.*
    - *If communicating with other services,* ***Private Google Access*** *must be enabled for the worker's subnetwork. Otherwise, services like* ***Cloud NAT*** *are required for internet access.*