Estuary Developers Documentation

Estuary Tech, Inc

Apr 29, 2021

OVERVIEW

Warning: This documentation is deprecated. The new documentation is available at docs.estaury.dev. This repo is archived for reference, but may soon be deleted to avoid confusion.

Estuary Flow unifies technologies and teams around a shared understanding of an organization's data, that updates continuously as new data records come in.

Flow is primarily targeted to backend engineers who must manage various continuous data sources, with multiple use cases and stakeholders. It makes it easy for engineers to turn sources – e.g. streaming pub/sub topics or file drops – into pristine S3 "data lakes" that are documented and discoverable to analysts, ML engineers, and others using their preferred tooling (e.g. via direct Snowflake / Spark integration).

Engineers can then go on to define *operational transforms* that draw from the same data – with its complete understanding of history – to build new data products that continuously materialize into databases, pub/sub, and SaaS. All with end-to-end latency in the milliseconds.

Flow's continuous transform capability is **uniquely powerful**. Build complex joins and aggregations that have unlimited historical look-back, with no onerous windowing requirements, and which are simple to define and evolve. Once declared, Flow back-fills transformations directly from the S3 lake and then seamlessly transitions to live updates. New data products – or fixes to existing ones – are assured of consistent results, every time. The Flow runtime manages scaling and recovers from faults in seconds, for true "hands-free" operation.

Flow is configuration driven and uses a developer-centric workflow that emphasizes version control, composition & re-use, rigorous schematization, and built in testing. It's runtime takes best advantage of data reductions and cloud pricing models to offer a surprisingly low total cost of ownership.

This documentation lives at https://github.com/estuary/docs, and is browse-able at https://estuary.readthedocs.io.

Warning: Flow is currently in release preview. It's ready for local development and prototyping, but there are sharp edges, open issues, and missing features.

Slides (Direct Link)

CHAPTER

ONE

RUNNING EXAMPLES

This documentation is interactive! You can directly open it on GitHub using Codespaces, or you can clone this repo and open using the VSCode Remote Containers extension (see our guide). Both options will spin up an environment with the Flow CLI tools, add-ons for VSCode editor support, and an attached PostgreSQL database for trying out materializations.

```
# Build this documentation repository's Flow catalog.
$ flowctl build

# Run all catalog tests.
$ flowctl test

# Start a local Flow instance and deploy the catalog to it
$ flowctl develop
```

CHAPTER

TWO

GAZETTE

Flow is built upon Gazette. A basic understanding of Gazette concepts can be helpful for understanding Flow's runtime and architecture, but isn't required to work with Flow.

CHAPTER

THREE

TABLE OF CONTENTS

3.1 How Flow Helps

3.1.1 Data Integrity

"My pipeline blew up in production because it thought a field wasn't null-able (and it was)."
"Another team changed their schema, my team didn't realize, and it broke our pipeline."
"An upstream bug caused my pipeline to read and propagate bad data."
"I need to update my pipeline. How can I be confident I won't break downstream use cases?"

Data errors and integrity issues are a common problem within larger teams and organizations. Mistakes are a fact of life, teams don't always communicate proactively, and vendors or partners may start sending you bad data at any time, making *their* bug *your* problem.

While Flow can't prevent every mistake, it's designed to identify and surface integrity issues as quickly as possible – before bad data has a chance to propagate, and often before pipelines are even deployed to production.

- Every Flow document has an associated JSON schema, which is verified every time a document is read or written. Full stop.
- JSON schema makes it easy to express *validation tests* over data within a document. For example, that a latitude is always between -90 and 90 degrees, or that a timestamp string is in the right format.

  You can encode expectations that have typically required entire testing libraries (e.x. Great Expectations), right into the schema.
- You can integrate un-trusted sources using an "Extract Load Transform" pattern, where documents are first written to a collection with a permissive schema – ensuring documents are never lost – and are then verified against a more restrictive schema while being transformed.

  If a validation error occurs, Flow halts the pipeline to provide an opportunity for the schema and transform to be fixed. The pipeline can then be resumed, and while downstream derivations will see a delay, they won't see bad data.
- JSON schema lets you hyperlink directly to the authoritative schema for a data source, as provided by another team or a vendor. Upstream schema changes are automatically picked up and verified with every catalog build.
- Flow lambdas are strongly typed, using TypeScript types which are drawn from schemas. This allows Flow to catch an enormous range of common bugs – such as typos, missing null-able checks, and more – at catalog build time.
- Flow makes it easy to write *contract test* that verify the integrated, end-to-end behavior of multi-stage processing pipelines. Tests can be run during local development and within a CI system (e.x. Github Actions or Jenkins).

3.1.2 Historical Replays

"I have to back-fill my new streaming pipeline, with replayed historical data"
"We messed up a dataset or pipeline, and must rebuild with complete history"

Mixing batch and streaming paradigms today often requires manual "replays" where historical data – usually stored in a separate batch system – is fed back through streaming pub/sub in order to pre-populate a continuous data pipeline.

Getting the ordering semantics and cut-over points right is a tedious, manual process. It's especially not fun when being undertaken due to a mistake, where there's an ongoing data outage until it's resolved.

Flow *unifies* batch and streaming data into a single concept – the *collection* – that makes it trivial to build (and rebuild) derivations & materialization that automatically backfill over all historical data, and thereafter transition to continuous updates.

3.1.3 Unbounded Look-back

"I want to compute a customer's lifetime value from a stream of interactions"
"I want to join two streams, where events may occur months apart"
"I want to consider my last 20 customer interactions, no matter when they occurred"

If you've spent time with continuous data systems that offer stream-to-stream joins (e.x. Flink, Spark, ksqlDB), you've no doubt read quite a bit on their various ways of expressing windowing semantics for stream-to-stream joins: event-time windows, hopping windows, rolling windows, session windows, etc.

Each has a slightly different expressions and semantics, but at their core, they're asking you to scope the portions of the joined streams which they have to consider at any given time.

This is fine if your problem is of some simpler forms – e.x. joining events that closely co-occur – but the semantics are arbitrary and pretty limiting in the general case.

Flow uses a simpler and more powerful conceptual model – the *register* – which allows for multi-way streaming joins and aggregations with arbitrary look-back.

With Flow, for example, it's trivial to convert a stream of customer purchases into a stream of customer lifetime value. Or to join a purchase with the user's first engagement with that product, over a month ago.

3.1.4 Data Reductions

"My PostgreSQL database can't keep up with my event bus."
"I'm hitting rate limits of my partner or vendor API."
"I have many frequent events of the same key, that cause performance hot-spots."

Flow is designed so that collection documents can be meaningfully combined at any time, grouped by the collection key, using *reduction annotations* of the document's schema.

Within the map/reduce paradigm, "combiners" have long been crucial to building high-performance pipelines, because they can substantially reduce the data volumes which must be written to disk, sent, sorted, and reduced at each pipeline stage.

Flow is unique in that it brings automatic combiners to the domain of continuous data processing. Documents are combined "early and often": when being ingested, derived, and materialized. Users are free to publish lots of small documents, without concern to the performance cost of doing so.

When materializing a rolled-up view, the target system only needs to be fast and large enough to process updates of the roll-up – and *not* the inputs that built it. As a materialization target – database, API, or stream – becomes busier or slower, Flow becomes more efficient at a combining more inputs into fewer writes to the target system.

3.1.5 Evolving Requirements

"I want to alter a calculation in my transform. . . "

"I want to join a new dataset into an existing transform. . . "

"I want to tweak how events are windowed. . . "

> ". . . without having to rebuild from scratch."

Requirements tend to change over time, and Flow's *derived collections* are designed to evolve with them. Add or remove transforms, update the register schema, or tweak lambda implementations, and the new behavior will be used on a go-forward basis – without having to completely rebuild the view, which can be expensive, involve downtime, and manual downstream migrations.

Of course if you *do* want to rebuild, that's easy too.

Some tools for continuous data processing (e.x. Spark, Flink, ksqlDB, Materialize) offer SQL as a primary means of defining transformations. While SQL is a wonderful query language, it's not a great fit for long-lived transforms with evolving requirements (e.x. left-join against a new dataset).

3.1.6 Versioned Documentation

"I don't know what data products are available within my organizations."

"How do I start using data produced by another team?"

"I need to understand how this metric is derived."

Flow catalogs are completely declarative, and are designed to be cooperatively developed by multiple self-service teams, coordinating through version control.

This means that everyone has a complete description of the data products managed by the organization in their git checkout, and can quickly start developing, locally running, and testing new data products that build off of their teammate's work.

In the near future, the Flow tooling will also generate human-friendly documentation for exploring a catalog, that can integrate directly into Github pages and be versioned with your repo. This keeps product, analyst, and business stakeholders "in the loop" with comprehensive documentation that updates as part of the engineering workflow.

3.1.7 Cheap Stream-to-Stream Joins

"I have a huge stream to join, and it's expensive to index!"

A canonical example is joining advertising "views", of which there are many, with a later "click", of which there are very few, joined over a common impression ID.

Typically this is done – either explicitly, or under the hood as part of an execution plan – by processing the view event first, and indexing it on impression ID within a local "state store", an attached mutable index offering very fast reads (eg, Flow uses RocksDB). Later, should a click event come, it's matched against the indexed view and emitted.

The trouble is that all of that indexed state needs to live *somewhere*, and as its quantity increases, you need more local stores and fast storage to back them. Flow is no exception here.

Flow *does* offer a unique "read delay" feature that lets you flip the problem on its head, by indexing each *click* and joining against views read with, say, a 10 minute delay. This can be **far** more efficient, as it re-orients to what local stores are great at: many very fast reads over fewer indexed states.

3.1.8 Tyranny of Partitioning

"Our topic has N partitions, but we've grown and now that's not enough"

Some systems (e.x. Kafka, Pulsar, Kenesis) require that you declare how many partitions a topic has. On write, each event is then hashed to a specific partition using its key. When reading, one consumer "task" is then created (usually, automatically) for each partition of the topic. Consumers leverage the insight that all events of a given key will be in the same partition.

This makes the choice of partitions a *crucial* knob. For one, it bounds the total read and write rate of the topic, though usually that isn't the primary concern. What is, is that the number of partitions determines the number of consumer tasks, and the number of associated task "state stores" – stores that hold inner transformation state like event windows and partial aggregates.

If those stores grow too large you *can't* simply increase the number of topic partitions, because that invalidates all stateful stream processors reading the topic (by breaking the assumption that all instances of a key are in the same partition).

Instead, standard practice is to perform an expensive manual re-partitioning, where the topic – in it's entirety – is copied into a new topic with updated partitioning, and downstream transformations are then rebuilt.

Flow **completely** avoids these issues. Collection partitions and derivation tasks are decoupled from one another, and can be scaled independently as needed and without downtime. In the future, scaling will be fully automated.

3.2 Comparisons

Flow is unique in the continuous processing space. It has similarities to, and is at the same time wholly unlike a number of other systems & tools.

3.2.1 Google Cloud Dataflow

Flow's most apt comparison is to Google Cloud Dataflow (aka Apache Beam), with which Flow has the most conceptual overlap.

Like Beam, Flow's primary primitive is a *collection*. One builds a processing graph by relating multiple collections together through procedural transformations (aka lambdas). As with Beam, Flow's runtime performs automatic data shuffles, and is designed to allow for fully automatic scaling. Also like Beam, collections have associated schemas.

Unlike Beam, Flow doesn't distinguish between "batch" and "streaming" contexts. Flow unifies these paradigms under a single *collection* concept.

Also, while Beam allows for optionally user-defined combine operators, Flow's runtime *always* applies combine operators built using the declared semantics of the document's schema.

Finally, Flow allows for stateful stream-stream joins without the "window-ization" semantics imposed by Beam. Notably, Flow's modeling of state – via it's per-key *register* concept – is substantially more powerful than Beams "per-key-and-window" model. For example, registers can trivially model the cumulative lifetime value of a customer.

3.2.2 PipelineDB / ksqlDB / Materialize

ksqlDB and Materialize are new SQL databases which focus on streaming updates of materialized views. PipelineDB was a PostgreSQL extension which, to our knowledge, pioneered in this space (and deserves more credit!).

Flow is not – nor does it want to be – a database. It aims to enable all of your *other* databases to serve continuously materialized views. Flow materializations use the storage provided by the target database to persist the view's aggregate output, and Flow focuses on mapping, combining, and reducing in updates of that aggregate as source collections change.

While Flow tightly integrates with the SQL table model (via *projections*), Flow can also power document stores like Elastic and CosmosDB, that deal in Flow's native JSON object model.

3.2.3 BigQuery / Snowflake / Presto

Flow is designed to integrate with Snowflake and BigQuery by adding Flow collections as external, Hive-partitioned tables within these systems.

First Flow is used to capture and "lake" data drawn from a pub/sub topic, for which Flow produces an organized file layout of compressed JSON in cloud storage. Files are even named to allow for Hive predicate push-down (ex "SELECT count(*) where utc_date = '2020-11-12' and region = 'US'), enabling substantially faster queries.

These locations can then be defined as external tables in Snowflake or BigQuery – and in the near future, we expect Flow will even produce this SQL DDL.

For data which is read infrequently, this can be cheaper than directly ingesting data into Snowflake or BigQuery – you consume no storage or compute credits until you actually query data.

For frequently read data, a variety of options are available for materializing or post-processing for native warehouse query performance.

3.2.4 dbt

dbt is a tool that enables data analysts and engineers to transform data in their warehouses more effectively. As they say, that's their elevator pitch.

In addition to – and perhaps more important than – it's transform capability, dbt brought an entirely new workflow for working with data. One that prioritizes version control, testing, local development, documentation, composition and re-use.

Fishtown Analytics should take it as sincere complement that Flow's declarative model and tooling has as many similarities as it does, as dbt provided significant inspiration.

However, there the similarities end. dbt is a tool for defining transformations, executed within your analytics warehouse. Flow is a tool for delivering data to that warehouse, as well as continuous *operational* transforms that are applied everywhere else.

They can also make lots of sense to use together: Flow is ideally suited for "productionizing" insights first learned in the analytics warehouse.

3.2.5 Spark / Flink

Spark and Flink are generalized execution engines for batch and stream data processing. They're well known – particularly Spark – and both are actually available "runners" within Apache Beam. Spark could be described as a batch engine with stream processing add-ons, where Flink as a stream processing engine with batch add-ons.

Flow best inter-operates with Spark through its "lake" capability. Spark can view Flow collections as Hive-partitioned tables, or just directly process bucket files.

For stream-stream joins, both Spark and Flink roughly share the execution model and constraints of Apache Beam. In particular, they impose complex "window-ization" requirements that also preclude their ability to offer continuous materializations of generalized stream-to-stream joins (eg, current lifetime value of a customer).

3.2.6 Kafka / Pulsar

Flow is built on Gazette, which is most similar to log-oriented pub/sub systems like Kafka or Pulsar. Flow also uses Gazette's consumer framework, which has similarities with Kafka Streams. Both manage scale-out execution contexts for consumer tasks, offer durable local task stores, and provide exactly-once semantics (though there are key differences).

Unlike those systems, Flow + Gazette use regular files with no special formatting (eg, compressed JSON) as the primary data representation, which powers its capabilities for integrating with other analytic tools. During replays historical data is read directly from cloud storage, which is strictly faster and more scalable, and reduces load on brokers.

Gazette's implementation of durable task stores also enables Flow's novel, zero-downtime task splitting technique.

3.3 Getting Started with VS Code

The easiest way to try out Flow yourself is to clone this repository and use VSCode Remote Containers. This guide will walk you through getting started.

1. Make sure you have VS Code Installed.
2. Follow the steps to setup VSCode Remote Containers
3. Clone the flow docs repo (`git clone https://github.com/estuary/docs flow-docs`)
4. Open the newly created `flow-docs` directory in VS Code. Select `Remote-Containers: Open Folder In Container...` from the command pallette and select the `flow-docs` directory. Alternatively, you can just open the `flow-docs` directory normally, and then click on "Reopen in Container" when the notification pops up.
5. You're ready to use Flow! Try opening a terminal within VS Code and running the tests for the example catalog:

```
# Build this documentation repository's Flow catalog.
$ flowctl build

# Run all catalog tests.
$ flowctl test
```

3.4 Concepts

Warning: This section is pretty dense. We need to break up and refactor content into smaller, more comprehensible chunks. You may want to take a look at examples first, and pick through topics here as more of a reference.

3.4.1 Collection

Flow's central concept is a **collection**: an append-only set of immutable JSON documents. Every collection has an associated schema that documents must validate against. Collections are either *captured*, meaning documents are directly added via Flow's ingestion APIs, or they're *derived* by applying transformations to other source collections, which may themselves be either captured or derived. A group of collections are held within a **catalog**.

Collections are optimized for low-latency processing.

> As documents are added to a collection, materializations & other derivations which use that collection are immediately notified (within milliseconds). This allows Flow to minimize end-to-end processing latency.

Collections are "Data Lakes".

> Collections organize, index, and durably store documents within a hierarchy of files implemented atop cloud storage. These files are Flow's native, source-of-truth representation for the contents of the collection, and can be managed and deleted using regular bucket life-cycle policies.
>
> Files hold collection documents with no special formatting (eg, as JSON lines), and can be directly processed using Spark and other preferred tools.

Collections can be of unbounded size.

> The Flow runtime persists data to cloud storage as soon as possible, and uses machine disks only for temporary data & indexes. Collection retention policies can be driven only by your organizational requirements –- not your available disk space.

A new derivation or materialization will efficiently back-fill over all collection documents – even where they span months or even years of data – by reading directly out of cloud storage.

Crucially, a high scale back-fill that sources from a collection doesn't compete with and cannot harm the collection's ability to accept new writes, as reads depend *only* on cloud storage for serving up historical data. This is a guarantee that's unique to Flow, through its Gazette-based architecture.

Collections may have logical partitions.

Logical partitions are defined in terms of a JSON-Pointer: i.e., the pointer `/region` would extract a partition value of "EU" from collection document `{"region": "EU", ...}`.

Documents are segregated by partition values, and are organized within cloud storage using a Hive-compatible layout. Partitioned collections are directly interpretable as external tables by tools that understand Hive partitioning and predicate push-down – like Snowflake, BigQuery, and Hive itself.

Each logical partition will have one or more *physical* partitions, backed by a corresponding Gazette journal. Physical partitions are largely transparent to users, but enable Flow to scale out processing as the data-rate increases, and may be added at any time.

Collections must have a declared key.

Keys are specified as one or more JSON-Pointer locations. When materializing a collection, its key carries over to the target system, and a key with more than one location becomes a composite primary key in SQL.

Collections are immutable, so adding a document doesn't *erase* a prior document having the same key – all prior documents are still part of the collection. Rather, it reflects an *update* of the key, which by default will flow through to replace the value indexed by a materialization. Far richer semantics are possible by using *reduction annotations* within the collection's schema.

3.4.2 Catalogs

Flow uses YAML or JSON source files, called *catalog sources*, to define the various entities which Flow understands (collections, schemas, tests, etc).

One or more sources are built into a *catalog database* by the `flowctl build` CLI tool. Catalog databases are SQLite files holding a compiled form of the catalog, and are what the Flow runtime actually executes against.

Catalog sources are divided into sections.

import section

A goal of catalogs is that they be composable and re-useable: catalog sources are able to import other sources, and it's recommended that authors structure their sources in ways that make sense for their projects, teams, and organization.

The `import` section is a list of partial or absolute URLs, which are always evaluated relative to the base directory of the current source. For example, these are possible imports within a catalog source:

```
# Suppose we're in file "/path/dir/flow.yaml"
import:
  - sub/directory/flow.yaml        # Resolves to "file:///path/dir/sub/directory/flow.
↪yaml".
  - ../sibling/directory/flow.yaml # Resolves to "file:///path/sibling/directory/flow.
↪yaml".
  - https://example/path/flow.yaml # Uses the absolute url.
```

The import rules are designed so that a catalog doesn't have to do anything special in order to be imported by another source, and flowctl can even directly build remote sources:

```
# Build this documentation repository's Flow catalog.
$ flowctl build -v --source https://raw.githubusercontent.com/estuary/docs/developer-
↪docs/flow.yaml
```

JSON schemas have a `$ref` keyword, by which local and external schema URLs may be referenced. Flow uses these same import rules for resolving JSON schemas, and it's recommended to directly reference the authoritative source of an external schema.

flowctl fetches and resolves all catalog and JSON Schema sources at build time, and the resulting catalog database is a self-contained snapshot of these resources *as they were* at the time the catalog was built.

collections section

The `collections` section is a list of collection definitions within a catalog source. A collection must be defined before it may be used as a source within another collection.

Derived collections may also reference collections defined in other catalog sources, but are required to first import them (directly or indirectly).

materializationTargets section

`materializationTargets` define short, accessible names for target systems – like SQL databases – that can be materialized into.

They encapsulate connection details and configuration of systems behind a memorable, authoritative name. See *Materializations* for more.

tests section

Flow catalogs can also define functional *contract tests* which verify the integrated end-to-end behaviors of one or more collections. You'll see examples of these tests throughout this documentation.

Tests are named and specified by the `tests` section, and are executed by the "flowctl test" command against a local instance of the Flow runtime. A single test may have one or more steps, where each is one of:

ingest:

> Ingest the given document fixtures into the named collection. Documents are required to validate against the collection's schema.
>
> All of the documents written by an ingest are guaranteed to be processed before those of a following ingest. However, documents *within* an ingest are written in collection key order.

verify:

> Verify runs after all prior "ingest" steps have been fully processed, and then compares provided fixtures to the contents of a named collection.
>
> Comparisons are done using fully combined documents, as if the collection under test had been materialized. Notably this means there will be only one document for a given collection key, and documents always appear in collection key order.
>
> Test fixture documents are *not* required to have all properties appearing in actual documents, as this can get pretty verbose. Only properties which are present in fixture documents are compared.

3.4.3 Schemas

Flow makes heavy use of JSON Schema to describe the expected structure and semantics of JSON documents. If you're new to JSON Schema, it's an expressive standard for defining JSON: it goes well beyond basic type information, and can model tagged unions, recursion, and other complex, real-world composite types. Schemas can also define rich data validations like minimum & maximum values, regular expressions, date/time/email & other formats, and more.

Together, these features let schemas represent structure *as well as* expectations and constraints which are evaluated and must hold true for every collection document, *before* it's added to the collection. They're a powerful tool for ensuring end-to-end data quality: for catching data errors and mistakes early, before they can cause damage.

Inference

A central design tenant of Flow is that users need only provide a modeling of their data *one time*, as a JSON schema. Having done that, Flow leverages static inference over the schema to provide translations into other schema flavors:

- Most *Projections* of a collection are automatically inferred from its schema, for example, and inference is used to map to appropriate SQL types and constraints.
- Inference powers many of the error checks Flow performs when building the catalog, such as ensuring that the collection key must exist and is of an appropriate type.
- Flow generates TypeScript definitions from schemas, to provide compile-time type checks of user lambda functions. These checks are immensely helpful for surfacing mismatched expectations around e.g. whether a field must exist, which otherwise usually blow up in production.

Reduction Annotations

JSON Schema introduces a concept of "Annotations", which allow schemas to attach metadata at locations within a validated JSON document. For example, `description` can be used to describe the meaning of a particular property:

```
properties:
    myField:
        description: "A description of myField"
```

Flow extends JSON Schema with *reductions* that define how one document is to be combined into another. Here's an integer that's summed:

```
type: integer
reduce: { strategy: sum }

# [ 1, 2, -1 ] => 2
```

What's especially powerful about annotations is that they respond to *conditionals* within the schema. A tagged union type might alter the `description` of a property depending on which variant of the union type was matched. This also applies to reduction annotations, which can use conditionals to *compose richer behaviors*.

Reduction annotations are a Flow super-power. They make it easy to define *combiners* over arbitrary JSON documents, and they allow Flow to employ those combiners early and often within the runtime – regularly collapsing a torrent of ingested documents into a trickle.

Note: Flow never delays processing in order to batch or combine more documents, as some systems do (commonly known as *micro-batches*, or time-based *polling*). Every document is processed as quickly as possible, from end to end.

Instead, Flow uses optimistic transaction pipelining to do as much useful work as possible, while it awaits the commit of a previous transaction. This natural back-pressure affords *plenty* of opportunity for data reductions, while minimizing latency.

3.4.4 Projections

Flow documents are arbitrary JSON, and may contain multiple levels of hierarchy and nesting. However, systems that Flow integrates with often model flattened tables with rows and columns, but no nesting. Others are somewhere in between.

Projections are the means by which Flow translates between the JSON documents of a collection, and a table representation. A projection defines a mapping between a structured document location (as a JSON-Pointer) and a corresponding column name (a "field") in, e.g., a CSV file or SQL table.

Many projections are inferred automatically from a collection's JSON Schema, using a field which is simply the JSON-Pointer with its leading slash removed. For example, a schema scalar with pointer `/myScalar` will generate a projection with field `myScalar`.

Users can supplement by providing additional collection projections, and a document location can have more than one projection field that references it. Projections are also how logical partitions of a collection are declared.

Some examples:

```
collections:
- name: example/sessions
    schema: session.schema.yaml
    key: [/user/id, /timestamp]
    projections:
        # A "user/id" projection field is automatically inferred.
        # Add an supplemental field that doesn't have a slash.
        user_id: /user/id
        # Partly decompose a nested array of requests into a handful of named␣
↪projections.
        "first request": /requests/0
        "second request": /requests/1
        "third request": /requests/2
        # Define logical partitions over country and device type.
        country:
            location_ptr: /country
            partition: true
        device:
            location_ptr: /agent/type
            partition: true
```

Logical Partitions

A logical partition of a collection is a projection which physically segregates the storage of documents by the partitioned value. Derived collections can in turn provide a *partition selector* which identifies a subset of partitions of the source collection that should be read:

```
collections:
- name: example/derived
  derivation:
    transform:
        fromSessions:
```

(continues on next page)

(continued from previous page)

```
            source:
                name: example/sessions
                partitions:
                    include:
                        country: [US, CA]
                    exclude:
                        device: [Desktop]
```

Partition selectors are very efficient, as they allow Flow to altogether avoid reading documents which aren't needed by the derived collection.

Partitions also enable *predicate push-down* when directly processing collection files using tools that understand Hive partitioning, like Snowflake, BigQuery, and Spark. Under the hood, the partitioned fields of a document are applied to name and identify the Gazette journal into which the document is written, which in turn prescribes how journal fragment files are arranged within cloud storage.

For example, a document of "example/sessions" like `{"country": "CA", "agent": {"type": "iPhone"}, ...}` would map to a Gazette journal prefix of `example/sessions/country=CA/device=iPhone/`, which in turn produces fragment files in cloud storage like: `s3://bucket/example/sessions/country=CA/device=iPhone/pivot=00/utc_date=2020-11-04/utc_hour=16/<name>.gz`.

Tools that understand Hive partitioning are able to take query predicates over "country", "device", or "utc_date/hour" and push them "down" into the selection of files which must be read to answer the query – often offering much faster query execution because far less data must be read.

Note: "pivot" identifies a *physical partition*, while "utc_date" and "utc_hour" reflect the time at which the journal fragment was created.

Within a logical partition there are one or more physical partitions, each a Gazette journal, into which documents are actually written. The logical partition prefix is extended with a "pivot" suffix to arrive at a concrete journal name.

Flow is designed so that physical partitions can be dynamically added at any time, to scale the write & read throughput capacity of a collection.

3.4.5 Ingestion

Flow offers a variety of ingestion APIs for adding documents into captured collections: gRPC, WebSocket streams of JSON or CSV, and POSTS of regular JSON over HTTP.

Ingestion within Flow is transactional. For example, the JSON POST API accepts multiple documents and collections to which they're written, and returns only after the ingestion has fully committed. If a fault occurs, or a document fails to validate against its collection schema, than the ingestion is rolled back in its entirety.

Many data sources are continuous in nature. Flow provides WebSocket APIs for use by browser agents, API servers, log sources, and other connected streaming systems. Documents sent on the WebSocket streamed are collected into transaction windows, and Flow regularly reports back on progress as transactions commit.

Note: Estuary has plans for additional means of ingestion, such as Kenesis and Kafka integrations, as well as direct Database changed data capture.

What's implemented today is a minimal baseline to enable early use cases.

3.4.6 Derivations

A derived collection is built from one or more *transforms*, where each transform reads a *source* collection and applies mapping *lambda* functions to its documents.

The transforms of a derivation are able to share state with each other through derivation *registers*, which are JSON documents that transformations can read and update. The applicable register is keyed by a data *shuffle* defined by each transform, extracted from its source documents.

Sources

A source is an "upstream" collection being consumed by a derived collection. Sources can be either captured or derived, however a derived collection cannot directly or indirectly source from itself.

In other words, collections must represent a directed acyclic graph (not having any loops), such that document processing will always halt. Of course, that doesn't stop you from integrating a service which adds a cycle, if that's your thing.

Selectors

Sources can specify a selector over partitions of the sourced collection, which will restrict the partitions which are read. Selectors are efficient, as they allow Flow to altogether avoid reading data that's not needed, rather than performing a read and then filtering it out.

Source Schemas

Collection schemas may evolve over time, and documents read from the source are re-validated against the current collection schema to ensure they remain valid. A schema error will halt the execution of the derivation, until the mismatch can be corrected.

Sources can optional provide an alternative *source schema* to use. This is helpful if the sourced collection reflects an external data source that the user doesn't have control over, and which may evolve its schema over time.

In this case, the captured collection can use a permissive schema that ensures documents are never lost, and the derived collection can then assert a stricter source schema. In the event that source documents violate that schema, the derivation will halt with an error. The user is then able to update their schema and transformations, and continue processing from where the derivation left off.

Delayed Reads

Event-driven workflows are usually a great fit for reacting to events as they occur, but they aren't terribly good at taking action when something *hasn't* happened:

- A user adds a product to their cart, but then doesn't complete a purchase.
- A temperature sensor stops producing its expected, periodic measurements.

Event driven solutions to this class of problem are challenging today. Minimally it requires integrating another system to manage many timers and tasks, bringing its own issues.

Engineering teams will instead often shrug and switch from an event streaming paradigm to a periodic batch workflow – gaining ease of implementation, but adding irremovable latency.

Flow offers another solution, which is to add an optional *read delay* to a transform. When specified, Flow will use the read delay to *gate* the processing of documents, with respect to the timestamp encoded within each document's UUID,

assigned by Flow at the time the document was ingested or derived. The document will remain gated until the current time is ahead of the document's timestamp, plus its read delay.

Similarly, if a derivation with a read delay is added later, the delay is also applied to determine the relative processing order of historical documents.

Note: Technically, Flow is gating the processing of a physical partition which is very efficient due to Flow's architecture. Documents that are closely ordered within a partition will also have almost identical timestamps.

For more detail on document UUIDs, see their Gazette documentation.

Read delays open up the possibility, for example, of joining a collection *with itself* to surface cases of shopping cart abandonment, or silenced sensors. A derivation might have a real-time transform that updates registers with a "last seen" timestamp on every sensor reading, and another transform with a five minute delay, that alerts if the "last seen" timestamp hasn't been updated *since* that sensor reading.

Lambdas

Lambdas are anonymous pure functions taking documents and returning zero, one, or more output documents. In map/reduce terms lambdas are "mappers", and the Flow runtime performs combine and reduce operations using the reduction annotations provided with schemas.

The Flow runtime manages the execution contexts of lambdas, and a derivation may be scaled out to many contexts running over available machines. Assignments and re-assignments of those contexts are automatic, and the runtime maintains "hot" standbys of each context for fast fail-over.

Under the hood, lambda execution contexts are modeled as *shards* within the Gazette consumers framework.

Lambda functions are typed by the JSON schemas which constitute their inputs and outputs. Output documents are validated against expected schemas, and an error will halt execution of the derivation. Where applicable, Flow will also map JSON schemas into corresponding types in the lambda implementation language, facilitating static type checks during catalog builds.

Note: Flow intends to support a variety of lambda languages in the future, such as Python, SQLIte, and jq.

TypeScript Lambdas

TypeScript is typed JavaScript that compiles to regular JavaScript during catalog builds, which Flow then executes on the NodeJS runtime. JSON Schemas are mapped to TypeScript types with high fidelity, enabling succinct and performant lambdas with rigorous type safety. Lambdas can also take advantage of the NPM package ecosystem.

Remote Lambdas

Remote endpoints are URLs which Flow invokes via JSON POST, sending batches of input documents and expecting to receive batches of output documents in return.

They're a means of integrating other languages and environments into a Flow derivation. Intended uses include APIs implemented in other languages, running as "serverless" functions (AWS lambdas, or Google Cloud Functions).

Registers

A register is an arbitrary JSON document which is shared between the various transformations of a derivation. It allows those transformations to communicate with one another, through updates of the register's value. Registers enable the full gamut of stateful processing workflows, including all varieties of joins and custom windowing semantics over prior events.

Like collections, the registers of a derivation always have an associated JSON schema. That schema may have reduction annotations, which are applied to fold updates of a register into a fully reduced value.

Each source document is mapped to a corresponding register using the transform's *shuffle*, and a derivation may have **lots** of distinct registers. Flow manages the mapping, retrieval, and persistence of register values.

Under the hood, registers are backed by replicated, embedded RocksDB instances which co-locate 1:1 with the lambda execution contexts that Flow manages. As contexts are assigned and re-assigned, their DBs travel with them.

If any single RocksDB instance becomes too large, Flow is able to perform an online "split" which subdivides its contents into two new databases (and paired execution contexts), re-assigned to other machines.

Shuffles

Transformations may provide a shuffle key as one or more JSON-Pointer locations, to be extracted from documents of the transform's sourced collection. If multiple pointers are given, they're treated as an ordered composite key. If no key is provided, the source's collection key is used instead.

During processing, every source document is mapped through its shuffle key to identify an associated register. Multiple transformations can coordinate with one another by selecting shuffle keys which reflect the same identifiers – even if those identifers are structured differently within their respective documents.

For example, suppose we're joining two collections on a user accounts: one transform might use a shuffle key of [/id] for "account" collection documents like `{"id": 123, ...}`, while another uses key [/account_id] for "action" documents like `{"account_id": 123, ...}`. In both cases the shuffled entity is an account ID, and we can implement a left-join of accounts and their actions by *updating* the register with the latest "account" document, and *publishing* "action" documents enriched by the latest "account" stored in the register.

At catalog build time, Flow checks the shuffle keys align on their composition and schema types.

Shuffle keys are named as they are because, in many cases, a physical "data shuffle" must occur where Flow redistributes source documents to the execution contexts that are responsible for their associated registers. This is a well known concept in the data processing world, and "shuffle" acknowledges and ties the role of a shuffle key to this concept. However, data shuffles are transparent to the user, and in many cases Flow can avoid them altogether.

Transforms

Transforms put sources, shuffles, registers, and lambdas all together: transforms of a derivation specify a source and (optional) shuffle key, and may have either or both of an *update* lambda and a *publish* lambda.

"Update" lambdas update the value of a derivation register. These lambdas are invoked with a source document as their only argument, and return zero, one, or more documents which are then reduced by Flow into the current register value.

"Publish" lambdas publish new documents into a derived collection. Publish lambdas run *after* an update lambda of the transformation, if there is one. They're invoked with a source document, its current register value, and its previous register value (if applicable). In turn, publish lambdas return zero, one, or more documents which are then incorporated into the derived collection.

Note that documents returned by publish lambdas are not *directly* added to collections. They're first reduced by Flow into a single document update for each encountered unique key of the derivation, within a given processing

transaction. In map/reduce terms, this is a "combine" operation, and it's a powerful data reduction technique. It means that "publish" lambdas can return many small documents with impunity, confident that the runtime will combine their effects into a single published document.

To accomplish a stateful processing task, generally an "update" lambda will update the register to reflect one or more encountered documents of interest (often called a *window*), using reduction annotations that fold semantically meaningful updates into the register's value for each document. This might mean storing a last-seen value, updating counters, sets, or other structures, or simply storing a bounded array of prior documents wholesale.

A "publish" lambda will then examine a source document, its current register, and prior register. It might filter documents, or compose portions of the source document & register. It can compare the prior & current registers to identify meaningful inflections, such as when a sum transitions between negative and positive. Whatever its semantics, it takes action by returning documents which are combined into the derived collection.

One might wonder why "update" lambdas aren't invoked with / allowed to examine the present register. The short answer is "performance". If update lambdas received a current register value then that implies that, for a given shuffle key, update lambdas must be invoked in strict sequential order. This could be *very* slow, especially if each invocation requires network round trips (e.g. with remote lambdas). Instead, Flow's formulation of "update" and "publish" allows the runtime to process large windows of source documents through "update" or "publish" concurrently, even where many may share a common shuffle key.

Note: While Flow is an event-driven system, the update/publish formulation has a direct translation into a traditional batch map/reduce paradigm, which Flow may offer in the future for even faster back-fills over massive datasets.

3.4.7 Materializations

Materializations are the means by which Flow "pushes" collections into your databases, key/value stores, publish/subscribe systems, WebHook APIs, and so on. They connect a collection to a target system, via a *materialization* of the collection that continuously updates with the collection itself.

Wherever applicable, materializations are always indexed by the collection key. For SQL specifically, this means components of the collection key are used as the composite primary key of the table.

Many systems are document-oriented in nature, and can accept unmodified collection documents. Others are table-oriented, and when materializing into these systems the user first selects a subset of available projections, where each projection becomes a column in the created target table.

Note: For the moment, Flow offers PostgreSQL and SQLite as available materialization targets. Again, what's implemented today is a minimal baseline to enable early use cases. We have lots planned here.

Transactions

Flow executes updates of materializations within transactions, and if the materialized system is also transactional, then these transactions are integrated for end-to-end "exactly once" semantics. At a high level, transactions:

- *Read* current documents from the store for relevant collection keys (where applicable, and not already cached by the runtime).
- *Reduce* one or more new collection documents into each of those read values.
- *Write* the updated document back out to the store (or stream, etc).

One thing to note is that Flow issues at most one store read, and just one store write per collection key, per transaction. That's irrespective to the number of collection documents that were ultimately reduced within the transaction. An

implication is that a "flood" of collection documents can frequently be reduced to a comparative "trickle" of database updates, allowing the database to be scaled independently of the collection's raw rate.

Flow also pipelines transactions: while one store transaction is in the process of committing, Flow is reading and reducing documents of the *next* transaction as it awaits the prior transaction's commit. As a target database becomes busier or slower, Flow becomes more efficient at combining many documents into fewer table updates.

Creation Workflow

Flow offers an interactive command-line workflow (`flowctl materialize`) by which materializations are created, using collections, projections, and *materialization targets* defined in the Flow catalog.

A materialization target simply defines a target system (eg, PostgreSQL), and the connection parameters which are necessary to reach it. It gives the target a short and memorable.

Once the workflow is completed, the Flow runtime creates and manages the long-lived execution context which will continuously keep the materialization up-to-date.

3.5 Reductions

Flow implements a number of reduction strategies for use within schemas, which tell Flow how two instances of a document can be meaningfully combined together.

3.5.1 Guarantees

In Flow, documents having the same collection key and written to the same logical partition have a "total order", meaning that one document is universally understood to have been written *before* the other.

This doesn't hold for documents of the same key written to *different* logical partitions. These documents can be considered "mostly" ordered: Flow uses timestamps to understand the relative ordering of these documents, and while this largely does the "Right Thing", small amounts of re-ordering are possible and even likely.

Flow guarantees exactly-once semantics within derived collections and materializations (so long as the target system supports transactions), and a document reduction will be applied exactly one time.

Flow does *not* guarantee that documents are reduced in sequential order, directly into a "base" document. For example, documents of a single Flow ingest transaction are combined together into one document per collection key at ingestion time – and that document may be again combined with still others, and so on until a final reduction into the base document occurs.

Taken together, these "total order" and "exactly-once" guarantees mean that reduction strategies must be *associative* [e.g. (2 + 3) + 4 = 2 + (3 + 4)], but need not be commutative [2 + 3 = 3 + 2] or idempotent [S u S = S]. They expand the palette of strategies which can be implemented, and allow for more efficient implementations as compared to, e.g., CRDTs.

In documentation, we'll refer to the "left-hand side" (LHS) as the preceding document, and the "right-hand side" (RHS) as the following one. Keep in mind that both the LHS and RHS may themselves represent a combination of still more ordered documents (e.g, reductions are applied *associatively*).

Note: Estuary has many future plans for reduction annotations:

- More strategies, including data sketches like HyperLogLogs, T-Digests, etc.
- Eviction policies and constraints, for bounding the sizes of objects and arrays with fine-grained removal ordering.

What's here today could be considered a minimal, useful proof-of-concept.

3.5.2 append

`append` works with arrays, and extends the left-hand array with items of the right-hand side.

```
collections:
  - name: example/reductions/append
    schema:
      type: object
      reduce: { strategy: merge }
      properties:
        key: { type: string }
        value:
          # Append only works with type "array".
          # Others will error at build time.
          type: array
          reduce: { strategy: append }
      required: [key]
    key: [/key]

tests:
  "Expect we can append arrays":
    - ingest:
        collection: example/reductions/append
        documents:
          - { key: "key", value: [1, 2] }
          - { key: "key", value: [3, null, "abc"] }
    - verify:
        collection: example/reductions/append
        documents:
          - { key: "key", value: [1, 2, 3, null, "abc"] }
```

The right-hand side *must* always be an array. The left-hand side *may* be null, in which case the reduction is treated as a no-op and its result remains null. This can be combined with schema conditionals to "toggle" whether reduction reduction should be done or not.

3.5.3 firstWriteWins / lastWriteWins

`firstWriteWins` always takes the first value seen at the annotated location. Likewise `lastWriteWins` always takes the last. Schemas which don't have an explicit reduce annotation default to lastWriteWins behavior.

```
collections:
  - name: example/reductions/fww-lww
    schema:
      type: object
      reduce: { strategy: merge }
      properties:
        key: { type: string }
        fww: { reduce: { strategy: firstWriteWins } }
        lww: { reduce: { strategy: lastWriteWins } }
      required: [key]
    key: [/key]
```

(continues on next page)

(continued from previous page)

```
tests:
  "Expect we can track first- and list-written values":
    - ingest:
        collection: example/reductions/fww-lww
        documents:
          - { key: "key", fww: "one", lww: "one" }
          - { key: "key", fww: "two", lww: "two" }
    - verify:
        collection: example/reductions/fww-lww
        documents:
          - { key: "key", fww: "one", lww: "two" }
```

3.5.4 merge

`merge` reduces the LHS and RHS by recursively reducing shared document locations. The LHS and RHS must either both be objects, or both be arrays.

If both sides are objects then it performs a deep merge of each property. If LHS and RHS are both arrays then items at each index of both sides are merged together, extending the shorter of the two sides by taking items of the longer:

```
collections:
  - name: example/reductions/merge
    schema:
      type: object
      reduce: { strategy: merge }
      properties:
        key: { type: string }
        value:
          # Merge only works with types "array" or "object".
          # Others will error at build time.
          type: [array, object]
          reduce: { strategy: merge }
          # Deeply merge sub-locations (items or properties) by summing them.
          items:
            type: number
            reduce: { strategy: sum }
          additionalProperties:
            type: number
            reduce: { strategy: sum }
      required: [key]
    key: [/key]

tests:
  "Expect we can merge arrays by index":
    - ingest:
        collection: example/reductions/merge
        documents:
          - { key: "key", value: [1, 1] }
          - { key: "key", value: [2, 2, 2] }
    - verify:
        collection: example/reductions/merge
        documents:
          - { key: "key", value: [3, 3, 2] }

  "Expect we can merge objects by property":
```

(continues on next page)

(continued from previous page)

```
    - ingest:
        collection: example/reductions/merge
        documents:
          - { key: "key", value: { "a": 1, "b": 1 } }
          - { key: "key", value: { "a": 1, "c": 1 } }
    - verify:
        collection: example/reductions/merge
        documents:
          - { key: "key", value: { "a": 2, "b": 1, "c": 1 } }
```

Merge may also take a `key`, which is one or more JSON pointers that are relative to the reduced location. If both sides are arrays and a merge key is present, then a deep sorted merge of the respective items is done, as ordered by the key. Arrays must be pre-sorted and de-duplicated by the key, and merge itself always maintains this invariant.

Note that a key of [“”] can be used for natural item ordering, e.g. when merging sorted arrays of scalars.

```
collections:
  - name: example/reductions/merge-key
    schema:
      type: object
      reduce: { strategy: merge }
      properties:
        key: { type: string }
        value:
          type: array
          reduce:
            strategy: merge
            key: [/k]
          items: { reduce: { strategy: firstWriteWins } }
      required: [key]
    key: [/key]

tests:
  "Expect we can merge sorted arrays":
    - ingest:
        collection: example/reductions/merge-key
        documents:
          - { key: "key", value: [{ k: "a", v: 1 }, { k: "b", v: 1 }] }
          - { key: "key", value: [{ k: "a", v: 2 }, { k: "c", v: 2 }] }
    - verify:
        collection: example/reductions/merge-key
        documents:
          - {
              key: "key",
              value: [{ k: "a", v: 1 }, { k: "b", v: 1 }, { k: "c", v: 2 }],
            }
```

As with `append`, the left-hand side of `merge` *may* be null, in which case the reduction is treated as a no-op and its result remains null.

3.5.5 minimize / maximize

`minimize` and `maximize` reduce by taking the smallest (or largest) seen value.

```yaml
collections:
  - name: example/reductions/min-max
    schema:
      type: object
      reduce: { strategy: merge }
      properties:
        key: { type: string }
        min: { reduce: { strategy: minimize } }
        max: { reduce: { strategy: maximize } }
      required: [key]
    key: [/key]

tests:
  "Expect we can min/max values":
    - ingest:
        collection: example/reductions/min-max
        documents:
          - { key: "key", min: 32, max: "abc" }
          - { key: "key", min: 42, max: "def" }
    - verify:
        collection: example/reductions/min-max
        documents:
          - { key: "key", min: 32, max: "def" }
```

Minimize and maximize can also take a `key`, which is one or more JSON pointers that are relative to the reduced location. Keys make it possible to min/max over complex types, by ordering over an extracted composite key.

In the event that a RHS document key equals the current LHS minimum (or maximum), then documents are deeply merged. This can be used to, for example, track not just the minimum value but also the number of times it's been seen:

```yaml
collections:
  - name: example/reductions/min-max-key
    schema:
      type: object
      reduce: { strategy: merge }
      properties:
        key: { type: string }
        min:
          $anchor: min-max-value
          type: array
          items:
            - type: string
            - type: number
              reduce: { strategy: sum }
          reduce:
            strategy: minimize
            key: [/0]
        max:
          $ref: "#min-max-value"
          reduce:
            strategy: maximize
            key: [/0]
      required: [key]
```

(continues on next page)

(continued from previous page)

```
    key: [/key]

tests:
  "Expect we can min/max values using a key extractor":
    - ingest:
        collection: example/reductions/min-max-key
        documents:
          - { key: "key", min: ["a", 1], max: ["a", 1] }
          - { key: "key", min: ["c", 2], max: ["c", 2] }
          - { key: "key", min: ["b", 3], max: ["b", 3] }
          - { key: "key", min: ["a", 4], max: ["a", 4] }
    - verify:
        collection: example/reductions/min-max-key
        documents:
          # Min of equal keys ["a", 1] and ["a", 4] => ["a", 5].
          - { key: "key", min: ["a", 5], max: ["c", 2] }
```

3.5.6 set

`set` interprets the document location as an update to a set.

The location must be an object having only "add", "intersect", and "remove" properties. Any single "add", "intersect", or "remove" is always allowed.

A document with "intersect" and "add" is allowed, and is interpreted as applying the intersection to the LHS set, followed by a union with the additions.

A document with "remove" and "add" is also allowed, and is interpreted as applying the removals to the base set, followed by a union with the additions.

"remove" and "intersect" within the same document is prohibited.

Set additions are deeply merged. This makes sets behave like associative maps, where the "value" of a set member can be updated by adding it to set again, with a reducible update.

Sets may be objects, in which case the object property serves as the set item key:

```
collections:
  - name: example/reductions/set
    schema:
      type: object
      reduce: { strategy: merge }
      properties:
        key: { type: string }
        value:
          # Sets are always represented as an object.
          type: object
          reduce: { strategy: set }
          # Schema for "add", "intersect", & "remove" properties
          # (each a map of keys and their associated sums):
          additionalProperties:
            type: object
            additionalProperties:
              type: number
              reduce: { strategy: sum }
            # Flow requires that all parents of locations with a reduce
            # annotation also have one themselves.
```

(continues on next page)

(continued from previous page)

```
            # This strategy therefore must (currently) be here, but is ignored.
            reduce: { strategy: lastWriteWins }

      required: [key]
    key: [/key]

tests:
  "Expect we can apply set operations to incrementally build associative maps":
    - ingest:
        collection: example/reductions/set
        documents:
          - { key: "key", value: { "add": { "a": 1, "b": 1, "c": 1 } } }
          - { key: "key", value: { "remove": { "b": 0 } } }
          - { key: "key", value: { "add": { "a": 1, "d": 1 } } }
    - verify:
        collection: example/reductions/set
        documents:
          - { key: "key", value: { "add": { "a": 2, "c": 1, "d": 1 } } }
    - ingest:
        collection: example/reductions/set
        documents:
          - { key: "key", value: { "intersect": { "a": 0, "d": 0 } } }
          - { key: "key", value: { "add": { "a": 1, "e": 1 } } }
    - verify:
        collection: example/reductions/set
        documents:
          - { key: "key", value: { "add": { "a": 3, "d": 1, "e": 1 } } }
```

Sets can also be sorted arrays, which are ordered using a provide `key` extractor. Keys are given as one or more JSON pointers, each relative to the item. As with "merge", arrays must be pre-sorted and de-duplicated by the key, and set reductions always maintain this invariant

Use a key extractor of [""] to apply the natural ordering of scalar values.

Whether array or object types are used, the type must always be consistent across the "add" / "intersect" / "remove" terms of both sides of the reduction.

```
collections:
  - name: example/reductions/set-array
    schema:
      type: object
      reduce: { strategy: merge }
      properties:
        key: { type: string }
        value:
          # Sets are always represented as an object.
          type: object
          reduce:
            strategy: set
            key: [/0]
          # Schema for "add", "intersect", & "remove" properties
          # (each a sorted array of [key, sum] 2-tuples):
          additionalProperties:
            type: array
            # Flow requires that all parents of locations with a reduce
            # annotation also have one themselves.
            # This strategy therefore must (currently) be here, but is ignored.
```

(continues on next page)

(continued from previous page)

```
            reduce: { strategy: lastWriteWins }
            # Schema for contained [key, sum] 2-tuples:
            items:
              type: array
              items:
                - type: string
                - type: number
                  reduce: { strategy: sum }
              reduce: { strategy: merge }

      required: [key]
    key: [/key]

tests:
  ? "Expect we can apply operations of sorted-array sets to incrementally build␣
↪associative maps"
  : - ingest:
        collection: example/reductions/set-array
        documents:
          - { key: "key", value: { "add": [["a", 1], ["b", 1], ["c", 1]] } }
          - { key: "key", value: { "remove": [["b", 0]] } }
          - { key: "key", value: { "add": [["a", 1], ["d", 1]] } }
    - verify:
        collection: example/reductions/set-array
        documents:
          - { key: "key", value: { "add": [["a", 2], ["c", 1], ["d", 1]] } }
    - ingest:
        collection: example/reductions/set-array
        documents:
          - { key: "key", value: { "intersect": [["a", 0], ["d", 0]] } }
          - { key: "key", value: { "add": [["a", 1], ["e", 1]] } }
    - verify:
        collection: example/reductions/set-array
        documents:
          - { key: "key", value: { "add": [["a", 3], ["d", 1], ["e", 1]] } }
```

3.5.7 sum

`sum` reduces two numbers or integers by adding their values.

```
collections:
  - name: example/reductions/sum
    schema:
      type: object
      reduce: { strategy: merge }
      properties:
        key: { type: string }
        value:
          # Sum only works with types "number" or "integer".
          # Others will error at build time.
          type: number
          reduce: { strategy: sum }
      required: [key]
    key: [/key]
```

(continues on next page)

(continued from previous page)

```
tests:
  "Expect we can sum two numbers":
    - ingest:
        collection: example/reductions/sum
        documents:
          - { key: "key", value: 5 }
          - { key: "key", value: -1.2 }
    - verify:
        collection: example/reductions/sum
        documents:
          - { key: "key", value: 3.8 }
```

3.5.8 Composing with Conditionals

Reduction strategies are JSON Schema annotations, and as such their applicability at a given document location can be controlled through the use of conditional keywords within the schema like `oneOf` or `if/then/else`. This means Flow's built-in strategies below can be combined with schema conditionals to construct a wider variety of custom reduction behaviors.

For example, here's a reset-able counter:

```
collections:
  - name: example/reductions/sum-reset
    schema:
      type: object
      properties:
        key: { type: string }
        value: { type: number }
      required: [key]
      # Use oneOf to express a tagged union over "action".
      oneOf:
        # When action = reset, reduce by taking this document.
        - properties: { action: { const: reset } }
          reduce: { strategy: lastWriteWins }
        # When action = sum, reduce by summing "value". Keep the LHS "action",
        # preserving a LHS "reset", so that resets are properly associative.
        - properties:
            action:
              const: sum
              reduce: { strategy: firstWriteWins }
            value: { reduce: { strategy: sum } }
          reduce: { strategy: merge }
    key: [/key]

tests:
  "Expect we can sum or reset numbers":
    - ingest:
        collection: example/reductions/sum-reset
        documents:
          - { key: "key", action: sum, value: 5 }
          - { key: "key", action: sum, value: -1.2 }
    - verify:
        collection: example/reductions/sum-reset
        documents:
          - { key: "key", value: 3.8 }
```

(continues on next page)

(continued from previous page)

```
  - ingest:
      collection: example/reductions/sum-reset
      documents:
        - { key: "key", action: reset, value: 0 }
        - { key: "key", action: sum, value: 1.3 }
  - verify:
      collection: example/reductions/sum-reset
      documents:
        - { key: "key", value: 1.3 }
```

3.6 Derivation Patterns

3.6.1 Where to accumulate?

When building a derived collection, the central question is where accumulation will happen: within derivation registers, or within a materialized database? Both approaches can produce equivalent results, but they do it in very different ways.

Accumulate in the Database

To accumulate in the database, you'll define a collection having a reducible schema with a derivation that uses only "publish" lambdas and no registers. The Flow runtime uses the provided annotations to reduce new documents into the collection, and ultimately keep the materialized table up to date.

A key insight is that the database is the *only* stateful system in this scenario, and that Flow is making use of reductions in two places:

1) To combine many published documents into partial "delta" states, which are the literal documents written to the collection.

2) To reduce "delta" states into the DB-stored value, reaching a final value.

For example, consider a collection that's summing a value:

Time	DB	Lambdas	Derived Document
T0	**0**	publish(2, 1, 2)	**5**
T1	**5**	publish(-2, 1)	**-1**
T2	**4**	publish(3, -2, 1)	**2**
T3	**6**	publish()	

This works especially well when materializing into a transactional database. Flow couples its processing transactions with corresponding DB transactions, ensuring end-to-end "exactly once" semantics.

When materializing into a non-transactional store, Flow is only able to provide weaker "at least once" semantics: it's possible that a document may be combined into a DB value more than once. Whether that's a concern depends a bit on the task at hand. Some reductions can be applied repeatedly without changing the result ("idempotent"), and some use cases are fine with *close enough*. For our counter above, it could give an incorrect result.

When materializing into a pub/sub topic, there *is* no store to hold final values, and Flow will publish delta states: each a partial update of the (unknown) final value.

Accumulate in Registers

Accumulating in registers involves a derivation that defines a reducible register schema, and uses "update" lambdas. Registers are arbitrary documents that can be shared and updated by the various transformations of a derivation. The Flow runtime allocates, manages, and scales durable storage for registers; you don't have to.

When using registers, the typical pattern is to use reduction annotations within updates of the register, and to then publish last-write-wins "snapshots" of the fully reduced value.

Returning to our summing example:

Time	Register	Lambdas	Derived Document
T0	**0**	update(2, 1, 2), publish(register)	**5**
T1	**5**	update(-2, 1), publish(register)	**4**
T2	**4**	update(3, -2, 1), publish(register)	**6**
T3	**6**	update()	

Register derivations are a great solution for materializations into non- transactional stores, because the documents they produce can be applied multiple times without breaking correctness.

They're also well suited for materializations that publish into pub/sub, as they can produce stand-alone updates of a fully-reduced value.

Example: Summing in DB vs Register

Here's a complete example of summing counts in the database, vs in registers:

```
import:
  - inputs.flow.yaml

collections:
  - name: patterns/sums-db
    schema: &schema
      type: object
      properties:
        Key: { type: string }
        Sum:
          type: integer
          reduce: { strategy: sum }
      required: [Key]
      reduce: { strategy: merge }
    key: [/Key]

    derivation:
      transform:
        fromInts:
          source: { name: patterns/ints }
          shuffle: [/Key]
          publish:
            nodeJS: |
              return [{Key: source.Key, Sum: source.Int}];

  - name: patterns/sums-register
    schema:
      # Re-use the schema defined above.
      <<: *schema
```

(continues on next page)

(continued from previous page)

```
      reduce: { strategy: lastWriteWins }
    key: [/Key]

    derivation:
      register:
        schema:
          type: integer
          reduce: { strategy: sum }
        initial: 0

      transform:
        fromInts:
          source: { name: patterns/ints }
          shuffle: [/Key]
          update:
            nodeJS: |
              return [source.Int];
          publish:
            nodeJS: |
              return [{Key: source.Key, Sum: register}];

tests:
  "Expect we can do sums during materialization or within registers":
    - ingest:
        collection: patterns/ints
        documents:
          - { Key: key, Int: -3 }
          - { Key: key, Int: 5 }
    - ingest:
        collection: patterns/ints
        documents: [{ Key: key, Int: 10 }]
    - verify:
        # "verify" steps fully reduce documents of the collection.
        # Under the hood, these are multiple delta updates.
        collection: patterns/sums-db
        documents: [{ Key: key, Sum: 12 }]
    - verify:
        # These are multiple snapshots, of which "verify" takes the last.
        collection: patterns/sums-register
        documents: [{ Key: key, Sum: 12 }]
```

3.6.2 Types of Joins

Note: Some schema of the examples below is omitted for brevity, but can be found here.

Outer Join accumulated in Database

Example of an outer join, which is reduced within a target database table. This join is "fully reactive": it updates with either source collection, and reflects the complete accumulation of their documents on both sides.

The literal documents written to the collection are combined delta states, reflecting changes on one or both sides of the join. These delta states are then fully reduced into the database table, and no other storage *but* the table is required by this example:

```
import:
  - inputs.flow.yaml

collections:
  - name: patterns/outer-join
    schema:
      $ref: schema.yaml#Join
      reduce: { strategy: merge }
      required: [Key]
    key: [/Key]

    derivation:
      transform:
        fromInts:
          source: { name: patterns/ints }
          shuffle: [/Key]
          publish:
            nodeJS: |
              return [{Key: source.Key, LHS: source.Int}];

        fromStrings:
          source: { name: patterns/strings }
          shuffle: [/Key]
          publish:
            nodeJS: |
              return [{Key: source.Key, RHS: [source.String]}];

tests:
  "Expect a fully reactive outer join":
    - ingest:
        collection: patterns/ints
        documents: [{ Key: key, Int: 5 }]
    - verify:
        collection: patterns/outer-join
        documents: [{ Key: key, LHS: 5 }]
    - ingest:
        collection: patterns/strings
        documents: [{ Key: key, String: hello }]
    - verify:
        collection: patterns/outer-join
        documents: [{ Key: key, LHS: 5, RHS: [hello] }]
    - ingest:
        collection: patterns/ints
        documents: [{ Key: key, Int: 7 }]
    - ingest:
        collection: patterns/strings
        documents: [{ Key: key, String: goodbye }]
    - verify:
        collection: patterns/outer-join
```

(continues on next page)

(continued from previous page)

```
        documents: [{ Key: key, LHS: 12, RHS: [hello, goodbye] }]
```

Inner Join accumulated in Registers

Example of an inner join, which is reduced within the derivation's registers. This join is also "fully reactive", updating with either source collection, and reflects the complete accumulation of their documents on both sides.

The literal documents written to the collection are fully reduced snapshots of the current join state.

This example *requires* registers due to the "inner" join requirement, which dictates that we can't publish anything until *both* sides of the join are matched:

```
import:
  - inputs.flow.yaml

collections:
  - name: patterns/inner-join
    schema:
      $ref: schema.yaml#Join
      reduce: { strategy: lastWriteWins }
      required: [Key]
    key: [/Key]

    derivation:
      register:
        schema:
          $ref: schema.yaml#Join
          reduce: { strategy: merge }
        initial: {}

      transform:
        fromInts:
          source: { name: patterns/ints }
          shuffle: [/Key]
          update:
            nodeJS: |
              return [{LHS: source.Int}];
          publish:
            nodeJS: &innerJoinLambda |
              // Inner join requires that both sides be matched.
              if (register.LHS && register.RHS) {
                return [{Key: source.Key, ...register}]
              }
              return [];

        fromStrings:
          source: { name: patterns/strings }
          shuffle: [/Key]
          update:
            nodeJS: |
              return [{RHS: [source.String]}];
          publish:
            nodeJS: *innerJoinLambda

tests:
  "Expect a fully reactive inner-join":
```

(continues on next page)

(continued from previous page)

```
    - ingest:
        collection: patterns/ints
        documents: [{ Key: key, Int: 5 }]
    - verify:
        # Both sides must be matched before a document is published.
        collection: patterns/inner-join
        documents: []
    - ingest:
        collection: patterns/strings
        documents: [{ Key: key, String: hello }]
    - verify:
        collection: patterns/inner-join
        documents: [{ Key: key, LHS: 5, RHS: [hello] }]
    - ingest:
        collection: patterns/ints
        documents: [{ Key: key, Int: 7 }]
    - ingest:
        collection: patterns/strings
        documents: [{ Key: key, String: goodbye }]
    - verify:
        # Reacts to accumulated updates of both sides.
        collection: patterns/inner-join
        documents: [{ Key: key, LHS: 12, RHS: [hello, goodbye] }]
```

One-sided Join accumulated in Registers

Example of a one-sided join, which publishes a current LHS joined with an accumulated RHS.

This example is *not* fully reactive. It publishes only on a LHS document, paired with a reduced snapshot of the RHS accumulator at that time.

```
import:
  - inputs.flow.yaml

collections:
  - name: patterns/inner-join
    schema:
      $ref: schema.yaml#Join
      reduce: { strategy: lastWriteWins }
      required: [Key]
    key: [/Key]

    derivation:
      register:
        schema:
          $ref: schema.yaml#Join
          reduce: { strategy: merge }
        initial: {}

      transform:
        fromInts:
          source: { name: patterns/ints }
          shuffle: [/Key]
          update:
            nodeJS: |
```

(continues on next page)

(continued from previous page)

```
            return [{LHS: source.Int}];
          publish:
            nodeJS: &innerJoinLambda |
              // Inner join requires that both sides be matched.
              if (register.LHS && register.RHS) {
                return [{Key: source.Key, ...register}]
              }
              return [];

        fromStrings:
          source: { name: patterns/strings }
          shuffle: [/Key]
          update:
            nodeJS: |
              return [{RHS: [source.String]}];
          publish:
            nodeJS: *innerJoinLambda

tests:
  "Expect a fully reactive inner-join":
    - ingest:
        collection: patterns/ints
        documents: [{ Key: key, Int: 5 }]
    - verify:
        # Both sides must be matched before a document is published.
        collection: patterns/inner-join
        documents: []
    - ingest:
        collection: patterns/strings
        documents: [{ Key: key, String: hello }]
    - verify:
        collection: patterns/inner-join
        documents: [{ Key: key, LHS: 5, RHS: [hello] }]
    - ingest:
        collection: patterns/ints
        documents: [{ Key: key, Int: 7 }]
    - ingest:
        collection: patterns/strings
        documents: [{ Key: key, String: goodbye }]
    - verify:
        # Reacts to accumulated updates of both sides.
        collection: patterns/inner-join
        documents: [{ Key: key, LHS: 12, RHS: [hello, goodbye] }]
```

3.6.3 Comparing Registers

Suppose we want to take action based on how a register is changing.

For example, suppose we want to detect "zero crossings" of a running sum, and then filter the source collection to those documents which caused the sum to cross from positive to negative (or vice versa).

We can use the `previous` register value to do so:

```
import:
  - inputs.flow.yaml
```

(continues on next page)

(continued from previous page)

```yaml
collections:
  - name: patterns/zero-crossing
    schema: schema.yaml#Int
    key: [/Key]

    derivation:
      register:
        schema:
          type: integer
          reduce: { strategy: sum }
        initial: 0

      transform:
        fromInts:
          source: { name: patterns/ints }
          shuffle: [/Key]
          update:
            nodeJS: return [source.Int];
          publish:
            nodeJS: |
              if (register > 0 != previous > 0) {
                return [source];
              }
              return [];

tests:
  "Expect we can filter to zero-crossing documents":
    - ingest:
        collection: patterns/ints
        documents:
          - { Key: key, Int: -5 } # => -5
          - { Key: key, Int: -10 } # => -10
    - ingest:
        collection: patterns/ints
        documents: [{ Key: key, Int: 13 }] # => -2
    - verify:
        collection: patterns/zero-crossing
        documents: []
    - ingest:
        collection: patterns/ints
        documents:
          - { Key: key, Int: 4 } # => 2 (zero crossing)
          - { Key: key, Int: 10 } # => 12
    - verify:
        collection: patterns/zero-crossing
        documents: [{ Key: key, Int: 4 }]
    - ingest:
        collection: patterns/ints
        documents:
          - { Key: key, Int: -13 } # => -1 (zero crossing)
          - { Key: key, Int: -5 } # => -6
    - verify:
        collection: patterns/zero-crossing
        documents: [{ Key: key, Int: -13 }]
```

3.7 Ingesting Data

Note: Flow's current ingestion capabilities should be considered a proof of concept. We plan to make it *much* simpler to express integrations into pub/sub systems, databases for "change data capture", and more.

There are a number of different options for how to ingest data into Flow:

- *HTTP PUT or POST requests*
- *Stream data over a Websocket* in either *CSV, TSV*, or *JSON* formats

3.7.1 flow-ingester

Flow ships with the `flow-ingester` binary, which provides network service endpoints for ingesting data into Flow Collections. There are currently two main APIs, a REST API accepts data in HTTP PUT and POST requests, and a websocket API that accepts data streamed over websocket connections. Only captured collections may ingest data in this way, not derivations.

When you run `flowctl develop`, the Flow Ingester will listen on `http://localhost:8081` by default.

Flow Ingester will always validate all documents against the collection's schema before they are written, so invalid data will never be added to the collection. Note that your collection schema may be as permissive as you like, and you can always apply more restrictive schemas in derivations if you want to.

Flow Ingester will also reduce all documents according to the collection key and reduction annotations on the schema, if present. This is done to optimize the storage space for collections that see frequent updates to the same key.

3.7.2 REST API

The REST API makes it easy to add data to one or more Flow collections transactionally. The endpoint is available at `/ingest` (e.g. `http://localhost:8081/ingest`). This endpoint will respond only to PUT and POST requests with a `Content-Type: application/json`. Any other method or content type will result in a 404 error response. The request body should be a JSON object where the keys are names of Flow Collections, and the values are arrays of documents for that collection. For example,

```
curl -H 'Content-Type: application/json' --data @- 'http://localhost:8081/ingest' <
↪<EOF
{
    "examples/citi-bike/rides": [
        {
            "bike_id": 7,
            "begin": {
                "timestamp": "2020-08-27 09:30:01.2",
                "station": {
                    "id": 3,
                    "name": "Start Name"
                }
            },
            "end": {
                "timestamp": "2020-08-27 10:00:02.3",
                "station": {
                    "id": 4,
                    "name": "End Name"
                }
```

(continues on next page)

(continued from previous page)

```
            }
        }
    ]
}
EOF
```

Running the above should result in output similar to the following:

```
{"Offsets":{"examples/citi-bike/rides/pivot=00":305},"Etcd":{"cluster_id
↪":14841639068965178418,"member_id":10276657743932975437,"revision":28,"raft_term":2}
↪}
```

In this example, we are ingesting a single document (beginning with `{ "bike_id": 7,...`) into the collection `examples/citi-bike/rides`. You may ingest any number of documents into any number of Flow Collections in a single request body, and they will be added in a single transaction. The response `Offsets` includes all of the Gazette journals where the data was written, along with the new "head" of the Journal. This is provided only to allow for applications to read data directly from Gazette or cloud storage if desired.

REST Transactional Semantics

Flow Ingester will ingest the data using a single Gazette transaction per REST request. For details on Gazette transactions, see the Gazette Transactional Append docs. The summary is basically that:

- If the HTTP response indicates success, then the documents are guaranteed to be written to the gazette brokers and replicated.
- If the HTTP response indicates an error, then the transaction will not be committed and no derivations will observe any of the documents.

3.7.3 Websocket API

The Websocket API provides an alternative for ingesting data, which is especially useful when you don't know how much data there is ahead of time, or when you don't need precise control over transaction boundaries. When ingesting over a websocket, the ingester will automatically divide the data into periodic transactions to provide optimal performance. The websocket API is also more flexible in the data formats that it can accept, so it's able to ingest CSV/TSV data directly, in addition to JSON. The websocket API is only able to ingest into a single collection per websocket connection, though.

The collection for websocket ingestions is given in the path of the URL, as in: `/ingest/<collection-name>`. For example, to ingest into the `examples/citi-bike/rides` collection, you'd use `ws://localhost:8081/ingest/examples/citi-bike/rides`.

For all websocket ingestions, the Sec-Websocket-Protocol header must be set when initiating the websocket connection. The value must be one of:

- `json/v1`
- `csv/v1`
- `tsv/v1`

If you're using the websocat CLI, then you can simply use the `--protocol` option.

Ingesting JSON over Websocket

When ingesting JSON, Flow Ingester accepts data over the websocket in "JSON-newline" (a.k.a. JSON Lines) format. Objects should not be enclosed within an array or have any separator characters between them except for whitespace. For example, to ingest a few rides into the `examples/citi-bike/rides` collection, lets start with the documents in JSON Lines format in the file `rides.jsonl`:

```
{"bike_id":7,"begin":{"timestamp":"2020-08-27 09:30:01","station":{"id":66,"name":
↪"North 4th St"}},"end":{"timestamp":"2020-08-27 10:00:02","station":{"id":23,"name":
↪"High St"}}}
{"bike_id":26,"begin":{"timestamp":"2020-08-27 09:32:01","station":{"id":91,"name":
↪"Grant Ave"}},"end":{"timestamp":"2020-08-27 09:50:12","station":{"id":23,"name":
↪"High St"}}}
```

Given the above content in a file named `rides.jsonl`, we could ingest it using `websocat` like so:

```
cat rides.jsonl | websocat --protocol json/v1 'ws://localhost:8081/ingest/examples/
↪citi-bike/rides'
```

This will add the data to the collection named `examples/citi-bike/rides`.

Ingesting CSV/TSV over Websocket

Flow Ingester is able to ingest a few different character-separated formats. Currently it supports Comma-separated (CSV) and Tab-separated (TSV) formats, using the *csv/v1* and *tsv/v1* protocols, respectively. Flow collections always store all data in JSON documents that validate against the collection's schema, so the tabular data in character-separated files must be converted to JSON before being written. Flow Ingester will convert these for you, based on the headers in the data and the projections for the Flow Collection. Each header in a character-separated ingestion must have the same name as a *projection* of the Collection. The projection will be used to map the field named by the header to the JSON pointer, which is used to construct the JSON document. For example, the `examples/citi-bike/rides` collection looks like this:

Given this, we could ingest a CSV file that looks like:

```
bikeid,starttime,"start station id","start station name",stoptime,"end station id",
↪"end station name"
7,"2020-08-27 09:30:01",66,"North 4th St","2020-08-27 10:00:02",23,"High St"
26,"2020-08-27 09:32:01",91,"Grant Ave","2020-08-27 09:50:12",23,"High St"
```

Assuming this was the content of `rides.csv`, you could ingest it using:

```
cat rides.csv | websocat --protocol csv/v1 'ws://localhost:8081/ingest/examples/citi-
↪bike/rides'
```

The actual JSON documents that would be written to the collection are:

```
{"bike_id":7,"begin":{"timestamp":"2020-08-27 09:30:01","station":{"id":66,"name":
↪"North 4th St"}},"end":{"timestamp":"2020-08-27 10:00:02","station":{"id":23,"name":
↪"High St"}}}
{"bike_id":26,"begin":{"timestamp":"2020-08-27 09:32:01","station":{"id":91,"name":
↪"Grant Ave"}},"end":{"timestamp":"2020-08-27 09:50:12","station":{"id":23,"name":
↪"High St"}}}
```

For example, the projection `bikeid: /bike_id` means that, for each row in the CSV, the value of the "bikeid" column was used to populate the `bike_id` property of the final document. Flow uses the collection's json schema to determine the required type of each property. Additionally, each document that's constructed is validated against the collection's schema prior to it being written.

Null, Empty, and Missing Values

In JSON documents, there's a difference between an explicit `null` value and one that's undefined. When Flow Ingester parses a character-separated row, it also differntiates between `null`, empty string, and undefined values. Empty values being ingested are always interpreted as explicit `null` values as long as the schema location allows for `null` values (e.g. `type: ["integer", "null"]`). If the schema does not allow `null` as an acceptable type, but it does allow `string`, then the value will be interpreted as an empty string. A row may also have fewer values than exist in the header row. If it does, than any unspecified column values will be undefined in the final document. In the following example, let's assume that the schema allows for the types in each column's name.

```
id,string,stringOrNull,integerOrNull
1,"","",""
2,,,
3,
4
```

Assuming simple direct projections, this would result in the following JSON documents being ingested:

```
{"id": 1, "string": "", "stringOrNull": null, "integerOrNull": null}
{"id": 2, "string": "", "stringOrNull": null, "integerOrNull": null}
{"id": 3, "string": ""}
{"id": 4}
```

Note how in rows `1` and `2`, empty `stringOrNull` values are mapped to `null`, regardless of the presence of quotes. In row `3`, the trailing comma indicates that the row has two values, and that the second value is empty (`""`), but the remainder are undefined. In row `4`, all values besides `id` are undefined.

Websocket Responses

Regardless of which format you ingest, all websocket ingestions will return responses similar to the following:

```
{"Offsets":{"examples/citi-bike/rides/pivot=00":545},"Etcd":{"cluster_id
↪":14841639068965178418,"member_id":10276657743932975437,"revision":28,"raft_term":2}
↪,"Processed":2}
```

The response will show the offsets of the transaction boundaries in the Gazette journals. If you ingest larger amounts of data, you will receive many such responses. In addition to the journal offsets, each response also includes the `Processed` property, which indicates the number of websocket frames that have been successfully ingested. This can be used to allow clients to resume where they left off in the case that a websocket ingestion fails partway through. For example, if you sent one json object per websocket frame, then you would know from the `Processed` field how many documents had been successfully ingested prior to the failure (`Processed` times the number of documents per frame).

3.8 Citi Bike System Data

We'll be using Flow to capture and process Citi Bike system data. The dataset is available in the S3 "tripdata" bucket as compressed CSV files of each ride taken within the system, by month.

3.8.1 Modeling Rides

Every Flow collection has an associated JSON Schema which describes its documents. Let's begin with an example of a ride document that we want to schematize:


```
{
   "bike_id": 26396,
   "duration_seconds": 1193,
   "user_type": "Customer",
   "gender": 0,
   "birth_year": 1969,
   "begin": {
      "station": {
         "geo": {
         "latitude": 40.711863,
         "longitude": -73.944024
         },
         "id": 3081,
         "name": "Graham Ave & Grand St"
      },
      "timestamp": "2020-09-01 00:00:12.2020"
   },
   "end": {
      "station": {
         "geo": {
         "latitude": 40.68402,
         "longitude": -73.94977
         },
         "id": 3048,
         "name": "Putnam Ave & Nostrand Ave"
      },
      "timestamp": "2020-09-01 00:20:05.5470"
   }
}
```


3.8.2 Ride Schema

We've already gone to the trouble of creating a JSON Schema which models Citi Bike rides, which you can find here. We can remotely link to it from our catalog so we don't have to repeat ourselves. A few things about it to point out:

It defines the *shape* that documents can take. A "ride" document must have a *bike_id*, *begin*, and *end*. A "location" must have a *latitude*, *longitude*, and so on. The `$ref` keyword makes it easy to re-use common structures.

***Validations* constrain the types and values that documents can take.** A "longitude" must be a number and fall within the expected range, and "gender" must be a value within the expected enumeration. Some properties are `required`, while others are optional. Flow enforces that all documents of a collection must validate against its schema before they can be added.

Flow is also able to *translate* many schema constraints (e.g. "/begin/station/id must exist and be an integer") into other kinds of schema – like TypeScript types and SQL constraints – which promotes end-to-end type safety and a better development experience.

***Annotations* attach information to locations within the document.** `title` and `description` keywords give color to locations of the document. They're machine-accessible documentation – which makes it possible to re-use these annotations in transformed versions of the schema.

3.8.3 Capturing Rides

To work with ride events, first we need to define a collection into which we'll ingest them. Simple enough, but a wrinkle is that the source dataset is CSV files, using header names which don't match our schema:

```
$ wget https://s3.amazonaws.com/tripdata/202009-citibike-tripdata.csv.zip
$ unzip -p 202009-citibike-tripdata.csv.zip  | head -5
"tripduration","starttime","stoptime","start station id","start station name","start␣
↪station latitude","start station longitude","end station id","end station name",
↪"end station latitude","end station longitude","bikeid","usertype","birth year",
↪"gender"
4225,"2020-09-01 00:00:01.0430","2020-09-01 01:10:26.6350",3508,"St Nicholas Ave &␣
↪Manhattan Ave",40.809725,-73.953149,116,"W 17 St & 8 Ave",40.74177603,-74.00149746,
↪44317,"Customer",1979,1
1868,"2020-09-01 00:00:04.8320","2020-09-01 00:31:13.7650",3621,"27 Ave & 9 St",40.
↪7739825,-73.9309134,3094,"Graham Ave & Withers St",40.7169811,-73.94485918,37793,
↪"Customer",1991,1
1097,"2020-09-01 00:00:06.8990","2020-09-01 00:18:24.2260",3492,"E 118 St & Park Ave",
↪40.8005385,-73.9419949,3959,"Edgecombe Ave & W 145 St",40.823498,-73.94386,41438,
↪"Subscriber",1984,1
1473,"2020-09-01 00:00:07.7440","2020-09-01 00:24:41.1800",3946,"St Nicholas Ave & W␣
↪137 St",40.818477,-73.947568,4002,"W 144 St & Adam Clayton Powell Blvd",40.820877,-
↪73.939249,35860,"Customer",1990,2
```

Projections let us account for this, by defining a mapping between document locations (as JSON Pointers) and corresponding fields in a flattened, table-based representation such as a CSV file or SQL table. They're used whenever Flow is capturing from or materializing into table-systems.

Putting it all together, let's define a captured "rides" collection:

```
collections:
  - name: examples/citi-bike/rides
    key: [/bike_id, /begin/timestamp]
    schema: https://raw.githubusercontent.com/estuary/docs/developer-docs/examples/
↪citi-bike/ride.schema.yaml
    # Define projections for each CSV header name used in the source dataset.
    projections:
      bikeid: /bike_id
      birth year: /birth_year
      end station id: /end/station/id
      end station latitude: /end/station/geo/latitude
      end station longitude: /end/station/geo/longitude
      end station name: /end/station/name
      gender: /gender
      start station id: /begin/station/id
      start station latitude: /begin/station/geo/latitude
      start station longitude: /begin/station/geo/longitude
      start station name: /begin/station/name
      starttime: /begin/timestamp
      stoptime: /end/timestamp
      tripduration: /duration_seconds
      usertype: /user_type
```

As this is a tutorial, we'll use a Flow ingestion API to capture directly from CSV. In a real-world setting, you could instead *bind* the collection to a pub/sub topic, S3 bucket and path, or a database table (via *change data capture*):

```
# Start a local development instance, and leave it running:
$ flowctl develop
```

(continues on next page)

(continued from previous page)

```
# In another terminal:
$ examples/citi-bike/load-rides.sh
```

3.8.4 Last-Seen Station of a Bike

We'll declare and test a collection that derives, for each bike, the station it last arrived at:

```
import:
  - rides.flow.yaml

collections:
  - name: examples/citi-bike/last-seen
    key: [/bike_id]
    schema:
      type: object
      properties:
        bike_id: { $ref: ride.schema.yaml#/properties/bike_id }
        last: { $ref: ride.schema.yaml#/$defs/terminus }
      required: [bike_id, last]

    derivation:
      transform:
        locationFromRide:
          source: { name: examples/citi-bike/rides }
          publish:
            nodeJS: |
              return [{ bike_id: source.bike_id, last: source.end }];
```

We can materialize the collection into a database table:

```
$ flowctl materialize --collection examples/citi-bike/last-seen --table-name last_
↪seen --target testDB
```

If you're in VSCode, you can query the attached database using the "SQLTools" icon on the left bar. Or, use `psql`:

```
select bike_id, "last/station/name", "last/timestamp" from last_seen limit 10;
```

Materialization tables always use the collection's key, and are update continuously to reflect ongoing changes of the collection.

3.8.5 Bike Relocations

Citi Bike will sometimes redistribute bikes between stations, when a station gets too full or empty. These relocations show up as "holes" in the ride data, where a bike mysteriously ends a ride at one station and starts its next ride at a different station.

Suppose we want a collection which is enriched with explicit "relocation" events. To derive it, we must determine if the start of a current ride is different than the end of a previous ride, for each bike. But, we don't have the prior ending station station available in the source document.

We can use *registers* to preserve a previous ending station, and compare it with a current starting station for each bike:

```yaml
import:
  - rides.flow.yaml

collections:
  - name: examples/citi-bike/rides-and-relocations
    key: [/bike_id, /begin/timestamp]
    schema:
      # Relocations are rides marked by a "relocation: true" property.
      $ref: ride.schema.yaml
      properties:
        relocation: { const: true }

    derivation:
      # Use a register to persist the last-arrived station for each bike.
      register:
        schema: ride.schema.yaml#/$defs/terminus
        initial:
          # Value to use if this is the first time seeing this bike.
          station: { id: 0, name: "" }
          timestamp: "0000-00-00 00:00:00.0"

      transform:
        fromRides:
          source: { name: examples/citi-bike/rides }
          shuffle: [/bike_id]
          update:
            nodeJS: return [source.end];
          publish:
            # Compare |previous| register value from before the update lambda was␣
↪applied,
            # with the source document to determine if the bike mysteriously moved.
            nodeJS: |
              if (previous.station.id != 0 && previous.station.id != source.begin.
↪station.id) {
                return [
                  { bike_id: source.bike_id, begin: previous, end: source.begin,␣
↪relocation: true },
                  source,
                ];
              }
              return [source];
```

Use `gazctl` to observe relocation events, as they're derived:

```
$ gazctl journals read --block -l estuary.dev/collection=examples/citi-bike/rides-and-
↪relocations \
  | jq -c '. | select(.relocation)'
```

3.8.6 Catalog Tests

We can uses *catalog tests* to verify the end-to-end, integrated behavior of collections. In fact, all of the collections in this tutorial have associated tests, but they're omitted here for brevity. You can find examples of comprehensive tests on GitHub.

Here's an example of a test for the rides & relocations derivation we just built:

```yaml
import:
  - rides-and-relocations.flow.yaml

tests:
  "Expect a sequence of connected rides don't produce a relocation event":
    - ingest:
        collection: examples/citi-bike/rides
        documents:
          # Bike goes from station 1 => 2 => 3 => 4.
          - &ride1
            bike_id: &bike 17558
            begin: &station1
              station: { id: 3276, name: "Marin Light Rail" }
              timestamp: "2020-09-01 09:21:12.3090"
            end: &station2
              station: { id: 3639, name: "Harborside" }
              timestamp: "2020-09-01 13:48:12.3830"
          - &ride2
            bike_id: *bike
            begin: *station2
            end: &station3
              station: { id: 3202, name: "Newport PATH" }
              timestamp: "2020-09-01 14:33:35.1020"
          - &ride3
            bike_id: *bike
            begin: *station3
            end: &station4
              station: { id: 3267, name: "Morris Canal" }
              timestamp: "2020-09-01 16:49:30.1610"
    - verify:
        collection: examples/citi-bike/rides-and-relocations
        documents: [*ride1, *ride2, *ride3]

  "Expect a disconnected ride sequence produces an interleaved relocation":
    - ingest:
        collection: examples/citi-bike/rides
        documents:
          - &ride1 { bike_id: *bike, begin: *station1, end: *station2 }
          - &ride2 { bike_id: *bike, begin: *station3, end: *station4 }
    - verify:
        collection: examples/citi-bike/rides-and-relocations
        documents:
          - *ride1
          - {
              bike_id: *bike,
              begin: *station2,
              end: *station3,
              relocation: true,
            }
          - *ride2
```

3.8.7 Station Status

Suppose we're building a station status API. We're bringing together some basic statistics about each station, like the number of bikes which have arrived, departed, and been relocated in or out. We also need to know which bikes are currently at each station.

To accomplish this, we'll build a collection keyed on station IDs into which we'll derive documents that update our station status. However, we need to tell Flow how to *reduce* these updates into a full view of a station's status, by adding `reduce` annotations into our schema. Here's the complete schema for our station status collection:

```
# Compose in the "station" definition from ride.schema.yaml,
# which defines "id", "name", and "geo".
$ref: ride.schema.yaml#/$defs/station

properties:
  arrival:
    description: "Statistics on Bike arrivals to the station"
    properties:
      ride:
        title: "Bikes ridden to the station"
        type: integer
        reduce: { strategy: sum }
      move:
        title: "Bikes moved to the station"
        type: integer
        reduce: { strategy: sum }

    type: object
    reduce: { strategy: merge }

  departure:
    description: "Statistics on Bike departures from the station"
    properties:
      ride:
        title: "Bikes ridden from the station"
        type: integer
        reduce: { strategy: sum }
      move:
        title: "Bikes moved from the station"
        type: integer
        reduce: { strategy: sum }

    type: object
    reduce: { strategy: merge }

  stable:
    description: "Set of Bike IDs which are currently at this station"
    type: object

    reduce:
      strategy: set
      # Use bike IDs as their own keys.
      key: [""]

    # Sets are composed of 'add', 'intersect', and 'remove' components.
    # Here, we're representing the set as an array of integer bike IDs.
    additionalProperties:
      type: array
```

(continues on next page)

(continued from previous page)

```
      items: { type: integer }
      reduce: { strategy: merge }

reduce: { strategy: merge }
```

Flow uses reduce annotations to build general "combiners" (in the map/reduce sense) over documents of a given schema. Those combiners are employed automatically by Flow.

Now we define our derivation. Since Flow is handling reductions for us, our remaining responsibility is to implement the "mapper" function which will transform source events into status status updates:

```
import:
  - rides-and-relocations.flow.yaml

collections:
  - name: examples/citi-bike/stations
    key: [/id]
    schema: station.schema.yaml
    projections:
      stable: /stable/add

    derivation:
      transform:
        ridesAndMoves:
          source:
            name: examples/citi-bike/rides-and-relocations
          publish:
            nodeJS: |
              if (source.relocation) {
                return [
                  {
                    departure: { move: 1 },
                    stable: { remove: [source.bike_id] },
                    ...source.begin.station,
                  },
                  {
                    arrival: { move: 1 },
                    stable: { add: [source.bike_id] },
                    ...source.end.station,
                  },
                ];
              } else {
                return [
                  {
                    departure: { ride: 1 },
                    stable: { remove: [source.bike_id] },
                    ...source.begin.station,
                  },
                  {
                    arrival: { ride: 1 },
                    stable: { add: [source.bike_id] },
                    ...source.end.station,
                  },
                ];
              }
```

Now we can materialize the collection into a PostgreSQL table, and have a live view into stations:

```
$ flowctl materialize --collection examples/citi-bike/stations --table-name stations -
↪-target testDB
```

```
-- Current bikes at each station.
select id, name, stable from stations order by name asc limit 10;
-- Station arrivals and departures.
select id, name, "arrival/ride", "departure/ride", "arrival/move", "departure/move"
   from stations order by name asc limit 10;
```

3.8.8 Idle Bikes

We're next tasked with identifying when bikes have sat idle at a station for an extended period of time. This is a potential signal that something is wrong with the bike, and customers are avoiding it.

Event-driven systems usually aren't terribly good at detecting when things *haven't* happened. At this point, an engineer will often reach for a task scheduler like Airflow, and set up a job that takes periodic snapshots of bike locations, and compares them to find ones which haven't changed.

Flow offers a simpler approach, which is to join the rides collection with itself, using a *read delay*:

```
import:
  - rides.flow.yaml

collections:
  # Derive idle bikes via two transforms of rides:
  # * One reads in real-time, and stores the ride timestamp in a register.
  # * Two reads with a delay, and checks whether the last
  #   ride timestamp hasn't updated since this (delayed) ride.
  - name: examples/citi-bike/idle-bikes
    schema:
      type: object
      properties:
        bike_id: { type: integer }
        station: { $ref: ride.schema.yaml#/$defs/terminus }
      required: [bike_id, station]

    key: [/bike_id, /station/timestamp]

    derivation:
      register:
        # Store the most-recent ride timestamp for each bike_id,
        # and default to null if the bike hasn't ridden before.
        schema: { type: [string, "null"] }
        initial: null

      transform:
        liveRides:
          source:
            name: examples/citi-bike/rides
          shuffle: [/bike_id]
          update:
            nodeJS: return [source.end.timestamp];

        delayedRides:
          source:
            name: examples/citi-bike/rides
```

(continues on next page)

(continued from previous page)

```
          shuffle: [/bike_id]
          # Use a 2-day read delay, relative to the document's ingestion.
          # To see read delays in action within a short-lived
          # testing contexts, try using a smaller value (e.g., 2m).
          readDelay: "48h"
          publish:
            nodeJS: |
              // Publish if the bike hasn't moved in 2 days.
              if (register === source.end.timestamp) {
                return [{ bike_id: source.bike_id, station: source.end }];
              }
              return [];
```

After the read delay has elapsed, we'll start to see events in the "idle-bikes" collection:

```
$ gazctl journals read --block -l estuary.dev/collection=examples/citi-bike/idle-bikes
```

3.9 Wikipedia Edits

We'll use to model Wikipedia page edits data, inspired by the Druid documentation.

3.9.1 Captured Edits

Our source dataset has documents like:

```
{
   "time": "2015-09-12T22:02:05.807Z",
   "channel": "#en.wikipedia",
   "cityName": "New York",
   "comment": "/* Life and career */",
   "countryIsoCode": "US",
   "countryName": "United States",
   "isAnonymous": true,
   "isMinor": false,
   "isNew": false,
   "isRobot": false,
   "isUnpatrolled": false,
   "metroCode": 501,
   "namespace": "Main",
   "page": "Louis Gruenberg",
   "regionIsoCode": "NY",
   "regionName": "New York",
   "user": "68.175.31.28",
   "delta": 178,
   "added": 178,
   "deleted": 0
}
```

Here's a captured collection for these page edits:

```
collections:
  - name: examples/wiki/edits
    key: [/time, /page]
```

(continues on next page)

(continued from previous page)

```
    # Inline schema which partially describes the edit dataset:
    schema:
      type: object
      required: [time, page, channel]
      properties:
        time: { type: string }
        page: { type: string }
        channel: { type: string }
        countryIsoCode: { type: [string, "null"] }
        added: { type: integer }
        deleted: { type: integer }
    # Declare channel (e.x. "#en.wikipedia") as a logical partition:
    projections:
      channel:
        location: /channel
        partition: true
```

We'll use Flow's ingestion API to capture the collection. In a production setting, you could imagine the collection instead being bound to a pub/sub topic or S3 bucket & path:

```
# Start a local development instance, and leave it running:
$ flowctl develop

# In another terminal:
$ examples/wiki/load-pages.sh
```

3.9.2 Page Roll-up

We can roll-up on page to understand edit statistics for each one, including a by-country break down (where the country is known):

```
import:
  - edits.flow.yaml

collections:
  - name: examples/wiki/pages
    key: [/page]
    # Inline schema which rolls up page edit statistics,
    # including a per-country breakdown:
    schema:
      $defs:
        counter:
          type: integer
          reduce: { strategy: sum }

        stats:
          type: object
          reduce: { strategy: merge }
          properties:
            cnt: { $ref: "#/$defs/counter" }
            add: { $ref: "#/$defs/counter" }
            del: { $ref: "#/$defs/counter" }

      type: object
      $ref: "#/$defs/stats"
```

(continues on next page)

(continued from previous page)

```
      properties:
        page: { type: string }
        byCountry:
          type: object
          reduce: { strategy: merge }
          additionalProperties: { $ref: "#/$defs/stats" }
      required: [page]

    # /byCountry is an object (which isn't projected by default),
    # and we'd like to materialize it to a column.
    projections:
      byCountry: /byCountry

    derivation:
      transform:
        rollUpEdits:
          source:
            name: examples/wiki/edits
          publish:
            nodeJS: |
              let stats = {cnt: 1, add: source.added, del: source.deleted};

              if (source.countryIsoCode) {
                return [{
                    page: source.page,
                    byCountry: {[source.countryIsoCode]: stats},
                    ...stats,
                }];
              }
              // Unknown country.
              return [{page: source.page, ...stats}];
```

Materialize pages to a test database:

```
$ flowctl materialize --collection examples/wiki/pages --table-name pages --target␣
↪testDB
```

Query for popular pages. As page edits are captured into the source collection, the page roll-up derivation and it's materialization will update. You can repeat the ingest if it completes too quickly:

```
SELECT page, cnt, add, del, "byCountry" FROM pages WHERE cnt > 10;
```

3.10 Network Traces

We'll tackle a few network monitoring tasks, using data drawn from this Kaggle Dataset of collected network traces.

> **Caution:** This example is a work in progress, and currently more of a sketch. Contributions welcome!

3.10.1 Trace Dataset

Our source dataset contains network traces of source & destination endpoints, packet flows, and bytes – much like what you'd obtain from tcpdump. It has many repetitions, including records having the same pair of endpoints and timestamp.

```
Source.IP,Source.Port,Destination.IP,Destination.Port,Protocol,Timestamp,Flow.
↪Duration,Total.Fwd.Packets,Total.Backward.Packets,Total.Length.of.Fwd.Packets,Total.
↪Length.of.Bwd.Packets
172.19.1.46,52422,10.200.7.7,3128,6,26/04/201711:11:17,45523,22,55,132,110414
10.200.7.7,3128,172.19.1.46,52422,6,26/04/201711:11:17,1,2,0,12,0
50.31.185.39,80,10.200.7.217,38848,6,26/04/201711:11:17,1,3,0,674,0
50.31.185.39,80,10.200.7.217,38848,6,26/04/201711:11:17,217,1,3,0,0
192.168.72.43,55961,10.200.7.7,3128,6,26/04/201711:11:17,78068,5,0,1076,0
10.200.7.6,3128,172.19.1.56,50004,6,26/04/201711:11:17,105069,136,0,313554,0
192.168.72.43,55963,10.200.7.7,3128,6,26/04/201711:11:17,104443,5,0,1076,0
192.168.10.47,51848,10.200.7.6,3128,6,26/04/201711:11:17,11002,3,12,232,3664
```

3.10.2 Capturing Peer-to-Peer Flows

We don't necessarily want to model the level of granularity that's present in the source dataset, within a collection. Cloud storage is cheap, sure, but we simply just don't need or want multiple records per second, per address pair. That's still data we have to examine every time we process the collection.

Instead we can key on address pairs, and lean on reduction annotations to aggregate any repeat records that may occur within a single ingestion transaction. Here's a schema:

```yaml
$defs:
  counter:
    type: number
    reduce: { strategy: sum }

  ip-port:
    type: object
    properties:
      ip: { type: string }
      port: { type: integer }
    required: [ip, port]

  stats:
    type: object
    reduce: { strategy: merge }
    properties:
      packets: { $ref: "#/$defs/counter" }
      bytes: { $ref: "#/$defs/counter" }

type: object
reduce: { strategy: merge }
properties:
  src: { $ref: "#/$defs/ip-port" }
  dst: { $ref: "#/$defs/ip-port" }
  timestamp: { type: string }
  protocol: { enum: [0, 6, 17] }
  millis: { $ref: "#/$defs/counter" }
  fwd: { $ref: "#/$defs/stats" }
  bwd: { $ref: "#/$defs/stats" }
```

And a captured collection into which we'll ingest:

```
collections:
  - name: examples/net-trace/pairs
    key: [/src/ip, /src/port, /dst/ip, /dst/port]
    schema:
      $ref: schema.yaml
      required: [src, dst, protocol, timestamp]

    projections:
      Source.IP: /src/ip
      Source.Port: /src/port
      Destination.IP: /dst/ip
      Destination.Port: /dst/port
      Protocol:
        location: /protocol
        partition: true
      Timestamp: /timestamp
      Flow.Duration: /millis
      Total.Fwd.Packets: /fwd/packets
      Total.Backward.Packets: /bwd/packets
      Total.Length.of.Fwd.Packets: /fwd/bytes
      Total.Length.of.Bwd.Packets: /bwd/bytes
```

Kick off streamed capture:

```
# Start a local development instance, and leave it running:
$ flowctl develop

# In another terminal:
$ examples/net-trace/load-traces.sh
```

3.10.3 Service Traffic by Day

A simplistic view which identifies *services* (endpoints having a port under 1024), and their aggregate network traffic by day:

```
import:
  - pairs.flow.yaml

# Package lambdas with an NPM dependency on Moment.js.
nodeDependencies:
  moment: "^2.24"

collections:
  - name: examples/net-trace/services
    key: [/date, /service/ip, /service/port]
    schema:
      type: object
      reduce: { strategy: merge }
      properties:
        date: { type: string }
        service: { $ref: schema.yaml#/$defs/ip-port }
        stats: { $ref: schema.yaml#/$defs/stats }
      required: [date, service]
```

(continues on next page)

(continued from previous page)

```
    derivation:
      transform:
        fromPairs:
          source: { name: examples/net-trace/pairs }
          publish:
            nodeJS: |
              // Use moment.js to deal with oddball timestamp format, and truncate to␣
↪current date.
              let date = moment(source.timestamp, "DD/MM/YYYYhh:mm:ss").format('YYYY-
↪MM-DD')
              let out = [];

              if (source.src.port < 1024) {
                source.src.ip = source.src.ip.split('.').slice(0, -1).join('.');
                out.push({
                    date: date,
                    service: source.src,
                    stats: source.fwd,
                });
              }
              if (source.dst.port < 1024) {
                source.dst.ip = source.dst.ip.split('.').slice(0, -1).join('.');
                out.push({
                    date: date,
                    service: source.dst,
                    stats: source.bwd,
                });
              }
              return out;
```

Materialize to a test database:

```
$ flowctl materialize --collection examples/net-trace/services --table-name services -
↪-target testDB
```

3.11 An Introduction to Flow

If you're a brand new Flow user, you're in the right place! We're going to walk through the basics of Flow by building a shopping cart backend.

3.11.1 Your First collection

To start with, we're going to define a Flow collection that holds data about each user. We'll have this collection accept user JSON documents via the REST API, and we'll materialize the data in a Postgres table to make it available to our marketing team. Our devcontainer comes with a Postgres instance that's started automatically, so all of this should "just work" in that environment.

Flow collections are declared in a YAML file, like so:

```
collections:
  - name: examples/shopping/users
    key: [/id]
    schema: user.schema.yaml
```

Note that the schema is defined in a separate file. This is a common pattern because it allows your schemas to be reused and composed. The actual schema is defined as:

Listing 1: user.schema.yaml

```yaml
description: "A user who may buy things from our site"
type: object
properties:
  id: { type: integer }
  name: { type: string }
  email:
    type: string
    format: email
required: [id, name, email]
```

We can apply our collection to a local Flow instance by running:

```
$ flowctl build && flowctl develop
```

Now that it's applied, we'll leave that terminal running and open a new one to simulate some users being added.

```
curl -H 'Content-Type: application/json' -d @- 'http://localhost:8081/ingest' <<EOF
{
    "examples/shopping/users": [
        {
            "id": 6,
            "name": "Donkey Kong",
            "email": "bigguy@dk.com"
        },
        {
            "id": 7,
            "name": "Echo",
            "email": "explorer@ocean.net"
        },
        {
            "id": 8,
            "name": "Gordon Freeman",
            "email": "mfreeman@apeture.com"
        }
    ]
}
EOF
```

This will print out some JSON with information about the writing of the new data, which we'll come back to later. Let's check out our data in Postgres:

```
$ psql 'postgresql://flow:flow@localhost:5432/flow?sslmode=disable' -c "select id,␣
↪email, name from shopping_users;"
id |         email          |      name
---+------------------------+----------------
6 | bigguy@dk.com          | Donkey Kong
7 | explorer@ocean.net     | Echo
8 | freeman@apeture.com | Gordon Freeman
(3 rows)
```

As new users are added to the collection, they will continue to appear here. One of our users wants to update their email address, though. This is done by ingesting a new document with the same `id`.

```
curl -H 'Content-Type: application/json' -d @- 'http://localhost:8081/ingest' <<EOF
{
    "examples/shopping/users": [
        {
            "id": 8,
            "name": "Gordon Freeman",
            "email": "gordo@retiredlife.org"
        }
    ]
}
EOF
```

If we re-run the Postgres query, we'll see that the row for Gordon Freeman has been updated. Since we declared the collection key of `[ /id ]`, Flow will automatically combine the new document with the previous version. In this case, the most recent document for each `id` will be materialized. But Flow allows you to control how these documents are combined using *reduction annotations*, so you have control over how this works for each collection. The users collection is simply using the default reduction strategy `lastWriteWins`.

Writing Tests

Before we go, let's add some tests that verify the reduction logic in our users collection. The *tests section* allows us to ingest documents and verify the fully reduced results automatically. Most examples from this point on will use tests instead of shell scripts for ingesting documents and verifying expected results.

```
tests:
  "A users email is updated":
    - ingest:
        collection: examples/shopping/users
        documents:
          - { id: 1, name: "Jane", email: "janesoldemail@email.test" }
          - { id: 2, name: "Jill", email: "jill@upahill.test" }
    - ingest:
        collection: examples/shopping/users
        documents:
          - id: 1
            name: Jane
            email: jane82@email.test
    - verify:
        collection: examples/shopping/users
        documents:
          - id: 1
            name: Jane
            email: jane82@email.test
          - id: 2
            name: Jill
            email: jill@upahill.test
```

Each test is a sequence of `ingest` and `verify` steps, which will be executed in the order written. In this test, we are first ingesting documents for the users Jane and Jill. The second `ingest` step provides a new email address for Jane. The `verify` step includes both documents, and will fail if any of the properties do not match.

We can run the tests using:

```
$ flowctl build && flowctl test
```

3.11.2 Next Steps

Now that our users collection is working end-to-end, Here's some good topics to check out next:

- Learn the basics of CSV ingestion by building the *Products collection*
- Explore reduction annotations by building the *Shopping Cart collection*

Products CSV Ingestion

We'll walk through how to populate our "products" Collection from a CSV file. Here's the schema for a product:

Listing 2: product.schema.yaml

```yaml
description: "A product that is available for purchase"
type: object
properties:
  id: { type: integer }
  name: { type: string }
  price: { type: number }
required: [id, name, price]
```

We want to ingest a CSV with all our products from the old system. This works by sending the CSV over a websocket to flow-ingester, which will convert each CSV row to a JSON document and add it to our products Collection. Here's a sample of our CSV data:

Listing 3: products.csv

```
product_num,price,name
1,0.79,Fruit Rollup
2,0.89,Fruit by the Foot
```

The `price` and `name` columns match the properties in our schema exactly, so it's obvious how those will end up in the final JSON document. But we'll need to tell Flow that the `product_num` column should be mapped to the `id` field. We do this by adding a *projection* to our products Collection.

```yaml
  - name: examples/shopping/products
    schema: product.schema.yaml
    key: [/id]
    projections:
      product_num: /id
```

With this projection, we'll be able to simply send the CSV to flow-ingester over a websocket:

```
cat products.csv | websocat --protocol csv/v1 'ws://localhost:8081/ingest/examples/
↪shopping/products'
```

We'll see the usual JSON response from flow-ingester. For larger CSVs, we may see many such responses as flow-ingester will break it down into multiple smaller transactions.

Next

- Continue the example with the *shopping cart implementation*.
- Learn about ingestion details in the *flow-ingester reference*.
- Learn about projection details in the *projections documentation*.

Shopping Carts

At this point, we have Captured collections setup for users and products, and we're ready to start building the shopping carts. We'll start by defining a Captured collection for cart updates, which will hold a record of each time a user adds or modifies a product in their cart. These cart updates will then be joined with the product information so that we'll have the price of each item. Then we'll create a "carts" collection that rolls up all the joined updates into a single document that includes all the items in a users cart, along with their prices.

Summing Item Quantities

Here's the collection and schema for the cart updates:

Listing 4: cart-updates.flow.yaml

```
collections:
  - name: examples/shopping/cartUpdates
    schema: cart-update.schema.yaml
    key: [/userId, /productId]
```

Listing 5: cart-update.schema.yaml

```yaml
type: object
description: Represents a request from a user to add or remove a product in their␣
↪cart.
properties:
  userId: { type: integer }
  productId: { type: integer }
  quantity:
    description: The amount to adjust, which can be negative to remove items.
    type: integer
    reduce: { strategy: sum }
required: [userId, productId, quantity]
reduce: { strategy: merge }
```

Each cartUpdate document represents a request to add or subtract a quantity of a particular product in a user's cart. If we get multiple updates for the same user and product, the quantities will be summed. This is because of the `reduce` annotation in the above schema.

Joining Cart Updates and Products

We'll define a new *derived collection* that performs a streaming inner join of `cartUpdate` to `product` documents.

```yaml
  - name: examples/shopping/cartUpdatesWithProducts
    key: [/action/userId, /product/id]
    schema:
      type: object
      properties:
        action: { $ref: cart-update.schema.yaml }
        product: { $ref: product.schema.yaml }
      required: [action, product]
      reduce: { strategy: lastWriteWins }

    derivation:
      register:
        initial: null
        schema:
          oneOf:
            - { $ref: product.schema.yaml }
            - { type: "null" }

      transform:
        products:
          source:
            name: examples/shopping/products
          update:
            nodeJS: |
              return [source]

        cartUpdates:
          source:
            name: examples/shopping/cartUpdates
          # Setting the shuffle key to "[/productId]" means that for each cartUpdate␣
↪document from
          # the source, Flow will use the value of the productId field to look up its␣
↪associated
```

(continues on next page)

(continued from previous page)

```
          # register value.
          shuffle: [/productId]
          publish:
            nodeJS: |
              // The register schema says this might be null, so we need to deal with␣
↪that here.
              // If we haven't seen a product with this id, then we simply don't␣
↪publish. This makes
              // it an inner join.
              if (register) {
                // The ! in register! is a typescript non-null assertion. It's␣
↪required since
                // the register schema says it may be null, and safe here because we␣
↪checked.
                return [{action: source, product: register!}];
              }
              return [];
```

There's two main concepts being used here. The first is the *register*. We'll have a unique register value for each product id, since that's the value we're joining on. For each product id, the register value can either be a product document, or null, and the `initial` value is always `null`.

Now let's look at the two *transforms*, starting with `products`. This will read documents from the products collection, and update the register for each one. Note that the default shuffle key is implicitly the key of the source collection, in this case the `/id` field of a `product`. The return value of `[source]` is simply adding the source product to the set of register values. We simply return the value(s) that we'd like to be saved in registers, rather than calling some sort of "save" function. The key for each value must be included in the document itself, and this is verified at build/compile time. We always return a single value here because we're doing a 1-1 join.

Whenever a document is read from the `cartUpdates` collection, the `cartUpdates` transform will read the current value of the register and publish a new document that includes both the `cartUpdate` event and the `product` it joined to. If the register value is not null, then it means that the `products` update lambda has observed a product with this id, and we'll emit a new `cartUpdatesWithProducts` document. This is what makes it an inner join, since we only return a document if the register is not null.

Rolling Up Carts

Now that we have the product information joined to each item, we're ready to aggregate all of the joined documents into a single cart for each user. This is an excellent use case for the `set` reduction strategy. In this case, we're going to apply the reduction annotations to the register schema, and leave the collection schema as `lastWriteWins`. This means that the state will accumulate in the register (one per `userId`), and the collection documents will each reflect the last known state.

Initially, all we'll need is a single transform:

```
import:
  - cart-updates-with-products.flow.yaml
  - cart-purchase-requests.flow.yaml

collections:
  - name: examples/shopping/carts
    schema: cart.schema.yaml
    key: [/userId]
    derivation:
      register:
```

(continues on next page)

(continued from previous page)

```
        initial: { userId: 0, cartItems: {} }
        schema:
          type: object
          properties:
            userId: { type: integer }
            cartItems:
              type: object
              reduce: { strategy: set, key: [/product/id] }
              additionalProperties:
                type: array
                items: { $ref: cart.schema.yaml#/$defs/cartItem }
          required: [userId, cartItems]
          reduce: { strategy: merge }

      transform:
        cartUpdatesWithProducts:
          source:
            name: examples/shopping/cartUpdatesWithProducts
          shuffle:
            - /action/userId
          update:
            nodeJS: |
              return [{
                userId: source.action.userId,
                cartItems: {
                  add: [source]
                }
              }]
          publish:
            nodeJS: |
              return [{
                userId: register.userId,
                items: register.cartItems!.add!,
              }]
```

In the update lambda, we're adding the combined update-product document to the `cartItems`. The use of the `set` reduction strategy means that the item we provided will be deeply merged with the existing set. So if there's already a product with the same `id` in the set, then the `sum` reduction strategy will apply.

Let's take a look at a test case that demonstrates it working end to end. Here we're ingesting some products followed by a series of cart updates. Then we verify the final cart.

Listing 6: cart-tests.flow.yaml

```
import:
  - carts.flow.yaml
  - cart-updates.flow.yaml
  - products.flow.yaml

tests:
  "shopping cart is populated from cartUpdates":
    - ingest:
        collection: examples/shopping/products
        documents:
          - { id: 333, name: "Fruit Rollup", price: 0.79 }
          - { id: 2222, name: "Fruit By The Foot", price: 0.89 }
          - { id: 4004, name: "Gushers", price: 2.95 }
```

(continues on next page)

(continued from previous page)

```
- ingest:
    collection: examples/shopping/cartUpdates
    documents:
      - userId: 1
        productId: 2222
        quantity: 2
      - userId: 1
        productId: 4004
        quantity: 1
      - userId: 1
        productId: 333
        quantity: 1
      - userId: 1
        productId: 2222
        quantity: 1
      - userId: 1
        productId: 333
        quantity: -1

- verify:
    collection: examples/shopping/carts
    documents:
      - userId: 1
        items:
          - product:
              id: 333
              name: Fruit Rollup
              price: 0.79
            action: { quantity: 0 }
          - product:
              id: 2222
              name: Fruit By The Foot
              price: 0.89
            action: { quantity: 3 }
          - product:
              id: 4004
              name: Gushers
              price: 2.95
            action: { quantity: 1 }
```

Next

- *Handle purchases* to learn how to reset the state of a cart.
- Check out the *Derivation Patterns* docs for more examples of how to manage stateful transformations.

Handling Purchases

To put it all together, we'll create a captured collection for requests to purchase a cart, and a final derived collection to hold the complete purchase details.

Here's the schema and Captured collection:

```
collections:
  - name: examples/shopping/cartPurchaseRequests
    schema:
      type: object
      description: "Represents a request from a user to purchase the items in their␣
↪cart."
      properties:
        userId: { type: integer }
        timestamp:
          type: string
          format: date-time
      required: [userId, timestamp]
    # The timestamp is part of the key in order to uniquely identify multiple␣
↪purchase requests for
    # the same user.
    key: [/userId, /timestamp]
```

We'll read these purchase events in a couple of places. First, we'll create a purchases derivation that stores the most recent cart for each user in a register. When it reads a purchase event, it will publish the complete cart contents.

```
import:
  - carts.flow.yaml
  - cart-purchase-requests.flow.yaml

collections:
  - name: examples/shopping/purchases
    schema: purchase.schema.yaml
    key: [/userId, /timestamp]
    derivation:
      register:
        initial: { userId: 0, items: [] }
        schema: cart.schema.yaml
      transform:
        carts:
          source:
            name: examples/shopping/carts
          update:
            nodeJS: return [source]

        purchaseActions:
          source:
            name: examples/shopping/cartPurchaseRequests
          shuffle: [/userId]
          publish:
            nodeJS: |
              return [{
                userId: register.userId,
                timestamp: source.timestamp,
                items: register.items,
              }]
```

The timestamp is again part of the key in order to uniquely identify multiple purchases from the same user. If we were

to materialize the purchases collection, we'd get a separate row for each purchase. We can see this work end to end in the following test case.

Listing 7: purchase-tests.flow.yaml

```yaml
import:
  - cart-updates.flow.yaml
  - carts.flow.yaml
  - cart-purchase-requests.flow.yaml
  - purchases.flow.yaml

tests:
  "shopping cart is purchased":
      # The "&products" here is a yaml feature that lets us re-use this object in
↪later steps with
      # "- ingest: *products".
    - ingest: &products
        collection: examples/shopping/products
        documents:
          - { id: 333, name: "Fruit Rollup", price: 0.79 }
          - { id: 2222, name: "Fruit By The Foot", price: 0.89 }
          - { id: 4004, name: "Gushers", price: 2.95 }

    - ingest: &cartItems
        collection: examples/shopping/cartUpdates
        documents:
          - userId: 1
            productId: 2222
            quantity: 2
          - userId: 1
            productId: 4004
            quantity: 1
          - userId: 1
            productId: 333
            quantity: 1
          - userId: 1
            productId: 2222
            quantity: 1
          - userId: 1
            productId: 333
            quantity: -1
    - ingest:
        collection: examples/shopping/cartPurchaseRequests
        documents:
          - userId: 1
            timestamp: '2020-01-04 15:22:01'

    - verify:
        collection: examples/shopping/purchases
        documents:
          - userId: 1
            timestamp: '2020-01-04 15:22:01'
            items:
              - { product: { id: 333 }, action: { quantity: 0 } }
              - { product: { id: 2222 }, action: { quantity: 3 } }
              - { product: { id: 4004 }, action: { quantity: 1 } }
    - verify:
        collection: examples/shopping/carts
```

(continues on next page)

(continued from previous page)

```
        documents:
```

The last thing we'll do is to reset the state of a user's cart after they complete a purchase. Here we'll leverage Flow's capability to have multiple readers of each collection, and add a `clearAfterPurchase` transform to our `carts` collection.

Here we have both update and publish lambdas. The update lambda clears the set of items in the register by intersecting it with `[]`, using the same `set` reduction strategy. The publish lambda ensures that other readers of the carts collection (and materializations) will observe the now empty cart. This behavior is required in order for the `cart is cleared after purchase` test case to pass:

```
  "cart is cleared after purchase":
    # Here *products and *cartItems refers to the same items defined above.
    - ingest: *products
    - ingest: *cartItems
    - ingest:
        collection: examples/shopping/cartPurchaseRequests
        documents:
          - userId: 1
            timestamp: '2020-01-04 15:22:01'

    # Add and purchase one more product to assert we get a separate purchase
    - ingest:
        collection: examples/shopping/cartUpdates
        documents:
          - userId: 1
            productId: 2222
            quantity: 50

    # Verify that the cart doesn't contain any items that were already purchased
    - verify:
        collection: examples/shopping/carts
        documents:
          - userId: 1
            items:
              - product:
                  id: 2222
                  name: Fruit By The Foot
                  price: 0.89
                action:
                  productId: 2222
                  quantity: 50

    - ingest:
        collection: examples/shopping/cartPurchaseRequests
        documents:
          - userId: 1
            timestamp: '2020-01-04 15:30:44'

    # Verify that we have two distinct purchases
    - verify:
        collection: examples/shopping/purchases
        documents:
          - userId: 1
            timestamp: '2020-01-04 15:22:01'
            items:
```

(continues on next page)

(continued from previous page)

```
        - { product: { id: 333 }, action: { quantity: 0 } }
        - { product: { id: 2222 }, action: { quantity: 3 } }
        - { product: { id: 4004 }, action: { quantity: 1 } }
    - userId: 1
      timestamp: '2020-01-04 15:30:44'
      items:
        - { product: { id: 2222 }, action: { quantity: 50 } }
```

You Made It!

If you've made it this far, then you've seen all the major elements of the Flow programming model. Some recommended next steps are:

- *Try out Flow yourself* using our pre-built dev container.
- Read about *how flow compares to other systems*.
- Check out some more examples.