Docs

Entity Builder
About
About
Getting Started
Cost
Tour
Architecture
Topology
Topology Configuration
Topology Modification
Optimization Guide
Containers
Database
Test Harness
Template Mapper
Transformer
Deployments
Help
Terms of Service
Beam
Getting Started
Deployments
Beam Operations
Help

Configuration

Overview

The Entity Builder topology is composed of steps that describe what should be done with the data. Below is a reference of the different steps that can be used to join, filter and transform data in entity builder. Steps are processed asynchronously. Entity Builder uses internal topics to ensure correct relationships.

Topology configuration is provided to Entity Builder in the join.yaml. Topologies must be targeted to building a single output topic.

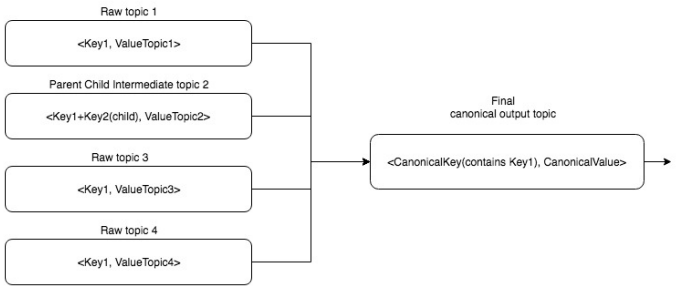
Step Types

This is a full reference of available topology step configuration. Often times it can be unclear how best to translate your data relationships into an optimal set of steps. For this purpose we have built a topology generator that is designed to be more straight forward to configure and will provide an optimal topology.

Shared Key

The shared key step type (`type: shared-key`) is a combination of joining data by a shared key and a transformation of the data using a mapper. It requires that all topics being joined contain all of the configured shared key fields in the message key (except in the case of the Join By Value variation). All other fields in the key must be included in additional keys. The shared key fields are read from the message key to build a partitioning key for grouping data for the same key. The fields are combined in the order they are configured. Changing the order of fields in the step configuration will require a reload of all data.

Diagram



Example Step Configuration

```
topology_steps:
  - step_name: person_with_phones
    type: shared-key
    models:
      - shared_keys:
          - personID
        topics:
          - type: person
          - type: phone
            additional_keys:
              - phoneID
```

A full list of configuration options can be found in Shared Key Configuration

How it works

1. Entity Joiner receives a message from a topic
2. Message is persisted to the persistence table. If the value of the message is null, then the persisted data row is deleted for the message instead.
3. A message is produced to the publish queue for the shared key.
4. Entity Transformer reads the publish queue message, reads data from the persistence table and sends it to the mapper.
5. Mapper returns a canonical message and the Transformer produces to the output topic.

Note: Shared key is the only type that produces messages to the publish queue. A shared key step is required in every topology to create output messages.

Notable Variations

There are several configurations that vary the functionality of the shared key step.

Join By Value

A shared key step can be configured to join a topic using fields from the value. This requires an additional `rekey lookup table` to maintain the relationship between the message key and the shared key built using message fields. This additional table is needed to handle clean up when a message with a null value is received or when the field value being joined by changes

Example Step Configuration

```
topology_steps:
  - step_name: team_sk
    type: shared-key
    models:
      - shared_keys:
          - teamID
        topics:
          - type: team
          - type: person
            shared_key_overrides:
              - shared_key: teamID
                from_value: true
            additional_keys:
              - personID
```

You indicate one or more of the shared keys comes from the value of the message by setting `from_value` on `shared_key_overrides`.

How it works

Create/Update

1. Entity Joiner receives a message from a topic
2. Reads rekey lookup table to get previous partitioning key
3. If previous partitioning key is different from the new partitioning key
    i. Row for the previous partitioning key is deleted from the persistence table

ii. A message is produced to the publish queue for the previous partitioning key
iii. rekey lookup table row is persisted for the new partitioning key

4. if a previous partitioning key was not found
   i. Row is written to rekey lookup table for the new partitioning key.
5. Row is persisted to the persistence table
6. A message is produced to the publish queue for the shared key.
7. Entity Transformer reads the publish queue message, reads data from the persistence table and sends it to the mapper.
8. Mapper returns a canonical message and the Transformer produces to the output topic.

Delete

1. Entity Joiner receives a message from a topic
2. Reads rekey lookup table to get previous partitioning key
3. If previous partitioning key exists
   i. Row for the previous partitioning key is deleted from the persistence table
   ii. A message is produced to the publish queue for the previous partitioning key
   iii. Row is deleted from the rekey lookup table
   iv. Entity Transformer reads the publish queue message, reads data from the persistence table and sends it to the mapper.
   v. Mapper returns a canonical message and the Transformer produces to the output topic.

Join Only

The `join_only` option changes shared key step so that rather than producing to the publish queue it produces to the join queue. Data is not sent to the Entity Transformer for transformation, instead it is persisted to be used in later topology steps. It is an intermediate topology step used to join two or more that have a common key, but that key is not the key desired for final output topic. This option offers the benefit of not needing an intermediate data

transformation step. This step is typically followed by a parent child step or a shared key join by value step

NOTE If the data for a single shared key could exceed 1MB you cannot use the join_only flag. This is due to the way the data is stored in downstream steps. You should instead use a standard shared-key step with a mapper that will produce a message smaller than 1MB with the data you need in downstream steps.

NOTE If not all the data is needed to build the final output message, for instance you need only a single active phone number, then you should consider using a standard shared key and transform the data into an intermediate output message that can be used in later steps.

Example Step Configuration

```
name: teamCanonical
topology_steps:
  - step_name: person_w_phones
    type: shared-key
    join_only: true
    models:
      - shared_keys:
         - personID
        topics:
         - type: person
         - type: phone
           additional_keys:
             - phoneID
           shared_key_overrides:
             - shared_key: personID
               override_key: ownerID
               from_value: true

  - step_name: team_sk
    type: shared-key
    models:
      - shared_keys:
          - teamID
        topics:
          - type: team
          - type: person_w_phones
            shared_key_overrides:
              - shared_key: teamID
```

Overview
Step Types
Shared Key
Parent Child
Rekey
Filter
Lookup
Transform Only
Pass Through
Fan Out
Internal Topics
Join Queue
Publish Queue
Low Priority Join Queue
Low Priority Publish Queue
Configuration Reference

Entity Builder
About
About
Getting Started
Cost
Tour
Architecture
Topology
Topology Configuration
Topology Modification
Optimization Guide
Containers
Database
Test Harness
Template Mapper
Transformer
Deployments
Help
Terms of Service
Beam
Getting Started
Deployments
Beam Operations
Help

```
            override_key: teamID>string
            row_type: person
            from_value: true
          additional_keys:
            - personID
```

General

How it works

This step works identically to the variation of shared key step that is configured without this option. It differs only in that it produces to the join queue instead of the publish queue. **Note** As this flag defines where to produce messages, this option is not allowed if `persist_only` is also set on the step.

Changed Field Tracking

Changed field tracking is a boolean field that creates a `change_field_tracking` element which entails the version (how many times this canonical schema has been changed) and the changed fields which details the path to the value and how that value was changed ("create","delete","update") which is obtained from the difference from the previously produced canonical. The comparing process is all done automatically in the transformer and all that needs to be done is to turn this feature on.

Example Result

```
"changeTracking": {
    "com.optum.exts.common.ChangedFieldsTrac
      "version": 0,
      "changedFields": {
        "array": [
          {
            "path": "publications[1]",
            "operationType": "create"
          }
        ]
      }
    }
  }
```

WARNING In you deploy.yaml you will need to set min.compaction.lag.ms to ensure no cases where canonical updates override each other in kafka before

they can be compared. This can cause records with relevant changes to be ignored as they will not be considered.

NOTE Changed Field Tracking is not suitable for all cases. Some potential use cases include highly utilized api's with many consumers (allowing consumers to have a higher level of granulairty with what updates they want to include) or canonicals that are primarily consumed into databases that have complexity issues with many updates (and stand to benefit from the potential exclusion of a wide variety of canonical updates)

Example Step Configuration

```
name: teamCanonical
topology_steps:
  - step_name: person_w_phones
    type: shared-key
    join_only: true
    models:
      - shared_keys:
         - personID
        topics:
         - type: person
         - type: phone
           additional_keys:
             - phoneID
           shared_key_overrides:
             - shared_key: personID
               override_key: ownerID
               from_value: true

  - step_name: team_sk
    type: shared-key
    change_field_tracking: true
    models:
      - shared_keys:
          - teamID
        topics:
          - type: team
          - type: person_w_phones
            shared_key_overrides:
              - shared_key: teamID
                override_key: teamID>string
                row_type: person
                from_value: true
            additional_keys:
```

```
    - personID
```

How it works

Change field tracking is an easy-to-use feature with a lot of work being done behind the scenes. When it is active, entity-builder will store canonical records in the persistence store. Then, after an update is processed, the stored record is compared to the new one. This process is done with generic diffing and will populate the value of `changedFields` on the canonical message. It also affects the amount of processing power the transformer requires.

Changed Field Tracking Array Comparisons

By default changed field tracking will only indicate and compare based on the indices of the elements that have changed in an array. The results can be misleading if the mapper produces messages in an inconsistent order or an element is removed from an array or inserted in the middle of an array. Optionally, for arrays of complex elements, you can configure the fields that make the elements of an array unique. When provided, array element will be compared to the element that has the same unique key. This is done by configuring a list of `array_comparisions` in `changed_field_tracking_settings` on the shared-key step. One array comparison config can be provided per array in the canonical schema. It consists of two fields path (the location of the array field the configuration applies to) and an optional list `element_key_fields` (the list of fields within each array element that makes the element unique). For primitive arrays only the path is required and the values will be used as the key.

The output for create and update `operationTypes` remains the same without the array comparison configuration, although the comparison will be made to the previous output that relates to its given key rather the indice in the array. For delete, `deletedArrayElements` will include the values for the configured `element_key_fields` that has been removed.

Example Result

```
"changeTracking": {
    "com.optum.exts.common.ChangedFieldsTrac
      "version": 0,
      "changedFields": {
        "array": [
          {
            "path": "publications[0]",
            "operationType": "create"
          },
          {
            "path": "publications[]",
            "operationType": "delete",
            "deletedArrayElements": {
              "array": [
                {
                "path": "publisher",
                "value": "Me"
                }
              ]
            }
          }
        ]
      }
    }
  }
```

Example Step Configuration

```
- step_name: book
  type: shared-key
  changed_field_tracking: true
  changed_field_tracking_settings:
    array_comparisons:
      - path: publications
        element_key_fields:
          - publisher
  models:
  - shared_keys:
      - id
    topics:
      - type: book_w_author
        additional_keys:
          - authorID
      - type: book_w_category
        additional_keys:
          - categoryID
      - type: publication
```

```
    shared_key_overrides:
      - shared_key: id
        override_key: bookID
    additional_keys:
      - id
```

Row Type Override

When row types might have potential collisions row_type_override can be used to rename the referenced name of a given row. The example below shows the row type person being overridden. This feature can be used on the child and parent.

Example Step Configuration

```
- step_name: person_w_phones
    type: shared-key
    join_only: true
    models:
      - shared_keys:
         - personID
        topics:
         - type: person
           row_type_override: prsn
         - type: phone
           additional_keys:
             - phoneID
           shared_key_overrides:
             - shared_key: personID
               override_key: ownerID
               from_value: true
```

How it works

Will change the referenced row type to the new overridden name. This will make it so future references to this row should be through the row_type_override name specified. For example references to the row type in future parts of the topology and in the rows received by the mapper.

Multi Model

Shared key steps can define multiple models. This allows different shared keys and different sets of topics to be defined. This variation can be used when two or more distinct sets of topics should be

transformed into the same output message format and topic.

Example Step Configuration

```
- step_name: coordination_of_benefits
    type: shared-key
    models:
      - shared_keys:
          - PARTN_NBR
          - CNSM_ID
          - SRC_CD
          - LGCY_SRC_ID
          - COB_COV_TYP_CD
          - OI_TYP_CD
          - COB_PRIMACY_EFF_DT
        topics:
          - type: cnsm_cob_prmcy_w_srch_and_
      - shared_keys:
          - PARTN_NBR
          - CNSM_ID
          - SRC_CD
          - LGCY_SRC_ID
          - MDCR_PART_TYP_CD
          - MDCR_PRIMACY_EFF_DT
        topics:
          - type: cnsm_mdcr_prmcy_w_srch_oth
            additional_keys:
              - COB_COV_TYP_CD
```

How it works

This step works the same as if two shared key steps were configured except that it shares a persistence table.

NOTE The mapper for this step will need to be able to handle data from both sets of topics to produce the correct output messages.

Parent Child

The parent child step type (`type: parent-child`) allows connecting two topics where there is a relationship between two topics and the key(s) that connect the two topics is in one of the topics' message value. A common use case for this step is joining topics on a foreign key relationship.

The parent-child step produces output messages to the join queue that are used in later steps. The key for the message is the combination of the parent message key and child message key

Example Step Configuration

```
topology_steps:
  - step_name: city_w_state
    type: parent-child
    parent_child_configs:
      child_config:
        type: city
        keys:
          - cityID
      parent_config:
        type: state
        keys:
          - stateID
```

A full list of configuration options can be found in Parent Child Configuration

How it works

Child message

Create/Update

1. Entity Joiner receives a message from the child topic
2. Reads child table to read previous parent key
3. If previous parent key is different from the new parent key
    i. Row for the previous parent key is deleted from the persistence table
    ii. A message is produced to the join queue for the previous parent key / child key
    iii. child lookup table row is persisted for the new partitioning key
4. if a previous parent key was not found in the child table
    i. Row is written to child table for the new parent key
5. Row is persisted to the persistence table
6. A message is produced to the join queue for the new parent key / child key

Delete

1. Entity Joiner receives a message from a topic
2. Reads child table to get previous parent key
3. If previous parent key exists
    i. Row for the previous parent key is deleted from the persistence table
    ii. A message is produced to the join queue for the previous parent key / child key
    iii. Row is deleted from the child table

Parent message

Create/Update

1. Entity Joiner receives a message from the parent topic
2. Row is written to persistence table
3. RowPKs (child keys) are read from persistence table
4. A message is produced to the join queue for each child key

Note as child key is needed for the message to the join queue, if there are no child records in the persistence table then no messages will be published.

Delete

1. Entity Joiner receives a message from a topic
2. Row is deleted from the persistence table
3. RowPKs (child keys) are read from persistence table
4. A message is produced to the join queue for each child key

Notable Variations

Persist Parent Only

A less common, but still useful in the right circumstances, variation of a parent-child step is persist parent only. In this case both child and parent topics are configured and consumed, but only the parent record data is persisted. For child records a dummy row with an empty value field is persisted.

This reduces data duplication and most importantly can even out partition size in the data store. As child records are stored on the parent records partition, if there are thousands of children this can result in uneven partitions in the persistence store.

Note This should not be used to deal with low cardinality lookup table data

*Example Step Configuration

```
- step_name: application_w_category
    type: parent-child
    parent_child_configs:
      persist_parent_only: true
      child_config:
        type: application
        keys:
          - applicationID
      parent_config:
        type: application_category
        keys:
          - categoryID
```

How it works

This works the same as a standard parent-child table except that child records are not stored in the value column. If child data is needed for mapping then the topic should be included in a later shared-key step.

From Key

In some cases you want the value that is pulled and compared to come from the key of the records linked. Some direct examples are when you use a persist-parent-only parent-child step and wish to link it to another parent-child step using the child key. Another potential case is when you want to simplify the process when your linking value is inside another value such as `DATA>key_value`.

Note In given example book_w_author is a persist-parent-only parent-child step with a child key being referenced

*Example Step Configuration

```
- step_name: book_w_author_and_category
    type: parent-child
    parent_child_configs:
      child_config:
        type: book_w_author
        keys:
          - authorID
          - id
          - categoryID
        parent_keys:
          - parent_key: id
            child_key: categoryID
            from_key: true
      parent_config:
        type: category
        keys:
          - id
```

How it works

This check simply pulls from the key instead of the value when specified.

Low Priority Parent updates

As parent updates trigger reprocessing of all related child keys, a parent update can lead to spikes in lag on the join-queue that can lead to delays in processing for other messages. A parent-child step can be marked as low priority which will divert messages triggered by parent updates to the low priority join queue. Processing of the messages on the low priority join queue is deferred until higher priority messages have been processed.

Note Downstream processing triggered by low priority parent updates will maintain low priority. They will produce to low priority join queue and low priority publish queue

Example Step Configuration

```
- step_name: application_w_category
    type: parent-child
    parent_child_configs:
      low_priority: true
      child_config:
```

```
        type: application
        keys:
          - applicationID
      parent_config:
        type: application_category
        keys:
          - categoryID
```

How it works

This works the same as a standard parent-child table except that parent updates produce messages for each related child to the low priority join queue to be processed after high priority messages are completed.

Row Type Override

When row types might have potential collisions row_type_override can be used to rename the referenced name of a given row. The example below shows the row type person being overridden. This feature can be used on the child and parent.

Example Step Configuration

```
- step_name: phone_w_person
    type: parent-child
    parent_child_configs:
      child_config:
        type: phone
        keys:
          - phoneID
        parent_keys:
          - parent_key: personID
            child_key: ownerID
            output_key: personID
      parent_config:
        type: person
        row_type_override: prsn
        keys:
          - personID
```

How it works

Will change the referenced row type to the new overridden name. This will make it so future references to this row should be through the row_type_override name specified. For example

Docs

Entity Builder
About
About
Getting Started
Cost
Tour
Architecture
Topology
Topology Configuration
Topology Modification
Optimization Guide
Containers
Database
Test Harness
Template Mapper
Transformer
Deployments
Help
Terms of Service
Beam
Getting Started
Deployments
Beam Operations
Help

references to the row type in future parts of the topology and in the rows received by the mapper.

Join by a subset of parent keys

Parent child steps can also be configured to join children to parents using only a subset of the parent keys. This will result in many children joined to many parents. This is configured by setting keys that are not joined to the child type in the additional_keys field of parent_config.

*Example Step Configuration

```
- step_name: book_by_author
    type: parent-child
    parent_child_configs:
      child_config:
        type: book_author
        keys:
          - bookID
          - authorID
        parent_keys:
          - parent_key: bookID
            child_key: bookID
            output_key: bookID
      parent_config:
        type: book
        keys:
          - bookID
        additional_keys:
          - edition
```

How it works

This works the same as a standard parent-child step except that multiple parent rows will be joined to each child, meaning downstream steps will also receive all parents.

Rekey

The rekey step allows you to add fields from the message value to the message key for joining purposes. This was previously achieved with a `parent-child` step with a parent type that was not configured in topics.yaml.

Example Step Configuration

```
topology_steps:
  - step_name: city_by_state_id
    type: rekey
    rekey_config:
      type: city
      keys:
        - cityID
      keys_from_value:
        - path: stateID
```

A full list of configuration options can be found in Rekey Configuration

How it works

Create/Update

1. Entity Joiner receives a message from the child topic and builds
2. Reads child table to read previous value keys for the message
3. If previous value keys are different from the new value keys
    i. A delete sent to downstream steps with keys_from_value added to the message key based on values in the child table
    ii. child lookup table row is persisted for the new value keys
4. if a previous key was not found in the child table
    i. Row is written to child table for the new keys_from_value
5. Message is passed along to downstream steps with the new keys_from_value added to the message key.

Delete

1. Entity Joiner receives a message from a topic
2. Reads child table to get previous value keys
3. If previous value keys exist
    i. Key is modified based on values from child table and delete is sent to downstream steps
    ii. Once downstream steps have all succeeded, row is deleted from the child table

Docs

Entity Builder
About
About
Getting Started
Cost
Tour
Architecture
Topology
Topology Configuration
Topology Modification
Optimization Guide
Containers
Database
Test Harness
Template Mapper
Transformer
Deployments
Help
Terms of Service
Beam
Getting Started
Deployments
Beam Operations
Help

Filter

The filter step type (`type: filter`) allows a topic to be filtered to remove messages that should not be processed in a subsequent step of the topology.

Example Step Configuration

```
topology_steps:
  - step_name: phone_filter
    type: filter
    filter_config:
      type: phone
      output_type_prefix: filtered
      any_match:
        - conditions:
            - field_name: active
              from_value: true
              blacklist:
                - "false"
            - field_name: phoneType
              whitelist:
                - cell
                - home
```

Filter steps can be configured with one to many matches in the any_match field. If any of these matches succeeds, the message will be passed along to downstream steps. Otherwise, the message will be filtered out. Steps that wish to use the filter messages should be configured to use the name of the filter step. Any step that uses the raw topic type will get the unfiltered messages.

Each match can be configured with one to many conditions. All conditions for a match must pass for a match to succeed. A condition can be defined for a field in the message key or message value using the `from_value` property.

Types of Conditions

Several types of conditions can be configured in a filter step.

Blacklist/Whitelist Condition

A blacklist/whitelist condition can define either a whitelist (the field must match one of the values in the list) or a blacklist (the field must not match one of the values in the list). Configuring "null" in the whitelist or blacklist will match null values for the field.

Examples of Blacklist/Whitelist Condition Configuration

```
- step_name: categorized_books
  type: filter
  filter_config:
    output_type_prefix: categorized
    type: book_data
    any_match:
      - conditions:
        - field_name: categoryID
          from_value: true
          blacklist:
            - "-1"
```

In this example categoryID values with -1 would be filtered out for not meeting the requirements set.

A blacklist/whitelist condition can also specify a wildcard match field which is a boolean value to specify the use of asterisks as a wildcard that wil match 0 or more characters.

```
- step_name: categorized_books
  type: filter
  filter_config:
    output_type_prefix: categorized
    type: book_data
    any_match:
      - conditions:
        - field_name: categoryID
          from_value: true
          wildcard_match: true
          blacklist:
            - "-*"
            - "*1"
```

In this example categoryID values that match regular expression values "-" *or* "1" would be filtered out for not meeting the requirements set.

Changed Fields Condition

A changed fields condition is used to filter messages so that only changes to configured fields trigger updates.

Example Changed Fields Condition Configuration

```
- step_name: book_title
  type: filter
  filter_config:
     type: book_data
     output_type_prefix: filtered
     any_match:
        - conditions:
             - changed_fields:
                  - title
```

In this example only changes made to the configured field title will trigger updates.

Ignore Changed Fields Condition

An ignore changed fields condition is used to filter messages so that changes to listed fields will not trigger updates, while all other changes to other fields will.

Example Ignore Changed Fields Condition Configuration

```
- step_name: book_title
  type: filter
  filter_config:
    type: book_data
    output_type_prefix: filtered
    any_match:
      - conditions:
        - ignore_changed_fields:
          - title
          - authorID
```

In this example any changes to *only* the fields title and/or authorID will be ignored. Changes to fields not in the list will *always* trigger updates.

Please note that you can only have a single ignore changed fields condition in a step, and you also cannot have both an ignore changed fields condition and a changed fields condition in a single step.

Ignoring fields in an array is not supported. You must specify avro schema defaults for ignored fields.

Field Length Condition

A field length condition defines what the length an input field string must be using less than and/or greater than integer configuration values.

Example Field length Condition Configuration

```
- step_name: state_cleaned
  type: filter
  filter_config:
    type: state
    output_type_prefix: filtered
    any_match:
      - conditions:
        - field_name: region
          from_value: true
          field_length:
            less_than: 8
```

In this example all state's region value that have a string length 8 or greater would be filtered out for not meeting the requirements set.

Numeric Condition

A numeric condition defines numeric boundaries on a numeric input field using less than/less than or equal to and/or greater than/greater than or equal to floating point configuration values.

Example Numeric Condition Configuration

```
- step_name: book_price
  type: filter
  filter_config:
    type: book_data
    output_type_prefix: filtered
    any_match:
      - conditions:
        - field_name: price
          row_type: book
          from_value: true
          numeric_value:
            less_than: 8.50
```

```
      greater_than_or_equal_to: 1.25
```

In this example all book prices that are 8.50 or greater or less than 1.25 would be filtered out for not meeting the requirements set.

Date Condition

A date condition defines date/timestamp boundaries on a date formatted string input field using before and/or after date/timestamp values. Before and after date configurations are not inclusive and must be in the format of yyyy-mm-dd or yyyy-mm-ddThh:mm:ss.sssZ.

Example Date Condition Configuration

```
- step_name: book_publish_date
  type: filter
  filter_config:
    type: book_data
    output_type_prefix: filtered
    any_match:
      - conditions:
        - field_name: publishDate
          row_type: book
          from_value: true
          date:
            before: 2019-01-01
            after: 2002-01-01
```

In this example all book publish dates that are after January 1st 2019 or before January 1st 2002 would be filtered out for not meeting the requirements set.

The results of a filter step are used in downstream steps by configuring a type of `output_type_prefix` _ `type`, not the filter step name. For the example above you would configure `filtered_phone`

A full list of configuration options can be found in Filter Configuration

How it works

Message not filtered

Docs

Entity Builder
About
About
Getting Started
Cost
Tour
Architecture
Topology
Topology Configuration
Topology Modification
Optimization Guide
Containers
Database
Test Harness
Template Mapper
Transformer
Deployments
Help
Terms of Service
Beam
Getting Started
Deployments
Beam Operations
Help

1. Entity Joiner receives a message from a topic
2. Configured matches are applied in order. As soon as one match is found that succeeds, the messages is passed through to the downstream steps.
3. Message key is stored in the filter table. This keeps track of messages that have passed the filter so if the message is filtered by value it can be cleaned up properly.

Filtered by key field condition

1. Entity Joiner receives a message from a topic
2. Configured matches are applied in order. No matches pass and at least one condition configured for a key field fails.
3. Offset for the message is committed and joiner moves on to the next message.

Note As Kafka message keys are immutable, filtering based on a key field takes precedence over filtering based on a value field. This is because filtering by key is more efficient than by value. No interactions are needed with the filter table and the message is not sent to downstream steps.

Filtered by value field condition

1. Entity Joiner receives a message from a topic
2. Configured matches are applied in order. No matches pass and all failed conditions are for value fields.
3. The filter table is queried for the message key. If the key is not found in the table, the offset for the message is committed, and it is not sent to downstream steps.
4. If the key is in the filter table the message is sent to downstream steps and treated as a delete message (null message value).
5. Once all downstream steps have cleaned up data for the message and produced deletes downstream the message key is removed from the filter table and the offset is committed.

Notable Variations

Docs

Entity Builder
About
About
Getting Started
Cost
Tour
Architecture
Topology
Topology Configuration
Topology Modification
Optimization Guide
Containers
Database
Test Harness
Template Mapper
Transformer
Deployments
Help
Terms of Service
Beam
Getting Started
Deployments
Beam Operations
Help

Multiple Topic Filter

Filter steps can be configured with multiple types.
Example Step Configuration

```
- step_name: delete_filter
    type: filter
    filter_config:
      output_type_prefix: filtered
      types:
        - phone
        - person
        - team
      any_match:
        - conditions:
            - field_name: headers>operation
              from_value: true
              blacklist:
                - "DELETE"
```

Each filtered type is used in downstream steps by configuring `output_type_prefix` _ `type`. The above example has 3 output types, `filtered_phone`, `filtered_person`, `filtered_team`.

Lookup

Lookup steps (`type: lookup`) are used to store data from a single raw topic in the persistence store which can then be queried directly from the mapper. Lookup steps can be used on either raw types or filtered raw types. This is a replacement for a common pattern of using `persist-only` shared key steps and querying the data from the mapper.

Note Lookup steps cannot be referenced by other steps.

Changes to a record specified in the `type` of a lookup steps configuration will not trigger a remapping of the messages that previously used it. A full example of this initialization can be seen in the lookup-table example mapper.

Cache

Docs

Entity Builder
About
About
Getting Started
Cost
Tour
Architecture
Topology
Topology Configuration
Topology Modification
Optimization Guide
Containers
Database
Test Harness
Template Mapper
Transformer
Deployments
Help
Terms of Service
Beam
Getting Started
Deployments
Beam Operations
Help

For those utilizing the `ebaas-mapper-common-golang` framework, you can see example usage here. The lookup cache will keep records in a cache in the memory of the mapper. Canonical messages can then be enriched with this data. The cache can be configured using some of the parameters found here. Cache size configuration is based on the number of unique keys. If you're using a grouping key lookup, you should pay extra attention to this. The number of grouping keys will be limited by cache size, not the total number of rows.

Example Step Configuration

```
- step_name: state_lookup
    type: lookup
    lookup_config:
      type: state
      keys:
        - stateCD
```

A full list of configuration options can be found in Lookup Configuration

How it works

1. Entity Joiner receives a message from a lookup type topic
2. Entity Joiner writes row to the Lookup table. This uses the persistence table, but with the table name `lookup`.
3. Offset for the message is committed and joiner moves on to the next message

Message deleted

1. Entity Joiner receives a message from a lookup type topic
2. Entity Joiner uses the key fields to delete the row from the Lookup table
3. Offset for the message is committed and joiner moves on to the next message

Transform Only

Docs

Entity Builder
About
About
Getting Started
Cost
Tour
Architecture
Topology
Topology Configuration
Topology Modification
Optimization Guide
Containers
Database
Test Harness
Template Mapper
Transformer
Deployments
Help
Terms of Service
Beam
Getting Started
Deployments
Beam Operations
Help

A transform only step (`type: transform`) offers a way to map a single raw record to a canonical message. By default, it is stateless, meaning no data is persisted during transformation. This step is handled entirely in the Entity Transformer and Mapper.

Note Because there is no state this step can only be used to map 1 raw record to a 1 canonical message.

Example Step Configuration

```
- step_name: member
    type: transform
    transform_config:
      type: raw_member
```

A full list of configuration options can be found in Transformation Configuration

How it works

1. Entity Transformer receives a message from the raw topic
2. Sends message to the Mapper
3. Mapper returns a canonical message and the Transformer produces to the output topic.

Pass Through

A Pass Through step (`type: pass-through`) allows you to use a code-based filter on a single topic, by providing a pass through container rather than a traditional mapper. It does not allow for the transformation of the messages themselves, it only allows for the filtering of which messages make it to the output topic.

Pass Through is generally transform only, and stateless. If a `persist-only` step, like lookup, is used then this is no longer the case, as the joiner and the persistence store will be utilized. It is also no longer stateless if `minimize_tombstones` is enabled.

One of the benefits of `pass-through`, as opposed to using `transform`, is that you do not need to maintain

canonical schemas within the containers. The schema of the canonical messages will always be the same as those of the raw topics. This also means that if your raw topic schema changes slightly, your container filtering logic may continue to work, as long as the changes are not relevant to the workings of that logic.

Pass Through has its own contract with the transformer, and all Pass Through containers must implement this contract. Currently, EBaaS supports pass through contract containers in its golang mapper library. A full library-utilizing container example can be found here. Users can also implement the contract in the language of their choice.

Example Step Configuration

```
- step_name: member
    type: pass-through
    pass_through_config:
      type: raw_member
```

How it works

1. Entity Transformer receives a message from the raw topic
2. Sends message to the Pass Through container, where the message is filtered
3. The container returns a status and the Transformer produces an unmodified message to the output topic based on this status.

Fan Out

The fan out step allows you to add fields from an array field in message value to the message key for joining purposes. Each element in the array will be sent as a message to downstream steps. An array with 5 elements will be sent to downstream steps 5 times each with a key containing the information pulled from that element.

Note A fan out step will only allow you to join by an array by pulling fields into the key. The original message value is sent to downstream steps unchanged.

Docs

Entity Builder
About
About
Getting Started
Cost
Tour
Architecture
Topology
Topology Configuration
Topology Modification
Optimization Guide
Containers
Database
Test Harness
Template Mapper
Transformer
Deployments
Help
Terms of Service
Beam
Getting Started
Deployments
Beam Operations
Help

Example Step Configuration

```
- step_name: fan_out_book
    type: fan-out
    fan_out_config:
      type: book
      keys:
        - bookID
      array_path: authors
      fan_out_keys:
        - path: authorID>int
          output_key: authorID
```

A full list of configuration options can be found in Fan Out Configuration

How it works

Create/Update

1. Entity Joiner receives a message from the child topic and builds.
2. Reads fan out table to read previous value keys for the message.
3. For each element in the array field to fan out by
    i. A message is created with the fan out key fields added to the message key
    ii. The modified message is sent to downstream steps to be processed
    iii. If fan out keys were not in the fan out table previously, a new row is written to the fan out table
4. For each row in the fan out table that was not found in the array to fan out
    i. A delete message is created with the fan out key fields added to the message key
    ii. The delete message is sent to downstream steps to be processed
    iii. The fan out table row is deleted

Delete

1. Entity Joiner receives a message from a topic
2. Reads fan out table to read previous value keys for the message.
3. For each row from the fan out table

i. A delete message is created with the fan out key fields added to the message key
ii. The delete message is sent to downstream steps to be processed
iii. The fan out table row is deleted

Notable Variations

Fan Out by Row Type

Fan out steps can also be configured to fan out based on values of a non array field within a row type. A common use for this is fanning out a join only shared key step by values from a row type with multiple rows. To configure a case like this, array_path is not provided and row_type is required.

Example Step Configuration

```
- step_name: fan_out_book
    type: fan-out
    fan_out_config:
      type: book_join
      row_type: book
      keys:
        - bookID
      fan_out_keys:
        - path: authorID>int
          output_key: authorID
```

How it works

This configuration works identically to the fan out step with notable difference. While a fan out step on an array field does not modify the message value at all (the array remains in the value and only the key is modified), fanning out by a row type limits the rows sent to downstream steps for the row type to those associated to the fanned out key.

Internal Topics

Entity-Builder ensures that all messages for the same key are processed synchronously, in order, to prevent data issues due to race conditions. This is the reason for parent-child and join by value shared-key steps. To accomplish this, entity-builder uses several internal

topics in order to take advantage of the kafka partitions and the order guarantees they provide.

Join Queue

The join queue is a topic used as the output topic for parent-child and join only shared-key steps.

The topic name is `ebaas_prefix`.`canonical_name`.join-queue`optional encrypted suffix`. `ebaas_prefix` is configured in `topics.yaml`. `canonical_name` is configured as the `name` field in `join.yaml`. The topic will have a `_encrypted` suffix if beam encryption is enabled.

Ex. `EBAAS.hemi.attperf.order_payment_schedules.join-queue_encrypted`

A single join queue is used for the entire entity-builder topology.

Message Format

Key

field	description
stepName	The name of the step that produced the message. This is used to determine which step config to use to process the message.
parentKeyFields	This is an array of the field names/values from the parent_config of the step that produced it. For shared-key join by value this is populated with the shared keys.
childKeyFields	This is an array of the field names/values from the child_config of the step that

field	description
	produced it. For shared-key join by value this is an empty array.

Example

```
{
  "stepName":"phone_w_person",
  "parentKeyFields":[{"name":"personID","val
  "childKeyFields":[{"name":"phoneID","value
}
```

Value

field	description
ChildType	The row_type for the child row in the step that produced the message.
ParentType	The row_type for the parent row in the step that produced the message.

Example

```
{
  "ChildType":"phone",
  "ParentType":"person"
}
```

Join message handling

Messages from the join queue are handled identically to a message that comes from a raw kafka topic, except that the "value" is read from the previous steps persistence table.

1. The `stepName` from the join queue message key as the previous step persistence table to read from.

2. The parent key is constructed from the `parentKeyFields`
3. The previous parent row is read from the previous step persistence table where:
   - the `partitioning_key` is equal to the parent key
   - the `row_type` is equal to the `ParentType` from the message value
   - row_primary_keys is empty
4. The child key is constructed from the `childKeyFields`
5. The previous child row is read from the previous step persistence table where:
   - the `partitioning_key` is equal to the parent key
   - the `row_type` is equal to the `ChildType` from the message value
   - row_primary_keys is equal to the child key

The remaining joiner functionality is the same with the parent and child rows used as the "value" of the message.

Note Unlike a raw topic, the join queue message is never a tombstone. Instead, a message is treated as a delete if no child row is found in the previous step persistence table.

Publish Queue

The publish queue is a topic used by joiner to communicate to Entity Transformer that the data for shared-key has changed and needs to be remapped. Like the join queue, there is a single publish queue per topology.

The topic name is `ebaas_prefix`.`canonical_name`.publish-queue`optional encrypted suffix`. `ebaas_prefix` is configured in `topics.yaml`. `canonical_name` is configured as the `name` field in `join.yaml`. The topic will have a `_encrypted` suffix if beam encryption is enabled.

Ex. `EBAAS.hemi.attperf.order_payment_schedules.publish-queue_encrypted`

Message format

Key

field	description
name	The shared-key step the message was produced by
key	The partitioning_key that needs to be remapped

Example

```
{
  "name":"team_canonical",
  "key":"2"
}
```

Value

field	description
data	An array of fields that make up the partitioning key. Used to construct the keyData map provided to the mapper

Example

```
{
  "data":[{"name":"teamID","value":"2","type
}
```

Low Priority Join Queue

The low priority join queue allows for prioritized joining. Message format is identical to the join queue.

The topic name is `ebaas_prefix`.`canonical_name`.low-priority-join-

queue. `ebaas_prefix` is configured in `topics.yaml`. `canonical_name` is configured as the `name` field in `join.yaml`.

Ex. `EBAAS.hemi.attperf.order_payment_schedules.low-priority-join-queue`

A single low priority join queue is used for the entire entity-builder topology.

Message handling

Messages are read from low priority join topic after the configured time without a message from a higher priority topic (Consume Priority Configuration).

1. Message is read from the low priority join queue
2. `ebaas-priority` header is added to the message
3. Message is produced to the join queue.

Messages are redirected to the standard join queue to prevent race conditions that could arise with the same join key being processed from both the join queue and low priority join queue. The `ebaas-priority` header is used to ensure that messages from the low priority join queue result in messages produced back to the low priority join queue and the low priority publish queue by down stream steps.

Low Priority Publish Queue

The low priority publish queue allows for prioritized transformation. Message format is identical to the publish queue.

The topic name is `ebaas_prefix`.`canonical_name`.low-priority-publish-queue. `ebaas_prefix` is configured in `topics.yaml`. `canonical_name` is configured as the `name` field in `join.yaml`.

Ex. `EBAAS.hemi.attperf.order_payment_schedules.low-priority-publish-queue`

Entity Builder
About
About
Getting Started
Cost
Tour
Architecture
Topology
Topology Configuration
Topology Modification
Optimization Guide
Containers
Database
Test Harness
Template Mapper
Transformer
Deployments
Help
Terms of Service
Beam
Getting Started
Deployments
Beam Operations
Help

A single low priority publish queue is used for the entire entity-builder topology.

Message handling

Messages are read from low priority join topic after the configured time without a message from a higher priority topic (Consume Priority Configuration).

1. Message is read from the low priority publish queue
2. Message is produced to the publish queue.

Messages are redirected to the standard publish queue to prevent race conditions that could arise with the same publish key being transformed from both the publish queue and low priority publish queue.

Configuration Reference

General configs

Field	
name	The name of to construct topic names
global_config	An optional f for the joiner
global_config.key_separator	You shouldn'
global_config.empty_key_substitute	You shouldn'
global_config.trim_front_whitespace	When set to whitespace f join. Applies key and mes
global_config.trim_back_whitespace	When set to whitespace f join. Applies key and mes

Field	
topology_steps	The list of joi
topology_steps.step_name	The name of the step. Res subsequent s the `type` in `parent_chi` `parent_chi` or `models.t`
topology_steps.type	The type of t

Shared Key Configuration

Field
topology_steps.minimize_tombstones

Field
topology_steps.changed_field_tracking
topology_steps.changed_field_tracking_settings.arra
topology_steps.changed_field_tracking_settings.arra

Field
topology_steps.changed_field_tracking_settings.arra
topology_steps.models
topology_steps.models.shared_keys
topology_steps.models.topics
topology_steps.models.topics.type
topology_steps.models.topics.row_type_override

Field
topology_steps.models.topics.additional_keys
topology_steps.models.topics.shared_key_overrides
topology_steps.models.topics.shared_key_overrides
topology_steps.models.topics.shared_key_overrides
topology_steps.models.topics.shared_key_overrides
topology_steps.models.topics.shared_key_overrides

Field
topology_steps.join_only
topology_steps.persist_only

Parent Child Configuration

Field
topology_steps.parent_child_configs
topology_steps.parent_child_configs.buckets

Field
topology_steps.parent_child_configs.child_config
topology_steps.parent_child_configs.child_config.typ
topology_steps.parent_child_configs.child_config.rov
topology_steps.parent_child_configs.child_config.key
topology_steps.parent_child_configs.child_config.par
topology_steps.parent_child_configs.child_config.par
topology_steps.parent_child_configs.child_config.par
topology_steps.parent_child_configs.child_config.par
topology_steps.parent_child_configs.child_config.par
topology_steps.parent_child_configs.child_config.par

Field
topology_steps.parent_child_configs.parent_config
topology_steps.parent_child_configs.parent_config.ty
topology_steps.parent_child_configs.parent_config.r
topology_steps.parent_child_configs.parent_config.k
topology_steps.parent_child_configs.parent_config.a
topology_steps.parent_child_configs.persist_parent_

Rekey Configuration

Field
topology_steps.rekey_config
topology_steps.rekey_config.type
topology_steps.rekey_config.keys

Field
topology_steps.rekey_config.keys_from_value
topology_steps.rekey_config.keys_from_value.path
topology_steps.rekey_config.keys_from_value.outpu
topology_steps.rekey_config.keys_from_value.row_t
topology_steps.rekey_config.keys_from_value.outpu

Filter Configuration

Field
topology_steps.filter_config
topology_steps.filter_config.type
topology_steps.filter_config.types
topology_steps.filter_config.output_type_prefix
topology_steps.filter_config.any_match
topology_steps.filter_config.any_match.conditions
topology_steps.filter_config.any_match.conditions.fie
topology_steps.filter_config.any_match.conditions.fro

Field
topology_steps.filter_config.any_match.conditions.wi
topology_steps.filter_config.any_match.conditions.bl
topology_steps.filter_config.any_match.conditions.wl
topology_steps.filter_config.any_match.conditions.ch
topology_steps.filter_config.any_match.conditions.ig
topology_steps.filter_config.any_match.conditions.fie

Field
topology_steps.filter_config.any_match.conditions.fie
topology_steps.filter_config.any_match.conditions.fie
topology_steps.filter_config.any_match.conditions.nu
topology_steps.filter_config.any_match.conditions.nu
topology_steps.filter_config.any_match.conditions.nu
topology_steps.filter_config.any_match.conditions.nu
topology_steps.filter_config.any_match.conditions.nu

Field
topology_steps.filter_config.any_match.conditions.da
topology_steps.filter_config.any_match.conditions.da
topology_steps.filter_config.any_match.conditions.da

Lookup Configuration

Field	De
topology_steps.lookup_config	Hol cor for ste
topology_steps.lookup_config.type	The tha the use the cac
topology_steps.lookup_config.keys	A li fiel rec

Field	De
	sav per sto que cac
topology_steps.lookup_config.additional_keys	A li ren key gro rec the loo

Transform Only Configuration

Field	Descriptio
topology_steps.minimize_tombstones	Allows the transformer to optimize tombstones by storing a additional row in the persistence store table that is checked after the mapping. If the last produce wa a delete, and the current mapper result is als a delete, transformer will not

Field	Descriptio
	produce to the canonical topic. Usefu for topologies where records frequently cause canonical deletes. Makes transform stateful
topology_steps.transform_config	Holds configuratic for a transform step
topology_steps.transform_config.type	The type that maps t the topic to be transformec This type can be filtered

Pass Through Configuration

Field	Descri
topology_steps.minimize_tombstones	Allows transfor to optim tombsto by stori additior

Field	Descri
	row in t persiste store ta that is checke after th mappin the last produce a delete and the current mapper result is a delete transfor will not produce the canonic topic. U for topolog where records frequen cause canonic deletes Makes through stateful
topology_steps.pass_through_config	Holds configu for a pa through
topology_steps.pass_through_config.type	The typ that ma the topi which

Docs

Entity Builder
About
About
Getting Started
Cost
Tour
Architecture
Topology
Topology Configuration
Topology Modification
Optimization Guide
Containers
Database
Test Harness
Template Mapper
Transformer
Deployments
Help
Terms of Service
Beam
Getting Started
Deployments
Beam Operations
Help

Field	Descri
	messag are bei passed type ca filtered

Fan Out Configuration

Field
topology_steps.fan_out_config
topology_steps.fan_out_config.type
topology_steps.fan_out_config.keys
topology_steps.fan_out_config.array_path
topology_steps.fan_out_config.row_type
topology_steps.fan_out_config.fan_out_keys

Field
topology_steps.fan_out_config.fan_out_keys.path
topology_steps.fan_out_config.fan_out_keys.output_

Consume Priority environment variables

There are several environment variables that can be used to tune consume priority

Env	De
CONSUME_PRIORITY_TIMEOUT_MS	The of mil tha pas witl hig pric me bei bef low pric me will

Env	De
CONSUME_PRIORITY_LOWEST_PRIORITY	Def low cor pric dep Hig pric alw hig nur hav pric is t pric low joir and pric pub que giv be cor tier cor for
CONSUME_PRIORITY_MIN_WAIT_MS	The mir nur mil tha pas a m fro par be App all exc hig

Entity Builder
About
About
Getting Started
Cost
Tour
Architecture
Topology
Topology Configuration
Topology Modification
Optimization Guide
Containers
Database
Test Harness
Template Mapper
Transformer
Deployments
Help
Terms of Service
Beam
Getting Started
Deployments
Beam Operations
Help

Env	De
	pric (pri A v zer hav mir del
CONSUME_PRIORITY_MAX_WAIT_MS	The ma nur mil the dep will witl rea me fro par App all exc hig pric (pri A v zer hav ma del

Logical Decimal Join Formatting

> **WARNING**
>
> Because this changes the format of the join key, changing this setting requires a full reload.

By default, logical decimal avro fields are formatted in the rational format provided by the goavro library that

Entity Builder uses. For instance, the integer 3 is formatted to staring as 3/1 and the decimal 1.5 is formatted as 3/2. This does not work very well when joining to another field type. You can change this formatting to be a more standard numeric format by providing an env `FORMAT_BYTES_DECIMAL_FOR_JOIN` set to true. When enabled, the string format used to join will be based on the scale property of the bytes decimal field. The scale property determines the number of decimal places to include in the string.

Examples

Rational Format	scale	Joined format
3/1	0	3
3/1	1	3.0
3/1	2	3.00
3/1	3	3.000

Env	Descri
FORMAT_BYTES_DECIMAL_FOR_JOIN	Formats decimal into stri represe of the n value w joining

This feature can be turned on in the test harness by providing the --decimal-join=true flag when executing test.sh or by adding the above env to eb_env

Topics

`topics.yaml` defines the relationship between types in `join.yaml` and Kafka topics. This allows for easier deployments to multiple data centers.

`join.yaml` can remain the same and only the topics must change to match the environment. Users need to specify all raw input topics and all (non `join_only`) shared-key steps. The type must match what is in `join.yaml` exactly.

Example:

```
output_topic: cdb.prod-eligibility-ctc.prod.
ebaas_prefix: EBAAS.cdb
topic_configs:
  - type: cnsm_cob_primacy
    topic_name: DE_STREAMS.ctc.cdc.cdb.cnsm_
  - type: cnsm_mdcr_primacy
    topic_name: DE_STREAMS.ctc.cdc.cdb.cnsm_
  - type: l_cnsm_srch
    topic_name: DE_STREAMS.ctc.cdc.cdb.l_cns
  - type: coordination_of_benefits
    topic_name: cdb.prod-eligibility-ctc.pro
```

In this example, the topology consumes off of raw input topics

- DE_STREAMS.ctc.cdc.cdb.cnsm_cob_primacy.v2
- DE_STREAMS.ctc.cdc.cdb.cnsm_mdcr_primacy.v2
- DE_STREAMS.ctc.cdc.cdb.l_cnsm_srch.v2

And produces to

- cdb.prod-eligibility-ctc.prod.canonical.eligibility.coordination-of-benefits.v3

Topic Priority

Individual topics can be given a priority between 0 (highest priority) and CONSUME_PRIORITY_LOWEST_PRIORITY. A topic with a priority set to a value other than 0 will wait for a configured amount of time without processing messages with a higher priority before a message is read from it (Consume Priority configuration. Steps consuming messages from topics with lowest priority

will produce to low priority join queue and low priority publish queue.

Example:

```
output_topic: cdb.prod-eligibility-ctc.prod.
ebaas_prefix: EBAAS.cdb
topic_configs:
  - type: cnsm_cob_primacy
    topic_name: DE_STREAMS.ctc.cdc.cdb.cnsm_
  - type: cnsm_mdcr_primacy
    topic_name: DE_STREAMS.ctc.cdc.cdb.cnsm_
    priority: 1
  - type: l_cnsm_srch
    topic_name: DE_STREAMS.ctc.cdc.cdb.l_cns
    priority: 2
  - type: coordination_of_benefits
    topic_name: cdb.prod-eligibility-ctc.pro
```

Topic configs

Field	Description	
output_topic	The output topic for this topology	required
ebaas_prefix	Prefix used for internal topics	required
topic_configs	A list of type to topic mappings	required, at l
topic_configs.type	The type for this topic. Used in `join.yaml`	required
topic_configs.topic	The topic that corresponds to this type	required

Field	Description	
topic_configs.priority	The priority of this topic.	min=0, max=CONSU