FLIP-107: Handling of metadata in SQL connectors

Discussion thread
Vote thread

JIRA

Release 1.12

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

- Motivation
- Examples
  - Kafka: Read, transform, and write back with key and value. All fields of the key are present in the value as well.
  - Kafka: Read metadata from Kafka's ConsumerRecord
  - Kinesis: Read other metadata from Kinesis
  - Kafka + Canal JSON Format: Both connector and format expose metadata
  - Kafka: Write metadata into Kafka's ProducerRecord
- Implementation Details
  - Syntax and Semantics
    - Implications to other components
  - Schema
  - LIKE clause
  - Handling of Data Types
  - Reading metadata via DecodingFormat
  - Writing metadata via DynamicTableSink
  - Writing metadata via EncodingFormat
  - Metadata for existing connectors and formats
    - Kafka
    - Debezium
  - Reading and writing from key and value
- Rejected alternatives
  - Option 1
  - Option 2
  - Option 3
  - Option 4
- Future Work
  - Kafka: Push down offset filter
  - Kinesis: Specify custom partitioning
  - Kafka: Manually specify target partition, overwrite partitioning by key.
- Related alternatives:
  - Writing to a specific partition/partitioning of source
  - Relation to FLIP-63

Motivation

Besides the main payload, the majority of connectors (and also many formats) expose additional information that should be readable and (depending on the use case) also writable as metadata.

It can simply be read-only metadata such as a Kafka read-offset or ingestion time. But can also add or remove header information (e.g. a message hash, or record version) to every Kafka ProducerRecord. Additionally, users might want to read and write only parts of the record that contain data but additionally serve different purposes (e.g. compaction by key).

We should make it possible to read and write data from all of those locations.

Kafka is the source with the most intricacies as it allows storing data in multiple different places of the records. Each of those places is/can be serialized differently. Moreover some of them might serve different purposes:

- all of them can be just a data container,
- key for partitioning (hash on the key),
- key for compacting (if topic is compacted records with same key within a partition are merged),
- timestamp for log retention
- header for metadata

Because of the reasons above Kafka will be used for a majority of the examples.

Also formats should be able to expose metadata, FLIP-132 is just one example where the Debezium format might expose a "db_operation_time" that is not part of the schema itself.

Other use cases could be exposing Avro version or Avro schema as meta information per record.
Examples

Kafka: Read, transform, and write back with key and value. All fields of the key are present in the value as well.

CREATE TABLE Kafka_table {
  id BIGINT,
  name STRING,
  col1 STRING,
  col2 STRING
} WITH {
  'connector' = 'kafka',
  'topic' = 'test-topic',
  'key.fields' = 'id, name',
  'key.format' = 'csv',
  'value.format' = 'avro'
}

An insert statement

```
INSERT INTO Kafka_table VALUES {
  (1, "ABC", "col1", "col2")
}
```

will result in a ProducerRecord as follows:

```
new ProducerRecord<{
  "test-topic",
  null,
  CSV.serialize(id, name),
  AVRO.serialize(id, name, col1, col2)
}
```

A slightly different use case is to manually control where to store the columns data:

CREATE TABLE Kafka_table {
  id BIGINT,
  name STRING,
  col1 STRING,
  col2 STRING
} WITH {
  'connector' = 'kafka',
  'topic' = 'test-topic',
  'key.format' = 'csv',
  'key.fields' = 'id, name',
  'value.format' = 'avro',
  'value.fields-include' = 'EXCEPT_KEY'
}

An insert statement

```
INSERT INTO Kafka_table VALUES {
  (1, "ABC", "col1", "col2")
}
```

will result in a ProducerRecord as follows:
```java
new ProducerRecord<>
("test-topic",
 null,
 CSV.serialize(id, name),
 AVRO.serialize(col1, col2)
)
```

**Kafka: Read metadata from Kafka's ConsumerRecord**

```sql
CREATE TABLE kafka_table {
    id BIGINT,
    name STRING,
    offset INT METADATA, -- access Kafka 'offset' metadata
    headers MAP<STRING, BYTES> METADATA  -- access Kafka 'headers' metadata
} WITH {
    'connector' = 'kafka',
    'topic' = 'test-topic',
    'format' = 'avro'
}
```

**Kinesis: Read other metadata from Kinesis**

Kinesis represents partition keys as strings:

```sql
CREATE TABLE kinesis_table {
    id BIGINT,
    name STRING,
    partitionId STRING METADATA FROM 'partition'  -- use different column name for metadata 'partition'
} WITH {
    'connector' = 'kinesis',
    'value.format' = 'avro'
}
```

```sql
SELECT * FROM kinesis_table;
```

```sql
-- Partition is a persisted column, therefore it can be written to:
INSERT INTO kinesis_table VALUES (1, "ABC", "shard-0000")
```

**Kafka + Canal JSON Format: Both connector and format expose metadata**

```sql
CREATE TABLE kafka_table {
    id BIGINT,
    name STRING,
    offset INT METADATA, -- from Kafka
    database STRING METADATA FROM 'value.canal-json.database'  -- from Canal
} WITH {
    'connector' = 'kafka',
    'topic' = 'test-topic',
    'format' = 'canal-json'
}
```

Using the following format:

```json
{
    "data": [ {
        "id": "102",
        "name": "car battery"
    } ],
    "database": "inventory",
    "table": "products",
    "es": 1589374013000,
    "ts": 1589374013680,
    "type": "DELETE"
}
```
Kafka: Write metadata into Kafka's ProducerRecord

CREATE TABLE kafka_table {
  id BIGINT,
  name STRING,
  timestamp BIGINT METADATA,
  offset INT METADATA VIRTUAL  -- offset cannot be written out, it is ignored in the query-sink schema
} WITH {
  'connector' = 'kafka',
  'topic' = 'test-topic',
  'format' = 'avro'
}

An insert statement could look like:

```
INSERT INTO kafka_table VALUES {
  (1, "ABC", 1599133672)
}
```

Or with no persisted metadata:

```
INSERT INTO kafka_table (id, name) VALUES {
  (1, "ABC")
}
```

Implementation Details

Syntax and Semantics

Let's assume the following example:

```sql
CREATE TABLE t (i INT, s STRING, timestamp TIMESTAMP(3) WITH LOCAL TIME ZONE METADATA, d DOUBLE)
```

The `timestamp` column is declared as a `METADATA` column. By default, the column name is used to map to a corresponding metadata key ("timestamp" in this case).

The data type of the column is used to perform an explicit cast of the original metadata data type. For example, a Kafka metadata timestamp is defined as `BIGINT`. A user can declare the column as `TIMESTAMP(3)` though and thus force an explicit cast from sources (BIGINT to TIMESTAMP) and to sinks (TIMESTAMP to BIGINT).

```sql
CREATE TABLE t (i INT, s STRING, myOffset INT METADATA FROM 'offset', d DOUBLE)
```

In order to use a different column name, it is possible to use `FROM` to reference the metadata key ("offset" in this case).

```sql
CREATE TABLE t (i INT, s STRING, offset INT METADATA VIRTUAL, d DOUBLE)
```

`offset INT METADATA` would be a valid read-only column for Kafka and can be extracted by the planner. However, since declared tables can be used for sources and sinks, we need to be able to exclude read-only metadata from writing.

The `VIRTUAL` keyword excludes the column from the query-to-sink schema which means that a query cannot write to a this metadata column.

Implications to other components

Because we are extending the DDL, this has implication to other components.

Schema

We propose to extend the `Schema` class from FLIP-129 by:
LIKE clause

Currently, the LIKE clause offers the following table features:

- CONSTRAINTS - constraints such as primary and unique keys
- GENERATED - computed columns
- OPTIONS - connector options that describe connector and format properties
- PARTITIONS - partition of the tables
- WATERMARKS - watermark declarations

We propose to extend the LIKE clause to add ‘METADATA’ as another table feature.

Metadata columns are not generated. They represent data that is present in the external system. Thus, it should not be categorized into the GENERATED feature.

Furthermore, METADATA is connector dependent. It is safer to have a fine-grained control over the table feature. The user should control whether metadata can be inherited or not.

For example, this is important when switching from filesystem to Kafka or vice versa.

METADATA supports OVERWRITING by column name.

The following example shows how to overwrite a metadata column with another metadata column:

```sql
CREATE TABLE t (i INT, s STRING, timestamp TIMESTAMP(3) WITH LOCAL TIME ZONE METADATA FROM 'timestamp', d DOUBLE);
CREATE TABLE t (timestamp TIMESTAMP(3) WITH LOCAL TIME ZONE METADATA FROM 'other-system.timestamp')
WITH (...
LIKE t (
    INCLUDING ALL
    OVERWRITING OPTIONS
    OVERWRITING METADATA
)
```

Reading metadata via DynamicTableSource

Let's assume the following example:

```sql
CREATE TABLE t (i INT, s STRING, offset INT METADATA VIRTUAL, d DOUBLE)
```

We suggest the following interfaces for integrating reading metadata into FLIP-95 interfaces:

```java
interface SupportsReadingMetadata {
    Map<String, DataType> listReadableMetadata();
    void applyReadableMetadata(List<String> metadataKeys, DataType outputDataType);
}
```
`listReadableMetadata()` allows for validation and provides information about expected data types for the output row.

`applyReadableMetadata(List<String> metadataKeys, DataType outputDataType)` provides the list of required metadata by the DDL and query. It incorporates information from projection push down and only requests metadata that is necessary for executing the given query.

A DynamicTableSource must append the requested metadata columns (such as ‘offset INT’ in the example) to the source’s original output row in the order of the given list. This also works for a row that is passed into SupportsComputedColumnPushDown.

In general, the method `applyReadableMetadata(...)` is very similar to `SupportsComputedColumnPushDown#applyComputedColumn`. The method provides an updated output data type to create TypeInformation similar to SupportsComputedColumnPushDown (i.e. projection pushdown is considered and applied before). This is the output type that also the planner expects.

The order of execution in a source is:

projection push down metadata appending computed column execution (in source or follow-up operator)

All other row data types do not change. This means:

| TableSchema.toPhysicalRowDataType => pure physical data, excluding computed columns and metadata columns |
| TableSchema.toRowDataType => full schema with all kinds of computed columns |

**Handling of Data Types**

For making the use of metadata easier and avoid nested casting such as:

```java
rowtime BIGINT METADATA FROM 'timestamp'
rowtimeCasted AS CAST(rowtime AS TIMESTAMP(3) WITH LOCAL TIME ZONE)
```

We allow explicit casting to a target data type:

```java
rowtime TIMESTAMP(3) WITH LOCAL TIME ZONE METADATA FROM 'timestamp'
```

A connector still produces and consumes the data type returned by `listMetadata()`. The planner will insert necessary explicit casts.

In any case, the user must provide a data type such that the metadata column receives a valid data type when constructing the table schema.

**Reading metadata via DecodingFormat**

We allow formats to expose metadata as well. By default, a format does not provide any metadata. The information is not used unless a source supports it.

```java
interface DecodingFormat {
    default Map<String, DataType> listReadableMetadata() {
        return Collections.emptyMap();
    }

    default void applyReadableMetadata(List<String> metadataKeys) {
        throw new UnsupportedOperationException();
    }
}
```

Similar to the source, `listReadableMetadata()` allows for validation and provides information about expected data types for the format's output row.

`applyReadableMetadata(...)` enables the modification of the format's produced row type to:

PHYSICAL COLUMNS + FORMAT METADATA COLUMNS

The source must call the methods above and forward them if it implements `SupportsReadingMetadata`.

Both source and format can offer metadata columns. We define the following behavior for a consistent interaction:

- The source must add metadata columns in the order as passed by the planner.
- The passed `List<String>` `metadataKeys` will be ordered according to the iteration order of `listReadableMetadata()` (so it might be beneficial that implementations return a LinkedHashSet, but don’t have to).
- Format metadata keys have higher precedence than source metadata keys in case of duplicate key names (this case should happen very rarely).

In the end an output row that leaves the source looks like:

PHYSICAL COLUMNS + FORMAT METADATA COLUMNS + SOURCE METADATA COLUMNS

For Kafka with key and value formats, the row would look like:
Writing metadata via DynamicTableSink

For being able to write out metadata from a query into a sink, we introduce the concept of virtual columns to exclude non-writable columns.

Let's assume the following example:

```
CREATE TABLE t (i INT, s STRING, timestamp BIGINT METADATA, offset INT METADATA VIRTUAL, d DOUBLE)
```

The `timestamp` column can be used for both symmetrically reading and writing metadata.

Non-virtual (i.e. persisted) metadata columns are part of the query-to-sink schema which means that they need to be specified in an INSERT INTO statement. In order to make those columns optional, we suggest to introduce the syntax:

```
INSERT INTO t (i, s, d) SELECT ...
```

This marks persisted columns as NULL.

Persisted metadata columns are part of a physical row that enters a sink (either passed by the query or NULL) it is never recomputed.

We suggest the following interfaces for integrating writing metadata into FLIP-95 interfaces:

```java
interface SupportsWritingMetadata {
    Map<String, DataType> listWritableMetadata();
    void applyWritableMetadata(List<String> metadataKeys);
}
```

The interface is similar to `SupportsReadingMetadata`. `listWritableMetadata()` is used for validation and metadata information.

The `applyWritableMetadata(...)” provides the final list of keys that is used in the DDL. The method also enables writing metadata in formats.

By definition, the planner will put persisted columns to the end of the physical row before passing it to a sink this makes it easier to split metadata and actual data. The sink can use `TableSchema.toPhysicalRowDataType` to extract the pure data as before.

For the example above, a row that enters a sink looks like:

```
ROW<i INT, s STRING, d DOUBLE, timestamp BIGINT>  -- 'timestamp' is moved to the end and has the expected metadata data type, 'offset' is not present
```

Writing metadata via EncodingFormat

Similar to DecodingFormat, we add similar default methods to EncodingFormat. That can be used by the DynamicTableSink.

```java
interface EncodingFormat {
    default Map<String, DataType> listWritableMetadata() { return Collections.emptyMap(); }
    default void applyWritableMetadata(List<String> metadataKeys) { throw new UnsupportedOperationException(); }
}
```

By calling `applyWritableMetadata(...)”, the format enables the modification of the format's consumed row type. The row that goes into the format during runtime will be:

```
PHYSICAL COLUMNS + FORMAT METADATA COLUMNS
```

Same rules as mentioned above for handling both sink metadata and format metadata apply.

In the end an input row into the sink looks like:

```
PHYSICAL COLUMNS + FORMAT METADATA COLUMNS + SINK METADATA COLUMNS
```
Metadata for existing connectors and formats

For completeness, we list the first metadata for existing Flink connectors and formats. The naming follows FLIP-122 which means:

- connector metadata has no prefix
- formats will have a prefix using the factory identifier
- key/value formats are always prefixed with 'key.' and 'value.' to avoid collisions

**Kafka**

<table>
<thead>
<tr>
<th>Key</th>
<th>Data type</th>
<th>Read/Write</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>topic</td>
<td>STRING</td>
<td>r</td>
<td>We don’t allow writing to different topics for now. Maybe we will allow that in the future via a property.</td>
</tr>
<tr>
<td>partition</td>
<td>INT</td>
<td>r</td>
<td>We don’t allow writing to different partitions for now. Maybe we will allow that in the future via a property.</td>
</tr>
<tr>
<td>headers</td>
<td>MAP&lt;STRING, BYTES&gt;</td>
<td>r/w</td>
<td></td>
</tr>
<tr>
<td>leader-epoch</td>
<td>INT</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td>BIGINT</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>timestamp</td>
<td>TIMESTAMP(3) WITH LOCAL TIME ZONE</td>
<td>r/w</td>
<td>Directly forward the underlying type.</td>
</tr>
<tr>
<td>timestamp-type</td>
<td>STRING</td>
<td>r</td>
<td>['NoTimestampType', 'CreateTime', 'LogAppendTime']</td>
</tr>
</tbody>
</table>

**Debezium**

<table>
<thead>
<tr>
<th>Key</th>
<th>Data type</th>
<th>Read/Write</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>debezium-json.schema</td>
<td>STRING</td>
<td>r</td>
<td>Pure JSON string, can be handled with Flink’s built-in JSON SQL functions</td>
</tr>
<tr>
<td>debezium-json.ingestion-timestamp</td>
<td>TIMESTAMP(3) WITH LOCAL TIME ZONE</td>
<td>r</td>
<td>Directly forward the underlying type. Do not try to abstract it into a TIMESTAMP.</td>
</tr>
<tr>
<td>debezium-json.source.timestamp</td>
<td>TIMESTAMP(3) WITH LOCAL TIME ZONE</td>
<td>r</td>
<td>Shortcut for debezium-json.source[ts_ms].</td>
</tr>
<tr>
<td>debezium-json.source.database</td>
<td>STRING</td>
<td>r</td>
<td>Unified shortcut for ‘debezium-json.source[db]’ across SQL vendors.</td>
</tr>
<tr>
<td>debezium-json.source.schema</td>
<td>STRING</td>
<td>r</td>
<td>Unified shortcut for ‘debezium-json.source[schema]’ across SQL vendors.</td>
</tr>
<tr>
<td>debezium-json.source.table</td>
<td>STRING</td>
<td>r</td>
<td>Unified shortcut for ‘debezium-json.source[table/collection]’ across SQL vendors.</td>
</tr>
<tr>
<td>debezium-json.source.properties</td>
<td>MAP&lt;STRING, STRING&gt;</td>
<td>r</td>
<td>All properties that Debezium provides under ‘source’. Frequently used properties have a dedicated key above. It seems that this is always a property list. So we can make it available as a map for easier access.</td>
</tr>
</tbody>
</table>

Reading and writing from key and value

We suggest defining the places where the particular columns come from in the source options sections.

We suggest introducing connector specific properties that allow specifying which fields should end up in which parts of the record. The reason why we are not suggesting having a key-value-metadata format is that there are most often differences across records of different systems, e.g.

- Kinesis, Pravega do not have a key where users can store data

As described previously Kafka has the most complex record structure. It is also the most important source for streaming use cases therefore we will discuss how to support its record below.

We suggest introducing additional properties for controlling different parts of ConsumerRecord:
- `key.fields`, `key.format.key.(<format-identifier>).(<properties-required-by-format>)` - this controls which fields should end up in Kafka's key and what should be the serialization format
- `value.fields-include`, this controls which fields should end up in the value as well, possible values
  - ALL (all fields of the schema, even if they are part of e.g. the key)
  - EXCEPT_KEY (all fields of the schema - fields of the key)
- `value.format.value.(<format-identifier>).(<properties-required-by-format>)` - The "value" prefix is optional
- (optional) `fields.verify-integrity` - controls if we should perform an equality check if a field is contained in different parts of consumer record (either in key or value)

Note: `key.format`, `value.format`, `key.fields` etc. are only available in Kafka. They are Kafka connector specific properties.

A full example looks like:

```sql
CREATE TABLE kafka_table (
id STRING,
timestamp TIMESTAMP,
coll ..., col2 ...
....
) WITH {
    'connector' = 'kafka',
    ...'key.fields' = 'id, coll'
    'key.format' = 'csv'

    -- optional: controls which fields of the schema should end up in the value (ALL by default) (options: ALL, EXCEPT_KEY)
    'value.fields-include' = 'ALL'
    'value.format' = 'avro'

    --optional: (false by default) if true and a field originates from more than a single location (e.g. both in key and value), a check will be performed both values are equal
    'fields.verify-integrity' = 'false'
}
```

Rejected alternatives
Copied from the long ML discussion.

Option 1

Declare everything via properties:

```sql
CREATE TABLE kafka_table {
    id BIGINT,
    name STRING,
coll STRING,
col2 STRING,
ts TIMESTAMP(3) WITH LOCAL TIME ZONE, -- ts is a normal field, so can be read and written.
offset AS SYSTEM_METADATA("offset")
} WITH {
    'connector' = 'kafka',
    'topic' = 'test-topic',
    'key.fields' = 'id, name',
    'key.format' = 'csv',
    'value.format' = 'avro',
    'timestamp.field' = 'ts' -- define the mapping of Kafka timestamp
}
```

Pros:
- "timestamp", "headers" are something like "key" and "value" that are stored with the real data. So why not define the "timestamp" in the same way with "key" by using a "timestamp.field" connector option?

Cons:
- "key" and "value" in the properties are a special case because they need to configure a format.
- We have way more metadata fields like headers, epoch-leader, etc. Having a property for all of this metadata would mess up the WITH section entirely. Furthermore, we also want to deal with metadata from the formats. Solving this through properties as well would further complicate the property design.

Option 2
Use computed columns:

```sql
CREATE TABLE kafka_table (
    id BIGINT,
    name STRING,
    timestamp AS CAST(SYSTEM_METADATA("timestamp") AS INT) PERSISTED,
    headers AS CAST(SYSTEM_METADATA("headers") AS MAP<STRING, BYTES>) PERSISTED
) WITH (...
)"
```

Pros:
- Allows to have full flexibility to compute the final column and avoid helper columns:
  ```sql
  timestamp AS adjustTimestamp(CAST(SYSTEM_METADATA("ts") AS TIMESTAMP(3)))
  ```

Cons:
- Mixes concepts of metadata and computed column.
- Would need the concept of PERSISTED computed columns but not with 100% the same semantics as other vendors
- Complicated syntax:
  a) CREATE TABLE t (a AS CAST(SYSTEM_METADATA("offset") AS INT))
     pro: readable, complex arithmetic possible, more SQL compliant, SQL Server compliant
     con: long
  b) CREATE TABLE t (a INT AS SYSTEM_METADATA("offset"))
     pro: shorter, not SQL nor SQL Server compliant
     con: requires parser changes, no complex arithmetic like `computeSomeThing(SYSTEM_METADATA("offset"))` possible
  c) CREATE TABLE t (a AS SYSTEM_METADATA("offset", INT))
     pro: shorter, very readable, complex arithmetic possible
     con: non SQL expression, requires parser changes

Option 3

Marking key columns in the schema section:

```sql
CREATE TABLE kafka_table (id STRING KEY,
    col1 ... KEY:HEADER("key"),
    col2 ...
) WITH (...

'connector.type' = 'kafka',
'format.type' = 'kafka-format'
'format.key.type' = 'csv'
'format.value.type' = 'avro'
)
```

There is a couple of issues with this approach:
- mixes logical schema definition with physical representation of format and/or source (kafka the only source that has a meaningful key, all other sources like pulsar/kinesis/pravega use key only as a partitioning hash and support string based keys only). Moreover it makes it harder to reason about when implementing CREATE TABLE ... AS SELECT ...

For CREATE TABLE ... AS SELECT ... there is no place to put the modifiers, as the schema is derived from the query.
- cumbersome or impossible to handle columns that are stored in multiple places (both in key and value)

Option 4

Specific per property functions
CREATE TABLE kafka_table (id STRING, offset AS SOURCE_OFFSET(), partitionId AS SOURCE_PARTITION())

WITH ('connector.type' = 'kafka')

Con: Would add too many built-in functions.

Future Work

Some topics that are not part of this FLIP anymore.

Kafka: Push down offset filter

CREATE TABLE kafka_table (id BIGINT, offset BIGINT AS SYSTEM_METADATA("offset"))

WITH ('connector.type' = 'kafka', 'value.format.type' = 'avro')

SELECT * FROM kafka_table WHERE offset > 123456;

Kinesis: Specify custom partitioning

CREATE TABLE kinesis_table (id BIGINT, name STRING)

PARTITIONED BY HASH(id)

WITH ('connector.type' = 'kinesis', 'value.format.type' = 'avro')

INSERT INTO kinesis_table VALUES(
  (1, 'ABC', 1),
  (1, 'ABC', 3)
)

Both records will end up in the same partition.

Kafka: Manually specify target partition, overwrite partitioning by key.

CREATE TABLE kinesis_table (id BIGINT, name STRING, partitionId BIGINT)

PARTITIONED BY HASH(partitionId)

WITH (connectorg.type' = 'kafka',
  'key.fields' = 'id, name',
  'key.format.type' = 'csv',
  'value.format.type' = 'avro')

INSERT INTO kafka_table VALUES(
  (1, 'ABC', 1),
  (1, 'ABC', 3)
)

The two records end up in different partitions even though the resulting Kafka’s keys will be equal for both records.
Rejected alternatives:

Writing to a specific partition/partitioning of source

A common requirement is to create a custom partitioning of the data. We should have a way to specify/compute target partition/shard for Kinesis/Pravega/Pulsar. In those cases it would be the only way to control partitioning.

In the case of Kafka we could use the key for data partitioning as described in the first section. However, it does not have to always correspond to the partition that a record originates from/is written to. The default partitioner uses the hash value of the key and the total number of partitions on a topic to determine the partition number. If you increase a partition number, then the default partitioner will return different numbers even if you provide the same key. Users might want to implement a custom Kafka’s partitioner to be able to rescale the topic.

Use PARTITIONED BY clause with different partitioning strategies (similar to ORACLE):

Partition by hash:

```sql
CREATE TABLE kafka_table (  
id STRING,  
name STRING,  
date: DATE,  
key: BINARY  
) WITH (  
'connector.type' = 'kafka'  
'key.fields' = 'key' --the key will not be used for partitioning  
) PARTITIONED BY HASH(id, name)
```

If necessary we can introduce more partitioning strategies E.g. explicit partitioning:

```
-- partitioning based on an existing column in data  
CREATE TABLE kafka_table (  
id BIGINT,  
name STRING,  
date: DATE,  
key: BINARY  
) WITH (  
'connector.type' = 'kafka'  
'key.fields' = 'key' --the key will not be used for partitioning  
) PARTITIONED BY EXPLICIT(id)

-- or on a computed column  
CREATE TABLE kafka_table (  
id BIGINT,  
name STRING,  
targetPartition BIGINT AS CAST(name AS BIGINT),  
date: DATE,  
key: BINARY  
) WITH (  
'connector.type' = 'kafka'  
'key.fields' = 'key' --the key will not be used for partitioning  
) PARTITIONED BY EXPLICIT(targetPartition)

-- Or with user defined functions in the partitioned by clause  
CREATE TABLE kafka_table (  
id BIGINT,  
name STRING,  
date: DATE,  
key: BINARY  
) WITH (  
'connector.type' = 'kafka'  
'key.fields' = 'key' --the key will not be used for partitioning  
) PARTITIONED BY EXPLICIT(USER_SCALAR_FUNCTION(name, date, SYSTEM_METADATA("numberOfPartitions")))
```
Relation to FLIP-63

Flip-63 introduced initial support for PARTITIONED BY clause to an extent that let us support Hive's partitioning. In majority of situations it is in line with the proposal above.

The only significant difference is that this proposal suggests adding partitioning strategies e.g. PARTITIONED BY HASH/EXPLICIT. Currently implemented strategy works as an EXPLICIT strategy that requires a list of any type as a partition identifier. We could make the EXPLICIT strategy default, which could let us support current PARTITION BY syntax.

The current implementation requires the column in PARTITIONED BY clause to be also defined in the schema part of the table. It is the same as in this proposal.

FLIP-63 discusses syntax as INSERT INTO ... PARTITION(...)  It does not interfere with this proposal. It just implies additional check that the row matches the partition requirements(columns in rows match the partition specification)

FLIP-63 discusses more in-depth filesystem specific partition pruning. This proposal leaves out that part to specific source implementations. Therefore it does not contradict with FLIP-63.