KSQL STRUCTURED KEYS

Up until this point KSQL has only supported VARCHAR / String keys. Enhancing KSQL to support more data types, (e.g. INT keys), and structured data, (e.g. JSON objects or Avro records, etc), within the key has been a much requested feature and one we've been kean to add.

While investigating what the new syntax would look like to allow us to define and manipulate keys we came up with two approaches that each look viable. Opinion within the team is split, often with people flipping back and forth, and so we'd like to ask the community for its opinion. Below is a side by side view of how different statements would look and behave. We are looking for feedback on which of the two customers prefer.

Before we get started let us first pin down some terminology, so that we're hopefully all on the same page:

We are looking to enhance KSQL to support key *schemas* that are more than just an unnamed VARCHAR. This will include the ability to have:

- A key with named fields, e.g. an Avro *record*, or a JSON *object*, or a Delimited key with multiple fields, or
- A key that is some raw type, e.g. an INT, BIGINT, DOUBLE, etc, as well as VARCHAR.
- A key that has an array as its top level entity, e.g. a JSON array of numbers.
- A key that has a map as its top level entity, e.g. an Avro map.

In this document, we will refer to the first of these as a *struct*, the second *primitive* keys and last two as *array* and *map* keys, respectively.

One of the design goals for this work is to allow KSQL to work with, and output, data of any schema.

The proposal is *not* looking to change the serialization *formats* that KSQL supports. This will remain JSON, Avro and Delimited for the time being, (though many may be happy to hear Google Protocol buffer support is coming). Nor is it looking to extend the primitive types KSQL supports in its schemas at the moment, e.g. we're not adding a BINARY or DECIMAL type as part of this work.

While backwards compatibility is almost always a key concern for us, in this instance we did not want to restrict the syntax we're designing for tomorrow by what we currently have today, for fear we may end up with some frankenstein's monster. So if you're already familiar with KSQL syntax, please try not to think too much about whether either of these is more or less backwards compatible. What we're much more interested in is which semantics and syntax *feels more natural*.

In the discussion below some details, e.g. the implicit ROWTIME column, have been ignored to keep things simple.

Importing data

When importing data into KSQL, via DDL statements, the schema of the imported stream or table must include the field or fields from the key. Below are some examples that cover how statements would look for importing data with different key schemas.

In each example the *value* fields v0 and v1 are included for completeness. In each comparison between the more logical and more physical syntax the underlying physical representation of data in the Kafka topic is the same.

More logical view	**More physical view**
The approach here is to have a single schema for the row, using a new **KEY** keyword to differentiate between fields that come from the message's key verses its value. For some formats, e.g. Avro, KSQL needs to know the exact schema of the key to be able to deserialize it correctly. For example, in the case of a DDL statement where the key contains only a single field, KSQL needs to know if that field is a raw primitive or within an Avro record. As this information is not part of the logical schema it will be provided within the WITH clause using the **KEY_FORMAT_TYPE** property, which can be set to either STRUCTURED or PLAIN. KEY_FORMAT_TYPE will only be required where this is ambiguous, i.e. for single field keys. The same KEY_FORMAT_TYPE property can be used in DML statements to control if a key with a single field is persisted within a structured key, (e.g. a JSON Object or Avro Record), or as a plain unnamed field. In the future, we will likely support a VALUE_FORMAT_TYPE to allow for primitive value types.	The approach here is to explicitly define the schemas of the key and the value via **ROWKEY** and **ROWVALUE** keywords. So the schema of a row is a combination of the ROWKEY and ROWVALUE schemas.

Import a stream with a structured key with two fields:
 	
CREATE STREAM FOO (**k0 INT KEY, k1 INT KEY**, v0 INT, v1 INT) WITH (FORMAT='JSON', ...);
	Example JSON key:
	{
	"k0": 1234,
	"k1": 6789
	}
	
	CREATE STREAM FOO (**ROWKEY STRUCT<k0 INT, k1 INT>**, ROWVALUE STRUCT<v0 INT, v1 INT>) WITH (FORMAT='JSON', ...);

Import a stream with a structured key containing a single field:
 	
CREATE STREAM FOO (**k0 INT KEY**, v0 INT, v1 INT) WITH (FORMAT='AVRO', **KEY_FORMAT_TYPE='STRUCTURED',** ...);
	Example JSON key:
	{
	"k0": 1234
	}
	
	CREATE STREAM FOO (**ROWKEY STRUCT<k0 INT>**, ROWVALUE STRUCT<v0 INT, v1 INT>) WITH (FORMAT='JSON', ...);

Import a stream with a primitive INT key: CREATE STREAM FOO (**k0 INT KEY,** v0 INT, v1 INT) WITH (FORMAT='JSON', **KEY_FORMAT_TYPE='PLAIN'**, ...); **NB:** the logical schema defined in this statement is the same as for the one above. However, here the WITH clause reflects the different key type.	Example JSON key: 1234 CREATE STREAM FOO (**ROWKEY INT**, ROWVALUE STRUCT<v0 INT, v1 INT>) WITH (FORMAT='JSON', ...); **NB:** notice that unlike the more logical approach to the left, here the statements dealing with a single numeric value and a struct containing a single numeric field are different. The schema defined in the statement more closely matches the physical structure of the data in Kafka.

Import a stream with that has a structured key containing a single array of ints:
 	
CREATE STREAM FOO (**k0 ARRAY<INT> KEY**, v0 INT, v1 INT) WITH (FORMAT='JSON', **KEY_FORMAT_TYPE='STRUCTURED'**, ...);
	Example JSON key:
	{
	  "k0": [123, 456]
	}
	
	CREATE STREAM FOO (**ROWKEY STRUCT<k0 ARRAY<INT>>**, ROWVALUE STRUCT<v0 INT, v1 INT>) WITH (FORMAT='JSON', ...);

Import a stream with that has an array of ints key:
 	
CREATE STREAM FOO (**k0 ARRAY<INT> KEY**, v0 INT, v1 INT) WITH (FORMAT='JSON', **KEY_FORMAT_TYPE='PLAIN'**, ...);
 	
NB: as above, the name 'k0' is not part of the key schema. It is just an arbitrary name assigned to the field in the statement.
	Example JSON key:
	[
	   123,
	    456
	]
	
	CREATE STREAM FOO (**ROWKEY ARRAY<INT>**, ROWVALUE STRUCT<v0 INT, v1 INT>) WITH (FORMAT='JSON', ...);

Query semantics

In KSQL at the moment a query, e.g. "select f0, f1, f2 from bar;", the projection part, i.e. "f0, f1, f2", defines the fields that will be part of the output *value's* schema. IF there is neither a GROUP BY or PARTITION BY part to the query then the *key* schema will be passed through unchanged. If a GROUP BY or PARTITION BY clause is included, then it controls the fields that will be part of the output *key's* schema. As the language semantics of GROUP BY and PARTITION BY are very similar, the examples below only cover PARTITION BY to avoid a lot of duplication.

For the context of this document we are not proposing changing these query semantics, only the syntax / concept between how we access the data in the key and value of the underlying Kafka message.

More logical view	**More physical view**
The row schema contains all the fields of the messages key and value, namespaced by **ROWKEY** and **ROWVALUE**. Where a field name is unambiguous, i.e. it exists only in either the key or the value, then the namespace can be omitted from the query statement.	The row schema is explicitly made up from the **ROWKEY** and **ROWVALUE** schemas. Where a field name is unambiguous, i.e. it exists only in either the key or the value, then the **ROWKEY / ROWVALUE** prefix can be omitted from the query statement.
Query on a stream with a structured key *Given:* CREATE STREAM FOO (**k0 INT KEY, k1 INT KEY**, v0 INT, v1 INT) WITH (...); *Then a query might be:* SELECT k0, v0 FROM FOO; *Which would the same as writing:* SELECT ROWKEY.k0, ROWVALUE.v0 FROM FOO; *Which would the same as writing:* SELECT FOO.ROWKEY.k0, FOO.ROWVALUE.v0 FROM FOO; *All of the above results in a row schema of:* (k0 INT KEY, k1 INT KEY, k0 INT, v0 INT) **NB:** here a fully qualified name is in the form <table>.<namespace>.<field name>, where <table> and <namespace> are optional where there is no ambiguity.	*Example JSON key:* *{* *“k0”: 1234,* *“k1”: 6789* *}* *Given:* CREATE STREAM FOO (**ROWKEY STRUCT<k0 INT, k1 INT>**, ROWVALUE STRUCT<v0 INT, v1 INT>) WITH (...); *Then a query might be:* SELECT k0, v0 FROM FOO; *Which would the same as writing:* SELECT ROWKEY->k0, ROWVALUE->v0 FROM FOO; *Which would the same as writing:* SELECT FOO.ROWKEY->k0, FOO.ROWVALUE->v0 FROM FOO; *All of the above results in a row schema of:* (ROWKEY<k0 INT, k1 INT>, ROWVALUE<k0 INT, v0 INT>) **NB:** here a fully qualified name is in the form <table>.<key/value struct>-><field name>, where <table> and <key/value struct> are optional where there is no ambiguity. Unlike the logical view on the left, the statement uses the ‘->’ notation because the ROWKEY and ROWVALUE are both STRUCTs.
Query on a stream with a primitive INT key:	Example JSON key: 1234

Given: CREATE STREAM FOO (**k0 INT KEY,** v0 INT, v1 INT) WITH (...); *Then a query might be:* SELECT k0, v0 FROM FOO; *Resulting schema:* (k0 INT KEY, k0 INT, v0 INT) **NB:** here, even though the key's schema contains no field name 'k0', a query can use the 'k0' field name of the primitive key.	*Given:* CREATE STREAM FOO (**ROWKEY INT**, ROWVALUE STRUCT<v0 INT, v1 INT>) WITH (...); *Then a query might be:* SELECT ROWKEY as k0, v0 FROM FOO; *Resulting schema:* (ROWKEY INT, ROWVALUE STRUCT<k0 INT, v0 INT>) **NB:** here, as the key's schema is a primitive type, queries can access the key using the **ROWKEY** field. Though, as **ROWKEY** is a reserved word we alias the field in the select to ensure the value schema of the output doesn't have a field name that is a reserved word.
Query on a stream with a structured key with ambiguous field names: *Given:* CREATE STREAM FOO (**f0** INT KEY**,** **f0** INT) WITH (...); *Then a query accessing f0 needs to be explicit about which f0:* SELECT ROWVALUE.f0 FROM FOO;	 *Given:* _CREATE STREAM FOO (ROWKEY STRUCT<**f0** INT>, ROWVALUE STRUCT<**f0** INT>) WITH (...);_ *Then a query accessing f0 needs to be explicit about which f0:* SELECT ROWVALUE->f0 FROM FOO;
Query that partitions by a single field in struct: *The following will partition data by a new structured key:* CREATE STREAM FOO WITH(**KEY_FORMAT_TYPE='STRUCTURED'**) AS SELECT * FROM BAR **PARTITION BY f0**;	 *The following will partition data by a new structured key:* CREATE STREAM FOO AS SELECT * FROM BAR **PARTITION BY STRUCT<f0>**;
Query that partitions by a primitive: *The following will partition the data by a new primitive key:* CREATE STREAM FOO WITH(**KEY_FORMAT_TYPE='PLAIN'**) AS SELECT * FROM BAR **PARTITION BY f0**;	 *The following will partition the data by a new primitive key:* CREATE STREAM FOO AS SELECT * FROM BAR **PARTITION BY f0**;
Query that partitions multiple fields: *The following will partition the data by a new structured key with two fields:*	 *The following will partition the data by a new structured key with two fields:*

CREATE STREAM FOO AS SELECT * FROM BAR **PARTITION BY f0, f1**;
 	
Example resulting JSON key:
{
“f0”: 123,
“f1”: “hello”
}
 	
NB: note that the KEY_FORMAT_TYPE is not needed as it is implicitly STRUCTURED as the resulting key has multiple fields.
 	
Conversely, the following would build a key with a nested struct:
CREATE STREAM FOO AS SELECT * FROM BAR **PARTITION BY STRUCT<f0, f1> AS bar;**
 	
Example resulting JSON key:
{
bar” : {
“f0”: 123,
“f1”: “hello”
}
}
	CREATE STREAM FOO AS SELECT * FROM BAR **PARTITION BY STRUCT<f0, f1>**;
	
	Example resulting JSON key:
	{
	“f0”: 123,
	“f1”: “hello”
	}
	
	Conversely, the following would build a key with a nested struct:
	CREATE STREAM FOO AS SELECT * FROM BAR **PARTITION BY STRUCT<bar STRUCT<f0, f1>>;**
	
	Example resulting JSON key:
	{
	bar” : {
	“f0”: 123,
	“f1”: “hello”
	}
	}

Summary

To summarise in a way that may help:

In the *logical* approach the schema of the key and value is *flattened* into a single schema. The KEY keyword is used to distinguish between fields in the value and the key. Namespacing is used to avoid field name clashes and the namespace ROWKEY or ROWVALUE is only needed where there is an ambiguity / clash. The schema within KSQL is a logical one, decoupled from the physical persisted schema in Kafka. Where this particularly comes to light is when dealing with keys that contain only a single field. With the logical approach the DDL and DML statements don’t care or know if a key with a single field is *persisted* as a primitive or as a single field within a struct, (i.e. JSON Object, Avro Record, etc). How the key is persisted is controlled by the WITH clause using KEY_FORMAT_TYPE both when reading in data and when writing it out.

In the *physical* approach the underlying persistence model of Kafka bleeds through more into the KSQL syntax. The exact schema of the persisted data can be expressed within in the DDL and DML statements. Again, this more notably comes to the fore when dealing with keys with a single field. The DDL and DML statements explicitly define if the single field is a primitive or a structured key.