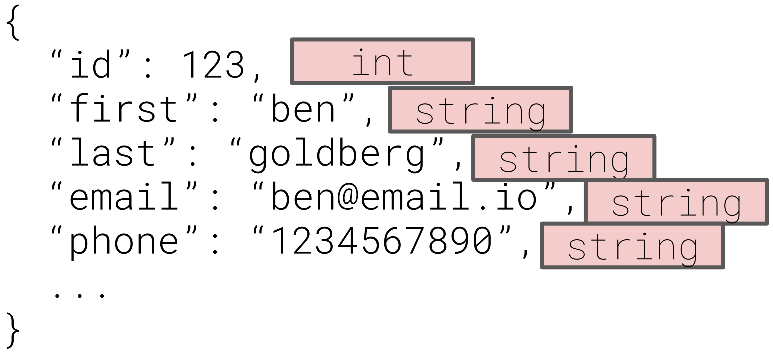
Data Schemas

Introduction to Data Schemas

Understanding Data Schemas

What is a Data Schema?

Data schemas set **expectations** about the shape and types of our data



```
{
  “id”: 123,                 int
  “first”: “ben”,            string
  “last”: “goldberg”,        string
  “email”: “ben@email.io”,   string
  “phone”: “1234567890”,     string
  ...
}
```

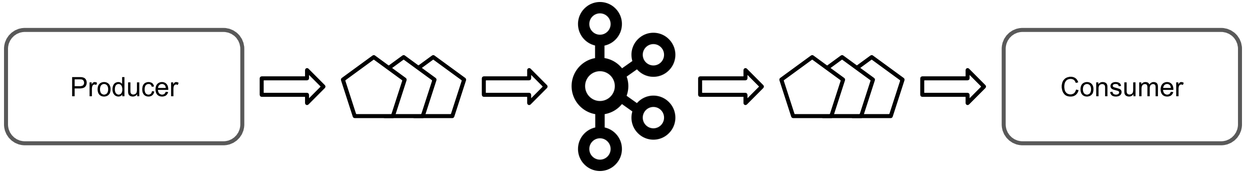
What is a Data Schema?

SQL Databases enforce schemas on tables

id int	first string	last string	email string	phone string
123	ben	goldberg	ben@email.io	1234567890
...				

What is a Data Schema?

Schemas decrease coupling between applications



Real-World Usage

Where are Data Schemas Used?

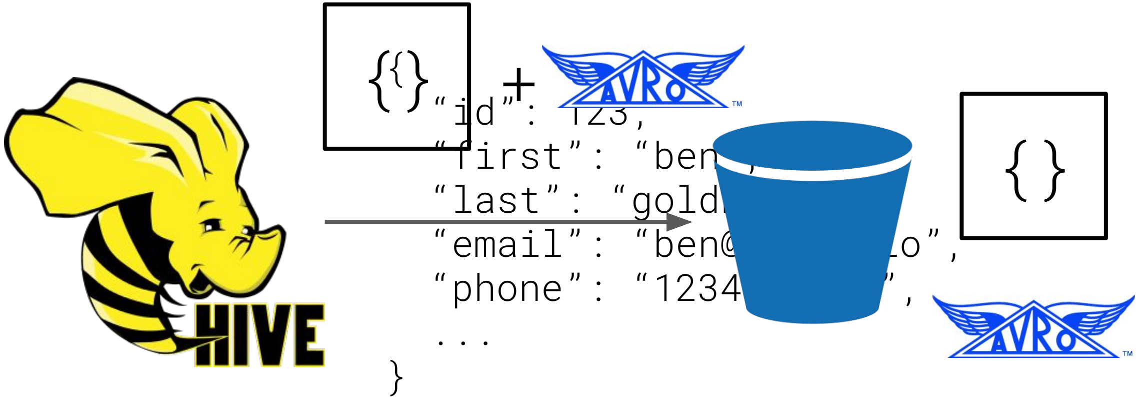
Declaring a table in Postgres or MySQL is an example of using a schema

```
CREATE TABLE store_location (
    name VARCHAR(80),
    city VARCHAR(40),
    latitude NUMERIC(10),
    longitude NUMERIC(10)
);
```

id	name	city	latitude	longitude
123	cool_clothing	chicago	67.14721	12.78431
...				

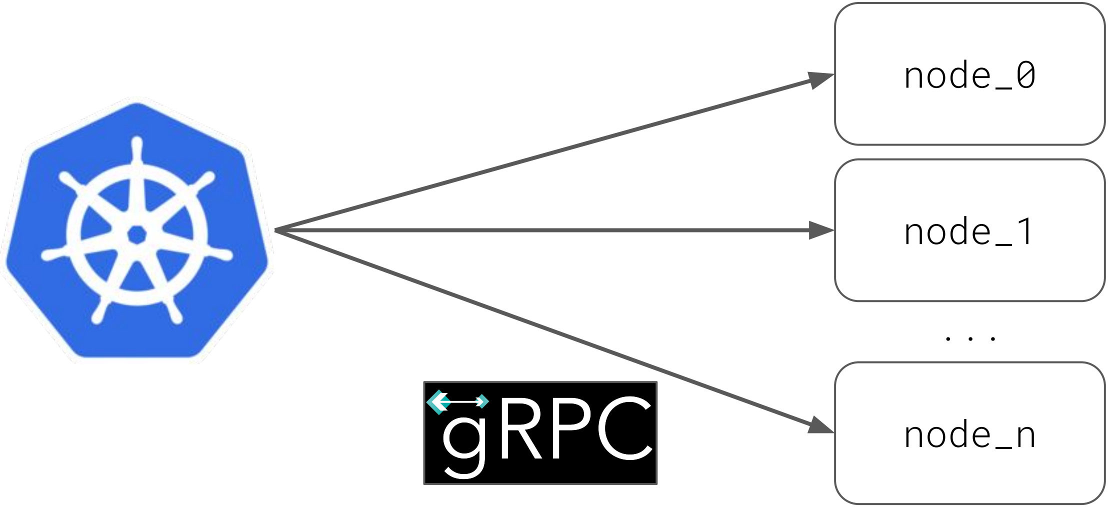
Where are Data Schemas Used?

The Hadoop ecosystem uses defined schemas to load data



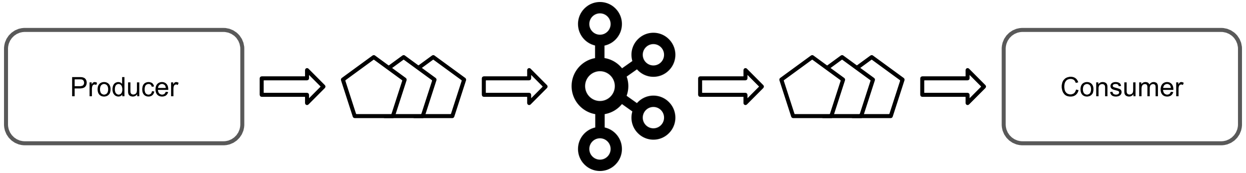
Where are Data Schemas Used?

Kubernetes uses gRPC to communicate with system components



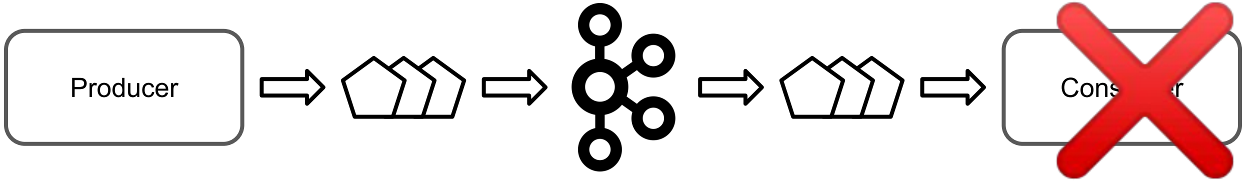
Data Streaming Without Schemas

Scenario: A system is released with no schema. All goes well at first.



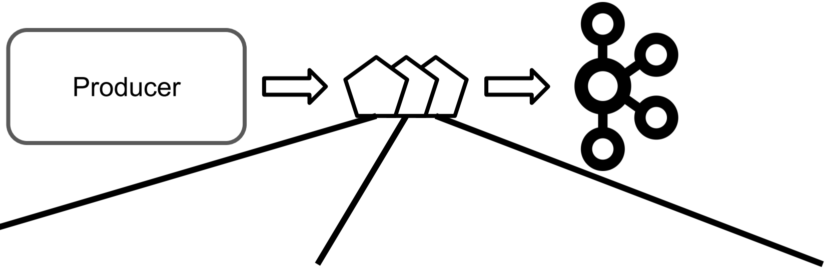
Data Streaming Without Schemas

Scenario: A few weeks later, our consumer mysteriously dies!



Data Streaming Without Schemas

Scenario: A renamed and missing field is crashing the consumer



```
message 3000:
{
  “id”: 123,
  “name”: “tom”
}
```

```
message 3001:
{
  “id”: 123,
  “name”: “tom”
}
```

```
message 3002: ❌
{
  “first_name”: “tom”
}
```

Demo: Data Streaming Without Schemas

Data Streaming with Schemas

Data Streaming with Schemas

Why they matter

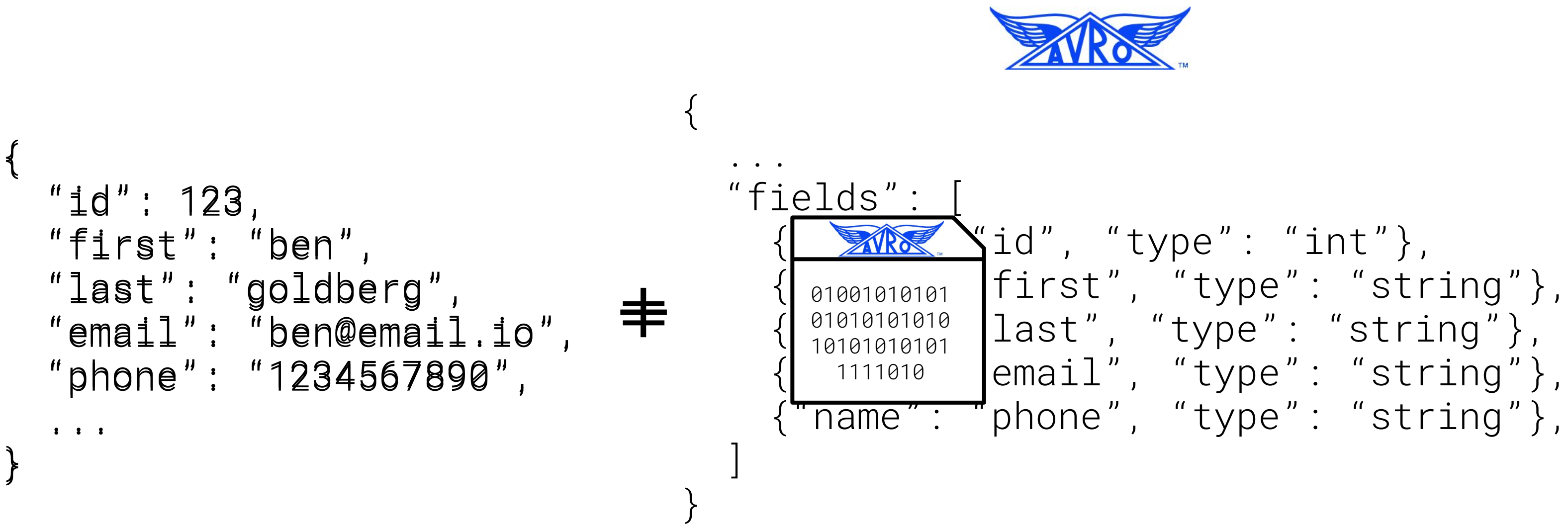
- Data streams are constantly evolving
- No schema = broken consumer on every data change
- Schemas allow consumers to function without updates
- Schemas provide independence and scalability
- Schemas can communicate version compatibility

Apache Avro

Apache Avro

What is Apache Avro?

Avro is a data serialization system that uses binary compression



```
{
  “id”: 123,
  “first”: “ben”,
  “last”: “goldberg”,
  “email”: “ben@email.io”,
  “phone”: “1234567890”,
  ...
}
```

≠

```
{
  ...
  “fields”: [
    {          “id”, “type”: “int”},
    {          first”, “type”: “string”},
    {          last”, “type”: “string”},
    {          email”, “type”: “string”},
    {“name”: “phone”, “type”: “string”},
  ]
}
```

How Avro Schemas are Defined

How Avro Schemas are Defined

Avro schemas are defined as JSON records

```
{
  “type”: “record”,
  “name”: “user”,
  “fields”: [
    {“name”: “id”, “type”: “int”},
    {“name”: “first”, “type”: “string”},
    {“name”: “last”, “type”: “string”},
    {“name”: “email”, “type”: “string”},
    {“name”: “phone”, “type”: “string”},
  ]
}
```

How Avro Schemas are Defined

The required **name** field identifies the Avro schema uniquely

```
{
  “type”: “record”,
  “name”: “user”,
  “fields”: [
    {“name”: “id”, “type”: “int”},
    {“name”: “first”, “type”: “string”},
    {“name”: “last”, “type”: “string”},
    {“name”: “email”, “type”: “string”},
    {“name”: “phone”, “type”: “string”},
  ]
}
```

How Avro Schemas are Defined

The optional **namespace** field groups the Avro schema with others

```
{
  “type”: “record”,
  “name”: “user”,
  “namespace”: “com.udacity”,
  “fields”: [
    {“name”: “id”, “type”: “int”},
    {“name”: “first”, “type”: “string”},
    {“name”: “last”, “type”: “string”},
    {“name”: “email”, “type”: “string”},
    {“name”: “phone”, “type”: “string”},
  ]
}
```

How Avro Schemas are Defined

All Avro schemas have a **type** and the root type is always **record**

```
{
  “type”: “record”,
  “name”: “user”,
  “namespace”: “com.udacity”,
  “fields”: [
    {“name”: “id”, “type”: “int”},
    {“name”: “first”, “type”: “string”},
    {“name”: “last”, “type”: “string”},
    {“name”: “email”, “type”: “string”},
    {“name”: “phone”, “type”: “string”},
  ]
}
```

How Avro Schemas are Defined

All Avro records have **fields** that define expected data keys and types

```
{
  “type”: “record”,
  “name”: “user”,
  “namespace”: “com.udacity”,
  “fields”: [
    {“name”: “id”, “type”: “int”},
    {“name”: “first”, “type”: “string”},
    {“name”: “last”, “type”: “string”},
    {“name”: “email”, “type”: “string”},
    {“name”: “phone”, “type”: “string”},
  ]
}
```

How Avro Schemas are Defined

Optional fields may be **null** or another primitive type

```
{
  “type”: “record”,
  “name”: “user”,
  “namespace”: “com.udacity”,
  “fields”: [
    {“name”: “id”, “type”: “int”},
    {“name”: “first”, “type”: “string”},
    {“name”: “last”, “type”: “string”},
    {“name”: “email”, “type”: “string”},
    {“name”: “phone”, “type”: [“null”, “string”]},
  ]
}
```

How Avro Schemas are Defined

The below fields consist of **primitive** types, ex: `null, string, int`

```
{
  “type”: “record”,
  “name”: “user”,
  “namespace”: “com.udacity”,
  “fields”: [
    {“name”: “id”, “type”: “int”},
    {“name”: “first”, “type”: “string”},
    {“name”: “last”, “type”: “string”},
    {“name”: “email”, “type”: “string”},
    {“name”: “phone”, “type”: [“null”, “string”]},
  ]
}
```

How Avro Schemas are Defined

The **record** type is a **complex** type, ex: `record, map, array`

```
{
  “type”: “record”,
  “name”: “user”,
  “namespace”: “com.udacity”,
  “fields”: [
    {“name”: “id”, “type”: “int”},
    {“name”: “first”, “type”: “string”},
    {“name”: “last”, “type”: “string”},
    {“name”: “email”, “type”: “string”},
    {“name”: “phone”, “type”: [“null”, “string”]},
  ]
}
```

Defining your first Avro Record Demonstration

Demo: Defining your first Avro Record

Apache Avro Data Types

Avro Data Types

Primitive Types

- null
- boolean *(true / false)*
- int, long, float, double *(1 / 123.37)*
- bytes *(b’AE002448FF’)*
- string *(“hello world”)*

Avro Data Types

Complex Types

- `record`
- `enum`
- `array`
- `map`
- `union`
- `fixed`

Avro Data Types

Enumerations are a set of named symbols

```
{
  “type”: “enum”,
  “name”: “us_states”,
  “symbols”: [“AL”, “AK”, “AZ”, “AR”, “CA”, ...]
}
```

Avro Data Types

Arrays store ordered fields of **primitive** or **complex** types

Primitive

```
{
  “type”: “array”,
  “items”: “string”,
}
```

Complex

```
{
  “type”: “array”,
  “items”: {
    “type”: “record”,
    “fields”: [
      {“name”: “id”, “type”: “int”}
    ]
  }
}
```

Avro Data Types

Maps store fields as a **string key** to **value** of **primitive** or **complex** type

Primitive

```
{
  “type”: “map”,
  “values”: “int”,
}
```

Complex

```
{
  “type”: “map”,
  “values”: {
    “type”: “record”,
    “fields”: [
      {“name”: “id”, “type”: “int”}
    ]
  }
}
```

Avro Data Types

Unions denote that more than one type may be used.

```
{
  “type”: “map”,
  “values”: {
    “type”: “record”,
    “fields”: [
      {“name”: “zipcode”, “type”: [“null”, “int”, “string”]}
    ]
  }
}
```

Avro Data Types

Fixed denotes a fixed size entry in **bytes**

```
{
  “name”: “md5”,
  “type”: “fixed”,
  “size”: 16
}
```

Complex Records in Apache Avro

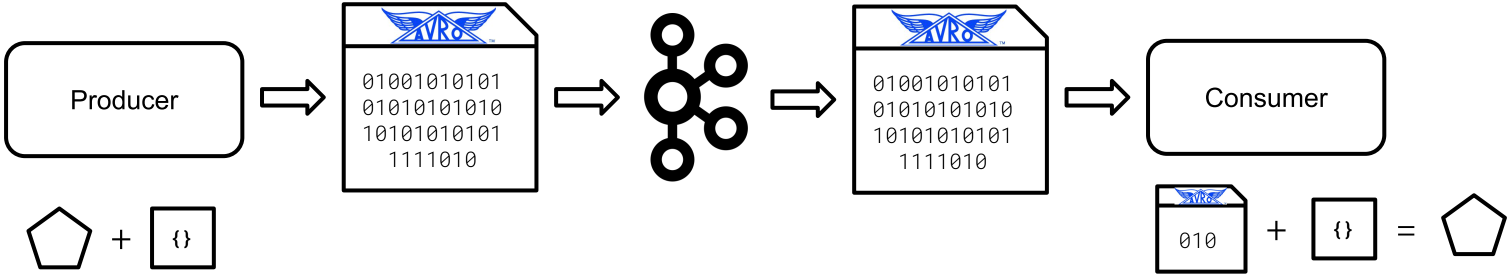
Demo: Defining a Complex Avro Record

Apache Avro and Kafka

Apache Avro and Kafka

Producing and Consuming Kafka Data with Apache Avro

The Producer must define an **Avro** schema and **encode the data**



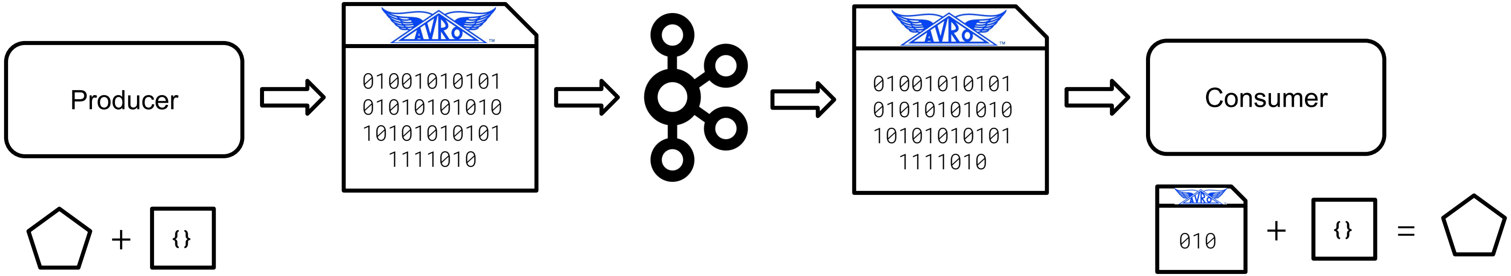
Schema Registry

Schema Registry

How Kafka and Schema Registry Integrate

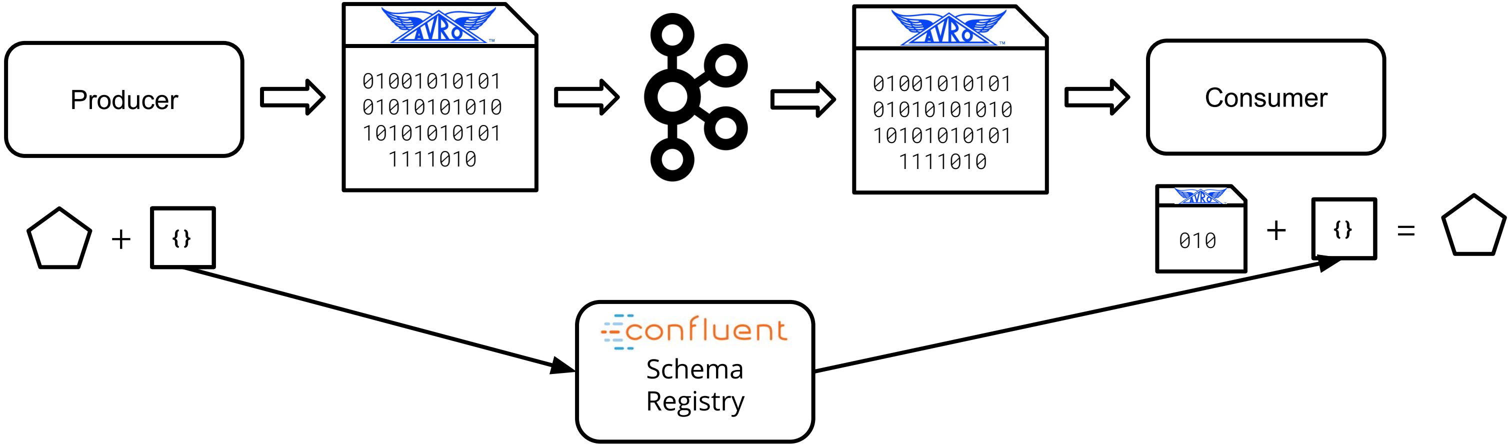
Producing and Consuming Data with Schema Registry

Sending a schema definition with every message **adds overhead**



Producing and Consuming Data with Schema Registry

Sending a schema definition with every message **adds overhead**



Key Points

Schema Registry

confluent

- Schema Registry stores state in Kafka itself
- Schemas only need to be sent to Schema Registry once
- Clients fetch schemas as needed from the registry
- Does not support deletes
- Has an HTTP REST Interface
- May use with any application, not just Kafka apps!

Architecture

Schema Registry

confluent

- Built in Scala and Java, runs on the JVM
- High portable, runs on nearly all OSes
- Stores all of its state in Kafka topics, not a database
- Exposes an HTTP web-server with a REST API
- Can run standalone or clustered with many nodes
- Uses ZooKeeper to choose leader in cluster mode

Integrating Schema Registry

Demo: Producing and Consuming Data with Schema Registry

Schema Evolution and Compatibility

Schema Evolution and Compatibility

Understanding Schema Evolution

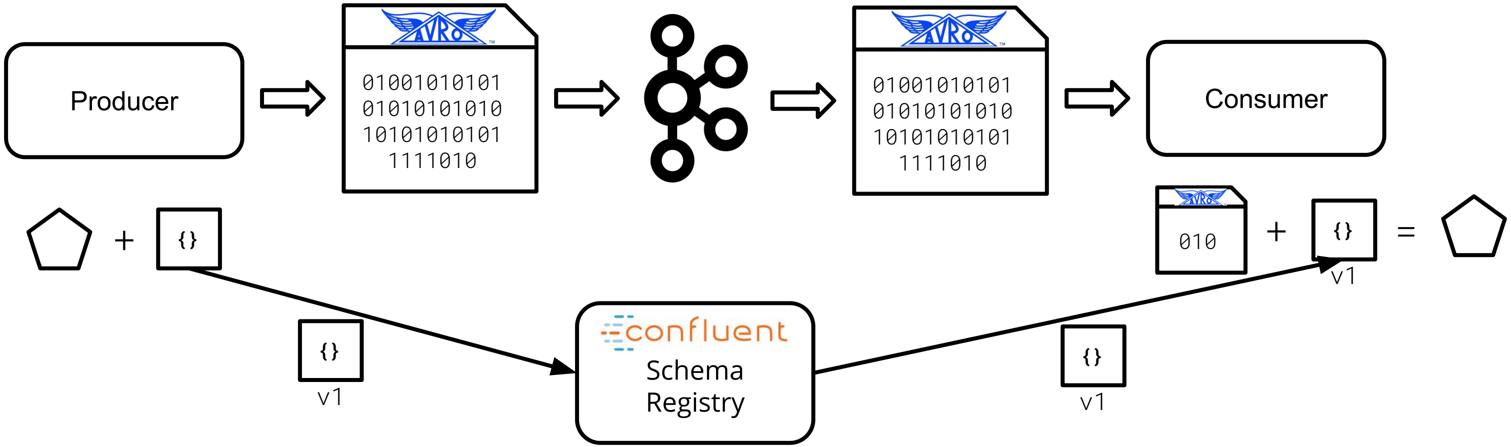
Schema Evolution

The process of changing the schema of a given dataset is referred to as **schema evolution**. Modifying, adding, or removing a field are all forms of schema evolution.

What is Schema Compatibility?

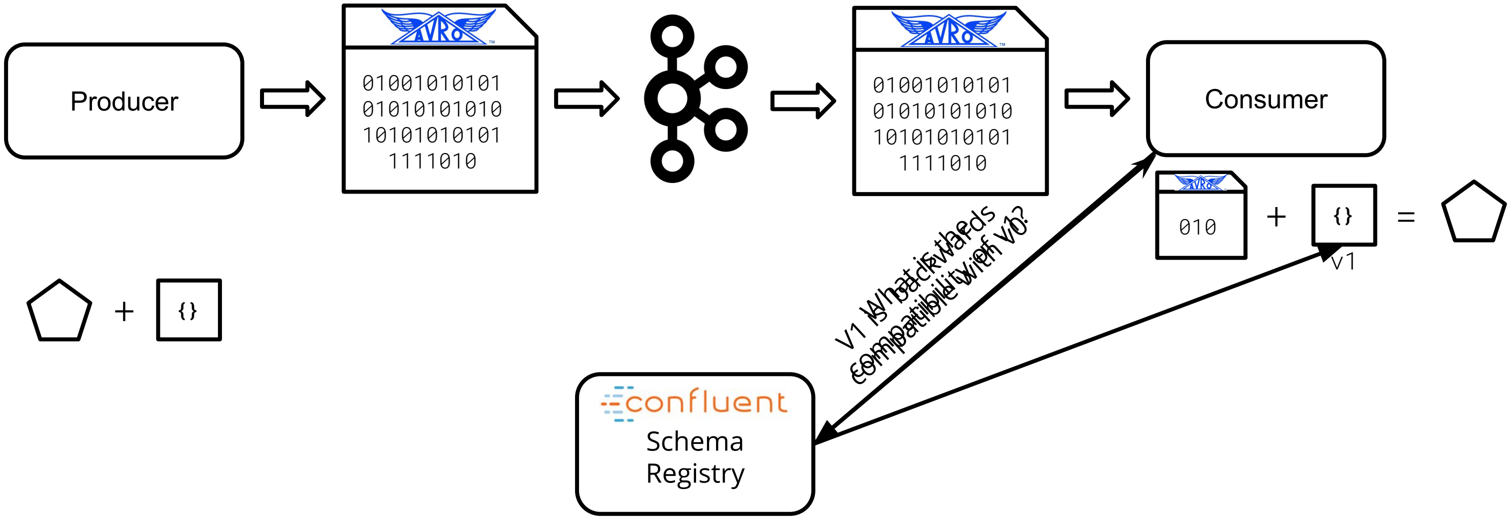
Schema Compatibility

Schema Registry **tracks compatibility** between schema versions



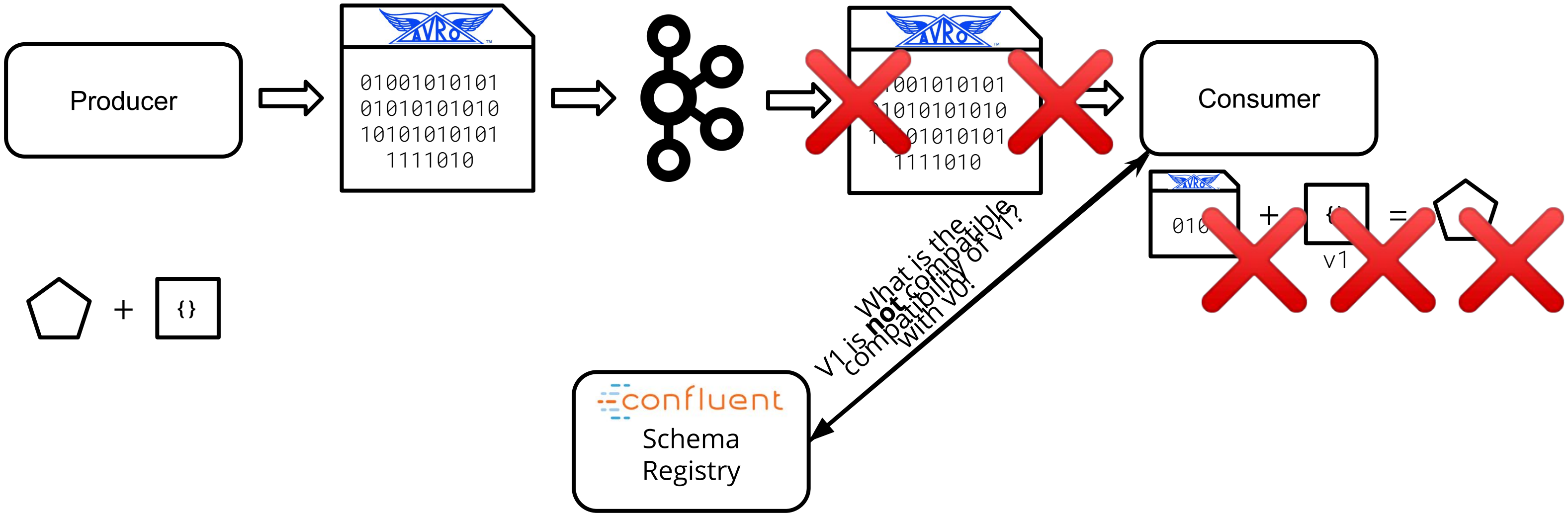
Schema Compatibility

If the schema is compatible, the consumer continues consumption



Schema Compatibility

If the schema is incompatible, the consumer will cease consumption



Backward Compatibility

Backward Compatibility

Consumers developed against the **latest** schema can use **older** data

- Addition of **optional** fields or the **deletion** of a field in the latest schema are backward compatible changes

```
{
  “type”: “record”,
  “name”: “purchase”,
  “fields”: [
    {“name”: “username”, “type”: “string”},
    ~~{“name”: “amount”, “type”: “float”},~~
    {“name”: “email”, “type”: “string”},
    {“name”: “memo”, “type”: [“null”, “string”]
  ]
}
```

Forward Compatibility

Forward Compatibility

Consumers developed against the **previous** schema can use the **latest**

- Addition of **new** fields or the **deletion** of a **optional** fields in the new schema are forward compatible changes

```
{
 “type”: “record”,
 “name”: “purchase”,
 “fields”: [
  {“name”: “username”, “type”: “string”},
  {“name”: “amount”, “type”: “float”},
  {“name”: “email”, “type”: “string”},
 ] ~~{“name”: “memo”, “type”: [“null”, “string”]~~,
} ] {“name”: “area_code”: “type”: “string”}
} ]
}
```

Full Compatibility

Full Compatibility

The change is both **backward** and **forward** compatible

- Changing the default for a field is fully compatible

```
{
  “type”: “record”,
  “name”: “purchase”,
  “fields”: [
    {“name”: “username”, “type”: “string”},
    {“name”: “amount”, “type”: “float”},
    {“name”: “email”, “type”: “string”},
    {“name”: “memo”, “type”: [“null”, “string”],
    {“name”: “area_code”: “type”: “string”},
    {“name”: “action”, “type”: “string”, default: “pending_payment”}
  ]
}
```

No Compatibility

None Compatibility

None compatibility indicates that compatibility is not tracked

- **Do not use** None compatibility!
- If your schema has changed in a breaking fashion, **always create a new topic** and update your consumers to use that topic
- None does *not* indicate a breaking change, it is more akin to “unknown” since Schema Registry no longer tracks the compatibility

Summary