Data encoding and metadata for streams

Very Topic

Such Partition

WOW !

Me at a glance

- My name is Jonathan Winandy (@ahoy_jon).
- I am a Data pipeline engineer :
  - I worked on a “DataLake” !
- I use tools in the larger Java ecosystem like Java, Scala, Clojure, Hadoop …
- And I am an “entrepreneur”.

> Introduction

I cofounded two companies and they use streams as their data backbone.

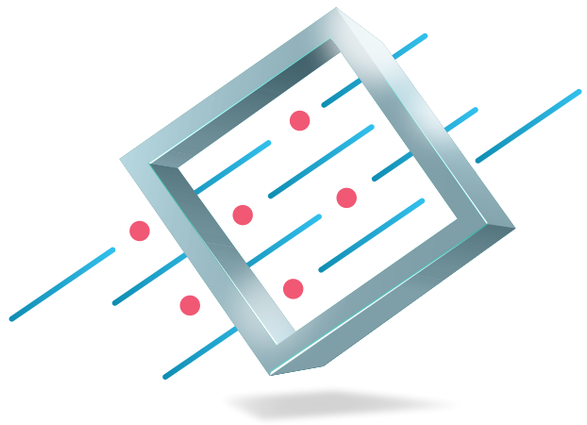
Health care oriented software engineering.

Provide :

Coordination for health care professionals.

> Introduction

I cofounded two companies and they use streams as their data backbone.



Primatice

@PrimaticeData

"Good dataviz, surreal backends."

Provide :

Tools and methods for Data capitalisation.

What are Streams ?

It's an abstract data structure with the following :

operations :

- append(bytes) -> void?
- readAt(int) -> null | bytes

rule 1 :

$\forall p \in \mathbb{N}$, for some definition of '=='

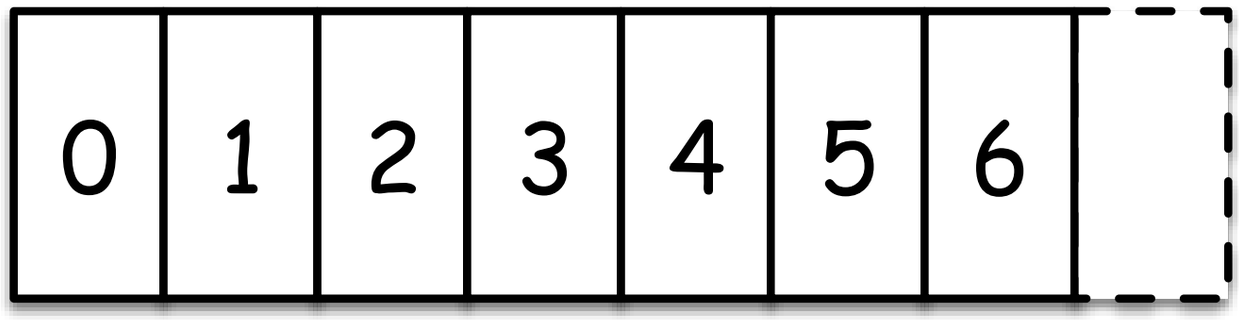
x := readAt(p)

y := readAt(p)

x != null => x == y

Rule 1 implies : Infinite cacheability
once the data is available at a position.

Streams are the simplest way to manage data.



And they are naturally compatible with the perception of information from a singular observer …

But be careful, streams are definitely not like queues, ESB, EAI, or what ever messaging solution comes to mind …

gregyoung
@gregyoung

Follow

I understand more now why people in the java community have such a hard time with event sourcing when they use rabbit in the middle

4:38 PM - 25 Oct 2014

2 RETWEETS **2** FAVORITES

There is a lot to tell on Streams

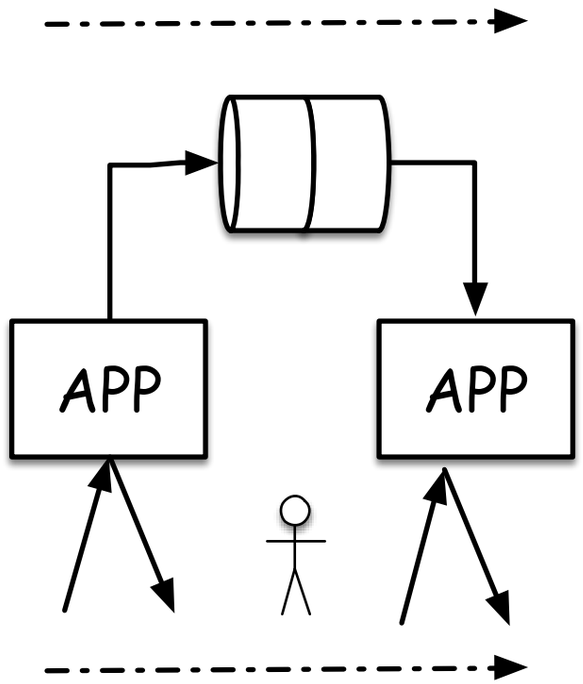
- Sub events : Events are pre-projected into …
- Quantum of action : A 'user' action generates zero or one event (no more).
- Structural sharing for large payload (cf. Content Addressable Storage).
- Garbage collection for append only data structures.

---

this presentation

- Causality enforcement in asynchronous contexts : On important request, causality is enforced.
- **Binary encoding and Metadata.**

A quick note on Causality



"Who is the fastest between the Data bus and the client ?"

You don't want to bet, especially under load.

If you don't ensure causality for web apps, some strange comportements may arise :

Sometimes, as a user, I cannot see my own "edits".

Sometimes, as a client, I cannot buy on the website after I checkout my basket.

Data encoding and metadata for streams

Very Topic

Such Partition

WOW !

> Content

Content :

- Data encoding
- Identity
- Metadata
- Datagram
- Conclusion

State of data encodings in the industry

- **As always worse is considered better.**
- Most of streams have data encoded in :
  - CSV/TSV
  - JSON
  - Platform specific serialisations (eg: Java serialisation, Kryo)

Why this is important ?

- Some streams may contains very large amount of Data, the chosen encoding must be cpu and space efficient.

- Streams are processed
  by many programs,
  and many intermediaries,
  for many years,
  the chosen encoding must be processable in a generic way.

JSON is the lower denominator

Plus :

- It reaches the browser, you can produce and consume data from inside a web page.

A lot of Cons :

- Inefficient,
- No dates, no proper numerics,
- Very basic data structures,
- Very error Prone.

We all need JSON, but we should use it only when we can't avoid it.

Eg : In our databases, we can avoid JSONs ;)

How bad JSON is ?

{“name”:"Bob",
“age":11,
"gender":"Male"}

39 Bytes for
10 Bytes of data

:02:06:62:6f:62:02:16:02:da:01

> Data encoding

relevant ones	popular binaries			low tech		cognitect			"papa ?!"	
	Avro	Thrift	Proto Buf	JSON	CSV	Fressian	Transit	EDN	XML	RDF
binary	YES			NO		YES	OK	NO	??	
generic	YES	??	NO	YES		YES			YES	
schema based	YES			NO	YES	NO			??	meta
specific encoding	YES			NO	"STRINGS"	YES			OK	Literals
reach the browser	YES		NO	+++++	OK	NO	YES		OK	
easy ?	NO	I PASS		"true"		YEP			HUM ?	...
safe ?	YES	HUM?		NO	NO				MISMATCH	YES
has dates?	Soon	NO		NO		YES			YES	

Identity

- Most mechanism around stream assure an "at most once delivery".
- An identity definition is necessary to ensure idempotency.

> Identity

There are 2 ways to refer to a message :

- with a fingerprint calculated from the message (digest).
- with an external identifier (like UUIDs).

> Identity

F0991FD1-D58A-4A5F-8D13-903F368882D1

8AA5C612-B365-4F8F-AF3F-DF623E1F6B22

93A87D37-0658-47C9-84F6-801E83A5821C

UUIDs allow :

- to manage things that are not encoded yet.
- to avoid the hashing and the parsing of payloads.

Recommandation : add an UUID (128bits) to every elements of the stream.

Metadata

- Metadata uses range from the very useful (like http headers) to the very meta meta[1].
- Metadata on Stream elements is most of the time implicit, like for example the Content-Type :
  - "It's a stream of JSONs" then every element of the stream has *"content-type=application/json"*.

[1] I am looking at you RDF !

> Metadata

What kind of metadata there are for streams element ?

- Content-type or data-encoding :
  `e.g. : application/json`

- Type or Profile : indicate that the given element is an instance of a given type.
  `e.g. : domain.model.MessageSent`

- Provenance information :

```
e.g. : {“env”:”test”,
         “application”:{“name”:”webapp”,
             “version”:{“commit”:”68546ca…”}}}
```

A quick note on provenance

The provenance is practical in distributed systems we want to know :

- from which node do a element comes.
- on the behalf of which agent this element is created.
- from which environment[1] a element comes.

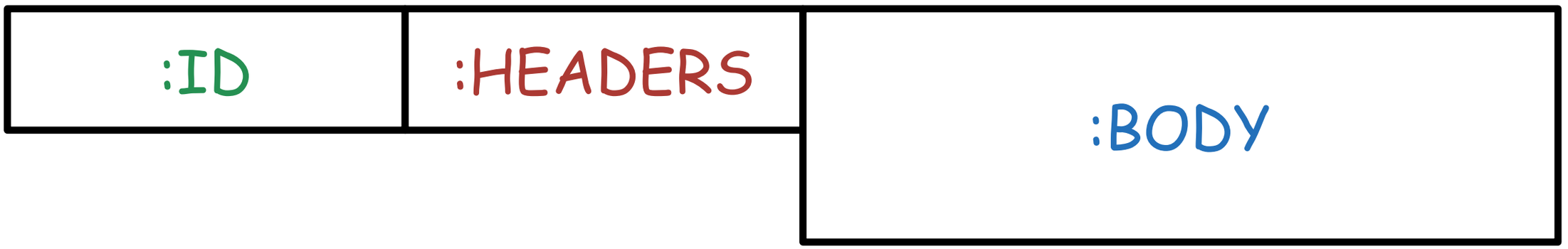
[1] with new architecture and Data Labs, environments are sometimes shared on the same infrastructure (eg : no Pre-Production platform). It's then very useful to safeguard against the pollution of data.

> Metadata

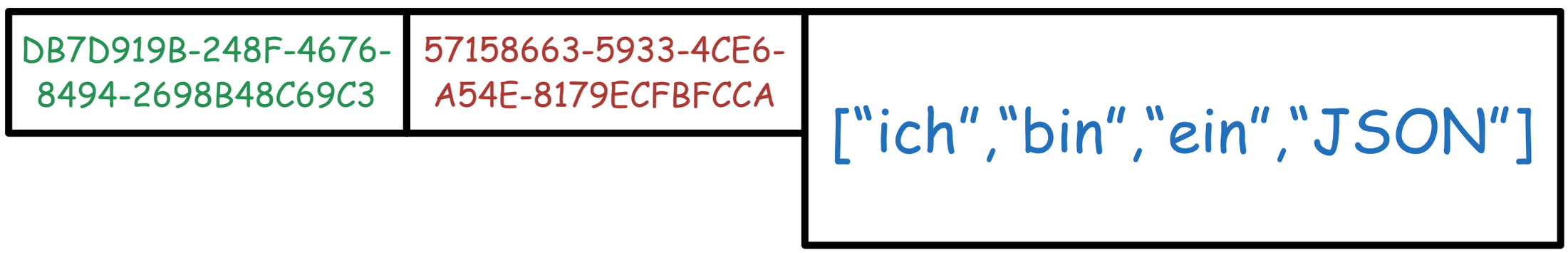

```
{
    "content-type":"application/json",
    "profile":"domain.model.MessageSent",
    "provenance":{
        "application":{
            "name":"Blaze",
            "version":"68546ca6e963981a8279aa327cc1e1362d15554e"
        },
        "node":{
            "environment":"test",
            "device":{
                "interface":{
                    "en0":{
                        "addresses":{
                            "192.168.0.13":{
                                "family":"inet",
                                "netmask":"255.255.255.0"
                            }
                        }
                    }
                }
            },
            "hostname":["Blaze"],
            "platform_family":"mac_os_x"
        }
    }
}
```


- The metadata of an element can represent a significant piece of data. Sometimes more than the data itself.
- !! The same piece of metadata can be shared across many elements. !!

Anatomy of an element



e.g.



> Datagram

1. Create and register your headers
(in a distributed Key/Store for example) .

4813EDF2-B04E-4B70-AB04-0F9EA456E032


```
{
   "content-type":"application/json",
   "profile":"domain.model.MessageSent",
   "provenance":{
      "application":{
         "name":"webapp",
         "version":"68546ca6e963981a8279aa327cc1e1362d15554e"
      },
      "node":{
         "environement":"test",
         "network":{
            "interface":{
               "en0":{
                  "addresses":{
                     "192.168.0.13":{
                        "family":"inet",
                        "netmask":"255.255.255.0",
                        "broadcast":"192.168.0.255"
                     }
                  }
               }
            }
         },
         "hostname":["Blaze"],
         "platform_family":"mac_os_x"
      }
   }
}
```

> Datagram

2. use it in your stream !

5462E738-ABAA-452F-87E0-FD38AEB9DF81	4813EDF2-B04E-4B70-AB04-0F9EA456E032

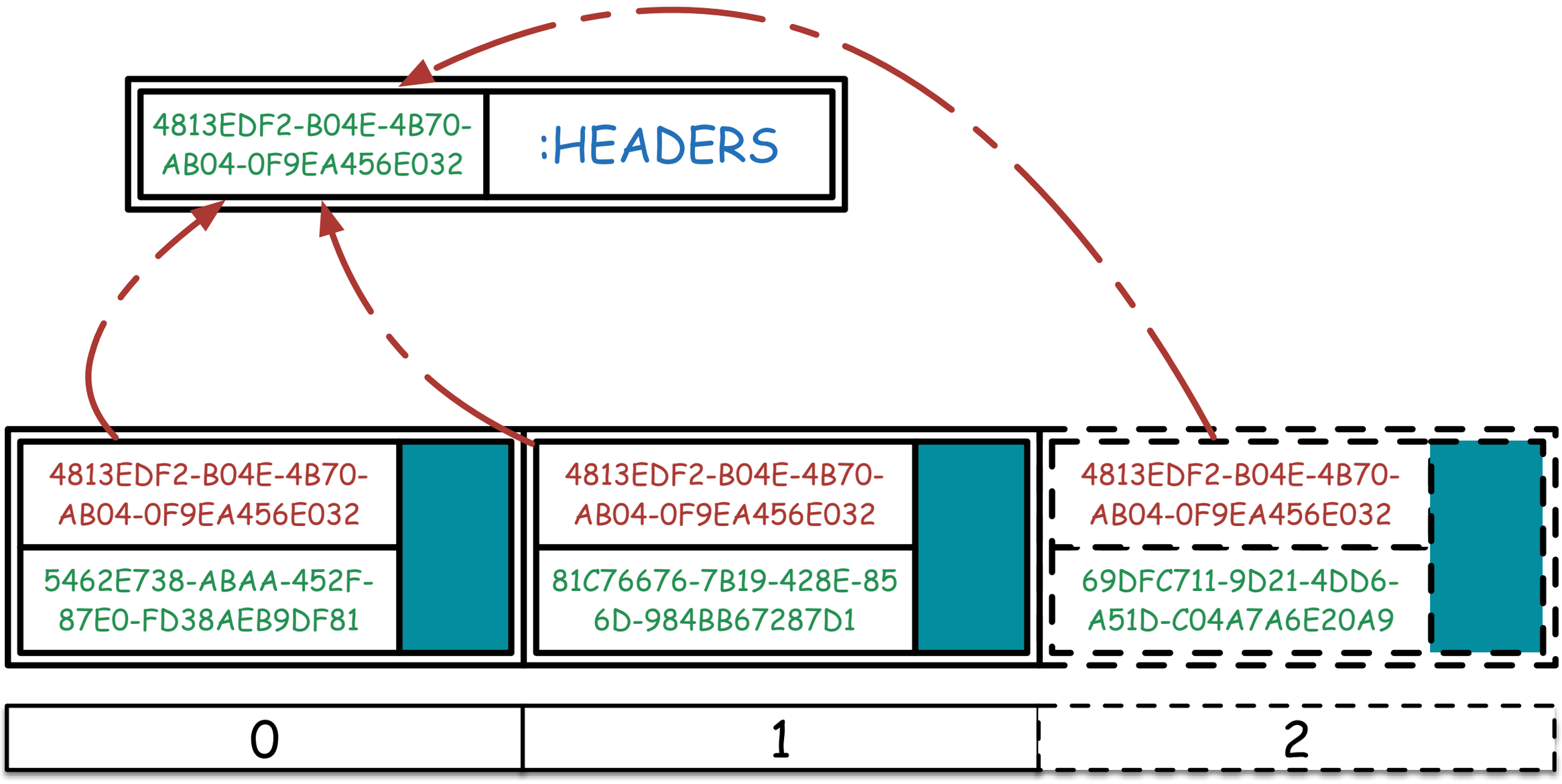
```
{"cid": {"idStr": "498683D2-1192-4794-8C23-5BE49EEEC763"},
 "userId":
  {"idStr": "BC3D8614-AF1F-48C8-B91F-0D907FD0FAF3"},
 "content": "  Contenu de message de test"}
```

81C76676-7B19-428E-856D-984BB67287D1	4813EDF2-B04E-4B70-AB04-0F9EA456E032

```
{"cid": {"idStr": "498683D2-1192-4794-8C23-5BE49EEEC763"},
 "userId":
  {"idStr": "BC3D8614-AF1F-48C8-B91F-0D907FD0FAF3"},
 "content": "  Contenu de message de test”}
```

> Datagram

Ho : You can have also have a stream of headers …



> Conclusion

If you don't yet use streams instead of databases, start to use one next Monday (even with JSON and no headers…).

If you do already use streams … Well, you know what to do ! ;)

Hi,

db.oplog.rs.count() shows 18878152 records

An external java app query periodically oplog.rs with the following code snippet:

```
 private DBCursor oplogCursor(final BSONTimestamp timestampOverride) {
                BSONTimestamp time = timestampOverride == null
                                ? MongoDBRiver.getLastTimestamp(client, definition) : timestampOverride;
                DBObject indexFilter = getOplogFilter(time);
                if (indexFilter == null) {
                        return null;
                }

                int options = Bytes.QUERYOPTION_TAILABLE
                                | Bytes.QUERYOPTION_AWAITDATA | Bytes.QUERYOPTION_NOTIMEOUT;

                // Using OPLOGREPLAY to improve performance:
                // https://jira.mongodb.org/browse/JAVA-771
                if (indexFilter.containsField(MongoDBRiver.OPLOG_TIMESTAMP)) {
                        options = options | Bytes.QUERYOPTION_OPLOGREPLAY;
                }
```

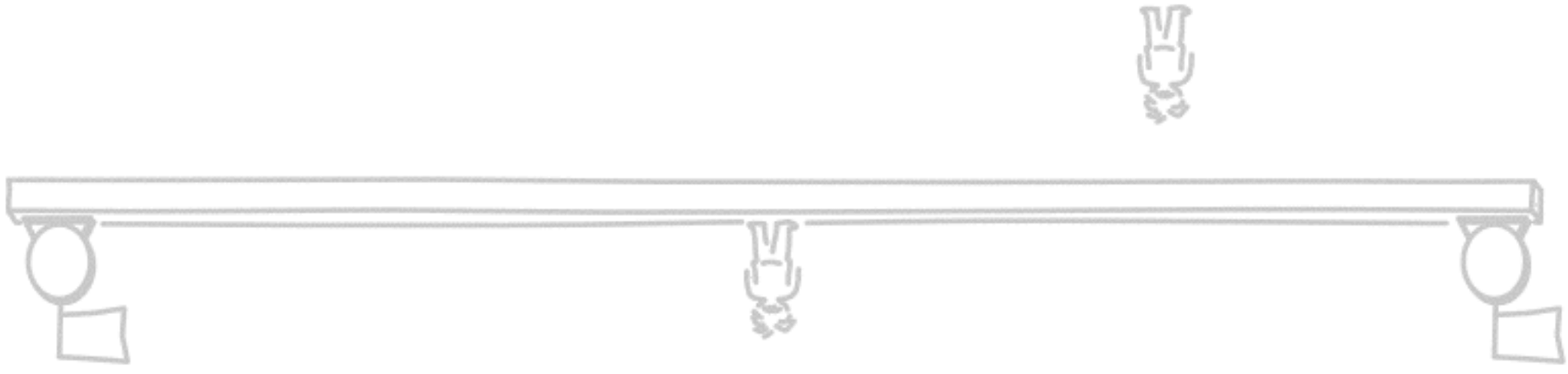
Tapping into MySQL replication stream

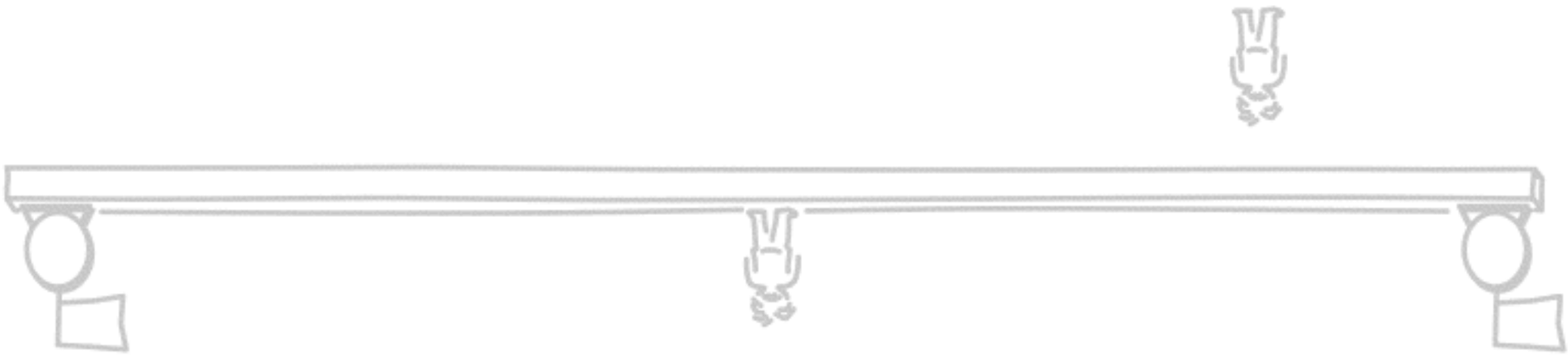
> PREREQUISITES: Whichever user you plan to use for the BinaryLogClient, he MUST have REPLICATION SLAVE privilege. Unless you specify binlogFilename/binlogPosition yourself (in which case automatic resolution won't kick in), you'll need REPLICATION CLIENT granted as well.

```
BinaryLogClient client = new BinaryLogClient("hostname", 3306, "username", "password");
client.registerEventListener(new EventListener() {

    @Override
    public void onEvent(Event event) {
        ...
    }
});
client.connect();
```

> By default, BinaryLogClient starts from the current (at the time of connect) master binlog position. If you wish to kick off from a specific filename or position, use client.setBinlogFilename(...) + client.setBinlogPosition(...).





Bonus :What is a CAS ?

A Content Adressable Storage is a specific "key value store" :

operations :

- store(bytes) -> key
- get(key) -> null | bytes

rule 1 :

key = h(data)

h being a cryptographic hash function like md5 or sha1.

rule 2 :

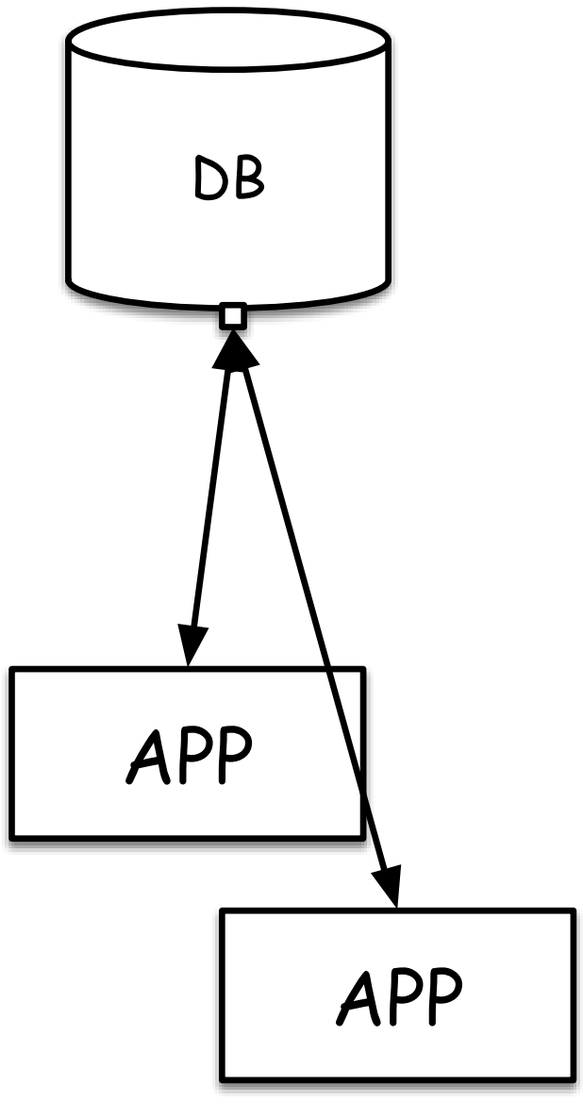
∀data

get(store(data)) = data

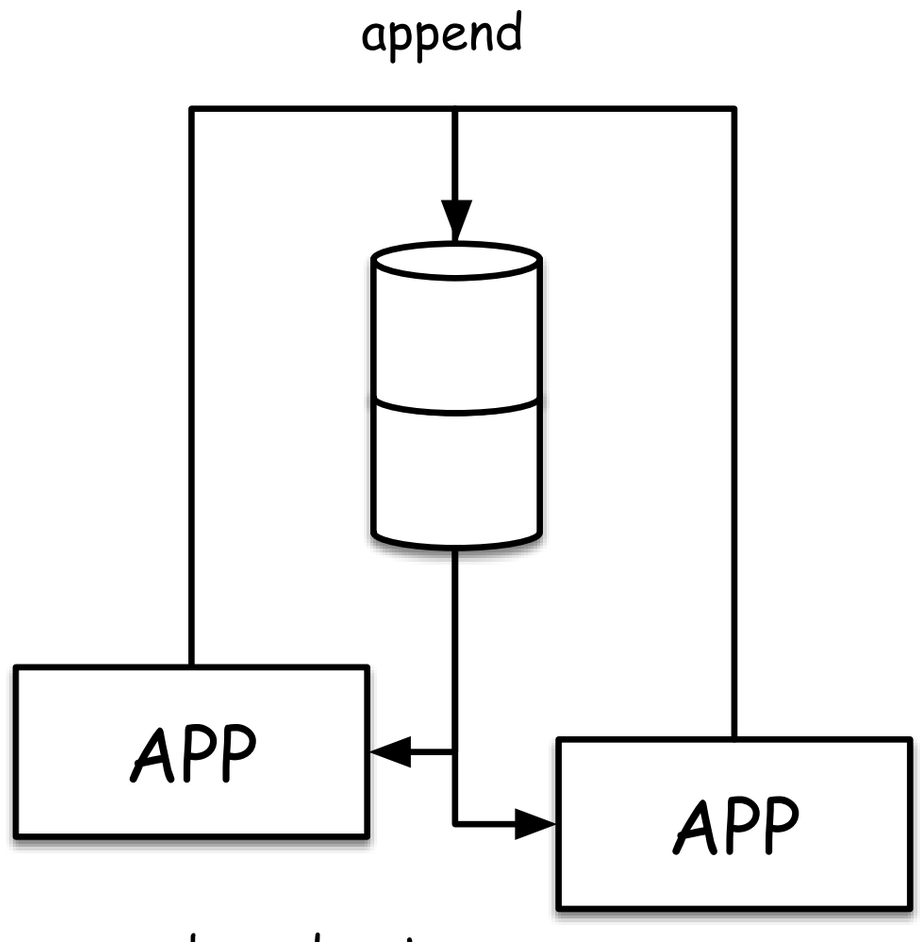
Rule 1 and 2 imply :
Infinite cacheability and scalability.

Exemple of architectures

CLASSICAL



WITH STREAMS



The broadcast mechanism is equivalent to a db replication mechanism.