Stream processing with R and Amazon Kinesis

Big Data Day LA 2016

Gergely Daroczi

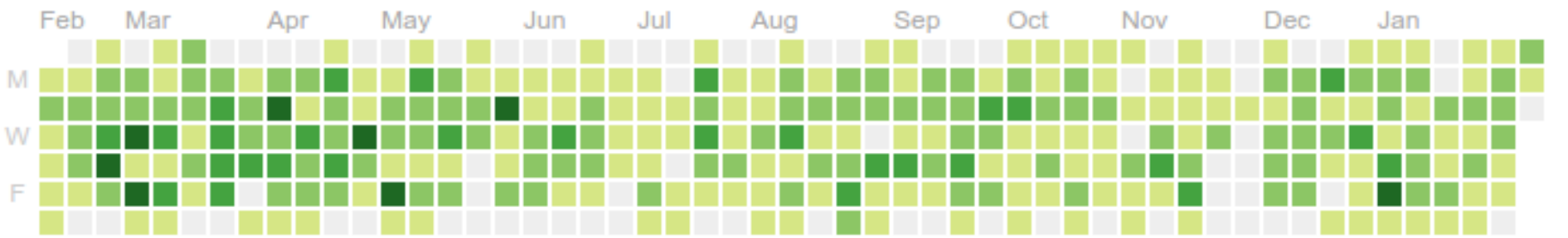
@daroczig

July 9 2016

About me

CARD.COM

Contributions



Summary of pull requests, issues opened, and commits. Learn how we count contributions.

Less More

Contributions in the last year	Longest streak	Current streak
3,477 total	22 days	2 days
Feb 9, 2015 – Feb 9, 2016	December 26 – January 16	February 7 – February 8

CARD.com's View of the World

Gergely Daróczi @daroczig · Apr 11

Just received my "I ♥ R" prepaid debit card from @CARD. Will be fun to use this #rstats designed card at #user2015 :)

RETWEETS 10 FAVORITES 16

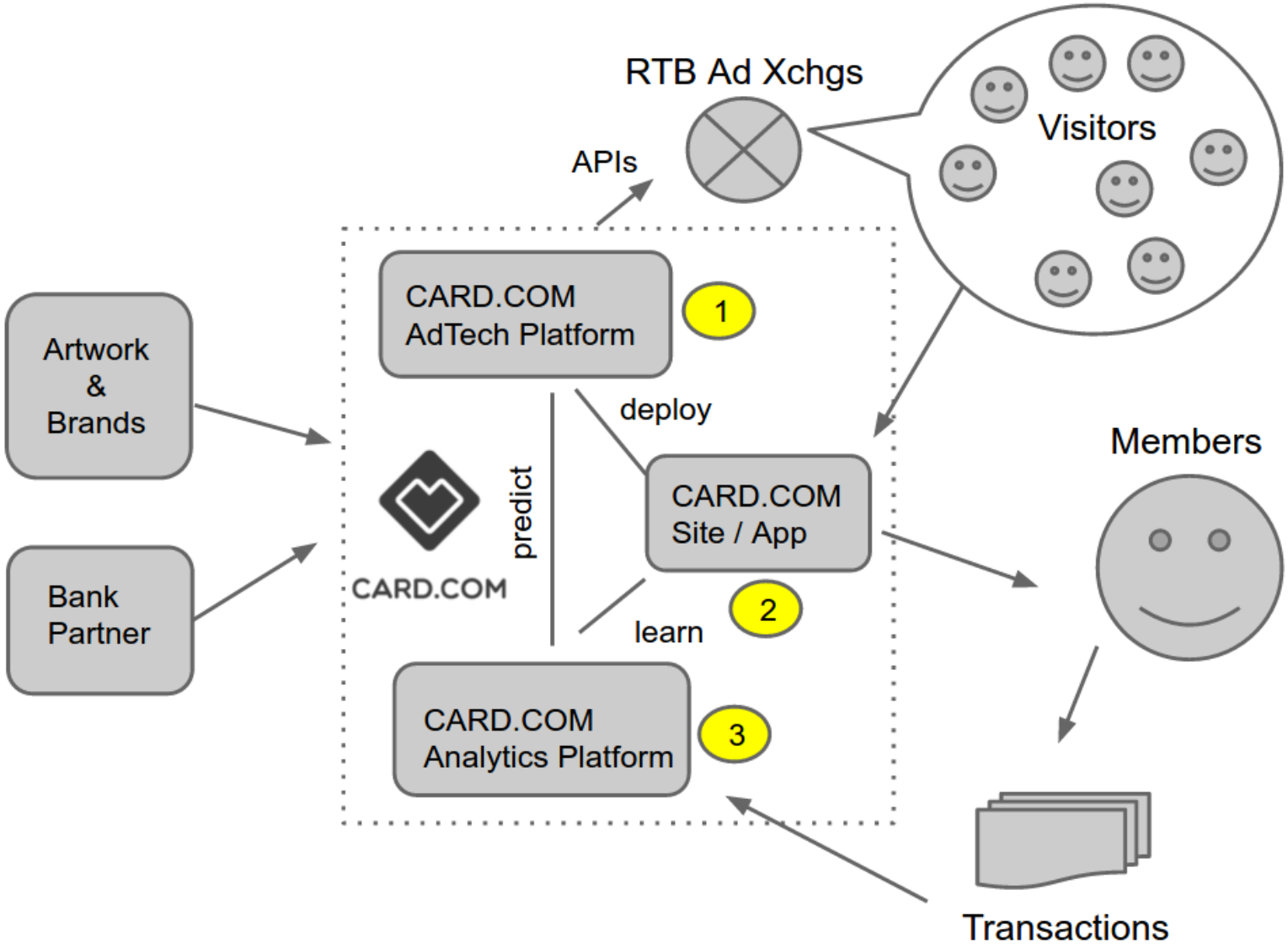
CARD.com's View of the World

CARD.COM

$\neq$ $\neq$

$\neq$

Modern Marketing at CARD.com



Further Data Partners

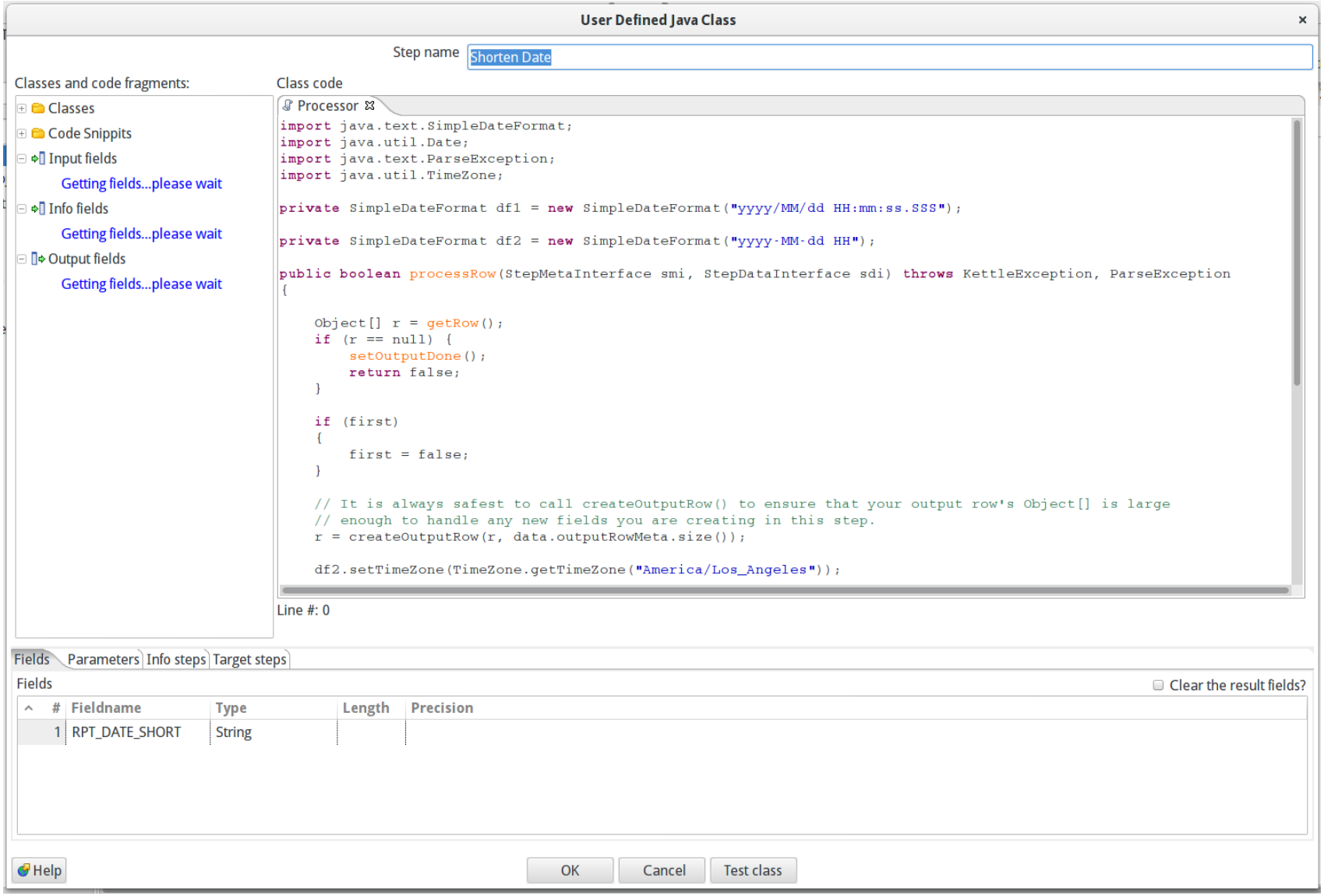
CARD.COM

- card transaction processors
- card manufacturers
- CIP/KYC service providers
- online ad platforms
- remarketing networks
- licensing partners
- communication engines
- others

Infrastructure

Why R?

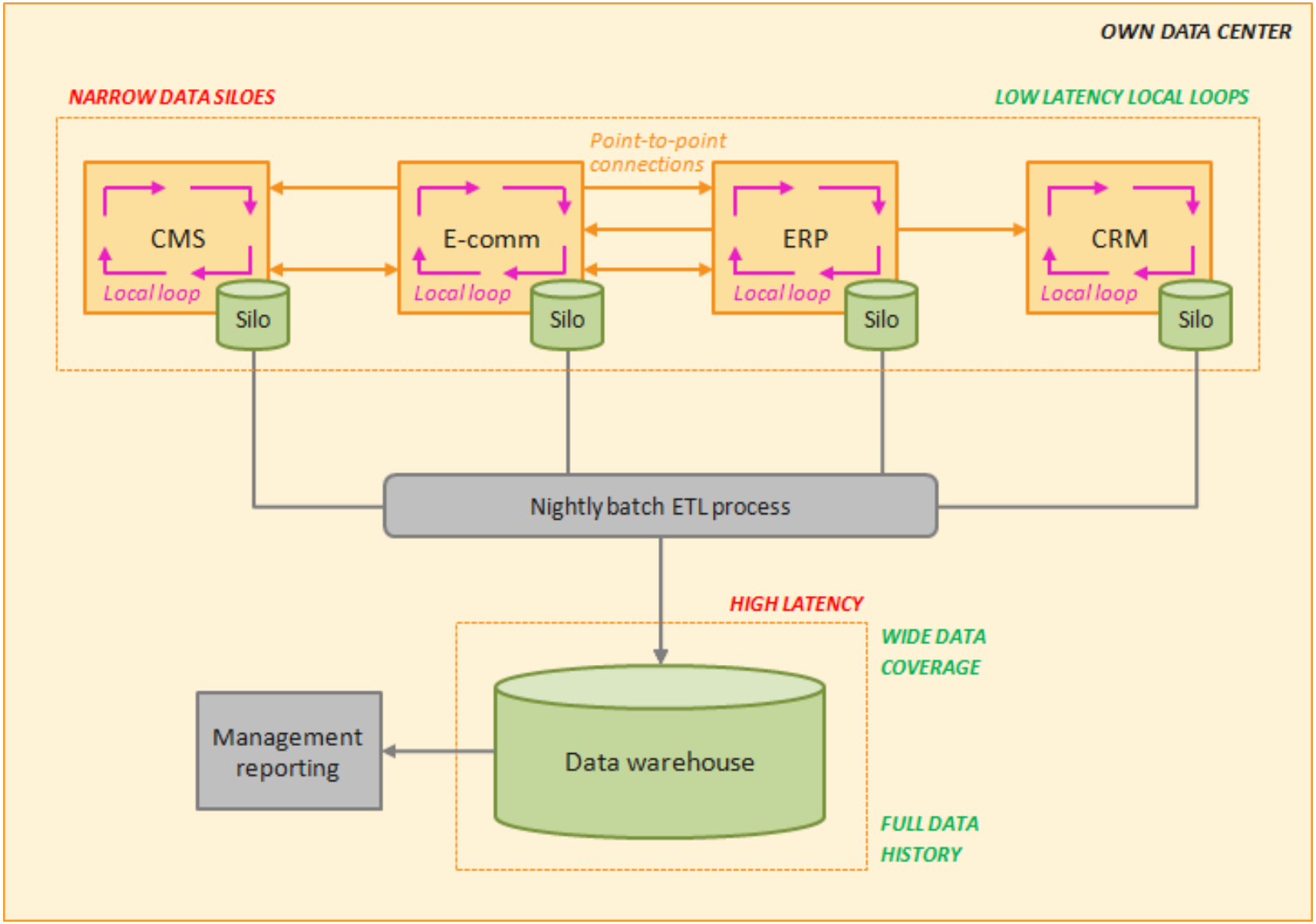
CARD.COM



Infrastructure

The Classic Era

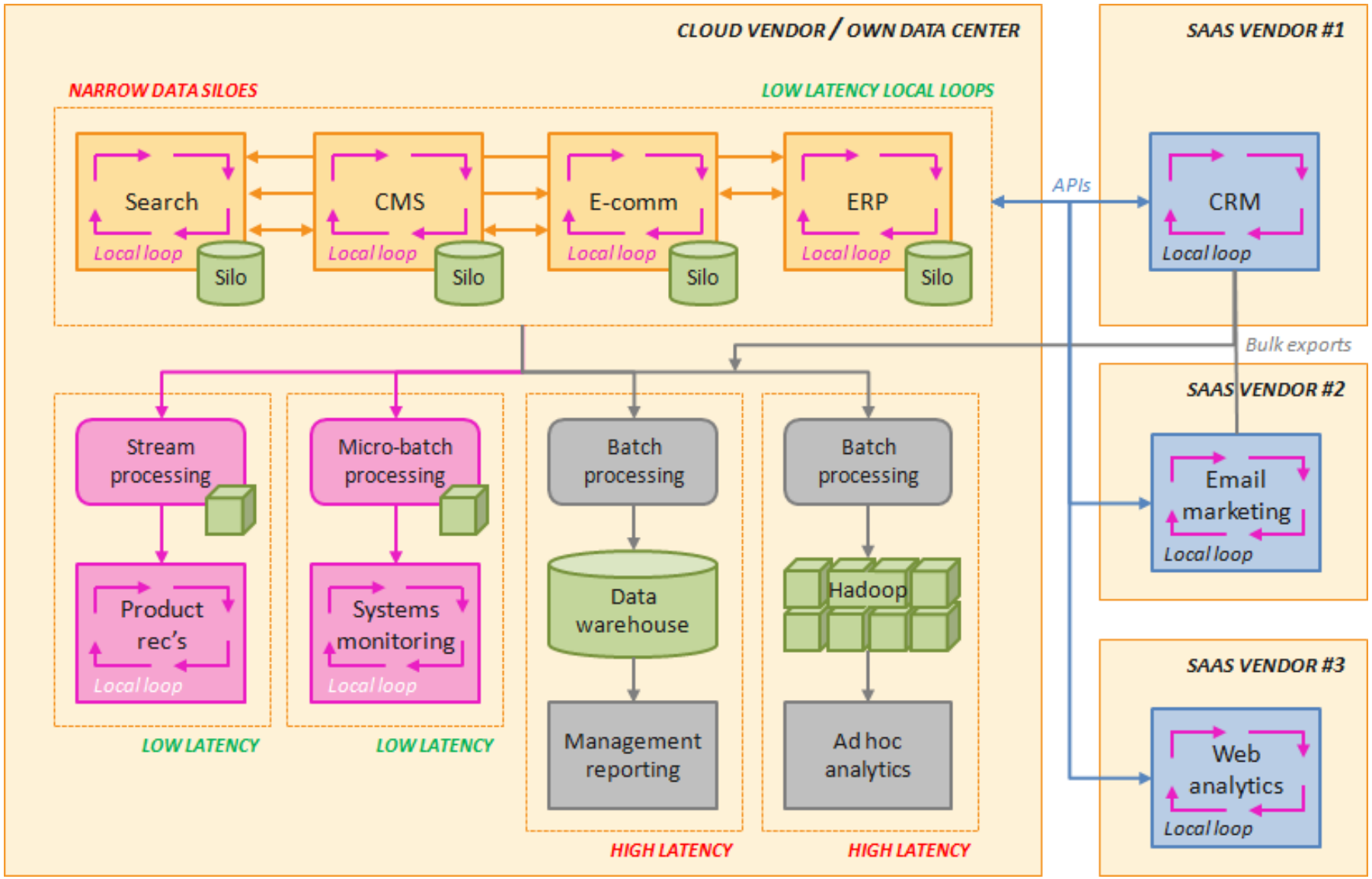
CARD.COM



Source: SnowPlow

The Hybrid Era

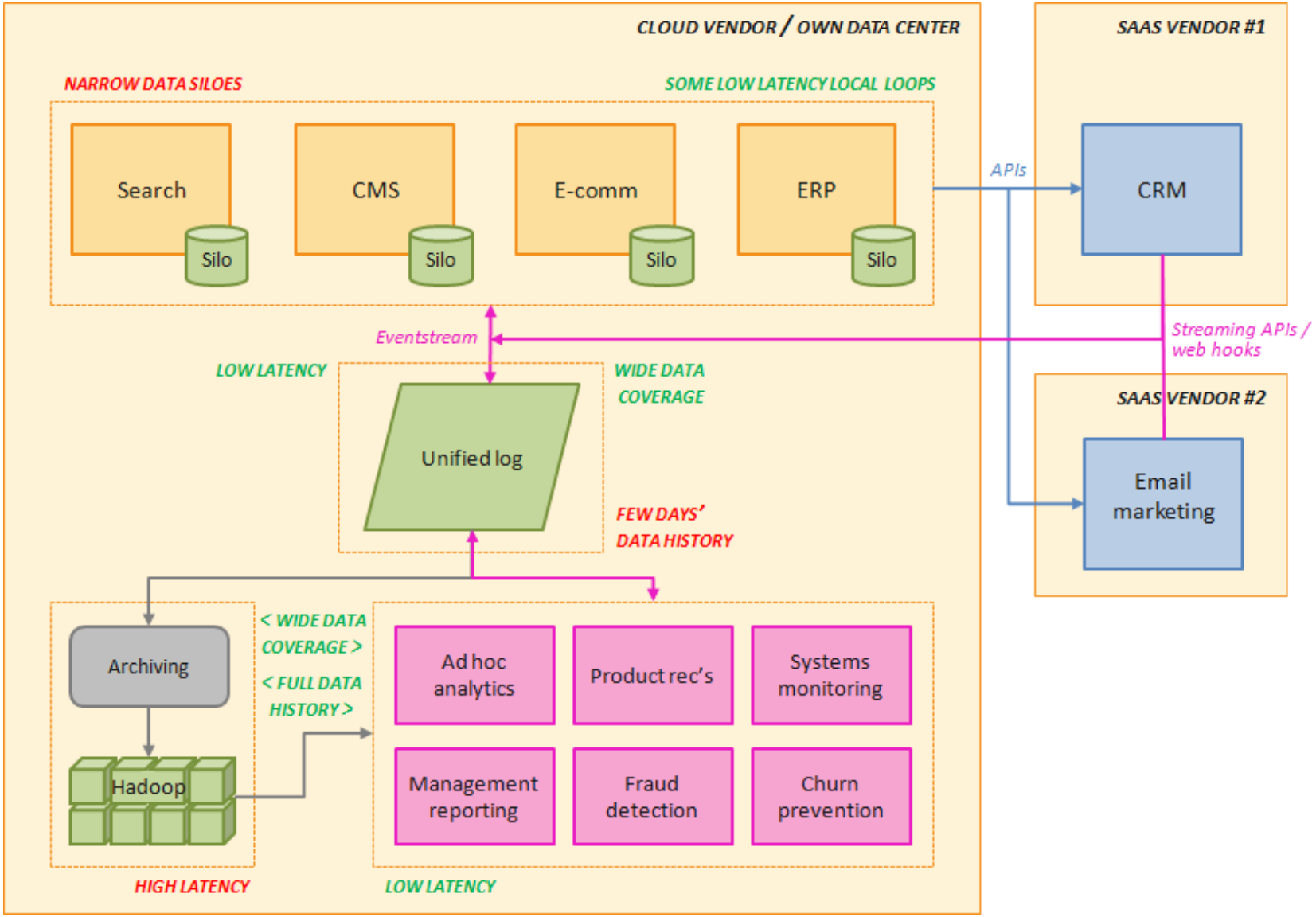
CARD.COM



Source: SnowPlow

The Unified Era

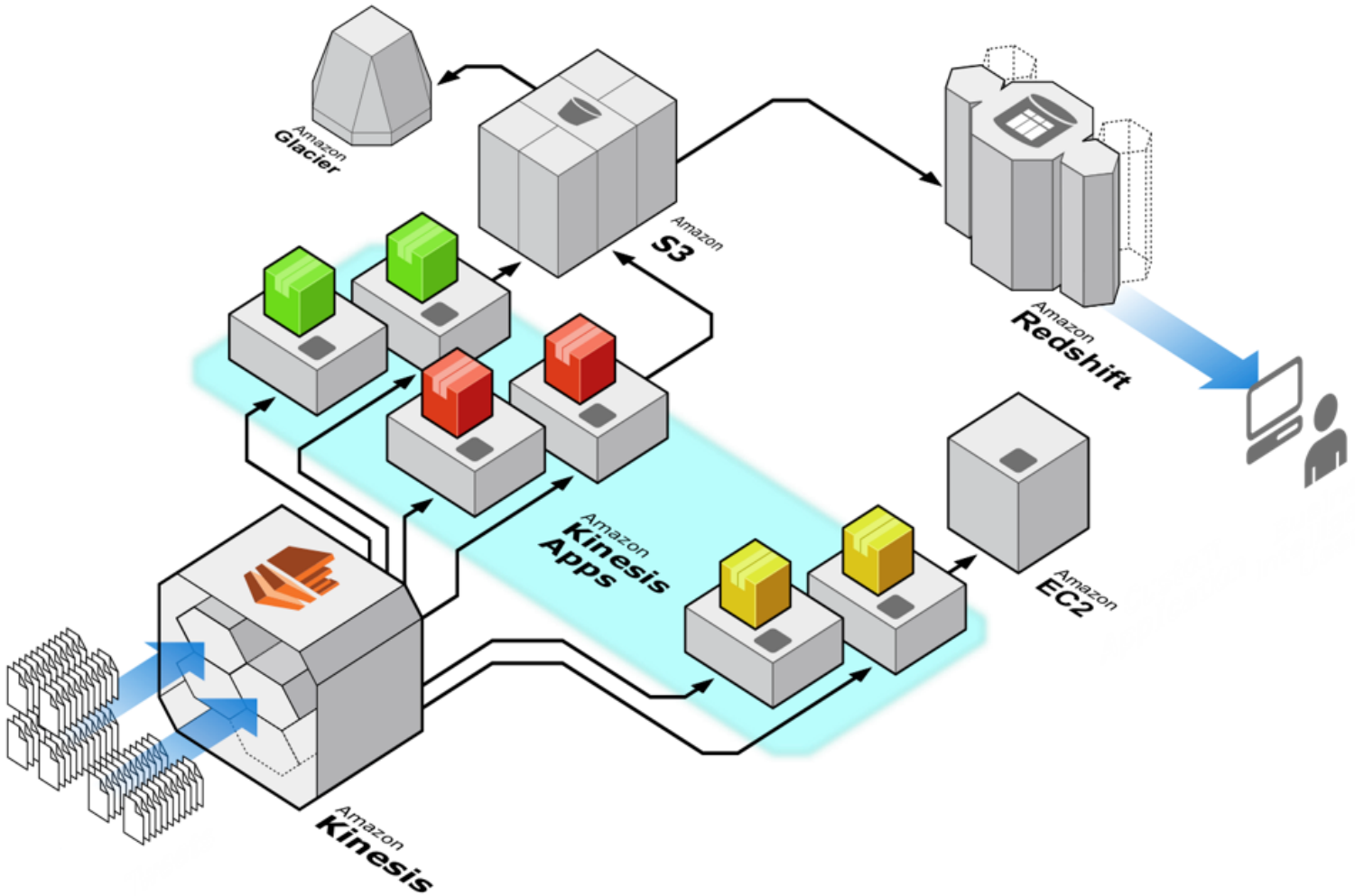
CARD.COM



Source: SnowPlow

Why Amazon Kinesis & Redshift?

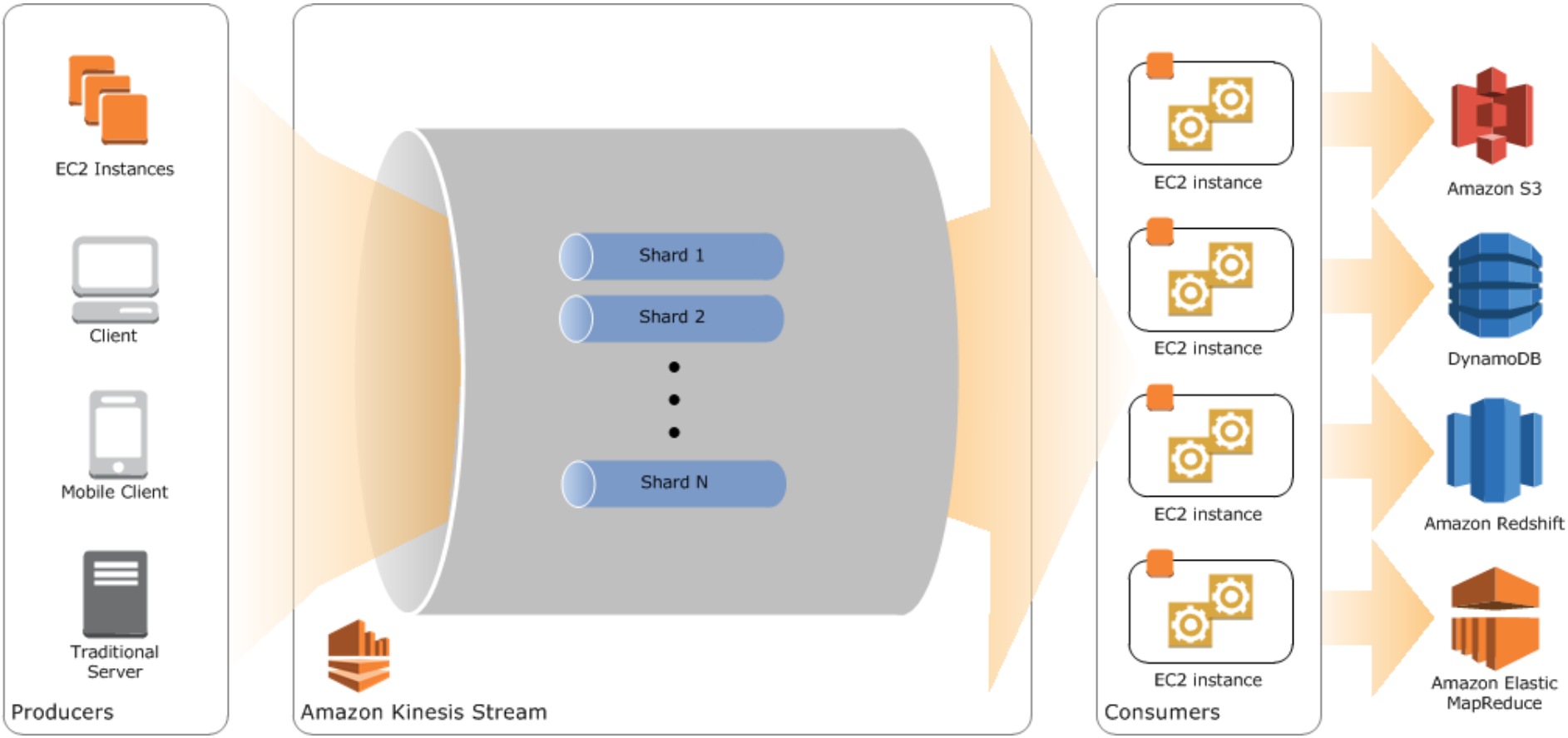
CARD.COM



Source: Kinesis Product Details

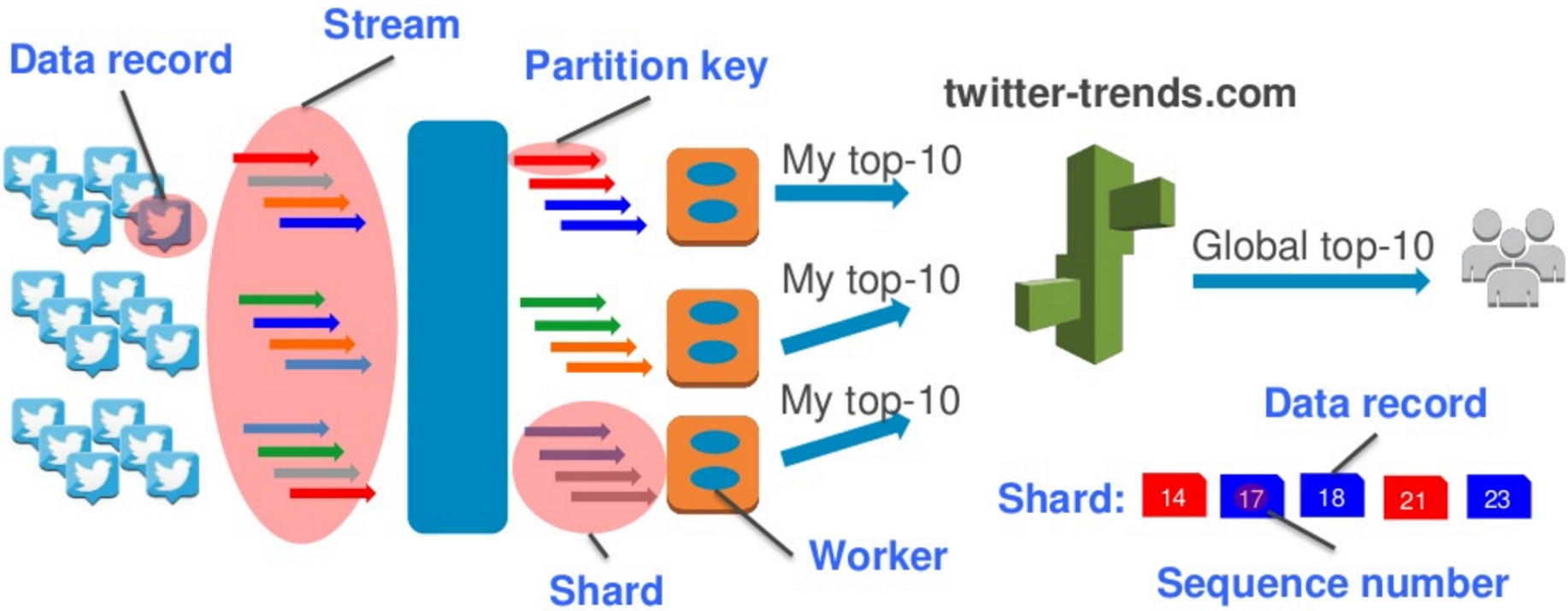
Intro to Amazon Kinesis Streams

CARD.COM



Source: Kinesis Developer Guide

Intro to Amazon Kinesis Shards



Source: AWS re:Invent 2013

How to Communicate with Kinesis

CARD.COM

Writing data to the stream:

- Amazon Kinesis Streams API (!)
- Amazon Kinesis Producer Library (KPL) from Java
- flume-kinesis
- Amazon Kinesis Agent

Reading data from the stream:

- Amazon Kinesis Streams API (!)
- Amazon Kinesis Client Library (KCL) from Java, Node.js, .NET, Python, Ruby

Managing streams:

- Amazon Kinesis Streams API (!)

Now We Need an R Client!

CARD.COM

Initialize connection to the SDK:

```
> library(rJava)
+ .jinit(classpath =
+         list.files(
+             '~/Projects/kineric/inst/java/',
+             full.names = TRUE),
+         parameters = "-Xmx512m")
```

What methods do we have there?

```
> kc <- .jnew('com.amazonaws.services.kinesis.AmazonKinesisClient')
> .jmethods(kc)
[1]  "public void com.amazonaws.services.kinesis.AmazonKinesisClient.splitShard(jav
[2]  "public void com.amazonaws.services.kinesis.AmazonKinesisClient.splitShard(com
[3]  "public void com.amazonaws.services.kinesis.AmazonKinesisClient.removeTagsFrom
[4]  "public com.amazonaws.services.kinesis.model.ListStreamsResult com.amazonaws.s
[5]  "public com.amazonaws.services.kinesis.model.ListStreamsResult com.amazonaws.s
[6]  "public com.amazonaws.services.kinesis.model.ListStreamsResult com.amazonaws.s
[7]  "public com.amazonaws.services.kinesis.model.ListStreamsResult com.amazonaws.s
[8]  "public com.amazonaws.services.kinesis.model.GetShardIteratorResult com.amazon
[9]  "public com.amazonaws.services.kinesis.model.GetShardIteratorResult com.amazon
[10] "public com.amazonaws.services.kinesis.model.GetShardIteratorResult com.amazon
```

CARD.COM

Set API endpoint:

```
> kc$setEndpoint('kinesis.us-west-2.amazonaws.com', 'kinesis', 'us-west-2')
```

What streams do we have access to?

```
> kc$listStreams()
[1] "Java-Object{{StreamNames: [test_kinesis],HasMoreStreams: false}}"
> kc$listStreams()$getStreamNames()
[1] "Java-Object{[test_kinesis]}"
```

Describe this stream:

```
> (kcstream <- kc$describeStream(StreamName = 'test_kinesis')$getStreamDescription()
[1] "Java-Object{{StreamName: test_kinesis,StreamARN: arn:aws:kinesis:us-west-2:595
> kcstream$getStreamName()
[1] "test_kinesis"
> kcstream$getStreamStatus()
[1] "ACTIVE"
> kcstream$getShards()
[1] "Java-Object{[
{ShardId: shardId-000000000000,HashKeyRange: {
StartingHashKey: 0,EndingHashKey: 170141183460469231731687303715884105723 24},Sequen
{ShardId: shardId-000000000001,HashKeyRange: {StartingHashKey: 170141183460469231 73
```

Writing to the Stream

CARD.COM

```
> prr <- .jnew('com.amazonaws.services.kinesis.model.PutRecordRequest')
> prr$setStreamName('test_kinesis')
> prr$setData(J('java.nio.ByteBuffer')$wrap(.jbyte(charToRaw('foobar'))))
> prr$setPartitionKey('42')
> prr
[1] "Java-Object{{StreamName: test_kinesis,
     Data: java.nio.HeapByteBuffer[pos=0 lim=6 cap=6],
     PartitionKey: 42,}}"
> kc$putRecord(prr)
[1] "Java-Object{{ShardId: shardId-000000000001,
    SequenceNumber: 49562894160449444332153346371084313572324361665031176210}}"
```

Get Records from the Stream

CARD.COM

First, we need a shared iterator, which expires in 5 minutes:

```
> sir <- .jnew('com.amazonaws.services.kinesis.model.GetShardIteratorRequest')
> sir$setStreamName('test_kinesis')
> sir$setShardId(.jnew('java/lang/String', '0'))
> sir$setShardIteratorType('TRIM_HORIZON')
> iterator <- kc$getShardIterator(sir)$getShardIterator()
```

Then we can use it to get records:

```
> gir <- .jnew('com.amazonaws.services.kinesis.model.GetRecordsRequest')
> gir$setShardIterator(iterator)
> kc$getRecords(gir)$getRecords()
[1] "Java-Object{[]}"
```

Get Records from the Stream

CARD.COM

First, we need a shared iterator, which expires in 5 minutes:

```
> sir <- .jnew('com.amazonaws.services.kinesis.model.GetShardIteratorRequest')
> sir$setStreamName('test_kinesis')
> sir$setShardId(.jnew('java/lang/String', '0'))
> sir$setShardIteratorType('TRIM_HORIZON')
> iterator <- kc$getShardIterator(sir)$getShardIterator()
```

Then we can use it to get records:

```
> gir <- .jnew('com.amazonaws.services.kinesis.model.GetRecordsRequest')
> gir$setShardIterator(iterator)
> kc$getRecords(gir)$getRecords()
[1] "Java-Object{[]}"
```

The data was pushed to the other shard:

```
> sir$setShardId(.jnew('java/lang/String', '1'))
> iterator <- kc$getShardIterator(sir)$getShardIterator()
> gir$setShardIterator(iterator)
> kc$getRecords(gir)$getRecords()
[1] "Java-Object{[{SequenceNumber: 49562894160449444332153346371084313572324361665
ApproximateArrivalTimestamp: Tue Jun 14 09:40:19 CEST 2016,
Data: java.nio.HeapByteBuffer[pos=0 lim=6 cap=6],
PartitionKey: 42}]}"
```

Get Further Records from the Stream

CARD.COM

```
> sapply(kc$getRecords(gir)$getRecords(),
+        function(x)
+        rawToChar(x$getData()$array()))
[1] "foobar"
> iterator <- kc$getRecords(gir)$getNextShardIterator()
> gir$setShardIterator(iterator)
> kc$getRecords(gir)$getRecords()
[1] "Java-Object{[]}"
```

Get the oldest available data again:

```
> iterator <- kc$getShardIterator(sir)$getShardIterator()
> gir$setShardIterator(iterator)
> kc$getRecords(gir)$getRecords()
[1] "Java-Object{[{SequenceNumber: 49562894160449444332153346371084313572324361665
```

Or we can refer to the last known sequence number as well:

```
> sir$setShardIteratorType('AT_SEQUENCE_NUMBER')
> sir$setStartingSequenceNumber('49562894160449444332153346371084313572324361665031
> iterator <- kc$getShardIterator(sir)$getShardIterator()
> gir$setShardIterator(iterator)
> kc$getRecords(gir)$getRecords()
[1] "Java-Object{[{SequenceNumber: 49562894160449444332153346371084313572324361665
```

Managing Shards

CARD.COM

No need for both shards:

```
> ms <- .jnew('com.amazonaws.services.kinesis.model.MergeShardsRequest')
> ms$setShardToMerge('shardId-000000000000')
> ms$setAdjacentShardToMerge('shardId-000000000001')
> ms$setStreamName('test_kinesis')
> kc$mergeShards(ms)
```

What do we have now?

```
> kc$describeStream(StreamName = 'test_kinesis')$getStreamDescription()$getShards()
[1] "Java-Object{[
{ShardId: shardId-000000000000,HashKeyRange: {StartingHashKey: 0,EndingHashKey: 170
SequenceNumberRange: {
StartingSequenceNumber: 49562894160427143586954815717376297430913467927668719618,
EndingSequenceNumber: 49562894160438293959554081028945856364232263390243848194}},
{ShardId: shardId-000000000001,HashKeyRange: {StartingHashKey: 170141183460469231731
SequenceNumberRange: {
StartingSequenceNumber: 49562894160449444332153346340517833149186116289174700050,
EndingSequenceNumber: 49562894160460594704752611652087392082504911751749828626}},
{ShardId: shardId-000000000002,
ParentShardId: shardId-000000000000,
AdjacentParentShardId: shardId-000000000001,
HashKeyRange: {StartingHashKey: 0,EndingHashKey: 340282366920938463463374607431768
SequenceNumberRange: {StartingSequenceNumber: 49562904991497673099704924344727019521
```

Next Steps

CARD.COM

Ideas:

- R function managing/scaling shards and attaching R processors
- One R processor per shard (parallel threads on a node or cluster)
- Store started/finished sequence number in memory/DB (checkpointing)

Example (Design for a simple MVP on a single node)

mcparallel starts parallel R processes to evaluate the given expression

Next Steps

CARD.COM

Ideas:

- R function managing/scaling shards and attaching R processors
- One R processor per shard (parallel threads on a node or cluster)
- Store started/finished sequence number in memory/DB (checkpointing)

Example (Design for a simple MVP on a single node)

mcparallel starts parallel R processes to evaluate the given expression

Problems:

- duplicated records after failure due to in-memory checkpoint
- needs a DB for running on a cluster
- hard to write good unit tests

Next Steps

CARD.COM

Ideas:

- R function managing/scaling shards and attaching R processors
- One R processor per shard (parallel threads on a node or cluster)
- Store started/finished sequence number in memory/DB (checkpointing)

Example (Design for a simple MVP on a single node)

mcparallel starts parallel R processes to evaluate the given expression

Problems:

- duplicated records after failure due to in-memory checkpoint
- needs a DB for running on a cluster
- hard to write good unit tests

Why not use KCL then?

Amazon Kinesis Client Library

CARD.COM

- An easy-to-use programming model for processing data

```
java -cp amazon-kinesis-client-1.6.1.jar \
  com.amazonaws.services.kinesis.multilang.MultiLangDaemon \
  test-kinesis.properties
```

- Scale-out and fault-tolerant processing (checkpointing via DynamoDB)
- Logging and metrics in CloudWatch
- The MultiLangDaemon spawns processes written in any language, communication happens via JSON messages sent over stdin/stdout
- Only a few events/methods to care about in the consumer application:
  1. initialize
  2. processRecords
  3. checkpoint
  4. shutdown

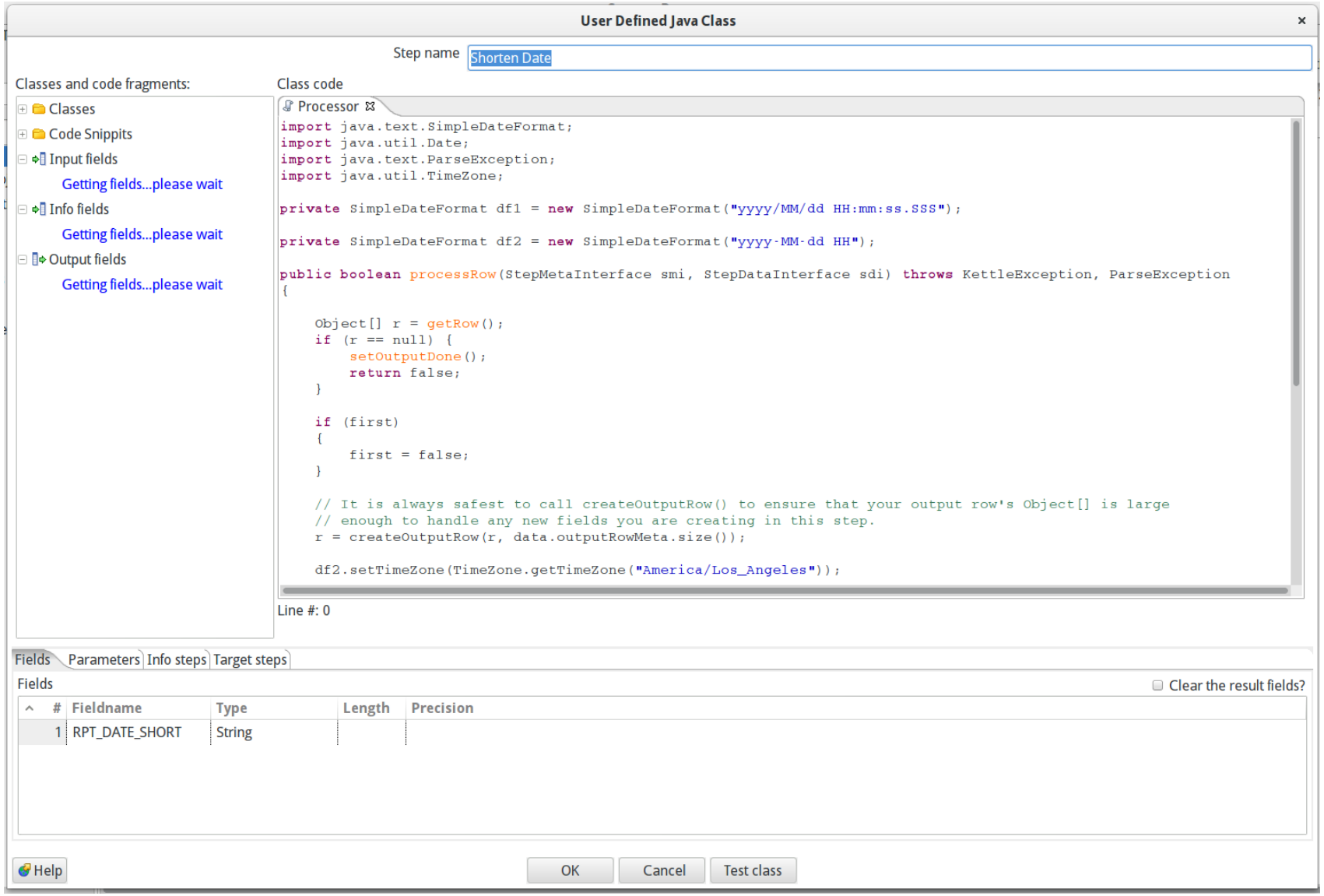
Messages from the KCL

CARD.COM

1. initialize:
   - Perform initialization steps
   - Write "status" message to indicate you are done
   - Begin reading line from STDIN to receive next action
2. processRecords:
   - Perform processing tasks (you may write a checkpoint message at any time)
   - Write "status" message to STDOUT to indicate you are done.
   - Begin reading line from STDIN to receive next action
3. shutdown:
   - Perform shutdown tasks (you may write a checkpoint message at any time)
   - Write "status" message to STDOUT to indicate you are done.
   - Begin reading line from STDIN to receive next action
4. checkpoint:
   - Decide whether to checkpoint again based on whether there is an error or not.

Again: Why R?

CARD.COM



R Script Interacting with KCL

CARD.COM

```
#!/usr/bin/r -i

while (TRUE) {

    ## read and parse messages
    line <- fromJSON(readLines(n = 1))

    ## nothing to do unless we receive a record to process
    if (line$action == 'processRecords') {

        ## process each record
        lapply(line$records, function(r) {

            business_logic(r)
            cat(toJSON(list(action = 'checkpoint', checkpoint = r$sequenceNumber)))

        })
    }

    ## return response in JSON
    cat(toJSON(list(action = 'status', responseFor = line$action)))

}
```

Examples

CARD.COM