Apache Spark as a Platform for Powerful Custom Analytics Data Pipeline

Misha Chernetsov, Grammarly

Apache Spark Meetup, SF

April 27, 2017

About Me: Misha Chernetsov

@chernetsov

Data Team Lead @ Grammarly

Kindle @ Amazon

Building Analytics Pipelines (5 years)

Coding on JVM (12 years), Scala + Spark (3 years)

Analytics @ Consumer Product Company

Tool that helps us better understand:

- Who are our users?
- How do they interact with the product?
- How do they get in, engage, pay, and how long do they stay?

Analytics @ Consumer Product Company

We want our decisions to be

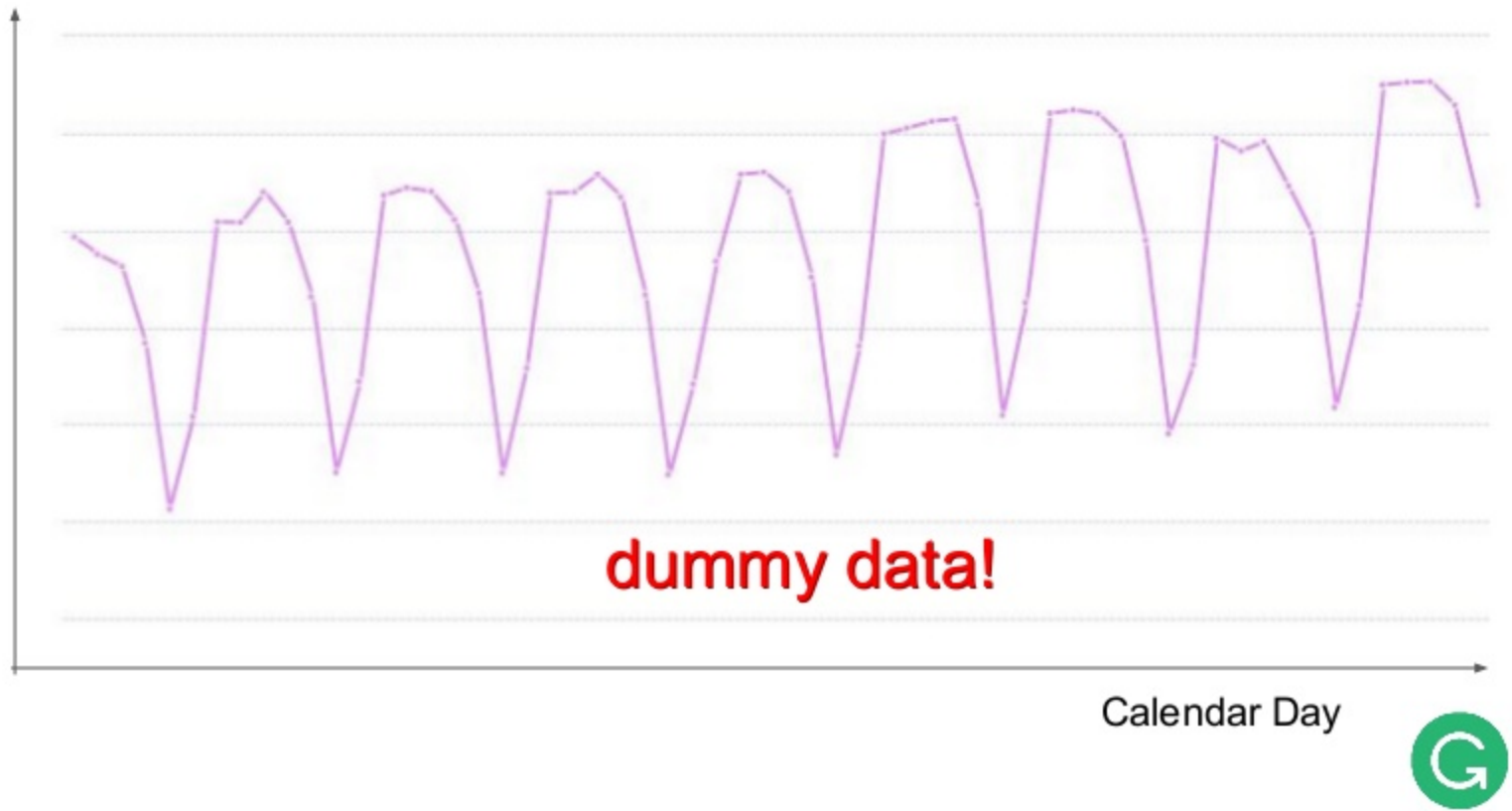
data-driven

Everyone: product managers, marketing, engineers, support...

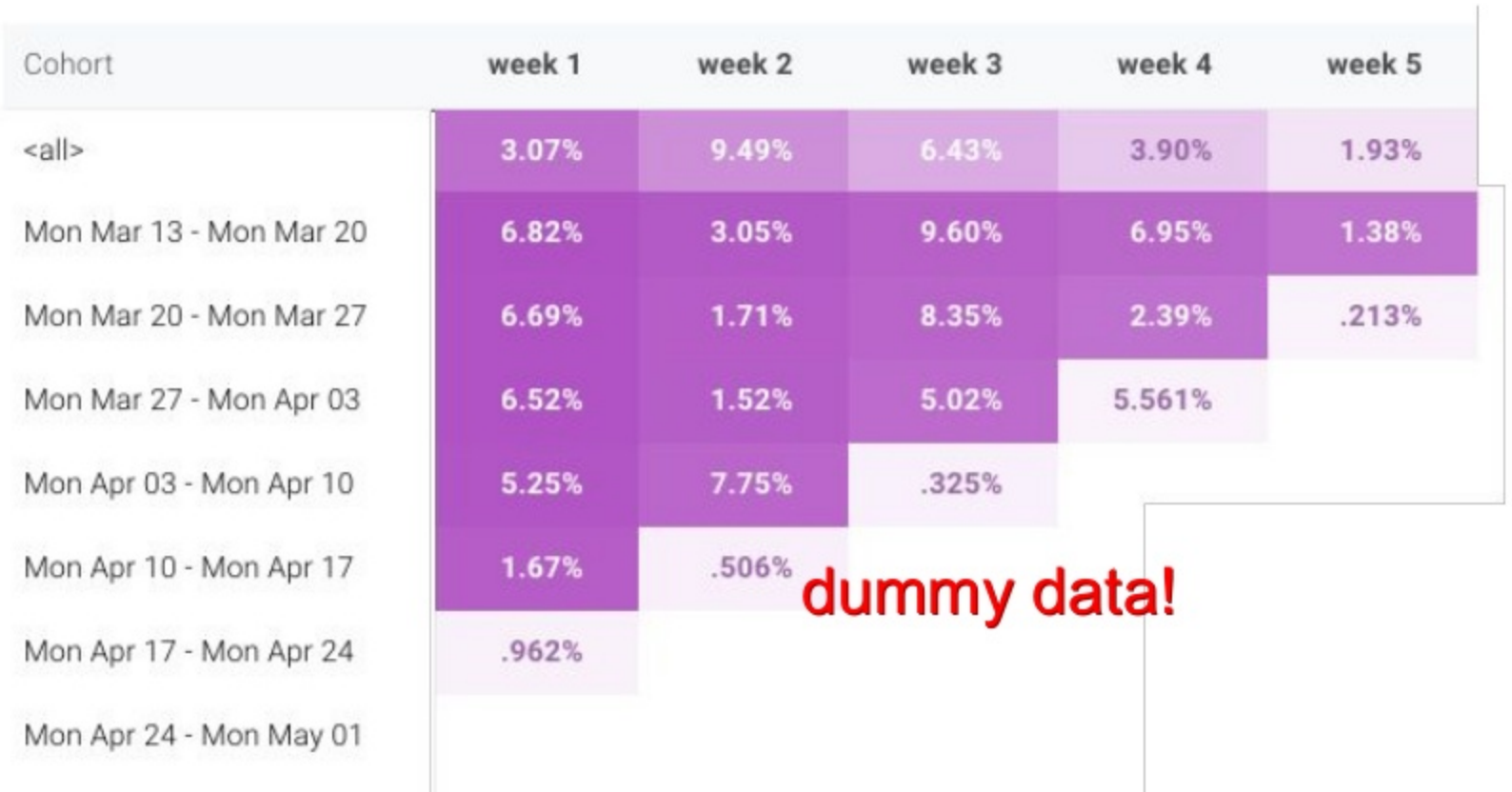
Analytics @ Consumer Product Company

Example Report 1 – Daily Active Users

Number of unique active users by day



Example Report 2 – Comparison of Cohort Retention Over Time



Cohort	week 1	week 2	week 3	week 4	week 5
<all>	3.07%	9.49%	6.43%	3.90%	1.93%
Mon Mar 13 - Mon Mar 20	6.82%	3.05%	9.60%	6.95%	1.38%
Mon Mar 20 - Mon Mar 27	6.69%	1.71%	8.35%	2.39%	.213%
Mon Mar 27 - Mon Apr 03	6.52%	1.52%	5.02%	5.561%	
Mon Apr 03 - Mon Apr 10	5.25%	7.75%	.325%		
Mon Apr 10 - Mon Apr 17	1.67%	.506%			
Mon Apr 17 - Mon Apr 24	.962%				
Mon Apr 24 - Mon May 01					

dummy data!

Example Report 3 – Funnel Analysis

Step 1. Users who visited a certain landing page

Step 2. Users who installed a product after Step 1.

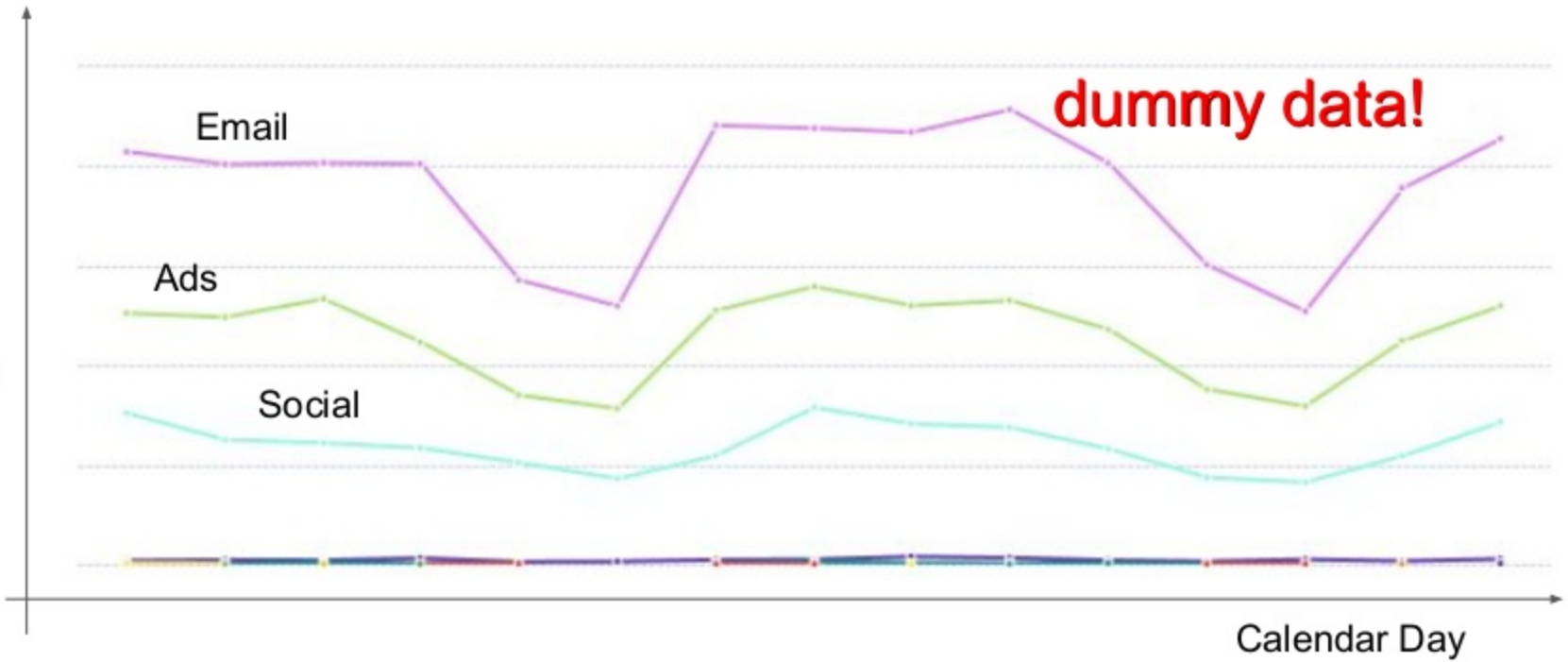
dummy data!

Step 3. Users who created an account after Step 2 and Step 1.

Example Report 4 – Payer Conversions By Traffic Source

Number of users who bought a subscription.

Split by traffic source type (where user came from)



Example: Data

- Landing page visit 
  - URL with UTM tags
  - Referrer

- Subscription purchased 
  - Is first in subscription

Example: Data

```
{
  "eventName": "page-visit",
  "url": "...?utm_medium=ad",
  …
}
```

```
{
  "eventName": "subscribe",
  "period": "12 months",
  …
}
```

Everything is an Event

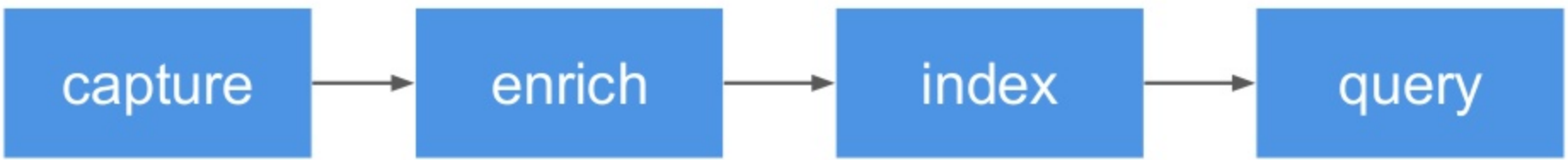
Example: Data

Slice by

Plot

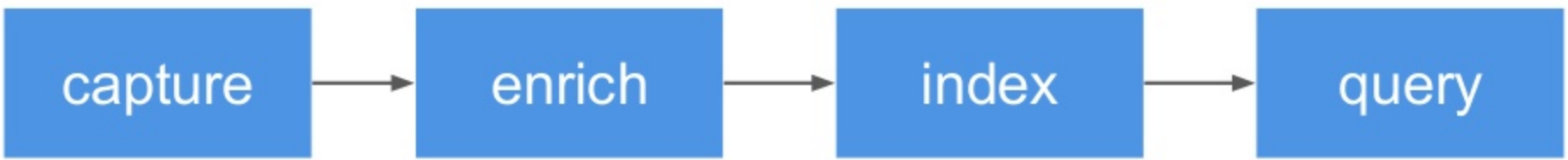
Enrich and/or Join

Analytics @ Consumer Product Company



Use 3rd Party?

1. Integrated Event Analytics
2. UI over your DB

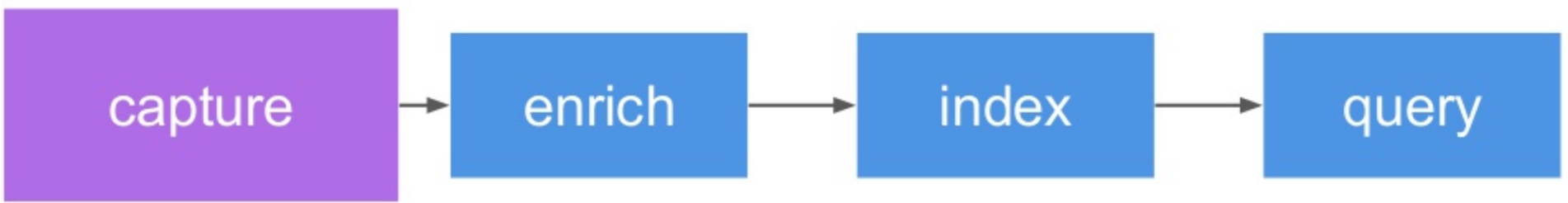


Pre-aggregation / enriching is still on you.

Reports are not tailored for your needs, limited capability.

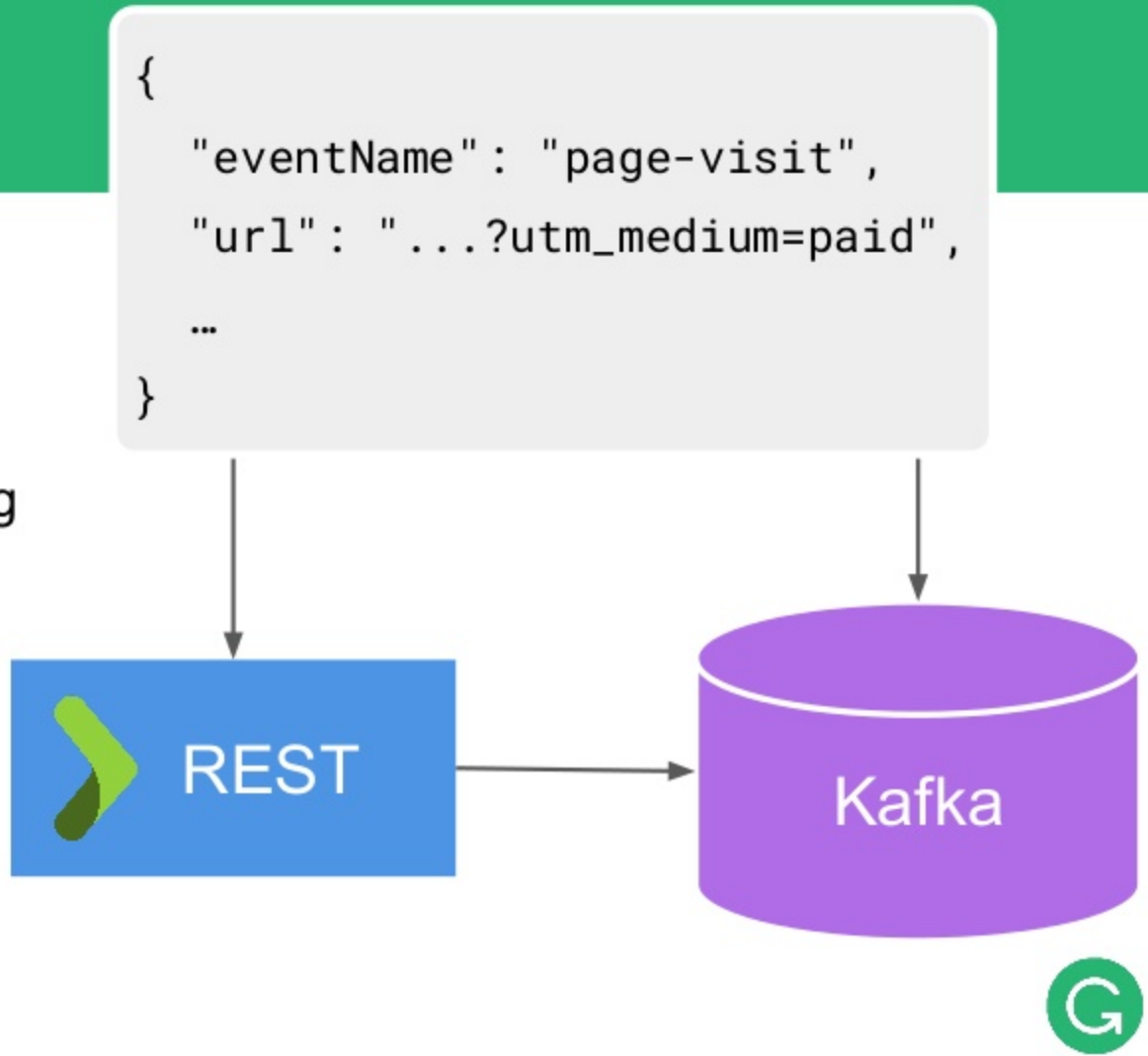
Hard to achieve accuracy and trust.

Build Step 1: Capture

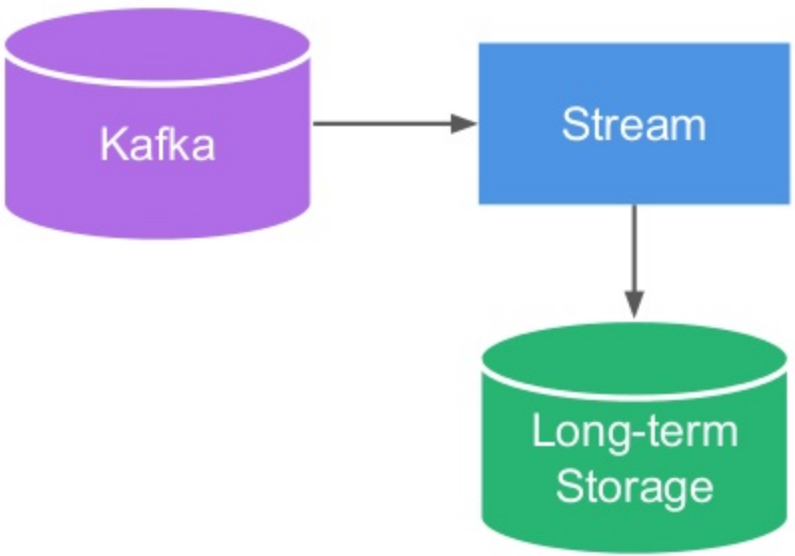


Capture

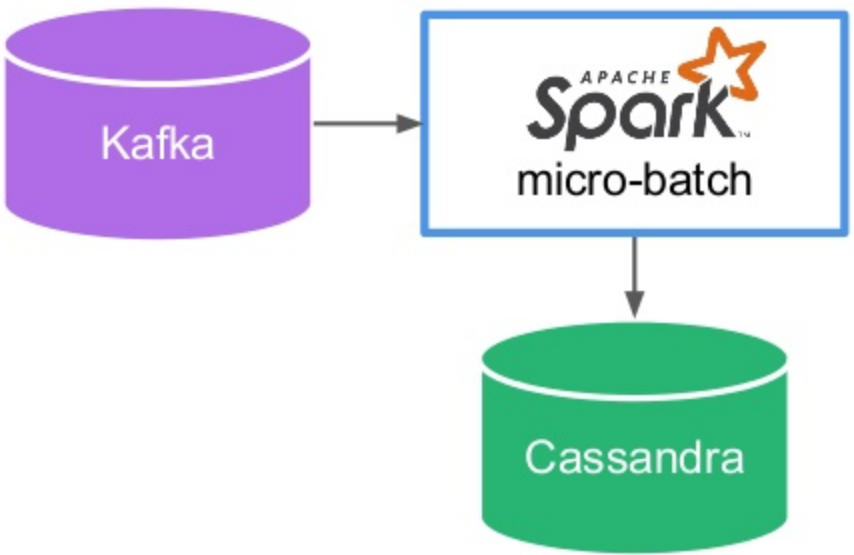
- Always up, resilient
- Spikes / back pressure
- Buffer for delayed processing



Save To Long-Term Storage

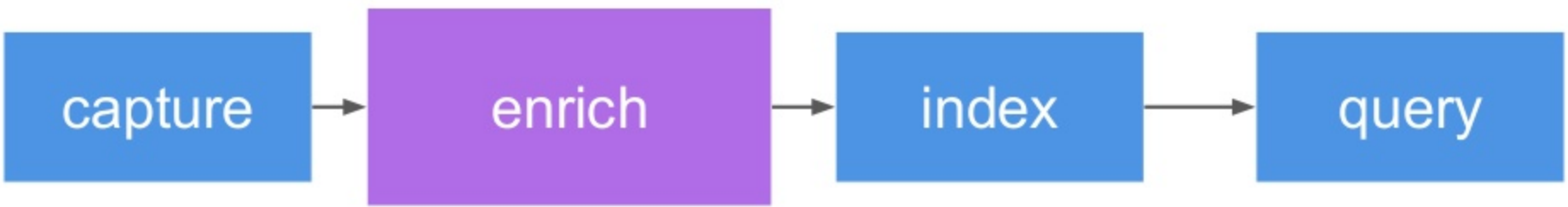


Save To Long-Term Storage




```
val rdd = KafkaUtils.createRDD[K, V](...)

rdd.saveToCassandra("raw")
```

Build Step 2: Enrich



Enrichment Phase 1: User Attribution

Enrichment Phase 1: User Attribution

```
{
  "eventName": "page-visit",
  "url": "...?utm_medium=ad",
  "fingerprint": "abc",
  …
}
```

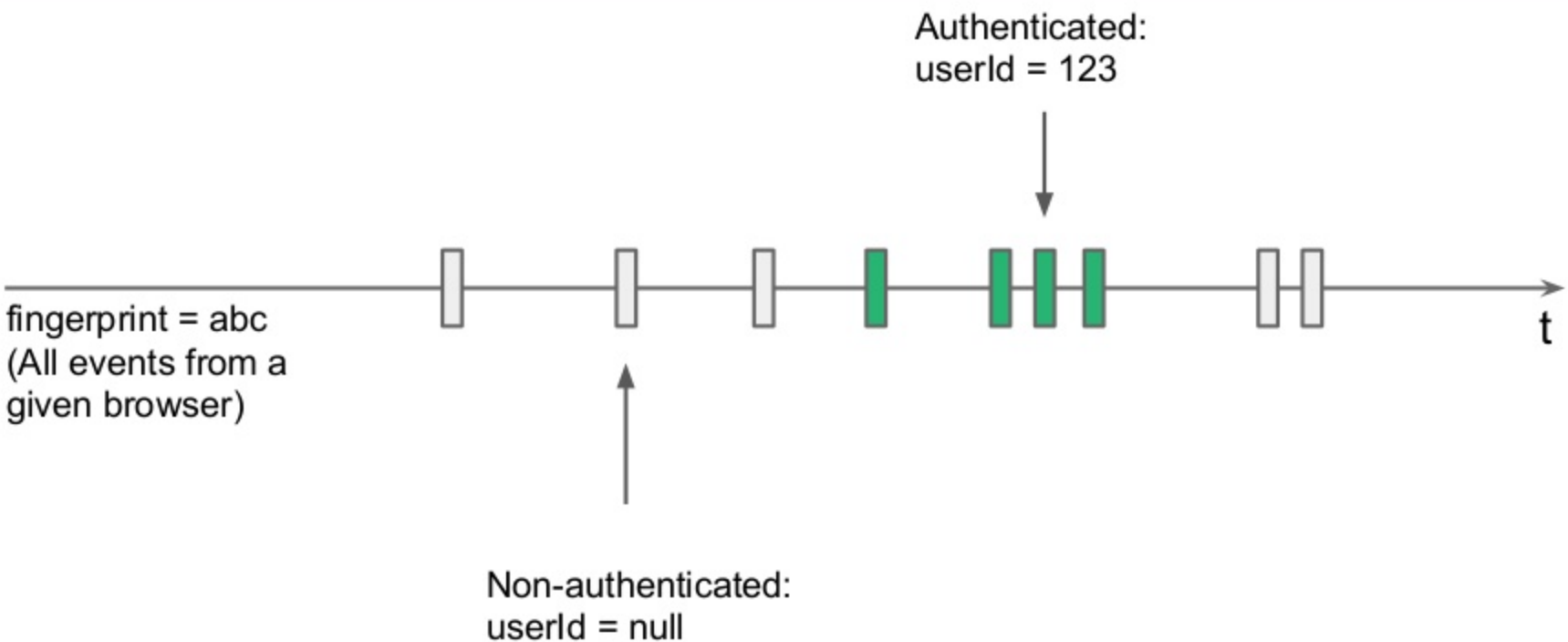
```
{
  "eventName": "subscribe",
  "userId": 123,
  "fingerprint": "abc",
  …
}
```

Enrichment Phase 1: User Attribution

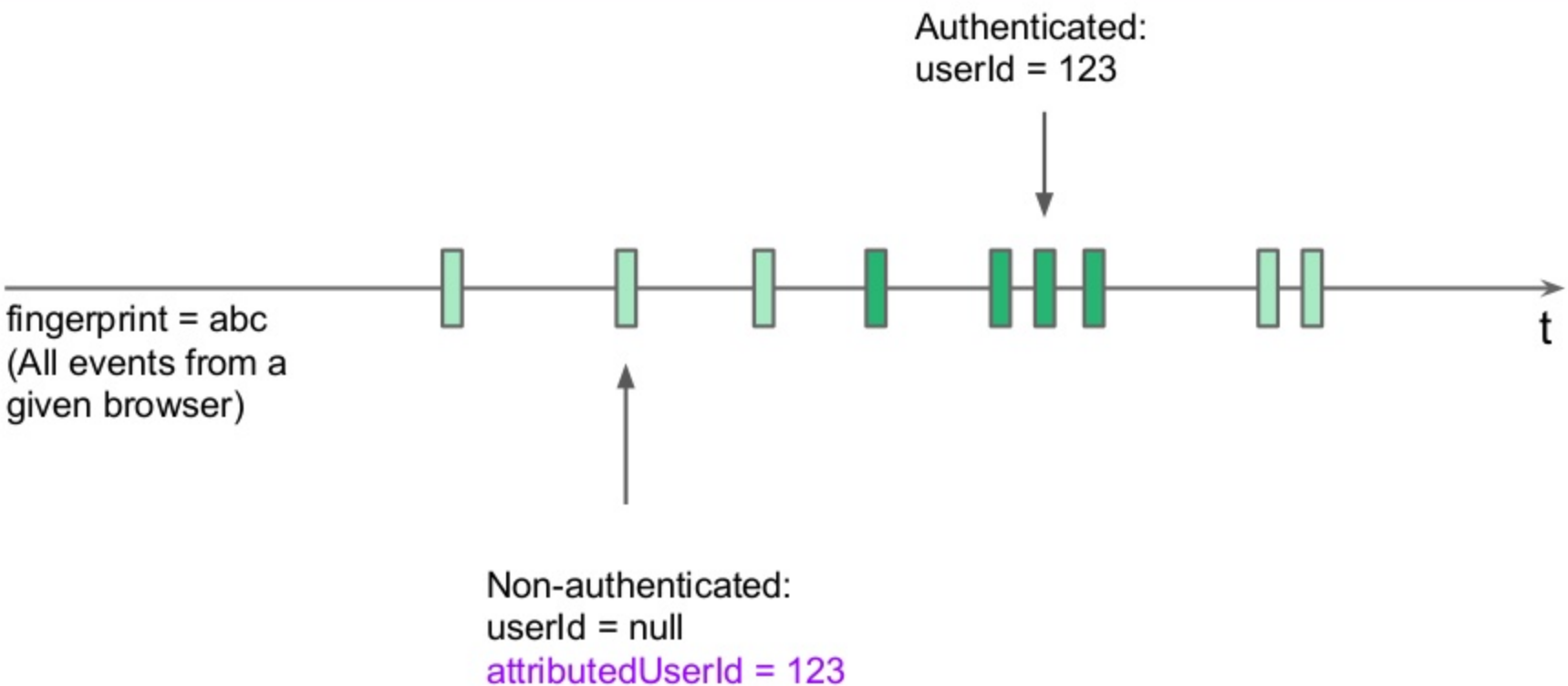
```
{
  "eventName": "page-visit",
  "url": "...?utm_medium=ad",
  "fingerprint": "abc",
  "attributedUserId": 123,
  …
}
```

```
{
  "eventName": "subscribe",
  "userId": 123,
  "fingerprint": "abc",
  …
}
```

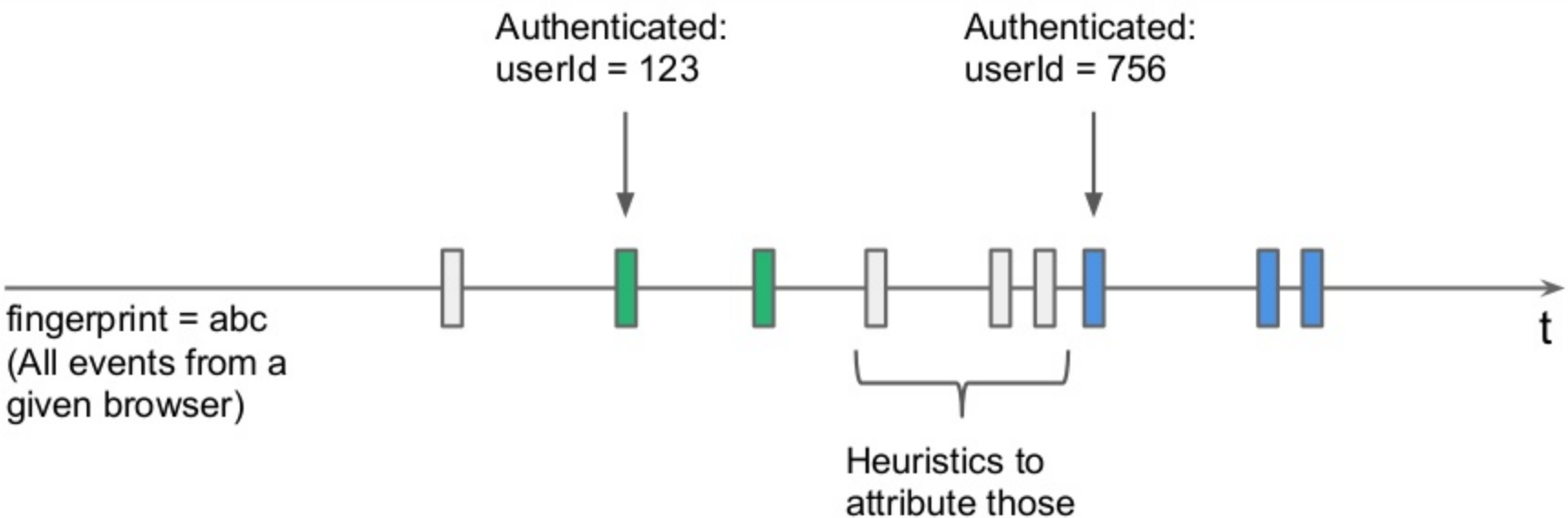
Enrichment Phase 1: User Attribution



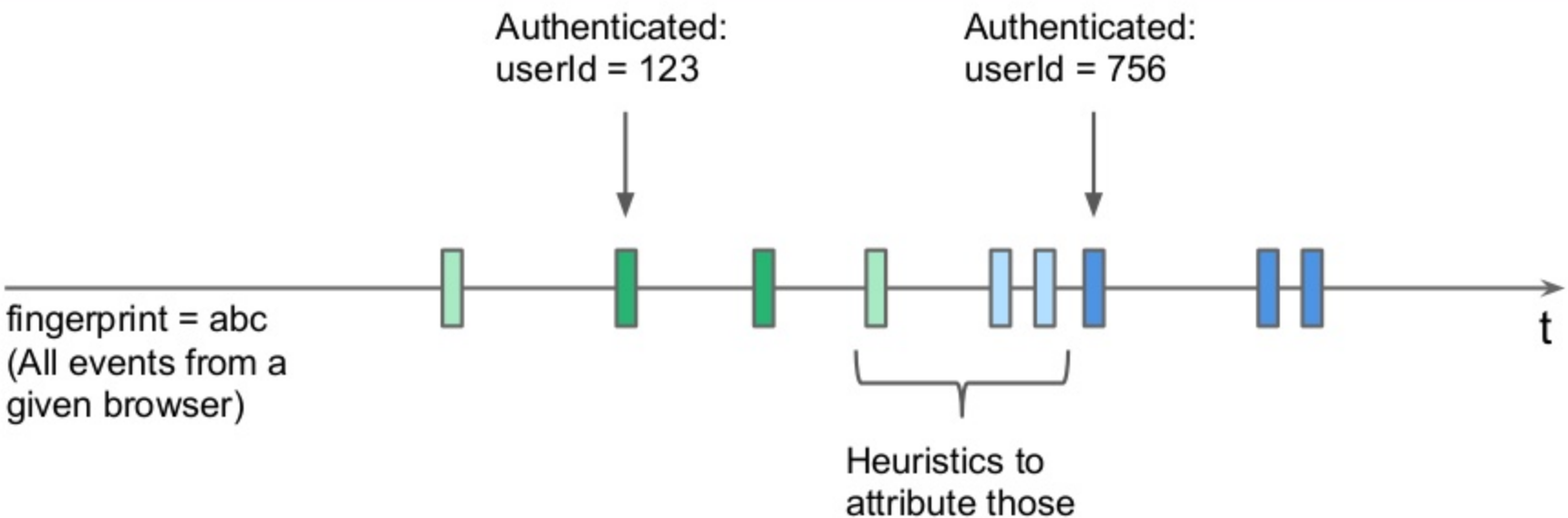
Enrichment Phase 1: User Attribution



Enrichment Phase 1: User Attribution



Enrichment Phase 1: User Attribution



Enrichment Phase 1: User Attribution

```
rdd.mapPartitions { iterator =>

}
```

Enrichment Phase 1: User Attribution

```
rdd.mapPartitions { iterator =>

    val buffer = new ArrayBuffer()









}
```

Enrichment Phase 1: User Attribution

```
rdd.mapPartitions { iterator =>

    val buffer = new ArrayBuffer()

    iterator
        .takeWhile(_.userId.isEmpty)
        .foreach(buffer.append)
    val userId = iterator.head.userId



}
```

Enrichment 1: User Attribution

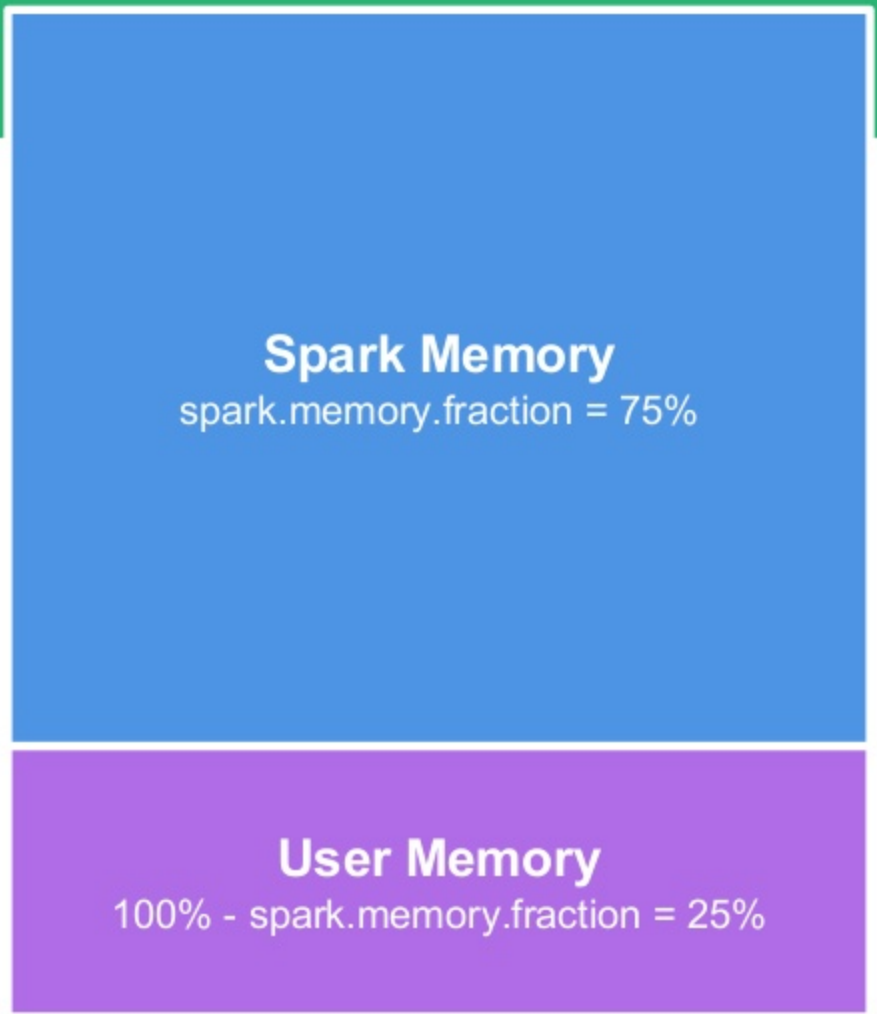
```
rdd.mapPartitions { iterator =>

    val buffer = new ArrayBuffer()

    iterator
        .takeWhile(_.userId.isEmpty)
        .foreach(buffer.append)
    val userId = iterator.head.userId

    buffer.map(_.setAttributedUserId(userId)) ++ iterator

}
```

Enrichment 1: User Attribution

```
rdd.mapPartitions { iterator =>

    val buffer = new ArrayBuffer()

    iterator
        .takeWhile(_.userId.isEmpty)
        .foreach(buffer.append)
    val userId = iterator.head.userId

    buffer.map(_.setAttributedUserId(userId)) ++ iterator

}
```

Can grow big and OOM your worker for outliers who use Grammarly without ever registering

Spark & Memory



Let's get into Spark Memory and use its safety features.

By default we should operate in User Memory (small fraction).

Spark Memory Manager & Spillable Collection

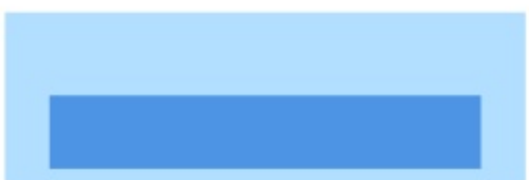
Memory

Disk

Spark Memory Manager & Spillable Collection

Memory

Disk



Spark Memory Manager & Spillable Collection

Memory

Disk

Spark Memory Manager & Spillable Collection

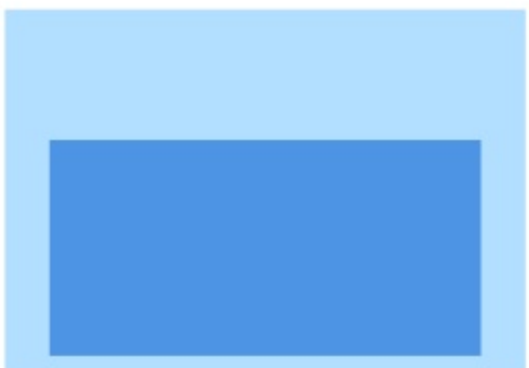
Memory

Disk

Spark Memory Manager & Spillable Collection

Memory

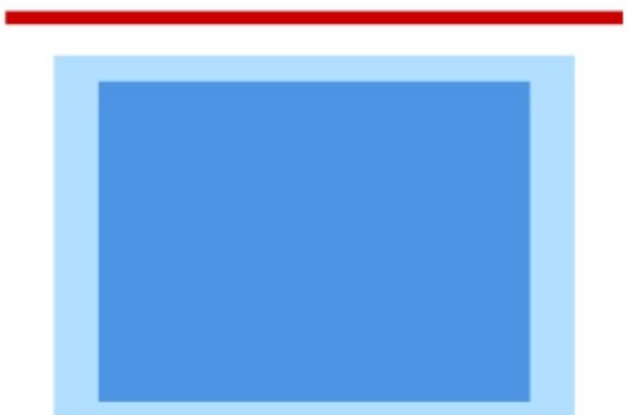
Disk



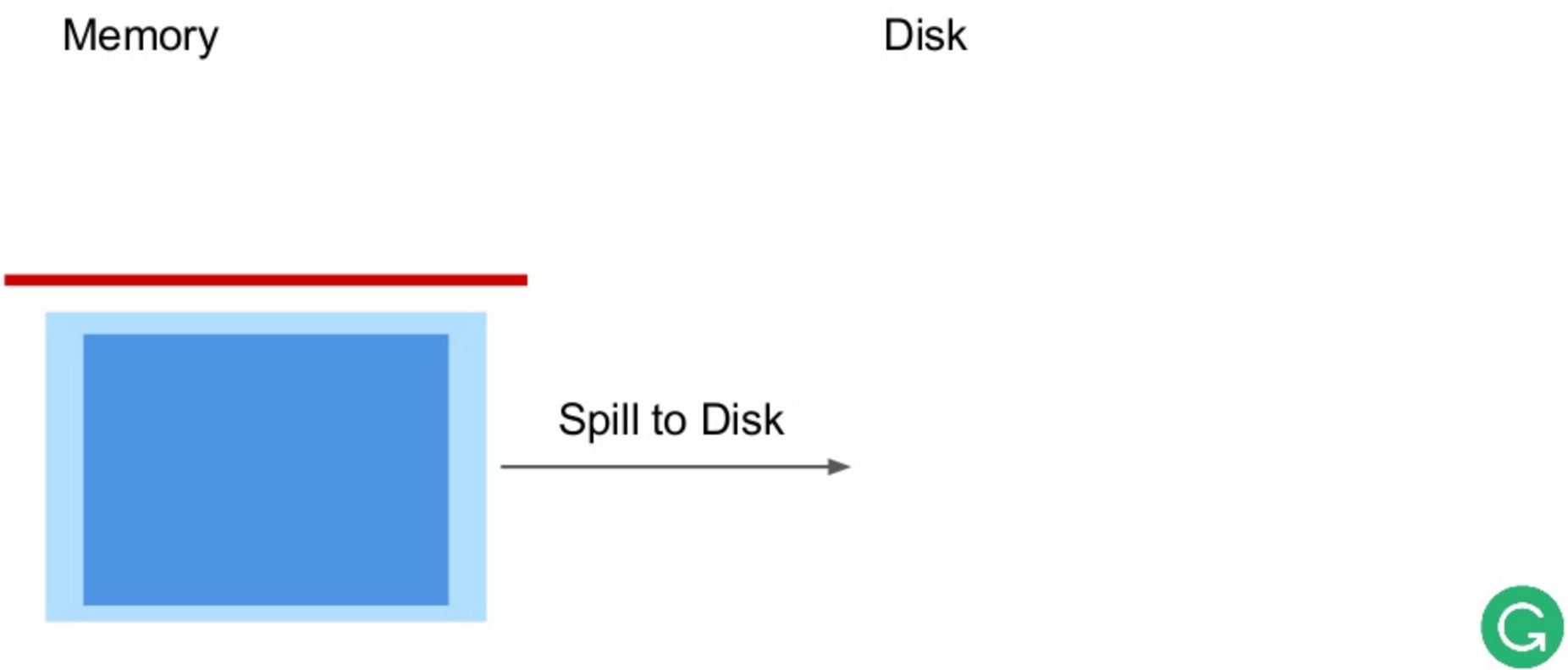
Spark Memory Manager & Spillable Collection

Memory

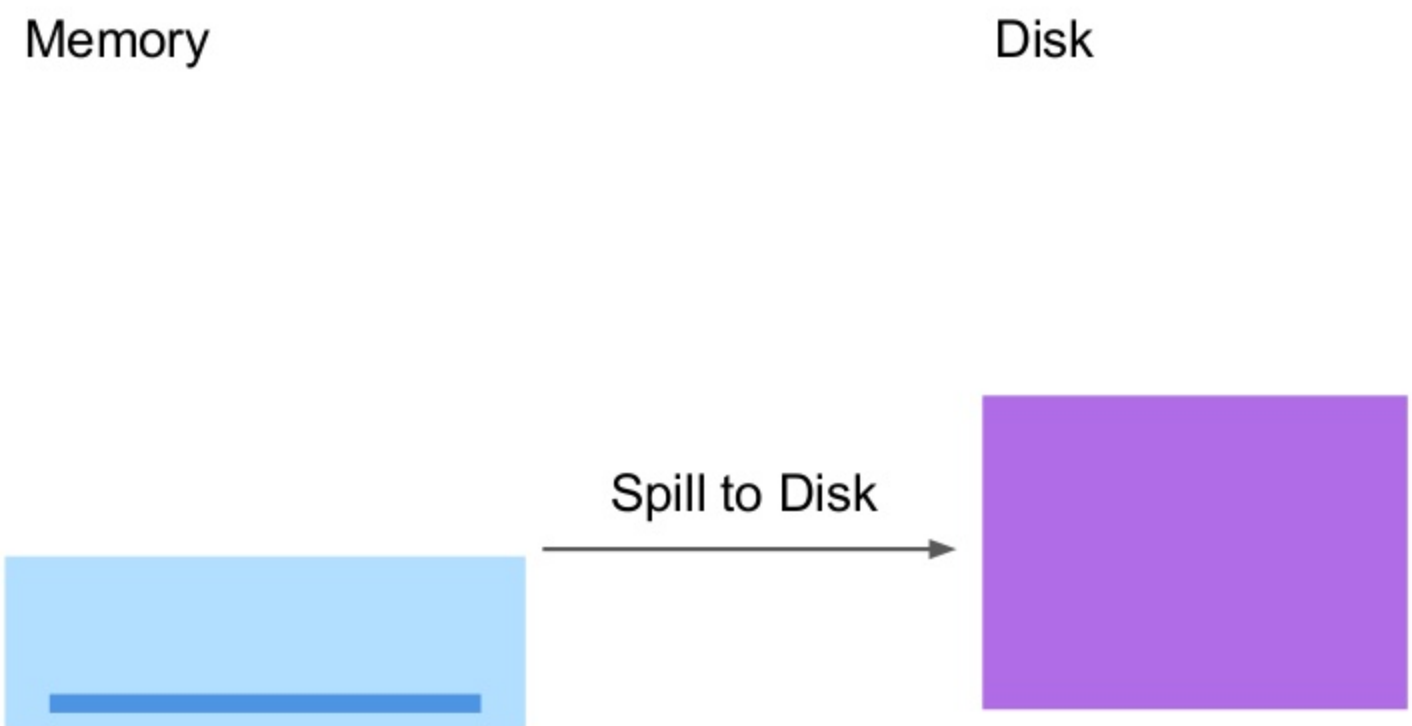
Disk



Spark Memory Manager & Spillable Collection

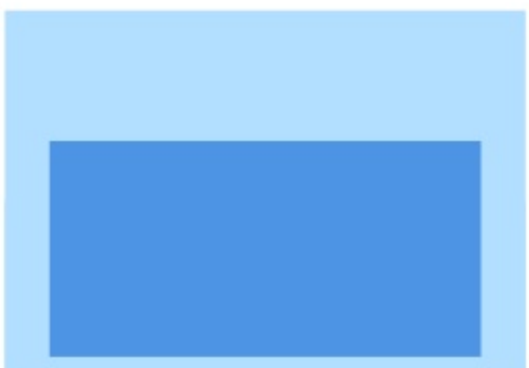


Spark Memory Manager & Spillable Collection



Spark Memory Manager & Spillable Collection

Memory



Disk

TaskMemoryManager

```
public long acquireExecutionMemory(long required, …)
```

```
public void releaseExecutionMemory(long size, …)
```

Spillable

```
trait Spillable {

    abstract def spill(inMemCollection: C): Unit

    def maybeSpill(currentMemory: Long, inMemCollection: C) {
        try x2 if needed
    }

}
```

call on every append to collection

SizeTracker

```
trait SizeTracker {

  def afterUpdate(): Unit = { … }

  def estimateSize(): Long = { … }

}
```

Call on every append.
Periodically estimates size
and saves samples.

Extrapolates

Custom Spillable Collection

- Get outside User Memory (25%), use Spark Memory (75%)
- Be safe with outliers
- Spark APIs: Could be a bit friendlier and high level

Enrichment Phase 1: User Attribution

```
rdd.mapPartitions { iterator =>

    val buffer = new SpillableBuffer()

    iterator
        .takeWhile(_.userId.isEmpty)
        .foreach(buffer.append)
    val userId = iterator.head.userId

    buffer.map(_.setAttributedUserId(userId)) ++ iterator

}
```

Can safely grow in mem while enough free Spark Mem. Spills to disk otherwise.

Enrichment Phase 2: Calculable Props

Enrichment Phase 2: Calculable Props

```
{
  "eventName": "page-visit",
  "url": "...?utm_medium=ad",
  "fingerprint": "abc",
  "attributedUserId": 123,
  …
}
```

```
{
  "eventName": "subscribe",
  "userId": 123,
  "fingerprint": "abc",
  …
}
```

Enrichment Phase 2: Calculable Props

```
{
  "eventName": "page-visit",
  "url": "...?utm_medium=ad",
  "fingerprint": "abc",
  "attributedUserId": 123,
  …
}
```

```
{
  "eventName": "subscribe",
  "userId": 123,
  "fingerprint": "abc",
  "firstUtmMedium": "ad",
  …
}
```

Enrichment Phase 2: Calculable Props Engine & DSL

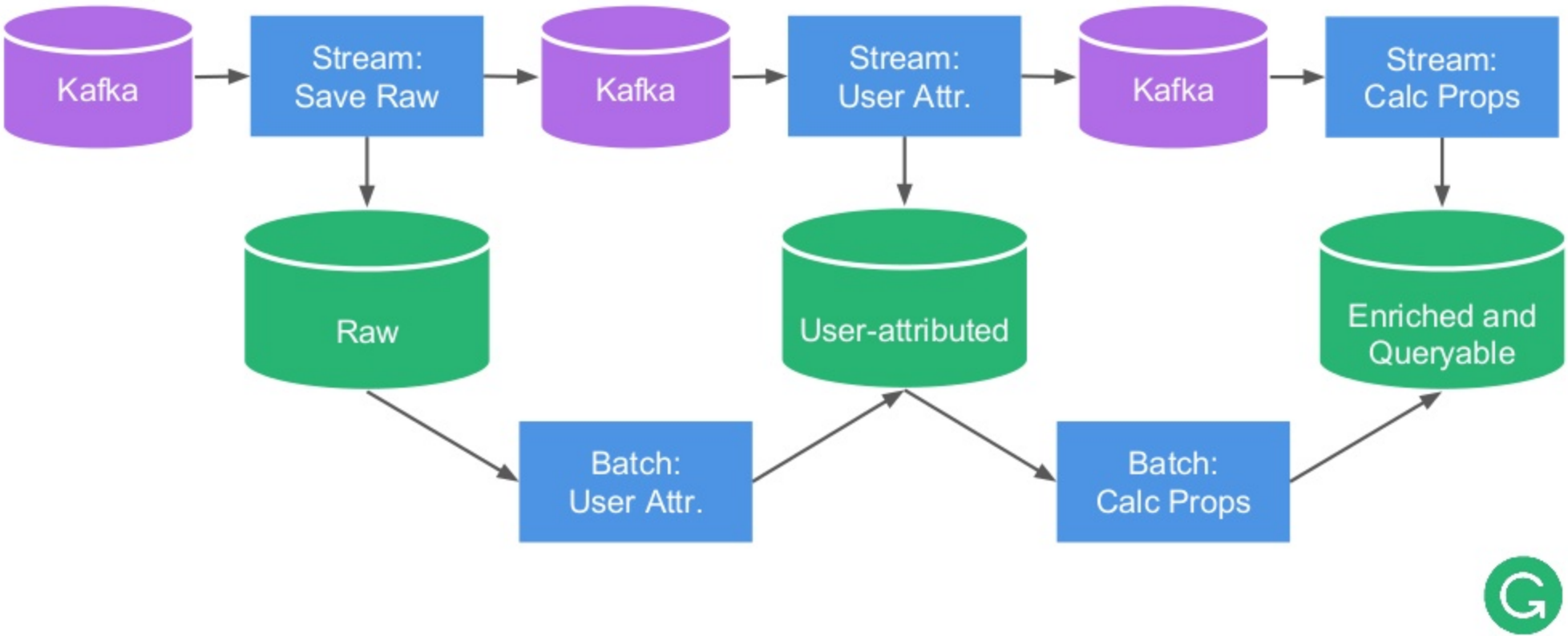
```
val firstUtmMedium: CalcProp[String] =
  (E \ "url").as[Url]
  .map(_.param("utm_source"))
  .forEvent("page-visit")
  .last
```

Enrichment Phase 2: Calculable Props Engine & DSL

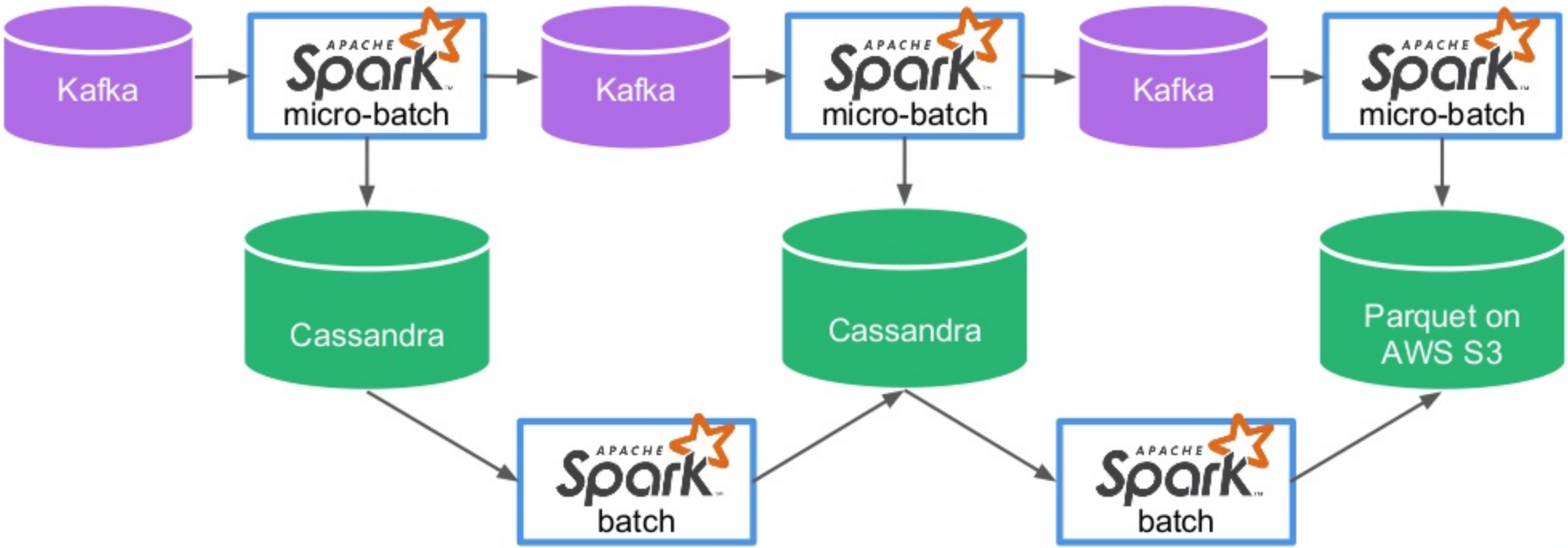
- Type-safe, functional
- Composable
- Similar to Scala collections API
- Batch & Stream (incremental)

Enrichment Phase 3: Spark Pipeline

Spark Pipeline



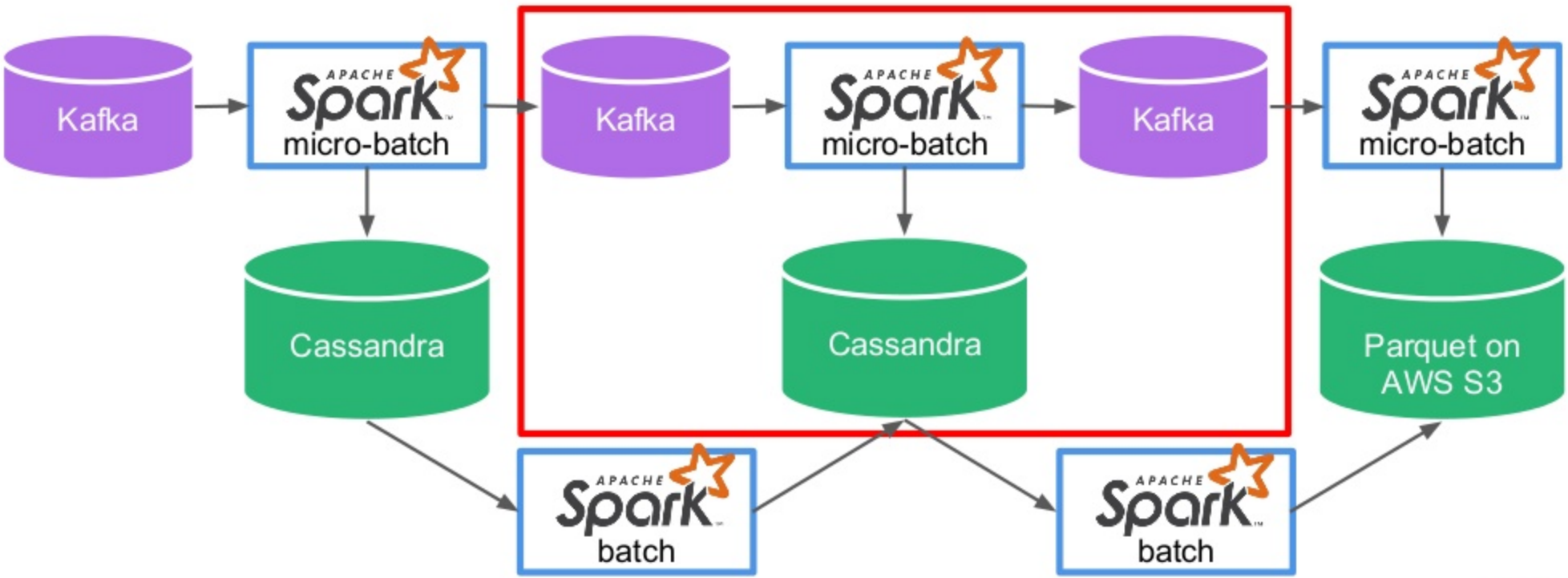
Spark Pipeline



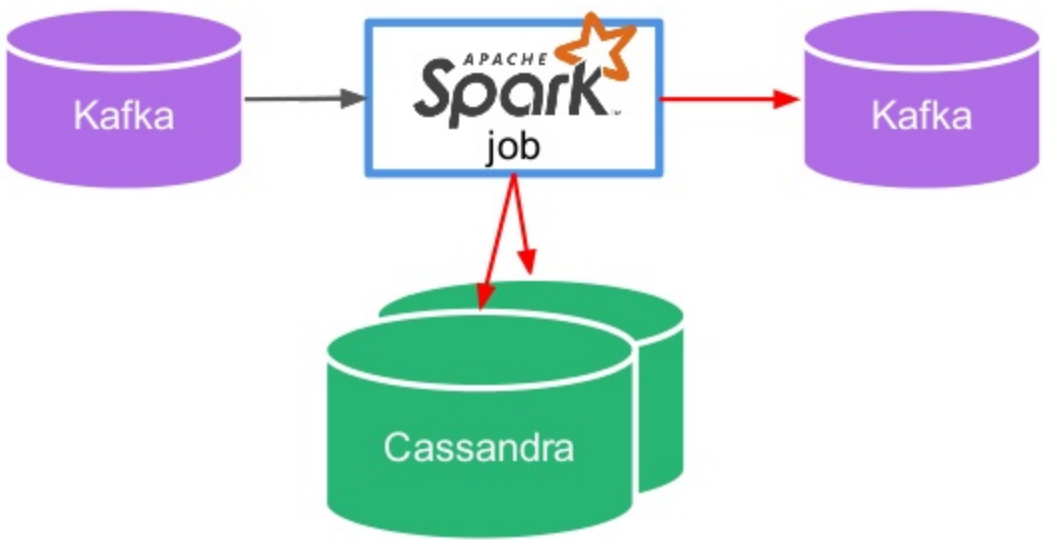
Enrichment Phase 3: Spark

- Connectors for everything
- Great for batch
  - Shuffle with spilling
  - Failure recovery
- Great for streaming
  - Fast
  - Low overhead

Save to Long-term Storage



Multiple Output Destinations



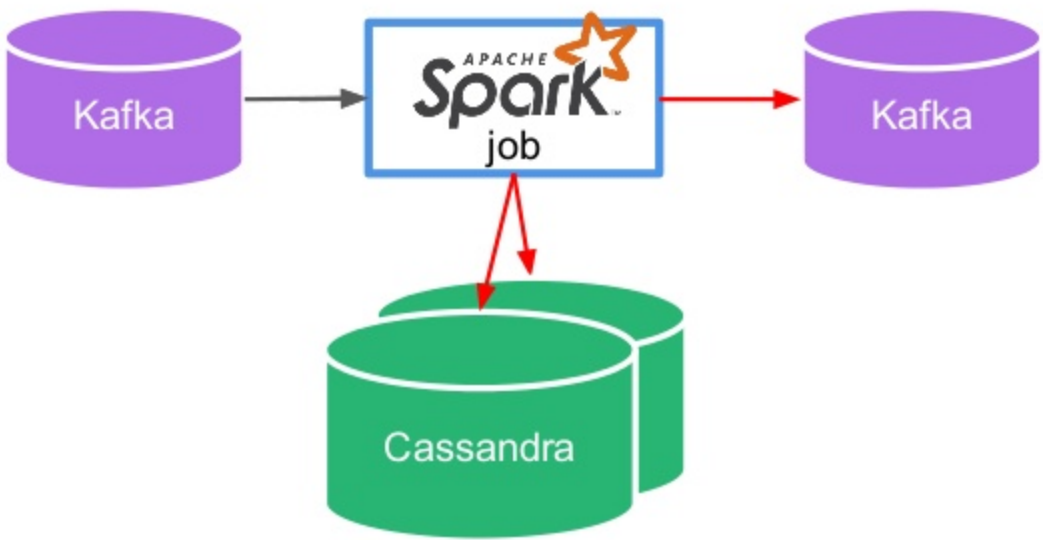
Multiple Output Destinations: Try 1

```
val rdd: RDD[T]

rdd.sendToKafka(“topic_x”)

rdd.saveToCassandra(“table_foo”)

rdd.saveToCassandra(“table_bar”)
```

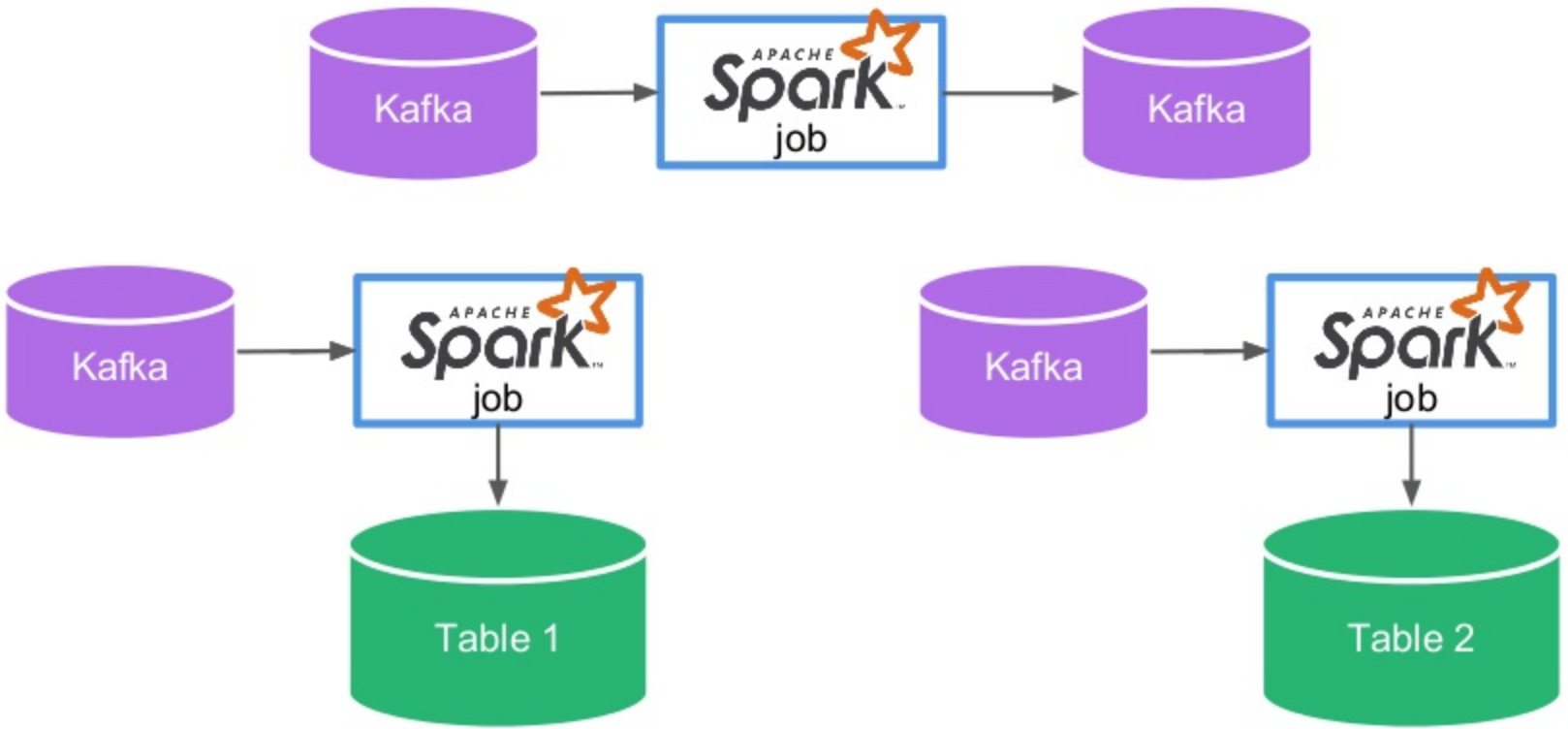
Multiple Output Destinations: Try 1

```
rdd.saveToCassandra(...)

        rdd.forEachPartition(...)

                sc.runJob(...)
```

Multiple Output Destinations: Try 1



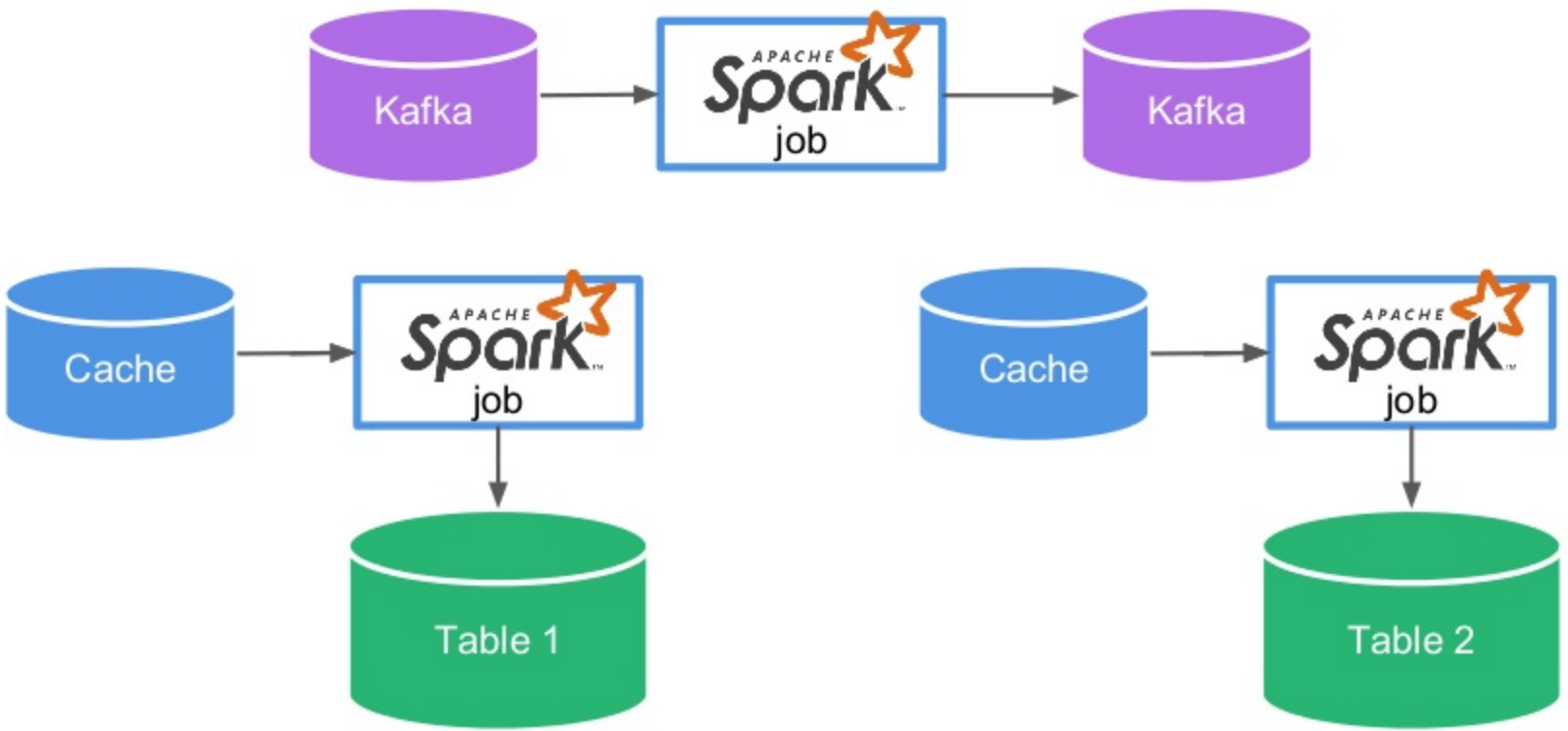
Multiple Output Destinations: Try 1 = 3 Jobs



Multiple Output Destinations: Try 2

```
val rdd: RDD[T]

rdd.cache()

rdd.sendToKafka(“topic_x”)

rdd.saveToCassandra(“table_foo”)

rdd.saveToCassandra(“table_bar”)
```

Multiple Output Destinations: Try 2 = Read Once, 3 Jobs



Writer

```
rdd.forEachPartition { iterator =>

}
```

Writer

```
rdd.forEachPartition { iterator =>

  val writer = new BufferedWriter(
    new OutputStreamWriter(new FileOutStream())
  )




}
```

Writer

```
rdd.forEachPartition { iterator =>

  val writer = new BufferedWriter(
    new OutputStreamWriter(new FileOutStream())
  )

  iterator.forEach { el =>
    writer.writeln(el)
  }


}
```

Writer

```
rdd.forEachPartition { iterator =>

  val writer = new BufferedWriter(
    new OutputStreamWriter(new FileOutStream(...))
  )

  iterator.forEach { el =>
    writer.writeln(el)
  }

  writer.close() // makes sure this writes
}
```

- Buffer
- Non-blocking
- Idempotent / Dedupe

AndWriter

```
andWriteToX = rdd.mapPartitions { iterator =>




}
```

AndWriter

```
andWriteToX = rdd.mapPartitions { iterator =>

  val writer = new XWriter()

  val writingIterator = iterator.map { el =>
    writer.write(el)
  }
}
```

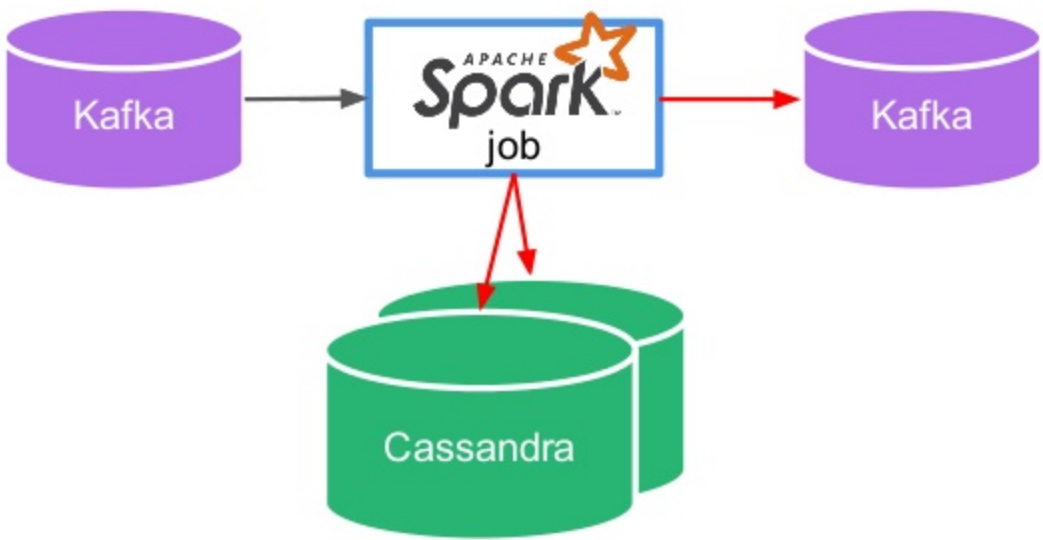
AndWriter

```
andWriteToX = rdd.mapPartitions { iterator =>

  val writer = new XWriter()

  val writingIterator = iterator.map { el =>
    writer.write(el)
  }.closing(() => writer.close)
}
```

Multiple Output Destinations: Try 3

```
val rdd: RDD[T]

rdd.andSaveToCassandra(“table_foo”)
    .andSaveToCassandra(“table_bar”)
    .sendToKafka(“topic_x”)
```

Multiple Output Destinations: Try 3



And Writer

- Kafka
- Cassandra
- HDFS

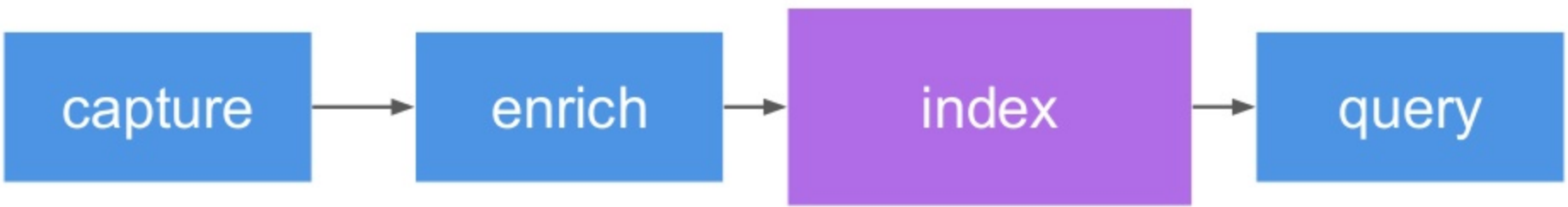
Important! Each andWriter will consume resources

- Memory (buffers)
- IO

Some Stats

- Thousands of events per second
- Terabytes of compressed data

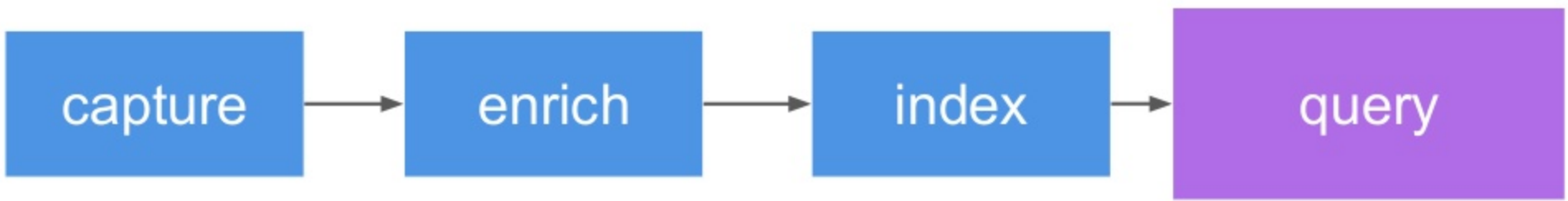
Build Step 3: Index



Index

- Parquet on AWS S3
- Custom partitioning: By eventName and time interval
- Append changes, compact + merge on the fly when querying
- Randomized names to maximize S3 parallelism
- Use and tweak s3a for max performance and read after write S3 consistency
- Support flexible schema, even with conflicts!

Build Step 4: Query



Hardcore Query

- DataFrames
- Spark SQL Scala dsl / Pure SQL
- Zeppelin

Casual Query

- Plot by day
- Unique visitors
- Filter by country
- Split by traffic source (top 20)
- Time from 2 weeks ago to today

Option 1: SQL

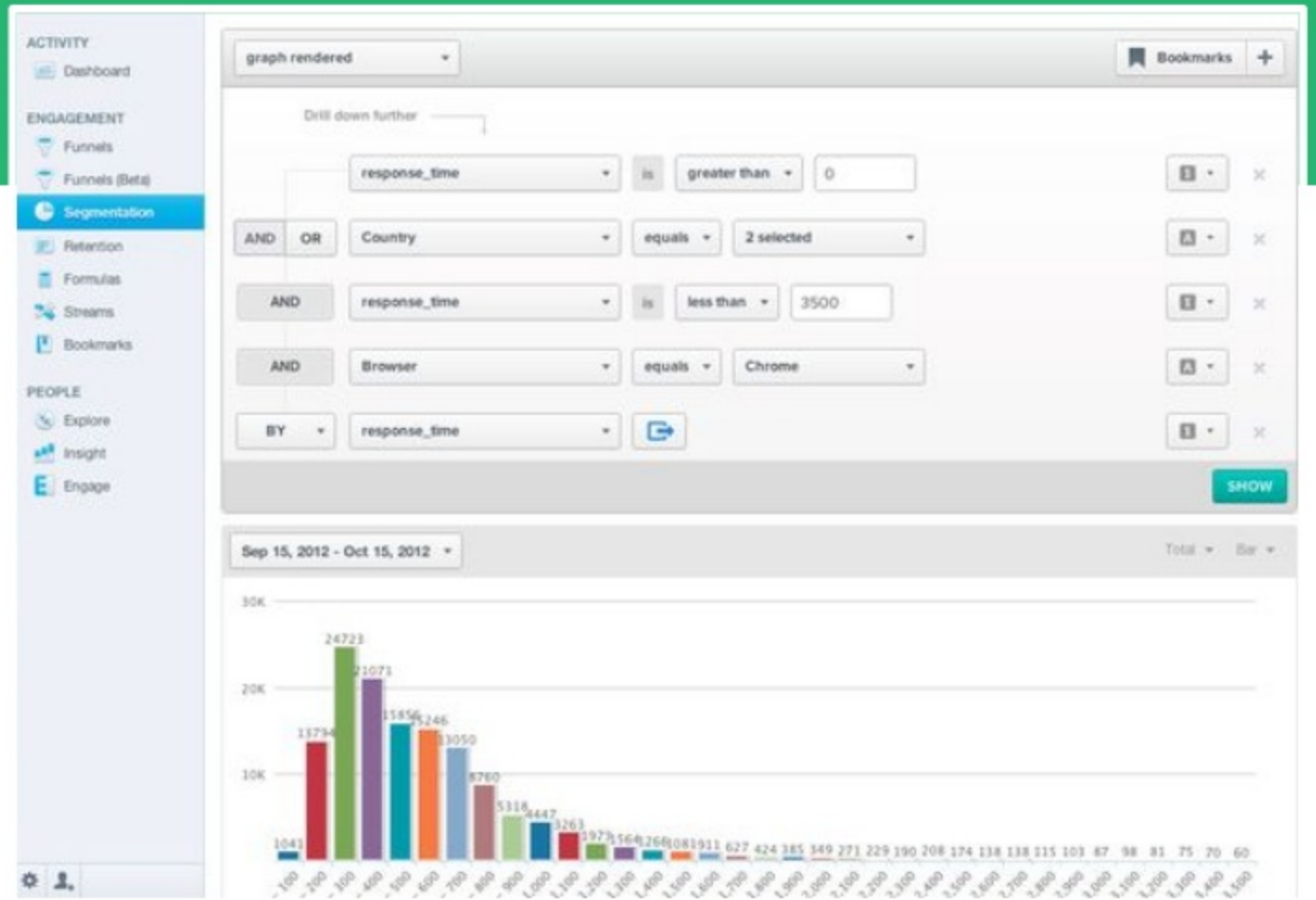
```
// 1) materialize query details

val aggValue: AggValue = segmentQuery.aggValue.getOrElse(AggValue(CountFun, Literal(1l, LongType)))

var byLimit: Int = segmentQuery.by.flatMap(_.limit).getOrElse(50)
byLimit = math.min(200, math.max(0, byLimit))

val now: Long = Platform.currentTime
val segmentTime: SegmentTime = SegmentTime.build(segmentQuery, now)

// 2) then the event names needed for the query

val eventNameFilter = EventNameFilter(segmentQuery.event)
val eventNames = sqlConnector.withConnection(implicit connection => IndexCellsDAO.listAllEventNames.filt


// 3) prepare predicate

val predicate = And(
  segmentQuery.filter.map(_.cast(BooleanType)).getOrElse(
  if (aggValue.aggFunction == CountFun) IsNotNull(aggValu
)


// 4) prepare projections

var projections: Seq[NamedExpression] = Seq()

// slot
projections :+= (if (segmentTime.isExplosive) {
  DaySlotExplosionExpression
} else if (segmentTime.isSubDay) {
  SubDaySlotExpression($"xts")
} else {
  DaySlotExpression
}).as("slot")

// value
if (aggValue.aggFunction.valueNeeded) {
  projections :+= aggValue.valueExpression.cast(DoubleType).as("value")
}
```

```
// by
if (segmentQuery.by.nonEmpty) {
  projections :+= segmentQuery.by.map(_.byExpression).get.cast(StringType).as("by")
}

// pidHash
if (segmentQuery.isUnique) {
  projections :+= $"pidHash"
}

// ts
if (segmentQuery.isUnique && (aggValue.aggFunction.valueNeeded || segmentQuery.by.nonEmpty && segmentQue
  projections :+= (if (segmentTime.isMultiDay) StsExpression($"xts") else $"xts").as("ts")
}

                                                                                    and df

                                                                                    r.withConnection { implicit connection =>
                                                                                    (eventNames, segmentTime.days)


// taking only the first layer for now
if (cells.isEmpty) return Future.successful(SegmentQueryResult.empty(now))

val traversal: Traversal = Traversal.build(cells, predicate, projections, ordered = false)
if (traversal.isEmpty) return Future.successful(SegmentQueryResult.empty(now))

val traversalPlan: TraversalPlan = traversalPlanBuilder.build(traversal)
if (traversalPlan.isEmpty) return Future.successful(SegmentQueryResult.empty(now))

val init = () => {
  SegmentTime.setCtx(segmentTime)
}

var df: DataFrame = indexTraverser.traverseDF(traversalPlan, init, sparkContext, sqlContext)


// 6) explode explosion into slot if needed
```

Quickly gets complex

Option 2: UI

Too expensive to build, extend and support

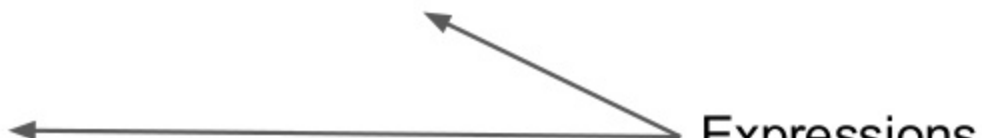


Option 3: Custom Query Language

```
SEGMENT “eventName”
WHERE foo = “bar” AND x.y IN (“a”, “b”, “c”)
UNIQUE
BY m IS NOT NULL
TIME from 2 months ago to today
     STEP 1 month SPAN 1 week
```

Option 3: Custom Query Language

```
SEGMENT “eventName”
WHERE foo = “bar” AND x.y IN (“a”, “b”, “c”)
UNIQUE
BY m IS NOT NULL
TIME from 2 months ago to today
     STEP 1 month SPAN 1 week
```



Spark Expressions

```
class Expression {
    def eval(input: InternalRow = null): Any
}
```

Spark Expressions

```
class Expression {
    def eval(input: InternalRow = null): Any
}

case class Add(left: Expression, right: Expression)
    extends Expression {

    override def eval(input: InternalRow = null): Any {
        left.eval(input).asInstanceOf[Int]
            + right.eval(input).asInstanceOf[Int]
    }
}
```

Scala Parser Combinators

```
val termExpression: Parser[Expression] = ???

val addParser = termExpression ~
    ("+"  ~> termExpression) ^^ {
       case e1 ~ e2 => Add(e1, e2)
    }
```

SEGMENT

```
lazy val eventParser: Parser[Seq[String]] =
    ((EVENT | FROM).? ~> rep1sep(strLit, ",")).map(_.flatten)

lazy val filterParser: Parser[Option[Expression]] =
    ((FILTER | WHERE) ~> expression).?

lazy val timeParser: Parser[SegmentTime] = …
```

Take expression parser
from Spark SQL Parser

SEGMENT

```
lazy val segmentQueryParser: Parser[SegmentQuery] = {

  SEGMENT ~> eventParser ~ filterParser ~
    byParser ~ timeParser ^^ {

    case event ~ filter ~ by ~ time =>
      SegmentQuery(event ~ filter ~ by ~ time)

  }

}
```

Option 3: Custom Query Language

- Segment, Funnel, Retention
- UI & as DataFrame in Zeppelin
- Spark <= 1.6 – Scala Parser Combinators
- Reuse most complex part of expression parser
- Relatively extensible

Quick Demo!

Conclusion

- Custom versatile analytics is doable and enjoyable
- Spark is a great platform to build analytics on top of
    - Enrichment Pipeline: Batch / Streaming, Query
- Would be cool to see even deep internals slightly more extensible


We are hiring!
CSS
zen
olivia@grammarly.com
https://www.grammarly.com/jobs

Questions?