# Ingesting streaming data into Graph Database

Analytics and Data Summit

Guido Schmutz
San Francisco, 22.3.2018



@gschmutz


guidoschmutz.wordpress.com

# Guido Schmutz

Working at Trivadis for more than 21 years

Oracle ACE Director for Fusion Middleware and SOA

Consultant, Trainer Software Architect for Java, Oracle, SOA and Big Data / Fast Data

Head of Trivadis Architecture Board

Technology Manager @ Trivadis

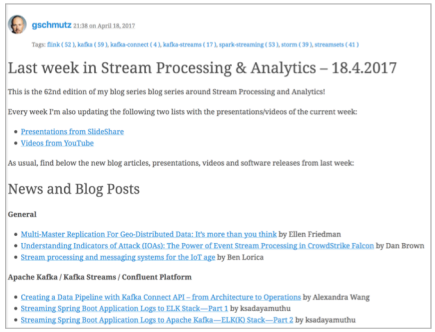
More than 30 years of software development experience


Contact: guido.schmutz@trivadis.com

Blog: http://guidoschmutz.wordpress.com

Slideshare: http://www.slideshare.net/gschmutz

Twitter: gschmutz

gschmutz 21:38 on April 18, 2017

Tags: flink ( 52 ), kafka ( 59 ), kafka-connect ( 4 ), kafka-streams ( 17 ), spark-streaming ( 53 ), storm ( 39 ), streamsets ( 41 )

Last week in Stream Processing & Analytics – 18.4.2017

This is the 62nd edition of my blog series blog series around Stream Processing and Analytics!

Every week I'm also updating the following two lists with the presentations/videos of the current week:

- Presentations from SlideShare
- Videos from YouTube

As usual, find below the new blog articles, presentations, videos and software releases from last week:

News and Blog Posts

General

- Multi-Master Replication For Geo-Distributed Data: It's more than you think by Ellen Friedman
- Understanding Indicators of Attack (IOAs): The Power of Event Stream Processing in CrowdStrike Falcon by Dan Brown
- Stream processing and messaging systems for the IoT age by Ben Lorica

Apache Kafka / Kafka Streams / Confluent Platform

- Creating a Data Pipeline with Kafka Connect API – from Architecture to Operations by Alexandra Wang
- Streaming Spring Boot Application Logs to ELK Stack—Part 1 by ksadayamuthu
- Streaming Spring Boot Application Logs to Apache Kafka—ELKK Stack—Part 2 by ksadayamuthu


trivadis
makes IT easier.

# Agenda

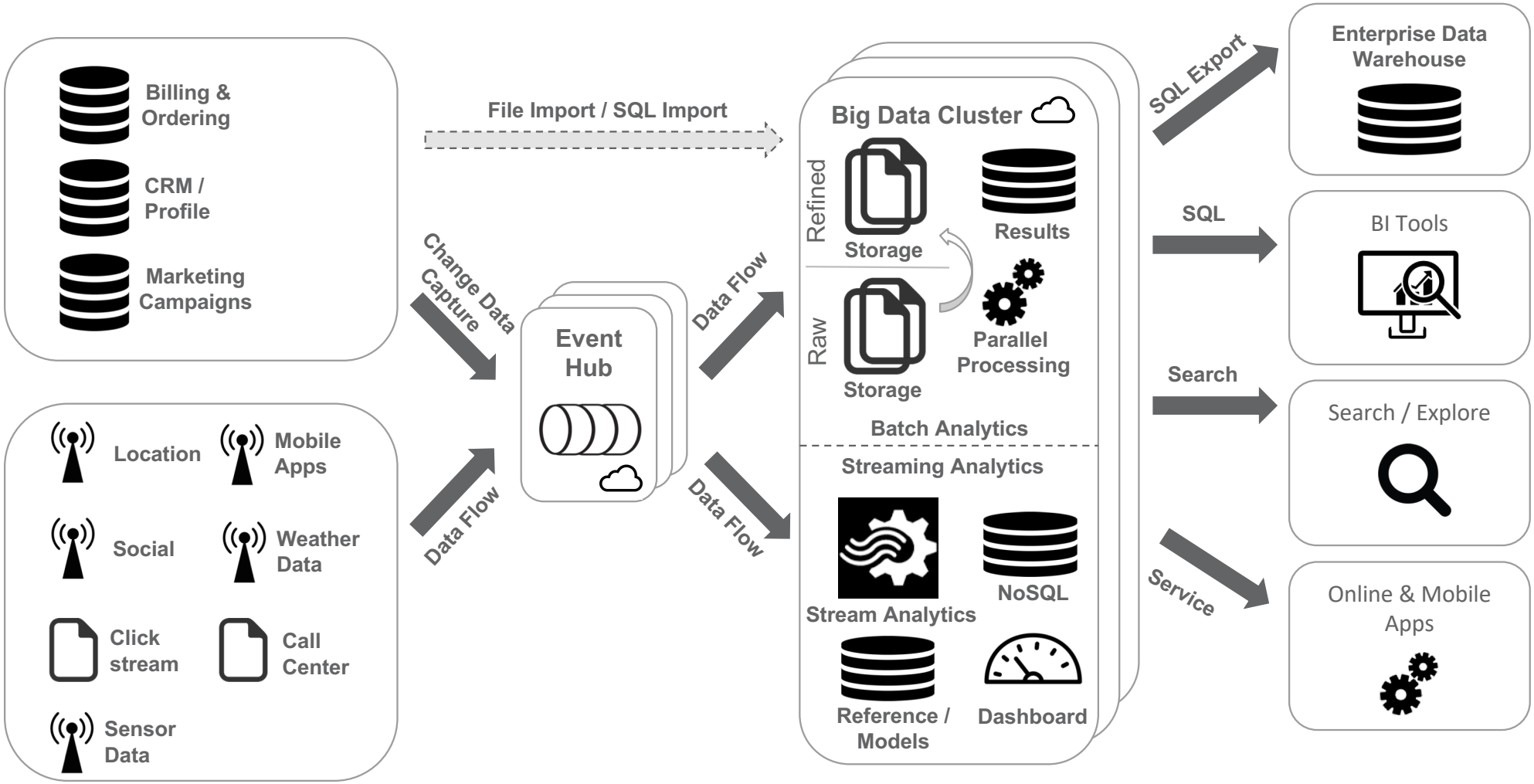
1. Introduction
2. DataStax (DSE) Graph
3. Use-Case 1 - Batch Data Ingestion of Social Graph
4. Use-Case 2 - Streaming Ingestion of Social Graph
5. Use-Case 3 - Streaming Ingestion into RDF Graph
6. Summary

trivadis
makes IT easier.

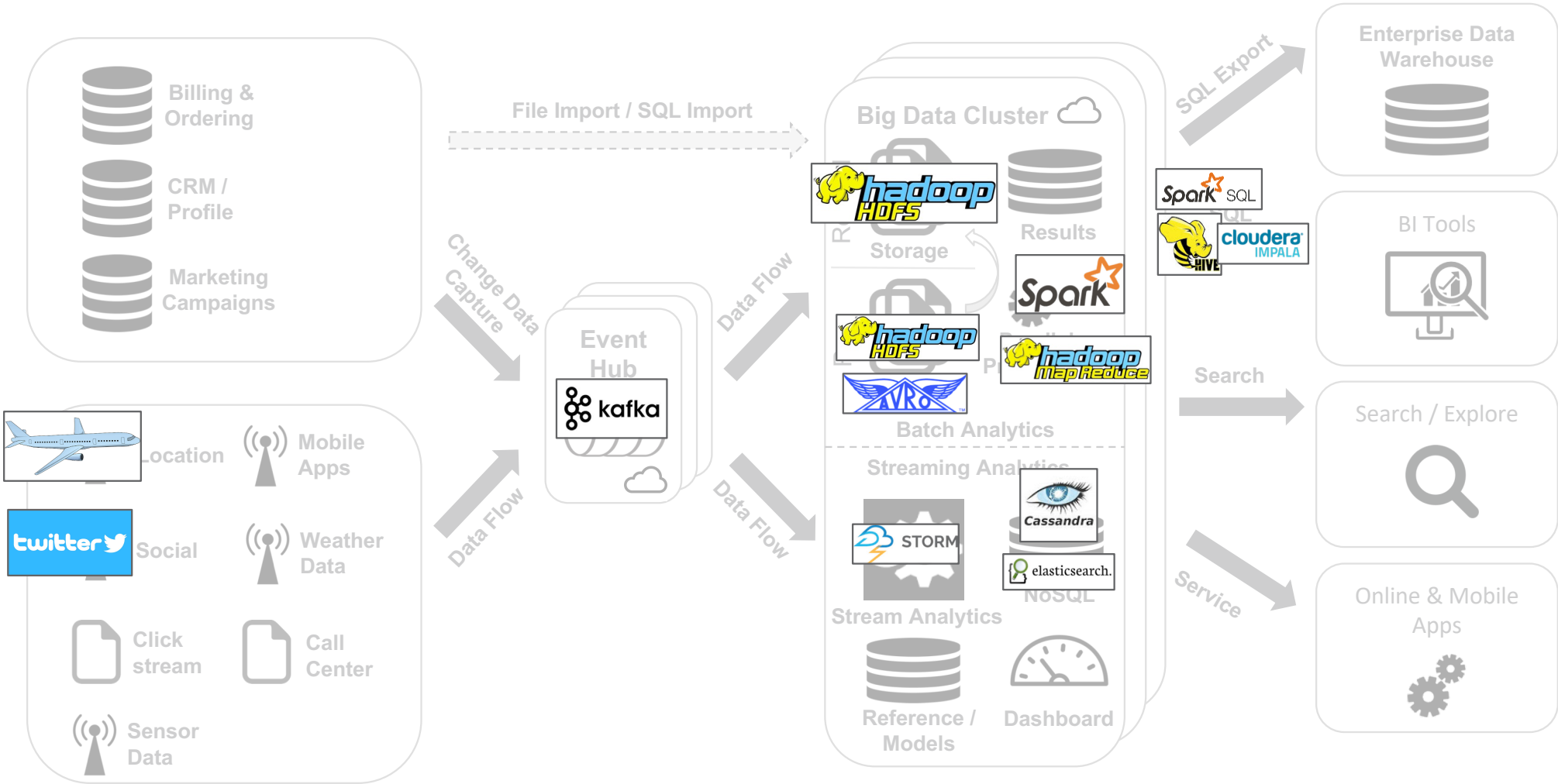
# Introduction

trivadis
makes IT easier.

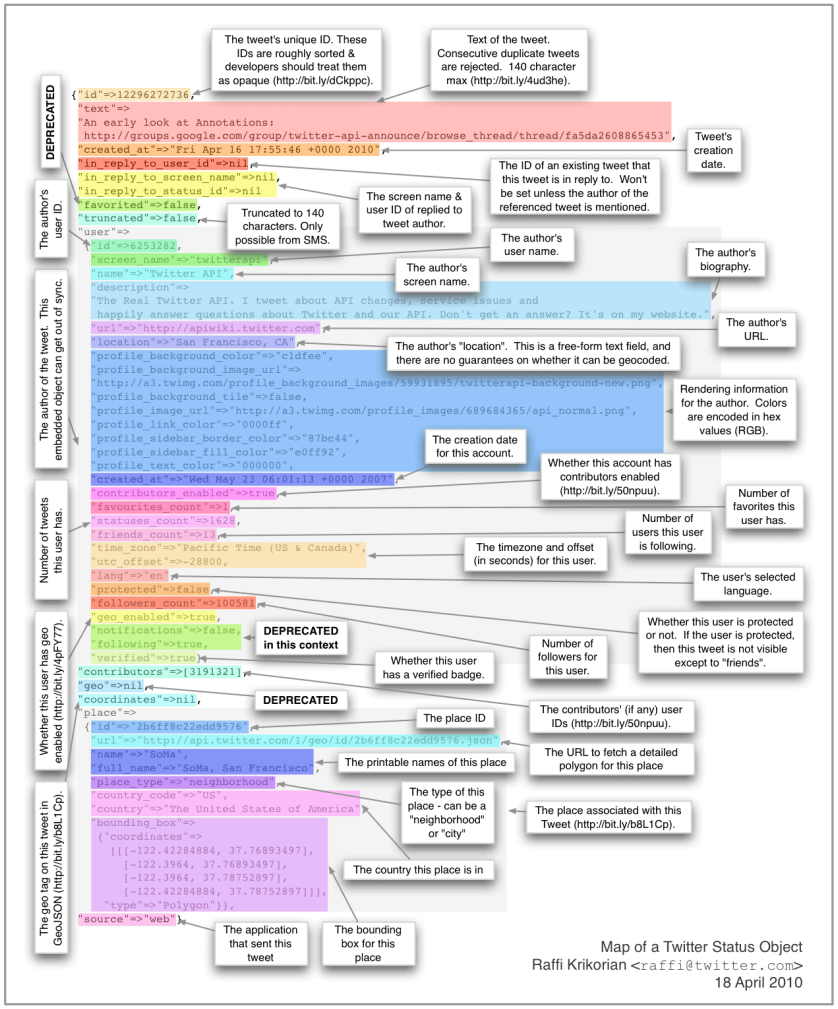
# Customer Context – Big Data Lab

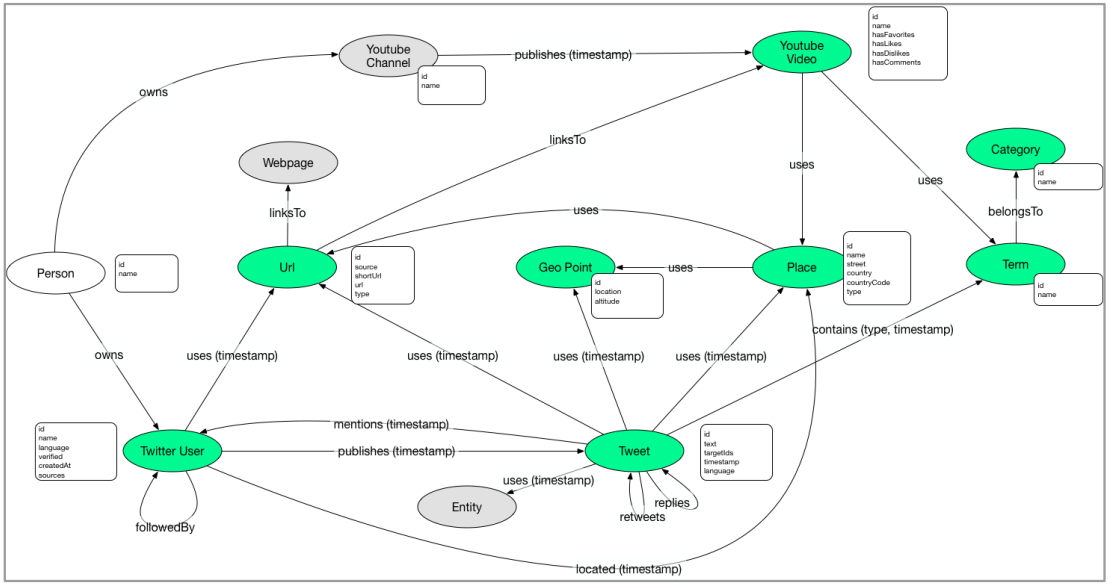


# Customer Context – Big Data Lab



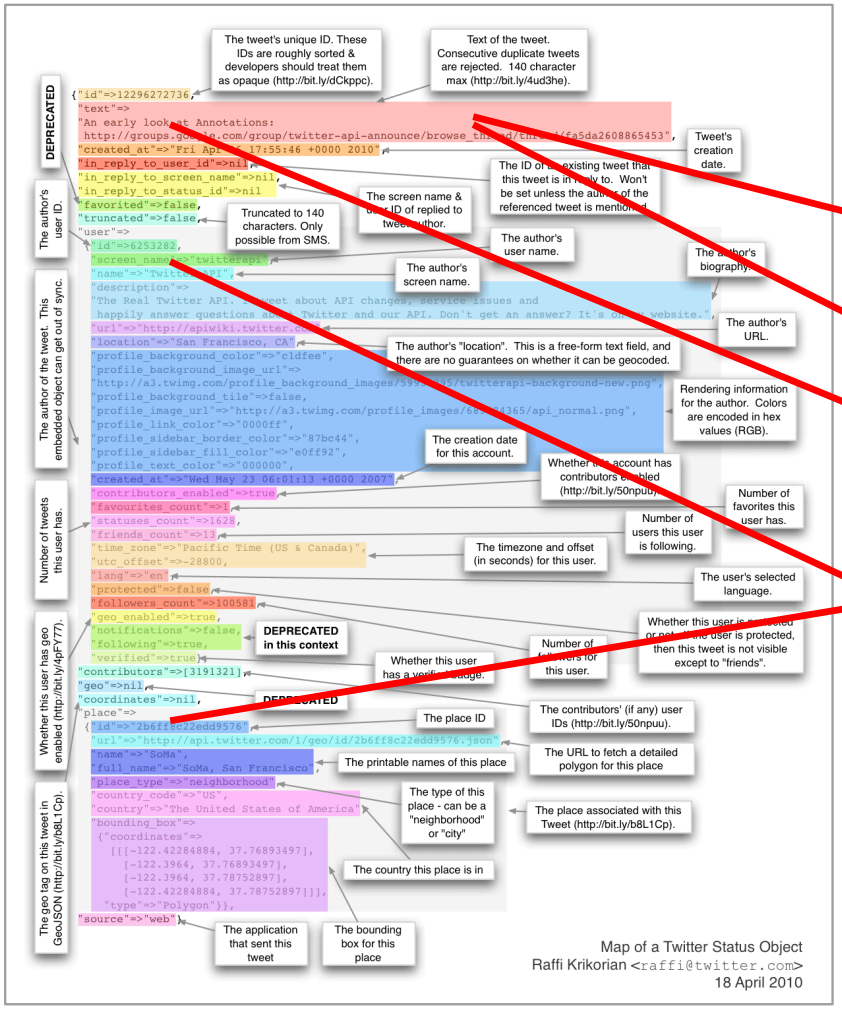
# Map of a Twitter Status object

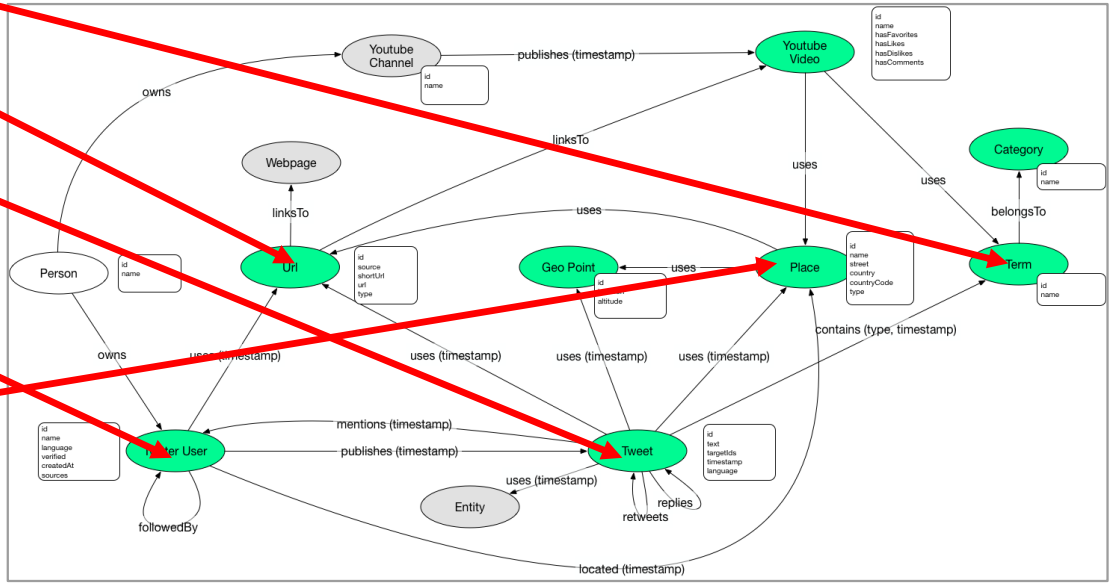



trivadis
makes IT easier.

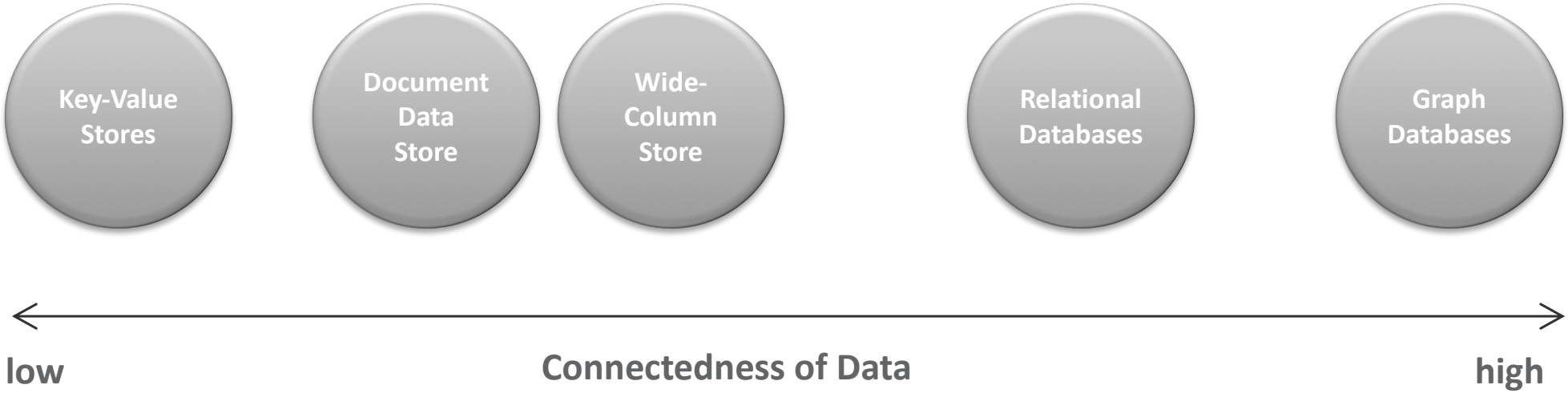
# Map of a Twitter Status object




trivadis
makes IT easier.

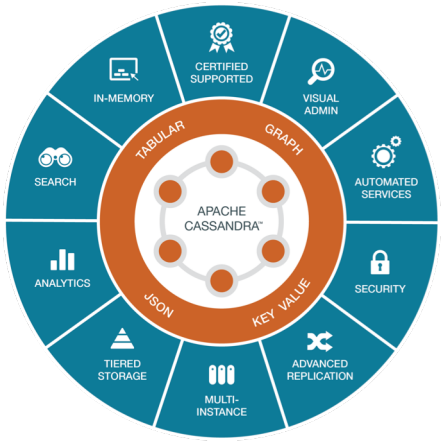
# Know your domain and choose the right data store!



# DataStax (DSE) Graph

trivadis
makes IT easier.

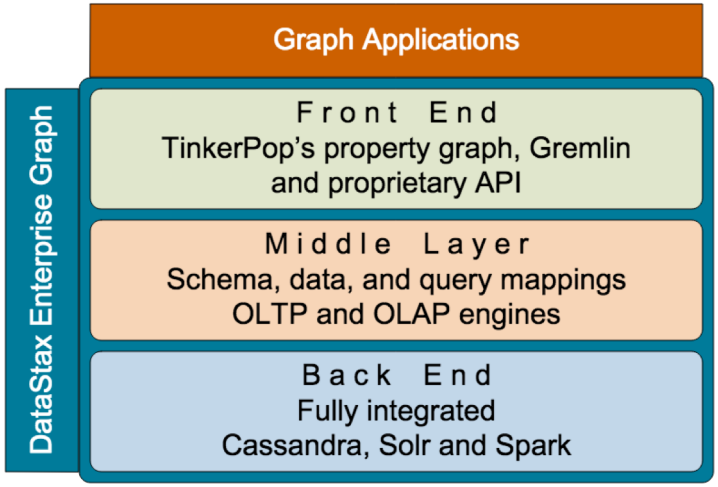
# DataStax Enterprise Edition – DSE Graph




A scale-out property graph database

Store & find relationships in data fast and easy in large graphs.

- Adopts **Apache TinkerPop** standards for data and traversal
- Uses **Apache Cassandra** for scalable storage and retrieval
- Leverages **Apache Solr** for full-text search and indexing
- Integrates **Apache Spark** for fast analytic traversal
- Supports data security




Images: DataStax

trivadis
makes IT easier.

# Lab cluster setup

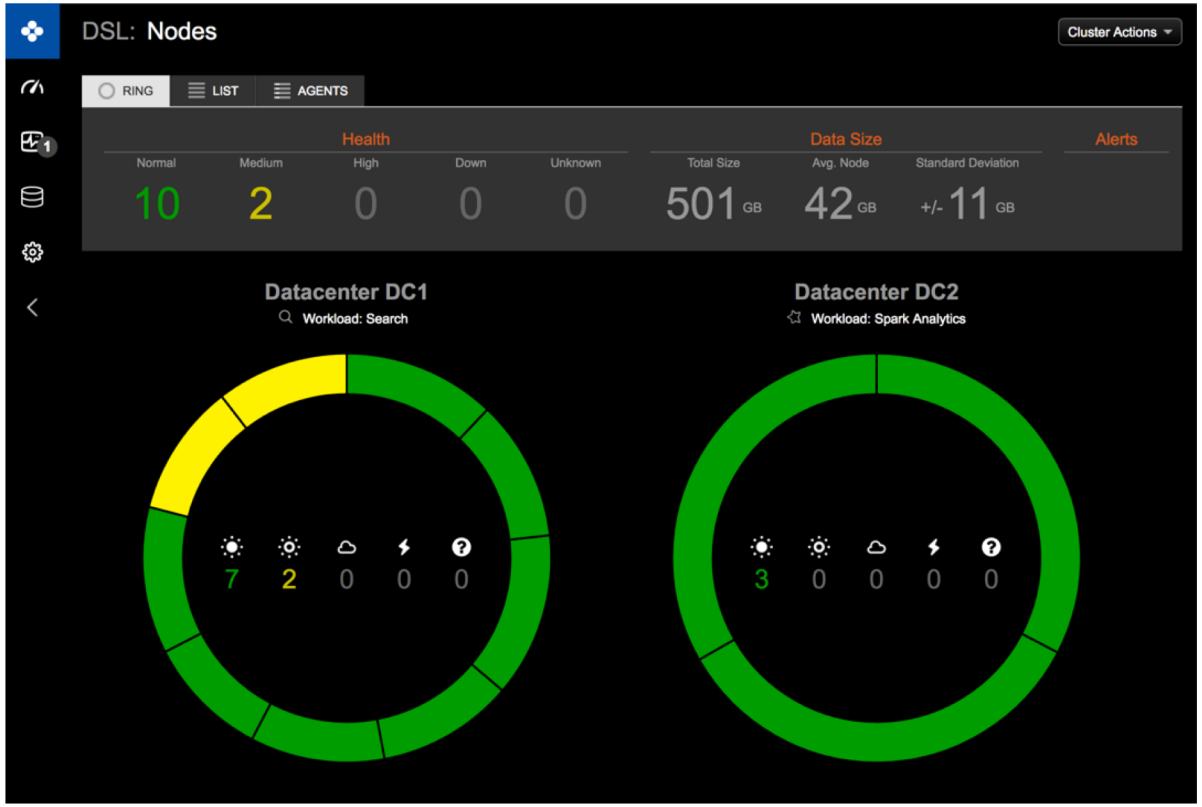
2 local data center for workload segregation

DC1

- 9 nodes
- OLTP workload incl. Search

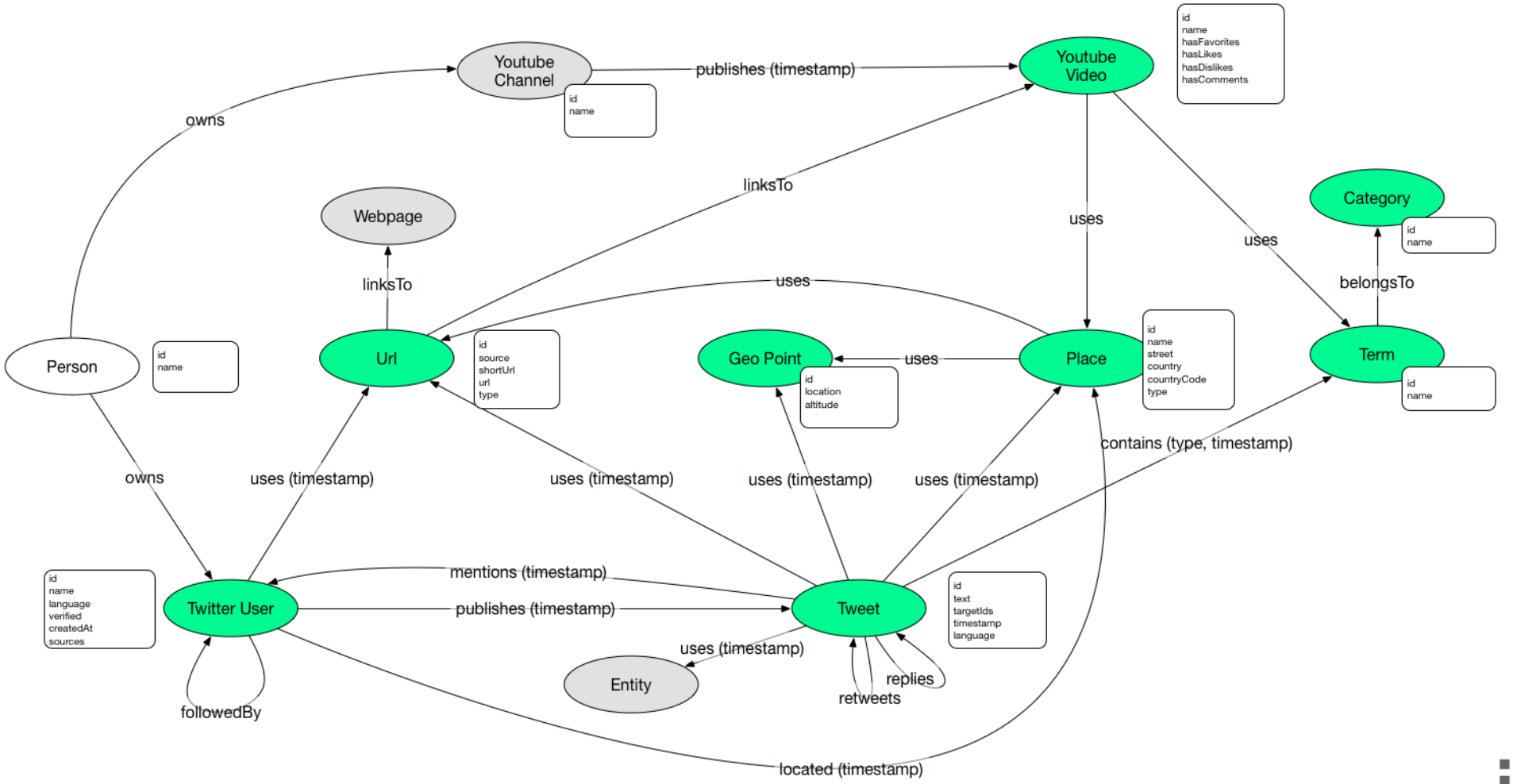
DC2

- 3 nodes
- OLAP workload with Spark



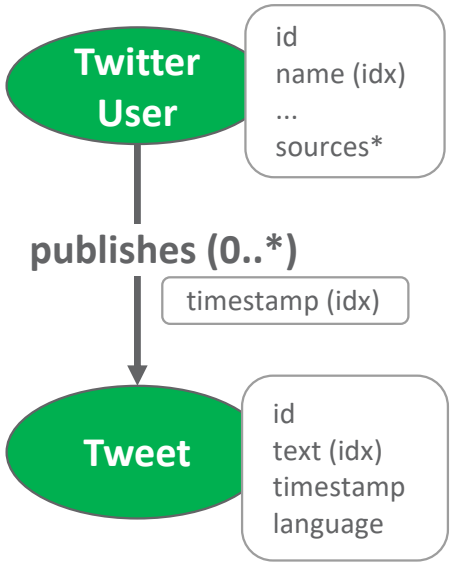

trivadis
makes IT easier.

# Social Network Graph Model




trivadis
makes IT easier.

# Creating the Graph Schema (I)




Define types of properties to be used with Vertices and Edges

```
schema.propertyKey("id").Long().single().create();
schema.propertyKey("name").Text().create();
schema.propertyKey("sources").Text().multiple().create();
```

Define types of vertices to be used

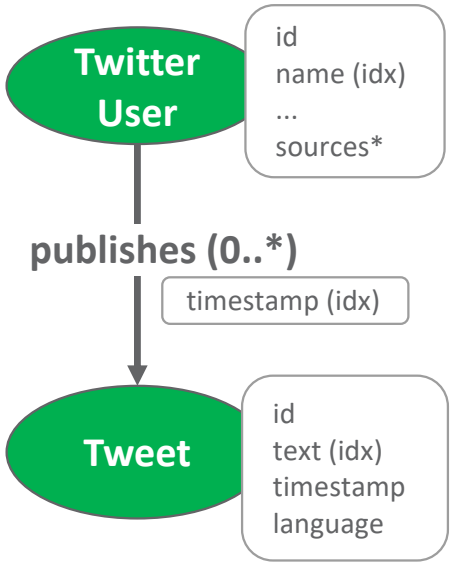
```
schema.vertexLabel("twitterUser").
    properties("id","name","language","verified","createdAt","sources").create();
schema.vertexLabel("tweet".
    properties("id","text","targetIds","timestamp","language").create()
```

Define types of edges to be used

```
schema.edgeLabel("publishes").properties("timestamp")
       multiple().properties("rating").
       connection("twitterUser","tweet").create();
```

trivadis
makes IT easier.

# Creating the Graph Schema (II)




Define global materialized index on vertex (good for high selectivity)

```
schema.vertexLabel("twitterUser").
         index('twitterUserNameIdx').
         materialized().by('name').ifNotExists().add()
```

Defining global full-text search index (SolR) on text of Tweet

```
schema.vertexLabel("tweet").
         index("search").
         search().by("text").asText().ifNotExists().add()
```

Define vertex-centric index on edge

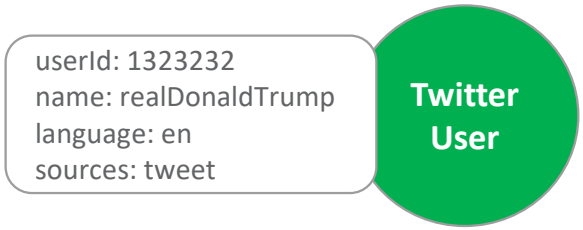
```
schema.vertexLabel("twitterUser").
         index("usesAtTimestamp").
         outE("publishes").by("timestamp").ifNotExists().add()
```

trivadis
makes IT easier.

# Creating Vertex Instances

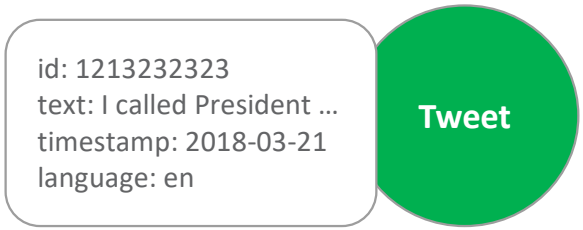
Adding a new **TwitterUser** vertex

```
Vertex u = graph.addVertex("twitterUser");
       u.property("id","1323232");
       u.property("name",”realDonaldTrump");
       u.property("language","en");
       u.property("sources","tweet");
```




Adding a new **Tweet** vertex

```
Vertex m = graph.addVertex(label, "tweet",
                      "id", "1213232323",
                      "text", "I called President Putin of
Russia to congratulate him on his election ...",
                      "timestamp", "2018-03-21",
                      "language", "en");
```

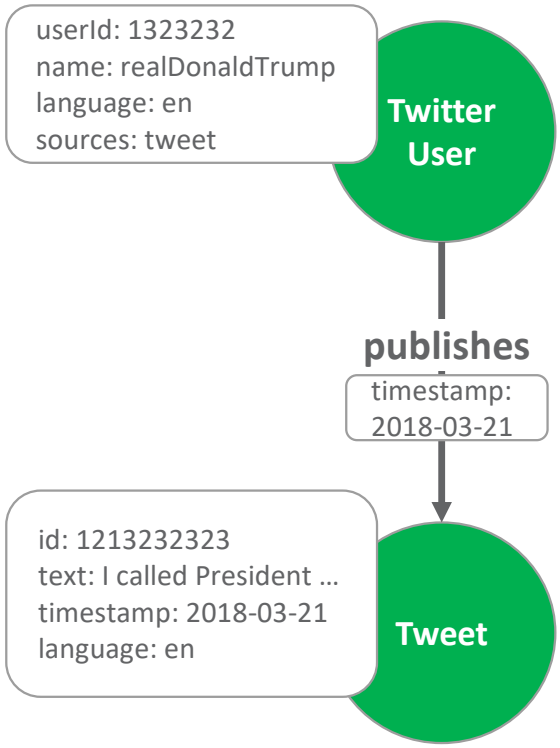



trivadis
makes IT easier.

# Creating Edges between Vertices

Adding an edge between the **TwitterUser** and the **Tweet** vertex

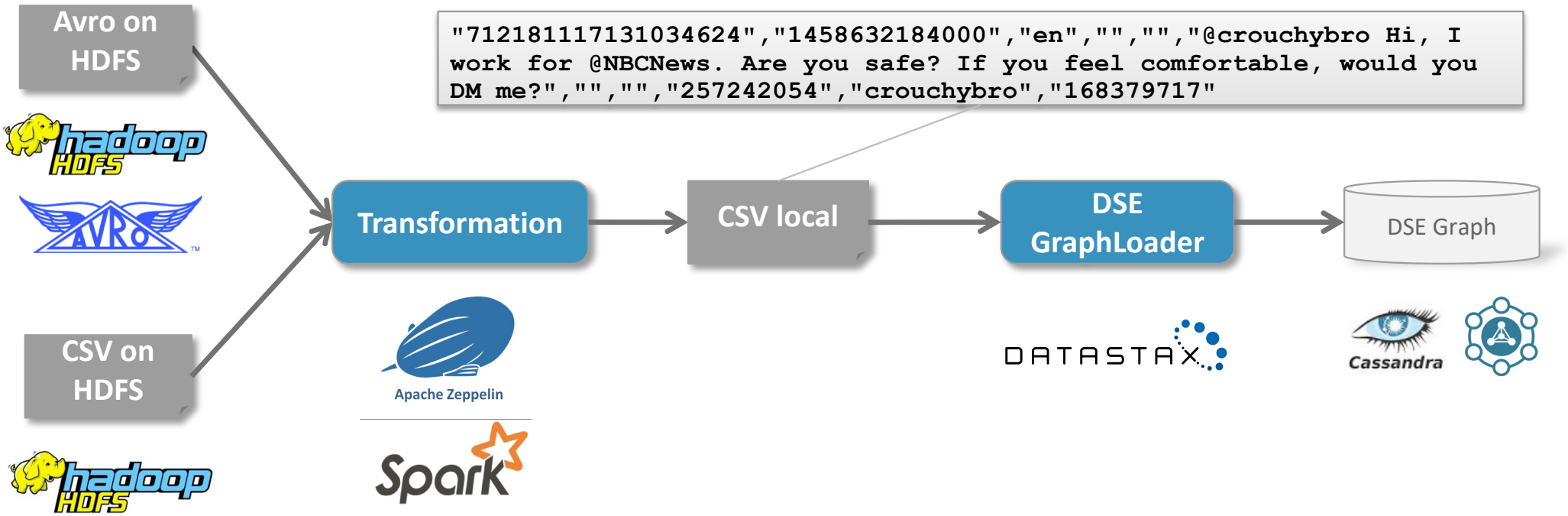
```
Vertex u = graph.addVertex("twitterUser", ...);
Vertex t = graph.addVertex("tweet", ...);

Edge e = u.addEdge("publishes", t);
     e.property("timestamp", "2018-03-21");
```



# Use Case 1 – Batch Data Ingestion of Social Graph

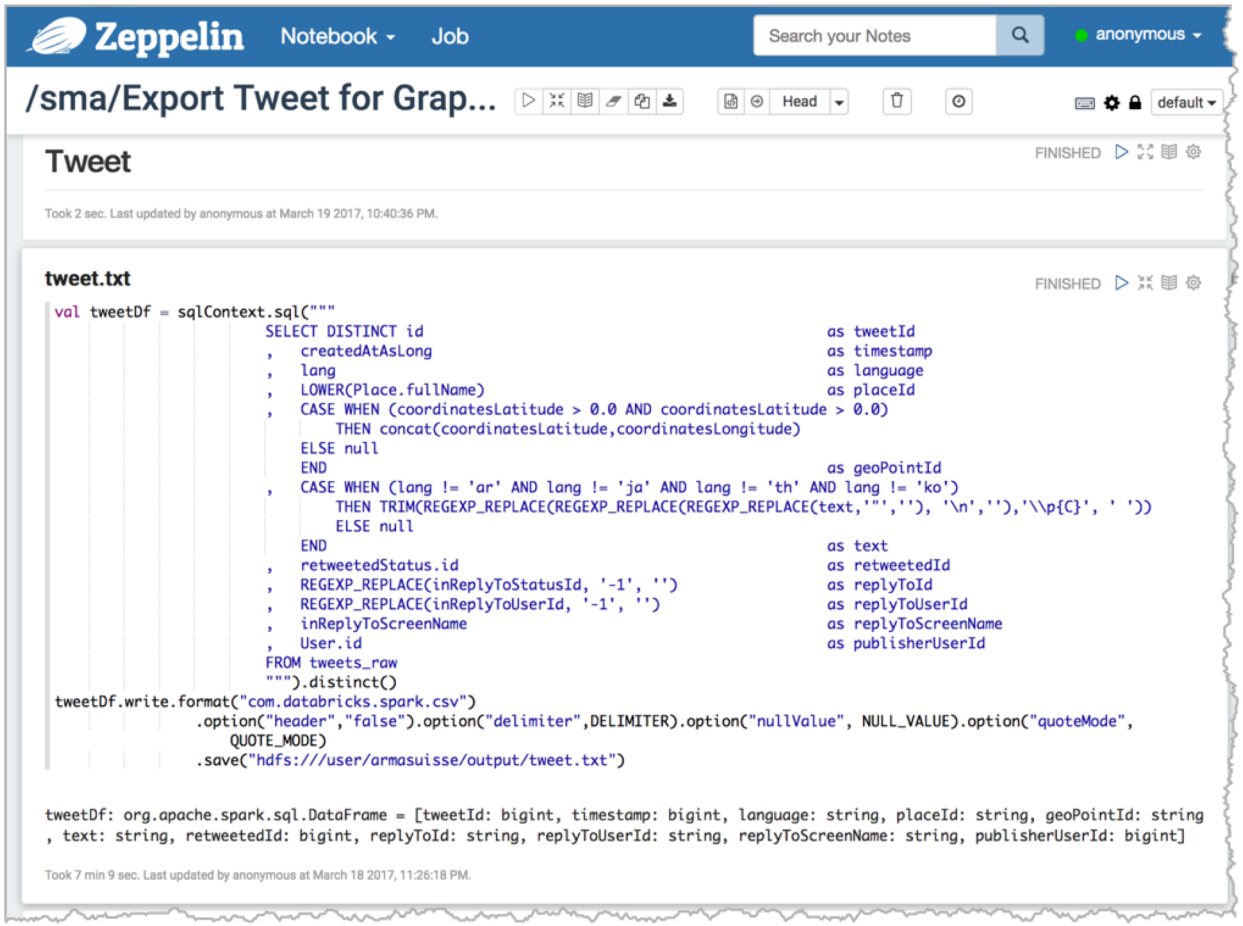
trivadis
makes IT easier.

# Batch Data Ingestion of Social Graph




trivadis
makes IT easier.

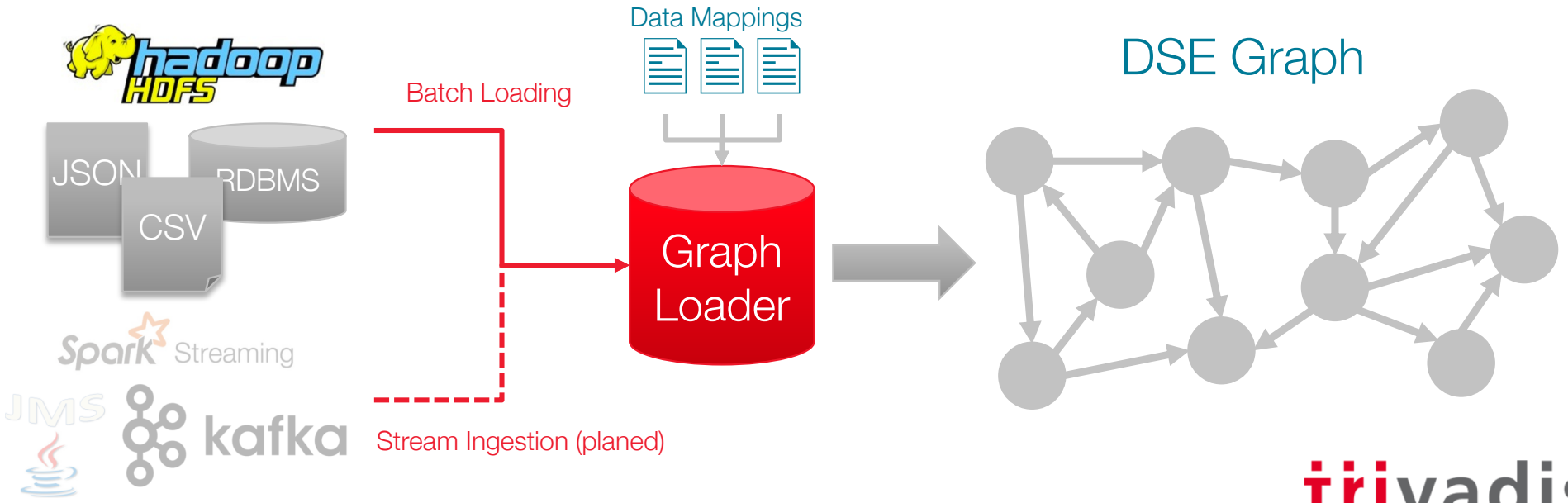
# Extracting and Preparing Data through Zeppelin




trivadis
makes IT easier.

# Batch Ingestion with DSE Loader

- Simplifies loading of data from various sources into DSE Graph efficiently
- Inspects incoming data for schema compliance
- Uses declarative data mappings and custom transformations to handle data
- Loads all vertices first and stores in cache then edges and properties




trivadis
makes IT easier.

# Batch Ingestion with DSE Loader – Example

```
"712181117131034624","1458632184000","en","","","@crouchybro
Hi, I work for @NBCNews. Are you safe? If you feel comfortable,
would you DM me?","","","257242054","crouchybro","168379717"
...
```

Extract

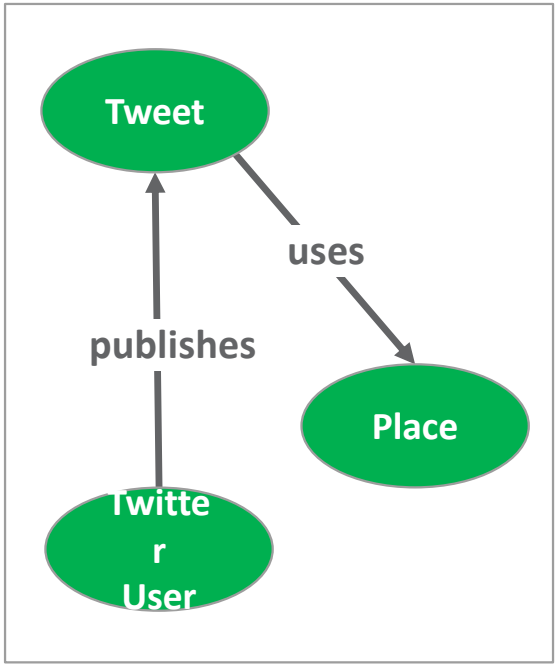
```
tweetInput = File.text("tweet.txt")
  .delimiter('","')
  .header('id','timestamp','language','placeId','geoPointId','text',
        'retweetedId','replyToId','replyToUserId','replyToScreenName',
        'publisherUserId')
```

Transform

```
tweetInput = tweetInput.transform { it['replyToId'] =
     it['replyToId'].equals('-1') ? '' : it['replyToId'];
     it.each {key,value -> it[key] = value.replaceAll('^\"|\"$', '')};
     it }
```

Load

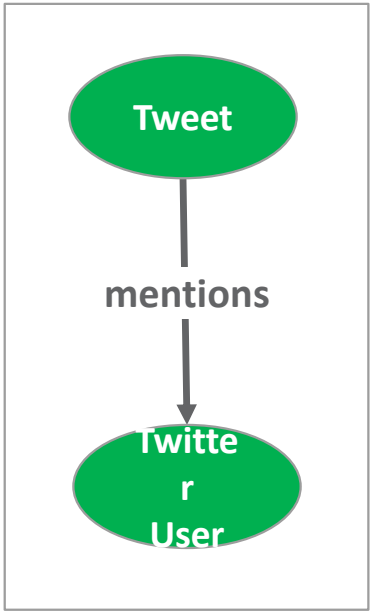
```
load(tweetInput).asVertices {
    isNew()
    label "tweet"
    key "id"
    ignore "placeId"
    ignore "publisherUserId "
    ...
    outV "placeId", "uses", {
        label "place"
        key "id"
    }
    ...
    inV "publisherUserId", "publishes", {
        label "twitterUser"
        key "id"
    }}
```




trivadis
makes IT easier.

# Batch Ingestion with DSE Loader - Example

```
tweetMentionsInput = File.text("tweet_mentions_user.txt")
   .delimiter('","')
   .header('tweetId','userId','screenName','timestamp')

tweetMentionsInput = tweetMentionsInput.transform {
       it.each {key,value -> it[key] = value.replaceAll('^\"|\"$', '')};
       it
}

load(tweetMentionsInput).asEdges {
    label "mentions"
    ignore "screenName"
    outV "tweetId", {
        label "tweet"
        key "tweetId":"id"
    }
    inV "userId", {
        label "twitterUser"
        key "id"
    }
}
```




```
graphloader load-sma-graph.groovy -graph sma_graph_prod_v10 -address
192.168.69.135 -dryrun false -preparation false -abort_on_num_failures
10000 -abort_on_prep_errors false -vertex_complete false
```

trivadis
makes IT easier.

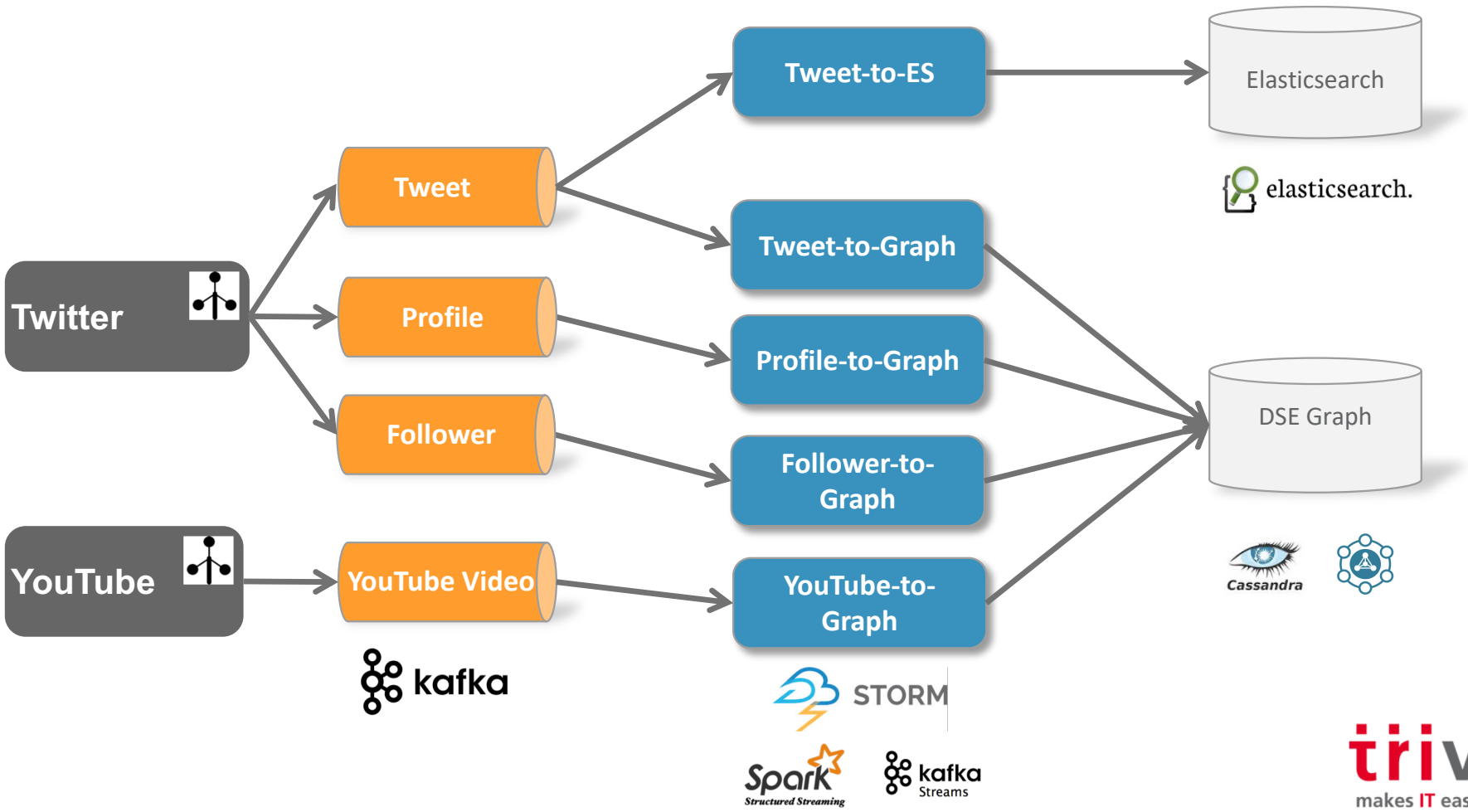
# Performance Tests DSE Graph Loader

| ID strategy | dataset | vertex threads | edge threads | number of tweets | time sec/min | tweets per sec |
|---|---|---|---|---|---|---|
| Gen | 1-day-14 | 0 | 0 | 212’116 | 541/9.01 | 392 |
| Gen | 1-day-14 | 4 | 12 | 212’116 | 419/6.93 | 506 |
| Gen | 1-day-14 | 10 | 10 | 212’116 | 334/5.56 | 635 |
| Gen | 1-day-14 | 40 | 40 | 212’116 | 170/2.83 | 1247 |
| Gen | 1-day-14 | 60 | 80 | 212’116 | 118/1.96 | 1797 |
| Cust | 1-day-14 | 60 | 80 | 212’116 | 101/1.68 | 2100 |
| Gen | 1-mth-14 | 60 | 80 | 7’403’667 | 3888/64.8 | 1904 |
| Gen | 1-mth-14 | 60 | 120 | 7’403’667 | 3685/61.4 | 2009 |
| Cust | 1-mth-14 | 60 | 120 | 7’403’667 | 3315/55.25. | 2233 |
| Gen | 1-mth-14 | 80 | 120 | 7’403’667 | 4005/66.75 | 1848 |
| Cust | 1-mth-14 | 80 | 120 | 7’403’667 | 3598/59.69 | 2057 |

# Use Case 2 – Streaming Ingestion of Social Graph

trivadis
makes IT easier.

# Streaming Ingestion of Social Network Graph

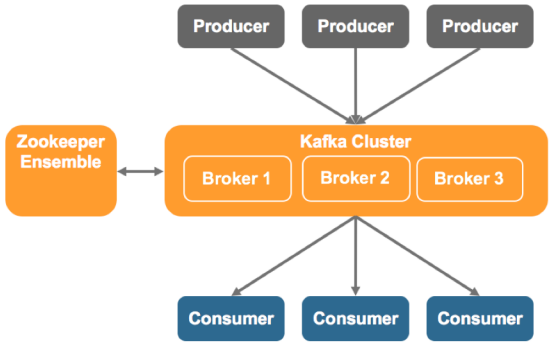



trivadis
makes IT easier.

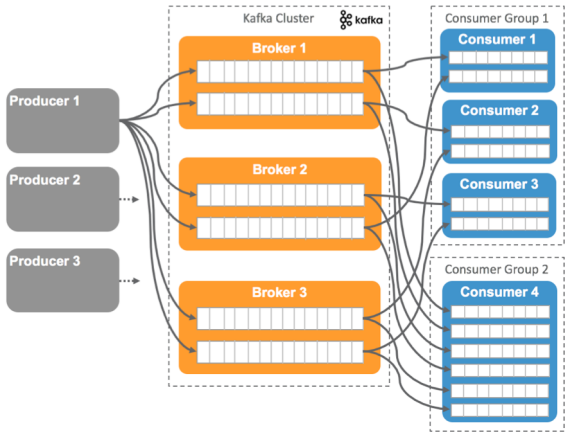
# Apache Kafka – A Streaming Platform

kafka Streams

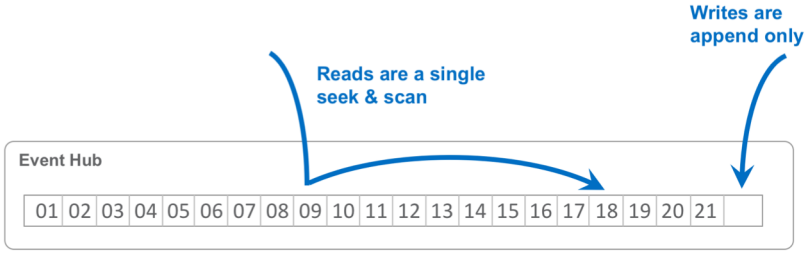
## High-Level Architecture



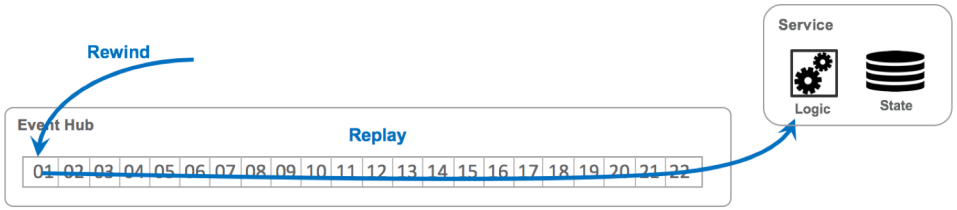

## Scale-Out Architecture




## Distributed Log at the Core




## Logs do not (necessarily) forget




trivadis
makes IT easier.

# Graph Ingestion through DataStax Java Driver

Tweet-to-Graph

Similar to programming with JDBC: execute your gremlin statement by creating it in a String variable => also as error prone as with JDBC ;-)

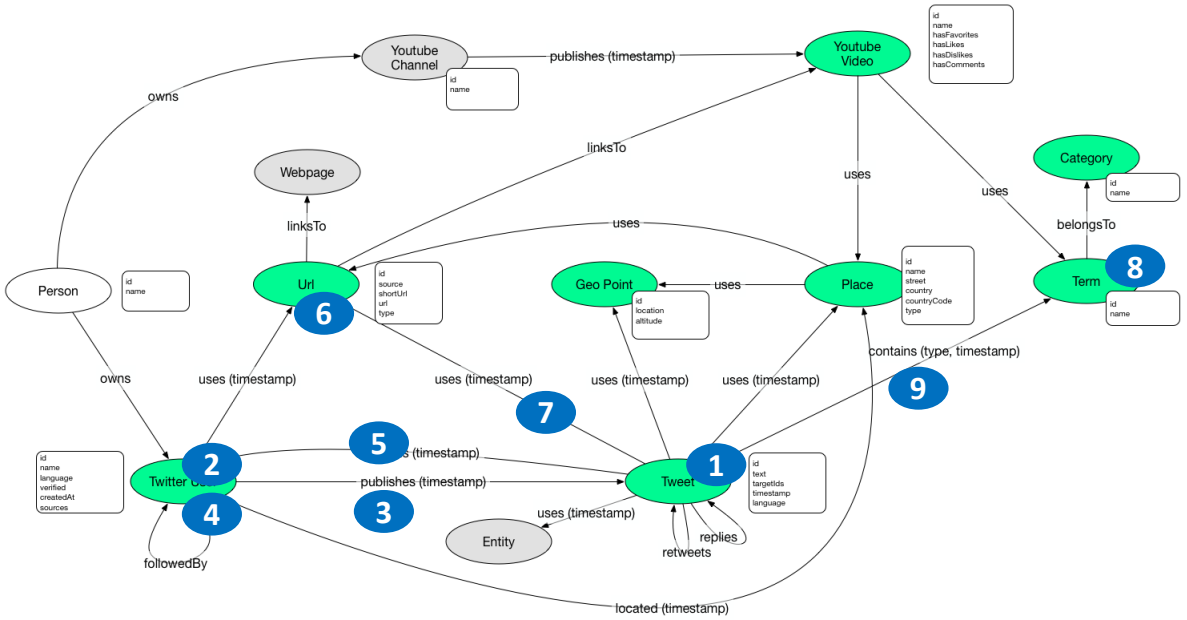
fluent Java-based API available with version of DSE (5.1)

```
String stmt = ""
    + "Vertex v" + "\n"
    + "GraphTraversal gt = g.V().has(b.vertexLabel, b.propertyKey, b.propertyKeyValue)" + "\n"
    + "if (!gt.hasNext()) {"  + "\n"
    + " v = graph.addVertex(label, b.vertexLabel, b.propertyKey, b.propertyKeyValue)" + "\n"
    + "} else {" + "\n" ...

Map<String,Object> map = ImmutableMap.<String,Object>of("vertexLabel", vertexLabel, "propertyKey",
propertyKey, "propertyKeyValue", propertyKeyValue);

vertex = session.executeGraph(new SimpleGraphStatement(stmt)
    .set("b"
    , ImmutableMap.<String,Object>builder().putAll(map).putAll(propertyParams.getRight()).build()
    )).one().asVertex();
```

trivadis
makes IT easier.

# Add information for new Event (tweet) to Graph – what it really means ….

1. Create new **Tweet** vertex and add retweets/replies edges
2. Create/Update **TwitterUser** vertex
3. Create new **publishes** edge
4. For Each Mention, Create/Update **TwitterUser** vertex
5. For each Mention, Create **mentions** edge
6. For each Link, Create/Update **Url** vertex
7. For each Link, Create **uses** edge
8. For each HashTag, Create/Update **Term** vertex
9. For each HashTag, Create **contains** edge
10. ….




trivadis
makes IT easier.

# Could we solve it by modeling differently?

# Single Insert into Social Network Graph

```
GraphTraversal gt = g.V().has(b.vertexLabel, b.propertyKey, b.propertyKeyValue)
if (!gt.hasNext()) {
  v = graph.addVertex(label, b.vertexLabel, b.propertyKey, b.propertyKeyValue)
} else {
  v = gt.next()
}
v.property("name",b.p
v.property("language"
v.property("verified"
```

```
GraphTraversal gt = g.V().has(b.vertexLabel, b.propertyKey, b.propertyKeyValue)
if (!gt.hasNext()) {
  v = graph.addVertex(label, b.vertexLabel, b.propertyKey, b.propertyKeyValue)
} else {
  v = gt.next()
}
v.property("timestamp",b.
v.property("language",b.p
```

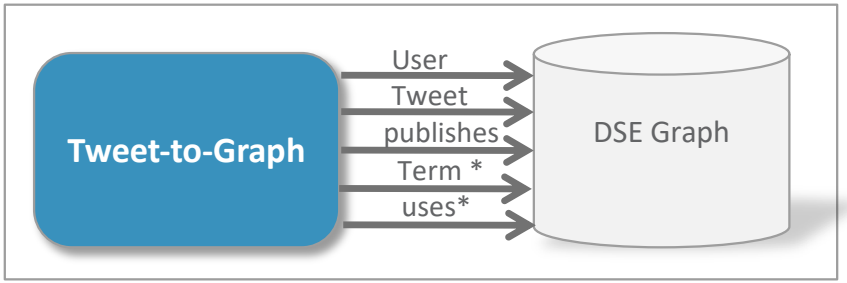
```
GraphTraversal gt = g.V().has(b.vertexLabel, b.propertyKey, b.propertyKeyValue)
```

```
GraphTraversal gt = g.V().has(b.vertexLabel, b.propertyKey, b.propertyKeyValue)
if (!gt.hasNext()) {
  v = graph.addVertex(label, b.vertexLabel, b.propertyKey, b.propertyKeyValue)
} else {
  v = gt.next()
}
v.property("type",b.propertyParam0)
```

```
Vertex from = g.V(f).next(); Vertex to = g.V(t).next(); if
(!g.V(f).out(b.edgeLabel).V(t).hasNext()) { from.addEdge(b.edgeLabel, to) }
```

```
Vertex from = g.V(f).next(); Vertex to = g.V(t).next(); if
(!g.V(f).out(b.edgeLabel).V(t).hasNext()) { from.addEdge(b.edgeLabel, to) }
```

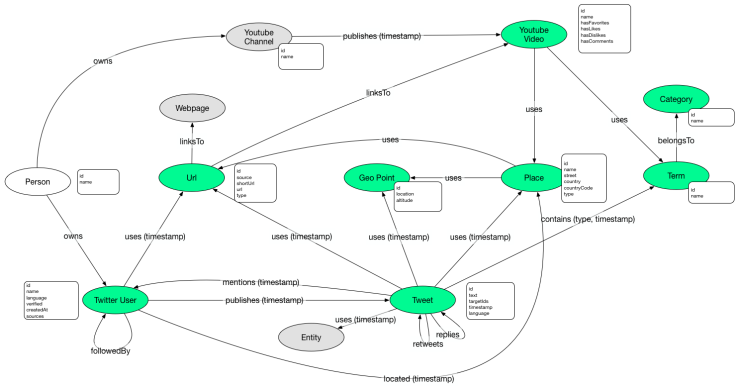
```
Vertex from = g.V(f).next(); Vertex to = g.V(t).next(); if
(!g.V(f).out(b.edgeLabel).V(t).hasNext()) { from.addEdge(b.edgeLabel, to) }
```



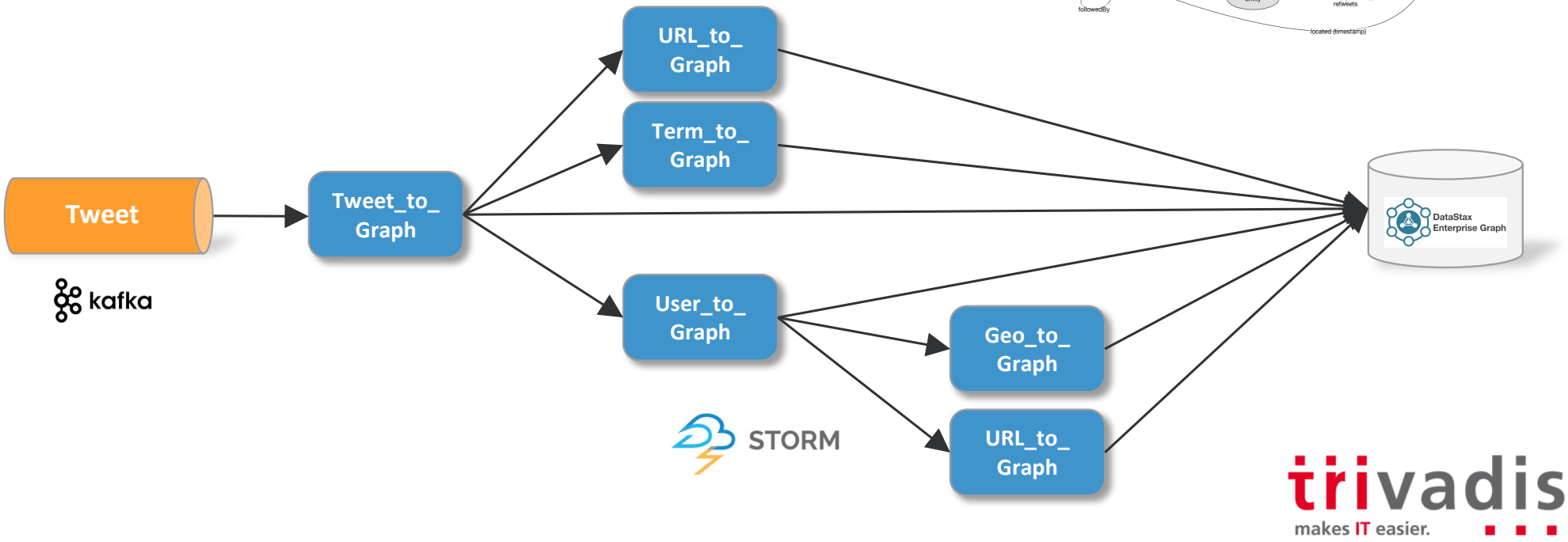

trivadis
makes IT easier.

# 2nd approach - ”Fan out ingestion driven by model”

1st approach => too slow

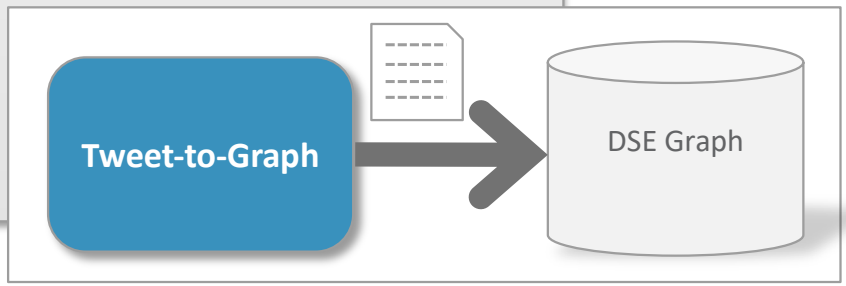

2nd approach => too difficult to implement and maintain




trivadis
makes IT easier.

# 3rd approach - Batch Insert into Social Network Graph

```
user = g.V().has('twitterUser', 'id', bUser.id).tryNext().orElseGet { g.addV('twitterUser').property('id', bUser.id).next() }
user.property('name',bUser.name)
user.property('language',bUser.language)
user.property('verified',bUser.verified)
tweet = g.V().has('tweet', 'id', bTweet.id).tryNext().orElseGet { g.addV('tweet').property('id', bTweet.id).next() }
tweet.property('timestamp',bTweet.timestamp)
tweet.property('language',bTweet.language)
if (!g.V(user).out('publishes').hasId(tweet.id()).hasNext()) {
   publishes = g.V(user).as('f').V(tweet).as('t').addE('publishes').from('f').next()
}
i = 0
Vertex[] term = new Vertex[10]
for (Object keyValue : bTerm.propertyKeyValues) {
   term[i] = g.V().has('term', 'name', keyValue).tryNext().orElseGet {g.addV('term').property('name', keyValue).next() }
  Map<String,Object> params = bTerm.params[i]
  if (params != null)
  for (String key : params.keySet()) {
    term[i].property(key, params.get(key))
  }
  i++
}
Edge[] usesTerm = new Edge[10]
for (i = 0; i < bUsesTerm.count; i++) {
  if (!g.V(tweet).out('uses').hasId(term[i].id()).hasNext()) {
    usesTerm[i] = g.V(tweet).as('f').V(term[i]).as('t').addE('uses').from('f').next()
  }
  Map<String,Object> params = bUsesTerm.params[i]
  if (params != null)
    for (String key : params.keySet()) {
      usesTerm[i].property(key, params.get(key))
    }
  }
return nof
```

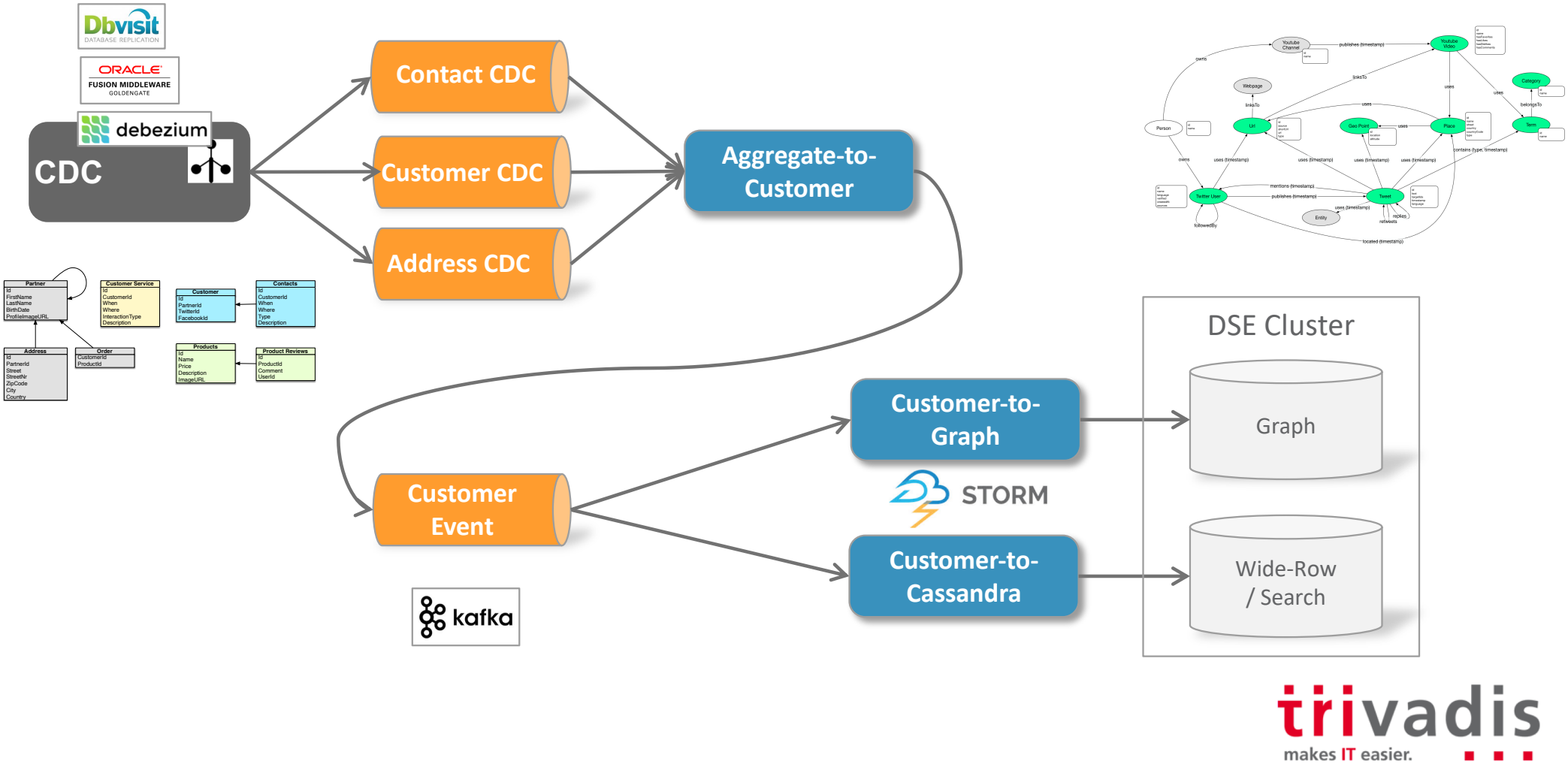



trivadis
makes IT easier.

# Benchmark Single  vs. Scripted Insert

| Test Case | Single Inserts | Scripted (Batch) Insert |
|---|---|---|
| 10 tweets with 10 hashtags each | 569ms - 690ms | 132ms – 172ms |
| 10 tweets with 30 hashtags each | 1311ms – 1396ms | 285ms – 364ms |
| 20 tweets with 30 hashtags each | 2734ms – 3385ms | 640ms – 767ms |
| 20 tweets with 40 hashtags each | 3576ms – 4089ms | 778ms – 992ms |
| 100 tweets with 10 hashtags each | 6340ms – 7171ms | 1673ms - 2428 |

# Process CDC (Change Data Capture) Events

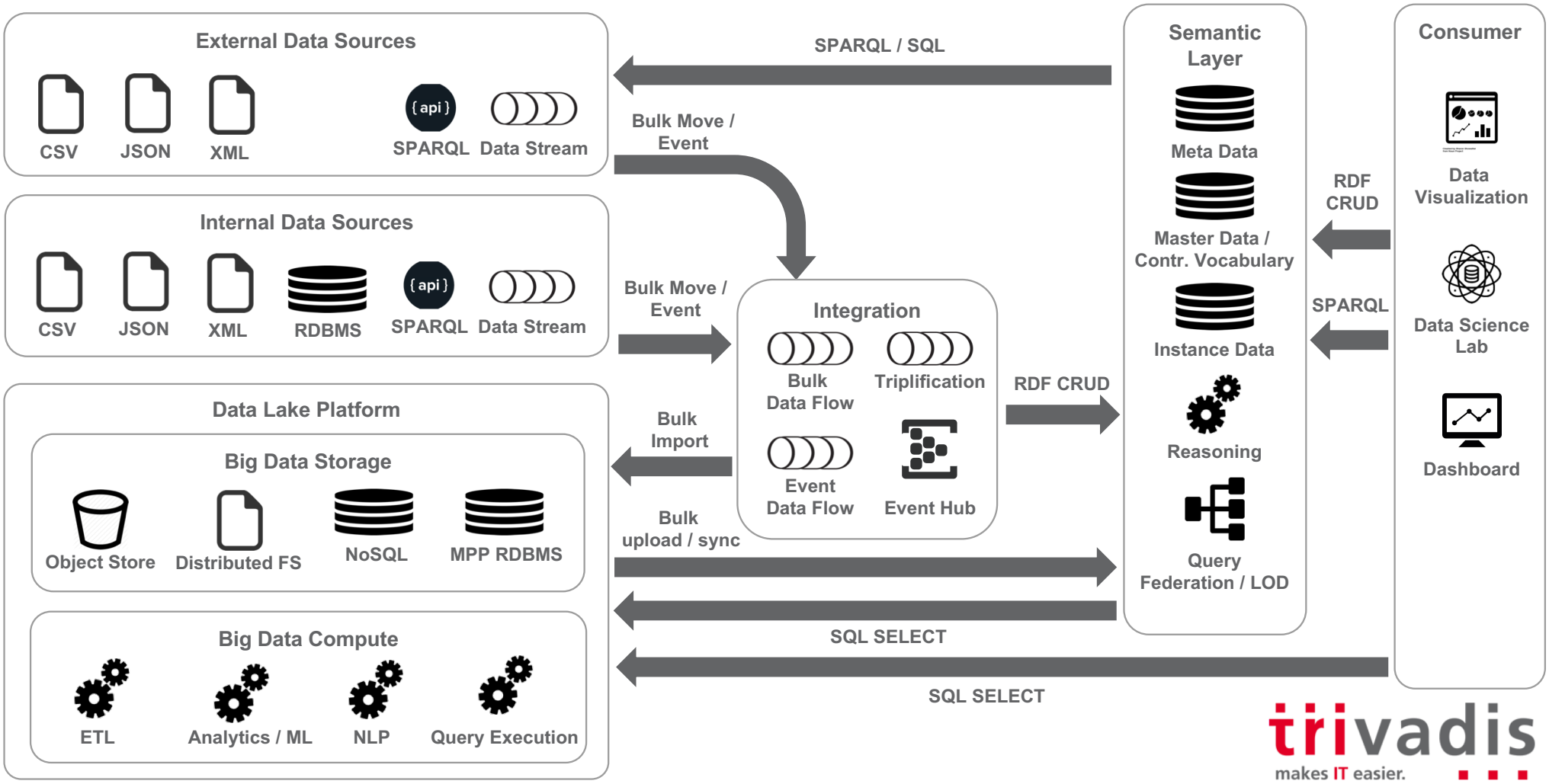



trivadis
makes IT easier.

# Use Case 3: Streaming Ingestion into RDF Graph

trivadis
makes IT easier.

# Semantic Layer 3 – streaming update of Triple store

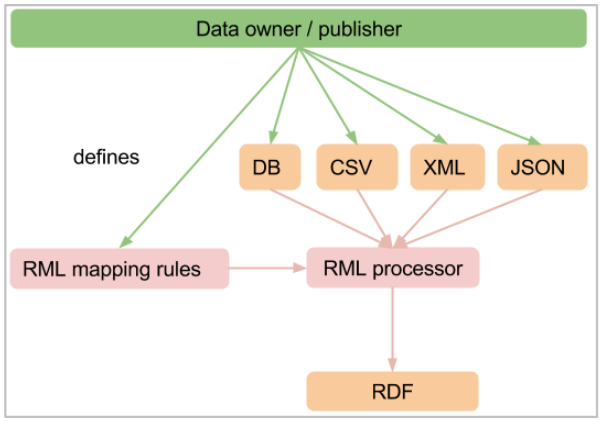


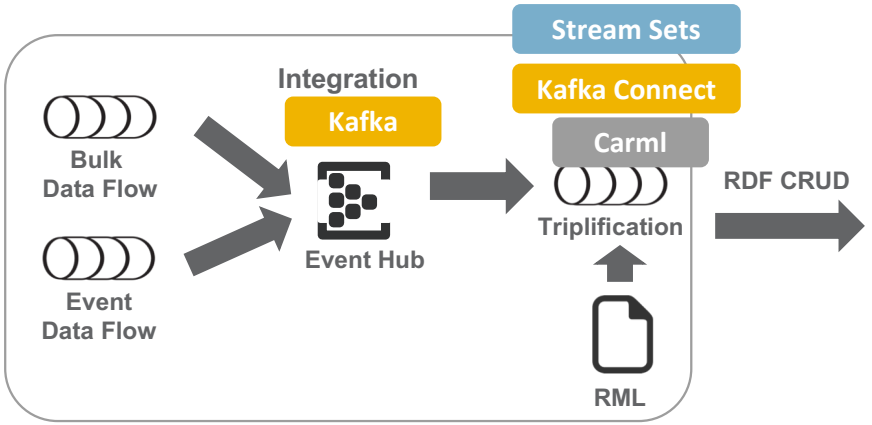

trivadis
makes IT easier.

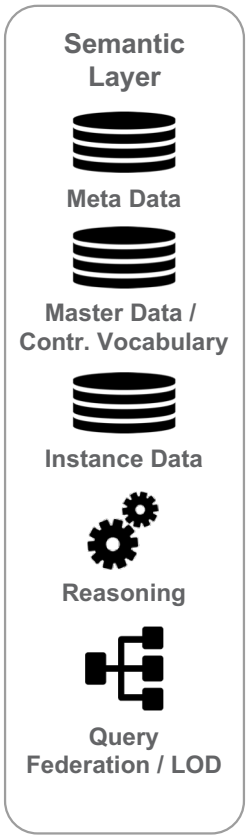
# “Triplification” – 1st option

Using RML (http://rml.io/) and Carml (https://github.com/carml/carml)

```
@prefix rml: <http://semweb.mmlab.be/ns/rml#>.
@prefix ex: <http://example.com/>.
@prefix ql: <http://semweb.mmlab.be/ns/ql#> .
@prefix carml: <http://carml.taxonic.com/carml/> .

<#SubjectMapping> a rr:TriplesMap;
    rml:logicalSource [
        rml:source [
            a carml:Stream;
            carml:streamName "input";
        ];
        rml:referenceFormulation ql:JSONPath;
        rml:iterator "$"
    ];

    rr:subjectMap [
        rr:template "http://example.com/person/{firstName}/{lastName}";
        rr:class ex:Person
    ];

    rr:predicateObjectMap [
        rr:predicate ex:likes;
        rr:objectMap [
            rml:reference "hobbies";
        ]
    ];

    rr:predicateObjectMap [
        rr:predicate ex:hasFirstName;
        rr:objectMap [
            rml:reference "firstName";
        ]
    ].
```




trivadis
makes IT easier.

# “Triplification” – 2nd option

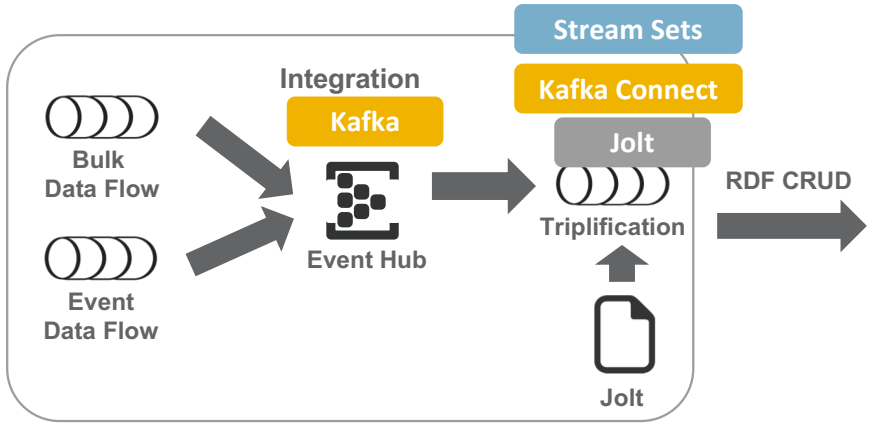
Using Jolt Transformation (https://github.com/bazaarvoice/jolt) to create JSON-LD

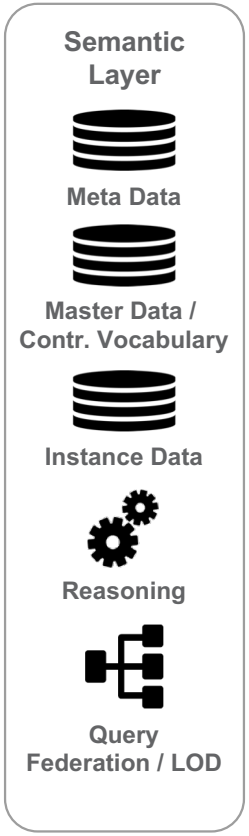
Demo: http://jolt-demo.appspot.com/#inputArrayToPrefix

```
[
  // Flatten an array of photo objects
  //  soup of properties.
  {
    "operation": "shift",
    "spec": {
      "Photos": {
        "*": {
          "Id": "photo-&1-id",
          "Caption": "photo-&1-caption"
          "Url": "photo-&1-url"
        }
      }
```

```
{
  "Photos": [
    {
      "Id": "327703",
      "Caption": "TEST>> photo 1",
      "Url": "http://bob.com/0001/32770
    },
    {
      "Id": "327704",
      "Caption": "TEST>> photo 2",
      "Url": "http://bob.com/0001/32770
    }
  ]
}
```

```
{
  "photo-0-id" : "327703",
  "photo-0-caption" : "TEST>> photo 1",
  "photo-0-url" : "http://bob.com/0001/32
  "photo-1-id" : "327704",
  "photo-1-caption" : "TEST>> photo 2",
  "photo-1-url" : "http://bob.com/0001/32
}
```




trivadis
makes IT easier.

# Summary - Lessons Learnt

trivadis
makes IT easier.

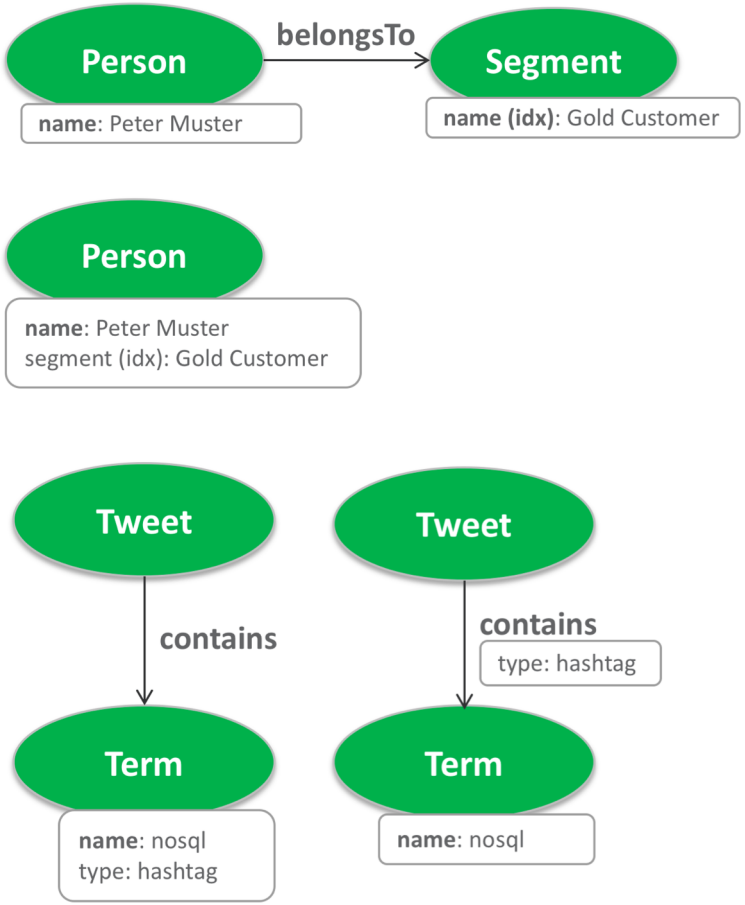
# Summary - Lessons Learnt

## Graph Schema

- Edges can not cross graph schemas
- Vertex or edge? Vertex or property?
- Custom Vertex IDs can be useful
  - idempotent
  - faster on insert (no id generation needed)
  - can lead to hotspots

## Data Ingestion

- Single Nodes & Edge Insert is slow => batching by Groovy Script to speed up
- DSE graph loader works good for batch data loading




trivadis
makes IT easier.


trivadis
makes IT easier.