Large-Scale Graph Processing with Apache Flink

GraphDevroom
FOSDEM ‘15

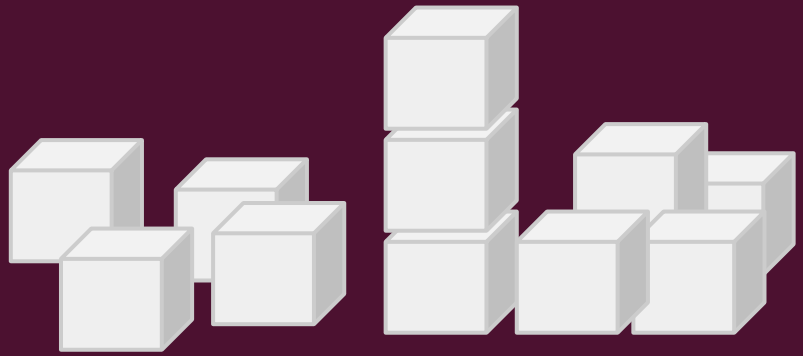
Vasia Kalavri
Flink committer & PhD student @KTH
vasia@apache.org
@vkalavri

Overview

- What is Apache Flink?
- Why Graph Processing with Flink:
  - *user perspective*
  - *system perspective*
- Gelly: the upcoming Flink Graph API
- Example: Music Profiles

Apache Flink

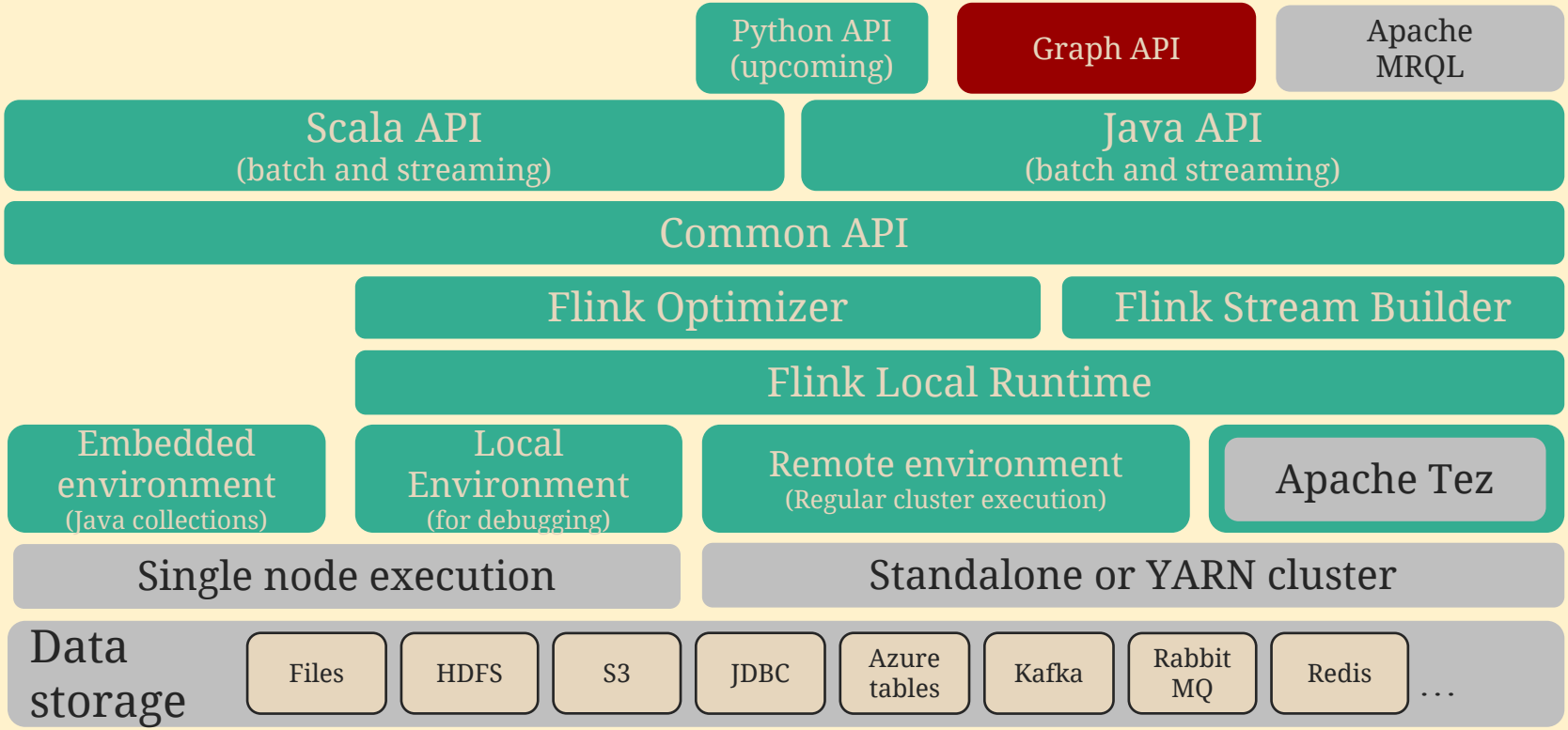
quick intro



What is Apache Flink?

- Large-scale data processing engine
- Java and Scala APIs
- Batch and Streaming Analytics
- Runs locally, on your cluster, on YARN
- Performs well even when memory runs out

The growing Flink stack



Available Transformations

- map, flatMap
- filter
- reduce, reduceGroup
- join
- coGroup
- aggregate
- cross
- project
- distinct
- union
- iterate
- iterateDelta
- ...

Word Count

Java

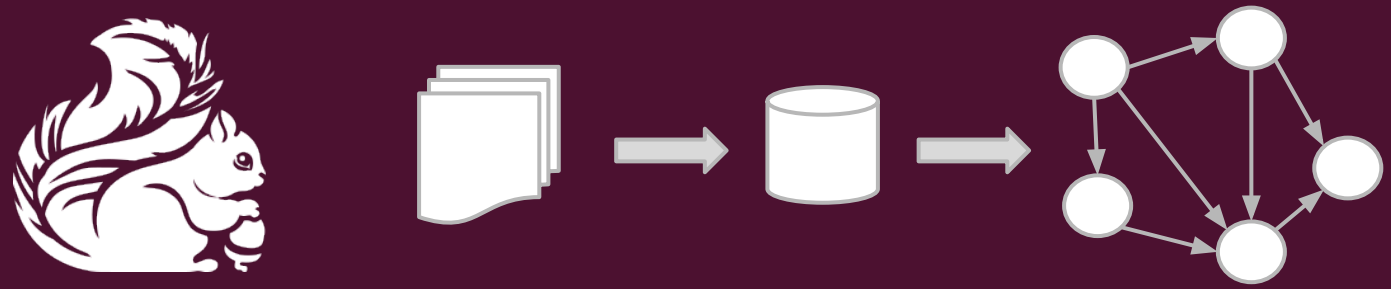
```
DataSet<String> text = env.readTextFile(input);

DataSet<Tuple2<String, Integer>> result = text
         .flatMap((str, out) -> {
              for (String token : value.split("\\W")) {
                   out.collect(new Tuple2<>(token, 1));
              })
         .groupBy(0)
         .aggregate(SUM, 1);
```

Scala

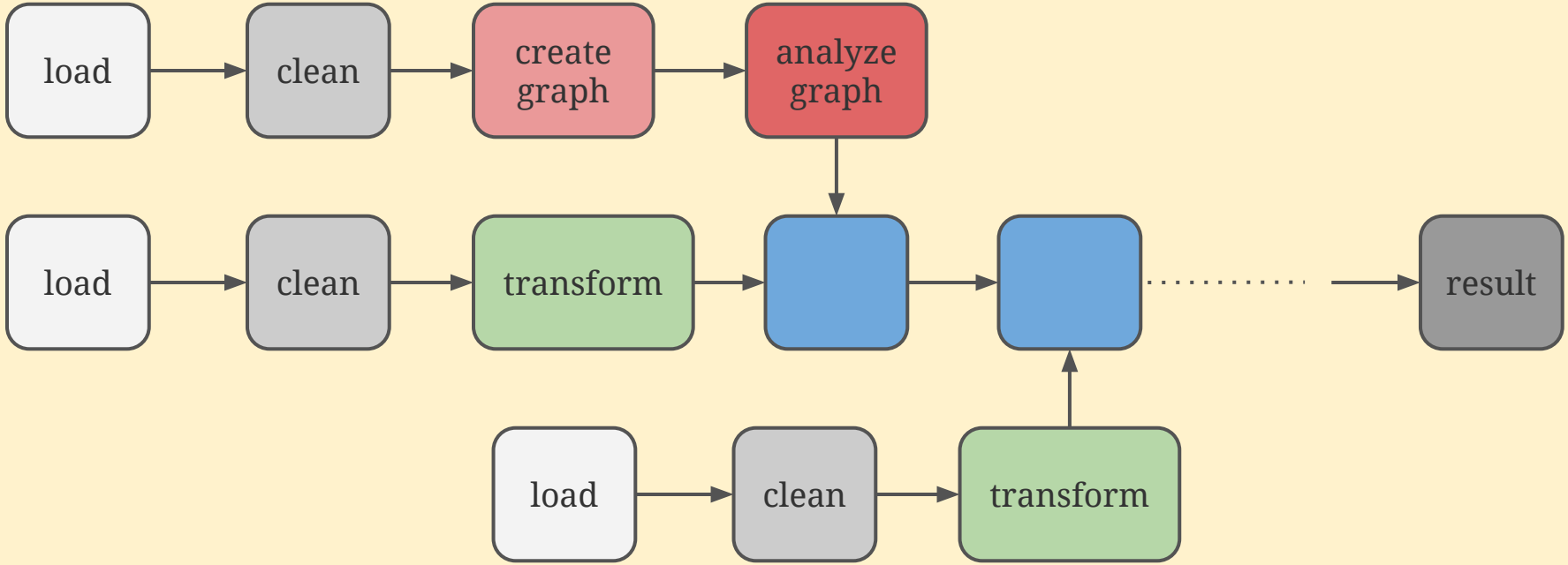
```
val input = env.readTextFile(input);
val words = input flatMap { line => line.split("\\W+")}
                  map { word => (word, 1)}
val counts = words groupBy(0) sum(1)
```

Why Graph Processing with Flink?

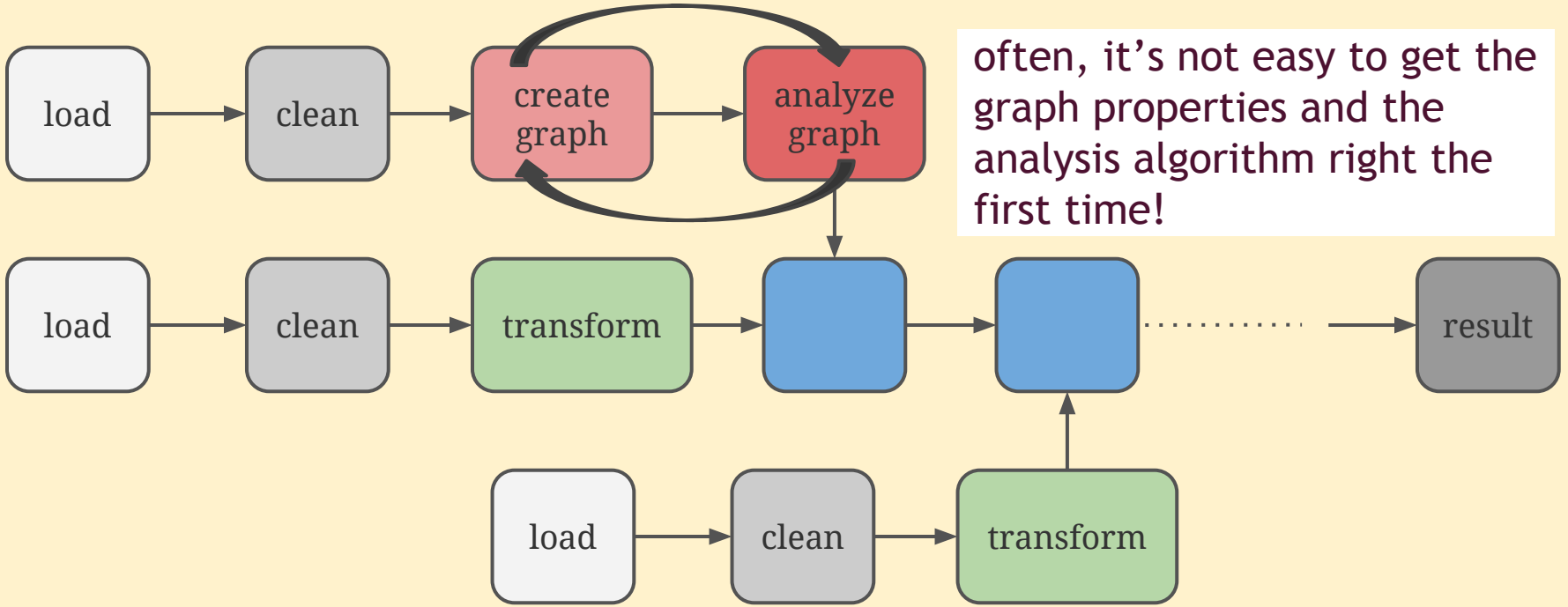
user perspective



Typical graph data analysis pipeline

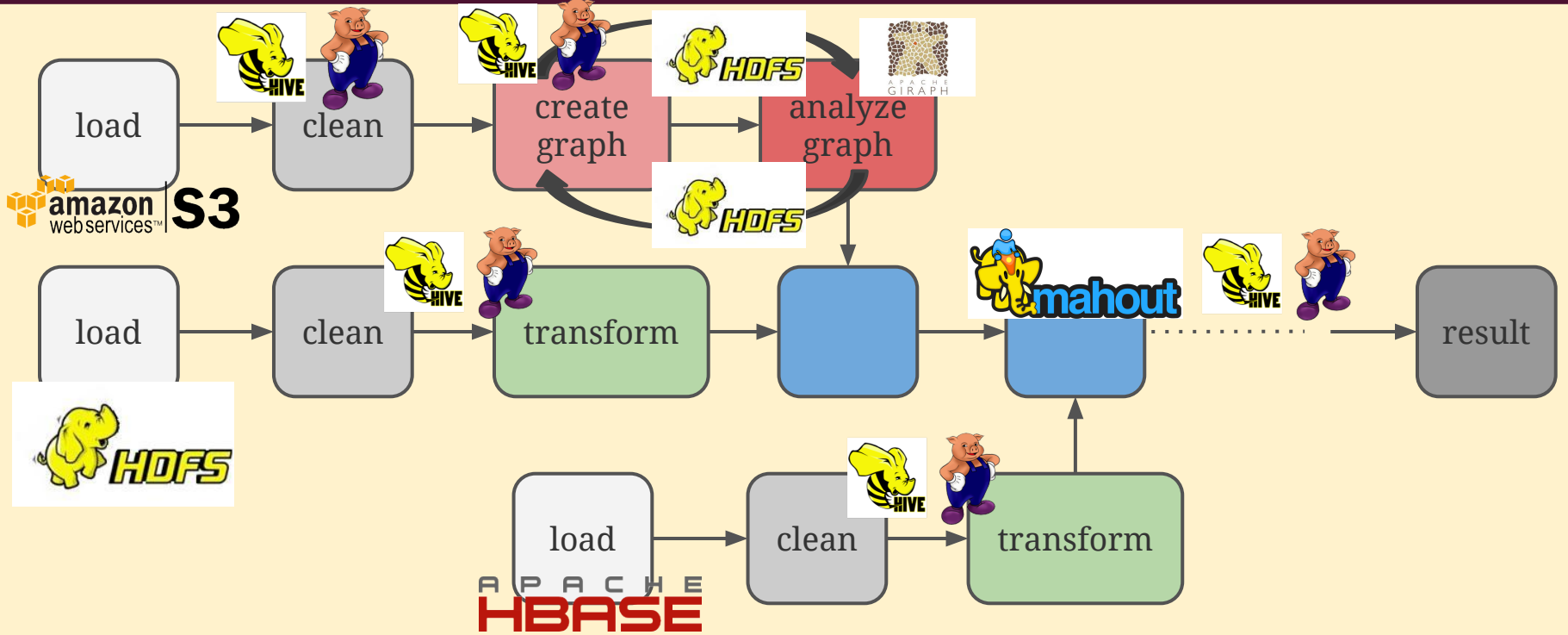


A more realistic pipeline



often, it's not easy to get the graph properties and the analysis algorithm right the first time!

A more realistic pipeline



A more *user-friendly* pipeline

General-purpose or specialized?

general-purpose

- \+ fast application development and deployment
- \+ easier maintenance
- — non-intuitive APIs

specialized

- — time-consuming
  - use, configure and integrate different systems
- — hard to maintain
- \+ rich APIs and features

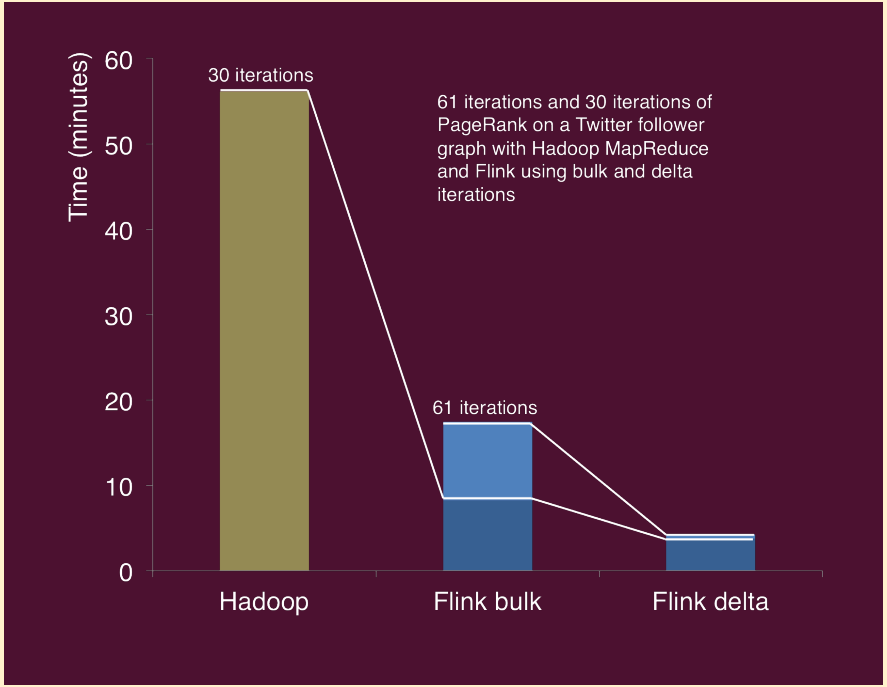
what about performance?

Why Graph Processing with Flink?

system perspective

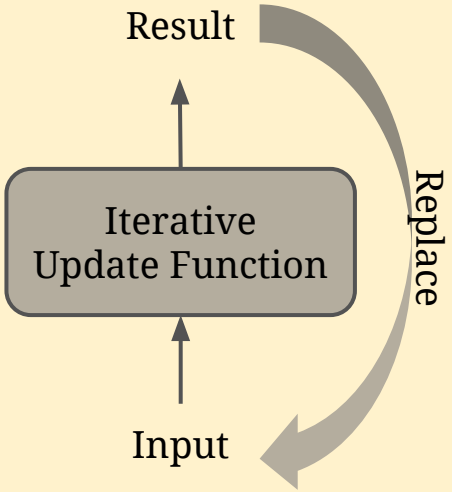
Efficient Iterations

- Fink supports iterations *natively*
  - the runtime is aware of the iterative execution
  - no scheduling overhead between iterations
  - caching and state maintenance are handled automatically

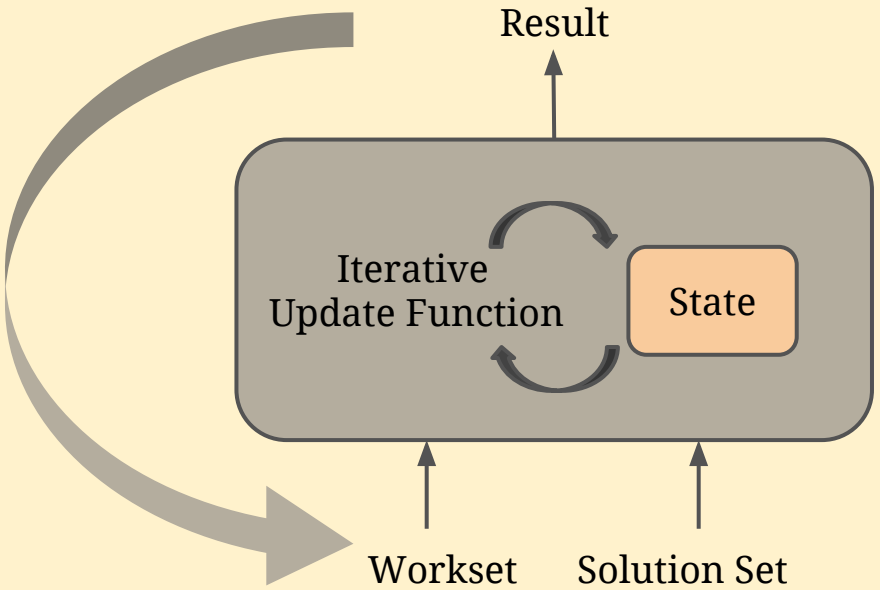


Flink Iteration Operators

Iterate

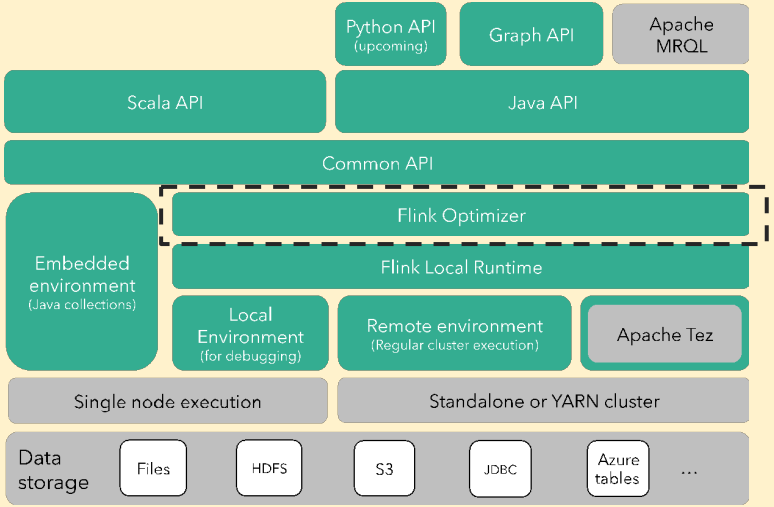


IterateDelta



Flink Optimizer

- The optimizer selects an *execution plan* for a program
- Think of an AI system manipulating your program for you

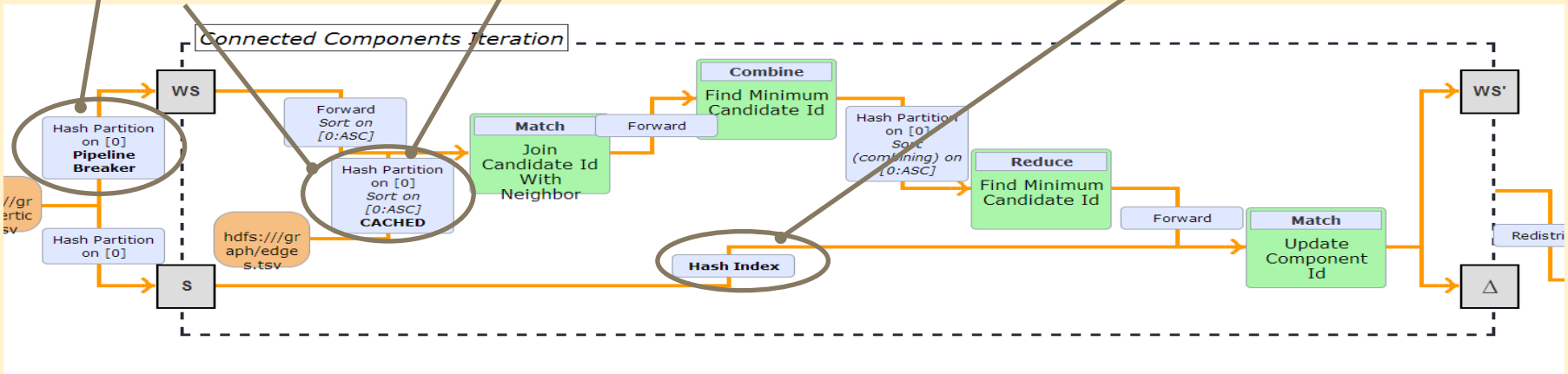


Optimization of Iterative algorithms

Pushing work "out of the loop"

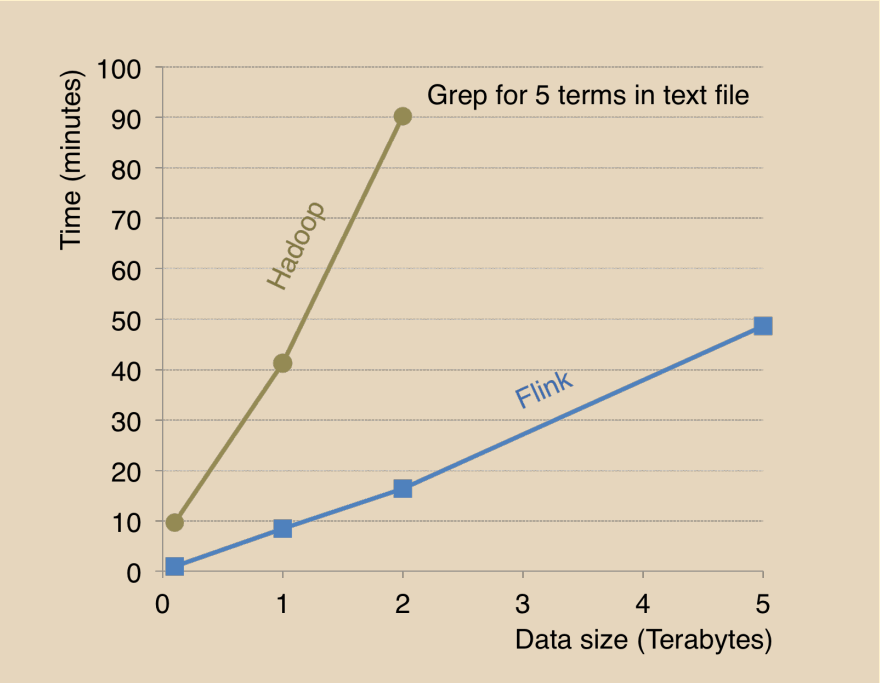
Caching Loop-invariant Data

Maintain state as index



Performance

- in-memory data streaming
- memory management
- serialization framework



Scalability

Computing Recommendations at Extreme Scale with Apache Flink and Google Compute Engine

dataArtisans
Makers of Apache Flink

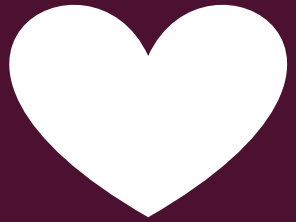
Experiments on Google Compute Engine

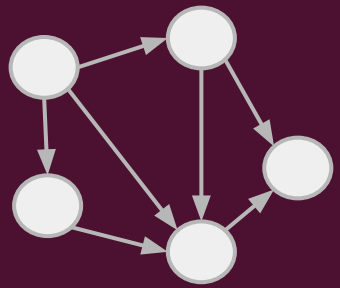
We ran a series of experiments with our ALS implementation on Google Compute Engine. We scaled the matrix to a size of **40 million users**, **5 million items**, and an average of 700 ratings per user, making it a total of **28 billion ratings**.
We ran all experiments with **50 latent factors**, for 10 iterations.

http://data-artisans.com/computing-recommendations-with-flink.html

Gelly

the upcoming Flink Graph API





Meet Gelly

- Java Graph API on top of Flink
- Initial version coming with Flink 0.9
- Can be seamlessly mixed with the standard Flink API
- Easily implement applications that use both record-based and graph-based analysis

Hello, Gelly!

In Gelly, a Graph is simply represented by a DataSet of Vertices and a DataSet of Edges:

```
Graph<String, Long, Double> graph = Graph.fromDataSet(vertices, edges, env);

Graph<String, Long, NullValue> graph = Graph.fromCollection(edges,
        new MapFunction<String, Long>() {
            public Long map(String value) {
                        return 1l;
            }
        }, env);
```

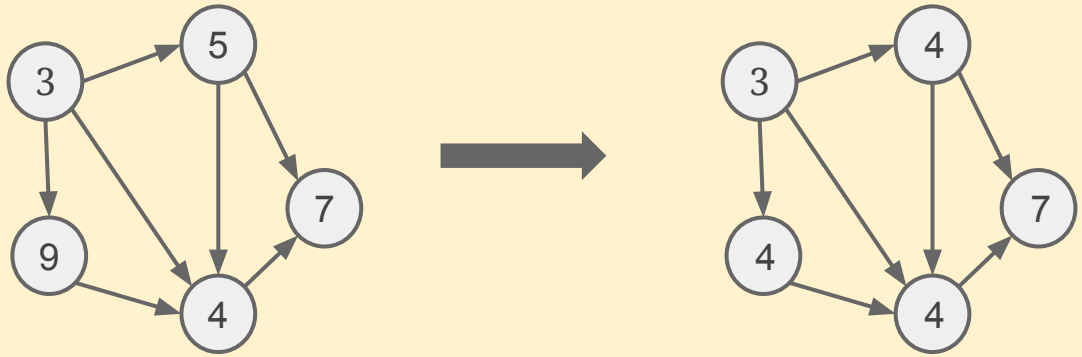
Available Methods

- Graph Properties
  - getVertexIds
  - getEdgeIds
  - numberOfVertices
  - numberOfEdges
  - getDegrees
  - isWeaklyConnected
  - ...
- Transformations
  - map, filter, join
  - subgraph, union
  - reverse, undirected
  - ...
- Mutations
  - add vertex/edge
  - remove vertex/edge

Neighborhood Methods

- Apply a reduce function to the 1st-hop neighborhood of each vertex in parallel

```
graph.reduceOnNeighbors(new MinValue(), EdgeDirection.OUT);
```



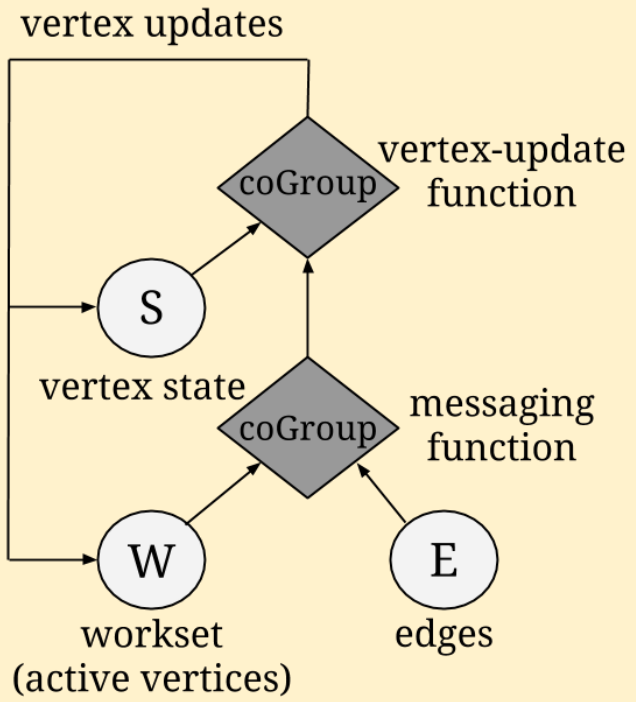
Graph Validation

- Validate a Graph according to given criteria
  - do the edge ids correspond to vertex ids?
  - are there duplicates?
  - is the graph bipartite?

```
edges = { (1, 2), (3, 4), (1, 5), (2, 3), (6, 5) }
vertices = { 1, 2, 3, 4, 5 }

graph = Graph.fromCollection(vertices, edges);
graph.validate(new InvalidVertexIdsValidator()); // false
```

Vertex-centric Iterations

- Wraps the Flink Spargel (Pregel-like) API
- The user only implements two functions
  - VertexUpdateFunction
  - MessagingFunction
- Internally creates a delta iteration



Vertex-centric SSSP

```
shortestPaths = graph.runVertexCentricIteration(
              new DistanceUpdater(), new DistanceMessenger()).getVertices();
```

DistanceUpdater: VertexUpdateFunction

```
updateVertex(K key, Double value,
             MessageIterator msgs) {

  Double minDist = Double.MAX_VALUE;
  for (double msg : msgs) {
    if (msg < minDist)
      minDist = msg;
  }
    if (value > minDist)
      setNewVertexValue(minDist);
}
```

DistanceMessenger: MessagingFunction

```
sendMessages(K key, Double newDist) {

for (Edge edge : getOutgoingEdges()) {
  sendMessageTo(edge.getTarget(),
      newDist + edge.getValue());
}
```

Library of Algorithms

- PageRank
- Single Source Shortest Paths
- Label Propagation
- Weakly Connected Components

Example

User Music Profiles

MAIN

- Browse
- Activity
- Discover
- Radio
- Follow
- Top Lists
- Messages 1
- Play Queue
- Devices
- App Finder
- Digster
- Last.fm
- Pitchfork
- Songkick Concerts

Top Lists

Artists ▾	for me ▾	Tracks ▾	for me ▾
1	Toundra	1	Every age by José González
2	Long Distance Calling	2	Cerveza Beer by Las Ruinas
3	MONO	3	Cooking Up Something Good by Mac Demarco
4	65daysofstatic	4	Carissa by Sun Kil Moon
5	As The Poets Affirm	5	Can't Do Without You by Caribou
6	ef	6	Weight by Mikal Cronin
7	Belle & Sebastian	7	Otitis by Mourn
8	José González	8	Bury Our Friends by Sleater-Kinney
9	Orchestral Manoeuvres In The Dark	9	Droguerías y Farmacias by Sr. Chinarro
10	Las Ruinas	10	Don't Wanna Lose by Ex Hex
11	Trentemøller	11	The Lord's Favorite by iceage
12	Pg.lost	12	All The Rage Back Home by Interpol
13	Mikal Cronin	13	Dark/Light by Mike Simonetti

Problem Description

Input:

- `<userId, songId, playCount>` triplets
- a set of bad records (not to be trusted)

Tasks:

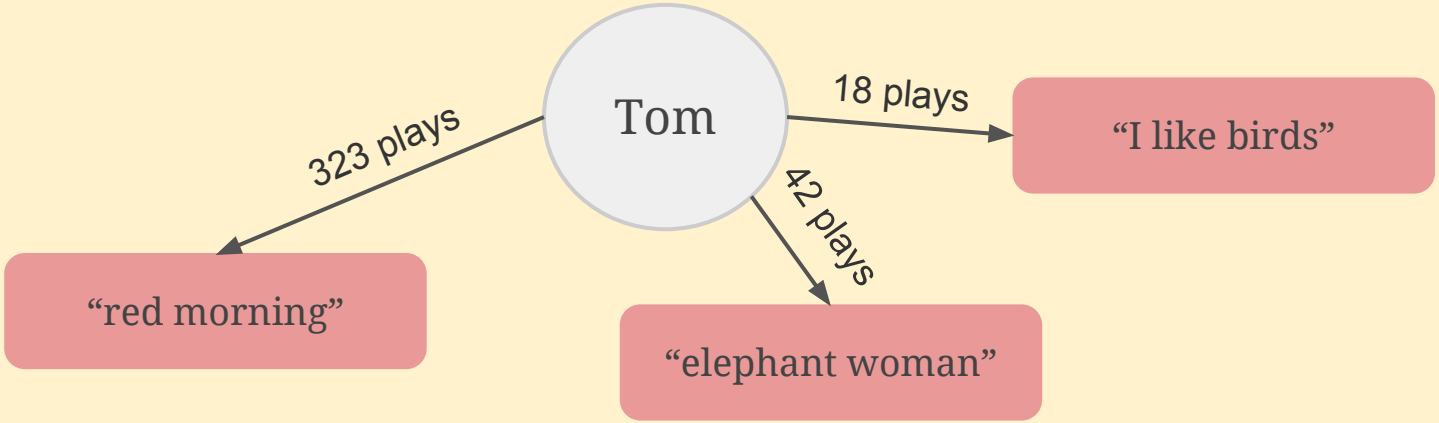
1. **filter** out bad records
2. compute the **top song per user** (most listened to)
3. create a **user-user similarity graph** based on common songs
4. **detect communities** on the similarity graph

1. Filter out bad records

```
/** Read <userID>\t<songID>\t<playcount> triplets */
DataSet<Tuple3> triplets = getTriplets();
/** Read the bad records songIDs */
DataSet<Tuple1> mismatches = getMismatches();
/** Filter out the mismatches from the triplets dataset */
DataSet<Tuple3> validTriplets = triplets.coGroup(mismatches).where(1).equalTo(0)
     .with(new CoGroupFunction {
          void coGroup(Iterable triplets, Iterable invalidSongs, Collector out) {
               if (!invalidSongs.iterator().hasNext())
                  for (Tuple3 triplet : triplets) // this is a valid triplet
                       out.collect(triplet);
          }
```

2a. Compute top song per user

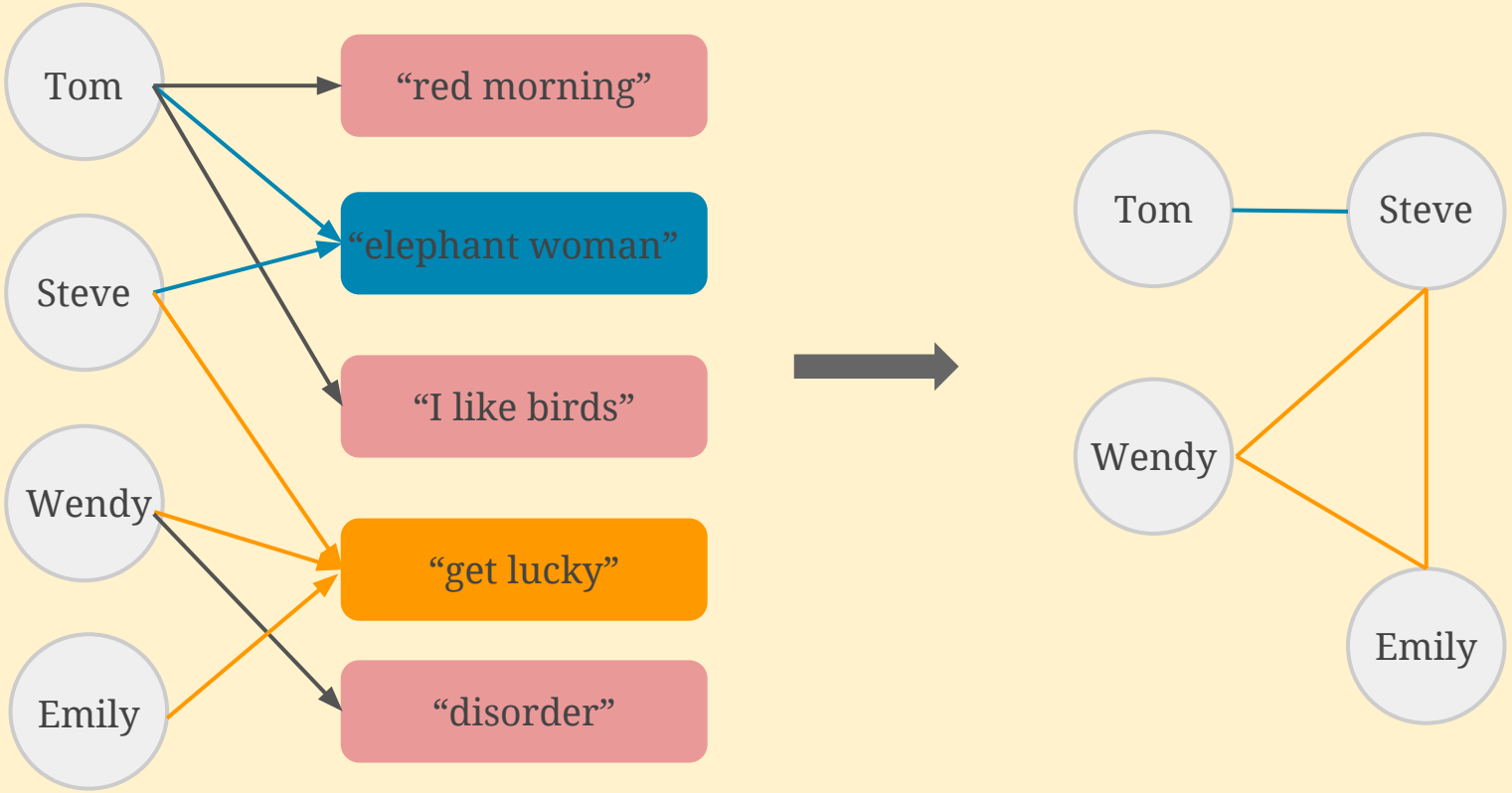
```
/** Create a user -> song weighted bipartite graph where the edge weights correspond
to play counts */
Graph userSongGraph = Graph.fromTupleDataSet(validTriplets, env);

/** Get the top track (most listened) for each user */
DataSet<Tuple2> usersWithTopTrack = userSongGraph
                    .reduceOnEdges(new GetTopSongPerUser(), EdgeDirection.OUT);
```



2b. Compute top song per user

```
class GetTopSongPerUser implements EdgesFunctionWithVertexValue {
    void iterateEdges(Vertex vertex, Iterable<Edge> edges) {
            int maxPlaycount = 0;
            String topSong = "";
            for (Edge edge : edges) {
                if (edge.getValue() > maxPlaycount) {
                    maxPlaycount = edge.getValue();
                    topSong = edge.getTarget();
                }
            }
            return new Tuple2(vertex.getId(), topSong);
        }
    }
```

user-song to user-user graph



3. Create a user-user similarity graph

```
/**Create a user-user similarity graph:
     two users that listen to the same song are connected */
DataSet<Edge> similarUsers = userSongGraph.getEdges().groupBy(1)
          .reduceGroup(new GroupReduceFunction() {
               void reduce(Iterable<Edge> edges, Collector<Edge> out) {
                    List users = new ArrayList();
                    for (Edge edge : edges)
                        users.add(edge.getSource());
                    for (int i = 0; i < users.size() - 1; i++)
                       for (int j = i+1; j < users.size() - 1; j++)
                           out.collect(new Edge(users.get(i), users.get(j)));
               }
          }).distinct();
Graph similarUsersGraph = Graph.fromDataSet(similarUsers).getUndirected();
```

4. Cluster similar users

```
/** Detect user communities using label propagation */
// Initialize each vertex with a unique numeric label
DataSet<Tuple2> idsWithLabels = similarUsersGraph
                    .getVertices().reduceGroup(new AssignInitialLabel());

// update the vertex values and run the label propagation algorithm
DataSet<Vertex> verticesWithCommunity = similarUsersGraph
                    .joinWithVertices(idsWithlLabels, new MapFunction() {
                         public Long map(Tuple2 idWithLabel) {
                              return idWithLabel.f1;
                         }
     }).run(new LabelPropagation(numIterations)).getVertices();
```

Music Profiles Recap

- Filter out bad records : record API
- Create user-song graph : record API
- Top song per user : Gelly
- Create user-user graph : record API
- Cluster users : Gelly

What’s next, Gelly?

- Gather-Sum-Apply
- Scala API
- More library methods
  - Clustering Coefficient
  - Minimum Spanning Tree
- Integration with the Flink Streaming API
- Specialized Operators for Skewed Graphs

Keep in touch!

- Gelly development repository
  http://github.com/project-flink/flink-graph
- Apache Flink mailing lists
  http://flink.apache.org/community.html#mailing-lists
- Follow @ApacheFlink