

Management & Analysis of Big Graph Data

Current Systems and Open Challenges

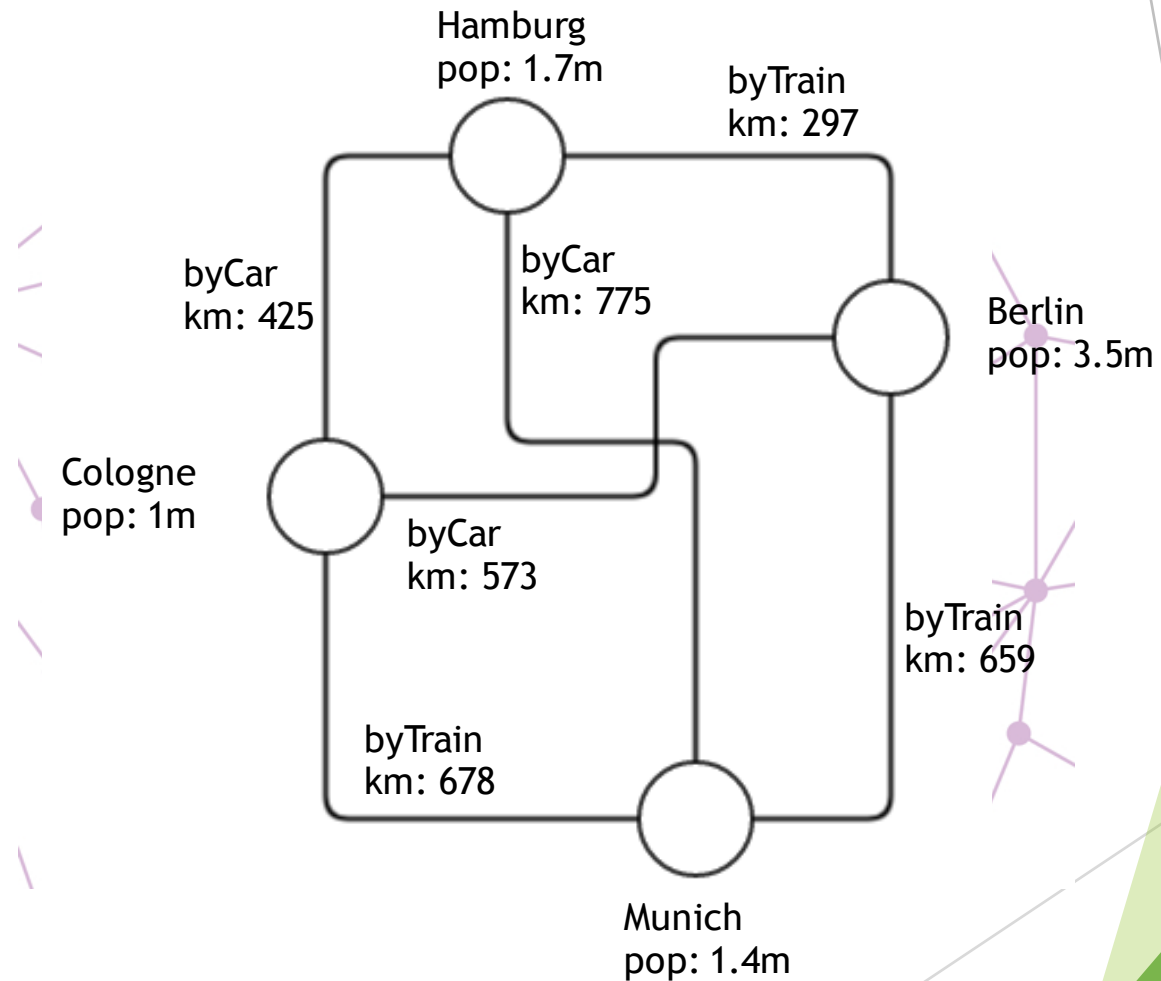
Outline

- ▶ Introduction
- ▶ Graph Databases
- ▶ Graph Processing
- ▶ Graph Dataflow Systems
- ▶ Gradoop
- ▶ Current Research and Open Challenges
- ▶ Conclusion

Introduction

Motivation

- ▶ “Graphs are everywhere”
- ▶ Facebook social graph
 - ▶ 100+ billion entities
- ▶ Twitter follower graph
- ▶ Google maps

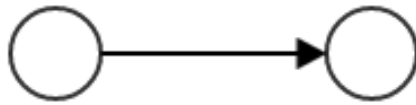


Introduction

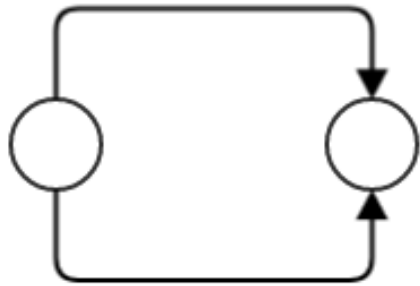
Graphs structures



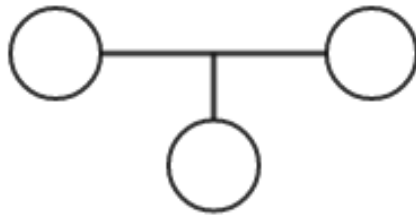
Undirected



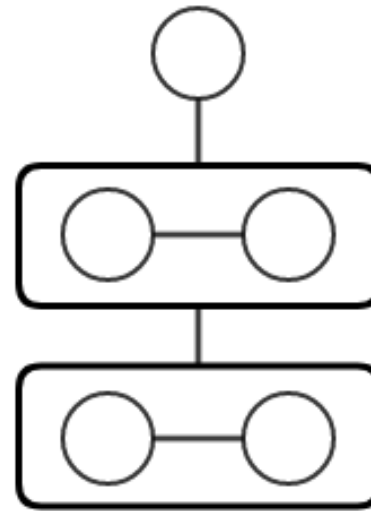
Directed



Directed Multigraph



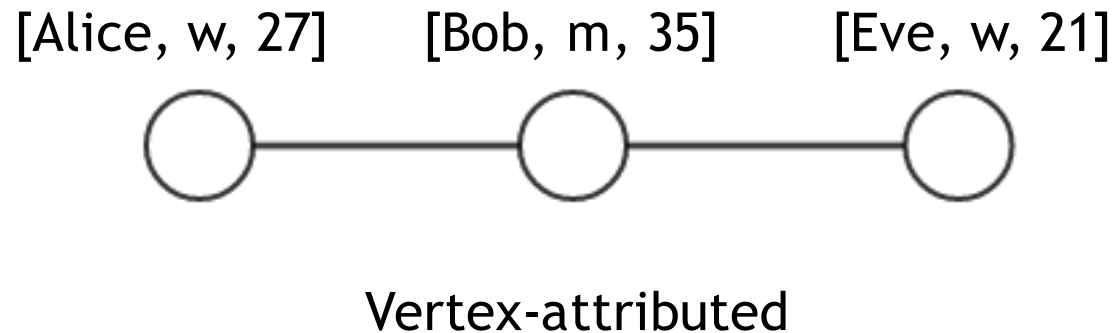
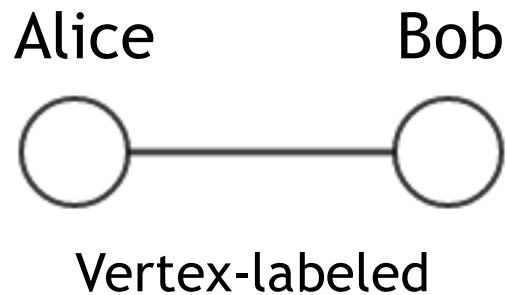
Hyperedge



Hypervertices

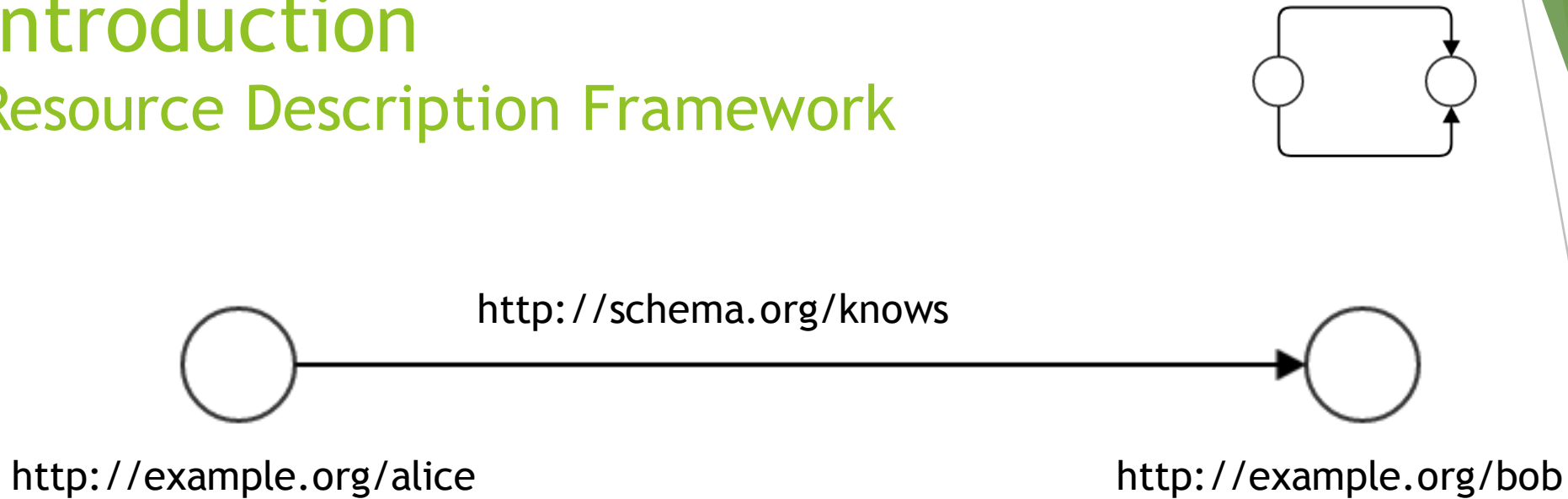
Introduction

Vertex- and edge specific data



Introduction

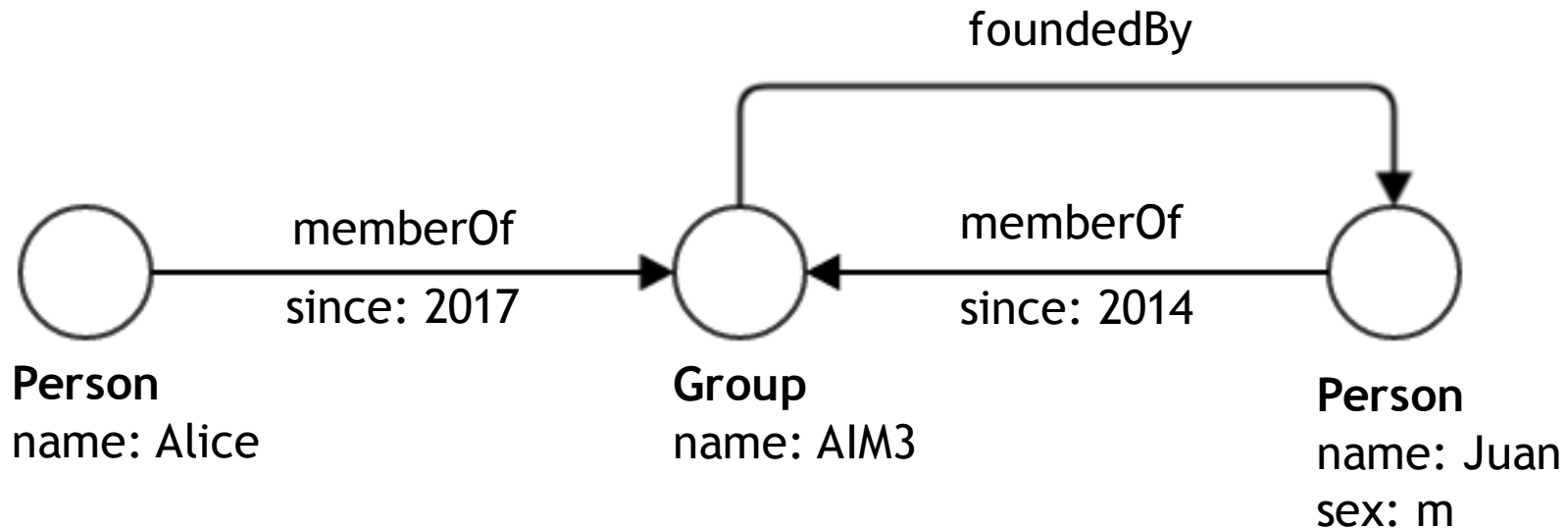
Resource Description Framework



- ▶ Machine-readable data exchange format
- ▶ Subject predicate object syntax
- ▶ Directed labeled multigraph
 - ▶ Edges between edges & vertices allowed
- ▶ `:knows :isA :Relationship`
- ▶ Most popular in the context of semantic web
- ▶ Additional information on handling huge RDF graphs: [2]

Introduction

Property Graph Model



- ▶ Most flexible graph data model
- ▶ Directed labeled multigraph
- ▶ Properties as key value pairs for vertices & edges
- ▶ “Schema-free” ➡ no standardization
- ▶ Property graphs **with** a fixed schema can be translated to RDF

Graph Databases

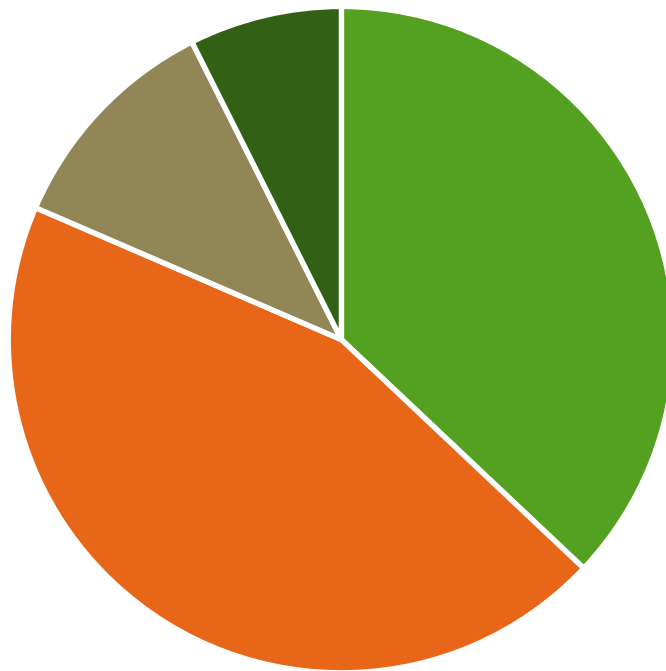
Recent graph database systems

- ▶ Based on graph data model
- ▶ Provide storage, query processing & graph-based operators such as neighbourhood traversal or pattern matching
- ▶ Differ based on:
 - ▶ Supported data models
 - ▶ Application scope
 - ▶ Storage techniques
 - ▶ Query language support
- ▶ For a complete overview see [1]

Graph Databases

supported data models & application scope

- ▶ Majority supports either RDF or PGM



■ RDF / SPARQL ■ PGM ■ Both ■ Generic

- ▶ Most focus on CRUD, query processing & transaction
- ▶ Some graph databases already have support for graph analytics



IBM System G





Oracle Big Data



Graph Databases

storage techniques & query language support

- ▶ Most use native storage specifically tailored to graph data
 - ▶ Typical: adjacency lists
- ▶ In contrast: some systems implement their database on top of an alternative data model (e.g. relational)
- ▶ BUT: Storage approach in general no hint for performance
- ▶ Half of the tested systems support partitioned graph storage & distributed query processing
- ▶ According to *Angles*, four operations specific to graph database query languages should be supported [3] :
 - ▶ Adjacency
 - ▶ Reachability
 - ▶ Pattern Matching
 - ▶ Aggregation Queries
- 
- ▶ Common limitation: not possible to execute any further graph operations on query results

Graph Processing

- ▶ There are many graph processing systems, i.e.
 - ▶ Apache Flink
 - ▶ Apache Spark
 - ▶ Google Pregel
- ▶ These systems provide graph processing capabilities but the kind of data we are dealing is based on graph centric processing or vertex centric processing. How many of you heard about weakly connected graph?
- ▶ If not then I can explain it now: A graph is weakly connected if we don't able to visit all vertex using one path. And the one which is being ignored by that path is weakly connected component.

Graph Processing

Graph Architecture

- ▶ The graph architecture uses two nodes,
 - ▶ Master node : For coordination
 - ▶ Worker nodes: For actual distribution
- ▶ The input graph partitioned among all workers nodes using Hash based partitioning on vertex label. And a worker node stores for each of vertex their vertex value all outgoing and incoming edges values.

Graph Processing

Think like a vertex

- ▶ The think like vertex or vertex centric approach is firstly used by Google Pregel. To perform this task we have write function named vertex compute function, and this function have these three steps.
 - ▶ Read all incoming message
 - ▶ Update the internal vertex
 - ▶ Send message to its neighbors
- ▶ As we already discuss, each vertex has local view of itself and its neighborhood. The vertex function are executed in super steps. Worker node execute the compute functions for all active vertices and make them inactive when `voteToHalt()` function executed and get all of their output messages.

Graph Processing

Think like a vertex

- ▶ **Variants:** There are many various vertex-centric graph processing systems provide specific features and optimizations techniques, i.e. mutate the graph or reduce the network traffic and etc.
- ▶ **Aggregation:** Many graph algorithms need global knowledge in term of aggregated values such as the number of the vertex and total sum of all vertex values.
- ▶ Many frameworks require a user function that will run on master node and have two phases.
 - ▶ `aggregateValue()` : aggregate all values
 - ▶ `getAggregatedValue()` : to access all the values.

Graph Processing

Think like a vertex

- ▶ **Reduce network communication:** In this approach we need to reduce the message between worker nodes. i.e. if a worker node have multiple message towards same vertex, and then it potentially combined into single message.
 - ▶ Gather-Apply-Scatter(GAS).
- ▶ **Asynchronous execution:** There are many nodes in our graph and every one need to be executed, but if there is a node which take more execution time. So other nodes have to wait a bit. So using asynchronous approach other nodes don't have to wait for executions.
- ▶ **Graph mutation:** Transformational algorithms such as graph coarsening or computing the minimum spanning tree need to modify the graph structure during the execution.

Graph Processing

Think like a graph

- ▶ Instead of writing a compute function executed on each vertex, in a graph- /partition-centric model.
- ▶ the user provides a compute function that takes all vertices managed by a worker node as input. The vertices of worker node n are called internal vertices to n .
- ▶ On each worker node n we then create a copy of each vertex that is not internal to n , but is directly connected to an internal vertex of n , known as boundary vertices.
- ▶ Each worker node executes its user-defined function and afterwards sends messages from all boundary vertices to their internal representation.

Graph dataflow systems

- ▶ The distributed in-memory data flow systems such as Apache Spark, Apache Flink or Naiad provide general-purpose operators (e.g., map, reduce, filter, join) to load and transform unstructured and structured data as well as specialized operators and libraries for iterative algorithms (e.g., for machine learning and graph analysis).
- ▶ We will discuss graph analytics on distributed data flow systems using Apache Flink as a representative system .We briefly introduce Apache Flink and its concept for iterations and will then focus on Gelly, a graph processing library integrated into Apache Flink. Gelly implements the Scatter-Gather and Gather-Sum-Apply programming abstractions for graph processing.

Graph dataflow systems

Apache Flink

- ▶ Apache Flink is the successor of the former research project Stratosphere and supports the declarative definition and distributed execution of analytical programs on batch and streaming dataflow .The basic abstractions of such programs are datasets and transformations. A dataset is a collection of arbitrary data objects and transformations describe the transition of one dataset to another one.
 - ▶ A Flink program may include multiple chained transformations. When executed, Flink handles program optimization as well as data distribution and parallel execution across a cluster of machines.
- Picture of code

Graph dataflow systems

Iterations in Apache Flink

- ▶ **Bulk iteration:** Flink's Bulk Iteration can be separated into four phases:
 - ▶ Iteration input
 - ▶ Take input and create new data set
 - ▶ Partial solution for next iteration
 - ▶ Iteration result
- ▶ **Delta iteration:** Flink's Delta Iteration can be separated into two phases:
 - ▶ Initial workset
 - ▶ An initial solution

Graph dataflow systems

Apache Flink Gelly

- ▶ Flink Gelly is a graph library integrated into Apache Flink and implemented on top of its dataset API. Gelly provides a wide set of additional operators to simplify the definition of graph analytical programs.
- ▶ **Graph Representation**
 - ▶ Gelly uses two classes to represent the elements of a graph:
 - ▶ Vertex
 - ▶ Edge.
 - ▶ A Vertex comprises a comparable, unique identifier (id) and a value,
 - ▶ An Edge consists of a source vertex id, a target vertex id and an edge value.

Graph dataflow systems

Apache Flink Gelly

- ▶ **Graph Transformations** : Applying methods on input graph and return new/modified graph. In Gelly it is perform into series of transformation.
 - ▶ **Mutation**: Enable adding and removing of vertices
 - ▶ **Map**: Allow modification by user define transformation function
 - ▶ **SubGraph**: Extract the new graph into user-defined predicates
 - ▶ **Join**: Combination of vertex and edge datasets with additional datasets and deliver updated datasets
 - ▶ **Undirected**: Change graph from directed to undirected i.e. removing and cloning.
 - ▶ **Union/difference**: Merge two graph into one i.e. user define function

Graph dataflow systems

Apache Flink Gelly

► Neighborhood methods

- This method applied to all incident edges and vertices of each vertex to get aggregate values.
 - ReduceOnEdges\neighbors : Aggregation of edge and vertex values by user define methods on pair of values then new dataset will have only one value per vertex.
 - Group ReduceOnEdges\neighbors : Again user-define methods to aggregate values, only if more than one aggregation is required to compute per neighborhood

Graph dataflow systems

Apache Flink Gelly

- ▶ Graph processing

- ▶ Scatter-Gather

- ▶ Known as “signal/collect” model

- ▶ Scatter: produces the messages that a vertex will send to other vertices.

- ▶ Gather: updates the vertex value using received messages.

- ▶ Gelly provides methods for scatter-gather iterations, corresponding to the scatter and gather phases.

- ▶ ScatterFunction

- ▶ GatherFunction

Graph dataflow systems

Apache Flink Gelly

- ▶ Graph processing

- ▶ Gather-Sum-Apply

- ▶ Like in the scatter-gather model, Gather-Sum-Apply also proceeds in synchronized iterative steps, called supersteps.
 - ▶ gather: a map on each $\langle \text{srcVertex}, \text{edge}, \text{trgVertex} \rangle$ that produces a partial value
 - ▶ sum: a reduce that combines the partial values
 - ▶ apply: join with vertex set to update the vertex values using the results of sum and the previous state.

Gradoop

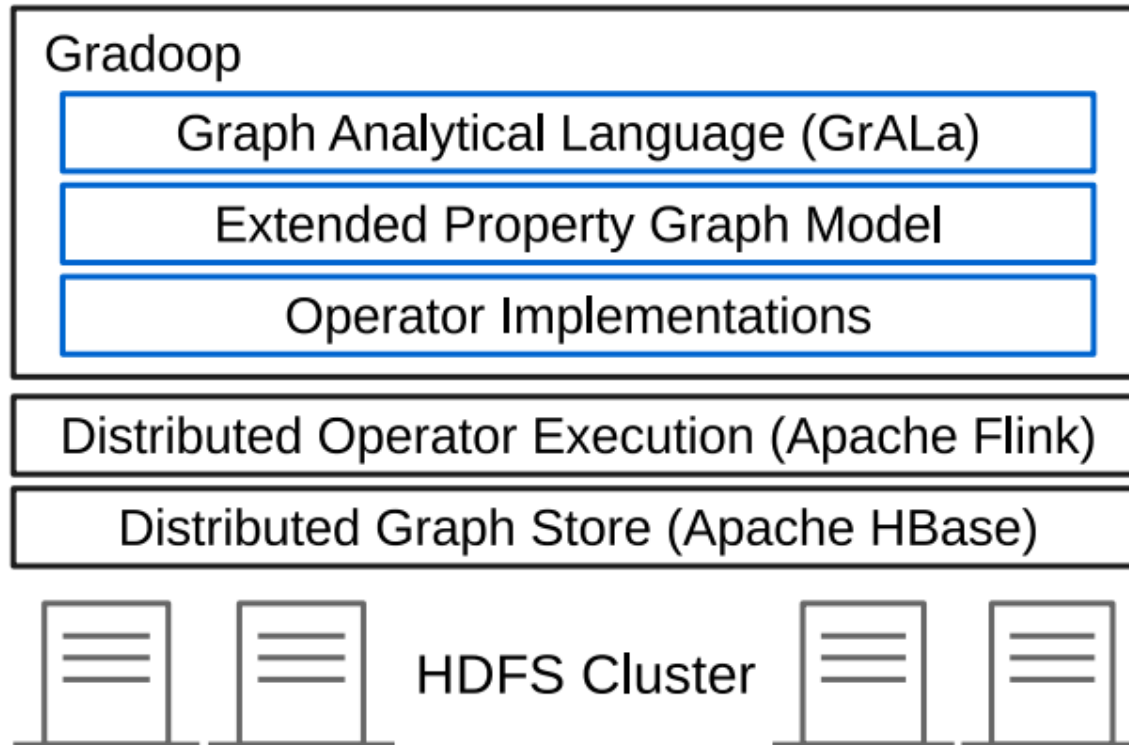
Motivation



- ▶ None of the systems so far has built-in support to manage collections of graphs
- ▶ Graph data model should provide a basic set of declarative operators on graphs & graph collections
- ▶ Operators can be combined to develop advanced graph analysis programs
- ▶ Gradoop resolution:
 - ▶ Framework for scalable graph data management and analytics for large & semantically expressive graphs
 - ▶ Horizontal scalability of storage & processing capabilities through Hadoop-based shared nothing architecture

Gradoop

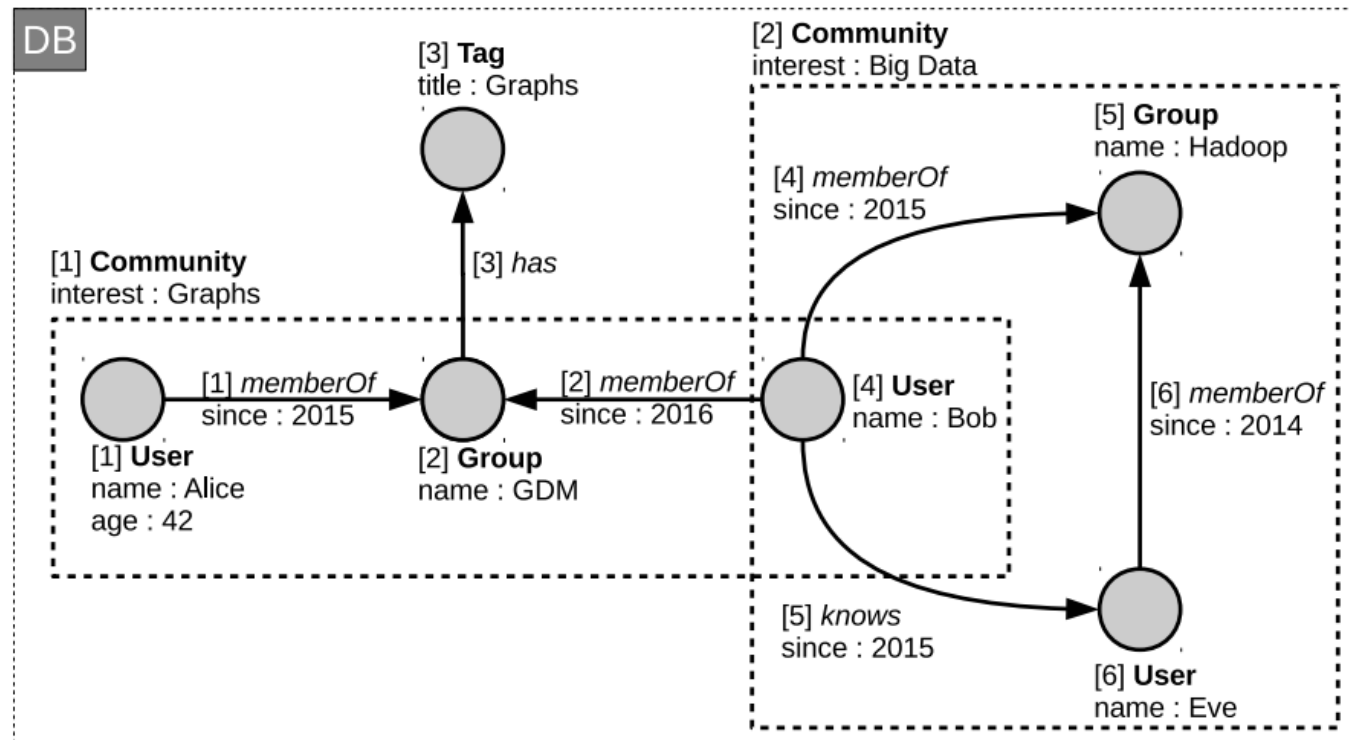
Architecture



Gradoop

Extended Property Graph Model

- Extends PGM by supporting graph collections and composable analytical operators using GrALa



[1] Management and Analysis of Big Graph Data

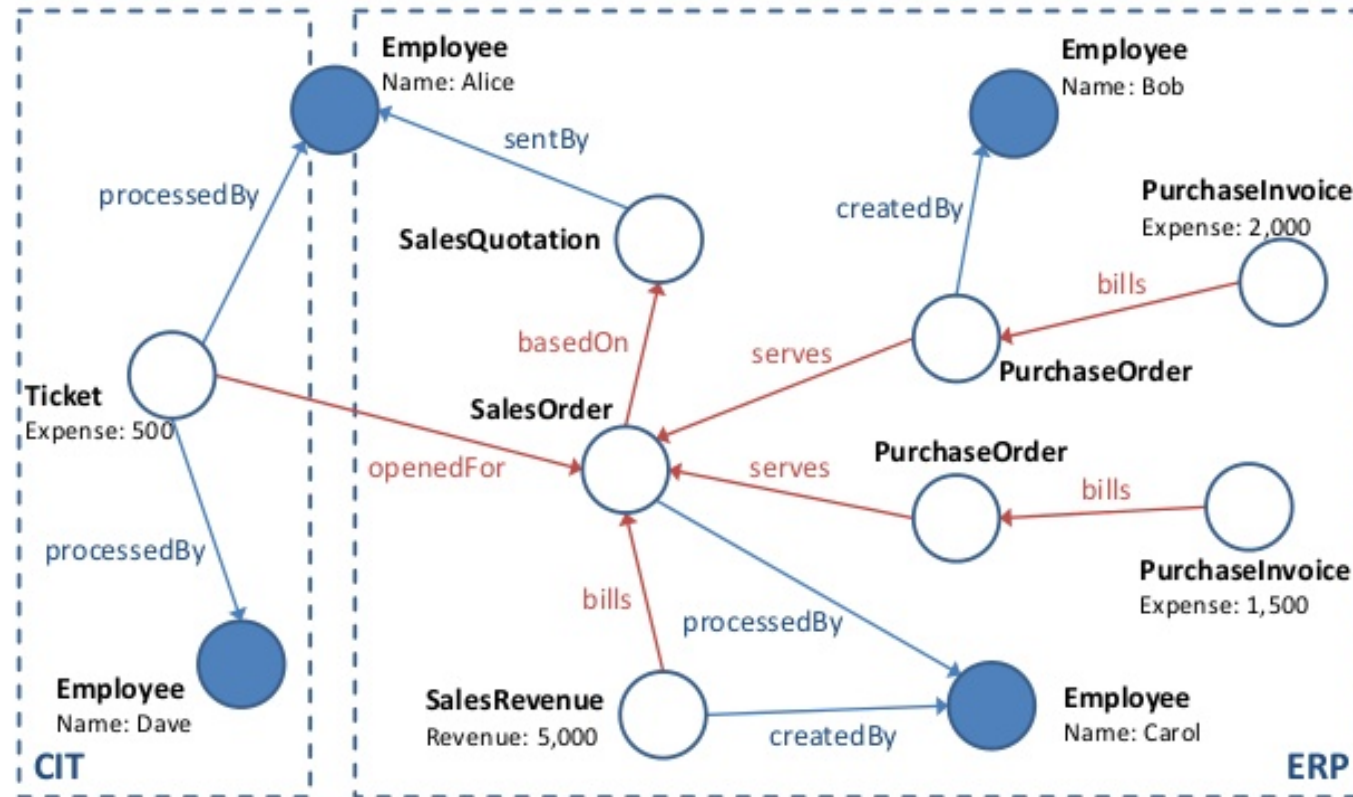
Gradoop Operators

- ▶ GrALA distinguishes among:
 - ▶ Graph operators / collection operators
 - ▶ Unary operators
 - ▶ Binary operators
 - ▶ Auxiliary operators
 - ▶ Aggregation
 - ▶ Grouping
 - ▶ Pattern Matching

Gradoop

Business Intelligence Use case

Business Transaction Graphs



Gradoop

Business Intelligence Use case

► Profit Aggregation:

```
// define profit aggregate function
aggFunc = ( Graph g =>
    g.V.values('Revenue').sum() - g.V.values('Expenses').sum()
)

// apply aggregate function and store results
Btgs = btgs.apply( Graph g =>
    g.aggregate('Profit', aggFunc)
)
```

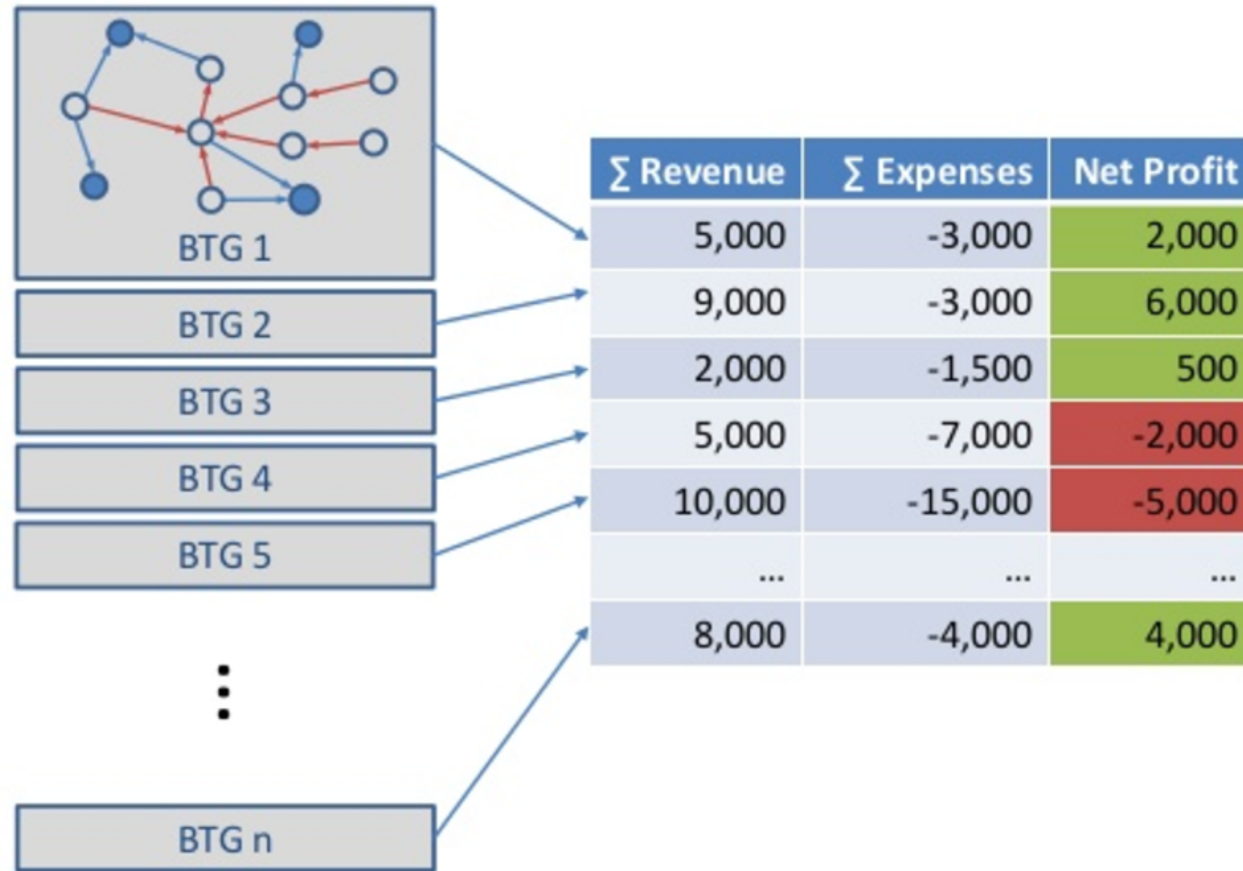
► BTG Clustering based on Profit:

```
// select profit and loss clusters
profitBtgs = btgs.select( Graph g => g['Profit'] >= 0 )

lossBtgs = btgs.difference(profitBtgs)
```

Gradoop

Business Intelligence Use case



Gradoop

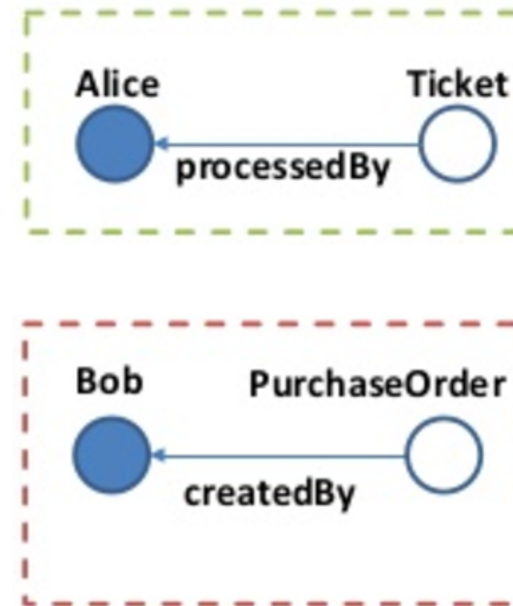
Business Intelligence Use case

- Find cluster characteristic patterns by mining most frequent subgraphs:

```
// apply magic
profitFreqPats = profitBtgs.callForCollection(
    :FrequentSubgraphs, {'Threshold': 0.7}
)

lossFreqPats = lossBtgs.callForCollection(
    :FrequentSubgraphs, {'Threshold': 0.7}
)

// determine cluster characteristic patterns
trivialPats = profitFreqPats.intersect(lossFreqPats)
profitCharPatterns = profitFreqPats.difference(trivialPats)
lossCharPatterns = lossFreqPats.difference(trivialPats)
```



Current research and open challenges

- ▶ The development of systems for graph analytics has made great progress in the past decade but there are still several areas requiring significant further improvement and research.
- ▶ Graph data allocation and partitioning
 - ▶ The efficiency of distributed graph processing substantially depends on a suitable data allocation (partitioning).
 - ▶ This data allocation should enable graph processing with a minimum of internode communication and data transfer while at the same time ensure a good load balancing such that all nodes can be effectively utilized.

Current research and open challenges

Graph data allocation and partitioning

- ▶ Current distributed graph data systems mostly follow technique like
 - ▶ hash-based partitioning
- ▶ This approach achieves an good distribution of vertices and load balancing and does not require a data structure to locate vertices.
- ▶ On the other hand, it frequently assigns neighboring vertices to different partitions leading to poor locality of processing and high communication overhead for many algorithms.
- ▶ The discussion shows that graph data allocation is a challenging problem that is not sufficiently solved with a single solution such as hash partitioning.

Current research and open challenges

Benchmarking and Evaluation of Graph Data Systems

- ▶ The large number of existing systems for analyzing graph data raise the question for potential users of such that which of the systems performs best for which kind of analysis tasks, datasets and platforms.
- ▶ By studying different systems, we find out that their results are mostly not comparable because they use different
 - ▶ real or synthetically generated graph datasets
 - ▶ different sets of queries
 - ▶ analytical tasks
 - ▶ different environments in terms of number of worker nodes and their characteristics such as memory size and number of cores.

Current research and open challenges

Analysis of dynamic graphs

- ▶ Previous approaches for graph analytics focus on static graphs that remain stable.
- ▶ Most graphs like
 - ▶ Social networks : Constantly changing so that analytical processes.
- ▶ Dynamic graph have two categories
 - ▶ Slowly evolving graphs: can be perform offline
 - ▶ Streaming networks: real time analysis is required
- ▶ Number of edges grow stronger then number of vertices in denser network.
- ▶ Despite the relatively large body of previous theoretical and experimental work on dynamic networks, little work has been done for big graph data which take this to further research.

Current research and open challenges

Graph based data Integration and Knowledge

- ▶ As for big data analysis in general, the graph data typically needs to be extracted from the original data sources (e.g.
 - ▶ social networks
 - ▶ web pages
 - ▶ tweets
 - ▶ relational databases, etc.
- ▶ Data should be transformed and cleaned.
- ▶ A particularly challenging kind of graph-based data integration becomes necessary for the generation and continuous maintenance of so-called knowledge graphs providing a large amount of interrelated information about many real-world entities.

Current research and open challenges

Interactive Graph Analysis

- ▶ Interactive graph analysis is currently only supported for query processing with graph databases. While graph analytics with the discussed distributed frameworks is largely batch-oriented.
- ▶ The currently existing separation between interactive query processing with graph databases and batch-oriented graph analytics should thus be overcome by providing all kinds of analysis in a unified, distributed platform with support for interactive and visual analysis.

References

1. Junghans, M., et al.: Management and Analysis of Big Graph Data: Current Systems and Open Challenges (2017)
2. Kaoudi, Z., Manolescu, I.: RDF in the Clouds: A survey. VLDB Journal 24(1) (2015)
3. Angles, R.: A comparison of current graph database models. In: Proc. ICDEW (2012)
4. Junghans, M.: Martin Junghans - Gradoop: Scalable Graph Analytics with Apache Flink, at Flink Forward (2015)

Conclusion

- ▶ Graphs come in different forms and shapes
- ▶ Most graph database systems support either RDF or PGM
- ▶ Used database storing technique no hint for performance
- ▶ Vertex-centric approach vs graph-centric approach needs further assessment for evaluation
- ▶ Apache Flink Gelly is a recent development and therefore needs some testing from the industry to decide whether it is a performant solution or not
- ▶ Gradoop provides framework for storage & processing of horizontal scalable graphs and supports high-level graph analytics using GrALA
- ▶ There is still room for research in the domains of data allocation, benchmarking, dynamic graph, etc. because there is no solid way to compare systems