UNIVERSITY OF COPENHAGEN

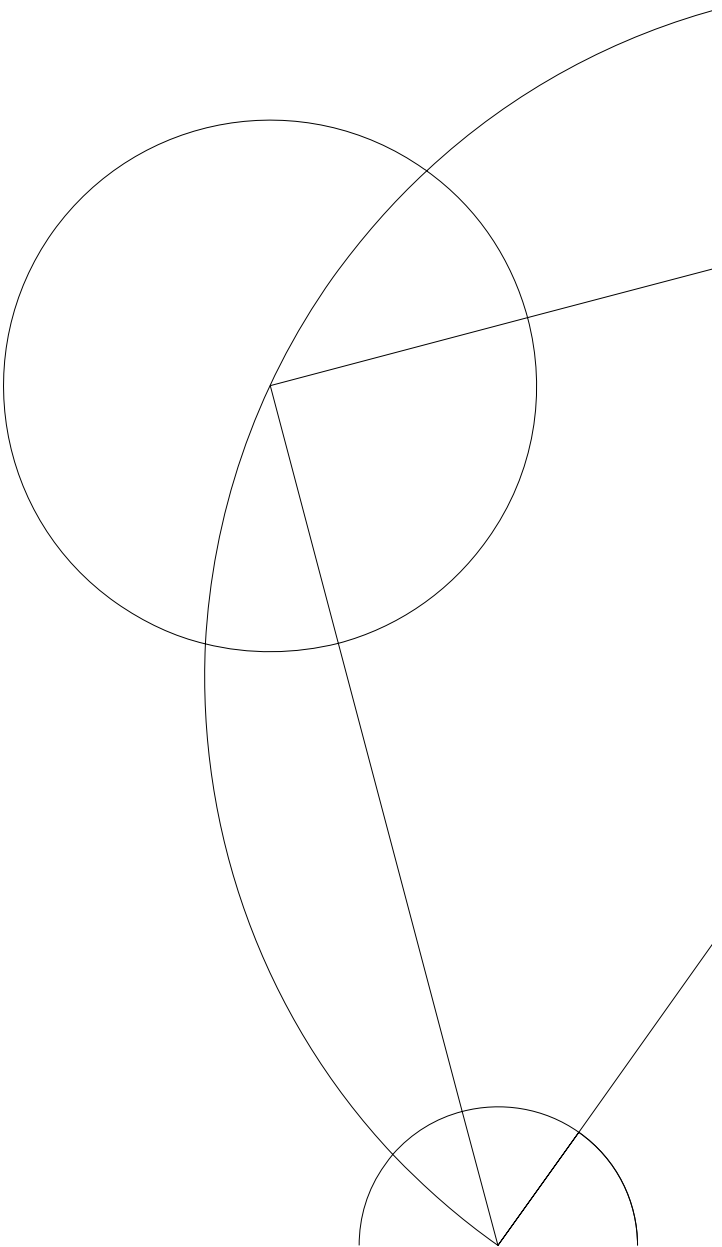
Assignment 1

Subgraph Pattern Matching

Big Data System

Chuang Wu whj433
Yiwen Wang vld772

University of Copenhagen
December 17, 2018



- **Task 1.1: Implement the class *EdgeTripletRDD***
  We implemented:
  - the constructor of *EdgeTripletRDD*.
  - compute() function: Return an iterator of *EdgeTriplet<ED,VD>*.
  - matchEdgePattern() function:
    * input parameters: *EdgePattern* edgePattern.
    * return a *JavaRDD<Match>* matches based on edgePattern.
    * Invokes matchEdgePattern() function from *EdgeTripletPartition* class to get a iterator *Tuple2<VertexId, VertexId>* matchedRDDIter for each *EdgeTripletPartion* in order to generate a list of *Match*.
  - getPartitions() function: Return an array of Partitions under class *EdgeTripletRDD*.
  - static fromEdgeTriplets() function: Generate a *EdgeTripletRDD<ED, VD>* given a *JavaRDD<EdgeTriplet<ED, VD»*.
- **Task 1.2: Implement the class *EdgeTripletPartition***
  We implemented:
  - iterator():
    * Return an iterator of *EdgeTripletPartition*
    * hasNext(): Return *boolean* result by checking if pointer has not pointed to the end of Iterator;
    * next(): Return an *EdgeTriplet<ED, VD>* edgetriplet which currently is pointed by pointer.
  - matchEdgePattern() functuion:
    * Input parameter *EdgePattern* edgePattern.
    * Generate the iterator of *Tuple2<VertexId, VertexId>* matches based on edgePattern.
    * If type of *VertexPredicate* srcpredicate/dstpredicate extracted from edgePattern is *VertexPredicate.Type.ATTR*, find matches by comparing vertices attribute(vertice labels); else, find mathces by comparing to *VertexId*
  - static fromEdgePartitionAndVertices() function: Generate *EdgeTripletPartition<ED, VD>* edgetripletpartition given input parameters *EdgePartition<ED>* edgePartition and *Iterator<Tuple2<VertexId, VD»* vertices.
  - getVertexAttrs: Return a vertexAttrs given a array position.
- **Task 2: Implement interfaces: *matchEdgePattern()*, *inDegrees()*,*outDegrees()* and *degrees()* through all abstraction levels (*Graph*, *RDD* and *Partition*)**
  We implemented:
  - matchEdgePattern() function in *EdgeTripletPartition*: Mentioned in Task 1.2.

- matchEdgePattern() function in *EdgeTripletRDD*: Mentioned in Task1.1.
- inDegrees()/outDegrees()/degrees() functions in *EdgePartition*:
  - Invokes calDegree() with different parameters.
  - calDegree() function calculate degree of src or/and dst inputs.
  - Obtain in-degrees, out-degrees and both-degrees respectively.
- inDegrees()/outDegrees()/degrees() functions in *EdgeRDD*:
  - Invokes calDegree() with different parameters with different input parameters such as *EdgeDirection.IN*, *EdgeDirection.OUT* and *EdgeDirection.BOTH*.
  - calDegree() function invokes corresponding functions in *EdgePartition* class given different inputs.
  - Obtain in-degrees, out-degrees and both-degrees respectively.
- inDegrees()/outDegrees()/degrees() functions in *Graph*:
  - Invokes inDegrees(), outDegrees() and degrees() from *EdgeRDD*.
  - Obtain in-degrees, out-degrees and both-degrees respectively.

- **Task 3: Implement Graph**
  We implemented:
  - shipVertexAttrs():
    - Based on *RoutingTable* in each *VertexPartition*, return a new *EdgeTripletRDD* by shipping vertex attributes from *VertexRDD* to corresponding *EdgeRDD*.
    - Invokes fromEdgePartitionAndVertices() which is from *EdgeTripletPartition* and has an Edgepartition and a vertices iterator as its parameters, and fromEdgePartitionAndVertices() will return an EdgeTripletPartition which is then used to construct an *EdgeTripletRDD*.
  - matchEdgePattern(): Mentioned in Task 2
  - match(): Mentioned in Task 6;
  - JoinMatches():
    - Input parameters: a *List<MatchesRDD>* matches;
    - Return a *List<MatchesRDD>* newmathes in which a match is joined with the rest of matches, for instance, if matches contain matchRDD1, matchRDD2 and matchRDD3, the return of new matches would be [matcheRDD1, matchRDD2, matchRDD3], [matchRDD2, matchRDD3] and matchRDD3.

- **Task 4: Implement the classes *Match*, *MatchesRDD* and *MatchMeta***
  We implemented:
  - match() from *Graph*:
    - Invokes toEdgePatterns() from *patternGraph* in order to generate an array of EdgePatterns.

    * Invokes fromEdgePattern() function from *MatchesRDD* with inputs as that EdgePatterns array generated in the last step.

- *Match*:
    * construct a Match instance by initiating a List of *VertexId*;
    * implement compareTo() given parameter of a *VertexId* named othervid in which returning 1 if owned vertex larger than othervid, returning -1 if smaller than othervid and returning 0 for the rest conditions;
    * implement iterator() to generate vertices iterator
- *MatchMeta*:
    * construct a Match instance by initiating a List of *VertexId*;
    * implement comparewith() given parameter of a *VertexId* named othervid in which returning index of List of *VertexId* if owned vertex equal to othervid and returning -1 for the rest conditions;
    * implement iterator() to generate vertices iterator
- *MatchesRDD*:
    * constructor:
        · Member variables includes *MatchMeta* meta and *List<Match>* matches;
        · invoke parent constructor with input matches.rdd(), and expose *Match* class;
        · initialize member variables by input parameters *MatchMeta* meta and *JavaRDD<Match>* matches whose iterator is utilized to initialize member variable *List<Match>* matches;
    * join():
        · The method 'join()' computes the joining results of two MatchesRDD instances, and the result is returned as a new MatchesRDD instance
        · input parameter is a *MatchesRDD<ED, VD>* other would be used to compared with class *MatchesRDD* itself;
        · a new *MatchesRDD* would be returned according to join();
        · mechanism of join() include several steps:
        · compare match metadata from itself and other and join metadata
        · compare matches and join matches given the metadata (edge pattern)
    * matchEdgePattern():
        · initiate *GraphLoader* class;
        · invoke getGraphInstance() which returns a data Graph graph, and involke getPatternInstance() which returns a *PatternGraph* patterngraph;
        · return *MatchesRDD* by involking graph.match() funciton with parameter patterngraph;

- **Task 5: Implement GraphLoader for reading both data graphs and pattern graphs from files**
  We implemented:
  - constructor
  - getGraphInstance():
    * read data from data file
    * for each line read by *BufferedReader*, split the string by space delimiter and distributed data into *JavaRDD<Edge<ED»* edges and *JavaPairRDD<VertexId, VD>* vertices;
    * return a *Graph* by invoking fromEdgesAndVertices() with input parameters as edges and vertices.
  - getPatternInstance():
    * read data from pattern file
    * for each line read by *BufferedReader*, split the string by space delimiter and distributed data into *List<QueryVertex>* edges and *List<QueryEdge>* vertices;
    * return a *PatternGraph* by invoking *PatternGraph* constructor function with input parameters as edges and vertices.

- **Task 6: Understand the query optimization method and implement it. Conduct some experiments to show that the method works by collecting measurements from SparkUI.**
  - Query optimization:
    * First, we generate a array of *EdgePattern* based on the input *PatternGraph* by calling function toEdgePatterns().
    * Then, we obtain a list of *MatchesRDD* based on every *EdgePattern*.
    * Next we use Map-Reduce to get *MatchesRDD* which were further sorted by their number of occurrences.
    * Finally, we join every two sorted MatchesRDDs to generate one MatchesRDD until we could no longer do join operation on that MatchesRDD
    * The final result is the sub-matchGraph we tend to get.
  - We have not conducted experiments about performance to show optimization works by collecting measurements from SparkUI.

- **Task 7: Test your implementations and explain your tests in your report.**

- EdgeRDDTest():
  - Implemented sampleEdges(), testEdge(), testDegree();
  - sampleEdges():Construct *List* of five sampled Edges in which for example there is an *Edge* with source *VertexId* VertexId(2L), destination *VertexId* VertexId(4L) and its edge label 1;
  - testEdge(): Test two *Edge*(getting from sampleEdges() function) by calling "compareTo" function; all tests passed;
  - testEdgeRDD():
    - Manually constructed two *EdgeRDD* by "EdgeRDD.fromEdges()" with input parameters "edges" through calling "sampleEdges()";
    - Compare two *EdgeRDD* by calling "compareTo" on top of their *List<Edge<Integer»*
  - testDegree():
    - Test inDegrees(), outDegrees() and degrees() functions;
    - Create a list of *List<Edge<Integer»* testEdges by invoking sampleEdges() in ordre to construct *EdgeRDD<Integer>* edgeRDD for further use of testing *EdgeRDD*'s inDegrees(), outDegrees() and degrees() functions;
    - Tests are carried out by comparing returns from inDegrees(), outDegrees() and degrees() with known ground truth value, say, if we made a vertex with indegrees 4 then inDegrees() funciton should return that number of degrees;
    - All tests passed.
- In VertexRDDTest class:
  - Implemented sampleVertices(), sampleEdges() and testVertexRDD();
  - sampleVertices(): Return a *List<Tuple2<VertexId, Integer»* of vertices;
  - sampleEdges: Return a *List<Edge<Integer»* of Edges;
  - testVertexRDD: Test *VertexRDD* by assserting *VertexRDD<Integer>* vertices number given that we knew the groud true number.
  - All tests passed.
- In new EdgeTripletRDDTest class:
  - Implemented testEdgeTriplet() to test edgeTriplet and testEdgeTripletRDD() to test edgeTripletRDD.fromEdgeTriplets() based on edgeTriplets;
  - sampleEdgeTriplets(): Generate *List<EdgeTriplet<Integer,Integer»* of EdgeTriplets;
  - testEdgeTriplet(): Test EdgeTriplet by comparing sampled edgeTriplet to known edgeTriplet instance; all test passed.
  - testEdgeTripletRDD():

        - Generate a EdgeTripletRDD by constructing *EdgeTriplet* and also invoking "EdgeTripletRDD.fromEdgeTriplets()" functions;
        - Test edgeTripletRDD by running "compareTo()" function: doing "collect()" actions on edgeTripletRDD in order to have them in *List<EdgeTriplet<Integer,Integer»* format; finally calling compare function among these EdgeTriplet's Lists;
        - All tests passed.

- In GraphTest class:
    * Implemented sampleVertices() to generate *List<Tuple2<VertexId, Integer»* of vertices;
    * sampleEdges(): Return *List<Edge<Integer»* of edges;
    * testGraph():
        - Invokes sampleVertices() and sampleEdges() to generate vertices and edges;
        - Instantiate a *Graph<Integer, Integer>* graph with input parameters edges and vertices;
        - Test numEdges() by assertTrue() function;
        - Test numVertices() by running assertTrue() funciton.
        - All tests passed

- In new MatchesRDDTest class:
    * implemented testMatchesRDD() to test MatchesRDD and testJoin() by using examples shown in join() function explaination;
    * sampleMeta(): Return two list<VertexId> to generate MatchMeta for two MatchesRDDs;
    * sampleMatches(): Return two List<Match> to generate Match for two MatchesRDDs;
    * testMatchesRDD(): Test MatchesRDD generate from MatchMeta and Match.
    * testJoin(): using two MatchesRDD generated from sampleMeta() and sampleMatches(), then join them and test the result with expection.
    * All test passed.

- In GraphLoaderTest class:
    * Implemented getGraphInstanceTest() and getPatternInstanceTest();
    * getGraphInstanceTest(): Read data graph from the data file (we create a new graph_forTest file) and check whether the read result is equal to what it should be ;
    * getPatterInstanceTest(): Read pattern graph from pattern file (we use pattern1 file for test) and check whether the read result is equal to what it should be;