2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems

Partitioning Large-scale Property Graph for Efficient Distributed Query Processing

Haohan Pang, Peng Gan, Pingpeng Yuan, Hai Jin, Qiangsheng Hua
National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab.
Clusters and Grid Computing Lab.
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
Email: {panghaohan,ppyuan,hjin,qshua}@hust.edu.cn

***Abstract*—Recently, many communities have published their domain knowledge in the form of graph. A significant trend is that the relations between pairs of nodes in the graph have become so complex that these data can be treated as property graphs. Complex properties imply semantic constraints which can distinguish elements of property graph into different semantic-related aspects. If the property graph can be partitioned according to these aspects, cross-partition communication would be reduced when evaluating SPARQL queries, for the reason that a query can often be evaluated in a single aspect. Conventional graph partition methods, however, pay little attention to semantic constraints in property graphs. To solve this problem, we propose a new property graph partition method for query optimization. First, we propose the property semantic reachable path as basic partition element, with which to capture the semantic-related aspects in a property graph. Second, to combine paths into partitions, we develop a space-efficient algorithm which merges vertices rather than entire paths. Extensive experiments over representative property graph data confirm the effectiveness of our semantic awareness approach. The performance of our approach is comparable with leading competitors, in terms of load balance, data redundancy, and network overhead.**

***Index Terms*—Property graph; Partitioning; Query processing**

I. Introduction

With the explosive growth of linked data sets in the last years, processing them has become more and more difficult. Especially, the relation between pairs of nodes in the graph has become so complex that these data sets can be treated as property graphs. Property graph is a linked data model where the edges, as well as the nodes, have abundant attributes (key-value pairs). Because of this feature, it is more suitable for modeling complex linked data. Property graph has widely used in industry and academia, such as knowledge management, recommendation engines, network management, and social networks analysis [1], [2] [3]. Figure 1 shows a simple property graph of aeronautical data. Each edge and node in this property graph have one or more properties, which contains abundant semantic information.

Although, property graphs provide extra information to achieve better partition results [4] [5] regarding to graph partition, graph partitioning methods for property graph is far less common. Complex properties in the property graph imply semantic constrains. Taken them into consideration, we can distinguish elements of property graph into different semantic-related aspects, which is shown in Figure 2. For distributed graph data management, the intrinsic advantages of property graph mentioned above can be exploited to guide the partition process. More specifically, if the property graph can be partitioned according to the semantic-related aspects in the property graph, cross-partition communication would be reduced when evaluating queries. Thus a query can often be evaluated in a single aspect which improves the efficiency of parallel operation in distributed environment greatly. Traditional graph partition methods may not work well in property graph, they pay little attention to different semantic-related aspects.

At present, there are many partitioning methods for common graphs. The most commonly used graph partitioning method is hash partitioning. It ensures that graph partitioning can be evaluated in parallel without distributed joins [6]. But it must execute distributed joins with a huge overhead to merge the intermediate results from queries, in order to get the final output. Undoubtedly, this is a resource-intensive query operation. The authors in [7] proposed a new partition method which employs the end-to-end paths as basic partition elements. According to dividing the paths to different partitions, it can reduce the number of cross-partition queries. Unfortunately, this method does not consider the semantic constraints in graph.

Although the reported experimental results of these partition methods seem to be appealing, but they still have opportunities to obtain better partition results in property graph because most of them do not consider the semantic constraints and the importance of semantic information in property graphs.

In this paper, we propose a new partition method in property graph to optimize the evaluation of query. We introduce the concept of the property semantic reachable paths in property graph and use property semantic reachable paths as the element to partition the data. According to this method, the query would be decomposed into small sub-queries, we aggregate the vertexes that shared by all the property paths from the same partition.

978-1-7281-2058-4/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCC/SmartCity/DSS.2019.00225

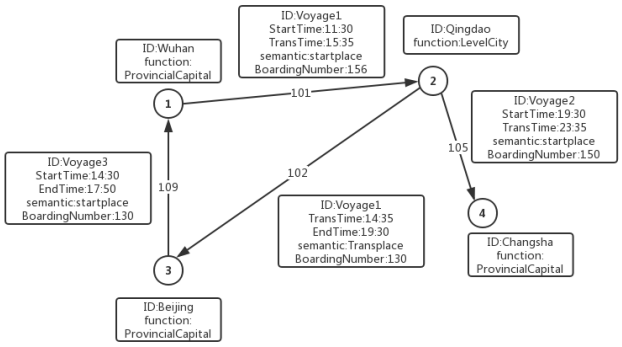

Fig. 1. Property graph: flight graph

Using this method, partitions are less semantic-related and relatively independent. Sub-queries can be executed in parallel with different partitions, which improves the parallelism. For the reason that sub-queries can often be evaluated in a single partition, cross-partition communication would be reduced. The intermediate results also would be reduced, which makes the communication overhead smaller and improves the system's throughput.

All in all, the main contributions of our method are following:

- We propose a scalable property graph partitioning framework. We define the property semantic reachable paths and use the definition as the partition elements. We aggregate vertices to represent the partitioning property paths combining them into partitions.
- We decompose query by property paths that can divide complex SPARQL query into minimize cross-node communications. Our partition solution decomposes a complex query to a small number of sub-queries. And due to the fewer sub-queries, we can get smaller intermediate results, which brings better scalability in evaluating SPARQL query over large-scale property graphs.
- We compared some leading algorithms with our solution. The experiment results show that our strategy has good performance in data redundancy and the load balance. Especially for complex queries, if there are clear semantic relations, the cost of cross-node communication is lower than other algorithms.

The remainder of this paper is organized as follows: Section II review the related work. Section III introduces background of our work. In Section IV and Section V, we define metrics to evaluate partitioning approaches and give our partitioning approach. We evaluate our approach in Section VI and conclude the paper in Section VII.

II. Related Work

Graph partitioning generally depends on actual graph applications. One of graph applications is graph store, which processes users' queries and return results to users. In distributed environment, graph stores need collect intermediates from distributed nodes which process sub-queries, join them together and output the final results. So, large intermediate results lead to huge overhead, including network and computation. For distributed graph stores, graph partitioning methods should reduce the intermediate results.

Graph partitioning is widely explored in past decades. General graph partitioning approaches which consider structural graph statistics in partitioning algorithms, divide vertices (edge-cut) or edges (vertex-cut) of a graph into several disjoint sets. Different from general graph, property graph has attribute/value pairs on vertices and edges. So property graph contains rich semantic information beyond structural statistics. So, previous graph partitioning methods may not work on property graphs because they do not consider properties attached to vertices or edges. The graph partitioning approaches are always used in graph processing systems which include cloud-based systems, graph partitioning based systems, and federated systems [8] [9].

Cloud-based graph systems [10]–[14] always use the existing cloud platforms, such as Hadoop, Spark to process data sets. It would cause lots of intermediate result, which may make the performance of joins worse. For example, Hadoop based data systems always store graph data sets into HDFS files, and then distribute the files into multiple nodes according to placement policies. They may lead to massive I/O cost and communication overhead if graph partitioning algorithms are not carefully designed [15].

Graph partitioning based systems first divide the graph to some partitions and place each of them in different nodes by a distributed and parallel system [15] [16]. These methods mainly divide the data sets into a collection of partitions, and divide a query to sub-queries [17] [18] [19]. When running a query operation, these methods decompose the query Q into several sub-queries. Then these sub-queries will be executed on the data partitions with some techniques similar to distributed relational databases. The best situation is that the queries can be answered at one node so that few intermediate results need to be merged at last. So the performance of query processing is dependent on partitioning methods. $TriAD$ [20] employs $METIS$ [21] to decompose graphs into several partitions. For each node, $TriAD$ maintains six large triple memory vectors that store corresponding permutations of graphs. At the same time, $TriAD$ maintains partition information by using a summary graph.

Federated systems [22] [23] [24] are built on legacy graph systems or systems owned by other organizations or persons. So, sub-systems of federated systems are independent and may not employ same technologies (e.g. graph partitioning, storage) for graph management. When queries are executed on federated systems, applications should assume there exists no regular patterns for the sub-graph in each node. In order to return correct results, federated systems can not prune intermediate results which may be final results.

Most of the systems mentioned above need to use specific partitioning method to break down the distributed graphs such as RDF graphs [25] which represent the data as a set of

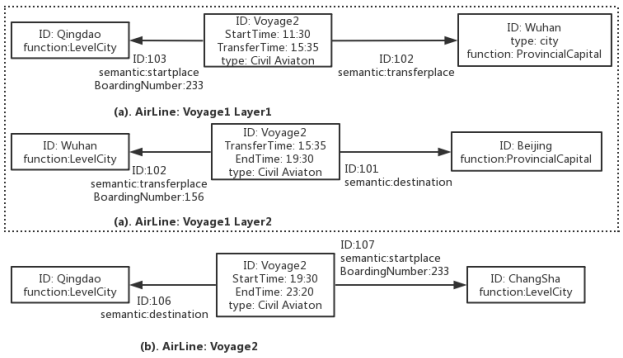

Fig. 2. Property graph aspect partitions

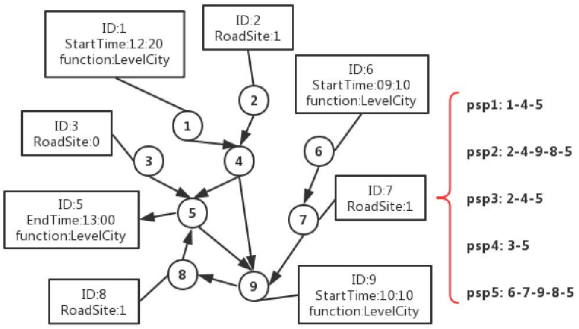

Fig. 3. Property semantic reachable paths

triples in a form <subject, predicate, object>. They may cause relatively much more sub-queries and data redundancy. More cross-partitions will reduce the performance of query. In addition, these methods that using query decomposition to execute the query would cause more intermediate results and they can not make full use of the semantic information in property graphs.

III. Background and Framework

The linked data can be modeled as graphs, which can be represented as RDF graph or property graph. Here, we model linked data as property graphs which can provide a straightforward way to represent complex information. Similar as RDF graph, the data model for property graphs is a directed labeled graph. However, different from RDF graph where each edge or vertex has a label, each edge or vertex of property graph may have many attribute/value pairs. So, property graph model is more powerful to represent semantic information. In the following, we will give the definitions.

Definition 1 (Property Graph): Let $\mathcal{A}$ be a property set. Directed property graph $\mathcal{G} = (V, E, \mathcal{A}, f_V, f_E)$.$f_V$ and f_E are functions which assign the property of vertex or edge property values. Namely, $v \in V, a \in \mathcal{A}$, $f_V(a, v) = u_v^a$. $e \in E, a \in \mathcal{A}$, $f_E(a, e) = u_e^a$.

According to the property graph model, we can make full use of the semantic information in graph structure to reduce the number of sub-queries.

Definition 2 (Path): Let function s, t return the source and destination of a directed edge respectively. There exists a path between vertex v_s and w_e if there is an ordered list of edges, $e_0,\ e_1,\ ..., e_i,\ ...,\ e_{k-1}$, such that $s(e_0) = v_s, t(e_{k-1}) = v_e$,and $s(e_{i+1}) = t(e_i)$ $(e_i \in E(0 < i < k \leq |E|))$. When a path from u to w exists, we say that vertex w is reachable by u.

Definition 3 (Property Path): There exists a path, $e_0,\ e_1,\ ...,\ e_i,\ ...,\ e_{k-1}$ between vertex v_s and w_e. Assume a_i $(0 < i < k)$ be the property of edge e_i, $u_{e_i}^{a_i}$ is the property value of property a_i on edge e_i. If $u_{e_0}^{a_0},\ ...,\ u_{e_i}^{a_i},\ ...,\ u_{e_{k-1}}^{a_{k-1}}$ are consistent, then $u_{e_0}^{a_0},\ ...,\ u_{e_i}^{a_i},\ ...,\ u_{e_{k-1}}^{a_{k-1}}$ are consisted of a property path.

Given a property graph $\mathcal{G}$, path $u_{e_0}^{a_0},\ ...,\ u_{e_i}^{a_i},\ ...,\ u_{e_{k-1}}^{a_{k-1}}$ is a semantic reachable property path when it satisfies the following conditions: (1) v_0 is the start vertex, or a vertex in a directed loop that does not contain any vertex with an incident edge from the outer vertex of the loop. (2) v_m is an ending vertex, or existing a vertex v_i in the path that must have $v_m = v_i$ (i.e. there is a directed loop). (3) the property of $u_{e_0}^{a_0}, ..., u_{e_i}^{a_i}, ..., u_{e_{k-1}}^{a_{k-1}}$ has to be semantic reachable such as Figure 3. The starting vertex represents that has zero in-degree. Similarly, a vertex having zero out-degree is an ending vertex.

Consider the example flight graph shown in Figure 1, where nodes represent cities with some properties and edges represent routes. For a route, there exist a starting city, ending city and/or transfer city. Assume starting city v_0 has the start time property, $u_{e_0}^{a_0} = 11:30AM$, and the transfer time of transfer city $u_{e_m}^{a_m} = 15:35PM$, the route is semantic reachable. Because time is irreversible, $u_{e_0}^{a_0} u_{e_m}^{a_m}$ are semantic reachable. If there is an end city which has the $6:00AM$ end time $u_{e_n}^{a_n}$, the voyage is not semantic reachable.

Proposition 1: Given a property graph $\mathcal{G}$, let it be broken down to a collection of property paths, each vertex and each edge must appear in at least one property path.

We can prove it in two parts:

1) Firstly, there exists at least one semantic reachable property path for any vertex $\in V$ and this path contains the vertex v. If v is a start vertex, v belongs to the path with its own start vertex. If v is not the start vertex and there is a beginning vertex v_s up to vertex v then there must be a path with v_s as the beginning vertex containing v. If it is not the start vertex and there is no beginning vertex v_s reaching vertex v, then it must belong to the directed ring or it is a property according to the Definition 3. So, there must be a beginning vertex v and a property semantic reachable path passing through the vertex v.

2) Then, there is at least one semantic reachable property path for any edge $(u, w) \in E$, so that this path contains this edge. For an edge $(u, w) \in E_{pg}$, according to the conclusion proved in (1), the vertex u must be included in at least one property semantic reachable path. Suppose this path is v_s ...,u,...v_e, where v_s is the starting vertex v_e is the ending vertex. If the vertex u is not the ending vertex of this path, then

there must be a path that passes (u, w) (the property graph has no edge to itself). If the vertex u is the ending vertex, according to the Definition 2, the vertex u appears twice in the path, that is to say, there must be a path with v as the starting vertex (u, w).

Definition 4 (Aggregated Vertex): If all of semantic reachable property paths including v are in a same part, vertex v is named as aggregated vertex.

Definition 5 (k-aspect Property Path Partition): The k-aspect property path partitions on property graph $\mathcal{G}$ are expanded partitions from $\{\mathcal{P}^1, \mathcal{P}^2, ..., \mathcal{P}^k\}$, by adding property paths that are within multi-set from any semantic reachable paths along the forward direction, denoted by $\{P_1^k,\ P_2^k,\ ...,\ P_m^k\}$, where each property path does not intersect each other. Each partitioning blocks $\mathcal{P}^1$ is a subset of the set of all paths in the $\mathcal{G}$. All the property semantic reachable paths of $\mathcal{G}$ are divided into k partitions according to **Proposition 1**.

For property graphs, there is no common and standard query language. One typical language for descriptive analytic on RDF graph is SPARQL, a SQL-like query language. It can query property graph. A SPARQL query contains a set of triple patterns, namely (subject, predicate, object). The subject, predicate, and object component of each triple pattern can be either constant (IRIs, literals etc.) or variable, which is marked by the use of either "?" or "$". There are three types of SPARQL queries: star, chain, and complex queries [26]. Triple patterns of a star query often share a join variable. Triple patterns of a chain query form a path where one triple pattern connect another patterns by a join variable between them. Star queries and chain queries can be composed into a complex query.

Definition 6 (Triple Pattern): The sets of IRIs, literals are denoted as $\mathcal{I}$, and $\mathcal{L}$. $\mathcal{IL} = \mathcal{I} \cup \mathcal{L}$. Assume an infinite set of variables Υ disjoints from $\mathcal{IL}$. The set of triple patterns $\mathcal{TP} = (\mathcal{IL} \cup \Upsilon) \times (\mathcal{I} \cup \Upsilon) \times (\mathcal{IL} \cup \Upsilon)$.

SPARQL query Q includes a set of triple patterns. Actually, all triple patterns are connected by the shared subject or object. Always, a join occurs on the shared subject or object between two joined triple patterns.

Definition 7 (Query Graph) : A SPARQL query is denoted as a query graph $Q = (V_Q, E_Q)$. $V_Q \subseteq V \cup V_{var}$ is the set of vertices where V is a vertex from property graph $\mathcal{G}$ and V_{var} is a set of variables; $E_Q \subseteq V_Q \times V_Q$Every edge has a label that is from property graph or is a variable.

Generally, distributed processing a SPARQL query can be decomposed into two kinds of sub-queries: one is the sub-queries executing on a node or a partition, which are named as local sub-queries; another is sub-queries to join local sub-queries. If a query can be executed in parallel on each compute node without any across node data communication, it is referred to as a local query. Otherwise, processing the query must join intermediate results from multiple nodes. The query is referred to as a cross-node query.

Proposition 2: Given any multi-aspect property path partition plan P, the only containing S-O joins queries such as chained queries are local queries.

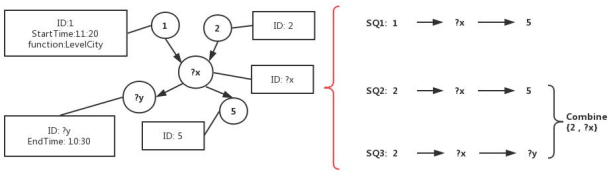

Fig. 4. Decomposing property query model

Given a property graph $\mathcal{G}$ and a query plan Q which only includes S-O connection. Every matches must be a property semantic reachable paths or a directed cycle or the property of some vertexes. So, there must be a property semantic reachable path that make $u(Q)$ be a sub-graph of $\mathcal{G}$.

IV. Query Processing Oriented Partitioning Metrics

With the increasing size of property graphs, the distributed processing queries on property graph is one choice. Due to the complex structures of property graphs, distributed processing queries on property graphs is more difficult than which in normal graphs. Each edge and node in property graph has one or more properties which causes the number of combined paths increased. It will lead to much overhead.

For distributed processing queries, the first step is to analyze whether or not the query Q can be answered on a partition. If not, the query is decomposed into a set of sub-queries which can be answered in a partition. Because a SPARQL query can be seen as a directed graph, the query graph also can be seen as a set of paths. For a query play Q, first of all, dividing the Q into a collection of paths, denoted as $\{SQ_1,\ SQ_2,\ \cdots,\ SQ_m\}$. Every sub-query SQ_i would be a chain query or a directed query that only contains S-O connections such as Figure 4.

All the sub-queries are local queries. If there are two sub-queries that have the same vertex, it means that there may be a distributed join between the sub-queries follow up, denoted as $SQ_i \cup SQ_j$, $V_{i,j}$ is the shared vertex of the two sub-queries. Finally, the system joins intermediate results returned by sub-queries to get the ?nal result.

A. Metrics for graph partitioning

For a property graph and given the number of partition k, there would be a lot of property path partition plan. How to choose the optimal plan is the key to efficient distributed processing queries on property graph. So, we define some metrics to assess the pros and cons of partitioning property graphs.

Load Balance. In distributed environments, the result of graph partitioning will seriously impact the performance of the query due to data skew and load imbalance etc.. Data skew could lead to severe computation tilt. The amount of data in the overload partition may exceed the storage and computation power of the compute node. Both of these problems can lead to bottlenecks in computing clusters. In our model, we use the

number of the property paths in a node to indicate the load of the node. For node i, its load or number of property paths is denoted as $|L_b(P_i)|$. The number of the property paths in each node or partition is different. In order to evaluate the load balance of one partitioning of a property graph, we compute the difference between the maximal partition (denoted by $|P_{max}|$) and minimal partition (denoted by $|P_{min}|$) as a percentage of total partition.

$$L_b(P) = \frac{|P_{max}| - |P_{min}|}{|P|} \times 100\% \quad (1)$$

Data Redundancy. Property paths may share same vertices, edges or property/value pairs. Since property paths are distributed into different partitions, there must be data redundancy (e.g. duplicate vertices, edges or property/value pairs) between partitions. Duplicate data may cause additional storage, communication and computation overhead. We use $D_r(P)$ to denote the redundancy rate of a property path partitioning plan.

$$D_r(P) = \frac{1}{\Sigma_{e\in E}\Sigma_{a\in A}|u_e^a|} \sum_{e\in E}(|P(e)| \cdot \sum_{a\in A}|u_e^a| - 1) \quad (2)$$

In Formula 2, $|u_e^a|$ is the number of all properties on edge e. The $|P(e)|$ is the number of partitions that contain edge e in partitioning plan P.

Number of Cross-node Sub-queries. As mentioned above, reducing the number of intermediate results sub-queries return can improve the query performance. If there are a large number of sub-queries, the number of cross-node queries would also increase, which leads to redundancy of intermediate results and reduces the query performance. The number of the sub-queries which need to be evaluated on multiple nodes is smaller, the performance to evaluate the query is better. Merging sub-queries according to Definition 4, it represents that it can merge vertexes to reduce the number of sub-queries. According to this way, the size of aggregated vertexes represents the cross-node sub-queries. The set of vertexes which are aggregated is denoted by $|AV_+|$. The size of $|AV_+|$ reflects the number of cross-node sub-queries which determines the query performance.

B. Problem formulation

As mentioned above, to find an optimal property path partition plan, we should keep a good load-balancing, reduce the data redundancy, and decrease the number of cross-node sub-queries. The load balance depends on the $|L_b(P)|$, so there is $|L_b(P)| \leq [\frac{n}{k}], n$ means the number of $\mathcal{G}$'s property paths.

The data redundancy (e.g. duplicate vertices, edges or property/value pairs) is caused by partitioning data into different nodes, it also involves cross-node sub-queries just like $|AV_+|$. We can easily deduce that the redundancy rate of data is limited to the range associated with $|AV_+|$. That is to say, the $|AV_+|$ is larger and the redundancy rate is smaller.

V. Property Path Based Partitioning Algorithm

A. Partitioning algorithm

According to empirical research [27], a SPARQL query is usually made up of joins between subject and/or object components of triple patterns. Based on the observation, we also partition property graph into a set of property paths. The property path based partition algorithm first decompose a property graph into a set of property paths and we record the beginning vertices of property paths. Different from paths, there exist much more property paths than paths in property paths. So it raises a challenge how to merge property paths. So, we aggregate beginning vertices to sets $AV+$, because the beginning vertices are initial vertices that can be aggregated. Then, the algorithm calculates the benefit ratio of merging a vertex and chooses the vertex that need to be merged later through the benefit ratio value. At last, the algorithm places the aggregated vertices into each node so that each node contains the similar size of the different aggregated vertices group.

According to this way, there are two advantages at least. First, if all beginning vertexes are merged, most frequent query types mentioned would be local queries, such as star query, chain query, and tree query. The second advantage is that the method saved computing space and cost. Because the number of property paths generated from a property graph is very large, if all property paths are generated and stored, it requires large storage space. However, aggregating all of beginning vertices could greatly reduce the spatial complexity. Because each beginning vertex can be used to represent property paths that contain this beginning vertex, all property paths can be restored by simple traversal. This way can save a lot of storage space and the computation overhead to generate all the paths.

The most important step is merging the remaining vertexes according to the benefit ratio of return. We regulate that the benefit ratio of merging a vertex is set to 1. The benefit ratio means that merging a vertex can make the benefit value. Before this step, it needs all the property semantic reachable paths as the input elements. So we should calculate all the beginning vertexes first and use beginning vertexes to generate property paths. Its pseudo-code is as follow. Generating all property paths is not the most time-consuming part, so we use a relatively simple strategy to reduce the complexity of the algorithm. According to the depth-first traversal of beginning vertexes, find out all eligible paths. Given property semantic reachable paths by comparing property of vertexes and paths.

After generating all property semantic reachable paths, we should start to merge vertexes. When we have counted all beginning vertexes, we need as much as possible to merge remaining vertexes. The specific operation of merging these vertexes is combining all paths (or path group) that contain this vertex into a single path group. At the same time, we also should control the number of all property semantic reachable path sets less than or equal to $[\frac{n}{k}]$. Ensure the size to meet a condition, we guarantee even division in the next step. In the last step, we need to distribute paths evenly to k nodes. But the need to pay attention to is that the size of property paths

is bigger than common paths, property paths contain more information so they can reduce the number of cross-node sub-queries.

Algorithm 1: Get List of Beginning Vertex

Input: $G = (V, E, \mathcal{A}, f_v, f_e)$, V_{start}
Output: V_b

```
1  V_b.add(V_start)
2  for each v ∈ V do
3      markindex(v);
4      if v ∈ V_start then
5          V_b.add(v); min(v) ← v
       end
       else
6          V_b ← 0; min(v) ← v
       end
   end
7  while Key = 1 do
8      for getMarkIndex(v) do
9          for u ∈ v.getNeighbor() do
10             V_u.add(V_v); min(u) ← min{min(u), v} if
               V_u or min(u) is marked then
12             markindex(u); Key ← 1
               end
               else
13             no-markindex(u);
               end
           end
       end
14     V_b.add(min(v));
   end
```

The algorithm initializes a set V_b to count all beginning vertices. If a vertex is a beginning vertex, then $V_b.add(v)$, or V_b does not take any treatment. Then the algorithm spreads the V_b of each vertex to out of the neighbor until there is no update information. After the above stage, when the algorithm is responsible for vertex merge initialization, each path group contains only one starting vertex, indicating that all paths starting from this starting vertex are included in this path group. Then, the algorithm remaining vertices are merged in reverse order according to the value of the benefit rate (the weight estimation formula is optional) and must satisfy the limit of the number of paths included in the path group. After the merging operation ends, the algorithm expands the beginning vertices into paths. Finally, assign the path group to the corresponding partition. During the time, we must choose the maximum benefit ratio value. The most important thing is keeping each property path semantic reachable.

B. Decomposing Query

We have introduced the property path partition algorithm in detail above. If a query Q is a local query, we can send it to each node to calculate the results, at last return all results for a summation. But not all queries are local queries, for a distributed query, it needs to be decomposed into several local sub-queries. Then, we can use Apache Spark frame to execute the distributed joins.

According to the decomposing query model, given a query Q, first of all, decomposing the Q to a set of sub-queries $SQ = \{SQ_1, \ldots SQ_m\}$, every SQ_i should contain the beginning vertexes and the set of vertexes and edges which they can reach in the query graph. So all of the sub-queries are star-query, chain-query or tree-query. Then, we can try to join the sub-queries. Given two queries SQ_i and SQ_j, we grant that they have the common vertexes. If one of the vertices is a constant and merged at the time of data division, you can merge the two sub-queries directly instead. If the common vertex is a variable, we can check whether the property paths are reached. Then the two sub-queries would be merged. The process continues until no two sub-queries can be merged, and the result of the query split is finally obtained.

VI. Experiments

We build a cluster to complete these experiments. We compare our method with other path partition methods which are as follows: semantic hash partition method and path partition method. In these experiments, all the algorithms are in the same environment.

The system used in the experiment consists of TripleBit [26] that is acted as the local query engine for compute nodes. For cross-node communication, we use Apache Spark framework [28] which is a popular big data framework. For a fair comparison, we use the same experimental environment, we call the hash semantic partitioning algorithm the **Hash algorithm**, and the top-level path merging algorithm the **Path algorithm**, our multi-aspect property path partition algorithm the **Property algorithm**.

The data sets of the whole experiment come from LUBM benchmarks and Aviation Data from the real world. Aviation Data set is provided by China Southern Airline in the real world, it is a true airline data set. It contains rich semantic information, so it accords with the demands that we can make full use of the semantic information. The other data sets are LUBM data sets. Although they do not contain rich semantic information, they are from benchmark generators. We need to show that our method works well on them. The sizes of them are shown in Table I. All the benchmark queries come from the standard data sets, as shown on our homepage [29].

TABLE I
The Data Sets

Data sets	Triples	Resource	Data size	Is Property Graph
LUBM-10	1050K	256K	216M	No
LUBM-50	7000K	1850k	1GB	No
Aviation Data	120M	47M	30GB	Yes

A. Comparison with other partitioning algorithms

Table II shows the partitioning time of the three algorithms. Comparing the results, we can know that the Hash algorithm has the minimum execution time, the other two algorithms

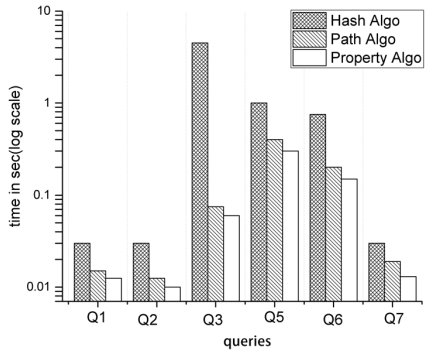

Fig. 5. Query processing on LUBM-50

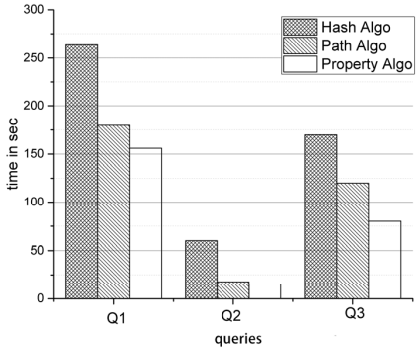

Fig. 6. Query processing on Aviation Data

have almost the same execution time. This is because the both algorithms take a certain amount of time to calculate the weight of the vertices. And the Property algorithm needs to calculate the property of data sets.

TABLE II
PARTITIONING TIME

	LUBM-10	LUBM-50	Aviation Data
Hash-Algo	16.6s	65.7s	32.6min
Path-Algo	21.5s	101.5s	43.5min
Property-Algo	22.7s	112.7s	40.7min

Table III shows the load balance, the data redundancy and the number of cross-node sub-queries of the three algorithms. The index of load balance is the max partition size in the total percentage. As we can see, the Hash algorithm has lots of data redundancy. Without the aggregated vertex, the Hash algorithm has a bad cross-node and lager data redundancy. The load of Path algorithm and Property algorithm is balanced, and the data redundancy is rather lower. Especially, property algorithm is the best in the large data sets. But the Hash algorithm has the smallest number of cross-node sub-queries, because it does not consider the aggregated vertex. The Path algorithm has a good effect because it can merge vertexes to reduce the cost of storing triples.

TABLE III
PERFORMANCE OF PARTITIONING ALGORITHMS

	Algorithms	Load balance	Redundancy	$\|AV+\|$
LUBM-10	Hash Algo	1.5%	1.56	56k
	Path Algo	1.8%	0.06	200k
	Property Algo	1.9%	**0.03**	197k
LUBM-50	Hash Algo	1.8%	2.07	233k
	Path Algo	2.2%	0.35	970k
	Property Algo	2.1%	**0.28**	877k
Aviation Data	Hash Algo	4.6%	2.56	12M
	Path Algo	3.5%	0.25	27M
	Property Algo	3.2%	**0.17**	35M

Our solution has the best effect on the number of the cross-node sub-queries. Because we not only aggregate the vertexes, but also reduce the number of some paths which are not semantic reachable. Especially, with the Aviation Data, we can get a better effect. Because the Aviation Data has the obvious semantic reachable relation of time. So, in this respect, we perform better.

B. Query performance

Because the number of LUBM-10 is so small that we do not need to compare the query performance between them, we only use data set LUBM-50 and Aviation Data set for the comparison with the three algorithms. Figure 5 and Figure 6 show the results of their query performance.

In Figure 5, Q1 and Q2 are star queries that only contain $S-S$ join. Q7 is a one-hop chain query. They are the cross-partition queries for these algorithms. Because of the their high selectivity, the local TripleBit engines can retrieve the results matching sub-queries efficiently. The reason for query performance difference is lots of data redundancy, which leads to duplicate results which need to be removed.

Q3 is a tree query. It is highly selective. So the hash algorithm has to decompose it into multiple sub-queries. It joins the intermediate results using the Spark framework. In this case, the property algorithm improves over these approaches.

Q5 and Q6 are complex queries. It means that both of them need to be decomposed into some sub-queries. The hash algorithm performs worst because of the large data redundancy. The property algorithm performs best. The semantic reachable paths can reduce the number of sub-queries. The path algorithm can not make full use of the semantic, so it is worse than the property algorithm.

In Figure 6, we find that the query performance results in Aviation Data are similar to those for LUBM-50. But as the size of the data sets grows larger, using the property algorithm and the path algorithm can achieve even greater performance improvements than the hash algorithm, because as the size of the data set increases, the size of the intermediate result becomes larger. As the Aviation Data is from the real world, it has more semantic information, the property algorithm can perform better than the path algorithm.

VII. Conclusion

In this paper, we propose a property path based partitioning method for a property graph. Property paths are a set of connected edges which are semantic reachable. The proposed method considers the semantic constraints of property paths and exploits them to partition property graph, which helps to reduce the intermediate result of a query. Because of partitioning the larger-scale property graph into several balanced portions, which are consisted of the property semantic reachable paths, many complex queries can be seen as a set of sub-queries on the local partition. So it reduces the cost of cross-node join with the large-scale data sets and improves the query performance on the partitions of a property graph produced by the method. Results of our experiment demonstrated that our method is effective for complex queries, especially with semantic attributes. At the same time, it maintains the load balance in a distributed environment and reduces data redundancy. The method offers another thought for thinking about large-scale association data query.

VIII. Acknowledgement

This research is supported by The National Key Research and Development Program of China (No. 2018YFB1004002), NSFC (No. 61672255), Science and Technology Planning Project of Guangdong Province, China (No.2016B030306003 and 2016B030305002), and the Fundamental Research Funds for the Central Universities, HUST.

References

[1] Georgios Drakopoulos, Andreas Kanavos, and Athanasios K. Tsakalidis. Evaluating twitter influence ranking with system theory. *Proc. of WEBIST'16*, pages 113–120, 2016.

[2] Shuang Zhou, Pingpeng Yuan, Ling Liu, and Hai Jin. Mgtag: a multi-dimensional graph labeling scheme for fast reachability queries. pages 1372–1375, 04 2018.

[3] Martin Junghanns, Max Kießling, Niklas Teichmann, Kevin Gómez, André Petermann, and Erhard Rahm. Declarative and distributed graph analytics with gradoop. *Proc. of VLDB'18*, pages 2006–2009, 2018.

[4] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. *Proc. of SIGMOD'18*, pages 1433–1445, 2018.

[5] Andrejs Abele, John P McCrae, Paul Buitelaar, Anja Jentzsch, and Richard Cyganiak. Linking open data cloud diagram 2017 (2017), 2018.

[6] Kisung Lee and Ling Liu. Scaling queries over big rdf graphs with semantic hash partitioning. *Proc. of VLDB'13*, pages 1894–1905, 2013.

[7] Buwen Wu, Yongluan Zhou, Pingpeng Yuan, Ling Liu, and Hai Jin. Scalable sparql querying using path partitioning. *Proc. of ICDE'15*, pages 795–806, 2015.

[8] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. Linked stream data processing engines: Facts and figures. *Proc. of ISWC'12*, pages 300–312, 2012.

[9] Lei Zou and M. Tamer Özsu. Graph-based rdf data management. *Data Science and Engineering*, 2(1):56–70, 2017.

[10] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2rdf: Rdf querying with sparql on spark. *Proc. of VLDB'16*, 9(10):804–815, 2016.

[11] Mohammad Farhan Husain, Pankil Doshi, Latifur Khan, and Bhavani Thuraisingham. Storage and retrieval of large rdf graph using hadoop and mapreduce. *Proc. of IEEE CLOUD'09*, pages 680–686, 2009.

[12] Mohammad Husain, James McGlothlin, Mohammad M. Masud, Latifur Khan, and Bhavani M. Thuraisingham. Heuristics-based query processing for large rdf graphs using cloud computing. *IEEE Access*, 23(9):1312–1327, 2011.

[13] Vaibhav Khadilkar, Murat Kantarcioglu, Bhavani Thuraisingham, and Paolo Castagna. Jena-hbase: A distributed, scalable and efficient rdf triple store. *Proc. of ISWC-PD'12*, pages 85–88, 2012.

[14] Mohammad Farhan Husain, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham. Data intensive query processing for large rdf graphs using cloud computing tools. *Proc. of IEEE CLOUD'10*, pages 1–10, 2010.

[15] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. Processing sparql queries over distributed rdf graphs. *Proc. of VLDB'16*, pages 243–268, 2016.

[16] Jiewen Huang, Daniel J Abadi, and Kun Ren. Scalable sparql querying of large rdf graphs. *Proc. of VLDB'11*, pages 1123–1134, 2011.

[17] Luis Galárraga, Katja Hose, and Ralf Schenkel. Partout: a distributed engine for efficient rdf processing. *Proc. of WWW'14*, pages 267–268, 2014.

[18] Katja Hose and Ralf Schenkel. Warp: Workload-aware replication and partitioning for rdf. *Proc. of ICDEW'13*, pages 1–6, 2013.

[19] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. *Proc. of VLDB'13*, 6(4):265–276, 2013.

[20] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. *Proc. of SIGMOD'14*, pages 289–300, 2014.

[21] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *Proc. of SIGMOD'98*, pages 359–392, 1998.

[22] Bastian Quilitz and Ulf Leser. Querying distributed rdf data sources with sparql. *Proc. of ESWC'08*, pages 524–538, 2008.

[23] Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, Kai-Uwe Sattler, and Jürgen Umbrich. Data summaries for on-demand queries over linked data. *Proc. of WWW'10*, pages 411–420, 2010.

[24] Fabian Prasser, Alfons Kemper, and Klaus A Kuhn. Efficient distributed query processing for autonomous rdf databases. *Proc. of EDBT'12*, pages 372–383, 2012.

[25] Valerie Bonstrom, Annika Hinze, and Heinz Schweppe. Storing rdf as a graph. *Proc. of LA-WEB'03*, pages 27–36, 2003.

[26] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. Triplebit: a fast and compact system for large scale rdf data. *Proc. of VLDB'13*, 6(7):517–528, 2013.

[27] Mario Arias, Javier D Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world sparql queries. *arXiv preprint arXiv:1103.5043*, 2011.

[28] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *Proc. of HotCloud'10*, page 95, 2010.

[29] Triplebit. http://grid.hust.edu.cn/triplebit/.