Dynamic Partitioning of Big Hierarchical Graphs*

Vasilis Spyropoulos
Athens University of Economics and Business
76 Patission Street
Athens, Greece
vasspyrop@aueb.gr

Yannis Kotidis
Athens University of Economics and Business
76 Patission Street
Athens, Greece
kotidis@aueb.gr

ABSTRACT

Hierarchical graphs are multigraphs, which have as vertices the leaf nodes of a tree that lays out a hierarchy, and as edges the interactions between the entities represented by these nodes. In this paper we deal with the management of records that are the edges of such a graph by describing a model that fits well in a number of applications, many of which deal with very big volumes of streaming distributed data that have to be stored in a way so as their future retrieval and analysis will be efficient. We formally define a partitioning schema that respects the hierarchy tree, and apply these ideas by using well known open source big data tools such as Apache Hadoop and HBase on a small cluster. We built a framework on which we examine some basic policies for the partitioning of such graphs and draw interesting conclusions regarding the quality of the partitions produced and their effectiveness in processing analytical queries drawn from the imposed hierarchy.

1. INTRODUCTION

There are numerous applications such as management and visualization of Telecommunications data [1], Web log mining [2] or Internet traffic analysis [3], in which data records can be described as edges between vertices of a *hierarchical graph*, i.e a directed multigraph whose vertices are also the leaf nodes in a hierarchy tree. As an example, Call Detail Records (CDRs) can be naturally depicted via a massive graph structure in which nodes represent customers' phone numbers and edges between them their calls. At the same time, the nodes of this graph are the leaves of a tree that indicates their location and superimposes a geographical hierarchy over this data [4].

You can see such an example in Figure 1 which presents a small part of the hierarchy of locations in Greece. In this Figure, Attiki and Messinia are states of Greece, while Athens, Piraeus and Kalamata are cities in these states. The unlabeled nodes represent subscribers of a telephony network. Then, each call serviced by the telephony network instantiates a new directed edge in the graph, between the respective vertices (caller and callee) located at the leaves of the hierarchy tree, as is shown in the Figure. This hierarchy is exploited in order to help pose queries that seek to retrieve certain records for further analysis. For instance in order to calculate statistics on the out-of-state calls originating in Athens, this intention may be described by a query edge between Athens and Greece in the tree. Similarly, query edge (Attiki, Attiki) denotes the set of calls originating and terminating within this state. The aforementioned query edges are shown in Figure 1 (dashed lines). Another example where similar hierarchical graphs exist, is social networks where users are organized in groups according to their location or other characteristics such as age or interests and we need to record the interactions between them. Users (e.g. analysts) of such data often need to answer queries regarding interactions or distributions of records between hierarchy groups not necessarily belonging to the same level of the hierarchy.

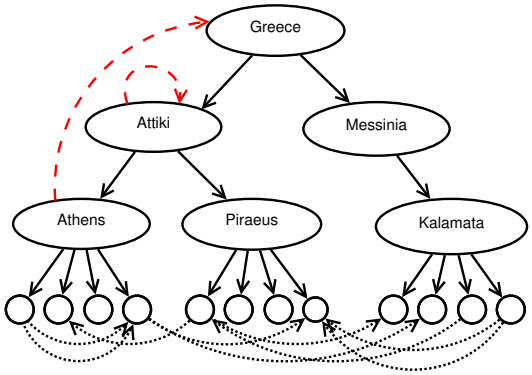


Figure 1: Example of a hierarchical graph - the graph consists of the unlabeled nodes, which are also the leaves of a hierarchy tree, and the edges/interactions between them

In the aforementioned applications, this kind of graphs can grow to enormous size. For instance a large telecom provider may service hundreds of millions of calls per day, each triggering a new edge in the graph. Moreover, this data is distributed by nature as it is being streamed from distant locations (e.g. call centers, web hosts, ip routers). Thus, we need solutions that can cope with the volume, but also with the streaming and distributed nature that characterize this kind of data.

In our work we address these challenges by moving the storage and processing of these graphs to the cloud. We propose a system that uses the distributed data store HBase [5] running on the

*This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: RECOST

Hadoop distributed file system [6], but also MapReduce [7] techniques so as to handle a continuous stream of updates efficiently. Our system leverages the available degree of hardware parallelism by devising a dynamic partitioning scheme over the streamed edges of the hierarchical graph. Our techniques aim at generating partitions that correspond to clusters of graph edges, which are naturally mapped to collections of nodes in the hierarchy tree, while respecting the distribution of the streamed records. In this way, analysis of the records based on the superimposed hierarchy can be performed in an efficient manner. Our contributions are:

- We revisit the problem of managing massive hierarchical graphs that are streamed by many applications of interest. Our techniques utilize emerging computational and data management platforms for manipulating large, dynamic and distributed collections of records in a cluster of machines. Available parallelism is exploited via a dynamic partitioning scheme we propose for the streamed records.
- We formally define the space of choices for partitioning the streamed graph, while respecting the hierarchy tree that is superimposed over its nodes. We then present a number of interesting partitioning policies, and describe the details of the system we built for implementing our framework while utilizing off-the-shelf tools.
- We present an experimental evaluation of our system using a small cluster of machines. Our results demonstrate the efficiency of our system in managing massive graphs scaling to millions of edges. We also provide a comparison among the partitioning policies we implemented based on the results of a number of experiments that we conducted.

The rest of the paper is organized as follows. In Section 2 we discuss related work. In Section 3 we formally introduce our framework, discuss the type of graph data and queries we consider. Then, we describe a partitioning scheme based on the tree hierarchy that accompanies the graph data, a number of partitioning policies that we implemented and discuss the architecture of our system. Section 4 presents our experiments and Section 5 contains concluding remarks.

2. RELATED WORK

Interest in graphs and their applications in data management has been renewed due to the wide spread of fields such as social networks and the semantic web. In the same time there is a profound need for the efficient management of big distributed data. As a result we can see a lot of recent work done in the area, ranging from graph databases to distributed graph processing or graph partitioning techniques. The latter mainly cope with the problem of splitting a large graph by assigning its vertices into independent partitions. While there are several variations of the problem, a typical objective is to obtain partitions such that the sum of the vertex weights across partitions is even while the sum of the inter-partition edges is minimized [8, 9]. The work in [10] proposes data partitioning that is guided by the user's queries. Another approach that aims at the partitioning of graphs across clusters of servers in a dynamic way by using queries during the runtime of the system can be found in [11]. Our query-driven partitioning policy described in Section 3 is motivated by these ideas but the actual setting is different. In [12] the authors present SPAR, a social partitioning and replication middle-ware that uses the social graph structure in order to achieve data locality. In our work we also use a structure to guide the partitioning but the structure we use is a hierarchy tree.

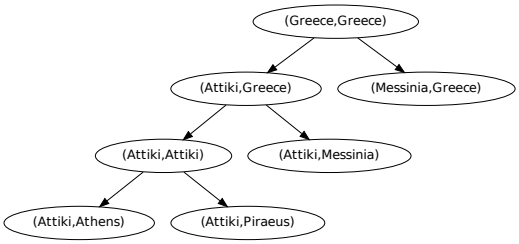


Figure 2: Example state of a partition tree T_P

The objective in our work is quite different to vertex partitioning since we actually do partitioning of the edges of a hierarchical multigraph. These edges represent interactions between nodes that need to be investigated according to the imposed hierarchy. In a typical scenario aggregation of these edges at the higher levels of the hierarchy tree is more important for the application, while in certain applications, such as analysis of call detail records, decision making based on fine-granularity statistics (i.e. low-level aggregations) is in-fact prohibited by law, so that certain carriers cannot obtain unfair advantage over their competitors. Our techniques can benefit systems built for visualizing hierarchical multigraphs (e.g. [1]). Moreover, indexing techniques for hierarchical multigraphs such as [4] and multi-dimensional indexes over key-value stores as in [13] can be incorporated in our system for providing fast access to individual records within the partitions created by our framework.

3. SYSTEM OVERVIEW

3.1 Definitions

Assuming a rooted hierarchy tree T, we denote the set of its leaves as $Leaves(T)$. We refer to a subtree rooted at a vertex $x \in T$ as T_x. For vertices u and v of T we say that v is *descendant* of u if there is a path descending from u to v, including the case that u equals v. Vertices u and v are called *comparable* in T if one of them is descendant of the other and *incomparable* in T if neither u nor v is a descendant of the other. Also, by $depth(u)$ we shall refer to the number of edges that have to be crossed so as to get from u to the tree root node.

A *hierarchical graph* is a multigraph $G(T, V, E)$, where T is a tree, V is the set of vertices in the graph, which are also the leaves of T (V=$Leaves(T)$) and E is a multiset of edges between the nodes in V. We assume that each instance of an edge is labeled with a unique identifier in order to be able to assign data on them (i.e. in the CDRs example, each edge is associated with a unique key that identifies the corresponding CDR).

In order to represent the set of edges between nodes in T and their relationships, we define the graph G_{T^2}. Each possible pair of nodes u and v in T is a vertex $[u, v]$ in G_{T^2} and we refer to u as the source node in T and v as the destination node in T. The edges in G_{T^2} imply a hierarchy inherited from the hierarchy described by tree T. In particular, there is an edge between nodes $[u_1, v_1]$ and $[u_2, v_2]$ in G_{T^2} if exactly one of the following conditions hold: (i) u_2 is a child of u_1 in T, (ii) v_2 is a child of v_1 in T.

Graph G_{T^2} is used in our framework in order to define our partitioning scheme on the edges of the hierarchical graph. Moreover, the nodes of G_{T^2}, as will be explained in Section 3.2, are used in order to model possible queries on this data. Then, the edges of G_{T^2} will determine partitions that contain relevant data for a query.

Each vertex $[u, v] \in G_{T^2}$ is a candidate partition $P_{u,v}$ to be ma-

terialized by our partitioning scheme. Keeping all G_{T^2} in memory is not a feasible solution at its size is quadratic on the size of T. As will be explained, our partitioning process progressively splits the hierarchical graph and constructs a *partition tree* T_P, which is a subgraph of G_{T^2}. When we need to locate partitions so as to insert new records or answer queries we use T_P as our lookup structure. An example partition tree T_P which follows the hierarchy of Figure 1 is shown in Figure 2

Finally, by $C_{u,v}$ we refer to a counter of how many records are contained in the partition $P_{u,v}$ and by $threshold$ to the value that when the number of records in a partition grows bigger than, the partition has to be split in smaller ones and then get dropped.

3.2 Hierarchical Queries

In our framework, retrieval of edges belonging to the hierarchical graph is accomplished via queries that are modeled using the hierarchy tree T. In particular a query $Q_{u',v'}$ is denoted as a query edge in T (see Figure 1). This query denotes our intention to retrieve all edges that have as source vertices the leaves of $T_{u'}$ and as destination vertices the leaves of $T_{v'}$. When such a query arrives we need to be able to decide which of the materialized partitions in T_P may contain relevant graph edges. In order to achieve that we traverse T_P in a top-down fashion and check each vertex $P_{u,v}$ against the query $Q_{u',v'}$. The partition is considered useful in answering the query when their respective source and destination vertices are comparable in T. We continue traversing T_P descending the useful partitions until we get to the active partitions (active partitions are these that contain data and are pointed by the leaf nodes of T_P as explained in Section 3.3), which are returned to the query for further processing (e.g. filtering of relevant edges). In case that during the traversal we come across a $P_{u,v}$ for which it stands that u is equal to u' and v is equal to v' then we stop the traversal and retrieve the leaf nodes in the subtree rooted at $P_{u,v}$. In that case, all the edges in the respective partitions are returned to the user.

In our running example, assuming that the partition tree T_P is at the state shown in Figure 2 and that we have to answer query $Q_{Athens,Messinia}$ that retrieves all CDRs from locations in Athens to locations in the state of Messinia, we first check the root of T_P which is [Greece, Greece]. Since Athens is comparable to Greece and Messinia is comparable to Greece we continue with examining the children of [Greece, Greece], which are [Attiki, Greece] and [Messinia, Greece]. Athens is comparable to Attiki and so is Messinia to Greece, so node [Attiki, Greece] is useful, but Athens is incomparable to Messinia so we do not have to further investigate the node [Messinia, Greece] or any node in $T_{Messinia,Greece}$. Next we have to check nodes [Attiki, Attiki] and [Attiki, Messinia] which are the children of [Attiki, Greece]. Athens is comparable to Attiki but Messinia is incomparable to Attiki so [Attiki, Attiki] is not considered useful. On the other hand [Attiki, Messinia] is useful since Athens is comparable to Attiki and Messinia is comparable to Messinia. [Attiki, Messinia] is a leaf node in T_P and so the traversal ends here and the partition pointed by [Attiki, Messinia] is the result returned to the query.

3.3 Overview of the Partitioning Process

A high level description of the partitioning process is as follows: In the beginning let T_P consist of just one vertex $[r, r]$ where r refers to the root of T. That means that we initially materialize just one global partition $P_{r,r}$ containing all possible edges amongst leaves in T. When $C_{r,r}$ grows greater than $threshold$ the split process is triggered. The split process decides whether $P_{r,r}$ will be split by its source or destination node, depending on the rules of the chosen partitioning policy, which is discussed later. Each outcome is encoded by a set of nodes that are reachable from node $[r, r]$ in graph G_{T^2}, depending whether the respective edge denotes a parent-child relationship on the source or destination node.

After the split, the vertices representing the new partitions are added to T_P. Vertex $[r, r]$ in T_P points no longer to an active partition but we keep it since it describes the records contained in the active partitions pointed by its descendants (the new vertices that we added) and we use this information when we traverse T_P in order to insert new records or answer a query. This process takes place for every active partition $P_{u,v}$ when $C_{u,v}$ grows greater than $threshold$ after the insertion of new records. This way the vertices in T_P that point, or previously have been pointing, to an active partition, form a hierarchical tree. At any moment the leaves of T_P point to the active partitions while the inner nodes, including the root, are "aggregations" of these partitions.

Any node in T_P can optionally maintain a series of useful application specific statistics such as the number of records in the partition, aggregations over measures of these records, calculations regarding heavy hitters such as top-k sources and top-k destinations, etc. Furthermore, when fast approximate answers are desired by the application (for example during exploratory data analysis or as a preview while the exact answer is computed) it is also possible to maintain synopses such as Sketches [14], Histograms [15, 16] or Wavelets [17, 18] on the nodes of T_P. Since these nodes are traversed while new data is added in the partitions, maintenance of these synopses can be easily incorporated in the process. While these extensions are applicable in our framework, their discussion is beyond the scope of this paper.

In what follows we describe the different split/partitioning policies that we implemented in our system and used in our experiments. First we describe two simple query agnostic policies and next, in more detail, a partitioning policy that we call Query-Driven Partitioning that decides the split to materialize by taking under consideration a set of queries that are most important to the user and makes the split decisions according to them.

Query-Agnostic Policies: The first two policies assume no previous knowledge about the interests of the user. Each of them though utilizes a different heuristic as explained below.

- **Round-Robin Partitioning:** Round-Robin is a simple approach to partitioning the hierarchical graph. Partitions in T_P that need to split, are split alternately by source or destination. This process results in creating balanced partitions in the sense that source and destination nodes in a partition have a maximum distance of one hierarchy level. That way the partitions created are not biased towards the source or destination nodes of the constituent edges.

- **Min-Split Partitioning:** Min-Split partitioning policy is a heuristic method, which tries to create the minimum number of new partitions, when an active partition overflows. This policy seems preferable when the goal is to create as few active partitions as possible, while keeping their size close to the selected threshold. Thus, when it has to make a split choice, it simply chooses between the candidate splits the one containing fewer partitions.

Query-Driven Partitioning: In Query-Driven Partitioning we assume that we have a prior knowledge of the queries that the users of the system are mostly interested in. So, when we have to make a split choice we chose the candidate split that suggests a partitioning better suited to answer the set of queries. In what follows we present the idea of Query-Driven Partitioning for hierarchical graphs in a formal way.

Let $P_{u,v}$ be a partition in the partition tree T_P and $Q_{u',v'}$ be a query asking for all records having as source and destination all the nodes that are leaves of the hierarchy tree's T subtrees $T_{u'}$ and $T_{v'}$, respectively. Recall that a partition is useful for answering a query if their respective source and destination nodes are comparable in T, otherwise the partition is pruned while navigating the partition tree in search of answers to the query.

For a useful partition, we can define a measure of the overhead that the retrieval of $P_{u,v}$ adds to the overall cost of answering the query by considering the number of records that belong to $P_{u,v}$ but are not part of the result of $Q_{u',v'}$. We can use this measure to make the split choice for a partition to be split, by calculating the query answering overhead for each of the candidate splits. Extending this, we can calculate the overhead not just for one query but for a set of queries that the users are mostly interested in.

In order to measure the overhead of a useful partition for a given query, we have to estimate the portion of the partition records that are not useful for the query, but will have to be retrieved when scanning the partition for relevant data. Since both the partition and the query follow the hierarchy implied by the hierarchy tree T we should check the partition's source and destination nodes u and v against the query's source and destination nodes u' and v', respectively. We will describe the procedure for u and u', but whatever we mention holds true also for v an v'.

Since nodes u and u' are comparable (otherwise the partition is not useful), we have to consider the following three cases: (i) u equals u', (ii) u is a descendant of u' ($depth(u) > depth(u')$, and (iii) u' is a descendant of u ($depth(u) < depth(u')$). Let $fitness(u', u)$ denote the portion of the leaf nodes in T_u that are also leaves in the subtree of $T_{u'}$. In the first and second cases we can safely infer that $fitness(u', u)$ equals to 1, since T_u is contained in $T_{u'}$. On the contrary, in the third case, $T_{u'}$ is contained in T_u and, thus, $fitness(u', u)$ is calculated by considering the ratio of the leaves of $T_{u'}$ over the leaves of T_u.

$$fitness(u', u) = \begin{cases} 0 & \text{, if } u \text{ and } u' \text{ are not comparable} \\ 1 & \text{, if } u \text{ and } u' \text{ are comparable} \\ & \text{and } depth(u) \geq depth(u') \\ \frac{|leaves(T_{u'})|}{|leaves(T_u)|} & \text{, otherwise} \end{cases}$$

Then, the fitness of the partition for the query is computed as:

$$fitness(Q_{u',v'}, P_{u,v}) = fitness(u', u) \cdot fitness(v', v)$$

Intuitively, this measure estimates the percentage of records in the partition that are useful for the query, assuming no additional knowledge on the data distribution is given.

In case the partition is split into k sub-partitions, assuming a uniform distribution of the records in $P_{u,v}$, then each of these partitions will receive $\frac{C_{u,v}}{k}$ of records, where $C_{u,v}$ is the size of the partition. Then, given that we have calculated the fitness for each of these smaller partitions denoted as $f_1, \ldots f_k$, respectively, we compute the *overhead* of the split as the number of non-useful to the query records expected to be retrieved from the set of partitions belonging to the candidate split as:

$$overhead(Q_{u',v'}, Split(P_{u,v})) = \frac{C_{u,v}}{k} \sum_{i=1\ldots k, f_i>0} (1 - f_i)$$

For a set of queries the cumulative overhead of the split is computed by summing the estimated overhead for each query. Thus, given a choice of splitting the partition by source of destination, we compare the overheads that we calculated for each of the splits and select to materialize the one with the lowest number. As have been explained, our calculations are based on the assumption that the data distribution within the partition is uniform. Of course, this is a bold assumption and we expect that the system may occasionally make wrong decisions, leading to suboptimal splits. An easy workaround is to consider additional statistics, for instance in the form of sketches, that will help better estimate the distribution of records within a partition, at the cost of increased overhead due to bookkeeping of these synopses. We leave exploration of such choices as future work.

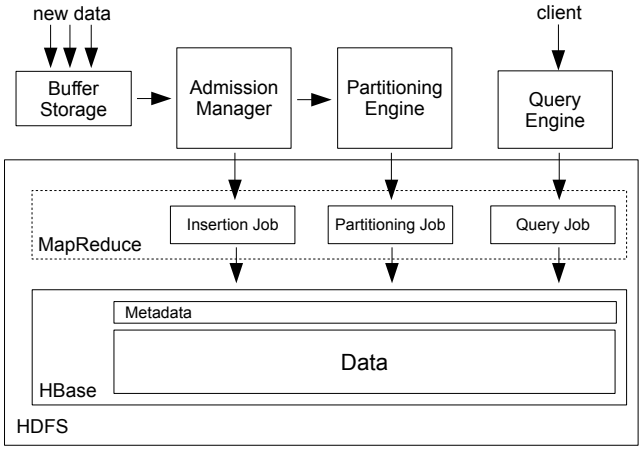


Figure 3: Framework overview

3.4 Implementation Overview

We implemented our system as a framework consisting of four main modules, the Buffer Storage, the Admission Manager, the Partitioning Engine and the Query Engine. It uses the well-known distributed data store HBase running on top of HDFS (the Hadoop Distributed File System) and also the Hadoop MapReduce engine in order to accomplish tasks such as loading and repartitioning of the data. We materialize each partition as an HBase table so as to make easier the retrieval of records belonging to one partition by involving the scan of one and only table. Another choice would be to use one big table to store all the records and define each partition space by applying a compound row-key design.

Figure 3 provides a high-level view of the framework. The Buffer Storage module is located at the input of the system and stores temporarily the new records that are streamed from distributed sources, into text files. When the size of these records grows bigger than a maximum buffer size, then the Buffer Storage module sends a message to the Admission Manager module, which executes the task of pulling the records from Buffer Storage and loading them into HBase. After each load of new records, Admission Manager checks for partitions that grew over the partitioning threshold. For any such partition, Admission Manager passes the required instructions to Partitioning Engine, which decides a split according to the selected policy amongst the ones described earlier and applies it.

For the support of these tasks, the system maintains a set of metadata, such as the hierarchy and partition trees that are used, updated and shared by all the different modules. The actual execution of the tasks is taking place as a number of MapReduce jobs, with the most important of them being the Insertion Job which puts each new record in the appropriate partition by looking up the hierarchy and partition trees, and the Partitioning Job which moves the data from a splitting partition to the new partitions.

Finally, we also implemented a Query Engine, which accepts a

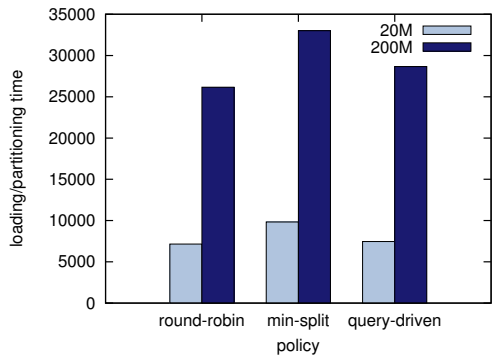


Figure 4: 20M vs 200M Records Loading Times

user query or set of queries and, by looking up the hierarchy and partition trees, discovers the partitions that possibly contain relevant records, scans them and returns the results to the user.

4. EXPERIMENTS

4.1 Experimental Setup

All experiments were conducted on a cluster of seven virtual machines located at the GRNET's cloud infrastructure Okeanos[1]. Each machine had 4 processors, 4 GB of memory, while the disk sizes varied from 40 to 100 GB. The operating system installed was Ubuntu Server 12.04 and we used Hadoop version 1.1.2 and HBase version 0.94.6.1. We used one machine as the system master running the NameNode, JobTracker and HMaster daemons, while each of the remaining six slave machines were running instances of the DataNode, TaskTracker and HRegionServer daemons.

In order to be able to generate massive graph data records, we wrote a custom CDR data generator. We used the geopolitical hierarchy of Greece as described by the Ministry of Interior[2] to create the hierarchy tree and then we generated a number of phone numbers for each city in it. The resulting hierarchy tree consisted of five levels, which from top to bottom are Country, District, State, City and Phone Number, each of them having a node count of 1, 13, 58, 324 and 1134698 nodes, respectively. The phone numbers consist of regular phone numbers that make and receive calls, inbound-only phones as those found in customer service departments or telemarketing, and outbound-only phones as those used by marketing agencies. Each record in the experiment data sets consists of a source and destination phone number and a unique call record id. Last, we created a random set of 10 queries which we used to guide the Query-Driven policy, but also to examine the performance of each policy in answering them. The queries were picked randomly amongst all possible queries having as source or destination inner nodes of the hierarchy tree, meaning we excluded the root (country) and the leaves (single phone numbers).

4.2 Loading/Partitioning Time

The first experiment we conducted is a comparison of the loading times for each of the partitioning policies we implemented. We created a data set of a total of 50 million CDR records and broke it into an initial ingest set of 20 million records and 10 append sets of 3 million records each. For each of these 11 steps we traced the total time it took to insert the records in HBase and perform the repartitioning of the schema, when needed. The results of this experiment are summarized in Figure 5. We can see that the Min-Split and Query-Driven policies spent slightly more time in repartitioning the hierarchical graph. This is explained since Min-Split follows a conservative approach of repartitioning the data by taking the minimum number of splits at each step, resulting in more subsequent splits. The Query-Driven policy is often keen to repartition the data by extending the partitioning tree in order to best fit the input queries that drive its selection process. Finally, in Figure 4 we compare the scalability of our framework using a larger number of input data (200 million records) for each policy. Compared to the smaller dataset, the Figure suggest a sublinear increase of the loading times, something that is contributed to the overhead times of the underlying frameworks (e.g. MapReduce jobs setup) that affects more (proportionally) the smaller input.

[1]https://okeanos.grnet.gr/
[2]http://www.ypes.gr/

4.3 Partitions' Quality

In order to examine the quality of the partitions created by each of the policies tested, we propose a measure that we call $distance$. This metric can be used in applications where the goal is to derive the fewer number of partitions that are each smaller or equal to the selected threshold. Given the actual final partition sizes $size_i$, where i=1 . . . p_m and p_m is the number of the active partitions for policy m at the end of the loading phase, we define $distance_m$ to be:

$$distance_m = \sqrt{\sum_{i=1}^{p_m} (size_i - threshold)^2}$$

Intuitively, a smaller distance value denotes a set of partitions that are created evenly near, but not exceeding, the selected threshold. We have calculated and present the value of the distance metric for each policy and after each loading/partitioning job in Figure 6(a).

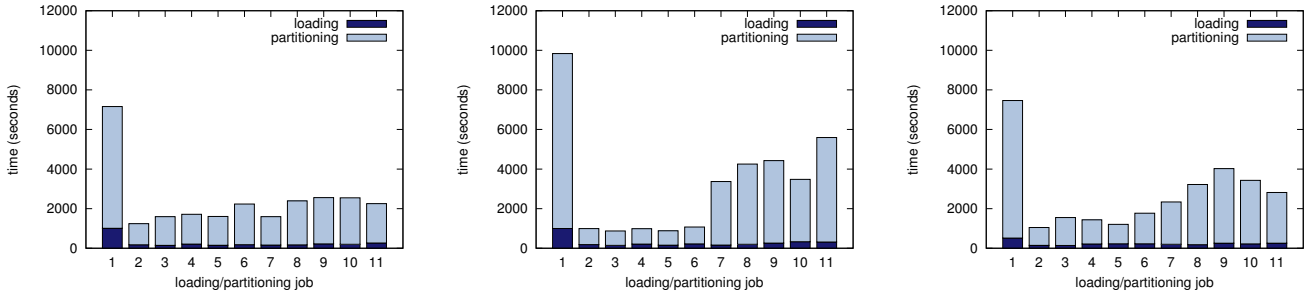
What is worth noticing in this Figure is that the Min-Split policy, while in the beginning was the best amongst the others, later on it created partitions that had a larger collective distance. This is explained by the fact that the choice of the smaller split, leads to many deep splits of the partition tree and, subsequently, when the leaves are reached, the deep partitions that need to be split are getting split in high levels of the hierarchy (on the opposite direction) since this is the only feasible split left. This fact also leads to the increase of the number of active partitions that is evident in Figure 6(b). Thus, even though at each step Min-Split makes (locally) optimal decisions regarding the split that leads to the smaller increase of the distance metric, the final resulting partitioning is worse that the ones achieved by the other policies.

4.4 Queries Answering

In order to examine the effectiveness of the dynamic partitioning schema, for each of the policies we ran the set of queries mentioned in Section 4.1. We used these same queries to guide the Query-Driven policy. In Figure 6(c) we can see a comparison of the total records retrieved by each of the policies, and the fraction of them that were useful in answering these queries. As expected the Query-Driven policy has better precision than the other two policies, while the Min-Split policy, which has gone deep in the hierarchy tree, was not able to create partitions suitable for the selected set of random queries.

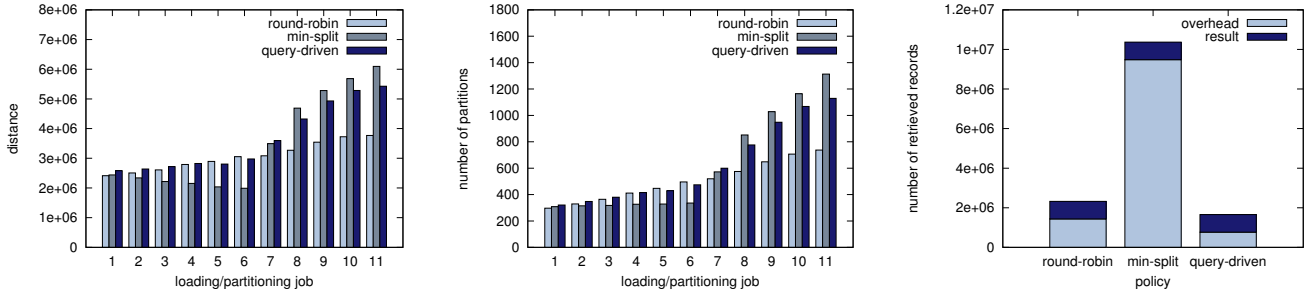
5. CONCLUSIONS

In this paper we considered the problem of managing big hierarchical graphs by exploiting the implied hierarchy so as to partition the data edges in a way that would better support future retrieval



(a) Round-Robin (b) Min-Split (c) Query-Driven

Figure 5: Policies' loading and partitioning times



(a) Partitions' distance (b) Number of active partitions (c) Queries

Figure 6: Comparison between policies

and analysis. We evaluated a number of dynamic partitioning policies using open source big data tools on a small cluster of nodes. From the policies considered, the Query-Driven partitioning policy lead to partition schemes that enable faster analysis of the records, assuming that some a-priori knowledge of the user queries is given. In an uncertain environment, the Round-Robin policy seems to result in more balanced partitions with good query performance. Moreover, it has been shown to have the best performance in the loading and partitioning processes.

6. REFERENCES

[1] J. Abello and J. Korn, "Visualizing massive multi-digraphs," in *Proceedings of INFOVIS*, pp. 39–47, 2000.

[2] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Comput. Netw.*, vol. 33, June 2000.

[3] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," *SIGCOMM Comput. Commun. Rev.*, vol. 29, pp. 251–262, Aug. 1999.

[4] J. Abello and Y. Kotidis, "Hierarchical graph indexing," *Proceedings of the twelfth international conference on Information and knowledge management*, 2003.

[5] "Apache HBase." `http://hbase.apache.org/`.

[6] "Apache Hadoop." `http://hadoop.apache.org/`.

[7] J. Dean and S. Ghemawat, "MapReduce : Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 1–13, 2008.

[8] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *Proceedings of IPDPS*, pp. 124–124, 2006.

[9] I. S. Dhillon, Y. Guan, and B. Kulis, "Weighted graph cuts without eigenvectors a multilevel approach," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, Nov. 2007.

[10] K. Tzoumas, A. Deshpande, and C. S. Jensen, "Sharing-aware horizontal partitioning for exploiting correlations during query processing," *Proc. VLDB Endow.*, vol. 3, pp. 542–553, Sept. 2010.

[11] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proceedings of ACM SIGMOD*, pp. 517–528, 2012.

[12] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The little engine(s) that could: scaling online social networks," *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 375–386, Aug. 2010.

[13] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "MD-HBase: A scalable multi-dimensional data infrastructure for location aware services," 2011.

[14] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[15] F. Reiss, M. N. Garofalakis, and J. M. Hellerstein, "Compact histograms for hierarchical identifiers," in *VLDB*, 2006.

[16] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "Fast, small-space algorithms for approximate histogram maintenance," in *STOC*, pp. 389–398, 2002.

[17] A. Deligiannakis, M. N. Garofalakis, and N. Roussopoulos, "Extended wavelets for multiple measures," *ACM Trans. Database Syst.*, vol. 32, no. 2, 2007.

[18] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "One-pass wavelet decompositions of data streams," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 3, pp. 541–554, 2003.