2013 IEEE International Conference on Big Data

Managing Massive Graphs in Relational DBMS

Ruiwen Chen
Simon Fraser University, Burnaby BC, Canada
Email: ruiwenc@sfu.ca

***Abstract*—Massive graphs emerge in many real-world applications. Practitioners often find relational databases are inefficient in graph data management. In this paper, we investigate the efficiency issue by analyzing both I/O and CPU costs. First, we find the storage of a graph in relational DBMS violates the *locality principle*: graph queries will always reference neighbors; however, the data locations of neighbors are almost random. To solve this problem, we introduce *partitioned graph storage* as a new database design option. It combines database partitioning with available graph-partitioning algorithms to restructure the storage such that neighbors are located close to each other. Second, we find graph queries expressed with SQL introduce unnecessary overheads. To overcome the CPU costs, we propose a new storage access method, which we call *graph scan*, to retrieve neighbors in one single operation.**

We show experimentally that partitioned graph storage and graph scan can significantly reduce I/O and CPU costs. We conclude that a relational DBMS could be a good graph store, as long as the storage respects the locality principle and SQL overheads are eliminated.

I. Introduction

Many real-world applications are being built on massive graphs, for example, social networks, web graphs, geographical networks, recommender systems, etc. It is highly desirable to have a general-purpose graph data management system. Such a system needs to address the following characteristics in massive graphs:

- Large. Real-world graphs have grown to have millions to billions of vertices: the web graph has hundreds of billions of vertices [13], and the largest social network has hundreds of millions of active users.
- Dynamic. The graph structures are frequently updated online, mostly by adding new vertices and edges.
- Irregular. Autonomously emerged graphs often exhibit extremal imbalances, that is, a non-negligible fraction of vertices have very large degrees.
- Query-intensive. Applications based on massive graphs require instant online responses for graph queries.

Efficient management of massive graphs is challenging. Many application developers resort to relational DBMS by storing graphs as relational tables. However, they often find this is not efficient enough [5], [20], [9]. The goal of this work is to find approaches to improving the efficiency of massive graph management in relational DBMS.

We are mainly concerned with the efficiency of local queries on graphs, where a local query is one that explores the neighborhood of a vertex. We propose a generic algorithmic template for processing such local queries. We address I/O and CPU costs of local queries from two perspectives: the storage structure of a graph in disk blocks and the accessing method to graph neighbors in query processing.

First, we find the storage of a graph in a relational DBMS violates the *locality principle* [10], which says that data locations accessed in a short period of time often close to each other. The violation is in the following sense: a local query will always reference neighbors in a graph, but the data locations of neighbors are almost unpredictable. Conventional optimization techniques such as clustered indexes, materialized views or vertical partitioning, which respect the locality principle, do not work on graphs. A first solution is to apply available graph-partitioning (clustering) algorithms to restructure the storage such that neighbors are physically located close to each other. This approach improves the storage structure. However, there are still several issues: (1) graph-partitioning algorithms usually do not scale to external memory; (2) irregular graphs may not have good partitioning at all. We propose a hybrid framework which combines database horizontal partitioning with graph partitioning, and addresses both the graph connectivity and the visiting frequencies of vertices. By a design of *partitioned graph storage*, our experimental results show that, the I/O costs for local queries can be reduced significantly.

The second issue we tackle is the SQL overheads for graph queries. Finding neighbors using recursive self-joins or loops of selections induces unnecessary overheads on the query optimizer and executor. We take a different perspective by viewing the neighborhood relationship on vertices as a selection condition, and propose a new storage access method, which we call *graph scan*, to retrieve graph neighbors in one single operation. We experimentally show that, this approach reduces the CPU costs noticeably.

Figure 1 shows our newly introduced components and their counterparts in a conventional relational DBMS. Our conclusion is that, a relational DBMS can still be an efficient graph store, as long as *its graph storage respects the locality principle and the SQL overheads are eliminated.*

The paper is organized as follows. Section 2 provides background on the data model, systems and datasets. Section 3 introduces local queries in our theme and also a local algorithm template which underlies our implementation of several graph algorithms. Section 4 addresses the I/O costs and introduces the partitioned graph storage. Section 5 focuses on the SQL overheads and introduces the graph scan access path. This is followed by discussions, related work, and our conclusion.

User Defined Functions (**Graph Package**)		
SQL Query Engine		
Sequence Scan	Index Scan	**Graph Scan**
Horizontal Partitioning	Vertical Partitioning	**Graph Partitioning**

Fig. 1. Components of a Relational DBMS.

II. Preliminaries

In this section, we list our assumptions on graphs and queries. We review available techniques of storing and querying graph data in both relational and graph databases. We also describe the graph datasets and our test environment.

A. Graphs and Application Scenarios

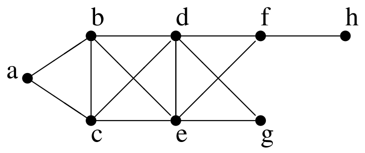
We consider directed graphs, with labeled vertices and edges. A *(directed) graph* is a pair $G = (V, E)$ where V is a set of vertices V and E is a set of edges; each *(directed) edge* is an ordered pair of vertices. We represent undirected graphs as directed graphs where each undirected edge is replaced by exactly two directed reciprocal edges. We allow vertices and edges labeled by attributes following a common schema. For instance, a typical social network has individuals as vertices and friendships as edges, where the vertices may be labeled by "name" and "age", and the edges may be labeled by "timestamp".

A *walk* is a sequence of vertices where each consecutive pair of vertices is an edge of the graph. When a walk has no repeated vertices, we call it a *path*. Given two distinct vertices, we define the *distance* between them as the length of a shortest path connecting them. Given a vertex v, its *d-hop neighborhood* is the subgraph induced by the set of all vertices within distance d from v.

In this paper, we consider graphs that are massive, sparse, and dynamic. That is, the graph may have hundreds of millions of vertices; meanwhile, vertices on average have relatively small degree. We also allow updates on graphs such as adding or removing vertices or edges, but we assume such updates are not frequent; this means that we can rebuild the storage when a certain fraction, say 20%, of the graph has changed. We note that such assumptions allow us to capture real-world scenarios such as social network applications and recommender systems. These assumptions distinguish our work from previous works which assume small or unchangeable graphs.

For queries on graphs, we mainly consider online queries regarding a small neighborhood of some specified vertex. We call such queries as *local queries*. As an example, in a social network, we may wish to find friends of an individual whose names satisfy a certain condition. We assume these queries are very frequent and require instant responses. Note that, rather than traversing the whole graph, these queries will only necessarily access a small part of the graph.

We consider graph traversals as offline operations, which will happen infrequently. Such operations are useful in activities such as building indexes and generating statistics.



AdjList

v	neighbor
a	[b, c]
b	[a, c, d, e]
c	[a, b, d, e]
. . .	. . .
h	[f]

(a)

EdgeTable

v	u
a	b
a	c
b	a
. . .	. . .
h	f

(b)

Fig. 2. A graph with its (a) adjacency list, and (b) edge table.

B. Graph Data in Relational DBMS

One can store a graph in a relational DBMS either as a table of adjacency lists or as a table of edges, as shown in the example in Figure 2. Although the adjacency-list representation does not meet first normal form (1NF), it saves storage and is more efficient for neighborhood exploration; this is supported by most systems with the "array" data type. The edge-table representation is necessary when the edges are labeled by other attributes. In this paper we will mainly use the adjacency-list representation; however, the techniques developed are also applicable to the edge-table representation.

There are several ways of find a neighborhood using SQL: self-joins, recursive queries, and loops of selections. For the graph in Figure 2, the following self-join query[1] finds vertices of distance 3 to a starting vertex 'a'.

```
SELECT DISTINCT t3.v, t3.neighbor
FROM AdjList t1, AdjList t2, AdjList t3
WHERE t1.v = 'a' AND t2.v = ANY(t1.neighbor)
    AND t3.v = ANY(t2.neighbor);
```

Practitioners often use self-join due to its simplicity; however, self-join queries can only express graph traversals up to a small constant distance.

Recursive queries provide an alternative approach. Recursive queries can express graph traversals of arbitrary distance, and thus have strictly stronger expressive power than self-joins [1]. The following recursive query retrieves the 3-hop neighborhood of 'a' in the graph in Figure 2.

```
WITH RECURSIVE SearchAdjList(v, neighbor, depth) AS (
        SELECT v, neighbor, 0
        FROM AdjList WHERE v = 'a'
    UNION ALL
        SELECT DISTINCT A.v, A.neighbor, S.depth + 1
        FROM AdjList A, SearchAdjList S
        WHERE A.v = ANY(S.neighbor) AND S.depth ≤ 2
) SELECT DISTINCT v, neighbor FROM SearchAdjList;
```

This query first initializes a result set with the starting vertex, and then iteratively joins the result set with the adjacency list until the result set does not change. Note that it has explicit duplicate eliminations in each recursion as well as in the final result.

[1]In this paper we follow the SQL syntax of PostgreSQL; other systems may have slightly different syntax specifications.

A third approach of neighborhood querying is to use a procedure language to define a loop of selections which visit the vertices one at a time.

Although we have shown three ways of neighborhood querying in relational DBMS, the efficiency of these queries is not promising; several work [20], [9], [5] have shown both analytically and experimentally that these queries are not efficient on large graphs. In this paper we establish several ways to reduce the I/O and CPU costs of these queries.

C. Datasets and Test Environment

We examine on the massive graphs listed in Table I. Except GRID-($10^4 \times 10^4$), all others are from real-world applications, including social networks, web graphs, road networks, internet networks. Dataset TWITTER is from the Social Computing Data Repository at ASU [21], and the others are from Stanford Large Network Dataset Collection[2]. We converted directed graphs into undirected graphs by adding reciprocal edges; the reason is that most applications require finding neighbors in both directions (in social network terms, both followers and followees).

The number of vertices in a graph ranges from 0.4 million to 100 million, and the number of edges ranges from 4.8 million to 400 million. The average degree of vertices is below 20, but in most cases (except GRID and ROADNETCA), the degree distribution is highly skewed (these graphs follow the power-law distribution, where a noticeable fraction of vertices have very large degrees).

We test on the relational DBMS PostgreSQL 9.0.4, configured with 512 MB shared buffers and 16 KB block size. We store the graphs as adjacency-list tables, and build B-Tree indexes on the vertex identifier attribute. The disk sizes of the tables and indexes are also in Table I.

III. LOCAL QUERIES

In this section we give an algorithmic template for processing local queries. We also analyze the visiting frequency of vertices based on a model of local queries.

A. A Local-Algorithm Template

A *local query* on a graph finds a neighborhood of a given vertex following certain criteria. For example, on a social network, one may wish to find friends of friends by exploring a 2-hop neighborhood; in recommender systems, the user would like to see items that are frequently bought together; on a road network, one may wish to find a nearby store location with road connections. A significant characteristic of such queries is that, for a given query, it is only necessary to look at a small part of the graph (of constant or sub-linear size).

Processing local queries on massive graphs could be more involved. To name a few obstacles, different queries may require different processing algorithms; in a highly connected graph, a constant-hop neighborhood may cover a big portion of the graph. To provide a unified algorithmic framework, Spielman and Teng [19] initiated a study of local algorithms for massive graphs. Roughly speaking, a *local algorithm* starts with a specified vertex, and then it iteratively examines only vertices which are neighbors of those seen before. There is no computation involving examining all vertices. For efficiency guarantees, one should provide stopping criteria and a threshold on the size of the explored neighborhood.

```
Input:
  A graph $G = (V, E)$ and a vertex $v \in V$;
  Thresholds $t_A$ and $t_F$, and stopping criteria.
Procedure:
  Initialize ACTIVE $= \{v\}$, FROZEN $= \{\}$.
  While ACTIVE is not empty:
    1) Expand ACTIVE with respect to threshold $t_A$;
    2) Truncate ACTIVE and generate a subset $F$;
    3) Append $F$ to FROZEN;
    4) Compute a goal based on ACTIVE $+$ FROZEN;
    5) Break if the size of FROZEN exceeds threshold
       $t_F$, or stopping criteria are met;
```

Fig. 3. A local-algorithm template.

We design a generic template for local algorithms as shown in Figure 3. The algorithm utilizes three sub-procedures **Expand**, **Truncate** and **Compute**, which are dependent on the queries. Following the template, a local algorithm starts from a given vertex, and then it iteratively examines neighbor vertices until thresholds or stopping criteria are reached. In particular, we divide vertices into three subsets: ACTIVE, FROZEN, and the unvisited. ACTIVE maintains vertices in the boundary of the neighborhood exploration; FROZEN maintains visited vertices which are not in the boundary but are still useful for computation. In each iteration, using **Expand** we add unvisited but promising neighbors to ACTIVE; then we reduce the size of ACTIVE using **Truncate**, where we may also move certain vertices from ACTIVE to FROZEN. The iteration terminates when thresholds or stopping criteria are met.

We implement the collection ACTIVE using a priority queue structure, where each vertex is associated with a priority value. FROZEN, which contains vertices with stabilized priority values, is append-only. We remark that the threshold t_A on the size of ACTIVE sacrifices the completeness of the search. When the threshold does not exist, we can cast many algorithms completely into the template, for example, breath-first search, best-first search, Dijkstra's algorithm, etc. However, in order for the algorithm to be feasible on massive graphs, the thresholds are necessary to tradeoff efficiency with completeness.

Several previous works also exploited the idea of truncating the searching boundary. The beam search algorithm [6], which is widely used on AI problems with exponentially-growing search space, runs a breath-first search but keeps only a small part of the search space at each depth level. The graph local-clustering algorithm in [19] maintains a small neighborhood by truncating random walks with low probabilities.

The running time of the algorithm depends on the sub-procedures provided. In this work we require **Expand**, **Truncate** and **Compute** to be running in time nearly-linear, that is $O(n \log^c n)$ where n is bounded by $t_A + t_F$ and c is a constant. Under such assumptions, the whole algorithm runs

[2]http://snap.stanford.edu/data/

Graph	$\lvert V\rvert$	$\lvert E^d\rvert$	$\lvert E\rvert$	AvgDeg.	MaxDeg.	Table	Index	Edge Description
GRID-($10^4 \times 10^4$)	100.00	400.00	400.00	4.00	4	7.6 GB	2.2 GB	Coordinate Neighbor
TWITTER	11.317	85.332	127.11	11.23	564795	1168 MB	253 MB	Following
LIVEJOURNAL	4.848	68.994	86.221	17.78	20334	635 MB	108 MB	Friendship
WIKITALK	2.394	5.021	9.319	3.89	100029	175 MB	54 MB	Talk to
ROADNETCA	1.965	5.533	5.533	2.82	12	136 MB	44 MB	Road connection
WEBGOOGLE	0.876	5.105	8.644	9.87	6332	86 MB	20 MB	Web link
AMAZONPROD	0.403	3.387	4.887	12.13	2752	53 MB	9 MB	Product co-purchase

TABLE I. GRAPH DATASETS. THE NUMBERS OF VERTICES AND EDGES ARE IN MILLIONS.

in time nearly-linear in the explored neighborhood size.

We will use this local-algorithm template to instantiate a local graph-partitioning algorithm in Section IV, and a graph-scan accessing method in Section V.

B. Local Computation

Let $G = (V, E)$ be a graph where $V = \{1, \ldots, n\}$ and let M be an $n \times n$ matrix, which could be a simple transformation from the adjacency matrix or the distance matrix of G. Let $x = (x_1, \ldots, x_n)$ be a vector indexed by vertices in the graph. Consider the following iterative process. Initialize a vector $x^{(0)}$, and then iteratively compute $x^{(t+1)} = x^{(t)}M$, until it converges. This iterative process can be instantiated to compute values such as shortest-path lengths and PageRank. However, it is not efficient. Suppose we run the process for t iterations, the total running time is $O(tn^2)$.

On sparse graphs, the computation simplifies using the following identity: $\Delta x^{(t+1)} = \Delta x^{(t)}M$, where we define $\Delta x^{(t+1)} = x^{(t+1)} - x^{(t)}$. One may expect that, for sparse graphs, most entities in the vector $\Delta x^{(t)}$ vanish, and from the computational perspective we only need maintain a small portion of the vector.

In the local-algorithm template we defined in Figure 3, we further impose a threshold function δ on the changes $\Delta x^{(t+1)} = \delta[\Delta x^{(t)}M]$, which guarantees that only a bounded number of entities $\Delta x^{(t)}$ will not vanish. The validity of such threshold functions can be found in [6], [19] for different computation problems. Therefore, roughly speaking, the collection ACTIVE maintains the changes at each iteration, while FROZEN maintains the necessary history along the computation. While the history helps computing the final goal function, in many cases, the history can be completely discarded if the goal function depends only on the priority values maintained in ACTIVE.

C. Vertex Visiting Frequency

Now we analyze how frequent a vertex will be visited by local queries. We establish a relationship between the visiting frequency of a vertex and its local structure.

Consider the following random querying model. We start from a random vertex v, branch out to a bounded number b of the neighbors of v, and then randomly pick up a fraction r of these neighbors to recursively continue the process; each branch terminates when reaching a distance d. The breadth bound b and distance bound d resembles the real-world resource bounds. A real-world scenario could be that, on a social network, a user first checks information on a bound number of friends, and then finds friends of a fraction of these friends.

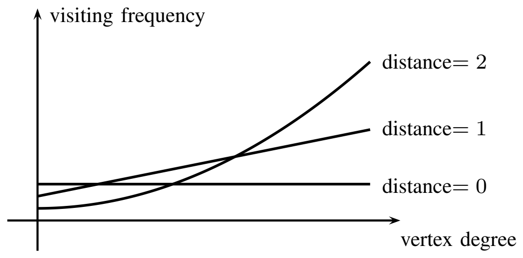


Fig. 4. Vertex visiting frequency vs. the degree.

For an individual vertex, its visiting frequency depends on its neighborhood structure. For the trivial case $d = 0$, each vertex is visited uniformly randomly, with no visits to neighbors. This corresponds to the conventional assumption on relational database queries. For $d = 1$, $r = 1$ and unbounded b, the visiting frequency is proportional to the degree plus 1. In this case each edge is visited uniformly randomly. This is the mostly assumed case in graph-partitioning research. For a constant d, $r = 1$ and unbounded b, the visiting frequency is proportional to the size of the d-hop neighborhood. On average, the visiting frequency increases polynomially as the degree increases.

A complete characterization would need specific assumptions on the degree profile and the connectivity of the graph. However, we can conclude that for distance $d > 1$, the visiting frequency increases super-linearly as the degree increases; this is different from the usual assumption in graph-partitioning research.

Another simple observation is that, for two vertices which are far from each other, they will not be visited together in one local query.

IV. PARTITIONED GRAPH STORAGE

In this section we propose a partitioned graph storage as a database-design option to tackle the I/O costs of graph queries. We investigate several graph-partitioning approaches in our querying context, and give a guideline on partitioning massive graphs in a relational DBMS. In particular, we stress that the partitioning should not only depend on the graph connectivity but also on the visiting frequencies of vertices.

A. Sequential Graph Storage

With the adjacency-list representation, each vertex and its neighbor edges are stored as one tuple in a database table. The whole graph is represented by a collection of such tuples in a table, and the tuples are distributed over disk blocks. At a top level, the block-based storage essentially partitions the graph into almost equal-sized parts. However, there is no rule guiding such a partitioning; the system is unaware of the graph connectivities, and this partitioning is far from optimal.

Let G be a graph with n vertices and m edges, and we wish to divide the vertices into disjoint parts, each of size at most a constant K. Practically this constant K depends on the database block size. A random partitioning is such that vertices are randomly distributed over the blocks. Consider an arbitrary edge. With a random partitioning, the probability that the two endpoints of the edge resides in the same block is $\frac{K-1}{n-1} \approx \frac{K}{n}$, which is tiny since the number of vertices n could be very large. By the linearity of expectation, this implies on average a fraction $(n-K)/n$ of the edges are crossing blocks. This storage structure means that retrieving a neighbor vertex almost always requires reading a new block. This strongly violates the locality principle [10], which says that data locations accessed in a short period of time should form clusters. The current relational DBMS have no ways to rectify this worst case.

This situation can be improved significantly by only a simple procedure. Consider a graph traversal path, which gives an ordering of vertices such that (most) consecutive pairs are connected. We sequentially store the vertices following the order; this gives a partitioning such that the fraction of edges with endpoints in the same block is $\frac{K-1}{K} \cdot \frac{n}{m}$. This is much better than a random partitioning. In the next two subsections, we take graph connectivity and vertex visiting frequencies into consideration, and that further improves the storage for local query processing.

B. Graph-Partitioning Algorithms

The *graph-partitioning problem* is to divide a graph into parts, such that the parts have about the same size and there are few connections between the parts. A good partitioning algorithm tries to minimize the number of edges between vertices in different parts. This problem is well-known to be NP-complete [12]. However, this hardness result only characterizes the worse case. Given its practical importance, graph partitioning has been widely studied in many fields such as parallel computing, sparse matrix computation, circuit design, social network analysis, etc.; severial efficient approximate algorithms have been successfully applied in practice.

The class of (heuristic or approximate) graph-partitioning algorithms is diversified. Below we analyze the applicability of these algorithms in partitioning massive graphs in relational DBMS.

Global vs. Local Algorithms. Several well-studied graph-partitioning algorithms produce approximately optimal solutions, including the spectral graph partitioning [8], Leighton-Rao flow-based algorithm [16], and Arora-Rao-Vazirani geometric embedding approach [3]. Another line of work is on heuristic methods, such as Kernighan-Lin algorithm [15]. Although these algorithms succeed in many applications, they are not efficient enough for processing massive graphs; the best implemented Arora-Roa-Vazirani algorithm runs in $O(n^{1.5})$ time [3], and Kernighan-Lin algorithm is in $O(n^2 \log n)$ time [15]. Researchers have resorted to the divide-and-conquer approach to partitioning large graphs, namely the *multilevel partitioning* [4], [14], which first "coarsen" the input graph into a smaller one, and then iteratively generate refined partitioning level by level. However, this approach uses the above-mentioned algorithms as sub-routines, which dominates the efficiency.

Spielman and Teng's local clustering algorithm Nibble [19] finds a cluster via local expansions on graphs. The algorithm starts from a given vertex, and expands to neighbors via *truncated random walks*; the cluster produced consists of vertices that are reachable by many walks. The graph-partitioning algorithm based on Nibble [19] runs in nearly-linear time, and statistically guarantees that the output is approximately optimal. Although the algorithm is still not practical due to a large hidden constant, it is promising that efficient local algorithms could be good even for hard problems like graph partitioning.

Partitioning vs. Ordering. For storing massive graphs in databases, it is more desirable to have an ordering of vertices instead of a collection of parts. This is because that data attributes associating with vertices may require varied storage spaces. An ordering of vertices provides more flexibility in generating partitions.

Many graph-partitioning algorithms can be viewed as embedding a graph into a low-dimensional metric space [3]. A good embedding is such that the connectivity in the graph is reflected by the distance in the metric space; with a good embedding, a random separation in the metric space should give a good partitioning of the graph. Upon this, it will be easier to derive an ordering and then a sequential partitioning of the vertices.

C. Partitioning Respecting Power Laws and Vertex Visiting Frequencies

Many real-world graphs follow the power-law distribution, that is, the fraction of vertices having degree k is proportional to $1/k^{\beta}$, where β is a number slightly larger than 2 [11], [8]. For power-law graphs, most graph-partitioning algorithms do not work well [2], since the algorithms tend to be dominated by high-degree vertices.

Even worse, almost all graph-partitioning algorithms are based solely on the graph connectivity; they do not consider the visiting frequencies of vertices, which is essential in our local-query context. As shown in Section III-C, the query visiting frequency increases super-linearly as the degree increases (as long as the distance in the queries is bigger than 1). The means a few portion of high-degree vertices receives the majority of the visits, while a large portion of low-degree vertices receive very few visits.

This characteristic leads to the following partitioning scheme: *partition the graph into three parts (1) high-degree vertices, (2) low-degree vertices, and (3) the rest.*

Let $G = (V, E)$ be a power-law graph with degree distribution $f(k) = \alpha k^{-\beta}$, where α is a constant and β is slightly bigger than 2. Assume that the visiting frequency of a k-degree vertex follows the distribution $g(k) = bk^{c}$, where b is a constant, and $c > 1$ is a number depending on the distance of local queries. Figure 5 plots the degree distribution in contrast with the query visiting frequency.

We wish to find a splitting degree k^* such that the number of vertices with degree greater than k^* is small, whereas the

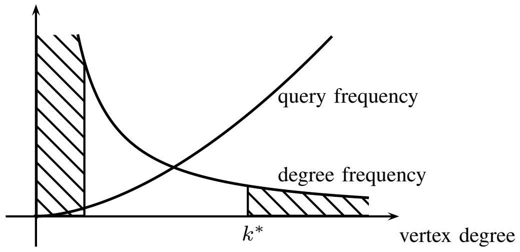


Fig. 5. Partitioning respecting power laws and query visiting frequencies.

accumulated frequency of visits to such vertices is large. More formally, let γ be a parameter balancing the two goals, and we wish to minimize the following:

$$\begin{aligned} & \sum_{k=k^*}^{\infty} f(k) - \gamma \sum_{k=k^*}^{\infty} f(k)g(k) \\ = & \sum_{k=k^*}^{\infty} \alpha k^{-\beta} - \gamma \sum_{k=k^*}^{\infty} \alpha k^{-\beta} b k^{c} \\ = & \frac{\alpha}{\beta-1} k^{*1-\beta} - \frac{\gamma\alpha b}{\beta-1-c} k^{*1+c-\beta}. \end{aligned}$$

In this degree-based partitioning, the part of high-degree vertices respects the locality principle since data locations that are frequently accessed cluster together. The part with low-degree vertices receives very few visits, and also the visiting cost is small because of the small neighborhood. For the rest part with medium-degree vertices, the degree distribution tends to be less skewed because of the removal of high-degree vertices; this follows from the result on network resilience under targeted attackes [7], [18]. This allows us to apply the conventional graph-partitioning algorithms discussed in Section IV-B.

D. Implementing Partitioned Graph Storage

An important fact in real-world graphs is that vertices are often associated with *ground-truth labels*, which indicate natural clusters. For example, in road networks, vertices are often associated with geographical locations; in the publication-author network, publication venues, such as conferences or journals, serve as natural identifiers of communities; in who-buys-what relationships, product categories are meaningful labels; and in web graphs, domain names naturally group the URLs. Although ground truth provides no guarantees on the partitioning quality, Leskovec et al. [17] shows that ground truth gives good natural clusters (communities) in many real-world networks. For massive graphs that are too large to be processed in memory, one may use ground truth to generate parts that are small enough to be fed into in-memory partitioning algorithms.

Real-world graphs may differ in many aspects: size, degree, connectivity, irregularity, etc. It is impossible to have one single partitioning algorithm which works on all types of graphs. A practical approach for a DBMS is to provide a library of schemes and algorithms. The user may choose specific algorithms and parameters based on the knowledge on the data, or even the user can plug in with their own algorithms. In our framework, we provide the following partitioning options:

- Partitioning based on ground truth. This is useful to partition big graphs into parts that fit in memory.
- Partitioning based on graph statistics such as vertex degree. This is useful for power-law graphs.
- A simplified implementation of the local-clustering algorithm Nibble [19]. It traverses the graph locally by simulating truncated random walks. The algorithm runs efficiently and generates an ordering of vertices.
- The mutilevel heuristic-based graph-partitioning scheme METIS [14], [2], which is widely used in experimental settings.

Our guideline for partitioning a massive graph is the following: (1) partition the graph based on some ground-truth labels; (2) if the graph is highly skewed, partition it based on graph statistics such as the degree profile; (3) apply in-memory partitioning algorithms on the generated subgraphs. An implementation of a partitioning requires the user to analyze ground truth and the degree profile, and specify in-memory partitioning algorithms and related parameters. The partitioning will either assign each vertex with a part identifier, or output an ordering of all vertices; then a post-processing follows to distribute vertices into disk blocks.

E. Experiments on Partitioned Storage

In this subsection we present our experimental results on partitioned graph storage. [3]

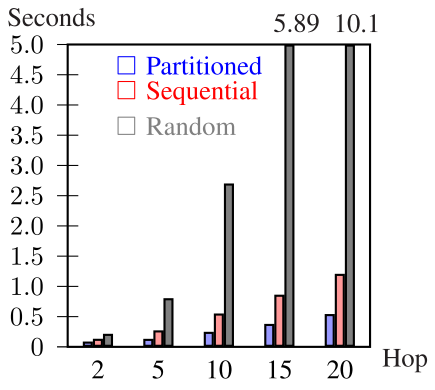
Small Graphs. We first applied partitioning on the relatively small graphs (storage size less than 512 MB), with results in Table II. These graphs have about $4 \sim 9$ millions of edges. We compared three types of storage layout: random ordering, sequential ordering on vertex identifiers, and partitioning. Since these graphs are small, instead of giving the actual I/O costs, we show two statistical metrics on a complete analysis of the storage: the percentage of edges that are between vertices in the same block, and the average number of blocks that contain a vertex and all of its neighbors (1-hop neighborhood). As shown in Table II, ROADNETCA has a very good partitioning, and we expect this will scale to larger road networks. WEBGOOGLE and AMAZONPROD also have good partitioning, with more than 70% of edges inside blocks and the 1-hop neighborhood clustering in $2 \sim 3$ blocks. The partitioning for WIKITALK is not as good as others. The result was from the local-clustering algorithm, while METIS gives a slightly worse result. This graph is highly skewed: the top 0.05% high-degree vertices (1197 out of 2.4 million) connect with 64% of the edges. For such graphs we suggest degree-based partitioning as discussed in Section IV-C.

Big Grid Graph. We use GRID-$(10^4 \times 10^4)$ as a testbed for big graphs. The storage size about 10 GB. We built three storage layouts: random ordering, sequential ordering by rows in the grid, and partitioning. The partitioning was a hybrid of horizontal partitioning and local clustering: we partition the grid by rows into 20 parts, and then apply local clustering

[3]The tests in this paper were conducted with the following environment: operation system: OS X V10.6.8, Linux kernel: Darwin 10.8.0, CPU: 2.3 GHz Intel Core i5, main memory: 4 GB 1333 MHz DDR3, hard disk rotational rate: 5400, and file system: Journaled HFS+.

Graph	%In-Block Edges (16k)			#Blocks 1-Hop Neigh.			Partition Alg.
	Rand.	Seq.	Part.	Rand.	Seq.	Part.	
WIKITALK	0.01%	0.54%	17.1%	4.60	3.91	3.36	LocalCluster
ROADNETCA	0.01%	71.2%	95.2%	3.81	1.68	1.13	METIS 10000
WEBGOOGLE	0.02%	0.02%	73.6%	10.72	10.72	2.24	METIS 6000
AMAZONPROD	0.03%	12.4%	70.4%	13.04	8.92	3.34	METIS 3000

TABLE II. STATISTICS ON PARTITIONED GRAPH STORAGE.



(a)

Hop	2	5	10	15	20
#Neighbors	13	61	221	481	840
#Read Part.	1.98	3.78	7.71	12.78	18.25
#Read Seq.	5.06	11.36	22.87	34.98	47.94
#Read Rand.	12.84	60.74	216.7	456.4	787.5

(b)

Fig. 6. Neighborhood queries on GRID-($10^4 \times 10^4$). (a) Running time. (b) Physical I/O counts.

on each part. We tested neighborhood queries starting from random vertices. The running time and physical I/O counts are reported in Figure 6. (All querying results in this paper were averaged over 300 runs of random queries.)

Large Power-Law Graphs. We tested on two large social networks LIVEJOURNAL and TWITTER, which follow the power-law distribution, and have large storage size. We first derive some statistics in Table III to assist us choosing appropriate partitioning schemes. LIVEJOURNAL has 4.8 million vertices, 86 million edges, with storage size exceeding 700 MB. Single partitioning algorithm does not scale well on this large power-law graph (neither local clustering nor METIS). Our approach is to first partition it by vertex degree: top 1% high-degree vertices, vertices with degree one, and the rest; and then apply graph partitioning on each part. The query we tested is the following: given a vertex v, find vertices u sharing common neighbors with v, and return the top 100 of them ordered by the number of common neighbors. In social network terms, this is to find friends via mutual friends. We expressed this query using join, aggregation and ordering, and tested on the storage without and with partitioning. Table IV reports the results. (In this case only, we configured the database with 64 MB shared buffers in order to better measure the I/O costs. We tested each query twice in sequence, in which the first query measures the total time and the second one roughly measures the CPU time.)

TWITTER graph has 11 million vertices and 127 million edges. We partitioned the graph into top 0.1% high-degree vertices, degree-one vertices, and the rest, which is further partitioned using local clustering. The query is to count the number of vertices within distance 2 from a random vertice. Table V shows the running time and the physical I/O counts.

	Total Time (ms)	CPU Time (ms)
No Part.	186.85	16.38
Partititoned	56.28	15.77

TABLE IV. RUNNING TIME OF FINDING TOP 100 VERTICES WITH COMMON NEIGHBORS ON LIVEJOURNAL.

	Total Time (ms)	# Physical Read
No Part.	73.34	6.80
Partititoned	19.57	4.34

TABLE V. RUNNING TIME AND PHYSICAL I/O FOR FINDING 2-HOP NEIGHBORHOOD ON TWITTER.

V. GRAPH SCAN

In this section we address the CPU costs of graph queries expressed with recursive self-joins and loops of selections. We introduce "graph scan" as a new storage access method to overcome the SQL overheads in graph queries.

A. *Graph Scan: A New Access Method*

As seen in Section 2, one may use a recursive self-join query or a loop of selections to retrieve the neighborhood of a vertex. This approach is widely adopted by developers due to its simplicity. However, the SQL overheads with these queries are relatively expensive. Using a recursive self-join query to find a d-hop neighborhood requires d rounds of joins, together with projections and duplicate eliminations. The costs of joins and projections are largely unnecessary. And such overheads grow super-linearly as the distance d grows. On the other hand, using a loop of selections requires running one selection query on each vertex encountered, which also introduces unnecessary SQL overheads.

In contrast, the prevalent graph database systems often natively implement graph-traversal functionalities. In particular, a programmer can invoke some lower-level interfaces to directly retrieve and manipulate neighbors of an individual vertex.

To overcome the unnecessary SQL overheads, we propose a new storage access method, which we call *graph scan*. It complements the conventional sequential scan and index scan. A graph scan fetches tuples by following a graph-traversal order. It starts from a given vertex, and then retrieves its neighbors by following its edges; it maintains a collection of vertex identifiers as scan keys, and recursively proceeds until certain stopping criteria is met.

This is analogous to a B-Tree index scan for a range query, in which we maintain two scan keys, namely the lower bound

Graph	AvgDeg.	MaxDeg.	Top 0.1% Deg.	Top 1% Deg.	% Degree-1 Vertices
TWITTER	11.23	564795	1207	104	59.6%
LIVEJOURNAL	17.78	20334	553	180	21.0%

TABLE III. STATISTICS ON LARGE POWER-LAW GRAPHS.

Hop	2	5	10	15	20
#Neighbors	13	61	221	481	840
Recursive Join (ms)	1.28	2.19	8.89	22.07	48.03
Selection Loop (ms)	0.79	2.12	7.11	16.07	36.77
Graph Scan (ms)	0.71	0.84	1.61	3.24	5.26

TABLE VI. CPU TIME OF NEIGHBORHOOD QUERIES ON GRID-($10^4 \times 10^4$).

Hop	2	5	10	15	20
#Neighbors	8.61	43.1	208	583	1250
Recursive Join (ms)	0.76	1.49	7.46	27.57	80.18
Selection Loop (ms)	0.67	1.71	6.56	19.46	51.17
Graph Scan (ms)	0.43	0.52	1.22	2.27	4.10

TABLE VII. CPU TIME OF NEIGHBORHOOD QUERIES ON ROAD NETWORK ROADNETCA.

and the upper bound, descend down the tree to locate the first entry, and then follow the link on the leafs to find other entries until the upper bound is met. In a graph scan, we maintain a queue of scan keys and also an explicit stopping condition for the traversal.

Graph scan can be implemented as a local-algorithm instance with customized searching order and stopping condition. This follows from the local-algorithm template defined in Section III. As a single operation, graph scan does not involve any joins or projections, and duplicate eliminations are done along the way.

B. Experiments on Graph Scan

We tested graph scan using neighborhood queries starting from random vertices. The results show that CPU costs are reduced significantly. In particular, we compared three expressions of neighborhood queries: (1) Recursive join query. We use standard SQL recursive query, which makes joins iteratively to find neighbors level by level. Projections and duplicate eliminations are done as early as possible in the iteration. (2) Loop of selections. We create a function using a procedure language; it maintains a queue of vertices and iteratively issues a selection query on each vertex. (3) Graph scan. We implemented graph scan for breath-first search. A query specifies a starting vertex and a distance; the graph scan retrieves all vertices up to the distance. Internally, it maintains a queue of vertex identifiers as scan keys.

We tested the three approaches on GRID-($10^4 \times 10^4$) and ROADNETCA to find the neighborhood of a vertex up to a given distance. In order to measure the CPU time, for GRID-($10^4 \times 10^4$), we executed each query twice, where the first one measures both I/O and CPU costs, and the second one measures only the CPU costs. Table VI shows the result.

For ROADNETCA which is of small size, we pre-loaded the data in memory and then executed the queries. Table VII gives the result. We see that, for neighborhoods of distance 10 to 20, graph scan is about $5 \sim 10$ times faster than recursive joins and loops of selections.

VI. CONCLUSION AND FUTURE WORK

In this paper we propose an algorithmic template for local queries on graphs. We introduce two novel features for managing massive graphs in relational DBMS: (1) partitioned graph storage, which respects the locality principle and reduces the I/O costs, and (2) graph scan access method, which reduces the CPU overheads. We conclude that a relational DBMS can still be an efficient graph store, as long as its storage respects the locality principle and the SQL overheads are eliminated.

REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] A. Abou-rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS). In*, pages 16–575. press, 2006.

[3] S. Arora, S. Rao, and U. Vazirani. Geometry, flows, and graph-partitioning algorithms. *Commun. ACM*, 51:96–105, October 2008.

[4] S. T. Barnard and H. D. Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency Practice and Experience*, 6(2):101–117, 1994.

[5] C. Becker. Rdf store benchmarks with dbpedia (http://www4.wiwiss.fu-berlin.de/benchmarks-200801/), 2008.

[6] R. Bisiani. Beam search. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. John Wiley & Sons, 1987.

[7] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38, June 2006.

[8] F. R. K. Chung. *Spectral Graph Theory*, volume 92. American Mathematical Society, 1997.

[9] P. Cudré-Mauroux and S. Elnikety. Graph data management systems for new application domains (tutorial). In *VLDB*, 2011.

[10] P. J. Denning. The locality principle. *Commun. ACM*, 48:19–24, July 2005.

[11] D. Easley and J. Kleinberg. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, 2010.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[13] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In WWW '05, pages 902–903. ACM, 2005.

[14] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, December 1998.

[15] B. W. Kernighan and S. Lin. An efficient heuristic for partitioning graphs. *Bell Systems Technical J*, 49:291–307, 1970.

[16] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46:787–832, November 1999.

[17] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In WWW '08, p 695–704.

[18] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. Anf: A fast and scalable tool for data mining in massive graphs. In ACM KDD'02, 1:81–90.

[19] D. A. Spielman and S. Teng. A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning. *CoRR*, abs/0809.3:1–25, 2008.

[20] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In ACM SE '10, pages 42:1–42:6. ACM, 2010.

[21] R. Zafarani and H. Liu. Social computing data repository at ASU, 2009.