A Framework for SQL-Based Mining of Large Graphs on Relational Databases

Sriganesh Srihari[1], Shruti Chandrashekar[2], and Srinivasan Parthasarathy[3]

[1] School of Computing, National University of Singapore, Singapore 117590
[2] New Jersey Institute of Technology, Newark, NJ 07102
[3] The Ohio State University, Columbus, OH 43210
srigsri@comp.nus.edu.sg, sc297@njit.edu, srini@cse.ohio-state.edu

Abstract. We design and develop an SQL-based approach for querying and mining large graphs within a relational database management system (RDBMS). We propose a simple *lightweight framework* to integrate graph applications with the RDBMS through a tightly-coupled *network layer*, thereby leveraging efficient features of modern databases. Comparisons with straight-up main memory implementations of two kernels - breadth-first search and quasi clique detection - reveal that SQL implementations offer an attractive option in terms of productivity and performance.

Keywords: Graph mining, SQL-based approach, Relational databases.

1 Introduction

Over the past few years data generated from real-world processes have increasingly attracted the attention of researchers from all domains. A lot of effort has gone into analyzing this data from different perspectives to extract valuable information. In this respect, mining of *graph data* has always demanded its share of lime-light. This is primarily because graphs are ubiquituous and many *real world* scenarios are modeled as graphs. For example, the physical interactions between proteins in an organism are modeled as a protein interaction graph with proteins as vertices and their interactions as edges. Protein interaction graphs are a major resource for knowledge discovery: detecting protein complexes, predicting protein functions and reliabilities of interactions [10].

There are many efficient techniques developed for storing and manipulating graphs in main memory (RAM): for traversals, answering reachability queries, mining frequent patterns, etc. [5] However, as more and more graph data is accumulated, it is not feasible to store and manipulate entire graphs in main memory. Therefore, graphs are stored on disks and efficiently fetched into main memory in parts for manipulation [2]. The computational power of processors is increasing, while the speed gap between main and secondary (disk) memories is widening. Therefore, graphs are compressed and stored on disks so that they can be retrieved in parts with as little I/O reads as possible, and uncompressed quickly in main memory [1]. To summarize, these approaches used to

M.J. Zaki et al. (Eds.): PAKDD 2010, Part II, LNAI 6119, pp. 160–167, 2010.

handle graphs can be classified broadly into two categories: (a) efficient storage and manipulation of graphs in main memory; (b) efficient storage and indexing of graphs on disks and their retrieval into main memory (out-of-core approach).

As graph sizes continue to grow larger, it will be interesting to look at alternative approaches to mine large graphs (any data in general). The **SQL-based approach** for integrating mining with RDBMS (relational database management systems) was proposed long back (in 1998) for association rule mining [9], followed by k-way join variants for association rule mining in 2003 [6], Subdue-based substructure mining in 2004 [3], and frequent subgraphs mining in 2008 [4]. The SQL-based approach proposed storing data in relational databases and mining it using SQL queries. Even though this approach was considered novel and promising, the idea was mainly constrained to transactional datasets, and never became popular for mining graphs. One probable reason, we believe, was the complications (awkwardness) involved in "mapping" graphs onto the *relational framework* of a RDBMS. This involved expressing the whole problem (graph data, storage and manipulation) declaratively (using SQL).

In spite of the non-trivial nature of the SQL-based approach, it can be very useful and promising. The RDBMS displays data to the designers and programmers as relational structures, while internally it stores this data on disk blocks using efficient disk-based data structures (example, B+ trees). Hence, if we can reasonably "map" graph data structures and algorithms onto relational structures, then we can leverage all the services RDBMS can offer: handling dynamic updates, buffer management, indexing and retrieval, and parallelism. After all, more than two decades of research has gone into making database systems fast, scalable, robust, and concurrent. Secondly, in many instances, main memory and out-of-core implementations can get exceedingly non-trivial. However, the development and deployment time of SQL-based code can be significantly shorter because one can avoid denormalizing data and storing into flat files prior to data mining, and also writting code for indexing and retrieval [9].

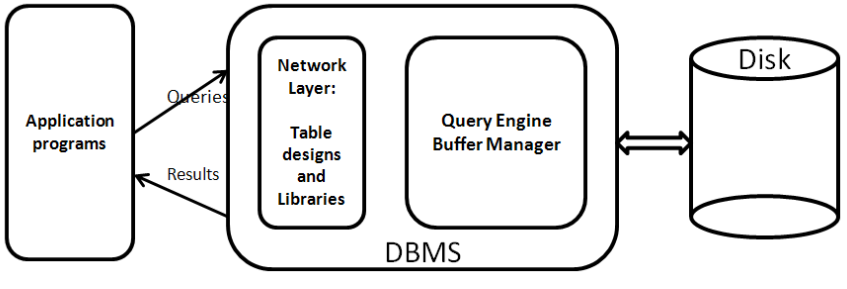


Fig. 1. Proposed framework for SQL-based mining of graphs on RDBMS

2 Our Proposed Framework

The main contribution of our work is to propose a **lightweight framework** for SQL-based mining of graphs on RDBMS. It is shown in Figure 1. This framework is designed for making effective use of the RDBMS features for efficient handling of large graphs. The **network layer** forms the most important component of the framework. Several graph mining applications can be "mapped" onto the RDBMS through the services offered by this layer.

2.1 The Network Layer

The network layer rests conceptually within the RDBMS (see Figure 1) and runs in the same address space as the RDBMS. It is implemented using procedural SQL (stored procedures using Oracle's PL/SQL [8]). The advantage of implementing this way is that the network layer is *tightly-coupled* to the RDBMS: it has direct access to all the services offered by the RDBMS. This layer provides the necessary *graph-abstraction* to abstract away all the complications involved in handling large graphs. It houses all the basic table designs and 'utilities'. Graph applications can either be implemented loosely-coupled or tightly-coupled to the RDBMS. For loosely-coupled applications, the network layer acts as a *translation layer* (For example, converting C or Java calls into SQL queries), while for tightly-coupled applications (written in procedural SQL), it provides ready-to-use libraries and utilities.

2.2 Efficient Storage of Graphs in the Network Layer

The basic schema design consists of storing all graphs $G = \{G_1, G_2, .., G_k\}$ in a hierarchical 'master-detail' fashion in the following tables: a graph table Graph(GraphId, NoOfVertices, NoOfEdges), a vertex table Vertex(GraphId, VertexId), and a connectivity table AdjMatrix (GraphId, Vertex1, Vertex2). For every graph $G_i = (V_i, E_i) \in G$, there is a record (tuple) in Graph, uniquely identified by the primary key $\{\text{GraphId}\} \leftarrow \{G_i\}$. For every vertex $v \in V_i$ of graph G_i, there is a record in Vertex, uniquely identified by primary key $\{\text{GraphId, VertexId}\} \leftarrow \{G_i, v\}$. The whole connectivity structure is then stored as records in AdjMatrix. For every edge $(u, v) \in E_i$, there is a record in AdjMatrix uniquely identified by the primary key $\{\text{GraphId, Vertex1, Vertex2}\} \leftarrow \{G_i, u, v\}$. Notice how *GraphId* is propagated as part of the primary key in all tables. The whole graph G_i can be uniquely queried from the tables using *GraphId*.

2.3 Implementing a Basic Utility within the Network Layer: BFS

We next describe how a basic utility like the breadth-first search (BFS) on a graph is efficiently implemented within the network layer.

The BFS algorithm on a graph $G_i = (V_i, E_i)$ and its SQL-based design are shown in Algorithm 1. We first store the graph G_i in the above-proposed tables. To simulate the FIFO queue used in BFS, we design a table Queue (Line: 1).

Algorithm 1. BFS(G, s)

```
1: Initialize queue Q; /* Create table: Queue(GraphId, VertexId, position) */
2: enqueue(Q, s);
3: while Q ≠ empty do
4:    v ← dequeue(Q); /* Query: SELECT record with MIN position */
5:    for each unvisited neighbor u of v do
6:       enqueue(Q, u); /* Insert into Queue. */
7:       mark u as 'visited'; /* Update: 'visited' in Discovery. */
8:       assign a discovery number to u;
9:    end for
10:   if commitCnt ≥ commitFreq then
11:      COMMIT and reset commitCnt; /* Controlled COMMITs to restrict I/O.*/
12:   end if
13: end while
14: Output the vertices in discovery sequence;
```

For every vertex $v \in V_i$ that is enqueued, there is a record in Queue, uniquely identified by {GraphId, VertexId} $\leftarrow \{G_i, v\}$. The *position* attribute in Queue gives the position of v in the queue. The smaller the *position*, the earlier v will be dequeued. Additionally, for every vertex $v \in V_i$, there is a record in table Discovery, uniquely identified by the primary key {GraphId, VertexId} $\leftarrow \{G_i, v\}$. There are attributes *visited* and *discoveryNo* to keep track of whether v has been visited and its order in the visited sequence.

The BFS algorithm begins by inserting the source s into Queue (Line: 2). In each iteration, the vertex v with the minimum *position* is selected (Line: 4) from Queue. All unvisited neighbors u of v (Line: 5) are then selected from the join: AdjMatrix A $\bowtie_{A.Vertex1=v \wedge A.Vertex2=D.VertexId \wedge D.Visited=FALSE}$ Discovery D. These vertices are inserted into Queue (Line: 6) and updated as 'visited' in Discovery (Line: 7, 8). These iterations continue till Queue is empty.

2.4 Extending to Graph Mining: Quasi Clique Detection

Quasi cliques are very interesting structures from the point of view of graph mining. Very simply, a quasi clique in a graph is an 'almost' complete subgraph. Quasi cliques are used to model real-world communities in protein networks, social networks, scientific collaboration networks, etc. [10]

There are several ways to model quasi cliques; one way is by the notion of a γ-*quasi clique*. Given a graph $G = (V, E)$, a subgraph $Q = (V_Q, E_Q)$, $V_Q \subseteq V$ and $E_Q \subseteq E$, is called a quasi clique with clustering co-efficient $0 \leq \gamma \leq 1$ if, $|E_Q|$ is at least a γ-fraction of the total possible number of edges in a subgraph of the same size. This is given by: $|E_Q| \geq \gamma.\binom{|V_Q|}{2}$. Therefore, the number of edges missing in Q is given by: $\lambda \leq (1-\gamma).\binom{|V_Q|}{2}$.

To study quasi clique detection on our framework, we chose the algorithm proposed in [10]. We only give the essense of the algorithm here so that the purpose of our work is served; for details see [10]. The inputs to the algorithm

are graph $G = (V, E)$, and fixed *parameters* $k > 0$ and $\lambda \geq 0$. The algorithm performs a bounded recursive search to find a quasi clique $Q \subseteq V$ of size at most k with at most λ edges missing. The time complexity of the algorithm is $O(3^{k+\lambda}.f_{poly}(|V|))$. The recursive search procedure makes the algorithm highly memory-intensive with the amount of additional memory required per search-path being $O((k+\lambda).g_{poly}(|V|))$, which can be very large. This also reflects how non-trivial the memory management can be in such applications, especially when implemented in-memory or out-of-core.

2.5 The RCR Strategy in SQL-Based Approach

In order to implement the quasi clique algorithm using the SQL-based approach, we made use of the earlier proposed table designs. Additionally, we designed the following interesting strategy, which we call **replicate-cleanup-rebuild** (RCR). This strategy can be adopted to other recursive algorithms as well.

Algorithm 2. bool QCRecursive $(G, Q, V \setminus Q, k, \lambda)$: recursive call

1: $I = \{G, Q, V \setminus Q, k, \lambda\}$; /* Input I from parent call.*/
2: c = generateCallNo();
3: Working(GraphId, CallNo, Info) $\leftarrow \{G, c, I\}$; /* Replicate into Working. */
4: Pick an edge (u, v);
5: $I' = \{G', Q' = Q \cup \{u\}, V \setminus Q', k' = k - 1, \lambda\}$; /* Include u into solution.*/
6: **if** Q' is the required solution **then** return TRUE along with Q';
7: r = QCRecursive$(G', Q', V \setminus Q', k', \lambda')$; /* Send new values to first child. */
8: **if** r = TRUE **then** return TRUE along with the solution;
9: $I' = \emptyset$; /* Clean-up current values. */
10: $I' \leftarrow$ Working(G, c); /* Rebuild from Working.*/
11: Repeat for subsequent children.

In this strategy, each call *replicates* (stores an additional copy) all the values received from its parent into a working table Working (see Algorithm 2). It makes its computations on the received values and passes the results to its child. When the child backtracks, instead of reverting back each computation, the current computed values are blindly *cleaned-up* (discarded), and the original values are *rebuilt* (queried) from Working. Subsequently, new computations are performed on these original values and sent to the next child. Also, when a child call backtracks, its records are permanently deleted from Working. The records stored for each call c are uniquely identified by the primary key {GraphId, CallNo} $\leftarrow \{G_i, c\}$ in Working. Considering $h_{poly}(|V|)$ number of records inserted per call, the total number of records in Working is $O((k+\lambda).h_{poly}(|V|))$.

Notice the intuition behind this strategy: to remove all the non-trivial memory management (local storage of values, and reverting back of computations from unsuccessful paths) within the calls and instead rely on the RDBMS for efficient mechanisms. It also illustrates how code development time can be significantly shorter using the SQL approach.

3 Empirical Evaluation

We compared SQL-based implementations against straight-up main memory implementations for: (a) breadth-first search (BFS) as a basic graph utility, and (b) quasi clique detection as a graph mining application. We implemented the main memory versions of the algorithms in C++ using the g++ 4.1.2 compiler on the GNU/Linux Debian distribution (2.6 kernel) installed on an Intel Xeon Dual Core 2.4GHz machine with 3GB RAM, 2.7GB swap partition and 250GB hard disk. Whenever the memory requirement was more than 3GB we relied on virtual memory. The procedural SQL versions were implemented in PL/SQL [8] on Oracle 10g on the same machine.

3.1 Evaluation of Breadth-First Search

We first compared the two implementations of BFS: (a) main memory (referred as BFSiMM) *versus* (b) procedural SQL (referred as BFSiSQL).

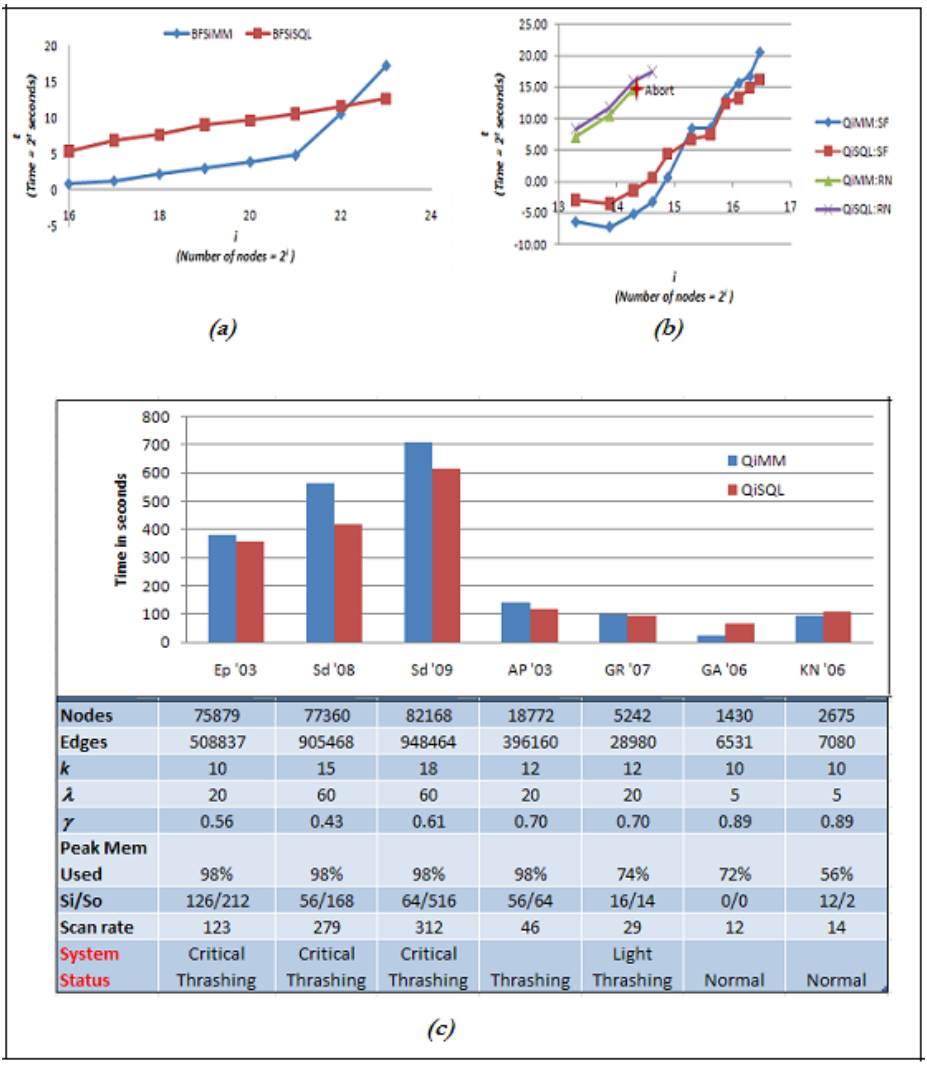
We generated random networks of n nodes and $m = 4n$ edges by replacement (that its, selecting m times nodes u and v such that $u \neq v$ and removing the edges between duplicated pairs). Figure 2(a) shows the comparison plots of runtimes (seconds) on networks for n between 2^{16} to 2^{23}. The figure shows that even though BFSiMM performed better than BFSiSQL for small networks, BFSiSQL outperformed BFSiMM for large networks.

3.2 Evaluation of Quasi Clique Detection

We next compared the two implementations of the quasi clique algorithm: (a) main memory (referred as QiMM) *versus* (b) procedural SQL (referred as QiSQL).

We generated scale-free networks with $n = 10$K to 90K ($\sim 2^{13.28}$ to $\sim 2^{16.45}$), and random networks with $n = 10$K to 40K ($\sim 2^{13.28}$ to $\sim 2^{15.28}$) vertices. We clustered them and stored co-clustered vertices on close-by disk blocks. Very small quasi cliques are easy to find and are not interesting, therefore we set $k = 25$ and $\lambda = 180$, giving $\gamma \geq \{\binom{k}{2} - \lambda\}/\binom{k}{2} = 0.4$. In each execution, 20 γ-quasi cliques were detected by iteratively deleting the current quasi clique and searching for the next one in the remaining network. Figure 2(b) shows the comparison of runtimes (in lg scale) for QiMM and QiSQL. It shows that even though QiMM performed better than QiSQL for small networks, QiSQL outperformed QiMM for large networks. For scale-free networks, this cross-over occurred around 60K ($\sim 2^{15.7}$) nodes. For random networks of size 25K ($\sim 2^{14.6}$), QiMM continuously aborted finding only 13 quasi cliques, while QiSQL found all 20 quasi cliques.

We next considered a variety of real-world networks obtained from [7]. These included social (Epinions: Ep'03, Slashdot: Sd'08 and Sd'09), scientific collaborations (Astro-physics: AP'03, General Relativity: GR'07) and protein interaction networks (Gavin: GA'06, Krogan: KN'06). Figure 2(c) shows the comparisons for fixed k and λ. It shows that QiSQL outperformed QiMM for all networks, except the small ones like PPI GA'06 and KN'06.



Nodes	75879	77360	82168	18772	5242	1430	2675
Edges	508837	905468	948464	396160	28980	6531	7080
k	10	15	18	12	12	10	10
λ	20	60	60	20	20	5	5
γ	0.56	0.43	0.61	0.70	0.70	0.89	0.89
Peak Mem Used	98%	98%	98%	98%	74%	72%	56%
Si/So	126/212	56/168	64/516	56/64	16/14	0/0	12/2
Scan rate	123	279	312	46	29	12	14
System Status	Critical Thrashing	Critical Thrashing	Critical Thrashing	Thrashing	Light Thrashing	Normal	Normal

(c)

Fig. 2. (a) BFSiMM Vs BFSiSQL; (b) QiMM Vs QiSQL on scale-free and random networks; (c) QiMM Vs QiSQL on real-world networks

Analysis of deteriorating performance of QiMM: Even though the synthetic and real-world networks considered in the quasi clique experiments resided completely in main memory, QiMM displayed worse behavior compared to QiSQL for the larger networks. This was primarily because of the significant amount of additional memory required for recursive calls, which subjected QiMM to heavy *thrashing*. See Figure 2(c). *Snapshots* of memory usage (from *top* and

vmstat) of the overall system when QiMM was executing showed 100% RAM and 100% CPU usage. The high swap-in (si) and swap-out (so) values (always zero while not thrashing) clearly indicated critical thrashing. The high scan indicated wastage of CPU cycles while waiting for the page handler to scan for free pages.

4 Conclusions and Future Work

In this work we have proposed a lightweight framework to extend the SQL-based approach to mine large graphs. We showed that this approach outperformed straight-up main memory implementations for BFS and quasi clique detection on large graph datasets. It will be interesting to realize our framework on grid technologies (like Oracle 10g/11g) for mining large graphs in a parallel distributed fashion.

Acknowledgements. We would like to thank Hon Wai Leong (NUS) and Anthony Tung (NUS) for the interesting discussions during this work, and the reviewers for their valuable comments. SP would like to acknowledge support from the following NSF grants – IIS-0917070, CCF-0702587, IIS-0347662. SS is supported under the NUS research scholarship.

References

1. Aggarwal, C., Yan, X., Yu, P.S.: GConnect: A connectivity index for massive disk-resident graphs. In: Very Large Databases (VLDB), vol. 2, pp. 862–873 (2009)
2. Chen, W., et al.: Scalable mining of large disk-based graph databases. In: ACM Knowledge Discovery and Data Mining (SIGKDD), pp. 316–325 (2004)
3. Chakravarthy, S., Beera, R., Balachandran, R.: DB-Subdue: Database approach to graph mining. In: Dai, H., Srikant, R., Zhang, C. (eds.) PAKDD 2004. LNCS (LNAI), vol. 3056, pp. 341–350. Springer, Heidelberg (2004)
4. Chakravarthy, S., Pradhan, S.: DB-FSG: An SQL-based approach for frequent subgraph mining. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) DEXA 2008. LNCS, vol. 5181, pp. 684–692. Springer, Heidelberg (2008)
5. Jin, R., et al.: Efficiently answering reachability queries on very large directed graphs. In: ACM Management of Data (SIGMOD), pp. 595–608 (2008)
6. Mishra, P., Chakravarthy, S.: Performance evaluation and analysis of k-way join variants for association rule mining. In: James, A., Younas, M., Lings, B. (eds.) BNCOD 2003. LNCS, vol. 2712, pp. 95–114. Springer, Heidelberg (2003)
7. Network datasets, `http://snap.stanford.edu/data/index.html`
8. Oracle PL/SQL, `http://www.oracle.com/technology/tech/pl_sql/index.html`
9. Sarawagi, S., Thomas, S., Agarwal, R.: Integrating mining with relational database systems: Alternatives and implications. In: ACM Management of Data (SIGMOD), pp. 343–354 (1998)
10. Srihari, S., Ng, H.K., Ning, K., Leong, H.W.: Detecting hubs and quasi cliques in scale-free networks. In: IEEE Pattern Recognition (ICPR), pp. 1–4 (2008)