PageRank for Billion-Scale Networks in RDBMS

Aly Ahmed and Alex Thomo
{alyahmed,thomo}@uvic.ca

University of Victoria, BC, Canada

Abstract. Data processing for Big Data plays a vital role for decision-makers in organizations and government, enhances the user experience, and provides quality results in prediction analysis. However, many modern data processing solutions make a significant investment in hardware and maintenance costs, such as Hadoop and Spark, often neglecting the well established and widely used relational database management systems (RDBMS's). PageRank is vital in Google Search and social networks to determine how to sort search results and how influential a person is in a social group. PageRank is an iterative algorithm which imposes challenges when implementing it over large graphs which are becoming the norm with the current volume of data processed everyday from social networks, IOT, and web content. In this paper we study computing PageRank using RDBMS for very large graphs using a consumer-grade server and compare the results to a dedicated graph database.

Keywords: PageRank · One Billion Graph · RDBMS · Graph Database · Big Data · Matrix partitioning

1 Introduction

With the amount of data produced daily, one of the main challenges facing big data is filtering out data and identifying the wheat from the precious. As search results tend to be in millions of pages, ranking search results becomes very crucial, however ranking pages tends to be one of the most difficult problems as the search engine is required to present a very small subset of results and order them by relevance. A variety of ranking features such as page content or hyperlink structure of the web are used by Internet search engines to come up with a good ranking. Many algorithms have been proposed to sort query results and return the most relevant pages first. Among them are PageRank [12], HillTop [4] and Hypertext Induced Topic Selection (HITS) algorithms [6].

PageRank [5, 12] developed by the Google founders is based on the hyperlink structure and on the assumption that high ranked pages usually contain links to useful pages, therefore giving more weight to pages that have more inbound links from high weighted paged. PageRank is an iterative algorithm and executed on the whole graph, which, in Google's case, is very large. Algorithms such as PageRank for big data sets require extensive hardware setups and, in many cases, distributed computing, such as Hadoop [14] and Spark [13]. This is because we

cannot fit the whole data set in one machine's memory. Building such a setup comes with significant investment and continuous running costs.

Nevertheless, some of the existing systems such as Relational Database Management Systems (RDBMS's) are still widely used and will not be deprecated in the future as the amount of investments on them is growing over the years. More specifically, RDBMS's have been around for over half-century [7] and proven to provide consistent performance, stability, and concurrency control. RDBMS's are currently the backbone of the IT industry and have been evolving over the past few decades for better performance.

On the other hand, using dedicated graph databases for graph processing is presumed to provide better performance and scalability over relational databases (c.f. [3]), however, graph databases still have a long way to reach the level of maturity of RDMBS's. From this prospective, using an RDBMS to implement graph algorithms seems logical and in fact more efficient. However computing graph algorithms using SQL queries is challenging and requires novel thinking. As such, there is active research on the use of novel methods to compute graph analytics on RDBMS (c.f. [1, 2, 9, 11]). These works have shown that RDBMS's often provide higher efficiency over graph databases for specific analytics task.

This paper presents a PageRank algorithm implementation using RDBMS with table partitioning and compares it with the implementation provided by a dedicated Graph Database.

The rest of this paper is organized as follows. A brief background review of the PageRank algorithm is given in Section 2. Our PageRank implementation using RDBMS with table partitioning is given in Section 3. Section 4 shows the results of the experiments. Section 5 concludes the paper.

2 Preliminaries

We denote by $G = (V, E)$ a graph with V as a set of vertices, and E as the set of edges. For each vertex v there will be a non-negative initial PageRank value $PR(v)$ and for each edge $e = (u, v)$ there is a weight of $1/n$ assigned to it, where n is the number of outgoing links from u.

We can use the following relational tables to store a graph. The TE table contains $\forall e = (u, v) \in E$ along with their weights, where u is denoted by *fid*, and v by *tid* and $w(u, v)$ by *cost*. We can construct a unique index on (fid,tid).

The TV table contains $\forall u \in V$ in the graph, denoted by *id* along with its rank $PR(u)$, denoted by *pagerank*. We can also construct a unique index on *id*.

The PageRank algorithm assigns a weight value to each page in the web or vertex in a graph; the higher the weight of a page or vertex, the more important it is. Web pages are represented as a directed graph where pages are vertices and links are edges. Below is an example of how we calculate PageRank for a small graph.

The graph in Figure 1 has four vertices representing four web pages. Page 1 has links to each of the other three pages; page 2 has links to 1 and 3 only; page 0 has a link only to 1, and page 3 has links to 2 and 0 only. Let us assume a

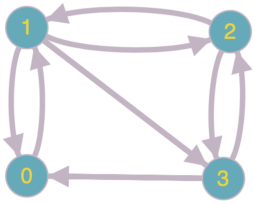

Fig. 1: Simple Directed Graph

random user is visiting page 1; this user will have a probability of 1/3 for each link (0,2,3) to follow and visit a next page. If the user is visiting page 0, then he will have a probability of 1 to visit page 1 as this is the only link available. If we follow the same logic, we will have a probability of 1/2 for each link on page 2 and page 3. The probability value for each link is the weight for each link, and based on this, we could build an adjacency matrix for the graph as a square matrix M with a number n of columns and rows. The PageRank algorithm proceeds in the following steps.

- Set an initial PageRank value for each page
- Repeat until convergence: compute PageRank using Equation 1

$$PR(A) = \sum_{i=1}^{n} \frac{PR(i)}{C(i)} \tag{1}$$

where $PR(A)$ is the PageRank value of vertex A, $PR(i)$ is the PageRank value of vertex i, and $C(i)$ is the number of outbound links (edges) of vertex i. Vertices i for $i \in [1, n]$ are all the vertices of the graph that contain links pointing to A. Usually, there is also a damping factor present in the computation of PageRank values but we ignore it in this paper for simplicity and because all techniques we present can be extended easily to that case.

The link probabilities ($1/C(i)$), as described above, could be represented as a matrix M. For the graph in Figure 1, the matrix will be as follows.

$$M = \begin{bmatrix} 0 & 1/3 & 0 & 1/2 \\ 1 & 0 & 1/2 & 0 \\ 0 & 1/3 & 0 & 1/2 \\ 0 & 1/3 & 1/2 & 0 \end{bmatrix}$$

Regarding the $PR(i)$ values, we can represent them all by a PageRank vector V. Then the computation given by Equation 1 can be written as $M \cdot V$, which captures the computation of PR values for all the vertices of the graph at the same time. We denote by V_t the version of V at iteration t. Then, PageRank is iteratively computed using equation 2, by multiplying matrix M and vector V_t and repeating until convergence.

$$V_{t+1} = M \cdot V_t \tag{2}$$

where V_{t+1} is the new vector holding the newly computed PageRanks for all the vertices. In each iteration, the newly computed PageRank values will get closer to the final PageRank values. We stop when PageRank values do not change much.

Observe that the PageRank value of a vertex A is dependent on the value of PageRank of vertices pointing to it. However, we do not know the Pagerank value of inbound vertices till we calculate the ones pointing to them and we will not know the Pagerank values of them till we calculate the PageRank values of vertices pointing to them too and this keeps on. So, to overcome this starting problem, we initially set an estimated PageRank value for each vertex. This can be represented as a vector

$$V_0 = [1/n, 1/n, \cdots, 1/n]$$

where n is the number of vertices in the graph.

3 PageRank in RDBMS

Representing the graph in a square matrix, M requires quadratic size. Computing Pagerank in its matrix representation requires the matrix to be fully loaded in memory; however, loading the graph into memory might not be possible for large graphs like the Google web or Facebook. However, since the matrix is very sparse, all the implementations exploit sparsness and do not materialize the matrix as is. Instead only the non-zero entries are stored in the format (i, j, m_{ij}).

Using RDBMS is quite efficient in this regard. First, matrix M could be saved as tuples (i, j, m_{ij}) of only connected vertices. Second, when computing PageRank for a vertex A, the edges that need to be considered are only those pointing to vertex A. This is a tiny subset of the matrix.

Figure 2 shows the SQL statement used to compute Pagerank using equation 2, where TE stores graph edges and TV stores vertices's Pagerank estimates. If we run this SQL query, it will produce the result of multiplying the matrix with the vector V_i. The multiplication is very efficient as we only do the calculation for existing edges in the matrix.

```
SELECT a.tid, SUM(a.rank*b.pgrank)
    From TE a, TV b
    WHERE a.fid=b.node
    GROUP BY a.tid;
```

Fig. 2: Compute Pagerank For one Iteration

Figure 3 shows the full SQL statement using the new Merge SQL [8] operation, which is very efficient in saving SQL results. This way, we save the new Pagerank estimate so that it can be used in the next iteration. The query will do a full table scan or index scan based on the table setup. RDBMS will need

to load parts of the table into memory to compute Pagerank. This process is acceptable when the loaded parts could be loaded into memory but cumbersome when graph size is hugely larger than available memory, which inevitably will lead to use data swap and, as a result, diminish the performance dramatically. In the following section, we solve the graph size problem by using table partitioning based on partitioning the matrix M and vector V_i into parts that can be loaded into memory.

```
MERGE  INTO TV as Target
        USING
         (SELECT a.tid, SUM(a.rank*b.pgrank)
                FROM TE a, TV b
                WHERE a.fid=b.node
                GROUP BY a.tid )
        AS  Source(node,pgrank)
        ON(target.node=source.node)
        WHEN MATCHED THEN
                UPDATE SET pgrank=source.pgrank
        WHEN NOT MATCHED THEN
                INSERET(node,pgrank)
```

Fig. 3: Compute Pagerank and update vector V

3.1 Table Partitioning

To overcome the matrix size problem, we partitioned both the matrix and the vector into k parts and saved each part in a separate table, TV_i, and TE_i, where $i \in [1, k]$. We divide the matrix into stripes of almost equal size, and we create vectors to have only the vertices that are needed to compute Pagerank for each matrix stripe.

Figure 4 shows how the matrix and vector are partitioned. Each matrix stripe will have a full set of inbound edges for a set of vertices and matched with a vector containing all the *fid*'s that exist in the partitioned matrix. This way, we will be able to compute Pagerank for the set of vertices of interest. A similar matrix partitioning scheme is also described in the Map-Reduce chapter of [10].

The main goal is to create as many stripes as needed so that the portions of the matrix in one partition can fit conveniently into memory. We used the SQL statements in Figure 5 to build the partitioned tables based on matrix partitions. Each TE_i table will have a subset of vertices along with all inbound edges, and each table TV_i will have all *fid*'s that exist in TE_i.

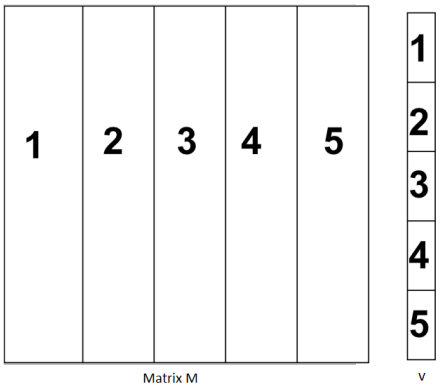

Fig. 4: Matrix and vector partitioning into k stripes.

```
--Create TE_i table
INSERT into TE_i (fid,tid,rank)
SELET fid, tid, rank FROM TE
where tid >=<val> AND tid < <val2>
--Create TV_i table
INSERT INTO TV_i(node,pgrank)
SELET DISTINCT  fid node, d FROM TE_i
```

Fig. 5: Creating partition tables TV_i and TE_i for $i \in [1, k]$.

4 Experimental Results

4.1 Setup Configurations

We executed the experiments on a consumer-grade server with Intel Core i7-2600 CPU @3.4 GHz 64 bit Processor, 12 G of RAM and running Windows 7 Home Premium, using Java JDK SE 1.8.

As RDBMS's we used the latest versions of a commercial database (which we anonymously call CD) and an open-source database (which we anonymously call OD). As graph database, we used the latest version of a graph database (which we anonymously call GD). We refrain from using the real names of these databases for obvious reasons.

We used four real datasets from Stanford's Data collection and a one-billion-edge graph from The Laboratory for Web Algorithmics. By default, we used three table partitions in the case of table partitioning experiments except stated otherwise. All the results shown are based on computing one Pagerank iteration. The real datasets are Web-Google, Pokec, Live-Journal and Orkut (from http://snap.stanford.edu), and UK 2005 (from http://law.di.unimi.it/webdata). Table 1 shows statistics about the datasets used.

4.2 Results

We observed that in all the datasets we used, OD and CD clearly out-perform GD significantly. Figure 6 shows how GD performs poorly with large datasets,

Table 1: Graph Datasets

Data Set	Nodes#	Edges#
Web-Google	875,713	5,105,039
Pokec	1,632,803	30,622,564
Live Journal	4,847,571	68,993,773
Orkut	3,072,441	117,185,083
UK 2005	39,459,921	936,364,282
IT 2004	41,291,594	1,150,725,436

such as Live Journal (LJ) or Orkut. Orkut was the largest data set that GD could manage to process without crashing out.

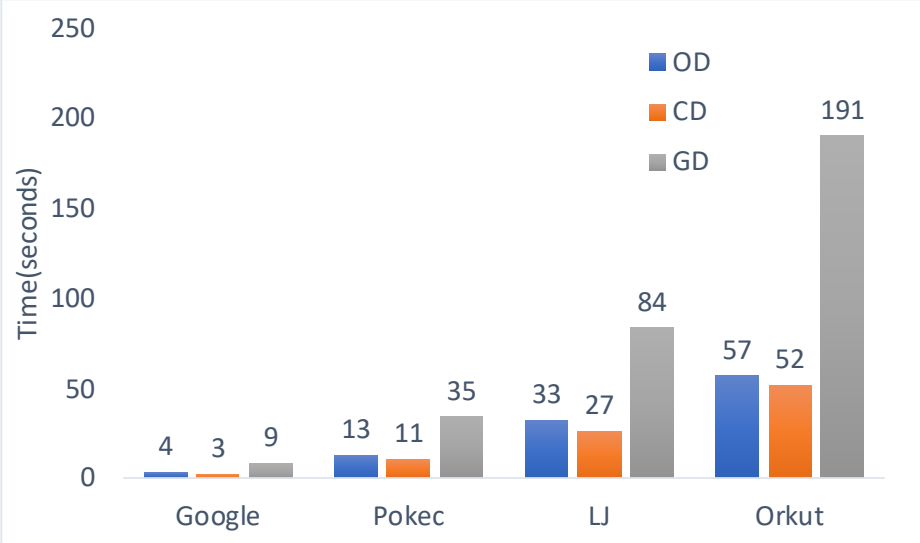

Fig. 6: Results of running PageRank using GD, CD, and OD.

Using table partitioning gives significant enhancement in managing memory load which in turn boosts PageRank processing time especially with large data sets such as LJ, Orkut and the large graph UK-2005. Figure 7 shows big performance differences between the GD processing time and both RDBMS approaches using table scan and table partitioning. The impact of table partitioning starts to appear once the datasets become larger, as shown in the chart. Table partitioning significantly improved over the approach of table scan especially for LJ and Orkut.

In our experiments we also wanted to decouple the processing time of computing PageRank from the time to save the results, hence we ran two separate experiments; one with saving the outcome and the other without saving the out-

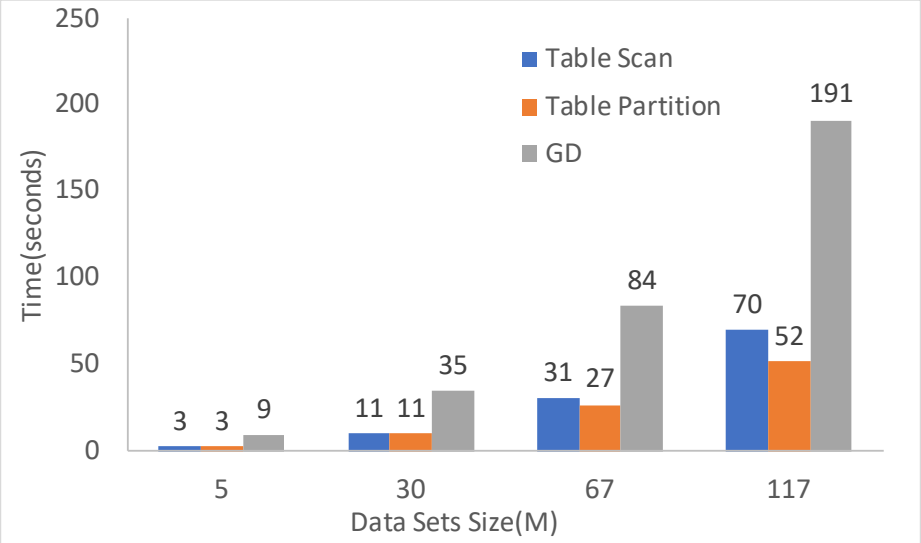

Fig. 7: Results of PageRank in RDBMS CD using Table Scan, Table Partitioning and GD. We show here only the dataset sizes as opposed to their names. The names are as in Figure 6.

come. Figure 8 compares the results of the experiments. We noticed that CD did a better job than OD in both operations and the time taken for saving data was noticeably shorter. We relate this to the Merge operation which exists in CD but does not in OD. The Merge operation showed to have superior performance over regular insert/update operations.

In addition to the above, OD performs poorly in computing PageRank using a non-clustered index scan. Figure 9 shows a big jump in time when we used non-clustered index scan in large data sets, in contrast to a clustered index scan or table scan. We relate this to the OD optimizer not being good enough in planning and executing the queries. Also the I/O cost was high which indicates the data retrieval process included high random access. Such random access was reduced significantly when the table was reordered as part of building the clustered index, hence the processing time was also reduced significantly.

Figure 10 shows the processing time in the case of using table scan and clustered index scan. Using an index did not help that much in reducing processing time and the results were very comparable to just table scan and differences were not noticeable. We relate this to the fact that the query used to calculate PageRank requires a full table retrieval hence using an index will not make that big of a difference.

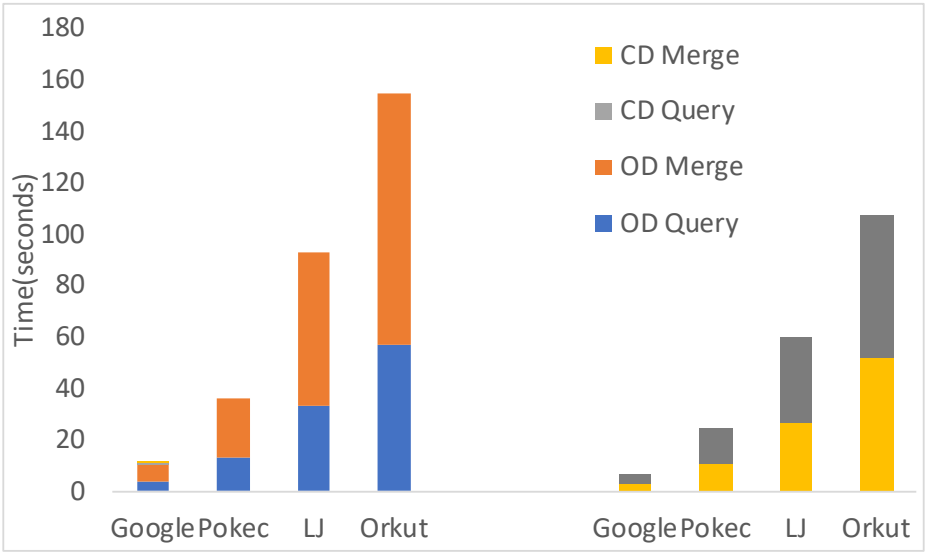

Fig. 8: Show the difference between the time taken to only calculate PageRank without saving the results and the time taken to do the same with saving the results.

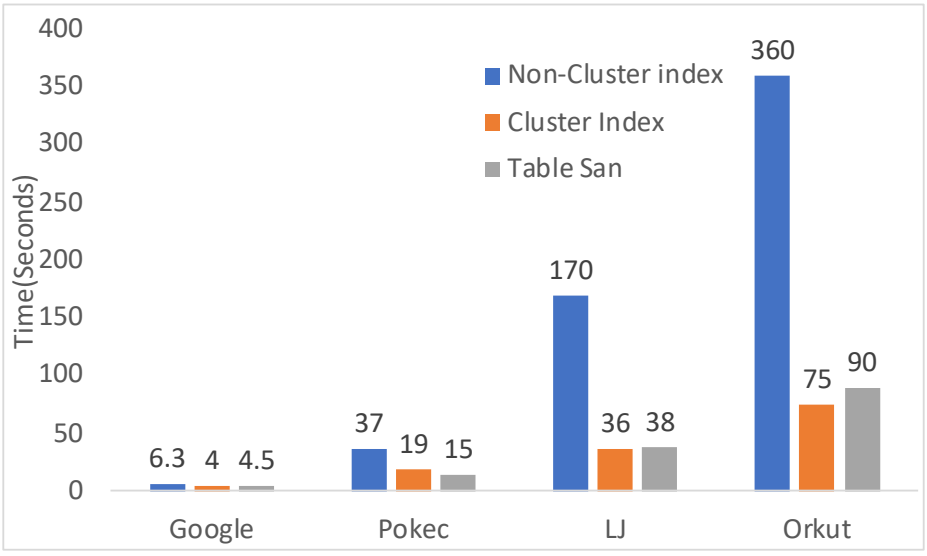

Fig. 9: OD performs poorly in the case of non-clustered index vs table scan or clustered index.

4.3 Experiments on Billion-Scale Networks

Here we show our experiments on two very large datasets, namely UK-2005 and IT-2004, the latter with more than a billion edges. They represent the web net-

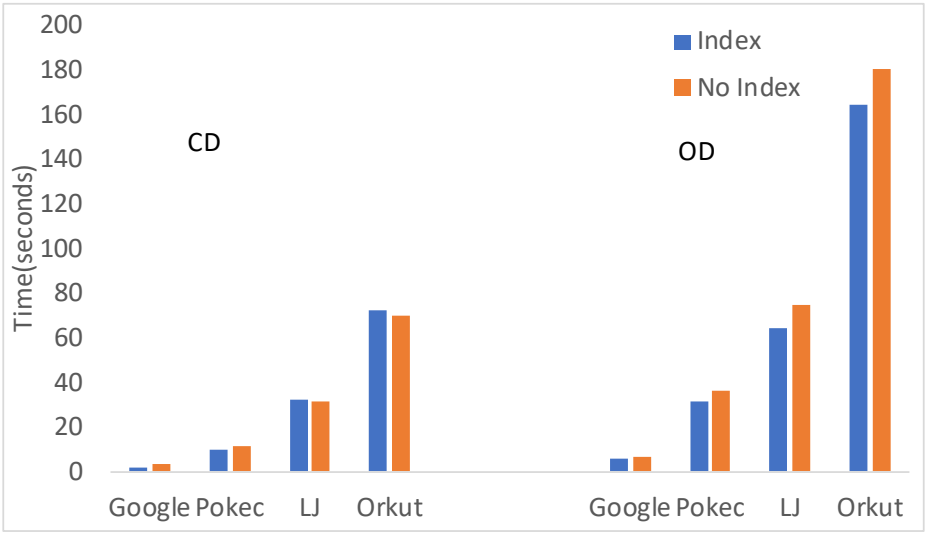

Fig. 10: Results of using table scan vs index scan in OD and CD

work of UK and Italy in 2005 and 2004. The precise number of nodes and edges is given in Table 2. We ran the PageRank algorithm using table partitioning. Both data sets were partitioned into 12 partitions and we sum up all the processing time to compute PageRank for each partition.

Figure 11 shows the runtime for each of the datasets. We used CD as it showed superiority over OD in I/O and memory management. GD could not be a part of the experiment as it failed to process any graph bigger than Orkut in our test environment setup. As Figure 11 shows, even with over one billion graph size, we managed to get a good processing time. More specifically, for IT-2004, we were able to complete the computation of PageRank for all the partitions in about 10 min (600 sec).

Table 2: Billion-size Datasets

Data Set	**Nodes#**	**Edges#**
IT 2004	41,291,594	1,150,725,436
UK 2005	39,459,921	936,364,282

5 Conclusion

We presented the implementation of the PageRank algorithm over RDBMS using different options, such as table-scan, non-clustered-index, clustered-index, and

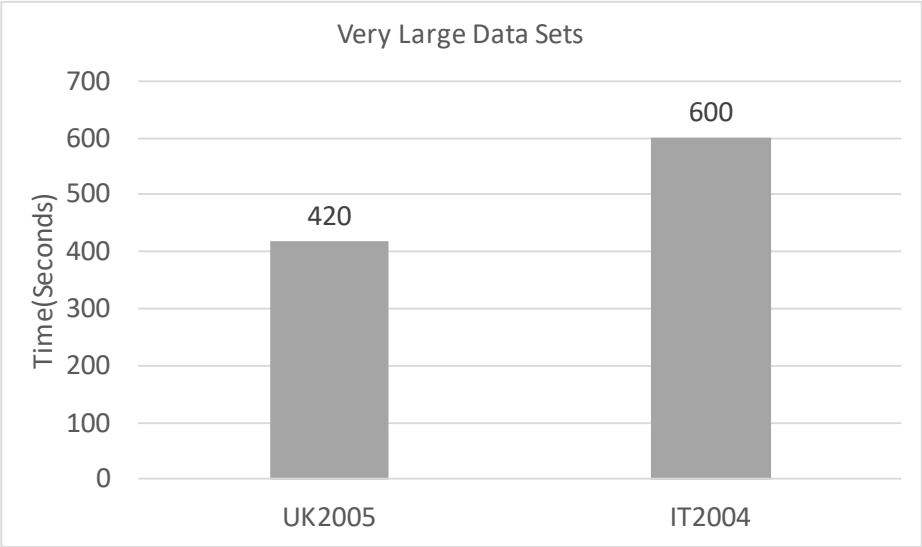

Fig. 11: Results of calculating PageRank on very large data sets, IT 2004: 1.15 billion edge graph and UK 2005:0.93 billion edge

table-partitioning. We showed that RDBMS's could perform better than GD and could process very big datasets in a consumer-grade server.

The experiments showed that the OD optimizer was not good enough for our task. For instance, it was not able to determine that using a non-clustered index is not a good choice for our queries. The OD clustered index behaved better but still there was no improvement compared to simple table-scan without any indexing at all. The CD commercial query optimizer is more intelligent than its open-source counterpart.

We observed that manually partitioning tables gives a significant improvement in the execution time of our queries. This tells us that RDBMS optimizers of today, even after many decades of development, still can be improved further in order to handle heavy analytical queries such as those computing PageRank.

CD did not consume significant processing time in order to merge the data but OD in large datasets consumed significant processing time, in some cases double the query time. We clearly observed that both RDBMS's we use, without using any indexing or partitioning still dramatically outperform graph database GD. This comes as a surprise because the latter was designed for handling graphs from the ground up. Therefore we conclude that specialized graph databases still have a lot of ground to cover in order to be good competitors to RDBMS engines for large datasets.

References

1. R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE transactions on knowledge and data engineering*, 5(6):914–925, 1993.
2. A. Ahmed and A. Thomo. Computing source-to-target shortest paths for complex networks in rdbms. *Journal of Computer and System Sciences*, 89:114–129, 2017.
3. R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.
4. K. Bharat and G. A. Mihaila. When experts agree: using non-affiliated experts to rank popular topics. In *Proceedings of the 10th international conference on World Wide Web*, pages 597–602, 2001.
5. S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. 1998.
6. S. Chakrabarti, B. E. Dom, S. R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins, D. Gibson, and J. Kleinberg. Mining the web's link structure. *Computer*, 32(8):60–67, 1999.
7. E. F. Codd. A relational model of data for large shared data banks. In *Software pioneers*, pages 263–294. Springer, 2002.
8. A. Eisenberg, J. Melton, K. Kulkarni, J.-E. Michels, and F. Zemke. Sql: 2003 has been published. *ACM SIGMoD Record*, 33(1):119–126, 2004.
9. J. Gao, J. Zhou, J. X. Yu, and T. Wang. Shortest path computing in relational dbmss. *IEEE Trans. Knowl. Data Eng.*, 26(4):997–1011, 2014.
10. J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive data sets*. Cambridge university press, 2020.
11. C. Ordonez and E. Omiecinski. Efficient disk-based k-means clustering for relational databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(8):909–921, 2004.
12. L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.
13. M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.
14. P. Zikopoulos, C. Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.