Computing Source-to-Target Shortest Paths for Complex Networks in RDBMS

Aly Ahmed, Alex Thomo
University of Victoria, BC, Canada
{alyahmed,thomo}@uvic.ca

ABSTRACT

How do we deal with the exponential growth of complex networks? Are existing algorithms introduced decades ago able to work on big network graphs? In this work, we focus on computing shortest paths (SP) from a source to a target in large network graphs. Main memory algorithms require the graph to fit in memory and they falter when this requirement is not met. We explore SQL-based solutions using a Relational Database Management System (RDBMS). Our approach leverages the intelligent scheduling that a RDBMS performs when executing set-at-a-time expansions of graph vertices, which is in contrast to vertex-at-a-time expansions in classical SP algorithms. Our algorithms perform orders of magnitude faster than baselines and even faster than main memory algorithms for large graphs. Also, we show that our algorithms on RDBMS outperform counterparts running on modern native graph databases, such as Neo4j.

1. INTRODUCTION

Large graphs are everywhere nowadays. They model social and web networks, knowledge networks, and product co-purchase networks, to name a few. We label these networks "complex" to distinguish them from "spatial" networks, such as road networks that have been studied extensively.Our focus in this paper is on complex networks.

Many classical graph algorithms face challenges when the graph is large. This is because they need random access to the vertices of the graph and their adjacency lists, and random access is expensive. While this is significantly more pronounced for data residing in external storage, it is also true for data that can fit in main memory (see [22] for discussions and experiments). For complex networks the situation is even more challenging because many optimization ideas for spatial networks are not applicable to complex networks (see [37] for a survey on shortest path approaches for different kinds of networks).

One family of graph algorithms that we identify as particularly demanding for random-access is graph search. Graph search algorithms seek subgraphs that satisfy some property, such as the shortest paths between a source and destination [7], the minimum spanning tree rooted at a vertex [30], the connected component containing a vertex [19], and so on. Most algorithms of this family have an expand-and-explore nature that exhibits an intensive random access pattern. Therefore, they are good candidates for re-engineering so that random access is reduced.

In this paper, we focus on *source-to-target* (s-t) shortest path queries (or simply s-t queries), also known as point-to-point queries in literature. These queries are central in social network analysis. For instance, graph distance (often referred to as social distance) can play an important role in deriving insights on user search in Linkedin, Facebook, and other social networks (see [20, 21, 38] for examples of using social distance). S-t queries have also been used to leverage trust in social links in online marketplaces [41] and shown to be an integral part of location and social aware search [36, 44].

S-t queries have a "local search" nature that is in contrast to the source-to-all queries which have a "global search" nature. As such, s-t queries are not a good fit for Pregel-like systems (such as Graphchi in [26]) which access the whole graph in each pass.

The approach we follow for computing s-t queries is to use relational databases as pioneered by [9]. Relational databases are a mature technology representing more than 40 years of active development. What relational databases offer is a set-at-a-time mode of operation, which allows data-access scheduling for grouping requests to disk blocks and thus reducing random access.

However, the main algorithm for finding shortest paths, the Dijkstra's algorithm, follows a vertex-at-a-time approach; it seeks to expand only the best vertex (path) discovered so far. On the other hand, other search algorithms, such as breadth first search (BFS), expand a set of vertices (paths) in each iteration, thus making possible to use the set-at-a-time mode of operation that a relational database offers. One can use BFS for s-t queries, however, the discovered paths might not be the best (shortest), and we need to re-expand vertices many times until we find the shortest paths.

The authors of [9] propose Bidirectional Restrictive BFS (B-R-BFS) which is an adaptation of BFS to reduce the number of vertex re-expansions. This is achieved by partitioning the table of graph edges into multiple tables based on the weights of edges. The algorithm is also bidirectional, meaning that it runs both from the source and the target until the two searches meet. The performance improvements over the Dijkstra's algorithm and pure BFS are impressive. However, deciding termination in B-R-BFS is challenging, and the condition proposed in [9] for checking termination is unfortunately not complete.

Our contributions in this paper are as follows.

First, we show the problem with termination in B-R-BFS and then propose a new termination algorithm. In general, in any bidirectional s-t algorithm that starts two search processes, one forward from s and the other backward from t, we need to determine whether both processes have *finalized* the distance of a vertex v from s and t, respectively. In such a case, we can successfully terminate the algorithm. In B-R-BFS, deciding whether a vertex v has its distance finalized is not easy as v can be expanded multiple times using different edge tables. The solution we propose is based on determining lower bounds for vertex distances and inferring a termination condition based on these bounds.

Second, we propose another algorithm, Bidirectional Level-based Frontier BFS (B-LF-BFS), for computing s-t queries in a set-at-a-time fashion. Differently from B-R-BFS, we achieve restrictive BFS not by splitting the edge table, but by selecting only a part of the visited vertices as a frontier to be expanded. The frontier contains only those vertices that have a distance estimate less than a "level" value. We show that, if the frontier is iteratively expanded until no more expansion is possible, then the distances of the vertices expanded during the current level are final. We, then, increase the current level to the next one (by adding a step value) and repeat the process. Since we have an explicit way to determine when vertices are finalized, we obtain a much simplified termination procedure.

Third, we enhance B-LF-BFS to use a graph representation where the neighbors of each vertex and their respective edge costs are compressed in an inverted-index style. We call this enhanced algorithm B-LF-BFS-C. We borrow ideas from Information Retrieval practice to perform compression by encoding the differences in neighbor ids using variable-byte encoding. The compression achieved is such that B-LF-BFS-C is able to handle graphs of an order of magnitude bigger than what B-R-BFS and B-LF-BFS can.

Finally, we present a detailed experimental study on real and synthetic datasets. We observe that all the above three algorithms outperform the vertex-at-a-time Dijkstra's algorithm in RDBMS by orders of magnitude. This strongly affirms the benefit of the set-at-a-time mode of operation offered by RDBMSs. Furthermore, we show that B-LF-BFS-C outperforms even a memory implementation of the Dijkstra's algorithm for a relatively large graph (Live Journal). We also show that B-LF-BFS-C can easily handle very large graphs, such as UK 2005, with close to one billion edges.

The rest of the paper is organized as follows. In Section 2, we discuss other systems for graph management and processing. In Section 3, we present preliminaries and explain the challenges of computing shortest path queries on RDBMS. In Section 4, we describe the problem with termination detection in B-R-BFS and the proposed solution to fix it. In sections 5 and 6, we present the B-LF-BFS and B-LF-BFS-C, respectively. In Section 7, we present our experimental results. In Section 8, we describe the related works. Finally, Section 9 concludes the paper.

2. OTHER SYSTEMS FOR GRAPH MANAGEMENT AND PROCESSING

Systems that allow for the storage and random access of big graphs are (native) graph databases. One of the main graph databases is Neo4j[1]. To make random access feasible, Neo4j builds indexes to quickly zoom in to a vertex and its neighborhood. A good index makes random access fast. However, if there are massive requests for random access during the execution of an algorithm, the performance will still suffer. As we show in our experiments, Neo4j is considerably slower than the proposed algorithms on RDBMS; it even fails to return results for our larger graphs.

A very different approach is followed by the systems geared towards graph analytics. As representatives of such systems, we mention Pregel [27] for a distributed setting and GraphChi [26] for a single machine. They do not offer random access to a graph. Instead, they present a vertex-centric (VC) computation paradigm [27] where each vertex independently runs the same algorithm and sends and receives messages to and from its neighbors. VC systems for a single machine, such as GraphChi, significantly reduce random access to only a negligible amount. The tradeoff is multiple sequential passes over the graph. VC computation is quite good for some problems. Global graph search can be nicely implemented as a VC computation (see [26] for a discussion). For instance, finding the shortest paths from a source vertex to *all* the other vertices of a graph or finding *all* the connected components of a graph can be efficiently done as VC computations [26]. However, a VC computation is not a good fit for more local graph search, such as finding s-t shortest paths. The latency is too high as the whole graph will be accessed.

Our goal in this paper is to provide algorithms for s-t shortest paths with a latency in the order of a few seconds (on a consumer-grade machine).

3. PRELIMINARIES

We denote a directed, edge-weighted graph by $G = (V, E, C)$, where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and $C : E \to \{x \in \mathbb{R} : x > 0\}$ is the edge-weight (or cost) function.

Let $p = [(u_0, u_1), \ldots, (u_{k-1}, u_k)]$, where $(u_{i-1}, u_i) \in E$ for $i \in [1, k]$, be a path from $u_0 \in V$ to $u_k \in V$. We denote by $c_p = \sum_{i=1}^{k} C(u_{i-1}, u_i)$ the *length* (or *cost*) of p.

Given two vertices s and t, we denote by $d(s, t)$ the length of the shortest path from s to t.

In this paper, we are interested in source-to-target (s-t) queries which specify a vertex pair s, t and ask for the shortest path from s to t.

3.1 Graphs and Shortest Paths in RDBMS

We store the edges of a graph in a RDBMS in a table TE with three columns, *fid*, *tid*, and *cost*, for the source vertex id, target vertex id, and weight (cost) of an edge, respectively. We also construct indexes on *fid* and *tid*. For the ease of exposition, we will blur the distinction between a vertex and its id.

[1]http://neo4j.com

SP algorithms start out from the source vertex s and *expand* it by *reaching its neighbors*. One or more of the neighbors are expanded in turn, and we continue like this until we reach the target vertex t.

To accommodate expansions, we need a table, called TA, which stores the set of vertices that have been visited so far. Table TA has four columns: (1) *nid* for the id of a vertex we have visited, (2) $d2s$ for the length of the best path we have discovered so far from s to nid, (3) $p2s$ for the id of of the vertex coming before nid in this path, and (4) f for flagging nid as finalized (the best path from s to nid has been discovered) or not. When a node u is finalized it means that the shortest path from source node s to u has been determined and the discovered distance estimate will not change to a lower value in a later iteration. The main goal of the shortest path problem is to finalize target node t. The computational challenge is to finalize t as quickly as possible.

Table TA is typically much smaller than TE, usually by one or two orders of magnitude. We do not create an index for TA as it is frequently updated.

In each iteration, we select from TA a set F of vertices for expansion. Vertex expansion is computed by joining F with TE. The newly visited vertices are merged into TA. Depending on the algorithm, a vertex can be visited multiple times. Each visit can (possibly) cause an update or insert into TA. We handle both cases using the *MERGE* operator in SQL.

Initially, $(s, 0, s, 0)$ is inserted into TA. At the end of an SP algorithm, we should have $(t, d(s,t), u_k, 1)$ in TA. Upon termination of the algorithm, we output the shortest path by following backwards the chain of tuples $(t, d(s,t), u_k, 1)$, $\ldots$, $(u_1, d(s,u_1), s, 1)$ in TA, where s, u_1, $\ldots$, u_k, t are the vertices along this path.

In order to speed up the computation, we can do bidirectional search and expansions. We start simultaneously from s in the forward direction and from t in the backward direction and discover paths that eventually meet at some intermediate vertex. We need two TA tables for bidirectional search, TA^f and TA^b. Also we refer to the F sets in the forward and backward directions as F^f and F^b, respectively.

Dijkstra's Algorithm on RDBMS Dijkstra's algorithm only expands one vertex at a time; the one with the smallest $d2s$ value. The expansion joins are fast individually as each one only involves one tuple from TA^f (or TA^b) that needs to be joined with TE. Unfortunately, these joins are too many and the overall latency is high.

Termination. For the bidirectional Dijkstra's algorithm, the termination condition is when the forward search finalizes a vertex that has also been finalized by the backward search (or vice-versa). A vertex is finalized in the forward (backward) direction when it is selected to be in F^f (F^b).

3.2 Set-at-a-time Evaluation

One of the strengths of an RDBMS is its set-at-a-time evaluation mode. In graph search, set-at-a-time is more efficient than vertex-at-a-time because it allows the database to perform intelligent scheduling of buffer content and disk blocks; access requests to the same block can be bundled and scheduled at the same time, thus allowing for a better query evaluation plan.

Consider the join of F with TE. In the Dijkstra's algorithm, F has only one vertex. As such, the database needs to retrieve the edges of only one vertex for each join. In the worst case there can be n such join queries. Clearly, a vertex-at-a-time mode of operation is quite inefficient in this case; there will be unnecessary I/Os for retrieving the edges of different vertices when they can happen to be in the same block. In contrast, in a set-at-a-time mode, a block can be read once and serve many vertex expansions.

On the other hand, there is significant overhead if the set-at-a-time strategy is taken to the limit, which, in our case, translates to pure breadth-first-search (BFS). In BFS, all newly visited vertices are selected to be in F, and the expansion of all these vertices is achieved with a single join operation. However, BFS may expand the same vertices multiple times, thus incurring significant overhead in the number of expansions compared to Dijkstra's algorithm. Therefore, we need to strike a balance between pure BFS and Dijkstra's algorithm.

In [9], the strategy proposed is a restrictive BFS. Similar to BFS, multiple vertices are selected to be in F. However, in each iteration, only a subset of edges is allowed to be used for expansion. More specifically, vertices are expanded first using the lightest edges, then using more heavier edges, and so on.

However, when performing bidirectional search under a BFS-like strategy, deciding termination becomes complicated. This is because when the two searches meet at some vertex v, we do not know whether v is finalized or not. Therefore, there is no guarantee that the path discovered is the shortest. In contrast, in the Dijkstra's algorithm, we have an easy way to finalize a vertex; this happens when the vertex is selected to be in F^f (F^b). This is not true for a BFS-like strategy.

3.3 Bidirectional Restrictive BFS (B-R-BFS)

Bidirectional Restrictive BFS (B-R-BFS) operates in set-at-a-time mode and performs much better than the Dijkstra's algorithm, however, its termination decision is not complete. In this section, we give an overview of B-R-BFS. In the next section, we show the problem with its termination and then present a correct termination procedure.

Partitioning the edge table. B-R-BFS starts by partitioning the TE table based on the edge weights. Formally it is done as follows.

Let pts be the desired number of partitioned tables and $[w_{min}, w_{max}]$ be the range of edge weights. We denote by $[w_0, \ldots, w_{pts}]$ the edge-weight *partitioning vector*, where $w_0 = w_{min}$, $w_{pts} = w_{max} + \epsilon$,[2] and $w_i < w_{i+1}$ for $0 \leq i < pts - 1$. We create pts partition tables, TE_0, $\ldots$, TE_{pts-1}.[3] For each edge e in the graph, if $w_i \leq C(e) < w_{i+1}$, then e is put into partition table TE_i.

[2] ϵ represents a very small number.

[3] In [9], the partition tables are numbered from 1 to pts. We choose to number them from 0 to $pts - 1$ in order to simplify the exposition of results later.

High-level overview of the algorithm. Recall the TA^f and TA^b tables we use for the forward and backward search, respectively. These tables, instead of the f column, will now have a different last column, fwd for TA^f and bwd for TA^b. fwd and bwd store the number of iteration during which the tuple was inserted or updated in TA^f or TA^b, respectively.

During an iteration, a vertex will only be expanded using one partition table. Once a vertex is selected to be in F^f (or F^b), it remains there for pts iterations until it is expanded using each of the partition tables. Initially, in the forward expansion, table TA^f will have the source node s. In the first iteration, s is selected to be in F^f and subsequently expanded using TE_0. The neighbors of s reachable using TE_0 are added in TA^f. Let the set of these neighbors be N_s^0. In the second iteration, we have $F^f = \{s\} \cup N_s^0$, and try to expand s using TE_1 while the vertices in N_s^0 using TE_0. In general, consider a vertex v that enters F^f in iteration i (a vertex enters F^f in the next iteration after it is inserted or updated in TA^f). In iterations i, $i+1$, ..., $i+(pts-1)$, vertex v will be expanded using tables TE_0, TE_1, TE_{pts-1}, respectively. An analogous logic is followed for the backward direction as well. In other words, a vertex can have *delayed expansions* during the pts iterations it remains in F^f (F^b).

4. TERMINATION PROCEDURE FOR B-R-BFS

In this section we present a correct termination procedure for B-R-BFS.

Consider table TA^f (or table TA^b). We call a $d2s$ ($d2t$) value in table TA^f (TA^b) a *distance estimation* (DE). This is because it can (possibly) be lowered and become a real distance later on during the execution of the algorithm.

Definition 1. *Let v be a visited vertex in the forward direction and $(v, d2s_v, p2s_v, fwd_v)$ be its tuple in TA^f at the end of iteration i in the execution of B-R-BFS. DE $d2s_v$ is called* distance *if and only if it cannot change in some later iteration $i' > i$.*

An analogous definition can be stated for $d2t_u$ of a visited vertex u in the backward direction.

Given a tuple $(v, d2s_v, p2s_v, fwd_v)$ in TA^f or $(u,\ d2t_u,\ p2t_u,\ bwd_u)$ in TA^b, it is not easy to determine whether $d2s_v$ or $d2t_u$ are distances.

We define by m_i^f and m_j^b the minimum DE's discovered in iterations i and j in the forward and backward directions, respectively. Formally,

$$\begin{aligned} m_i^f &= \min\{d2s_v : (v, d2s_v, p2s_v, i) \in TA^f\} \\ m_j^b &= \min\{d2t_u : (u, d2t_u, p2t_u, j) \in TA^b\}. \end{aligned}$$

These values can be easily obtained by simple MIN queries on the TA^f and TA^b tables. We have $m_i^f = d2s_v$ and $m_j^b = d2t_u$ for some vertices v and u. We might be tempted to declare m_i^f and m_j^b to be distances. This is not always true however because $d2s_v$ and $d2t_u$ can be lowered in later iterations as result of delayed expansions.

For an example see Fig. 1 where we want to compute the shortest path from s to t. For simplicity, we are only considering the forward search. Suppose the partition vector is $[1, 5, 10+\epsilon]$. We have two partition tables, $TE_0 = \{(s,u,1), (s,v,4), (v,t,4)\}$ and $TE_1 = \{(u,t,6), (t,s,10)\}$.

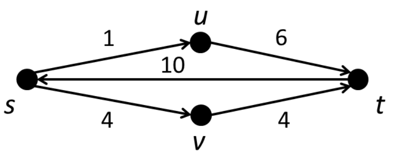


Figure 1: A graph illustrating the effect of delayed expansions.

Vertex s enters F^f in iteration 1, and stays there for iterations 1 and 2. Similarly, u enters F^f in iteration 2, and stays there for iterations 2 and 3. It is in iteration 3 that we find the shortest path from s to t via u. At the end of each iteration, F^f, TA^f and m_i^f are as follows.

Iteration 0: $F^f = \{\}$, $TA^f = \{(s,0,s,0)\}$, $m_0^f = 0$

Iteration 1: $F^f = \{(s,0,s,0)\}$,
$TA^f = \{(s,0,s,0), (u,1,s,1), (v,4,s,1)\}$, $m_1^f = 1$

Iteration 2: $F^f = \{(s,0,s,0), (u,1,s,1), (v,4,s,1)\}$,
$TA^f = \{(s,0,s,0), (u,1,s,1), (v,4,s,1), (t,8,v,2)\}$,
$m_2^f = 8$

Iteration 3: $F^f = \{(u,1,s,1), (v,4,s,1), (t,8,v,2)\}$,
$TA^f = \{(s,0,s,0), (u,1,s,1), (v,4,s,1), (t,7,v,3)\}$,
$m_3^f = 7$

As we can see, $m_2^f = 8$, corresponding to $d2s_t$, is not a distance because $d2s_t$ becomes 7 in iteration 3. Tuple $(u,1,s,1)$ is inserted into TA^f in iteration 1, and enters F^f in iteration 2. However, $(u,1,s,1)$ is not expanded using edge $(u,t,7)$ in iteration 2. This expansion is *delayed* to iteration 3.

As in [9], let l_i^f be *the maximal distance* finalized (discovered) from s after the i-th forward iteration. Similarly, let l_j^b be *the maximal distance* finalized (discovered) from t after the j-th backward iteration. In [9], l_i^f is proposed to be recursively computed as

$$l_i^f = \begin{cases} \min\{m_i^f, w_1\}, & \text{if } i = 1. \\ \min\{m_i^f, l_{i-1}^f + w_i - w_{i-1}\}, & i \geq 2. \end{cases}$$

and l_j^b is analogously computed replacing i by j and f by b.[4]

The above computations for l_i^f and l_j^b pose the following problem: what happens when i or j are larger than pts? In such a case, w_i or w_j are undefined and the above equalities do not work. This unfortunately is not addressed in [9].

In the following, we present another solution.It is inspired in part by an email communication we had with the authors of [9].

[4]In [9], l_i^f and l_i^b are used to terminate the algorithm if the following condition is met: minCost $\leq l_i^f + l_i^b$, where minCost is defined later in this section.

In fact, what we propose is computing *lower bounds* to l_i^f and l_j^b. We use ll_i^f and ll_j^b to denote the lower bounds for l_i^f and l_j^b and set them to be as follows.

First, $ll_0^f = l_0^f = m_0^f (= 0)$, $ll_0^b = l_0^b = m_0^b (= 0)$, and $ll_1^f = l_1^f = m_1^f$, $ll_1^b = l_1^b = m_1^b$.

Now, let $k = \min\{i-1, pts\}$ and $h = \min\{j-1, pts\}$. For $i, j \geq 2$, we set

$$\begin{aligned} ll_i^f &= min\{m_i^f, m_{i-1}^f + w_1, \ldots, m_{i-k}^f + w_k^f\} \quad (1) \\ ll_j^b &= min\{m_j^b, m_{j-1}^b + w_1, \ldots, m_{j-h}^b + w_h^b\}. \quad (2) \end{aligned}$$

Suppose, for instance that $i = 2, 3, 4, 5$ and $pts = 3$. Then, $k = 1, 2, 3, 3$, and for the forward direction, we have

$$\begin{aligned} ll_2^f &= min(m_2^f, m_1^f + w_1) \\ ll_3^f &= min(m_3^f, m_2^f + w_1, m_1^f + w_2) \\ ll_4^f &= min(m_4^f, m_3^f + w_1, m_2^f + w_2, m_1^f + w_3) \\ ll_5^f &= min(m_5^f, m_4^f + w_1, m_3^f + w_2, m_2^f + w_3). \end{aligned}$$

We show the following theorem.

Theorem 1. *$ll_i^f \leq l_i^f$ and $ll_j^b \leq l_j^b$.*

Proof. Consider ll_2^f. We have that m_2^f is a distance of a vertex v (from s) unless there exists a vertex u that has remained with $fwd = 1$ after the 2nd iteration. Vertex u did not have any luck to be expanded by edges in $[w_0, w_1)$, however, it may be expanded by edges in the next interval and possibly cause the DE of v to get lower. In this case, the distance of v should be at least $m_1^f + w_1 (< m_2^f)$.

Now consider ll_i^f. We have that m_i^f is a distance of a vertex v (from s) unless there exists a vertex u that has remained with $fwd \leq i-1$ after the i-th iteration. Suppose u has $fwd = i-1$. As such, u did not have any luck to join with edges in $[w_0, w_1)$, however again, it may be expanded by edges in the next interval and possibly cause the DE of v to get lower. In this case, the distance of v should be at least $m_{i-1}^f + w_1 (< m_i^f)$. Similarly, suppose u has $fwd = i-r$, for $r \in [1, k]$. As such, u did not have any luck to join with edges in $[w_0, w_1)$,..., $[w_{r-1}, w_r)$, however, it may be expanded by edges in the next interval and possibly cause the DE of v to get lower. In this case, the distance of v should be at least $m_{i-r}^f + w_r (< m_i^f)$. From all the above, $ll_i^f \leq l_i^f$.

An analogous argument can be made for the backward direction as well. □

Remark. We would like to emphasize here that m_i^f's (m_j^b's) are each time *recomputed*. For example, m_2^f is recomputed when computing ll_3^f. This is because the vertex achieving the old m_2^f might have been updated in the current iteration and can now have $fwd > 2$. Therefore another vertex with $fwd = 2$ will provide the new m_2^f. The new m_2^f is greater than the old one.

For the termination decision we define

$$\mathsf{minCost} = \begin{cases} \min\{d2s_v + d2t_v\}, & \text{if } TA^f \bowtie_{nid} TA^b \neq \emptyset \\ \infty, & \text{otherwise.} \end{cases} \quad (3)$$

Now, we give the following condition that we use in the termination procedure.

$$\mathsf{minCost} \quad \leq \quad ll_i^f + ll_j^b. \quad (4)$$

If the above condition 4 is true, then by Theorem 1, the following condition is true as well.

$$\mathsf{minCost} \quad \leq \quad l_i^f + l_j^b. \quad (5)$$

If the last condition is true, then we can safely terminate B-R-BFS (see [9]). Upon such termination, minCost will be the length of the shortest path from the source to the target vertex. In other words, even though we might not have computed yet l_i^f and l_j^b, we can infer that Condition 5 is satisfied based on Condition 4 using the lower bounds ll_i^f and ll_j^b.

With the modified termination condition provided in this section, each vertex in the shortest path p takes pts iterations to finalize, hence the algorithm is estimated to take $length(p) * pts$ iterations.

5. BIDIRECTIONAL LEVEL-BASED- FRONTIER BFS (B-LF-BFS)

Here we propose another set-at-a-time algorithm for computing shortest paths in RDBMS. Similarly to B-R-BFS, it works in a set-at-a-time fashion by expanding a set of vertices in each iteration. Differently from B-R-BFS, it achieves restrictive expansions not by splitting the edge table, but by selecting only a part of the visited vertices as a frontier to be expanded. The algorithm performs better in practice than B-R-BFS and it does not require partitioning the TE table into several tables as in B-R-BFS. This can be desirable as it does not need extra space for partition tables and code complexity is reduced.

The Bidirectional Level-based-Frontier BFS (B-LF-BFS) we propose uses the TE, TA^f and TA^b tables as defined before. We do not need to record the iteration number as in B-R-BFS, and we bring back the finalization flag f in TA^f and TA^b.

We define $F^f = F_i^f$, where F_i^f is the set of all the vertices in TA^f that are not finalized and their $d2s$ value is less or equal to L_i, where L_i is a *distance level*. We define F^b in an analogous way.

For the sake of explanation, let us assume we are working in the forward direction and that initially we set $L_1 = step$, where $step$ is a small constant. Now the algorithm will (a) expand all the vertices in F^f (i.e. unfinalized vertices in TA^f that have $d2s \leq L_1$), (b) merge all the new vertices into TA^f, then (c) iterate and do the same, until no more unfinalized vertices with $d2s \leq L_1$ exist in TA^f. At this point we say that level L_1 is *cleared*, set $L_2 = L_1 + step$, and repeat the above operations for L_2.

In general, once level L_i is cleared, we set $L_{i+1} = L_i + step$, and repeat the above operations for L_{i+1}. This continues until termination is achieved. The algorithm terminates when a vertex v is finalized by both the forward and backward directions, or if there are no more unfinalized vertices in TA^f or TA^b.

Now, we illustrate the algorithm with an example. For the sake of simplicity we will explain the forward expansion; the backward expansion is similar. Consider the graph in Figure 1 and assume $step = 3$. In order to calculate the shortest distance between s and t, we initialize $L_1 = step = 3$, minCost $= \infty$, and the TA^f table with the tuple $(s, s, 0, false)$. In the first iteration, the algorithm will expand the initial tuple in TA^f with $d2s = 0 \leq L_1$ and $f^f = false$. So, node s will be expanded and nodes u with $d2s$=1 and v with $d2s$=4 will be merged into TA^f and node s will be flagged as processed (its flag f^f becomes $true$). The algorithm will iterate to check if there are any tuples in TA^f with $d2s \leq L_1$ and $f^f = false$. Node u with $d2s = 1$ will be expanded and as a result node t with $d2s = 6$ will be merged into TA^f. At this point, no more nodes with $d2s \leq L_1$ are left to be processed, hence the algorithm declares that L_1 is cleared and sets $L_2 = L_1 + step = 6$. As node t is merged, the algorithm sets minCost $= 6$. Next, the algorithm will expand the nodes that have not been processed yet and have $d2s \leq L_2$. As a result, node v with $d2s = 4$ will be expanded resulting in rediscovering node t with higher cost $d2s = 8$, hence no tuple will be merged in TA^f. As no more nodes are left to process, level L_2 is cleared. At this point, node t will be flagged as finalized.

```
MERGE INTO TAf
USING (
  WITH

  --Compute frontier
  F(nid,p2s,d2s) AS (
    SELECT nid, p2s, d2s
    FROM TAf
    WHERE f_f='0' AND d2s <= Li
  ),

  --Join frontier with the edge table
  F_TE(nid,p2s,d2s) AS (
    SELECT TE.tid AS nid, TE.fid AS p2s, F.d2s+TE.cost AS d2s
    FROM F JOIN TE ON F.nid=TE.fid
  ),

  --For each nid, select the tuple with the smallest d2s value
  SELECT nid, p2s, d2s
  FROM (SELECT nid, p2s, d2s,
                 row_number()  OVER
                   (PARTITION BY nid ORDER BY d2s) AS rn
             FROM F_TE)
  WHERE rn = 1

  ) source
ON (TAf.nid=source.nid)
WHEN MATCHED THEN
   UPDATE SET d2s=source.d2s, p2s=source.p2s, f_f='0'
   WHERE source.d2s<TAf.d2s
WHEN NOT MATCHED THEN
   INSERT  (nid,d2s,p2s,f_f)
   VALUES (source.nid, source.d2s, source.p2s, '0');

--Node finalization
UPDATE TAf SET f_f='1' WHERE f_f='0' AND d2s <= Li;
```

Figure 2: SQL statements for the B-LF-BFS algorithm.

More formally, we give the following definitions for B-LF-BFS.

Definition 2. *The F^f and F^b sets in levels L_i and L_j are*

$$F_i^f = \{(nid, d2s, p2s, f^f) \in TA^f : f^f = 0 \text{ and } d2s \leq L_i\}$$
$$F_j^b = \{(nid, d2t, p2t, f^b) \in TA^b : f^b = 0 \text{ and } d2t \leq L_j\}.$$

Consider F_i^f, for some $i \geq 1$. The algorithm expands the vertices in F_i^f, then recomputes F_i^f. We give the following definition.

Definition 3. *We say that level L_i, for $i \geq 1$, is* cleared *in the forward direction, if F_i^f is empty. Likewise, level L_j, for $j \geq 1$, is* cleared *in the backward direction, if F_j^b is empty.*

Theorem 2. *If level L_i is cleared in the forward direction, then the $d2s$ values in TA^f, such that $d2s \leq L_i$, are final and represent distances to the source vertex s.*

Proof.
Suppose not, i.e. let us assume there exists a vertex $v \in V$ processed in level L_i and assigned a $d2s$ value d, but in a later level, $L_{i'} > L_i$, v is assigned a lower $d2s$ value $d' < d$. Vertex v will be discovered in level $L_{i'}$ through a vertex, say u, not processed in iteration L_i, which implies that vertex u has a $d2s$ value greater than L_i, therefore $d' > L_i > d$, which is a contradiction. $\square$

Similarly, we can show that

Theorem 3. *If level L_j is cleared in the backward direction, then the $d2t$ values in TA^b, such that $d2t \leq L_j$, are final and represent distances to the target vertex t.*

Once we clear a level L_i (L_j), we finalize all the vertices in TA^f (TA^b) with $d2s \leq L_i$ ($d2t \leq L_j$). We finalize those vertices by setting their f^f or f^b flag to 1 ($true$). Observe that as we go to the next level, L_{i+1} (L_{j+1}), the set F_{i+1}^f (F_{i+1}^f) will contain those vertices of TA^f (TA^b) that are unfinalized and have a $d2s$ value less than L_{i+1} (L_{j+1}). Since, all the vertices of TA^f (TA^b) with $d2s \leq L_i$ ($d2t \leq L_j$) have been finalized, we have that in level L_{i+1} (L_{j+1}), we only process vertices of TA^f (TA^b) with $L_i < d2s \leq L_{i+1}$ ($L_j < d2t \leq L_{j+1}$).

In the B-LF-BFS algorithm, we have a clear way to finalize vertices, hence the termination decision becomes easier; it happens when we find a vertex that is finalized by both the forward and backward directions, or if there are no more unfinalized vertices in TA^f or TA^b. The validity of this condition for any bidirectional shortest-path algorithm (that finalizes vertices) is shown in [18]. In contrast, we did not have the ability to easily decide how to finalize vertices in B-R-BFS, hence, we had to resort to a much more complex termination procedure.

The computed F^f and F^b sets in B-LF-BFS are only a part of the TA^f or TA^b tables. Therefore the joins with the TE table are restricted in size compared to a full BFS approach.

Whereas B-R-BFS achieves the join reduction by partitioning the TE table, B-LF-BFS achieves a similar effect by performing first a selection on the TA^f and TA^b tables to generate a smaller set to join with TE.

The pseudo-code for clearing a level in B-LF-BFS in the forward direction is given in Algorithm 1. Please refer to Fig. 2 for the SQL statements. Once a level is cleared, we go to the next level until no more expansions are possible. The backward direction is analogous. B-LF-BFS alternates between the forward and backward direction depending

Algorithm 1 B-LF-BFS expand and merge

```
 1: function ExpandAndMerge($TA^f$, $TE$, $L_i$)
 2:     do
 3:         Compute $F_i^f$ (view F)
 4:         Join $F_i^f$ with $TE$ (view F_TE)
 5:         For each $nid$ in F_TE,
 6:                 compute the tuple with the smallest $d2s$
 7:         Merge all the tuples thus produced into $TA^f$
 8:         $n \leftarrow$ number of merged tuples
 9:     while $n \neq 0$
10:     Finalize all vertices in $TA^f$ with $f^f = 0$ and $d2s \leq L_i$
11: end function
```

on the number of tuples merged in TA^f and TA^b choosing each time the direction with the fewer merges.

The computations in lines 3–7 of Algorithm 1 are implemented by the SQL *MERGE* statement in Fig. 2. There, we first create two views, F and F_TE.[5] View F contains set F^f. View F_TE contains the join of F^f with TE.

Next, for each vertex id, nid, we compute the tuple with the smallest $d2s$ in F_TE. For this we use the **row_number**() SQL window function[6]. Then comes the merge of thus obtained tuples into table TA^f. If we obtained a tuple that is better (with respect to $d2s$) than a tuple with the same nid in TA^f, then the latter will be replaced by the former. Also, the tuples that do not have counterparts in TA^f (with respect to nid) will be simply inserted into TA^f.

Finally, we finalize all the vertices in TA^f with $f^f = 0$ (f_f = '0') and $d2s \leq L_i$. The finalization of vertices (tuples) in TA^f is done by setting the value of their f^f flag (f_f) in TA^f to 1. This is done with the last SQL statement in Fig. 2.

Now we analyze the number of iterations the algorithm would take. Suppose first we only do forward search. Let $mc > 0$ be the cost of the lightest edge in the graph. Let mp be the cost of the shortest path from s to t. In order to clear a level, we need $\lceil mp/step \rceil$ iterations in the worst case. In order to discover the shortest path from s to t, we need $\lceil mp/step \rceil$ levels in the worst case. Therefore, we need $\lceil (mp/step) \rceil * (step/mc)$ iterations in the worst case. When we do bidirectional search, we will need about half this number of iterations. This analysis shows that the algorithm terminates in a finite number of iterations. Based on the above reasoning and Theorem 2, we conclude the correctness of the algorithm.

6. B-LF-BFS WITH COMPRESSED ADJACENCY LISTS (B-LF-BFS-C)

In this section, we present B-LF-BFS-C, which enhances B-LF-BFS using a compressed representation of the input graph.

A RDBMS often uses more space than necessary for storing numeric datatypes (cf. [22]). Also, an edge table, such as TE, has unnecessary redundancy. For example, if a highly connected vertex v has, say 1000 neighbors, $u_1, \ldots, u_{1000}$, then v's id will repeat 1000 times to represent the 1000 edges that connect v to its neighbors, i.e. we will have the triples $(v, u_1, c_1), (v, u_2, c_2), \ldots, (v, u_{1000}, c_{1000})$.

A better alternative is to use an adjacency list of neighbors and costs, e.g. for v, we would have a list such as $[u_1, c_1,\ u_2, c_2,\ \ldots,\ u_{1000}, c_{1000}]$. While this is an improvement, we can do better than just storing the numbers in their original form. In fact, we can compress an adjacency list quite efficiently.

We borrow the idea of variable-byte-encoding of postings lists from Information Retrieval practice (cf. [28, 3, 46]). A posting list for a term is a list of documents that contain the term. For example, for a term, say *dog*, we can have a posting list like $[334, 345, 350]$, where the numbers are document ids containing *dog*. Observe that document ids are sorted in ascending order.

There is a similarity between a posting list and an adjacency list; instead of document ids we have neighbor ids.

For representing a posting list, we do not store the original document ids; rather, we store the *gaps* (or differences) between document ids. So, in the previous example, the posting list becomes $[334, 11, 5]$, where 11 is $345 - 334$, and 5 is $350 - 345$. Now, variable-byte encoding is used for the modified posting list. Specifically, we need 2 bytes for 334, and only one byte for each of the other two numbers, for a total of 4 bytes. In contrast, the original posting list, with fixed-byte-encoded integers of, say 4 bytes, needs 12 bytes.

We applied this idea for encoding the graph adjacency lists. We stored the obtained byte-encodings as *BLOB*'s (Binary Large Objects) in the TE table. More specifically, the TE table has now only two attributes, *fid* (as before) and *ncb* (which stands for *neighbor-cost bytestream*).

The pseudo-code for encoding/decoding sorted adjacency lists is given in algorithms 2, 3, and 4.

A number is encoded by a list of bytes. The rightmost 7 bits in a byte are *content* and represent a part of the number. The leftmost bit is an indicator flag. If it is 1, it means that the byte is the last one in the number encoding. If it is 0, it means there are more bytes following up in the encoding. In Algorithm 3, we encode a list of neighbor/cost numbers. We iterate over the elements of this list in pairs of neighbor and cost. We encode the difference of the current neighbor from the previous one, then encode the cost of the edge reaching the neighbor. When decoding a list of bytes (see Algorithm 4), we check for the leftmost bit of the bytes we read in order to detect the end of a number encoding. To decode a number, we extract and put together the 7 rightmost bits of the bytes in its encoding. We also check to see if the number we decoded is a neighbor (gap) or a cost, and proceed accordingly.

Expansion. In the following we describe the forward expansion. The backward version is analogous. The set F^f is computed using a similar query as before (see view F in Fig. 3). The join of F^f with TE returns now a result set of $(nid, ncb, d2s)$ tuples. The ncb value is a list of bytes encoding the neighbors of nid and the costs to reach these neighbors. The $d2s$ value represents the distance estimation of nid from the source vertex.

[5]We are calling F and F_TE "views", however, they are more precisely called "factored subqueries" (created with the SQL keyword *WITH*).

[6]See `https://en.wikipedia.org/wiki/Select_(SQL)`

Algorithm 2 Encoding of a number a

```
function encode($a$)
    $bytelist \leftarrow \emptyset$
    do
        $bytelist \leftarrow bytelist.prepend(a \ \& \ 0x7F)$
        $a \leftarrow a >> 7$
    while $a > 0$
    $bytelist[bytelist.size] = bytelist[bytelist.size] \mid 0x80$
    return $bytelist$
end function
```

Algorithm 3 Encoding of a list of neighbor/cost numbers

```
function encode($list$)
    $bytelist \leftarrow \emptyset$
    $prev\_neighbor \leftarrow 0$
    for each $neighbor, cost$ in $list$ do
        $\delta \leftarrow neighbor - prev\_neighbor$
        $bytelist_1 \leftarrow encode(\delta)$
        $bytelist_2 \leftarrow encode(cost)$
        $bytelist.concatenate(bytelist_1)$
        $bytelist.concatenate(bytelist_2)$
        $prev\_neighbor \leftarrow neighbor$
    end for
    return $bytelist$
end function
```

Algorithm 4 Decoding of a list of bytes $bytelist$

```
function decode($bytelist$)
    $list \leftarrow \emptyset$, $a \leftarrow 0$
    $prev\_neighbor \leftarrow 0$, $is\_neighbor \leftarrow true$
    for each $byte$ in $bytelist$ do
        if $byte < 0x80$ then
            $a \leftarrow a << 7 \mid byte$
        else
            $byte \leftarrow byte \ \& \ 0x7F$
            $a \leftarrow (a << 7) \mid byte$
            if $is\_neighbor = true$ then
                $a \leftarrow prev\_neighbor + a$
                $prev\_neighbor \leftarrow a$
                $is\_neighbor \leftarrow false$
            else
                $is\_neighbor \leftarrow true$
            end if
            $list.append(a)$
            $a \leftarrow 0$
        end if
    end for
    return $list$
end function
```

The neighbor-cost byte lists in the join result need to be decoded first in order to obtain tuples that can be merged into TA^f. This is done by a procedure described in Algorithm 5. In this procedure, we populate a new table, $EX^f(nid, d2s, p2s)$, which will contain the tuples that will be merged into TA^f. We truncate (clean-up) this table at each expansion round.

After performing the clean-up of EX^f (first statement in Fig. 3), then the computation of F^f, and the join with TE, the procedure proceeds with the creation of the tuples for EX^f. Specifically, for each tuple in the join result, it decodes the ncb byte list and iterates over the produced numbers. The iteration is done in pairs a, b to account for neighbor id and cost (a is neighbor id, b is cost). Let t be the current tuple being processed from the join result. The $d2s$ and $p2s$ values of the new tuple we create are set to be $t.d2s + b$ and $t.fid$, respectively.

Finally, once we populate the EX^f table, we merge it with TA^f using the SQL *MERGE* statement in Fig. 3. Differently from Fig. 2, the view creations in Fig. 3 are not part of the *MERGE* statement. Within the *MERGE* statement, for each nid, we first compute the tuple with the smallest $d2s$ in EX^f. Then, we proceed with the merge of these tuples into TA^f. The vertex finalization query is the same as in Fig. 2. The main algorithm for clearing a given level is shown in Algorithm 6.

```
--Prepare the EXf table for a fresh expansion
TRUNCATE TABLE EXf;

WITH
--Compute frontier
F(nid,p2s,d2s) AS (
   SELECT nid, p2s, d2s
   FROM TAf
   WHERE f_f='0' AND d2s <= Li
),

--Join frontier with the edge table
F_TE(nid,ncb,d2s) AS (
   SELECT TE. d, TE.ncb, TA.d2s
   FROM F JOIN TE ON F.nid=TE.fid
),
SELECT * FROM F_TE;

--Merge into TAf
MERGE INTO TAf
USING (
  --For each nid, select the tuple with the smallest d2s value
  SELECT nid, p2s, d2s
  FROM (SELECT nid, p2s, d2s,
              row_number() OVER
                 (PARTITION BY nid ORDER BY d2s) AS rn
          FROM EXf)
  WHERE rn = 1
) source
ON (TAf.nid=source.nid)
WHEN MATCHED THEN
   UPDATE SET d2s=source.d2s, p2s=source.p2s, f_f='0'
   WHERE source.d2s<TAf.d2s
WHEN NOT MATCHED THEN
   INSERT (nid,d2s,p2s,f_f)
   VALUES(source.nid, source.d2s, source.p2s, '0')
```

Figure 3: SQL statements for B-LF-BFS-C.

7. EXPERIMENTAL RESULTS

Setup. All our experiments are conducted on a consumer-grade machine with Intel i7, 3.4Ghz CPU, and 12Gb RAM, running Windows 7 Professional. The hard disk is Seagate Barracuda ST31000524AS 1TB 7200 RPM. We used the latest versions of two commercial databases (which we anonymously call D1 and D2) as well as PostgreSQL 9.4.4 (PG).

We performed our analysis on six real and ten synthetic graph datasets. We show the results for the real datasets in Fig. 4 and for synthetic datasets in Fig. 5.

Algorithm 5 Populating expansion table EX^f

```
1: procedure populate($EX^f$, $TA^f$, $TE$, $L_i$)
2:     $EX^f \leftarrow \emptyset$ (truncate statement in Fig. 3)
3:     Compute $F_i^f$ (view F in Fig. 3)
4:     Join $F_i^f$ with $TE$ (view F_TE in Fig. 3)
5:     for each $t$ in F_TE do
6:         $list \leftarrow decode(t.ncb)$
7:         for each $a, b$ in $list$ do
8:             $nid \leftarrow a$, $d2s \leftarrow t.d2s + b$, $p2s \leftarrow t.fid$
9:             Insert $(nid, d2s, p2s)$ into $EX^f$
10:        end for
11:    end for
12: end procedure
```

Algorithm 6 B-LF-BFS-C expand and merge

```
1: function ExpandAndMerge($TA^f$, $TE$, $L_i$)
2:     do
3:         $populate(EX^f, TA^f, TE, L_i)$
4:         Merge $EX^f$ into $TA^f$ (MERGE in Fig.3)
5:         $n \leftarrow$ number of merged tuples
6:     while $n \neq 0$
7:     Finalize all vertices in $TA^f$ with $f^f = 0$ and $d2s \leq L_i$
8: end function
```

The real datasets are Web-Google, Pokec, Live-Journal (all three from `http://snap.stanford.edu`), and UK 2002, Arabic 2005, UK 2005 (all three from `http://law.di.unimi.it/webdata`).

The characteristics of the real datasets are as follows.

Name	# of vertices	# edges	Diameter
Web-Google	875,713	5,105,039	21
Pokec	1,632,803	30,622,564	11
Live Journal	4,847,571	68,993,773	16
UK 2002	18,520,486	298,113,762	21
Arabic 2005	22,744,080	639,999,458	22
UK 2005	39,459,925	936,364,282	23

The last three datasets, UK 2002, Arabic 2005, and UK 2002 are significantly bigger than the first three as well as the datasets considered in [9]. UK 2005, for instance, has close to one billion edges. We give a bar chart of the edge numbers in Fig. 4d.

The synthetic datasets vary in size from 1 million edges to 15 million edges. We generated five random graphs with sizes of 1, 2, 5, 10, and 15 million edges (denoted by 1M, 2M, 5M, and 15M), and five graphs of the same sizes using the preferential attachment model.

In figures 5b, 5c, and 5d, we compare the performance of B-R-BFS, B-LF-BFS, and B-LF-BFS-C on random vs. preferential attachment graphs. We see that their performance on the two types of graphs is more or less the same. Therefore, we show results using random graphs in the rest of the charts of Fig. 5.

For edge weights, we generated random numbers from 1 to 100 using uniform distributions for both real and synthetic datasets.

Regarding indexes we experimented with clustered and non- clustered indexes on the edge table. The results using clustered indexes are better (see figures 5k and 5l). For the TA tables, we did not create indexes as this slowed down the *MERGE* operations and the performance of all the algorithms suffered.

Each running time is given in seconds and obtained as an average over 100 random s-t queries.

In the following, we give the questions we aim to address with our experiments.

Questions.

Q1 How scalable are the algorithms we consider? How do they compare to each other? In particular, can they handle large and very large graphs, e.g. Live Journal (large) and UK 2005 (very large)?

Q2 How well do the algorithms perform against baselines, such as the following variants of the bidirectional Dijkstra's algorithm: (a) in memory (when the graph fits there), (b) in RDBMS, and (c) in a modern native graph database (Neo4j)?

Q3 What are the best parameters for B-R-BFS, B-LF-BFS, and B-LF-BFS-C (number of partitions, p, for the first, and step size, s, for the second and third)?

Q4 What is the processing time trend as the size of the dataset grows?

Q5 What is the relative cost of database operations?

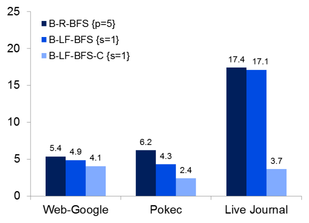
Q6 Is there a notable difference in the particular RDBMS chosen for this problem?

Answers.

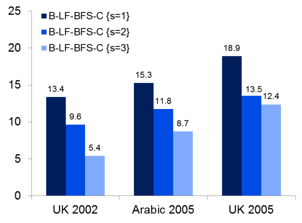
Q1. In figures 4a and 5a, we show the running times of the algorithms under their best parameter setup. For B-R-BFS, the datasets are partitioned into 5, 10, and 15 tables (p=5, p=10, p=15), respectively, and for B-LF-BFS and B-LF-BFS-C, *step* is set to 1,2 and 3 (s=1,s=2,s=3), respectively. Fig. 4a shows the running times for the real datasets, whereas Fig. 5a for the synthetic ones (all obtained using D2).

We see that B-LF-BFS outperforms B-R-BFS on all the datasets, with the difference being more pronounced for the random graphs. Recall though that our main contribution in B-LF-BFS is simplicity over B-R-BFS both in terms of algorithmic design as well as termination detection. The fact that B-LF-BFS performs better than B-R-BFS shows that we achieved simplicity without sacrificing performance.

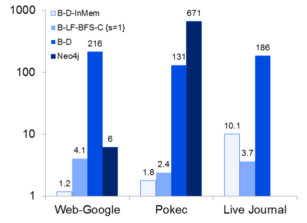
B-LF-BFS-C outperforms both B-LF-BFS and B-R-BFS, and the difference becomes quite significant for Live Journal. This behavior of B-LF-BFS-C is due to the fact that many fewer disk blocks are needed to store the TE table using the compression presented in Section 6. Therefore, there are less disk blocks to read to perform the main join. To see the compression achieved, please refer to Fig. 4l that shows the sizes of original and compressed TE tables for various datasets (using D2, the best performing RDBMS). The compression is quite significant for all the datasets, and for some, such as Arabic 2005 and UK 2005, it is by a factor of more than 20. This compression ratio shows that our compression is quite efficient; it also



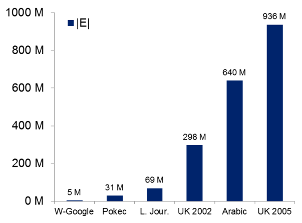
(a) Algorithms under their best parameter settings.



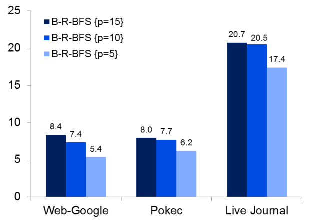
(b) B-LF-BFS-C, different step sizes, big graphs



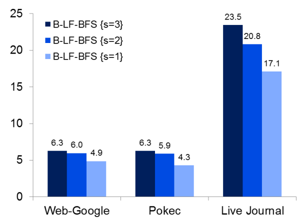
(c) B-LF-BFS-C vs. baselines.



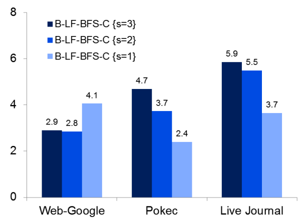
(d) Graph sizes in millions of edges (for reference).



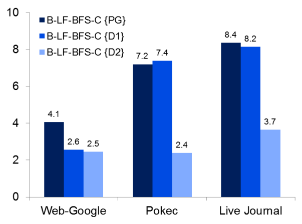
(e) B-R-BFS, different partition numbers.



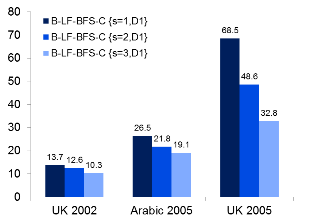
(f) B-LF-BFS, different step sizes.



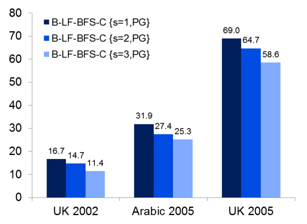
(g) B-LF-BFS-C, different step sizes.



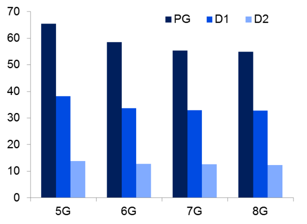
(h) B-LF-BFS-C {s=1}, different RDBMSs.



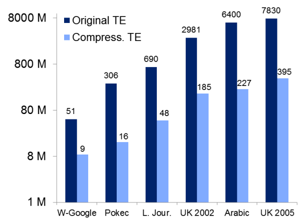
(i) B-LF-BFS-C, different step sizes, big graphs, D1.



(j) B-LF-BFS-C, different step sizes, big graphs, PG.



(k) Buffer size influence on run. time (B-LF-BFS-C).



(l) Original vs. Compressed TE (megabytes).

Figure 4: Experimental results for real datasets. All values on the vertical axes are times in *seconds*, except for 4d and 4l.

shows that there is a blowup factor when storing data uncompressed in a database. For example the CSV edge file of UK 2005 is 38 GB, which is less than half the size of the TE table (78 GB) for the same dataset. A similar blowup has also been observed in other works, e.g. [22].

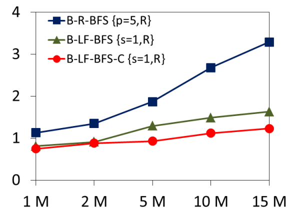
Regarding UK 2002, Arabic 2005, and UK 2005, B-LF-BFS-C is the only one to be able to handle them (see Fig. 4b). In fact, B-LF-BFS-C does on UK 2005 significantly better (by more than 27%) than what the other two algorithms can do for Live Journal, which is an order of magnitude smaller than UK 2005. B-R-BFS and B-LF-BFS were not able to handle the three largest datasets in our machine in a reasonable time; for instance, it took B-LF-BFS about two hours to compute a single s-t query on UK 2002.

We observe that the average time for B-LF-BFS-C is only 3.7 seconds on Live Journal, and 12.4 seconds on UK 2005. Fig. 4k shows the running time of B-LF-BFS-C on UK 2005 for different buffer allocations. We see there is only mild improvement as the buffer size grows. This applies to all three RDBMSs we used (D1, D2, PG). We discuss performance comparisons among RDBMSs later in this section.

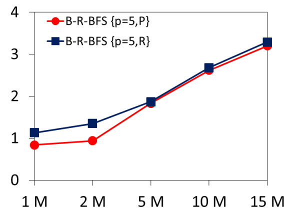
Q2. Regarding the baselines, we show the results in Fig. 4c. We compare there the baselines against each other and B-LF-BFS-C. As expected, the in-memory implementation of bidirectional Dijkstra's algorithm (B-D-InMem) is faster than its counterparts in RDBMS (B-D) and Neo4j. Also expected is the fact that B-LF-BFS-C does quite better than B-D and Neo4j; this is because B-LF-BFS-C benefits from a set-at-a-time approach (see Section 3.2 for a discussion). What is quite revealing though is that B-LF-BFS-C is a close contender to B-D-InMem and even outperforms it on Live Journal. This affirms the virtue of set-at-a-time evaluation, intelligent scheduling by the RDBMS, and graph compression in B-LF-BFS-C.

Q3. Now we focus on what the best parameters for the considered algorithms are. We see in Fig 4e that $p = 5$ is the best number of partitions for B-R-BFS in the real datasets. This is also confirmed by Fig. 5e for synthetic datasets. For the latter, we only show lines for $p = 5$ and $p = 10$ as the numbers for $p = 15$ were too large to be interesting. We explain this behavior of B-R-BFS as follows. While having more partitions (edge tables) helps to potentially achieve termination faster using the early joins (those using low-numbered edge tables), there is nevertheless an added penalty in terms of page scheduling by the RDBMS, if we are to join the same part of the F set with too many edge tables (which happens when the finalization is delayed). In other words, joins become too-small-too-many, and the performance suffers.

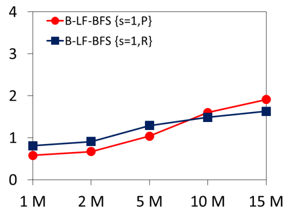
In Fig. 4f and 5f we see the B-LF-BFS performance for different step values on the real and synthetic datasets, respectively. We see that $s = 1$ is the best value for B-LF-BFS.



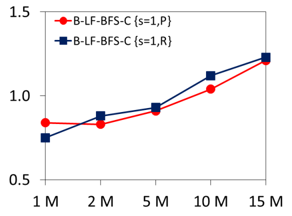
(a) Algorithms under their best parameter settings.



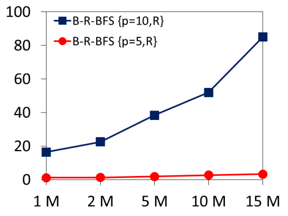
(b) B-R-BFS, pref. attachment and random graphs.



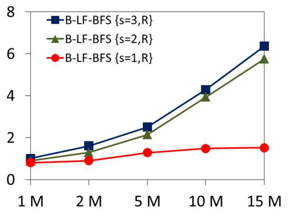
(c) B-LF-BFS, pref. attachment and random graphs.



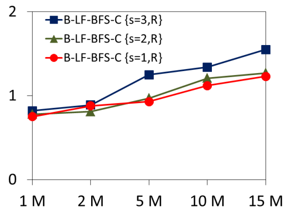
(d) B-LF-BFS-C, pref. attach. and rand. graphs.



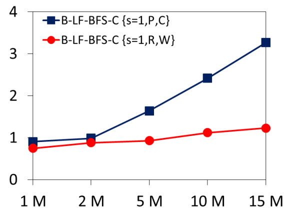
(e) B-R-BFS, different partition numbers.



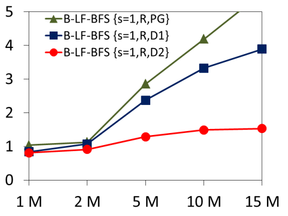
(f) B-LF-BFS, different step sizes.



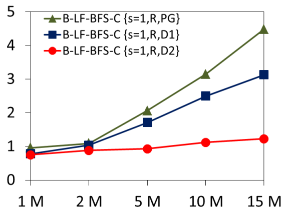
(g) B-LF-BFS-C, different step sizes.



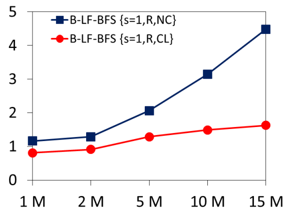
(h) B-LF-BFS-C, window functions vs. classical SQL.



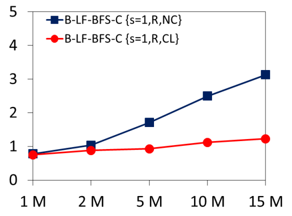
(i) B-LF-BFS, different RDBMSs.



(j) B-LF-BFS-C, different RDBMSs.



(k) B-LF-BFS, non-cluster. and cluster. ind.



(l) B-LF-BFS-C with non-cluster. and cluster. ind.

Figure 5: Experimental results for synthetic datasets. All values on the vertical axes are times in *seconds*.

Whereas the differences in running time for $s = 1, 2, 3$ are not big for the real datasets, they become quite noticeable for the synthetic datasets. For the latter, the performance for $s = 1$ is significantly better than for $s = 2, 3$. Recall that the greater the step size, the bigger the F sets become, and the more the B-LF-BFS gets closer to BFS. For B-LF-BFS-C, on the other hand, we find that $s = 3$ is the best value for the three big real datasets (Fig. 4b), whereas $s = 1$ is the best value for the medium real datasets (Fig. 4g) and synthetic datasets (Fig. 5g). This suggests that some parameter tuneup is needed depending on the graph.

The tuneup of the step size can be performed by randomly selecting a set of source-target pairs as we are doing in our experiments. Then, we test different values for $step$ starting from a value equal to the cost of the lightest edge and incrementing it by this amount each time. What we look for is some value for the step size such that the set-at-the-time evaluation has an opportunity to better schedule disk accesses while not doing too much non-optimal work. As our experiments show, moderate values for the step size give a good balanced evaluation.

Q4. In Fig. 5a, we see that the curve for B-R-BFS, even for $p = 5$, grows faster compared to B-LF-BFS and B-LF-BFS-C. This trend for B-R-BFS is more pronounced in Fig. 5e for $p = 10$. The curve for B-LF-BFS grows in general mildly, unless its step value is not tuned properly (see Fig. 5f). Finally, B-LF-BFS-C has the mildest growing curve of the three algorithms and is somewhat “forgiving” even when s is not tuned to the best value (see Fig. 5g).

Q5. The experiments in Fig. 6 were conducted using synthetic data sets (random graphs of 1-15 million edges) with $step = 3$. Fig. 6a shows the relative time for finding frontier nodes versus and the time taken for executing the join queries in B-LF-BFS. The Merge time was not considered in the figure as it was insignificant. We can see that the processing time is mainly consumed by join queries. Fig. 6b shows the relative time for B-LF-BFS-C to decode compressed tuples in table TE, find frontier nodes, and execute join queries. We can see that decoding did not consume the significant portion of the whole processing time. It is still the join time that dominates.

Q6. Figures 4h and 5j show the performance of different RDBMSs when running B-LF-BFS-C (best algorithm). We allocated the same amount of buffer space, 6G, and created the same index setup for all three of RDBMSs we used. We see that D1 and Postgres performed similarly, however, D2 significantly outperformed both of them. A

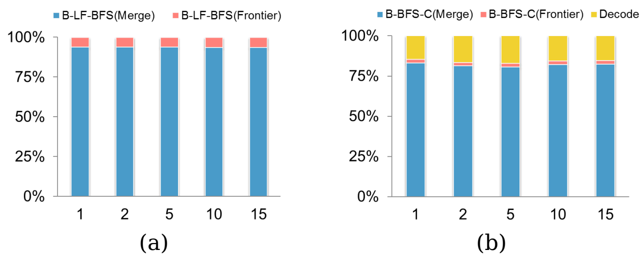


Figure 6: Relative cost of different database operations.

similar behavior can also be observed when running B-LF-BFS (see Fig. 5i).

To better see the performance of B-LF-BFS-C on different RDBMSs, we also give figures 4i (for D1) and 4j (for Postgres), which should be compared with Fig 4b (for D2).

These results suggest that even though well-developed RDBMSs (such as those we consider) are close competitors in well-known benchmarks, when it comes to handling specialized workloads (such as graph operations), they show noticeable differences.

8. RELATED WORK

Computing s-t queries on large complex networks has received considerable attention from the research community (cf. [1, 2, 23] and [6, 15, 29, 31, 42]). The first and second group of works compute exact and approximate answers, respectively. As we provide exact answers to s-t queries, our work is more related to the first group. The works in the first group achieve very good scalability, but focus on unweighted graphs, which is an easier problem to tackle. In unweighted graphs, the length of a path amounts to the number of its edges. Since social and web graphs have typically a small diameter, computing s-t shortest paths on unweighted graphs does not need to travel more than few hops. On the other hand, if edge weights are taken into consideration, the small diameter of the graphs is not that important anymore as shortest paths can contain an arbitrary number of edges, hence, many more expansions are needed. In this paper, in contrast with the aforementioned works, we focus on weighted graphs, which are more general and challenging to handle. While we do not show experimental results for unweighted graphs, we mention that we tested extensively with such graphs in RDBMS and our performance was considerably better than for weighted graphs.

Bidirectional computation for finding shortest paths from a source to a target has been suggested in several works (cf. [11, 18]. In [18], it is shown that proper termination is when both the forward and backward search finalize a graph vertex in common. Recall, this was possible for the bidirectional Dijkstra's algorithm and B-LF-BFS, but not for B-R-BFS (for which we need to resort to a more complex termination procedure). In [11], the benefits of bidirectional search are shown by experiments for graphs that fit in memory. Another part of [11] is about A* heuristics for speeding up the s-t shortest path computation. Such heuristics, however, were in practice observed in [11] to be mainly useful for spatial networks (and not much for complex networks). In this paper, we focus on complex networks (social and web networks). As such, we have not employed A* heuristics.

Relational Databases have been often used to store complex data, such as XML and RDF graphs (cf. [5, 32] and [4, 17], respectively). They have also been shown to be a good choice to support advanced applications, such as data mining [34, 45] and machine learning [10, 25].

For graph queries, as [16, 43] argue, relational technology can sometimes outperform more specialized solutions. An interesting work that uses relational technology for answering subgraph and supergraph queries is [33]. These queries are different from our source-to-target shortest path queries. Other works, such as [8, 24] use relational databases to build vertex-centric (VC) systems in the Pregel model. As we explained in Section 2, a VC computation is not a good fit for computing s-t shortest paths. In terms of table structure, the edge tables used in B-R-BFS and B-LF-BFS are similar to the edge table in [8]. All these works (including ours) suggest that using relational databases for graph management can be for some problems better than using specialized graph engines.

9. CONCLUSIONS

We showed that designing algorithms for RDBMS is a good avenue to pursue for graphs that do not fit in memory, and sometimes, even for graphs that can (e.g. Live Journal). Also, RDBMS technology is quite mature and can accommodate complex data and algorithms, sometimes, even better than special purpose systems.

We presented a correct procedure for deciding termination in B-R-BFS. We argued that it is challenging to determine whether a vertex has its distance finalized in B-R-BFS. Then we showed that we can decide termination by carefully deriving lower bounds on the forward and backward distances of vertices from the source and target.

We gave next a new algorithm, B-LF-BFS, which performs restrictive BFS by selecting only a part of the visited vertices as a set to be expanded. This was achieved by setting distance levels that need to be cleared (in terms of expansions) before going to the next level. We showed that, once a level is cleared, the vertices that were expanded in that level have their distance finalized. This allowed us to use a much simplified termination procedure.

Then, we presented B-LF-BFS-C, an algorithm that enhances B-LF-BFS by using a compressed representation of neighbor-cost lists. The compression achieved was such that B-LF-BFS-C was able to handle graphs of an order of magnitude bigger than what B-R-BFS and B-LF-BFS could.

Using detailed experiments, we showed that all three algorithms scale well (for their best parameter setup), and B-LF-BFS-C in particular can produce results in a reasonable time even on the largest dataset we experimented with, UK 2005, with close to one billion edges, using only a consumer-grade machine.

As future work, we would like to extend our results using RDBMS to shortest paths in graphs that are both weighted and labeled with symbols from a finite alphabet [12, 35, 40]. In this case, the paths allowed to follow are specified by regular expressions and the goal is to compute shortest paths that spell words in the query language [13, 14, 39].

10. REFERENCES

[1] R. Agarwal, M. Caesar, P. B. Godfrey, and B. Y. Zhao. Shortest paths in less than a millisecond. In *ACM Workshop on Online Social Networks*. ACM, 2012.

[2] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*. ACM, 2013.

[3] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[4] H. Chen, Z. Wu, H. Wang, and Y. Mao. Rdf/rdfs-based relational database integration. In *ICDE*, pages 94–94. IEEE, 2006.

[5] L. J. Chen, P. Bernstein, P. Carlin, D. Filipovic, M. Rys, N. Shamgunov, J. F. Terwilliger, M. Todic, S. Tomasevic, D. Tomic, et al. Mapping xml to a wide sparse table. *Knowledge and Data Engineering, IEEE Transactions on*, 26(6):1400–1414, 2014.

[6] A. Das Sarma, S. Gollapudi, M. Najork, and R. Panigrahy. A sketch-based distance oracle for web-scale graphs. In *WSDM*. ACM, 2010.

[7] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[8] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[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] W. Gatterbauer, S. Günnemann, D. Koutra, and C. Faloutsos. Linearized and single-pass belief propagation. *PVLDB*, 8(5):581–592, 2015.

[11] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, pages 156–165, 2005.

[12] G. Grahne and A. Thomo. Query answering and containment for regular path queries under distortions. In *International Symposium on Foundations of Information and Knowledge Systems*, pages 98–115. Springer, 2004.

[13] G. Grahne and A. Thomo. Regular path queries under approximate semantics. *Annals of Mathematics and Artificial Intelligence*, 46(1-2):165–190, 2006.

[14] G. Grahne, A. Thomo, and W. Wadge. Preferentially annotated regular path queries. In *International Conference on Database Theory*, pages 314–328. Springer, 2007.

[15] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM*, pages 499–508. ACM, 2010.

[16] A. Gubichev and M. Then. Graph pattern matching: Do we have to reinvent the wheel? In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–7. ACM, 2014.

[17] S. Harris and N. Shadbolt. Sparql query processing with conventional relational database systems. In *Web Information Systems Engineering–WISE 2005 Workshops*, pages 235–244. Springer, 2005.

[18] M. Holzer, F. Schulz, D. Wagner, and T. Willhalm. Combining speed-up techniques for shortest-path computations. *ACM Journal of Experimental Algorithmics*, 10, 2005.

[19] J. E. Hopcroft and R. E. Tarjan. Efficient algorithms for graph manipulation. 1971.

[20] H.-P. Hsieh, C.-T. Li, and R. Yan. I see you: Person-of-interest search in social networks. In *SIGIR*, pages 839–842. ACM, 2015.

[21] S.-W. Huang, D. Tunkelang, and K. Karahalios. The role of network distance in linkedin people search. In *SIGIR*, pages 867–870. ACM, 2014.

[22] A. Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.

[23] R. Jin, N. Ruan, Y. Xiang, and V. Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *SIGMOD*, pages 445–456. ACM, 2012.

[24] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. Vertexica: your relational friend for graph analytics! *PVLDB*, 2014.

[25] M. L. Koc and C. Ré. Incrementally maintaining classification using an rdbms. *PVLDB*, 4(5):302–313, 2011.

[26] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, volume 12, pages 31–46, 2012.

[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[28] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge university press, 2008.

[29] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, pages 867–876. ACM, 2009.

[30] R. C. Prim. Shortest connection networks and some generalizations. *Bell system technical journal*, 36(6):1389–1401, 1957.

[31] M. Qiao, H. Cheng, L. Chang, and J. X. Yu. Approximate shortest distance computing: A query-dependent local landmark scheme. *IEEE Trans. Knowl. Data Eng.*, 26(1):55–68, 2014.

[32] M. Rys, D. Chamberlin, and D. Florescu. Xml and relational database management systems: the inside story. In *SIGMOD*, pages 945–947. ACM, 2005.

[33] S. Sakr and G. Al-Naymat. Efficient relational techniques for processing graph queries. *Journal of Computer Science and Technology*, 25(6):1237–1255, 2010.

[34] X. Shang, K. U. Sattler, and I. Geist. Sql based frequent pattern mining without candidate generation. In *SAC*, pages 618–619. ACM, 2004.

[35] M. Shoaran and A. Thomo. Fault-tolerant computation of distributed regular path queries.

Theoretical Computer Science, 410(1):62–77, 2009.

[36] P. Singla and M. Richardson. Yes, there is a correlation:-from social networks to personal behavior on the web. In *WWW*. ACM, 2008.

[37] C. Sommer. Shortest-path queries in static networks. *ACM Computing Surveys (CSUR)*, 46(4):45, 2014.

[38] N. V. Spirin, J. He, M. Develin, K. G. Karahalios, and M. Boucher. People search within an online social network: Large scale analysis of facebook graph search query logs. In *CIKM*. ACM, 2014.

[39] D. Stefanescu and A. Thomo. Enhanced regular path queries on semistructured databases. In *International Conference on Extending Database Technology*, pages 700–711. Springer, 2006.

[40] D. C. Stefanescu, A. Thomo, and L. Thomo. Distributed evaluation of generalized path queries. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 610–616. ACM, 2005.

[41] G. Swamynathan, C. Wilson, B. Boe, K. Almeroth, and B. Y. Zhao. Do social networks improve e-commerce? a study on social marketplaces. In *First Workshop on Online Social Net.* ACM, 2008.

[42] K. Tretyakov, A. Armas-Cervantes, L. García-Bañuelos, J. Vilo, and M. Dumas. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *CIKM*, pages 1785–1794. ACM, 2011.

[43] A. Welc, R. Raman, Z. Wu, S. Hong, H. Chafi, and J. Banerjee. Graph analysis: do we have to reinvent the wheel? In *First International Workshop on Graph Data Manag. Experiences and Systems*. ACM, 2013.

[44] S. A. Yahia, M. Benedikt, L. V. Lakshmanan, and J. Stoyanovich. Efficient network aware search in collaborative tagging sites. *PVLDB*, 1(1), 2008.

[45] B. Zou, X. Ma, B. Kemme, G. Newton, and D. Precup. Data mining using relational database management systems. In *Advances in Knowledge Discovery and Data Mining*, pages 657–667. Springer, 2006.

[46] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, pages 59–59. IEEE, 2006.