Of Hammers and Nails: An Empirical Comparison of Three Paradigms for Processing Large Graphs

Marc Najork
Microsoft Research
Mountain View, CA, USA
najork@microsoft.com

Dennis Fetterly
Microsoft Research
Mountain View, CA, USA
fetterly@microsoft.com

Alan Halverson
Microsoft Corporation
Madison, WI, USA
alanhal@microsoft.com

Krishnaram Kenthapadi
Microsoft Research
Mountain View, CA, USA
krisken@microsoft.com

Sreenivas Gollapudi
Microsoft Research
Mountain View, CA, USA
sreenig@microsoft.com

ABSTRACT

Many phenomena and artifacts such as road networks, social networks and the web can be modeled as large graphs and analyzed using graph algorithms. However, given the size of the underlying graphs, efficient implementation of basic operations such as connected component analysis, approximate shortest paths, and link-based ranking (*e.g.*PageRank) becomes challenging.

This paper presents an empirical study of computations on such large graphs in three well-studied platform models, *viz.*, a relational model, a data-parallel model, and a special-purpose in-memory model. We choose a prototypical member of each platform model and analyze the computational efficiencies and requirements for five basic graph operations used in the analysis of real-world graphs *viz.*, PageRank, SALSA, Strongly Connected Components (SCC), Weakly Connected Components (WCC), and Approximate Shortest Paths (ASP). Further, we characterize each platform in terms of these computations using model-specific implementations of these algorithms on a large web graph. Our experiments show that there is no single platform that performs best across different classes of operations on large graphs. While relational databases are powerful and flexible tools that support a wide variety of computations, there are computations that benefit from using special-purpose storage systems and others that can exploit data-parallel platforms.

Categories and Subject Descriptors

G.2.2 [**Discrete mathematics**]: Graph Theory—*Graph algorithms, path and circuit problems*; H.2.4 [**Database management**]: Systems—*Distributed databases*

General Terms

Algorithms, Experimentation, Performance

Keywords

Very large graphs, graph algorithms, databases, data-parallel computing, graph servers

1. INTRODUCTION

Large scale networks are increasingly finding their place beyond traditional networks such as the Internet and telecommunication networks. Social and professional networks such as Facebook and LinkedIn have seen explosive growth in recent years. The social graph of Facebook, for example, contains more than 750 million active users[1] and an average friend count of 130. The widespread use of GPS applications and devices is built upon the graph of roads, intersections, and topographical data. The underlying graph in all these networks allows a wide range of analyses to measure different properties of these networks such as communities, size of the largest component, and distances between nodes. For example, the affinity between people in a social network can be modeled as a connectivity problem between nodes in the underlying graph where edges denote shared interests between any two people in the network. Computation of the popularity of a web page and of the similarity between two web pages are widely studied problems.

The analysis of large graphs presents challenging algorithmic problems. Fundamental applications like component analysis using breadth-first search (BFS) or depth-first search (DFS), PageRank computation, and shortest path computation become prohibitively expensive in the absence of suitable representation of the underlying graph. For example, the classic algorithm of Dijkstra [11] would take days to run on a web graph containing tens of billions of nodes and trillions of edges. Even in a distributed setting, the sequentially dependent nature of Dijkstra's algorithm would require huge amounts of communication. For the analysis of large graphs, research has focused on the relational model, the streaming model, and other special-purpose and often application-specific models.

Given the explosive growth of both the number and size of graph datasets as well as the increased complexity of analyzing large graphs, choosing a scalable platform for graph data storage and analysis is of paramount importance. Data parallel computation models such as MapReduce [14] provide a natural candidate platform. However, the complexity of implementing diverse and rich graph applications as MapReduce jobs produces the need for a higher level language to express the implementations in. Several languages for data-parallel systems exist (e.g. Sawzall [30], Pig [27], and DryadLINQ [39]) with a common goal of providing a declarative programming surface that allows the programmer to ask for "what" data they want rather than "how" to get it. Many of these languages are inspired by SQL, the standard language surface of relational database management systems (RDBMS). Thirty years of

[1]`http://www.facebook.com/press/info.php?statistics`

RDBMS research experience suggest that it should be considered as a candidate platform as well. Still, sometimes the "how" of data retrieval and processing is important. For example, an in-memory graph store will be extremely efficient in retrieving the neighborhood of a random node and therefore a very effective model for component analysis in a graph. We therefore have a "hammers and nails" problem – that is, the hammers are the data storage and processing platforms and the nails are the graph applications to evaluate over the data. Given the collection of hammers and nails, how should we best match them for efficiency, cost, and performance?

In this study, we undertake an extensive empirical study of three representative systems of the above models and their effectiveness in solving some prototypical graph algorithms using a large web graph. The contributions of our study are two-fold. First, given the size of the graph, we tailored the graph algorithms to work on large-scale data under the three settings. Second, we provide a first-of-a-kind study to show the effectiveness of each model to solve different classes of graph algorithms. We describe the implementations of five basic graph algorithms, *viz.*, PageRank [28], SALSA [20, 25], SCC [36, 4], WCC [11], and ASP [13], on a prototypical member of each model and further compare these implementations in terms of performance, scalability, and ease of implementation.

2. RELATED WORK

Recently, a lot of research has been directed toward studying large-scale graph algorithms necessitated by applications in large networks such as road networks, wireless networks, the Internet, and social networks [21, 10, 32, 8, 12]. The implementations in these studies have mostly relied on application specific storage and access models for the underlying graphs. Studies have also focused on developing graph algorithms in the streaming model [16, 17].

On problems involving large data sets, studies using relational databases have focused on issues related to scale-up and scale-out. Traditional ways to scale a database included upgrading to faster CPUs, adding more RAM and disks. However, these approaches often run into technical and economic limitations. In the limit, twice the money does not buy twice the performance. Recently, there has been a plethora of work on "scaling out" as exemplified by systems such as Netezza, column-based storage systems [33, 2, 26, 9], and managing data in cloud databases [1].

In the class of data-parallel systems, MapReduce [14] is the first system to use clusters of commodity hardware for distributed computing on large data sets. The *map* step in a MapReduce system typically involves mapping different parts of the problem to different nodes in the system. Each subproblem is processed at the worker node. The *reduce* step involves the master node aggregating the answers from all the worker nodes and processing it to generate the final output. The order of computation and the dataflow between nodes is represented using a directed graph known as the *dataflow graph*. The scalability and simplicity of this paradigm led the open-source community to develop Hadoop [6]. In fact, there are hybrid architectures that combine a MapReduce framework with a RDBMS, such as HadoopDB [3], Hive [37], and Aster Data. One of the drawbacks of MapReduce and Hadoop is that the dataflow graph has a fixed topology. Dryad [18] was developed to overcome this drawback. The dataflow graph in Dryad is computed dynamically based on the data at run-time. There are related platforms that are built on top of either MapReduce, Hadoop, or Dryad. Sawzall [30] was developed as an interpreted procedural language to work on MapReduce; Pig [27] was developed for analyzing large data sets on Hadoop; and DryadLINQ [39] is a programming language developed for writing applications on Dryad.

In the space of special-purpose graph stores, the Connectivity Server [5] is a link-server built to access the linkage information for all web pages indexed by the AltaVista search engine. Suel and Yuan [35] built a similar server to store a compressed web graph consisting of several hundred million nodes and a billion links. A number of systems exploit different compression schemes to scale traditional in-memory stores to large graphs [31, 7]. The system that is representative of this class in our study is the Scalable Hyperlink Store [24]. It differs from the earlier in-memory stores in that it is distributed, following the client-server model.

Pavlo et al. [29] performed a similar study as ours in which they provide benchmarks for Hadoop, Vertica, and an unnamed RDBMS on an identical cluster of 100 commodity nodes. Their workload consisted of different flavors of `grep`, database queries with joins and simple aggregations on clickstream data. They showed that the RDBMS in general outperformed Hadoop. While we agree that databases are incredibly useful pieces of infrastructure that can support an extremely wide set of applications, there are applications when more specialized tools are more appropriate. One example is time-constraints – applications that have extremely low latency requirements but do not require the guarantees that come with databases, such as transactional semantics. Another example is economic constraints – when a service will be run on a million computers, the potential savings in capital expenditure justify the cost of implementing a bespoke solution. More recently, a pair of articles by Stonebraker et al. [34] and Dean and Ghemawat [15] appeared in CACM to continue the discussion by arguing both for and against the use of an RDBMS for general data processing, respectively. Stonebraker et al. argue that MapReduce provides an excellent extract-transform-load (ETL) tool for pre-processing data to load into an RDBMS, but question the repeated parsing of lines that the Hadoop implementation does. Dean and Ghemawat counter by pointing to their Protocol Buffer serialization for data as a way to avoid repeated line parsing, and provide guidelines for MapReduce system usage to avoid some of the overheads pointed out in [29]. In our study, we will focus on the special case of large graph storage and query processing in our three chosen platforms.

Another paradigm for graph computations that supports iterative processing of messages between vertices is proposed in [22]. In each iteration each vertex processes messages received in the previous iteration and sends out messages that can potentially change the graph topology. Finally, we note that there are many other graph databases based on the paradigms that we study in this paper. The reader is referred to the Wikipedia article on graph databases [38] and the references within. While comparing these products would be an interesting study in itself, we note that the focus of this work has been to study and compare three paradigms of computation on graph databases instead of the products themselves.

3. EXPERIMENTAL SETUP

This section describes the data sets we used as the input to our experiments, we hardware we ran them on, and the software platforms (SQL Server PDW, DryadLINQ, and SHS) on top of which we implemented the five graph algorithms.

3.1 The data sets

We perform our experiments on the ClueWeb09 datasets[2]. The larger "Category A" dataset was derived from 1 billion web pages crawled during January and February 2009. It is comprised of two files, one containing all unique URLs (325 GB uncompressed, 116 GB compressed) and the other containing all the outlinks (71 GB uncompressed, 29 GB compressed). The corresponding graph con-

[2]`http://boston.lti.cs.cmu.edu/Data/clueweb09/`

sists of 4.77 billion unique URLs and 7.94 billion edges. The maximum out-degree is 2,079 and the maximum in-degree is 6,445,063. The smaller "Category B" dataset contains the first 50 million English pages in Category A. It is comprised of two files, one containing all unique URLs (30 GB uncompressed, 9.7 GB compressed) and the other containing all the outlinks (3.8 GB uncompressed, 1.6 GB compressed). The corresponding graph consists of 428 million unique URLs and 454 million edges. The maximum out-degree is 1,055 and the maximum in-degree is 310,483.

For the evaluation of the SALSA algorithm, we use 171 queries sampled from the Bing query log. For each query, we extracted up to 5,000 top URLs (based on BM25F, a text-based ranking feature) from the ClueWeb09 Category A or B pages, thereby obtaining a total of 831,591 and 826,364 associated URLs, respectively.

3.2 The hardware

We performed all our experiments on a cluster of 17 Dell PowerEdge 2950 servers, all running Windows Server 2008 R2. Each server had two Intel Xeon E5430 processors clocked at 2.66 GHz, 16 GB of DDR2 RAM, and six 146 GB, 15K RPM SAS drives. The computers were each connected to a Force10 S50-01-GE-48T-AC 48-port full-crossbar Gigabit Ethernet switch via Gigabit Ethernet. For the set of experiments where we used a single server, we ran the client/control program on the same machine; otherwise we used 16 machines to run the platform and one machine to run the client.

3.3 The platforms

We next describe the three platforms we use, representing the three paradigms for processing large graphs. We also discuss how the web graph is preprocessed, loaded and distributed into each of the three systems, since the performance of the graph algorithms depends heavily on the way the data is configured on the systems.

SQL Server PDW

Microsoft SQL Server 2008 R2 Parallel Data Warehouse [23] (SQL Server PDW) is a multi-user client/server RDBMS developed and commercialized by Microsoft Corporation. It provides standard RDBMS functionality – transactions, concurrency control, recovery, and a SQL language surface plus extensions (T-SQL). The relational data model is at the heart of data processing for SQL Server PDW. Set-oriented operations are preferred to row-by-row processing due to overheads associated with query parsing, algebraization, optimization, and execution that are incurred for each SQL statement passed to the system. SQL Server PDW uses a Control Node to accept a user query, produces a parallel execution plan, and executes the plan on a set of Compute Nodes. The Control Node is also responsible for presenting a "single system image" to users of the system, allowing expression of T-SQL queries as if all data is stored in a single database instance. SQL Server PDW is an appliance product that provides a choice of hardware configurations to users. The experiments we performed for this paper utilized the hardware described in Section 3.2, and therefore cannot be considered indicative of real-world SQL Server PDW performance.

The chosen relational schema for the ClueWeb09 data contains two relations: the `nodes` relation with attributes `id` of type bigint and `url` of type VARCHAR(900)[3]; and the `edges` relation with attributes `src` and `dest`, both of type bigint. By contrast, the original ClueWeb09 edges file contains an adjacency list per source node. We partitioned the edges and nodes files into 8 pieces to facilitate a parallel bulk insert operation into SQL Server PDW.

[3]In SQL Server PDW, index keys are limited to 900 bytes, and providing efficient URL-to-ID lookups requires an index on URL.

DryadLINQ

DryadLINQ [39][4] is a system for large scale distributed data-parallel computing using a high-level programming language. Compared to programming environments such as SQL, MapReduce, and Dryad, DryadLINQ supports general-purpose imperative and declarative operations on datasets within a traditional high-level programming language and thereby provides a higher layer of abstraction. A DryadLINQ program is a sequential program composed of LINQ expressions and can be written and debugged using standard .NET development tools. LINQ expressions are similar to SQL statements, however they are commonly used in conjunction with lambda expressions (specified with `=>` operators) where each expression will be applied to every element of the stream. DryadLINQ adds several methods to enable data-parallel computations, including `PartitionTable.Get` and `ToPartitionedTable` which will respectively read from and write to a data stream spread across multiple machines. Distribution of data is done using the `HashPartition` and `RangePartition` methods. The `Apply` method both consumes and produces a stream of elements, which is useful for algorithms that employ a sliding window. The DryadLINQ system automatically and transparently translates the data-parallel portions of the program into a distributed execution plan which is passed to the Dryad execution framework. By design, DryadLINQ is inherently specialized for streaming computations. In general, DryadLINQ is suitable for computations that can be expressed using LINQ constructs. However, executing a DryadLINQ program incurs at least a few seconds of overhead. Hence DryadLINQ is not appropriate for all applications, e.g. applications requiring low-latency distributed database lookups.

The processed ClueWeb09 data is partitioned into the cluster machines based on a hash value of the page identifier such that all pages and their associated link identifiers are present in the same machine. After the data is loaded into the cluster, whenever a new job is started, we incur a small overhead as the binaries need to be copied to all the machines.

Scalable Hyperlink Store

The Scalable Hyperlink Store (SHS) [24] is a distributed in-memory database for storing large portions of the web graph. It partitions the nodes (URLs) and the edges (hyperlinks) of the web graph over a set of servers, maintains all the data in main memory to allow extremely fast random access, and uses data compression techniques that leverage properties of the web graph (namely, the prevalence of relative links) to achieve fairly good compression. SHS was designed to enable research on structural properties of the web graph as well as new link-based ranking algorithms. It is not well suited for graph applications that require attaching arbitrary metadata to nodes and edges (such as anchor texts and weights) as doing so would result in decreased compression ratios. To achieve good compression, SHS maintains a bijection between textual URLs and numerical UIDs, and edges are represented in terms of UIDs. Initially the SHS Builder distributes the web graph data across a set of available SHS servers, by partitioning the URLs and their associated links according to the URLs' host components. This preserves link locality, that is, all URLs on a host and all links between pages on the same host are kept in the same SHS server. The clerk can determine the SHS server responsible for a URL by performing a local computation (hash function). Similarly the SHS server ID is encoded in UIDs, so that a client can determine the SHS server owning a UID locally. Client applications communicate with SHS via the `ShsClerk` class whose key methods are as follows:

[4]Available at `http://connect.microsoft.com/Dryad`

```
public class ShsClerk {
  public ShsClerk(string shsName) {...}
  public IEnumerable<long> Uids {...}
  public bool IsLastUid(long uid) {...}
  public long[] BatchedUrlToUid(string[] urls) {...}
  public string[] BatchedUidToUrl(long[] uids) {...}
  public long[][] BatchedGetLinks(bool fwd, long[] uids) {...}
  public long[][] BatchedSampleLinks(bool fwd, long[] uids,
      int numSamples, bool consistent) {...}
  public UidState<T> AllocateUidState<T>() {...}
  ...
}
public class UidState<T> {
  public void Dispose() {...}
  public T Get(long uid) {...}
  public void Set(long uid, T val) {...}
  public T[] GetMany(long[] uids) {...}
  public void SetMany(long[] uids, T[] vals) {...}
  public IEnumerable<UidVal<T>> GetAll() {...}
  public void SetAll(Func<long,T> f) {...}
}
public struct UidVal<T> {
  public readonly long uid;
  public readonly T val;
  public UidVal(long uid, T val) {...}
}
public struct UidBatch {
  public UidBatch(int sz) {...}
  public void Add(long uid) {...}
  public static implicit operator long[] (UidBatch x) {...}
  public bool Full { get {...} }
  public void Reset() {...}
}
public class UidMap {
  public UidMap(long[][] items) {...}
  public int this[long item] {...}
  public static implicit operator long[] (UidMap x) {...}
  ...
}
```

The `ShsClerk` constructor creates a new clerk object connected to the given SHS service. UIDs can be enumerated using the C# enumerator `Uids`, and `IsLastUid(u)` returns whether `u` is the final UID in that enumeration. `BatchedUrlToUid` and `BatchedUidToUrl` map URLs to UIDs and vice versa. The method `BatchedGetLinks` returns out-links/in-links associated with UIDs depending on the value of `fwd`. The method `BatchedSampleLinks` can be used to return either a uniformly random sample or a consistent sample of such links. The `Batched` methods allow clients to send many UIDs or URLs at once, thereby allowing them to amortize the RPC overhead, which is crucial for performance.

The version of SHS described in [24] provided a read-only store of the vertices and edges of the web graph. Since then, we have augmented the system to also provide a per-vertex read-write storage service. Calling `AllocateUidState<T>()` allocates a value of type `T` for each vertex in the graph, distributed over the SHS servers in the same way as the vertices themselves, and returns a `UidState` handle to this distributed mapping. The `UidState` provides a `Dispose` method for freeing up the server-side state, and methods for reading and updating the values. `Get` and `Set` read and update the value associated with a single UID, `GetMany` and `SetMany` read and update the values associated with a batch of UIDs, `GetAll` returns an enumeration of all UID/value pairs, and `SetAll` sets all values functionally dependent on their UIDs.

Among the auxiliary classes contained in the SHS package, two should be mentioned here: The `UidBatch` class makes it easy to aggregate UIDs into a bounded-size batch, for use with methods such as `BatchedGetLinks`. The `UidMap` class maps a multi-set of UIDs as returned by `BatchedGetLinks` to a set of UIDs realized as a `long` array that can be passed to `UidState.GetMany` and `UidState.SetMany`, and provides a way for obtaining the index of a given UID in that array.

We preprocess the ClueWeb09 dataset using DryadLINQ to produce a file containing the URLs of all pages and their links, which is then ingested by the SHS Builder to produce an SHS store.

4. ALGORITHM IMPLEMENTATION AND EVALUATION

We evaluated the five graph algorithms on the datasets described in Section 3.1 on each of the three systems. For each algorithm, we provide a short description of the graph algorithm and its implementation on each system. Before we delve into the details of our implementation of each algorithm, we describe the evaluation criteria, both quantitative and qualitative, for comparing the systems.

- Performance: We measure performance in terms of the running time for each implementation of each algorithm.
- Scalability: We characterize how the performance of each implementation changes as the size of the cluster and/or the size of the graph increases.
- Ease of implementation: We consider how elegant, concise and natural each implementation is.

We report performance in terms of running time in seconds. Table 1 shows the running time of the implementations of the five algorithms on each of the three platforms, processing the ClueWeb09 Category B graph using a single server. Table 2 shows running times of the same programs processing the same graph using sixteen servers. As the tables show, SHS experiences a substantial slowdown for every algorithm. The reasons for this are two-fold: first, SHS runs in client-server paradigm where the data resides in the servers and the computation happens in the client. Therefore, increasing the number of servers only scales up the available storage, but not the available computing resources, since the algorithms run on the client machine; On the other hand, both DryadLINQ and SQL Server PDW admit data-parallel computation and therefore scale up gracefully on both the data and computation as more servers are added to the system. We note that DryadLINQ is optimized for streaming access of data and this model is especially useful for computations like ASP and WCC where one needs to access all the nodes in the graph. However, this advantage is not present in the case of SALSA and to a lesser extent SCC where the access to nodes is mostly random. We would also like to point out that in the case of the single server, SHS also benefits from having the client and server running on the same machine.

Another key measure of effectiveness of an algorithm is its scalability. In our case, the key question to ask is *how does each of the platforms perform when the size of the graph increases*? Table 3 shows similar running times of the same programs processing the ClueWeb09 Category A graph using sixteen servers. We note that this graph has an order-of-magnitude more nodes than the ClueWeb09 Category B graph. While both DryadLINQ and SQL Server PDW scale gracefully on the larger data set, SHS suffers from the bottleneck introduced by having the computation performed in the client even as it scales well in terms of using more servers to handle the larger data. We discuss the scalability of each platform in more detail as we describe the implementation of each algorithm in the following sections.

Performance aside, we are also interested in how elegant and concise each implementation is. A quantitative measure of conciseness is the number of lines of code, which we report in Table 4. For the first algorithm (PageRank), we discuss the code in detail to give a flavor of implementations in each of the systems.

Algorithm	SQL Server PDW	DryadLINQ	SHS
PageRank	122,305	83,472	63,771
SALSA	5,873	4,843	37
SCC	1,147	3,243	94,346/816
WCC	63,972	74,359	1,801
ASP	138,911	175,839	77,214

Table 1: Running time in seconds for each algorithm implementation in each system, processing the Category B graph on one server.

Algorithm	SQL Server PDW	DryadLINQ	SHS
PageRank	8,970	4,513	90,942
SALSA	2,034	439	163
SCC	475	446	214,858/1,073
WCC	4,207	3,844	1,976
ASP	30,379	17,089	246,944

Table 2: Running time in seconds for each algorithm implementation in each system, processing the Category B graph on sixteen servers.

4.1 PageRank

PageRank [28] (Algorithm 1) computes a query-independent score of web pages based on endorsement (in the form of links) of each web page by other web pages. It is well-suited for data-parallel architectures, since links and one score vector can be easily streamed.

SQL Server PDW

The T-SQL implementation uses three temporary tables, for current (*score_cur*) and previous (*score_prev*) scores and the count of outlinks (*link_counts*) for each node. Pre-computing the outlink counts provides a straightforward way to normalize the score being passed from a source node to its destination outlinks. The *link_counts* table is sparse, meaning that only nodes with non-zero out-degree are present. In each iteration, the scores in *score_prev* are passed on to outlinks, and new rank is provided via the random jump probability as the two queries concatenated by the `UNION ALL` statement. The results are grouped by the node-id and summed to populate the *score_cur* table. At the end of each iteration, the *score_prev* table is dropped and the *score_cur* table is renamed to *score_prev*.

DryadLINQ

The DryadLINQ implementation creates a distributed stream of $\langle$*page-id*, *link-ids*$\rangle$ for each node in the graph and a distributed stream of $\langle$*id*, *score*$\rangle$ pairs such that each score is $1/n$ and a pair with a given *id* resides on the same machine as the link entry with the matching *page-id*. Each PageRank iteration is performed in parallel using a `join` of these two streams on *page-id* and *id*. All tuples with the same key u are aggregated in the machine responsible for u and then the join on u is locally computed on this machine. Denoting the adjacency list of u by $links[u]$, the output of the join is a new stream of $\langle v, score[u]/|links[u]|\rangle$ pairs for each node v in $links[u]$. Then all pairs in the new stream about the same node v are aggregated using a `group by` operation (so that the probability contributions from every in-link to v are grouped together) and the score for v is updated. At the end of each iteration, the scores are written to disks across the cluster. Note that the PageRank implementation makes full use of data-parallel computation in DryadLINQ.

Algorithm	SQL Server PDW	DryadLINQ	SHS
PageRank	156,982	68,791	836,445
SALSA	2,199	2,211	124
SCC	7,306	6,294	-/15,903
WCC	214,479	160,168	26,210
ASP	671,142	749,016	2,381,278

Table 3: Running time in seconds for each algorithm implementation in each system, processing the Category A graph on sixteen servers.

Algorithm	SQL Server PDW	DryadLINQ	SHS
PageRank	34	23	38
SALSA	116	165	77
SCC	-/247	-/118	66/174
WCC	86	40	181
ASP	78	144	108

Table 4: Number of lines of code for each algorithm implementation in each system.

SHS

The SHS implementation performs batched invocations to the server to amortize network latency. During each iteration, we enumerate the UIDs in a fixed order using the C# enumerator `Uids` and create batches of UIDs using a `UidBatch` object. Whenever the batch becomes full or the last UID is reached, we obtain the forward links for all UIDs in the batch using the method `BatchedGetLinks`. We then retrieve the scores corresponding to these links from the servers, update them appropriately, and send them back to the servers. After going through all the UIDs, the scores are retrieved from the servers and written to a file on the client's disk, which is then read in the next iteration in a streaming fashion. Writing to the disk avoids having to maintain two sets of scores in main memory.

Performance Evaluation

For PageRank computation, we observed that on a single server, SHS outperforms both DryadLINQ and SQL Server PDW. However, when either the number of machines or the data is increased, the other two platforms take advantage of the increase in storage and compute power and perform significantly better than SHS. The fact that PageRank is highly parallelizable is the main reason behind the improvement.

As can be seen from the code snippets for PageRank and from Table 4, all the algorithms could be implemented using a few lines of code on all the platforms. We notice that for each system, the algorithms that are most natural to that paradigm take the fewest lines of code to implement. In the case of SQL Server PDW, each row of the edges table is complete – that is, the source and destination node IDs are both represented. This creates both opportunity and limitations for the programmer. For example, finding edges whose source node ID is greater than the destination node ID can be expressed as `SELECT src, dest FROM edges WHERE src > dest`. However, we need to keep the context explicit in the query, as the output schema is fixed. The programmer cannot query for a list of destination IDs for a given source ID when the list is of variable length depending upon the source ID. In the above example, it would be space-efficient to write `SELECT src, listOfDest FROM edges WHERE src > dest`. Some relational systems have been extended with arrays and/or nested relations to provide support for such operations, but code using these extensions is not portable.

4.2 SALSA

The Stochastic Approach to Link-Sensitivity Analysis (SALSA) [20] is a query-dependent link-based ranking algorithm inspired by HITS [19] and PageRank [28]. SALSA computes an authority score for each node in a query-specific neighborhood graph, by performing a random walk on the neighborhood graph (Algorithm 2). In the computation of this neighborhood graph, $C_t(X)$ refers to a consistent unbiased sample of t elements from set X, as described in [25]. We used consistent sampling parameters, $a = b = 5$. The random walk on the neighborhood graph commences on a node with in-degree greater than 0. Each step in the walk consists of following an edge in the backward direction and then following an edge in the forward direction. The stationary probability distribution of the random walk is returned as the authority score vector s. We next describe the implementations in the three systems.

Algorithm 1 PAGERANK(G)

Input: A directed graph $G = (V, E)$ with n nodes, random jump probability d, number of iterations z.
Output: A vector s of stationary probabilities for the nodes in G.

```
 1: for each $u \in V$ do
 2:   $s[u] := 1/n, s'[u] := 0$
 3: for $k = 1$ to $z$ do
 4:   for each $u \in V$ do
 5:     $links[u] := \{v | (u, v) \in E\}$
 6:     for each $v \in links[u]$ do
 7:       $s'[v] := s'[v] + \frac{s[u]}{|links[u]|}$
 8:   for each $u \in V$ do
 9:     $s[u] := d/n + (1-d)s'[u]$
10:     $s'[u] := 0$
```

```
public class ShsPageRank {
  private static string Name(int i) { return "scores-" + i; }
  public static void Main(string[] args) {
    Shs.ShsClerk shs = new Shs.ShsClerk(args[0]);
    double d = double.Parse(args[1]);
    long n = shs.NumUrls();
    var ob = new OutBuf(Name(0));
    for (long i = 0; i < n; i++) ob.WriteDouble(1.0 / n);
    ob.Close();
    var scores = shs.AllocateUidState<double>();
    var uidBatch = new Shs.UidBatch(50000);
    for (int k = 0; k < 100; k++) {
      scores.SetAll(x => d / n);
      var ib = new InBuf(Name(k));
      foreach (long u in shs.Uids) {
        uidBatch.Add(u);
        if (uidBatch.Full || shs.IsLastUid(u)) {
          var linkBatch = shs.BatchedGetLinks(true, uidBatch);
          var linkMap = new Shs.UidMap(linkBatch);
          var scoreArr = scores.GetMany(linkMap);
          foreach (var links in linkBatch) {
            double f = (1.0 - d) * ib.ReadDouble() / links.Length;
            foreach (long link in links) {
              scoreArr[linkMap[link]] += f;
            }
          }
          scores.SetMany(linkMap, scoreArr);
          uidBatch.Reset();
        }
      }
      ib.Close();
      ob = new OutBuf(Name(k+1));
      foreach (var us in scores.GetAll()) ob.WriteDouble(us.val);
      ob.Close();
      System.IO.File.Delete(Name(k));
    }
  }
}
```

SHS implementation of PageRank

```
CREATE TABLE link_counts WITH (DISTRIBUTION=HASH(id)) AS
      SELECT src AS id, COUNT(dest) AS cnt
      FROM edges GROUP BY src;
Declare @CNODES bigint = (SELECT COUNT_BIG(*) FROM nodes);
CREATE TABLE score_prev WITH (DISTRIBUTION=HASH(id)) AS
      SELECT n.id, 1.0 / @CNODES AS score FROM nodes n;
Declare @ITER int = 0, @d float = 0.15;
While @ITER < 100
BEGIN
      CREATE TABLE score_cur WITH (DISTRIBUTION=HASH(id)) AS
      SELECT s.curid AS id, SUM(s.newscore) AS score FROM
      (SELECT * FROM (
      SELECT e.dest AS curid,
            SUM((1.0-@d)*(sp.score/lc.cnt)) AS newscore
      FROM score_prev sp, edges e, link_counts lc
      WHERE sp.id = lc.id AND sp.id = e.src
      GROUP BY e.dest
      WITH (DISTRIBUTED_AGG)) A
      UNION ALL
      SELECT sp.id AS curid, (@d / @CNODES) AS newscore
      FROM score_prev sp
      WHERE sp.id <> @CNODES) s GROUP BY s.curid;

      INSERT INTO score_cur
      SELECT CAST(@CNODES AS bigint) AS id,
      SUM(CASE WHEN lc.cnt IS NULL
          THEN sp.score ELSE 0 END) * (1.0 - @d)
      FROM score_prev sp
           LEFT OUTER JOIN link_counts lc on sp.id = lc.id;

      DROP TABLE score_prev;
      RENAME OBJECT score_cur TO score_prev;
      SET @ITER += 1;
END
DROP TABLE link_counts; DROP TABLE score_prev;
```

SQL Server PDW implementation of PageRank

```
using LD = LinqToDryad.Pair<long, double>;
using GR = LinqToDryad.Pair<long, long[]>;

public class DryadLINQPageRank {
  public static IEnumerable<LD> InitScores(long n) {
    for (long i = 0; i < n; i++) yield return new LD(i, 1.0 / n);
  }
  static void Main(string[] args) {
    var pages = PartitionedTable.Get<GR>("tidyfs://cw-graph");
    long n = pages.LongCount();
    double d = double.Parse(args[0]);
    IQueryable<LD> scores = pages.Apply(x => InitScores(n))
         .HashPartition(x => x.Key, pages.PartitionCount)
         .ToPartitionedTable("tidyfs://scores-0");
    for (int i = 1; i <= 100; i++) {
      var scores1 = from p in pages
                    join s in scores on p.Key equals s.Key
                    from v in p.Value
                    select new LD(v, s.Value / p.Value.Length);
      scores = from s in scores1
               group s.Value by s.Key into g
               select new LD(g.Key, d / n + (1.0 - d) * g.Sum());
      scores.ToPartitionedTable("tidyfs://scores-" + i);
    }
  }
}
```

DryadLINQ implementation of PageRank

SQL Server PDW

The SALSA implementation for SQL Server PDW is a set-oriented implementation of the algorithm. For all input queries, we compute the graph based on the input URLs and random sampling of the edges, keeping track of which query-id the graph belongs to. We perform a random walk back and forth on the graphs 100 times and return the top input urls for each query according to the calculated score. Due to the interpreted nature of a T-SQL batch script in SQL

Algorithm 2 SALSA(G)

Input: A directed web graph $G = (V,E)$, a result set $R \subseteq V$ of URLs that satisfy a query q, consistent sampling parameters a, b.
Output: A vector s of SALSA authority scores for each URL in a query-specific neighborhood graph.

```
 1: $V_R := \bigcup_{r\in R}\{r\} \cup C_a(\{u|(u,r)\in E\}) \cup C_b(\{v|(r,v)\in E\})$
 2: $E_R := \{(u,v)\in E | u\in V_R \wedge v\in V_R\}$
 3: $V_R^A := \{u\in V_R | in(u) > 0\}$
 4: for each $u \in V_R$ do
 5:     $s[u] := \frac{1}{|V_R^A|}$ if $u\in V_R^A$; 0 otherwise
 6: repeat
 7:     for each $u \in V_R^A$ do
 8:         $s'[u] := \sum_{(v,u)\in E_R} \sum_{(u,w)\in E_R} \frac{s[w]}{out(v)\cdot in(w)}$
 9:     for each $u \in V_R^A$ do
10:         $s[u] := s'[u]$
11: until $s$ converges
```

Server PDW, each short random walk query is re-compiled and optimized. This adds some overhead to the total query execution time, but the overhead is amortized across all input queries.

The program performs consistent sampling by joining the edges table to a table containing a pre-computed mapping of node ids to random numbers, which are obtained in ascending order using a `ROW_NUMBER() OVER()` analytic function, and only the rows with a row number less than 5 are included in the graph.

DryadLINQ

The DryadLINQ implementation computes both the neighborhood graph and the SALSA authority scores in a distributed data-parallel fashion. First, a distributed stream of $\langle$*page-id*, *link-ids*$\rangle$ corresponding to graph G is created and then streams of forward and backward edges are generated from this stream. Similarly a distributed stream R_s of $\langle$*query-id*,*page-id*$\rangle$ pairs is generated by joining the $\langle$*query-id*, *URL*$\rangle$ dataset with the given URL/id mapping. The stream R_s is joined with streams of forward and backward edges and then sampled to generate the SALSA neighborhood set V_R associated with each query (as a stream). This neighborhood set stream is then joined with the forward graph stream to obtain the neighborhood graph stream for each query (a stream of $\langle$*query-id*,*page-id*,*link-id*$\rangle$ tuples). Using a series of distributed joins, we aggregate the relevant information for each query (query-id, result set R, neighborhood set V_R, and edge set E_R) at one cluster node and compute the SALSA authority scores locally on that node. Finally, we map the scores to the URLs by joining the stream of computed SALSA scores with the URL/id mapping. The implementation benefits tremendously from DryadLINQ's ability to perform distributed joins over data streams.

SHS

The SHS implementation computes the neighborhood graph by communicating with the SHS store and then performs the random walk locally. The neighborhood graph corresponding to the given set of result URLs is computed as follows. The set of result URLs is first converted to UIDs (using one invocation of `BatchedUrlToUid`). Then `BatchedSampleLinks` is called twice to obtain a random sample of forward and backward links for each URL in the result set. As the method `BatchedSampleLinks` performs sampling at the server end, the network overhead is kept to a minimum. Finally the set of URLs pointed to by the vertices in the neighborhood graph is obtained (using one invocation of `BatchedGetLinks`) for use in the remaining local computations. Thus batching over many UIDs (or URLs) helps to significantly reduce the RPC overhead.

Algorithm 3 SCC(G)

Input: A directed graph $G = (V,E)$.
Output: A list of SCCs where each SCC is represented as a set of nodes belonging to the SCC.

```
 1: for each $u \in V$ do
 2:     $fwdVisited[u] := false, fwdParent[u] := Nil$
 3: $time := 0$
 4: for each $u \in V$ do
 5:     if $fwdVisited[u] = false$ then
 6:         DFS-VISIT($u, E, fwdVisited, fwdParent, f$)
 7: $E^T := \{(v,u)|(u,v)\in E\}$
 8: for each $u \in V$ do
 9:     $bwdVisited[u] := false, bwdParent[u] := Nil$
10: for each $u \in V$ in decreasing order of $f[u]$ do
11:     if $bwdVisited[u] = false$ then
12:         DFS-VISIT($u, E^T, bwdVisited, bwdParent, b$)
13: Output the vertices of each tree in the backward depth-first forest as a
    separate strongly connected component.
```

Algorithm 3 DFS-VISIT($u, E, visited, parent, f$)

```
    $time := time + 1$
    $links[u] := \{v|(u,v)\in E\}$
    for each $v \in links[u]$ do
        if $visited[v] = false$ then
            $parent[v] := u$
            DFS-VISIT($v, E, visited, parent, f$)
    $visited[u] := true,\ f[u] := time$
```

Performance Evaluation

We observe that SALSA is inherently based on random access to the node information. As noted earlier, SHS is very suitable for such applications. In fact, as both tables show SHS does indeed perform as much as 160 times better than both DryadLINQ and SQL Server PDW. SHS does not benefit from the increase in the number of servers from one to sixteen, while the other two platforms exploit the additional storage and compute power to their benefit. The performance difference between SQL Server PDW and DryadLINQ can be attributed to DryadLINQ performing the power iteration on a single machine instead of distributed over the cluster, whereas SQL Server PDW has to persist the result after each iteration to distributed storage.

4.3 Strongly Connected Components

We next consider the problem of decomposing a directed graph into its strongly connected components [36, 4]. This problem is important since many problems on large directed graphs require such a decomposition. Typically an original problem can be divided into subproblems, one for each strongly connected component.

The standard linear-time algorithm for computing strongly connected components in a directed graph (Algorithm 3) uses two depth-first searches, one on the graph and one on the transpose graph. However, depth-first search is highly sequential and hence this algorithm is not suitable for execution on very large graphs. We instead implement a "hybrid" version which copies a pruned version of the graph locally and runs Algorithm 3 locally. The graph is pruned as follows: All nodes with no outlinks (sinks) or no inlinks (sources) are guaranteed to be singleton SCCs and hence we rule out these nodes and their incident edges. As most of the nodes in a large web graph tend to be sinks, we get a substantial reduction in the graph size so that the new graph fits into memory and DFS computations can be performed locally. However this approach does not scale with the size of the graph. We briefly describe the SCC implementations in the three systems. We also implement the original version of Algorithm 3 on SHS.

SQL Server PDW

We describe the SQL Server PDW implementation of SCC using the hybrid version. First, SQL Server PDW is used to filter the nodes and edges to remove nodes that cannot possibly end up in an SCC larger than size 1. By definition, nodes in a multi-node SCC must have at least one outlink and at least one inlink other than a reflexive edge. The filtered node list comes from intersecting the src and dest sets in the edges table. To aid in the computation, a mapping function is constructed to a new, dense ID space. The edges are then filtered and mapped into the new ID space. Given the reduced size of the nodes and edges table, both are read out of SQL Server PDW into main memory where Algorithm 3 is performed. The edges are read into adjacency arrays twice, first ordered by src (forward direction), then by dest (backward direction).

DryadLINQ

We next describe the DryadLINQ implementation of the hybrid version of Algorithm 3. First we create a distributed stream of $\langle$*page-id*, *link-ids*$\rangle$ for each node in the graph and obtain the number of nodes using a `count` operation. We then create a stream of $\langle$*source-id*, *destination-id*$\rangle$ for edges in the graph, from which we generate streams of non-sink nodes (nodes with non-zero out-degree) and non-source nodes (nodes with non-zero in-degree) using `select` and `distinct` operations. Intersecting these two streams provides a stream of nodes in the pruned graph, which we then map to a new, dense ID space. Then we stream through the distributed stream of edges, loading the forward graph corresponding to the above nodes into main memory of the client machine, and locally perform forward DFS. Then we similarly load the backward graph into main memory and perform backward DFS locally. Finally the SCCs (including the singleton SCCs) are written to disk.

SHS

The SHS implementation of the original version of Algorithm 3 involves a clever implementation of depth-first search to handle the large graph size. As either out-links or in-links can be retrieved for each node, we do not have to compute the transpose graph G^T explicitly. In order to maintain the nodes in the order of their finishing times in the first DFS, we implement a custom stack that is optimized for disk access. Due to the sequential nature of depth-first search, we do not parallelize (batch) accesses and hence incur a significant network overhead.

We also implemented the hybrid version of Algorithm 3. We perform `BatchedGetDegree` requests to the store to obtain forward and backward degrees for all nodes, thereby determining nodes guaranteed to be singleton SCCs. We map the remaining nodes to a new, dense ID space. Then we load the forward graph into user space (main memory of the client machine) using a series of `BatchedGetLinks` requests, and perform forward DFS locally. Then we load the backward graph into user space (reusing memory) and perform backward DFS locally. While this approach improves the performance by about two orders of magnitude due to the reduced network overhead, it is inherently non-scalable.

Performance Evaluation

The performance of the non-hybrid SHS implementation of SCC is largely constrained by the exploration of one node at a time in DFS, which requires one roundtrip to the server per node in the DFS.

For the hybrid version, SHS performs best for the one server Category B experiment, but falls behind SQL Server PDW and DryadLINQ in both of the sixteen node experiments. The SCC implementation for all three platforms under study is identical, with the exception of the first phase. At the beginning, each implementation must prune out nodes that will be in singleton SCCs and remap the remaining node ids into a dense id space that starts with zero. The edges containing references to these nodes are then rewritten to use the mapped node ids as well. This allows a very compact in-memory representation – for example, even if the original node ids required an eight byte integer representation, the mapped ids can utilize a four byte integer. In the case of SQL Server PDW and DryadLINQ, this mapping effort can be done completely inside the cluster and only the mapped and pruned edges need be pulled out to the client SCC memory. By contrast, the SHS implementation must pull the forward and backward link counts for every node and build the id mapping in the client. It further must pull all of the links for any node found to have nonzero in and out degree, and is not able to prune the edges for those nodes on the server side.

Algorithm 4 WCC(G)

Input: A directed graph $G = (V, E)$ on n nodes, where $V = \{1, \ldots, n\}$ and $E \subseteq V \times V$.
Output: A mapping rep where all nodes in the same WCC map to the same representative element.

```
 1: for each $u \in V$ do
 2:     $rep[u] := u$
 3: repeat
 4:     $progress := false$
 5:     for each $u \in V$ do
 6:         $links[u] := \{v | (u, v) \in E\}$
 7:         $minid := rep[u]$
 8:         for each $v \in links[u]$ do
 9:             if $rep[v] \neq minid$ then
10:                 $minid := \min(minid, rep[v])$
11:                 $progress := true$
12:         $rep[u] := minid$
13:         for each $v \in links[u]$ do
14:             $rep[v] := minid$
15: until $\neg progress$
```

4.4 Weakly Connected Components

A related problem is to compute the weakly connected components in a directed graph. Determining the connected components and their sizes is crucial to understanding the network structure of massive graphs. Moreover other graph computations such as minimum spanning tree or approximate shortest path are defined only over individual components.

Weakly connected components can be computed by performing depth-first search or breadth-first search on the underlying undirected graph. Our algorithm uses the latter since all nodes in the BFS frontier can be explored in parallel. Algorithm 4 has resemblance to the algorithm for computing WCC using disjoint-set operations [11]. Initially each node is in its own connected component (with component value equal to itself). We then iterate until no component mapping gets updated. In each iteration, for each node u, we check whether any of its neighbors is mapped to a different component and if so, update the node and its neighbors to map to the smallest component value amongst them.

SQL Server PDW

The SQL Server PDW implementation of WCC starts by precomputing the “flattened graph” – that is, the union of the directed edge set with the set of edges reversed. To speed computation, all nodes not present in the flattened graph are removed from further processing since they are guaranteed to be in singleton WCCs. The propagation of the *minid* in each loop is done as a join of the current *id*-to-*minid* table with the flattened graph, UNIONed with the *id*-to-*minid* table. We perform a GROUP BY *id* with a MIN() function applied to the *minids* from that union-ed set. The loop continues until the *minid* for all *ids* does not change across an iteration.

DryadLINQ

The DryadLINQ implementation of WCC generates a distributed stream E of graph edges from the input graph and maintains a distributed stream C of $\langle node, componentID \rangle$ pairs. C is initialized with the identity mapping so that each node is in its own component. Then the components are merged by growing balls around the nodes and checking whether nodes in intersecting balls are assigned to the same component. This "growing" step is implemented as follows. Let C' denote the stream obtained by joining C with E ($C' = \{(y, id) | (x, id) \in C \wedge (x, y) \in E\}$). We compute the union $C' \cup C$ and then generate the stream with the new component mapping, $C_{new} = \{(x, minid) | minid = min(id : (x, id) \in C \cup C')\}$ using a `group by` operation. When all the components have been assigned correctly, C_{new} will be identical to C. If C_{new} is different from C, we update C to C_{new} and iterate. As the component ID for any node can only decrease, it is sufficient to check whether the sum of component IDs over all nodes has changed.

SHS

The SHS implementation of WCC is based on disjoint-set operations [11]. The union-find forest is represented as a distributed `UidState` array, which can be updated in constant time, and performance is further improved through batching.

Performance Evaluation

We observe that SHS dramatically outperforms SQL Server PDW and DryadLINQ, due to its ability to express the union-find algorithm. As described earlier, DryadLINQ is limited by the overhead associated with evaluating the loop termination. To amortize this overhead, the DryadLINQ program performs partial loop unrolling, checking the termination condition every five iterations. The SQL Server PDW implementation does the loop termination check on every iteration, and therefore suffers from an overhead that the DryadLINQ implementation does not.

4.5 Approximate Shortest Path

We next consider the problem of finding shortest paths between pairs of nodes in a large graph, which is a fundamental operation with many applications. However, for web-scale graphs, computation of exact shortest paths is challenging since these graphs do not fit in main memory on a single machine. Even in a distributed setting, the computation would require huge amounts of communication. Furthermore, in a real-time application, we would like to compute shortest paths with limited memory and processing resources. Hence we are interested in computing approximate shortest paths between nodes.

We use the algorithm proposed in [13]. The essential idea behind that algorithm is the following: In an offline computation, sample a small number of sets of nodes in the graph (sets of seed nodes). Then, for each node in the graph, find the closest seed in each of these seed sets. The sketch for a node is composed of the closest seeds, and the distance to these closest seeds. The offline steps are described in Algorithm 5. Then, in an online step, the sketches can be used to estimate the distance between any pair of nodes (as described in [13]). As the most important component of computation of the approximate shortest paths is computation of the offline sketch for all nodes u, we perform our evaluations on this step, treating the graph as undirected.

Algorithm 5 OFFLINE-SKETCH(G)

Input: An undirected graph $G = (V, E)$.
Output: Vectors of nearest seeds $seed[u]$ and their distances $dist[u]$ for each node u.

```
 1: $C := \{u \in V | degree(u) \geq 2\}$
 2: $I := \{0, 1, \ldots, \lfloor log|C| \rfloor\}$
 3: for each $i \in I$ do
 4:     $S_i :=$ Random sample of $2^i$ elements of $C$
 5: for each $u \in V$ and each $i \in I$ do
 6:     $seed[u][i] := u$
 7:     $dist[u][i] := 0$ if $u \in S_i$; $\infty$ otherwise
 8: repeat
 9:     $progress := false$
10:     for each $(u, v) \in E$ and each $i \in I$ do
11:         if $dist[u][i] < dist[v][i]$ then
12:             $seed[v][i] := seed[u][i]$
13:             $dist[v][i] := dist[u][i] + 1$
14:             $progress := true$
15: until $\neg progress$
```

SQL Server PDW

We have implemented ASP in T-SQL in an iterative manner. For each chosen seed set size, the next BFS search is computed in a single query for all seeds. When a given node is encountered along multiple paths originating from different seeds, the seed with the minimum ID value is chosen as the seed. Beyond this conversion, the algorithm is implemented in a straightforward manner.

DryadLINQ

The DryadLINQ implementation uses the input stream of graph edges to compute a stream of (undirected) degrees of nodes using a `group by` operation, and then obtains a stream of candidate nodes C by removing nodes with degree < 2. It then determines the count of C to to set $r = \lfloor \log |C| \rfloor$ and also to set the initial value of the number of nodes reachable from all the sampled sets. Next it samples sets of sizes, $1, 2, 2^2, 2^3, \ldots, 2^r$ by generating a stream of $\langle node, initial\text{-}sketch \rangle$ pairs, where for each node u, the corresponding *initial-sketch* represents membership of u in the r sampled sets. DryadLINQ's flexibility allows us to use a single series of joins to process all of the seed sets, thereby performing the BFSs for the $\log n$ seed sets in parallel. For each sampled set S, in parallel, we grow a ball around S and thereby compute distances of all other nodes to S. At the end of each iteration, we check whether the number of nodes reachable from all the sampled sets increases (using a distributed sum operation) and if not, terminate the loop.

SHS

First we compute the candidate set C by probing degrees of all nodes and ruling out nodes with degree < 2 and represent C using a bit vector. This step is batched to reduce network overhead. Then we sample sets of different sizes $(1, 2, 2^2, 2^3, \ldots)$ uniformly at random from the candidate set of vertices. For each sampled set S, we compute distances of all other nodes to S by growing a ball around S, until no new nodes are added to the ball (that is, when distances to all nodes from S have been computed). This step is also performed in batches.

Performance Evaluation

In this case, the trade-off in using SHS is in memory vs. roundtrips. The bulk of the relatively large cost of computing the approximate shortest paths for SHS can be attributed to computing the BFS one dimension at a time. This constraint arises because the entire sketch for all the nodes cannot be stored in memory on a single machine. Such a constraint does not exist in the case of DryadLINQ where the sketch is stored on disk and read and written in a streaming fashion, and all the dimensions can therefore be handled concurrently. This explains the relative speedup of DryadLINQ compared to SHS. The SQL Server PDW implementation processes one seed set at a time and therefore pays a sequential processing penalty that the DryadLINQ implementation avoids.

5. DISCUSSION AND CONCLUSION

This study compared the efficiency of three paradigms of computation on very large graphs used in real-world applications. We explained, using five basic graph algorithms, the effectiveness of each paradigm. The graph algorithms were chosen in a way to favor different data and computation models. For example, the efficiency of SALSA hinges on the random access to node data while ASP does not have such a restriction. In fact, ASP allows a natural streaming data model and data-parallel computation on this model. We conclude that SHS is a good platform choice when random access to the data is involved and relatively little computation is required (e.g. SALSA); DryadLINQ is a good choice for data-parallel computations on streaming data (e.g. PageRank); and is appropriate for scenarios where indices can be leveraged or transactions are required. DryadLINQ provides richer data types (e.g. arrays) and a more expressive language (C#) to programmers than SQL Server PDW does with T-SQL; but this flexibility comes at a price: programmers *can* optimize DryadLINQ programs by choosing the number of stream partitions or annotating user-defined operators with hints that trigger partial aggregation, but on the flip-side they *have to* perform these optimizations to extract the best performance from the system. SQL Server PDW, on the other hand, is much more of black box: programmers write their queries in a completely declarative fashion, and the system generates what it believes is the best query plan.

6. REFERENCES

[1] D. Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Data Engineering Bulletin*, 32(1):3–12, 2009.

[2] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column oriented database systems. In *VLDB*, 2009.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *VLDB*, 2009.

[4] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[5] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The connectivity server: Fast access to linkage information on the web. In *WWW*, 1998.

[6] A. Bialecki, M. Cafarella, D. Cutting, and O. O-Malley. Hadoop: A framework for running applications on large clusters built of commodity hardware, 2005. `http://lucene.apache.org/hadoop/`.

[7] P. Boldi and S. Vigna. The webgraph framework I: Compression techniques. In *WWW*, 2004.

[8] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *WWW*, 2000.

[9] M. Castellanos, U. Dayal, and T. Sellis. Beyond conventional data warehousing–massively parallel data processing with greenplum database. In *BIRTE*, 2008.

[10] B. V. Cherkassky, L. Georgiadis, A. V. Goldberg, R. E. Tarjan, and R. F. F. Werneck. Shortest path feasibility algorithms: An experimental evaluation. In *ALENEX*, pages 118–132, 2008.

[11] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. The MIT press, 2001.

[12] N. Craswell and M. Szummer. Random walks on the click graph. In *SIGIR*, pages 239–246, 2007.

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

[14] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[15] J. Dean and S. Ghemawat. MapReduce: A flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[16] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005.

[17] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the data-stream model. *SIAM J. Comput.*, 38(5):1709–1727, 2008.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[19] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *SODA*, 1998.

[20] R. Lempel and S. Moran. The stochastic approach for link-structure analysis (SALSA) and the TKC effect. *Computer Networks*, 33(1-6):387–401, 2000.

[21] M. W. Mahoney, L.-H. Lim, and G. E. Carlsson. Algorithmic and statistical challenges in modern large-scale data analysis are the focus of MMDS 2008. *CoRR*, abs/0812.3702, 2008.

[22] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, 2010.

[23] Microsoft SQL Server 2008 R2 Parallel Data Warehouse. `http://www.microsoft.com/sqlserver/en/us/solutions-technologies/data-warehousing/pdw.aspx`.

[24] M. Najork. The scalable hyperlink store. In *HT*, 2009.

[25] M. Najork, S. Gollapudi, and R. Panigrahy. Less is more: Sampling the neighborhood graph makes SALSA better and faster. In *WSDM*, 2009.

[26] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4):680–710, 1984.

[27] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[28] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[29] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparative study of approaches to cluster-based large scale data analysis. In *SIGMOD*, 2009.

[30] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[31] K. Randall, R. Stata, J. Wiener, and R. Wickremesinghe. The link database: Fast access to graphs of the web. In *DCC*, 2002.

[32] S. D. Servetto and G. Barrenechea. Constrained random walks on random graphs: Routing algorithms for large scale wireless sensor networks. In *WSNA*, 2002.

[33] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A column-oriented DBMS. In *VLDB*, 2005.

[34] M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: Friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[35] T. Suel and J. Yuan. Compressing the graph structure of the web. In *DCC*, 2001.

[36] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[37] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a Map-Reduce framework. In *VLDB*, 2009.

[38] Wikipedia. Graph database. `http://en.wikipedia.org/wiki/Graph_database`.

[39] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.