An Evaluation of Alternative Physical Graph Data Designs for Processing Interactive Social Networking Actions

Shahram Ghandeharizadeh(✉), Reihane Boghrati, and Sumita Barahmand

Department of Computer Science, University of Southern California, Los Angeles, CA 90089, USA
shahram@dblab.usc.edu

Abstract. This study quantifies the tradeoff associated with alternative physical representations of a social graph for processing interactive social networking actions. We conduct this evaluation using a graph data store named Neo4j deployed in a client-server (REST) architecture using the BG benchmark. In addition to the average response time of a design, we quantify its SoAR defined as the highest observed throughput given the following service level agreement: 95 % of actions to observe a response time of 100 ms or faster. For an action such as computing the shortest distance between two members, we observe a tradeoff between speed and accuracy of the computed result. With this action, a relational data design provides a significantly faster response time than a graph design. The graph designs provide a higher SoAR than a relational one when the social graph includes large member profile images stored in the data store.

1 Introduction

A graph database provides an intuitive representation of a social graph. It supports vertices that may represent members and edges that may represent a relationship such as friendship between two members. Queries may filter vertices of interest and navigate edges to retrieve relevant data. Updates may insert and delete a vertex, add and remove edges between vertices, and change the property value of edges and vertices. Facebook's TAO [1] is an example graph data store that serves a social graph to hundreds of millions of users on a daily basis.

One may represent a social graph using different physical graph representations. To illustrate, consider the friendship relationship between two members A and B. It may start with one member, say Member A, extending a friend invitation to Member B. And, Member B accepting this invitation. Two physical representations, termed *Labeled* and *Distinct*, are as follows. With Labeled, the friendship edge between Member A and B is assigned a value to identify it as a friendship invitation. Once Member B accepts A's invitation, the value of this edge changes to denote a confirmed friendship. With Distinct, there are two types of edges, one for a pending friend invitation and a second for a confirmed friendship. When Member B accepts A's invitation, the system deletes the edge corresponding to the friend invitation and creates a confirmed friendship edge between them. This design creates and deletes edges more frequently than the Labeled design.

A research topic is what are the tradeoff associated with these alternative designs for different workloads? And, how do they compare with data stores that implement a different data model such as relational database management systems (RDBMSs)? To investigate these research topics, we had a choice of benchmarks including BG [6,7,14], LinkBench [4], LDBC [2,11], or a micro-benchmark such as [3,16]. After a careful analysis, we decided to use BG for two reasons. First, BG is a stateful benchmark that quantifies both the average response time of a data store and its throughput given a pre-specified service level agreement (SLA). The latter is termed Social Action Rating, SoAR [7], and is similar to the tps rating[1] defined by the TPC-C benchmark [12,15]. As reported in Sect. 4, an RDBMS may provide an average response time that is faster than Neo4j for some actions while Neo4j outperforms the RDBMS when considering SoAR with certain database settings. Second, BG quantifies the amount of stale, inconsistent, or invalid data (collectively, termed *unpredictable data* [7,8]) produced by a data store. This is useful because certain social networking actions such as computing the shortest distance between two members may utilize heuristic search techniques that do not produce correct results, see discussions of Fig. 4 in Sect. 3.

The primary contribution of this study are two folds. First, it identifies four physical graph data designs for processing interactive social networking actions, see Fig. 3. Second, it evaluates these designs using the Neo4j [22] data store and the BG benchmark. This includes extensions of BG with the following three graph oriented actions: Get Shortest Distance, List Common Friends, and List Friends-of-Friends. The main findings of our evaluation are as follows. The Distinct physical graph design provides a superior performance when compared with the Labeled design. With the three new graph oriented actions, an industrial strength relational database management system (SQL-X) provides faster response times than Neo4j configured with a variant of the Distinct design named StoredDistinct (see description of Fig. 3 for details). One reason for this is the normalization guideline of the relational data model that represents a many-to-many friendship relationship as a table. This enables the graph oriented actions to fetch a smaller amount of data from a single table to provide faster response times. With a workload consisting of a mix of actions, SQL-X provides a higher SoAR than Neo4j when the social graph consists of no images. When large profile images are stored in SQL-X, Neo4j provides a higher SoAR than SQL-X.

The rest of this paper is organized as follows. We survey the related work in Sect. 2. Section 3 describes an implementation of the BG benchmark using Neo4j, detailing four physical graph data designs and their performance characteristics for different mix of actions. Section 4 quantifies the tradeoffs associated with a graph and a relational data design. Our future research directions are contained in Sect. 5.

[1] SoAR is different than tps in that the SLA can be changed depending on the requirements of an application while TPC-C's specified SLA is fixed.

2 Related Work

Evaluation of graph data stores has been a subject of active research during the past few years. The average response time of different actions of a microbenchmark is presented in [3] to compare two graph databases (Neo4j and Dex) with a RDF store (RDF-3X) and two relational database management systems (PostgreSQL and Virtuoso). Similarly, in [16], the response time of several social networking actions is used to compare the performance of alternative graph query languages using Neo4j with Java Persistent API (JPA) using the MySQL relational database management system. Both studies consider Neo4j deployed in either embedded or a client-server (REST) mode.

This study is different than [3,16] along two dimensions. First, we focus on Neo4j Cypher REST to investigate the alternative physical designs of a social graph, see the taxonomy of Fig. 2 and its discussion in Sect. 3.1. Second, we use the BG benchmark to analyze both the average response time and SoAR of the different designs. This analysis includes both read and write actions. (Both [3,16] focus on read actions only.) A key findining is that a design that provides a high performance with infrequent write actions may not perform well when the frequency of write actions is higher, see Table 4 and its discussion in Sect. 3.2. A novel feature of BG is its ability to quantify the amount of erroneous data produced by a data store. We use this capability of BG to show that one may trade performance for accuracy of results with an action such as Get Shortest Distance. To the best of our knowledge, these findings are novel and have not been presented else where.

3 BG Benchmark and Its Implementation Using Neo4j

Figure 1 shows the conceptual design of BG's social graph used for this evaluation. (See [6,7,9] for a comprehensive description of BG.) The Members entity set contains those users with a registered profile. It consists of a unique identifier and a fixed number of string attributes[2]. One may configure BG to create a social graph with or without images. In this paper, we consider both possibilities. With images, all experimental results are obtained using a social graph configured with a 2 KB thumbnail image and a 12 KB profile image. Thumbnail images are displayed when listing friends of a member and the higher resolution profile image is displayed when a member visits a profile. A member may extend a friend invitation to another member or be friends with a member, represented using "Invite" and "Friend" relationship sets, respectively. A resource may pertain to an image, a posted question, a technical manuscript, etc. These entities are captured in one set named "Resources". In order for a resource to exist, a member must "Own" that resource. A member may post a resource, say an image, on the profile of another member, represented as a "Posted on" relationship between two members and a resource. A member may comment on a resource. This is implemented using the "Manipulation" relationship set.

[2] The size of these attributes is configurable [6].

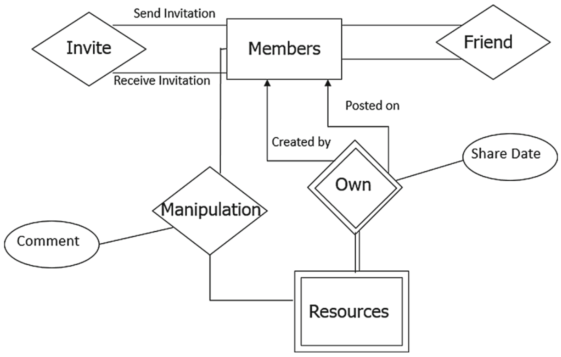


Fig. 1. BG benchmark's conceptual schema.

BG uses a closed emulation model to generate a workload of actions for a data store. With this model, a thread emulates a Member A who performs an action on another member or resource. This member who is performing the action is termed a *socialite*. A thread does not emulate another socialite until the pending action of the current socialite is processed. BG controls the load imposed on a data store by varying the number of threads used to emulate concurrent socialites performing actions, see [6,7] for details.

Figure 2 shows four different graph representations of this conceptual data model. We describe these alternatives when presenting the different actions that constitute the core of BG's workload. This discussion presents the average response time ($\overline{RT}$) and Social Action Rating (SoAR) of the alternative graph models using a single node Neo4j deployment. $\overline{RT}$ is quantified with BG emulating a single socialite issuing a mix of actions by issuing one action at a time. It is the average amount of time elapsed from when a socialite issues a request to the time Neo4j completes servicing the request. SoAR is the highest throughput observed with a service level agreement (SLA) that requires 95 % of actions to observe a response time of 100 ms or faster with no stale data.

The target hardware platform consists of two PCs connected using a Gigabit switch. Each PC consists of an i7-4770 processor, 16 GB of memory, one TB of disk storage, and a Gigabit networking card. The operating system of each PC is a 64 bit Windows 2012 Server. The version of Neo4j server is 2.0.1 and we used Neo4j's Cypher[3] query language to implement the Client that performs the interactive social networking actions (termed BGClient). All experiments assume a social graph consisting of 100,000 members with 100 friends per member (ϕ) and 100 resources per member (ρ).

We classify BG's actions into read and write. Below, we present them in turn.

[3] Cypher is a declarative language similar to SQL.

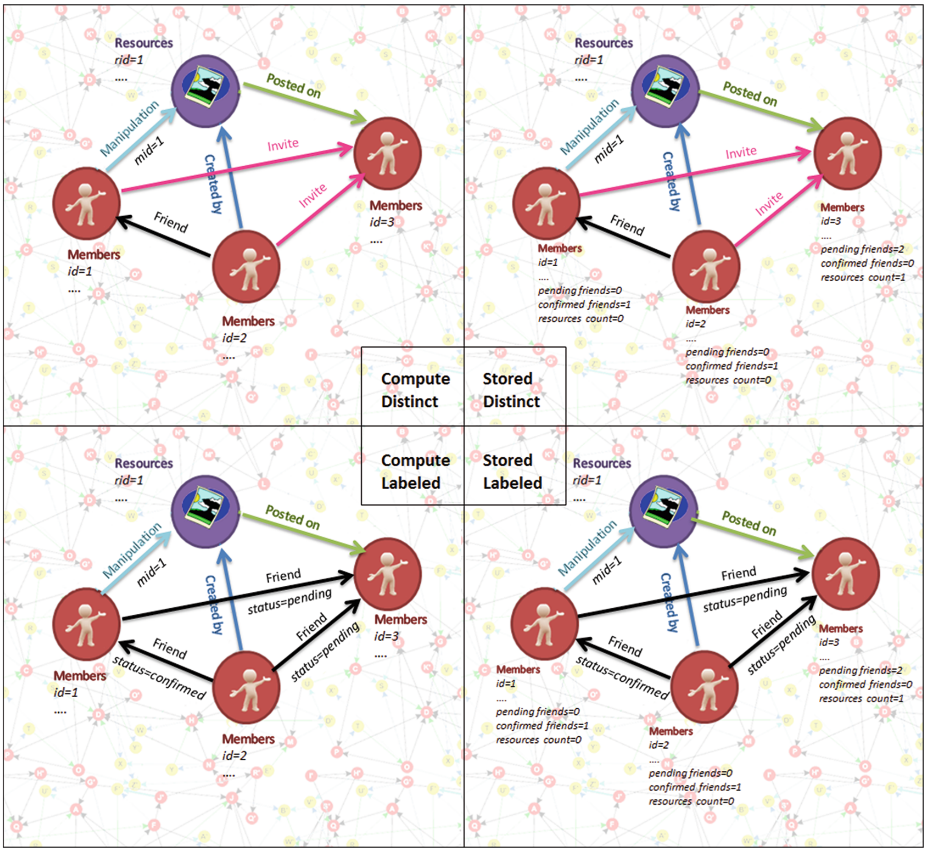


Fig. 2. Four physical graph representations of BG's database.

3.1 BG's Read Actions

BG's actions and their graph implementation are as follows. First, the **View Profile** (VP) action emulates a Socialite with member id A visiting the profile of a member with id U_r. BG generates A and U_r as input to VP. A may equal U_r, emulating a socialite referencing her own profile. The output of VP is the profile information of U_r, including U_r's attributes and the following two simple analytics: U_r's number of friends and number of posted resources on her wall. If the socialite is referencing her own profile (A equals U_r) then VP retrieves a third simple analytic, U_r's number of pending friend invitations.

The observed system performance with the VP action depends on the physical representation of the graph database. Figure 3 shows four different physical representations using a two dimensional quad, see also Fig. 2. The two dimensions correspond to the alternative representations of the simple analytics and friendship. One may implement the simple analytics using a Cypher query that computes the required value every time, see the first column of Table 2.

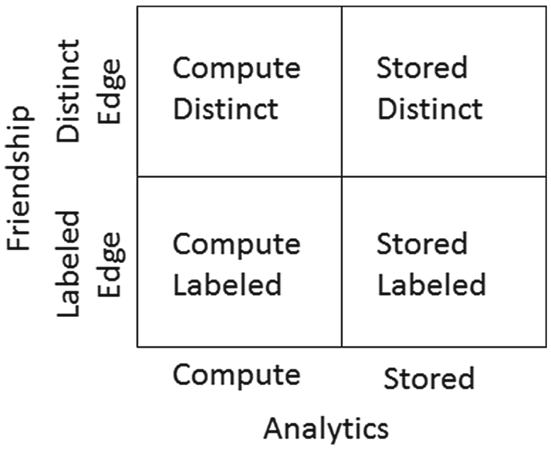


Fig. 3. Four physical graph designs.

Table 1. $\overline{RT}$, in milliseconds, for the alternative physical graph representations using Neo4j with a 100 K social graph, $\phi = 100$ friends per member, $\rho = 100$ resources per member.

	ComputeLabeled	ComputeDistinct	StoredLabeled	StoredDistinct
View Profile (VP)	308	93	12	8
List Friend (LF)	435	293	520	313

Alternatively, one may store the value of these simple analytics and update them in the presence of write actions, enabling the VP action to simply look up the stored value, see the last column of Table 2. These two alternatives are termed[4] *Compute* and *Stored*, respectively.

With the friendship relationship, one may represent pending friend invitations and the confirmed friendships as unique edges (relationships) independent of one another. This design is termed *Distinct* friendship. Alternatively, one may represent both as one edge and label the edge to identify either a pending invitation or a confirmed friendship. This design is termed *Labeled* friendship. These two alternatives constitute the rows of Fig. 3, resulting in four physical graph designs shown in the quad.

The first row of Table 1 shows the average response time, $\overline{RT}$, observed with the alternative designs for the VP action. The StoredDistinct is clearly the fastest of the alternatives. Its SoAR with VP is more than twice higher than Compute, see the first row of Table 4.

The **List Friend** (LF) action of BG emulates a socialite A viewing member U_r's list of friends. Similar to the discussion of VP, A may equal to U_r emulating

[4] They are termed Basic and Manual in [10] with a relational and JSON representation of BG social graph.

Table 2. Cypher queries that implement the View Profile action with four different data models.

Data Model	Query
ComputeLabeled	a. MATCH (u:‘Members’)-[f:‘Friend’]-(uu:‘Members’)
	WHERE u.userid=profileOwnerID AND f.status=Confirmed
	RETURN COUNT (uu) AS total
	b. MATCH (u:‘Members’)<-[f:‘Friend’]-(uu:‘Members’)
	WHERE u.userid=profileOwnerID AND f.status=Pending
	RETURN COUNT (uu) AS total
	c. MATCH (u:‘Members’)<-[c:‘Postedon’]- (r:‘Resources’)
	WHERE u.userid=profileOwnerID RETURN COUNT(r) AS total
	d. MATCH (u:‘Members’) WHERE u.userid = profileOwnerID
	RETURN u.userid, u.username, u.lname, u.fname,
	u.gender, u.dob, u.jdate, u.ldate, u.address,
	u.email, u.tel, u.pic
ComputeDistinct	a. MATCH (u:‘Members’)-[f:‘Friend’]-(uu:‘Members’)
	WHERE u.userid=profileOwnerID
	RETURN COUNT (uu) AS total
	b. MATCH (u:‘Members’)<-[f:‘Invite’]-(uu:‘Members’)
	WHERE u.userid= profileOwnerID
	RETURN COUNT (uu) AS total
	c. MATCH (u:‘Members’)<-[c:‘Postedon’]-(r:‘Resources’)
	WHERE u.userid= profileOwnerID
	RETURN COUNT(r) AS total
	d. MATCH (u:‘Members’) WHERE u.userid = profileOwnerID
	RETURN u.userid, u.username, u.lname, u.address, u.gender,
	u.dob, u.jdate, u.ldate, u.fname, u.email, u.tel, u.pic
StoredLabeled/	MATCH (u:‘Members’) WHERE u.userid = profileOwnerID
StoredDistinct	RETURN u.userid, u.username, u.lname, u.fname, u.gender,
	u.dob, u.jdate, u.ldate, u.address, u.email, u.tel,
	u.friendsCount, u.pendingfCount, u.resourcesCount, u.pic

the socialite viewing her own list of friends. LF retrieves the profile information of each friend including their thumbnail image and excluding their profile image. We implement LF using the following Cypher query: `MATCH (u1:Members)-[f:Friend]- (u2:Members) WHERE u1.userid =` U_r `AND f. status=Confirmed RETURN u2.userid, u2.username, u2.fname, u2.lname, ..., u2.thumbnail`.

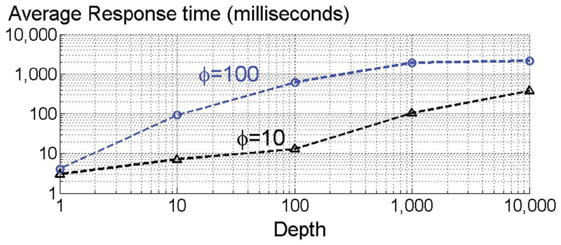
Table 1 shows representation of a friendship as a distinct edge is faster than using labeled edges. With the latter, the query must incur the additional overhead of examining the value of each label (pending versus confirmed friendship) to process the LF action. However, the alternative designs provide comparable SoAR, see the second row of Table 4.

The **Get Shortest Distance** (GSD) action of BG computes the distance between two members in the social graph. If these two members are the same user then their shortest path is zero. If they are friends then their shortest path is one. If they belong to two disjoint social graphs then their shortest path is MAX-INT. The Cypher query to implement GSD is: `MATCH p=shortestPath((u:Members)-[:Friend*.. depthToTraverse] -(u2:Members)) WHERE u.userid=`U_r `and u2.userid=`U_p `RETURN length(p) as total`. The parameter `depthToTraverse` defines the number of levels (termed depth) of friendship relationship traversed by the shortestPath function of Neo4j, striking a balance between the observed response times and the accuracy of the computed value. Increasing depth may enhance the accuracy of GSD and slow down its processing, resulting in a higher response time.

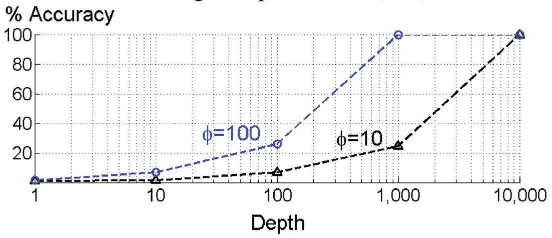
Figure 4a show the average response time of GSD as a function of the depth traversed with 10 and 100 friends per member. BG quantifies the percentage of GSD actions that observe incorrect results, termed *unpredictable data* [7], τ. Figure 4b shows the percentage of GSD requests that observe accurate results, termed Accuracy (100-τ), as function of the depth with different number of friends per member. As we increase the traversed depth on the x-axis, the computed distance becomes more accurate (i.e., τ decreases [7]) and the system becomes slower as the shortestPath function visits many more vertices. A sufficiently high depth value causes the shortestPath to visit all vertices and terminate, producing 100 % accurate results. The response time level off beyond this depth.

More formally, the response time levels off when the depth traversed multiplied by the number of friends equals the total number of members, resulting in 100 % accurate results. For example, in Fig. 4a, with the 100 K social graph and 100 friends per member, the response time levels off at a dept of 1,000. It levels of at a depth of 10,000 with 10 friends per member. The first row of Table 3 shows the observed response time with a depth of 20,000 with different number of friends per member, ϕ. With this depth, GSD provides 100 % accurate resuls and its response time levels off with all three ϕ values.

With a fixed depth for the shortestPath function, the response time is faster with fewer friends per member as this function visits fewer vertices. Hence, its accuracy is also lower. To illustrate, consider a depth of 100 on the x-axis of Fig. 4. The observed response time with 10 friends per member is six time faster, 100 versus 600 ms. Moreover, the accuracy is significantly lower, 7 % versus 25 %, as its traversal of each depth visits fewer vertices (10 times lower) and



3.a Average Response Time ($\overline{RT}$)



3.b Accuracy

Fig. 4. Average response time and accuracy of GSD as a function of the traversed depth with a 100 K member social graph and two different settings for the number of friends per member (ϕ = 10 and 100).

its likelihood of visiting the vertex of interest is lower. The first row of Table 3 shows the response time increases as a function of ϕ as GSD must process many more edges.

The **View Friend Request** (VFR) action of BG retrieves Socialite A's pending friend request, retrieving the profile information of each member who has generated a friend request for member A. The behavior of VFR with Neo4j is similar to the discussion of LF.

A socialite uses the **View Comments on Resource** (VCR) action to display the attributes of comments posted on a resource with a unique RID. Its Cypher query is as follows: `MATCH (u:Members)-[m:Manipulation]->(r:Resources) WHERE r.rid=RID RETURN u.userid, r.rid, m.mid, m.type, m.content, m.timestamp`. The socialite may post and delete comments on a resource (PCR and DCR) that creates and deletes edges between a member and a resource vertex, respectively.

The **View Top-K Resources** (VTR) enables a socialite (Member A) to retrieve and display her top k resources posted on her wall. Both the value of k and the definition of "top" are configurable. Our Cypher implementation uses the unique id assigned to a resource (rid) as the definition of top: `MATCH (u:Members) <-[cf:PostedOn]- (r:Resources) WHERE u.userid=A ORDER BY r.rid LIMIT k`.

The **List Common Friends** (LCF) action computes the common friends of two members. If these two members are the same member then their common friends is an empty set. If they are friends then LCF retrieves their common

Table 3. $\overline{RT}$, in milliseconds, of the StoredDistinct physical graph design as a function of the number of friends (ϕ) with a 100 K social graph and $\rho = 100$ resources per member.

	$\phi = 10$	$\phi = 100$	$\phi = 1{,}000$
Get Shortest Distance (GSD)	402	2,733	41,027
List Common Friends (LCF)	2,120	4,368	34,630
List Friends-of-Friends (LFF)	12	212	7,939

friends excluding themselves. Otherwise, if their distance is three or higher, then the result is an empty set. The set is defined as the members who are a distance of one from both members. `Match (u1:Members), (u2:Members),(mf:Members) WHERE u1.userid=`U_p `AND u2.userid=`U_r `AND (u1)-[:Friend]-(mf)-[:Friend]-(u2) RETURN mf.userid`. The response time of LCF increases as a function of the number of friends per member, ϕ. (See the second row of Table 3.) At times, the result of the LCF action might be the empty set as its input members may have no common friends. The likelihood of this is lower with higher values of ϕ, explaining the higher average response time.

The **List Friends-of-Friends** (LFF) action computes those members who are a distance of two from the specified member, including their common friends. The Cypher query to implement this action is as follows: `MATCH (u1:Members)-[:Friend *2..2]-(u2:Members) WHERE u1.userid=`U_p `and NOT (u1)-[:Friend]-(u2) RETURN distinct u2.userid`. The third row of Table 3 shows the response time of the LFF action increases superlinearly as a function of ϕ. With LFF, a ten fold increase in the value of ϕ results in a ten fold increase in the number of retrieved userids. More precisely, given M members, BG constructs the social graph by assigning members (i+j)%M as friends of Member i where the value of j varies from 1 to[5] $\frac{\phi}{2}$. Hence, LFF retrieves 2ϕ userids. For example, with $\phi = 10$ and 100, LFF retrieves 20 and 200 members, respectively. While this explains the higher response time as a function of ϕ, there appears to be additional overhead that causes the response time of Neo4j to increase superlinearly.

3.2 BG's Write Actions

BG supports four write actions that impact the friendship relationship (edges) between members (vertices). These are Invite Friend (IF), Accept Friend Request (AFR), Reject Friend Request (RFR), and Thaw Friendship (TF). All involve Socialite A invoking the action on Member U_r. These actions modify either the presence of edges or the attribute value of an edge between vertices. For example, the Cypher `create edge` command for the IF action with the labeled design is as follows: `MATCH (u1:Members), (u2:Members) WHERE u1.userid=A AND u2.userid=`U_r `CREATE (u1)-(:Friend{status:pending})->(u2)`.

[5] The torus characteristics of the mod function guarantees ϕ friends per member.

Table 4. SoAR of the four physical graph models with workloads consisting of VP only, LF only, and a mix of read and write actions.

Workload	ComputeLabeled	ComputeDistinct	StoredLabeled	StoredDistinct
View Profile (VP)	971	714	2,205	2,251
List Friend (LF)	93	119	112	118
0.1 % Write Actions	117	459	819	835
1 % Write Actions	46	369	435	499
10 % Write Actions	32	162	0	100

With the Stored representations, these write actions must maintain the simple analytics attribute values of a vertex (member) up to date. For example, the AFR action must increment the number of friends of the vertices corresponding to Members A and U_r. Moreover, it must decrement[6] the number of pending friend invitations for Member A.

Table 4 shows the SoAR of the alternative physical graph designs for a mix of read and write actions. The first column increases the frequency of the write actions such as Invite Friend and Thaw Friendship, see Table 5. This reduces the SoAR of all designs shown in Fig. 2. With a mix consisting of 10 % write actions, computing the analytics of the View Profile action provides a higher performance than the stored designs due to their overhead to maintain the value of simple analytics up to date.

Representing pending and confirmed friendship relationships with unique edges provides a higher performance when compared with labeled edges, compare ComputeDistinct and ComputeLabeled columns in Table 4. Both slow down as a function of an increasing mix of write actions. With ComputeDistinct, when a member confirms a pending friendship invitation, the system deletes an edge and inserts a new one. With ComputeLabeled, the same action changes the value associated with a property of an edge. This consumes more of system resources with our workloads, resulting in a lower SoAR.

4 Comparison of Neo4j with SQL-X Using BG

This section compares the performance of an industrial strength relational database management system (RDBMS) named[7] SQL-X with Neo4j using BG. The schema used for the RDBMS to represent the social graph is as follows:

- Users(userid, username, pw, fname, lname, gender, dob, jdate, ldate, address, email, tel, profileImage, thumbnailImage, #Friends, #FriendInvitations, #Resources)
- Friendship(*inviter, invitee*, status)

[6] BG is a stateful benchmark that generates valid actions only. When it invokes the AFR action involving Member A and U_r, it does so based on its knowledge of U_r having a pending friend invitation from A. See [7] for details.

[7] Due to licensing agreement, we cannot disclose the identity of this system.

Table 5. Three mixes of social networking actions.

BG Social Actions	Type	Very Low (0.1 %) Write	Low (1 %) Write	High (10 %) Write
View Profile, VP	Read	40 %	40 %	35 %
List Friends, LF	Read	5 %	5 %	5 %
View Friend Requests, VFR	Read	5 %	5 %	5 %
Invite Friend, IF	Write	0.04 %	0.4 %	4 %
Accept Friend Request, AFR	Write	0.02 %	0.2 %	2 %
Reject Friend Request, RFR	Write	0.02 %	0.2 %	2 %
Thaw Friendship, TF	Write	0.02 %	0.2 %	2 %
View Top-K Resources, VTR	Read	40 %	40 %	35 %
View Comments on a Resource, VCR	Read	9.9 %	9 %	1 %

Table 6. $\overline{RT}$ in milliseconds with maximum depth = 1,000.

	SQL-X	Neo4j
Get Shortest Distance (GSD)	718	2,588
List Common Friends (LCF)	14	4,317
List Friends-of-Friends (LFF)	26	163

– Resource(*rid, creatorid, walluserid*, type, body, doc)
– Manipulation(*mid, rid, modifierid, creatorid*, timestamp, type, content)

Underlined attributes are indexed and serve as the primary key of a table. An italicized attribute represents a foreign key relationship. A confirmed friendship between two members is represented as two rows.

Except for the LCF and the GSD actions, an implementation of BG's actions using the SQL query language is straightforward and described in [6,7,10]. We implement LCF(A,B) using a single query: `SELECT DISTINCT f1.inviteeid FROM Friendship f1, f2 WHERE f1.inviteeid=f2.inviteeid and f1.inviterid= A and f2.inviterid=B and f1.status=Confirmed and f2.status=Confirmed`. Figure 5 shows an implementation of the Breadth First Search (BFS) algorithm to implement GSD using the SQL query language. This algorithm issues a SQL query for each level of BFS starting with one member of the social graph, identified by UserID1. It terminates once it encounters the other member of the social graph (UserID2), exhausts all the members of the social graph, or exceeds its maximum allowed depth.

Table 6 shows the average response time of GSD, LCF, and LFF with SQL-X and Neo4j for a social graph consisting of 100 K members, 100 fpm, and 100 rpm. SQL-X is faster than Neo4j for processing each of these commands. An SQL implementation of these commands reference a single table, Friendship, that is a vertical slice of the data. For example, The GSD algorithm of Fig. 5 queries the

```
Algorithm GSD(UserID1, UserID2, MaxDepth):
If UserID1 equals UserID2 return 0
If MaxDepth equals 0 return MAX-INT
Initialize Visited ← {}
Initialize SRC ← {UserID1}
Initialize CurrentDepth ← 0
While (true):
(1) CurrentDepth ← CurrentDepth+1
(2) If (CurrentDepth > MaxDepth) return MAX-INT
(3) If (Visited contains all members) return MAX-INT
(4) Qry ← "SELECT unique inviteeid FROM Friendship WHERE "
(5) For each userid in SRC:
          Extend Qry with the clause "inviterid=userid"
              using boolean or connective
(6) Visited ← Visited ∪ SRC
(7) Execute Qry using RDBMS to obtain a result set R
(8) If (UserID2 ∈ R) return CurrentDepth
(9) SRC = R - (R ∩ Visited)
(10)If (SRC is empty) return MAX-INT
```

Fig. 5. Get Shortest Distance using SQL-X.

Friendship table repeatedly in Step 5. Neo4j, on the other hand, may retrieve a vertex that contains several property values of a member including a 12 KB profile image. It is possible to further enhance the reported GSD numbers with SQL-X by implementing the algorithm of Fig. 5 as a stored procedure.

Table 7 shows the observed SoAR with SQL-X and Neo4j (using the Stored-Distinct design, see Fig. 3) for the three mix of write actions shown in Table 5. We consider a BG database configured with either images or no images. The latter lacks the 12 KB profile image and the 2 KB thumbnail image. With both, the schema of SQL-X stores the simple analytics of a member as an attribute value of a row and requires a write action to maintain these values up-to-date [10].

SQL-X performs poorly when required to store images larger than 4 KB [10, 19] and Neo4j outperforms it by a wide margin. With a social graph that has no images, SQL-X outperforms Neo4j by a wide margin, see last two columns of Table 7. SoAR of Neo4j is also enhanced when the social graph has no images. In [10], we show that storing profile images in the file system, termed Boosted SQL-X design, enhances the SoAR of SQL-X by more than ten folds. A future research direction is to analyze Neo4j with images stores in the file system (similar to the discussion of Boosted SQL-X). We speculate its performance to fall between the two extremes shown in Table 7.

Table 7. SoAR with 100 K members, $\phi = 100$ fpm, and $\rho = 100$ rpm, with and without images.

	With images		No images	
	SQL-X	Neo4j	SQL-X	Neo4j
0.1 % Write Action	360	835	20,550	1,460
1 % Write Action	290	499	16,135	688
10 % Write Action	0	100	2,095	150

5 Future Research Direction

We are extending this study by considering additional graph data stores, characterizing their scalability and their role in processing more complex social networking actions. We describe these in turn.

We are using BG to complete an evaluation of Neo4j and other graph databases such as G* [18] and OrientDB [23]. This includes an analysis of their scalability characteristics and a comparison with data stores that support alternative data models, e.g., document stores, extensible stores, key-value stores and relational DBMSs. We also intend to analyze the overhead of an Object Graph Model (OGM) such as Blueprint when compared to using the native interface of a graph data store [16].

Moreover, we intend to investigate alternative physical graph designs for processing more complex social networking actions, namely, feed following actions such as Share Resource (SR) and View New Feed (VNF) [9]. These model a member producing events for consumption by others and displaying the events generated by other members and entities, typically their friends or those that they follow. Both the highly variable fan-out of the follows graph along with its dynamically changing structure (e.g., a member thaws friendship with another member) makes an implementation of feed following challenging [9,20]. One may introduce different designs and implementations to address these challenges [5,17,21]. One is to materialize the feed of a member and maintain it up to date when new events are produced by those she follows [21]. A graph database such as Neo4j may be suitable for this Push paradigm because it supports extensions of a vertex with new attributes. A design may split a vertex into multiple vertices once it increases beyond a certain size [13]. Finally, edges may maintain the relationship between older and newer feed as a member's feed grows in size. An alternative to Push is to Pull events and may include clever designs that synergizes those members with mutual friends by maintaining one news feed for them. We plan to investigate these alternative implementations with Neo4j and other graph data stores.

References

1. Amsden, Z., Bronson, N., Cabrera III, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Hoon, J., Kulkarni, S., Lawrence, N., Marchukov, M., Petrov, D., Puzar, L., Venkataramani, V.: TAO: how facebook serves the social graph. In: SIGMOD Conference (2012)
2. Angles, R., Boncz, P., Larriba-Pey, J., Fundulaki, I., Neumann, T., Erling, O., Neubauer, P., Martinez-Bazan, N., Kostev, V., Toma, I.: The Linked data benchmark council: a graph and RDF industrybenchmarking effort. SIGMOD Rec. **43**, 27–31 (2014)
3. Angles, R., Prat-Pérez, A., Dominguez-Sal, D., Larriba-Pey, J.: Benchmarking database systems for social network applications. In: First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013 (2013)
4. Armstrong, T., Ponnekanti, V., Borthakur, D., Callaghan, M.: LinkBench: a database benchmark based on the facebook social graph. In: ACM SIGMOD, June 2013
5. Bai, X., Junqueira, F.P, Silberstein, A.: Cache refreshing for online social news feeds. In: CIKM (2013)
6. Barahmand, S.: Benchmarking interactive social networking actions. Ph.D. thesis, Computer Science Department, USC (2014)
7. Barahmand, S., Ghandeharizadeh, S.: BG: a benchmark to evaluate interactive social networking actions. In: Proceedings of 2013 CIDR, January 2013
8. Barahmand, S., Ghandeharizadeh, S.: Benchmarking correctness of operations in big data applications. In: Proceedings of IEEE MASCOTS (2014)
9. Barahmand, S., Ghandeharizadeh, S.: Extensions of BG for testing and benchmarking alternative implementations of feed following. In: ACM SIGMOD Workshop on Reliable Data Services and Systems (RDSS), June 2014
10. Barahmand, S., Ghandeharizadeh, S., Yap, J.: A comparison of two physical data designs for interactive social networking actions. In: CIKM (2013)
11. Boncz, P.: LDBC: benchmark for graph and RDF data management. In: IDEAS, October 2013
12. Transaction Processing Performance Council. TPC Benchmarks. http://www.tpc.org/information/benchmarks.asp
13. Nishtala, R., et al.: Scaling memcache at Facebook. In: NSDI (2013)
14. Ghandeharizadeh, S., Barahmand, S.: A mid-flight synopsis of the BG social networking benchmark. In: Rabl, T., Raghunath, N., Poess, M., Bhandarkar, M., Jacobsen, H.-A., Baru, C. (eds.) WBDB 2013. LNCS, vol. 8585, pp. 19–31. Springer, Heidelberg (2013)
15. Gray, J.: The Benchmark Handbook for Database and Transaction Systems, 2nd edn. Morgan Kaufmann, San Mateo (1993). ISBN 1055860-292-5
16. Holzschuher, F., Peinl, R.: Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4J. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops, EDBT 2013 (2013)
17. Junqueira, F.P., Leroy, V., Serafini, M., Silberstein, A.: Shepherding social feed generation with sheep. In: SNS (2012)
18. Labouseur, A., Olsen, P., Hwang, J: Scalable and robust management of dynamic graph data. In: VLDB Workshop on Big Dynamic Distributed Data (2013)
19. Sears, R., Ingen, C.V., Gray, J.: To BLOB or not to BLOB: large object storage in a database or a filesystem. Technical report MSR-TR-2006-45, Microsoft Research (2006)

20. Silberstein, A., Machanavajjhala, A., Ramakrishnan, R.: Feed following: the big data challenge in social applications. In: DBSocial (2011)
21. Silberstein, A., Terrace, J., Cooper, B.F., Ramakrishnan, R.: Feeding frenzy: selectively materializing users' event feeds. In: SIGMOD Conference (2010)
22. The Neo4j Team. The Neo4j Manual V2.1.1, 29 May 2014. http://www.neo4j.org
23. Tesoriero, C.: Getting Started with OrientDB. Packt Publishing Ltd, Birmingham (2013)