# Building Social Networking Services Systems Using the Relational Shared-Nothing Parallel DBMS

Kyu-Young Whang[a]*, Inju Na[a], Tae-Seob Yun[a], Jin-Ah Park[a], Kyu-Hyun Cho[a],
Se-Jin Kim[a], Ilyeop Yi[a], Byung Suk Lee[b]
[a]School of Computing, KAIST, Daejeon, Korea
[b]Department of Computer Science, University of Vermont, Burlington, Vermont, U.S.A.
Emails: {kywhang, ijna, tsyun, jinah, khcho, sjkim, iyyi}@mozart.kaist.ac.kr, bslee@cs.uvm.edu

September 26, 2019

**Abstract**

We propose methods to enable the relational model to meet scalability and functionality needs of a large-scale social networking services (SNS) system. NewSQL has emerged recently indicating that shared-nothing parallel *relational* DBMSs can be used to guarantee the ACID properties of transactions while keeping the high scalability of NoSQL. Leading commercial SNS systems, however, rely on a *graph* – not relational – data model with key-value storage and, for certain operations, suffer overhead of unnecessarily accessing multiple system nodes. Exploiting higher semantics with the relational data model could be the remedy. The solution we offer aims to perform a transaction as a set of independent local transactions whenever possible based on the conceptual semantics of the SNS database schema. First, it hierarchically clusters entities that are sitting on a path of frequently navigated one-to-many relationships, thereby avoiding inter-node joins. Second, when a multi-node delete transaction is performed over many-to-many relationships, it defers deletion of related references until they are accessed later, thereby amortizing the cost of multi-node updates. These solutions have been implemented in Odysseus/SNS – an SNS system using a shared nothing parallel DBMS. Performance evaluation using synthetic workload that reflects the real SNS workload demonstrates significant improvement in processing time. We also note that our work is the first to present the entity-relationship schema and its relational representation of the SNS database.

## 1 Introduction

The social networking services (SNS) system is an online platform that supports the cultivation and maintenance of social relationships among users through open communication

*Corresponding author

and information sharing. Some of the commercial SNS systems have grown very large in scale, with several hundred million or even a billion active users. Thus, these SNS systems need an efficient "scale-out" base system to store and process massive data produced by an ever increasing number of users in a distributed environment.

Figure 1 shows how large-scale data management systems have evolved. Ever since MapReduce came about, the NoSQL system became popular as a highly scalable system. The NoSQL system typically uses the key-value storage format, in which all data values under the same key are stored together and accessed together fast. It, however, lacks the high-level functionality of the relational model because of its low-level storage format and compromises the ACID properties [7]. In this regard, there has been a transition from NoSQL to NewSQL recently [2, 11]. A NewSQL system is essentially a relational DBMS (with a support for SQL, index, and schema) that provides the same kind of scalability as that of NoSQL while guaranteeing the ACID properties of transactions. MySQL Cluster is one representative example. Its base architecture is a shared-nothing parallel DBMS, which has been shown to outperform MapReduce in terms of processing large-scale database and query load [19, 20].

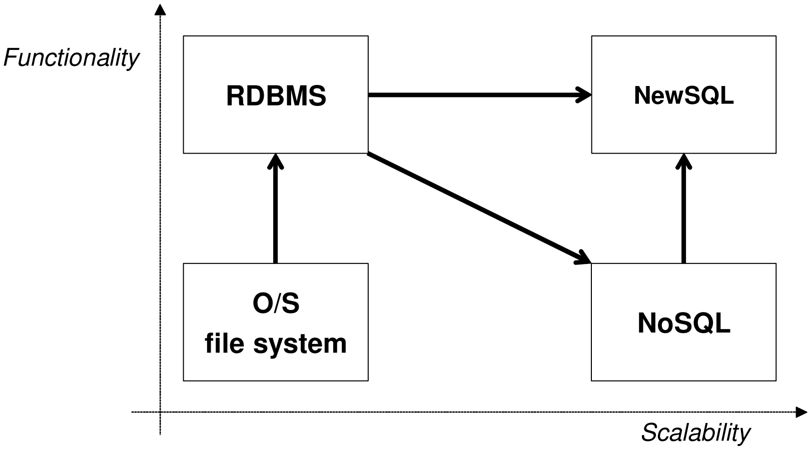


Figure 1: Evolution of data management systems.

The base system used by leading commercial SNS systems (e.g., Facebook) is standing half way between NoSQL and NewSQL – it stores data in a massively-parallel DBMS (specifically, MySQL Cluster), which is NewSQL, but represents data using a graph (or object-association) model [1, 5] in the key-value format, which is NoSQL. This graph model represents data at such a low level that it ends up with distributing objects randomly over different machines, and therefore, incurs a significant overhead of accessing multiple machines to retrieve the objects (more on this issue in Section 2.1). This problem of the graph data model can be much alleviated by using a higher-level data model like the entity-relationship model or relational model, which can aggregate the objects into entity sets and their relationships or relations.

In this paper we demonstrate that the relational data model (as opposed to the graph data model) can be used to implement a scale-out SNS system based on the parallel DBMS while resolving the problems of the graph data model. Using the semantics of the relational model, we process global transactions as separate local (i.e., single-node) transactions as much as possible. In particular, we address the problem of *inter-node join* in different machines, and the problem of *multi-node updates* that require accesses to tuples stored in multiple machines. To resolve these problems, we propose the following solutions. We first start with a conceptual (entity-relationship) schema that captures the semantics of common SNS operations. Then, we exploit the notion of *hierarchical clustering* that relates entities in multiple relations by modelling one-to-many relationships to sequences of identifying relationships, through which a relation inherits the primary key of the root relation of the path – called the *identifying key.* Then, by distributing entities hashed by the identifying key, we partition related entities into the same node and as a result avoid inter-node joins, thereby implementing the operation as a single-node transaction. For many-to-many relationships, unlike one-to-many, inter-node joins are unavoidable. In this case, we use a *deferred update* strategy whereby, when an entity in one node is deleted, deletion of the references to it in other nodes are deferred until they are actually accessed later and, as a result, decompose a multi-node transaction into multiple *single-node-update transactions*(to be defined in Section 3.3). As explained in Section 3, the decomposed execution of a transaction does not affect the consistency of the original multi-node delete transaction. These ideas have been implemented in Odysseus/SNS [24], an SNS system using a shared-nothing parallel DBMS, which is an extension of the Odysseus DBMS [24].

For performance evaluation, we measured the processing time of SNS operations implemented in our system using the proposed methods. The experiments were conducted in a small-scale computer cluster using a synthetic workload (i.e., data, queries) proportionately scaled according to the real SNS system workload. The results show that our system significantly reduces the processing time of newsfeed and timeline operations, which together account for about 80% of the query load, and also greatly reduces the processing time of delete operations. The former is attributed to the idea of clustering by identifying keys, and the latter to the idea of deferred delete strategy. This scalability comes largely from the fact that, with clustering, a typical query can be processed by forwarding it to only one or a few specific nodes without having to access the data distributed in a large number of nodes.

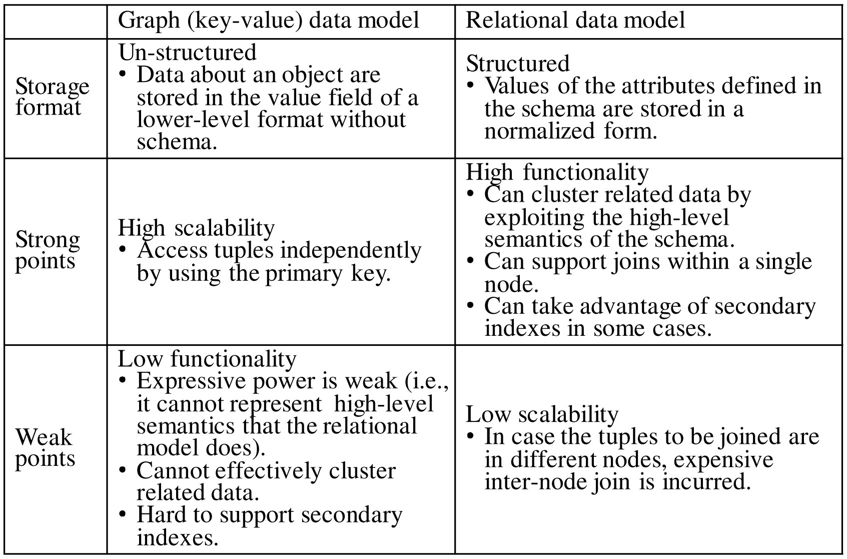
Contributions of this paper can be summarized as follows. First, this paper is the first to present the conceptual (entity-relationship) schema and its relational representation of social networking services operations. Second, we propose a method of removing inter-node joins over one-to-many relationships by hierarchical clustering. Third, we propose a deferred delete strategy to handle a multi-node transaction over a many-to-many relationship as multiple single-node-update transactions.

The rest of this paper is organized as follows. Section 2 provides relevant background information. Section 3 discusses our SNS database schema design and the proposed query processing methods with the relational data model. Section 4 presents performance evaluation. Section 5 concludes the paper.

## 2 Background

Table 1 highlights the differences between the graph data model and the relational data model in terms of the storage format, strengths, and weaknesses.

Table 1: Differences between the graph data model and the relational data model.



| | Graph (key-value) data model | Relational data model |
|---|---|---|
| Storage format | Un-structured<br>• Data about an object are stored in the value field of a lower-level format without schema. | Structured<br>• Values of the attributes defined in the schema are stored in a normalized form. |
| Strong points | High scalability<br>• Access tuples independently by using the primary key. | High functionality<br>• Can cluster related data by exploiting the high-level semantics of the schema.<br>• Can support joins within a single node.<br>• Can take advantage of secondary indexes in some cases. |
| Weak points | Low functionality<br>• Expressive power is weak (i.e., it cannot represent high-level semantics that the relational model does).<br>• Cannot effectively cluster related data.<br>• Hard to support secondary indexes. | Low scalability<br>• In case the tuples to be joined are in different nodes, expensive inter-node join is incurred. |

### 2.1 Graph data model

In the graph model, vertices represent objects (e.g., users, posts, comments) and edges represent associations between objects, as illustrated in Figure 2. Vertices and edges may contain data in the key-value format. According to Bronson et al. [5], the key-value storage has the following format:

- for a vertex, (id, (object_type, (attribute, value)*)
- for an edge, ((id$_1$, association_type, id$_2$), (time, (attribute, value)*))

where the key is the object ID (i.e., id) for a vertex and the triplet (id$_1$, association_type, id$_2$) for an edge, and the value (or 'payload") is the data comprising the object or the association (see Figure 3 for an illustration)

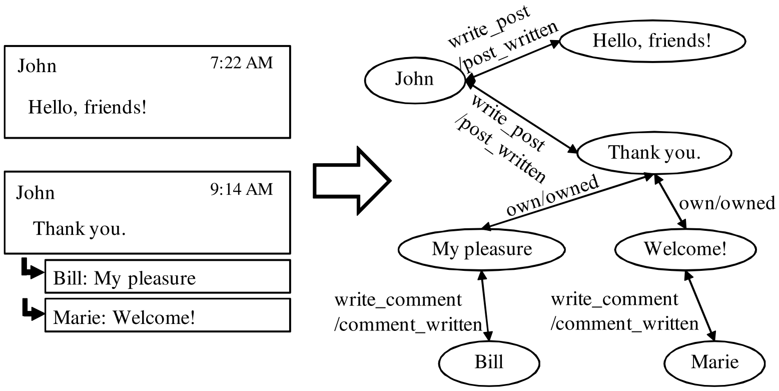


Figure 2: An example of the graph data model.

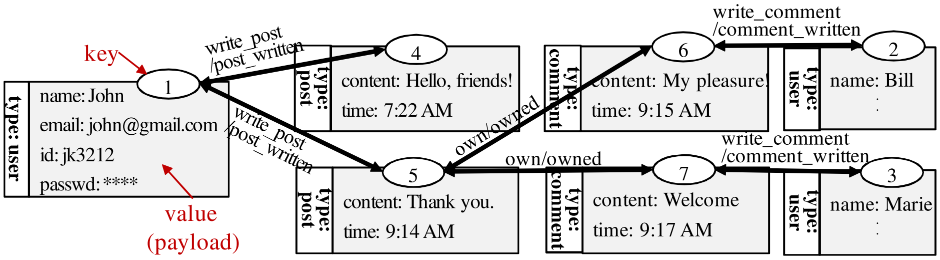
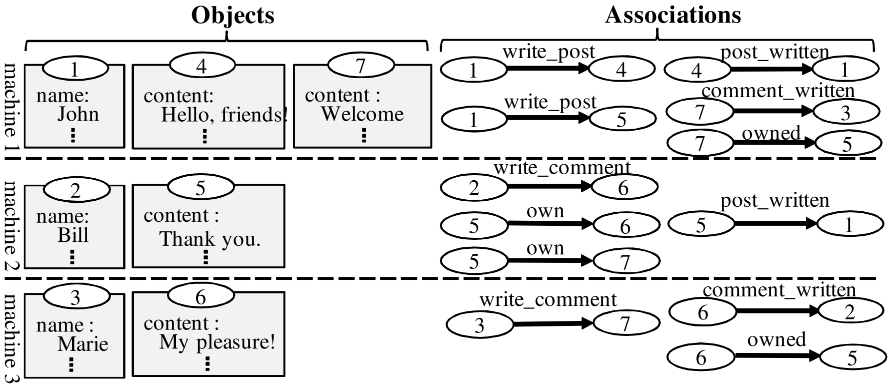


Figure 3: An example of graph data in the key-value format.

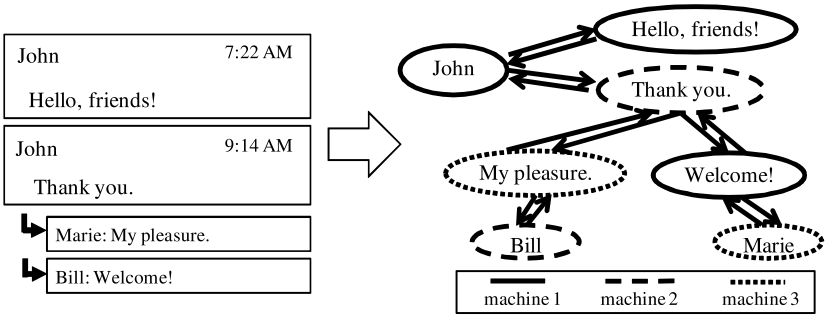
Data distribution and storage rely on hashing. For vertices, the hashing key is the object ID (i.e., id) and, for edges, the source object ID (i.e., id). Figure 4 illustrates the data storage distributed over three machines.

This graph model (with key-value storage) has serious drawbacks stemming from the low-level functionality of the model. It is such a simple low-level model that it misses out much of the semantics that can be represented in a higher-level model like the relational model. For instance, we can only represent that two individual vertices (instances) are related to each other by using edges, but we cannot represent the concept of a set of objects sharing the same type (or format) and their structural relationships (e.g., one-to-many, many-to-many), and thus, we cannot take advantage of these semantics to effectively cluster the data. Moreover, in the graph model, vertices and edges are randomly distributed over different machines unnecessarily, thereby incurring random access to them. Further, the key-value storage makes it difficult to support secondary indexes [7, 10]. As an example, Figure 5 illustrates the overhead of having to access all three machines in order to view a user's timeline.



Machine ID to store an object = object ID % 3
Machine ID to store an association = source object ID % 3

Figure 4: An example of graph data in the key-value format distributed over three nodes.



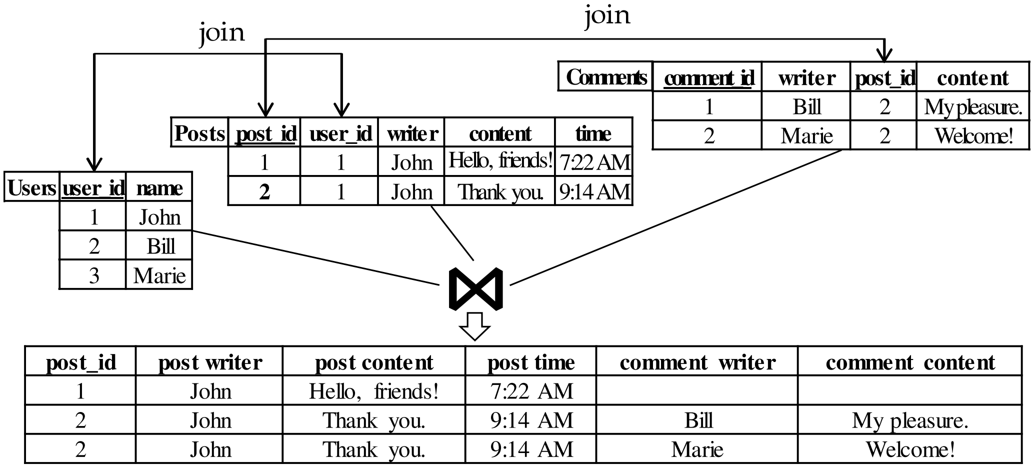
All three machines need to be accessed in order to view John's timeline.

Figure 5: An example of the timeline operation on graph data in the key-value format.

## 2.2 Relational data model

Figure 6 shows the relational model view of the data shown in the graph model view of data in Figure 3. As we can see in the figure, the relational model brings the advantage that data are aggregated into relations, relationships are represented through foreign keys, and joins can be processed using the foreign keys.

For data distribution and storage, we can partition a relation and distribute the tuples by the primary key. Simple distribution, however, does not guarantee that the related tuples are clustered in the same machine (or node)(see Figure 7). As a result, if the tuples to be joined are stored in different nodes, then we have to access multiple nodes, that is, *inter-node join* is needed, as shown in Figure 7. The problem of inter-node join is the network cost incurred to access multiple nodes to perform join across them. We thus need a method to cluster the related data (i.e., tuples to be joined) in the same node.



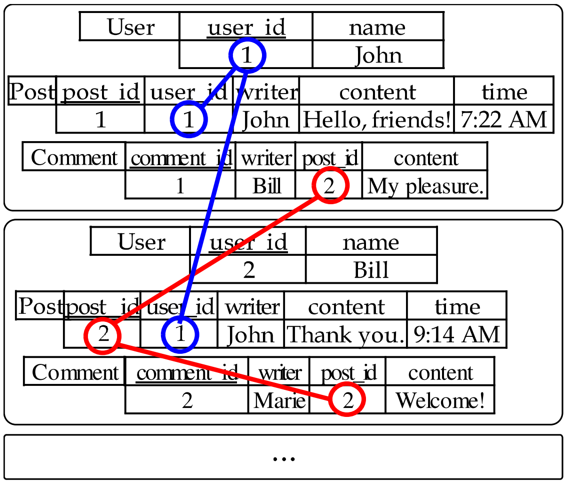
| Users | user_id | name |
|---|---|---|
| | 1 | John |
| | 2 | Bill |
| | 3 | Marie |

| Posts | post_id | user_id | writer | content | time |
|---|---|---|---|---|---|
| | 1 | 1 | John | Hello, friends! | 7:22 AM |
| | 2 | 1 | John | Thank you. | 9:14 AM |

| Comments | comment_id | writer | post_id | content |
|---|---|---|---|---|
| | 1 | Bill | 2 | My pleasure. |
| | 2 | Marie | 2 | Welcome! |

| post_id | post writer | post content | post time | comment writer | comment content |
|---|---|---|---|---|---|
| 1 | John | Hello, friends! | 7:22 AM | | |
| 2 | John | Thank you. | 9:14 AM | Bill | My pleasure. |
| 2 | John | Thank you. | 9:14 AM | Marie | Welcome! |

Figure 6: An example of the relational model view of data.



| User | user_id | name |
|---|---|---|
| | 1 | John |

| Post | post_id | user_id | writer | content | time |
|---|---|---|---|---|---|
| | 1 | 1 | John | Hello, friends! | 7:22 AM |

| Comment | comment_id | writer | post_id | content |
|---|---|---|---|---|
| | 1 | Bill | 2 | My pleasure. |

| User | user_id | name |
|---|---|---|
| | 2 | Bill |

| Post | post_id | user_id | writer | content | time |
|---|---|---|---|---|---|
| | 2 | 1 | John | Thank you. | 9:14 AM |

| Comment | comment_id | writer | post_id | content |
|---|---|---|---|---|
| | 2 | Marie | 2 | Welcome! |

Inter-node joins occur between node 1 and node 2 to execute a query 'User $\bowtie_{User.user\_id=Post.user\_id}$ Post $\bowtie_{Post.post\_id=Comment.post\_id}$ Comment".

Figure 7: An example of relational data distribution and storage.

# 3 Database Design with the Relational Model

## 3.1 SNS entity-relationship schema

Figure 23 in Appendix shows the conceptual schema of entities and relationships representing the types of SNS objects and associations where weak entity types and their identifying relationship types are shown as rectangles and diamonds in double-lines and only primary keys (underlined by a solid line) or partial keys (underlined by a broken line) are shown as attributes. For the sake of easy implementation, we simplify Figure 23 to Figure 8 through the following methods. We model the post_id attributes of the User-Post and Group-Post entity types as the primary keys implementing them as globally unique identifiers. We do the same for the comment_id attributes of the User-Post-Comment and Group-Post-Comment entity types. Then, we integrate the User-Post and Group-Post entity types into the Post entity type whose primary key is post_id and the User-Post-Comment and Group-Post-Comment entity types into the Comment entity type whose primary key is comment_id. We also integrate the relationship types that relate those entity types: the write user-post and the write group-post relationship types into the write post relationship type, the recommend user-post and recommend group-post relationship types into the recommend post relationship type, the write user-post-comment and write group-post-comment relationship types into the write comment relationship type, and the recommend user-post-comment and recommend group-post-comment relationship types into the recommend comment relationship type. In addition, to avoid clutter, we focus on the entity and relationship types that we explain in this section by omitting the entity and relationship types for thumbnail and photo.

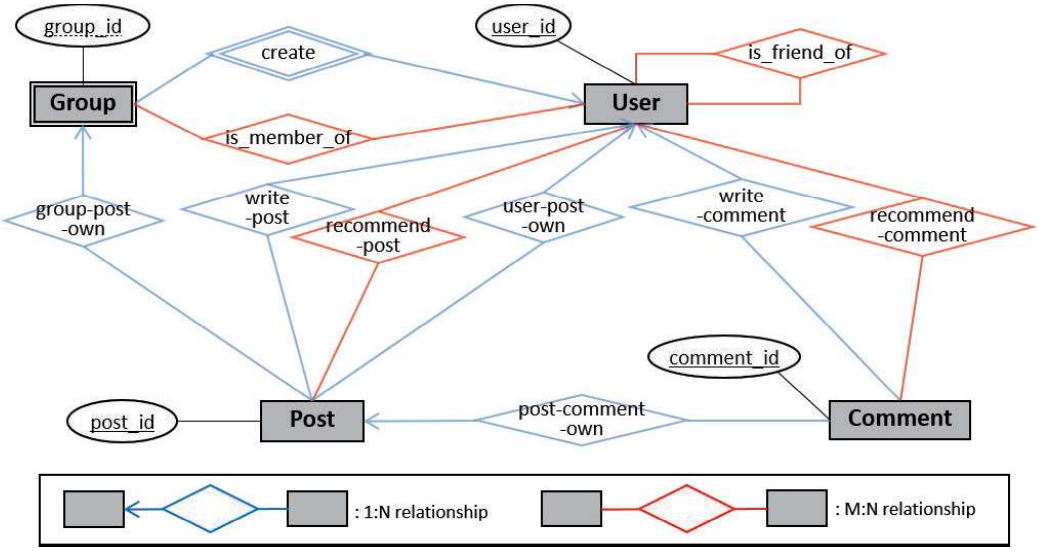
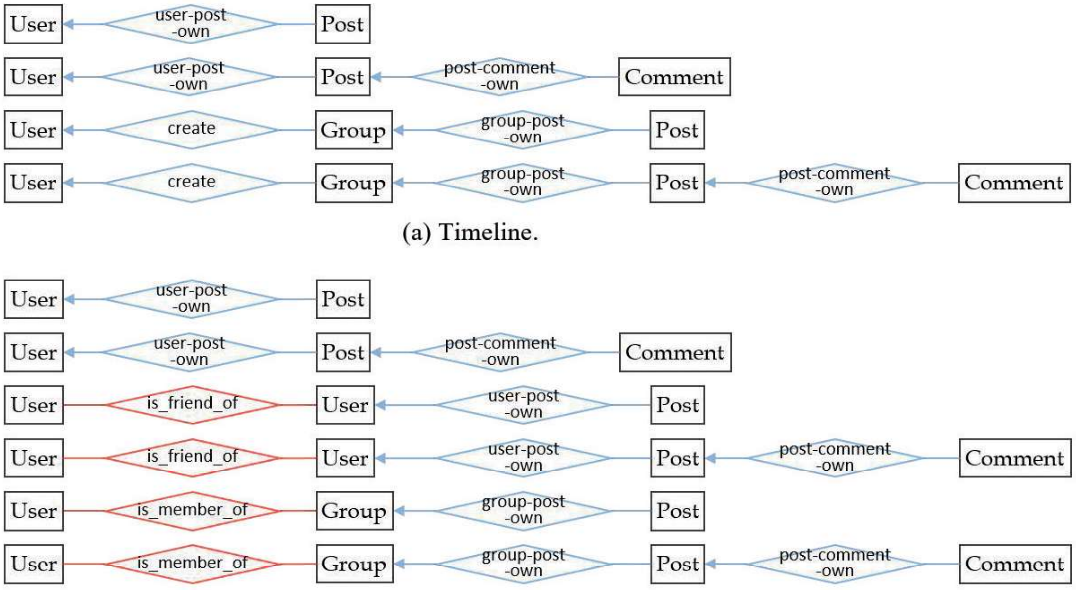


Figure 8: SNS entity-relationship schema.

Table 2 shows commonly used SNS operations categorized by the service type. The primary operations in Table 2 are executed most commonly in SNS. The timeline and newsfeed operations, in particular, comprise a majority of SNS operations. Timeline shows 10 recent posts and related comments owned by a certain user or the groups created by the user. Newsfeed shows 10 recent posts and related comments owned by a certain user, all of the user's friends, and all groups the user is a member of. Figure 9 shows the sequences of entity types accessed via relationship types in order to perform these operations. Both operations access multiple relationship types, and, therefore, the database design should make these multi-relationship operations efficient.

Table 2: Social networking services operations.

| Class | Type | | Description |
|---|---|---|---|
| Primary | Post | | View timeline |
| | | | View newsfeed |
| | | | Write/delete/modify/recommend post |
| | | | Write/delete/modify/recommend comment |
| | | | View photo |
| Secondary | General | | Sign up/deactivate |
| | | | Login/logout |
| | | | Create/delete group |
| | | | Join/unjoin group |
| | | | Make/accept friend request |
| | | | Unfriend a friend |
| | Information | User | View/modify user profile |
| | | | View the friend list of user |
| | | | View the group list of user |
| | | Group | View the user list of a group |
| | | | Search for a member of a group |
| | Search | | Find users |
| | | | Find groups |
| | | | Find posts on a timeline |



(a) Timeline.

(b) Newsfeed.

Figure 9: SNS timeline and newsfeed operations.

### 3.2 One-to-many relationship

As explained in Section 2.2, expensive inter-node joins occur if entities to be accessed together via one-to-many relationships are stored in different nodes, and they incur severe performance penalty due to network communication overhead. The novel solution we propose avoids inter-node joins by modelling those one-to-many relationships as sequences of identifying relationships and hierarchically clustering the entities related by those identifying relationships. Each sequence of identifying relationship types has a *root entity type.* An *identifying key* is the primary key of the root entity type. The identifying key is inherited in all entity types connected via the sequence of identifying relationship types. Then, by distributing entities over multiple nodes based on the identifying key as the partitioning attribute (through hashing), we can store entities that have the same identifying key value in the same node. Hence, the entities that are connected by a sequence of identifying relationships are *hierarchically clustered* (i.e., all entities connected to a root entity via a sequence identifying relationships are clustered) in one node and we can avoid inter-node joins when processing the multi-relationship operations[1].

Figure 10 shows a sub-schema including only one-to-many relationship types from the SNS entity-relationship schema in Figure 8. Here, the Post entity type can be considered a weak entity type. The reason is as follows. There are two one-to-many identifying relationship type paths between the User entity type and the Post entity type: 1) User-user-post-own-Post and 2) User-create-Group-group-post-own-Post. Although these two paths seem to share the post entity type and do not form hierarchies, actually those two paths are independent since the set of entities from the Group-Post entity type is disjoint from that from the User-Post entity type in Figure 23. That is, the Post entity type can be a weak entity type in each path. The Comment entity type also can be considered a weak entity type for the same reason. Hence, we model the Post and Comment entity types as if they were weak entity types. Then, we select the relationship types that are frequently traversed by SNS primary operations (See Figure 9) among those that relate the Post, Comment, and User (or Group) entity types and model the selected ones as if they were identifying relationship types. Those are shown in Figure 10 as the diamonds and rectangles with double-lines.

The primary key user_id attribute of the root entity type User is inherited being cascaded through the sequence of identifying relationship types, that is, from User to Post and to Comment and from User to Group to Post and to Comment as shown in Figure 11. Then, a tuple is stored in the node corresponding to the hash value of its identifying key, which is user_id in this example (see Figure 12). As a result, a set of hierarchically related tuples are stored in the same node. By the same token, a query is allocated to the node corresponding to the same hash value, as indicated in the predicate of the following

[1] The concepts of the identifying relationship and clustering can be generalized to ternary relationships. However, since all relationships in the SNS database (See Figure 8) are binary, all discussions on ternary relationship are out of our focus, so that in this paper, we focus on binary relationships.

example: “select * from User, Post, Comment where User.user_id = 1 and User.user_id = Post.user_id and Post.post_id = Comment.post_id”.

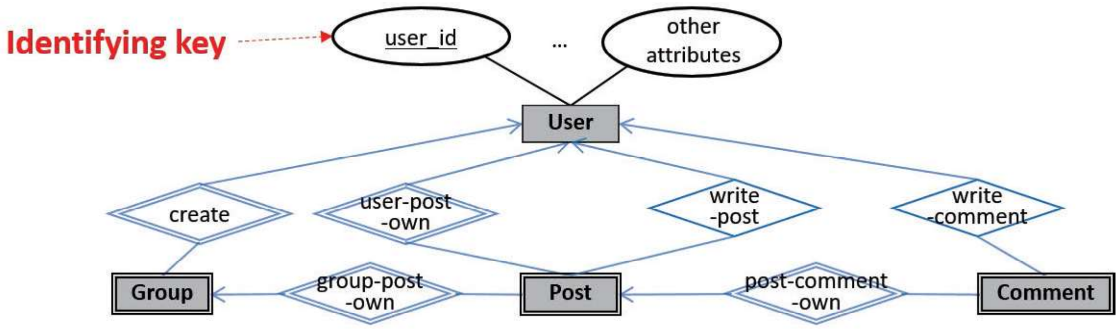


Figure 10: An example of modelling one-to-many relationship types as identifying relationship types.

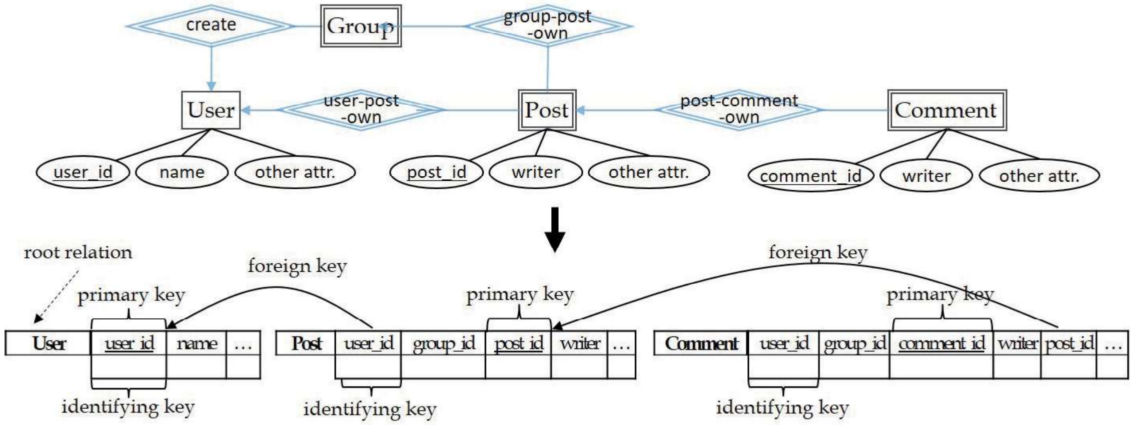
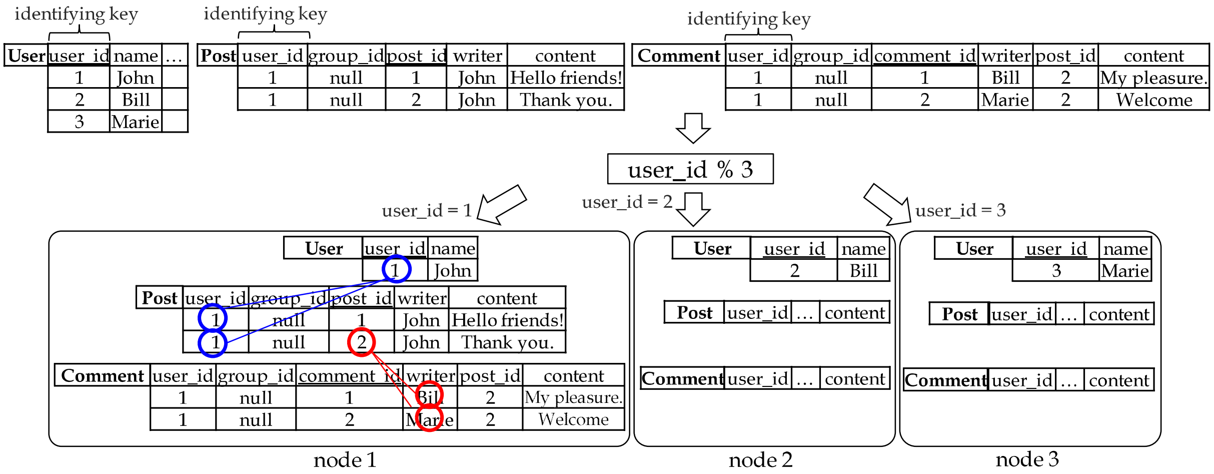


Figure 11: An example of mapping identifying relationship types to relational schema.

Figure 13 shows the sequences of identifying relationships traversed to execute timeline and newsfeed operations. For the timeline operation(see Figure 13(a)), all the related data for processing the timeline operation are totally clustered in a single node by the identifying relationships. Therefore, we need to access only one node to process the timeline operation. For the newsfeed operation(see Figure 13(b)), however, the related data are clustered by the sequence of identifying relationships except for the sequences of User—is_friend_of—User and User—is_member_of—Group. Thus, we have to access a significantly smaller number of nodes to process the newsfeed operation than without clustering. As a result, we can effectively decrease the number of nodes to be accessed by the timeline and newsfeed operations, which comprise a majority of SNS operations.



| User | user_id | name | … |
|---|---|---|---|
| | 1 | John | |
| | 2 | Bill | |
| | 3 | Marie | |

| Post | user_id | group_id | post_id | writer | content |
|---|---|---|---|---|---|
| | 1 | null | 1 | John | Hello friends! |
| | 1 | null | 2 | John | Thank you. |

| Comment | user_id | group_id | comment_id | writer | post_id | content |
|---|---|---|---|---|---|---|
| | 1 | null | 1 | Bill | 2 | My pleasure. |
| | 1 | null | 2 | Marie | 2 | Welcome |

node 1:

| User | user_id | name |
|---|---|---|
| | 1 | John |

| Post | user_id | group_id | post_id | writer | content |
|---|---|---|---|---|---|
| | 1 | null | 1 | John | Hello friends! |
| | 1 | null | 2 | John | Thank you. |

| Comment | user_id | group_id | comment_id | writer | post_id | content |
|---|---|---|---|---|---|---|
| | 1 | null | 1 | Bill | 2 | My pleasure. |
| | 1 | null | 2 | Marie | 2 | Welcome |

node 2:

| User | user_id | name |
|---|---|---|
| | 2 | Bill |

| Post | user_id | … | content |
|---|---|---|---|

| Comment | user_id | … | content |
|---|---|---|---|

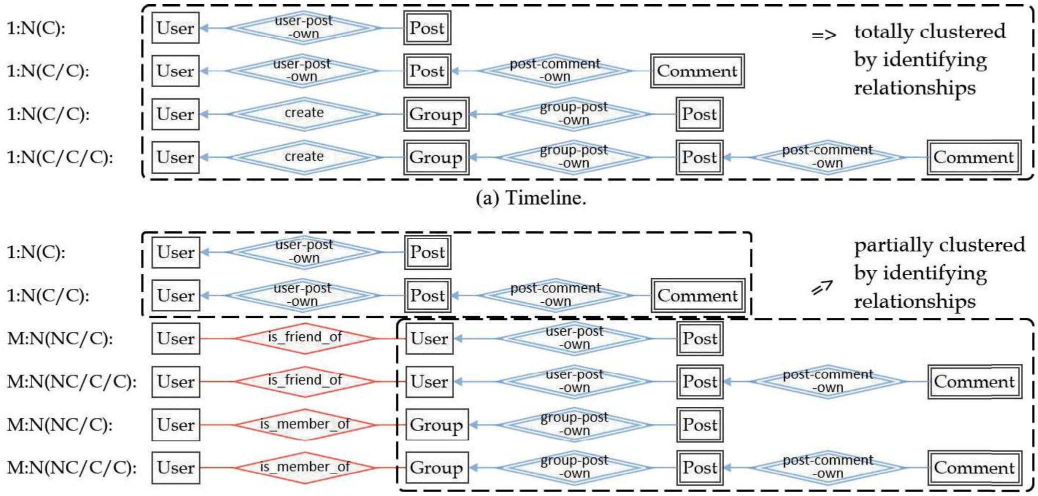
node 3:

| User | user_id | name |
|---|---|---|
| | 3 | Marie |

| Post | user_id | … | content |
|---|---|---|---|

| Comment | user_id | … | content |
|---|---|---|---|

Figure 12: An example of partitioning relations by identifying key.



(a) Timeline.

(b) Newsfeed.

C: “clustered”. NC: “not clustered”.

Figure 13: SNS timeline and newsfeed operations(clustered by identifying relationships).

The hierarchical clustering method based on identifying relationships can be used for any large-scale systems whose entities are distributed over multiple nodes based on hashing. For any such systems, we identify frequently traversed sequences of one-to-many relationships and model them as sequences of identifying relationships. This way, we can realize hierarchical clustering of entities in one node making a transaction accessing all entities related by a sequence of identifying relationships a single-node transaction. Examples are E-commerce systems such as Amazon, Ebay, and Auction where User, Transaction, and

Item can be modelled as entity types related by a sequence of identifying relationships (User-Transaction and Transaction-Item).

### 3.3 Many-to-many relationship

Figure 14 shows many-to-many relationships in the simplified SNS entity-relationship schema (Figure 8). Unlike in an one-to-many relationship, in a many-to-many relationship, tuples that are accessed together cannot be stored in the same relation. Thus, inter-node joins cannot be avoided. More importantly, multi-node updates (which require two-phase commit) are needed for update transactions. Our solution to this problem is (a) to decompose a global transaction into a set of single-node-update transactions without two-phase commit (defined later in this section) through careful analysis of transaction semantics.

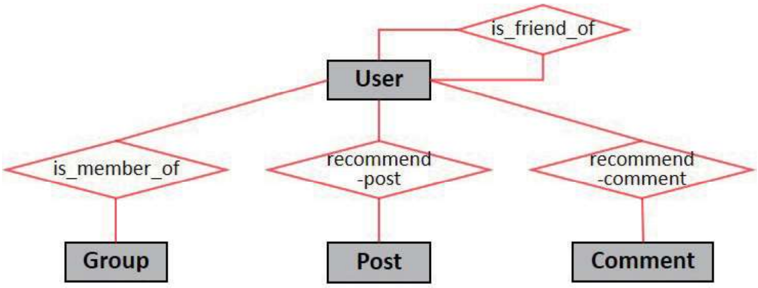
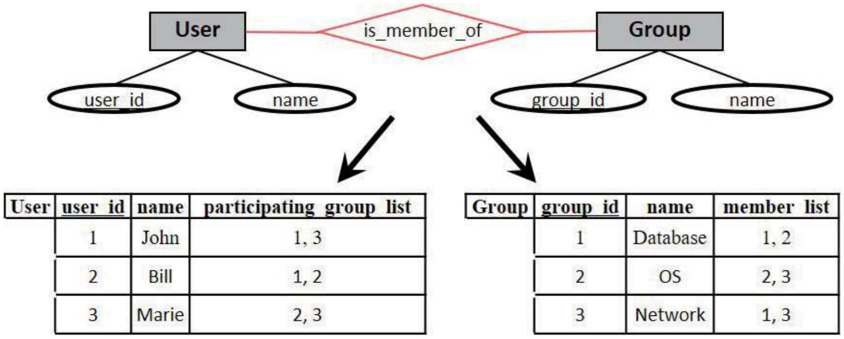


Figure 14: Many-to-many relationships in the SNS entity-relationship schema of Figure 8.

Instead of introducing a third relationship table, in Odysseus/SNS, we simplify the relational schema for a many-to-many relationship by representing it directly through a set (or list) type, which is an object-relational feature available in the DBMS. Figure 15 shows a mapping for a many-to-many relationship, is_member_of between User and Group. This direct mapping has the effect of performing pre-joins between the entity relations (i.e., User, Group) and the relationship relation (i.e., is_member_of) to pre-populate related tuples in each tuple of the entity relations.

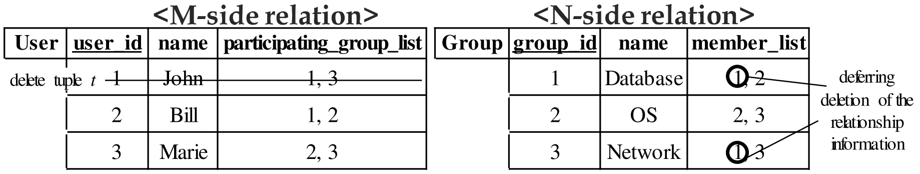


| User | user_id | name | participating_group_list |
|---|---|---|---|
| | 1 | John | 1, 3 |
| | 2 | Bill | 1, 2 |
| | 3 | Marie | 2, 3 |

| Group | group_id | name | member_list |
|---|---|---|---|
| | 1 | Database | 1, 2 |
| | 2 | OS | 2, 3 |
| | 3 | Network | 1, 3 |

Figure 15: Mapping a many-to-many relationship using a set type.

For delete transactions, we use a deferred delete strategy to complete the transactions without two-phase commit. The delete operations (deactivate, delete a group, and delete a post in the experiment in Section 4) need to delete both of a tuple and the references to it. The deferred delete strategy immediately delete the tuple in one node and defer deletion of the references to that tuple in other node until they try to access that tuple. For instance, Figure 16 shows the scenario of an example deferred delete transaction. When the tuple $t$ (with user_id 1) is deleted from the M-side relation, the id 1 must be dropped from the list of id's in each related tuple of the N-side relation. The tuples updated on the N-side have the group_id 1 and 3, respectively, and both tuples contain the user id 1 in their member_list. Hence, a two-phase commit over multiple nodes would be required to complete the deletion operation. However, under the deferred delete strategy, at the time $t$ is deleted, the transaction ends without deleting the references to $t$ in the N-side tuples and, instead, delete a reference later when some transaction tries to access the tuple $t$ through the member_list of the N-side relation (e.g., when executing the view member list operation). We note that the delete operations under the deferred delete strategy ar single-node-update transactions. Although the operations accessing the tuples from the references are two-node transactions, the update is processed in only one node without two-phase commit. Hence, we call these type of operations *single-node-update transactions*. In effect, we are decomposing a multi-node transaction into multiple single-node or single-node-update transactions executed on demand. We note that the deferred delete does not affect the consistency of the original multi-node delete using two-phase commit since, at the time the tuple $t$ is deleted, the references to $t$ in the N-side tuples are effectively immediately invalidated (i.e., they are pointing to a non-existing tuple). That is, deleting the references to $t$ in the N-side tuples can be considered only a post-transaction operation(amounting to garbage collection) that does not affect the transaction semantics.



<M-side relation>

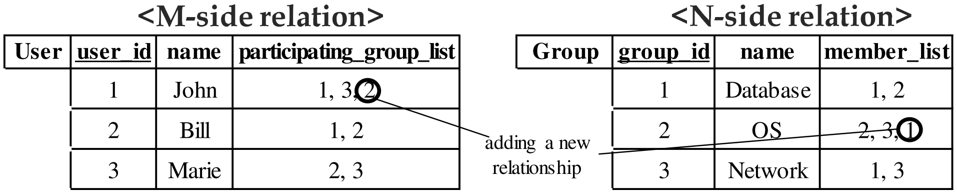
| User | user_id | name | participating_group_list |
|---|---|---|---|
| delete tuple $t$ | ~~1~~ | ~~John~~ | ~~1, 3~~ |
| | 2 | Bill | 1, 2 |
| | 3 | Marie | 2, 3 |

<N-side relation>

| Group | group_id | name | member_list |
|---|---|---|---|
| | 1 | Database | 1, 2 |
| | 2 | OS | 2, 3 |
| | 3 | Network | 1, 3 |

Figure 16: An example of deferred delete of a relationship.

For insert transactions, deferred commit is not an option. As shown in Figure 17, we need to update the list in the tuples on both sides of the relationship. (The same situation happens for "unclustered" one-to-many relationships, that is, those not modelled as identifying relationships.) However, the overhead of this two-phase commit is not significant because update transactions like this one involve only *two-node* transactions since all relationships in the SNS database (see Figure 8) are binary. (Identifying relationships for clustering a sequence of relations through the identifying key are an exception, but they involve only single-node transactions that do not need two-phase commit.)



**<M-side relation>**

| User | user_id | name | participating_group_list |
|---|---|---|---|
| | 1 | John | 1, 3, 2 |
| | 2 | Bill | 1, 2 |
| | 3 | Marie | 2, 3 |

**<N-side relation>**

| Group | group_id | name | member_list |
|---|---|---|---|
| | 1 | Database | 1, 2 |
| | 2 | OS | 2, 3, 1 |
| | 3 | Network | 1, 3 |

Figure 17: An example of inserting a new relationship.

# 4 Performance Evaluation

The objective of performance evaluation is to examine the benefit of the methods proposed in reducing the overhead of multi-node access in a global transaction. The first set of experiments (Section 4.2) examines how much the query response times of important SNS operations are reduced with respect to a designated reference operation in Odysseus/SNS when compared with SNS-A – a widely used commercial SNS. The second set of experiments examines how Odysseus/SNS's performance is affected when the query arrival rate is increased (i.e., "scaled up") (Section 4.3) and when the number of system nodes is varied (i.e., "scaled out") (Section 4.4).

## 4.1 Workload model

Odysseus/SNS used in the experiments consists of nine nodes – one master node (with 2.93GHz quadcore CPU, 4GB memory, and a 500GB hard disk) and eight slave nodes (each with 3.2GHz quadcore CPU, 8GB memory, and a 2TB hard disk). Each slave runs its own local DBMS and communicates with the master over the network, and performs tasks assigned by the master and returns the result it.

We built a workload model for generating a synthetic SNS workload (data set, query set) that reflects the real-life scale workload per node. The "real-life scale workload" was constructed based on the SNS-A system's published statistics and reports[1, 4, 8, 9, 13, 18, 22] as well as our own survey. The parameters used in the workload model are shown in Table 3, where the sources of published/surveyed parameters are indicated.

Almost all published parameter values are dated around the year 2010, which was selected as a common time point for the values. We then followed their distributions (e.g., power law) and scaled the size of the data set to that used in LinkBench(more specifically, the total number of objects, which is 1.2 billion per node) [1]. As a result, the size of the database is 0.23 TBytes per node[2].

[2] This is equivalent to 13.03 petabytes in total if we use 58,000 nodes in the entire system as in a real-life system(see Table 3).

Table 3: Workload parameters and the sources of their values (if available).

| **Parameter** | **Description** | **Value** |
|---|---|---|
| $N_{servers}$ | Total number of servers | 60 thousand [8] |
| $N_{log\_servers}$[a] | Total number of log servers | 2 thousand [22] |
| $N_{nodes}$ | Total number of nodes (i.e., data servers) | |
| $N_{users}$ | Total number of users | 350 million [13] |
| $N_{groups}$ | Total number of groups | 620 million [18] |
| $N_{posts}$ | Total number of posts stored | |
| $N_{comments}$ | Total number comments stored | |
| $N_{photos}$ | Total number of photos stored | 260 million [4] |
| $NPD_{write\_post}$ | Number of writing post per day | |
| $NPD_{write\_comment}$ | Number of writing comment per day | 300 million [13] |
| $NPD_{modify\_post}$ | Number of modifying post per day | |
| $NPD_{modify\_comment}$ | Number of modifying comment per day | |
| $NPD_{delete\_post}$ | Number of deleting post per day | |
| $NPD_{delete\_comment}$ | Number of deleting comment per day | |
| $NPD_{recommend\_post}$ | Number of recommending post per day | 105 million [13] |
| $NPD_{recommend\_comment}$ | Number of recommending comment per day | |
| $NPD_{view\_timeline}$ | Number of viewing user/group timeline per day | |
| $NPD_{view\_newsfeed}$ | Number of viewing newsfeed per day | |
| $NPD_{view\_full\_photo}$ | Number of viewing full photo image per day | |
| $NPD_{upload\_photo}$ | Number of uploading photo per day | 142.8 million [4] |
| $NPD_{view\_photo}$ | Number of viewing photo per day | 90 billion [4] |
| $NPD_{all\_queries}$ | Total number of queries per day | |
| $R_{modify\_post}$ | Rate of modification after post | 13.2%[b] |
| $R_{modify\_comment}$ | Rate of modification after comment | 8.6%[b] |
| $R_{delete\_post}$ | Rate of deletion after post | 9.7%[b] |
| $R_{delete\_comment}$ | Rate of deletion after comment | 9.1%[b] |
| $R_{recommend\_post}$ | Rate of recommendation of post among all recommendations | 74.9%[b] |
| $R_{recommend\_comment}$ | Rate of recommendation of comment among all recommendations | 25.1%[b] |
| $R_{view\_timeline}$ | Rate of user/group timeline viewing among all post viewings | 28.7%[b] |
| $R_{view\_newsfeed}$ | Rate of newsfeed viewing among all post viewings[c] | 71.3%[b] |
| $R_{view\_thumbnail}$ | Rate of thumbnail viewing among all photo viewings | 84.4% [4] |
| $R_{view\_full\_photo}$ | Rate of full photo viewing among all photo viewings | 5.2% [4] |
| $R_{photo\_in\_post}$ | Rate of photo being contained in a post | 75% [9] |
| $FRIENDS\_PER\_USER$ | Average number of friends per user | 44[d] |
| $GROUPS\_PER\_USER$ | Average number of groups per user | 12 [13] |
| $POSTS\_PER\_PAGE$ | Number of posts per message board page | 10[e] |

[a]: Hadoop HDFS cluster for log server (as opposed to MySQL cluster for data server) in SNS-A

[b]: rates obtained through a web-based survey participated by 108 users

[c]: $R_{view\_timeline} + R_{view\_newsfeed} = 100\%$

[d]: an average obtained from SNAP dataset [14]

[e]: a default number of posts per page on SNS-A

### SNS objects

The data set parameters ($N_{users}$, $N_{groups}$, $N_{posts}$, $N_{comments}$, $N_{photos}$) in Table 3 reflect the data scale of an SNS system. Among them, the base values of $N_{users}$, $N_{groups}$, and $N_{photos}$ were obtained from the published sources of SNS-A (see 'Value" in Table 3), and the base values of $N_{posts}$ and $N_{comments}$ were derived as follows.

- $N_{posts} = N_{photos}/R_{photo-in-post}$ = 260 million/0.75 = 347 billion
- $N_{comments} = N_{posts} \times (NPD_{write\_comment}/NPD_{write\_post})$ = 347 billion × (300 million/190 million) = 547 billion (see SNS relationships below for the value of $NPD_{write\_post}$)

Table 4 shows the base values and the proportions of SNS objects of different types. We then scaled up the numbers of objects of individual types in Odysseus/SNS so that the total number of objecs (except photos) per node is 1.2 billion according to the LinkBench benchmark [1].

Table 4: The proportions and numbers of SNS objects in the data set.

| **SNS object type** | **Base number of objects** | **Proportion** | **Actual number of objects per node** |
|---|---|---|---|
| User | 350 million | 0.03% | 470.34 thousand |
| Group | 620 million | 0.05% | 826.80 thousand |
| Post | 347 billion | 30.04% | 466.44 million |
| Comment | 547 billion | 47.37% | 735.54 million |
| Photo | 260 million | 22.51% | 349.44 million |

The total number of all objects(except photos) per node = 1.2 billion (LinkBench [1]).
The number of system nodes = 8.

Individual SNS objects were instantiated as summarized in Table 5. SNS entities (i.e., users, groups, posts, comments) are instantiated as tuples in the corresponding relations, and photos are instantiated as files.

### SNS relationships

All SNS relationships follow the power law [6, 12, 15, 21], so each relationship is instantiated randomly according to the power law distribution, where the mean values are set as summarized in Table 6. For one-to-many (1:N) relationships, the mean of N-side distribution is calculated from the numbers in Tables 3 and 4. For many-to-many (M:N) relationships, the means of the distributions on the M-side and N-side, denoted as $\overline{M}$ and $\overline{N}$, respectively, are obtained as follows (see Table 3 for the workload parameters and values).

- Is_member_of:

$$\overline{M} = GROUPS\_PER\_USER = 12$$
$$\overline{N} = GROUPS\_PER\_USER \times \frac{N_{users}}{N_{groups}} = 12 \times \frac{350 \text{ million}}{620 \text{ million}} = 7$$

Table 5: Instantiation of SNS objects in the data set.

| SNS object type | Attribute | Instantiation |
|---|---|---|
| User | name | randomly selected from list on the Web [26] |
| | job | randomly selected from list on the Web [27] |
| | password | 10-digit random number |
| | phone number | 10-digit random number |
| | nickname | "Name" followed by 4-digit random number |
| Group | name | 5-character random alphabet string followed by 5-digit random number |
| Post | content | string of random length extracted from the book "Pride and Prejudice" [3], where the random length follows the distribution in Nierhoff [17] (see Figure 18). |
| Comment | content | string of random length extracted in the same way as Post using the same distribution but scaled down to 37, which is the average length of comments (obtained from 100 randomly selected comments randomly selected from popular timelines on SNS-A) |
| Photo | N/A | image randomly selected from a website [16] and set to the size of 8KB for thumbnail image and 64KB for full image, the same as in Beaver et al. [4] |

- Is_friend_of:
  $\overline{M} = \overline{N} = \mathit{FRIENDS\_PER\_USER} = 44$

- Recommend_post:
  $\overline{M} = \frac{NPD_{recommend\_post}}{NPD_{write\_post}} = \frac{NPD_{recommend\_post}}{NPD_{photos\_updated}/R_{photo\_in\_post}} = \frac{105 \text{ million}}{142.8 \text{ million}/0.75} = 0.55$
  $\overline{N} = N_{posts} \times \frac{NPD_{recommend\_post}}{NPD_{write\_post}} \times \frac{1}{N_{users}} = 347 \text{ billion} \times \frac{105 \text{ million}}{190 \text{ million}} \times \frac{1}{350 \text{ million}} = 547$

- Recommend_comment:
  $\overline{M} = \frac{NPD_{recommend\_comment}}{NPD_{write\_comment}} = \frac{35.1 \text{ million}}{300 \text{ million}} = 0.03$ (see SNS queries below for $NPD_{recommend\_comment}$)

$$\begin{aligned}\overline{N} &= N_{comments} \times (NPD_{recommend\_post} \\ &\times \frac{R_{recommend\_comment}}{R_{recommend\_post}}) \times \frac{1}{NPD_{write\_comment}} \times \frac{1}{N_{users}} \\ &= 547 \text{ billion} \times (105 \text{ million} \times \frac{0.251}{0.749}) \times \frac{1}{300 \text{ million}} \times \frac{1}{350 \text{ million}} = 183\end{aligned}$$

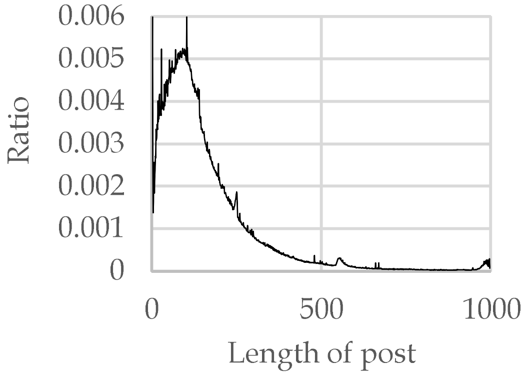


Figure 18: Distribution of the lengths of posts [17].

Table 6: Instantiation of SNS relationships in the data set.

(a) One-to-many (1:N) relationships.

| **1-side relation** | **N-side relation** | **Relationship** | **Mean of N-side distribution** |
|---|---|---|---|
| User | Post | write_post | 991 $(=N_{posts}/N_{users})$ |
| User | Comment | write_comment | 1562 $(= N_{comments}/N_{users})$ |
| User | Group | create | 1.77 $(=N_{groups}/N_{users})$ |
| User, Group | Post | own | 357 $(= N_{posts}/(N_{users} + N_{groups}))$ |
| Post | Comment | own | 1.5 $(=N_{comments}/N_{posts})$ |

(b) Many-to-many (M:N) relationships.

| **M-side relation** | **N-side relation** | **Relationship** | **Mean of M-side distribution** | **Mean of N-side distribution** |
|---|---|---|---|---|
| User | Group | is_member_of | 7 | 12 |
| User | User | is_friend_of | 44 | 44 |
| User | Post | recommend_post | 0.55 | 547 |
| User | Comment | recommend_comment | 0.03 | 183 |

## SNS queries

For the SNS query set, 10,000 queries of primary operations (see Table 2) are randomly generated in proportion to the distribution of the frequencies of SNS-A queries. (Queries of the secondary operations are not included because they are executed too infrequently to avail any workload statistics.) The frequencies of queries per day for each operation are obtained as follows (see Table 3 for the workload parameters and values).

- $NPD_{write\_post} = \frac{NPD_{upload\_photo}}{R_{photo\_in\_post}} = \frac{142.8 \text{ million}}{0.75} = 190$ million
- $NPD_{write\_comment} = 300$ million
- $NPD_{modify\_post} = NPD_{write\_post} \times R_{modify\_post} = \frac{NPD_{update\_photo}}{R_{photo\_in\_post}} \times R_{modify\_post} = \frac{142.8 \text{ million}}{0.75} \times 0.132 = 25.13$ million
- $NPD_{modify\_comment} = NPD_{write\_comment} \times R_{modify\_comment} = 300$ million $\times\ 0.086 = 25.8$ million
- $NPD_{delete\_post} = NPD_{write\_post} \times R_{delete\_post} = \frac{NPD_{upload\_photo}}{R_{photo\_in\_post}} \times R_{delete\_post} = \frac{142.8 \text{ million}}{0.75} \times 0.097 = 18.46$ million
- $NPD_{delete\_comment} = NPD_{write\_comment} \times R_{delete\_comment} = 300$ million$\times 0.091 = 27.3$ million
- $NPD_{recommend\_post} = 105$ million
- $NPD_{recommend\_comment} = NPD_{recommend\_post} \times \frac{R_{recommend\_comment}}{R_{recommend\_post}} = 105$ million $\times\ \frac{0.251}{0.749} = 35.1$ million
- $NPD_{view\_timeline} = NPD_{view\_photo} \times \frac{R_{view\_thumbnail}}{R_{photo\_in\_post}} \times R_{view\_timeline} \times \frac{1}{POSTS\_PER\_PAGE} =$ 90 billion $\times\ \frac{0.844}{0.85} \times 0.287 \times \frac{1}{10} = 2.9$ billion
- $NPD_{view\_newsfeed} = NPD_{view\_photo} \times \frac{R_{view\_thumbnail}}{R_{photo\_in\_post}} \times R_{view\_newsfeed} \times \frac{1}{POSTS\_PER\_PAGE} =$ 90 billion $\times\ \frac{0.844}{0.75} \times 0.713 \times \frac{1}{10} = 7.22$ billion
- $NPD_{view\_full\_photo} = NPD_{view\_photo} \times R_{view\_full\_photo}$ 90 billion $\times\ 0.052 = 4.5$ billion

The query arrival rate per day to the SNS-A system, $NPD_{total}$, is the summation of the frequencies of all SNS operations, which equals 15.22 billion per day. For each query operation $qo$, the proportion of its query frequency is $\frac{NPD_{qo}}{NPD_{total}}$, as summarized in Table 7.

### Performance measure

The performance measure is the average query response time for each SNS query operation in Table 2. Queries are issued by a separate machine, which issues queries to the master through network. We first measure the performance of Odysseus/SNS both in warm start and in cold start. For warm start, the query arrival rate is varied from 1.0 to 3.5 million queries per day with Poisson distribution.

We note that the proportionally scaled-down query arrival rate for an 8-node Odysseus/SNS, which is equivalent to SNS-A's 15.22 billion queries per day(= $NPD_{total}$ above in SNS queries) for 58,000 nodes(= $N_{nodes}$, obtained as $N_{servers} - N_{log\_servers}$; see Table 3), is 2.09 million queries per day. For cold start, the query arrival rate is varied from 0.2 to 1.0 million queries per day (1.0 million is the maximum query arrival rate that Odysseus/SNS can handle with cold start). We expect that the average query response time of Odysseus/SNS

Table 7: The proportions of different query operations.

| SNS query type | Proportion |
|---|---|
| write post | 1.25% |
| write comment | 1.97% |
| modify post | 0.13% |
| modify comment | 0.17% |
| delete post | 0.15% |
| delete comment | 0.18% |
| recommend post | 0.69% |
| recommend comment | 0.2% |
| view timeline | 16.62% |
| view newsfeed | 49.08% |
| view full photo | 29.57% |

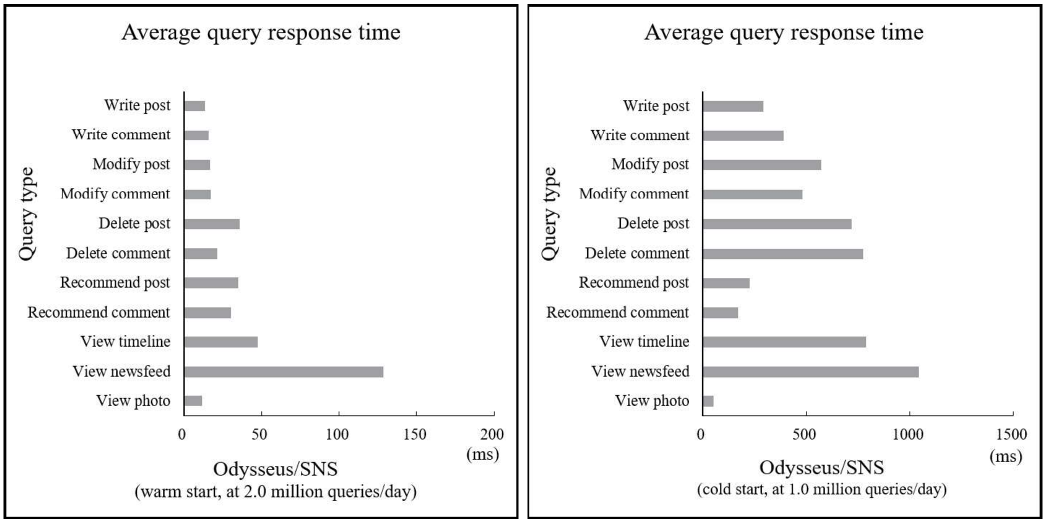
in the real-world environment will show between those of warm start and cold start(as a reference, SNS-A processes 95% of queries in memory [23].)

We then compare the "relative" performance among various query operations of SNS-A with that among those of Odysseus/SNS. Direct comparison with SNS-A is not relevant because of the difference in the scale (e.g., number of nodes) and the computing environment. Thus, when comparing with SNS-A, the query response times are normalized by that of the reference operation ("recommend comment" for primary query operations, "login" for secondary query operations) for SNS-A and Odysseus/SNS to compare relative performances among various query operations regardless of the scales. We choose the reference operation that is simple so that it operates similarly independent of the architecture. The focus of this comparison is on examining the effects of the particular techniques that are used in Odysseus/SNS but not in SNS-A. To measure the query response time in SNS-A, queries are repeated at different times to cover users in different time zones of the world clock[3]. In each measurement, to measure the server processing time only, the round trip time (i.e., ping time) between the web sever and SNS-A is subtracted from the elapsed time.

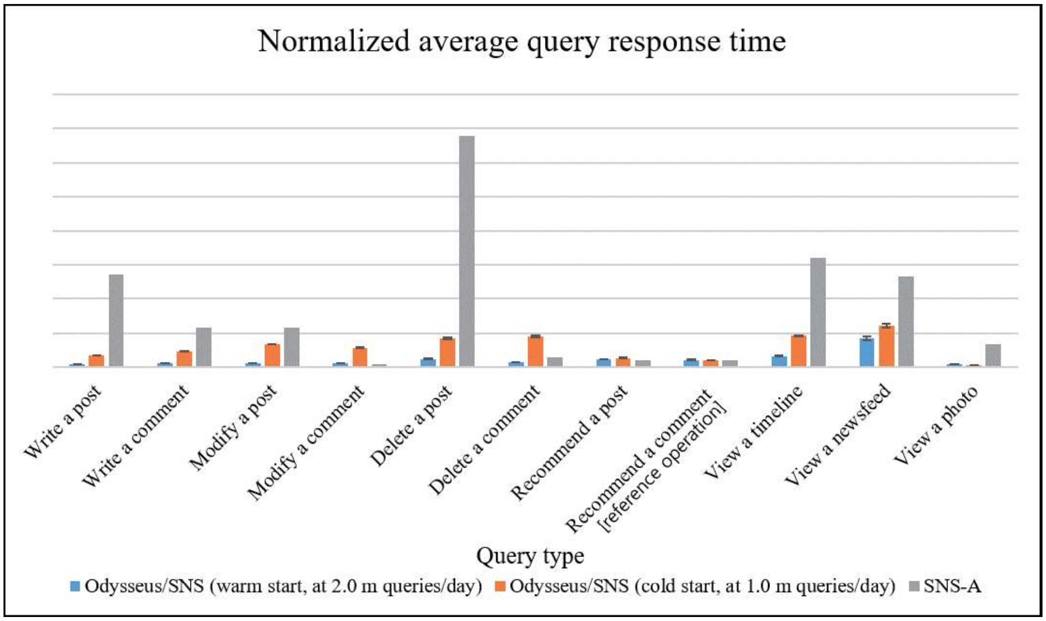
## 4.2 Query response times in Odysseus/SNS and SNS-A

Figure 19 shows the query response times in each system for the primary query operations. For Odysseus/SNS, the arrival rates are at 2.0 million queries per day for warm start and 1.0 million queries for cold start, and Figure 20 shows those for the secondary query operations. For Odysseus/SNS at warm start and cold start, we also show standard deviations in error bars.

[3] We measured the query response time of each operation six times on Jan. 28th, 2017, with a 4-hour interval(i.e., at 0, 4, 8, 12, 16, and 20 o'clock PT). In the case of "sign up" and "deactivate", we measured it only once (at 0 o'clock) since repeated trial of those operations is not allowed by SNS-A.



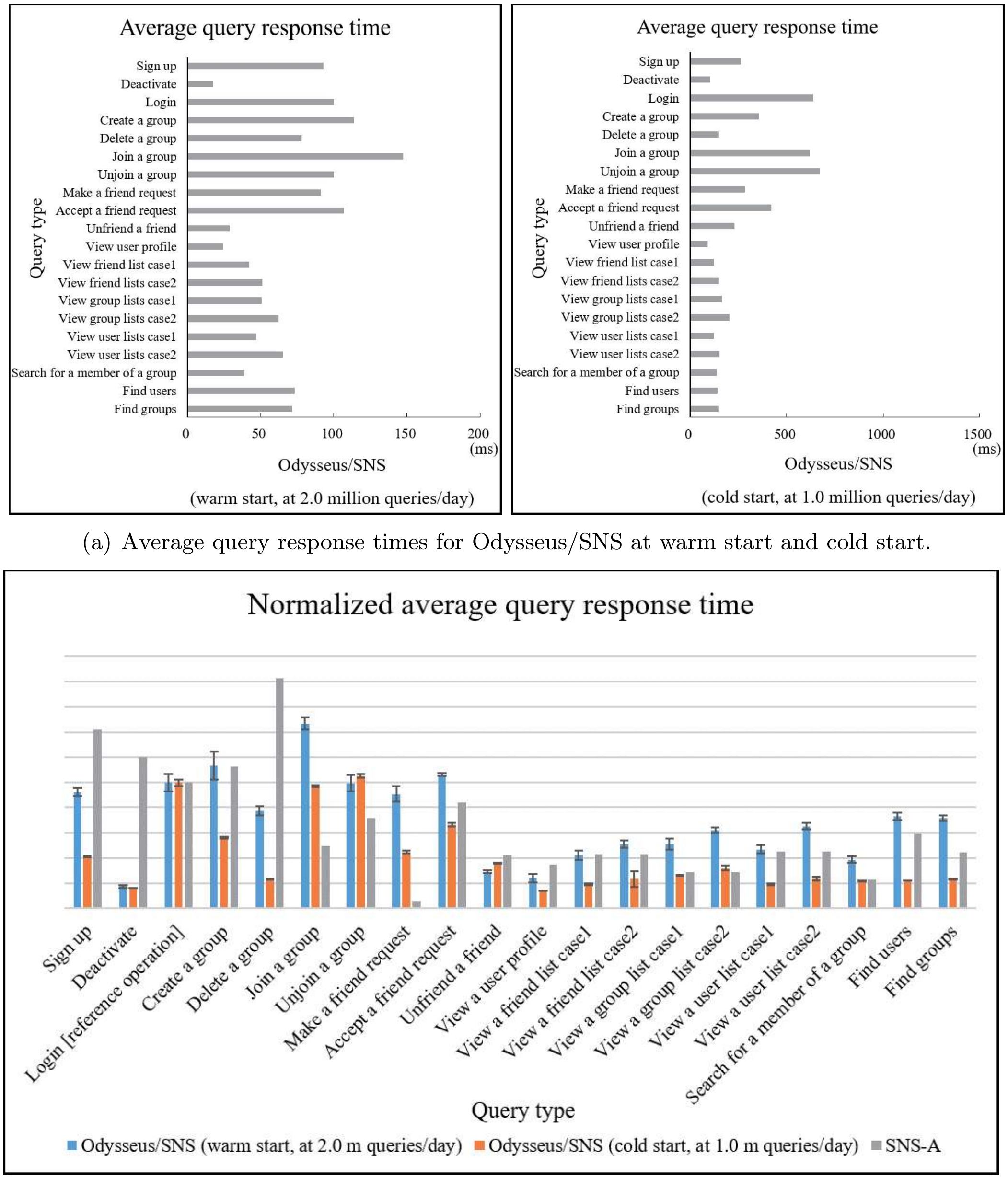
(a) Average query response times for Odysseus/SNS at warm start and cold start.



(b) Clustered bar graph comparing Odysseus/SNS at warm start and cold start and SNS-A without vertical scale.

Figure 19: Average query response times of the primary query operations.

In Figure 19, the normalized processing times of newsfeed and timeline, which together take about 66% of query load, are significantly reduced (by 53.8∼68.0% for newsfeed; 71.0∼90.1% for timeline) in Odysseus/SNS compared with those of SNS-A. This reduction is a direct benefit of clustering relations by an identifying key as we have explained in Section 3.2(Fig.13).



(a) Average query response times for Odysseus/SNS at warm start and cold start.

(b) Clustered bar graph comparing Odysseus/SNS at warm start and cold start and SNS-A without vertical scale.

Figure 20: Average query response times of the secondary query operations. Since cases 1 and 2 are not differentiated for SNS-A, we show the same data for both cases.

In addition, the processing time of “delete a post” in Figure 19 and “delete a group” and “deactivate” in Figure 20 show that the processing times of delete operations are drastically reduced (by 87.5∼96.5% for delete a post;by 57.5∼87.3% for delete a group; 85.3∼86.3% for deactivate), which is attributed to the deletion cost amortized under the deferred delete strategy. With this strategy, "view friend lists" and "view user lists" take a bit longer "after deactivate" if they find that the user has deactivated the account. Likewise, "view group lists" may take a bit longer after "delete a group" if it finds that the group has been deleted. This increase of response time is a direct consequence of amortization. Comparing the response time between the cases of deferred deletion occurring (case 2) and not occurring (case 1) showed no more than 39% difference, which was not significant.

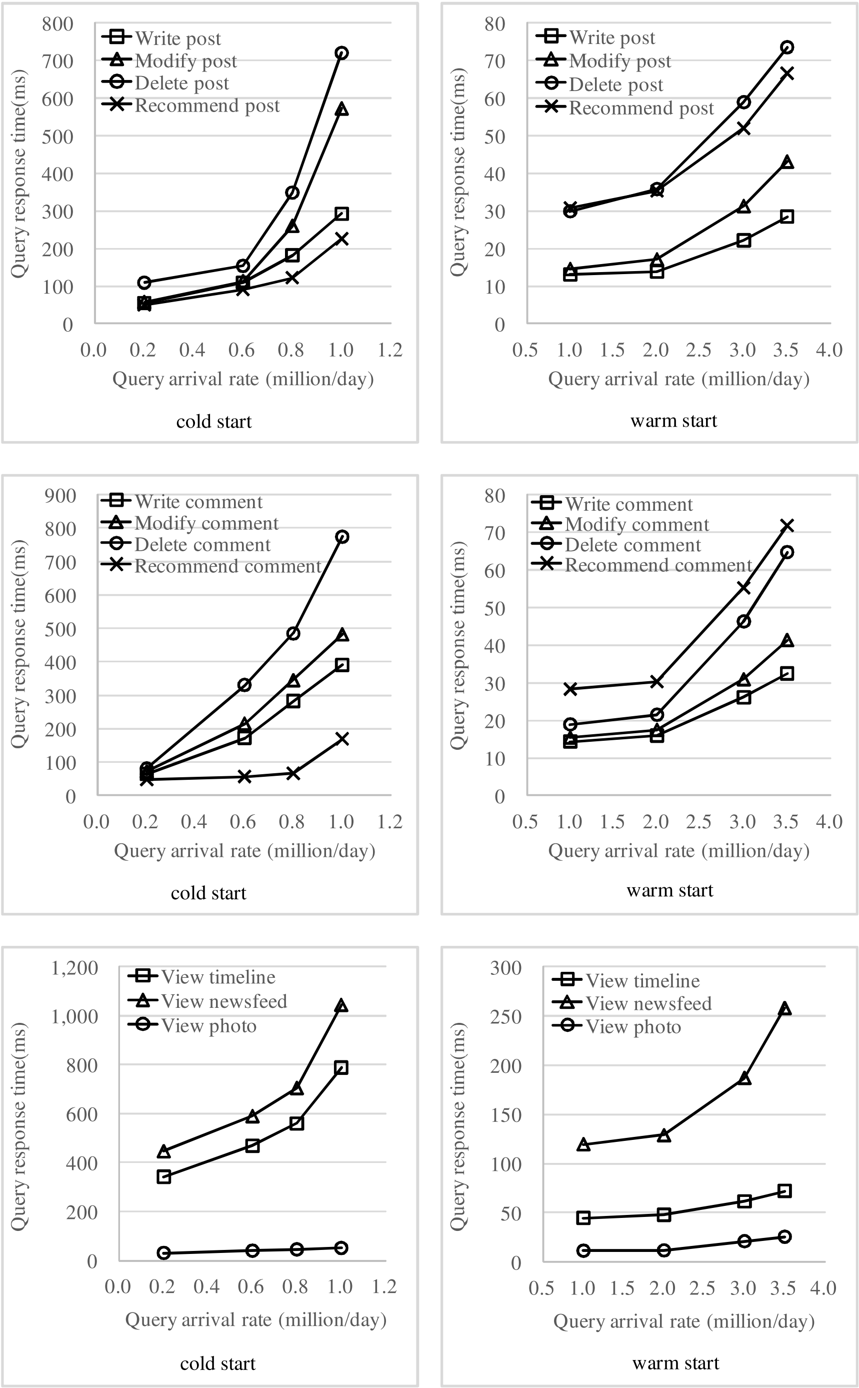
## 4.3 Query response times of Odysseus/SNS as the query arrival rate is varied

Figure 21 shows the result for all primary query operations as the query arrival rate is varied from 1.0 to 3.5 million queries per day at warm start and from 0.2 to 1.0 million queries per day at cold start. The results for the secondary query operations are similar and omitted here to avoid redundancies. For every query operation, the query response time increases as the query arrival rate increases. This trend makes sense considering the queuing effect on each node as the query arrival rate increases. Experiments show that Odysseus/SNS can process up to 3.5 million queries per day, which is larger than the 2.09 million queries per day that we have estimated for an 8-node equivalent of SNS-A's query load (see "Performance measure" in Section 4.1). This result demonstrates Odysseus/SNS's ability to handle query arrival rates typical of the SNS workload.

## 4.4 Query response times of Odysseus/SNS as the number of system nodes increases

Figure 22 shows the response time of primary query operations when the number of nodes varies, divided into three groups. For every query operation, it clearly shows that the query response time hardly changes as the number of nodes varies. (The mean and standard deviation of the range (i.e., maximum − minimum) over minimum for all query operations are only 3.00% and 1.90%, respectively.)[4] The reason for this is that, in most cases, the number of nodes that need to be accessed to execute a query is limited to be a small number and does not increase as the number of nodes increases, which is characteristic of SNS operations. We believe that the ideas (i.e., avoiding inter-node joins and multi-node two-phase commit) implemented in Odysseus/SNS also helps with scalability, due to its positive effect on reducing the number of nodes accessed during global transaction processing. The result thus suggests the potential of Odysseus/SNS to scale out as the number of system nodes increases.

[4] We present only the warm-start results since cold-start results are rather unstable due to random disk accesses.



cold start

warm start

cold start

warm start

cold start

warm start

Figure 21: Average response times of query operations in Odysseus/SNS as the query arrival rate is varied.

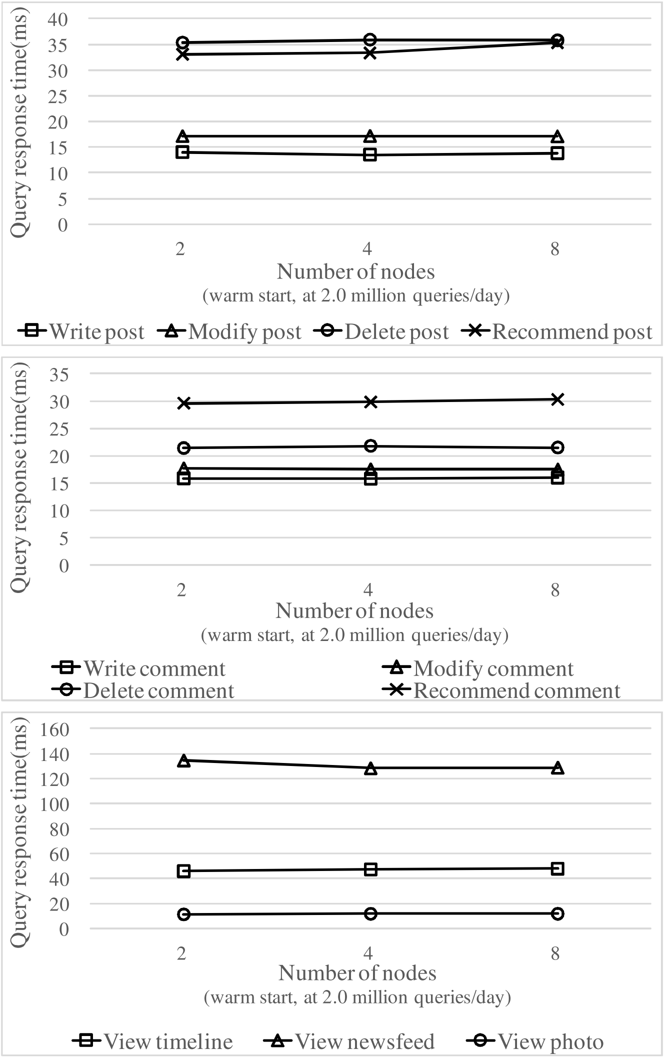


Figure 22: Average response times of query operations in Odysseus/SNS when the number of nodes is varied.

## 5 Conclusions

In this paper we have discussed an approach to implementing a large-scale social networking services (SNS) system by using a relational DBMS. In particular, we have shown a case where we can build a scaled-out system using a shared-nothing parallel DBMS. We have first proposed the entity-relationship conceptual model and its relational representation of the SNS database, and then, by using the high-level semantics provided by the schema, have proposed the methods processing global transactions involving multiple nodes as local transactions on single nodes as much as possible resulting in (1) avoiding joins across different nodes (for queries via one-to-many relationships) and (2) amortizing the cost of updates across different nodes (for deletion via many-to-many relationships). This paper is the first to present the entity-relationship schema and its relational representation of the SNS database. Performance evaluation, conducted using a synthetic workload of the scale of Linkbench[1], demonstrated the significant benefit of the methods, especially in timeline and newsfeed – the two dominant SNS query operations – and in various delete operations.

## 6 Acknowledgement

This work was supported by the National Research Foundation of Korea(NRF) grant funded by Korean Government(MSIT) (No. 2016R1A2B4015929).

This work has been recorded as a technical report of School of Computing of KAIST [25]. A short version in Korean has been published in Journal of The National Academy of Sciences Republic of Korea (NAS), which is a non-copyrighted technical report of NAS distributed to Government institutes and public and university libraries in Korea [25].

# References

[1] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: A database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196, New York, NY, USA, 2013.

[2] M. Aslett. How will the database incumbents respond to NoSQL and NewSQL? 451 TechDealmaker, The 451 group, April 2011.

[3] J. Austen. Pride and prejudice (ebook). https://www.gutenberg.org/ebooks/1342, referenced in November 2014.

[4] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX International Conference on Operating Systems Design and Implementation*, pages 1–8, Vancouver, Canada, October 2010.

[5] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 49–60, Berkeley, CA, USA, 2013.

[6] S. Catanese, P. De Meo, E. Ferrara, G. Fiumara, and A. Provetti. Extraction and analysis of Facebook friendship relations. In *Computational Social Networks*, pages 291–324. Springer, June 2012.

[7] R. Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, December 2010.

[8] Datacenter Knowledge. The Facebook data center FAQ. http://www.datacenterknowledge.com/the-facebook-data-center-faq, referenced in November 2014.

[9] Digital Marketing Ramblings. By the numbers: 85 amazing Facebook page statistics. http://expandedramblings.com/index.php/facebook-page-statistics, referenced in November 2014.

[10] R. Escriva, B. Wong, and E. Sirer. HyperDex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, October 2012.

[11] K. Grolinger, W. A. Higashino, A. Tiwari, and M. A. Capretz. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):1–24, 2013.

[12] H. Halpin and A. Capocci. The Berners-Lee hypothesis: power laws and group structure in Flickr. In *Proceedings of the 23rd International Conference on World Wide Web Companion*, pages 885–890, April 2014.

[13] P. Kirschner and A. Karpinski. Facebook and academic performance. *Computers in Human Behavior*, 26(6):1237–1245, 2010.

[14] J. Leskovec and A. Krevl. SNAP datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, referenced in November 2014.

[15] L. Muchnik, S. Pei, L. C. Parra, S. D. Reis, J. S. Andrade Jr., S. Havlin, and H. A. Makse. Origins of power-law degree distribution in the heterogeneity of human activity in social networks. *Scientific Report*, 3(1783):1–7, May 2013.

[16] D. Niblack. Imagebase (absolutely free photos). http://imagebase.net, referenced in November 2014.

[17] M. H. Nierhoff. Research: Short posts on Facebook, Twitter and Google+ seem to get more interactions. Quintly-Social Media Analytics Blog, https://www.quintly.com/blog/2013/12/short-posts-on-facebook-twitter-google-more-interactions, referenced in November 2014.

[18] N. O'Neill. Google now indexes 620 million Facebook groups. SocialTimes, http://allfacebook.com/google-now-indexes-620-million-facebook-groups_b10520, referenced in November 2014.

[19] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 165–178, New York, NY, USA, 2009.

[20] M. Stonebraker, D. 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, Jan. 2010.

[21] R. A. van Compernolle and L. B. Abraham. Interactional and discursive features of English language Weblogs for language learning and teaching. In *Electronic Discourse in Language Learning and Language Teaching*, pages 191–212. John Benjamins, 2009.

[22] J. Vijayan. Facebook moves 30-petabyte Hadoop cluster to new data center. Computer World,

http://www.computerworld.com/s/article/9218752/Facebook_moves_30_petabyte_Hadoop_-cluster_to_new_data_center, referenced in November 2014.

[23] S. Warmuta and D. Delfrate. Memcached. Slideshare, http://www.slideshare.net/harpastum/memcached-2699652 (referenced in November 2014), November 12, 2009.

[24] K. Whang, J. Lee, M. Lee, W. Han, M. Kim, and J. Kim. DB-IR integration using tight-coupling in the Odysseus DBMS. *World Wide Web Journal*, 18(3):491–520, May 2015.

[25] K. Whang, T. Yun, J. Park, K. Cho, S. Kim, I. Yi, I. Na, and B. Lee. Building social networking services systems using the relational shared-nothing parallel DBMS. CS-TR-2018-419, School of Computing, KAIST, August 2018. Also, in Journal of The National Academy of Sciences Republic of Korea, Natural Sciences, Vol. 57, No. 2, Dec. 2018(in Korean).

[26] Wikipedia. Anthroponymy. http://en.wikipedia.org/wiki/Anthroponymy, referenced in November 2014.

[27] Wikipedia. List of occupations. http://en.wikipedia.org/wiki/Lists_of_occupations, referenced in November 2014.

# Appendix

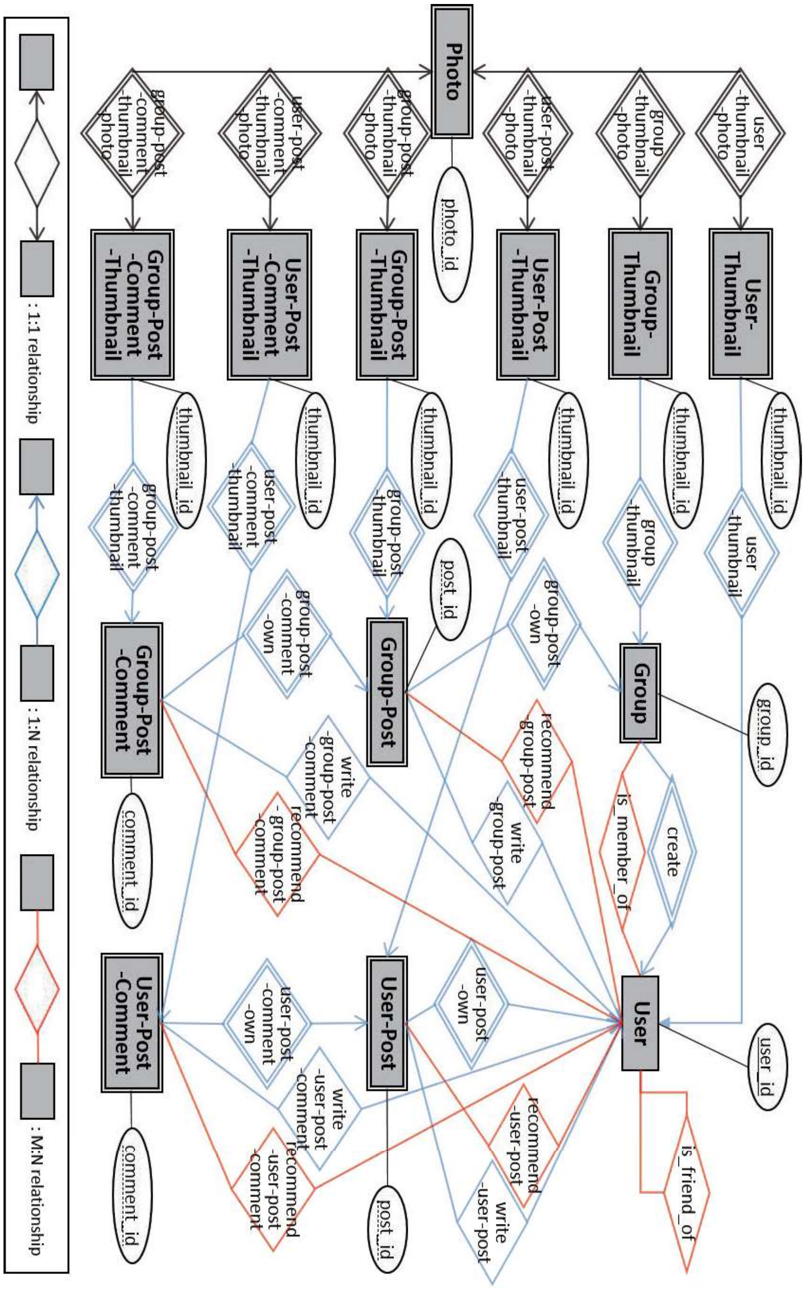


Figure 23: Entire SNS entity-relationship schema.