Data & Knowledge Engineering 125 (2020) 101756

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

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, VT, USA

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 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.

Fig. 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 [1]. In this regard, there has been a transition from NoSQL to NewSQL recently [2,3]. 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 [4,5].

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 [6,7] in the key–value format, which is NoSQL. This graph model represents data at such a low level

* Corresponding author.
E-mail addresses: kywhang@mozart.kaist.ac.kr (K.-Y. Whang), ijna@mozart.kaist.ac.kr (I. Na), tsyun@mozart.kaist.ac.kr (T.-S. Yun), jinah@mozart.kaist.ac.kr (J.-A. Park), khcho@mozart.kaist.ac.kr (K.-H. Cho), sjkim@mozart.kaist.ac.kr (S.-J. Kim), iyyi@mozart.kaist.ac.kr (I. Yi), bslee@cs.uvm.edu (B.S. Lee).

https://doi.org/10.1016/j.datak.2019.101756
Received 21 November 2018; Received in revised form 27 September 2019; Accepted 4 October 2019
Available online 28 October 2019
0169-023X/

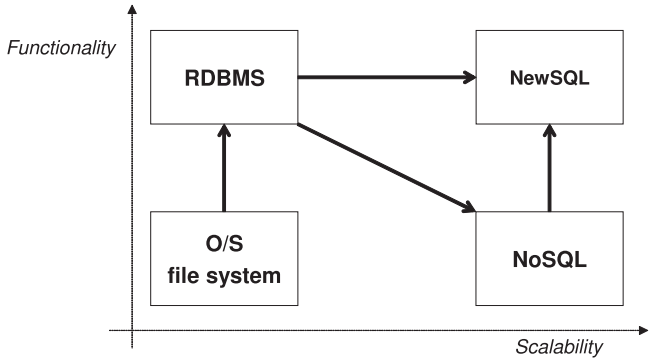


Fig. 1. Evolution of data management systems.

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 [8], an SNS system using a shared-nothing parallel DBMS, which is an extension of the Odysseus DBMS [8].

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.

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 Fig. 2. Vertices and edges may contain data in the key–value format. According to Bronson et al. [7], the key–value storage has the following format:

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

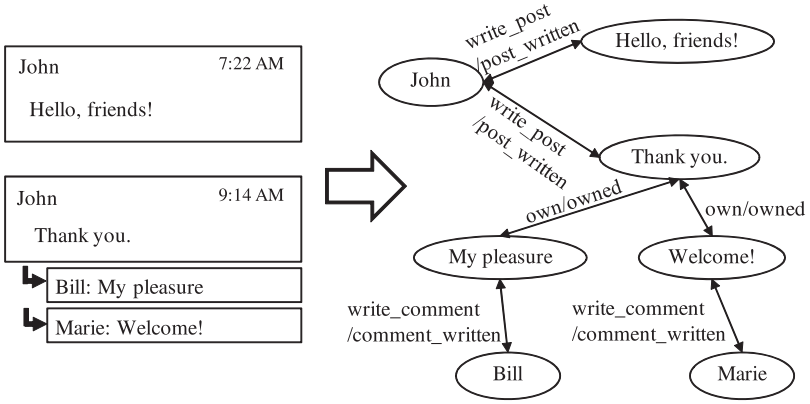


Fig. 2. An example of the graph data model.

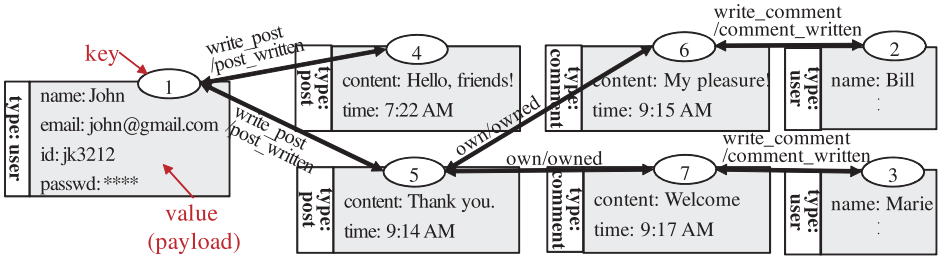


Fig. 3. An example of graph data in the key–value format.

- 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 Fig. 3 for an illustration)

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). Fig. 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

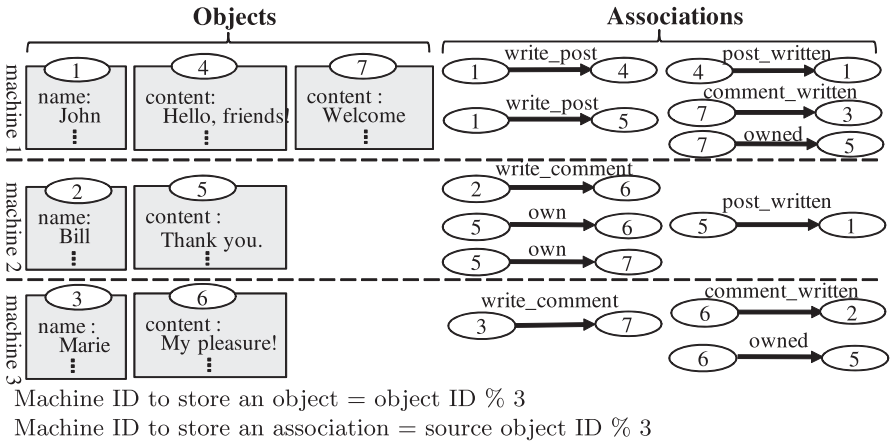
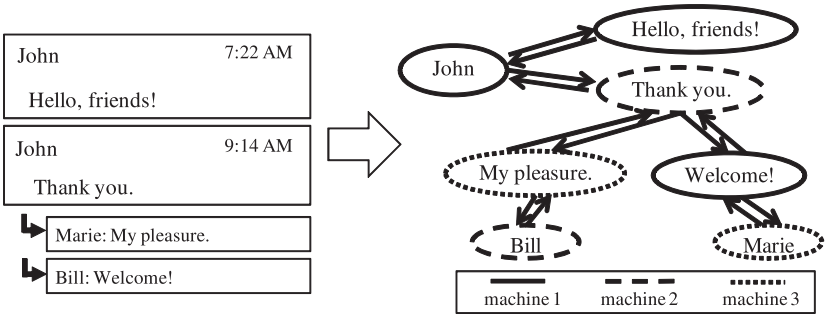


Fig. 4. An example of graph data in the key–value format distributed over three nodes.



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

Fig. 5. An example of the timeline operation on graph data in the key–value format.

(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 [1,9]. As an example, Fig. 5 illustrates the overhead of having to access all three machines in order to view a user's timeline.

2.2. Relational data model

Fig. 6 shows the relational model view of the data shown in the graph model view of data in Fig. 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 Fig. 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 Fig. 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.

3. Database design with the relational model

3.1. SNS entity-relationship schema

Fig. 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 Figs. 8 to 23 through the following methods. We model the post_id attributes of the

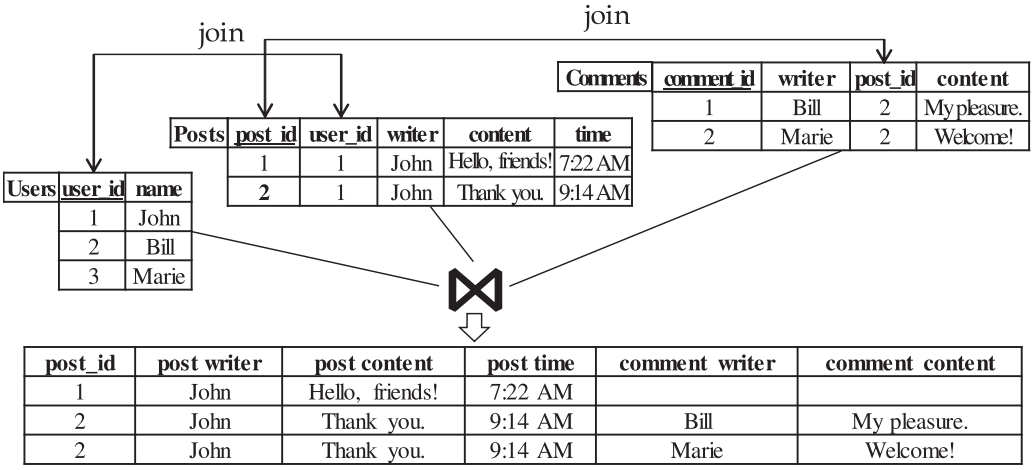


Fig. 6. An example of the relational model view of data.

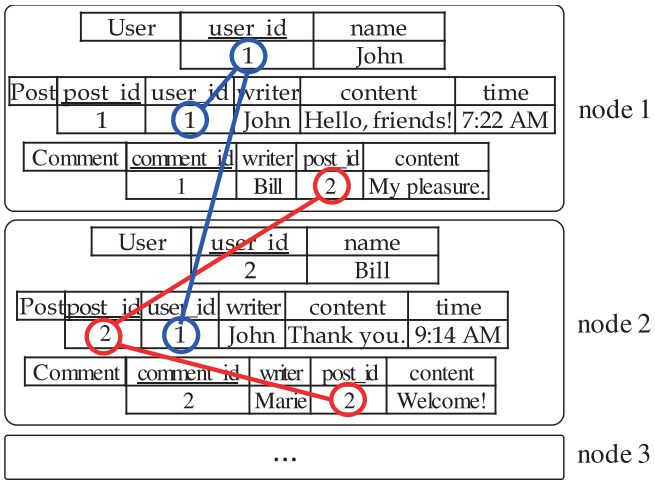


Fig. 7. An example of relational data distribution and storage.

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.

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. Fig. 9 shows

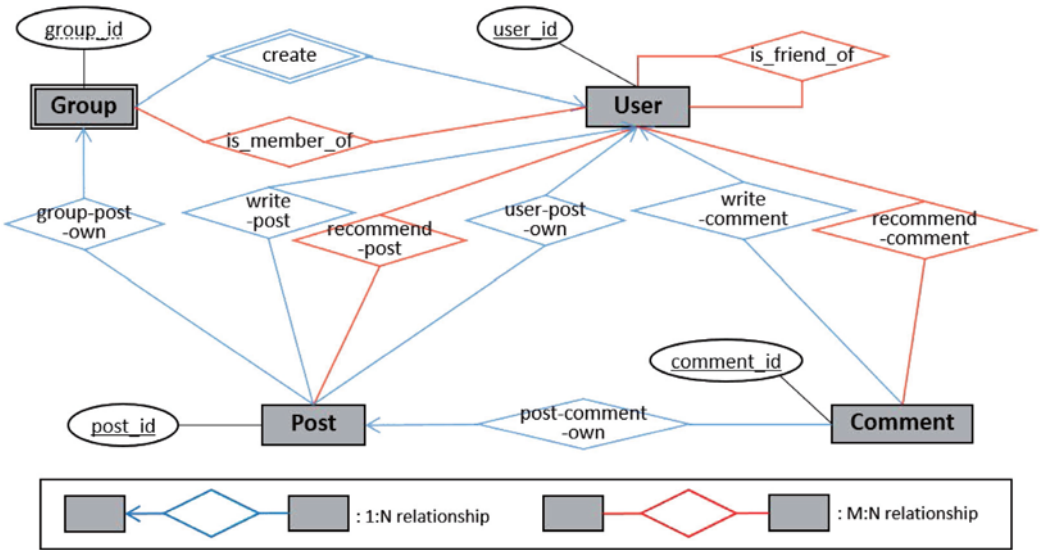


Fig. 8. SNS entity-relationship schema.

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

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.

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

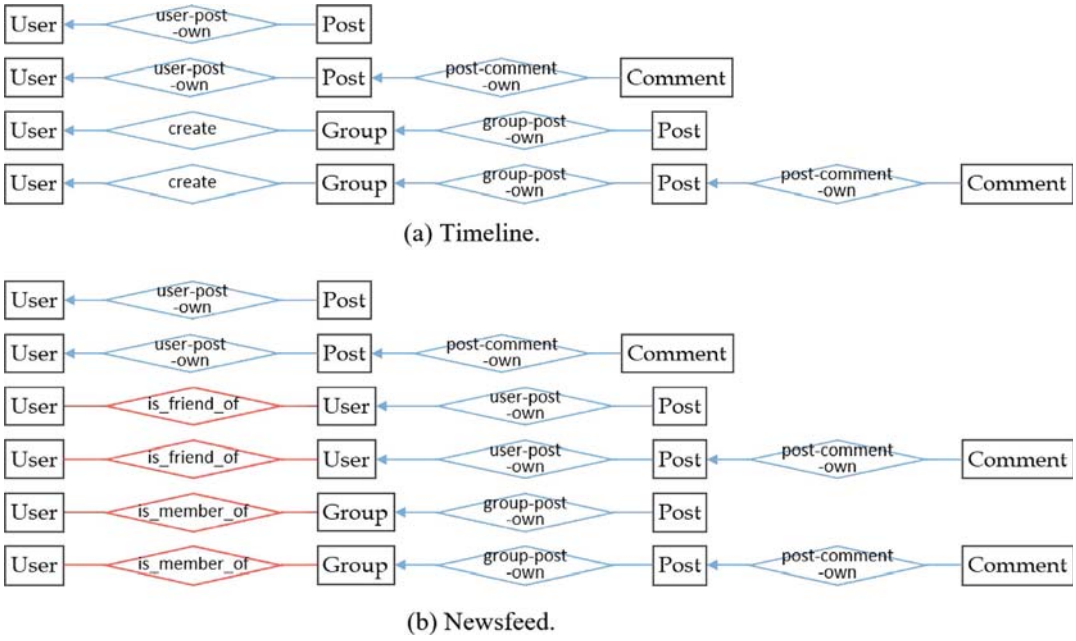


Fig. 9. SNS timeline and newsfeed operations.

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]

Fig. 10 shows a sub-schema including only one-to-many relationship types from the SNS entity-relationship schema in Fig. 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 Fig. 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 Fig. 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 Fig. 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 Fig. 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 Fig. 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 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".

Fig. 13 shows the sequences of identifying relationships traversed to execute timeline and newsfeed operations. For the timeline operation (see Fig. 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 Fig. 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.

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).

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

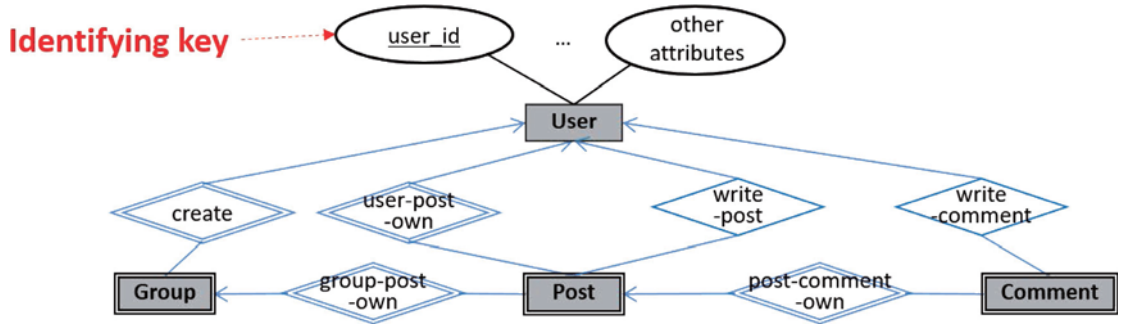


Fig. 10. An example of modelling one-to-many relationship types as identifying relationship types.

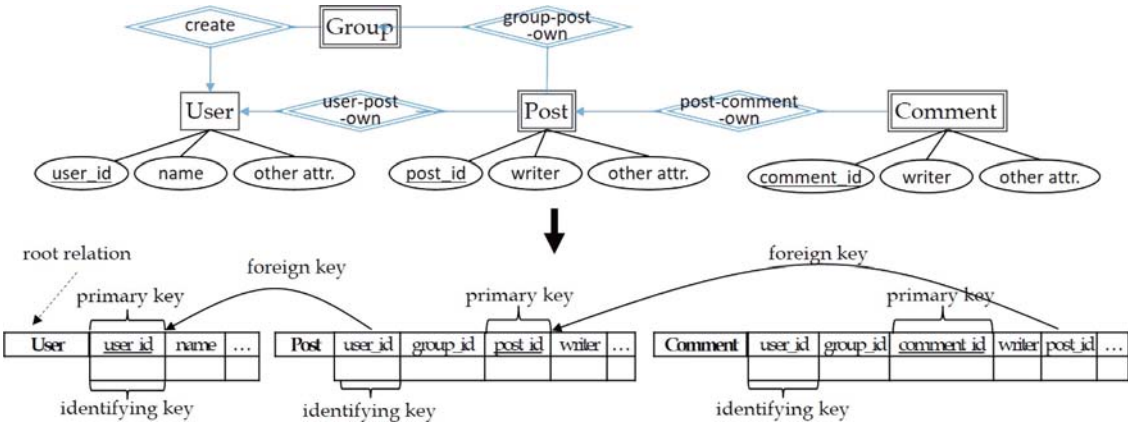


Fig. 11. An example of mapping identifying relationship types to relational schema.

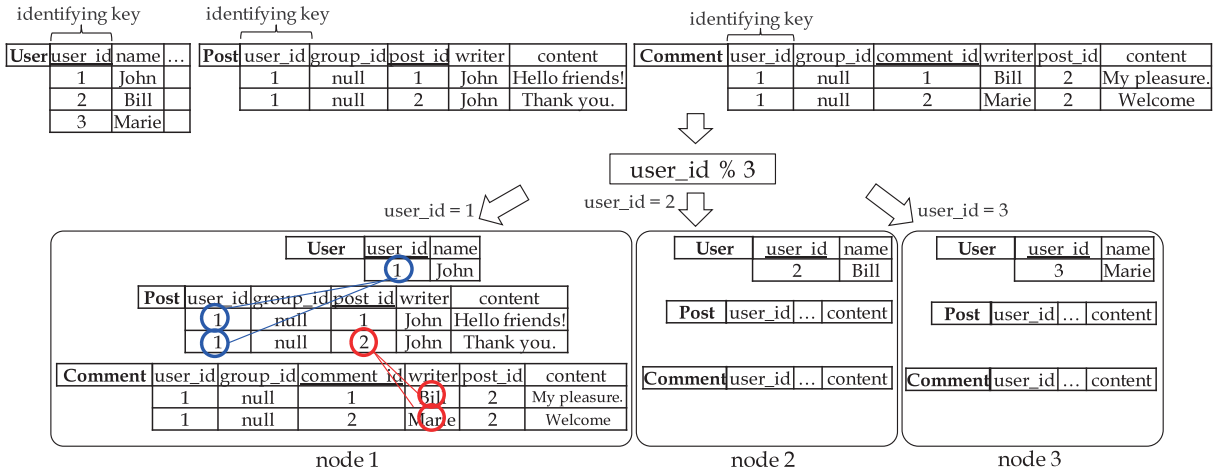


Fig. 12. An example of partitioning relations by identifying key.

3.3. Many-to-many relationship

Fig. 14 shows many-to-many relationships in the simplified SNS entity-relationship schema (Fig. 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

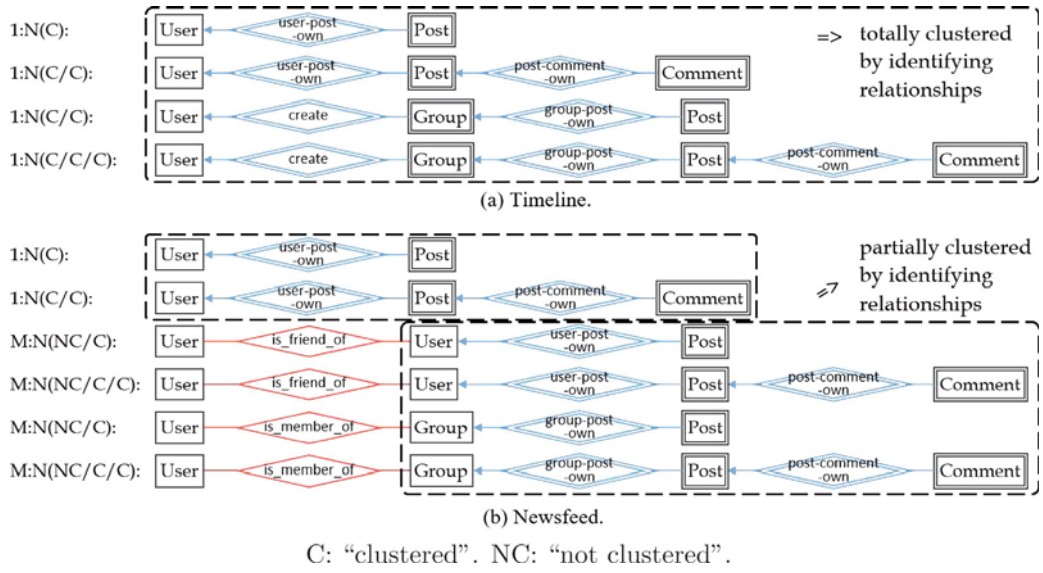


Fig. 13. SNS timeline and newsfeed operations (clustered by identifying relationships).

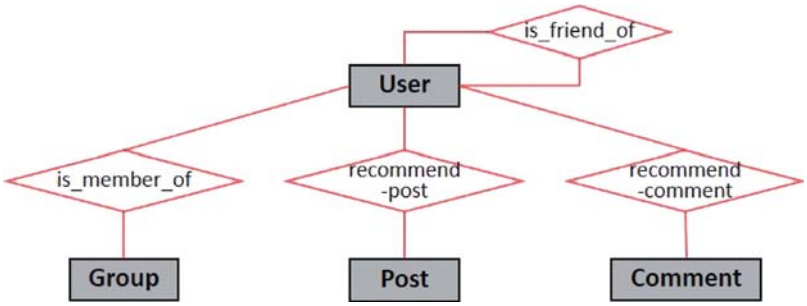
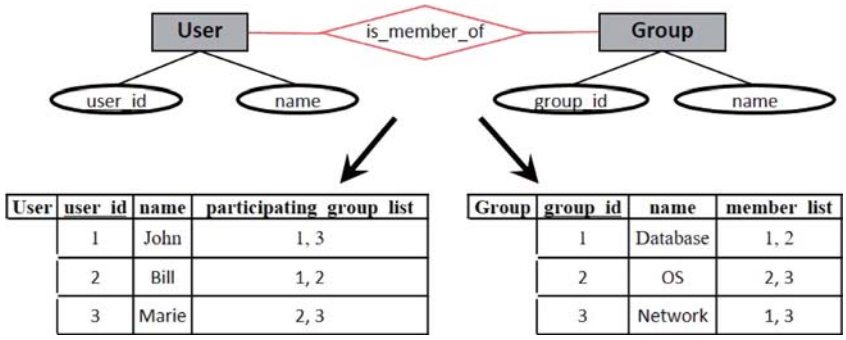


Fig. 14. Many-to-many relationships in the SNS entity-relationship schema of Fig. 8.

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.

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. Fig. 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.

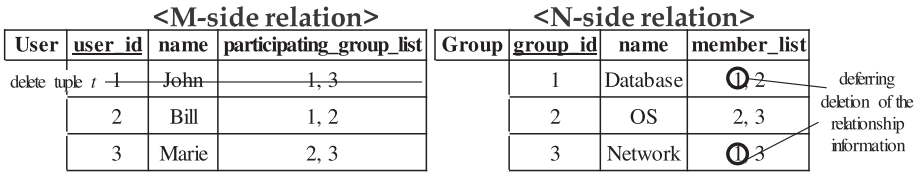
For delete transactions, we use a deferred delete strategy to complete the transactions without two-phase commit. The delete operations (deactivate, delete a group, and delete a post in the experiment in Section 4) need to delete both of a tuple and the references to it. The deferred delete strategy 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, Fig. 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 are 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



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

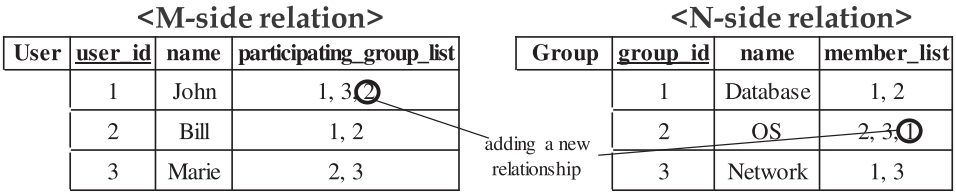
Fig. 15. Mapping a many-to-many relationship using a set type.



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

Fig. 16. An example of deferred delete of a relationship.



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

Group	group_id	name	member_list
	1	Database	1, 2
	2	OS	~~2, 3~~ ①
	3	Network	1, 3

Fig. 17. An example of inserting a new relationship.

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.

For insert transactions, deferred commit is not an option. As shown in Fig. 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 Fig. 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.)

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 2.93 GHz quadcore CPU, 4 GB memory, and a 500 GB hard disk) and eight slave nodes (each with 3.2 GHz quadcore CPU, 8 GB memory, and a 2 TB hard disk). Each

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

Parameter	Description	Value
$N_{servers}$	Total number of servers	60 thousand [10]
$N_{log\_servers}$[a]	Total number of log servers	2 thousand [11]
N_{nodes}	Total number of nodes (i.e., data servers)	
N_{users}	Total number of users	350 million [12]
N_{groups}	Total number of groups	620 million [13]
N_{posts}	Total number of posts stored	
$N_{comments}$	Total number comments stored	
N_{photos}	Total number of photos stored	260 million [14]
NPD_{write_post}	Number of writing post per day	
$NPD_{write\_comment}$	Number of writing comment per day	300 million [12]
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 [12]
$NPD_{recommend\_comment}$	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 [14]
NPD_{view_photo}	Number of viewing photo per day	90 billion [14]
$NPD_{all\_queries}$	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% [14]
$R_{view\_full\_photo}$	Rate of full photo viewing among all photo viewings	5.2% [14]
$R_{photo\_in\_post}$	Rate of photo being contained in a post	75% [15]
FRIENDS_PER_USER	Average number of friends per user	44[d]
GROUPS_PER_USER	Average number of groups per user	12 [12]
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 [16].
[e] A default number of posts per page on SNS-A.

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 [6,10–15] 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) [6]. As a result, the size of the database is 0.23 TBytes per node.[2]

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.

[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 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 [6]). The number of system nodes = 8.

Table 5
Instantiation of SNS objects in the data set.

SNS object type	Attribute	Instantiation
User	Name	Randomly selected from list on the Web [17]
	Job	Randomly selected from list on the Web [18]
	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" [19], where the random length follows the distribution in Nierhoff [20] (see Fig. 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 [21] and set to the size of 8KB for thumbnail image and 64KB for full image, the same as in Beaver et al. [14]

- $N_{posts} = N_{photos} / R_{photo-in-post}$ = 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 objects (except photos) per node is 1.2 billion according to the LinkBench benchmark [6].

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 [22–25], 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:

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

- Is_friend_of:

$$\overline{M} = \overline{N} = FRIENDS\_PER\_USER = 44$$

- Recommend_post:

$$\overline{M} = \frac{NPD_{recommend\_post}}{NPD_{write\_post}} = \frac{NPD_{recommend\_post}}{NPD_{photos\_updated} / R_{photo\_in\_post}} = \frac{105 \text{ million}}{142.8 \text{ million}/0.75} = 0.55$$

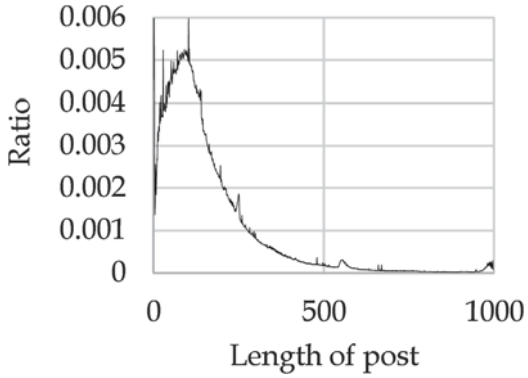


Fig. 18. Distribution of the lengths of posts [20].

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

$$\overline{N} = N_{posts} \times \frac{NPD_{recommend\_post}}{NPD_{write\_post}} \times \frac{1}{N_{users}} = 347 \text{ billion} \times \frac{105 \text{ million}}{190 \text{ million}} \times \frac{1}{350 \text{ million}} = 547$$

- Recommend_comment:

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

$$\overline{N} = N_{comments} \times (NPD_{recommend\_post} \times \frac{R_{recommend\_comment}}{R_{recommend\_post}}) \times \frac{1}{NPD_{write\_comment}} \times \frac{1}{N_{users}}$$

$$= 547 \text{ billion} \times (105 \text{ million} \times \frac{0.251}{0.749}) \times \frac{1}{300 \text{ million}} \times \frac{1}{350 \text{ million}} = 183$$

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).

- $NPD_{write\_post} = \frac{NPD_{upload\_photo}}{R_{photo\_in\_post}} = \frac{142.8 \text{ million}}{0.75} = 190 \text{ million}$
- $NPD_{write\_comment} = 300 \text{ million}$
- $NPD_{modify\_post} = NPD_{write\_post} \times R_{modify\_post} = \frac{NPD_{update\_photo}}{R_{photo\_in\_post}} \times R_{modify\_post} = \frac{142.8 \text{ million}}{0.75} \times 0.132 = 25.13 \text{ million}$
- $NPD_{modify\_comment} = NPD_{write\_comment} \times R_{modify\_comment} = 300 \text{ million} \times 0.086 = 25.8 \text{ million}$
- $NPD_{delete\_post} = NPD_{write\_post} \times R_{delete\_post} = \frac{NPD_{upload\_photo}}{R_{photo\_in\_post}} \times R_{delete\_post} = \frac{142.8 \text{ million}}{0.75} \times 0.097 = 18.46 \text{ million}$
- $NPD_{delete\_comment} = NPD_{write\_comment} \times R_{delete\_comment} = 300 \text{ million} \times 0.091 = 27.3 \text{ million}$
- $NPD_{recommend\_post} = 105 \text{ million}$

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%

- $NPD_{recommend\_comment} = NPD_{recommend\_post} \times \frac{R_{recommend\_comment}}{R_{recommend\_post}} = 105 \text{ million} \times \frac{0.251}{0.749} = 35.1 \text{ million}$
- $NPD_{view\_timeline} = NPD_{view\_photo} \times \frac{R_{view\_thumbnail}}{R_{photo\_in\_post}} \times R_{view\_timeline} \times \frac{1}{POSTS\_PER\_PAGE} = 90 \text{ billion} \times \frac{0.844}{0.85} \times 0.287 \times \frac{1}{10} = 2.9 \text{ billion}$
- $NPD_{view\_newsfeed} = NPD_{view\_photo} \times \frac{R_{view\_thumbnail}}{R_{photo\_in\_post}} \times R_{view\_newsfeed} \times \frac{1}{POSTS\_PER\_PAGE} = 90 \text{ billion} \times \frac{0.844}{0.75} \times 0.713 \times \frac{1}{10} = 7.22 \text{ billion}$
- $NPD_{view\_full\_photo} = NPD_{view\_photo} \times R_{view\_full\_photo}$ 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 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 [26].)

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 server and SNS-A is subtracted from the elapsed time.

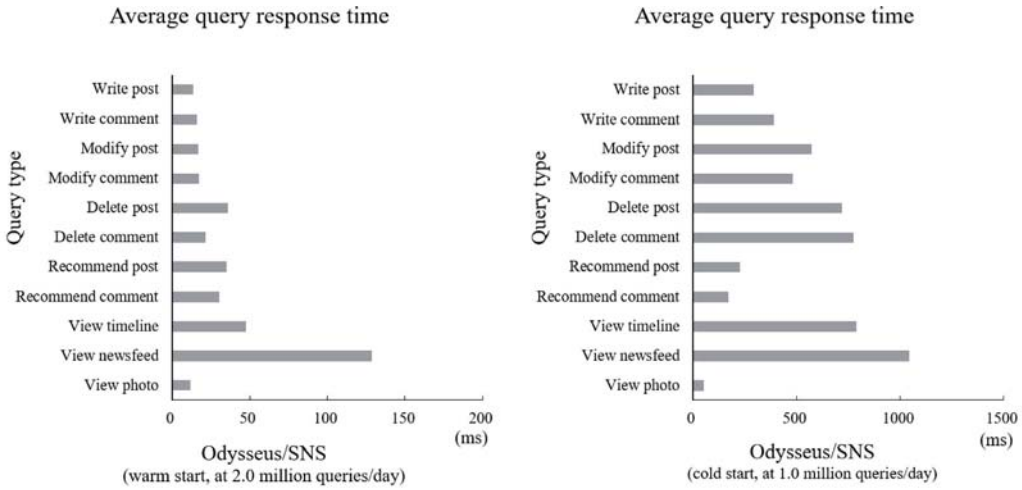
4.2. Query response times in Odysseus/SNS and SNS-A

Fig. 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 Fig. 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.

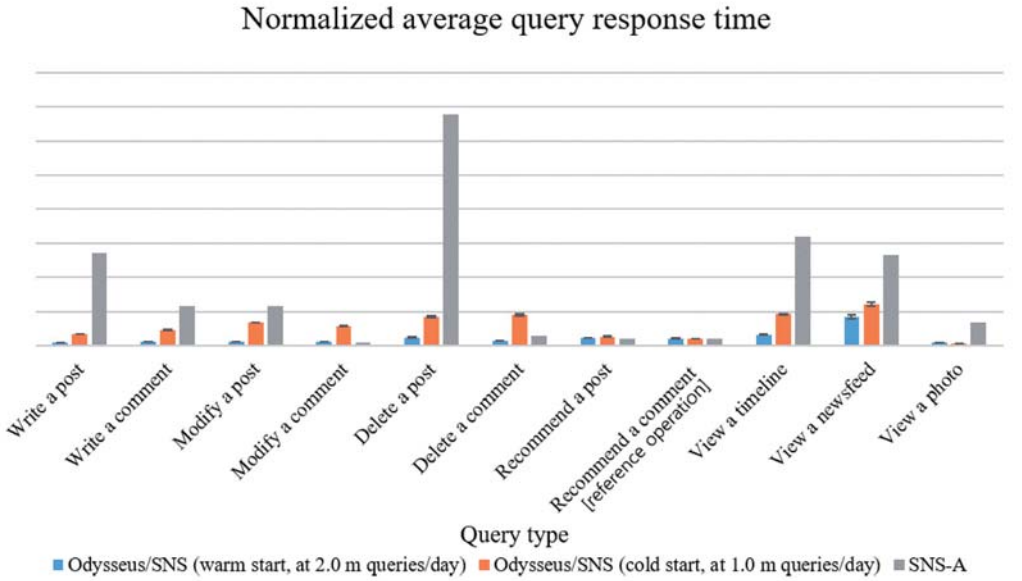
In Fig. 19, the normalized processing times of newsfeed and timeline, which together take about 66% of query load, are significantly reduced (by 53.8 ~ 68.0% for newsfeed; 71.0 ~ 90.1% for timeline) in Odysseus/SNS compared with those of SNS-A. This reduction is a direct benefit of clustering relations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

In addition, the processing time of "delete a post" in Fig. 19 and "delete a group" and "deactivate" in Fig. 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.

[3] We measured the query response time of each operation six times on Jan. 28th, 2017, with a 4-h 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.

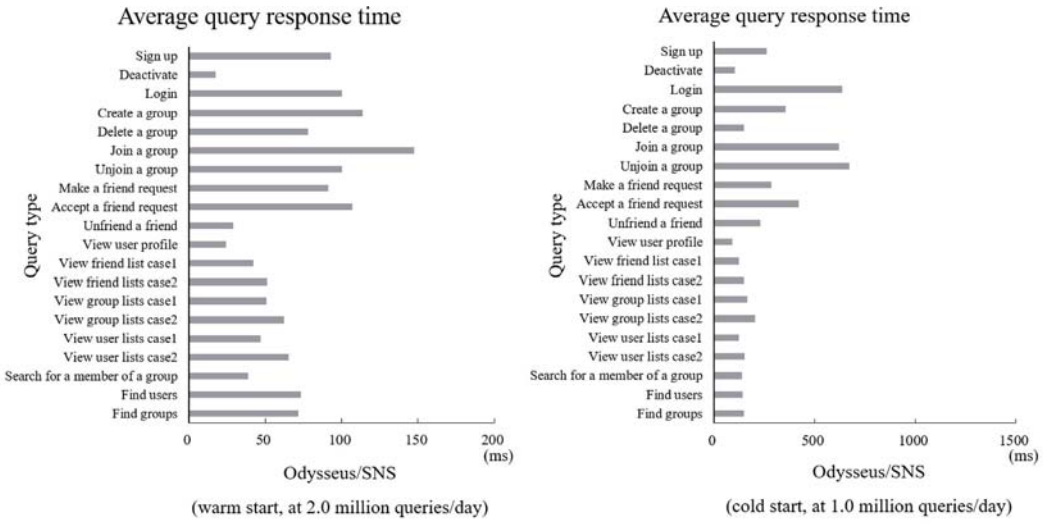


(b) Clustered bar graph comparing Odysseus/SNS at warm start and cold start and SNS-A without vertical scale.

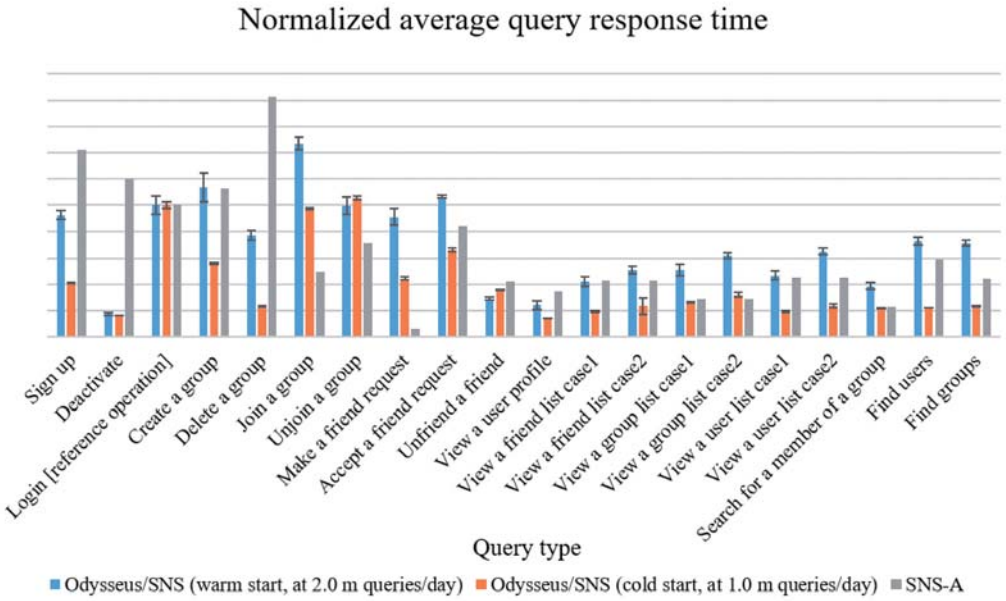
Fig. 19. Average query response times of the primary query operations.

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

Fig. 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.



(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.

Fig. 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.

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

Fig. 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

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

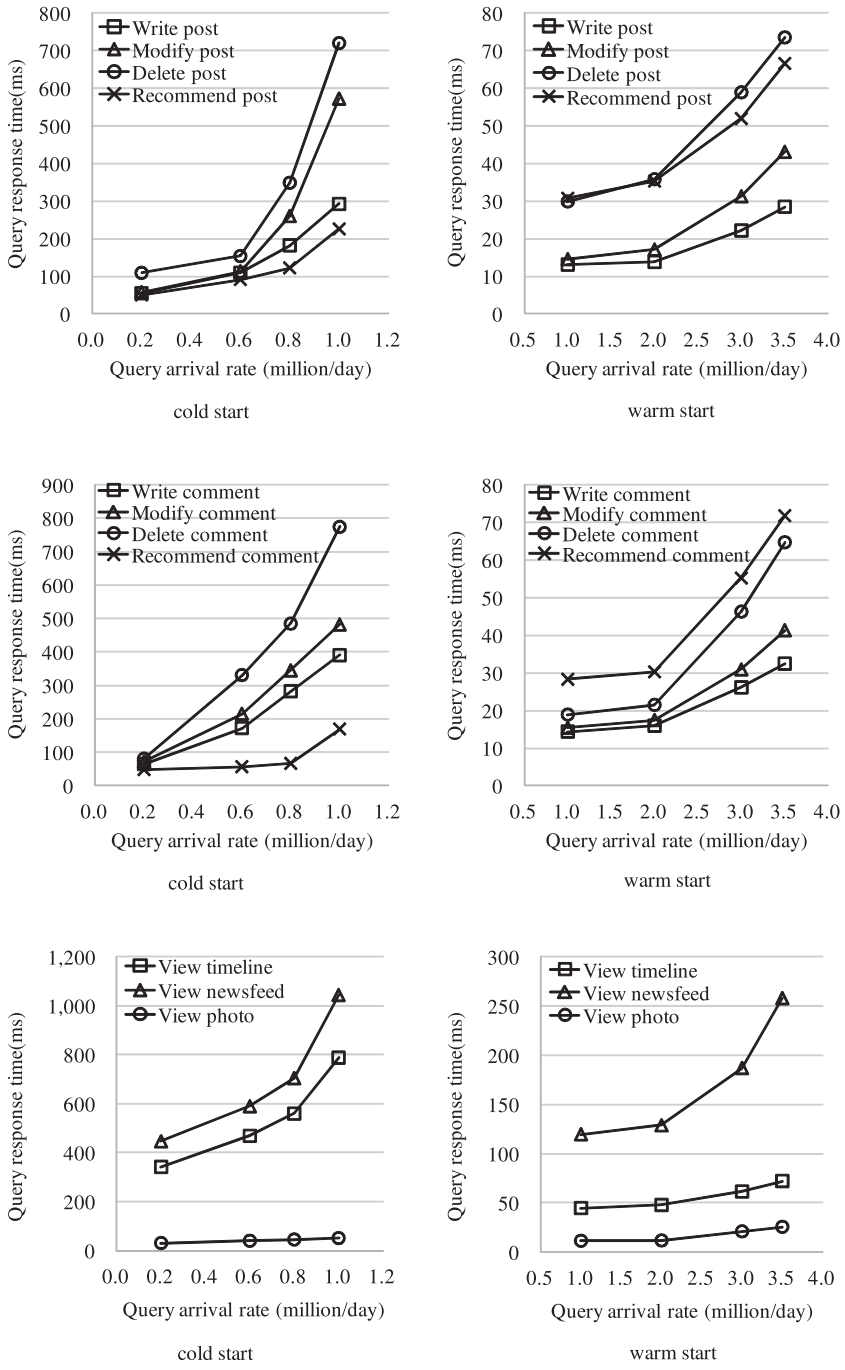


Fig. 21. Average response times of query operations in Odysseus/SNS as the query arrival rate is varied.

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.

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

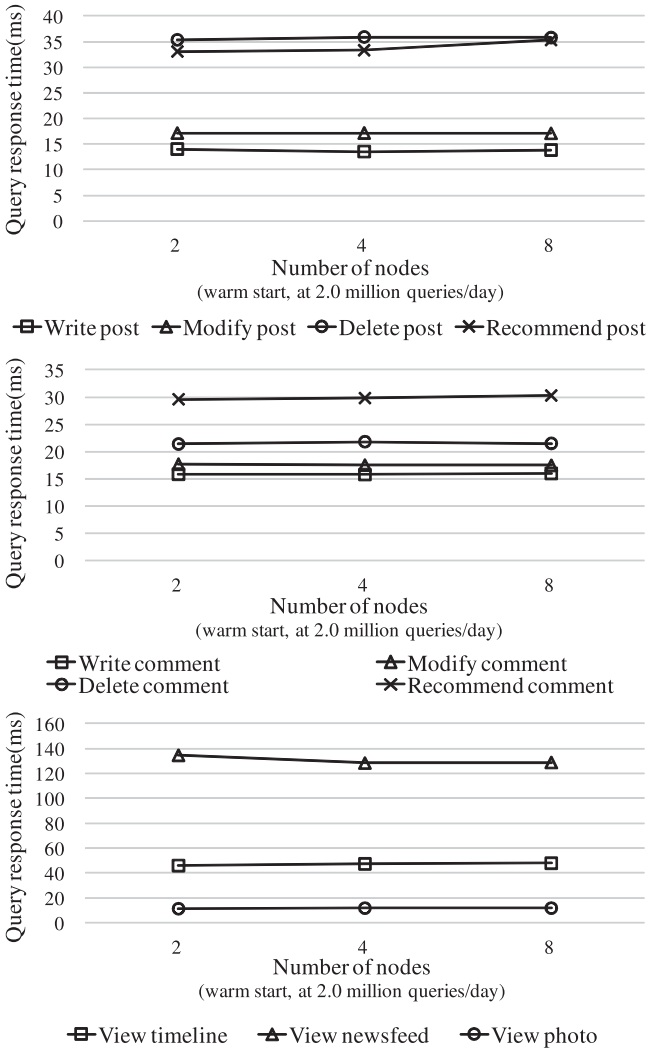


Fig. 22. Average response times of query operations in Odysseus/SNS when the number of nodes is varied.

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 [6], demonstrated the significant benefit of the methods, especially in timeline and newsfeed – the two dominant SNS query operations – and in various delete operations.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

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 [27]. 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 [27].

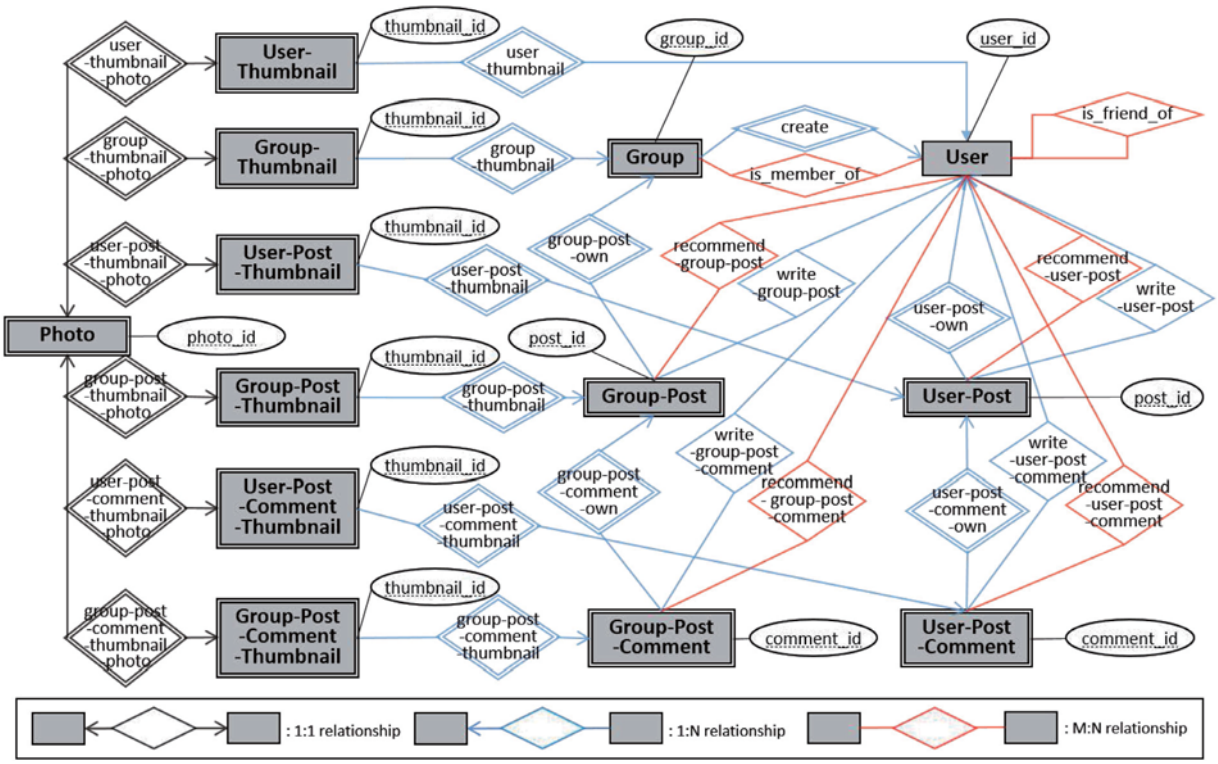


Fig. 23. Entire SNS entity-relationship schema.

Appendix

See Fig. 23.

References

[1] R. Cattell, Scalable SQL and NoSQL data stores, ACM SIGMOD Rec. 39 (4) (2010) 12–27.
[2] M. Aslett, How will the database incumbents respond to NoSQL and NewSQL? 2011, 451 TechDealmaker, The 451 group.
[3] K. Grolinger, W.A. Higashino, A. Tiwari, M.A. Capretz, Data management in cloud environments: NoSQL and NewSQL data stores, J. Cloud Comput. 2 (1) (2013) 1–24.
[4] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, M. Stonebraker, A comparison of approaches to large-scale data analysis, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 2009, pp. 165–178.
[5] M. Stonebraker, D. Abadi, D.J. DeWitt, S. Madden, E. Paulson, A. Pavlo, A. Rasin, MapReduce and parallel DBMSs: friends or foes? Commun. ACM 53 (1) (2010) 64–71.
[6] T.G. Armstrong, V. Ponnekanti, D. Borthakur, 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, New York, NY, USA, 2013, pp. 1185–1196.
[7] 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, 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, Berkeley, CA, USA, 2013, pp. 49–60.
[8] K. Whang, J. Lee, M. Lee, W. Han, M. Kim, J. Kim, DB-IR Integration using tight-coupling in the Odysseus DBMS, World Wide Web J. 18 (3) (2015) 491–520.
[9] R. Escriva, B. Wong, E. Sirer, HyperDex: A distributed, searchable key-value store, ACM SIGCOMM Comput. Commun. Rev. 42 (4) (2012) 25–36.
[10] Datacenter Knowledge, The Facebook data center FAQ, 2014, http://www.datacenterknowledge.com/the-facebook-data-center-faq, referenced in November 2014.
[11] J. Vijayan, Facebook moves 30-petabyte Hadoop cluster to new data center, 2014, Computer World. http://www.computerworld.com/s/article/9218752/Facebook_moves_30_petabyte_Hadoop_-cluster_to_new_data_center, referenced in November 2014.
[12] P. Kirschner, A. Karpinski, Facebook and academic performance, Comput. Hum. Behav. 26 (6) (2010) 1237–1245.
[13] N. O'Neill, Google now indexes 620 million Facebook groups, 2014, SocialTimes. http://allfacebook.com/google-now-indexes-620-million-facebook-groups_b10520, referenced in November 2014.
[14] D. Beaver, S. Kumar, H.C. Li, J. Sobel, 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, Vancouver, Canada, 2010, pp. 1–8.
[15] Digital Marketing Ramblings, By the numbers: 85 amazing Facebook page statistics, 2014, http://expandedramblings.com/index.php/facebook-page-statistics, referenced in November 2014.
[16] J. Leskovec, A. Krevl, SNAP datasets: Stanford large network dataset collection, 2014, http://snap.stanford.edu/data, referenced in November 2014.
[17] Wikipedia, Anthroponymy, 2014, http://en.wikipedia.org/wiki/Anthroponymy, referenced in November 2014.

[18] Wikipedia, List of occupations, 2014, http://en.wikipedia.org/wiki/Lists_of_occupations, referenced in November 2014.
[19] J. Austen, Pride and prejudice (ebook), 2014, https://www.gutenberg.org/ebooks/1342, referenced in November 2014.
[20] M.H. Nierhoff, Research: Short posts on Facebook, Twitter and Google+ seem to get more interactions, 2014, Quintly-Social Media Analytics Blog. https://www.quintly.com/blog/2013/12/short-posts-on-facebook-twitter-google-more-interactions, referenced in November 2014.
[21] D. Niblack, Imagebase (absolutely free photos), 2014, http://imagebase.net, referenced in November 2014.
[22] S. Catanese, P. De Meo, E. Ferrara, G. Fiumara, A. Provetti, Extraction and analysis of Facebook friendship relations, in: Computational Social Networks, Springer, 2012, pp. 291–324.
[23] H. Halpin, 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, 2014, pp. 885–890.
[24] L. Muchnik, S. Pei, L.C. Parra, S.D. Reis, J.S. Andrade Jr., S. Havlin, H.A. Makse, Origins of power-law degree distribution in the heterogeneity of human activity in social networks, Sci. Rep. 3 (1783) (2013) 1–7.
[25] R.A. van Compernolle, 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, John Benjamins, 2009, pp. 191–212.
[26] S. Warmuta, D. Delfrate, Memcached, 2009, Slideshare, http://www.slideshare.net/harpastum/memcached-2699652. (Referenced in November 2014).
[27] K. Whang, T. Yun, J. Park, K. Cho, S. Kim, I. Yi, I. Na, B. Lee, Building Social Networking Services Systems Using the Relational Shared-Nothing Parallel DBMS, CS-TR-2018-419, School of Computing, KAIST, 2018, Also, in Journal of The National Academy of Sciences Republic of Korea, Natural Sciences, Vol. 57, No. 2, Dec. 2018 (in Korean).

Kyu-Young Whang earned his Ph.D. from Stanford University in 1984. From 1983 to 1991, he was a Research Staff Member at IBM T. J. Watson Research Center. In 1990, he joined KAIST, where he currently is a Distinguished Professor and a Professor Emeritus at School of Computing. His research interests encompass database systems/storage systems, search engines, object-oriented databases, geographic information systems, big data management, and data mining. He was the general chair of VLDB2006. He served as an Editor-in-Chief of the VLDB Journal from 2003 to 2009. Dr. Whang is a Fellow of ACM and a Life Fellow of IEEE. He served as the chair of IEEE Technical Committee on Data Engineering from 2013 to 2014. Dr. Whang is a member of National Academy of Sciences Republic of Korea.

Inju Na is a Ph.D. candidate in school of computing from Korea Advanced Institute of Science and Technology. (KAIST). He received the B.S. in school of computer science and engineering from Kyungpook National University in 2012. His research interests include graph data, database systems and storage systems.

Tae-Seob Yun received the B.S. in computer engineering from Kyungpook National University in 2008, the M.S. in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2010, and the Ph.D. in computer science from KAIST in 2016. His research interests include information retrieval, large-scale parallel search engines, distributed systems, and social networking systems.

Jin-Ah Park received the B.S. in computer engineering from Hanyang University in 2011 and the M.S. in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2013. Her research interests include big data analytics.

Kyu-Hyun Cho received the B.S. in computer science from Sogang University in 2013 and the M.S. in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2015. His research interests include information retrieval and big data analytics.

Se-Jin Kim is a Ph.D. candidate in the Graduate School of Knowledge Service Engineering, Korea Advanced Institute of Science and Technology, Republic of Korea, and earned his master's degree in Computer Science at Korea Advanced Institute of Science and Technology. His research interests include parallel processing, trajectory data mining, and taxi demand prediction.

IlYeop Yi graduated from Yonsei University in 2006, received an M.S. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2011, and earned a Ph.D. degree in Computer Science from KAIST in 2016. From 2016 to 2018, he was a Postdoctoral Researcher at Industrial Engineering & Management Research Institute in KAIST. Currently, he is working at Samsung Electronics. His research interests include database systems, storage systems, distributed database systems, cloud database systems, big data management platforms, search engines, data mining systems, graph database systems, and complex event processing / data stream processing.

Byung Suk Lee is a Professor of Computer Science at the University of Vermont. He received his Ph.D. from Stanford University, MS from KAIST, and BS from Seoul National University. His current research interests include database systems, data stream processing, and data mining. His teaching has been mainly on subjects such as algorithms, database systems, and data structures.