sonSchema: A Conceptual Schema for Social Networks *

Zhifeng Bao Y.C. Tay Jingbo Zhou

National University of Singapore

Abstract. sonSQL is a MySQL variant that aims to be the default database system for social network data. It uses a conceptual schema called sonSchema to translate a social network design into logical tables. This paper introduces sonSchema, shows how it can be instantiated, and illustrates social network analysis for sonSchema datasets. Experiments show such SQL-based analysis brings insight into community evolution, cluster discovery and action propagation.

Keywords: schema, social network

1 Introduction

The proliferation of online social networks is now as unstoppable as the spread of the Web. Such a network needs a database system to properly manage its data. Many social network services are started by small teams of developers and typically use some free, off-the-shelf database system, e.g. MySQL. If the service is successful and the team grows to include professional database administrators, the dataset may be so large already that any re-engineering of the early decisions becomes difficult.

Our contribution to solving this problem is **sonSQL** (http://sonsql.comp.nus.edu.sg), a variant of MySQL that we hope to develop into the default database management system for social networks. We provide sonSQL with an interactive user interface and a rule-based expert system to translate a developer's design for a social network into a set of db-relational[1] tables. These tables are logical instantiations of a conceptual schema, called **sonSchema** ("son" for "social network"), that we designed for social network data. The objective of this paper is to introduce sonSchema.

We follow two guidelines in our design of sonSchema: **(G1: Generality)** The schema should be sufficiently general that it can model any social network design from the developer, and **(G2: Service-Oriented)** The entities and relationships in the schema must correspond naturally to social network activity and services.

Given our goal of building a database system for social networks, why chose a db-relational system (MySQL) as the code base? There is a trend in the convergence of OLTP and OLAP[2], so social network analysis (SNA) may increasingly run on live data.

* This research was supported in part by MOE Grant No. R-252-000-394-112 and carried out at the SeSaMe Centre, which is supported by the Singapore NRF under its IRC@SG Funding Initiative and administered by the IDMPO.

[1] Following Angles and Gutiérrez [2], we use "db-relation" to refer to a database table, while "sn-relation" refers to a relationship in a social network.

[2] http://www.greenplum.com/products/greenplum-uap

Can a sonSchema dataset support SNA effectively? Most SNA in the literature are based on graph models, so should we start with a graph database system instead?

We studied one well-known problem in SNA, namely link prediction [15]. We developed **sonLP**, a predictor that applies principal component regression to features from multiple dimensions in the data [5]. For the example of a social graph defined by coauthorship among ACM authors, the links are in one dimension, while affiliation, research areas, etc., are in other dimensions. Experiments on such data show that sonLP outperforms HPLP+, a state-of-the-art predictor that is based entirely on the social graph [16]. This suggests that SNA should work with features *beyond* the graph topology.

By itself, sonLP cannot make the case for choosing db-relations instead of graphs as the data model. A fuller discussion of this choice is in our technical report [4]. Perhaps the most compelling argument, for us, is this: A database system for social network data must have an expressive query language, query optimization, indices, integrity constraints, concurrency control, crash recovery, batch processing and data exploration tools. Implementing this list to bring a prototype to market is highly nontrivial for any database system, and the only ones to have done so are db-relational. This may be why the core databases for Facebook (http://www.facebook.com) and Twitter (http://www.twitter.com) remain db-relational, even though some of their data use NoSQL (e.g. Cassandra for Twitter) [18].

The NoSQL movement (nosql-database.org) has argued that db-relational systems do not suit massive datasets because ACID consistency would compromise partition tolerance. However, there are other failures besides network partitioning and, again, the task of implementing some non-ACID consistency to handle various failures can be overwhelming [21]; this is particularly true for start-ups that need off-the-shelf systems.

We therefore chose to go with a db-relational system. It remains for us to show (later in this paper) that SQL-based SNA can even provide better insight than analyzing some graph extracted from the data.

This paper makes three contributions: (1) We introduce sonSchema, a conceptual schema for social network data. (2) We show how social network analysis can be done with SQL queries on a sonSchema dataset. (3) We present insights on community evolution, cluster discovery and action propagation. Such insights are hard to extract from just the social graph because they require multi-dimensional access to the raw data, using the full range of db-relational query operators.

We begin by introducing sonSchema in Sec. 2, and present several examples of instantiations of sonSchema in Sec. 3. Sec. 4 then demonstrates how three well-known problems in social network analysis can be studied for a sonSchema dataset. Results from experiments, reported in Sec. 5, show that these techniques are effective and provide new insights for these problems. Sec. 6 reviews related work, before Sec. 7 concludes with a brief description of current work.

2 From guidelines to conceptual and logical schemas

Sec. 2.1 first characterizes a social network. The guidelines (G1) and (G2) then lead us to the conceptual schema. The db-relational form of this schema is sonSchema, which we present in Sec. 2.2, together with examples of translation into logical schemas.

2.1 Social network entities and relationships

For generality (G1), we start with the following fundamental characterization [25]: *An online social network is a group of users who interact through social products.* This informal definition focuses on social interaction, and explicitly points out the role of products (games, events, songs, polls, etc.). It suggests four entities for our model:

(E1) **user** is generic; it can be Jim, an advertiser, a retailer, etc.
(E2) **group** has details (names, membership size, etc.) of an interest group.
(E3) **post** may be a blog, tag, video, etc. contributed by a user; it includes the original post, comments on that post, comments on those comments, etc.
(E4) **social_product** is a product with some intended consumers, like an event created for a group, a retailer's coupon for specific customers, etc. A **post** can be considered as a special case of a **social_product** (that has no intended consumer).

Similarly, there are four natural relationships:

(R1) **friendship** among users may refer to Twitter followers, ACM coauthors, etc.
(R2) **membership** connects a user to a group.
(R3) **social_product_activity** connects a user to a social product through an activity (buy a coupon, vote in a poll, etc.).
(R4) **social_product_relationship** connects two social products, like between a meeting and a poll, or between a charity event and a sponsor's discount coupons.

Interaction creates another entity and relationship:

(E5) **private_msg** is a message that is visible only to the sender and receiver(s).
(R5) **response2post** is a relationship between a tag and an image, a comment and a blog, a comment and another comment, etc. If **post** is considered a **social_product**, then one can consider **response2post** as a special case of **social_product_relationship**.

The above exhausts the list of entities (users and products) and relationships (user-user, user-product, product-product) in any online social network, in line with the generality requirement (G1). The refinements of **social_product** into **post** and **private_msg** reflect services that are usually provided by social networks (G2). Service orientation (G2) also guided our model for the interactions that give life to a social network; e.g. we split a comment into a **post** ((E3) modeling the content) and a **response2post** ((R5) modeling the interaction).

2.2 From conceptual to logical schemas

sonSchema is the db-relational form, shown in Fig. 1, of the conceptual schema in Sec. 2.1. In the following, we use **bold** for a schema, *italics* for an instance of the schema, and `typewriter` font for attributes.

The table **friendship** stores user pairs, since most db-relational systems do not provide an attribute type suitable for lists of friends. Fig. 1 shows an optional attribute `group_id` in **post** for the case where a post (e.g. a paper) belongs to a group (e.g. journal). There is another optional attribute, `product_group`, for the category that this

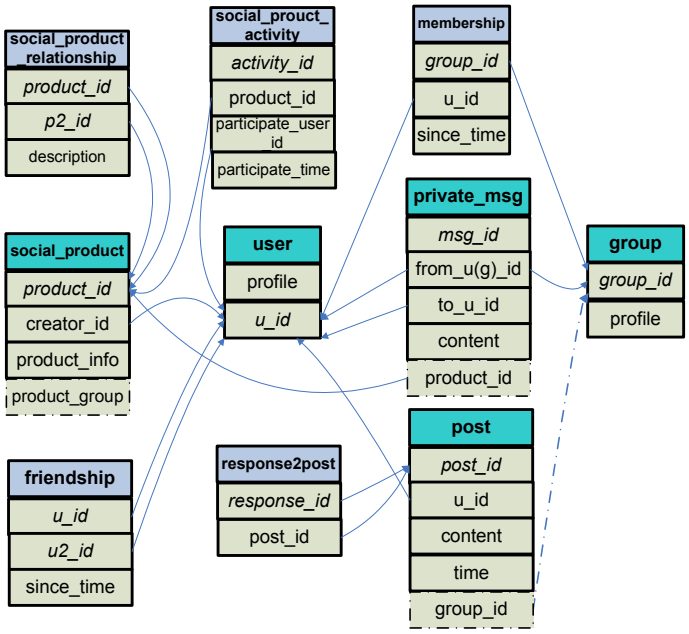

Fig. 1: **sonSchema. Table names are in bold, primary keys are in *italics*, and edges point from foreign key to primary key. The 5 green tables are for entities, the 5 blue tables for relationships. Some attributes (e.g. `group_id` in post) are optional.**

product belongs to. E.g. a social network for sharing and recommendation, like Douban (http://www.douban.com), may categorize products into books (further classified by topics like history, fiction, etc), movies, etc.

Fig. 1 may appear to be a logical schema, with a table each for the 5 entities and 5 relationships in Sec. 2.1. Rather, like the snowflake schema [14], sonSchema is in fact a conceptual schema that can be translated into different logical schemas.

For example, **private_msg** can be instantiated as two tables, one for messages between two users, and one from a group to its members (hence the attribute `from_u(g)_id` in Fig. 1). Similarly, **social_product** in Fig. 1 has a `creator_id` that may be a `u_id` or a `group_id`. Thus, Facebook events (http://socialmediadiyworkshop.com/2010/03/manage-your-facebook-event/), like a wedding organized by a bride or a rally organized by a union, can belong to different tables. Also, to allow a **post** to be shared (e.g. retweeted) within a restricted group, one can add a *shared_to* table, as an additional instance of **social_product_activity**.

Adding new attributes is costly for a db-relational system. To overcome this problem, we make sonSchema extensible through multiple instantiations of its table schemas. A social network service provider that wants to introduce a new service (image repository, event organizer, etc.) just adds corresponding tables for **social_product**, **post**, etc.

This again illustrates how the sonSchema design is service-oriented (G2), and its extensibility provides generality (G1).

3 Application: social network data management

We now verify that sonSchema can be instantiated for some social network services, and consider two applications: the ACM Digital Library (we will later use it to analyze the social network of ACM authors) and coupon dissemination via a social network.

3.1 Online social networks

Online social networks are driven by an entity (image, video, etc.), thus generating a corresponding set of tables. E.g. when Jay uploads a photograph (a **post**), Kaye may comment on it and Elle may tag Kaye (two separate instantiations of **response2post**), while mum may email Jay about the photograph (a **private_msg**). We have designed sonSchema to match this general structure of social interactions, so we expect its restricted form to suffice for current and future social networks.

The following considers the instantiation in greater detail:

Undirected social graphs Facebook, Renren and Linkedin are major social networks, where **friendship** models classmates, colleagues, etc. **post** or **social_product** models "status", "note", "photo", "event", etc., and **response2post** or **social_product_activity** models "like", "comment", "share", "join_event", "tag", "via", etc.

Directed social graphs Twitter and Sina Weibo (http://www.weibo.com) are large social networks with directed follower-followed sn-relationships. Again, **post** would model tweets, and **response2post** would model activities like "retweet", "comment" and "reply".

Mixed social graphs For a service like Flickr, **friendship** can have two instantiations, one for directed and the other for undirected links in user contact lists.

3.2 ACM Digital Library (ACMDL)

Facebook users often belong to overlapping social networks, some implicitly defined. Can sonSchema model such networks? Since Facebook data is not publicly available, we use ACMDL as a proxy: the publications therein define at least two social networks — an explicit bidirectional coauthorship, and an implicit directional citation.

In detail [4], sonSchema can model ACMDL with **user** instantiated as *author*, **post** as *papers* (with `group_id` identifying the publication venue), **friendship** as *coauthorship*, **response2post** as *citation*, and **group** as *conference/journal*. Many implicit social networks can be found via selection (e.g. `affiliation`) and joins of these tables.

3.3 Coupon dissemination

To extract value from a social network, a service provider like Groupon (http://www.groupon.com) may use it to disseminate coupons for businesses. sonSchema can model such a service by instantiating **user** as *business* and *consumer* (2 tables), **social_product** as *coupon*, and **private_msg** as *coupon_dissemination*; coupon forwarding among members of a social group [19] is a separate instance of **private_msg**.

4 Application: social network analysis

We want to verify that db-relations are better than graphs for modeling and analyzing social network data. To do so, we examine three well-studied problems in the literature, namely community structure (Sec. 4.1), cluster discovery (Sec. 4.2) and action propagation (Sec. 4.3). In particular, we show how such social network analysis can be done with SQL queries, instead of graph algorithms.

4.1 Community structure

Current techniques for studying community definition and evolution invariably use graphs. We now examine how this can be done differently, using sonSchema.

Community definition Some social networks are explicitly declared [3] (e.g. LiveJournal, http://www.livejournal.com), but many are only implicitly defined. With sonSchema, one can easily discover, say, camera fans or bird watchers in the Flickr data by querying **response2post** for relevant tags. Similarly, one can extract the communities for data mining and cloud computing from ACMDL by specifying relevant journals and conferences, or keywords in the attribute `abstract` for *papers*.

In social network analysis, the interaction graph is most important [25]. Since interaction is explicitly represented by **private_msg**, **response2post** and **social_product_activity**, extracting the interaction graph is easy with sonSchema. In the Flickr example, if one defines an interaction as two users commenting on each other's photographs, then such user pairs can be retrieved with an appropriate join query.

Community growth One way of studying the growth of a community C is to determine the probability that someone on the fringe of C joins C [3]. Let $fringe(C)$ consist of users who are not in C but have a friend in C. For integer $K \geq 0$, let $prob(C|K, \Delta T)$ be the probability that a user $x \in fringe(C)$ will join C within the next time period ΔT if x has K friends in C.

We can compute $prob(C|K, \Delta T)$ in three steps: (i) run queries to take two snapshots of C — C_1 at time T_1 and C_2 at time $T_2 = T_1 + \Delta T$; (ii) retrieve those x in $fringe(C_1)$ with K friends in C_1. (iii) count how many x from (ii) are in C_2.

Note that, in the above, there are other, natural variations in the definition of $fringe(C)$ and $prob(C|K, \Delta T)$, as illustrated later in Sec. 5.1.

4.2 Cluster discovery

A social network often contains clusters, whose members interact more among themselves than with others outside their cluster. E.g. Facebook users may have clusters from the same school or club. We now illustrate cluster discovery via sonSchema.

For better clarity, we use the sonSchema model of ACMDL from Sec. 3.2. (Here, a cluster may form around Codd, or Knuth.) Let coauthorship frequency of an author x be the sum of number of coauthors on all papers written by x (if coauthor y appears on n papers, y is counted n times).

Algorithm 1: Cluster discovery for a social network of authors

input : sonSchema dataset for publications, K: choice of top-K clusters, τ: threshold for a qualified coauthorship frequency
output: a set of clusters: $CSet$

1 **for** *each* $au \in$ *author* **do**
  `/* coauthor_freq(author_id,freq,isVisited) is a table */`
2 $coauthor\_freq(au).freq$ = number of coauthorships per author au (**SQL1**);
3 **repeat**
4 let $a = au$ that has the highest $coauthor\_freq.freq$ and not isVisited (**SQL2**);
5 let cluster $C = \{a\}$ initially;
6 let queue Q=$\{a\}$ initially;
7 let $P = \emptyset$ initially;
8 **repeat**
9 let a=Q.pop();
10 let B={b | ($\langle a, b\rangle \in$coauthor OR $\langle b, a\rangle \in$coauthor)) AND (!b.isVisited)} (**SQL3**);
11 **for** *each* $b \in B$ **do**
12 let $P_{a,b}$ = set of `paper_id`s coauthored by $\langle a, b\rangle$;
13 **if** *(coauthor_freq(b,C)* $> \tau$*) AND (*$\exists p \in P_{a,b}$ *such that* $p \notin P$*) (**SQL4**)* **then**
14 C.add(b);
15 Q.push(b);
16 P.add(p);
17 *coauthor_freq*(b).isVisited = true;
  `/* mark b as visited in table author_freq */`
18 **until** *(Q is empty)* ;
19 Output cluster C;
20 **until** *(The K-th result has been found)* ;

Our Algo. 1 (above) for cluster discovery returns the top-K clusters, defined by using an appropriate quality metric, like coauthorship frequency and number of papers. It has an outer loop to find K clusters, and an inner loop to grow each cluster. It finds the coauthorship frequency of all authors (**SQL1**), then picks the author with the highest frequency to grow a cluster C (**SQL2**). The latter is done by iteratively picking from coauthors of those in C (**SQL3**) and applying the quality metric (**SQL4**). This metric has a threshold τ for coauthorship frequency between a pair of authors.

A reasonable choice is $\tau = f/n$, where f is the total number of coauthorships and n is the number of authors in the cluster, so $coauthor\_freq(b, C) > f/n$ (**SQL4**) means that adding b to C would raise the average coauthorship in the cluster. Thus, only authors with strong coauthorship ties with C are admitted.

The other quality metric (**SQL4**) requires that b cannot join C if b cannot add a new paper for C. Intuitively, a cluster with more papers is better.

Algo. 1 stops trying to grow C when it cannot be expanded (line18). It then picks another author as seed to grow the next cluster, unless there are already K clusters.

The crucial parts of Algo. 1 are straightforward SQL select-join queries, and critical calculations concerning quality can be easily coded and efficiently executed. There is no use of graph algorithms like network flow [8], spectral algorithms [1] and hierarchical decomposition [6]. Clustering quality is judged semantically by coauthorships per author and number of papers per cluster, rather than syntactically by conductance [10]

and modularity [6], etc. One can use a small K to generate only the most significant clusters, without having to decompose the entire graph. Similarly, one does not need to load the whole graph into memory (a problem for massive datasets); rather, Algo. 1's memory requirement is of the order of the cluster size, which is usually small [13].

4.3 Action propagation

Social network analysis should focus on user interaction, as that is the very reason people join these networks. In current efforts to extract value from social networks (e.g. coupon dissemination in Sec. 3.3), one key idea is that friends influence one another.

Analyzing action propagation through a network starts with a log of user actions [9]. With sonSchema, we can extract this log as a table $Action(u, \alpha, t_u)$ from, say, **social_product_activity**, indicating user u performed action α at time t_u. We say α propagates from u to v if and only if $\langle u, v\rangle$ is an edge, $\langle u, \alpha, t_u\rangle \in Action$ and $\langle v, \alpha, t_v\rangle \in Action$ for some $t_u < t_v$, and $\langle u, v\rangle$ formed before t_v.

This definition for action propagation can be easily evaluated with a SQL query, and Sec. 5.3 will demonstrate this with a multi-dimensional analysis of Twitter dynamics.

5 Experiments

We now present experiments with the algorithms described in Sec. 4. The experiments are done with one dataset from the ACM Digital Library[3] and another from Twitter (publicly available). Our ACMDL dataset has 445656 authors, 265381 papers, 4291 journals and conferences, 1741476 citation pairs and 170020 coauthorships. The schemas for these datasets are described in Sec. 3.1 and Sec. 3.2. All experiments are run on a Linux machine with 1.15GHz AMD Opteron(tm) 64-bit processor and MySQL (version 5.1.51). No index is built on non-key attributes.

Before experimenting with our SQL-based algorithms for social network analysis, we should do a head-to-head comparison between MySQL and a graph database system. Where a social network is modeled as a graph and for queries that are expressible as graph traversal, one expects a graph database system will have an advantage over a db-relational system in processing the queries. This is because graph traversal corresponds to table joins, which is an expensive operation. However, graph traversal in db-relational systems can be accelerated via the use of an appropriate index.

For the comparison, we use Neo4j (a leading graph database system) and equip MySQL with a GRIPP index [23] that is implemented as stored procedures to support efficient graph traversal. We interpreted *coauthorship* in our ACMDL dataset as an undirected graph, with 59695 nodes (authors) and 106617 edges (coauthorships).

We use a canonical query in graph traversal, namely reachability, for the comparison. Let *distance* d between nodes x and y be the smallest number of edges connecting x and y. For each d, we randomly pick 1000 $\{x, y\}$ pairs and measure the time to determine reachability at distance d between x and y, using Neo4j and GRIPP.

Fig. 2 shows that Neo4j is indeed faster for small distances. However, its query time accelerates as d increases because Neo4j uses the Dijkstra algorithm (which has

[3] http://dl.acm.org/ Many thanks to Craig Rodkin at ACM HQ for providing the dataset.

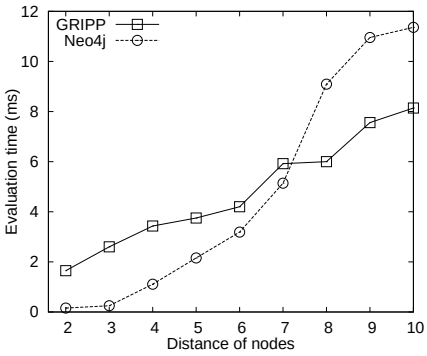

Fig. 2: **Comparison of average query time for reachability test**

time periods	average community size	total number of papers	avg. #coauthors per paper
1975-1980	5	144	2.1
1981-1985	6	356	2.6
1986-1990	6	366	2.9
1991-1995	9	511	4.2
1996-2000	11	535	4.7

Table 1: **ACMDL community statistics over time.**

quadratic time complexity). In contrast, traversal time with GRIPP scales linearly. It is even faster than Neo4j for large distances, which is an advantage for large datasets.

5.1 Community structure

We now use ACMDL to demonstrate how, with sonSchema, community definition is easy and can provide insight into community dynamics.

Community definition We illustrate the ease of community definition (Sec. 4.1) with this question: *How do coauthor communities evolve over time?*

Some author lists include people who are actually not active in the research collaboration. To help exclude such authors, we define a coauthor community C to include only those who coauthored at least two papers with others in C. On the other hand, to avoid isolating students and fresh PhDs, we add them to their supervisors' community.

We select only papers in SIGMOD conferences and divide time into 5-year periods. Table 1 shows that the average community size increases with time. To understand this increase, we run queries to count the number of papers and coauthors. Table 1 shows that the average number of coauthors per paper also increases over time. The insight we get is: *coauthor communities get larger because there are more authors per paper.*

A sample query to get the number of coauthors in a time period is:

```
select count(*) from coauthor, proceedings p, conference c
where coauthor.paper_id=p.paper_id
  and p.proceeding_id=c.proceeding_id
  and year(c.publication_date)>1995
```

and year(c.publication_date)<=2000
and c.proc_profile like '%SIGMOD%'

Note the use of aggregation (count), joins, selection (year) and non-key attributes (SIGMOD). Such a multi-dimensional analysis using a combination of operators is easy for SQL, but hard to formulate as a query on any graph-based model of coauthorship.

Community growth Sec. 4.1 describes how one can study the growth of a community C by measuring the probability that someone in $fringe(C)$ later joins C. We now demonstrate how sonSchema facilitates the analysis of social dynamics with this question: *How does coauthorship history affect the joining probability?*

A graph metric often used for cluster discovery is conductance [10], whose value is smaller for a tighter community (see Sec. 5.2). Analysis via sonSchema complements such graph-based analysis; we illustrate this now by choosing 10 communities from year 1999 that have conductance smaller than 0.06, and size from 4 to 8 authors. For each community C, we define $fringe(C)$ as those authors x not in C but have written a paper with someone in C in the last 10 years (i.e. 1990–1999). These x are further classified into: **Class (I)** 0 coauthorship in 1997–1999; **Class (II)** 1 to 3 coauthorships in 1997–1999; **Class (III)** 4 or more coauthorships in 1997–1999.

For each class, we then compute $prob(C|K, 1\text{ year})$ that an x in $fringe(C)$, who has more than K coauthorships with members of C over the last 10 years, joins C between 1999 and 2000. For $K = 8$, the experiments find that:

• A Class(I) author x, i.e. who has 0 coauthorship with C in 1997–1999, has a low probability 0.16 of joining C, even though x has more than 8 coauthorships with C.

• A Class(III) author x, i.e. who has 4 or more coauthorships with C in 1997–1999, has a high probability 0.94 of joining C.

(For Class(II), the probability is 0.74.) The insight we get is: *recent coauthorship affects* $prob(C|K, 1\text{ year})$ *more than a high coauthorship count (that is spread over 10 years).*

Again, extracting such an insight involving time and aggregation, etc., is easy with sonSchema, but difficult if ACMDL is modeled as a graph.

5.2 Cluster discovery

To evaluate our SQL-based Algo. 1 for cluster discovery, we compare it to the heuristic-based hierarchical decomposition [6] and the approximation-based local spectral algorithm [12], which represent the two major graph-based discovery techniques [13]. Our Algo. 1 uses the author with the highest coauthorship frequency to seed each cluster, so we rank the clusters in the order that they are generated. An empirical comparison of current techniques found that minimizing conductance produces better clusters [13], so we use conductance to rank (smaller is better) clusters for the graph algorithms.

To compare the algorithms, we use data from the data mining community (identified by the keywords, title and abstract of each paper in ACMDL). The choice of threshold τ and stop condition for Algo. 1 are as mentioned in Sec. 4.2.

Table 2 shows that the heuristic- and approximation-based techniques generate big clusters: their minimum top-20 cluster sizes are about 30-50, and the approximation-based technique have average cluster size of 1452 authors.

	maximum		minimum		average	
algo.	all	top-20	all	top-20	all	top-20
SQL	34	24	3	3	5	9
heuristic	2358	112	3	32	6	51
approx.	7733	150	3	46	1452	105

Table 2: **Number of authors in a cluster (excluding trivial clusters of size 1 and 2).**

	maximum		minimum		average	
algo.	all	top-20	all	top-20	all	top-20
SQL	296	208	4	4	17	75
heuristic	8826	247	2	69	16	164
approx.	20251	505	3	203	3925	355

Table 3: **Number of coauthorships in a cluster.**

	maximum		minimum		average	
algo.	all	top-20	all	top-20	all	top-20
SQL	20.3	16.9	1.3	1.3	3.1	8.9
heuristic	14.0	5.2	0.7	1.7	1.6	3.3
approx.	6.7	5.2	1.0	3.0	2.7	3.5

Table 4: **Number of coauthorships per author in a cluster.**

	maximum		minimum		average	
algo.	all	top-20	all	top-20	all	top-20
SQL	0.96	0.96	0.008	0.024	0.381	0.556
heuristic	0.20	0.00	0.000	0.000	0.001	0.000
approx.	0.14	0.01	0.002	0.003	0.014	0.006

Table 5: **Conductance of a cluster.**

Besides size, we also examine the *quality* of the clusters. Table 3 shows that the number of coauthorships per cluster is large for the approximation-based technique. Moreover, it generates many clusters of more than 1000 coauthorships each, and the average cluster has 3925 coauthorships; these numbers are arguably too high for the clustering to be meaningful. Table 4 further shows that our SQL-based technique is significantly better in terms of coauthorships per author. For example, the heuristic-based clusters have an average of only 1.6 coauthorships per author, partly because they have bigger sizes (Table 2).

Algo. 1 thus identifies clusters that are smaller in size and higher in quality. Yet, as Table 5 shows, the clusters found by the graph algorithms have very small conductance. It also shows that the SQL-based clusters have very high conductance. The insight we get is: *Conductance minimization is neither sufficient nor necessary for discovering good clusters.*

Finally, we compare the size distribution for Algo. 1 and the heuristic-based technique. (We omit the approximation-based technique because it gives just one cluster for each size [13].) Fig. 3 shows that the heuristic-based technique generates many small clusters (5083 clusters of 2 authors each) and another 7341 clusters that include huge ones (up to 2358 authors). In contrast, Algo. 1 finds 2814 clusters (none with size 1 or 2), the largest of which has 34 authors. Thus, Algo. 1 divides the social network into a small number of congenial clusters, whereas the heuristic-based technique produces many clusters that have unfriendly size (too small or too big).

Note that one can easily modify Algo. 1 to use some other preferred metrics.

5.3 Action propagation

We now demonstrate how our sonSchema model of Twitter data (Sec. 3.1) can support an analysis of action propagation (Sec. 4.3); statistics for the Twitter dataset are given in Table 6. Consider this question: *Does the number of followers a user has affect how far her tweet propagates?*

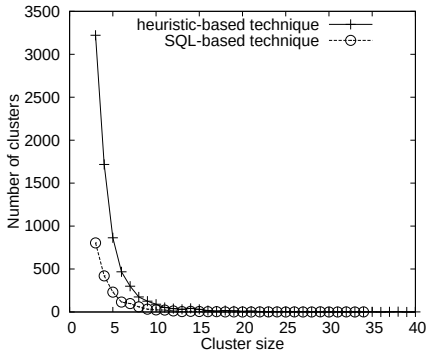

Fig. 3: **Comparing cluster size distribution. (The heuristic-based technique generates 5083 clusters of size 2, which are not shown here.)**

time period	2008-11-11 to 2009-11-06
number of users	456107
maximum/average/minimum number of followers	500/332/1
maximum/average/minimum number of people that a user follows	198/4/1
number of followed-follower pairs	815211
number of user pairs who follow each other	2494
number of tweets	28688584
number of tweets that are original tweets	26161245
number of tweets that are retweets	498991
number of users who have posted or replied to at least one tweet	274315
maximum/average/minimum number of tweets that are a users's original post	383/95.5/1
maximum/average/minimum number of tweets that are a users's reply	200/12.6/1
maximum/average/minimum number of tweets that are just retweets	195/4.4/1

Table 6: **Statistics for Twitter dataset.**

We first prepare a table *retweet* that stores all retweet information for each tweet up to 4 hops. We then group users according to how many followers they have, as shown in Table 7; e.g. there are 352 users who each has 1–50 followers, 161 users who each has 51–100 followers, etc. To compute Table 7, we first compute table *follower_cnt* that stores `user_id` and her number of followers, and table *follower_nhop* that stores `user_id` and her followers up to 3 hops. We then run the query

select fc.`uid`, count(rt.`tid`)
from *follower_cnt* fc, *follower_nhop* fn, *retweet* rt
where fn.`uid` = fc.`uid` and fc.`uid`=rt.`uid`
  and fc.`f_count`$\geq$1 and fc.`f_count`$\leq$50
  and rt.`ruid`=fn.`fid` and fn.`hop`=1
group by fc.`uid`

Computation for other ranges are similar. This query illustrates the need for several joins and aggregation in multiple dimensions (followers and retweets) if one is to gain insight into network dynamics.

Table 8 presents our query results on propagation depth. Consider the row for 3 hops: It shows that the 3-hop penetration increases from 0% for a user with 1–50 followers to 0.5% for one with 201–300 followers. Beyond that, the 3-hop penetration

n	1–50	51–100	101-200	201–300	301–400	401–500
#users with n followers	352	161	242	147	141	1433

Table 7: **Breadth of following: 352 users who each has 1–50 followers, etc.**

	range in number of followers					
K-hop	1–50	51–100	101–200	201–300	301–400	401–500
1	97.7%	97.9%	98%	97.6%	98.5%	99.4%
2	2.3%	1.8%	1.7%	1.9%	1.4%	0.4%
3	0%	0.3%	0.3%	0.5%	0.1%	0.1%
$\geq$4	0%	0%	0%	0%	0%	0%

Table 8: **Depth of propagation: for a user u with 201–300 followers, 0.5% of retweets of u's messages were by 3-hop followers, etc.**

actually drops for users with more followers. This shows that it is not true that the more followers u has, the farther u's messages will propagate, i.e. the insight we get is: *breadth of following does not determine depth of penetration.*

This observation would be relevant to merchants who are considering coupon dissemination via social networks (Sec. 4.3).

6 Related work

Most of the related work were already cited above. We now mention some others.

To see the novelty in sonSchema, one can compare it to the schemas for SoQL [17] and NetIntel [24]; they consist of just nodes, edges, node types and edge types. In contrast, sonSchema explicitly models social products and interactions, as well as user-product and product-product relationships.

A survey of the literature on community structure [3,12,20], link prediction [11,15] and social influence [9,22] shows that graphs are the predominant model for social networks. However, a social graph suffices for some of these studies because they do not analyze the interactions (which we do through **response2post** and **social_product_activity**).

Cluster discovery techniques can be classified under heuristics and approximations [13]; Sec. 4.2 therefore compares our SQL-based technique to a representative of each.

Several papers have studied DBLP as a social network [3,26]. Instead, we use ACMDL, which is much richer in terms of information (citation, affiliation, keywords, abstracts, etc.). Goyal et al.'s study of action propagation [9] requires an action log for Flickr, which we do not have. Instead, we use data from Twitter (with *tweet* as *Action*).

7 Current work

We can use sonSchema's restricted form to re-engineer MySQL for scalability. E.g. we believe it is possible to incorporate its schema graph into a concurrency control and thus provide strong consistency, but without the ACID bottleneck [21].

For now, we are studying the structure that sonSchema imposes on the space of all join trees. This study may identify bushy strategies for multi-way joins that execute faster than strategies that are produced by current optimizers [7].

References

1. R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *FOCS*, pages 475–486, 2006.
2. R. Angles and C. Gutiérrez. Survey of graph database models. *Comput. Surv.*, 40(1), 2008.
3. L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *Proc. KDD*, pages 44–54, 2006.
4. Z. Bao, Y. C. Tay, and J. Zhou. A conceptual schema for social networks. http://sonsql.comp.nus.edu.sg/rsn.pdf.
5. Z. Bao, Y. Zeng, and Y. C. Tay. sonLP: Social network link prediction by principal component regression. In *Proc. ASONAM*, 2013 (to appear).
6. A. Clauset, M. E. J. Newman, , and C. Moore. Finding community structure in very large networks. *Physical Review E*, pages 1– 6, 2004.
7. S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *Proc. ICDT*, pages 54–67, 1995.
8. G. W. Flake, R. E. Tarjan, and K. Tsioutsiouliklis. Graph clustering and minimum cut trees. *Internet Mathematics*, 1(4), 2003.
9. A. Goyal, F. Bonchi, and L. V. Lakshmanan. Learning influence probabilities in social networks. In *WSDM*, pages 241–250, 2010.
10. R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *J. ACM*, 51(3):497–515, 2004.
11. H. Kashima and N. Abe. A parameterized probabilistic model of network evolution for supervised link prediction. In *Proc. ICDM*, pages 340–349, 2006.
12. J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In *WWW*, pages 695–704, 2008.
13. J. Leskovec, K. J. Lang, and M. Mahoney. Empirical comparison of algorithms for network community detection. In *Proc. WWW*, pages 631–640, 2010.
14. M. Levene and G. Loizou. Why is the snowflake schema a good data warehouse design? *Inf. Syst.*, 28(3):225–240, 2003.
15. D. Liben-Nowell and J. M. Kleinberg. The link-prediction problem for social networks. *JASIST*, 58(7):1019–1031, 2007.
16. R. N. Lichtenwalter, J. T. Lussier, and N. V. Chawla. New perspectives and methods in link prediction. In *Proc. KDD*, pages 243–252, 2010.
17. R. Ronen and O. Shmueli. SoQL: A language for querying and creating data in social networks. In *Proc. ICDE*, pages 1595–1602, 2009.
18. M. Rys. Scalable SQL. *Commun. ACM*, 54(6):48–53, June 2011.
19. S. Shakkottai, L. Ying, and S. Sah. Targeted coupon distribution using social networks. *SIGMETRICS Perf.. Eval. Rev.*, 38:26–30, Jan 2011.
20. M. Spiliopoulou. Evolution in social networks: A survey. In *Social Network Data Analytics*, pages 149–175. Springer, 2011.
21. M. Stonebraker and R. Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, 54:72–80, June 2011.
22. J. Tang, J. Sun, C. Wang, and Z. Yang. Social influence analysis in large-scale networks. In *Proc. KDD*, pages 807–816, 2009.
23. S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proc. SIGMOD*, pages 845–856, 2007.
24. M. Tsvetovat, J. Diesner, and K. Carley. NetIntel: A database for manipulation of rich social network data. Technical Report CMU-ISRI-04-135, Carnegie Mellon University, 2005.
25. C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *Proc. EuroSys*, pages 205–218, 2009.
26. O. R. Zaiane, J. Chen, and R. Goebel. DBconnect: mining research community on DBLP data. In *Proc. WebKDD/SNA-KDD*, pages 74–81, 2007.