S3G2: a Scalable Structure-correlated Social Graph Generator

Minh-Duc Pham[1] Peter Boncz[1] Orri Erling[2]

[1] CWI, The Netherlands,
duc@cwi.nl, boncz@cwi.nl,
[2] OpenLink Software, U.K.
oerling@openlinksw.com

Abstract. Benchmarking graph-oriented database workloads and graph-oriented database systems is increasingly becoming relevant in analytical Big Data tasks, such as social network analysis. In graph data, structure is not mainly found inside the nodes, but especially in the way nodes happen to be connected, i.e. structural correlations. Because such structural correlations determine join fan-outs experienced by graph analysis algorithms and graph query executors, they are an essential, yet typically neglected, ingredient of synthetic graph generators. To address this, we present S3G2: a Scalable Structure-correlated Social Graph Generator. This graph generator creates a synthetic social graph, containing non-uniform value distributions and structural correlations, which is intended as test data for scalable graph analysis algorithms and graph database systems. We generalize the problem to decompose correlated graph generation in multiple passes that each focus on one so-called *correlation dimension*; each of which can be mapped to a MapReduce task. We show that S3G2 can generate social graphs that (i) share well-known graph connectivity characteristics typically found in real social graphs (ii) contain certain plausible structural correlations that influence the performance of graph analysis algorithms and queries, and (iii) can be quickly generated at huge sizes on common cluster hardware.

Data in real life is correlated; e.g. people living in Germany have a different distribution in names than people in Italy (location), and people who went to the same university in the same period have a much higher probability to be friends in a social network. Such correlations can strongly influence the intermediate result sizes of query plans, the effectiveness of indexing strategies, and cause absence or presence of locality in data access patterns. Regarding intermediate result sizes of selections, consider:

```
SELECT personID FROM person
WHERE firstName = 'Joachim' AND addressCountry = 'Germany'
```

Query optimizers commonly use the *independence assumption* for estimating the result size of conjunctive predicates, by multiplying the estimates for the individual predicates. This would underestimate this result size, since Joachim is more common in Germany than in most other countries; similar would happen e.g. when querying for `firstName` 'Cesare' from 'Italy'. Overestimation can also easily happen, if we would query for 'Cesare' from 'Germany' or 'Joachim' from 'Italy' (i.e. *anti-correlation*).

This correlation problem has been recognized in relational database systems as relevant, and some work exists to detect correlated properties inside the same table (e.g., see [13]). Still, employing techniques for the detection of correlation is hardly mainstream in relational database management, and this is even more so when we start considering correlations between predicates that are separated by joins. Consider for instance the DBLP example of co-authorship of papers that counts the number of authors that have published both in TODS and in the VLDB Journal:

```
SELECT COUNT(*)
FROM paper pa1 JOIN journal jn1 ON pa1.journal = jn1.ID
     paper pa2 JOIN journal jn2 ON pa2.journal = jn2.ID
WHERE pa1.author = pa2.author AND
      jn1.name = 'TODS' AND jn2.name = 'VLDB Journal'
```

The above query is likely to have a larger result size than a query that substitutes 'TODS' for 'Bioinformatics', even though Bioinformatics is a much larger publication than TODS. The underlying observation is that database researchers are likely to co-publish in TODS and The VLDB Journal, but are much less likely to do cross-disciplinary work. For database technology, this example poses (i) a challenge to the optimizer to adjust the estimated join hit ratio of `pa1.author = pa2.author` downwards or upwards depending on other (selection or join) predicates in the query (i-i) provide indexing support that can accelerate this query: the anti-correlated query (Bioinformatics and The VLDB Journal) has a very small result size and thus could theoretically be answered very quickly. However, just employing standard join indices will generate a large intermediate result for the Bioinformatics sub-plan containing all Bioinformatics authors, of which only a minute fraction is actually useful for the final answer.

Summarizing, correlated predicates are still a frontier area in database research, and such queries are generally not well-supported yet in mature relational systems. This holds still more strongly in the emerging class of graph database systems, where we argue the need for correlation-awareness in query processing is even higher.

In the particular case of RDF, its graph data model is expressly chosen to work without need for an explicit schema, such that graph datasets get stored as one big pile of edges (in particular, subject-property-object "triples"). Here we see a dualism between structure and correlation: in the relational model, certain structure is explicit in the schema, whereas in RDF such structure only re-surfaces as structural *correlation*. That is, it will turn out a journal paper (subject) always happens to have one `title` property, one `issue` property, one `journalName`, etc; and that these properties exclusively occur in connection to journal issues. The extreme flexibility of RDF systems in the data they can store, thus poses a significant challenge to SPARQL query optimizers, as they need to understand such correlations to get the planning of even basic queries right. Other graph database systems which use a richer data model, where nodes have a declared structure, suffer less from this problem. Still, when considering that graph analysis queries often involve a combination of (property) value constraints and structural constraints (pattern matching), it is likely that correlations between the structure of the graph and the values in them will strongly affect the performance of systems and algorithms. Yet, systems are not sufficiently aware of this, and existing graph benchmarks do not specifically test for this; and synthetic graphs used for benchmarking do not have such structure correlations. As such, we argue that for *benchmarking* graph data analysis systems and algorithms, it would be very worthwhile if a data generator could generate *synthetic graphs* which such *correlated structure*. To our knowledge, there exists no solution for generating a scalable random graph with value and structure correlations. Existing literature on random graph generation [4, 10, 6, 8] either does not consider node properties at all or ignores correlations between them.

In this paper, we describe the Scalable Structure-correlated Social Graph Generator (S3G2), and its underlying generic conceptual correlated graph generation framework. This framework organizes data generation in multiple phases that each center around a *correlation dimension*. In the case of our social graph use case, these dimensions are (i) education and (ii) personal interests. The data generation workflow is constrained by *correlation dependencies*, where certain already generated data influences the generation of additional data. A graph generator generates new nodes (with property values), and edges between these nodes and existing nodes. The probability to choose a certain value from a dictionary, or the probability to connect two nodes with an edge are thus influenced by existing data values. For instance, the birth location of a person influences probability distribution of the `firstName` and `university` dictionaries. As another example, the probability to create a friendship edge is influenced by (dis)agreement on `gender`, `birthYear` and `university` properties of two person nodes.

A practical challenge in S3G2 is that a naive approach to correlated graph generation would continuously access possibly any node and any edge in order to make decisions regarding the generation of a next node or edge. For generating graphs of a size that exceeds RAM, such a naive algorithm would grind down due to expensive random I/O. To address this challenge, we designed a S3G2 graph generation algorithm following the MapReduce paradigm. Each pass along one correlation dimension is a Map phase in which data is generated, followed by a Reduce phase that sorts the data along the correlation dimension that steers the next pass. We show that this algorithm achieves good parallel scale-out, allowing it e.g. to generate 1.2TB of correlated graph data in half an hour on a Hadoop cluster of 16 machines.

Contributions of our work are the following: (1) we propose a novel framework for specifying the generation of correlated graphs, (2) we show the usefulness of this framework in its ability to specify the generation of a social network with certain plausible correlations between values and structure, and (3) we devise a scalable algorithm that implements this generator as a series of MapReduce tasks, and verify both quality of its result as well as its scalability. In our vision, this data generator is a key ingredient for new benchmarks for graph query processing.

Outline. In Section 1, we present our framework for the generation of correlated graphs, and describe how such it maps on a MapReduce implementation. In Section 2 we use our framework to generate a synthetic social network graph. In Section 3 we evaluate our approach, confirming that the generated data has typical social network characteristics, and showing the scalability of our generator. Finally, in Section **??**, we review related work before concluding in Section 4.

1 Scalable Structure-correlated Social Graph Generator (S3G2)

We first formally define the end product of S3G2 which is essentially a directed graph of objects, and introduce the main ingredients of the S3G2 framework. Then, we describe the MapReduce-based generation algorithm that follows from these ingredients.

S3G2 generates a directed labeled graph, where the nodes are objects with property values, and their structure is determined by the *class* a node belongs to. Such a data model is commonly provided by graph database systems, and is more structured than RDF (though it can of course be represented in RDF, and our S3G2 implementation in fact generates RDF output).

Definition 1. *S3G2 produces a graph G(V, E, P, C) where V is a set of nodes, E is a set of edges, P is a set of properties and C is a set of classes.*

$$V = L \cup \bigcup_{c \in C} O^c$$
$$P = \left\{P^{L(x)} \,|\forall x \in C\right\} \cup \left\{P^{E(x,y)} \,|\forall x, y \in C\right\}$$
$$E = \left\{(n_1, n_2, p)|n_1 \in O^x \wedge ((n_2 \in L \wedge p \in P^{L(x)}) \vee (n_2 \in O^y \wedge p \in P^{E(x,y)}))\right\}$$

in which O^c is an object of class c in C; L is the set of literals; $P^{L(x)}$ is set of literal properties of class x in C; $P^{E(x,y)}$ is the set of properties representing relationship edges that go from instances of class x to class y.

1.1 Main S3G2 Concepts

We now discuss the conceptual framework behind S3G2, which are (i) property dictionaries, (ii) simple subgraph generation, and (iii) edge generation along correlation dimensions.

Property Dictionary. Property values for each literal property $l \in P^{L(x)}$ are generated following a property dictionary specification $PD_l(D, R, F)$, consisting of a dictionary D, a ranking function R and a probability function F (if the context is unclear, we can also write D_l, R_l and F_l).

A dictionary D is simply a fixed set of values: $D = \{v_1, .., v_{|D|}\}$. The ranking function R is a bijection $R : D \to \{1, .., |D|\}$ which gives each value in a dictionary a unique rank between 1 and $|D|$. The probability density function $F : \{1, .., |D|\} \to [0, 1]$ steers how the generator chooses values; i.e. by having it draw random numbers $0 \leq p \leq 1$, it chooses the largest rank r such that $F'(r) < p$, where F' is the cumulative version of F, that is $F' = \sum_{i=1}^{r} F(i)$. It finally emits the value v_{pos} from dictionary

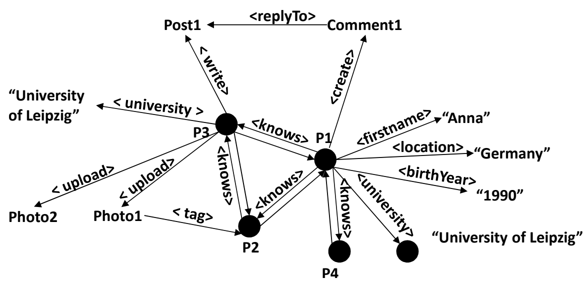


Fig. 1. Example S3G2 graph: Social Network with Person Information.

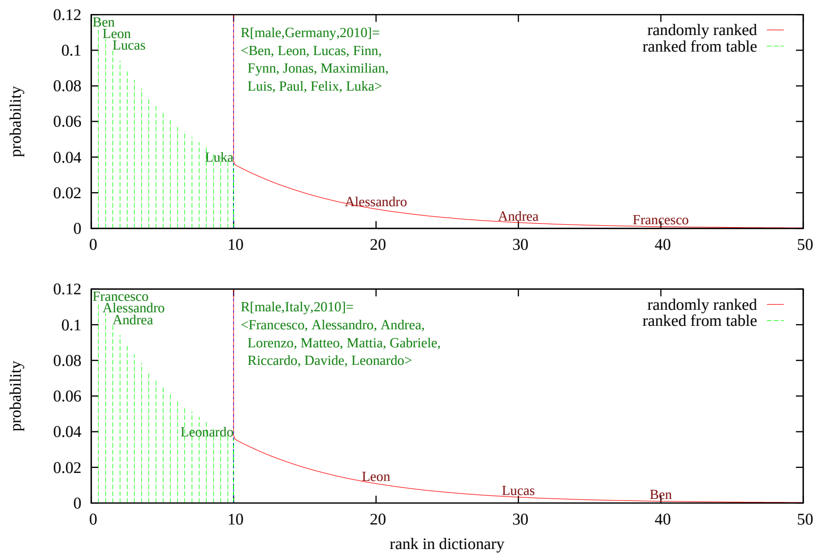


Fig. 2. Compact Correlated Dictionary Distributions: boy names in Germany (up) vs. Italy (lo)

D from position $pos = R(r)$. Thus, our framework can generate data corresponding to any discrete probability distribution.

The idea to have a separate ranking and probability function comes from generating correlated values. In particular, the ranking function $R[z](c)$ is typically parametrized by some parameters z; which means that depending on the parameter z, the value ranking is different. For instance, in case of a dictionary of `firstName` we could have $R[g, c, y]$; e.g. the popularity of first names, depending on `gender` g, `country` c and the year y from the `birthDate` property (let's call this `birthYear`). Thus, the fact that the popularity of first names in different countries and times is different, is reflected by the different ranks produced by function $R()$ over the full dictionary of names. Name frequency distributions do tend to be similar in shape, which is guaranteed by the fact that we use the same probability distribution $F()$ for all data of a property.

Thus, the S3G2 data generator must contain property dictionaries D_l for all literal properties in $l \in L(x)$, and it also must contain the ranking functions R_l, for all literal properties defined in all classes $x \in C$. When designing correlation parameters for a ordering function R_l, one should ensure that the amount of parameter combinations such as (g, c, y) stays limited, in order to keep the representation of such functions compact. We want the generator to be a relatively small program and not depend needlessly on huge data files with dictionaries and ranking functions.

Figure 2 shows how S3G2 compactly represents $R[g, c, y]$, by keeping for each combination of (g, c, y) a small table with only the top-N dictionary values (here N=10 for presentation purposes, but it is typically larger). Rather than storing an ordering of all values, a table like $R[male, Germany, 2010]$ is just an array of N integers. A value j here simply identifies value v_j in dictionary D. This table thus only stores the top-N ranked values. In order to avoid storing more information, the values ranked lower than N get their ranks assigned randomly. Given that in a monotonically decreasing probability function like the geometric distribution used here, the probabilities below that rank are very small anyway, this approximation only slightly decreases the plausibility of the generated values. In Figure 2 we see on the left that for *(male,Germany,2010)* we keep the 10 most popular boys names, which get mapped on the geometric distribution. All other dictionary values (among which Italian names) get some random ranks > 10. On the right, we see that for *(male,Italy,2010)* these Italian names are actually the most popular, and the German names get arbitrary (and low) probabilities.

Simple Graph Generation. Edges are often generated in one go together with new nodes, essentially starting with an existing node n, and creating new nodes to which it gets connected. This process is guided by a degree distribution function $N(n)$ that first determines how many such new children (or descendants) to generate. In many social networks, the amount of neighbour edges h is distributed following a *power law* distribution ($N(h) \sim \gamma.h^{-\lambda}$).

In the S3G2 framework, it is possible to make the function $N(n_i)$ that determines the degree of a node n_i correlated with its properties, e.g. by having these properties influence λ or γ. For instance, people with many friends in a social network will typically post more pictures than people with few friends (hence, the amount of friend nodes in our use case influences the amount of posted comment and picture nodes).

Generating new nodes and connecting them on the fly among mostly themselves and to an existing node n_i leads to isolated subgraphs that are dangling off the main graph connected to it by n_i. Typically, such subgraphs are small or have the shape of shallow trees if they are larger.

Correlation Dimensions. To generate correlated and highly connected graph data, we need a different approach that generates edges *after* generating many nodes. This is computationally harder than generating edges towards new nodes. The reason is that if node properties influence their connectivity, a naive implementation would have to compare the properties of all existing nodes with all nodes, which could lead to quadratic computational cost and a random access pattern, so the generation algorithm would only be fast as long as the data fits in RAM.

Data correlation actually alleviates this problem. We observe that the probability that two nodes are connected is typically skewed with respect to some similarity between the nodes. Given node n_i, for a small set of nodes that are somehow similar to it, there is a high connectivity probability, whereas for most other nodes, this probability is quite low. This observation can be exploited by a graph data generator by identifying *correlation dimensions*.

For a certain edge label $e \in P^{E(x,y)}$ between node classes O^x and O^y, a correlation dimension $CD_e(M^x, M^y, F)$ consists of two *similarity metric* functions $M^x : n \rightarrow [0, \infty]$, $M^y : n \rightarrow [0, \infty]$, and a probability distribution F :[1,$W.t$]→[0,1]. Here the $W.t$ is a window size, of W tiles with each t nodes, as explained later. Note that in case of friends in a social network, both start and end of the edges are of the same class persons ($O^x = O^y$), so a single metric function would typically be used. For simplicity of discussion we will assume $M = M^x = M^y$ in the sequel.

We can compute the similarity metric by invoking $M(n_i)$ on all nodes n_i, and sort all nodes on this score. This means that similar nodes are brought near each other, and we observe that the larger the distance between two nodes, their similarity difference monotonically increases. Again, we use a geometric probability distribution for $F()$ that provides a probability for picking nodes to connect with that are between 1 and $W.t$ positions apart in this similarity ranking. To fully comply with a geometric distribution, we should not cut short at $W.t$ positions apart, but consider even further apart nodes. However we observe that for a skewed distribution like geometric, the probability many positions away will be minute, i.e. $\leq \epsilon$ ($F(W.t) = \epsilon$). The advantage of this window shortcut is that after sorting the data, it allows S3G2 to generate edges using a fully sequential access pattern that needs little RAM resources (it only buffers $W.t$ nodes).

An example of a similarity function $M()$ could be `location`. Location, i.e., a place name, can be mapped to (longitude,latitude) coordinates, yet for $M()$ we need a single-dimensional metric that can be sorted on. In this case, one can keep (longitude,latitude) at 16-bits integer resolution and mix these by bit-interleaving into one 32-bits integer. This creates a two-dimensional space filling curve called Z-ordering, also known in geographic query processing as QuadTiles[3]. Such a space filling curve "roughly" provides the property that points which are near each other in the Euclidean space have a small z-order difference.

Note that the use of similarity functions and probability distribution functions over ranked distance drive *what* kind of nodes get connected with an edge, not *how many*. The decision on the degree of a node is made prior to generating the edges, using the previously mentioned degree function $N(n_i)$, which in social networks would typically be a power-law function. During data generation, this degree of node n_i is fulfilled by randomly picking the required number of edges according to the correlated probability distributions as described before in the example with person who have many friends generating more discussion posts. In case of multiple correlations, and thus multiple passes along correlation dimensions, we determine for each pass the number of edges to generate for a certain node. The overall degree function $N(n) = \sum N_{CD}(n)$ is the sum of the degree distribution functions over all correlation dimensions CD.

Random Dimension. The idea that we only generate edges between the $W.t$ most similar nodes in all correlation dimensions is too restrictive: unlikely connections in a social network that the data model would not explain or make plausible, will occur in practice. Such random noise can be modeled by partly falling back onto uniformly random data generation. In the S3G2 framework this can be modeled as a special case of a correlation dimension, by using a purely random function as rough metric, and a uniform probability function. Hence, data distributions can be made more noisy by making a pass in random order over the data and generating (a few) additional random edges.

1.2 MapReduce S3G2 Algorithm

In the previous discussion we have introduced the main concepts of the S3G2 framework: (i) correlated data dictionaries (ii) simple graph generation (iii) edge generation according to correlation dimensions. We now describe how a MapReduce algorithm is built using these ingredients.

[3] See http://wiki.openstreetmap.org/wiki/QuadTiles

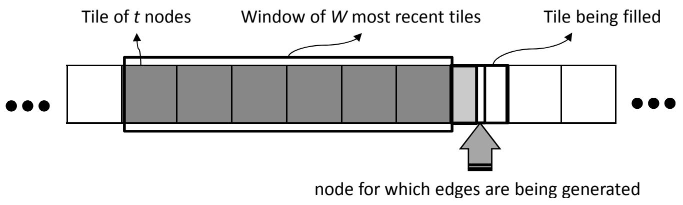

Tile of *t* nodes
Window of *W* most recent tiles
Tile being filled
node for which edges are being generated

Fig. 3. Sliding window of W tiles along the graph.

In MapReduce, a Map function is run on different parts of the input data on many cluster machines in parallel. Each Map function processes its input data item and produces for each a result with a key attached. MapReduce sends all produced results to Reduce functions that also run on many cluster machines; the key determines to which Reducer each item is sent. The Reduce function then processes this stream of data.

In the S3G2 algorithm, the key generated between Map and Reduce is used to *sort* the data for which edges need to be generated according to the similarity metric (the M^x, M^y functions) of the current correlation dimension. As mentioned, there may be multiple correlation dimensions, hence multiple successive MapReduce phases. Both the Map and Reduce functions can perform simple graph generation, which includes generation of (correlated) property values using dictionaries, as described before in the example with boys names in Germany vs. Italy. The main task of the Reduce function is sorting on correlation dimension and subsequent edge generation between existing nodes using a sliding window algorithm described in Algorithm 1.

The main idea of the *sliding window* approach to correlated edge generation is that when generating edges, we only need to consider nodes that are sufficiently similar. By ordering the nodes according to this similarity (the rough metric M^x, M^y) we can keep a sliding window of nodes (plus their properties and edges) in RAM, and only consider generating edges between nodes that are cached in this window. If multiple correlations influence the presence of an edge, multiple full data sorts and sequential sliding window passes are needed (i.e. multiple MapReduce jobs). Thus, each correlation dimension adds one MapReduce job to the whole process, that basically re-sorts the data. One may remark that if the simple graph generation activities that kick off graph generation already generate data ordered along the first correlation dimension, we can save one MapReduce job (as data is already sorted).

The sliding window approach is implemented by dividing the sorted nodes conceptually in tiles of t nodes. When the Reduce function accepts a data item it adds the item to the current tile (an in-memory data structure). If this tile is full, and it has W tiles already in memory, the oldest tile is dropped from memory. This process is visualized in Figure 3. The window approach could also have been implemented with a per-tuple granularity (t=1), but larger tiles make it easier to do memory management.

The Reduce function generates edges for a new incoming node using Algorithm 1, implementing the windowing approach and generating edges along a correlation dimension, by picking for each node n_i a number of $N(n_i)$ nodes to connect to, by having a function $F()$ picks them a certain number of positions back in the window of $W.t$ nodes. This function is most likely to pick nearby nodes; since successive nodes do the same, there is a high likelihood that similar (nearby) nodes have some overlapping neighbours (e.g. friends).

Algorithm 1 GenerateEdges(n_i, $S()$, $F()$)

```
Input: $n_i$: node to generate edges for
Input: $N$: a function that determines the degree of node $n_i$
Input: $F'$: turns a similarity [1,$m$] into a cumulative probability.
 1: for $j = 0; j < N(n_i); j++$ do
 2:     generate a uniform random number $p$ in [0,1]
 3:     $\delta$ = the largest $1 < \delta < W.t$ such that $F'(\delta) < p$.
 4:     if window contains at least $\delta$ nodes then
 5:         if $n_{i-\delta}$ not yet connected to $n_i$ then
 6:             createEdge($n_i$,$n_{i-\delta}$)
 7:         end if
 8:     end if
 9: end for
10: if addNodeToTile(curTile,$n_i$) == $t$ then
11:     if ++curTile == $W$ then
12:         curTile=0;
13:     end if
14:     flushTile(curTile);
15: end if
```

Depending on the node class and its property (correlations), and the various correlation dimensions to use, as well as simple graph generation steps, one needs multiple MapReduce jobs to generate a correlated data graph. In principle, simple graph generation only requires local information (the current node), and can be performed as a Map task, but also as a post-processing job in the Reduce function. Note that node generation also includes the generation of the (correlated) properties of the new nodes.

We should mention that data correlations introduce *dependencies*, that impose constraints on the order in which generation tasks have to be performed. For instance, if the `firstName` property of a person node depends on the `birthYear` and `university properties`, then within simple node generation, the latter properties need to be generated first. Also, if the discussion posts forum that a user might have below a posted picture involves the friends of that user, the discussion node generation should follow the generation of all `friend` edges. Thus, the correlation rules one introduces, naturally determine the amount of MapReduce jobs needed, as well as the order of actions inside the Map and Reduce functions.

2 Case study: generating social network data

In this section, we show how we applied the S3G2 framework for creating a social network graph generator. The purpose of this generator is to provide a dataset for a new graph benchmark, called the Social Intelligence Benchmark (SIB).[4] As we focus here on correlated graph generation, this benchmark is out of scope for this paper. Let us state clearly that the purpose of this generator is *not* to generate "realistic" social network data. Determining the characteristics of social networks is the topic of a lot

[4] See: www.w3.org/wiki/Social_Network_Intelligence_Benchmark

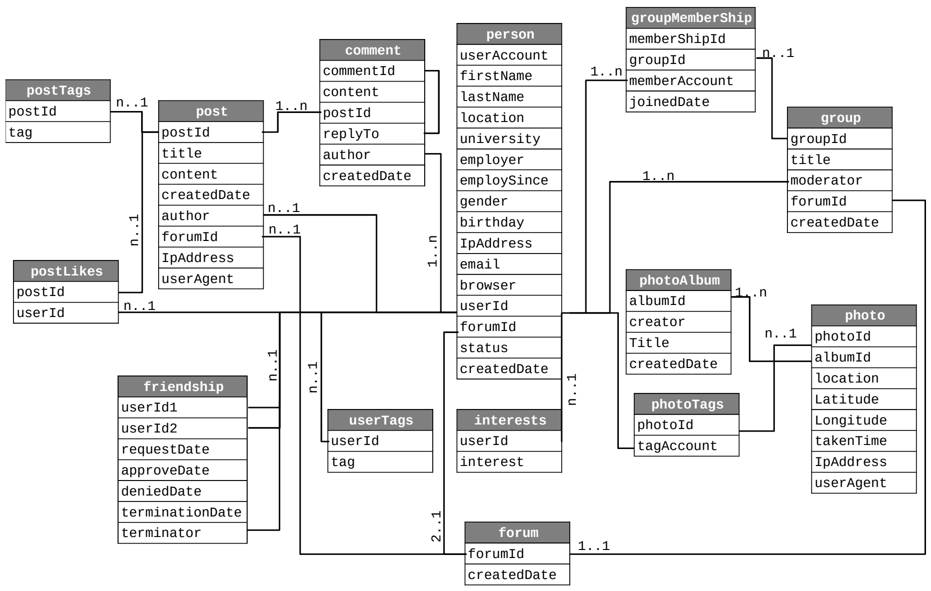


Fig. 4. The Generated Social Network Schema (SIB).

of research, and our use case uses some of the currently insights as inspiration (only). Our data generator introduces some plausible correlations, but we believe that real life (social network) data is riddled with many more correlations; it is a true data mining task to extract these. Given that we want to use the generated dataset for a graph database benchmark workload, having only a limited set of correlations is not a problem; as in a benchmark query workload only a limited set of query patterns will be tested.

Figure 4 shows the ER diagram of the social network. It contains persons and entities of social activities (posted pictures, and comments in discussions in the user's forum) as the object classes of C. These object classes and their properties (e.g., user name, post creation date, ...) form the set of nodes V. E contains all the connection between two persons including their friendship edges and social activity edges between persons and a social activity when they all join a social activity (e.g., persons discussing about a topic). P contains all attributes of a user profile, the properties of user friendships and social activities.

Correlated Dictionaries. A basic task is to establish a plausible dictionary (D) for every property in our schema. For each dictionary, we subsequently decide on a frequency distribution. As mentioned, in many cases we use a geometric distribution, which is the discrete equivalent of the exponential distribution, known to accurately model many natural phenomena. Finally, we need to determine a ranking of these values in the probability distribution (the $R()$ function). For correlated properties, this function is parameterized ($R[z]()$) and is different for value of z. Our compact approximation stores for each z value a top-N (typically N=30) of dictionary values.

The following property value correlations are built in ($R_x[z]$ denoted as $z \rightsquigarrow x$):

- (person.location,person.gender,person.birthDay) $\rightsquigarrow$ person.firstName
- person.location $\rightsquigarrow$ person.lastName

- `person.location` $\rightsquigarrow$ `person.university`
- `person.location` $\rightsquigarrow$ `person.employer`
- `person.location` $\rightsquigarrow$ `person.employSince`
- `(person.location,person.Gender,person.birthDay)` $\rightsquigarrow$ `person.firstName`
- `(person.location,person.Gender,person.birthDay)` $\rightsquigarrow$ `person.interests.interest`
- `person.location` $\rightsquigarrow$ `person.photoAlbum.photo.location`
- `person.location` $\rightsquigarrow$ `person.photoAlbum.photo.location`
- `person.employer` $\rightsquigarrow$ `person.email`
- `person.birthDate` $\rightsquigarrow$ `person.createdDate`
- `person.createdDate` $\rightsquigarrow$ `person.photoAlbum.createdDate`
- `photoAlbum.createdDate` $\rightsquigarrow$ `photoAlbum.photo.takenTime`
- `photoAlbum.photo.location` $\rightsquigarrow$ `photoAlbum.photo.latitude`
- `photoAlbum.photo.location` $\rightsquigarrow$ `photoAlbum.photo.longitude`
- `friendship.requestDate` $\rightsquigarrow$ `friendship.approveDate`
- `friendship.requestDate` $\rightsquigarrow$ `friendship.deniedDate`
- `(friendship.userId1,friendship.userId2)` $\rightsquigarrow$ `friendship.terminator`
- `person.createdDate` $\rightsquigarrow$ `person.forum.createdDate`
- `forum.createdDate` $\rightsquigarrow$ `forum.groupmembership.joinedDate`
- `forum.createdDate,forum.post.author.createdDate` $\rightsquigarrow$ `forum.post.createdDate`
- `post.createdDate` $\rightsquigarrow$ `post.comment.createdDate`

Our main source of dictionary information is DBpedia [2], an online RDF version of Wikipedia, extended with some ranking information derived with internet search engine scripts. From DBpedia one can obtain a collection of place names with population information, which is used as `person.location`. Form the place names, DBpedia also provides population distributions. We use this actual distribution as found in DBpedia to guide the generation of `location`.

The `person.university` property is filled with university names as found in DBpedia. The sorting function $R_{university}[location]$ ranks the universities by distance from the person location, and we keep for each location the top-10 universities. The geometric distribution is used as $F_{university}$ and its parameters are tuned such that over 90% of persons choose one of the top-10. Arguably, it is not plausible that all persons have gone to university, but absolute realism is not the point of our exercise.

From the cities, DBpedia allows to derive country information. DBpedia contains a large collection of person names (first and lastnames) and their country of birth, plus certain explicit information on popularity of first-names per country, which was used as well. Other information was collected manually on the internet, such as a distribution of browser usage, which is not correlated with anything, currently. A special rule for dates is applied that ensures that certain dates (e.g. the date a user joined the network) precede another date (the date that a user became friends with someone). This is simply done by repeating the process of randomly picking a date until it satisfies this constraint.

Correlation Dimensions. In our social network graph, the graph with most complex connectivity is the friends graph. The main correlations we have built in are (i) having studied together (ii) having common interests (hobbies). Arguably, the current schema allows more plausible correlations like working in the same company, or living really close, but these can easily be added following our framework. Further, the concept of `interest` is currently highly simplified to favorite musical artists/composers.

Consequently, there are three correlation dimensions, where the first is studying together, the second is musical interests and the third is random (this will create random connections). The degree of the persons (function $N(n)$) is a power-law distribution that on average produces 30 friends per person node n; it is divided over the three correlation dimensions in a 45%, 45%, 10% split: $N_{study}(n) = N_{interest}(n) = 0.45 * N(n), N_{random}(n) = 0.1 * N(n)$.

For having studied together we use the $M_{study}()$ function described as example in Section 1.1. It depends on `gender`, `university` and `birthYear`, to give highest probability for people of same gender who studied together to be friends. The similarity metric $M_{study}()$ hashes the `university` to the highest 20 bits of an integer; the following 11 bits are taken by filled with the `birthYear` and the lowest bit by `gender`.

The musical-interests correlation dimension is also a multi-valued function, because the persons have a list of favorite artists/composers. The similarity metric $M_{interests}$ creates a vector that holds a score for each genre (S3G2 has predetermined genre vectors for all artists, and the result vector contains the maximum value of all favorite artists for each genre). Then, like the previous example with `location`, z-ordering is used to combine the various genre scores (the genre vector) into a single integer metric.

Graph Generation. The generation of the social graph kicks off by generating person nodes; and all its properties. This "simple graph" generation process forms part of the first MapReduce job and is executed in its Map function. The data *is* generated in a specific order: namely M_{study}. Because this order happens to be the same order as the one needed by the first correlation dimension (studying-together), we do not need a sort phase, and inside the Map function we can already use the window algorithm to generate the first 45% percent of friendship edges using the F_{study} probability distribution.

Because the members of the forum groups of a user (who tag photos and comment on discussions of the user page) and their activity levels are correlated with the user's friends, the objects for these social activities cannot be generated before all friends have been generated. Therefore, the algorithm first continues with all correlation dimensions for friendship. Thus, the Reduce phase of the first MapReduce job sorts the data on $M_{interests}$. The Map phase of the second MapReduce job then executes the window algorithm to generate another 45% of friendship edges using the $F_{interests}$ probability distribution. The key produced is the one for the final, random, correlation dimension, emitting a random number (note that all randomness is deterministic, so the generated dataset is always identical for identical input parameters). The Reduce phase of the second MapReduce job creates this random sort order on person objects. Note that we must sort both persons and their generated friendship edges (user IDs), which are stored together with the person objects.

The Reduce function further performs *simple graph generation* for the social activities. These social activities are subgraphs with only "local" connections and shallow tree-shape, hence can be generated on-the-fly with low resource consumption. Here, the discussion topics are topics from DBpedia articles, and the comments are successive sentences from the article body (this way the discussions consist of real English text, and is kind-of on-topic). The forum group members are picked using a ranking function that puts the friends of a user first, and adds some persons that are in the window at lower ranks; using a geometric probability distribution.

3 Evaluation

We evaluate S3G2 both qualitatively and quantitatively (scalability). We do remark that the qualitative evaluation, namely the realistic-ness of the generated social network is not the main point of S3G2; which rather is the ability to have *some* plausible correlation in the data that future benchmark queries can test. But, the more a S3G2 generated data resembles a real social graph, the better.

Existing literature studying social networks has shown that popular real social networks have the characteristics of a small-world network [12, 15, 5]. We consider the three most robust measures, i.e., the social degrees, the clustering coefficient, and the average path length of the network topology. We empirically show that S3G2 generates a social graph that has these characteristics. In this experiment, we generated small social graphs of 10K, 20K, 40K, 80K, and 160K users. Note that in data volumes are considerable in the default (admittedly verbose) RDF output format: all data associated with one person, which has on average 30 friends and a few hundreds pictures and posts is 1MB. Thus 160K users is 160GB on disk.

Table 1. Graph measurements of the generated social network.

# users	Diameter	Avg. Path Len.	Avg. Clust. Coef.
10000	5	3.13	0.224
20000	6	3.45	0.225
40000	6	3.77	0.225

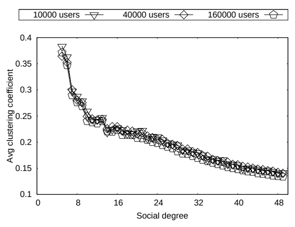
Clustering coefficient. Table 1 shows the graph measurements of the generated social network while varying the number of users. According to the experimental results, the generated social networks have high clustering coefficients of about 0.22 which adequately follow the analysis on real social networks in [15] where the clustering coefficients range from 0.13 to 0.21 for Facebook, and 0.171 for Orkut. Figure 5(a) shows the typical clustering coefficient distribution according to the social degrees that indicates the small-world characteristic of social networks.

Average path length. Table 1 shows that the average path lengths of generated social graphs range from 3.13 to 3.77 which are comparable to the average path lengths of real social networks observed in [15]. These experimental results also conform to the aforementioned observations that average path length is logarithmically proportional to the total number of users. Since we used a simple all-pair-shortest-path algorithm which consumes a lot of memory for analyzing large graphs, Table 1 only shows the results of the average path length for a social graph of 40K users.

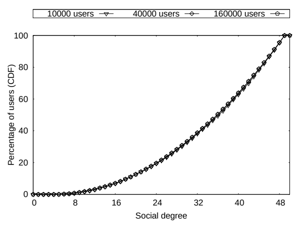
Social degree distributions. Figure 5(b) shows the distribution of the social degree with different number of users. All our experimental results show that the social degree follows a power-law distribution with an alpha value of roughly 2.0.

Scalability. In order to show the scalability of S3G2, we conduct experiments on a cluster of 16 nodes which are connected by 1 Gigabit Internet. Each node is a PC with an Intel i7-2600K, 3.40GHz CPU, 4-core CPU and 16 GB RAM [5]

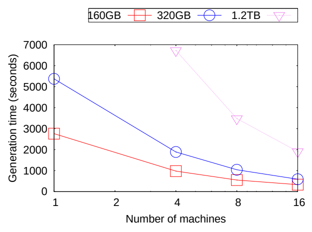
[5] We used the SciLens cluster at CWI: www.scilens.org



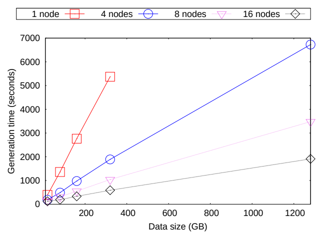
(a) Clustering coefficient



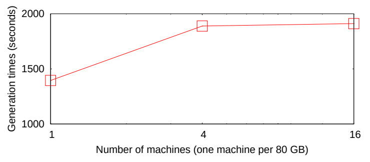
(b) User distribution



(c) Speed-Up Experiments



(d) Scale-Up Experiments



(e) Scale-Out Experiments

Fig. 5. Experimental Evaluation of S3G2

In Figure 5(d), for a specific number of nodes, we increase the data size. These results shows that the generation times is linear according to the size of the data. Interestingly, most of the computational effort is in the first Map function that generates all person nodes and its properties and in the last Reduce that generates the social activities (photos, forum posts). Thus, the effect of the sorting, which due to is $N.log(N)$ would causes super-linear scaling is not visible yet at these data sizes.

Figure 5(c) shows the speed-up of the generator when adding nodes and keeping data size fixed. It shows the MapReduce approach works well, and speed-up is especially good at the larger data sizes.

Figure 5(e) shows the scale-out capability of S3G2 increasing together the dataset size and amount of cluster machines. In these experiments we keep the data generated per machine at 80GB; hence with 4 machines we generate 320GB and with 16 this is 1.2TB. The experimental result shows that performance remains constant at half an hour when scaling out from 4 machines to 16 nodes. This suggests that S3G2 can generate extremely large graphs quickly on a Hadoop cluster with large resources. There is a lot of work studying the characteristics of social networks [11, 7, 12, 15, 5, 1, 9] and also on the generation of random graphs having global properties similar to a social network [14, 3, 4, 10, 6, 8]. However, to the best of our knowledge, there is no generator that creates a synthetic social graph with correlations. The existing graph generators mostly consider the topology and the structures of the generated graph, i.e., global properties, not the individual connections of nodes and their correlations.

One of the first studies to generate social-network-like random graph is [14]. This graph generator with small world properties such as a high clustering coefficient and low path lengths, by connecting a node with its k-nearest-neighbors and then rewiring edges. To satisfy the degree distributions [3] introduced the model of preferential attachment which was subsequently improved by [4]. The main idea of this model is that, for a new vertex, the probability that an edge is created between this vertex to an existing vertex depends on the degree of that vertex. Leskovec et al.[10] proposed a tractable graph that matches several properties of a social graph such as small diameter, heavy-tails in/out degree distribution, heavy-tails eigenvalues and eigenvectors by recursively creating a self-similar graph based on Kronecker[6] multiplication. None of these algorithms considers the correlation of a node attributes in the social graph.

Recently, Bonato et al.[6] studied the link structure of a social network and provided a model that can generate a graph satisfying many social graph properties by considering the location of each graph node by ranking each node. In this model, each node is randomly assigned a unique rank value and has a region of influence according to its rank. The probability that an edge is created between a new node and an existing node depends on the ranking of the existing node. Similar to the approach of using influent regions [8] constructed a set of cliques (i.e., groups) over all the users. For each new node (i.e., a new user), an edge to an existing node is created based on the size of cliques they have in common. These models are approaching the realistic observation that users tend to join and connect with people in a group of same properties such as the same location. However, the simulation of realistic data correlations is quite limited and both do not address the correlations between different attributes of the users.

Additionally, all of the existing models need a large amount of memory for storing either the whole social graph or its adjacency matrix. Leskovec et al. [10] may need to store all stages of their recursive graph. Although Batagelj et al. aimed at providing a efficient space-requirement algorithm, the space-requirement is $O(|V| + |E|)$ where V is the set of vertices and E is the set of edges [4].

4 Conclusion

In this paper, we have proposed S3G2, a novel framework for scalable graph generator that can generate huge graphs having correlations between the graph structure and graph data such as node properties. While current approaches at generating graphs require holding it in RAM, our graph generator can generate the graph with little memory by using a sliding window protocol, and exploit parallelism offered by the MapReduce paradigm. It thus was able to generate 1.2GB of tightly connected, correlated social graph data, on 16 cluster machines using only limited RAM.

In order to address the problem of generating correlated data and structure together, which has not been handled in existing generators, we propose an approach that separates value generation (data dictionaries) and probability distribution, by putting in between a value ranking function that can be parametrized by correlating factors. We also showed a compact implementation of such correlated ranking functions.

Further, we address correlated structure generation by introducing the concept of correlation dimensions. These correlation dimensions allow to generate edges efficiently by relying on multiple sorting passes; which map naturally on MapReduce jobs.

We demonstrate the utility of the S3G2 framework by applying it to the scenario of modeling a social network graph. The experiments show that our generator can easily

[6] http://en.wikipedia.org/wiki/Kronecker_product

generate a graph having important characteristics of a social network and additionally introduce a series of plausible correlations in it.

As future work, is to apply the S3G2 framework to other domains such as telecommunications networks, and possible direction is to write a compiler that automatically generates a MapReduce implementation from a set of correlation specifications. As we believe that correlations between value and structure are an important missing ingredient in today's graph benchmarks, and intend to introduce the Social Intelligence Benchmark (SIB), that uses our data generator, to fill that gap.

References

1. Y. Ahn, S. Han, H. Kwak, S. Moon, and H. Jeong. Analysis of topological characteristics of huge online social networking services. In *Proc. WWW*, 2007.
2. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: A nucleus for a web of open data. *Semantic Web Journal*, pages 722–735, 2007.
3. A. Barabási, R. Albert, and H. Jeong. Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: Statistical Mechanics and its Applications*, 281(1-4):69–77, 2000.
4. V. Batagelj and U. Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.
5. F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *Proc. SIGCOMM*, 2009.
6. A. Bonato, J. Janssen, and P. Prałat. A geometric model for on-line social networks. In *Proc. Conf. on Online Social networks*, 2010.
7. I. de Sola Pool and M. Kochen. Contacts and influence. 1978.
8. I. Foudalis, K. Jain, C. Papadimitriou, and M. Sideri. Modeling social networks through user background and behavior. *Algorithms and Models for the Web Graph*, pages 85–102, 2011.
9. H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proc. WWW*, 2010.
10. J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. *PKDD*, pages 133–145, 2005.
11. S. Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.
12. A. Mislove, M. Marcon, K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Proc. SIGCOMM*, 2007.
13. M. Stillger, G. Lohman, V. Markl, and M. Kandil. Leo-db2's learning optimizer. In *Proc. VLDB*, 2001.
14. D. Watts and S. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393(6684):440–442, 1998.
15. C. Wilson, B. Boe, A. Sala, K. Puttaswamy, and B. Zhao. User interactions in social networks and their implications. In *Proc. European Conference on Computer Systems*, 2009.