MARVIN: distributed reasoning over large-scale Semantic Web data[☆]

Eyal Oren[a], Spyros Kotoulas[a], George Anadiotis[b], Ronny Siebes[a], Annette ten Teije[a], Frank van Harmelen[a]

[a] *Vrije Universiteit Amsterdam, de Boelelaan 1081a, Amsterdam, NL*
[b] *IMC Technologies, Athens, Greece*

Abstract

Many Semantic Web problems are difficult to solve through common divide-and-conquer strategies, since they are hard to partition. We present MARVIN, a parallel and distributed platform for processing large amounts of RDF data, on a network of loosely-coupled peers. We present our *divide-conquer-swap* strategy and show that this model converges towards completeness.

Within this strategy, we address the problem of making distributed reasoning scalable and load-balanced. We present SPEEDDATE, a routing strategy that combines data clustering with random exchanges. The random exchanges ensure load balancing, while the data clustering attempts to maximise efficiency. SPEEDDATE is compared against random and deterministic (DHT-like) approaches, on performance and load-balancing. We simulate parameters such as system size, data distribution, churn rate, and network topology. The results indicate that SPEEDDATE is near-optimally balanced, performs in the same order of magnitude as a DHT-like approach, and has an average throughput per node that scales with $\sqrt{i}$ for i items in the system. We evaluate our overall MARVIN system for performance, scalability, load balancing and efficiency.

1. Introduction

In recent years, large volumes of Semantic Web data have become available, to the extent that the data is quickly outgrowing the capacity of storage systems and reasoning engines. Through the "linking open data" initiative, and through crawling and indexing infrastructures [14], datasets with millions or billions of triples are now readily available. These datasets contain RDF triples and many RDFS and OWL statements with implicit semantics [6].

Since the datasets involved are typically very large, efficient techniques are needed for scalable execution of analysis jobs over these datasets. Traditionally, scaling computation through a divide-and-conquer strategy has been successful in a wide range of data analysis settings. Dedicated techniques have been developed for analysis of Web-scale data through a divide-and-conquer strategy, such as MapReduce [5].

In contrast to other analysis tasks concerning Web data, it is not clear how to solve many Semantic Web problems through divide-and-conquer, since it is hard to split the problem into independent partitions. However, to process, analyse, and interpret such datasets collected from the Web, infrastructure is needed that can scale to these sizes, and can exploit the semantics in these datasets.

To illustrate this problem we will focus on a common and typical problem: computing the deductive closure of these datasets through logical reasoning. Recent benchmarks [2] show that current RDF stores can barely scale to the current volumes of data, even without this kind of logical reasoning.

To deal with massive volumes of Semantic Web data, we aim at building RDF engines that offer *massively scalable* reasoning. In our opinion, such scalability can be achieved by combining the following approaches:

- using *parallel hardware* which runs *distributed algorithms* that exploit hardware varying from tens to many hundreds of processors.
- designing *anytime algorithms* that produce

[☆]This work is supported by the European Commission under the LarKC project (FP7-215535).

sound results where the degree of completeness increases over time.

- our novel *divide-conquer-swap* strategy, which extends the traditional approach of divide-and-conquer with an iterative procedure whose result converges towards completeness over time.

We have implemented our approach in MARVIN[1] (MAssive RDF Versatile Inference Network), a parallel and distributed platform for processing large amounts of RDF data. MARVIN consists of a network of loosely-coupled machines using a peer-to-peer model and does not require splitting the problem into independent subparts. MARVIN is based on the approach of *divide-conquer-swap*: peers autonomously partition the problem in some manner, each operate on some subproblem to find partial solutions, and then re-partition their part and swap it with another peer; all peers keep re-partitioning, solving, and swapping to find all solutions.

In this paper, we present our general approach called *divide-conquer-swap* and show that the model is sound, converges, and reaches completeness eventually. We then focus on efficient computation: we introduce an *exit-door policy* for handling produced duplicates, and introduce our SPEEDDATE approach that combines efficient deductions while balancing the load equally amongst computation nodes. We report on simulation results with our SPEEDDATE approach and provide experimental results using MARVIN on RDF graphs.

2. Related work

In general, logical reasoning allows us to expand RDF graphs with implicit information. For example, by combining `(Amsterdam locatedIn Netherlands)` and `(Netherlands locatedIn Europe)`, we can derive `(Amsterdam locatedIn Europe)`. Given the size of the data, scalable reasoning is a major challenge [16, 8]

2.1. Distributed reasoning

For distributed reasoning, triples that share common elements ("Netherlands" in the example above) should be co-located at the same machine and combined into additional triples: triples should "meet" each other in one of the distributed peers. The challenge in this scenario lies in *assigning rendezvous points for triples*.

Our baseline approach uses random rendezvous points: triples are sent around randomly until they happen to produce some deduction. This random approach is load-balanced (all nodes hold the same number of triples at any point in time) but inefficient and not scalable: with a growing number of nodes, triples have less chance to meet.

Several distributed reasoning techniques have been proposed based on deterministic rendezvous points using a distributed hashtable (DHT) [12]. Here, each triple is sent to three rendezvous peers (one for each of its terms: subject, predicate, and object), which ensures that triples with common terms will be co-located [4]. However, given the size and distribution of the data (many billions of triples, with terms occurring according to a power-law [14]) the rendezvous peers will suffer from highly unbalanced load distributions.

Note that standard techniques for load-balancing [11] will not work in our situation, since: (a) some popular URIs appear in very many triples, thus we have more items *sharing one key* than can fit in a single node so replication and caching will not help, and (b) we need all items with the same key to meet each other, so sub-dividing the keyspace over multiple responsible nodes will also not help.

Fang *et al.* [7] have an iterative forward-chaining procedure similar to ours but do not address load-balancing issues. Kaoudi *et al.* [10] propose a backward-chaining algorithm which seems promising, but no conclusions can be drawn given the small dataset (10^4 triples) and atypical evaluation queries employed. Battré *et al.* [1] perform limited reasoning over locally stored triples and introduce a policy to deal with load-balancing issues, but only compute a fraction of the complete closure.

2.2. Federated reasoning

Schenk and Staab [17] introduce *networked graphs*, allowing transparent data integration and querying of remote RDF endpoints, with an initial evaluation over small datasets. DARQ [15] uses a similar approach but adds query optimisation based on endpoint descriptions and statistics, which improves query performance. Neither approach addresses inferencing.

[1] named after Marvin, the paranoid android from the Hitchhiker's Guide to the Galaxy. Marvin has "a brain the size of a planet" which he can seldomly use: the true horror of Marvin's existence is that no task would occupy even the tiniest fraction of his vast intellect.

Serafini and Tamilin [19] perform distributed description logic reasoning; the system relies on manually created ontology mappings, which is quite a limiting assumption, and its performance is not evaluated. Schlicht and Stuckenschmidt [18] distribute the reasoning rules instead of the data: each node is only responsible for performing a specific part of the reasoning process. Although efficient by preventing duplicate work, the node with the most limited computational resources in this setup becomes an immediate bottleneck and a single-point-of-failure, since all data has to pass all nodes for the system to function properly.

Hogan *et al.* [9] use a modified ruleset which allows reasoning using a single pass over the data. This approach is orthogonal to ours and could be integrated into MARVIN, since we treat each single reasoner as a black box.

3. Our approach: Divide-conquer-swap

MARVIN operates using the infinite main loop shown in Algorithm 1, which is run on a set of compute nodes. These nodes have the same functionality and we will also refer to them as peers. In this loop, nodes grab some partition of the data, compute the closure on their partition, and then repartition and swap with another node to find more inferences. The nodes keep re-partitioning, solving, and swapping to find all solutions.

In this paper, we focus on step *swap*. Initial partitioning is accomplished by reading some random data from disk, subsequent partitioning is actually dictated by step *swap*. For step *conquer*, we use an off-the-shelf reasoner to compute the closure, which we consider a black box. Note that our model only supports monotonic logics.

This "divide-conquer-swap" approach raises two questions:

- Does the approach converge and do we ever reach logical completeness?
- Is the approach efficient and scalable: what is the base performance of this model and how much performance gain is given by additional compute resources?

We will answer these questions in the next two sections. First, in Section 4, we show that the model does indeed reach completeness eventually. Next, in Section 5 we show that the distributed computation generates many duplicate triples which need to be detected and removed for efficient performance. Finally, in Section 6 we show that this model is inefficient if nodes exchange data randomly and we show how to strongly improve efficiency without sacrificing load-balance through our SPEEDDATE technique.

Algorithm 1 Divide-conquer-swap

```
repeat
  divide: read a partition of data
  conquer: compute closure
  swap: repartition and exchange with peers
until forever
```

4. Eventual completeness

In this section we will provide a qualitative model to study the completeness of MARVIN. Assuming a sound external procedure in the "conquer" step, overall soundness is evident through inspection of the basic loop, and we will not discuss it further.

The interesting question is not only *whether* MARVIN is complete: we want to know *to what extent* it is complete, and how this completeness *evolves over time*. For such questions, tools from logic do not suffice since they treat completeness as a binary property, do not analyse the degree of completeness and do not provide any progressive notion of the inference process. Instead, an elementary statistical approach yields more insight.

Let C^* denote the deductive closure of the input data: all triples that can be derived from the input data (we consider only the derived triples, to obtain the complete closure, the original data should be added). Given MARVIN's soundness, we can consider each inference as a "draw" from this closure C^*. Since MARVIN derives its conclusions gradually over time, we can regard MARVIN as performing a series of repeated draws from C^* over time. The repeated draws from C^* may yield triples that have been drawn before: peers could re-derive duplicate conclusions that had been previously derived by others. Still, by drawing at each timepoint t a subset $C(t)$ from C^*, we gradually obtain more and more elements from C^*.

In this light, our completeness question can be rephrased as follows: how does the union of all sets $C(t)$ grow with t? Will $\cup_t C(t) = C^*$ for some value of t?

At what rate will this convergence happen? Elementary statistics tells us that if we draw t times

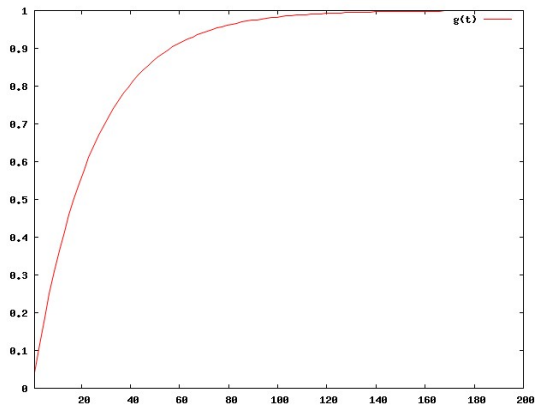


Figure 1: Predicted rate of unique triples produced

a set of k elements from a set of size N, the number of distinct drawn elements is expected to be $N \times (1 - (1 - k/N)^t)$. Of course, this is the *expected* number of distinct drawn elements after t iterations, since the number of drawn duplicates is governed by chance, but the "most likely" (expected) number of distinct elements after t iterations is $N \times (1 - (1 - k/N)^t)$, and in fact the variance of this expectation is very low when k is small compared to N.

In our case, $N = |C^*|$, the size of the full closure, and $k = |C(t)|$, the number of triples jointly derived by all nodes at time t, so that the expected completeness $\gamma(t)$ after t iterations is:

$$\gamma(t) = (1 - (1 - \frac{|C(t)|}{|C^*|})^t)$$

Notice that the boundary conditions on $\gamma(t)$ are reasonable: at $t = 0$, when no inference has been done, we have maximal incompleteness ($\gamma(0) = 0$); for trivial problems where the peers can compute the full closure in a single step (i.e. $|C(1)| = |C^*|$), we have immediate full completeness ($\gamma(1) = 1$); and in general if the peers are more efficient (i.e. they compute a larger slice of the closure after each iteration), then $|C(t)|/|C^*|$ is closer to 1, and $\gamma(t)$ converges faster to 1, as expected. The graph of unique triples produced over time, as predicted by this model, is shown in figure 1. The predicted completeness rate fits the curves that we find in experimental settings, shown in the next section.

This completeness result is quite robust: in many realistic situations, at each timepoint the joint nodes will only compute a small fraction of the full closure ($C(t) \ll |C^*|$), making $\gamma(t)$ a reliable expectation with only small variance. Furthermore, completeness still holds when $|C(t)|$ decreases over t, which would correspond to the peers becoming less efficient over time, through for example network congestion or increased redundancy between repeated computations.

Our analytical evaluation shows that *reasoning in* MARVIN *converges and reaches completeness eventually.* Still, convergence time depends on system parameters such as the size of internal buffers, the routing policy, and the exit policy. In the next section, we report on empirical evaluations to understand the influence of these parameters.

5. Duplicate detection and removal

Since our aim is to minimise the time spent for deduction of the closure, we should spend most of the time computing new facts instead of recomputing known facts. Duplicate triples can be generated for several reasons, including redundancy in the initial dataset, sending identical triples to several peers or deriving the same conclusions from different premises.

In reasonable quantities, duplicate triples may be useful: they may participate, in parallel, in different deductions. In excess, however, they pose a major overhead: they cost time to produce and process and they occupy memory and bandwidth. Therefore, we typically want to limit the number of duplicate triples in the system. On the other hand, we want to keep at least one copy of each triples, to guarantee eventual completeness. Therefore, we mark one copy of each triple as a *master copy* (using a bit flag) which will never be deleted.

To remove duplicates from the system, they need to be detected. However, given the size of the data, peers cannot keep a list of *all* previously seen triples in memory as this approach would not scale: for example, even using an optimal data structure such as a Bloom filter [3] with only 99.9% confidence, storing the existence of 1 billion triples would occupy some 1.8GB of memory on each peer.

We tackle this issue by distributing the duplicate detection effort, implementing a *one-exit door policy*: we assign the responsibility to detect each triple's uniqueness to a single peer, using a uniform hash function: $exit\_door(t) = hash(t) \bmod N$, where t is a triple and N is the number of nodes.

The exit door uses a Bloom filter to detect previously encountered triples: it marks the first copy of each triple as the *master copy*, and removes all other subsequent copies.

For large numbers of nodes however, the *one-exit door policy* becomes less efficient since the probability of a triple randomly appearing at its exit door is $\frac{1}{N}$ for N number of nodes. Therefore, we have an additional and configurable *sub-exit door policy*, where some k peers are responsible for explicitly routing some triples to an exit door, instead of waiting until the triples arrive at the designated exit door randomly. A final optimisation that we call the *dynamic sub-exit door policy* makes k dependent on the number of triples in each local output buffer - raising k when the system is loaded and lowering it when the system is underutilized. This mechanism effectively works as a pressure valve, relieving the system when pressure gets too high. This policy is implemented with two thresholds: if the number of triples in the *output pool* exceeds t_{upper} then we set $k = N$, if it is below t_{lower} then we set $k = 0$. Section 8.3 presents evaluation results for this technique.

6. Efficient deductions

In this section we address the problem of finding inferences efficiently in a distributed system. As discussed earlier, to draw some conclusions, all triples involved in the deduction need to be co-located on the same machines.

We showed in Section 4 that randomly exchanging triples ensures that each combination of triples is co-located in a machine at some point in time. However, random exchanges are inefficient since also irrelevant triples are co-located at one machine. With increasing number of nodes the chance for relevant triples to be co-located decreases strongly.

Assigning deterministic rendezvous points to each triple (sending it to some specific peer) as is done in DHT-based approaches is more efficient, but suffers from load-balancing problems, as mentioned in Section 2. For example, in a scenario where some popular term (e.g. rdf:type) appear 10% of the time, a single node will be called to store at least 10% of all triples in the system (remember that triples in deterministic partitioning approaches are constantly collocated). This will obviously not scale beyond a handful of nodes. Thus, deterministic partitioning in not feasible for large scale data. Instead, in this section we propose an improved strategy for exchanging triples called SpeedDate which does not disturb load-balance and is more efficient that random exchanges.

6.1. Problem

Let us describe our situation in more abstract terms: we have a set of items (triples) with particular keys (URIs), where multiple items may have the same key (different triples share some URI). We also have a set of nodes, each of them being able to store a number of items. The nodes do not have any special rights over items: any node can store any item.

Our goal is to have as many items encounter or meet their "buddies" (other items with the same key). By encounter we mean: the items must at some point in time be located in the same node. We do not care whether keys are always at the same node, we just want, over time, that items meet others with the same key. Furthermore, encounters have *at least once* semantics, we are not interested in how many times an item encounters another item with the same key, as long as they meet at least once.

In other words, the items want to speed-date each other. There are many rooms (nodes) that can hold only a finite number of items at a time. Items want to meet as many other items with the same interest (key) as possible yet minimise the time spent travelling.

The convergence (or completeness) criterion for our system is that all items with the same key have encountered each other. In our case, this means being collocated on the same node at some point: $\forall k \in keys, \forall i,j \in data_k : \exists t,n : located(i,n,t) \wedge located(j,n,t)$.

Note that some logics (such as some parts of OWL) require *three* or more items (triples) with the same key to meet simultaneously. Although our approach should improve performance in these logics as well, we do not consider them in our evaluation.

6.2. Intuition

Our approach combines the balanced load of random exchanges and the efficiency of deterministic rendezvous nodes. Instead of a deterministic rendezvous location for each key, data is moved around randomly, but with a (strong) bias towards some nodes. This bias is determined using item keys and node identifiers (similar to distributed hashtables). The bias improves clustering and encounter probability, while the random exchanges ensure that

items are distributed evenly among the nodes. Let us illustrate the algorithm with an example.

node	initial items	eventual items
A	p, s, s, t	p, q, r
B	r, u, v	s, s, t, u
C	q, s, t, t	s, t, t, v

Table 1: Data distribution over nodes. The lower case letters represent items with a given key. For example, there are three items with key t.

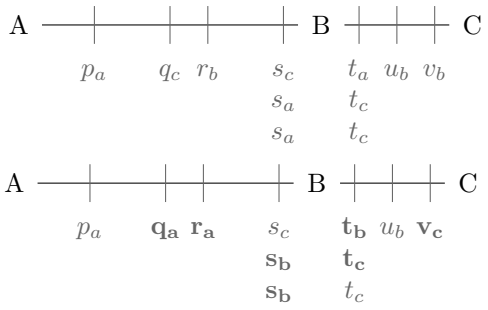


Figure 2: Position of keys and location of items in nodes. Lower case letters represent items with a given key and the subscript represents the node the items is located in. The top line shows the initial state from Table 1 and the bottom line shows the eventual state. Bold font signifies items that have been moved.

Example 1. *We have three nodes, A, B and C, and a set of items, with an uneven distribution of keys. Initially, items are distributed randomly amongst nodes, as shown in Table 1. We can position node identifiers and item keys on a space that wraps-around (e.g. using a hash function and modulo arithmetic), as shown in Figure 2. In this space, the position of A, B, and C signifies their node IDs. Items are positioned according to their key, not according to their current location: their location is indicated graphically with subscripts, e.g. p_a means item p is located at node A.*

In our algorithm, each node autonomously selects some other node (the selection mechanism will be explained later), and exchanges data. For example, A will ask B for one item. When asked by A, B will return the item it has whose key is closest to A on the joint space. In this case item, B will return r_b (the items p and q, which are closer to A, are located at nodes A and C). In return, B will receive an item from A that is closest to B, in this case one s_a.

After exchanging several data items, the data distribution will start clustering, as we can see in the lower part of Figure 2. Since nodes keep asking each other for data, the distribution never converges but keeps oscillating around a clustered state. Eventually (shown in the right column of Table 1), node A will mostly contain the items p, q and r, B will mostly contain s, and C will contain t, u, and v.

We can see that through biased random exchanges, a responsibility area emerged for each node, and was adjusted to ensure a balanced distribution of the initial data. These responsibility areas are an emergent property of the algorithm and are not explicitly defined.

In contrast, a DHT-like approach would use rigid responsibility areas (based on item key and node ID) and thus assign most items to node B, leading to load-balancing problems.

We will now describe our SpeedDate rendezvous algorithm, which was used in the example above. Since our algorithm is a mixture between the efficient approach of deterministic rendezvous points and the balanced approach of randomised rendezvous points, we first briefly explain these.

6.3. Reference approaches

A deterministic rendezvous algorithm uses some deterministic mechanism to map item keys to rendezvous points maintained by rendezvous nodes. The convergence criterion would be met quickly and efficiently, since all items with the same key would be sent to a given rendezvous, where they would meet each other.

It is straightforward to implement such a function with a DHT or with a broker. The main drawback of this approach is that it suffers from load balancing issues: What if there are more items for a given key than any single host can hold?

Algorithm 2 Random rendezvous

```
I: set of my items
P: set of IDs of my neighbouring nodes
procedure MAIN
    while true do
        i ← pick_uniform_random(I)
        p ← pick_uniform_random(P)
        send(i, p)
    end while
end procedure
```

Nodes may also exchange items randomly, shown in Algorithm 2. At every time step, they select a random subset of their items and send it to a randomly selected node. Both nodes and items are picked according to a uniform random probability function, i.e. all nodes and items have equal probability of being picked. Here, we assume that the network is fully connected, i.e. nodes can send messages to any other node. Section 7 will also evaluate the performance of random exchanges in partially connected networks.

Since the set of buddies encountered after each exchange is random, the method meets our eventual consistency criterion. The random approach does not suffer from load-balancing problems, since all nodes have an equal probability of receiving an item.

6.4. SpeedDate

The SpeedDate algorithm is a hybrid between a random exchange and a deterministic rendezvous. Nodes are assigned a random ID in the same space with the keys using a uniform distribution. Unlike DHTs, this ID does not need to be unique. Furthermore, every node has a set of neighbours, of which it knows the ID. Again, we can first assume that the network is fully connected; Section 7 will evaluate the performance in partially connected networks.

Similar to the random approach, nodes ask their neighbours for items. The uniqueness of SpeedDate lies in:

- Instead of returning random items, nodes return the items that are closest to the ID of the asker. This enables data clustering, thus increasing performance.
- Instead of picking random neighbours for exchange, nodes prefer neighbours with IDs close to their own. This ensures that data remains clustered, and improves the likelihood of data encounters.
- Instead of only returning optimal data items, nodes always return some fixed number of items when asked. If nodes do not have any items with keys close to the key of the asker, they will still return the best they have. This ensures that the system is load balanced.

The SpeedDate approach is shown in Algorithm 3. Each node repeats the following eternally:

Algorithm 3 SpeedDate rendezvous

```
I: set of my items
P: set of IDs of my neighbouring nodes
Self: my own node ID
procedure MAIN
    while true do
        p ← pick_gaussian_random(P, Self)
        i ← pick_item_for(p)
        send(i, p)
    end while
end procedure
procedure PICK_ITEM_FOR(p)
    repeat
        if ∃e ∈ I : key(e) = p then
            return e ▷ return item with given key
        else                    ▷ else, return closest item
            return arg(min_{e∈I}{|key(e) − p|})
        end if
    until forever
end procedure
```

1. Pick a neighbour using a Gaussian (normal) distribution on the node ID. Each node maintains a Gaussian distribution, with mean μ on his own ID, and some given variance σ. When run with low variance, nodes will mostly select their close neighbours for exchange (whose ID is close to their own). With high variance, nodes will exchange with nodes selected more uniformly across all other nodes.

2. Pick the items with keys closest to the ID of the selected neighbour. Each node contains a set of items, and each item has one key. Having selected one neighbour for exchange, nodes will select the data item whose key is closest to the ID of that selected neighbour.

   Note that this selection does not necessarily entail iterating over all items: nodes could preprocess their items into buckets or use sorted data structures such as trees.

3. Send those items and receive some of the neighbour's items in return. Nodes exchange items symmetrically, which ensures load balancing (all nodes have the same amount of data). The neighbour selects items according to the same principle as explained above, thus returning items whose keys are closest to the sender's ID.

6.5. Visual comparison of data clustering

The three algorithms are visually compared in Figures 4–5. The images shown are snapshots of the data distribution for a small example of 100 nodes, 100 keys and 10,000 items. They are taken after 33, 66 and 100 time units respectively.

The vertical dimension shows the nodes, the horizontal dimension shows the keys. Brightness represents the number of items with the given key at the given node. Key popularity is unevenly (linearly) distributed: not all keys appear in the same number of items. In the images, keys are sorted according to increasing popularity: keys at the top of the image are more popular than keys at the bottom.

In the random approach, shown in Figure 3, items with the same key are dispersed throughout the nodes, and the concentration of items in one node is never large enough to produce a bright pixel (the picture looks almost completely dark, but is not). Although this uniform distribution is not good for performance, all nodes have a similar load.

In the DHT-based approach, shown in Figure 4, items quickly move to the nodes responsible for them (optimal distribution already reached at the second snapshot). This results in a strongly partitioned data distribution, all items with the same key are located on the same node, producing bright white pixels. However, as we will show in Section 7, for realistic data distribution the partitioning results in poor load balancing (not obvious in the figure).

Figure 5 shows snapshots of the data distribution in SpeedDate. Over time, all items with some key gather in the neighbourhood of one node, increasing their encounter rate. Nodes share responsibility for a key, resulting in better load balancing. Furthermore, notice that the horizontal lines for the more popular key (i.e. the keys at the bottom of the image) are longer than those in the top, meaning that more nodes share responsibility for keys with more items.

7. Simulation results

In this section, we focus on the performance of SpeedDate compared to the random and DHT-based approaches in a simulated environment. We also explore the parameter settings and sensitivity of SpeedDate. Next, in Section 8 we focus on the performance of the complete Marvin system in experiments on real RDF datasets.

7.1. Simulation setup

For the SpeedDate experiments we have used a purpose-built simulator, which runs on Java 1.6. The simulation proved to be computationally demanding, so we have used a quad-processor 2.3GHz server-class machine with 32GB of main memory. We use a simulation clock. Every clock cycle, nodes may send a fixed number of messages.

7.1.1. External parameters

Experiments were performed using an artificial dataset, under the following conditions:

Nodes We have n nodes in the system; all nodes are considered to be homogeneous, with the same functionality and specifications. Nodes have a limited capacity for items which cannot be exceeded. Nodes have a limited bandwidth of messages per clock cycle.

Items There is a total of i items in the system. Each item has one (non-unique) key. There is a total of k unique keys in the system.

Key distribution The number of items with a given key follow one of the following distributions:

- **Uniform** All keys appear in the same number of items: $items(key) = c$, with $c = \frac{\#items}{\#keys}$.
- **Linear** The number of items per key are linearly distributed: $items(key_r) = c * r$ where r is the popularity rank of the key, for some constant c.
- **Zipf** The Zipf distribution for n outcomes is defined as $Zipf_n(r) = 1/(r * \sum_{j=1}^{n}(1/j))$, where r is the rank of the outcome (ordered by probability), and is a realistic distribution for Web documents and Semantic Web data [14]. Note that the probability density function is very steep: for 5000 outcomes, the top-3 has a total probability of more than 20%.

Churn We assume a fail-silent model for nodes. I.e. in every cycle, each node has a fixed probability *churn* of being offline. This means that no messages are sent or received by that node. Message senders do not know whether nodes are online, so there is a chance that messages are lost. In this case, we assume that the

Figure 3: Simulated random data distribution (at 33, 66, and 100 time units respectively)

Figure 4: Simulated data distribution in DHT (at 33, 66, and 100 time units respectively)

Figure 5: Simulated data distribution in SpeedDate (at 33, 66, and 100 time units respectively)

sender can detect that the receiver did not receive the message and retain the items that would be sent in its own data. In our setting, churn is the inverse of availability.

7.1.2. Internal parameters

Our algorithm has the following parameters:

Connectivity Each node keeps a connection to a number of other nodes, ranging from $1 \dots n$, the latter being a fully connected network. The higher this number, the more maintenance is required on the overlay network.

Topology When the network is not fully connected, neighbours are selected either randomly or according to their ID. We will refer to the former topology as "random" and to the latter as "proximity". In the proximity topology, nodes prefer neighbours with IDs similar to their own, resembling the topology of the Chord DHT [20].

Neighbour bias The variance σ represents the bias of nodes to select neighbours with similar IDs to their own for exchanges. For readability, we show the inverse: $1/\sigma$. A high $1/\sigma$ means highly selective bias: nodes mostly select peers close to them.

7.1.3. Evaluation criteria

We use the following criteria to evaluate our algorithm:

Recall per key We measure recall of encounters as the ratio of actual encounters vs. possible encounters for each key:

$$recall = \frac{1}{k}\sum_{k}\left(\frac{actual\ encounters\ for\ key\ k}{possible\ encounters\ for\ key\ k}\right)$$

Recall per item We also calculate recall per item to indicate system behaviour towards more-popular keys (which would normally overpower the normal, per key, recall):

$$w.\,recall = \frac{\sum_{k}(actual\ encounters\ for\ key\ k)}{\sum_{k}(possible\ encounters\ for\ key\ k)}$$

Load balance We measure, after each clock cycle, the data load in each node and the number of messages sent and received. We evaluate data load balancing in terms of maximum and standard deviation of $load_d$ and $load_m$ across all nodes.

Unless otherwise noted, all experiments use the default parameters from Table 2 which were obtained through a series of exploratory experiments.

parameter	**value**
nr. items	100.000
nr. keys	5.000
nr. nodes	500
nr. neighbours	$\log_2(nr.\ nodes)$
key distribution	Zipf
topology	proximity
selection bias σ	0.25
node capacity	200
node bandwidth	1

Table 2: Default parameters

7.2. Recall and load balance

Figure 6 compares the performance of SpeedDate with the random and deterministic rendezvous approaches. The figure shows, on the left, per-key rendezvous recall over time, and on the right, per-item recall over time. We can see that in both cases, the deterministic approach outperforms SpeedDate and that SpeedDate clearly outperforms the random approach. Note that the per-key recall of SpeedDate is higher than the per-item recall, which means that we favour rare items over very popular items.

Table 3 compares the load-balancing properties of these approaches, showing the standard deviation and maximum number of items stored and messages received per node. The presented values are the highest across all cycles in one simulation.

We can see that the random approach is clearly load-balanced, both in terms of data load and messages received (low deviation, low maxima). The deterministic approach suffers from severe imbalance, one node stores over 11k items while the average is 200. In reality, this would pose serious scalability issues. The same holds for message load. The load balance in SpeedDate is comparable to random, with low deviation and low maxima.

approach	σ_{data}	max	σ_{msgs}	max
random	13.18	242	10.48	244
deterministic	620.99	11021	316.71	5685
SpeedDate	14.08	242	12.52	281

Table 3: Load-balancing (lower is better)

In short, we can see that SpeedDate combines the efficiency of a deterministic approach with the load balance of a random approach.

7.3. Topologies

We evaluate SpeedDate under various overlay topologies and number of neighbours per node. Furthermore, we want to find the optimal values for neighbour bias σ. Table 4 shows these results which are also summarised in Figure 7(a). From these we can conclude the following:

Topologies Proximity and full clearly outperform the random topology. It is not clear whether proximity outperforms full.

Number of neighbours The proximity topology is not very sensitive to the number of neighbours per node. We can see that it has similar performance for 3 neighbours per node up to 500 neighbours per node (fully connected network). The fact that only 3 neighbours are required to achieve acceptable performance is very encouraging given the maintenance cost for neighbours.

Sensitivity to the value of sigma The proximity topology is not very sensitive to the setting of neighbour bias σ. We can see that the system produces similar results for a $1/\sigma$ ranging from 0.1 to 5. Note that the fully connected network performs best when we use a very steep distribution to select neighbours. The proximity topology performs well even if we use an almost uniformly random distribution. This is attributed to the fact that the set of neighbours already contains the nodes with the closest ID to the node.

Note that for very low sigmas (high $1/\sigma$), the proximity topology performs very badly, which is attributed to the symmetry of the topology (because nodes select their left and right neighbours using a symmetric function). For a very high selection bias, nodes will only exchange items pair-wise with one neighbour, which has a negative effect on recall.

7.4. Data distribution

Figure 7(b) shows the recall rate over time for different data distributions. Initially, the algorithm performs slightly better on data following a Zipfian distribution, which we attribute to it favouring rare keys (which appear more frequently in the Zipfian distribution). Overall, the algorithm performs slightly better on uniform and linear distributions, which we attribute to the large amounts of potential encounters that must be found for popular data.

7.5. Churn

Figure 8 shows how SpeedDate performs under node failures. We use the default settings except for churn, which ranges from 0% to 90%. Figure 8(a) shows recall over time, Figure 8(b) shows time needed to reach 90% recall.

For every node that is unavailable we lose the messages that it would send plus the messages that it would receive. This amounts to a loss in messages equal to the square of the loss in availability: for example, for 50% availability, only half of the nodes will send messages, out of which half will be lost (since the receivers are down), amounting to 25% of the messages that would have been sent with 100% availability.

In general, the ratio of the messages delivered in a system with availability x to a system with 100% availability is x^2 (x are sent, of which x arrive); since churn α is defined as $1 - x$, messages delivered as function of churn is $1 - (2\alpha - \alpha^2)$. In a perfect system without recovery mechanisms, loss in messages would exactly equal loss in recall (all undelivered messages are lost encounter opportunities).

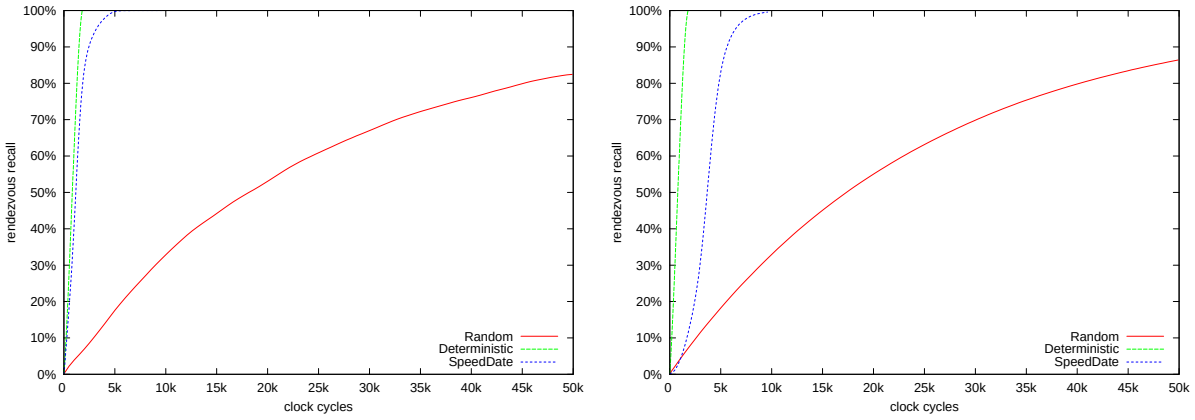
Indeed, our simulation results indicate that the performance loss in SpeedDate, in the presence of peer failures, follows this model. Figure 8(b) compares our simulated results against this prediction.

The construction and maintenance of the overlay network is orthogonal to our approach; thus coping with permanently failed connections is outside the scope of this paper.

7.6. Scalability

In evaluating the scalability of SpeedDate, we have made the following assumptions: (a) node capacity stays constant, thus as the number of items increases, so does the number of nodes; and (b) as the number of items increases, so does the number of keys. Note that Zipf is a scale-free distribution; thus, with an increasing number of items, item popularity still follows the same distribution.

Figure 9 summarises our results, showing recall over time on the left and time needed to reach 90% recall on the right.

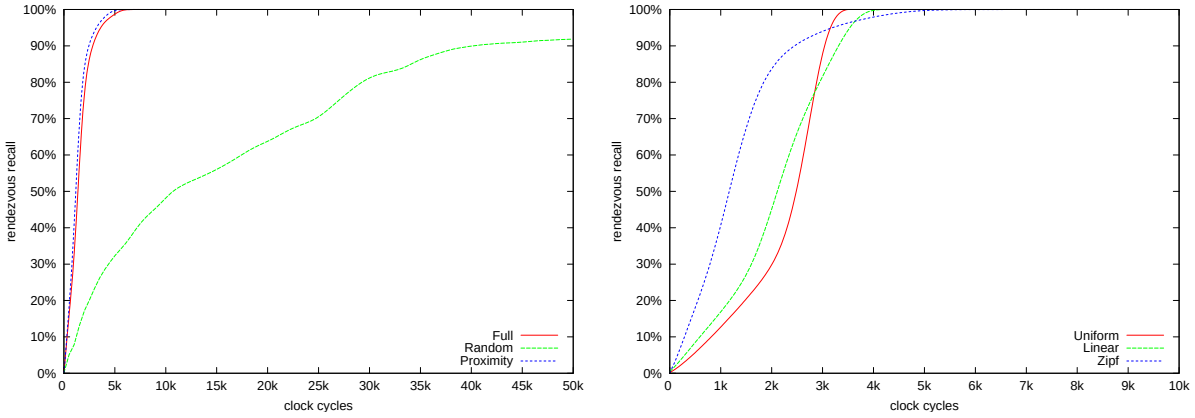


(a) (b)

Figure 6: Comparison of approaches, using (a) default recall and (b) per-item recall

1/σ / nr. neigh.	0.1	0.2	0.3	0.5	0.8	1	2	3	4	5	6	7	8	9	10	25	50	100	150	200
FULL																				
	346	361	331	351	325	393	319	289	163	133	131	98	98	75	77	22	23	19	34	55
PROXIMITY																				
252	19	19	20	22	18	27	18	17	18	22	33	49	83	115	404	500	500	500	500	500
128	23	19	19	16	20	20	19	16	20	22	32	38	65	153	490	500	500	500	500	500
66	24	26	18	21	21	17	23	24	19	27	34	44	71	118	289	500	500	500	500	500
45	21	20	20	25	19	17	23	18	20	19	28	53	75	176	334	500	500	500	500	500
35	19	21	19	18	22	18	22	19	18	24	43	57	88	111	173	500	500	500	500	500
16	25	21	25	22	19	22	25	23	19	20	35	52	108	132	280	500	500	500	500	500
14	22	24	19	31	19	17	24	21	18	22	33	52	86	101	187	500	500	500	500	500
12	20	21	32	27	17	22	21	19	22	29	31	45	65	146	192	500	500	500	500	500
11	33	26	23	28	23	21	19	17	23	29	39	40	48	72	131	500	500	500	500	500
10	23	24	23	26	19	27	27	21	17	37	23	35	70	93	192	500	500	500	500	500
9	20	20	22	25	27	24	22	21	22	22	42	51	83	177	344	500	500	500	500	500
6	25	28	18	32	30	20	19	30	42	27	34	45	70	139	221	500	500	500	500	500
5	20	24	23	20	19	26	19	20	17	24	29	60	46	152	271	500	500	500	500	500
4	22	28	26	29	25	29	23	27	27	30	42	58	61	115	77	500	500	500	500	500
3	22	21	27	25	23	23	24	19	21	27	27	47	73	107	180	500	500	500	500	500
2	102	102	82	100	87	89	68	85	92	127	131	254	378	500	500	500	500	500	500	500
1	93	124	101	87	72	80	84	100	88	130	87	69	109	73	76	129	99	88	117	102
RANDOM																				
252	409	414	369	421	397	361	323	280	219	231	151	102	108	89	90	51	43	52	87	180
128	406	300	360	304	395	359	451	263	242	254	158	194	108	99	85	348	500	500	500	500
66	377	318	371	500	378	396	457	440	224	242	210	148	172	161	124	500	500	500	500	500
45	424	390	470	479	429	444	379	422	306	207	193	231	321	170	185	500	500	500	500	500
35	410	398	500	489	350	404	500	348	295	338	234	172	264	207	325	500	500	500	500	500
16	404	411	493	449	375	401	500	358	424	388	378	500	500	500	500	500	500	500	500	500
14	380	454	363	418	448	400	500	379	449	401	352	500	500	500	500	500	500	500	500	500
12	348	423	341	356	397	387	472	321	358	427	500	500	500	500	500	500	500	500	500	500
11	499	407	474	500	331	463	500	434	474	355	500	500	500	500	500	500	500	500	500	500
10	460	500	494	426	500	500	500	473	399	430	500	500	500	500	500	500	500	500	500	500
9	396	478	500	500	417	418	418	500	500	359	500	500	500	500	500	500	500	500	500	500
6	500	500	500	433	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500

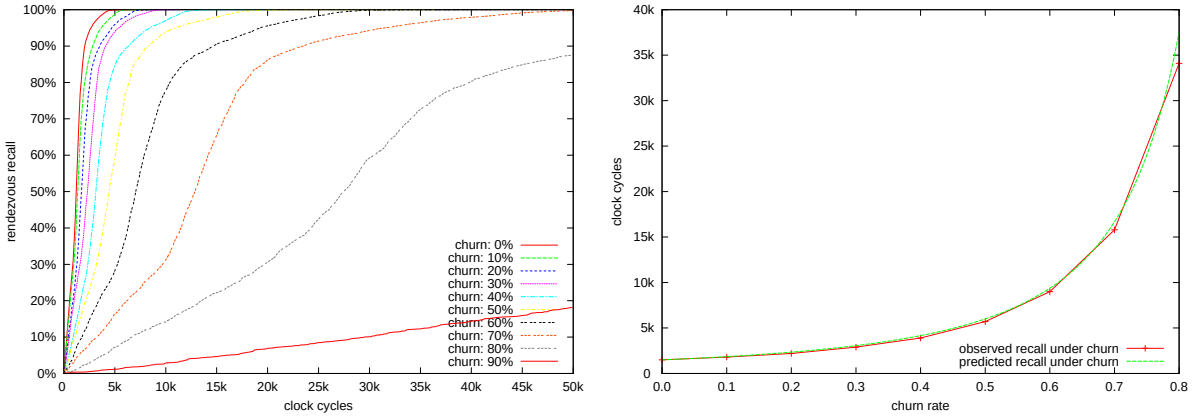
Table 4: cycles (×100) to reach 90% recall as a function of topology, number of neighbours and σ. The horizontal axis shows $1/\sigma$ to enhance readability. Simulations were limited to a maximum of 500 cycles. Lower is better.



(a) recall in different topologies

(b) recall for different distributions

Figure 7: Recall rate for different topologies and data distributions



(a) recall over time

(b) time to reach 90% recall

Figure 8: Recall with increasing churn

Note that the different settings in Figure 9(a) do not show a linearly increasing number of items. In fact, we notice that for 20× the number of items, we have 5× worse performance, for 3000× more items, we have 10× performance loss, which indicates good scalability properties.

As indicated in Figure 9(b), the performance of SpeedDate as a function of growing system size seems to follow a curve of $\sqrt{i}$, where i is the total number of items in the system.

7.7. Anytime behaviour

As we can see in the Figures 6– 9, recall does not increase linearly with time. Instead, we can observe three phases in the algorithm: first the clustering phase in which items are being moved towards their target neighbourhood, then the exploitation phase in which items are exchanged within clusters and the encounter rate is high, and a final phase where the remaining items slowly fight all odds.

SpeedDate has good anytime behaviour: in all settings, 80% of the results are produced in the initial and exploitation phases, which take around one-third of total running time.

8. Experimental results

The SpeedDate algorithm is focused on maximising the number of triples that meet with their "buddies", to allow nodes to produce inferences. The overall Marvin system implements the SpeedDate routing strategy, and performs the actual reasoning phase using an arbitrary off-the-shelf RDF/OWL reasoning library. As explained before, the Marvin system also includes the *one-exit door policy* for duplicate detection and removal.

We have implemented Marvin in Java, on top of Ibis, a high-performance communication middleware [13]. Ibis offers an integrated solution that *transparently* deals with many complexities in distributed programming such as network connectivity, hardware heterogeneity, and application deployment.

Experiments were run on the Distributed ASCI Supercomputer 3 (DAS-3), a five-cluster grid system, consisting in total of 271 machines with 791 cores at 2.4Ghz, with 4Gb of RAM per machine. All experiments used the Sesame in-memory store with a forward-chaining RDFS reasoner. All experiments we limited to a maximum runtime of one hour, and were run on smaller parts of the DAS-3, as detailed in each experiment.

The datasets used were RDF Wordnet[2] and SwetoDBLP[3]. Wordnet contains around 1.9M triples, with 41 distinct predicates and 22 distinct classes; the DBLP dataset contains around 14.9M triples, with 145 distinct predicates and 11 distinct classes. Although the schemas used are quite small, we did not exploit this fact in our algorithm (e.g. by distributing the schemas to all nodes a priori) because such optimisation would not be possible for larger or initially unknown schemas.

8.1. Baseline: null reasoner

To validate the behavior of the baseline system components such as buffers and routing algorithms, we created a "null reasoner" which simply outputs all its input data. We thus measure the throughput of the communication substrate and the overhead of the platform.

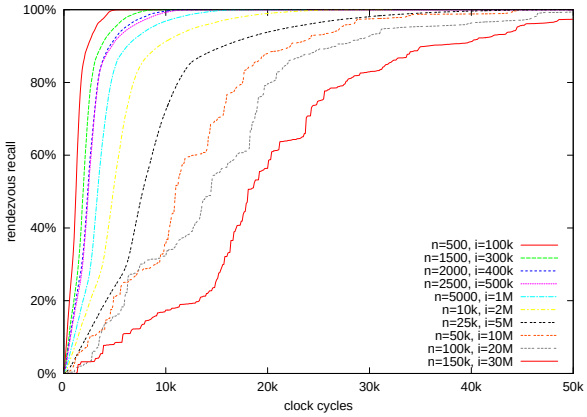
In this setup, the totality of the system reached a sustained throughput of 72.9 Ktps (thousand triples per second) per node, amounting to an aggregate throughput rate of some 2.3 Mtps on 32 nodes. Since typical loading times for RDF data (around 40 Ktps) [2] are well below this number, inter-node communication throughput (in the network used during our experiments) seems not to be a performance bottleneck. Note that we do not compare reasoning throughput here; these numbers merely show that transferring data is not the problem.

8.2. Scalability

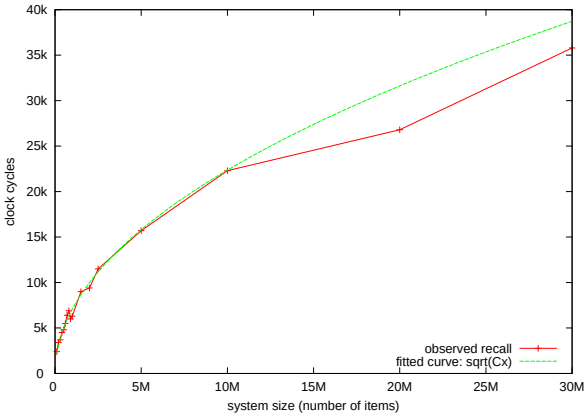
We have designed the system to scale to a large number of nodes. The Ibis middleware is based on reliable grid technology which allows Marvin to scale to a large number of nodes. Figure 10 shows the speedup gained by adding computational resources (using random routing, on the SwetoDBLP dataset), showing the number of unique triples produced for a system of 1–64 nodes.

Figure 10(a) shows the growth curves for different numbers of nodes. The sharp bends in the growth curves (especially with a small number of nodes) are attributed to the dynamic exit doors opening up: having reached the t_{upper} threshold, the nodes start sending their triples to the exit door, where they are counted and copied to the storage bin.

[2] http://larkc.eu/marvin/experiments/wordnet.nt.gz
[3] http://larkc.eu/marvin/experiments/swetodblp.nt.gz

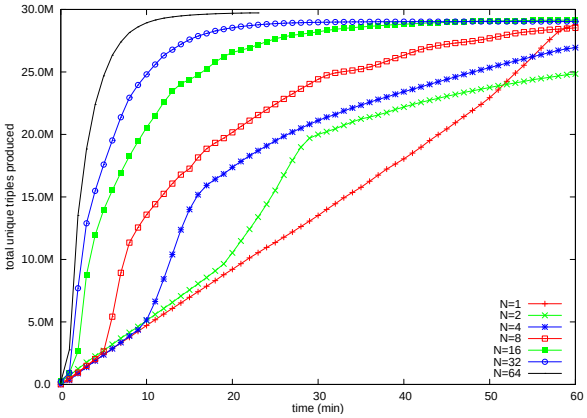


(a) recall over time

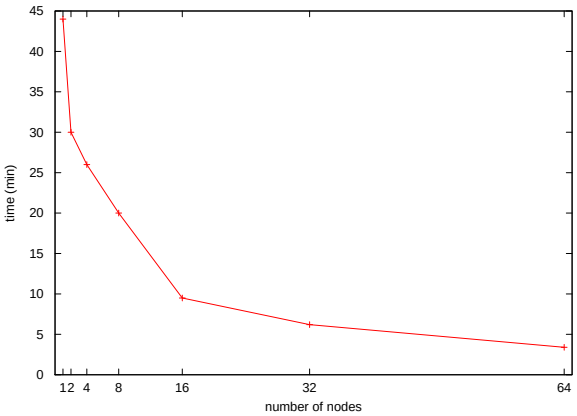


(b) time to reach 90% recall

Figure 9: Recall with increasing system size



(a) triples produced over time



(b) time to produce 20M triples

Figure 10: Triples derived using an increasing number of nodes

nodes	time (min)	speedup	scaled speedup
1	44	–	–
2	30	1.47	0.73
4	26	1.69	0.42
8	20	2.20	0.28
16	9.5	4.63	0.29
32	6.2	7.10	0.22
64	3.4	12.94	0.20

Table 5: Speedup for SwetoDBLP dataset

Table 5 shows the time needed to produce 20M triples from in the SwetoDBLP dataset. The same result is shown graphically in Figure 10(b). The table and graph show the amount of time needed over different numbers of nodes, the corresponding speedup (total time spent compared to time spent on a single node) and the scaled speedup (speedup divided by number of nodes). A perfect linear speedup would equal the number of nodes and result in a scaled speedup (speedup divided by number of nodes) of 1. To the best of our knowledge no relevant literature is available in the field to compare these results, but a sublinear speedup is to be expected in general. As we can see, the system scales gracefully.

8.3. Duplicate detection and removal

We have experimented with three different settings of the *dynamic sub-exit door*: "low" where

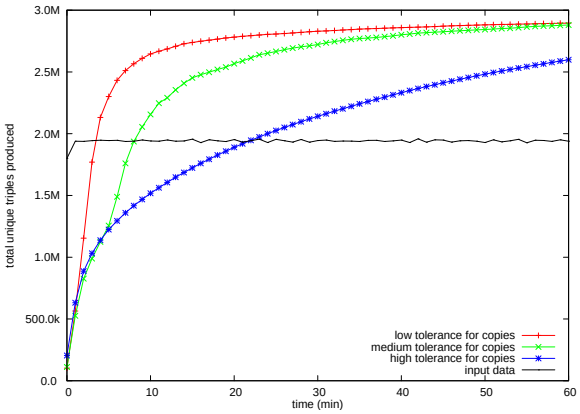


Figure 11: Triples derived using the dynamic exit-door policy

$t_{lower} = \alpha, t_{upper} = 2\alpha$; "medium" where $t_{lower} = 2\alpha, t_{upper} = 4\alpha$; "high" where $t_{lower} = 4\alpha, t_{upper} = 8\alpha$, where α is the number of input triples / N.

These different settings were tested on the Wordnet dataset, using 16 nodes with the *random* routing policy. The results are shown in Figure 11. As we can see, in the "low" setting, the system benefits from having low tolerance to duplicates: they are removed immediately, leaving bandwidth and computational resources to produce useful unique new triples. On the other hand, the duplicate detection comes at the cost of additional communication needed to send triples to the exit doors (not shown in the figure).

9. Conclusion

We have presented a platform for analysing Web data, with a focus on the Semantic Web. To process and interpret these datasets, we need an infrastructure that can scale to Web size and exploit the available semantics. In this paper, we have focused on one particular problem: computing the deductive closure of a dataset through logical reasoning.

We have introduced MARVIN, a platform for massive distributed RDF inference. MARVIN uses a peer-to-peer architecture to achieve massive scalability by adding computational resources through our novel *divide-conquer-swap* approach. MARVIN guarantees eventual completeness of the inference process and produces its results gradually (anytime behaviour). Through its modular design, MARVIN provides a versatile experimentation platform with many configurations.

We have developed SPEEDDATE, a scalable and load-balanced rendezvous mechanism. The benefits of our method are: (a) it is load-balanced, as opposed to a DHT-based approach, achieving a 40× smaller standard deviation in data load per node, (b) it clearly outperforms a random approach by orders of magnitude, performs within 3× better than the DHT-based approach and scales in general with $\sqrt{i}$, (c) it functions even using a small number of connections, namely 3, per node, (d) it is robust against failures and handles churn rates of up to 50% with less that 3× performance loss, (e) it is simple to implement and does not require expensive network maintenance.

We have experimented with various reasoning strategies using MARVIN. The experiments presented show that MARVIN scales gracefully with the number of nodes, that communication overhead is not the bottleneck during computation, and that duplicate detection and removal is crucial for performance.

Future work. As future work, we aim to evaluate MARVIN on larger datasets. We also plan to experiment with more expressive logics such as OWL Horst. Finally, we wish to investigate query answering on top of the MARVIN peer-to-peer model.

References

[1] D. Battré, A. Höing, F. Heine, and O. Kao. On triple dissemination, forward-chaining, and load balancing in DHT based RDF stores. In *Proceedings of the VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*. 2006.

[2] C. Bizer and A. Schultz. Benchmarking the performance of storage systems that expose SPARQL endpoints. In *Proceedings of the ISWC Workshop on Scalable Semantic Web Knowledge-base systems*. 2008.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] M. Cai and M. Frank. RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network. In *Proceedings of the International World-Wide Web Conference*. 2004.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pp. 137–147. 2004.

[6] M. Dean. Towards a science of knowledge base performance analysis. In *Proceedings of the ISWC Workshop on Scalable Semantic Web Knowledge-base systems*. 2008.

[7] Q. Fang, Y. Zhao, G. Yang, and W. Zheng. Scalable distributed ontology reasoning using DHT-based partitioning. In *Proceedings of the Asian Semantic Web Conference (ASWC)*. 2008.

[8] D. Fensel, F. van Harmelen, *et al.* Towards LarKC: A platform for Web-scale reasoning. In *Proceedings of the International Conference on Semantic Computing*, pp. 524–529. 2008.

[9] A. Hogan, A. Polleres, and A. Harth. Saor: Authoritative reasoning for the web. In *Proceedings of the Asian Semantic Web Conference (ASWC)*. 2008.

[10] Z. Kaoudi, I. Miliaraki, and M. Koubarakis. RDFS reasoning and query answering on top of DHTs. In *Proceedings of the International Semantic Web Conference (ISWC)*. 2008.

[11] D. R. Karger and M. Ruhl. Simple efficient load-balancing algorithms for peer-to-peer systems. *Theoretical Computer Science*, 39(6):787–804, 2006.

[12] K. Lua, J. Crowcroft, M. Pias, R. Sharma, *et al.* A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, pp. 72–93, 2004.

[13] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, *et al.* Ibis: a flexible and efficient Java based grid programming environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, 2005.

[14] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, *et al.* Sindice.com: A document-oriented lookup index for open linked data. *International Journal of Metadata, Semantics and Ontologies*, 3(1):37–52, 2008.

[15] B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *Proceedings of the European Semantic Web Conference (ESWC)*, pp. 524–538. 2008.

[16] K. Ruhloff, M. Dean, I. Emmons, D. Ryder, *et al.* An evaluation of triple-store technologies for large data stores. In *Proceedings of the ISWC Workshop on Scalable Semantic Web Knowledge-base systems*. 2007.

[17] S. Schenk and S. Staab. Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the web. In *Proceedings of the International World-Wide Web Conference*, pp. 585–594. 2008.

[18] A. Schlicht and H. Stuckenschmidt. Distributed resolution for ALC. In *Proceedings of the International Workshop on Description Logics*. 2008.

[19] L. Serafini and A. Tamilin. DRAGO: Distributed Reasoning Architecture for the Semantic Web. In *Proceedings of the European Semantic Web Conference (ESWC)*, pp. 361–376. 2005.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, *et al.* Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, 2001.