Open access • Proceedings Article • DOI:10.1109/PDCAT.2014.15

Dealing with Skewed Data in Structured Overlays Using Variable Hash Functions — Source link

Maeva Antoine, Fabrice Huet

Institutions: University of Nice Sophia Antipolis

Published on: 09 Dec 2014 - Parallel and Distributed Computing: Applications and Technologies

Topics: Hash function, Load balancing (computing), Data Web, Semantic Web and Big data

Dealing with Skewed Data in Structured Overlays using Variable Hash Functions

Maeva Antoine, Fabrice Huet

Dealing with Skewed Data in Structured Overlays using Variable Hash Functions

Maeva Antoine, Fabrice Huet
University of Nice Sophia Antipolis, CNRS, I3S, UMR 7271
06900 Sophia Antipolis, France
Email: FirstName.LastName@inria.fr

***Abstract*—Storing highly skewed data in a distributed system has become a very frequent issue, in particular with the emergence of semantic web and Big Data. This often leads to biased data dissemination among nodes. Addressing load imbalance is necessary, especially to minimize response time and avoid workload being handled by only one or few nodes. Our contribution aims at dynamically managing load imbalance by allowing multiple hash functions on different peers, while maintaining consistency of the overlay. Our experiments, on highly skewed data sets from the semantic web, show we can distribute data on at least 300 times more peers than when not using any load balancing strategy.**

***Keywords*—hash functions; load balancing; structured overlays; CAN; semantic web; RDF**

I. Introduction

Nowadays, many applications need to integrate data at web scale to extract information and knowledge. Semantic web technologies, such as RDF (Resource Description Framework [1]), provide useful tools for describing knowledge and reasoning on web data. With the advent of Big Data, it becomes incredibly difficult to manage realistic datasets on a single machine. Peer-to-peer (P2P) systems, such as Structured Overlay Networks (SON), are an efficient and scalable solution for data storage in large distributed environments. Many distributed RDF repositories based on a SON cluster semantically close terms, which appears as the best solution to efficiently processing range queries and reasoning on data [2] [3] [4]. Since RDF data is made of Unicode characters, distributing data among peers frequently boils down to partitioning the Unicode space. However, RDF datasets are often highly skewed, depending on the knowledge represented. As a result, a small subset of peers will have an arbitrarily large volume of triples to manage. Such a biased data distribution can lead to large workloads sent to a single node, which may impact system performance.

Various solutions have been proposed to achieve load balancing in SON (see Section II). We propose a very different technique that preserves the ordering of data and does not require to change the network topology. Our contribution aims at dynamically managing load imbalance by allowing multiple hash functions, while maintaining consistency of the overlay. More precisely, we allow a peer to change its hash function to reduce its load. Since this can be done at runtime, without *a priori* knowledge regarding data distribution, this provides a simple but efficient adaptive load balancing mechanism. Our approach is based on a CAN (Content Addressable Network [5]) storing semantic web data based on the RDF model. However, the technique presented here could also be used for other SON, such as Chord [6], and other data types. To summarize, we show in this paper that a SON can be consistent even when all peers do not apply the same hash function on data. We also describe a protocol which allows a peer to change its hash function at runtime. Then, we demonstrate through simulations that this strategy greatly improves load balancing of real RDF data.

The rest of the paper is structured as follows. In Section II, we present existing load balancing solutions for P2P systems. Section III introduces RDF data storage in a CAN. Section IV describes how peers can use different hash functions and change them at runtime. Based on this, Section V presents different load balancing strategies we propose. Section VI describes our experimental results and Section VII concludes the paper.

II. Existing Load Balancing Solutions

Many solutions have been proposed to address the load imbalance issue in P2P systems.

Hash functions can allow preventive load balancing. In [7], a peer having to insert an item applies n hash functions on the item's key and gets back n identifiers. Then, a probing request is sent to each peer managing one of these identifiers to retrieve their load state. Finally, the peer with the lowest load is chosen to store the item. This technique provides uniform data distribution, but destroys the natural ordering of information, useful for solving range queries.

Mercury [8] stores data contiguously on a Chord-like ring. Hence, range queries can be executed on contiguous peers, which would be impossible if randomizing hash functions were used for placing data. Mercury uses node migration as a load balancing solution: underloaded peers from lightly loaded areas are moved to overloaded areas.

Node replication, as used by Meghdoot [9], consists in replicating an overloaded peer's zone to a new peer joining the system. The authors recommand this approach when overload is due to high processing load, for example due to requests for popular items. Similar strategies replicating data [10] are useful in case of a node failure but consume more disk space and have high consistency constraints on updates.

However, all these techniques do not address the skewed data problem. Furthermore, some of them are very costly regarding network communication, if an overloaded peer first has to find a suitable underloaded peer in the overlay. Also, they may impose changes in the network topology, if peers change of neighbors. Our solution, described in the following sections, addresses the load imbalance problem from a different angle. Solving the data skewness issue is at the heart of our approach, while allowing efficient range query processing and without having to change the network topology.

III. RDF Storage within a CAN

A. Content Addressable Network (CAN)

A CAN network is a decentralized Peer-to-Peer infrastructure that can be represented as a *d-dimensional* coordinate space containing n nodes (denoted as peers). Each peer is responsible for the zone it holds in the network (a set of intervals in this space). All dimensions have a minimum and a maximum CAN-based value C_{min} and C_{max} (0 and 1 in Figure 1). Each peer p owns an interval $[min_p^d; max_p^d[$ delimited by 2 bounds (lower and upper bounds) between C_{min} and C_{max} on each dimension d. A CAN-based interval is constant and can only be modified during *join* or *leave* node operations. Each peer can only communicate with its neighbors, thus routing from neighbor to neighbor has to be done in order to reach remote zones in the network. The CAN topology is a torus which means, in Figure 1, that peers $p1$ and $p3$ are neighbors on the horizontal dimension.

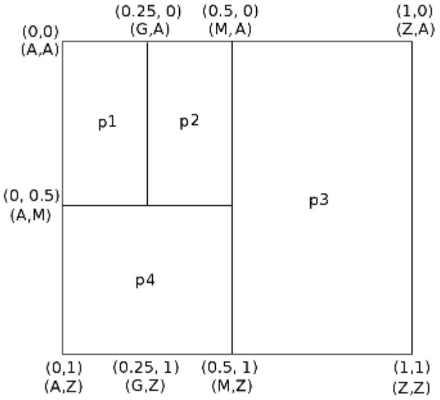


Figure 1: Example of a 2-dimensional Unicode CAN

B. Resource Description Framework (RDF)

The Semantic Web refers to W3C's vision of the Web of linked data[1], providing machine-understandable information. RDF provides a powerful abstract data model for structured knowledge representation and is used to describe semantic relationships between data. Resources are represented as *triples* in the form of $<subject;predicate;object>$ expressions. The $subject$ of a triple indicates the resource that the statement is about, the $predicate$ defines the property for the $subject$, and the $object$ is the value of the property. For example, when storing these two triples: $<John;fatherOf;Helen>$ and $<John;fatherOf;Elsie>$, a machine should be able to understand that $Helen$ and $Elsie$ are sisters and form a family along with $John$ (i.e. semantic relations).

C. Storage

Our approach for implementing a distributed storage of RDF data uses a 3-dimensional CAN in order to benefit from the 3-part structure of an RDF triple. This technique is based on the architecture of the Event Cloud [4], a CAN-based distributed RDF repository. Data is stored in a lexicographical order and each dimension corresponds to a part of a triple: $subject$, $predicate$ and $object$ (Figure 2). For all dimensions, minimum and maximum Unicode values U_{min} and U_{max} are set to determine the Unicode range that can be managed within the CAN (in Figure 2, U_{min} is equal to A and U_{max} to Z). The lexicographic order preserves the semantic information of data, by clustering triples sharing common values in contiguous peers. As a result, range queries, for instance, can be resolved with a minimum number of hops.

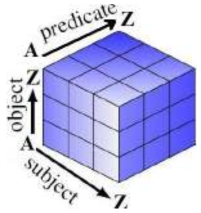


Figure 2: Structure of a 3-dimensional lexicographical CAN.

A Unicode value is associated to each of the peer's CAN bounds and a peer is responsible for storing Unicode values falling between these bounds on each dimension. For example, on Figure 1, $p1$ is responsible for data between $[A; G[$ (corresponding to CAN-based interval $[0; 0.25[$) on the horizontal dimension (for clarity, we will represent schemas as 2-dimensional CAN in this paper). The mapping relation between CAN-based and Unicode-based values can be seen as a form of hash function. Indeed, for each Unicode value corresponds a CAN-based value between C_{min} and C_{max}, obtained by applying a given hash function on the Unicode value. This hash function provides CAN-based coordinates that determine where a triple should be stored. The default hash function applied by all peers of Figure 1 for all dimensions is shown on Figure 3.

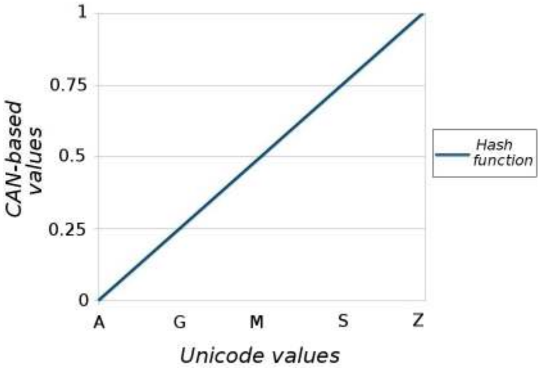


Figure 3: Default hash function.

We assume each peer stores RDF data and can easily sort triples alphabetically (using index trees for instance). A peer receiving a new triple to insert or a query to execute has to hash it to check whether it is responsible for it or not. For example, the hash value of string $ProductType1$ in the context of a CAN storing worldwide data (wide Unicode range, up to code point value 2^{20}) would be equal to CAN value 0.00004673 (i.e. at the far-left of the CAN). By default, strings made of Latin characters have a low hash value, whereas strings made of any Asian characters have high values (close to C_{max}). If the hash value does not match the peer's CAN coordinates, it means the peer is not responsible for the corresponding triple. In this case, the peer chooses a dimension and forwards the

[1] http://www.w3.org/standards/semanticweb/

triple/query on the same dimension until it reaches the correct coordinate for this dimension, then on another dimension and so on until the message reaches the right peer.

D. *Skewed data*

The order-preserving storage technique presented above suffers from a major drawback regarding data distribution. Indeed, having a system covering the whole Unicode range means potential overloaded areas may appear, depending on data distribution. Figure 4-(1) describes a system where only triples made of Latin characters are stored, which means only a small area of the CAN is targeted when inserting or querying data. In consequence, peer $p1$ becomes overloaded, while the rest of the network stores nothing as it is dedicated to other Unicode characters. Based on this observation, our contribution aims at dynamically adapting the size of skewed Unicode areas in a CAN, by changing hash functions to determine where data should be stored. We will present hereafter our notion of variable hash function and how it helps balance the load of a storage system.

IV. Variable Hash Functions

A. *Definition*

For each CAN-based interval $[min_p^d; max_p^d[$ of a peer p on a dimension d, corresponds a Unicode interval $[Umin_p^d; Umax_p^d[$. As a consequence, the matching between p's CAN interval and p's Unicode interval can be seen as the hash function applied by p for dimension d. A peer can use different hash functions on different dimensions. As in any standard structured overlay, all peers initially use the same uniform hash function. In the present case, our function linearly matches the minimum Unicode value that can be stored within our CAN U_{min} (resp. U_{max}) to the minimum CAN-based value C_{min} (resp. C_{max}). This is shown on Figure 3, where 0 and 1 respectively match A and Z.

When a peer is overloaded, it has to reduce the Unicode interval it is responsible for, in order to store less data. Changing the Unicode value of one of its bounds on a given dimension implies for the peer to change the hash function it applies on data. The general idea of this paper is to enable peers to change their hash function in order to improve skewed data distribution. For example, in Figure 4-(1), the default hash function used by all peers places data made of Latin characters on peer $p1$. New hash functions successively applied by $p1$, $p2$ and $p3$ evenly distribute data load between these peers (Figure 4-(2)), by enlarging the Latin interval. Empty areas are shrunk in order to give more space to the currently stored data. If, for instance, large Thai datasets are later inserted, peer $p5$ will have to change its hash function in order to balance its load (with peer $p6$, in Figure 4-(3)). Therefore, by changing a hash function, a skewed Unicode interval will be managed by more peers, and data distribution improved among them.

B. *Updating hash functions*

As there is no central coordination in an overlay, it is very difficult to change the overall hash function applied in the network. However, it is not necessary for all peers to use the same hash function to get a fully functional CAN. Peers can use different functions as long as these rules are observed:

(a) A peer should know the hash function of its neighbors for each dimension, and should notify them as soon as it changes its hash function.

(b) Neighboring peers using different hash functions should have the same Unicode value on their common bound.

(c) All peers on a given CAN-based bound should apply the same hash function on this bound. Peers whose CAN interval includes this bound should also know this hash function, without needing to apply it.

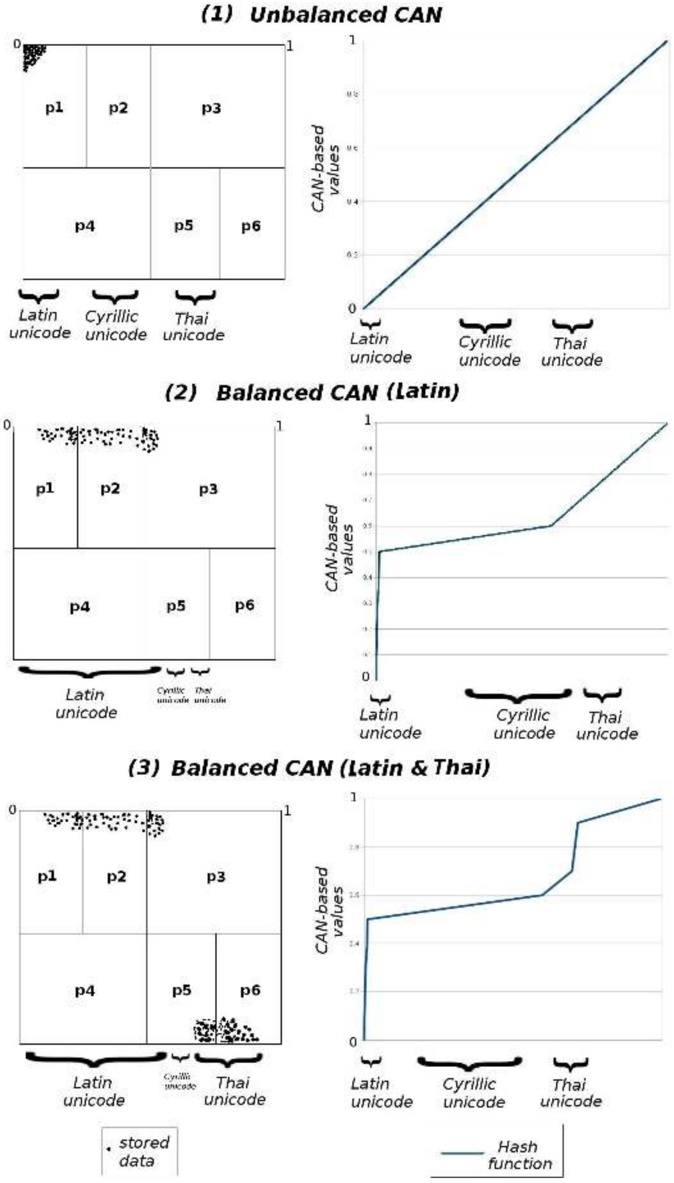


Figure 4: Hash function change depending on data skewness.

Rule (a) is required to ensure the arrival of new peers (division of an existing zone between two peers) is correctly handled. Also, when routing data in the network, this allows a peer to choose among its neighbors which one will be the most efficient for routing a particular data. In the remaining of the paper, we consider that each peer knows its neighbors and the hash function they use for each dimension. Rule (b) ensures overall consistency of the overlay: two peers sharing a common CAN-based value should share a common Unicode-based value, even if they use different hash functions. Given two neighbor peers $p1$ and $p2$ using $h1$ and $h2$ as their respective hash functions on dimension d, if $max^d_{p1} = min^d_{p2}$ then $h1(max^d_{p1}) = h2(min^d_{p2})$. This rule prevents concurrent hash function changes from creating overlap or empty areas in the CAN. Finally, rule (c) keeps in line with rule (b) and is necessary to maintain optimal routing in the overlay. Figure 5 describes the impact of rule (c) by showing the scope of a hash function change in a CAN. Peer $p1$ decides to change its hash function to reduce its Unicode interval on CAN-based bound 0.75. Thus, all peers on 0.75 (the grey area in Figure 5) should also apply this new hash function. A multicast message is sent by $p1$ and routed to all concerned peers, which also includes $p2$. Indeed, $p2$ will not apply the new hash function but must be aware of it to respect rule (a), as its CAN interval $[0.5; 1[$ comprises 0.75. This way, if a new peer wants to join $p2$ and split on its horizontal interval, $p2$ and the new peer will both know which Unicode value to apply on their new 0.75 CAN bound. Also, $p2$ must continue routing the multicast message in order to reach $p3$ and the other concerned peers.

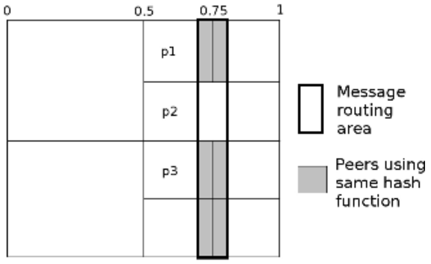
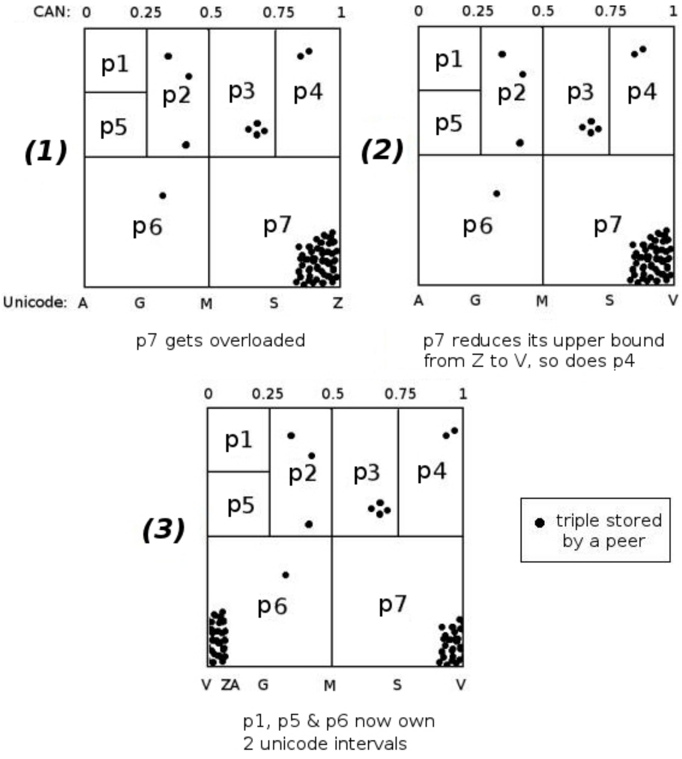


Figure 5: Hash function update message routing.

We will now focus on the process that happens when a peer wants to change its hash function. For the sake of simplicity, we will only allow a peer to reduce its upper bound. The possible negative consequences of this limitation are addressed in Section IV-B2.

1) Message propagation: If a peer decides to change its hash function (see Section V for details on how the new function is calculated), it applies the update on itself then sends a multicast update bound message. This message is propagated to all concerned peers on this CAN-based bound to make them change their associated Unicode value. Indeed, all other peers having one of their bounds on the same CAN-based value on the same dimension must apply the new value, too. This is very important in order to keep the same topology (same neighbors) and avoid inconsistency within the CAN (empty zones no one is responsible for, or overlap zones managed by two peers). The overloaded peer cannot predict if this change will overload some of the concerned peers since they may not be its neighbors. However, this can be easily addressed, as peers applying this new hash function can also induce their own hash function change later on.

2) Bound reduction: Peers having their CAN-based upper bound equal to the maximum authorized CAN value C_{max} should also be able to reduce their upper Unicode bound. Otherwise, if, for example, a large amount of non-Latin RDF triples is sent into the system, they would probably be stored at the bottom right corner of the CAN with no possibility for the responsible peer to balance its load. To avoid this situation, we allow peers located at the far right of the CAN to reduce their upper Unicode bound like any other peer. To do so, we take advantage of the tore-ring topology of the network, and consider $C_{min} = C_{max}$, which means all Unicode updates associated to C_{max} must also be applied to all peers on C_{min}. As a consequence, peers on C_{min} can become responsible for two Unicode intervals within a single CAN-based interval. This is shown in Figure 6, representing a CAN where the minimum Unicode value allowed U_{min} is A and the maximum U_{max} is Z. At step 3, peers $p1$, $p5$, and $p6$ own two different intervals: respectively $[[V; Z]; [A; M[]$ for $p6$ and $[[V; Z]; [A; G[]$ for $p1$ and $p5$. Therefore, when a peer is responsible for two Unicode intervals, the first interval must stop at U_{max} and the second one must start at U_{min}.



p7 gets overloaded

p7 reduces its upper bound from Z to V, so does p4

p1, p5 & p6 now own 2 unicode intervals

Figure 6: Unicode reduction process on the C_{max} bound.

3) Concurrency and message filtering: For a given bound, a peer p receiving an update message will allow the new Unicode value only if it is lower than p's current Unicode value for the same CAN-based value. Thus, it is forbidden to ask for a forwards reduction, in order to avoid inconsistent concurrent reductions (i.e. on different directions). For example, in Figure 6, if $p6$ and $p7$ decided in a short time interval to modify the Unicode value for the same CAN bound (0.5) but on different directions (backwards: from M to J for $p6$ and forwards: from M to Q for $p7$), it would become difficult to tell which one has priority over the other, especially since their multicast messages may not be received in the same order by all concerned peers ($p2$, $p3$, $p6$, $p7$). Moreover, if messages containing different hash functions are sent by different peers at the same time and for the same CAN-based bound, we ensure all peers will end up choosing the same value: the lowest one. Thus, peers will reject the other messages and stop their multicast routing. The only case where a higher Unicode value would be accepted is when reduction is made by a peer containing two intervals on one dimension (see IV-B2) and the new value is included in the first interval. On Figure 6, if, after step 3, $p1$ wants to reduce from G to Y, it would no longer be responsible for two Unicode intervals (only $[V; Y[$), but its neighbor $p2$ would $([[Y; Z]; [A; M[])$.

4) Data movement: After a peer p has changed its hash function, it has to send all triples it is no longer responsible for to its neighbor(s). To do so, p has to wait until they apply the update as well (it should be done quickly as they are only one hop away from p). After sending its triples, p waits until it receives a storage acknowledgement message from its neighbor(s) before deleting triples, in order to ensure no triple is lost during this process.

V. Load Balancing

A. Computing a new hash function

In this section, we explain how adaptive hash functions are calculated to manage load imbalance. We consider the number of RDF triples a peer stores as the imbalance criterion, assuming all triples are of a similar data size.

Let p be a peer storing a given number of triples noted as $load_p$ on a Unicode interval $[Umin_p, Umax_p[$. If p is overloaded, it has to send all triples above a certain limit, lim, to its neighbor(s) on the forward direction on a dimension d, such as the new value of $load_p \leq lim$. Basically, p has to move its upper Unicode bound backwards, such as only lim data remain in p's zone. This is achievable by simply changing p's hash function from $h1$ to $h2$ such that the new value of $h2(max_p^d)$ (i.e. the new value of $Umax_p^d$) is equal to the Unicode value of p's $(lim + 1)^{th}$ triple (as we consider p can sort triples alphabetically). An example is shown in Figure 7, where we consider a peer is overloaded if it stores more than 6 triples. Thus, in Figure 7-(1), peer $p1$ is overloaded and has to change its hash function on max_{p1}^d, corresponding to Unicode value $mango$. A new hash function $h2$ should transform max_{p1}^d into the Unicode value of $p1$'s 7^{th} triple (as lim is equal to 6), i.e. fig. In Figure 7-(2), $p1$ and $p2$ have changed their hash function to $h2$ respectively on max_{p1}^d and min_{p2}^d, and their load is evenly balanced.

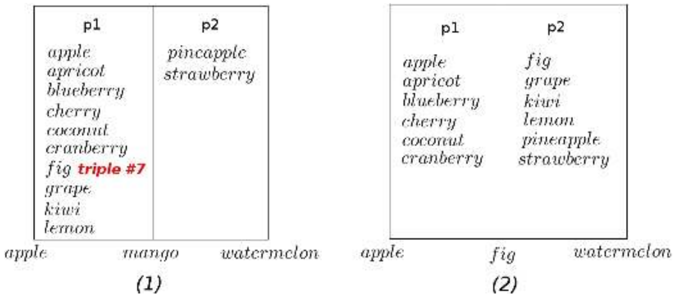


Figure 7: Load distribution before and after rebalancing.

B. Load balancing policies

We propose hereafter three different strategies a peer can use to check whether it should induce a rebalancing, and describe how the limit is calculated.

1) Threshold-based policy: A peer p is said to be overloaded when it stores more data than the limit set by a threshold $thres$. If a peer is overloaded, it has to send all its triples above $thres$ to its neighbour(s) on the forward direction for a randomly chosen dimension d. This approach is similar to the implementation on top of a Skip Graph proposed by [11]. The new Unicode value $h2(max_p^d)$ would be equal to the Unicode value of p's $(thres + 1)^{th}$ triple.

2) Locally-based policy: Each peer p periodically contacts its forward neighbors $Fneighbors_p^d$ on a random dimension d to calculate the average load $localAvg$ in its neighborhood. If p finds out it stores more data than $localAvg$ and more data than a given threshold $localThres$, then p can induce a rebalance. We use $localThres$ to ensure a peer storing very few triples will not be allowed to change its hash function if it is surrounded by peers storing few triples, too. However, as the data distribution and load are not known in advance, relying on a threshold can be problematic. One way to address this would be for peers to independently update their threshold value at runtime, depending on their knowledge of the system's load state. Then, p calculates how many triples $triplesToMove_p$ it has to send so that it stores as many triples as the sum of triples stored by $Fneighbors_p^d$. Then, $h2(max_p^d)$ would be equal to the Unicode value of p's $(load_p - triplesToMove_p)^{th}$ triple.

3) Overall-based policy: In the present case, we assume each peer knows an estimate of the average network load, thanks to a gossip protocol or by asking a peer having a global knowledge on the system. As the overall load may be very low in a network made of thousands of nodes, we consider a peer p is overloaded if it stores at least n times more triples than the average overall load $overallAvg$. If so, $h2(max_p^d)$ would become equal to the Unicode value of p's median triple, in order to send half of p's triples forwards in the network, without underloading p.

VI. Experiments

A. Experimental setup

To validate our approach, we ran extensive experiments in a cycle-based simulator: PeerSim [12]. A cycle represents the time required by a peer to perform a basic operation (message routing, load checking, etc.). We simulate a 3-dimensional CAN composed of 1000 peers. Maximum Unicode value supported U_{max} is set to code point 2^{20} on all dimensions, to encompass the whole Unicode characters table. In order to make realistic experiments, we used a dataset made of triples from three different sources. The first two are only composed of Latin Unicode characters, from the Berlin BSBM benchmark [13] and the LC Linked Data Service[2]. The third source is extracted from DBpedia[3] and contains Japanese Unicode characters. One million triples are inserted into the network in the space of 15 cycles, to simulate bursty traffic.

A perfect data distribution would correspond to all 1000 peers storing 1000 triples. However, this is not achievable in practice for two reasons. First, the CAN topology may not be perfect (some zones larger than others). Secondly, real RDF data is often very skewed, due to the large scope of the whole Unicode and the fact RDF triples may contain very similar values, like ID numbers that differ by a single character.

We ran experiments for each strategy described in V-B. For the threshold-based (resp. locally-based) approach, we set a limit of 8000 (resp. 30000) triples. For the overall-based strategy, we used a coefficient of 15, which means a peer was overloaded if it stored more than $overallAvg \times 15$ triples.

Table I: Load balancing results for each strategy.

Strategy	Peers storing data	Changes of hash function	Moved triples	Cycles to achieve balance	Standard deviation (triples)
None	2	0	0	0	235702
Threshold	652	156	10333076	80	1618
Local	818	170	5153092	100	2675
Overall	602	90	4626023	45	1900

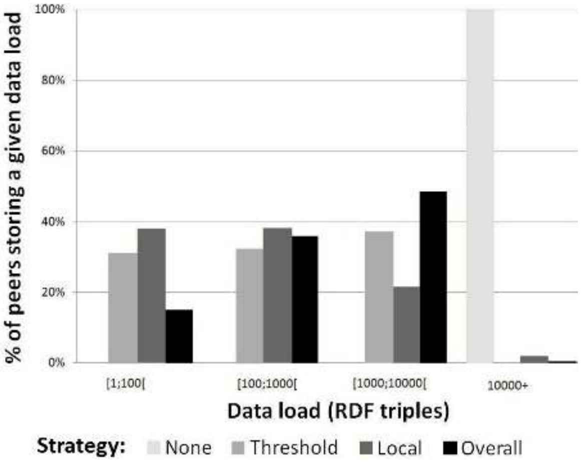


Figure 8: Load distribution (excluding peers storing no data)

B. Results

Table I summarizes obtained results for each load balancing strategy at the end of simulations. Results are expressed in terms of number of peers storing data (out of 1000), hash functions changed by overloaded peers, cumulated total number of triples moved from peers in order to balance their load, cycles to achieve balance[4] (after all triples are stored by the very first peer) and standard deviation in terms of RDF triples stored. Figure 8 represents the load distribution at the end of our experiments, excluding peers storing no data. For example, using the threshold-based strategy, 31% of peers storing data hold between 1 and 99 triples, 32% store between 100 and 999 triples, 37% between 1000 and 9999, while no peer stores more than 9999 triples.

Without applying any load balancing strategy, results show that only 2 peers would store data, because of the large bias between datasets (Latin vs Japanese characters). These two peers respectively correspond to the one at the far left of the CAN on each dimension, responsible for storing all Latin triples, and the peer at the far right of the CAN on each dimension, responsible for storing all Japanese triples.

The threshold-based strategy offers the lowest standard deviation between the load of peers but requires moving a large number of triples (over 10 million). The locally-based strategy distributes data across more peers than other strategies, but has a higher standard deviation and requires slightly more updates on hash functions (170). As a consequence, it takes more time to achieve balance. The overall-based strategy requires less communication between nodes (only 90 changes of hash function and 4626023 triples moved), which also means peers will store more data, but balance will be achieved sooner.

To ensure that load balancing operations do not affect the consistency of the overlay, 200 random queries were sent to random peers throughout the experiments. For all of them, a result was received within a reasonable time (an average of 15 hops to route queries or results). This is because a triple is always stored at least on one peer. Moreover, moving data is only done once an update bound message is applied by the sender and the receiver. Thus, reaching the new responsible peer for a given triple should not require more than a few hops in the network, at the very most.

These results show that our approach efficiently improves load distribution when storing highly skewed data, while maintaining consistency regarding storage and network topology. Using no load balancing strategy, only 2 peers store data. Therefore, it is very likely that they would quickly become overloaded. Conversely, up to 818 peers were able to share workload, when allowing different hash functions within the overlay.

VII. Conclusion

In this paper, we have presented a load balancing technique for storing highly skewed data in SON. We have shown it is not necessary for all peers to use the same hash function, while continuing to maintain optimal routing and consistency in a CAN overlay. Our approach is based on a dynamic adaptation of hash functions to data skewness, in order to improve data distribution among peers. We applied this technique in the context of distributed RDF data storage, using highly skewed but nonetheless realistic data. We simulated an overlay composed of 1000 peers and injected 1 000 000 triples. Load balancing in the overlay was greatly improved, the number of peers storing data going from 2 up to 818. This work is performed without *a priori* knowledge about data distribution. Although we used a CAN overlay throughout this paper, the ideas and techniques presented could be used by any DHT overlay network.

References

[1] G. Klyne and J. J. Carroll, "Resource description framework (rdf): Concepts and abstract syntax," 2006.

[2] K. Aberer, P. Cudré-Mauroux, M. Hauswirth, and T. Van Pelt, "Gridvine: Building internet-scale semantic overlay networks," in *The Semantic Web–ISWC 2004*. Springer, 2004, pp. 107–121.

[3] L. Ali, T. Janson, and G. Lausen, "3rdf: Storing and querying rdf data on top of the 3nuts overlay network," in *Database and Expert Systems Applications, International Workshop on*. IEEE, 2011, pp. 257–261.

[4] I. Filali, L. Pellegrino, F. Bongiovanni, F. Huet, and F. Baude, "Modular P2P-based Approach for RDF Data Storage and Retrieval," in *Proceedings of The Third International Conference on Advances in P2P Systems (AP2PS 2011)*, 2011.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," vol. 31, no. 4, pp. 161–172, October 2001.

[6] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4. ACM, 2001, pp. 149–160.

[7] J. Byers, J. Considine, and M. Mitzenmacher, "Simple load balancing for distributed hash tables," in *Peer-to-peer systems II*. Springer, 2003, pp. 80–87.

[8] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: supporting scalable multi-attribute range queries," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 353–366, 2004.

[2] http://id.loc.gov/

[3] http://dbpedia.org

[4] By achieving balance, we mean no more peer is overloaded, according to a given strategy.

[9] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi, "Meghdoot: content-based publish/subscribe over p2p networks," in *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Springer-Verlag New York, Inc., 2004, pp. 254–273.

[10] M. Cai and M. Frank, "Rdfpeers: a scalable distributed rdf repository based on a structured peer-to-peer network," in *Proceedings of the international conference on WWW*. ACM, 2004, pp. 650–657.

[11] I. Konstantinou, D. Tsoumakos, and N. Koziris, "Fast and cost-effective online load-balancing in distributed range-queriable systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 8, pp. 1350–1364, 2011.

[12] A. Montresor and M. Jelasity, "Peersim: A scalable p2p simulator," in *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*. IEEE, 2009, pp. 99–100.

[13] C. Bizer and A. Schultz, "The berlin sparql benchmark," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 5, no. 2, pp. 1–24, 2009.