Saturn: Range Queries, Load Balancing and Fault Tolerance in DHT Data Systems

Theoni Pitoura, *Member*, *IEEE*, Nikos Ntarmos, and Peter Triantafillou

Abstract—In this paper, we present *Saturn*, an overlay architecture for large-scale data networks maintained over Distributed Hash Tables (DHTs) that efficiently processes range queries and ensures access load balancing and fault-tolerance. Placing consecutive data values in neighboring peers is desirable in DHTs since it accelerates range query processing; however, such a placement is highly susceptible to load imbalances. At the same time, DHTs may be susceptible to node departures/failures and high data availability and fault tolerance are significant issues. *Saturn* deals effectively with these problems through the introduction of a novel multiple ring, order-preserving architecture. The use of a novel order-preserving hash function ensures fast range query processing. Replication across and within data rings (termed vertical and horizontal replication) forms the foundation over which our mechanisms are developed, ensuring query load balancing and fault tolerance, respectively. Our detailed experimentation study shows strong gains in range query processing efficiency, access load balancing, and fault tolerance, with low replication overheads. The significance of *Saturn* is not only that it effectively tackles all three issues together—i.e., supporting range queries, ensuring load balancing, and providing fault tolerance over DHTs—but also that it can be applied on top of any order-preserving DHT enabling it to dynamically handle replication and, thus, to trade off replication costs for fair load distribution and fault tolerance.

Index Terms—Distributed databases, distributed applications, fault tolerance, query processing, internet applications.

1 Introduction

Structured peer-to-peer (P2P) systems have provided the P2P community with efficient and combined routing and location primitives. This goal is accomplished by maintaining a structure in the system, emerging by the way that peers define their neighbors. Different structures have been proposed, most popular of which being 1) distributed hash tables or DHTs (CAN [31], Pastry [32], Chord [35], Tapestry [38]), using hashing schemes to map peers and data keys to a single, circular identifier space, and 2) distributed balanced trees (P-Grid [1], PHT [30], BATON [21], etc.), where nodes are placed in a tree-based topology.

One of the biggest shortcomings of DHTs, having spurred considerable research, is that they only support exact-match queries. Therefore, the naive approach to deal with range queries over DHTs—i.e., to individually query each value in the range—would be greatly inefficient and thus inapplicable in most cases. Although there are many research papers that support range queries over DHTs more "cleverly" and, thus, more efficiently [3], [17], [33], [36], all of them suffer from access load imbalances in the presence of skewed data-access distributions. Only a few approaches deal with both problems, i.e., load balancing and efficient range query processing in DHTs [7], [15], [1] or other structures [4], [14], [21]. However, these solutions are based on data migration which is often inadequate in skewed data access distributions since transferring, for instance, a single popular value from a heavy loaded peer to another simply transfers the problem. In such cases, access load balancing is best addressed using replication of popular values to distribute the access load among the peers storing such replicas.

In this work, we propose solutions for efficiently supporting range queries over order-preserving DHTs while also providing fair load distribution and fault tolerance using tunable replication mechanisms. Order-preserving hashing has already been proposed for centralized databases, especially for decision support (such as aggregation, moving sums, top-N, or bottom-N, etc.) queries, where the ordering of the results is necessary (e.g., order-preserving hash joins [10]). In large-scale DHT-based databases, order-preserving hashing can place consecutive data values on neighboring peers to efficiently support range queries (OP-Chord [36], MAAN [9], etc.); thus, collecting the values of a range query can be achieved by single-hop neighbor-to-neighbor visits. However, order-preserving DHTs may exacerbate load imbalances because order-preservation and load balancing conflict. Replicating popular data and also preserving their placement order is not simple: if the replicas of a popular value are placed in neighboring peers, the access load balancing problem still exists in this neighborhood of peers that is already overloaded. On the other hand, if the replicas are randomly distributed, additional hops are required each time a replica is accessed during range query processing. Successfully addressing these two conflicting goals and also ensuring data availability are the main targets of the proposed architecture.

- *T. Pitoura is with the Department of Computer Engineering and Informatics, University of Patras, Sisini 12-14, Patras 26225, Greece. E-mail: pitoura@ceid.upatras.gr.*
- *N. Ntarmos is with the Department of Computer Science, University of Ioannina, PO Box 1186, Ioannina 45110, Greece. E-mail: ntarmos@cs.uoi.gr.*
- *P. Triantafillou is with the Department of Computer Engineering and Informatics, University of Patras, Rio 26504, Patras, Greece. E-mail: peter@ceid.upatras.gr.*

Manuscript received 10 Feb. 2010; revised 31 July 2010; accepted 16 Oct. 2010; published online 23 Dec. 2010.
Recommended for acceptance by K.-L. Tan.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2010-02-0083.
Digital Object Identifier no. 10.1109/TKDE.2010.266.

Specifically, we develop a multiring order preserving architecture on top of an order-preserving DHT overlay. The proposed architecture consists of:

1. A novel multiring hash function used for data placement on top of the order-preserving DHT ring, both preserving the order of values and handling value replication in multiple data rings.
2. A load-driven replication scheme which dynamically replicates popular data across rings to ensure access load balancing. This scheme also supports tunable replication: by tweaking the maximum degree of replication, a system parameter, and per-node load thresholds, it trades off replication costs for access load balancing.
3. A fault tolerance mechanism which detects peer failures and retrieves lost data. An additional replication scheme is proposed, the horizontal k-replication, which both guarantees data availability and efficiency of range query processing, even with high-failure probabilities (fpr).

We coin this architecture *Saturn*[1] because of its multiple data rings created by replication. *Saturn* employs load-driven replication (visualized as "vertical rings") to ensure access-load balancing, and horizontal k-replication (visualized as "horizontal rings") to ensure data availability and to also preserve the order of values to support efficient range query processing.

We comprehensively evaluate *Saturn* and compare it against baseline competitors for efficiency of range query processing, load balancing, and fault tolerance. Results show significant speedups in range query processing, fairly distributed accesses in DHT peers, and high data availability even in cases of frequent peer failures, all while requiring reasonable replication costs. As the range query size or data-access skewness increase, so do the benefits of our solution. We further study the tradeoffs between replication costs and the load balancing achieved.

The rest of the paper is organized as follows: Section 2 describes a generic order-preserving DHT overlay, to be used as a starting point for building our solutions. Section 3 presents the *Saturn* infrastructure, its multiring hash function, and its load-driven replication mechanism. Section 4 describes the range query processing algorithm in *Saturn* and Section 5 presents its fault tolerance mechanism. Section 6 discusses various general aspects of the proposed architecture, focusing on its flexibility and its ability to tradeoff replication and maintenance costs for a fairer load distribution and/or a higher fault tolerance, while maintaining its range query processing efficiency. Section 7 experimentally evaluates *Saturn* in comparison to traditional DHTs, testifying to its superiority in efficiently handling range query processing, load balancing, and fault tolerance concurrently, and exhibiting its ability to tune replication. Finally, Section 8 presents related work and Section 9 concludes this work.

1. *Saturn* is the second largest planet of our solar system, named after the Greek god *Kronos* (or *Saturnus* by the Romans), the Titan father of Zeus. Its planetary rings make it the most visually amazing planet.

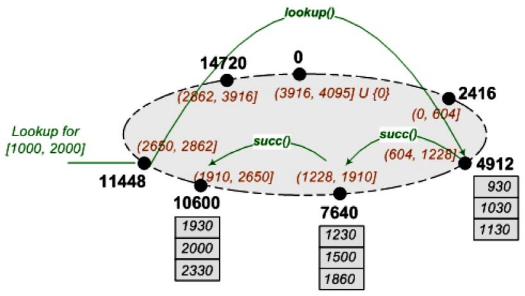

Fig. 1. The underlying order-preserving ring-based DHT.

2 The Underlying DHT Overlay

In this section, we overview the underlying DHT overlay that our proposed architecture is built upon. *Saturn* can be implemented on top of any ring-based order-preserving (i.e., where semantically close data are hashed into the same or consecutive peers) DHT, such as OP-Chord [36], MAAN [9], Mercury [7], etc. However, it can also be applied with small adaptations on top of any other order-preserving structure, such as skip lists [29] (e.g., Skip Graphs [4], [5], SkipNet [19]), tries (e.g., an adaptation of P-Grid in [13]), etc.

We assume that our data objects are tuples of a x-attribute database relation $R(A_1, A_2, \ldots, A_x)$, with each A_i also being used as a single-attribute index of the tuples in R. Furthermore, every tuple t in R is uniquely identified by its primary key, calculated using one or more of the tuple's attributes. We further assume these attributes take on integer values from a set of respective domains $DA_i, 1 \leq i \leq x$; other value types can be handled by conversion to an integer form.

In DHT-based networks, peers and data are assigned unique identifiers from a circular m-bit identifier space. Traditional DHTs use secure or quasi-random hash functions to randomly and uniquely assign identifiers to peers and data. As with many existing DHTs (CAN [31], Pastry [32], Chord [35], etc.), we assume that tuples are mapped to peers using *consistent hashing* [23]: a tuple with identifier *id* is stored at the peer whose identifier is either equal to *id* or the closest larger one among all peers; this peer is called the successor of *id* and is denoted by *succ*(*id*). Similarly, the peer whose identifier is the closest smaller one to an *id* is called its predecessor and denoted by *pred*(*id*). Each peer maintains routing state in the form of a *routing table*, consisting of pointers to 1) its predecessor and successor, and 2) O(log N) peers responsible for id's at exponentially increasing distances from its own id. Routing a message from one peer to another requires O(log N) messages in the worst case, where N is the number of peers in the network.

We assume that each tuple is stored on the peer mapped by securely hashing its primary key. Moreover, pointers to the latter are stored on x nodes whose identifiers are produced by hashing the values of each of the tuple's x attributes using a set of respective order-preserving hash functions, $hash_i()$. We should mention here that we could also store full instances of each tuple on each of these x nodes. At this point we make no distinction between these two cases.

Example 1. Fig. 1 illustrates data placement in a 14-bit order-preserving Chord-like ring, i.e., the id space is [0, 16384).

We assume tuples with a single-attribute A taking values in the interval $DA_A = [0, 4{,}096)$. Let $\mathrm{N} = 7$ peers with identifiers 0, 2,416, 4,912, 7,640, 10,600, 11,448, and 14,720 inserted in the network. Each peer is responsible for storing a partition of DA_A in an order-preserving way, as shown.

Range queries progress from the peer responsible for the range's lowest value to the peer responsible for its highest value following successor pointers. Thus, if there are n′ such nodes for a given range, including those responsible for the range's edge values, this algorithm requires $\mathrm{O}(\log \mathrm{N} + \mathrm{n}')$ hops to compute the result set [36].

Example 2. Fig. 1 illustrates how the range query [1,000, 2,000] initiated at peer 11,448 is answered. Using the look-up operation (*lookup*()) of the underlying DHT network, we initially move to peer 4,912, responsible for the lower limit of the range, and gather all locally stored tuples satisfying the query predicate. Then, the query is continuously forwarded to the successor peers (using *succ*()) until it reaches peer 10,600 responsible for the upper limit of the range.

Although this scheme accelerates the processing of range queries, it cannot handle load balancing in the case of skewed access distributions, when peers that store popular data become overloaded. *Saturn* deals with this problem while still attaining the efficiency of range query processing. Without loss of generality, from this point forward we assume that R consists of a single index attribute A with domain DA (each attribute is handled independently, and no extra routing structure is needed).

3 The Saturn Architecture

Although order-preservation in DHT-based data networks (such as OP-Chord [36], MAAN [9], etc.) can support efficient range query processing, it comes at a price: in the presence of skewed access distributions, order-preserving data placement can exacerbate the implicit access load imbalances. The proposed architecture, *Saturn,* built over such an order-preserving DHT, achieves a fair access load distribution (i.e., an almost uniform distribution [27]) by detecting overloaded peers and randomly distributing their access load among peers in the system by replicating their popular data items, all while maintaining the order-preserving data placement.

The main idea behind *Saturn* is a novel hash function which 1) handles multiple data replicas to preserve the data ordering of the underlying DHT, and 2) randomly distributes accesses among peers to balance access load. We should mention that instances of the algorithms run at each peer and no global schema knowledge is required. The notation used is summarized in Table 1.

TABLE 1
Notation

Notation	Definition
$\rho_{max}(A)$ or ρ_{max}	Maximum replication degree (i.e. number of instances) of tuples with attribute A
$\rho_{min}(A)$ or ρ_{min}	Minimum replication degree of tuples with attribute A
$\rho(v)$	Current replication degree of tuples with value v for attribute A
$Rhos_p[]$	Local array of current replication degrees of all values stored on peer p at any ring
$copiedRanges_p[]$	Local array of ranges whose values have instances on peer p
$a_p(v,\delta)$	Number of accesses of value v on peer p at ring δ
a_{min}, a_{max}	Locally set lower and upper limits for $a_p(v,\delta)$
$R_p(\delta) =$ $[lowest_p(\delta), highest_p(\delta)]$	Range of values peer p is responsible for at ring δ
$avgR_p(\delta) =$ $[avgLow_p(\delta), avgHigh_p(\delta)]$	Average lower and upper bounds of range queries processed by peer p at ring δ
$repR_p(\delta)$	Range of values being replicated/deleted as a consequence of peer p being overloaded/underloaded at ring δ

3.1 The Infrastructure

We assume that, for each of its attributes $A_i,$ a tuple can have a maximum number $\rho_{max}(A_i)$ (or simply ρ_{max}) of instances. An "instance" is either the original tuple or one of its replicas. This limit is a system parameter, set when *Saturn* is deployed, and is the same for all values of an attribute. We then define the variable $\delta \in [1, \rho_{max}]$ to indicate a specific instance, so that the original tuple corresponds to $\delta = 1$ (the first instance), its first replica corresponds to $\delta = 2$ (the second instance), and so on. If the value of attribute A is v, then its δth instance is assigned an identifier according to the following novel hash function.

Definition 1. *For any value* $v \in DA$ *and* $\delta \in [1, \rho_{max}]$, *the* ***Multiring Hash Function (mrhf())*** is defined as

$$\mathrm{mrhf}(v,\delta) = (\mathrm{hash}(v) + (rotation[\delta] - 1)\cdot s) \mathrm{mod}\, 2^m, \quad (1)$$

where rotation[] is a random permutation of $\{1, 2, ..\rho_{max}\}$, $rotation[1] = 1; s = 2^m/\rho_{\max}$*is the rotation unit; hash(v) is the order-preserving hash function of the underlying DHT.*

It is obvious that for $\delta = 1$, *mrhf*() is identical to *hash*() and is thus a one-to-one order-preserving mapping from domain *DA* to $\{0, 1, \ldots, 2^m - 1\}$. This means that, if $v, v' \in DA$ and $v \leq v'$, then $mrhf(v,1) = hash(v) \leq hash(v') = mrhf(v',1)$. For any $\delta > 1$, $mrhf()$ preserves the order and distance of values on the ring in a clockwise direction, i.e., if v precedes v' on the ring by i, then $mrhf(v,\delta)$ also precedes $mrhf(v',\delta)$ by i.

Therefore, a tuple t with an attribute value v is initially placed on peer $succ(mrhf(v,1))$—that is, $succ(hash(v))$—according to the underlying DHT. When the δth instance of t is created, i.e., its $(\delta - 1th)$ replica for $\delta > 1$, it is placed on the peer responsible for *hash*(v) advanced clockwise by $(rotation[\delta] - 1)\cdot s$ positions in the identifier circle (modulo 2^{m}). This can be illustrated as a clockwise rotation of the identifier ring by $(rotation[\delta] - 1)\cdot s$, where s is the *rotation unit*, and δ denotes the number of rotations.

Example 3. Fig. 2 illustrates a *Saturn* network where ring 1 is identical to the network of Fig. 1 and where some values have been replicated once (i.e., for which $\rho(v) = 2$). We assume that peers 4,912 and 7,640 were

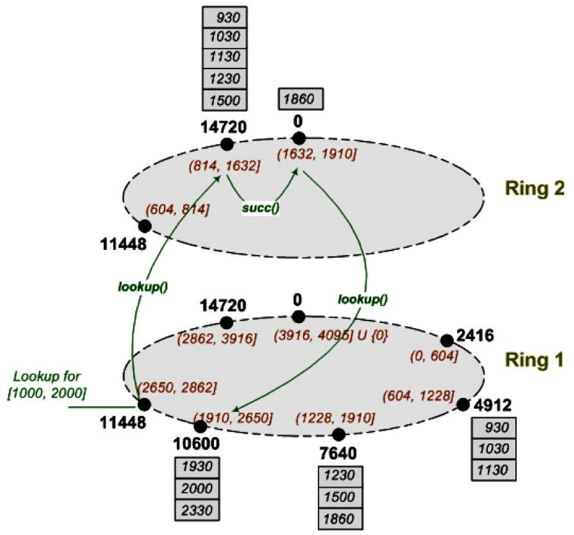

Fig. 2. A *Saturn* network with two rings.

overloaded and thus created replicas of their tuples at peers 11,448, 14,720, and 0 (assuming $\mathrm{rotation}[2] = 2$ and $s = 8{,}192$, i.e., half the identifier space). The figure also illustrates the partitions of domain DA that each peer is responsible for at ring 2.

Function *mrhf*() leverages the existence of a maximum of ρ_{max} replicas per value v, thus being able to choose one out of ρ_{max} possible positions for v in the system. That way, it fights back the effects of load imbalances caused by $hash()$. Note that $mrhf()$ is independent of the actual attribute domains and the only "global" piece of information required is the value of ρ_{max}. Also note that *randomly* selecting replicas using *rotation*[] aims at a uniform load distribution among replica holders. The result can be thought of as superimposing multiple rotated identical rings (as far as data are concerned) on each other and projecting them to the original ring. Thus, overloaded ("hot") and underloaded ("cold") areas of the rings are combined through rotation to give a uniform overall "temperature" across all peers.

3.2 Dynamic Load-Driven Data Replication

Each peer p keeps a local set of counters, $\alpha_p(v, \delta)$, indicating the number of times each one of its values, v in DA, was accessed at ring δ during a time interval T, autonomously set by each peer. We say that a value v is *popular* (or *"hot"*) when its access count $\alpha_p(v, \delta)$ at peer p at ring δ exceeds an upper limit, α_{max}, i.e., $\alpha_p(v, \delta) > \alpha_{max}$. Similarly, a value is *unpopular* (or *"cold"*) when its access count is below a lower limit, α_{min}, i.e., $\alpha_p(v, \delta) < \alpha_{min}$. In general, each peer can set its own α_{min}, α_{max} limits for the values it stores, based on its own capabilities. Moreover, a range of values is *popular* (*"hot"*) when at least one of its values is popular, and *unpopular* (*"cold"*) when all of its values are unpopular. Similarly, a peer is *overloaded* (*"hot"*) when at least one of the values that it is responsible for is popular, and *underloaded* (*"cold"*) when all values are unpopular.

In our scheme, *"hot" ranges of values* are replicated and rotated over the identifier space. Thus, the identifier space can be visualized as a number of replicated, rotated, and overlapping rings, the *Hot Ranges/Rings of Data* (Fig. 2)—so came the name of *HotRoD* [26], a preliminary version of *Saturn*. An instance of *Saturn* consists of an order-preserving DHT ring and a number of virtual rings where values are addressed using the multiring hash function, *mrhf*(). By "virtual" we mean that these rings materialize only through *mrhf*(), without any additional links (such as *succ*()) to connect peers between rings.

We denote by $\rho(v)$ the *replication degree* (i.e., the current number of instances) of a value $v \in DA$; obviously, $\rho(v) \leq \rho_{max}$. Each peer must have "write" access to this measure during replication (see function putRho() below), and "read" access during query processing (see getRho() below). We assume that all peers know the value of ρ_{max}. Moreover, we denote by $R_p(\delta) = [lowest_p(\delta), highest_p(\delta)]$ the range that peer p is responsible to store at ring δ (we describe later how $R_p(\delta)$ can be calculated locally), and by $avgR_p(\delta) = [avgLow_p(\delta), avgHigh_p(\delta)]$ the range defined by the average low and high bounds of the range queries it processes at ring δ during a time interval T. The load-driven replication algorithm is defined in Fig. 3.

First (line 1), if the access count $\alpha_p(v, \delta)$ of any value v stored at peer p on a ring δ exceeds the maximum access count limit, α_{max}, then the peer is deemed overloaded and initiates replication. In order to accomplish both a fair load distribution and efficient range query processing (to be discussed shortly), instead of only creating replicas of the tuples with the specific hot value v, we decide to replicate *all tuples kept on the nodes which are responsible to hold any value in the range* $repR_p(\delta = avgR_p)$ (δ) (lines 2-3). Assuming all $\rho(v)$ instances of v receive on average the same number of accesses, the total number of accesses of value v in the network can be approximated by $\alpha_p(v, \delta) \cdot \rho(v)$. Thus, to ensure that $\alpha_p(v, \delta) \leq \alpha_{max}, \forall v \in repR_p(\delta)$, the target replication degree *rho* for the replicated range is calculated by (lines 10-13)

$$rho = \max_{v \in repR_p(\delta)} \left(\left\lceil \frac{a_p(v, \delta) \cdot \rho(v)}{a_{\max}} \right\rceil \right).$$

The replication is executed through messages of type REPLICATE. When a peer p' receives such a message, it: 1) creates additional replicas of its relevant tuples until all of them have *rho* replicas (lines 14-24), and 2) forwards the replication message to its successor or predecessor, depending on which peer sent the message to p' (lines 25-28), until all peers of the arc that stores tuples with values in the range $repR_p(\delta)$ have been reached. When creating replicas, we can choose to either replicate all of a peer's value (lines 18-19) or just the fraction of the of its values in $repR_p(\delta;)$ (lines 15-16). In order to create replicas of a tuple t' with value v', peer p' sends a message of type COPY to peers $succ(mrhf(v', \delta))$ for $\delta \in (\rho(v'), rho]$. Peer p'' which receives such a message, stores the replica of the received tuple(s) and keeps the range being copied, i.e., $repR_p(\delta)$, in the δth position of a local array named $copiedRanges_{p''}[]$, used during range query processing.

Otherwise, when the access counts $\alpha_p(v, \delta)$ of all values v stored on peer p at ring δ are below the minimum access count α_{min}, (line 4), then p initiates deletion of unnecessary

```
// DYNAMIC LOAD-DRIVEN REPLICATION
// Executed periodically at each peer 'p'
1.  if (∃ v∈$R_p(δ)$,δ∈[1, $ρ_{max}$]: $α_p$[v, δ]>$α_{max}$) {
2.    rho = compute_rho(avgLow$_p$(δ),
         avgHigh$_p$(δ), δ);
3.    p.rcv_msg(REPLICATE, avgLow$_p$[δ],
         avgHigh$_p$[δ], δ, rho);
4.  } else if (∃ δ∈[1, $ρ_{max}$]:
                       $α_p$[v, δ]<$α_{min}$, ∀v∈$R_p(δ)$)) {
5.    rho = compute_rho(lowest$_p$(δ),
         highest$_p$(δ), δ);
6.    for v' in [lowest$_p$(δ), highest$_p$(δ)] {
7.      for r in [rho + 1, Rhos[v']]
8.        succ(mrhf(lowest$_p$(δ), r)).rcv_msg(
             DELETE, tuples(v'), r,
             lowest$_p$[δ], highest$_p$[δ]);
9.      putRho(v', rho); }

function compute_rho(low, high, δ)
// Computes the target replication degree for
// values in the interval [low, high] at ring
// δ on a peer p so that both the local $α_{max}$
// load limits and the global $ρ_{max}$ replication
// limit are satisfied.
10. rho = getRho(low);
11. for v' in [low, high] at ring δ
12.   rho = max(rho,
        ceil($α_p$[v',δ]*getRho(v')/$α_{max}$));
13. return min(rho, $ρ_{max}$);

function rcv_rep_msg(low, high, δ, rho)
// Executed upon arrival of a message of
// type 'REPLICATE' at peer 'p'.
// low, high: the boundaries of the range
// being replicated; δ: the starting ring;
// rho: the target replication degree
14. if (partial node replication allowed) {
15.   localLow = max(low, lowest$_p$[δ]);
16.   localMax = min(high, highest$_p$[δ]);
17. } else {
18.   localLow = lowest$_p$[δ];
19.   localMax = highest $_p$[δ]; }
20. for v' in [localLow, localHigh] {
21.   if (Rhos[v'] < rho) {
22.     for r in [Rhos[v'] + 1, rho]
23.       succ(mrhf(v', r)).rcv_msg(COPY,
             tuples(v'), r, low, high);
24.     putRho(v', rho); } }
25. if (sender == pred(p) and
        highest$_p$[δ] < high)
26.   succ(p).rcv_msg(REPLICATE, low, high,
        δ, rho);
27. else if (sender == succ(p) and
             lowest$_p$[δ] > low)
28.   pred(p).rcv_msg(REPLICATE, low, high,
        δ, rho);
```

Fig. 3. Dynamic load-driven replication algorithm.

replicas. Following the policy of dealing with ranges of values, we delete replicas such that all values of the range $R_p(\delta)$ have the same replication degree, *rho*, calculated as above except now only considering values in $R_p(\delta)$ (lines 5-9). Replica deletion is accomplished through messages of type DELETE sent to each peer $succ(mrhf(v', \delta))$ for δ in $(rho, \rho(v')]$. Each peer receiving such a message deletes the corresponding replica(s) and updates its local array of copied ranges (in both cases we also update the metadata information, as discussed later).

In any case, before a node starts creating/deleting replicas of a range of values, it first contacts the node(s) responsible for that range at the base ring (ring 1), informing them of its load values, limits, and the desired operation. In the case that multiple nodes holding replicas of the same values decide simultaneously to create or delete replicas, the base-ring nodes act as synchronization/serialization points; they are responsible for calculating the minimum amount of additional replicas required to satisfy the load limits of all requesting nodes, denying any extraneous replication requests, and for defining different target rings (δ values) for each accepted replication request.

The boundaries $lowest_p(\delta)$ and $highest_p(\delta)$ of range $R_p(\delta)$ can be calculated using the inverse *mrhf*() function. If pID is the identifier of peer p, then by definition $pID = mrhf(highest_p(\delta), \delta)$, thus $highest_p(\delta) = mrhf^{-1}(pID, \delta)$. Similarly, $lowest_p(\delta)$ can be calculated using $highest_{pred(p)}(\delta)$ of $pred(p)$, since $lowest_p(\delta)$ follows $highest_{pred(p)}(\delta)$ in the domain DA.

3.3 Handling Replication Metadata

The replication degree, $\rho(v)$, of each value $v \in DA$ is stored in a local array, $Rhos_p$[], on ρ_{max} peers dictated by $succ(mrhf(v, \delta)), \delta \in [1, \rho_{max}]$. Equivalently, each peer maintains the replication degrees of all values in its range of responsibility $R_p(\delta)$, for $\delta \in [1, \rho_{max}]$. Peers can retrieve and set the value of $\rho(v)$ for any $v \in DA$ via the functions *getRho*(v) and *putRho*(v, ρ), respectively. The former either retrieves the requested replication degree from $Rhos_p$[], if stored locally, or randomly chooses among and contacts one of its ρ_{max} holders. The latter visits all ρ_{max} holders and informs them of the new replication degree.

This placement strategy, in addition to contributing to fairly distributing the accesses for the replication degrees across the network, has the added benefit that *getRho*(v) invocations for values in the range of responsibility of a given peer (line 10 in Fig. 3) require no network access. On the other hand, a *putRho*(v, ρ) invocation requires one DHT lookup to update the corresponding *Rhos*[] entry in each of the $\rho_{max} (or \rho_{max} - 1$, if the caller is responsible for v at some ring) peers, for a grand total of O($\rho_{max} \cdot logN$) network hops.

We would like to note here that this approach is obviously geared toward an environment where *getRho*() invocations are very frequent, while *putRho*() is used much less frequently. This is a natural choice for our setting, as the replication degrees of values need to be read many times during both the load-driven replication algorithm and during range query processing (discussed shortly), while they are updated only when instances of a value are created or deleted. We would like to mention that the actual algorithm behind *putRho*() is somewhat orthogonal to our work and pertains more to the consistency requirements; for example, we could also adopt a lazy update mechanism, for the added "risk" of some extra hops during range query processing.

```
// RANGE QUERY PROCESSING
// Peer $p_{init}$ posing the query [$v_{low}$, $v_{high}$],
// uniquely identified by the identifier
// queryID
1.  rho = getRho($v_{low}$);
2.  δ = random(1, rho);
3.  succ(mrhf($v_{low}$, δ)).rcv_msg(RANGE, queryID,
       $p_{init}$, $v_{low}$, $v_{high}$, $v_{low}$, δ);
4.  while (more results)
5.    $p_{init}$.rcv_msg(RESULTS, queryID, tuples);

function rcv_range_msg(id,$p_{init}$,$v_{low}$,$v_{high}$,$v_{cur}$,δ)
// executed upon arrival of a message of
// type 'RANGE' at (current) peer p
6.  result = Ø;
7.  for cr ∈ CopiedRanges$_p$[δ]
8.    if (cr.highest > $v_{cur}$) {
9.      rangeHigh = cr.highest;
10.     break; }
11. for tuples t ∈ [cr.lowest, cr.highest]
12.   result.append(t);
13. $p_{init}$.rcv_msg(RESULTS, id, result);
14. $v_{next}$ = min(rangeHigh, highest$_p$[δ]) + 1;
15. if ($v_{next}$ > $v_{high}$) return;
16. rho = getRho($v_{next}$);
17. if (rho >= δ) {
18.   succ(p).rcv_msg(RANGE, id, $p_{init}$, $v_{low}$,
                      $v_{high}$, $v_{next}$, δ);
19. } else {
20.   $δ_{next}$ = random(1, rho);
21.   succ(mrhf($v_{next}$, $δ_{next}$)).rcv_msg(RANGE,
         id, $p_{init}$, $v_{low}$, $v_{high}$, $v_{next}$, $δ_{next}$); }
```

Fig. 4. Range query processing algorithm.

4 Multiring Range Query Processing

4.1 Range Querying in Multiple Rings

Consider a query for the range $I = [v_{low}, v_{high}]$ on attribute A initiated at peer p_{init}. Peer p_{init} 1) retrieves $\rho(v_{low})$, the replication degree of v_{low}, using *getRho*(), 2) randomly selects a number δ from 1 to $\rho(v_{low})$, and 3) sends the query to peer p_{low}: $succ(mrhf(v_{low}, \delta))$, i.e., the peer which is responsible to store v_{low} at ring δ. Peer p_{low} searches for matching tuples at ring δ, returns them to p_{init}, and calculates $v_{next} \in I$, $v_{next} = copiedRanges_p[\delta].highest + 1$. The process is then repeated for the subrange $[v_{next}, v_{high}]$, until all values of I have been looked up. The pseudocode is illustrated in Fig. 4.

Specifically, upon finishing looking for and returning matching tuples (lines 6-13), peer p forwards the query to the peer responsible for the next value of DA, v_{next}, whose δth instance is not found at p. If *rangeHigh* is the upper bound of the subrange processed by the current peer at ring δ, then v_{next} is the minimum between *rangeHigh* and $highest_p[\delta]$, plus 1 (line 14). The algorithm terminates if we have already covered the queried range (line 15). If v_{next} exists in a peer at ring δ (i.e., $\rho(v_{next})$ is equal to or higher than δ), p forwards the query to its successor (lines 17-18). Otherwise, p sends the query to a peer at a lower level ring selected randomly from 1 to $\rho(v_{next})$ (lines 20-21).

Example 4. Fig. 2 illustrates how the range query [1,000, 2000] is processed in *Saturn*. First, p_{init} (i.e., peer 11,448) randomly selects ring 2, and forwards the query to peer *succ*(*mrhf*(1,000, 2)), i.e., peer 14,720, using the lookup operation of the underlying DHT. Moving through immediate successors, the query reaches peer 0. When value 1,911 is not found at ring 2, the query randomly "jumps" to ring 1 and peer *succ*(*mrhf*(1911, 1)), i.e., peer 10,600, where it finishes.

4.2 Efficiency of Range Querying

The getRho(v_{low}) call at line 1 of Fig. 4 needs one DHT lookup, translating to O(log$N$) hops (N being the number of peers in the network). The getRho(v_{next}) call at line 16 only needs one hop; as we explained earlier, the replication degree of any value is kept at all ρ_{max} peers that may store its instances. Thus, during range query processing, a peer gets access to v_{next}'s replication degree by simply accessing its successor. In addition, a range query which is forwarded to a peer at another ring selected using function *mrhf*() (lines 3 and 21) requires one DHT lookup, whereas a range query which is forwarded to a successor at the same ring (line 18) requires only one hop.

In the underlying order-preserving DHT, each peer is responsible for a different partition of the domain DA. Since DA is uniformly distributed among peers, all partitions can be considered of equal size on average. Let P be the size of such a partition. Since peers are uniformly distributed in the identifier ring, we may also assume that P is the average size of the partition of any single peer at any ring in *Saturn*.

The best case for the efficiency of range query processing in *Saturn* is when all values of a range query are found at the same ring. This translates to O(log$N + n'$) hops, where n' is the number of peers that the query accesses at the ring: one DHT lookup to access the replication degree for the range's low bound, another DHT lookup to send the query to the node responsible for this value, and $n' - 1$ successor hops at the same ring. We may assume that, on average, a range query will move through $(n' - 1)/2$ jumps at different rings, each translating to O(log N) hops, and $(n' - 1)/2$ single-hop forwards to immediate successors. Given a range query spanning r values in DA, we can assume that $n' = \lceil r/P \rceil$. Thus, the average hop-count cost sums up to

$$O(\log N) \cdot (2 + (\lceil r/P \rceil - 1)/2) + 1 \cdot (\lceil r/P \rceil - 1)/2$$
$$\Rightarrow O(\log N) \cdot (\lceil r/P \rceil + 3)/2 + (\lceil r/P \rceil - 1)/2.$$

Note, however, that the expected number of jumps among different rings is significantly reduced by the fact that we have chosen to replicate ranges of values (as opposed to single "hot" values); this in essence means that, during range query processing, we expect that the single-hop forwards will outnumber the distant jumps—an intuition also verified by our experimental evaluation.

5 Fault Tolerance

Although load-driven replication can also help with data availability, *Saturn* includes an additional mechanism for fault detection and lost data restoration, making it even more resilient to failures. In the following sections, "data" indicate a peer's tuples (original and replicas), and "metadata" the peer's local $copiedRanges_p$[] and $Rhos_p$[] arrays.

5.1 Fault Detection

During range query processing, peers should be able to detect if a peer is missing due to a failure or an unexpected

```
function rcv_hrange_msg(id,p_init,v_low,v_high,v_cur,δ)
// executed upon arrival of a message of
// type 'RANGE' at (current) peer p;
// k: degree of horizontal replication
1.  result = Ø;
2.  predHighest = pred(p).rcv_msg(HIGHEST,1);
3.  if (lowest_p(1) == predHighest + 1)
4.    < execute lines 7-21 of Fig. 4 >;
5.  else {
6.    v_next = v_cur;
7.    < execute lines 7-21 of Fig. 4 using
        the local k-replication data store >;
8.    < execute lines 14-21 of Fig. 4 >;
9.    predp = pred(p);
10.   for m in [1, k] {
11.     predp.rcv_msg(COPYREQUEST);
12.     predp = pred(predp); }    }
```

Fig. 5. Range query processing algorithm with fault tolerance.

departure. The underlying DHT maintenance mechanisms may have already repaired the broken links (i.e., the pointers in the routing tables), but data/metadata may have been lost.

Saturn can easily detect missing successive peers at any ring during range query processing, by simply checking whether the visited peers are also successive in ring 1. Specifically, when a peer p receives a range query from its predecessor *pred*(p), it can detect whether any intermediate peers have failed between p and *pred*(p) by checking if the following condition holds:

$$lowest_p(1) == highest_{pred(p)}(1) + 1. \tag{2}$$

If a failure is detected, *Saturn* is able to calculate the range of values that was lost at ring 1 without any additional cost, since this range is obviously: $[highest_{pred(p)}(1), lowest_p(1)]$.

5.2 Data Availability Due to Load-Driven Replication

To achieve fault tolerance during range query processing, we add a slight modification to the previously presented algorithm (Fig. 4): when a peer failure is detected (using condition (2)), a new ring is selected to resume processing of the range query, until either there are no more rings to select from for the missing value or the query processing algorithm terminates. A *restoration* method is also applied, restoring data and metadata lost due to peer failures. Specifically, when a fault is detected and the range query is forwarded from peer p to peer p' at another ring, data and metadata are copied from peer p' to peer *succ*(p), which is now responsible to hold the missing data. Obviously, in case that all peers responsible for a given value at some ring have failed, all instances of this specific value have been lost.

As a first step, to enhance data availability, we define an extra parameter, $\rho_{min}(A)$, or simply ρ_{min}, specifying the minimum number of instances that each value of attribute A can have. We can thus adjust the minimum degree of replication and hence increase the degree of availability of all values.

5.3 Horizontal k-Replication

The above approach is capable of retrieving lost tuples when at least one of their replicas exists at a vertical ring. However, we pay extra messaging costs during range query

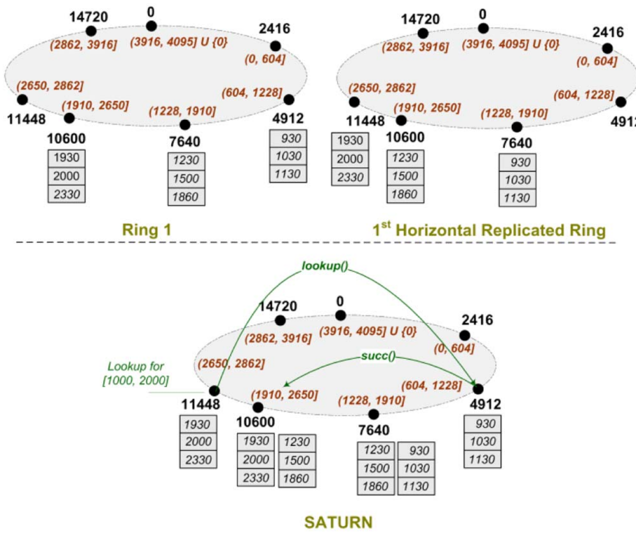

Fig. 6. Horizontal 1-replication in *Saturn*.

processing due to the additional jumps among rings. To this extent, we propose an enhanced mechanism to provide fault tolerance in *Saturn* while retaining its efficiency both in terms of messaging costs and data retrieval rate.

The core of this mechanism is based on *horizontal* replication, as opposed to the *vertical* replication provided by the load-driven replication scheme outlined earlier. In this scheme, each peer creates k replicas of its data and metadata to its k immediate successors ($k > 0$). Consequently, each peer p holds replicas of the data and metadata of each of its k predecessors. *Horizontal k-replication* is executed when a peer is inserted in the network and each time a change in its data or metadata occurs. Parameter k is called the *degree of horizontal replication*. The pseudocode in Fig. 5 describes rcv_hrange_msg(), the modified version of rcv_range_msg() of Fig. 4, enhanced with horizontal k-replication, fault detection, and lost data restoration.

Peer failures are detected, using condition (2), upon arrival of the range query message at a peer p (line 3). If a fault is detected, peer p first gathers any locally stored tuples from its k-replication data store and returns them to p_{init} (lines 6-7). The query processing algorithm resumes then from the last value retrieved during this step, by jumping to a new ring (line 8). Last, it updates and replicates the data/metadata structures of its own and its k predecessors/successors by sending a COPYREQUEST message to each of its k predecessors (lines 10-12). Each node receiving such a message must then make sure (through special messages) that there are k replicas of their data and metadata in their k immediate successors. The algorithm in Fig. 5 can be easily enhanced to also allow utilizing vertical replicas to retrieve lost data and metadata.

Example 5. Fig. 6 illustrates how the range query of example 2 is processed in a *Saturn* network enhanced with horizontal 1-replication. Each peer creates one replica of its data to its immediate successors (peers 4,912, 7,640, and 10,600 create replicas at peers 7,640, 10,600, and 11,448, respectively). We may illustrate each level of horizontal replication as a complete replica of Ring 1, where data are shifted by the k-replication level (e.g., in the mth horizontal replication ring, data are shifted by m steps clockwise). The final network is illustrated as an overlap of both rings. Thus, in the event of peer 7,640 failing, the range query initiated at peer 11,448 continues processing at peer 10,600, since the latter also holds the data and metadata of the faulty peer.

6 Discussion

One of the main premises of *Saturn* is its indexless nature, in the sense that no additional routing structure is required other than what is already provided by the underlying DHT overlay. This in essence means that, other than the DHT's own stabilization/maintenance procedure, *Saturn* incurs *no* extra cost to maintain routing table entries of participating nodes. However, as with all data replication strategies, there is a trade-off between load balancing, fault tolerance benefits, and replica management costs. The latter depend on ρ_{min}, ρ_{max}, and k, as well as on the replica management algorithm, dictated by the desired semantics and level of consistency. The maintenance of each "vertical" replica requires O(log N) network hops, while "horizontal" replica maintenance requires one-hop operations.

First, ρ_{max} defines the maximum number of vertical rings in the system and directly affects the level of load distribution fairness that can be achieved in the worst case; the load of the "hottest" value can be shared by at most ρ_{max} peers, or equivalently scaled down by a factor of up to ρ_{max}. Second, k and ρ_{min} define the number of "horizontal" and minimum number of "vertical" replicas, respectively, per data item, thus imposing an upper bound of $k + \rho_{min}$ peer failures before *Saturn* loses data. k further defines a limit on the number of failures of successive peers before *Saturn* resorts to looking for lost data items in remote replica holders, thus trading off horizontal replication for efficient range query processing under failures. In any case, a data item can have between $k + \rho_{min}$ and $k + \rho_{max}$ instances.

Moreover, the required level of consistency between replicas of data and metadata directly affects the choice of the replica maintenance algorithm. We would like to note that we consider the choice of the actual algorithm to be orthogonal to our work. Throughout this paper, we have chosen the more strict approach in which every update is routed to all replica holders. However, as also discussed in Section 3.3, if strict consistency is not a requirement *Saturn* may choose a lazy update scheme for both data and metadata, tradingoff data/metadata freshness for less hops during data insertion and replica creation/deletion and possibly extra hops during query processing.

However, the most significant property of *Saturn* is its flexibility. Different environments lend themselves to different combinations of the above system variables in order to achieve the best balance between fair load distribution, query processing performance, fault tolerance, and replication costs. For example, in a system with volatile or fault-prone nodes, we would set k to a higher value than in a system with stable nodes where the probability of abrupt node departures would be small. On the other hand, in a system handling a highly skewed workload we would opt for a higher ρ_{max} value, to let *Saturn* more leeway to adapt to load imbalances.

Moreover, the ability of every *Saturn* node to set its own access count limits $(\alpha_{min}, \alpha_{max})$, allows every peer to autonomously tweak the amount of load it receives during query processing, possibly trading it off for replica maintenance costs. We can further introduce a global limit, rep_{max}, on $\rho_{max} + k$, defining the maximum allowed number of vertical and horizontal replicas that any data item can have, thus imposing a global upper bound on the replication/maintenance costs of any peer. In the following section we will also see that, in many cases, both load fairness and high data retrieval rates can be achieved with a relatively small number of replicas, indicating that the relevant replication costs are usually low.

Saturn can further be configured for an even more dynamic and autonomic setting. For example, the degree of horizontal replication can be tweaked on a per-node basis, based on local reliability metrics: nodes that are by default volatile or that connect to the overlay through unreliable network infrastructures may autonomously opt for a higher k value. ρ_{max} can also be made dynamic, if we choose an extensible permutation scheme for computing the offset of each vertical ring of (1). For example, one alternative is to substitute $rotation[] \cdot s$ of (1) by function *offset*(), recursively defined as $offset(i) = offset(i - 2^{floor(log i)}) + 2^{m-i}, offset(1) = 0$, placing replicas in opposing "quantiles" of the identifier circle. Replica creation/deletion can be further tweaked too, by requiring that data items should be hot/cold for a number of time intervals before replicas are created/deleted, or by using some sort of "exponential back-off" algorithm, where the number of load distribution related replicas is increased rapidly (e.g., exponentially) and decreased slowly (e.g., linearly), to allow the system to reach a "replica equilibrium" faster and to react to such phenomena as flash crowds or abrupt changes in load access patterns. Further examination of these routes is left as future work.

7 Performance Evaluation

In this section, we experimentally evaluate *Saturn* by simulation. The *Saturn* simulator uses OP-Chord as the underlying DHT overlay and is in essence a heavily modified version of the internet-available Chord simulator [35], enhanced with order-preserving data placement [36] and our load-driven replication (Fig. 3), multiring range query processing (Fig. 4), and fault tolerance (Fig. 5) algorithms. We compare *Saturn* against:

- *Plain Chord* (*PC*), using the Chord simulator;
- an imaginary *enhanced Chord* (*EC*), assuming that for each range the system knows the identifiers of the peers that store all values of the range; and
- *OP-Chord*, an order preserving Chord-based network [36], equivalent to *Saturn* with only one ring.

7.1 Evaluation Criteria

The results are presented in terms of:

1. *the efficiency of query processing*, measured by the number of messages needed to process a range query;
2. *the fairness of the access load distribution*, measured by the Gini coefficient, G [12], [27], summarizing the

total amount of access load imbalances. Specifically, G is defined as the relative mean difference of the access loads of all peers

$$G = \sum_{i=1}^{N} (2 \cdot i - N - 1) \cdot l_i / N^2 \cdot \bar{l},$$

where l_i, $i = 1, 2, \ldots, N$ is the access load of the ith peer (N being the number of peers), and $\bar{l}$ their mean access load. The lower G, the less load imbalances and thus, the fairer the load distribution;

3. *the achieved fault tolerance*, measured by *recall*, defined as the ratio of the size of the query result set (i.e., the number of matching tuples retrieved) to the size of the original query result set (i.e., the number of matching tuples before failures) in a system where peers fail with a specific probability;
4. *the overhead costs*, measured by the degree of replication of peers and tuples.

7.2 Simulation Model and Workload

The experiments were conducted in N-node networks, where $N = 1{,}000$ and $10{,}000$, with routing tables of log N immediate successor and log N remote pointers. We used a T- tuple relation with a single-index attribute A, where $T = 5{,}000$ and $50{,}000$, respectively, taking *uniformly* random integer values in domain DA, where $DA = [0, 10{,}000)$ and $DA = [0, 100{,}000)$, respectively, (so that both in the smaller and larger-scale networks a node is responsible for about 10 attribute values).

We report on a series of $Q = 20{,}000$ and $200{,}000$ range queries generated as follows: the midpoint of a range is selected using a Zipfian distribution over DA with a skew parameter θ taking values from 0.2 (a near uniform distribution) to 1.2 (a very skewed distribution), in accordance to existing studies [8], [25], [34] testifying that the popularities of web and P2P file sharing objects follow Zipf's law with such values for parameter θ. The lower and the upper bounds of the range are randomly computed using a *maximum* range size equal to $2{\cdot}r$, for a given parameter r (thus, r is equal to the average range size, or there are r integer values in the queried range, on average). In our experiments, r ranged so as to yield an average range selectivity from 0.02 to 8 percent of the domain size ($|DA|$).

The measurements reported were taken when the system had entered a *steady state* with respect to the peer population and replication degrees. If not mentioned otherwise, experiments in *Saturn* were executed with a maximum replication degree of $\rho_{max} = 256$, high enough to allow unrestricted load-driven replication of popular ranges in almost all settings.

7.3 Efficiency of Range Query Processing

Assume a node poses a *range query* of size r. We assume that the matching tuples are stored on n peers under PC and EC and on n' peers under OP-Chord and *Saturn*. Thus, the expected [35] hop-count cost of range query processing is estimated as:

- PC: r exact-match queries are needed to gather all possible results (one for each value of the range) for an overall expected hop count cost of $1/2 \cdot r \cdot logN$.

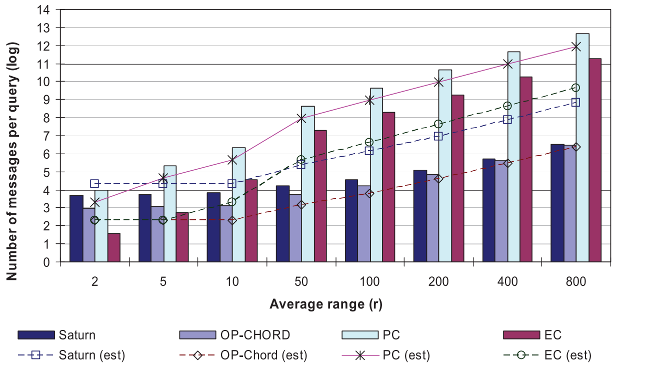

Fig. 7. Efficiency of range queries for *Saturn*, OP-Chord, PC, and EC (experimental results and theoretical estimations) for skewness $\theta = 0.8$ and different average range sizes r ($N = 1{,}000$).

- EC: n exact-match queries must be executed to gather all possible results. If there are T distinct tuples in the system each having a different attribute value uniformly selected from its DA, then each of the r values in the requested range will have a probability $(T/|DA|)$ of actually existing in a tuple, and thus $n = r \cdot (T/|DA|)$, for an overall expected hop-count cost of $1/2 \cdot (r \cdot T/|DA|) \cdot logN$.
- OP-Chord: one Chord lookup operation is needed to reach the peer holding the lower value of the range and n'-1 one-hop jumps to immediate successors; if P is the size of the partition of domain DA each peer is responsible for $(P > 1)$, then $n' = r/P$, for an overall hop-count cost of $(r/P - 1) + 1/2 \cdot logN$;
- *Saturn*: the range query needs $\alpha \cdot 1/2 \cdot logN + \beta \cdot (r/P - 1)$ hops, where α and β depend on the number of jumps performed among the rings, as discussed earlier.

Fig. 7 depicts the query processing hop-count cost for $\theta = 0.8$ and various values of the average range size r, for all four systems $(N = 1{,}000, T = 5{,}000, DA = [0, 10{,}000),\ Q = 20{,}000)$ in contrast to their theoretically estimated results. Compared against OP-Chord we see, as expected, that *Saturn* is more expensive. *Saturn*'s few extra messages (about 5 on average) are due to getRho() operations; this slight overhead pays back though when it comes to load distribution fairness, to be discussed shortly. As r increases, the message-count benefits of OP-Chord/*Saturn* versus PC/EC increase. The differences between the experimental with the estimated results in *Saturn* are because that actually less than one jump per query on average happened among rings in the experiments (that proves that our *range* replication scheme almost extinguishes "holes" in replicated ranges).

Fig. 8 depicts the performance of OP-Chord and *Saturn* of the larger scale experiment against the optimal range query performance of an order-preserving DHT overlay requiring only n' messages, i.e., one message for each peer that stores the values of the range. We see that the higher the range size r is, the closer OP-Chord and *Saturn* get to the optimal performance (the conclusions for smaller-scale networks are similar).

Testing with various θs our experiments showed that the skewness of the underlying Zipfian query distribution has

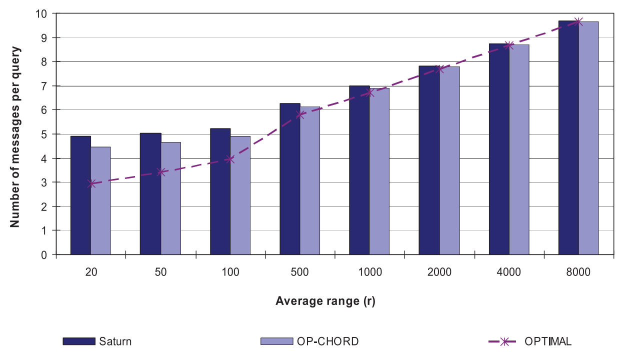

Fig. 8. Efficiency of range queries for OP-Chord and *Saturn* for different average range sizes *r*, compared to the optimal range-queriable DHT performance ($\theta = 0.8$, $N = 10{,}000$).

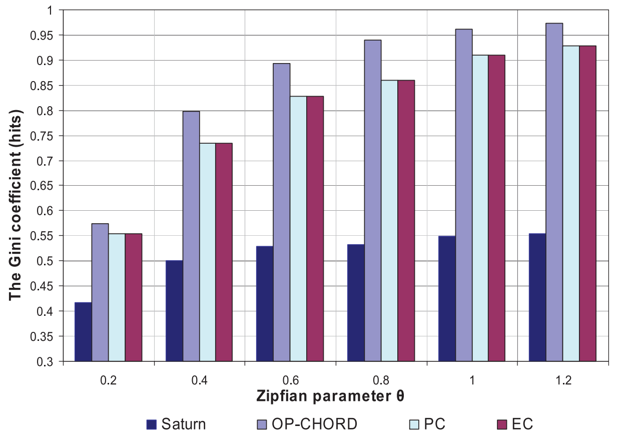

Fig. 9. The Gini coefficient (G) for *Saturn*, OP-Chord, PC, and EC for different Zipfian query distributions ($r = 50$, $N = 1{,}000$).

negligible effects on the efficiency of range queries. In the case of exact-match queries, PC, EC, and OP-Chord require one Chord lookup operation, translating to O(log N) hops in the worst case, or $1/2 \cdot \log N$ hops on average [35]. In *Saturn*, such queries require O(log N) additional hops in the worst case but log N hops on average, since two Chord lookup operations are needed: one for the getRho() operation, to retrieve the replication degree of the requested value and select a random ring to retrieve it from, plus one to retrieve the value from that ring.

7.4 Load Balancing

In these experiments, we evaluate the efficiency of *Saturn* in balancing the access load among the nodes of the network using the Gini Coefficient of the number of *hits* per peer. In the following experiments, we set $\rho_{max} = 256$ and the upper access count limit per value, a_{max}, equal to $2 \cdot r$. The various performance metrics were computed when the system had entered a *steady state* with respect to the peer population and the number of replicas.

We ran experiments with different average range sizes r and values of the Zipf parameter θ. Fig. 9 illustrates the Gini coefficient (G) of the hits distribution for $r = 50$ and various θ values ($N = 1{,}000, T = 5{,}000, DA = [0, 10{,}000)$, $Q = 20{,}000$).

It is apparent that *Saturn* has the lowest value for G compared to the other systems. More specifically, in *Saturn* G ranges from 0.41 for $\theta = 0.2$ to 0.55 for $\theta = 1.2$, whereas in

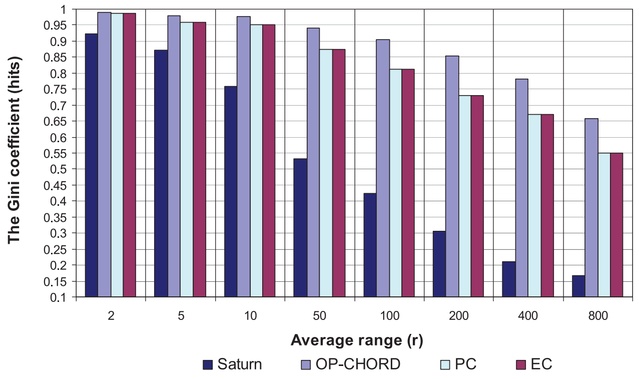

Fig. 10. The Gini coefficient (G) for *Saturn*, OP-Chord, PC, and EC for $\theta = 0.8$ and for different average range sizes r ($N = 1{,}000$).

OP-Chord, PC, and EC, G ranges from 0.55 for $\theta = 0.2$ to approximately 0.97 for $\theta = 1.2$. The most remarkable results with regard to *Saturn* are that 1) it succeeds to balance the load even in highly skewed query distributions (high θ values), and 2) the fairness of the load distribution does not seem to be affected by the skewness of the query workload (especially for high θ). This is reasonable since the higher θ is, the more replicas *Saturn* creates, and thus the load tends to be more uniformly distributed.

Please, notice that all simple DHT-based systems (PC, EC, and even OP-Chord) exhibit quite an unfair load distribution, especially for high θ values. This is an artifact of the absence of data replication, since neither random data placement nor data migration can solve the issues created by skewed query distributions.

Fig. 10 illustrates the fairness of the load distribution for $\theta = 0.8$ and for different range sizes r, for all four systems ($N = 1{,}000, T = 5{,}000, DA = [0, 10{,}000)$, $Q = 20{,}000$).

In general it holds that the higher the range size, the less load imbalances. This is expected since higher range sizes distribute the query load among more peers, thus resulting in a more fair hits load distribution. However, we can see that, for all values of r, *Saturn* distributes the load among the peers in the most fair manner, achieving a very low value for G for medium range sizes (i.e., $50 \leq r \leq 200$)—from 0.3 to 0.5—and an even lower value for larger range sizes (i.e., $r \geq 400$)—from 0.15 to 0.2. The value of G is higher—specifically from 0.76 to 0.92—for very low range sizes (i.e., $r \leq 10$), due to the fact that, in tandem with a high θ of the query access distribution, very few values are the recipients of almost the entire query load, and the replication degree limits imposed by our load-driven replication scheme do not allow *Saturn* to reach a fairer load distribution. Note, however, that even in this "hostile" setting, *Saturn* still achieves a consistently considerably lower value of G compared to the other three systems. In other words, *Saturn* manages to fairly distribute load in medium and large range sizes; even in very low range sizes it manages to reduce load imbalances keeping a very low replication cost. Access load distribution in PC and EC is the same, since the load is distributed among the same peers.

Similar results were depicted running experiments in 10 times larger scales. See, for example, Fig. 11 for N = 10,000, $T = 50{,}000$, $DA = [0, 100{,}000)$, $Q = 200{,}000$, where G

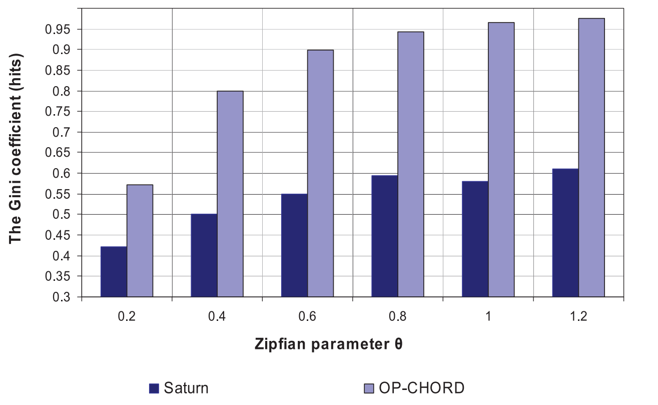

Fig. 11. The Gini coefficient (G) for *Saturn* and OP-Chord for $r = 500$ and for different Zipfian query distributions (*N* = 10,000).

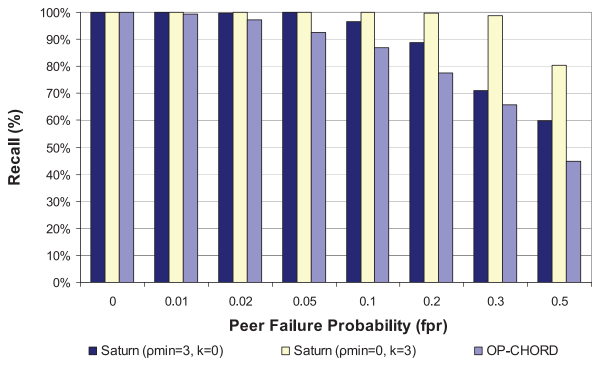

Fig. 12. *Recall* over different peer failure probabilities for OP-Chord and *Saturn* with ($k = 3, \rho_{min} = 0$) and without horizontal replication ($k = 0, \rho_{min} = 3$) ($r = 50$, $\theta = 0.8$, $N = 1{,}000$).

ranges from 0.42 for $\theta = 0.2$ to 0.61 for $\theta = 1.2$ in *Saturn*, and from 0.58 for $\theta = 0.2$ to around 0.98 for $\theta = 1.2$ in OP-Chord.

In all experiments, ρ_{max} was set to 256, resulting in a total of per-value replicas ranging from 8 (for $r = 2$) to 25 percent (for $r = 800$) of the number of nodes. This is definitely realistic, given typical sharing network applications [35], [38]; however, we stress that a fair load distribution can be achieved using even fewer replicas, as we shall see later in Section 7.6.

7.5 Performance under Failures

In this experiment, each peer may fail with a specific probability *fpr*, called the *peer failure probability*. We assume that the stabilization process of the underlying protocol (i.e., Chord in our experiments) has corrected the broken routing links (successors, predecessors, fingers), but *Saturn*'s data loss restoration algorithm has not been engaged yet. We evaluate the system's fault tolerance in terms of *recall*, defined as the ratio of the size of the retrieved query result set to the size of the original query result set, while also looking at the hop-count cost of query processing. We also compare results versus OP-Chord, lacking any fault tolerance mechanism (the reported results refer to the smaller-scale *Saturn* network of $N = 1{,}000$ nodes, but results for large-scale networks are similar).

We have examined two different settings of *Saturn*: 1) without horizontal replication (i.e., $k = 0$) and with a minimum vertical replication degree of $\rho_{min} = 3$, and 2) with a horizontal replication degree $k = 3$ and a minimum vertical replication degree of $\rho_{min} = 0$ (see Section 5.3). Results are averaged over different executions of the experiments using the same setup and various values for *fpr* ranging from 0.01 to 0.5.

Fig. 12 illustrates *recall* results for query ranges of average size $r = 50$ with a Zipfian query distribution with $\theta = 0.8 (N = 1{,}000, T = 5{,}000,\ DA = [0, 10{,}000), Q = 20{,}000)$. We can see that a relatively small horizontal replication degree $k = 3$ suffices to retrieve almost 100 percent of matching tuples for peer failure probabilities up to 0.3 (i.e., 30 percent of peers have failed) and more than 80 percent for $fpr = 0.5$ (i.e., in case half of the peers in the system have failed), which is remarkably high. With horizontal replication turned off and assuming a minimum replication degree $\rho_{min} = 3$, the achieved *recall* figures drop for higher values of *fpr* (e.g., about 70 percent for $fpr = 0.3$ and 60 percent for $fpr = 0.5$). As expected, OP-Chord manages to retrieve a percentage of tuples almost equal to the percentage of nonfailed peers (i.e., for $fpr = 0.2$ the *recall* is almost 0.80), since OP-Chord has no mechanism to retrieve tuples lost due to peer failures.

In addition *Saturn* with horizontal replication makes query processing more efficient in terms of communication costs. This happens because searching for missing tuples in successive horizontal replicas only requires one-hop, whereas jumping to other rings to search for vertical replicas requires O(log N) hops. Fig. 13 compares query processing efficiency with and without horizontal replication in terms of the number of hops required to retrieve all matching tuples for $r = 50$ and $\theta = 0.8$.

Saturn with horizontal replication is more efficient than without horizontal replication, as expected. In addition, the higher the peer failure probability, the larger the difference in hop-count efficiency. For example, for $fpr = 0.3$, without horizontal replication *Saturn* needs about 30 messages per query, whereas with horizontal replication it only requires about 20. Query processing in OP-Chord is significantly faster (since the number of active peers are reduced and thus, lookup hops are also reduced), but this result is largely artificial as its performance in terms of recall is significantly lower, as we showed in the previous figure. Therefore, horizontal replication in *Saturn* both increases recall (i.e., availability of tuples) and makes query processing more efficient (i.e., less messages required).

7.6 Replication Costs against Load Balancing

An important property of *Saturn* is its ability to trade off replication costs for a fair load distribution by tweaking the replication degree, as discussed earlier. We expect that the higher ρ_{max} is (recall that ρ_{max} is the maximum vertical replication degree), the more vertical rings are created and, thus, the fairer the load distribution becomes. Moreover, for each simulation case, we measure the number of per-tuple replicas created to estimate the replication maintenance costs (in all cases, we had disabled the horizontal k-replication, i.e., $k = 0$, since it does not play any role in balancing the query load among nodes).

Saturn was simulated for logarithmically increasing ρ_{max} values. Fig. 14 illustrates the role of ρ_{max} in load distribution and the number of replicas for $r = 50$ and $\theta = 0.8$; ρ_{max} ranges

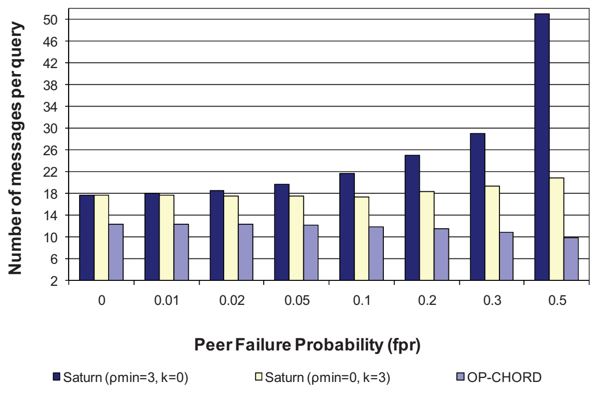

Fig. 13. Efficiency of range query processing over different peer failure probabilities for OP-Chord and *Saturn* with ($k = 3, \rho_{min} = 0$) and without horizontal replication ($k = 0, \rho_{min} = 3$) ($r = 50$, $\theta = 0.8$, N $= 1{,}000$).

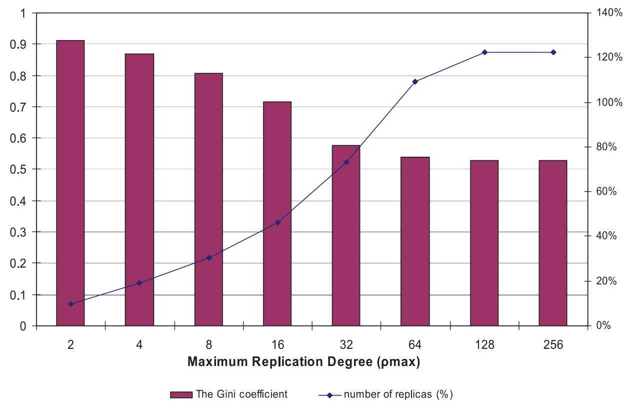

Fig. 14. Effects of the maximum replication degree, $\rho_{max}$to 1) load imbalances, expressed by G, and 2) the number of tuple replicas as a percentage of the original number of tuples ($r = 50$, $\theta = 0.8$).

from 2 to 256 ($N = 1{,}000, T = 5{,}000, DA = [0, 10{,}000)$, $Q = 20{,}000$). We see that, as ρ_{max} increases, more tuple replicas are created and G is decreasing. However, when ρ_{max} exceeds a value, no more replicas are created by the load-driven replication scheme and G is almost constant (about 0.50). Specifically, for $\rho_{max} = 91$, the number of tuple replicas reaches a maximum value of 120 percent of the original tuples (i.e., a little more than one replica per tuple), which is a reasonable cost. Beyond this "limit," the algorithm does not create more replicas and so, there are diminished returns in load balancing. This means that with reasonable replication costs we achieve a fair enough access load distribution.

Experiments with different θ's and r's showed that, the lower the skew parameter θ, or the higher the range size r, the more replicas required and, consequently, the higher the replication costs. The most important conclusion, however, is that when the query distribution is of medium or high skewness, or when the average size of the queries is small or medium, *Saturn* assures a fair load distribution with reasonable replication costs. Fig. 15 illustrates the relation among the skew parameter θ, the replication degree and load balancing when $N = 1{,}000, T = 5{,}000, DA = [0, 10{,}000)$, $Q = 20{,}000$ (the relation concerning range size is similar).

Moreover, we executed experiments with different Zipfian data distributions in order to study *Saturn*'s behavior in case of a nonuniform data distribution. Specifically, tuples' index attribute A takes integer values

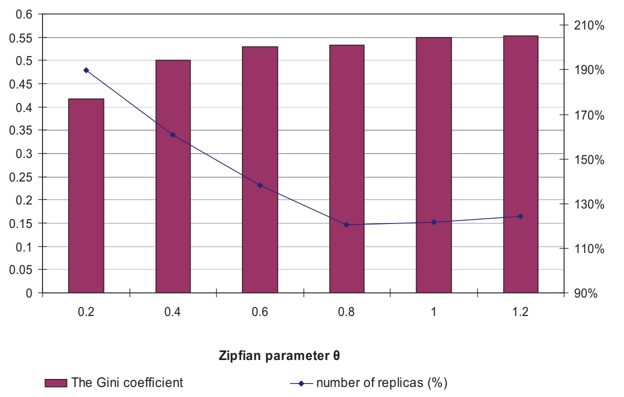

Fig. 15. Effects of the skewness of the distribution, θ to 1) load imbalances, expressed by G, and 2) the number of tuple replicas as a percentage of the original number of tuples ($r = 50$).

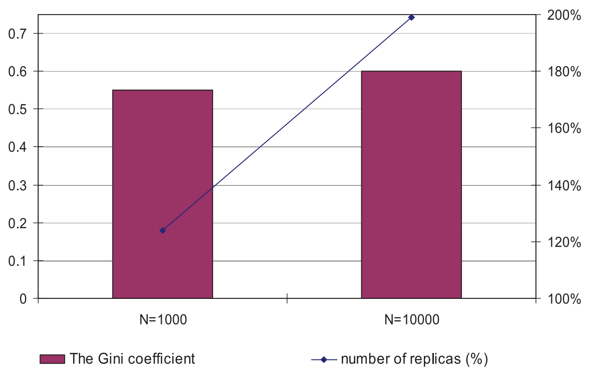

Fig. 16. Tuple replication degree and load balancing in *Saturn* networks of different sizes ($\theta = 1.2$, $r = 0.05\%$ of domain size, $|DA|$).

from a Zipfian distribution in $DA = [0, 10{,}000)$ with various skewness parameters θ_α. The results also exhibited *Saturn*'s efficiency and ability to tune load balancing against replication costs. For example, in a 1,000-node *Saturn* where tuples are distributed according to a Zipfian distribution with $\theta_\alpha = 0.8$, load imbalances are significantly low (i.e., $G < 0.55$), efficiency of range query is good (i.e., <18 messages) and the replication degree relatively small (i.e., 120 percent) for range queries with $r = 50$.

Finally, it is very interesting to show how the network size relates with the replication degree and load balancing. Fig. 16 depicts an example of the relation of the replication degree with load balancing (G) in *Saturn* networks of different sizes for $\theta = 1.2$ and average range selectivity 0.05 percent of domain size $|DA|$. It seems that the larger the network size, the larger the replication degree; the fairness of load distribution is similar in both cases. This shows that *Saturn* dynamically tunes the replication degree in order to achieve a fair load distribution in networks of different sizes. In addition, to reduce the replication costs in larger scale networks, we may increase the upper limit, α_{max}.

All results not only prove the scalability of our approach, but also show the ability of *Saturn* to tune replication costs in different workloads and different network sizes to achieve efficiency and load balancing.

8 Related Work

Almost as early as the first P2P systems appeared confined to support file sharing and efficient simple lookups, database researchers started investigating how these systems could be enriched with enhanced data management facilities ([16], [6], [18], [20], [28], etc.). Complex query and especially range query processing in P2Ps were among the first to be investigated. However, all initial works aiming to support range queries over popular P2Ps had many disadvantages, the most significant of which being that they suffered from load imbalances.

There are currently quite a few solutions supporting range queries, but the vast majority of them cannot both support load balancing and guarantee data locality ([3], [9], [17], [33], [36], [5], [19], etc.). Unfortunately, several of those works which deal with both range queries and load balancing issues ([4], [14], etc.) do not also provide fault tolerance. Those which, like *Saturn*, support efficient range queries, load balancing, and fault tolerance follow.

PHT [30] superimposes a Prefix Hash Tree onto an arbitrary structured overlay network. Although its universality is desirable, it is highly inefficient: due to the additional level of indexing to locate semantically close data, too many overlay network queries are required to answer range queries. P-Ring [11] is a P2P index structure that supports range queries, is fault tolerant, and provides logarithmic search performance even for highly skewed data distributions.

P-Ring provides a search performance of $O(\log_d N + m)$ hops, where N is the number of nodes, m is the number of peers with data items in the query range, and d is a tunable parameter. However, the higher d is, the higher the cost of storing and maintaining extra links to peers' routing tables becomes; also access load balancing is not discussed.

BATON [21] is a balanced binary tree overlay network which can support range queries and query load balancing by data migration between two, not necessarily adjacent, nodes. BATON* [22] is an improvement of BATON enhanced with a multiway fan out which provides reduced search costs, and with support of multiattribute queries. In addition to BATON*, Mercury [7] also supports multiattribute range queries and explicit load balancing, using random sampling.

In all the above approaches, load balancing is based on data migration. We expect that this will prove inadequate in highly skewed access distributions where some values may be so popular that single-handedly make the peer that stores them overloaded (transferring hot values from peer to peer only transfers the problem). Related research in web proxies has testified for the need of replication [37]. Replication can also offer a number of important advantages, such as fault tolerance and high availability [24], albeit at the expense of storage and update costs. *Saturn* can often keep these costs sufficiently low, as we experimentally indicated.

Finally, there are two solutions that both provide efficient range query processing and load balancing using replication: RHT [15] and P-Grid [1], [13], [2].

RHT [15], a Range Search Tree, is conceptually similar to PHT [30] in the sense that it uses a logical tree data structure on top of a DHT overlay. It supports multiattribute range queries and provides both storage and query load balancing by data partitioning and replication. However, the more popular a data item is, the more are the partitions that it is split into and thus, the more messages a query needs to be resolved. But even in case that data are uniformly distributed over peers, a range query is resolved with $O(\log r \cdot \log N)$ messages, for a range size r and a network with N nodes, which is less efficient than our approach which needs $\alpha \cdot O(\log N) + \beta \cdot (r/P - 1)$ messages with α close to 1 (as experimentally showed), and $P > 1$ where P is the average size of the partition that a node is responsible for.

The other approach is based on P-Grid [1], [13], [2], a distributed, prefix routing, trie-structured scheme used as DHTs. Datta et al. [13] describe how range queries can be handled in P-Grid using order-preserving hashing to map lexicographically sorted strings to binary strings. P-Grid also provides storage load balancing by bisecting skewed key partitions so that each partition carries the same storage load, and balances query load by associating more than one peer with each such partition (i.e., using replication) and maintaining multiple references in the routing tables to provide alternative access paths [2]. However, P-Grid replicates using only the knowledge of the data distribution for the sake of storage load balancing. Thus, access load balancing is not controlled, i.e., there is no guarantee that access load imbalances may not appear. *Saturn*, on the other hand, dynamically controls replication using the load-driven replication scheme and, thus, it manages to trade off range query processing efficiency for fair access load distribution.

In general, *Saturn* provides a completely indexless solution which means that there are no extra costs for building and maintaining additional routing state. On the contrary, both RHT and P-Grid build an additional tree-based network structure over the underlying DHT. Moreover, none of them specifically describe any mechanism for fault tolerance.

9 Conclusion

This paper presents the first (to our knowledge) attempt at concurrently attacking three key problems in structured P2P data networks: 1) efficient range query processing, 2) data-access load balancing, and 3) fault tolerance. The key observation is that replication-based load balancing techniques as well as frequent peer failures tend to obstruct techniques for efficiently processing range queries. Thus, solving these problems concurrently is an important goal and a formidable task. Some researchers claim that existing DHTs are ill suited to range queries since their property of uniform distribution is based on randomized hashing, which does not comply with range partitioning (i.e., [7]). However, *Saturn* succeeded in combining the good properties of DHTs with range partitioning using a novel hash function which is both order-preserving and randomized (in the sense that queries are processed in randomly selected—replicated—partitions of the identifier space).

We have taken an encouraging step toward solving the three key aforementioned problems through the *Saturn* architecture. *Saturn* reconciles and tradesoff message-count efficiency gains for improved data access load distribution fairness among the peers, on one hand, and replication costs for high data retrieval rate (i.e., recall) in case of peer failures, on the other. Compared to base architectures, our detailed experimentation clearly shows that *Saturn* achieves very good message-count efficiency coupled with a significant improvement in the overall access load distribution among peers with small replication overheads. Specifically, with the extra cost of a couple DHT lookups over the optimal performance of an order-preserving DHT, and a small replication degree, *Saturn* succeeds in balancing load even in highly skewed query distributions (for example, in medium range sizes with a replication degree a little more than 1, *Saturn* achieves G below 0.55). In addition, experimentation results show that *Saturn*'s fault tolerance techniques provide a high recall with very good message-count efficiency, for an extra, relatively low, replication degree. Specifically, even in a network with a high peer failure probability, *Saturn* is capable of retrieving almost all matching tuples with reasonable messaging and replication costs. Finally, *Saturn* can be superimposed over any underlying DHT infrastructure ensuring wide applicability and impact.

REFERENCES

[1] K. Aberer, "P-Grid: A Self-Organizing Access Structure for P2P Information Systems," *Proc. Ninth Int'l Conf. Cooperative Information Systems,* pp. 179-194, 2001.

[2] K. Aberer, A. Datta, M. Hauswirth, and R. Schmidt, "Indexing Data-Oriented Overlay Networks," *Proc. 31st Int'l Conf. Very Large Databases,* pp. 685-696, 2005.

[3] A. Andrzejak and Z. Xu, "Scalable, Efficient Range Queries for Grid Information Services," *Proc. IEEE Second Int'l Conf. Peer-to-Peer Computing,* pp. 33-40, 2002.

[4] J. Aspnes, J. Kirsch, and A. Krishnamurthy, "Load Balancing and Locality in Range-Queriable Data Structures," *Proc. 23rd Ann. ACMSIGACT-SIGOPS Symp. Principles of Distributed Computing,* pp. 115-124, 2004.

[5] J. Aspnes and G. Shah, "Skip Graphs," *Proc. 14th Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 384-393, 2003.

[6] P.A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu, "Data Management for Peer-to-Peer Computing: A Vision," *Proc. Fifth Int'l Workshop the Web and Databases,* pp. 89-94, 2002.

[7] A. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting Scalable Multi-Attribute Range Queries," *Proc. ACM Ann. Conf. Special Int'l Group Data Comm.,* pp. 353-366, 2004.

[8] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-Like Distributions: Evidence and Implications," *Proc. Conf. Computer Comm.,* pp. 126-134, 1999.

[9] M. Cai, M. Frank, J. Chen, and P. Szekely, "MAAN: A Multi-Attribute Addressable Network for Grid Information Services," *Proc. Fourth Int'l Workshop Grid Computing,* pp 184, 2003.

[10] J. Claussen, A. Kemper, D. Kossmann, and C. Wiesner, "Exploiting Early Sorting and Early Partitioning for Decision Support Query Processing," *Very Large Databases J.,* vol. 9, no. 3, pp. 190-213, 2000.

[11] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram, "P-Ring: An Efficient and Robust P2P Range Index Structure," *Proc. 27th Int'l Conf. Management of Data,* pp. 223-234, 2007.

[12] C. Damgaard and J. Weiner, "Describing Inequality in Plant Size or Fecundity," *J. Ecology,* vol. 81, pp. 1139-1142, 2000.

[13] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer, "Range Queries in Trie-Structured Overlays," *Proc. IEEE Fifth Int'l Conf. Peer-to-Peer Computing,* pp. 57-66, 2005.

[14] P. Ganesan, M. Bawa, and H. Garcia-Molina, "Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems," *Proc. 30th Int'l Conf. Very Large Databases,* pp. 444-455, 2004.

[15] J. Gao and P. Steenkiste, "An Adaptive Protocol for Efficient Support of Range Queries in DHT-Based Systems," *Proc. IEEE 20th Int'l Conf. Network Protocols,* pp. 239-250, 2004.

[16] S.D. Gribble, A.Y. Halevy, Z.G. Ives, M. Rodrig, and D. Suciu, "What Can Database Do for Peer-to-Peer?," *Proc. Fourth Int'l Workshop the Web and Databases,* pp. 31-36, 2001.

[17] A. Gupta, D. Agrawal, and A.E. Abbadi, "Approximate Range Selection Queries in Peer-to-Peer Systems," *Proc. First Biennial Conf. Innovative Data Systems Research,* 2003.

[18] M. Harren, J.M. Hellerstein, R. Huebsch, B.T. Loo, S. Shenker, and I. Stoica, "Complex Queries in DHT-Based Peer-to-Peer Networks," *Proc. First Int'l Conf. Peer-to-Peer Systems,* pp. 242-259, 2002.

[19] N. Harvey, M.B. Jones, S. Saroiu, M. Theimer, and A. Wolman, "SkipNet: A Scalable Overlay Network with Practical Locality Preserving Properties," *Proc. Fourth USENIX Symp. Internet Technologies and Systems,* 2003.

[20] R. Huebsch, J.M. Hellerstein, N. Lanham, B.T. Loo, S. Shenker, and I. Stoica, "Querying the Internet with PIER," *Proc. 29th Int'l Conf. Very Large Databases,* pp. 321-332, 2003.

[21] H.V. Jagadish, B.C. Ooi, and Q.H. Vu, "BATON: A Balanced Tree Structure for Peer-to-Peer Networks," *Proc. 31st Int'l Conf. Very Large Databases,* pp. 661-672, 2005.

[22] H.V. Jagadish, B.C. Ooi, K.-L. Tan, Q.H. Vu, and R. Zhang, "Speeding up Search in Peer-to-Peer Networks with a Multi-Way Tree Structure," *Proc. 26th Int'l Conf. Management of Data,* pp. 1-12, 2006.

[23] D. Karger, E. Lehman, F.T. Leighton, R. Panigraphy, M.S. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," *Proc. 29th Ann. ACM Symp. Theory of Computing,* pp. 654-663, 1997.

[24] A. Mondal, K. Goda, and M. Kitsuregawa, "Effective Load-Balancing via Migration and Replication in Spatial Grids," *Proc. Int'l Conf. Database and Expert Systems,* pp. 202-211, 2003.

[25] V. Padmanabhan and L. Qiu, "The Content and Access Dynamic of a Busy Web Site: Findings and Implications," *Proc. ACM Ann. Conf. Special Interest Group Applications, Technologies, Architectures, and Protocols for Computer Comm.,* pp. 111-123, 2000.

[26] T. Pitoura, N. Ntarmos, and P. Triantafillou, "Replication, Load Balancing and Efficient Range Query Processing in DHTs," *Proc. 10th Int'l Conf. Extending Database Technology,* pp. 131-148, 2006.

[27] T. Pitoura and P. Triantafillou, "Load Distribution Fairness in P2P Data Management Systems," *Proc. IEEE 23rd Int'l Conf. Data Eng.,* pp. 396-405, 2007.

[28] T. Pitoura and P. Triantafillou, "Self-Join Size Estimation in Large-Scale Distributed Data Systems," *Proc. IEEE 24th Int'l Conf. Data Eng.,* 2008.

[29] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Comm. ACM,* vol. 33, no. 6, pp. 668-676, June 1990.

[30] S. Ramabhadran, S. Ratnasamy, J. Hellerstein, and S. Shenker, "Brief Announcement: Prefix Hash Tree," *Proc. 23rd Ann. ACMSIGACT-SIGOPS Symp. Principles of Distributed Computing,* p. 368, 2004.

[31] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," *Proc. ACM Ann. Conf. Special Int'l Group Data Comm.,* pp. 161-172, 2001.

[32] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale P2P Systems," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms (Middleware),* pp. 329-350, 2001.

[33] O.D. Sahin, A. Gupta, D. Agrawal, and A.E. Abbadi, "A Peer-to-Peer Framework for Caching Range Queries," *Proc. IEEE 20th Int'l Conf. Data Eng. (ICDE,),* pp. 165-176, 2004.

[34] S. Saroiu, K.P. Gummadi, and S.D. Gribble, "A Measurement Study of Peer-to-Peer File Sharing Systems," *Proc. Multimedia Computing and Networking,* 2002.

[35] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Proc. ACM Ann. Conf. Special Int'l Group Data Comm.,* pp. 149-160, 2001.

[36] P. Triantafillou and T. Pitoura, "Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks," *Proc. First Int'l Workshop Databases, Information Systems, and Peer-to-Peer Computing,* pp. 169-183, 2003.

[37] K. Wu and P.S. Yu, "Replication for Load Balancing and Hot-Spot Relief on Proxy Web Caches with Hash Routing," *ACM J. Distributed and Parallel Databases,* vol. 13, no.2, pp. 203-220, 2003.

[38] Y.B. Zhao, J. Kubiatowitcz, and A. Joseph, "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing," Technical Report UCB/CSD-01-1141, Univ. of California, Berkeley, Apr. 2001.

Theoni Pitoura received the BEng degree in computer engineering and informatics from the University of Patras, Greece, the MSc degree in information technology from Imperial College, United Kingdom, and the PhD degree from the University of Patras, Greece. She has worked as a technical manager and research engineer for companies and research institutes and as a lecturer with the Hellenic Open University and the Technological Educational Institute of Patras. She is currently working at the public sector in Greece as a computer engineer. Her research interests include large-scale resource management with emphasis on query processing, resource distributions, and load balancing in peer-to-peer data networks. She is a member of the IEEE Computer Society and the IEEE.

Nikos Ntarmos received the Diploma in engineering from the Technical University of Crete, Greece, and the MSc and PhD degrees from the University of Patras, Greece. He is an adjunct assistant professor in the Computer Science Department, University of Ioannina, Greece. He has worked as a research engineer in numerous international research projects, as a technical advisor, security specialist, and chief systems administrator for groups of companies, research labs, and computer centers of many research and academic institutes, and as a software engineer in the RnD of various software systems. He also serves as a regular contributor to several open-source operating and software systems.

Peter Triantafillou received the PhD degree from the Department of Computer Science, University of Waterloo in 1991 and has previously held professorial positions in the School of Computing Science at Simon Fraser University, and in the Department of Electronic and Computer Engineering at the Technical University of Crete, Greece. He is currently a professor in the Department of Computer Engineering and Informatics at the University of Patras, director of the Network Centric Information Systems Laboratory, and director of the Software Division. His research efforts currently focus on large-scale distributed system infrastructures for content delivery, sharing, and integration, with emphasis on peer-to-peer systems, distributed event-based systems, social networking, and cloud computing. In the recent past, he worked on high performance storage systems and on distributed servers, with special emphasis on supporting multimedia applications. Earlier research activities focused on distributed file systems, multidatabases, and highly available distributed databases.